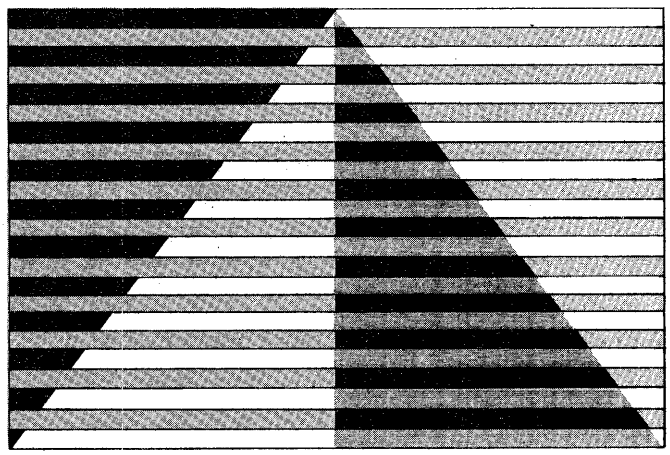
CONFERENCE PROCEEDINGS

The 2nd Annual Symposium on COMPUTER ARCHITECTURE



Sponsored by the IEEE Computer Society
and the Association for Computing Machinery

in cooperation with the University of Houston

JANUARY 20-22, 1975

Additional copies may be ordered from:
IEEE Computer Society Publications Office
5855 Naples Plaza, Suite 301, Long Beach, CA 90803
acm Association for Computing Machinery
1133 Avenue of the Americas, New York, N.Y. 10036

75CHO916-7 C

Symposium Chairman

WILLIS K. KING

Program Chairman

OSCAR GARCIA

Program Committee

HARVEY G. CRAGON
TSE-YUN FENG
REINER W. HARTENSTEIN
E. DOUGLAS JENSEN
TED A LALIOTIS
HAROLD LORIN
ELLIOTT I. ORGANICK
STEVE SHERMAN
DANIEL P. SIEWIOREK
HAROLD S. STONE
STEPHEN Y. SU
BRUCE WALD
RODNAY ZAKS

Symposium Committee

JAMES BARGAINER
OSCAR GARCIA
JUNG-CHANG HUANG
OLIN JOHNSON
JACK LIPOVSKI
DUANE PYLE
HUGH WALKER

Co-Sponsors:

ACM SIGARCH
Computer Science Department, University of Houston
Computer Society of the IEEE

CHAIRMAN'S REMARKS

A professional symposium should provide the attendees with the opportunities, 1) to gain new knowledge and keep abreast of new developments in the field and 2) to express and exchange ideas with other attendees. We believe the Second Annual Symposium on Computer Architecture will achieve these goals.

First of all we trust that anyone who reads this volume would agree with us that we have an abundance of high quality papers assembled here. As a matter of fact, it was most gratifying to us in organizing the symposium to find the tremendous enthusiastic response from the practitioners of the field in the form of the large number and high quality of papers submitted. The program committee, especially its chairman, of course, had to be faced with the difficult task of deciding how to limit the size of the technical program without leaving out any topic of importance. We are, however, pleased to say that every paper submitted was reviewed by no less than three referees and all papers accepted were rated as top quality by at least two of the referees. The result is a technical program consisting of twelve sessions with topics ranging from the architecture of computer networks to computer architecture education. Within this broad spectrum, most every computer architect will find some novel and exciting topic that will benefit him professionally.

No less encouraging and significant than the large number of papers being submitted, is the fact that over one third of them come from foreign countries. We are well aware of the fundamental contributions in computer architecture made by many European countries in the past. The full participation by those countries and others assure us that the state of art of research in computer architecture in the world will be reflected in the conference.

Furthermore, we were able to organize in conjunction with the symposium a one day tutorial on microcomputers --a subject that really attracts the interest and fancy of many of us. So, for those who come here with the intention of keeping themselves abreast in the field, they will not be disappointed.

As far as facilitating the exchange of ideas among the attendees is concerned, we provide the following: We have scheduled an evening of panel discussion with a list of distinguished speakers leading the discussion. We hope and expect to have some exciting and perhaps even heated debates on the current issues of computer architecture not only among the panelists but also with active participation from the floor. We have arranged a coffee break of half an hour after every one and a half hours of technical session so that the audience can have an additional chance to question the speakers closely while the subject is still fresh in everybody's mind. We provide a cocktail hour so that people can talk in a relaxed and informal atmosphere. We hold our meetings on the campus of the University of Houston away from the big hotels downtown so that attendees can seek one another out easily.

Beginning this year, we established a best paper award for the symposium. Steve Szygenda, as program chairman of the previous symposium, assumes the chairmanship of the award committee. The committee will not only examine the papers but also evaluate the oral presentation of the candidates. The award will be presented to the winner in the next symposium.

We are indebted to a great many people, too numerous, in fact to mention them all individually. However, we do want to acknowledge our appreciation to the following in particular. We would like to thank the chairmen of our sponsoring organizations, Jack Lipovski of TCCA and Chuck Casale of SIGARCH, for their continuous support. Mike Flynn, in spite of his busy schedule, never failed to provide his valuable advise and guidance when asked. Harry Haymann of IEEE, with his vast experience and contact, not only helped us in publicizing the symposium, but also in publishing the proceedings. The members of the symposium provide us with indispensable help, particularly, J. C. Huang who serves as treasurer and registration chairman, and Olin Johnson who serves not only as our publication chairman but also as the acting chairman of the Department of Computer Science, provided the extensive use of its facilities. To each of them, we would like to express our thanks. However, most of all we would like to send our gratitude to our program chairman, Oscar Garcia, and his program committee. Due to the extraordinarily large number of papers submitted and the strict standard of review we adhere to, it would have been impossible to get the job done were it not for the tireless effort and boundless resourcefulness of Oscar. We understand that before the technical program was finally hammered out, over forty people got themselves involved in the process of reviewing papers for the symposium.

Finally, we wish to acknowledge the assistance of the secretaries, particularly, I want to thank Alice Sand for her infinite patience and didication and help in answering all the correspondence and putting this proceedings in its final shape.

Willis K. King
Symposium Chairman

TABLE OF CONTENTS

PAGE

"Decentralized Priority Control in Data Communication,"
L. Nisnevich and E. Strasbourger, Technion 1

"A Loop Network for Simultaneous Transmission of Variable-Length Messages,"
Cecil C. Reames and Ming T. Liu, Ohio State University 7

"The Architecture of THE PICTURE SYSTEM,"
James F. Callan, Evans and Sutherland Computer Corp. 13

"A Fast Display-Oriented Processor,"
John Staudhammer, Jeffrey F. Eastman and James N. England, N.C. State University 17

"Computer Display of Colored Three-Dimensional Objects,"
Jeffrey F. Eastman and John Staudhammer, N.C. State University 23

"A Microprogrammed Processor for Interactive Computer Graphics,"
Henry D. Kerr, Adage, Inc. 28

"Functional Memory Techniques Applied to the Microprogrammed Control of an Associative Processor,"
C.V.W. Armstrong, University of Edinburgh. 34

"Instruction Design to Minimize Program Size,"
James F. Wade and Paul D. Stigall, University of Missouri. 41

"HMO, A Hardware Microcode Optimizer,"
James O. Bondi and Paul D. Stigall, University of Missouri 45

"The Computer Aided Design of Processor Architectures,"
A.M. Peskin, Brookhaven National Laboratory. 51

"Intermodule Protocol for Register Transfer Level Modules: Representation and Analytic Tools,"
W.H. Huen and D.P. Siewiorek, Carnegie-Mellon University 56

"Picture Systems, PS, and the Design of a Channel-to-Channel Computer Interface,"
Portia Isaacson, Xerox Corp. 63

"Reference Concepts in a Tree Structured Address Space,"
Lennart Löfgren, Saab-Scania 71

"A Virtual Memory for Microprocessors,"
Judith A. Anderson, NASA Kennedy Space Center and G.J. Lipovski, University of Florida 80

"The Performance Enhancement of Descriptor-Based Virtual Memory Systems Through the Use of Associative Registers,"
R.E. Brundage and A.F. Batson, University of Virginia. 85

"SPEAC—Special Purpose Electronic Area Correlator,"
Orin E. Marvel, Honeywell Government and Aerospace Products. 91

"Architecture Advances of the Space Shuttle Orbiter Avionics Computer System,"
James M. Satterfield, NASA Lyndon B. Johnson Space Center. 95

"Design Study of an Avionics Navigation Microcomputer,"
Uno R. Kodres and William McCracken, Naval Postgraduate School 99

"An Iteratively Structured Information Processor,"
Gerald R. Kane, University of Tulsa. 106

"Hardware-Software Interactions in SYMBOS-2R's Operating System,"
Hamilton Richards, Jr. and A.E. Oldehoeft, Iowa State University 113

"The Design and Evaluation of the Array Machine: A High-Level Language Processor,"
Pierre Sylvain and Maniel Vineberg, U.C.L.A. 119

"A Preliminary Architecture for a Basic Data-Flow Processor,"
Jack B. Dennis and David P. Misunas, M.I.T. 126

"Reduction Languages for Reduction Machines,"
K.J. Berkling, Gessellschaft für Mathematik und Datenverarbeitung, Bonn. 133

PAGE

"Output Devices Sharing by Minicomputers,"
Willis K. King and Fulvio Carbonaro, University of Houston 141

"On Relating Small Computer Performance to Design Parameters,"
S. Rannem, V.C. Hamacher, S.G. Zaky, P. Connolly, University of Toronto. 146

"Advantages of Structured Hardware,"
Harold W. Lawson, Jr., Linköping Högskola and Bengt Magnhagen, Datasaab. 152

"Concepts of the MATHILDA System,"
Peter Kornerup, University of Aarhus 159

"SOCRATES,"
Caxton C. Foster, University of Massachusetts. 165

"Conjoined Computer Systems: An Architecture for Laboratory Data Processing and Instrument Control,"
Donald F. Wann and Robert A. Ellis, Washington University. 170

"A Distributed Function Computer for Real-Time Control,"
E. Douglas Jensen, Honeywell, Inc. Systems and Research Center 176

"Switched Multiple Instruction, Multiple Data Stream Processing,"
C.H. Radoy and G.J. Lipovski, University of Florida. 183

"Sequentially Encoded Data Structures that Support Bidirectional Scanning,"
Robert J. Lechner, Honeywell Information Systems 188

"An Instruction Class for an Extensible Interpreter,"
Martin Freeman, American University. 195

"STARLET – A Computer Concept Based on Ordered Sets as Primitive Data Types,"
W.K. Giloi and H. Berg, University of Minnesota. 201

"A Cellular General Purpose Computer,"
R.G. Cornell, Bell Laboratories and H.C. Torng, Cornell University 207

"A Machine-Oriented Resource Management Architecture,"
Barry C. Goldstein and Thomas W. Scrutchin, IBM. 214

"A Design-Oriented Computer Engineering Program,"
M.E. Sloan, Michigan Technological University. 220

"An Educational Laboratory in Contemporary Digital Design,"
Janis B. Baron and D.E. Atkins, University of Michigan 225

REFEREES

The persons listed below have contributed their time and efforts to the reviewing process for this symposium, and their invaluable assistance is gratefully acknowledged.

M. D. Abrams
D. E. Atkins
J. F. Callan
K. M. Chandy
H. G. Cragon
D. C. Davis
J. B. Dennis
M. P. deRegt
R. A. Ellis
T. Y. Feng
E. A. Feustel
C. C. Foster
S. J. Garrett
H. Glass
M. J. Gonzalez
T. G. Hagan
R. W. Hartenstein
J. M. Hemphill
V. K. Jain
E. D. Jensen
S. Kamal
H. D. Kerr
W. K. King
T. A. Laliotis
G. J. Lipovski
H. Lorin
A. B. Marcovitz
P. N. Marinos
O. E. Marvel
E. I. Organick
L. Presser
D. M. Robinson
S. W. Sherman
D. P. Siewiorek
M. E. Sloan
J. Staudhammer
H. S. Stone
S. A. Szygenda
B. Wald
R. Zaks

DECENTRALIZED PRIORITY CONTROL IN DATA COMMUNICATION

L. Nisnevich & E. Strasbourger

Computer Science Department
Technion, Haifa, Israel

This paper describes a new principle for the control of data transmission within real time parallel systems. Control is effected by a number of identical units which are uniformly distributed among sender-receivers. When senders desire a transmission channel, their units try to capture the channel. The unit having the highest priority captures the channel. We describe a procedure for assigning and changing unit priorities under the constraint that customer service indices remain above given levels. The suggested procedure can be used to assign priorities in real time systems.

Index Terms. real time, parallel processing, decentralized control, priority control, data communication, channel control, priority assigning.

Introduction

In this paper, we are concerned with real time parallel systems and in particular with their data communication subsystems. In parallel systems decentralized control increases flexibility and reliability. The following comment[1] deals with this important feature of parallel system organization:

> "While graceful degradation is desirable in many commercial applications, it is essential in any military system where the results of complete system failure for even a short period of time could be catastrophic. The importance of this aspect of parallel processor techniques is further enhanced in military systems, since they face the loss of part of the system hardware not only from normal equipment and hardware failures, but also from the results of enemy action such as shell or bomb damage. In the latter case, however, it is important to note that this advantage of a parallel organization may be largely nullified unless the components of the parallel system are distributed physically as well as conceptually."

Recently, some publications have appeared[2,3,4] related to a data communication system with decentralized control proposal by Pierce[5]. Control is effected by distributing control units throughout the system. A Pierce system has many advantages, but data communication control in a real time system may require interrupts and priority servicing. It is difficult to include such features in a Pierce system. Other proposals[6,7,8] for decentralized data communication priority control in real time parallel computers have been based on the ideas of associative memory. Using these ideas, it is possible to achieve complete decentralization of priority control[13,14,15]. For such systems it is interesting to investigate the possibility of controlling priority parameter values in decentralized ways. This paper contains the outline of some results relating to decentralized priority control. First we shall consider schemes for the realization of dispersed control and then consider a scheme for changing priority values. We shall limit our discussion and consider only those systems in which the order of servicing is defined by absolute static priorities. The suggested methods may be extended to other systems.

Changing the order of servicing involves a reassignment of customer priorities. The purpose of such changes in this context is to find permissible priorities, i.e. to generate permissible values for multiple service indices. Such indices describe the level of customer service. Their values should not exceed the limits specified by the conditions under which customers operate. Examples of service indices are: average time between arrival and fulfillment of customer requirements for service, average number of customer requirements awaiting service more than a fixed limit. Such problems are important in real time computer systems. Their solution can have a strong influence on the structure of those hardware and software components which interact among the subsystems.

Channel Capturing Units (CCU)

Consider a channel connecting some sender-receivers (SR). We shall describe the decentralized control of the transmission time distribution to sender-receivers where the addresses of the SR's are fixed. This method was suggested in 1970[13]. The address of all

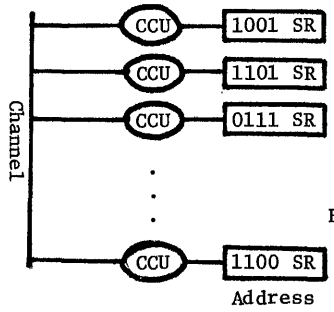


Fig. 1
Example Channel

SR's are binary numbers having an identical number of bits which determine the priority assigned to the SR. Each SR is provided with a channel capturing unit (CCU) (Fig. 1). If it is necessary to transmit a message, SR_i switches CCU_i into the active state. While in this state, CCU_i is watching the state of the channel.

The channel may be in three states: "transmit 1", "transmit 0", or "no transmission". The "no transmission" signal causes all of the active CCU's to transmit the highest bit (left most) of their addresses into the channel. In other words, after a message has been transmitted and the channel turns into the "no transmission" state, the active CCU's start transmitting their addresses into the channel. If at least one of the transmitted bits is one then the channel is in the state "transmit 1", while if all the transmitted bits are zero, the channel is in the state "transmit 0". CCU_i of SR_i compares the state of the channel with its own highest bit. If the channel is in the state "transmit 1" and CCU_i has transmitted a "0", then it switches itself off and awaits the next "no transmission" state. Otherwise, CCU_i remains connected to the channel.

Each CCU which remains connected to the channel transmits into the channel its second address bit and the channel turns into state of either "transmit 1" or "transmit 0". All of the CCU's which have sent their second bit behave as they did following the transmission of their first bit. In other words, if the channel is in the state "transmit 1" then all the CCU's which have sent the signal "0" are switched off and await the next "no transmission" state. This process

is repeated until all the address bits have been transmitted.

After the last bit is transmitted, only one SR remains connected to the channel (its CCU having captured the channel). The address of this SR is greater than the addresses of all of the other SR's that have been trying to occupy the channel during this period.

Example

Let the senders that require transmission time have the addresses 1001, 1101, 0111, and 1100 (as shown in Fig. 1). During the transmission of the highest bit the SR having the address 0111 will stop transmitting its address. The rest will start transmitting the second bit of their addresses. At this moment the SR having the address 1001 will stop its transmission. The remaining two SR's will transmit the third bits of their addresses which are 0. Therefore, they will continue to transmit their addresses. While the fourth bit is being transmitted, the SR having the address 1100 will stop transmitting. Thus, the channel will be occupied by the SR with address 1101.

After all the bits of the address have been transmitted, the CCU of the sender occupying the channel transmits the destination address and having received an answer as to whether it is free either starts sending a message for its SR or switches off and after some time repeats its attempt to establish communication with the same receiver.

Parallel Channel Capturing Units

Now consider a parallel scheme realizing decentralized control of the channel (Fig. 1). This scheme incorporates priority interrupt, whereby a higher priority CCU can preempt the channel from a lower priority CCU that is currently using the channel. For this approach the channel must have a line for each address bit plus a line for the message. Each line may be in one of two states: "transmit 0" if the line is dormant, or "transmit 1" if the line is set by one or more CCU's. Each active CCU continually attempts to capture the channel as follows: (see Fig. 2).

If the most significant address bit of an active CCU is a 1, then the CCU sets the most significant address line to "transmit 1" and activates the next most significant bit capture logic. Also, if the most significant bit is a 0 and the most significant address line is "transmit 0", the CCU activates the next most significant bit capture logic. However, if the most significant bit is a 0 and the corresponding address line is a"transmit 1", then the CCU simply waits for either a change in state of the address line (indicating that a higher priority CCU has relinquished the channel) or a change of state of the address bit (indicating a raise in priority of the CCU).

Thus, if the most significant address bit of an active CCU matches the corresponding address line of the channel, then the next most significant address line is set or compared and so on throughout the address lines or until a mismatch occurs. Only the address of the highest priority SR will completely match the channel address and force channel capture by its CCU. However, there is a transient time when two or more SR's might think they match the channel address (i.e. a logical 1 occurring at point "address match" in Fig. 2). This is because of transmission time lag from the time a higher priority CCU sets an address line to "transmit 1" and the time that the same address line at a lower priority SR switches from "transmit 0" to "transmit 1". This transient effect is eliminated by the settle detect, which has an output of 1 if and only if the "address match" stays at 1 for a sufficient period of time T as explained below.

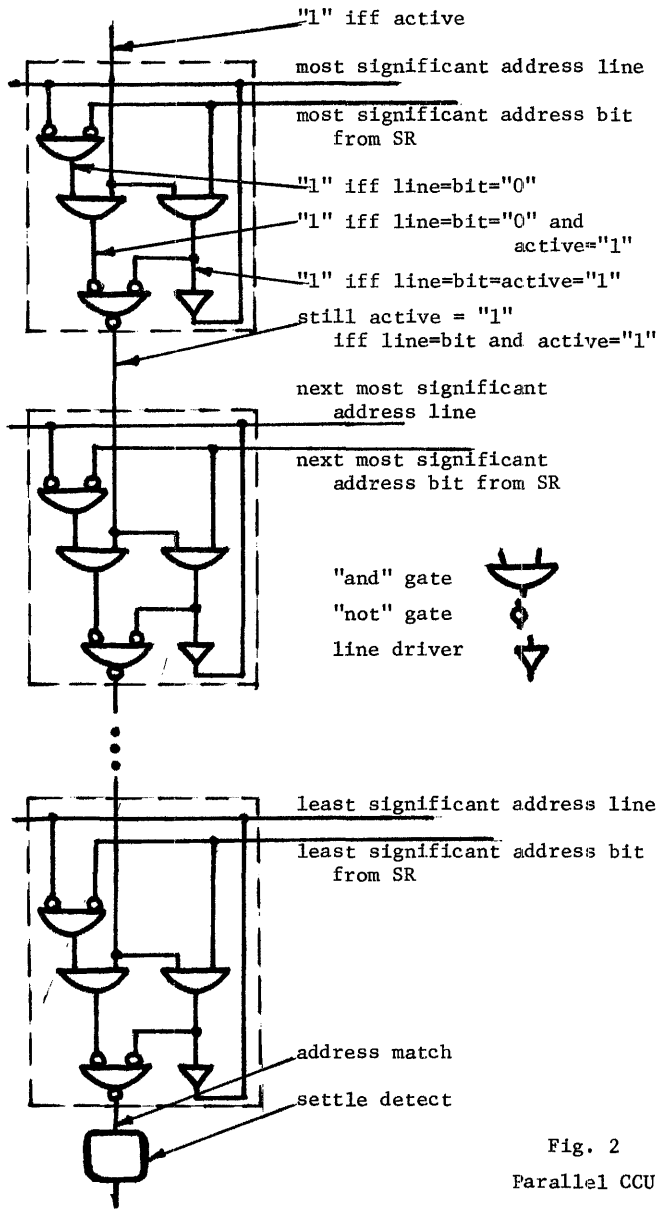


Fig. 2
Parallel CCU

For example consider the channel in Fig. 1 with the number n of bits per address equals 4, with maximum channel transmission time t_L, and with maximal gate delay time τ. Let the SR with address 0111 have control of the channel and let an SR with address 1111 become active which is $t < t_L$ time along the channel away from the 0111 SR. The 0111 SR will not release the channel until the following chain of events occurs: (see Fig. 3)

1. The"transmit 1" on the most significant address line propagates from the 1111 SR to the 0111 SR (time $t < t_L$)

2. The most significant address bit mismatch propagates through the 3 gates of the capture logic at each of the n = 4 address bit positions (time < $3n\tau$).

Therefore the 1111 SR has to wait a maximum time of $t_L + 3n\tau$ before the 0111 SR (or any other SR) releases control of the channel. A settle detect element of time delay $T = t_L + 3n\tau$ is thus needed after detection of a complete address match before a CCU can capture channel control for its SR. The CCU settle detect (Fig. 2)

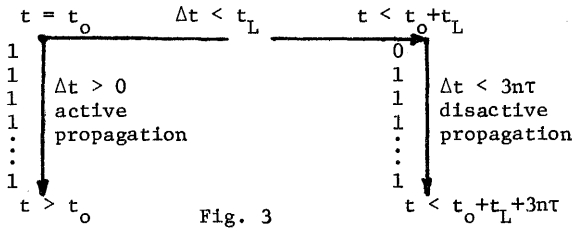


Fig. 3

Worst Case Timing While Waiting
For Channel To Clear

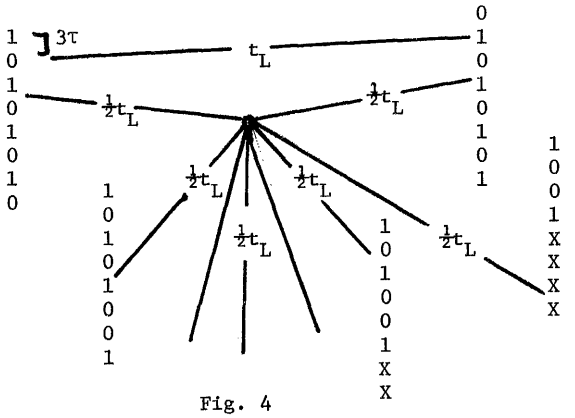


Fig. 4

Worst Case Timing Until Start Of Waiting Time
(Star Channel Configuration)

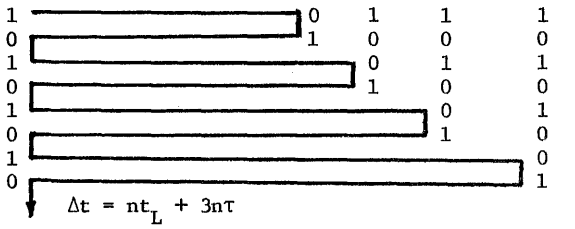


Fig. 5

Worst Case Active CCU Active Pulse Propagation
Of Star Configuration (Shown Linearly For Clarity)

provides this delay in a decentralized manner and assures that the channel is controlled by at most one SR.

It is also interesting to examine the maximum time that an active SR with the highest priority must wait while its active pulse propagates through its CCU until it reaches its settle detect element. Consider the following worst case as illustrated in Fig. 4 and Fig.5.

Let a channel address contain n address bits and let the SR's be arranged in a star such that each SR is t_L distant along the channel from all others. Furthermore, let an SR with address 101010...10 become active while an SR with address 0101...01 is in control. Also assume that SR's with addresses 1001..., 101001..., etc., become active at their critical worst effect times. Then the active signal propagation of the CCU with address 1010...10 can at worst follow the path as shown in Fig. 4 and Fig. 5, taking $\Delta t = nt_L + 3n\tau$ time. This sequence of events is unlikely but possible. Adding this active propagation time $nt_L + 3n\tau$ to the settle detect time of $t_L + 3n\tau$ yields a time of $(n+1)t_L + 6n\tau$ as a maximum bound on waiting before the highest priority active SR obtains control of the channel.

Notice that the higher the SR priority with the most consecutive address bits set to 1, then the less the waiting time for active pulse propagation. In particular, an SR with address containing all 1 bits waits only $2n\tau$ time until its delay element is activated.

Two advantages of the parallel scheme are:

1. Channel capture is much faster (at least an order of magnitude).

2. Priority interrupt is built in to the scheme alleviating the need to incorporate separate protocol and logic.

A disadvantage of the parallel scheme is the additional address lines (or higher bandwidth) needed in the channel.

Changing Priority Parameters

The general approach to the problem of selecting priority parameters is as follows. Assume that a system serves some fixed number N of customers denoted by i; i = 1, 2, ..., N. On the basis of some a priori considerations all the customers are divided into several groups which are assigned different priorities. Quantitative characteristics of system behavior are then specified only for the whole system and for the selected groups. These characteristics determine the sort of necessary priorities. Since each group is generally considered to have a unique priority the search routines which select adequate priority sets must use priority permutations.

A system in which customers are separated into L groups where each group has a different priority can be described as follows. Let us assign each customer i (i = 1, 2, ..., N) a priority parameter P_i and let the value of P_i be equal to the priority of the group to which the customer belongs. Therefore, all customers belonging to the same group will have identical parameter values, while the total number of different values will be L. The queue discipline is determined by priority parameter P_i in the following manner. Customer i has a higher priority with regard to customer j if $P_i > P_j$. Requirements of customers having identical values of priority parameters are serviced in the order "first come, first serve". Using this approach the value of each priority parameter P_i can be changed independent of the other priority parameters.

To illustrate, consider a given number L of priority groups dividing all the customers. Pick any customer i and let his priority parameter value change continuously from $-\infty$ to $+\infty$. Observe the changes taking place in the system. At the outset i will represent a separate group having the lowest priority. Then as the value of P_i becomes equal to the parameter value of the lowest priority group this customer will be included in it. Subsequently he will constitute a separate group again having a higher priority than this first group but a lower priority than the rest. This process will continue until customer i forms a separate group with the highest priority.

Priority Parameter Value Changing To Get Adequate Service

The possibility of changing priorities independently allows a system to respond in a very simple way to customer dissatisfaction when the level of service falls below a permissible value. Under such conditions the values of priority parameters for dissatisfied customers should be increased while leaving the rest unchanged.

In order to appreciate the need for demonstrating the validity of this simple procedure, consider another simple process of selecting priority parameters. Choose

the group of customers with the highest priority among all the groups whose level of service is inadequate. Interchange the priority of this group with that of the group having the next higher priority. Then continue this process of interchanging group priorities until the level of service is adequate for all groups in the system.

Such a process does not always lead to a satisfactory queueing discipline even when there is such a discipline. By way of illustration, select three groups of customers whose assigned priorities correspond with their numbers (1, 2, 3). Assume that under such a queueing discipline the level of service is inadequate only for the first group. Also assume that if we interchange priorities of the first and second groups then the level of service will be inadequate only for the second group. Continuing in this manner, we will return to the initial state and the process will cycle. If assigning the third group the lowest priority would in fact solve the problem, then this process will not find the solution.

Let us now return to the process of independent parameter change. This process ensures a solution of the problem whenever service indices possess certain properties described below. This method also permits the assignment of customers to a specified number of priority levels within the constraint of adequate service, providing such an assignment is possible at all.

For a fixed arrival and service pattern, the values of service indices depend only on the vector $\bar{P} = (P_1, P_2, \ldots, P_N)$. Denote the total set of service indices which have the property that each index is associated with exactly one customer by $\phi_1(\bar{P})$, $\phi_2(\bar{P}),\ldots,$ $\phi_K(\bar{P})$. Some subset of these indices indicate the service level of the first customer, another subset that of the second, etc. Denote the permissible level of the $\phi_k(\bar{P})$ index by C_k. Let the components of vector $\bar{P}^*$ be a permissible set of priority values for which the following system of inequalities holds:

$$\phi_k(\bar{P}^*) \geq C_k \quad (k = 1, 2, \ldots, K)$$

If we want to find a permissible set of parameters for a system in which the number of priority levels does not exceed L then we have to introduce an additional constraint that the components of the vector $\bar{P}^*$ take no more than L different values.

Note that any vector $\bar{P}$ whose components differ from the corresponding components of a vector $\bar{P}^{(0)}$ by the same value defines the same queueing discipline. Therefore, the service indices $\phi_k(\bar{P})$ do not change along the straight line defined by:

$$\begin{aligned} P_1 &= P_1^{(0)} + t \\ P_2 &= P_2^{(0)} + t \\ &\cdots\cdots\cdots\cdots \\ P_N &= P_N^{(0)} + t \end{aligned}$$

where $\bar{P}^{(0)}$ is any fixed vector and $-\infty < t < +\infty$.

Many service indices used in practice such as the examples cited earlier for the case of service by a single machine or a single channel have a so called monotonicity property which is explained as follows. Let $\phi(\bar{P})$ be a service index of customer i. If the vector $\bar{P}^{(1)}$ is generated from the vector $\bar{P}^{(2)}$ by decreasing the value of any priority parameter excluding the i-th, then $\phi(\bar{P}^{(1)}) \geq \phi(\bar{P}^{(2)})$.

For monotonic systems in which all service indices $\phi_1(\bar{P})$, $\phi_2(\bar{P})$, ..., $\phi_K(\bar{P})$ possess the monotonicity property, the problem of finding a permissible set $\bar{P}^*$ of priority values is solved by the process which we will call "simple descent" and denote by Π. The initial set of parameter values may be represented by any integer component vector $\bar{P}^{(0)}$. If at some stage of the process Π we have determined a set of parameter values $\bar{P}$ then the next stage will involve the following procedure. Choose a service index $\phi_k(\bar{P})$ for which the inequality $\phi_k(\bar{P}) \geq C_k$ does not hold for the vector found at the previous stage. Let $\phi_k(\bar{P})$ be the service index of customer i which means that this customer receives inadequate service for this index ϕ_k. We will assume that inadequate service for one index means inadequate service in general. The value of parameter P_i in the vector $\bar{P}$ is replaced by (P_i+1) while the rest of the components do not change. It has been shown[9] that the process Π terminates in a finite number of stages not exceeding $N(N-1)$ and results in a permissible set of priority parameter values provided such a set actually exists.

If we initially adopt the queueing discipline "first come, first serve" for all the requirements of all customers (this discipline is specified by a vector $\bar{P}^{(0)}$ with equal components) then the process Π will result in a set of priority parameter values in a number of stages less than $N(N-1)/2$. This set of values determines an adequate service level for all the customers with a minimum possible number of priority levels. In case an adequate set of parameter values does not exist, the process will result in a situation demonstrating this fact during the same number of stages. Therefore, the process having an initial vector with equal components solves the problem of assigning customers to priority levels when the number of levels must be limited.

At any stage of the process Π the values of service indices of several customers might lie below corresponding permissible values. In such a case one can choose one of these indices and raise the value of the priority parameter associated with the corresponding customer. It is possible to raise the values of several priority parameters simultaneously. The set of parameter values generated by the process Π does not depend on this choice, but is determined by the initial vector $\bar{P}^{(0)}$.

It is important to stress that in order to realize the search process one does not need to know the values of the functions $\phi_k(\bar{P})$ $(k = 1,2,\ldots,K)$. The only essential information is whether these values lie below specified levels. In other words it is only necessary to know if each customer is getting adequate service when the set of priority values is given.

Example

There are four customers in the system. Their requests arrive independently in conformity with a Poisson distribution. The customers are serviced by a single channel with exponential service times. The parameters of the corresponding distributions for each customer are:

$\lambda_1 = 1/5$ $\quad \lambda_2 = 1/20$ $\quad \lambda_3 = 1/10$ $\quad \lambda_4 = 1/20$

$\mu_1 = 1$ $\quad \mu_2 = 1/4$ $\quad \mu_3 = 1/2$ $\quad \mu_4 = 1/4$

The average total time (ATT) any requirement is in the system should not exceed 4 units for the first customer, 5 for the second, 16 for the third, and 30 for the fourth.

To find the values of the priority parameters satisfying these constraints, consider the operations of process Π. At the outset assign priorities in the order of decreasing the values of the given constraints (ATT), i.e. the initial vector $\bar{P}^{(0)}$ will be equal to (4,3,2,1). Under this queueing discipline the average total times are 1.25 for the first customer, 8.75 for the second, 9.17 for the third, and 37.50 for the fourth. As we can observe, this apparently natural set of priorities does not satisfy customers 2 and 4.

At the first stage of process Π raise the priority parameter values of customers 2 and 4. Then we obtain

priority vector $\overline{P}^{(1)} = (4,4,2,2)$. Under this queueing discipline, the average total times are 2.07, 5.07, 14.33, and 16.33. Now only the second customer is not satisfied. By raising his priority parameter value, we obtain the vector $\overline{P}^{(2)} = (4,5,2,2)$, which gives times of 3.33, 5, 14.33, and 16.33, thereby satisfying all of the customers. In this case partitioning the customers into three priority levels is sufficient and in addition is the only possible solution.

This method of changing parameter values in order to determine an adequate queueing discipline can be applied when the order of servicing depends on priority in other ways, i.e. in other classes of priority systems[10]. To ensure efficiency of the process Π, it is sufficient that service indices remain monotonic functions of priority parameters. Other papers[11] are devoted to the study of such systems and contain proofs of the efficiency of the simple descent procedure.

For real time systems the process Π may serve as a means to introduce dynamic priorities. Specifically, such priority will adapt to changes in the system and modify priority parameters when the values of service indices approach their limits provided external conditions change sufficiently slowly[9,10].

Distributed Control of a Multiplex Channel For Real Time Systems

Consider now the channel capture units as described earlier. Let the address of each SR consist of three parts: A = p s a.

Part p is the binary value of the customer priority parameter. This part changes independently as determined in the simple descent mode Π as just described. A modification of Π is required however, because of the upper limit of priority values, to prevent all customer priorities settling at the highest level. When an SR attains the highest priority level and is still not receiving adequate service, then its CCU will send the pulse "1" into the auxiliary channel ψ (Fig. 6). Each other CCU will respond by lowering its priority parameter p.

Part s is employed to provide the order of servicing "first come, first serve" among the SR's having equal priority values. This part of A is formed in the following manner. The value of s is 0 for all inactive SR's. When a requirement arrives and the CCU becomes active, it sends the signal "1" into the auxiliary channel S (Fig. 6). From this moment each "1" which is entered into the auxiliary channel S due to the activation of a channel capture unit is added to the value of s for each CCU awaiting transmission time. Among the senders belonging to the same priority group, the one whose waiting time for message transmission is the longest has the greatest value of s. After a message is transmitted and its SR becomes passive, its s returns to 0.

Address part a is the number (personal address) of an SR. This part is necessary only because address part p s may have the same value for two SR's. The auxiliary channels ψ and S can be combined into one line by special coding of the limit and the longevity pulses.

Thus in such a data communications system with decentralized control, transmission time is granted to an SR with the highest priority parameter value. In case of ties, an SR with the longest waiting time is chosen. If ties still remain, the SR having the greatest personal address is serviced.

Priority interrupt is built in to the parallel CCU scheme. For the serial CCU scheme a multiplex channel with interrupt capabilities may be organized by dividing the channel into two channels: one to transmit the addresses of the SR's and the other to transmit the regular messages (see Fig. 6). In this case the SR's try to capture the message channel by sending their addresses along the address channel after a "no transmission" state has appeared in the address channel. The CCU occupying the address channel transmits into this channel the address of the required receiver and connects its sender to the message channel. After this it creates a "no transmission" state in the address channel (i.e. switches off this channel) but remains active. When the "no transmission" state appears, the CCU's start to fight for the channel by trying to transmit addresses into the address channel. A CCU only disconnects its sender from the message channel when it has completed sending the message or when it has been preempted by another CCU. In the latter case its transmission time is interrupted and another SR with higher priority connects to the message channel and transmits its message. When a sender has completed its message it causes its CCU to enter the passive state and the CCU does not make any attempt to reference the address channel until it again becomes active.

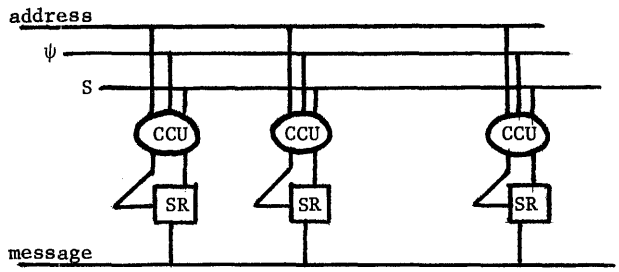


Fig. 6

Priority Interrupt Channel With Distributed Control

Papers[14,15] are devoted to the problems of broadening the functional possibilities of multichannel and multiloop data communication systems based on the principles discussed above. These papers also consider design methods for such systems.

The decentralized distributed control systems described above have a great deal of flexibility. In case such a system connects highly organized devices such as computer units, computer terminals, or computers, the address of an SR may be modified according to the transmitted message.

Conclusions

Parallel computers inherently have a communication control problem. Schemes have been presented in this paper to realize the following control capabilities:

1. decentralized priority control
2. decentralized priority changing
3. priority interrupt handling

Decentralized priority control using small channel capturing units enables efficient communication path utilization between parallel processors. Decentralized priority changing using independent priority determining automata alleviates unacceptable service conditions and lockouts. Priority interrupt handling allows minimal response time for most critical tasks. The implications of these capabilities upon parallel architecture should be investigated.

References

1. L.C. Hobbs and D.J. Theis. "Survey of Parallel Processor Approaches and Techniques". Parallel Processor Systems, Technologies and Applications, Ed. Hobbs, New York - Washington, 1970.

2. W.J. Kropfl. "An Experimental Data Block Switching System". The Bell System Technical Journal, Vol.51, No.6, 1972.

3. C.H.Coker. "An Experimental Interconnection of Computers through a Loop Transmission System". The Bell System Technical Journal, Vol.51, No.6, 1972.

4. E.E. Newhall, A.N. Venetsanopoulos. "Computer Communications-Representative Systems". Proc. of the IFIP Congress 71, Vol.1, Amsterdam-London, 1972.

5. J.R. Pierce. "Network for Block Switching of Data". The Bell System Technical Journal, Vol.51, No.6,1972.

6. R.J. Gountanis, N.L. Viss. "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System". Proc. of the IEEE, Vol.54, No.12, 1966.

7. M. Lehman. "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors". Proc. IEEE, Vol.54, No.12, 1966.

8. J.A. Githens. "An Associative, Highly-Parallel Computer for Radar Data Processing". Parallel Processor Systems, Technologies, and Applications, Ed. Hobbs, Spartan Books, New York - Washington,1970.

9. L.B. Nisnevich, G.G. Stetsyura. "Organization of Choice of Static Priorities in Systems with Decentralized Control". I. II. Automation and Remote Control, No. 6,7, 1971.

10. L.B. Nisnevich. "Organization of Service in Class of θ-Priorities". Automation and Remote Control, No. 8, 1969.

11. L.B. Nisnevich, G.G. Stetsyura. "Decentralized Control of Customers Interaction in Single-Step Systems of Resource Flow Allocation I,II". Automation and Remote Control, No. 5,6, 1972.

12. L.B. Nisnevich, G.G. Stetsyura. "Decentralized Control by the Simplest Descent Method Using Inaccurate Information". Automation and Remote Control, No.29, 1972.

13. G.A. Kotyujanski, L.B. Nisnevich, G.G. Stetsyura. "Decentralizovanoye prioritetnoye upravleniye v odnokanalnoy systeme obmena danimi". Izv. ANSSSR, Technitcheskaya Kibernetika, No.6, 1970.

14. L.B. Nisnevich, G.G. Stetsyura. "Mnogokanalnaya decentralizovanaya prioritetnaya sistema". Izv.ANSSSR, Technitcheskaya Kibernetika, No.2, 1971.

15. L.B. Nisnevich, G.G. Stetsyura. "Application of Principles of Decentralized Priority Control in Integral Systems of Data Transmission". Automation and Remote Control, No.5, 1971.

A LOOP NETWORK FOR SIMULTANEOUS TRANSMISSION OF VARIABLE-LENGTH MESSAGES

Cecil C. Reames and Ming T. Liu
Department of Computer and Information Science, and
Mechanized Information Center
The Ohio State University
Columbus, Ohio 43210

Abstract

A loop (ring) system is proposed for distributed computer networks which: 1) allows simultaneous transmission of variable-length message frames, 2) minimizes loop access and transmission times, and 3) provides a form of automatic traffic regulation. The ring interface transmitter which performs these three functions is described, and a conceptual model of its operation is developed. The model illustrates a technique by which the ring interface transmitter can delay incoming messages by hardware buffering just long enough for a variable-length output message to be placed on the loop. It is shown how advantage can be taken of gaps between incoming messages to clear out the delay buffer and to make room for future outgoing messages. It is further demonstrated that an interesting form of automatic message traffic regulation results from use of the proposed technique. Possible hardware implementations of the model are also considered, using a variable-length shift register for the incoming message delay buffer. The probable effects of the proposed technique on message transmission are discussed, and ongoing analytic and simulation studies are described.

I. Introduction

The loop topology is becoming increasingly popular today for the design of distributed computer networks [3,4,5,6,7,8,13,14,17,18]. A loop network consists of a high-speed digital communication channel, arranged as a closed loop (ring), to which the nodes (which may be processors, peripherals or terminals) are attached through simple ring interfaces. Messages are sent from a source node as addressed blocks of data which travel around the loop from interface to interface until picked off by the destination node. Some of the major advantages of a loop network are: 1) the ease of controlling information, 2) the simplicity of message routing, 3) the high rate of data transmission, 4) the facility to attach dissimilar nodes through a standard ring interface, and 5) the low costs of construction and incremental expansion.

With only a few exceptions, most loop systems studied or constructed have employed the concept of fixed-size frames (slots) for message transmission [3, 4,5,8,9,13,17]. Such systems, as illustrated by Fig.1a, are sometimes called Pierce loops. The reason for using a fixed-size frame is that message transmission protocol and ring interface hardware are thereby greatly simplified. The loop is initially filled with an integral number of empty frames, where a bit in the control field of each frame specifies if that frame is empty (not in use) or full (in use). If a node has a message it wishes to output which is shorter than the frame size, it simply waits for an empty frame to appear and then inserts its message into that frame; the unused frame space is wasted. On the other hand, should the output message be longer than the frame size, it must be broken down into frame-size packets and transmitted one packet at a time. The disadvantages of this approach are obvious: message space on the loop is wasted for short messages, while longer messages require elaborate buffering and disassembly/assembly techniques. Clearly, it would be more efficient if variable-length messages could be transmitted directly as variable-length frames.

A few attempts have been made to provide loop systems for transmission of variable-length message frames, but such systems, using Newhall loops as shown in Fig. 1b, have been severely handicapped by their inability to allow simultaneous message transmission [6,14]. Newhall loops operate under a round-robin control-passing mechanism which permits only one node at a time to transmit an arbitrary length message onto the loop. This restriction, although highly inefficient as far as loop utilization and message transmission is concerned, is absolutely necessary. Otherwise, two or more messages transmitted onto the loop during overlapping time periods could completely destroy each other.

It is primarily for this reason that most loop designs have favored fixed-size frames for message transmission. It seems entirely reasonable to suppose, however, that if variable-length frames could be transmitted as easily as fixed-size frames, it would be advantageous to do so. A loop system constructed for variable-length message transmission should result in faster message transmissions and better loop utilization, since no frame space (and thus time) would ever be unnecessarily wasted.

In this paper, a new ring interface design is proposed which does allow simultaneous transmission of variable-length message frames in a loop system. It will be argued that the proposed system does have the desirable properties mentioned above. It will also be asserted that the proposed scheme provides a bonus in the form of automatic traffic regulation, in the sense that access to the loop is controlled by the ring interface hardware based on the current system load and previous individual accesses. Preliminary investigation by the authors has already established the feasibility of the proposed scheme; analytic and simulation studies are now underway to verify the claims made for improved performance [18].

II. Design Considerations

Rather than immediately launch into a description of the proposal, it would be instructive to consider first a few of the motivating factors which led to the particular design chosen. The desired goal was a variable-length message transmission technique which would not restrict use of the loop to a single node at a time. Any such scheme must ensure that two or more messages, transmitted from different nodes during the same time period, never meet and interfere with each other. But if a completely distributed loop network is assumed, with each node acting independently and without knowledge of other nodes' actions, then it is impossible to guarantee that simultaneous transmissions will not cause messages to overlap and destroy

This research was supported in part by the National Science Foundation under grants GN-534.1 and GN-27458 and in part by the Office of Naval Research under contract N00014-67-A-0232-0022.

each other.

The obvious means of overcoming this problem is to buffer any incoming messages received at a node while an output message is being transmitted. The loop network then takes on some characteristics of a message store-and-forward network. While such a solution is obvious, it is not so easy to apply to a distributed loop network, for the nodes in such a network may well be unintelligent peripheral devices that are incapable of buffering messages.

In fact, in a loop network the node itself never sees messages not addressed to it; only the ring interface sees all messages that pass by on the loop. So if any buffering of incoming messages is to occur, it must be done entirely by the ring interface. But this requirement poses a new problem: the ring interface is a relatively simple, inexpensive ($200-$600) hardware device with little data buffering capability. It was deliberately designed that way to be cheap and reliable. Replacing it by a much more expensive minicomputer is not a feasible alternative, although in the future a microprocessor system might be economical. For the present time, however, it seems that the function of the ring interface must be expanded by giving it some hardware buffering capability.

Before describing how the ring interface is to accomplish this buffering of incoming messages while output is in progress, some other desirable characteristics for such a loop system will be listed for inclusion in the new design:

1) Incoming messages should experience the minimum delay possible in passing through the ring interface, the amount depending solely on the quantity of previously buffered data.

2) Outgoing message transmission should be possible at the completion of any incoming message, even if another message is incoming and must be delayed.

3) The quantity of data buffered by a ring interface should be reduced whenever possible and should be strictly limited so as to avoid unreasonable transmission delays.

4) All nodes should be allowed access to the loop as much and as often as the system can handle, but no node should be permitted to dominate the loop to the exclusion of other nodes.

III. Proposal for New Ring Interface

With these thoughts in mind, the major functions of the ring interface proposed in this paper will now be described. The scheme chosen allows the ring interface to delay an incoming message by hardware buffering just long enough for a variable-length message to be output onto the loop. The maximum length of an output message which can be transmitted at any moment is determined by the available space in this hardware delay buffer. More space is made available by taking advantage of gaps between messages to reduce the amount of data buffered, gaps appearing either from a lack of message transmission or due to removal of a message from the loop by its receiving node. Since an output message cannot be transmitted until sufficient buffer space is available to delay possible incoming messages, and since space is made available by the accumulation of intermessage gaps whose arrival rate depends on the total system traffic, it can be seen that output message transmission is automatically regulated as a function of traffic load.

A conceptual model of the proposed ring interface transmitter will be developed in the next section, so that its operation can be explained and understood without worrying about hardware details. Following the model presentation, Section V will consider hardware implementations of the model, with several approaches being suggested. A discussion of the effects the proposed scheme might have on message transmission in a loop system will be taken up in Section VI.

IV. Conceptual Model Development

The operation of the proposed ring interface transmitter will be explained through the development of a hardware-independent conceptual model; the design of the interface receiver is not considered in this paper. Fig. 2 is an illustration of the data buffers and paths for the model. It shows an Output Buffer and a multifunctioned Delay Buffer as the two data storage devices in the transmitter. Incoming messages enter this part of the ring interface from the loop communication channel bit-serially over the Input Line. The same incoming messages (perhaps after some delay) and outgoing messages from the attached node exit the transmitter onto the loop bit-serially on the Output Line. The Input and Output Lines are synchronized at the same data rate, so that for each bit input, another is output.

The function of the Output Buffer is simply to hold the output message as it is being assembled by the attached node and while it is waiting for its transmission onto the loop to begin. When transmission actually is to start, the output message will be parallel-transferred into a portion of the Delay Buffer, freeing the Output Buffer so that it can again be filled by the attached node. Since messages are of variable length in this system, only a portion of the Output Buffer may actually be used for a particular message transmission. The attached node is allowed to output a message of any length that does not exceed the capacity of its Output Buffer.

The Delay Buffer is slightly more complicated, for it has several functions to accomplish. It is physically one storage device, but logically it can be thought of as partitioned into two distinct storage devices, the boundary of this partition being changed from time to time. In the illustration of Fig. 2, this partitioning is shown as active and inactive portions of the Delay Buffer. Consider for the moment just the active portion.

The active portion of the Delay Buffer acts as a FIFO queue that delays the retransmission of incoming messages for a specified time period. Assume the entire Delay Buffer to have a capacity for storing N bits of data, and assume that r bits are presently allocated to the active portion, $1 \leq r \leq N$, leaving N-r bits for the inactive portion. Let the bit positions in the Delay Buffer be labeled DB_0, DB_1, ...DB_{r-1}, DB_r, ...DB_{N-1}. Choose a time scale for the rate of data transmission on the loop so that message bits arrive every one time unit. Then if the next message bit is to arrive at time t+r, the active portion of the Delay Buffer (DB_0, DB_1, ...DB_{r-1}) represents a FIFO delay queue which contains the message bits that arrived at times t, t+1, ..., t+(r-1).

Since the capacity of this FIFO delay queue is only r bits, at time t+r the longest delayed message bit (contained in DB_0) is removed from the queue and is output to the loop on the Output Line. Simultaneously, the incoming message bit from the Input Line is added to the queue (stored in DB_{r-1}) so that the length of the delay queue remains constant at r bits. Thus at the completion of these activities at time t+r, the new contents of DB_0, DB_1, ...DB_{r-1} will be the message bits received at times t+1, t+2, ... t+r. Stated in hardware terms, the contents of DB_0, DB_1, ...DB_{r-1}, Input have been shifted into Output, DB_0, DB_1, ...DB_{r-1}.

The size of this FIFO delay queue is adjustable at each time period (if r>1), the object being to reduce r to 1 as quickly as possible. A reduction can take place only between incoming messages, when the data being received is not part of any message and thus does

not need to be saved for later transmission. Thus, the preceeding explanation of the FIFO delay queue operation should be modified to state that if r>1 and the bit input on the Input Line is not part of any incoming message, then the input is discarded (not stored in DB_{r-1}) and r is reduced to r' = r-1. In effect, the logical boundary of the partition of the Delay Buffer is changed one position, so that the active portion is smaller and the inactive portion larger. When r reaches 1, no further reduction is possible, and the input is always stored in DB_0.

The space in the inactive portion of the Delay Buffer, DB_r, DB_{r+1}, ...DB_{N-1}, is not just wasted, but also plays a very important role in the ring interface transmitter operation. It is here that output messages are transferred from the Output Buffer when they are to be transmitted onto the loop. Assume an s-bit output message has been prepared by the attached node and is in the Output Buffer awaiting transmission. Assume also that at least s+1 bits of space are available in the inactive portion of the Delay Buffer (which is of size N-r bits). That being the case, the s bits of the Output Buffer are parallel-transferred into the inactive portion of the Delay Buffer at DB_r, DB_{r+1}, ...$DB_{r+(s-1)}$, directly adjacent to the active portion. Then if no incoming message is being received over the Input Line, r is immediately changed to r' = r+s+1; otherwise, the change is made at the end of the current incoming message.

This sudden change in r means that now the active portion of the Delay Buffer includes its previous contents (DB_0, ...DB_{r-1}), the message to be output (DB_r, ...$DB_{r+(s-1)}$) and an extra position for delaying new input (DB_{r+s}). In effect this means that the FIFO delay queue has been expanded by s+1 positions and has had an s-bit message pushed into it. New incoming messages will enter the delay queue through DB_{r+s} and will thus be transmitted onto the loop following the output message. Thus an output message has been inserted onto the loop between two incoming messages.

It may be the case, however, that when the output message in the Output Buffer is ready to be transferred into the Delay Buffer, insufficient space is available in the inactive portion of the Delay Buffer (N-r<s+1) for the transfer to take place. If that is true, then no output message can be transmitted by the ring interface until sufficient space becomes available in the Delay Buffer. Space is made available, however, only between incoming messages when the size of the FIFO delay queue can be reduced. Furthermore, the size of the delay queue is increased only when transmitting an output message onto the loop. Thus insufficient space for output message transmission can only happen when a node tries to output too many messages, of too long a length, over too short a time for the loop to handle. Under those circumstances, it seems entirely justifiable to force the offending node to wait for the system load to decrease before transmitting any more messages, especially since other nodes with shorter delay queues are still allowed to transmit their messages.

It should be apparent from the above discussion that the capacity of the Delay Buffer must be at least as large as that of the Output Buffer. Furthermore, it should be obvious that the greater the Delay Buffer capacity, the less likely is the chance that output message transmission will be blocked because of insufficient space being available. Thus, certain nodes can in effect have higher output priority than others, simply by increasing the sizes of their Delay Buffers. The size of the Delay Buffer is thus a design parameter which can be adjusted differently for each node to suit individual output requirements. However, it should also be observed that for messages already on the loop, transmission time is minimized by having the sum of all possible delays be as small as possible. Thus, optimization of overall system performance requires that Delay Buffer sizes be neither too excessive nor too small.

The complete operation of the model just developed is summarized in the flowchart of Fig. 3. Hopefully, the model has demonstrated (at least in concept) a plausible method by which variable-length message frames can be transmitted in a distributed loop system. In any case, it is now time to consider possible hardware implementations of the model.

V. Hardware Implementations

A direct translation of the model into hardware might look something like the device depicted by Fig.4. The essential feature of this implementation is a variable-length shift register which serves as the model's Delay Buffer. Only the left-most r bits of this shift register are activated (shifted), corresponding to the r-bit FIFO delay queue of the model. A 1-to-N decoder (Input Selector) is provided to gate the incoming message bits from the Input Line into the right-most active position of the shift register. The Input Selector thus defines the logical partition of the shift register into active and inactive portions. A similar K-to-N decoder (Output Selector) performs the more complicated operation of parallel-transferring s bits of an output message from the Output Buffer into the inactive portion of the shift register.

While the hardware illustrated by Fig. 4 could certainly be constructed, it would likely prove to be too expensive for a ring interface transmitter. The variable-length shift register is essential in any implementation for a Delay Buffer and thus cannot be eliminated. The same is true for some kind of Input Selector to direct incoming data into the shift register. The most expensive hardware component, however, seems to be the Output Selector, since its function is so complicated.

The Output Selector can be totally eliminated by separating the Delay Buffer into two distinct registers. The variable-length shift register will then be used exclusively as a FIFO message delay queue, while a second shift register will function for the inactive portion of the Delay Buffer. A further saving of hardware is possible by letting this second shift register also serve as the Output Buffer. If that is done, output messages can be assembled in the Output Buffer as before and then directly shifted out onto the loop.

This latter, more economical hardware implementation of the model is shown in Fig. 5. Now there are two shift registers which may be connected to the Output Line, the Delay Buffer and the Output Buffer, but only one or the other may be activated (shifted) at a time. If no output transmission is in progress, the Delay Buffer will be shifted out onto the loop. During receipt of incoming messages, the length of this shift register will remain constant; between messages when no valid information is being received, the length of the Delay Buffer will be decreased by one at each time period (until it is completely empty or a new incoming message arrives).

When an output message is assembled in the Output Buffer and is ready for transmission onto the loop, the control circuitry of the ring interface scans the Output Line, watching for the end of the current incoming message (if any is being received). When the end is detected and if sufficient space is available in the Delay Buffer (equal to the output message length), then the Delay Buffer is deactivated and the Output Buffer is activated, thus shifting the output message onto the loop. While its transmission is in progress, incoming message bits are saved in the Delay Buffer by increasing its length by one at each time

period; between incoming messages the length of the Delay Buffer remains constant. When the last bit of the output message has been shifted onto the loop, the Output Buffer is deactivated and the Delay Buffer is reactivated to operate again as previously described.

By restricting or imposing certain limitations on the conceptual model developed in Section IV, further hardware savings are possible. It is likely that some of these limited implementations might be much more economical and just as useful for certain applications. There is no reason that several implementations of varying degrees of sophistication could not be used in the same loop network. For further information in this regard, the reader is referred to the authors' previous work [18].

VI. Effects on Message Transmission

Since analytic and simulation studies of this proposed loop system are still in progress, no precise statements can be made at this time concerning performance. On the other hand, a number of performance factors can be evaluated, and certain improvements can be plausibly argued. Consider the following factors concerning message transmission in the proposed system:

1) The queueing delay a ready message in the Output Buffer may experience before being transmitted onto the loop depends on its length and the amount of data already stored in the Delay Buffer. If sufficient delay space is available, transmission can begin at the end of the current incoming message. Otherwise, the queueing delay will be extended for whatever time is required to make space in the Delay Buffer. This time obviously depends in a complicated manner on total system traffic and loop utilization.

2) The transmission delay a message will experience in getting from its source to its destination node once it is on the loop is simply the sum of the delays imposed by each intermediate node. Each of these delays depends solely on the amount of data already in each Delay Buffer when the message arrives there. The transmission delay will obviously increase rapidly with increasing system load but for a worst-case upper limit can be no greater than the total capacity of all Delay Buffers.

3) For moderate to heavy traffic loads, nodes that transmit messages only infrequently should experience little queueing delay, while nodes that desire to transmit almost constantly will undoubtedly be slowed down because of limited Delay Buffer space. Thus heavy, frequent users of the loop are restricted while less frequent users still receive reasonably prompt service. Of course, the transmission delays will be about the same for all nodes.

In comparing the proposed loop system with previous fixed-size and/or variable-length loop systems, certain performance improvements and advantages can be recognized:

1) Variable-length message frames can be transmitted directly.

2) Simultaneous message transmission is allowed by independently acting nodes.

3) Nearly immediate access to the loop is given to any node with sufficient space in its Delay Buffer, regardless of the traffic on the loop.

4) Subsequent loop access is granted by the hardware in a manner determined by individual past accesses and current traffic.

5) Automatic access regulation tends to favor infrequent or light users and to prevent domination of the loop by one or several nodes.

6) Loop utilization and message transmission are improved due to less wasted frame space and simultaneous message transmissions.

VII. Conclusion

As previously mentioned, verification of these performance claims is now underway. Research is proceeding in the areas of analytic modeling and simulation. An analytic model of the queueing and transmission delays in the proposed system is being sought as one goal. In conjunction with that work, a simulation study of loop networks is being performed. These studies are expected to yield considerable insight as to the actual operation and performance of the proposed system. The results of this research will be made available in a subsequent report [18].

It is felt that distributed loop systems with variable-length message transmission facilities could play an important role in areas as diverse as credit-card verification systems, mini-computer networks, and telephone switching systems. It is hoped that the ideas presented here may be of some interest in the future development of these and other application areas.

Acknowledgements

The authors wish to thank Drs. Marshall C. Yovits, Gerald J. Lazorick, and David K. Hsiao for their support and constant encouragement during the period of this research.

References

1. Anderson, Richard R., Jeremiah F. Hayes and David N. Sherman. "Simulated Performance of a Ring-Switched Computer Network," IEEE Trans. on Comm., COM-20, 3 (June 1972), 576-591.

2. Chu, Wesley W. "Some Recent Advances in Computer Communications," Proc. First USA-JAPAN Computer Conference, Tokyo, Japan, October 1972, pp. 514-519.

3. Coker, C.H. "An Experimental Interconnection of Computers through a Loop Transmission System," Bell Syst. Tech. Jour., LI (Jul./Aug. 1972), 1167-1175.

4. Farber, David J., et al. "The Distributed Computer System," Proc. COMPCON 73, Feb. 1973, pp. 31-34.

5. _____and Kenneth C. Larson. "The System Architecture of the Distributed Computer System — The Communication System," Proc. Symp. on Computer-Communications Networks and Teletraffic, Polytechnic Institute of Brooklyn, Apr. 1972, pp. 21-27.

6. Farmer, W.D., and E.E. Newhall. "An Experimental Distributed Switching System to Handle Bursty Computer Traffic," Proc. ACM Symp. on Data Communications, Pine Mountain, Ga., Oct. 1969, pp. 1-33.

7. Hafner, E.R., Z. Nenadal and M. Tschanz. "A Digital Loop Communication System," IEEE Trans. on Comm., COM-22, 6 (June 1974), 877-881.

8. Hassing, Thomas E., et al. "A Loop Network for General Purpose Data Communications in a Heterogeneous World," Proc. DATACOMM 73, St. Petersburg, Fla., Nov. 1973, pp. 88-96.

9. Hayes, Jeremiah F. "Modeling an Experimental Computer Communication Network," Proc. DATACOMM 73, St. Petersburg, Fla., Nov. 1973, pp. 4-11.

10. _____ and David N. Sherman. "Traffic Analysis of a Ring-Switched Data Transmission System," Bell Syst. Tech. Jour., L, 9 (Nov. 1971), 2947-2978.

11. Kaye, A.R. "Analysis of a Distributed Control Loop for Data Transmission," Proc. Symp. on Computer-Communications Networks and Teletraffic, Polytechnic Institute of Brooklyn, Apr. 1972, pp. 47-58.

12. Konheim, Alan G. "Service Epochs in a Loop System," Proc. Symp. on Computer-Communications Networks and

Teletraffic, Polytechnic Institute of Brooklyn, Apr. 1972, pp. 125-143.

13. Kropfl, W.J. "An Experimental Data Block Switching System," Bell Syst. Tech. Jour., LI, 3 (Jul./Aug. 1972), 1147-1165.

14. Manning, Eric G., and R.W. Peebles. "A Homogeneous Network for Data Sharing — Communications," Technical Report CCNG-E-12, Computer Communications Network Group, University of Waterloo, Mar. 1974.

15. Peebles, R.W., D.L. Buck and E.G. Manning. "Simulation Studies of a Homogeneous Computer Network for Data Sharing," Technical Report CCNG-E-19, Computer Communications Network Group, University of Waterloo, Mar. 1974.

16. Pierce, John R. "How Far Can Data Loops Go?" IEEE Trans. on Comm., COM-20, 3 (June 1972), 527-530.

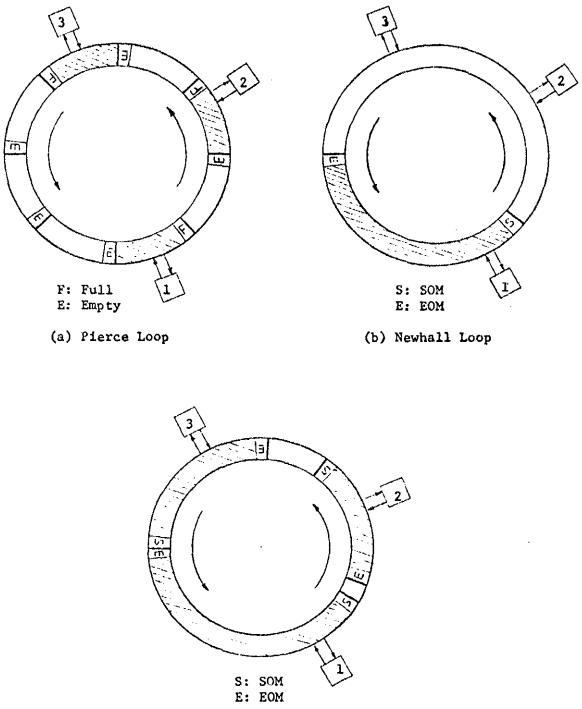
17. _____. "Network for Block Switching of Data," Bell Syst. Tech. Jour., LI, 3 (Jul./Aug. 1972), 1133-1143.

18. Reames, Cecil C., and Ming T. Liu. "Variable-Length Message Transmission for Distributed Loop Computer Networks, Part I," Technical Report OSU-CISRC-74-2, Dept. of Computer and Information Science, The Ohio State University, June 1974. (Part II is in preparation).

19. West, Lynn P. "Loop Transmission Control Structures," IEEE Trans. on Comm., COM-20, 3 (June 1972), 531-539.

20. Yuen, M.L., E.E. Newhall, et al. "Traffic Flow in a Distributed Loop Switching System," Proc. Symp. on Computer-Communications Networks and Teletraffic, Polytechnic Institute of Brooklyn, Apr. 1972, pp. 29-46.

21. Zafiropulo, Pitro. "Performance Evaluation of Reliability Improvement Techniques for Single-Loop Communications Systems," IEEE Trans. on Comm., COM-22, 6 (June 1974), 742-751.



(a) Pierce Loop

(b) Newhall Loop

(c) Proposed Loop

Figure 1. Loop Structures

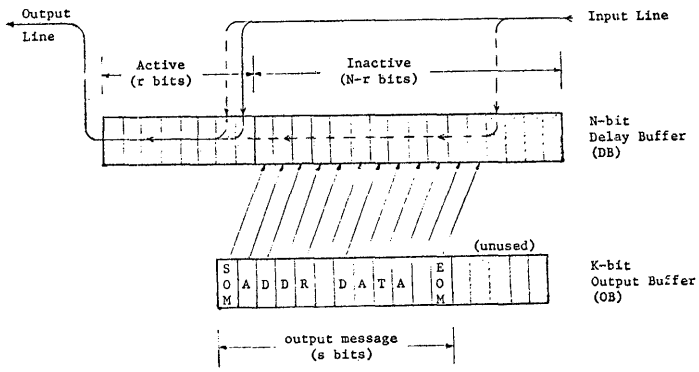


Figure 2. Conceptual Model of Ring Interface Transmitter

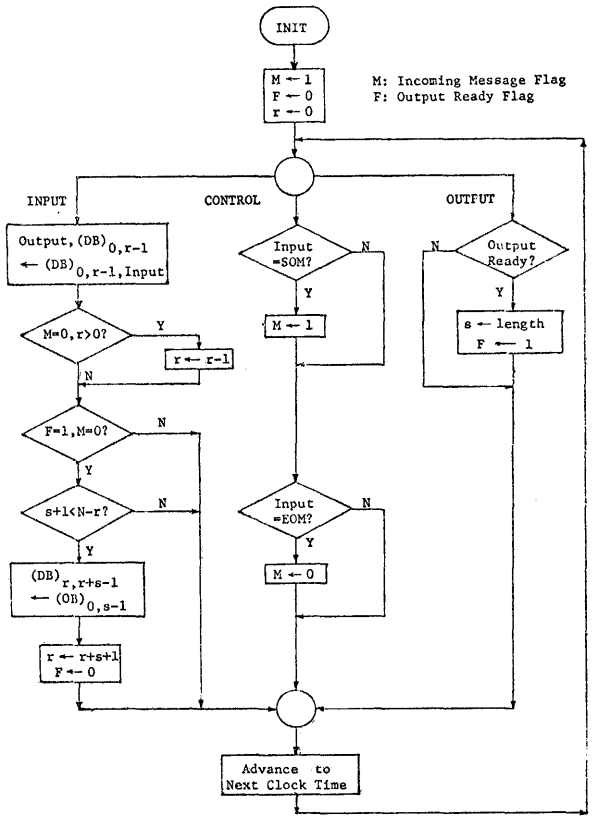


Figure 3. Cycle of Operations for Conceptual Model

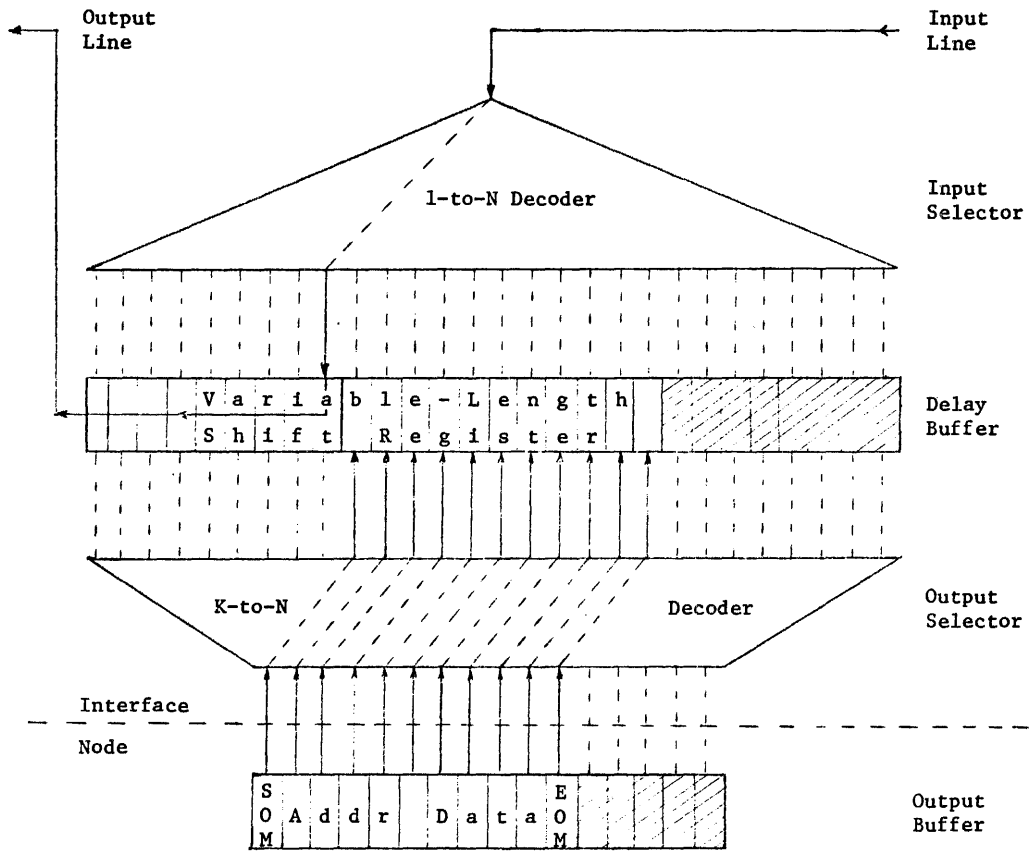


Figure 4. Direct Translation of Model to Hardware

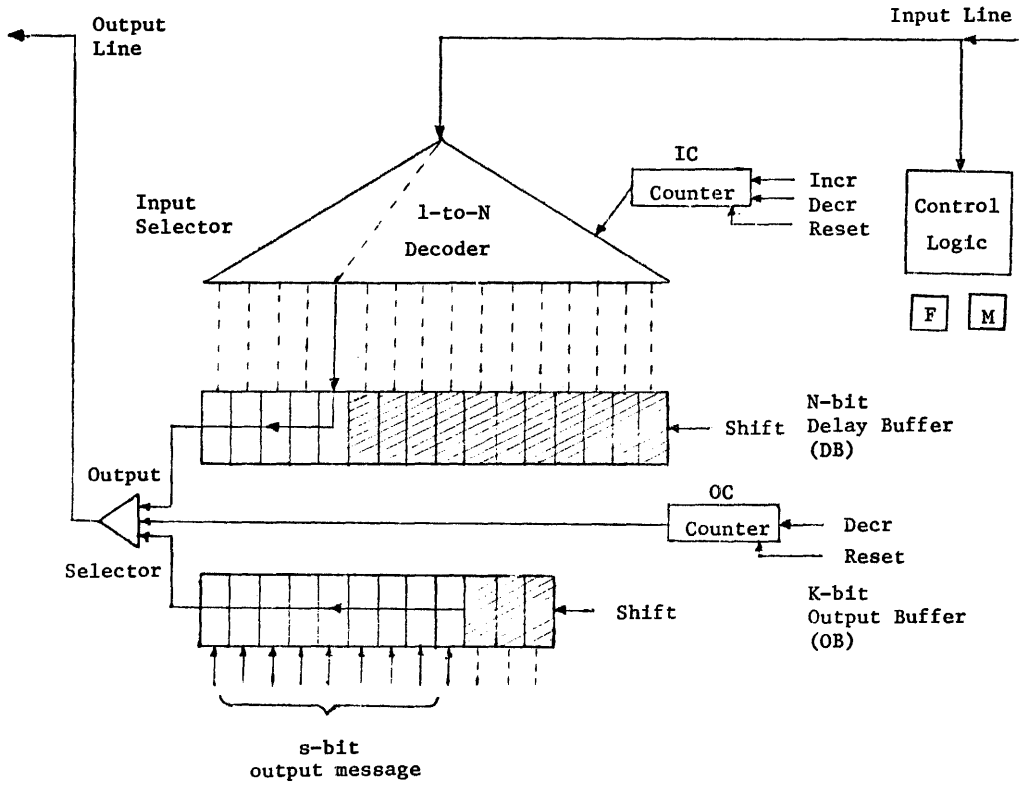


Figure 5. A More Economical Hardware Implementation of Model

THE ARCHITECTURE OF THE PICTURE SYSTEM

James F. Callan
Evans & Sutherland Computer Corporation
3 Research Road, Salt Lake City, Utah 84112

Abstract

THE PICTURE SYSTEM is a self-contained, stand-alone line drawing system for presenting dynamically moving pictures of two- or three-dimensional objects. This highly interactive system can display smoothly changing true perspective views of 3-D objects in real time. Objects can be rotating, translating, and changing in scale. Individual sub-objects can assume independent or compound motion. Lines or parts of lines outside the chosen viewing window are clipped. All these transformations are performed digitally to avoid inaccuracy and range limitations.

In order to perform these calculations fast enough to show smooth motion, support high-level interaction, and also sustain flicker-free pictures of 11000 line segments or more, a new graphics system architecture has been developed. This paper will present each of the hardware components of the system and discuss what each contributes to the production of pictures.

THE PICTURE SYSTEM represents a significant new graphics system architecture in several respects. This paper will describe the system including data flow, the functions of each major component, and some of the advantages that accrue from the approaches taken.

Figure 1. shows a block diagram of THE PICTURE SYSTEM.

The Picture Controller

Evans & Sutherland selected the DEC PDP-11 as the standard controller for THE PICTURE SYSTEM because of its popularity and because its architecture is well suited to supporting high-performance graphics systems. THE PICTURE SYSTEM requires no modifications to the PDP-11 at all. Therefore, it can be equipped with any standard peripherals, utilize any DEC supported system software, and can be put to general-purpose use when not executing a graphics program.

The UNIBUS which serves as the universal link connecting the PDP-11 processor, memory, and peripherals also serves as the connecting link to the controlling registers and graphic peripherals of THE PICTURE SYSTEM.

Prior to issuing any drawing commands, the Picture Controller passes parameters to the Picture Processor to indicate how subsequent coordinate data is to be interpreted and what transformations are to be made to that data. The data interpretation parameters indicate connectivity (i.e., how points are to be connected), and point of origin (i.e., whether the coordinate data is absolute or relative to preceding coordinates). It further states whether subsequent coordinate data is two-dimensional, three-dimensional, or homogeneous to provide a much larger effective dynamic range than is normally available with 16-bit words.

After all these modes have been set up, the Picture Controller passes the actual coordinate data from its memory to the Picture Processor via a DMA interface. The program contains DMA setups for each block of coordinate data that can be sent without mode changes.

This process continues for the data describing all objects to be displayed, at which point the program waits for a clock interrupt indicating that the pre-specified amount of time allocated for a frame has elapsed. The program then jumps back to draw the next frame perhaps with a slightly different set of transformation parameters, to produce the illusion of smooth motion in the picture.

When there is a great deal of data to be drawn a mechanism is provided for organizing this data efficiently so that the entire frame can be passed to the Picture Processor within the time desired. This mechanism entails arranging into tables coordinate data which have identical mode settings so that the issuing of data will not have to be interrupted to reset any modes. To make this feasible on a large scale, the modes are made very general.

Thus it is possible to issue a command to draw an entire table of:

a. connected lines from the current position;

b. connected lines from a new position;

c. unconnected lines from the current position;

d. unconnected lines from a new position;

e. dots; or

f. characters.

Likewise it is possible to specify that the coordinates of points in a table are:

a. all absolute;

b. all relative to their predecessor; or

c. all relative except the first, which is absolute.

Since coordinate data is being fetched from PDP-11 memory by the DMA, it is possible for the CPU to execute code simultaneously. This overlapping of operations substantially reduces the time required to process a frame.

The display program has access to a 120 cycle clock (which is actually in the Picture Processor) for setting refresh and update rates. This 120 cycle clock offers more choices than the usual 60 cycle clock, including the optimal refresh rate of 40 hz.

The Picture Processor

The Picture Processor receives data sent by the Picture Controller in the form of two- or three-dimensional line endpoint coordinates. It operates on this data a point at a time, based on parameters passed previously by the Picture Controller, and deposits the processed data in the Refresh Buffer.

The Picture Processor starts by converting relative coordinate data to absolute. Conversion at this point, prior to any coordinate transformations, avoids accumulating the roundoff errors that may be introduced by transformation.

The Picture Processor then transforms the data by any pre-specified combination of translations, rotations, and changes in scale. Translation may be in any direction in three space; rotation may occur about any axis in three space; and scaling may be in any dimension in three space. These transformations are imparted by expressing the coordinates of each point as a four-component vector (by appending one or two pre-specified coordinates, if necessary, to those emanating from memory), expressing the desired transformation as a four-by-four matrix, and multiplying the two together to form a four-component vector representing the transformed point. The homogeneous nature of the representation introduces much flexibility and permits much larger transformation and scale values than would otherwise be possible.

Transformed data is next "clipped"; i.e., it is compared with a pre-specified viewing window, and lines or parts of lines outside this window are eliminated. Clipping is the most general way to perform windowing, and allows zooming into a displayed structure to any desired magnification.

In two-dimensions, the viewing window is a rectangular region in the drawing space. In three-dimensions it can be either a cubic region (for orthographic views) or a section of a pyramid whose apex is at the eye point (for perspective views). Clipping is performed with respect to all six surfaces of the viewing window.

At this point three-dimensional coordinates are converted to two dimensions by computing perspective or, if desired, orthographic views. An intensity value proportional to depth is also computed here for use in depth-cueing if desired.

The final stage in the Picture Processor's digital processing is a linear mapping of points from the objects' coordinate system into that of the Picture Display. The mapping is from the viewing window onto a program-specified region of the Picture Display called a viewport. The viewport may also be thought of as six-sided, since in addition to the four geometric boundaries there are intensity maximum and minimum values which may be specified as well. When depth-cueing is used, lines at the front clipping plane of the 3D viewing window are assigned the maximum intensity and lines at back are assigned the minimum. Constant intensity is specified very simply by making the maximum and minimum the same.

The advantages of program-specified viewports are that they free the programmer from having to be concerned with the coordinate system of the output device in preparing his data, and that they provide a convenient way to show several views or pictures on the same output device.

The operations discussed above are performed serially on each point in the data base. That is, the coordinates of a point enter the Picture Processor, are converted to absolute homogeneous coordinates, multiplied by the transformation matrix, clipped to the window, put in depth-cued perspective, mapped into viewport coordinates, and deposited in the Refresh Buffer. A new point then enters the Picture Processor and is treated by the same operations. The total time required for all these processes ranges from 10 usec to 36 usec per point, depending on the complexity of the clipping operation, and averages about 20 usec. Thus the Picture Processor can process about 3300 points in 1/15 second, the largest frame time which supports smooth dynamics.

An important related feature of the Picture Processor is its ability to concatenate transformations. The matrices representing two or more simple transformations can be multiplied together to form a matrix representing a compound transformation, which is left in the Picture Processor for transforming points.

For temporary storage of matrices, a four-deep push-down stack is provided in the Picture Processor. Thus if a certain transformation must be compounded with another but will be needed subsequently, it may be pushed onto the stack before compounding and later popped off the stack for further usage. There is provision for overflowing the stack into PDP-11 memory in case a stack depth of four is not adequate for some program. The overflow has the same properties as the main transformation stack except that transfers to and from it are slower.

Data computed by the Picture Processor is normally deposited only in the Refresh Buffer, but can be read back into Picture Controller memory as well. This capability is useful for obtaining hard copy output and also for performing additional processing on pictorial output in software, such as slow time shading or hidden line removal. Results are available after the transformation stage, after the clipping stage, and after the viewport mapping stage. The viewport boundaries may exceed those of the Picture Display when output is to be returned to memory so that full 16-bit accuracy is maintained for cre-

ation of high-quality hard copy.

A final feature of the Picture Processor is its "hit test" capability, which allows the system to detect whether any part of a given picture element is within a program-specified region in the drawing space. Hit testing is used for implementing the pointing function with a data tablet, eliminating the need for a light pen.

The Refresh Buffer

Coordinate data output by the Picture Processor, still in digital form, is ready to be drawn on the Picture Display. It is not drawn immediately, however, but is instead deposited in a memory called the Refresh Buffer. Once deposited in the buffer, the data can be used to refresh the Picture Display any number of times before a new frame is ready for display. Therein lies the value of the Refresh Buffer and the reason why it is in the system: picture refresh rate and picture update rate need not be identical, so that each may be independently optimized. The refresh rate may be specified as any of 16 submultiples of 120 hz - usually 60, 40, or 30 hz. The generation of new frames need not proceed so fast, and so the update rate is usually lower than the refresh rate, typically 15 or 20 hz.

Because of the Refresh Buffer, the Picture Generator never has to sit idle while its input is being prepared. Therefore it can display many more lines in a given time than would be possible otherwise.

The buffer contains 8K 36-bit words, and can be optionally expanded to 16K 36-bit words. Each word can contain any of the following data types:

a. a point to be moved to with the beam off

b. a point to be moved to with the beam on

c. a point where a dot is to appear

d. three character codes

e. mode settings for the Picture Generator or Character Generator.

In the first three types, the point is defined by a 12-bit "X" field, a 12-bit "Y" field, and an 8-bit intensity field. In the character words, there are three 8-bit character codes. In the last type, called "status", there are bits for indicating scope selection, dashed mode, blink mode, character height and width, and character orientation. These bits set up modes which remain in effect until another status word is encountered.

Data may be single-buffered, in which case the Refresh Buffer contains one frame of data which is overwritten by the next frame, or double-buffered, which entails splitting the Refresh Buffer into two areas so that a frame in one area remains intact for refresh purposes until a new frame is completed in the other. The roles of the two areas are then reversed and a third frame is begun. Double-buffering is recommended when frames are small enough to fit in half of the Refresh Buffer, because single-buffered pictures can show any of a number of unpleasant effects that result from trying to display data that is simultaneously being updated.

A four-word area of the Refresh Buffer (two four-word areas in double-buffer mode) is devoted to a cursor so that a highly dynamic cursor can be maintained even if the update rate of the rest of the picture is slow.

Picture Generator and Picture Display

These two PICTURE SYSTEM components are discussed together since their specifications are so intertwined. The Picture Generator converts line endpoint and intensity coordinates, expressed digitally, to analog voltages for the Picture Display. The Picture Display contains a cathode-ray tube (CRT) whose electron beam sweeps out straight lines at any angle based on the analog voltages supplied by the Picture Generator. The beam briefly illuminates the phosphor coating of the CRT, which if repeated frequently enough results in a steady visible picture.

Lines may be solid or dashed, and may be made steady or blinking. Constant intensity of picture elements may be chosen from 256 levels. Lines are drawn at a constant rate which assures uniform brightness for the chosen intensity level. Depth cueing allows the intensity of lines to vary continuously with depth. In order to present a uniform variation in brightness, the intensity control of the Picture Display treats the Z coordinate data as the logarithm of the intensity to be displayed. The contrast control of the Picture Display is completely independent of the intensity control.

The following display rate examples assume a 30 hz refresh rate:

a. 11,000 half inch lines connected end-to-end

b. 1600 ten inch lines connected end-to-end

c. 6600 uniformly distributed dots

The Picture Display has a 21" diagonal rectangular CRT with a quality viewing area of 10" x 10". Line endpoints may be centered on any point in a raster of 4096 x 4096 addressable points on the display surface. Line width and spot size do not exceed .020". Line endpoints match to within .020".

The standard phosphor of the Picture Display is P4, a white phosphor which fades very quickly after illumination so that no trail or smear is left by moving picture elements.

The Character Generator

When character words are read out of the Refresh Buffer, the Character Generator unpacks them into codes which access a read-

only memory containing character stroking data. The strokes are read out of the read-only memory one by one, multiplied by a pre-specified sizing parameter, and drawn by the Picture Generator on the Picture Display.

The read-only memory contains the 96-character extended ASCII character set, including upper and lower case alphabetics, ten numerics, and thirty-four special characters.

There are eight character sizes available under program control ranging from .07" high in increments of .07" to .56" high on a 10" x 10" viewing area. The character width is also under program control with eight different widths selectable for each size. Characters may be displayed horizontally or in a 90° counter-clockwise orientation. Inter-character spacing is handled automatically by the Character Generator.

Such flexibilities as clipping, rotation, and continuous sizing cannot be imparted to characters generated by the Character Generator. When such operations are desired, the Picture Controller should produce characters just like other picture elements.

To place the Character Generator in front of the Picture Processor would severely and unnecessarily impact the picture update rate, since each individual character stroke would have to be transformed, clipped, etc.

The Tablet

The tablet serves as the standard, general-purpose graphic input device in THE PICTURE SYSTEM. Associated with the tablet is a pen whose coordinates are read by the Picture Controller. Normally a cursor is drawn on the Picture Display to indicate the position of the pen on the tablet.

The tablet is inherently well-suited to entry of precise positional information. With the aid of the Picture Processor's "hit test" capability discussed earlier, it can also be used conveniently for pointing to picture elements, eliminating the need for a light pen. An extremely versatile device, the tablet can also be used for command selection, entry of rate information, and even entry of alpha-numerics.

Options

The components described above comprise the so-called Standard Configuration of THE PICTURE SYSTEM. A number of options of a peripheral nature are also available such as keyboards, function switches, hard copy units and so forth. These units are interfaced directly to the UNIBUS. Additional Picture Displays and an expansion of the Refresh Buffer are also optional.

It is significant that there are no options within the central processing path of the system. All the processing features which E&S believes a high-performance system should have are standard in the system, not optional. The reason is that to make such a feature unpluggable is to make it expensive. It would have to have interfaces at either end to fit into the system; it would have to be separately packaged; and a great deal of care would have to be taken to make sure the system worked gracefully without it, hardware-wise, software-wise, diagnostic-wise and documentation-wise. Standardization is perhaps the most important reason why the system can be marketed for a reasonable price.

Conclusion

THE PICTURE SYSTEM's architecture represents a distillation of technology incorporated into earlier digital graphics systems as well as some new approaches. Widespread user acceptance indicates that the right set of features are included, and the relatively low system cost indicates that they are packaged in an efficient manner.

References

1. "Line Drawing System Model 1 System Reference Manual", Evans & Sutherland Computer Corporation, Salt Lake City, Utah January 1970.

2. "Line Drawing System Model 2 System Reference Manual", Evans & Sutherland Computer Corporation, Salt Lake City, Utah August 1971.

3. Callan, J. F., "Key Decisions in Designing THE PICTURE SYSTEM", SID Journal, Vol. II, No. 1, Jan/Feb 1974.

4. Newman, W. M., and Sproull, R. F., "Principles of Interactive Computer Graphics", McGraw Hill, 1973.

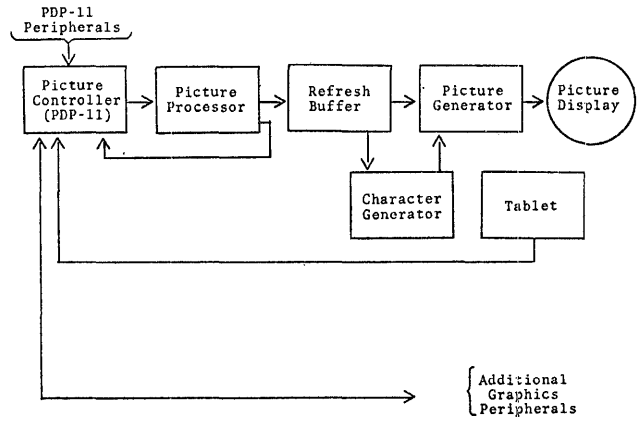


Figure 1.

A FAST DISPLAY-ORIENTED PROCESSOR

John Staudhammer
Jeffrey F. Eastman
James N. England

Department of Electrical Engineering
North Carolina State University
Raleigh, N. C. 27607

Abstract

A detailed study of the basic operations of scan graphics processes led to the design of this general-purpose expandable processor. Full advantage is taken of evolving trends in ECL technology and large-scale integrated circuits. The design allocates hardwired instructions to critical display functions and provides general-purpose flexibility in externally microprogrammable asynchronous processors. The machine uses instruction set partitioning and is optimized to execute relatively short programs operating on large data bases. The processor is thus not limited to display generation but is rather well suited for other real-time tasks, particularly those operating with autonomous processors which can be added as external devices.

1. Background

Over the last three years a considerable amount of work on color scan-display systems was done in the Signal Processing Laboratory at North Carolina State University. The aim of this work was to study the problems in generating realistic-looking images of three-dimensional objects. Some of the findings are reported in [1] together with performance data of a simple display system capable of generating studio quality color TV images in a few seconds. This paper describes the details of the logical next step in this research, the generation of such images in fractions of a second. The heart of this speed-up is a very fast processor which implements the basic operations required for display generation:[1,2]

a. Efficient conditional branching;
b. Fast output;
c. Handling of linked list structures;
d. Very fast special arithmetic operations.

The display requirements are for a modified NTSC television receiver, capable of displaying 512 lines with a spatial resolution of 512 positions on each line. Each picture element may be colored with any combination of 32 levels (5 bits) of red, green, and blue. Hence the color definition is 2^{15}, i.e., over 32,000. Since the entire picture is generated in about 30 ms, each displayed line must be completed in about 50 μs (allowing for beam retrace time). Hence each picture element must be generated within about 100 ns.

This paper describes a small processor capable of full cycle speeds of this order. Details of the central processor design were presented in [10]; this paper gives further particulars and stresses the design of the action processors.

2. Processor

In addition to the fast logical operations outlined above and detailed in [1], a real-time graphics processor must have a large memory in order to store complicated objects. To provide sufficient accuracy and resolution, calculations must be done either in floating-point representation or with large integer words, and they must above all be done fast due to the inherent complexity of visual information, and the constraints of real-time operation.

Examination of the requirements reveals some features which are normally associated with existing commercial computer systems, and some features which are not. Typical of the former are the requirements for large memories, wide words, and linked-list prone architectures. Some systems also offer fast multiply-divide operations suitable for array transformations although these are generally large "number crunchers" like the CDC 6600/7600 which cannot reasonably be dedicated to the graphics process alone. Digital array multipliers and Watkins' Clippers are generally not obtainable in commercial CPU's except microprocessors which are too slow for real-time work. Thus the burden of scan-graphics generation has fostered expensive, dedicated, extensively pipelined and parallel systems constructed solely for image generation. A number of such systems have been built and are currently in operation.[11,12,13] The processor described herein can be adapted to solve the scan-graphics problem in specific, but which retains enough generality to allow re-configuration to solve other classes of problems.

2A. Processor Organization

The basic block diagram of this processor is shown in Figure 1. Note that the machine is a 32-bit multibus device, organized around 32 registers. By addressing the Program Counter (R00) and the Instruction Register (R01) as operational registers efficient Conditional Transfer and Exchange Instructions may be mechanized. To set the conditional flags, the basic CPU is equipped with an Arithmetic/Logical Unit with a flexible instruction set. I/O is controlled through a memory paging system and supervisory memory management is provided by an external host computer.

While 8 K words of memory is barely adequate for performing scan conversion/hidden surface removal, it is certainly not adequate for storing the large data bases associated with graphics images. As such, a 65 K core memory system and memory paging on the fast RAM's is provided. This gives a larger virtual memory from which to operate and increase the power of the system. By including rotating on-line storage and magnetic tapes in a 4-layer memory hierarchy, one could operate in a 32-bit virtual address space which should be more than adequate for most jobs. The "action processors" which will first be constructed are to bear directly on our goal of real-time scan graphics production. A hidden surface removal algorithm[2] will be implemented to drive a raster-scan color display system similar to that already in operation on a slower speed (1.8 μs) minicomputer. It is anticipated that a typical reduction in hidden surface removal calculation time on the order of 40 to 1 will be realized over the present system.

2B. Instruction Set

Tables 1 and 2 indicate the basic processor instruction set.

Table 1

Condition Codes

CCC	Condition Code	CCC	Condition Code
000	UNCOND	100	Overflow
001	.LT.	101	.GE.
010	.GT.	110	.LE.
001	.NE.	111	.EQ.

Table 2

Instruction Format

31 24	23 19	18 17	16 12	11 0
OP-CODE	Source One	I D	Source Two	Displacement

I select	D select	Source two contents
0	0	Memory Location addressed by Source Two register ‡ Displacement
0	1	contents of Source Two register ‡ Displacement
1	0	Memory location addressed by bits 0 - 16 of I.R.
1	1	Sign extended bits 0 - 16 of I.R.

OP CODE	Operation Performed
0 0 0 0 0 C C C	S1 register gets contents of S2 bus on condition
0 0 0 0 1 C C C	S1 register exchanged with S2 register on condition
0 0 0 1 0 C C C	S1 register written into memory at S2I address on condition
0 0 0 1 1 C C C	S1 register exchanged with memory at S2I address on condition
0 0 1 0 E 0 0 0	S2 2's complemented
0 0 1 0 E 0 0 1	S1 Exclusive-OR S2
0 0 1 0 E 0 1 0	S1 OR S2 E = 0: set condition codes
0 0 1 0 E 0 1 1	S1 AND S2
0 0 1 0 E 1 0 0	S1 minus S2 with borrow
0 0 1 0 E 1 0 1	S1 minus S2 E = 1: set condition codes and return result to S1 register
0 0 1 0 E 1 1 0	S1 plus S2 with carry
0 0 1 0 E 1 1 1	S1 plus S2

All other OP-CODES are undefined and may be used by action processors connected to the S1, S2, & IR busses.

2C. Design Details

The entire basic processor is designed with Emitter-Coupled Logic (ECL) and uses medium-scale integration (MSI) extensively.

ECL was chosen over other available logic families because of its high speed capabilities. A typical gate propagation delay for ECL is 2.0 ns compared to 3.0 ns for Schottky Transistor-Transistor Logic (TTL-S), the closest competitor. In more complicated MSI packages, the ECL advantage in speed rises to approximately 2 to 1 over TTL-S. In a processor such as this one with a high speed architecture well suited to MSI implementation, the speed advantage offered by ECL is quite important. ECL offers a further advantage in that gates have complementary outputs greatly reducing the need for separate inverters and their consequent delays. Additionally, the open emitter outputs of the ECL 10,000 series logic chosen for this design enable the use of wire-OR techniques (not available with totem-pole output TTL) resulting in a further decrease in package count. The low output impedance and open emitter design facilitate the use of terminated line interconnections. The lack of internally generated current spikes (as with totem pole TTL) and the constant current design of ECL insure less problems with noise in such a large and dense system as this processor design.

Specifically the ECL 10,000 line of logic has been chosen for use in the processor. The multi-source availability of complex MSI elements, well suited to computer architecture, along with the decreasing cost of plastic packages in this line led to this choice. The slower edge speeds of ECL 10,000 enable the use of wirewrapped interconnection techniques with attendant flexibility.

The design has made full use of such currently available MSI functions as hex latches, counters, data selectors, etc. The operating registers will be implemented with register files to be available soon. As an example of the MSI package and ECL speed advantages combined in the ECL 10,000 line, the ALU of the processor can perform a 32-bit ADD in a typical time of 19 ns. This is achieved using only ten packages (eight 10181 4-bit ALU's and two 10179 lookahead carry generators).

As a final consideration, the availability of ECL to TTL and TTL to ECL translators enables the easy interconnection of TTL, NMOS or CMOS "action processors," memory, or peripheral elements as necessary. Slightly more complex translation circuitry is necessary only with PMOS logic systems.

The circuit realization allows a worst-case instruction execution time of 80 ns. Since instruction fetch will be overlapped, this then becomes the actual full machine cycle time. For comparisons in the succeeding sections a machine cycle time of 100 ns is used.

3. Peripherals

The basic processor efficiently implements the tasks of conditional branching and register transfers. Arithmetic other than integer addition, transcendental function calculation and graphics-specific arithmetic are performed by plug-in processors. It should be noted that the basic architecture is quite general in nature. It is the use of plug-in processors suited to graphics that results, in this instance, in a specialized graphics processor. This section describes

a fast multiplier, a CORDIC processor and a display rotation subsystem. Also included is a brief description of the color display system.

3A. Fast Multiplier Using Cellular Logic[5]

Simple circuits for multiplication deal with one bit of the multiplier at a time. While ECL technology can be used to speed up this operation, a far more fruitful way is to work with more than one bit of the multiplier per iteration. Such a scheme may be viewed as multiplication in radix r where $r = 2^k$, with k equal the number of bits inspected per iteration. Use of a higher radix has the disadvantage of requiring additional multiples of the multiplicand. In the case in which it uses a nonredundant number system, multiplication in radix r requires the multiples 0, 1, 2, ... (r-1) times the multiplicand. In the case in which a redundant number system is adopted, then the multiples stated above may be transformed into the multiples $-r/2, (-r/2-1), \ldots, \bar{1}, 0, 1, \ldots, (r/2-1), r/2$ (for even radices). In this new set of multiples, half of them are merely the complement of the others. For the case of multiplication simultaneously by two digits of the multiplier (r=4), the set {0,1,2,3} may be replaced by the set $\{\bar{2},\bar{1},0,1,2\}$. In the second set we have redundancy since there are more than r (in this case five) digit symbols. The multiple of 3 in the first set is awkward or costly to form, but in the second set all multiples may be formed by shifting and complementation.

The recoding scheme of the multiplier adopted was suggested by Wallace.[6] The recoding actually requires the parallel inspection of three bits of the multiplier. If m_i is the low-order bit of the multiplier, then the bits inspected are m_{i-1}, m_i and m_{i+1}. The bit m_{i+1} is an extra position at the right of the least significant bit of the multiplier. It is initially 0, but in the second iteration it will equal the previous m_{i-1}. The recoding is shown in Table 3. It will accommodate a negative number in two's complement representation.

Table 3

m_{i-1}	m_i	m_{i+1}	Recoded Digit/Multiple Selected
0	1	1	+2
0	1	0	+1
0	0	1	+1
0	0	0	0
1	1	1	0
1	1	0	-1
1	0	1	-1
1	0	0	-2

Using this recoding scheme a cellular logic array can be built which can achieve simultaneous multiplication by two digits of the multiplier. As before, let the two n-bit numbers be M and m where

$$m = m_0\, 2^0 + m_1\, 2^{-1} + m_2\, 2^{-2} + .. \; m_n\, 2^{-n} \quad (m_i = 0, 1)$$

$$M = M_0\, 2^0 + M_1\, 2^{-1} + M_2\, 2^{-2} + .. \; M_n\, 2^{-n} \quad (M_i = 0, 1)$$

and a similar notation is used for M. The steps of the multiplication algorithm are summarized below. The rules for the multiplication apply to digits m_n, m_{n-2}, $\cdots m_i$, m_{i-2}, m_{i-4}, ... 1(n is odd), starting with the least significant bit.

1. If $m_{io} = \bar{m}_{i-1}\, \bar{m}_i\, \bar{m}_{i+1} + m_{i-1}\, m_i\, m_{i+1} = 1$, shift the existing sum of partial products two places to the right.
2. If $m_{i1} = \bar{m}_{i-1}\, m_i\, \bar{m}_{i+1} + \bar{m}_{i-1}\, \bar{m}_i\, m_{i+1} = 1$, add M to the existing sum of partial products and shift the new sum two places to the right.
3. If $m_{i\bar{1}} = m_{i-1}\, m_i\, \bar{m}_{i+1} + m_{i-1}\, \bar{m}_i\, m_{i-1} = 1$, subtract M from the existing sum of partial products and shift the new sum two places to the right.
4. If $m_{i2} = \bar{m}_{i-1}\, m_i\, m_{i+1} = 1$, add 2M to the existing sum of partial products and shift the new sum two places to the right.
5. If $m_{i\bar{2}} = m_{i-1}\, m_i\, m_{i+1} = 1$, subtract 2M from the existing sum of partial products and shift the new sum two places to the right.

No shift is requested after the last operation is carried out. Clearly, if the multiplicand is shifted instead of the partial sum, the same end result is obtained.

This algorithm can be implemented by a cellular logic array having the basic cell shown in Fig. 2. Four-bit arithmetic logic unit used in this basic cell performs three arithmetic operations, which are selected by applying the appropriate binary word to the select inputs ($\bar{S}_0$ through $\bar{S}_3$), as follows:

1. If $\bar{S}_0 = \bar{S}_1 = \bar{S}_2 = \bar{S}_3 = 1$ and $\bar{C}_{i+3}$ of LSB is 1, then arithmetic operation is F = A plus 0;
2. If $\bar{S}_1 = \bar{S}_2 = 1$ and $\bar{C}_{i+3}$ of LSB is 1, but $\bar{S}_0 = \bar{S}_3 = 0$, then arithmetic operation is F = A plus M';
3. If $\bar{S}_0 = \bar{S}_3 = 1$ and $\bar{C}_{i+3}$ of LSB is 1, but $\bar{S}_1 = \bar{S} = 0$, then arithmetic operation is F = A minus M' minus 1.

The selection of these three arithmetic operations is achieved by the inputs m_{i12} and $m_{i\bar{1}\bar{2}}$, where

$$m_{i12} = m_{i-1} + \bar{m}_1 \cdot \bar{m}_{i+1}$$

$$m_{i\bar{1}\bar{2}} = \bar{m}_{i-1} + m_i \cdot m_{i+1} \qquad (1)$$

The arithmetic logic unit operates correctly when one's complement of its inputs are used. The inputs $m_{i1\bar{1}}$ and $m_{i2\bar{2}}$ are used to achieve at the inputs of the arithmetic logic units either $\bar{M}$ or $2\bar{M}$ (the multiplicand shifted one place to the left), where

$$m_{i1\bar{1}} = m_i \oplus m_{i+1}$$

$$m_{i2\bar{2}} = m_i \odot m_{i+1} \qquad (2)$$

In order that the arithmetic logic unit perform operation F = A minus M' instead of F = A minus M' minus 1, it is necessary that the input $\bar{C}_{i+3}$ of the least significant bit be "0" instead of "1". For this purpose it is sufficient to connect the input $m_{i\bar{1}\bar{2}}$ to the $\bar{C}_{i+3}$ of the least significant cell of the row.

An 8-bit by 8-bit multiplier, using ECL gates and ECL arithmetic logic units (ALUs) having the basic cell presented in Fig. 2, is shown in Fig. 3. At the left-hand edge of the array, m_{i12}, $m_{i\bar{1}\bar{2}}$, $m_{i1\bar{1}}$, and $m_{i2\bar{2}}$ (i.e., m_i') are obtained as function of m_{i-1}, m_i, m_{i+1}, according to the logical equations (1) and (2).

Because after recoding the multiplier, multiple selected 2 or $\bar{2}$ is not followed by the same multiple as well as due to the fact that there is an extra bit at the left-hand edge of every row, the problem of overflow does not arise. The problem of alignment of the sign bit does arise because, during shifting, the sign bit is shifted along with the other bits. The correct alignment is obtained by appending two digits before the sign bit of magnitude equal to the sign bit.

The typical multiply time can be calculated from the worst delay path through the multiplier and depends on the length of the operands.

In the case of a cellular logic array for multiplication of two signed binary numbers, both being n-bit numbers including the sign bit, the typical delays are:

4 bit-by-4 bit multiplier	24 ns
8 bit-by-8 bit multiplier	46 ns
16 bit-by-16 bit multiplier	90 ns
32 bit-by-32 bit multiplier	179 ns
(2 full machine cycles)	

For a 32-bit by 32-bit multiplier 144 cells are necessary. These can be implemented with 144 MC10181, 16 Exclusive OR/Exclusive NOR and 608 AND-OR gates. A similar multiplication method is used in a commercially available TTL-S 4 x 2 multiplier cell[14] but of course the speed is much slower than the ECL multiplier described here.

3B. CORDIC Processor

When a solid object, such as a cube, is displayed rotated at some arbitrary angle, the cosine of the angle formed by the normal to the surface and the viewing line is taken for the modification of the actual color. In order to provide this function and to provide general trigonometric and exponential function generation capability to the computer, a CORDIC processor was designed.

The CORDIC[7] principle is used to generate trigonometric, inverse trigonometric and the corresponding hyperbolic functions. Basically the process is capable of generating a wide range of functions by a series of pseudo-rotations of an argument. Roughly speaking, each such step produces another bit in the result and the operations required are only addition/subtraction and shifting. Without going exhaustively into the principles of CORDIC we wish to convey the basic ideas involved. In this process two registers (X and Y) are added to or subtracted from a shifted amount of another as a control parameter ($\emptyset$) is modified by a likewise addition or subtraction of a set of coefficients retrieved sequentially from a stored array. The decision for addition or subtraction is determined by a (greater or equal) or (less than) comparison of a register preset with a predefined constant. The constants may be calculated by using the standard function expansions; these values become the stored constants for the process. The main advantage of CORDIC is its simplicity. We utilize fast ROMs for the storage of the control constants and optimized hardware for speed with only secondary consideration for component count or costs. The basic block diagram is shown in Fig. 4.

Communication with the CORDIC processor is through the I/O bus. Initiation of action occurs when a command is received; a control block will decode this instruction and initialize the processor. The argument E is then received and is latched into the shift control to initialize the shift control counters for their corresponding shifters. This argument is also used to address the K_E table and results with the initial value of X (the addressed value from the K_E table) being latched into the X register. The Y and $\emptyset$ registers are also initialized. The M-type is also set by the first instructions. The M-type tells the processor what basic type of function group (linear, polar, or hyperbolic) it will be processing. Finally the Compare/Output Select is set. Depending on what function is being performed, a particular register (X, Y, or $\emptyset$) will be compared to the compare register to determine whether the adder-subtractor adds or subtracts. The processor will go through 32 iterative cycles, after which it will signal the computer that it is finished and is ready to output the results. The computer must then get the processed results out of the registers. Depending on what type of function is being processed (sine, cosine, tangent, etc.) the desired result or results are in different registers and the CORDIC control must put this data out on the I/O bus when the computer requests it.

To speed up the shift operations involved in the CORDIC process, two levels of selectors are used. The basic operations and their timing are:

1.	Add/subtract	20 ns
2.	Shift	20 ns
3.	Compare	10 ns
4.	Settling Times	35 ns

Hence the CORDIC processor can complete the calculation of a function in about 2.7 μs; an additional 300 ns is allowed for initialization. Thus the basic CORDIC calculation time is 3 μs, or about 30 full machine cycles.

3C. Display Rotator

The three-dimensional rotation of images of vector drawings requires the creation of an [x,y,z] data triplet for every new rotation angle. This is accomplished by keeping an image description in core and performing a matrix multiplication prior to outputting to the display generator. In creating a perspective drawing an additional piece of data is necessary: the distance (w) of the point (after rotation) from the observer. These four data elements form the homogeneous coordinate of a point in 4-space; conversion to 3-space is accomplished by mathematical projection: the value of w is divided out. One of the properties of homogeneous coordinates is that the parameter w may be rotated and scaled along with the x,y,z data by simple matrix multiplication. This peripheral device is designed to accept 16-bit x,y,z and w arguments, perform the 4 x 4 homogeneous coordinate rotation and output the result in 2 μs.

The Display Rotator uses four parallel 16-bit multipliers for speed; each of the multipliers is a shift-and-add device in order to keep costs reasonable. Again extensive use is made of medium-scale integrated ECL devices; a 16-bit adder with a 16-bit shifter of the partial product forms the heart of each of four multipliers. These are implemented with 10181 and 10179 integrated circuits. The multiplicand is selected into the adder with one-out-of-four selectors. The partial product shifter is loaded from the multipliers (z,x,y,w) in sequence as the multiplication and addition progresses. Overflow indication will abort the operation.

Total hardware used is about 200 chips. The bulk of the operation time is ADD/Shift sequences. The

design cycle time is 2 μs or 20 machine cycles. Design details are presented in [9].

3D. Color Display System

The raster scan color display system is described in detail in [1]. The system requires only hue and run-length data from the processor. The reduction in input data rate obtained by this coding will result in the possibility of real time generation of moderately complex color images.

4. Summary

This paper presented the rationale and details of implementation of a very fast processor. Part of the speed is due to asynchronous instruction fetch and execute overlap, a technique not usually employed in small machines; the other major contributor is the use of ECL technology. While some of the peripherals described are specifically designed for graphics processing, the architecture of the machine is well suited for controlling autonomous peripheral processors.

The machine is a significant step towards obtaining realistic-cost interactive color displays and is a test bed for proving out hardware designs. Initial operation of the machine is expected by January 1975.

5. References

1. Eastman, J. F., Staudhammer, J., "Computer Display of Colored Three-Dimensional Objects," This Conference.

2. Eastman, J. F., "A Scan Conversion and Hidden Surface Algorithm for Minicomputers," Report No. 2, Signal Processing Laboratory, North Carolina State University, Raleigh, N. C., September 1973.

3. Anagnostopoulos, P. et al, "Computer Architecture and Instruction Set Partitioning, AFIPS Proc., p. 519, 1973.

4. Reddi, S., Ramamoorthy, C. V., "Sequencing Strategies in Pipeline Computer Systems," AFOSR-TR-72-1952, Electronics Research Center, University of Texas, August 1972.

5. Toma, C. I., "Cellular Logic Array for Multiplication of Signed Binary Numbers," Report No. 6, Signal Processing Laboratory, North Carolina State University, November 1973.

6. Wallace, C. S., "Suggested Design for a Very Fast Multiplier," Dept. of Computer Sci., University of Illinois, Urbana, Rept. 133, February 11, 1963.

7. Walther, J. S., "A Unified Algorithm for Elementary Functions," Spring Joint Computer Conference, 1971.

8. Fischer, J., Holland, J., Knott, J., Mayberry, W., "CORDIC Processor," Report No. 13, Signal Processing Laboratory, North Carolina State University, September 1974.

9. Meares, R., Boersma, R. J., "Matrix Multiplier," Report No. 14, Signal Processing Laboratory, North Carolina State University, September 1974.

10. Eastman, J. F., Wooten, D. R., "A General Purpose, Expandable Processor for Real-Time Graphics," accepted for publication in Computers and Graphics.

11. Evans & Sutherland Computer Corporation, "The Evans & Sutherland Shaded Picture System," Salt Lake City, October 1973.

12. Hanson, D. F., "Outlining and Shading Generation for a Color Television Display," Dept. of Computer Sci., University of Illinois, Urbana, UIUCDCS-R-72-512, June 1972.

13. Rougelot, R. S., "Digitally Computed Images for Visual Simulation," PLUS-183, General Electric Co., August 1969.

14. Fairchild Semiconductor, "93543 4 by 2-Bit Multiplier" in The TTL Applications Book, Mountain View, California, August 1973.

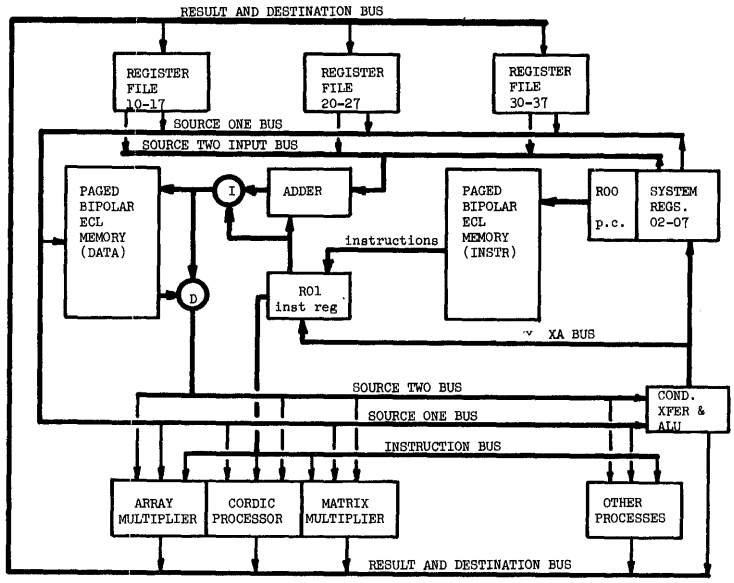


Figure 1

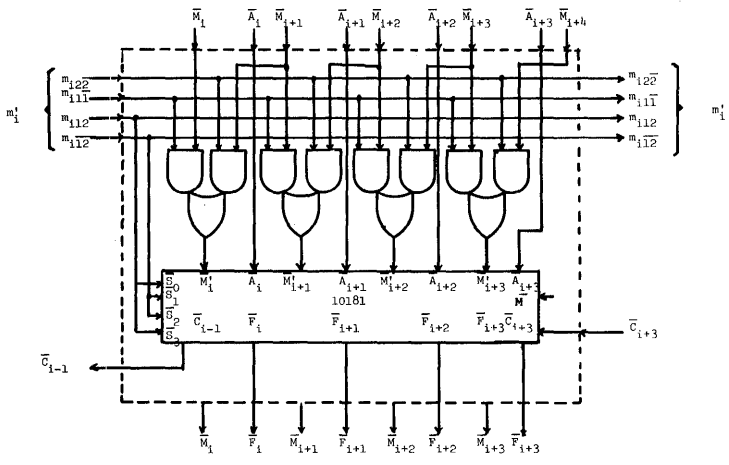
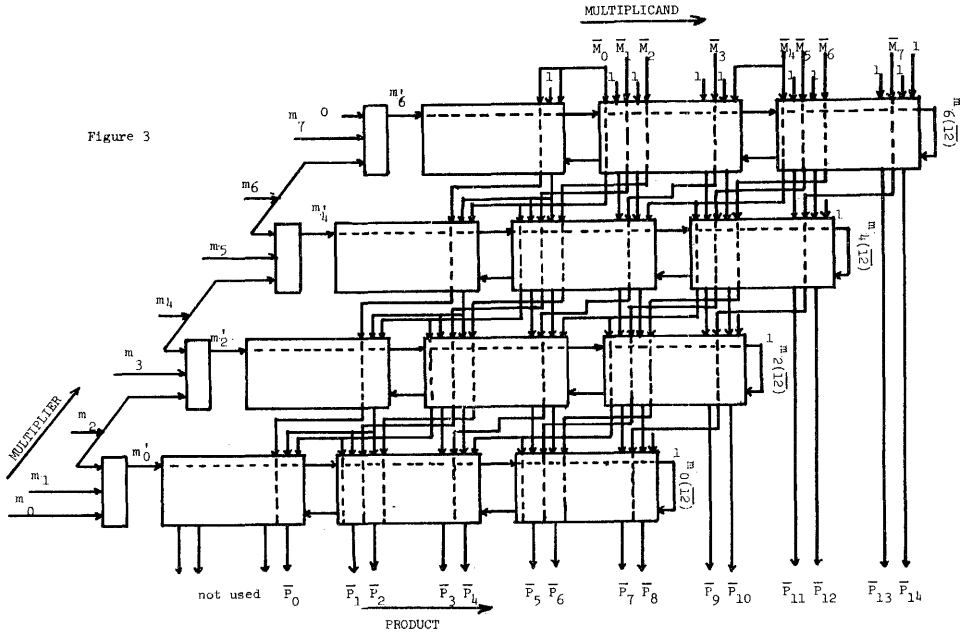


Figure 2

Figure 3



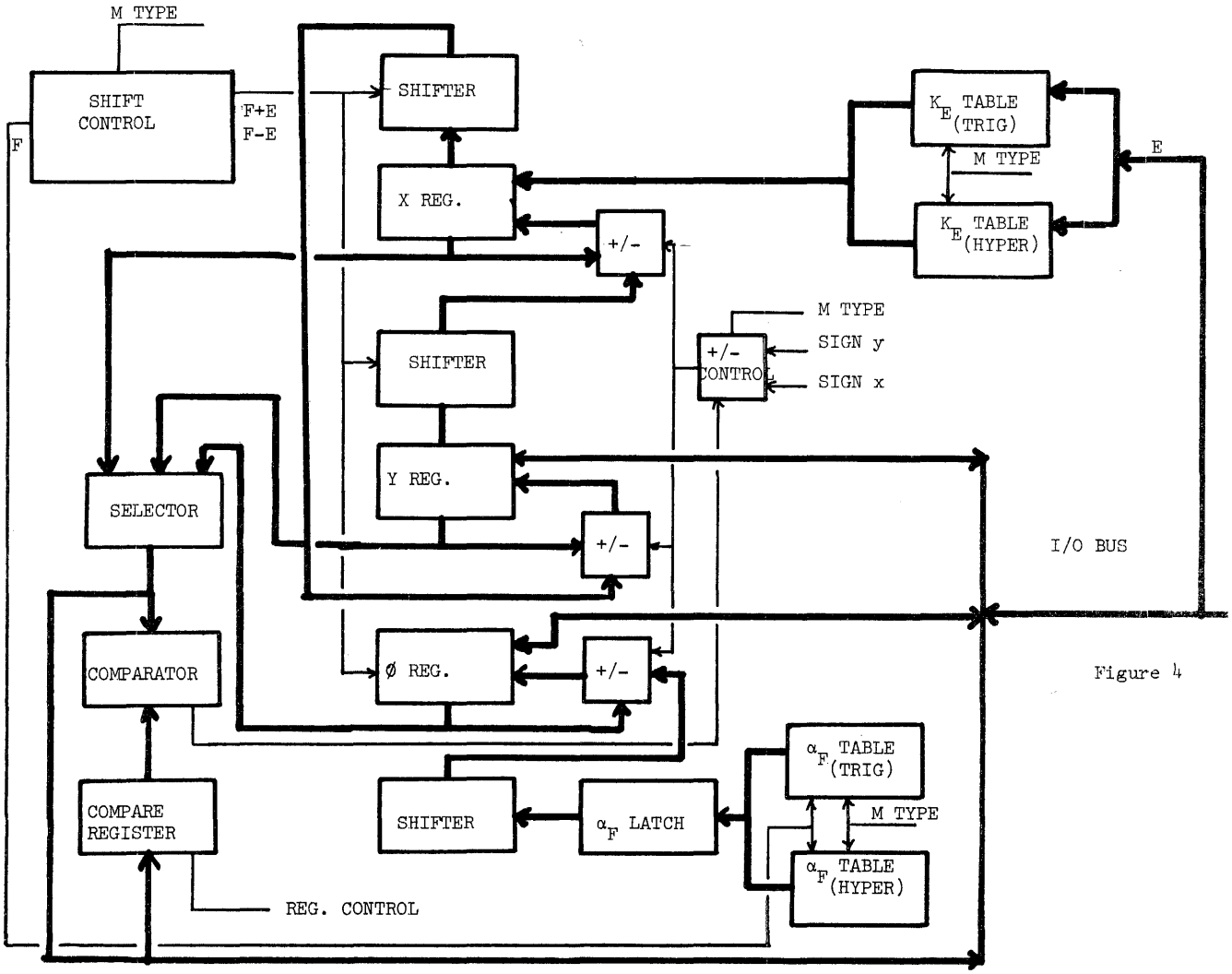


Figure 4

COMPUTER DISPLAY OF COLORED THREE-DIMENSIONAL OBJECTS

Jeffrey F. Eastman
John Staudhammer

Department of Electrical Engineering
North Carolina State University
Raleigh, North Carolina

Abstract

A television-based computer display system is described which uses a minicomputer to calculate scan lines of the display image. A display resolution of 512 lines by 512 positions is achieved with each position definable with 15 bits of color information (over 32,000 colors). The scan conversion process incorporates hidden surface removal, calculation of intersections, surface shading and, optionally, perspective. Object description must be by polygonal planar surfaces; the surfaces can have a solid color and are considered opaque. Object complexity is limited only by main storage size.

The design criteria which motivated the system architecture are discussed along with pertinent details on the particular implementation. The system described effectively mates minicomputer technology with that of the television industry to produce an economical, fast turnaround, color display system for 2-D and 3-D computer graphics. This system relies upon a special scan-line generator and upon a skillful hardware/software trade-off to achieve an economical and realistic color display device.

Performance data and examples of display objects are given.

1.1 Introduction

Cathode ray tube display devices are often used in computer graphics because they allow a large amount of information to be output from the computer at a rate limited only by the deflection speed of the electron beams. These devices may be subdivided into two groups by the manner in which the electron beam is utilized. Early computer graphics displays were primarily vector drawing displays. In vector displays, the electron beam in the CRT is used to draw intensified lines on the screen, and the rate at which a given image can be drawn is its refresh rate. If the refresh rate drops much below 30 Hz, an annoying flicker will be perceived due to the decay in intensity of the phosphors used. Vector displays are suitable for display of line drawings and text, but are generally ill-suited for shading large areas of the screen due to their random scan nature.

In situations where large areas of the screen are to be intensified, such as the display of shaded solid objects, raster scan displays are preferred. In raster displays, the electron beam is scanned across the screen in an orderly left-right, top-bottom manner as in a television receiver. Information is displayed by modulating the intensity (color) of the beam as it scans the CRT front. In computer graphics modeling, the dissimilarity between the image data structure which"looks" like the image, and the raster vectors which are ultimately displayed necessitates an intermediate scan conversion step. For images generated from 3-D computer models, this scan conversion process must include a visible surface determination so that only the surfaces which face the viewer are displayed. This paper reports on the development of an economical, minicomputer-driven color display subsystem which incorporates scan conversion and hidden surface removal, and uses commercial television technology to produce a colored raster display.

Application areas for this subsystem include:
a. Clutter Reduction in Large Displays. Color can be used to tag new or important data for the operator's attention in any display. Hence this display unit becomes the analog of a viewgraph with its attendant advantages. Color can be utilized to convey more information to the operator without significantly increasing the display data rate.
b. Three-Dimensional Displays. Various views of objects can be generated with the hidden parts removed for a realistic colored display of objects. Pairs of such displays can be used with polarizing light filters to display stereo image pairs for truly three-dimensional effect. These views are in full color. Uses for this display mode are in crystallography and molecular modeling as well as in architectural design.
c. Two-Dimensional Displays. While the system described here was designed initially for three-dimensional object display, it has been used for the design of flat patterns. A simple graphics tablet is used for inputting both color information (an "electronic palette") and position information. Application areas include textile and paper patterning.

In this paper the three-dimensional display capabilities are emphasized as the system architecture evolved primarily as a result of these design considerations. The evolution of this display system is discussed followed by pertinent details of the software, engineering details of the display generator and performance data on the display system. A set of photographs is enclosed.

1.2 Background

While the idea of using the computer to generate pictures has been around since the early days of computing, the high initial and operating costs of these systems precluded much activity in the area until the early 1960's. Then, with the advent of transistorized printed circuits and, later, integrated circuits, the trend towards interactive computer graphics became firmly entrenched. Early developments in raster display technology centered at the University of Utah, the University of Illinois, and G.E.'s Electronics Research Center at Syracuse, N.Y. At Utah, Wylie et al[1] and Warnock[2] produced some of the earliest images of shaded, solid objects on a POP-10 computer with a precision slow-scan CRT system and a film chain. By making three successive exposures of a film plate and by using colored filters, they were also able to generate colored images. This system was characterized by the production of excellent high resolution photographs which took several minutes to produce, and several more to develop. Thus it was primarily a batch-oriented system. At about the same time (1967), an interactive color display system was developed at G.E. for NASA to study docking and space maneuvers via simulation.[3,4] This system, while able to produce animated

television pictures in real-time (30 new frames per second) relied entirely on a special drum memory which contained one word for every picture element (pixel) on the screen. At a display resolution of 600 x 600 the system required a dedicated 360 k word, head-per-track drum memory which was very expensive.

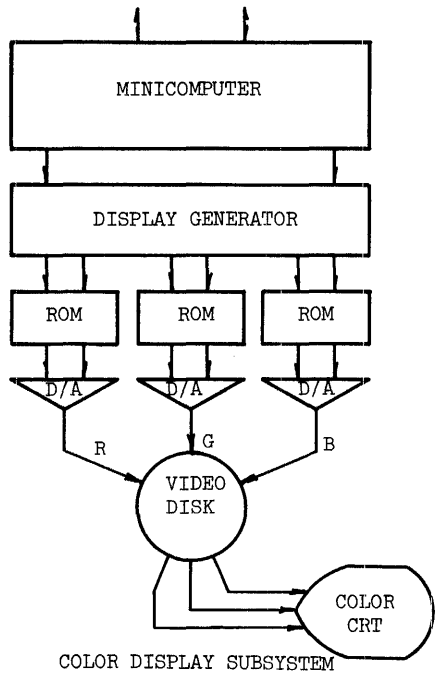
In 1969, Bouknight and Kelly adapted Warnock's[2] scan conversion and hidden surface removal algorithm for use on their CDC 1604 computer at the University of Illinois. Bouknight[5] subsequently wrote his own algorithm, Kelly extended it to include shadows[6], and they started producing shaded halftone images--again using the slow-but-sure time photographic process. At the time, researchers were interested in the idea of real-time halftone graphics but wanted a more inexpensive, generalized system than that at G.E.

Back at Utah, Watkins[7] demonstrated in 1970 that a scan conversion algorithm which utilized the structural similarity of adjacent scan lines (scan line coherence) and a fast recursive clipping process (clipping divider[8]) for surface visibility determinations could generate real-time images if implemented in suitable* hardware. Unfortunately, the FORTRAN-coded simulation of this process still required minutes to produce photographic plates which still required more minutes to develop, and so the graphics user with a modest budget was again stuck with batch processing. The hardware prototype of the Watkins Processor developed by the Evans & Sutherland Computer Corporation (and now marketed[9]) is operational at Utah and is used to generate black and white 256 x 256-resolution previews of the images produced on the slower high resolution system. Its commercial cousin is excellent but very expensive.

With the decline in memory cost in the early 1970's, some commercial companies started marketing scan-graphics displays. These units, such as the Ramtek GX-100, Data Disc 6600, Audin 5214, and Comtal 8000, are basically lower resolution versions of the system developed at G.E.--word per pixel storage and resolution up to 512 x 512 or more at a price. None of these vendors address the generalized 3-D scan conversion problem, although all provide optional 2-D vector generators and character generators with programmable fonts. The main markets for these devices are apparently the picture processing, process control, and management information industries which have little need of generalized 3-D modelling capabilities, yet.

Our advent into the computer graphics area began in 1972 with a National Science Foundation grant to study the feasibility of incorporating off-the-shelf television industry hardware and minicomputers into a color computer graphics system which could be produced at a reasonable cost. We felt (and still do) that the main need of the interactive graphics user is a low cost color display system which produces directly viewable images quickly but not necessarily in real time. To solve the viewable display problem, we utilized a video disk "frame grabber" storage medium and a standard color television monitor. In this system (Fig. 1.1), the image is stored as three black and white television video tracks on the disk. A special display generator described later in this paper changes the image stored on the disk by writing complete scan lines on the disk as analog signals. There are 512 addressable scan lines and each scan line may be specified to a resolution of $\frac{1}{512}$ of its total length. Each track on the disk contains a 32-level encoded video signal so that a total of 32,728 colors may be assigned to each of the 262 k pixels. Due to the interlaced nature of the display used, adjacent scan lines on the display are located approximately 180° from each other on the disk. This results in a latency of 1/60 second between writing opportunities if the new scan lines are calculated monitonically as they are in our system. The color display interface is driven by an 8 k, 1.8 μ sec Varian 620 minicomputer which is able to calculate next line data for a wide range of images in less than 1/60 second using a scan conversion algorithm SCHSA[10] developed by Eastman. Thus a new image may be generated in 8 seconds or more depending on image complexity. SCHSA incorporates the scan line coherence of Watkins' Algorithm, but uses more conventional arithmetic means for visible surface determination. Given a "suitable" hardware implementation[11,12], it too will be a real-time algorithm; however, its main strength lies in its ease of implementation on a general purpose computer and in its efficient use of storage. In execution speed, SCHSA is well matched to its display system and rarely requires more than half a minute to produce a viewable image which may then be photographed if desire

*However, one could say that any algorithm could be real-time given a "suitable" hardware implementation.



COLOR DISPLAY SUBSYSTEM
Figure 1.1

Display Generator

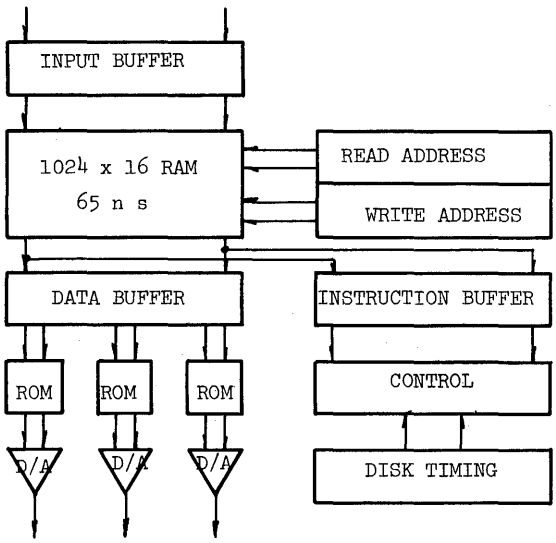
2.1 Design Considerations

The display generator interface connects the output of a general purpose minicomputer to the input of a special purpose video disk. For efficient operation it must be able to receive bursts of data at cpu speeds defining new scan lines to be written. Since there is likely to be a latency period up to 1/30 second before the disk is in the correct position for writing, it must be able to buffer some or all of this data for the latency interval. Also the video disk must be supplied with new scan line data at real-time rates. This requires a very high buffer memory bandwidth, since to attain a 512 point horizontal resolution requires 512 raster points to be output in the 53 μ sec scan line interval. Thus a new data point must be generated every 104 nanoseconds during disk writing operations.

Computer generated images tend to be very simple by television standards. This is because they are generated from relatively simple mathematical models which typically define long segments of a scan line to be a constant color. Efficient utilization of this phenomena results in a tremendous reduction in the I/O bandwidth required of the driving processor if scan lines may be run length encoded. Also the interface should embody an exponential intensity mapping (gamma correction) to account for the non-linearities of the phosphors used on CRTS. Otherwise the apparent light intensity viewed will not be a linear function of the value specified and a large reduction in usable dynamic range will result.

2.2 Implementation

The display generator interface (Fig. 2.1) is basically a high speed first-in-first-out (FIFO) stack with instruction processing ability. The memory used is 65 n sec bipolar RAM, and most of the control logic is Schottky TTL. A READ and WRITE ADDRESS COUNTER each point into the 1024 x 16 bit memory and are used for stack control. Every time a word is written into the memory from the computer, the WRITE counter is incremented, and every time a word is read out of the memory the READ counter is incremented. Whenever both counters are equal, the stack is either full or empty, depending upon the operation (WRITE or READ) which caused the condition. Detection of a counters-equal condition after a write operation means that the stack is full and vice versa. Writing into the stack is done by the connected processor while reading is under control of block definition instructions in the data itself and control logic slaved to the video disk.



DISPLAY GENERATOR
Figure 2.1

A typical scan line definition contains a START XXX instruction, a number of HUE and FILL XXX instructions which define colored segments (length=XXX) on that scan line, a STOP and an EOB instruction. The argument of the START instruction specifies the scan line on the disk which is to be overwritten by the following data up to the EOB instruction. A block counter insures that an entire line block is present in the memory before a disk write operation is initiated.

During a disk write operation, the interface may need continual access to the stack to maintain the 104 n sec per word data rate in a worst case condition. Data transfers from the computer are thus inhibited during this interval. Data output from the stack is in the form of 3 5-bit words which specify the amounts of red, green, and blue to 32 intensity levels. These 5-bit words are then used as the addresses into 3 32 x 8 ROMS which have been programmed to perform the exponential gamma correction function. The resulting 8-bit intensity values are then converted into analog waveforms by high speed Digital to Analog Converters, frequently modulated on an 8 MHz carrier and written on the video disk as normal video signals. When the three heads are not being used to write new lines on the disk, they are connected by a diode matrix to three read amplifiers which supply high level video to the color monitor(s). The required sync signals are generated by the disk electronics using a sync track on the disk and supplied to the monitor also.

Display Examples

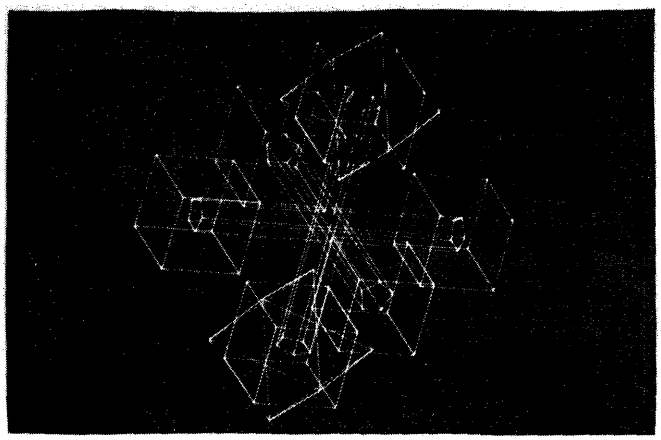
3.1 Introduction

The test objects shown in this section were generated interactively using the BUILD[13] system which we developed for our Adage AGT-30. In this system, objects are defined by collections of closed, planar polygons in 3-space. The edges which define these polygons may be displayed in real-time using the hybrid array processor and vector generator associated with the AGT-30 system. Images shown on this system are difficult to assess because of the ambiguity associated with depth determination and the lack of hidden line removal (see Fig. 3.1). However, the operator may create "snapshots" of his image by passing it to the color display subsystem for scan conversion, hidden surface removal and ultimate display on the color television monitor (Fig. 3.2). Part of this handling process involves modification of the defined hue of each polygon to account for its orientation with respect to the viewer. This gives solid objects a realistic shaded appearance (Figs. 3.2 to 3.6).

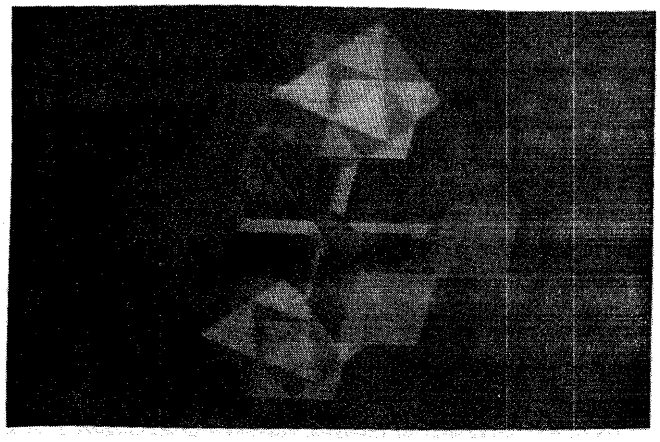
Conversion times and program statistics for five objects (Figs. 3.2 to 3.6) are shown in Table 3.1. The table lists:

A. Total number of polygons in the object.
B. Total number of edges contained in the object.
C. The maximum number of polygons that cross the same scan line. The objects shown here contain mostly triangular and rectangular surfaces.
D. Average number of polygons in 512 scan lines.
E. Number of intersection calculations performed.
F. Actual number of intersections detected.
G. Compute time in seconds for the scan conversion algorithm.
H. Total image generation time in seconds. This includes the disc latency time.

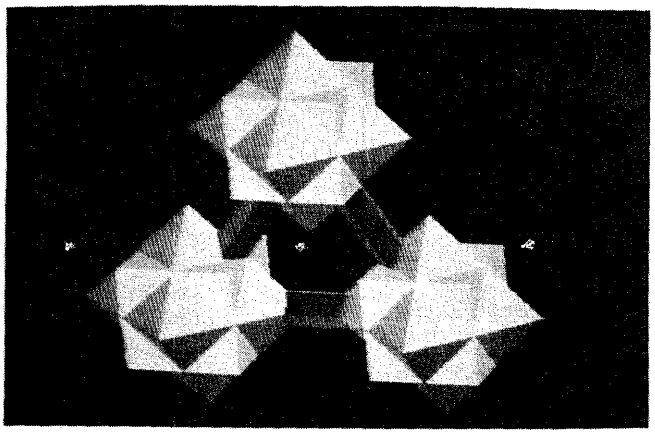
From the table it is obvious that quite complex images may be displayed with ease. The display system is rugged and cost effective. Further work proceeds in displaying quadratic and higher order surfaces and on a faster display processor.[11,12]



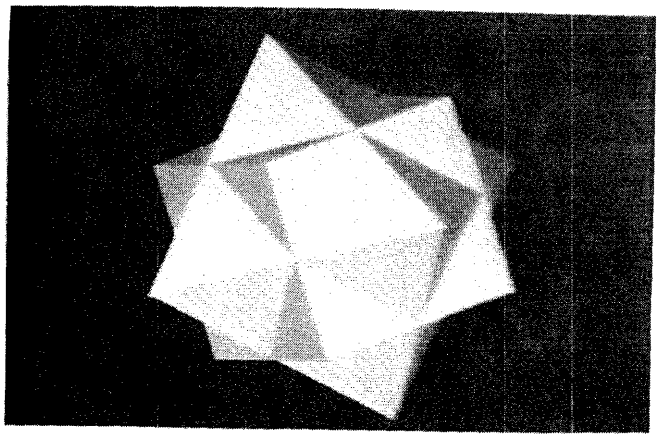
STKØ1
Figure 3.1



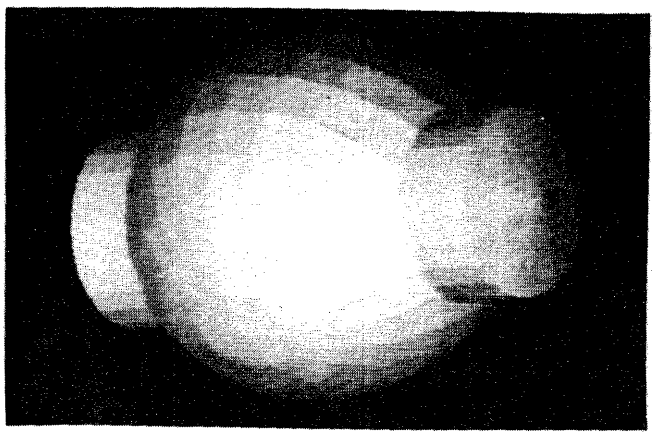
STKØ1
Figure 3.2



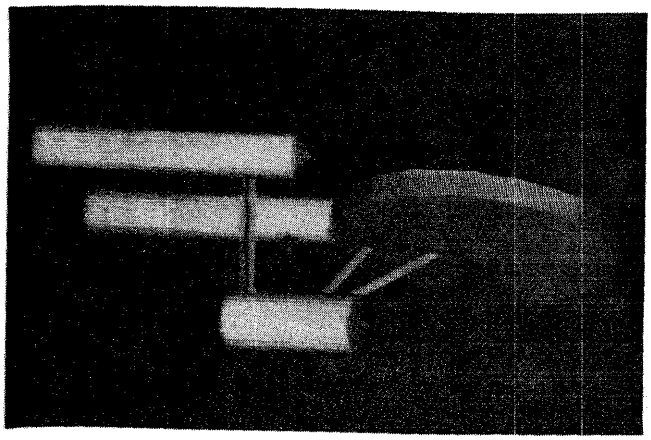
STKØ2
Figure 3.3



STKØ3
Figure 3.4



STKØ4
Figure 3.5



STKØ5
Figure 3.6

Table 3.1

OBJECT	STKØ1	STKØ2	STKØ3	STKØ4	STKØ5
A. TOTAL # POLYS	73	63	14	432	522
B. TOTAL # EDGES	282	234	48	1584	1756
C. MAX POLYS/LINE	27	34	11	94	113
D. AVG POLYS/LINE	9.45	9.35	3.52	20.9	14.3
E. # INTER CALC	903	1720	576	245	265
F. # INTERSECTS	814	1296	569	240	246
G. COMPUTE TIME	9	9	4	32	25
H. TOTAL TIME	11	11	8	33	26

Conclusion

The work which led to the results reported in this paper concerned itself with a critical evaluation of the entire display process. In particular, we have proven that TV technology is compatible with minicomputer technology. The generation of faster graphics in full shaded color is achievable if picture elements can be generated in about 100 n sec. Hence a second generation graphics system has been designed which is basically an expandable minicomputer optimized for display processing. The use of ECL technology will result in a calculated worst-case cycle time for instruction fetch and execute of about 100 n sec.[10,11]

References

1. Wylie, C., et al, "Half-tone Perspective Drawings by Computer," AFIPS Proc, FJCC 31, p. 49, 1967.

2. Warnock, J., "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," Technical Report 5-15, Computer Science, University of Utah.

3. Elson, B., "Color T.V. Generated by Computer to Evaluate Spaceborne Systems," Aviation Week and Space Technology (October 1967).

4. Burbaum, W., "Visual Simulation: Computer vs. Conventional," G.E. News Bureau Release 3439-967-39B (October 1967).

5. Bouknight, W. J., "An Improved Procedure for Generation of Halftone Computer Graphics Presentations," Report R-432, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, September 1969.

6. Kelly, K. C., "A Computer Graphics Program for the Generation of Half-tone Images with Shadows," Report R-444, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, (November 1969).

7. Watkins, G. S., "A Real-time Visible Surface Algorithm," UTEC-CSV-70-101, Computer Science, University of Utah, Salt Lake City, Utah, June 1970.

8. Sproull, R., Sutherland, I. E., "A Clipping Divider," AFIPS Proc, FJCC-33, pp. 765-776, 1968.

9. "The Evans and Sutherland Shaded Picture System," Evans and Sutherland Computer Corp., Salt Lake City, Utah, 1973.

10. Eastman, J. F., "An Efficient Scan Conversion and Hidden Surface Removal Algorithm," Proc. ACM Conference on Interactive Graphics, Boulder, Colorado, July 1974 (To be published in Computers and Graphics).

11. Eastman, J. F., Wooten, D. R., "A General Purpose, Expandable Processor for Real-Time Computer Graphics," Proc. ACM Conference on Interactive Graphics, Boulder, Colorado, July 1974 (To be published in Computers and Graphics).

12. Staudhammer, J., Eastman, J. F., England, J. N., "A Fast Display-Oriented Processor," This Conference.

13. Ogden, D. J., Staudhammer, J., "Computer Graphics for Half-Tone Three-dimensional Object Images," Proc. ACM Conference on Interactive Graphics, Boulder, Colorado, July 1974 (To be published in Computers and Graphics).

A MICROPROGRAMMED PROCESSOR FOR INTERACTIVE COMPUTER GRAPHICS

Henry D. Kerr
Adage, Inc.
Boston, Massachusetts

Summary

This paper describes the architecture of the microprogrammed processor which is the heart of the Adage GP/400 interactive graphics system, and discusses some of the factors which influenced its design.

Background

Adage, Inc. manufactures high-performance interactive refresh graphics systems. These systems are used typically in applications requiring a high degree of image structure. This structure includes functions such as loading or concatenation of coordinate transformation values for subsequently displayed picture parts, transfer of control commands such as image branches, or sub-image calls, and commands for setting and reading console devices.

In previous generations of Adage graphics systems, the image structure was processed by a computer program, invoked by an interrupt from the graphics hardware when it "saw" a graphical command it could not process. Even though these software-implemented commands were performed by a relatively fast, medium-scale digital processor, in highly structured images the number of vectors and characters displayed in each refresh cycle could be reduced to 25% or less of what the system was capable of displaying. The reduction of this overhead was one of the primary objectives to be met by the Adage GP/400 system architecture.

A second important factor which heavily influenced the design was that of programmability. One graphical command set is not always the best for many different applications. For example, the fastest and most efficient graphics language for situation display (i.e., aircraft trainers, simulators, etc.) is not the best for an interactive computer aided design (CAD) application. In addition, Adage had two initial uses for the GP/400. One was as a graphics peripheral system which was oriented to a 16-bit command format and could be attached to a variety of different small-, medium-, or large-scale digital computers. The other use was a graphics front-end attached to the Adage DPR4 processor to provide a new system (the GS/300), compatible with the existing Adage AGT/100 Series graphics systems.

The command sets of the GP/400 and GS/300, while functionally alike, are quite different in actual format. Furthermore, differences in various potential host computers for the GP/400 might require changes in image command processing (e.g., the DEC PDP-11's processing of the right byte of a word first in string operations instead of left byte first as on other systems). Also, we had expected to be called upon to implement other application-specific command sets and to emulate other graphics systems such as the IBM 2250.

The third major factor was the desire for a lower-cost, digital implementation of coordinate-transformation hardware. Previous Adage systems used fast, high-precision multiplying DACs, with the top-of-the-line, 3-D system, employing sixteen multiplying DACs to solve the following equations for each vector:

$$X' = PS[DX + SCL(R_{11}X + R_{12}Y + R_{13}Z)] \quad (1)$$

$$Y' = PS[DY + SCL(R_{21}X + R_{22}Y + R_{23}Z)] \quad (2)$$

$$INT = PS[DZ + SCL(R_{31}X + R_{32}Y + R_{33}Z)]IS + ID \quad (3)$$

where X, Y, and Z are the 3-dimensional image coordinates of the vector endpoint, and X', Y', and INT are the display screen X-, and Y-axis endpoints and intensity respectively. It was highly desirable to replace the multiplying DACs with a fast digital multiplier for solving the above equations as well as for the calculation of transform concatenation (i.e., the matrix multiplication of one transformation function -- rotation, displacement, scale -- against the current transformation set).

System maintainability was of prime importance to the design of the system. A digitally oriented system, plus microprogramming capability, would allow built-in firmware diagnostic routines and facilitate system maintenance.

As the need was established for local computing power in the GP/400 to interpret the image structure and to provide an "intelligent" graphics system, available commercial processors were evaluated for inclusion in the system. It was very quickly determined that commercial mini-computers were too slow to provide the speed necessary for the high performance desired. Systems which offered user-specified microprogramming were evaluated. In commercial mini-computers, such as the Interdata Model 85, the microprogramming available was most powerful in implementing the target machine instruction set, not in providing the facilities needed for our application.

Systems which were specifically designed to be microprogrammable,[3,4] such as the Microdata MICRO 1600 and the Digital Scientific Corporation META 4, were also examined. Although the microprogramming capability in these systems was more general, firmware implementation of the graphical commands we consider essential would still have been too slow to meet our performance goals.

As no commercially available processor would satisfy our needs, we decided to develop a fast, high-performance, microprogrammed processor of our own.

System Development

Initial system design of the GP/400 began in the fall of 1972. The first-pass design employed a 32-bit microinstruction word. As it was desired to implement the system with TTL devices, this design was abandoned because the large amount of decoding required prevented system operation at the desired speed. The overall requirements pointed to a more horizontal, parallel approach for the next design iteration.

A hardware design based on a 56-bit microinstruction word was completed early in 1973. In March, a simulator for the processor was written on the Adage DPR4 computer. Even though this simulator ran about one thousand times slower than real-time, microprograms similar to those running now on actual systems were run. This simulation led to further refinement of the hardware design. Although minor changes and re-arrangements of the microinstruction format were made during the development of the processor, the 56-bit microinstruction word was retained.

Although the market need for a system with writable control storage (WCS) was expected from the beginning, it became obvious in the early part of 1973 that writable control storage was also essential for prototype development. Therefore, the initial system design was with WCS, with the design goal that both WCS and read-only memory (ROM) would have identical execution speed.

Prototype checkout was started in the fall of 1973, and proceeded rapidly, thanks to the wire-wrap fabrication technique used. A fully working prototype was complete in December and the first shipment was made in March of 1974.

The Final Design

Figure 1 is a block diagram of the GP/400 system. Except for the Remote Console Interface (RCI), the system is implemented on 14" by 16" wire-wrap panels housed together in a rack-mounted card cage. The Host Computer Interface (HCI) occupies one panel, the High-Speed Stroke Generator (HSG) occupies three panels, the microprogrammed processor (DGC) occupies two panels, and the Control Store (CS) occupies four panels if bipolar RAM (WCS), or one panel for fusible-link PROM. Analog elements for stroke generation, intensity control, and circle-arc generation are built on "daughter boards" which plug into two of the HSG wire-wrap panels. A separate RCI is located at each of up to four remote CRT consoles. The Maintenance Control Panel (MCP), rack-mounted above the card cage, contains switches and indicators used for maintenance trouble-shooting of the GP/400, and for microprogram debugging on systems with Writable Control Storage. The internal structure of the DGC is shown in Figure 2. This figure depicts the data paths, arithmetic units, storage elements, and registers which make up the DGC. Data paths, except where noted, are 16-bits wide.

The Control Store (CS) contains 56-bit microinstructions which are addressed by the Location Counter (LC). Versions of the Control Store include both fusible-link PROM and bipolar RAM (WCS). Typical memory sizes range from 1K to 2K words, but the CS may be expanded up to 4K.

The Scratch Pad (SP) is a high-speed bipolar RAM containing 256 16-bit words. SP is used to store dynamic data values. It is the primary storage element for the microprogram as the CS is not writable by the DGC itself (only from the host computer). The SP may be addressed directly by the microinstruction, or indirectly by the lower 8 bits of the MS register.

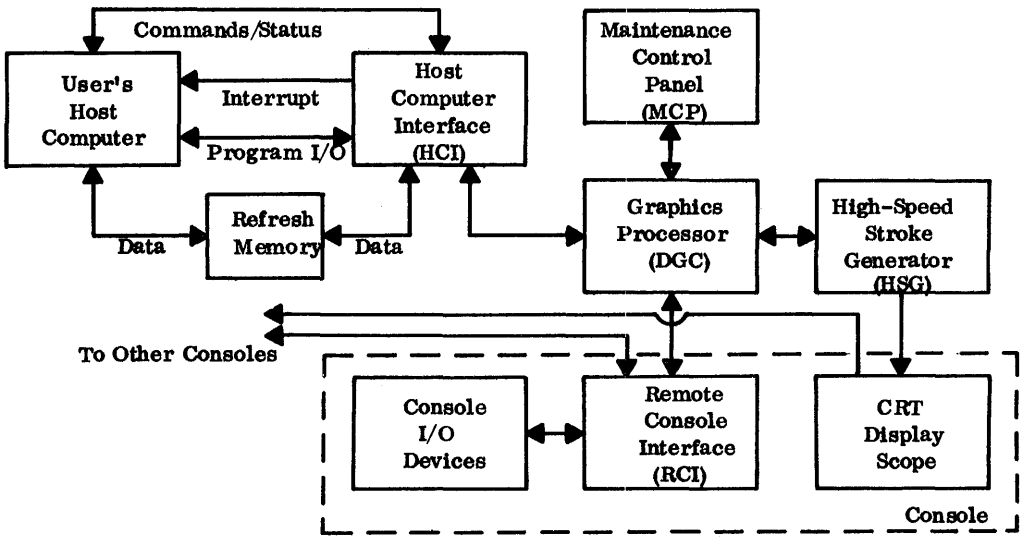


Figure 1. GP/400 Graphics Peripheral Organization

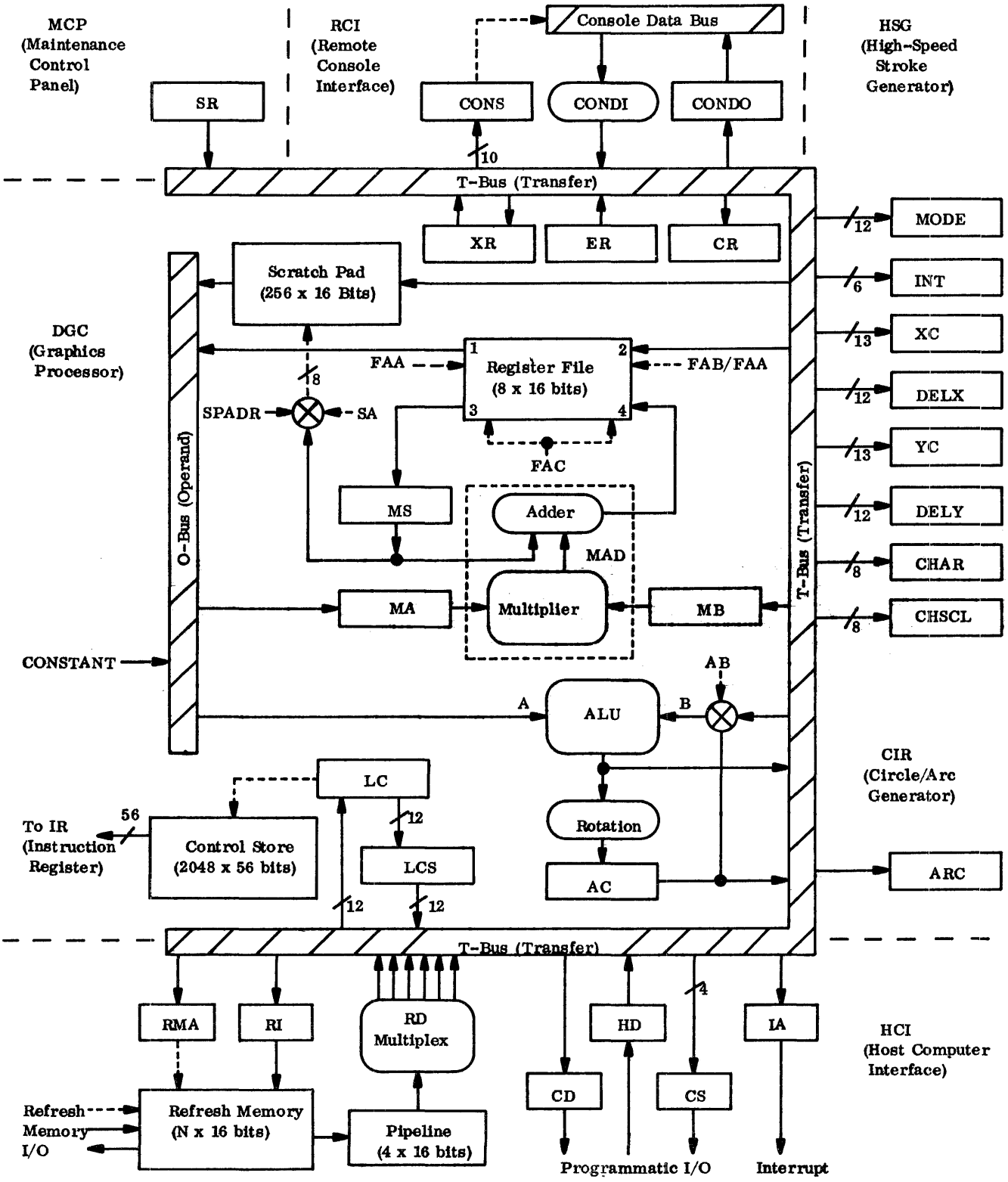


Figure 2. Graphics Processor Organization

The Register File (RF) contains eight 16-bit registers, arranged with two parallel input ports and two parallel output ports. One input port and one output port are tied to the major data-transfer busses, as in the SP. The other input and output ports are connected to the high-speed multiplier-adder. The Register File is used to store local variables and to transform vector endpoints in conjunction with the multiplier-adder.

The two arithmetic elements are the Arithmetic-Logic Unit (ALU) and the Multiplier-Adder (MAD). The ALU is a 16-bit general-purpose digital computation element which may perform any of the possible 16 logical functions of two variables, or one of 16 arithmetic functions. A low-order carry bit is provided to the ALU by the DGC, which provides 32 arithmetic functions, although some are duplicates of the logical functions and others are not generally useful.

The ALU result may be transferred to other destinations in the DGC, and can be passed through rotation logic and loaded into the ALU Accumulator register (AC). The rotation logic provides a direct copy, a one-bit left rotation, a one-bit right arithmetic shift, or eight-bits (halfword) rotation.

The "A-operand" input to the ALU comes from the Operand Bus (O-Bus) which may be the output of the SP, the RF, or a 16-bit constant provided by the microinstruction. The "B-operand" may be, under microprogram control, the current contents of the AC register, or may be the contents of the Transfer Bus (T-Bus).

The MA register is loaded from the O-Bus and the MB register is loaded from the T-Bus. The MS register is loaded from the Register File, through the output port associated with the MAD. The MS register may be also used to indirectly address the Scratch Pad.

Although the MAD computes its product-sum in approximately 230 nanoseconds, the effective multiply-add time is 400 nanoseconds (two microinstruction executions). On the first microinstruction the appropriate values are loaded into MA, MB, and MS. The resultant product-sum is stored in the Register File on the next microinstruction.

The multiplier consists of an array of 2-bit by 4-bit MSI multiplier chips, with adders to the sum the resultant partial products.

Various other registers are assigned for the purpose of storing specific data values, specifying control signals, and testing status signals. Most of these are accessible to the DGC as sources or destinations of the 16-bit T-Bus (although not all registers utilize the full 16 bits). Several of the source/destination registers are associated with HSG, RCI, HCI, and MCP, and provide the interface between the DGC and these subsystems. The Console Data Bus, a bi-directional 16-bit data bus, runs in a daisy-chain fashion from the DGC to each RCI located at each CRT console. The direction of the bus, the console select, and console data register address are controlled by bits loaded into the console bus control register (CONS).

The 56-bit microinstruction format is shown in Figure 3. Bits Ø through 55 specify the various fields of the microinstructions. Bit 63 is a memory parity bit used only with Writable Control Store. The OP field, bits Ø and 1, is the only field which changes the meaning of other fields in the microinstruction. This field performs a dual function of selecting the data presented to the O-Bus and specifying a "branch" microinstruction (OP = 3). For OP = Ø, the Scratch Pad word addressed by SPADR (or addressed by the MS register if SA = 1) is placed on the O-Bus. For OP = 1, or OP = 3, the Register File register specified by FAA is on the O-Bus. For OP = 2, the constant value in bits 16-31 of the microinstruction is placed on the O-Bus.

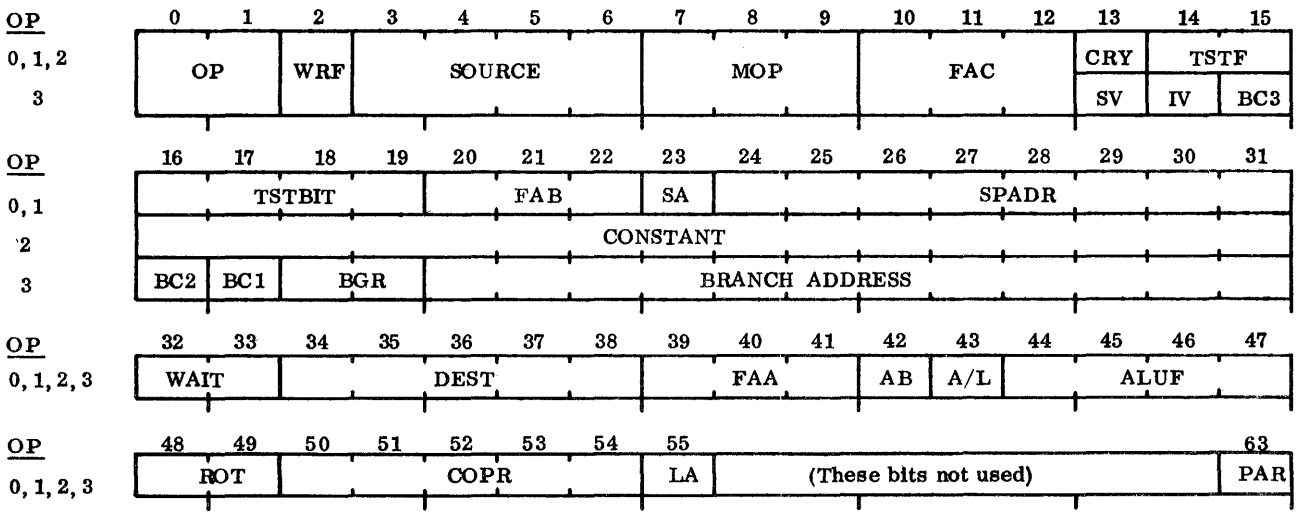


Figure 3. Microinstruction Format

The SOURCE field specifies the data to be placed on the T-Bus. Eight of these sources specify values from the Host Computer Interface (HCI). One of these values typically is a 16-bit register used in programmatic I/O transfers to the GP/400. The other seven sources specify various bit-mappings of a 16-bit value read from the host memory via a high-speed DMA channel or memory-port interface. This value is the output of a four-register "pipeline" built into the HCI to minimize word-fetch latencies.

The other eight sources select other registers or values such as the output of the ALU, the Accumulator (AC) register, the Index Register (XR), the 16-bit Switch Register (SR) from the Maintenance Control Panel (MCP), etc.

At the end of each microinstruction, the value on the T-Bus is loaded into the specified destination (DEST). Eight destination codes are assigned to the HCI and consist of registers for the address of data to be read from host-computer core, for data to be written into core, for programmatic I/O transfer of data to the host computer, for interrupt address to the host computer, etc. Sixteen destination codes are assigned to the High-Speed Stroke Generator (HSG) and to other optional graphics devices such as the Circle-Arc Generator and the Hardware Windowing Device. These destinations are used for loading values such as vector end-point coordinates, vector modes, intensity, character code and scale, vector texture and display scope enables. The other eight destination codes, used by the DGC itself, select destinations such as the Scratch Pad (same addressing as for reading), the Location Counter (LC), the Index Register (XR) and others. The Register File may be loaded from the T-Bus in addition to the selected destination (if WRF = 1) and is addressed for writing by the FAB field (for OP = Ø or 1) or the FAA field (OP = 2 or 3).

Several fields of the microinstruction control the ALU and Accumulator (AC). The ALUF field specifies the ALU function code and A/L selects an arithmetic or logical function. The AB bit selects either the T-Bus or the contents of the AC as the ALU "B" input. The low-order carry into the ALU is controlled by CRY. The output of the ALU (with possible rotation specified by ROT) will be loaded into the AC if LA is set.

As the T-Bus may be an input or an output of the ALU, microinstructions may specify several ways for data to flow through the system. For example, both the O-Bus and the T-Bus may be inputs to a two-variable ALU function and the result stored in the AC (after possible rotation). Note that the T-Bus value may also be written into a destination register and/or the Register File. Another possibility would allow a value on the O-Bus to be operated upon by the ALU (a one-variable function, or a two-variable function with the AC as the ALU "B" input) and placed on the T-Bus for writing into the selected destination and/or Register File. A variation of the latter case would allow the O-Bus value to be loaded into the AC, while the AC (or some other source) is sent on the T-Bus to the selected destination.

A bit-testing facility is provided by the TSTF and TSTBIT fields. The DGC maintains three general-purpose program flags (F1, F2, and F3). A selected flag (specified by TSTF) may be set to the state of a bit on the T-Bus specified by the TSTBIT field. These flags may be tested by branch conditions, and may be cleared or complemented by control functions (see below).

The high-speed multiplier-adder (MAD) is controlled by the MOP field and the Register File address field FAC. The MOP field specifies the multiplier function, such as no operation, storing the product-sum into the Register File, or selecting various combinations of loading the MA register from the O-Bus, loading MB from the T-Bus, or loading MS from the Register File.

A full multiply-add takes place in two successive microinstructions in which the first loads the desired combination of MAD input registers, while the second stores the product-sum into the Register File. The first microinstruction, in general, uses the major busses and transfer paths. However, the second only stores the product-sum in the Register File and the microinstruction is available for other microprogram operations.

The COPR field provides control "pulses" to various subsystems of the GP/400. Eight functions are assigned to the HCI and are used for starting operations such as requesting words from the host computer's core, writing data into core, requesting a host-computer interrupt, etc. The HSG is assigned eight functions for starting vector or character drawing, and resetting the HSG subsystem. The remaining sixteen control functions are assigned to the DGC and are used for operations such as clearing or complementing the microprogram flags (F1, F2, F3), capturing light-pen "hits", counting the Index Register (XR), sending data to the remote consoles, and halting the microprogram execution.

To minimize microprogram latency, the processor may "pause" until the occurrence of a particular event. The WAIT field may specify a "pause" until either a word is available from the host-computer core memory, or until the HSG has completed the drawing of the previous vector or character. This "pause" occurs at the start of execution of the microinstruction in which the WAIT field is specified.

The branch operation (OP = 3) modifies the function of several of the microinstruction fields. Bits 20 through 31 provide the 12-bit branch address. The fields BC3, BC2, BC1, and BGR provide 32 possible branch conditions (with one code always "true" to provide an unconditional branch). The IV bit inverts the sense of the condition. If the resultant branch condition is true, the branch address is loaded into the Location Counter (LC). If the SV bit is set, the LC value is saved in the LCS register prior to being loaded. Branch conditions include indicators and conditions such as the state of the microprogram flags, the AC being zero, negative, or normalized, the ALU overflow and carry indicators, HSG busy state, and events such as a light-pen "hit", keyboard strike, and frame-clock indicator. Other branch conditions assigned to the HCI test completion of host-computer memory operations and programmatic I/O "hand-shaking".

The branch operation is performed early in the microinstruction execution cycle to allow for the overlapped fetching of the correct next instruction. Only one hardware subroutine level is provided; however, any number of levels may be provided by saving the contents of the LCS register in the Scratch Pad or Register File.

A second method of transferring microprogram control is the use of the LC register as a destination of the T-Bus (a "dispatch"). As the transfer occurs at the end of the microinstruction execution, the next (following) microinstruction

has already been fetched and will be executed prior to the execution of the microinstruction at the new location. The "dispatch" is the normal form of subroutine return. This deferred branching creates several types of special instruction sequences. For example, a second dispatch instruction may follow immediately after the first, causing only one instruction at the first dispatch address to be executed, followed by sequential execution starting at the second dispatch address. For a second example, a conditional branch may follow the dispatch instruction, overriding the dispatch and branching if the condition is true, or allowing the dispatch to occur if false. Conditional subroutine returns may be coded in this manner.

External control of the GP/400 processor comes from two parallel sources. Located on the Maintenance Control Panel (Figure 4) are seven pushbuttons, which provide manual control of GP/400 system reset, place the DGC into run mode or halt mode, and provide the halt-mode functions of STEP (indexing LC), CYCLE (single microinstruction execution), BRANCH (loading the LC), and LOAD (loading microinstructions). The BRANCH and LOAD operations use data from the 16-bit toggle-switch register on the MCP. The LOAD function is used to load microinstructions into the Writable Control Storage, or in systems with PROM Control Storage, will load the 56-bit Instruction Register (IR) for single-step execution. The MCP also contains 32 LED indicators which are multiplexed by a four-position rotary switch to provide 128 bits of display information, such as the contents of IR, T-Bus, LC, Console Data Bus, etc.

Seven control commands, equivalent to the pushbuttons on the MCP, are provided to the host computer, with the programmatic I/O data path used for values required by the BRANCH and LOAD functions. A fifth position of the MCP rotary switch allows the host computer to selectively read back the 128 bits of display data in 8 words of 16-bits each. With the ability to load and execute microinstructions from the host computer (even on PROM systems) and to read back major data paths and registers, a powerful diagnostic environment is provided.

Conclusion

The performance improvement brought about by the new microprogrammed graphics processor can be illustrated best by an example. Essential to graphics display applications, which involve rotation of picture parts, are sine/cosine calculations.

The calculation of the sine and cosine of an angle has previously been performed by a software routine running in the Adage DPR4 processor. Execution time on this 1-microsecond cycle-time machine is on the order of 100 microseconds for about 14 bits of precision of sine and cosine. By comparison, when executed by a microsubroutine in the DGC, sine and cosine results are generated with 15 bits of precision in only 5.6 microseconds, including the call and return. The subroutine occupies less than 60 words in the Control Storage, including a 32-word lookup table. Other functions implemented in microprograms show similar speed improvements, averaging 10 to 30 times better than execution times on the host processor.

Although the system design of the GP/400 is at least more than once around the "wheel of reincarnation" of Myer and Sutherland[5], the introduction of a microprogrammed processor as the heart of an interactive graphics system has led to enhanced flexibility, superior performance, and more effective maintainability, at a lower cost, compared with previous-generation hybrid graphics systems.

References

1. "Adage GP/400 Graphics Peripheral System, User's Reference Manual", Adage, Inc. document GP/400/URM/D, June 1974.

2. "GRAFX, AMOS/2 Image Display Operator, Programmer's Reference Manual", Adage, Inc. document GRAFX/PRM/B, December 1971.

3. "META 4 Computer System, Microprogramming Reference Manual", Digital Scientific Corporation document 7043MO, June 1972.

4. "Microprogramming Handbook", Second Edition, Microdata Corporation document, 1971.

5. Myer, T.H.; Sutherland, I.E., "On the Design of Display Processors", CACM 11,6,410, June 1968.

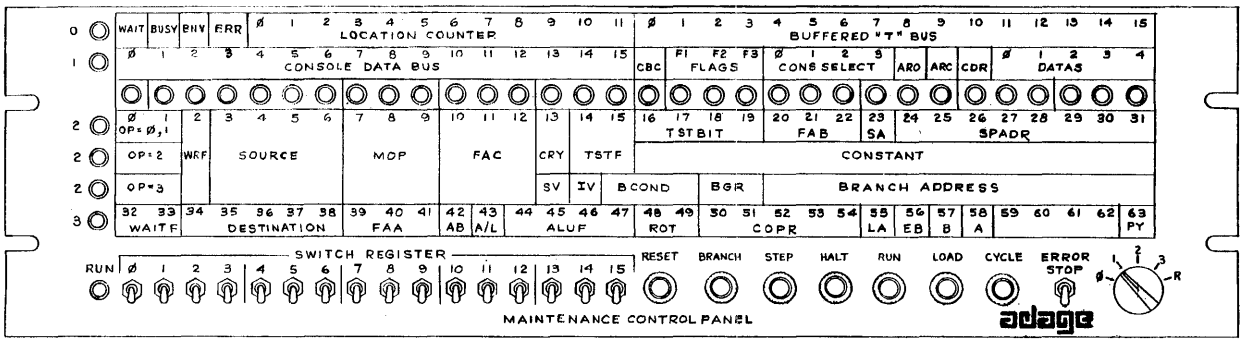


Figure 4. Maintenance Control Panel

FUNCTIONAL MEMORY TECHNIQUES APPLIED TO THE MICROPROGRAMMED CONTROL OF AN ASSOCIATIVE PROCESSOR

C.V.W. Armstrong

Dept. of Computer Science, University of Edinburgh

Edinburgh, Scotland

Summary - Consideration is given to the separation of the data and control structures of a microprogrammed processor. The use of functional memory techniques to provide a suitable medium for containing and processing the control structure and giving one possible solution to this separation problem is described. The design considered is that of an associative processor but the techniques involved are applicable to other types of processors. Some of the advantages of this approach are given, together with their implications in the light of advances in microprocessor technology and cellular logic.

Introduction

The Control Structure

The microprogrammed control unit of a processor has the task of fetching and executing instructions. These two functions can be separated out and handled by different control units. Similarly, the execution unit can be divided into a section that processes the data and a section that processes the control structure related to that data. Separation of this sort is useful because each control section is specialised for a particular type of operation and can work in parallel with the operation of the other sections. It seems possible to separate out these control sections[1] and in fact the use of a microprogrammed control unit can go some way towards making this separation. However, when the control structure requires processing, the microprogrammed control unit may borrow resources provided primarily for the processing of the data structure and parallel operation is no longer possible.

Take for example the case of the Interdata Model 70 and related models, a minicomputer where vertical microprogramming predominates. Testing may require the use of both S and B busses and does not allow multiway branches to take place. This, for example, would be very useful in the processing of interrupts. The processing of the control structure preempts any processing of the data structure during microroutine execution. In fact, the processing of the control structure is constrained by the data processing architecture of the processor[2]. A number of instructions have the same microroutines except for minor changes, which are restricted to small differences in the processing of the data structure. A means of separating out these two control sections may go some way in compacting the amount of space required to hold the differing microroutines into one where no such replications are necessary.

The attempt is made here to show how the use of functional memory techniques may allow a single microprogrammed control unit to process, in parallel, the data and control operations involved in executing instructions. This approach may be useful when a number of modules, such as microprocessors, must be controlled in parallel.

Functional Memory.

The term functional memory[3,4] was used to describe the use of a special type of memory to realise various combinational and sequential functions. A cellular structure was considered in which each cell could have three possible values : 0, 1 or X (corresponding to "don't care"). A rectangular array of these cells could perform searching operations on selected fields of rows of cells and data fields from selected rows could be ORed together during output. Functional memory had the advantages of regular circuitry such as RAMs and ROMs when implemented in LSI. The approach given here does not require customized LSI.

The Design of an Associative Processor.

An associative processor was chosen as a particular example of a subset of parallel processors, the subset of all single instruction stream - multiple data stream processors. The microprogrammed control of such a processor was studied. It is an example where the data structure is considerably different from the control structure and where control problems could be acute. Associative processors are examples where much of the cost of the machine is concentrated in the control unit and other supporting modules, and where improvements in control unit design can have a significant effect. In Goodyear's STARAN processor, an array of 256 associative elements uses 2,500 integrated circuits whilst the circuits necessary to control and support this array number 6,500.[5]

There are no fundamental associative properties in the processor which was developed. These associative properties are provided by the management of the processing elements by the microprogrammed control unit. The associative operations are emulated by standard arithmetic, logical and testing operations on 16-bit word-slices of a 256-bit word, as opposed to bit-slices. The control unit could alternatively emulate a different type of parallel processor in this subclass.

A hardware realization of an 8 processing element associative processor was built using the inventory of DEC RTM modules available in this Department, with the addition of a small number of other integrated circuits.

Architectural Decisions.

It was necessary to choose applications for which the associative processor could be used whilst its operation was being studied. It was decided to consider simple programs for:

(i) air traffic control conflict detection, and
(ii) information retrieval query processing.

Both these applications are suitable for associative processing. They also seemed complementary in their use of available operations of an associative processor.

The hardware realization was constrained by the inventory of DEC RTM modules and other circuits available, and this meant that the associative processor would have 8 processing elements with each element storing 256 bits as 16 words of 16 bits. Extensive processing capability was available at the processing element level.

The associative processor is interfaced to a sequential processor - an Interdata Model 74. This stores the program for the associative processor together with the data for its processing elements, which can only be accessed in sub-blocks. Thus, the sequential processor emulates the fetch section of the ideal associative processor being considered.

There would be I/O traffic of data for PEs during associative processing which would be reduced as the number of PEs increased. The point could be reached when all the data necessary for a particular search or data processing operation is loaded in one block.

The instruction stream is provided by the sequential processor and instructions modifying this stream are trapped by the sequential processor. If the modification is conditional, it is based on status information provided by the associative processor. The associative processor microprogrammed control unit can determine whether the status information in the sequential processor requires modification and transmits this new status, or whether the new status is for local control and can remain in the associative processor status register.

The following policies were pursued:-

(a) Apart from supplying the instructions and data, the sequential processor would be free to process programs in parallel with associative processor operation.
(b) An operation or design guideline would be adopted for the hardware realization only if it could be employed effectively or even more effectively in a larger (full-scale) associative processor.
(c) By careful choice of instructions, loops would be kept at the level of the microcode where they could be handled more efficiently, thus bringing the instruction stream as close to a sequential stream as possible. This is much easier to do in an associative processor, where iteration can be handled by exploiting parallelism.
(d) Delays due to modification of the instruction stream would be minimised by careful choice of the status information to be transmitted to the sequential processor and transmission would be well before this data is required for modifying the instruction stream.

The Data Processing Structure

The data processing structure consists of eight processing elements (PEs). Each PE has 16 16-bit words, a simple ALU, two accumulators, and a means of communicating with the MCU. It is a simple data processor which could be replaced by an LSI microprocessor.

In examples of associative processors such as STARAN, serial processing of a 256-bit word allows several fields of differing length starting at differing places to be processed. Here, processing is in parallel on 16-bit words, so the data fields must be integral values of 16-bits and aligned on a 16-bit word boundary. The STARAN example offers considerably more flexibility in the layout of data. However, the necessary control is correspondingly more complex and may require a lower level of microprogramming to achieve results such as the addition of two 16-bit fields.

The possible operations of a PE can be seen from the first part of Fig. 1.

As shown in Fig. 2, the PEs are connected in a ring structure, each being able to transmit a 16-bit word to one neighbour and receive a 16-bit word from its other neighbour. In addition, the microprogram control unit (MCU) can transmit a 16-bit word to all enabled PEs. The MCU can receive the ORing of 16-bit words transmitted from all the PEs enabled. The MCU can also receive an 8-bit status word from all the PEs enabled with each PE represented by a corresponding bit of the 8-bit word.

Note that this processing structure is a general purpose parallel processor with a simple interconnection pattern and that the associative properties are provided by the MCU which alternatively could emulate a different sort of parallel processor. The interconnection structure is easily modified in the hardware realization if this should prove necessary.

The Microprogram Control Unit.

The objective is to supply minimally encoded microinstructions to both the PEs and the MCU. Here, "minimally encoded" means that the microinstruction is decoded as much as economically possible with respect to the number of control lines and the corresponding number of pins on functionally organized LSI chips such as microprocessors, RAMs and ROMs. Thus, the size of the microinstruction wordlength is not considered a restraint in the ideal case.

The requirement, in this case, is that the MCU must perform a mapping from a user instruction of 16-bits to a variable number of 64-bit microinstructions where the first 24 bits are minimally encoded for use by the PEs (the data structure) and the second 40 bits are minimally encoded for use by the MCU (the control structure). Note that the MCU microorders are at a slightly higher level in order to reduce the number of bits required in the microinstruction.

The MCU is essentially as shown on Fig. 3. Each block named SPn or RPn represents a 16*16 scratchpad memory for the select phase or read phase, respectively. These will be called functional memories because of the role that they play in the design.

The basic interpretation cycle is as follows:- The select phase register breaks up the 16-bit word stored in it into 4 4-bit fields which address 4 16-bit words in the select phase functional memories. The 4 16-bit words so accessed are ORed together and form the 16-bit word stored in the read phase register. This word in turn is broken up into 4 4-bit fields and used to address the read phase functional memories. This provides the 64 bits of the microinstruction. The first 24 bits are used to control all the PEs enabled by the enable register. The second 40-bit field controls the MCU and either generates another 16-bit word for loading the select phase register or uses the next 16-bit user instruction to load it. The same interpretation cycle continues with the only variations allowed being the way in which the next 16-bit word for the select phase register is determined.

Fig. 1 shows the format that was used for the microinstruction in the design of this associative processor. Note that a few bits are as yet unused and may have their roles assigned later.

A number of other registers, mainly self-explanatory, are shown in Fig. 3. With proper timing, one register could hold the 16-bit word for both the select phase and the read phase. The user instruction or portions thereof can be loaded directly into the select phase register. In the DEC RTM realization, the microinstruction is used directly. If this was not possible due to synchronization problems then a 64-bit master-slave register could be used with the MCU and the PEs controlled by the master flipflops whilst the slave flipflops are being prepared with the next microinstruction.

A simple example of the use of this MCU is now given. Consider that the functional memories are loaded as shown in Fig. 4. We consider the example of the user instruction 0001 0010 0111 0001 to read in a variable amount of data into a variable number of PEs. Now the user instruction is in general broken into 4 4-bit fields as follows:-

(1) First Field - the instruction operation code. For example - 0001 - read in a variable amount of data into a variable number of PEs.
(2) Second Field - any necessary information for the operation and the PEs, including data or information where data is to be found (such as data or address in the next 16-bit word of the instruction stream). For example - 0010 - load the first (2+1) scratchpad words of each PE.
(3) Third Field - any necessary information for the MCU. For example - 0111 - load (7+1) PEs consecutively. This field is stored in the X register of the MCU.
(4) Fourth Field - the control level - corresponds to a particular microprogram. This is inspected by the MCU to cause switching from one set of functional memories to another and/or loading or preloading of functional memories. For example - 0001 - the control level 1 microprogram.

Operation commences as follows:-

The fourth field is used for checking that the appropriate microprogram is loaded. The first field is loaded into the first 4-bit field of the select phase register and the second field into the second field of the select phase register. The third field is loaded into the X register of the MCU. Thus, initially the select phase register is
0001 0010 0000 0000
Only SP1 and SP2 are used during the initial user instruction interpretation. This causes the read phase register to become
0010 0000 1101 0000
Note that the one's complement of 0010 (the count of the number of words to load into a PE) has been formed in the third field. The first microinstruction is output (see Fig. 1 for the operations) and the select phase register is now loaded directly from this microinstruction with
0001 1111 1101 0000
Note that a particular bit of the microinstruction causes the first field of the select register to be loaded from the first field of the instruction. The next read phase register contents are
0010 0001 1110 0001
Note that the second field has been incremented to 1 to access the next word in the PE and that the fourth field also has this value, which will be used in the next cycle for incrementing the second field again. Note that the third field has been incremented. The test whether the right number of words have been input to a PE has been made implicit because when the third field reaches 1111, it will cause RP3 to become all zero. This will cause the user instruction to be used in loading the select phase register as described above. If the third field of the user instruction as stored in the X register of the MCU is non-zero, its value is decremented and the same instruction used again. Otherwise, the next instruction is used.

The following points need to be made about this microroutine example:-
(1) The functional memories have enough space for more instructions than was shown above. The corresponding write instruction would be
0010 0010 0111 0001
There is space for 14 other instructions which require the performing of a particular operation sequentially on a variable number of 16-bit fields of a variable number of PEs. There are many other possible methods of using the MCU of which only one example has been given here. Space restrictions do not permit a detailed description of the other microprograms for parallel and associative operations. In the above example, the minimum amount of interaction between select phase functional memories in the generation of the read phase functional memories data has taken place. It is possible for two select phase functional memories to contribute alternate bits to a 4-bit field of the read-phase register and thus allow multiway branching within the one interpretation cycle. This could prove useful in instructions where considerable decoding and decision making was being employed.
(2) Although primarily designed for associative processing, the MCU is suitable for other types of single instruction stream - multiple data stream processing. Possibly this would require some extension of the interconnection pattern considered here.
(3) In addition, it is possible to pick different modes of interpretation by suitable use of the mode of interpretation field in the microinstruction. For example, in the mode of interpretation considered above, the select phase register is loaded from the previous microinstruction except in the case when RP3 is zero. In this case, the previous instruction is used again if the X register of the MCU is non-zero, otherwise the next user instruction is loaded.

Another mode tests the activate bit in the micro-instruction and ORs the value in the X register into the SPR before the execution of the next stage of interpretation. Other modes could make direct use of the status bits in the loading of the select phase register, thus allowing much of the branching and iteration to remain at the level of the microprogram. Although there is much repetition in particular fields of the above microprogram example, which is provided for explanatory purposes, the fact remains that another microprogram when loaded can use these same fields for entirely different control purposes and with other forms of repetition. See (vi) below. Note that the incorporation of the mode of interpretation as a field in the microinstruction allows a microroutine to change the mode dynamically during user instruction interpretation.
(4) Four different microprograms are now available and can be loaded corresponding to the four possible control levels specified in the last field of the user instruction. (Sixteen possible microprograms could be specified). There would be a delay during loading. Since a user instruction never requires the use of more than one of the microprograms, if two function memory "units" were available, one could be loaded whilst the other was being used. This is especially practical since the instruction stream is close to a sequential stream, and the fetch control unit could look ahead to the last 4-bit field of the next instruction. The second functional memory unit could be loaded by looking ahead for the first instruction which does not use the current microprogram and starting to load the required microprogram in parallel with the rest of the MCU operation. When this instruction is reached, the MCU switches over to the second functional memory unit and the above process can be applied to the first functional memory unit. Having three functional memory units would take care of the worst case of user instruction branching with minimum delay. With a judicious choice of the instruction classes and the corresponding microprograms for these classes, the amount of reloading can be decreased.[6] By increasing the permitted size and number of functional memories, this problem could be eliminated altogether.

The use of the above microprogramming technique has the following additional advantages:-
(i) This form of microprogramming does not seem to be any more difficult than the microprogramming schemes of many existing machines, especially when a simulator is available. It has the advantage of postponing until late in the design process, the microprogramming stage, the specification of the control structure and the operations on it. User microprogramming although possible is not the main objective of this approach.
(ii) In considering the firmware-software interaction, this approach allows much of the software to be concentrated in the firmware. The user writes instructions at a variety of levels corresponding to the control level specified in the instruction. At present, the microprograms are used as follows:-
(1) control level 0 - resetting, load, store, shifts, tests of PE condition codes, inter-PE communication
(2) control level 1 - input and output
(3) control level 2 - addition and subtraction with provision for multiple precision arithmetic, multiplication
(4) control level 3 - general associative operations - the first field of the instruction gives the type of test. The second field gives the scratchpad location on which the test is to take place. The third field directs what will be done with the result of the test.

The user can write programs using the higher level instructions (such as the general I/O instruction and the general associative operation) descending to the lower level instructions only when the final elements of control are unavailable higher up the hierarchy (such as instructions that set particular bits in the enable register). This leads to more compact code and faster operation.
(iii) In considering the hardware-firmware interaction, many of the simple flags and small field operations that a control unit may use, such as condition and emit fields, are eliminated completely by using this particular firmware approach. In particular, tests, multiway branches, complementation, incrementing, shifting, masking and reformatting can take place implicitly. Key variables or bits of instructions can be stored and used when necessary without requiring any additional emit fields, flags or registers. Apart from the functional memories, the barest minimum of extra circuitry and registers are required.
(iv) Fault recovery can be facilitated if this is a critical factor. A special microprogram could test all the PEs in parallel, and also test the MCU. The isolation of the malfunctioning PEs can be accomplished by microprogrammed control of the enable register. Since the MCU consists mainly of identical functional memories, a spare can be switched in if one should fail. The only critical circuitry requiring consideration are the small number of MCU registers and the combinational networks. The necessary redundancy is therefore limited to these circuits and a small number of PEs and functional memories. This would be a negligible cost for a large associative processor.
(v) There is a minimum delay in processing the data structure, since by the time the data structure has been acted upon by a microinstruction, the MCU has cycled back in parallel and produced the next microinstruction. Status information when required need never come from the current microinstruction execution. This holds true even when a new user instruction interpretation has commenced. In particular, there is no delay when a test has to be made at the microprogram level, where hopefully the majority of all branches will take place.
(vi) Many instructions can use the same microinstruction fields (4 16-bit fields) and only the fields which are different require placement in a word of the read phase functional memories. The other existing fields can be used unchanged. This is one answer to the field combination problem.[4]

Microprogrammed Processors.

The previous section has attempted to show how it is possible to gain an increase in the capability of containing and processing the control structure of a processor by the addition of a minimum amount of extra complexity in the microprogram control unit. This was achieved by:-
(i) breaking up decoders for the functional memory units (or control memory in the conventional case) into a number of separate decoders,
(ii) breaking up the read phase of the control memory into a select phase and then a read phase, and
(iii) performing an ORing of the output from the select phase functional memories. This requires no additional circuitry when negative logic and open-collector drivers are used as in the DEC RTM TTL implementation.

Thus, this is one possible solution to the problem of designing processors with separate data and control structures. It would be interesting to consider this approach in different types of processors.

Microprocessors

In the design of the associative processor, each processing element was a 16-bit parallel processor with a limited number of possible low level operations. This approach was used because of availability, speed and simplicity. It simplified the microprogramming and did not require the lower level of control that a serial processor would require.

Consider however the case where a processing element is replaced with a microprocessor. This could still provide enough processing power and storage per chip if present trends continue. If microprocessors cost $20-$30 each in large quantities, then a 1K processing element associative processor would have a hardware cost of $20,000-$30,000 plus the cost of the control and supporting equipment. The fact that suitable microprocessors are available and that a possible MCU uses standard RAMs, ROMs and MSI logic circuitry means that the development costs are reduced. For example, the RCA COSMAC LSI microprocessor is very similar to the PE considered here.

Cellular Logic.

Looking further into the future and considering the continuum described by Weinberger[7], by increasing the number of decoders further, the point is reached when a decoder is addressing one of two possible output lines. The functional memory has then reached the cellular complexity of those originally considered by Flinders, et al[3]. Thus, it is possible to envisage the same sorts of operations considered here, in a cellular logic implementation with all of the many advantages covered by Kautz[8].

Acknowledgement : I am indebted to Peter Gardner for mentioning how 16 word memories could be considered as an extension of functional memories.

REFERENCES

(1) C.G. Bell and J. Grason. The Register Transfer Module Design Concept. Computer Design, May 1971, pp.87-94.
(2) P.C. Anagnostopoulos, M.J. Michel, G.H. Sockut, G.M. Stabler and A. van Dam. Computer Architecture and Instruction Set Design. AFIPS 1973 National Computer Conference Proceedings, Vol 42, pp 519-527.
(3) M. Flinders, P.L. Gardner, R.J. Llewelyn and J.S. Minshull. Functional Memory as a General Purpose Systems Technology. Technical Report T.R. 12.088, IBM United Kingdom Laboratories Ltd, Hursley Park, Winchester Hampshire, July 1970.
(4) P.L. Gardner. Functional Memory and its Microprogramming Implications. IEEE Trans. on Comp., Vol. C-20, No. 7 (July 1971) pp.764-775
(5) J.D. Feldman and O.A. Reimann. RADCAP : An Operational Parallel Processing Facility. AFIPS 1974 National Computer Conference Proceedings, Vol. 43, pp. 7-15.
(6) C.C. Foster and R. Gonter. Conditional Interpretation of Operation Codes. IEEE Trans. on Comp, Vol. C-20, No. 2, (January 1971), pp. 108-111.
(7) A. Weinberger. Hybrid Associative Memory Concept. Computer Design, January 1971.
(8) W.H. Kautz. Cellular Logic-in-Memory Arrays. Trans. on Comp., Vol. C-18, No. 8, (August 1969), pp. 719-727.

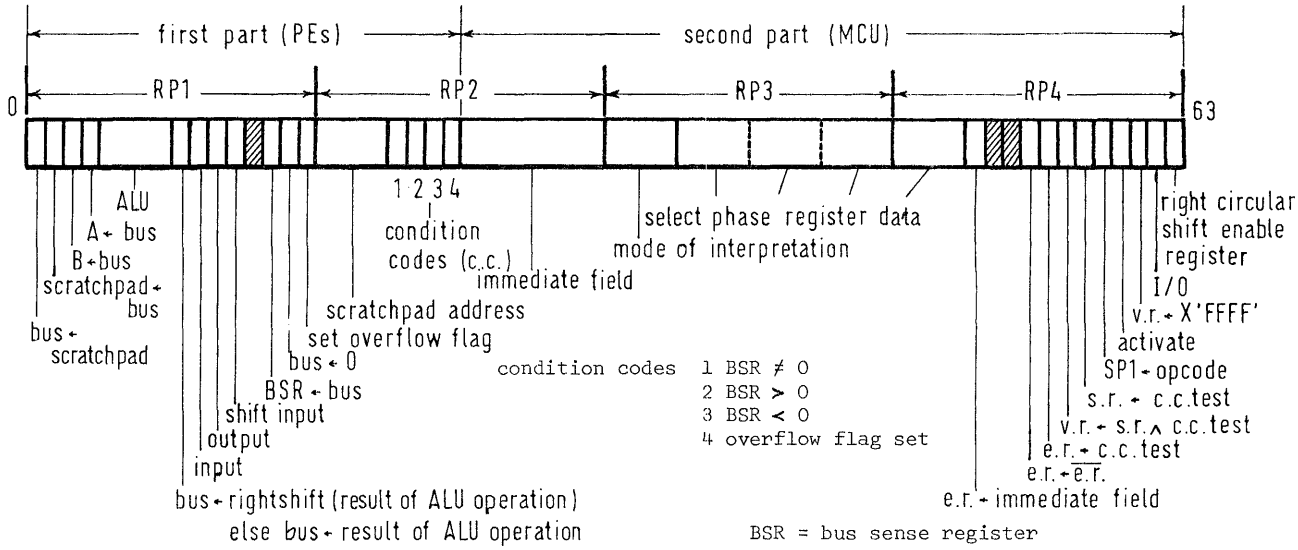


Fig.1. Microinstruction.

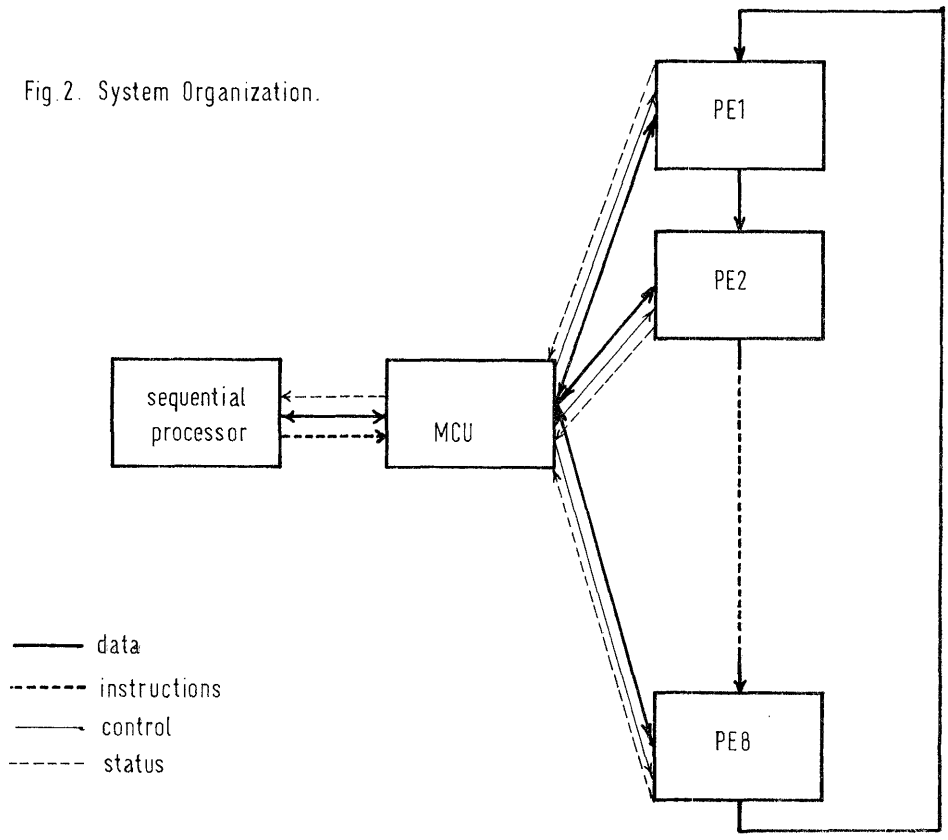


Fig.2. System Organization.

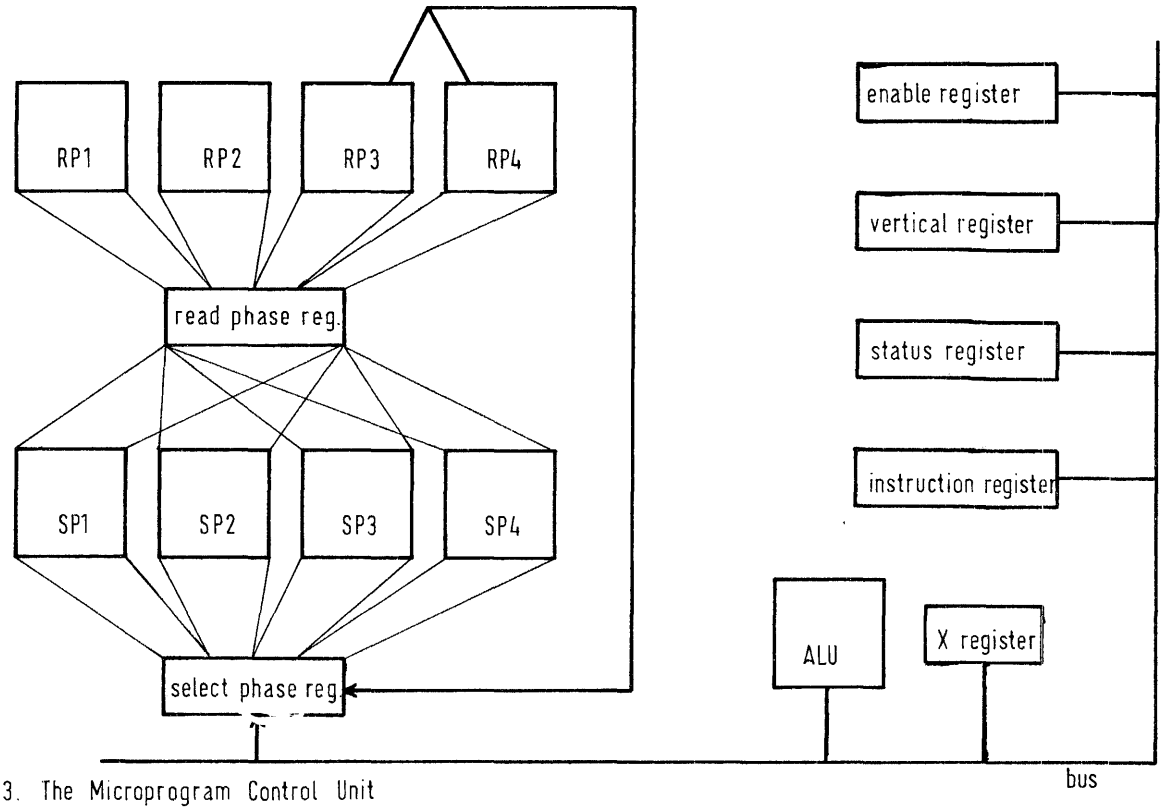


Fig.3. The Microprogram Control Unit

Fig.4. Microprogram (for I/O)

	SP1	SP2	SP3	SP4
0	0000000000000000	0010000011110000	0000000000010000	0000000100000001
1	0010000000000000	0000000011100000	0000000000100000	0000001000000010
2	0001000000000000	0000000011010000	0000000000110000	0000001100000011
3	0000000000000000	0000000011000000	0000000001000000	0000010000000100
4	0000000000000000	0000000010110000	0000000001010000	0000010100000101
5	0000000000000000	0000000010100000	0000000001100000	0000011000000110
6	0000000000000000	0000000010010000	0000000001110000	0000011100000111
7	0000000000000000	0000000010000000	0000000010000000	0000100000001000
8	0000000000000000	0000000001110000	0000000010010000	0000100100001001
9	0000000000000000	0000000001100000	0000000010100000	0000101000001010
10	0000000000000000	0000000001010000	0000000010110000	0000101100001011
11	0000000000000000	0000000001000000	0000000011000000	0000110000001100
12	0000000000000000	0000000000110000	0000000011010000	0000110100001101
13	0000000000000000	0000000000100000	0000000011100000	0000111000001110
14	0000000000000000	0000000000010000	0010000011110000	0000111100001111
15	0000000000000000	0000000000000000	0010000011111111	0000000000000000

	RP1	RP2	RP3	RP4
0	0000000000000000	0000000000000000	0000000011110000	0000000000010011
1	1000000000100000	0001000000000000	0000000011110001	0001000000010010
2	0100000001000000	0010000000000000	0000000011110010	0010000000010010
3	0000011100100000	0011000000000000	0000000011110011	0011000000010010
4	0000000000000000	0100000000000000	0000000011110100	0100000000010010
5	0000000000000000	0101000000000000	0000000011110101	0101000000010010
6	0000000000000000	0110000000000000	0000000011110110	0110000000010010
7	0000000000000000	0111000000000000	0000000011110111	0111000000010010
8	0000000000000000	1000000000000000	0000000011111000	1000000000010010
9	0000000000000000	1001000000000000	0000000011111001	1001000000010010
10	0000000000000000	1010000000000000	0000000011111010	1010000000010010
11	0000000000000000	1011000000000000	0000000011111011	1011000000010010
12	0000000000000000	1100000000000000	0000000011111100	1100000000010010
13	0000000000000000	1101000000000000	0000000011111101	1101000000010010
14	0000000000000000	1110000000000000	0000000011111110	1110000000010010
15	0000000000000000	1111000000000000	0000000000000000	1111000000010010

RAYTHEON
10-0541 (1-65)

INTERNAL MEMORANDUM

To

Date

From

Subject

Shannon, C. E. "A Mathematical Theory of Communication," Bell System Tech. J. 27 (1948) 379-423, 623-656

INSTRUCTION DESIGN TO MINIMIZE PROGRAM SIZE*

James F. Wade
Computer Science Department

Paul D. Stigall
Electrical Engineering and Computer Science Departments
University of Missouri-Rolla
Rolla, Missouri 65401

Introduction

With the introduction of dynamic microprogrammer computers, it becomes practical to reconfigure the architecture of a computer to more efficiently represent programs. A number of methods have been proposed and investigated for reducing the redundancy of computer op codes, address and data items. The greatest gains have come from frequency based encoding. Wilner[9] reports 40 to 70 percent reduction of program size using frequency based encoding for instructions and addresses combined with tailored instruction sets.

This paper examines the effect on program size of various architectural features assuming frequency based encoding. It is primarily oriented toward the reduction of program memory by selection of instructions and features to microcode, but applicable to the structure of the underlying micro machine and the compressed storage of any type of symbol string.

Simplest Information Theory Model

A program is assumed to consist of a string of symbols from a given alphabet A of N symbols. Each symbol S_i occurs with probability P_i. The probability of occurrence of a symbol is assumed to be independent of previous symbols.

From information theory[1] the information per symbol in bits is given by $-\text{Log}_2(P_i)$. The average information per symbol (entropy) $H = -\sum_{i=1}^{N} P_i \text{Log}_2(P_i)$. The lower bound on the number of bits needed to represent a string of L symbols under these assumptions is then LH. If symbols are coded individually as variable length bit strings with unique prefix encoding, the most efficient encoding is given by Huffman[3].

Evaluation Method

It will be assumed that the symbol string is encoded minimally redundant assuming independence. After the technique under study is applied the resulting string is assumed to be re-encoded. Thus the method of evaluation becomes computing the entropy of the string after the reduction and comparing with the entropy of the string before reduction. If there is a change in the number of symbols required to represent the string, the entropy is adjusted to normalize to the length of the original string.

*Supported in part by a National Science Foundation Grant, GJ-32596.

Replacement of Repeated Strings

The simplest and most natural process for reducing the length of a symbol string is the identification of repeated symbol strings and the introduction of a new symbol to replace the string. This is a simple model of subroutines, macros, and the introduction of new instructions.

If one assumed the string $S_1S_2\ldots S_k$ occurs with probability P_s and is replaced by the new symbol S' wherever it occurs, then the new value of entropy is given by:

$$H' = -\frac{P_s}{1-(k-1)P_s}\,\text{Log}\left(\frac{P_s}{1-(k-1)P_s}\right) - \sum_{i=1}^{k}\frac{P_i-P_s}{1-(k-1)P_s}\,\text{Log}\left(\frac{P_i-P_s}{1-(k-1)P_s}\right) - \sum_{i=k+1}^{N}\frac{P_i}{1-(k-1)P_s}\,\text{Log}\left(\frac{P_i}{1-(k-1)P_s}\right). \quad (1)$$

The number of symbols needed to represent a string is reduced by the factor $1-(k-1)P_s$. The resulting change in entropy normalized to the length of the original string is given by:

$$Ha - H = (1-(k-1)P_s)\,H' - H. \quad (2)$$

substituting (1) in (2) gives

$$Ha - H = -P_s\,\text{Log}\left(\frac{P_s}{1-(k-1)P_s}\right) - \sum_{i=1}^{k}(P_i-P_s)\,\text{Log}\left(\frac{P_i-P_s}{1-(k-1)P_s}\right) - \sum_{i=k+1}^{N}P_i\,\text{Log}\left(\frac{P_i}{1-(k-1)P_s}\right) + \sum_{i=1}^{N}P_i\,\text{Log}\,P_i.$$

Which reduces to:

$$Ha - H = (1-(k-1)P_s)\,\text{Log}(1-(k-1)P_s) - \sum_{i=1}^{k}P_i\left(1-\frac{P_s}{P_i}\right)\text{Log}\left(1-\frac{P_s}{P_i}\right) + P_s\sum_{i=1}^{k}\text{Log}\,P_i - P_s\,\text{Log}\,P_s. \quad (3)$$

To determine the value of P_s such that replacement results in a decrease of entropy, solve the inequality $H_a-H<0$ for P_s. For an approximate solution to the resulting nonlinear equation, use the Taylor expansion:

$$(1-x)\,\text{Log}_e(1-x) = -x + \frac{x^2}{2} + \text{H.O.T.}$$

to simplify the first two terms in (3). This gives

$$0 > P_s \left(\sum_{i=1}^{k} \text{Log}_e P_i - \text{Log}_e P_s + 1 + \frac{(k-1)^2 P_s}{2} - \sum_{i=1}^{k} \frac{P_s}{2P_i} + \text{H.O.T.} \right).$$

Rearranging yields:

$$\text{Log}_e P_s > \sum_{i=1}^{k} \text{Log}_e P_i + 1 + \frac{(k-1)^2 P_s}{2} - \sum_{i=1}^{k} \frac{P_s}{2P_i} + \text{H.O.T.} \qquad (4)$$

For small P_s, the Log terms dominate giving:

$$P_s > e \prod_{i=1}^{k} P_i . \qquad (5)$$

Thus a decrease in entropy results if $P_s > 2.72 \prod_{i=1}^{k} P_i$.

This result quantifies the intuituve idea that a rarely occurring long string of symbols can often be replaced by a new symbol and save memory, while a shorter string must occur more often to result in savings.

A somewhat less obvious result is that a rarely occurring symbol which can be replaced by a sequence of very frequently occurring symbols can result in a storage saving. This compares with Fosters' observation[8] that a few instructions make up the greatest percentage of most programs and elimination of infrequently used instructions causes little loss of programming power.

Processor State Change

The second major technique of minimization is the setting of a state which modifies the interpretation of the symbols that follow it. This models changing processor states, changing control program, base register addressing, address bank selection, op code grouping, shift level coding, and other similar techniques.

Let a string of L symbols from an alphabet of size N be partitionable into K contiguous groups such that in each group there is a set U_j of B symbols which appear in no other group. To encode the string, a new symbol is introduced to identify each group and inserted before each group in the string. Each symbol, $S_{ij} \varepsilon U_j$ is replaced by a new symbol S'_i. For notational convenience, let P_{ij} the probability of occurrence of $S_{ij} \varepsilon U_j$, let P_{io} for $1 \leq i \leq M = N - KB$ be the probability of occurrence of each symbol not in any set U_j. Note that

$$\sum_{i=1}^{B} \sum_{j=1}^{K} P_{ij} + \sum_{i=1}^{M} P_{io} = 1 . \qquad (6)$$

The entropy before combining symbols is:

$$H = - \sum_{i=1}^{B} \sum_{j=1}^{K} P_{ij} \text{ Log } P_{ij} - \sum_{i=1}^{M} P_{io} \text{ Log } P_{io} .$$

To calculate the entropy after the encoding, note that the probability of occurrence of a symbol will be reduced by the factor $\frac{L}{L+K}$ by the introduction of the K switch symbols. Thus the probability of occurrence of S'_j becomes

$$\sum_{j=1}^{K} \frac{L}{L+K} P_{ij} . \qquad (7)$$

The entropy of the string after encoding is:

$$H' = - \sum_{i=1}^{B} \left(\sum_{j=1}^{K} \frac{LP_{ij}}{L+K} \right) \text{Log} \left(\sum_{j=1}^{K} \frac{LP_{ij}}{L+K} \right) - \sum_{i=1}^{M} \frac{LP_{io}}{L+K} \text{ Log } \frac{LP_{io}}{L+K}$$

$$-K \left(\frac{1}{L+K}\right) \text{ Log } \left(\frac{1}{L+K}\right) .$$

To compare, it is necessary to adjust H' for the additional symbols introduced. Adjusted entropy is:

$$H_a = \frac{L+K}{L} H', \text{ or}$$

$$H_a = - \sum_{i=1}^{B} \left(\sum_{j=1}^{K} P_{ij} \right) \text{Log} \left(\sum_{j=1}^{K} \frac{LP_{ij}}{L+K} \right) - \sum_{i=1}^{M} P_{io} \text{ Log } \left(\frac{LP_{io}}{L+K}\right) - \frac{K}{L} \text{Log} \left(\frac{1}{L+K}\right) .$$

Isolating terms involving $\frac{L}{L+K}$ yields

$$H_a = - \sum_{i=1}^{B} \left(\sum_{j=1}^{K} P_{ij} \right) \text{Log} \left(\sum_{j=1}^{K} P_{ij} \right) - (\text{Log } \frac{L}{L+K}) \sum_{j=1}^{B} \sum_{j=1}^{K} P_{ij} - \sum_{i=1}^{M} P_{io} \text{ Log } P_{io} - (\text{Log } \frac{L}{L+K}) \sum_{i=1}^{M} P_{io} - \frac{K}{L} \text{Log} \left(\frac{1}{L+K}\right) . \qquad (8)$$

Recalling (6) that $\sum_{i=1}^{B} \sum_{j=1}^{K} P_{ij} + \sum_{i=1}^{M} P_{io} = 1$, and substituting in (8) yields:

$$H_a = - \sum_{i=1}^{B} \left(\sum_{j=1}^{K} P_{ij} \right) \text{Log} \left(\sum_{j=1}^{K} P_{ij} \right) - \sum_{i=1}^{M} P_{io} \text{ Log } P_{io} - \text{Log } \frac{L}{L+K} - \frac{K}{L} \text{Log} \left(\frac{1}{L+K}\right) .$$

The last two terms may be regrouped to

$$\frac{K}{L} \text{ Log } L + \left(1+\frac{K}{L}\right) \text{Log} \left(1+\frac{K}{L}\right) .$$

The change in entropy becomes

$$H_a - H = -\sum_{i=1}^{B}\left(\sum_{j=1}^{K} P_{ij}\right) \operatorname{Log}\left(\sum_{j=1}^{K} P_{ij}\right) + \sum_{i=1}^{B}\sum_{j=1}^{K} P_{ij} \operatorname{Log} P_{ij}$$

$$+ \frac{K}{L} \operatorname{Log} L + \left(1 + \frac{K}{L}\right) \operatorname{Log}\left(1 + \frac{K}{L}\right). \qquad (9)$$

Examining equation (9), we note that the first two terms represent the savings generated by combining symbols, and the last two terms represent the cost of introducing the K switch symbols. The first term is sensitive to the way the symbols in each independent group are paired. If the S_{ij} are sequenced so that $P_{1j} \geq P_{2j} \geq \ldots \geq P_{3j}$ for every j, then the first term in (9) will be minimized.

Under the above ordering assumption a more usable expression can be derived by using an approximation. The ratios $P_{i1}:P_{i2}:\ldots:P_{iK}$ can be assumed to be approximately independent of i. Thus $P_{ij} \simeq Q_i\ (1 + D_j)$. Where $Q_i = \frac{1}{K}\sum_{j=1}^{K} P_{ij}$ = average of P_{ij} over j. And

$$D_j = \frac{\frac{1}{B}\sum_{i=1}^{B} P_{ij}}{\frac{1}{BK}\sum_{i=1}^{B}\sum_{j=1}^{K} P_{ij}} - 1$$

Using this approximation:

$$H_a - H = -\sum_{i=1}^{B} KQ_i \operatorname{Log} KQ_i + \sum_{i=1}^{B}\sum_{j=1}^{K} Q_i(1+D_j) \operatorname{Log}(Q_i(1+D_j)) +$$

$$\frac{K}{L} \operatorname{Log} L + \left(1 + \frac{K}{L}\right) \operatorname{Log}\left(1 + \frac{K}{L}\right).$$

Now $\sum_{i=1}^{B} Q_i = \frac{1}{K}\sum_{i=1}^{B}\sum_{j=1}^{K} P_{ij}$, and $\sum_{j=1}^{K} D_j = 0$.

Regrouping and substituting yields:

$$H_a - H = \frac{1}{K}\left(\sum_{i=1}^{B}\sum_{j=1}^{K} P_{ij}\right)\left(-K \operatorname{Log} K + \sum_{j=1}^{K} (1+D_j) \operatorname{Log}(1+D_j)\right)$$

$$+ \frac{K}{L} \operatorname{Log} L + \left(1+\frac{K}{L}\right) \operatorname{Log}\left(1+\frac{K}{L}\right). \qquad (10)$$

Setting $H_a - H < 0$, and rearranging yields:

$$\sum_{i=1}^{B}\sum_{j=1}^{K} P_{ij} \geq \frac{\frac{K}{L} \operatorname{Log} L + \left(1+\frac{K}{L}\right) \operatorname{Log}\left(1+\frac{K}{L}\right)}{\operatorname{Log} K - \frac{1}{K}\sum_{j=1}^{K} (1+D_j) \operatorname{Log}(1+D_j)}. \qquad (11)$$

Equation (11) gives the conditions for a savings in storage. Table 1 is a tabulation of equation (11). All D_j are assumed to be 0. This gives a lower bound on the break even point. For distributions with D_j of significant size, the values will be somewhat higher.

NUMBER OF GROUPS (K)

L	2	3	4	6	8	10
10	.98	-	-	-	-	-
20	.58	.56	.59	.69	.80	.91
50	.28	.27	.29	.33	.38	.43
100	.16	.15	.16	.19	.22	.25
200	.09	.09	.09	.11	.12	.14
500	.04	.04	.04	.05	.06	.06

Table 1. Break Even Point

It is seen that for a short string, the symbols involved in the switch must account for most of the probability before any savings in storage is realized. For longer strings the break even point is quite low.

$\Sigma\Sigma\ P_{ij}$

L	.2	.4	.6	.8	1.0
10	-	-	-	-	.02
20	-	-	.02	.22	.42
50	-	.12	.32	.52	.72
100	.04	.24	.44	.64	.84
200	.11	.31	.51	.71	.91
500	.16	.36	.56	.76	.96

Table 2. Maximum Savings in Bits/Symbol for Number of Groups (K) = 2.

Table 2 is a tabulation of equation (10) for K = 2. Assuming $D_1 = D_2 = 0$ gives an upper bound on the savings. Using Log base 2 will give the savings in bits per symbol.

A savings of 0.5 bits/symbol represents about a 10 per cent reduction of a typical program. Thus we see that this type of feature saves significant amounts of storage only if the group sizes are on the order of 100 symbols.

Limitations and Extensions

An important limitation is the ignoring of implementation cost. While one may rationalize that the cost of hardware is rapidly decreasing, the implementation of features that result in very small savings is not cost effective.

The derivations can be modified to more closely model specific architectural features for more precise results.

Conclusions

This type of analysis helps evaluate the worth of a particular type of architectural feature. It shows what memory savings are possible over those obtained by frequency

based encoding. It finds the limits on memory savings for a particular type of feature, and the conditions under which the most savings are realized.

References

1. Shannon, C. E. "A Mathematical Theory of Communication," Bell System Tech. J. 27 (1948), 379-423, 623-656.
2. Oliver, B. M., "Efficient Coding," Bell System Tech. J. 21, 4 (July 1952), pp. 724-750.
3. Huffman, D. A. "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE 40 (1952) pp. 1098-1101.
4. Schwartz, E. S., "A Dictionary for Minimum Redundancy Encoding," J. ACM 10, 4 (Oct. 1963) pp. 413-439.
5. Schwartz, E. S. and Kallick B., "Generating a Cannonical Prefix Encoding," CACM 7, 3 (Mar. 1964).
6. Young, J. F., Information Theory, Wiley Interscience, New York, 1971.
7. Foster, C. C., and Gonter, R. H., "Conditional Interpretation of Operation Codes," IEEE Trans. Computers C-20, 1 (Jan. 1971) pp. 108-111.
8. Foster, C. C., Gonter, R. H., and Riseman, E. M., "Measures of Op-Code Utilization," IEEE Trans. Computers C-20, 5 (May 1971) pp. 582-584.
9. Wilner, W. T., "Burrough B1700 Memory Utilization," AFIPS Conference Proceedings 41, (Dec. 1972) pp. 579-586.

Appendix

Example of Frequency Based Coding

A tabulation of 1900 opcodes from text of programs for the MICRODATA 1621 computer was made. The MICRODATA 1621 has 8 bit opcode and variable length address and immediate fields. Of the approximately 250 defined opcodes, 132 appeared in the sample. A tabulation of results using Huffman coding with different number bases and encoding into fixed size bit fields is given. The split into bit fields is chosen to minimize average bits/symbol for the given maximum length.

	Average Bits/Symbol	Max Length in Bits
Entrophy	5.90	———
Huffman		
base 2	5.93	11
base 4	6.02	12
base 8	6.19	12
base 16	6.32	12
bit fields		
5 - 4	6.67	9
6 - 4	6.59	10
6 - 5	6.61	11
6 - 6	6.66	12
5 - 2 - 2	6.48	9
5 - 2 - 3	6.25	10
5 - 3 - 3	6.18	11
5 - 3 - 4	6.16	12
5 - 3 - 5	6.16	13
5 - 3 - 6	6.16	14

HMO, A HARDWARE MICROCODE OPTIMIZER

James O. Bondi
Department of Electrical Engineering

Paul D. Stigall
Departments of Electrical Engineering and Computer Science

University of Missouri-Rolla
Rolla, Missouri 65401

Abstract

This paper discusses an algorithm for optimizing the density and parallelism of microcoded routines in microprogrammable machines. Besides the algorithm itself, the algorithm's uses, adaptability to conventional machine characteristics, and architectural requirements are also discussed and analyzed. Even though the paper proposes a hardware implementation of the algorithm, the algorithm is viewed as an integral part of the entire microcode generation and usage process, from initial high-level input into a software microcode compiler down to machine-level execution of the resultant microcode on the host machine. It is believed that, by removing much of the traditionally time-consuming and machine-dependent microcode optimization from the software portion of this process, the algorithm can improve the overall process.

Introduction

Since the advent of microprogrammable machines in recent years, a frenzy of research has occurred on developing good software compilers to generate user-designed microprograms, or microcode, for chosen target machines [1], [2]. The traditional argument against such compilers is that they will never be able to generate the completely compact microcode needed in a typical high-usage microprogram. The traditionalists thus conclude that the tedious and complex task of microprogramming is best left solely to the hardware designers [3], [4], [5], [6]. On the other hand, many machine users have long desired a machine whose instruction repertoire they could tailor to their particular needs [5], [6]. These users argue that a microprogram compiler would drastically reduce microcode production time, thus making even medium-to-low usage, less highly compact microprograms practical [4].

Two important characteristics usually sought by proponents of such compilers are (1) a powerful, high-level input language and (2) a high degree of target-machine independence for the user. Typical versions of such compilers are structured in two basic phases conducive to these characteristics. The first phase is a complete compiler taking high-level input source into intermediate-level text. The second phase is a simple, direct translator chosen by the user to transform this intermediate text into actual microcode for his target machine [3], [7].

Although microprogram compilers such as those just mentioned have proved quite promising, one particularly annoying problem remains. This problem is the compactness, or degree of optimization, of the microcode output versus the required compilation time. To be feasible, even medium-to-low-usage microprograms require a fair degree of optimization. Furthermore, such microprograms require short compilation times to make them worthwhile producing. These two requirements are inherently conflicting, especially since microprograms and their formats are traditionally highly target-machine-dependent while the compiler attempting to optimize these microprograms is designed to be highly target-machine-independent. In other words, it is extremely difficult to efficiently optimize a machine-dependent process by means of a machine-independent mechanism [2], [7], [8].

One possible solution to this problem is to relieve the microprogram compiler of a large part of its optimization chores. The authors propose moving many local optimization duties out of the compiler and across the software-hardware boundary into the hardware realm of the target machine. The hardware microcode optimizer, HMO, is a simple hardware algorithm capable of condensing a sequence of essentially horizontal microinstructions to increase their bit density and parallelism. It is reasonable to expect that a hardware implementation of such a hardware-dependent process can be both fast and cost-effective [9]. Furthermore, by improving the efficiency of software microprogram compilers, the HMO algorithm can increase the practicality of a truly user-microprogrammable computer system.

It must be stressed that the overall microcode optimization process being proposed in this paper would consist of two basic levels, or phases. The first level, performed by the software microprogram compiler, would be the more complex, global, primarily machine-independent type of optimization procedures. The second level, performed by the HMO algorithm and associated hardware (after receiving the software compiler's generated microcode), would consist ideally of as much as possible of the less complex, local, highly machine-dependent type of optimization.

I. Description of Basic HMO Algorithm

Consider how the major internal hardware components of a computer are involved with the flow of data, or information, throughout the machine. With respect to the HMO algorithm, the following classification of such components is useful: 1) a fixed source, or data constant (e.g., a pseudo-register which supplies a hardwired constant of 0 or 1 to other components), 2) a data transformer (e.g., an adder, shifter, working register, main memory during a load-from-memory instruction, etc.), or 3) a data sink (e.g., main memory during a store-into-memory instruction). However, since the production of data constants

is a fixed operation, with no inputs on which to perform a function, HMO need not be concerned with such constants. Their control is inherently covered in the control of the transformers and sinks to which they supply inputs.

Concerning the control of active, functional components such as transformers and sinks, two major areas of interest are the supplying of inputs and the calling for outputs, with only the former area actually being needed for sinks. If we consider now a flexible microprogrammable architecture such as that shown in Fig. 1, these two areas become nothing more than particular groups of horizontal microinstruction bits controlling appropriate register transfers. One other area of interest for both transformers and sinks is timing, or the time interval required for them to complete their respective functions. This timing requirement implies a certain needed minimal distance between some microinstructions, or microwords, in any microinstruction stream. Assume for now that the microcycle time of HM1 in Fig. 1 is such that this needed distance is only one microcycle. This means, for example, that it is acceptable for one microword to excite an adder "input supply" and the microword immediately following to excite the corresponding adder "output call."

Notice that the "latching" type architecture of HM1 affords the microprogrammer virtually complete timewise independence of when inputs are supplied to a data transformer such as the adder. He may, in fact, "latch" in adder inputs during different microcycles. All he must do is make certain all desired inputs are fed at least one microcycle before he calls for the corresponding transformer output. Thus, the HMO algorithm can simply sequence through a stream of microinstructions, condensing (essentially combining) all microinstructions containing "input supply" bits into one instruction, until it reaches the point where the next instruction contains an "output call" bit corresponding to the already condensed "input supplies." At this point, the algorithm must temporarily stop condensing, save (or execute) the newly formed condensed instruction, and then proceed to condense again starting with the next microinstruction in the stream. What all this means is that the HMO algorithm can produce, from a microinstruction stream which exercises HM1's hardware in a purely serial fashion, a corresponding condensed stream which exercises HM1's hardware in a highly parallel fashion.

Unlike data transformers, data sinks, which don't require "output call" bits, make it difficult for the HMO algorithm to spot the point where condensing must temporarily stop. This problem can be solved by requiring that, following the desired sink inputs, a succeeding microinstruction appear containing a "1" bit which actually excites, or causes, the sinking of these preceding inputs. By controlling sinks in this manner, these sinks appear identical to data transformers as far as the HMO algorithm is concerned. It always sees a series of "input supplies" followed at least one microcycle later by a microword containing a control bit which, for transformers, calls for passage of the transformed data to some other point and, for sinks, causes the actual sinking action to be performed. Therefore, the HMO algorithm can now handle transformers and sinks with equal facility. The major hardware needed is a simple set of combinational logic "inhibit" functions which are driven both from the condensed instruction being formed and from the next instruction in the stream. At least one of these functions is activated when the next instruction contains an "output call" corresponding to "input supplies" in the condensed instruction. Further condensing is thus inhibited and the algorithm starts anew on the next instruction.

Note that Fig. 2 allows the option of either saving a condensed result for later use (pre-pass compilation) or executing this result immediately without saving it (interpretive execution). Interpretive execution would be inefficient for all but extremely low-usage microprograms, as it would require repeated condensing of repeatedly executed blocks of microcode. Therefore, all discussion that follows assumes that the HMO algorithm is being used as a pre-pass condensing compiler.

Fig. 3 contains two examples illustrating the algorithm's use. Note that the second example illustrates how the authors would ideally like to handle conditional branch microinstructions. This ideal method would be essentially to allow the HMO algorithm to condense "past" conditional branches along one of the two available paths (hopefully, the "non-branch" path, or path expected to be taken most of the time). Then, later, the algorithm could be restarted separately along the yet untouched (hopefully "branch") path.

Finally, Fig. 4 depicts one example of the "inhibit" functions which provide the logical signals to control the HMO algorithm. These figures should be studied before proceeding to the architectural discussion of Section II, as that discussion leans heavily upon their contents.

II. Architectural Requirements

Although necessarily brief, the Section I. HMO algorithm description was of sufficient depth to support the following discussion of architectural characteristics dictated as desirable for easy and efficient usage of the algorithm.

A. General Characteristics

1. Two-Step Structure. One architectural requirement for HM1 (Hypothetical Machine 1), already hinted at in Section I., is that all basic operations under microprogrammed control consist of two elementary steps. These two steps, for a data transforming unit such as the adder, were quite naturally chosen as "the supplying of inputs" followed by the "calling for outputs." However, for a data sinking operation such as storing data into main memory, two such steps were not found so naturally.

In fact, the author's original scheme for main memory was different from that now shown in Fig. 1. Originally, the MIR register (now used to accept the passage of data to be stored) did not exist. Instead, a "READ" and a "WRITE" flip-flop existed. As seen by the microprogrammer, a "store-into-memory" operation was then done in only one step by passing the MAR an address, the MBR the data to be stored, and the "READ" and "WRITE" flip-flops a "0" and a "1" respectively. The actual details of storing the data were then handled

by the memory controller during the next microcycle. Hardwarewise, this scheme indeed worked, but examples were found for which the HMO algorithm yielded condensed code not equivalent to the original uncondensed code.

This problem was finally eliminated by the hardware now shown in Fig. 1, for which a "load-from-memory" and a "store-into-memory" operation are both seen by the HMO algorithm as two-step operations. In other words, the hardware of HM1 must be such that all elementary operations under microprogrammed control consist of a "starting" step and a "finishing" step.

2. Latching Architecture. Another desirable architectural feature suggested in Section I. is the "latching" type architecture shown in Fig. 1, where "latching" refers to the use of, even on a unit such as the adder which could have its operand inputs fed directly from the source registers during a given add [3], a separate set of operand-holding input latches. Not only does this structure permit the microprogrammer (and software compiler) much hardware-timing independence (of when adder inputs are supplied for a given add, for example [10]), but it also helps to readily break down operations (such as addition) into the two necessary "starting" and "finishing" steps mentioned previously. Another possible advantage of this "latching" type structure will be pointed out later.

B. Microinstruction Formats

As might be expected, the HMO algorithm dictates certain characteristics as desirable in the area of microinstruction formats.

1. Control Section Encoding. First, in the area of control section formats, the algorithm is most easily implemented from a horizontal, virtually completely unencoded control section, having essentially one bit present for each possible register transfer. With such an unencoded control format, the inhibit functions (Fig. 4) can be driven directly from the control register (Fig. 2) and from the control memory output lines feeding the control register. While a typical encoding scheme such as encoding the mutually exclusive input sets of each hardware register, or latch, can be employed, the result is that the "output calls" section (Fig. 4) of each inhibit function can no longer be driven directly from the control memory output lines. This is true simply because a coding scheme which encodes according to register input sets does not possess sufficient information to directly feed the "output calls" section of an inhibit function. In summary, then, a horizontal, unencoded control section format permits, for inhibit functions, the easiest, fastest realization having the fewest levels of logic.

2. Basic Addressing Flexibility. Similarly, in the other area of microinstruction formats, namely addressing formats, the algorithm again dictates certain characteristics as desirable. Consider the examples of Fig. 3. They illustrate that the HMO algorithm, starting from a particular point, always condenses as many microinstructions as possible into one condensed instruction and then restores that condensed instruction for later use. However, what Fig. 3 does not show is that the algorithm always restores each condensed result in the location of the first, or "top," instruction of the original group of uncondensed instructions. This restoration at the "top" is necessary to assure that the rest of the instructions in each original group of uncondensed instructions remain intact between the restored condensed results. Research [11] has shown that these instructions which are left intact between restored condensed results, although they appear to be garbage, may prove to be needed if one of them is looped back to from some point later in the microprogram. Even when the algorithm is completely through with its one and only condensing pass, there is no real need to remove these garbage instructions, as the restored condensed results have been linked together so that, during execution, the garbage instructions are circumvented by a series of "leap frog" style jumps.

It should thus be obvious that more addressing flexibility is needed than would be provided by using a microprogram counter register similar to the program counter register, or PGC, usually used for addressing machine instructions in main memory. Such a counter, or pointer, register is readily suited to mostly-sequential addressing, but not to the "leap frog" style addressing just mentioned. At the very least, a microinstruction format having one complete "next address" in each word of control memory is needed (see Fig. 2).

3. Conditional Branch Dictates. Consider the second example of Fig. 3 which, remember, illustrates an extreme, idealistic scheme for handling conditional branch microinstructions. Note that when the conditional branch instruction was encountered, condensing proceeded down a selected, "favored" path, as suggested at the end of Section I. Further note that the condensed code on the right has, in its restored conditional branch instruction, the two transfers "AI1←PGC" and "AI2←0." The astute reader will notice that these two transfers will always be executed in the condensed code, no matter which path is taken, whereas, in the original uncondensed code, they would have been executed only if the "favored" path were taken. Thus, along the other, unfavored path (here FETCH), the program state when using the condensed code is slightly different from the program state when using the uncondensed code. This situation could obviously cause errors and unexpected results. However, this situation can be remedied by increasing the width of each control memory word, allowing room to place the bits representing these two transfers in a separate conditional control section which is executed only if the "favored" path is taken during execution.

Research [11] has also shown that, in order to condense, during the formation of one condensed result, not only "up to and including" a conditional branch instruction but "past" it as well (as in Fig. 3), two completely independent "next addresses" must be available to the conditional branch instruction. Again, this could be accomplished by further widening each control memory word to allow room for a second, optional "next address."

One possibility to consider at this point is a double-width scheme in which each word of control memory has room for essentially two complete control sections and two complete "next addresses." Such a scheme

would permit the storing of two regular or one conditional branch microinstruction in each control memory word. Indeed, this scheme is feasible from a hardware standpoint, the major requirement being the capability of reading or writing "single-length" or "double-length" control memory words. The real problem with this scheme surfaces when one attempts to apply the HMO algorithm in its present simple, unrestricted form. Whenever the algorithm restored a condensed, "double-length", conditional-branch result, it would generally be destroying one "single-length" garbage instruction (discussed previously) and possibly trying to store this "double-length" result starting on an "odd" boundary, or the midpoint of a "double-length" control memory word (an action not permitted in some "single/double-length" addressing schemes, such as IBM 360/Model 50 main memory).

Significantly, this whole problem disappears if one makes the simple restriction of not allowing the algorithm to condense "past" a conditional branch instruction. With this restriction, many format schemes between the extremes already discussed become possible, even schemes for which the two "next addresses" are interrelated rather than completely independent. However, research [11] has shown that schemes employing two independent "next addresses" for conditional branch microinstructions are preferable from the standpoint of adaptability to the HMO algorithm. Just as an example, one workable scheme is the basic "single-length" scheme shown in Fig. 2 augmented by allowing, for conditional branch instructions, a choice between the "next address" found in the instruction itself and a fixed, hardwired optional "next address" (pointing, perhaps, to an "increment the program counter (PGC) and then go to FETCH" microroutine).

4. Summary. The HMO algorithm does not dictate an obvious best choice for a microinstruction format. Instead a myriad of possibilities exist, each with its own good and bad points concerning the tradeoff areas of microprogramming flexibility, algorithmic adaptability, and hardware complexity and efficiency.

C. Control Memory Characteristics

1. Microcycle Rate. It was suggested in Section I. that the microcycle time, or cycling speed of control memory, be such that one microcycle is sufficient time to complete all basic operations under microprogrammed control. While this "one-microcycle assumption" is not absolutely necessary, it does permit the simplest, neatest realization of the HMO algorithm. The alert reader will note, however, that this characteristic implies essentially that control memory must be cycled at least as slowly as the slowest basic operation in HM1 (say one complete main memory cycle for HM1 as presented in this paper). One reasonable technique for helping to achieve this "one-microcycle assumption" would be to use the same type (and speed) of memory, say core, for both main and (user) control memories, a type of design employed, in varying degrees, on real, production machines such as the IBM 360/Model 25 [12] and the Burroughs B 1700 [13].

Obviously, however, this "one-microcycle assumption" could, in the extreme, result in a microcycle time for HM1 which is prohibitively slow. Another promising technique would be to take advantage of the "latching" type architecture previously suggested for HM1 and use it to pipeline the slower operations of HM1 (whatever they may be) to a sufficient degree such that all micro-operations can be completed at least as quickly as one cycle of control memory. For example, by insisting that the AO1 register of Fig. 1 be a real, physical latching register (not assumed thus far), the overall process of addition (from operand source registers to result destination registers) would then consist of three elemental stages instead of the present two stages. That is, the overall addition process would then be a three-stage pipeline, with each stage being a smaller part of the entire process than is the case with the present two-stage pipeline.

Thus, this "one-microcycle assumption" is not necessarily as impractical as it may have originally seemed, the two techniques just discussed (and others [11], [14]) helping make this assumption reasonable and practical. Although modifications to the HMO algorithm itself [11], [14] can also be helpful, this discussion has concentrated on hardware, or architectural, means of achieving and improving the practicality of the "one-microcycle assumption."

2. Possible Structure. Of the many possible methods which could be used to actually implement the HMO algorithm, a firmware implementation's flexibility is particularly attractive. To achieve a firmware implementation, one feasible approach would be to employ two separate control memories (or at least two separate sections of one memory). One section would contain the HMO algorithm, loader routines, and other factory-fixed routines of no condensing interest to the algorithm. In other words, this section would essentially contain routines that the microprogrammer was forbidden to alter. The other section would basically contain the user-written microprograms [11]. Thus, during the run mode, the machine could be under the control of either control memory section, but not both. While performing the HMO algorithm, the machine would be under control of the user-forbidden section and would be operating on the user-accessible section to condense some user-written microprogram stored therein. Note that this type of control memory structure employing two separate sections, one being user-accessible and the other not, is found, in varying degrees, on real, production machines such as the Burroughs B 1700 [13] and the Microdata 1600 [15], a fact indicating the practicality of this approach.

Conclusion

This paper has proposed a hardware algorithm which could enable a microprogrammable machine to do its own local, machine-dependent optimization of user-written microprograms, leaving the global, machine-independent optimization to an associated software compiler. In fact, one software microprogram compiler could efficiently serve a group of logically different, but architecturally similar, machines, each possessing an implementation

of the HMO algorithm enabling it to do its own machine-dependent condensing and "cycle squeezing." Such a system should be the ideal environment for a software compiler which can efficiently serve several different machines but still present the user with a maximum degree of machine independence as he writes a microprogram for a particular, chosen machine.

In addition, this paper has discussed several of the architectural design implications of this HMO algorithm. It is encouraging to note that the architectural criteria arrived at are not exotic, expensive, impractical criteria, but instead, many are actually found on production machines, thus implying their cost effectiveness.

References

[1] R. K. Clark, "Mirager, the 'Best-Yet' Approach for Horizontal Microprogramming," Proceedings of ACM '72, Association for Computing Machinery, New York, 1972, pp. 554-560.

[2] M. Hattori, M. Yano, and K. Fujino, "MPGS: A High-Level Language for Microprogram Generating System," Proceedings of ACM '72, Association for Computing Machinery, New York, 1972, pp. 572-581.

[3] S. G. Tucker, "Microprogram Control for System/360," IBM Systems Journal, Vol. 6, No. 4, pp. 222-241, 1967.

[4] R. H. Eckhouse, Jr., "A High-Level Microprogramming Language (MPL)," AFIPS Conference Proceedings, Vol. 38 (SJCC 1971), pp. 169-177.

[5] R. F. Rosin, "Contemporary Concepts of Microprogramming and Emulation," Computing Surveys, Vol. 1, No. 4, pp. 197-212, Dec., 1969.

[6] M. J. Flynn and R. F. Rosin, "Microprogramming: An Introduction and a Viewpoint," IEEE Transactions on Computers, Vol. C-20, No. 7, pp. 727-731, July, 1971.

[7] S. S. Husson, Microprogramming: Principles and Practices. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1970, pp. 125-144.

[8] C. V. Ramamoorthy, M. Tabandeh, and M. Tsuchiya, "A Higher Level Language for Microprogramming," $MICRO_6$ The Sixth Annual Workshop on Microprogramming, College Park, Maryland, Sept., 1973 (Preprints), pp. 139-144.

[9] H. Falk, "Hard-Soft Tradeoffs," IEEE Spectrum, Vol. 11, No. 2, pp. 34-39, Feb., 1974.

[10] J. O. Bondi and P. D. Stigall, "HMO, An Integrated Hardware Microcode Optimizer," Proceedings of the Third Annual Texas Conference on Computing Systems, Austin, Texas, Nov., 1974 (Preprints).

[11] J. O. Bondi, Towards a Design of HMO, An Integrated Hardware Microcode Optimizer, University of Missouri-Rolla, Ph.D.E.E. Dissertation, Rolla, Missouri, Dec., 1974.

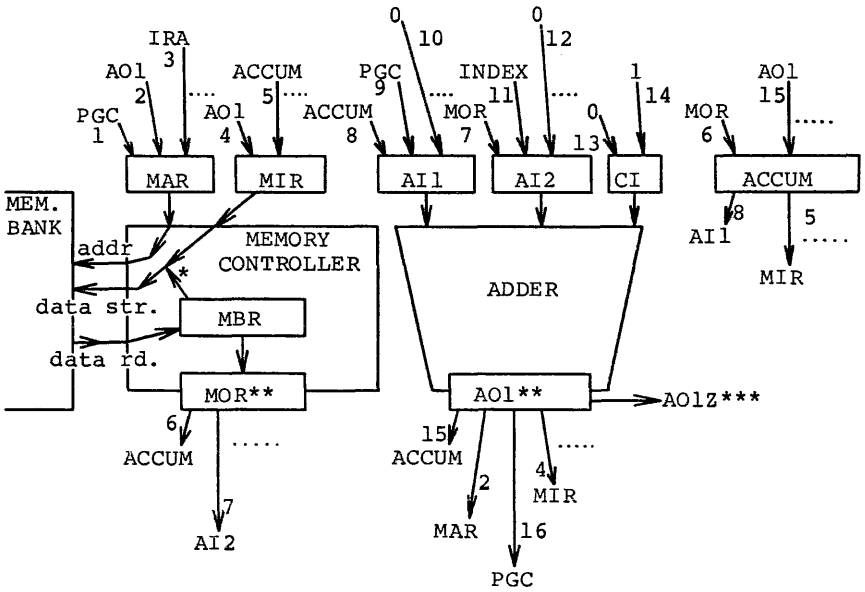
[12] C. G. Bell and A. Newell, Computer Structures: Readings and Examples, United States of America: McGraw-Hill, Inc., 1971, pp. 567-569.

[13] Burroughs B 1700 Systems Reference Manual, Preliminary Edition, Burroughs Corporation, Systems Documentation, Technical Information Organization, TIC-Central, Detroit, Michigan, 1972, pp. 1.7-1.8, 1.10, 3.1.

[14] J. O. Bondi and P. D. Stigall, "Designing HMO, An Integrated Hardware Microcode Optimizer," $MICRO_7$ The Seventh Annual Workshop on Microprogramming, Palo Alto, California, Sept., 1974 (Preprints), pp. 268-276.

[15] Microprogramming Handbook, Second Edition, Microdata Corporation, Santa Ana, California, 1971, pp. 317-318.

PGC - Program Counter, IRA - Instruction Register
Address Portion,
MIR - Memory Input Register,
MOR - Memory Output Register,
etc.



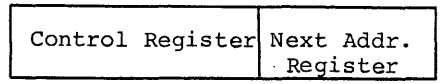
* Write cntrl bit determines gating of either MBR or MIR here.

** These can be real or pseudo registers.

*** This adder cond' code = 1 iff AO1 ≠ 0 (cond' code = 0 implies AO1 = 0). The algorithm can treat this cond' code as an adder output.

NOTE: The #'s indicate the microinstruction bit controlling a transfer.

Fig. 1 Subset of HM1 (Hypothetical Machine 1)



Master (Control) Register, or MCR
(Contains 1 Microword)

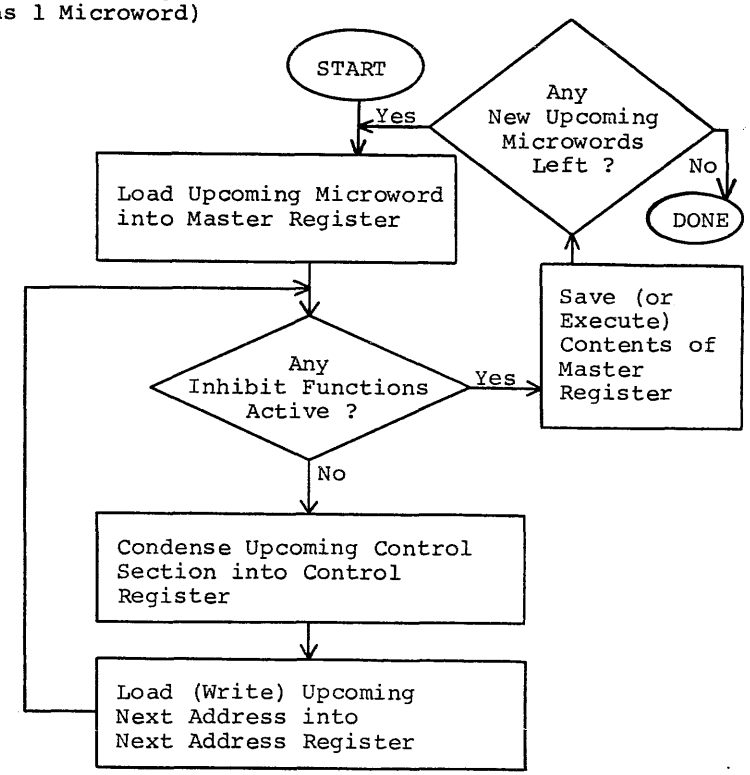


Fig. 2 Flow Chart of Basic HMO Algorithm

The following example illustrates condensing of an "add" with direct address that performs ACCUM←ACCUM + MEM(IRA);

```
1: MAR←IRA; to 2;                1: MAR←IRA; to 2;
2: AI2←MOR; to 3;      con-      2: AI2←MOR; AI1←ACCUM; CI←0;
3: AI1←ACCUM; to 4;    dense        to 5;
4: CI←0; to 5;
5: ACCUM←AO1; to FETCH;          5: ACCUM←AO1; to FETCH;
```

uncondensed microcode — condensed microcode

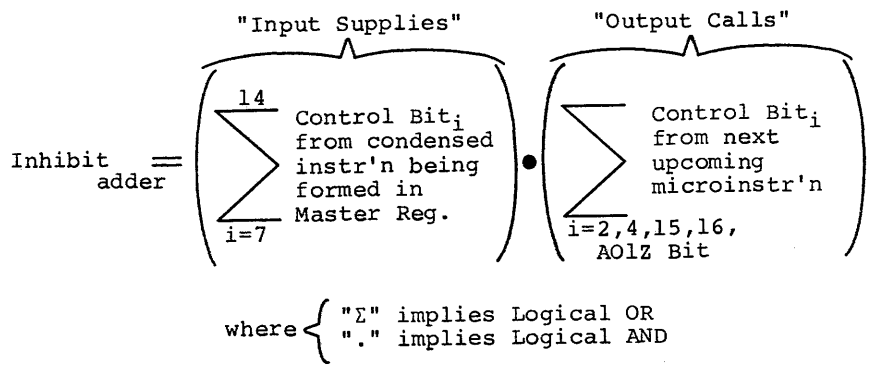
NOTE: The label #'s shown above are symbolically representative of control memory addresses and thus, in reality, could correspond to virtually any absolute physical address.

The following example depicts how the author would ideally hope to handle conditional branch microwords. The example is a "mem. increment and skip next instr. if result is 0" instruction. Note that "EFF ADDR" means Effective Address.

```
1: MAR←EFF ADDR; to 2;           1: MAR←EFF ADDR; to 2;
2: AI2←MOR; to 3;      con-      2: AI2←MOR; AI1←0; CI←1; to 5;
3: AI1←0; to 4;        dense
4: CI←1; to 5;
5: MIR←AO1; to 6;                5: MIR←AO1; to 6;
6: WRITE CNTRL=1; to 7;  con-    6: WRITE CNTRL=1; AI1←PGC;
   /* Above implies      dense       AI2←0; to(AO1Z) 10,FETCH;
      "MEM←MIR" during               /* In cond'l branches such
      data restore  */                  as above, parenthesized
7: to(AO1Z) 8,FETCH;                    quantity is a binary-
   /* No reg. xfers in                  valued cond' code, or
      above, only cond'l                CC.  If this CC=0, left
      branch on cond'                   next address (here "10")
      code AO1Z  */                     is used; if CC=1, right
8: AI1←PGC; to 9;                       next address (here
                                        "FETCH") is used.  */
9: AI2←0; to 10;
10: PGC←AO1; to FETCH;           10: PGC←AO1; to FETCH;
```

uncondensed microcode — condensed microcode

Fig. 3 Some "Before & After" Examples



NOTE: Refer to Fig.'s 1 & 2 for explanation of "Master Reg.", various control bit #'s, etc. (In above, "AO1Z Bit" refers to the microinstruction bit which performs a cond'l branch based on value of AO1Z.)

NOTE: "Inhibit" functions for other components in HM1 are formed in a similar manner to the one shown above for the adder.

Fig. 4 "Inhibit" Function Example

THE COMPUTER AIDED DESIGN OF PROCESSOR ARCHITECTURES*

A. M. Peskin
Applied Mathematics Department
Brookhaven National Laboratory
Upton, New York 11973

Abstract

A design automation system is described which has certain features that make it particularly well suited to the automation and analysis of entire digital systems; including hardware, firmware and software. The code includes facilities for automated synthesis, simulation and documentation. The languages, called LINDA and MODEL, and their processing algorithms are described.

Examples of computer architecture studies which have been conducted using LINDA and MODEL are also presented. These include development of an associative processor and investigations related to a logic-in-memory architecture. It is stressed that reliance on the appropriate design automation procedures have expedited the work and reduced the cost of research by eliminating the implementation phase as a necessary prerequisite to the study of machine concepts.

Introduction

One ironic aspect of design automation is that those who have been intimately aware of the capabilities of digital computers longer than anyone else, the computer designers themselves, lag behind their counterparts in many other disciplines in adapting these machines to solve their problems. Users in fields as diverse as astronomy and accounting have automated large segments of their workload, yet computer design has been a manual art until very recently. To be sure, the problem of adapting design automation techniques to entire systems is not a simple one, yet the fruits of such an endeavor appear to be well worth the effort. The advent of register transfer languages and special purpose simulators have increased the usage of computers in design of processor architectures, but each of the techniques introduced heretofore seem to have one or more serious deficiencies which detracts from large scale acceptance by the computer design community. The worth of total system design automation and the progress (or lack of it) in specific areas such as synthesis, has been brought out clearly in recent survey articles.[1,2]

A design automation system is described which has certain features that make it particularly well suited to the automated design and analysis of entire digital systems. Unlike register transfer languages, it provides all the precision and fidelity of gate level or instruction level descriptive models. It provides for the specification of hardware, software, and firmware at any desired level of detail with respect to timing, logical interaction, or cost. The descriptive formats used are evocative of the designers own "jargon", thus fostering a measure of immediate familiarity for new users. And the codes are written entirely in FORTRAN, to enhance portability among computer installations.

A description of the system follows. In addition, some applications of its use in investigation of computer architectures are offered.

The Design Automation System

The design automation system provides the modular facilities of automatic synthesis, layout, documentation, simulation and generation of diagnostic sequences. It is a unified system centered around a base language called MODEL which is the output file of the synthesis task and the input file of the simulation task. The interrelationship of the various program modules and data bases are given in Figure 1.

To synthesize a system, a generalized behavior description formatted in a language called LINDA is fed into the synthesis task. The output of that task is a network of interrelationships of system building blocks presented in the MODEL network description format. Typically this is a gate-level logic circuit description but facilities exist for utilizing complex macromodules and instruction sequences as well. From the MODEL base file, a task can be called to generate automated documentation, such as flow and logic diagrams and, in the case of gate-level circuits, layout and routing information as well. A simulation task can also be called, which, when externally excited by a test sequence, will provide indications of the system response, such as oscillographs or parameter dumps.

The specific behavior of the system under analysis, as represented by the simulator

*Work performed under the auspices of the U.S. Atomic Energy Commission.

output, can then be compared to the generalized behavior description which was fed into the synthesis task (as indicated by the check-out path of Figure 1). Many iterations are generally necessary before the system has been exhaustively tested and is satisfactorily producing prescribed behavior. But the point is that a logical system can be designed and thoroughly tested at any desired level of detail with a minimal amount of designer intervention. Thus, architectural concepts can be explored quickly with a vastly reduced level of system implementation activity and correspondingly lower development cost.

The MODEL and LINDA programs were originally developed for the synthesis and simulation of logic designs. It was recognized, however, that a fundamental requirement of any practical design automation code is the ability to encompass macrofunctions, macrosequences, read-only-storages and writeable storages as well as Nand gates and flip-flops. The resulting code is a design automation facility that treats all aspects of a digital system in a unified and consistent manner.

The elements of LINDA, the behavior specification language, and MODEL, the network description language are given in Tables 1 and 2. Details of the use of these languages and programs with examples are published elsewhere.[3] Obviously, these languages were originally meant for pure hardware applications, but since the formats and algorithms were sufficiently general to include "softer" aspects, MODEL and LINDA have evolved into tools that could be applied equally well to the analysis of architectural questions. The programs, in fact, continue to evolve and adapt to new applications, with newer versions containing expanded features for software.

Extensability to software in the simulation task is quite simple because all MODEL devices, both primitives and macrofunctions, are handled as "black boxes" with tabulated responses to given binary input vectors. Hence, by considering an instruction sequence as the behavior table of a firmware or processing storage module, it is immediately and naturally integrated into the system under test.

There are at least two viable approaches for accommodating an instruction reperatory in the synthesis algorithm. The first is to recognize certain subnetworks as functions for which there is a predefined instruction. Then, if the timing constraints will allow, this portion of the design can be partitioned into the software area, and the operands and results need only be accessible to general registers. The second approach, and the one which is most frequently taken, is to simply extend the complement of LINDA operators beyond those listed in Table 1.

Internals of the Tasks

The MODEL simulator uses an event-based, table-driven algorithm similar to that used in many stochastic simulation systems. The main simulation table has one row per device and up to sixty parameters or columns. It is distributed over several data set media, with the more frequently used columns being assigned to the more accessible media in a hierarchical structure.

The simulator begins by processing action statements until it reaches one or more signal generation statements. There is only one possible type of external event that can excite the circuit; the occurrence of a signal generated on an input line as represented by GENSIG. When this occurs, all devices whose inputs are affected are checked for prospective instabilities directly caused by this change. As the instabilities give way to output value changes, the devices to which these outputs connect become themselves new candidates for instability, and so on until the system becomes quiescent or the simulation time equals that of the next GENSIG.

The instability test for any device depends, of course, upon the device type. The test does, however, lend itself to a generalized treatment. For any k-input device i, the n-th output is a function of the ordered inputs I_j and, in some cases, the current state.

$$(1)\quad \phi_{i,n} = F_i(I_1, I_2, \ldots I_j, \ldots I_k; \phi_{i,n-1})$$

If, then, the state of the inputs is such that the output should be modified, a criterion labeled B is set

$$(2)\quad B_i = \begin{cases} 1 \text{ if } \phi_{i,n-1} \neq F_i(I_1, I_2, \ldots I_k; \phi_{i,n-1}) \\ 0 \text{ otherwise} \end{cases}$$

When this condition is first noted, the current simulation clock value is added to the appropriate device propagation delay (TPD) to schedule the next output transition at time

$$(3)\quad TNEXT_i = TPD_i + T$$

Then, criterion A_i is set

$$(4)\quad A_i = \begin{cases} 1 \text{ if } T \geq TNEXT_i \\ 0 \text{ otherwise} \end{cases}$$

The two criteria are added arithmetically and tested

$$(5)\quad C_i = A_i + B_i - 1$$

Figure 1

MØDEL/LINDA SYSTEM DESCRIPTION

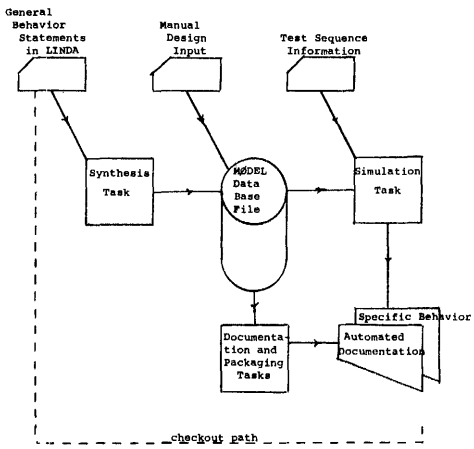


TABLE 1

ELEMENTS OF SPECIFICATION LANGUAGE
(LINDA)

Operators

Conditional Operators (CDØP)

DURM,T	duration no more than T
DURL,T	duration no less than T
X,N	occurred N times
PREV	previous occurrence of
AFT,T	delay

Boolean Operators

C	complement of
+	or
.	and

Syntax

Variable ← Input
Variable ← (Variable + Variable)
Variable ← (Variable . Variable)
Variable ← (CDØP . (Variable))
Variable ← (C(Variable))
Output ← Variable

TABLE 2

MODEL LANGUAGE STATEMENTS

Device Definition Statements:

NAME(I,J)/TYPE/INPUTS/OUTPUTS/DELAY

(Types are: JK, RS, AND, OR, INV, NAND, NOR, and TIE)

NAME(I,J)/FUNCTION/INPUTS/OUTPUTS

Connection Statements:

FROMTO/Name,Output No./Name, Input No.
BUS/Name(I,J), Output K/Name (M,N),
Output L/I=A,B/J=C,D/M=E,F/N=G,H

Monitoring Statements:

TRACE/variable list
SCOPE/output list
SNAP/output list

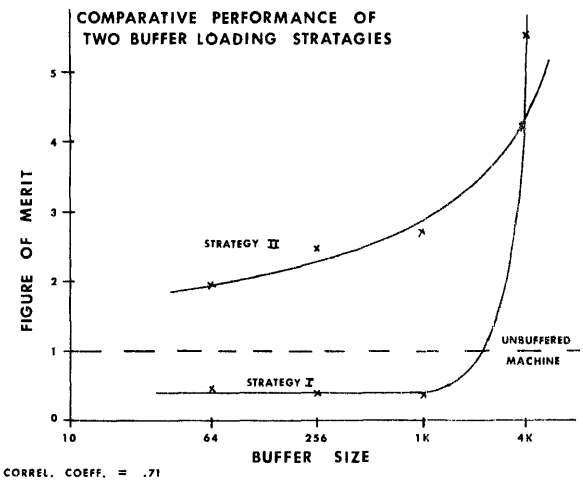
Control Statements:

END FUNCTION
END MACRO

Action Statements:

START
STOP
MACRO
IF/LABEL
GOTO/LABEL
GENSIG/VAR/Duration
WAIT
Q=(Q is a general register)
T=(T is the simulation clock)
QUØTE

FIGURE 2



$C_i = 1$ means that the device is ready for a transition. $C_i = 0$ indicates a stable condition; either the outputs agree with the input and no event is scheduled, or a transition is pending and recognized as such. $C_i = -1$ indicates that the function of the input conditions have changed twice in the time of one propagation delay or less, and a "spike" will appear on the output of that device. This usually indicates a design error and it is appropriate to flag this condition with a warning to the user.

The synthesis task consists of parsing the statements from the innermost parenthesized nesting level outward, and then optimizing the resulting forest of directed rooted tree graphs by combining those variables recognized as being equivalent and allowing the fan-out to exceed one. Most internal states or storage requirements of the circuit are recognized in the synthesis process. [An output which is explicitly set in one statement and explicitly cleared in another will be implemented with an R-S flip flop. The results of the PREV or X operators will also call for utilization of multistable devices.] Other simplifications performed by the program include those recognizing the economies of inverting logic and expanded fan-in, i.e.

$$C(A) + C(B) = C(A \cdot B)$$

$$C(A) \cdot C(B) = C(A + B)$$

$$A + (B + C) = A + B + C$$

and $$A \cdot (B \cdot C) = A \cdot B \cdot C$$

Error checking in the synthesis task includes testing for the grammar and syntax of the specification statements and tests for timing problems, particularly those where the accumulated unit delays in a generating tree exceed the expected generation time as specified by the (AFT,N) conditional operator.

The relationship between behavior and structure that suggests extensions of the algorithms to other classes of systems has been formally explored with the algebraic theory of categories. In the terminology of category theory, the statement is that behavior is left adjoint to minimal realization as functors between certain categories of machines and behaviors. The implication is that almost any system which has a descriptive algebra can be minimally realized using the method of the LINDA synthesis algorithm.[4]

Applications in Computer Architecture Studies

Associative Processor

The first versions of the design automation codes described above were written expressly for the analysis of novel computer architectures in studies conducted at Brookhaven National Laboratory. The first machine developed and analyzed in this program was an associative processor.[5] A 1024 word content addressable storage was connected to a Digital Equipment Corporation PDP11/20 minicomputer via a "unibus" interface controller. Within the controller was logic enabling the augmented PDP11 to process several new instructions in addition to the standard PDP11 reperatory. The newly defined instructions provided for the parallel association of the entire 1024 word array for simultaneous comparison with an instruction operand. In addition to the test for equality, the tests for greater-than and less-than, search for maxima and minima and other useful extensions were implemented via hardwired algorithms.

These new instructions behave exactly as any other machine instruction. They are trapped by the controller which then becomes master of the unibus. Control is returned to the central processor only when the instruction is complete. Hence, except for the new and powerful associative capability, the machine modifications are transparent to the user.

There are many references in the literature citing advantages of associative processors. The extended capability is useful in the solution of data management problems, processing systems of differential equations, communications and many other applications. The Brookhaven approach provided for an inexpensive method of developing such a processor. The extensive use of design automation techniques increased the cost effectiveness of the project as well. Simulations were run to test the storage array and the controller, giving the designers a basis for early evaluation and providing for considerable checkout before implementing the computer. Instruction level simulations enabled application codes to be written, debugged and evaluated prior to hardware availability.

Logic-in-Memory Processor

Subsequent to the associative memory processor, a more advanced architecture was investigated. It was structured around a distributed logic storage device or Processing Memory. This logic-in-memory architecture, unlike the associative processor, was not to be ever fabricated, but was rather meant only as a vehicle for evaluating its concepts through simulation. This project was also the first to make extensive use of the synthesis capabilities of the design automation code.[6]

One of the criteria for evaluating the logic-in-memory processor was its suitability for large-scale-integration (LSI) technologies. This imposed the requirement on the

MODEL and LINDA languages to also exhibit suitability for LSI. Hence, it was necessary to have a convenient method for generating and describing networks of large numbers of similar or identical subnetworks, as required by the economics of LSI implementation. The macro definition capabilities of the MODEL formats allow the design automation system to meet this requirement very well. In fact, it was found that the number of MODEL statements necessary to describe a system is a good measure of the optimization of that system for LSI implementation, i.e., the circuit with the smaller number of statements was more likely to be a better design for LSI.

Loading Strategies

Other computing techniques outside the framework of MODEL and LINDA have been used at Brookhaven to aid the system architect. One example arose in the logic-in-memory study but is applicable to almost all current and future processors. Most computers have storage hierarchies ranging from small, fast, and perhaps logic-augmented storage to very large, slow, sequential storage. For these systems, high performance is very much a function of the probability that the operands of interest are in the fastest store or scratchpad. Consequently scratchpad loading strategies must be carefully defined and analyzed.

It is difficult to find a deterministic method that can explore and resolve questions such as which loading strategy is best. It is rather easy to write a computer program, however, which can simulate and compare various strategies if a set of typical storage address sequences are available. This is exactly what was done in the logic-in-memory machine study. Large samples of storage address sequences were extracted from several different types of existing computers, both real and simulated. They were applied to various strategies and tested at several different buffer sizes. The resulting curves for each strategy were plotted so that the designer could select the buffer size and/or loading strategy best suited for the computer. An example of such graphical results appears in Figure 2. With the advent of writeable control storage, both strategies and buffer sizes can be changed dynamically, so it is even more important to develop "a priori" knowledge of the various characteristics. Then the proper control storage program can be invoked for a given set of circumstances (i.e. type of job, field length, user estimate of CPU utilization, degree of multiprogramming, etc.).

Other Applications

There are many other types of problems faced by the computer architecture designer for which computer aided solutions are advantageous. Conventional queuing models, mathematical programming optimization models, communications traffic and sampled data analysis models can all be used to good advantage for computer systems.[7,8] Special languages have been developed to describe computer structures and instruction sets. And, of course, small programs can always be written to expedite the solution of almost any large problem.

In summary, there is an entire spectrum of computer related facilities available to the digital system designer. One of the very real challenges of the future may very well be to select and adapt the most appropriate of these to the project at hand.

References

1. M. Breuer, Recent Developments in Augomated Design and Analysis of Digital Systems, Proceedings of the IEEE, vol. 60, no. 1, January 1972.

2. S. Su, Computer Aided Logic Design, Computer Design, vol. 13, no. 2, February 1974.

3. A. Peskin, A Users' Manual for the MODEL and LINDA Design Automation Programs, Brookhaven National Laboratory, BNL 50416, February 1974.

4. J. Goguen, Realization is Universal, University of Chicago, C00-2094-4, Chicago 1970.

5. N. Schumburg, Content Addressable Memory Processor, Proceeding of ACM 73, Atlanta, Georgia, August 1973.

6. A. Peskin, A Logic-in-Memory Architecture for Large-Scale-Integration Technologies, Proceedings of ACM 72, Boston, Mass., August 1972.

7. F. Kuo and J. Kaiser, System Analysis by Digital Computer, John Wiley and Sons, Inc., New York, 1966.

8. G. Gordon, System Simulation, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.

INTERMODULE PROTOCOL FOR REGISTER TRANSFER LEVEL MODULES: REPRESENTATION AND ANALYTIC TOOLS

W. H. Huen
Department of Computer Science
Illinois Institute of Technology

D. P. Siewiorek
Departments of Computer Science
and Electrical Engineering
Carnegie-Mellon University

Abstract

A distinguishing feature of modular design from ad hoc design is the establishment of an intermodule protocol to which all modules adhere. The problem of representing and analyzing intermodule protocol for the control portion of register transfer level systems is outlined. An introduction to two existing graph models of computation indicates that existing register transfer level module sets are representable by various "token flow" models. A single model that is capable of representing the token flow models and some of its analytical properties are illustrated by example. Finally, three examples of deadlocks in existing modules sets are presented. These deadlocks were uncovered by the analytic properties of the new model. One example is due to incorrect interconnection of modules at the user level. The other two illustrate incorrect signaling conventions between modules necessitating a redesign of some modules.

1. Introduction

As technology has evolved the primitive components available to a digital system design have increased in complexity. Twenty-five years ago the designer constructed his systems out of circuit level components such as resistors and diodes. Subsequently switching circuit level components, as represented by gates and flip-flops, became available as small scale integration (SSI) components. With the introduction of medium scale integration (MSI) register transfer (RT) level components appeared: arithmetic and logic units, registers, etc. The advent of large scale integration (LSI) has made memories and even processors primitive components from which systems are designed.

One model of the design process is as follows. The designer partitions the system into "modules" that are each, in turn, designed from the available primitive components. The advantages of this "modular" design approach are well documented [Bell 73; Davidow 72; Parnas 71] and a partial list might include:

1. Reduced development time by allowing the design task to be partitioned, making better use of the resources (time and manpower)

2. Increased flexibility by allowing the alteration of specifications and the redesign of modules

3. Comprehensibility by allowing the students of the system to concentrate in well defined pieces of the final object

4. Maintainability by allowing the identification of faulty components which can then be replaced or repaired

5. Economy of scale by mass producing a small number of standard modules rather than supporting custom designs

6. Significantly decrease system construction or modification time by conducting all design with high level modules as primitives. This is different from (1) and (2). Here we are talking about building systems with predefined modules, while (1) and (2) are concerned with the specification of the original modules.

Many of the advantages of modular design stem from the definition of an intermodule protocol for transfer of data and control signals. A module belongs to a particular set if it adheres to the signaling convention for that set. This paper presents two representational methods from which is derived a notation sufficiently powerful to represent register transfer level module sets. Analytical tools based on this notation enable us to evaluate the intermodule protocol of existing and proposed module sets.

The paper is divided into six sections. Section Two describes existing register transfer level module sets while Section Three discusses two theoretical models of computation. A unifying notation and some of its analytical properties are presented in Section Four. Section Five demonstrates the application of the model and tools to existing module sets. The final section defines some open research problems.

In a paper of this length we cannot present all the details of our research. Rather we hope to motivate and illustrate our results by example. Full details can be found in [Huen 73].

2. Existing Register Transfer Level Module Sets

Typically register transfer (RT) level module sets are divided into a control part and a data part. An example of an RT level module set is the Register Transfer Modules (RTMs) designed by Digital Equipment Corporation and Carnegie-Mellon University [Bell 72a, 72b].

RTMs use a distributed control scheme and currently there are approximately half a dozen control module types. As an economic decision, all the data modules (approximately a dozen data module types) are interconnected via a single bus. However, provision exists for RTM systems to have more than one data bus when increased performance is required. Figure 1 depicts the RTM implementation of a system to sum the integers from 1 to N. The connections shown in the figure are all that are required to construct the physical system. A detailed description of RTMs is given in [Bell 72a]. However, the flowchart format for RTM notation is so familiar that the detailed reference probably need not be read to understand the following discussion. Note that the primitive functions are very similar to those available at the assembly language level of programming. Other RT level module sets have been developed at MIT [Patil 72], Washington University [Clark 67], and the

The research in this paper was performed at Carnegie-Mellon University and supported by National Science Foundation Grant GJ32758X.

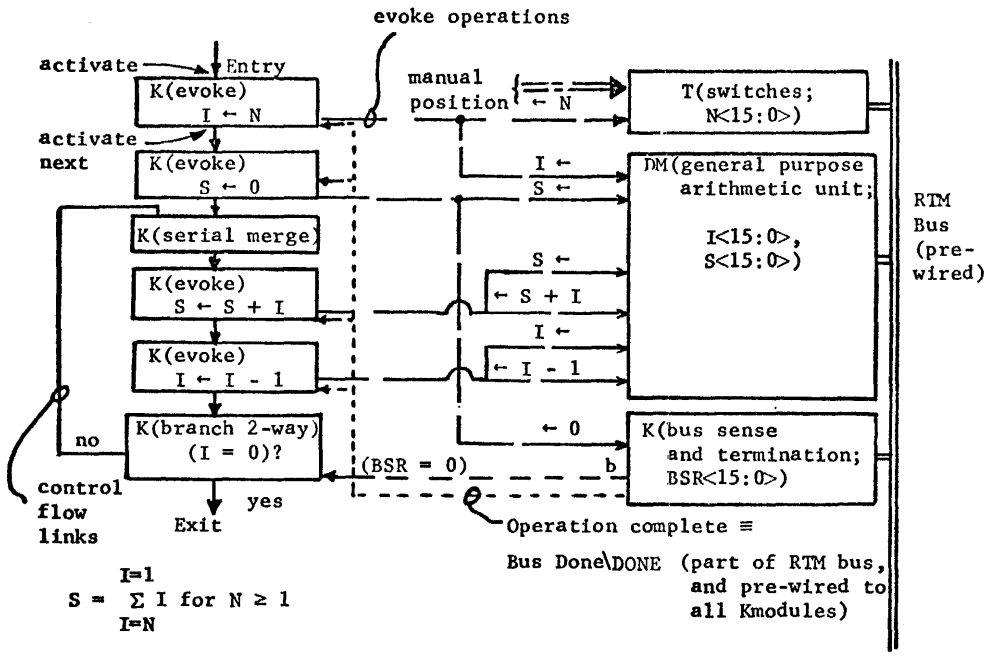


Fig. 1. RTM diagram for sum of integers from 1 to N.

University of Delaware [Robinson 73].* An interesting feature of the latter module set is that the data part is composed solely of commercially available MSI chips.

As exemplified by the University of Delaware module set, commercially available MSI/LSI chips can be used for the data part of RT module sets. However, there is only a bewildering array of SSI components to perform control functions [Fuller 73]. Thus motivated we examined control from a graph theoretic viewpoint and established tools for analyzing intermodule protocol signals. The tools were subsequently used to evaluate existing RT module sets.

3. Graph Models of Computation

There are a number of models which may be applicable for the description of RT modular systems. These are:

1. Conventional Representations which are the flowchart, the state diagram and the state table;

2. Token Flow Models which include the Petri Net, Parallel Program Schema [Karp & Miller 67,69], Flow Graph Schemata [Slutz 68], Graph Model of Computation (GMC) [Gostelow 71];

3. Data Flow Models which include Computation Graph [Karp & Miller 66], Program Graph [Rodriguez 67] and Adams' Computational Model [68];

4. Register Transfer Languages examples are ISP [Bell 71] and CDL [Chu 70].

Most of the above models are proposed mainly for parallel processing instead of specifically for Register Transfer Control Systems. Many of these models are based on graphs. The existing modular RT systems have the characteristics of being capable of asynchronous and concurrent operations. The properties of graphs mak these representations very attractive as representations of RT control. Nodes of a graph can correspond to control modules. Concurrent actions are easily represented by parallel arcs. The asynchronous mode of operation is represented by the graph property that for two nodes A and B joined by an arc directed from A to B, control is considered to pass from A to B if the RT operation associated with A is complete. The RT operation is assumed to take a finite but unspecified time.

The various models of parallel computation were surveyed and a taxonomy was developed [Huen 73]. Their properties were classified by the taxonomy in order to determine their suitability as representations of RT level module sets. For a survey of graph models, the reader is referred to [Huen 73] or [Baer 73]. Two models will be discussed here.

A Petri Net as presented by [Holt and Commoner 70] and [Dennis 70] is a directed graph with two types of nodes, namely transitions and places. Transitions represent events and are drawn as a bar in the graph. A place represents a condition and is drawn as a circle (Figure 2). The arcs in the graph must be directed from a transition to a place or from a place to a transition. If an arc is directed from a place to a transition, the place is an <u>input place of the transition</u>. If an arc is directed from a transition to a place, the place is an <u>output place of the transition</u>. Thus in Figure 2, P_1 is an input place to T_1 while P_2 and P_3 are output places of T_1.

*The advantages of design with these module sets are dramatic. A PDP-8 like minicomputer could be designed and build in six-seven man-months using discrete components. A similar processor built from SSI components might take two-three man-months and from MSI/LSI components about one man-month to design and construct [Bell 74]. A PDP-8 like minicomputer has been designed, constructed, and debugged with RTMs in eight-ten man-hours. As in the case with all the RT module sets, the translation from paper design to hardware implementation is a one-for-one process. A large majority of the systems work the first time power is applied. The module sets provide a very clean intermodule communications protocol which eliminates any timing problems.

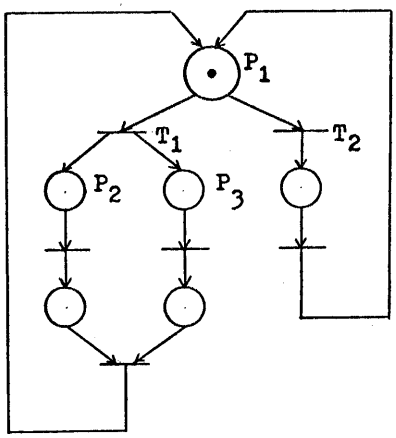


Fig. 2. A Petri Net

Each place may contain tokens. A place containing a token represents holding of the condition that is associated with the place. Graphically a token is represented by a dot in a place. A marking of a Petri Net with r places is a mapping from the set of r places to a set of r-dimensional vector of nonnegative integers, each of which represents the number of tokens in the corresponding place.

A transition having tokens in all of its input places is said to be enabled. Semantically a conjunction of input conditions enables a transition. Only enabled transitions can fire. The firing of a transition removes a token from each of its input places and a token is deposited in each of the output places of the transition, thus the conjunction of the output conditions hold. A transition cannot selectively put tokens in output places. The FORK and JOIN mechanisms [Conway 63] are provided by this rule. Alternate control flows are depicted by transitions having common input places. In Figure 2 both transitions T_1 and T_2 are enabled but the firing of one transition disables the other. A simulation is a sequence of transition firings that are permitted by the net. A marking M' is said to be reachable from a marking M if there exists a firing sequence which transforms marking M into M'. A marking class of a Petri Net is the set of all markings reachable from the initial marking.

Structurally the Petri Net is a token flow model in which a token carries no attributes as compared with Adams' Computation Graph. Only the positions of the tokens matters. Since each place corresponds to a boolean condition, it is desirable to have a Petri Net constructed so that it never has more than one token in a place in any marking. Such a Petri Net is said to be safe.

A Petri Net models control branches by showing only the possible branches of token flow. No mechanism is provided to show how the branch decision is reached.

Another interesting property is the liveness of Petri Nets. A live Petri Net is one in which every transition can be executed infinitely often. [Holt and Commoner 70] first studied necessary and sufficient conditions for live and safe state machines and marked graphs which are proper subclasses of Petri Nets. [Hack 72] has presented the necessary and sufficient conditions for live and safe Free Choice Nets which include state machines and marked graphs. Free Choice Nets, a proper subclass of Petri Nets, are not complex enough for the analysis of most of the Petri Net descriptions for the behavior of MIT Asynchronous Modules given in [Dennis 70]. [Huen 73] has proved the necessary and sufficient conditions for Petri Nets. We shall discuss the result in Section Five and show how they can be used to analyze RT control flow.

Another token flow model is the Graph Model of Computation (GMC) [Gostelow 71], which is a graph with a single node type, the vertex. The vertices are similar in function to the transitions of Petri Nets. The arcs are containers of tokens. The GMC is more general than Petri Nets in two aspects: input and output logic conditions and input and output thresholds. An input logic condition, either disjunctive, '+', or conjunctive, '*', is specified among input arcs of a vertex. Each input or output arc of a vertex is associated with a threshold, a positive integer. An output logic condition, '+' or '*', is specified for all the output arcs of a vertex. A disjunctive logic condition specifies that any input arc, say α, of a vertex V must contain tokens equal in number or exceeding the token threshold for arc α in order to fire V. The conjunctive input logic requires each input arc to contain as many tokens as the specified threshold to fire the vertex. Similarly, on the firing of a vertex, one or more tokens can be placed by the vertex in selective or all output arcs depending on whether the output logic condition is disjunctive or conjunctive, respectively.

Figure 3 shows the GMC graph equivalent of the Petri Net in Figure 2. The integers in a vertex against an arc represent the thresholds. [Gostelow 71] has proved that GMCs with thresholds equal to one are equivalent to Petri Nets.

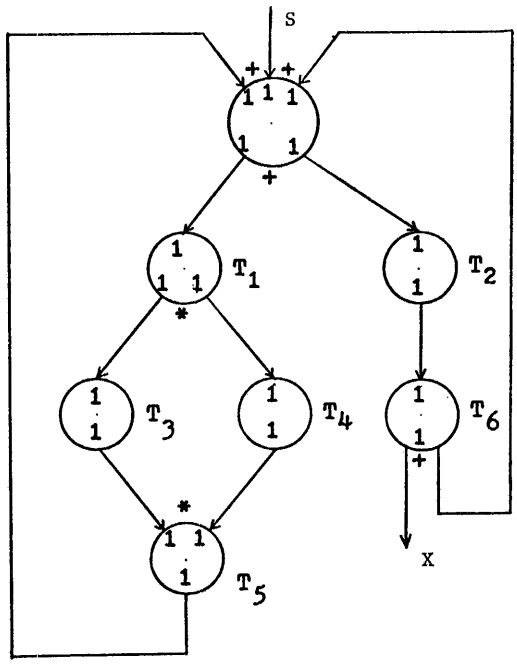


Fig. 3. A GMC

GMC graphs are organized so that there is an entry arc S and exit arc X. A properly terminating GMC is one such that given a token only at its entry arc S initially, there will be a token at its exit arc X and no other arc will have a token when the token flow stops. Necessary and sufficient conditions for a properly terminating GMC will be discussed in Section 4.

4. A Unifying Notation for RT Level Control

It may be observed from Section 3 that token flow models are suited for RT control description. Token flow models are in general directed graphs which

show explicitly tokens on the arcs or vertices of the graphs. The configuration of tokens determine if a vertex (transition) may fire. A vertex (transition) may be interpreted to represent a portion of a control module and a token describes a control signal. The firing of a vertex (transition) corresponds to the activation of a control module. The parallel flowcharts proposed for the Macromodules and the RTMs are actually special cases of token flow models. Moreover, some control flow concepts, i.e. Proper Termination (PT) and Liveness and Safeness (LS) have already been introduced in token flow models. We may borrow these concepts for the analysis of deadlocks.

The token flow models have different assumptions and notations. It was shown in [Huen 73] that all token flow models can be unified in a geometrical structure, the Vector Addition System (VAS), which was first studied by [Karp & Miller 67, 69]. We shall define the VAS and show that liveness and safeness of Petri Nets and properly terminating GMCs can be expressed as properties of the VAS.

Definition. An r-dimensional vector addition system (VAS) V is a pair $V=(m_0,D)$ where

1. $m_0 \in N$ where N is the set of nonnegative integers,

2. D is a finite set of r-dimensional integer vectors d_i which are called displacement vectors, i.e. $D=\{d_1,d_2,\ldots,d_n\}$.

The Reachability Set R(V) is the set of vectors of the form $m_0+d_1+d_2+\ldots+d_s$ such that $d_i \in D$ and $m_0+d_1+d_2+\ldots+d_i \geq 0$, $i=1,2,\ldots,s$.

[Karp & Miller 69] shows that the possibly infinite reachability set R(V) of a vector addition system V can be represented by a finite control flow tree T(V). We will illustrate this by example.

Figure 4 depicts a Petri Net and its corresponding VAS. The VAS consists of an initial marking vector m_0 and a set of displacement vectors D which correspond to transitions. Each component of the vector corresponds to a place in the Petri Net. All valid firings (new markings) of the Petri Net can be determined by adding the d_j displacement vectors to the current marking m_i. Those additions which result in all marking vector components being nonnegative are valid markings and can be used to establish subsequent valid markings. For example, the only valid markings from the initial marking m_0 resulting from the addition of a single displacement vector in Figure 4 are (0,1,0,0) and (0,0,0,1). The displacement vector d_3 does not lead to a valid marking since the result of its addition to m_0 is (2,-1,1,0).

A control flow tree, depicting all possible markings (or states) of the VAS can be constructed as shown in Figure 4. An 'ω' indicates that a marking is identical to another marking further up in the tree except that the component of the successor marking vector is greater than its predecessor. For example, consider the initial marking (1,0,0,0). Adding d_1 and d_3 sequentially yields (1,0,1,0). Since $(1,0,1,0) \geq (1,0,0,0)$ componentwise, the sequence of displacement vectors d_1d_3 can be used again. A second addition of d_1d_3 yields (1,0,2,0). The sequence of displacement vectors can be used a large number of times causing the third component of the marking to increase to an arbitrarily large number. The 'ω' is a notation to indicate this indefinite growth of a component.

Nodes are appended to the tree until for each leaf either its marking is identical to that of one of its ancestors or no displacement vectors can be applied; in either case the node is called a leaf.

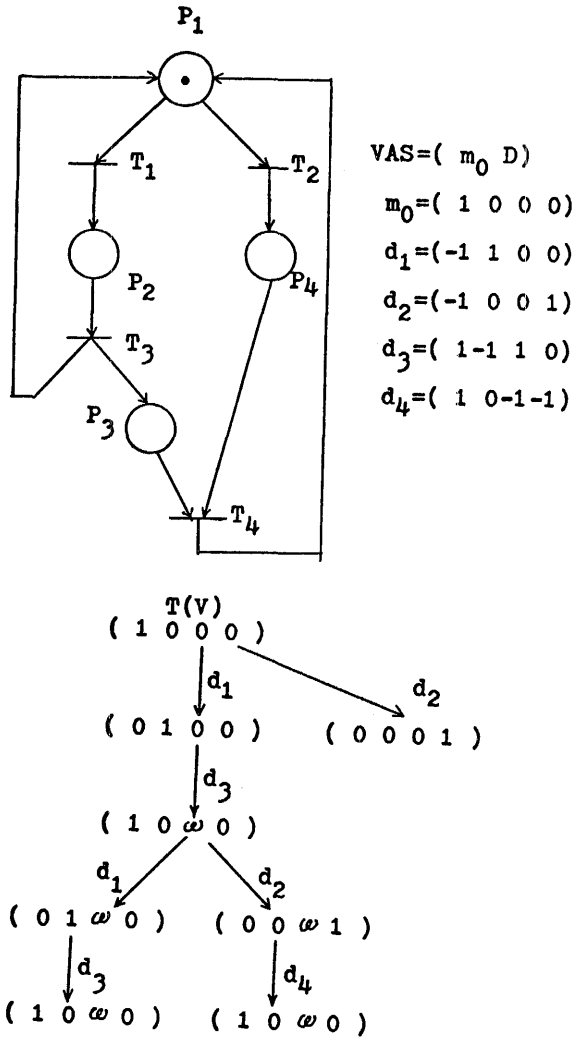


Fig. 4. Petri Net, its VAS and control flow tree T(V)

Properties of this tree can be used to detect properties of the Petri Net. For example, the marking (0,0,0,1) has no successors and is not represented anywhere else in the tree. Once in the state represented by (0,0,0,1) the system will never move to another state. This Petri Net is not 'live'.

The necessary and sufficient conditions for a Petri Net to be live and safe are as follows:

1. it has no "ω" in any marking in its control flow tree T(V);

2. for each leaf there exists at least one ancestor with an identical marking such that in traversing the tree from the ancestor to all possible leafs, all displacement vectors in D are encountered.

A properly terminating GMC has a control flow tree such that the marking of every leaf is identical to that of a single token at the exit arc X.

5. Deadlocks in RT Modules Sets

Unless the intermodule protocol is carefully designed a situation can arise where a module initiates an activity and awaits a response which never occurs. Deadlocks are a common problem with control protocol in RT module sets. The two concepts: Proper Terminating GMC and Live and Safe Petri Nets can be used to analyze deadlocks. A vertex in a GMC models the action of (a portion of) a control module and a token models a control signal. With this interpretation, the Properly Terminating condition (PT) has the physical meaning that when the control flow stops and no control module is initiated to wait for a control signal which will never occur. A live Petri Net is one in which every transition can be executed infinitely often. Physically, liveness means that if a control network is cyclic, none of its component control modules can be ruled out for operation after a finite number of operations. Safeness guarantees that no module receives a new control signal until it has completed its control action associated with a previous activation.

For RT modules, deadlock can occur at two levels. The first is an erroneous connection of modules by the user. [Keller 68] has studied implementation errors which are introduced by implementation of an algorithm as a concurrent process network. These errors belong to the user level. The user usually works with a high level description of the module set which we shall call the functional level. Figure 5 depicts an RTM system, its functional level VAS and a portion of the tree that demonstrates deadlock. For the RTM system a K.div evokes all its successor paths, a Kb2 activates only one of its two successors, a Kpm awaits signals on all incoming paths before evoking its successor, and a Ksm evokes its successor if any incoming path is activated. In the functional level, a K.evoke of the RTM can be represented by a vertex with a single input arc and a single output arc; a K.div by a vertex with conjunctive output logic; a K.parallel-merge by a vertex with conjunctive input logic; a K.serial-merge by a vertex with disjunctive input logic and K.b2 by a vertex with disjunctive output logic.

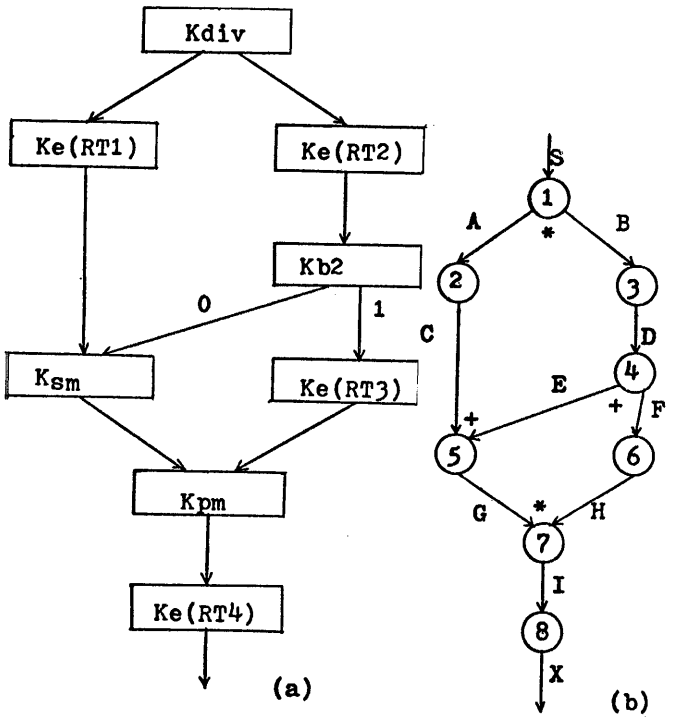


Fig. 5

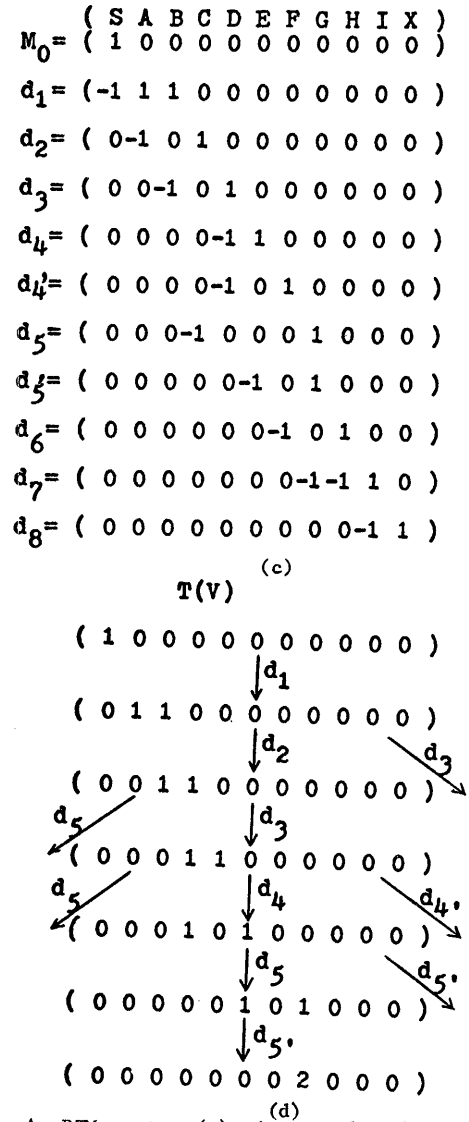


Fig. 5. An RTM system (a), its graphical representation (b), its VAS (c), and one path of its control flow tree (d).

The components of a marking in Figure 5d represent the arcs S,A,B,C,D,E,F,G,H,I,X in that order. The deadlock is indicated by the fact that the GMC in Figure 5b is not properly terminating because the marking of the leaf of the tree T(V) is not (0,0,0,0,0,0,0,0,0,0,0,1).

Deadlock can also arise from an incorrectly designed intercontrol module protocol that would not be detected from the functional level descriptions of the modules. A signal level description is required. Using the VAS and properties developed for the control flow tree, deadlocks were discovered in the signal level analysis of existing RT module sets. Figure 6 shows a system built from Asynchronous Modules that is deadlock free from a functional level analysis but exhibits deadlocks under a signal level analysis. Signal level descriptions, in Petri Nets, of individual Asynchronous Modules are given in [Dennis 70]. A Petri Net for the configuration in Figure 6 can be easily constructed. It is shown in [Huen 73] that the Petri Net is not live and safe. The deadlock can be explained as follows.

The UNION module corresponds roughly to the RTM serial merge (Ksm), the DECIDER to a two way branch (K.b2) and the SEQUENCE to an evoke. Each control link of an Asynchronous Module is composed of two

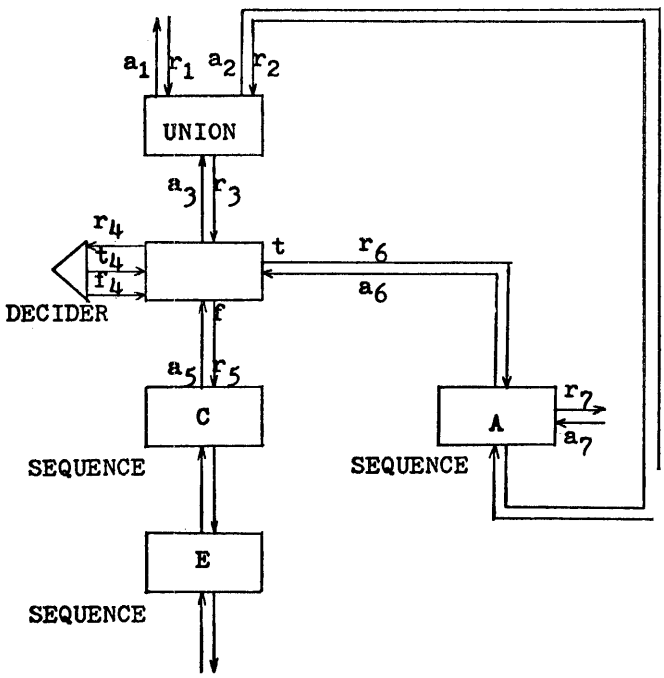


Fig. 6. An Asynchronous Module system that has signal level deadlock.

control lines, ready (r) and acknowledge (a), which are directed in opposite directions. A control link is defined to be active if $r \oplus a = 1$ and idle if $r \oplus a = 0$. Identical subscripts are attached to r and a to identify the control lines in the same link.

The protocol of the UNION stipulates that:

1. if only one input link is active, the UNION activates its output link and waits. When the output link becomes idle, the UNION idles the active input link;

2. if before an activating input link is reset to idle, the other input link becomes active, the UNION still waits for the idle state of its output link. When the output link becomes idle, the UNION resets the first input link and sets its output link active again;

3. if both input links become active at the same time, the UNION prevents hazards by picking one input at random and performs the cycle of activating its output, waiting for the output idle signal and idling the requesting input link. Next it repeats the cycle for the other input.

Assume that all links in Figure 6 are initially idle. Link 1 of the UNION module becomes active by a change in level in line r_1. Link 3 in turn becomes active. The DECIDER thus sends a r_4 signal to the data structure which replies with a control signal to either t_4 or f_4. If a control signal is received along t_4, then link 6 becomes active, otherwise link 5 is active. The SEQUENCE A is assumed to be connected to a RT operation in the data structure. It evokes the RT operation by a signal on line r_7. When link 7 returns to idle, i.e. when an acknowledge is received on a_7, the SEQUENCE activates its output link 2 which is also an input link for the UNION module. At this stage, the UNION must wait for link 3 to become idle. Link 3 becomes idle when link 6 becomes idle. Input link 6 to SEQUENCE A becomes idle when its output link 2 becomes idle. Link 2 cannot become idle unless link 3 turns idle first. A deadlock is thus formed.

This is an example of a well accepted concept in programming being unacceptable because of conflicts with the protocol of a specific RT system. A way out of the dilemma is to establish rules forbidding the use of UNION's in loops.

An RTM configuration that will deadlock was also discovered by the analytic procedure. It is shown in Figure 7. At one stage in the design of RTMs, K.div was just a set of divergent wires from a common source. The deadlock originates from the protocol of the K.evoke The K.evoke is initiated by an activation signal which is also sent directly to the associated RT operation. When the RT operation is completed, the bus broadcasts the Done signal to all control modules on the bus. It is the duty of the control predecessor of the current K.evoke to reset the activation signal upon receipt of the Done before the current K.evoke can activate its output. In a multibused system in which the buses are not connected, the Ke3 and the Ke4 on the branches will be waiting forever for the reset signal. The Done signals from these buses do not reach the contorl module before the K.div. The redesign of the K.div so that it intercepts the Done signals from the branch buses solves this problem.

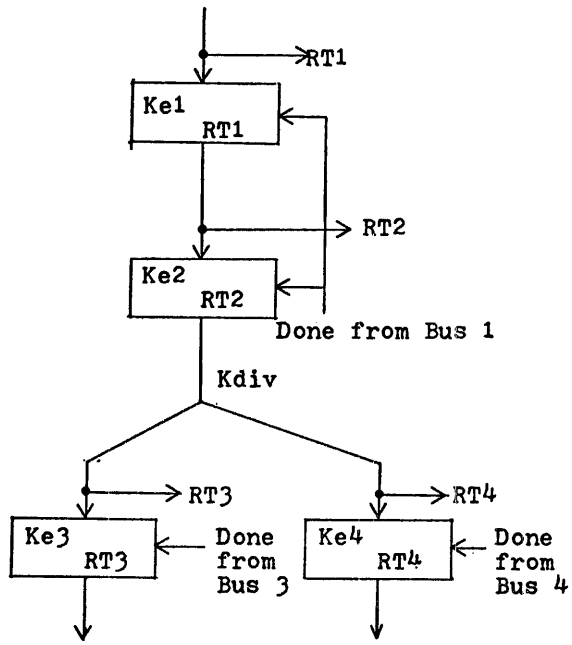


Fig. 7. Multibused RTM system exhibiting signal level deadlock.

6. Conclusions

The VAS seems a particularly powerful model to use in the analysis of intermodule protocol. Designers and users of modular systems can easily determine the correctness of their designs. Even though a VAS may have infinitely many reachable states, it has been shown [Karp & Miller 69] that the set of reachable states may be represented by a finite control flow tree T(V). The properties of proper termination or liveness and safeness can be expressed as properties of a control flow tree. Although a control flow tree is known to be finite its rate of growth as a function of a VAS is an open research question.

Finally, as module complexity increases the inter-module signals will become more complex and likely have unique identities (attributes). An analog to the VAS for Data Flow Models would be a valuable tool for analyzing multiprocessor structures.

References

[Adams 68] Adams, D. A., "A Computational Model with Data Flow Sequencing," Ph.D Thesis, Computer Science Dept., Stanford University, 1968.

[Baer 73] Baer, J. L., "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys, Vol. 5, No. 1, March 1973, pp. 31-80.

[Bell 71] Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, N.Y., 1971.

[Bell 72a] Bell, C. G., J. L. Eggert, J. Grason, and P. Williams, "The Description and the Use of Register Transfer Modules (RTMs)," IEEE Transactions on Computers, Vol. C-21, No. 5, May 1972, pp. 495-500.

[Bell 72b] Bell, C. G., J. Grason, and A. Newell, Designing Computers and Digital Systems," Digital Press, Digital Equipment Corporation, 1972.

[Bell 73] Bell, C. G., 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 Computer Society Internation. Conference, CompCon73, March 1973, pp. 177-180.

[Bell 74] Bell, C. G. (Private Communication)

[Chu 70] Chu, Y., Computer Organization and Microprogramming, Prentice-Hall, 1970.

[Clark 70] Clark, W. A., S. M. Ornstein, M. J. Stucki, A. S. Blum, T. J. Chaney, R. E. Olsen, R. A. Dammkoehler, W. E. Ball, C. E. Molnar, and A. Anne, "Macromodular Computer Systems," AFIPS Conference Proceedings, Vol. 30, SJCC 1967, pp. 335-402.

[Conway 63] Conway, M. E., "A Multiprocessor System Design," AFIPS Conf. Proc., Vol. 24, 1963, pp. 139-164.

[Davidow 72] Davidow, W. H., "General Purpose Microcontrollers. Part 1: Economic Considerations," Computer Design, Vol. 11, No. 7, July 1972, pp. 75-79.

[Dennis 70] Dennis, J. B., "Modular Asynchronous Control Structure for a High Performance Processor," record of the Project MAC Conference on Concurrent Systems and Parallel Computation, June 2-5, 1970, Woods Hole, Mass.

[Fuller 73] Fuller, S. H. and D. P. Siewiorek, "Some Observations on Semiconductor Technology and the Architecture of Large Digital Modules," report of the Workshop on the Architecture and Applications of Digital Modules held on June 7-8, 1973, IEEE Computer, Vol. 6, No. 10, October 1973, pp. 14-21.

[Gostelow 71] Gostelow, K. P., "Flow of Control, Resource Allocation and the Proper Termination of Programs," UCLA-10P14-106, UCLA-ENG-7179, Ph.D dissertation, Computer Science Dept., University of California, Los Angeles, December 1971.

[Hack 72] Hack, M. H. T., "Analysis of Production Schemata by Petri Nets," S. M. thesis, Dept. of Electrical Engineering, MIT, February 1972.

[Holt & Commoner 70] Holt, A. W. and F. Commoner, "Events and Condition," Applied Data Research, New York, 1970.

[Huen 73] Huen, W. H., "A Unifying Notation and Analysis of Modular Register Transfer (RT) Control," Ph. D thesis, Computer Science Dept., Carnegie-Mellon University, December 1973, available from the National Technical Information Service, Springfield, Va. 22151 under number PB-235 896 ($17.75 paper, $2.25 microfiche).

[Karp & Miller 66] Karp, R. M. and R. E. Miller, "Properties of a Model Parallel Computation: Determinacy, Terminations, Queueing," SIAM J. Appl. Math., 14, November 1966, pp. 1390-1411.

[Karp & Miller 67] Karp, R. M. and R. E. Miller, "Parallel Program Schemata: A Mathematical Model for Parallel Computations," IEEE Conference Record of the 8th Annual Symposium on Switching and Automata Theory, October 1967, pp. 55-61.

[Karp & Miller 69] Karp, R. M. and R. E. Miller, "Parallel Program Schemata," JCSS 3, 4, May 1969, pp. 147-195.

[Keller 68] Keller, R. M. and D. F. Wann, "Analysis of Implementation Errors in Digital Computing Systems," M.Sc. thesis, Washington University, March 1968 (also available as Computer Systems Laboratory Technical Report No. 6).

[Parnas 71] Parnas, D. L., "On the Criteria to be Used in Decomposing a System into Modules," Dept. of Computer Science, Carnegie-Mellon University, August 1971.

[Patil 72] Patil, S. and J. B. Dennis, "The Description and Realization of Digital Systems," IEEE Computer Society International Conference, CompCon72, September 1972, pp. 223-226.

[Robinson 73] Robinson, D. M., "Digital Systems Design with Control Modules," IEEE Computer Society International Conference, CompCon73, February 1973, pp. 207-210.

[Rodriguez 67] Rodriguez, J. E., "A Graph Model for Parallel Computations," Ph.D thesis, Dept. of Electrical Engineering, MIT, Cambridge, Mass., September 1968.

[Slutz 68] Slutz, D. R., "The Flow Graph Schemata Model of Parallel Computation," Report MAC-TR-53, Project MAC, MIT, Cambridge, Mass. 1968.

PICTURE SYSTEMS, PS, AND THE DESIGN OF A CHANNEL-TO-CHANNEL COMPUTER INTERFACE

Portia Isaacson
Xerox Corporation
Dallas, Texas

Summary

This paper introduces a new simulation tool called picture systems. A picture system is (1) a set of pictures - one representing each state of a modeled computer system and (2) a transition graph which relates each picture to the set of pictures that may follow it. Picture systems can be used to model computer systems at any level of detail; however, this paper is concerned with modeling hardware/software systems at relatively high architectural levels. Picture systems, as a simulation tool, are useful to the computer architect. Perhaps more importantly, they provide an unexcelled means of communicating computer system mechanisms between people.

The construction of picture systems from descriptions of the components of a computer system has been automated in PS. This paper describes a model of a channel-to-channel computer interface mechanism consisting of both hardware and software. Transition graph analysis by PS is briefly described. This powerful aid to computer system modeling eases the identification of problems such as deadlock, looping, and races.

Introduction

The day of the computer architect has arrived - basic building blocks are no longer gates and flip-flops; but buses, memories, and processors. The solution of an information processing problem involves making the right choices from the variety of components offered and then designing the hardware/software interface mechanisms between the components. Methods of specifying and simulating mechanisms involving both hardware and software at varying levels of detail are needed as tools.[1] PS is such a tool that is also designed to ease the problem of communicating hardware/software mechanisms between people.

One way of showing the utility of a tool for computer system design is to demonstrate its use on a reasonably difficult problem. This technique will be used to introduce a tool which consists of picture-system models along with PS to automate model generation and analysis.[2,3,4] The problem is the design of a channel-to-channel computer interface mechanism involving both hardware and software, in which the operating system in either computer can initiate transfer of data in either direction. We will show that PS, as a simulation tool is useful in designing computer systems. Perhaps more importantly, PS and the picture systems it produces provide a means of communicating computer system mechanisms between people.

Picture-system Models

Picture-system models grew from a study of a variety of notations used for describing computer systems at various levels of abstraction.[5,6,7] Primary among the notations studied were programming language notations such as the Vienna definition language,[8] graph-based notations such as Petri nets,[9,10] and application-specific graphic representations such as the contour model.[11,12,13] Of these, the application-specific graphic techniques are easily seen to be superior for people-oriented communication of computer system mechanisms. Picture-system models are a generalization of application-specific graphic techniques which encourages the representation of a computer system with drawings which are closely related to the abstraction being modeled.

The time-history representation of a computer system is necessary for its definition.[1] In picture-system models the time history takes the form of a sequence of snapshots, called a movie. From one snapshot to the next in a sequence there is a change in the drawing representing a change in the system state. To define a computer system by manually enumerating all possible movies would be a tedious undertaking. The next paragraphs explain how PS solves this problem.

Computer systems are naturally described as collections of finite-state elements which are interfaced to one another in various ways.[1] A finite-state element may be a passive element (its state changes are caused by other elements to which it is an interface) or an active element (it has transition rules which may change the state of the element or its interfaces). A computer system is a closed collection (no element has an interface outside the collection) of finite-state elements called a finite-state structure. The PS language provides a means of associating a unique drawing with each state of each element of a finite-state structure as shown in Figure 1.

From the description of a finite-state structure along with the associated graphics which is provided by the system designer, the PS system generates a finite-state system (computer system) and its isomorphic picture-system model, as shown in Figure 2. The picture-system model consists of a picture-set containing a picture for each system state (conglomerate of element states) and the state-transition graph for the system. The state-transition graph and the picture set fully determine all possible movies. Thus the PS system automatically generates an application-specific graphic representation of a computer system from a specification of the system as a collection of interfaced elements with their associated graphics.

This work was conducted while the author was an instructor at North Texas State University and a Ph.D. candidate at Southern Methodist University.

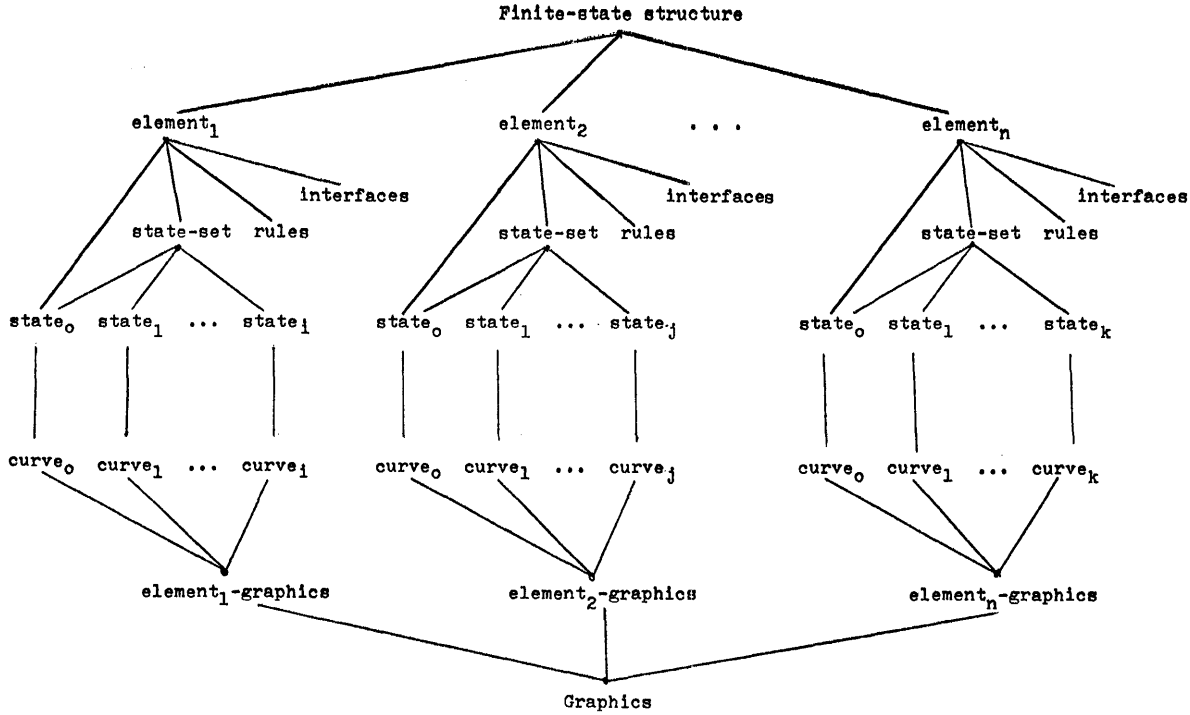


Figure 1.--Finite-state structure with graphics

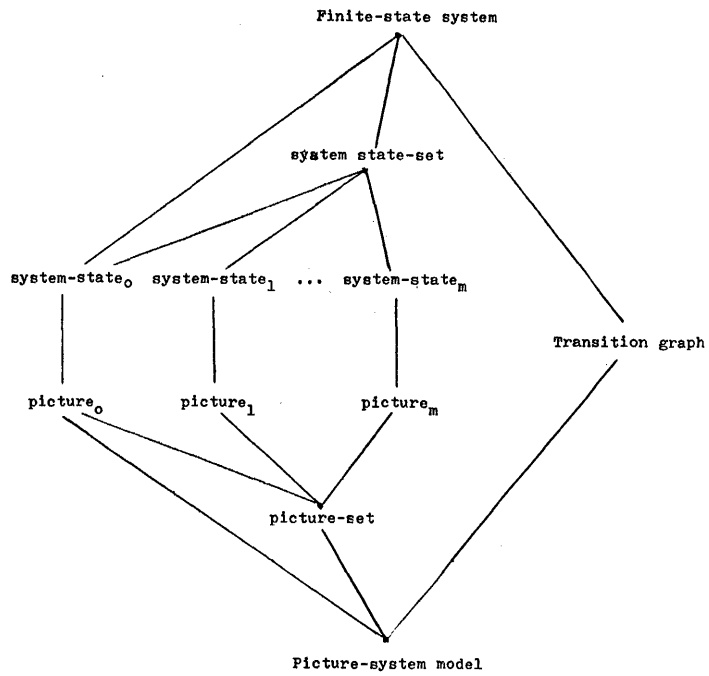


Figure 2.--Computer system and its picture-system model

The Computer Interface Mechanism

The computer interface mechanism to be designed as an illustrative example has five hardware components, namely: two central processing units with their respective operating system, two channels, and an interface unit. Transfer of data from CPUA to CPUB is denoted A2B. Similarly B2A denotes transfer the other direction. The mechanism for accomplishing a transfer is complicated by the fact that the computer requesting the transfer cannot predict when the other computer will respond. During the waiting time the channel must be free for use by other devices. This requires a two operation sequence by the requesting computer. The first operation is a request for the transfer. The second operation is the actual connection for the transfer. An example is in order.

Suppose CPUA wants to do an A2B transfer and no other operations are considered. The following events take place:

(1) CPUA transmits an A2B operation on channel A.
(2) The interface unit releases channel A and interrupts CPUB with the request.
(3) Eventually CPUB transmits an A2B operation.
(4) The interface unit leaves channel B connected while it interrupts CPUA with the matched operation.
(5) CPUA responds immediately by connecting channel A.
(6) The interface unit transmits the data then releases both channels.

At several points in this sequence CPUA may decide to also request a B2A transfer and/or CPUB may initiate transfer either direction.

The number of possible event sequences gets very high and the design gets complicated quickly. The actual complexity is well documented by the number of states in the picture-system model.

The PS Specification

The PS specification of a finite-state structure consists of:

(1) a list of all elements,
(2) the possibly null set of states associated with each element,
(3) the initial state of each element,
(4) the drawing associated with each state of each element,
(5) the possibly null set of interfaces for each element, and
(6) the possibly null set of state-transition rules for each element.

The PS language[4] eases the specification of similar elements by allowing references to previously defined element models (essentially macros). Each reference supplies its unique set of interfaces and can tailor the drawings associated with the element. In PS drawings are specified very concisely as curves using a method of encoding a curve as a sequence of operators.[14]

In the PS specification of the computer interface mechanism there are four types of elements (element models), namely:

(1) A CPU queue element type which defines the state of a queued or in-progress transfer from the point-of-view of the operating system in a CPU. There are four instances of this element type - one for each transfer direction in each CPU.
(2) The channel-to-channel interface unit (only one of these element types) which coordinates the four interface unit processes.
(3) An interface unit process element type which tracks a single CPU with respect to one direction of transfer. There are four interface unit processes. These processes are paired according to common transfer direction so that they can synchronize the actions of the two CPUs.
(4) A channel element type which is a passive element providing an interface line between a CPU queue element type and the interface unit with its processes. There are two channels, one attached to each CPU.

With the state of each element in the finite-state structure the PS specification associates a drawing. These drawings are used by the PS system to construct a picture representing each system state. Each picture can contain a background drawing over which the state-dependent drawings are superimposed. The background drawing for the computer interface model is shown in Figure 3. The drawing shows the various components of the system. It leaves space for the state-dependent drawings of the elements to be filled in for a particular system state. The following paragraphs discuss the states and their associated drawings for each element in the computer interface model.

Each of the four CPU queues have seven possible states which are represented in either the upper half (A2B transfer) or the lower half (B2A transfer) of a CPU. The seven possible states and their representations are:

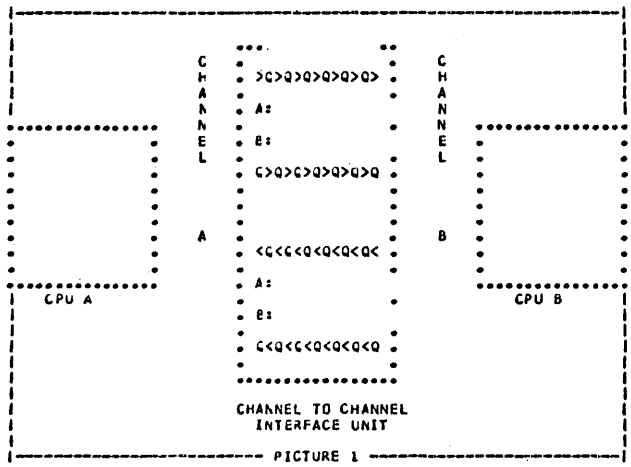


Figure 3.--Background drawing

(1) ( ) - no activity.
(2) ( ) (WANTED) - the operation has been requested by the other CPU.
(3) (WANTED) (WRITE) - the operation must be retransmitted due to a busy interface unit on an earlier try.
(4) (WRITE) (WRITE) - both the request and the operation are queued waiting to be sent.
(5) (WRITE) ( ) - the request has been placed on the channel.
(6) ( ) (WRITE) - the request has been accept- by the interface unit.
(7) ( ) ( ) - the operation has been placed on the channel.

Each of the channels have seven possible states which are represented in the area between a CPU and the interface unit. The states and their representations are:

(1) - no activity.
(2) >>>>>>>> - A2B operation or request being transmitted.
(3) <<<<<<<< - B2A operation or request being transmitted.
(4) --BUSY-- - interface unit response of busy to operation.
(5) --CEND-- - interface unit response of channel-end (operation-complete) to operation.
(6) --A<-B-- - interface unit interrupting CPU with A2B request.
(7) --A->B - - interface unit interrupting CPU with B2A request.

The interface unit has five states which are represented either in the top border of the interface unit box or, in the case of a transmitting state, in the center by a line connecting the two channels. The non-transmitting states and their representations are:

(1) IDLE - no activity.
(2) BUSY-A2B - the interface unit is in an uninterruptible sequence.

(3) BUSY-B2A - the interface unit is in an uninterruptible sequence.

The transmitting states and their representations are:

(1) >>>>>>>> - transmitting A2B.
(2) <<<<<<<< - transmitting B2A.

Each interface unit process has four states which are represented in the area of the interface unit labeled for the appropriate transfer direction and CPU. The states and their representations are:

(1) EMPTY - no activity.
(2) REQUESTED - the CPU has requested the operation.
(3) READY - the CPU has been notified of the request from the other CPU or has initiated a matching request prior to notification.
(4) CONNECTED - the CPU is connected by its channel to the interface unit waiting for the transfer.

The Picture-system Model

Several submodels of the computer-to-computer interface mechanism are of special interest. The one to be considered here is called the A2B submodel. It considers only transfers in the A2B direction. This submodel shows that the system can start in an idle state, initiate an A2B transfer from either or both CPUs, complete the transfer, and return to the idle state. The A2B submodel has 66 states. The transition graph is shown in Figure 4. Since the graph is in straight order[15] and the backward arcs are drawn, it is clear that the graph is a single strongly connected region; therefore, any state is reachable from any other state and no hang-up states or cycles exist in the A2B submodel. A movie derivable from the transition graph is shown in Figure 5. This movie illustrates the case where CPUA initiates an A2B transfer and before the interface unit has noticed the request, CPUB initiates a matching A2B transfer. The interface unit need not notify CPUB of CPUA's request, but must match the requests.

It is interesting to consider the B2A submodel along with the A2B submodel. Due to the symmetry of the system, the models are alike except that the roles of the CPUs are interchanged. A lower bound on the number of states in the full model of the channel-to-channel computer interface mechanism has been established at 1,781 by considering the relationship of the full model to its A2B and B2A submodels.[2] This number of states exceeds the number for which a transition graph can be reasonably generated and analyzed by PS. Therefore an alternate method of validating the model is desirable. The validation was done using knowledge of submodels. The A2B and B2A submodels show that the system operates properly when there is no channel contention by opposite transfer requests. The cases where contention is involved were validated by constructing three additional submodels.[2] The largest of these generated 339 states.

Conclusions

This paper has shown, via an illustrative example, that picture systems along with PS to automate their generation and analysis are useful in defining, simulating, and communicating

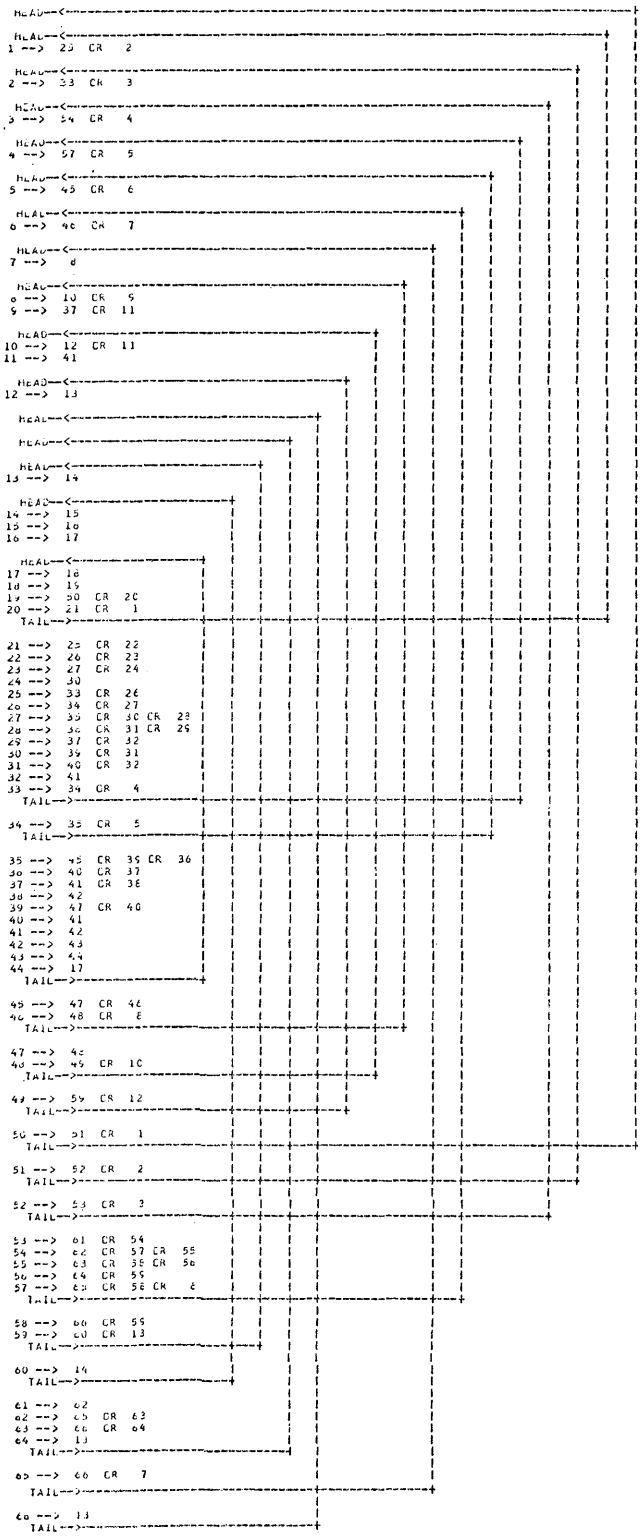


Figure 4.--Transition graph

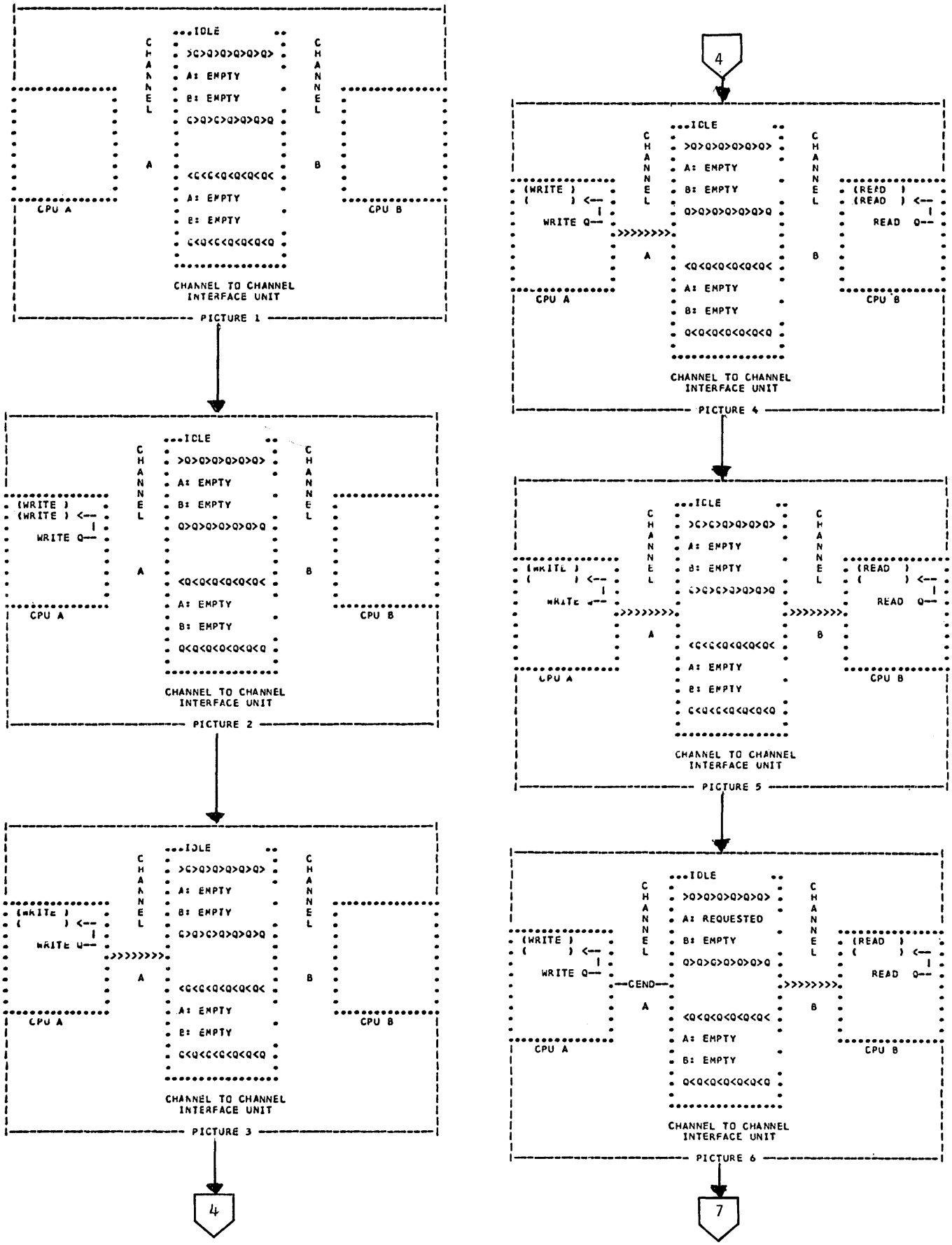


Figure 5.--Movie

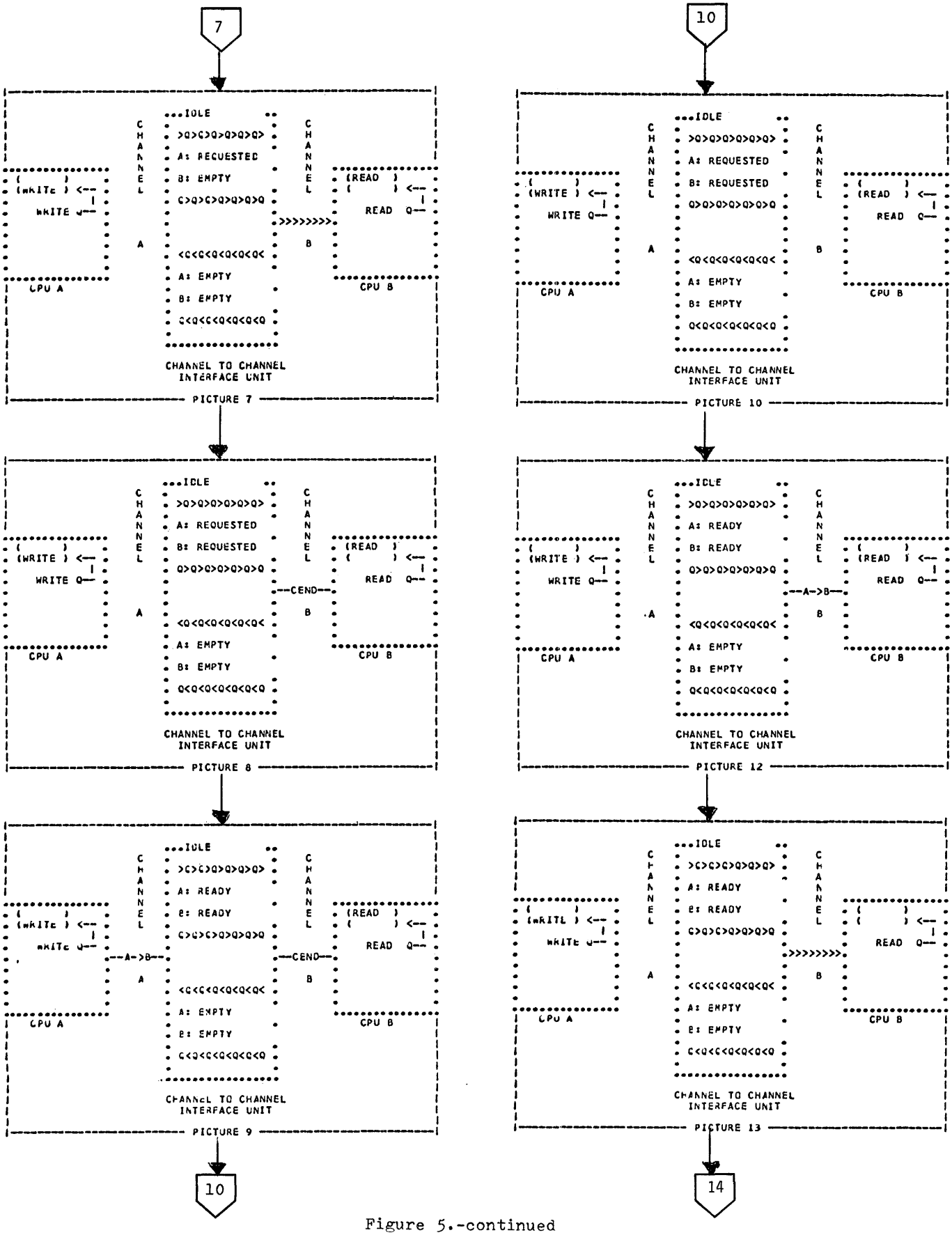


Figure 5.-continued

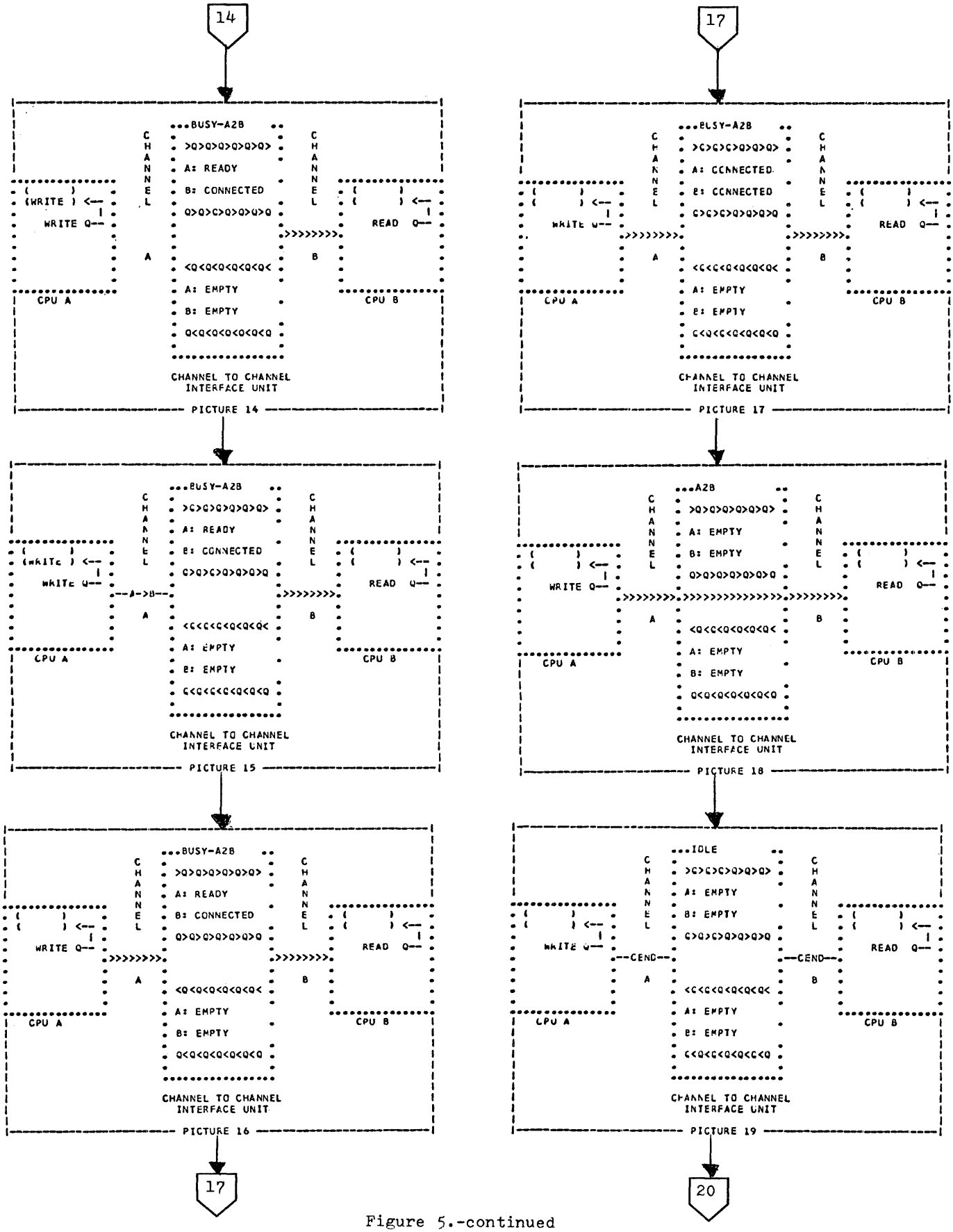


Figure 5.-continued

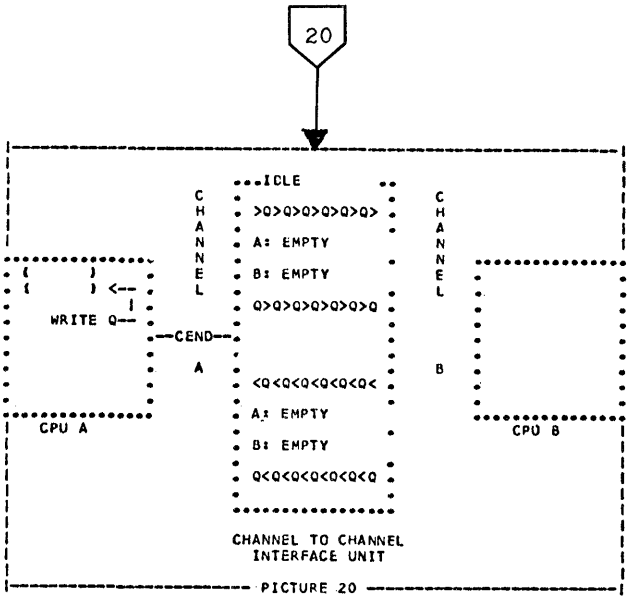


Figure 5.-continued

computer system mechanisms comprised of hardware and software at the early design stages when the descriptions must necessarily be at a high level. The need for modeling of highly parallel asynchronous computer system mechanisms at the early design stages is shown by the very large number of states in the relatively simple mechanism given as the example in this paper. The difficulty of communicating these mechanisms is well-known. The usual problem associated with using a model as a means of communication is that the modeling language must be understood by all concerned. Achieving common understanding of a modeling language is difficult enough in a hardware group or in a software group. To achieve such understnading across these two disciplines, which is necessary for document-hardware/software mechanisms, would appear impossible. The use of PS avoids this difficulty. Only the picture set and the transition graph need be broadly understood. Knowledge of the PS language can be restricted to a single individual who writes the PS specification of the finite-state structure from which the PS system generates the picture-system model. The pictures in the picture-system model can be chosen, as they were in the example given here, to build visually the understanding of a computer system by giving our minds a crutch for thinking about the system. It is not mere coincidence that the phrase "I see" is used to mean "I understand."

Acknowledgements

The author gratefully acknowledges helpful discussions with Dennis frailey, James Isaacson, Bob Korfhage, Bill Nylin, Dan Scott, and Rob Smith.

References

1. Bell, C. G., and Newell, A., Computer Structures: Readings and Examples, Mcgraw-Hill, New York, 1971.

2. Isaacson, P., "Picture-system models and computer system design," Ph.D. dissertation in preparation, Computer Science Department, Institute of Technology, Southern Methodist University, Dallas, Texas, 1974.

3. P. Isaacson, "PS language definition," ACM SIGDA Newsletter 4(September 1974), pp. 20-28.

4. Isaacson, P., "PS: a tool for building picture-system models of computer systems," Proceedings of the Third Texas Conference on Computing Systems, November, 1974.

5. Baer, J. L., "A survey of some theoretical aspects of multiprocessing," ACM Computing Surveys 5 (1), pp. 31-81.

6. Brewer, M.A., Design Automation of Digital Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

7. Miller, R. E., "A comparison of some theoretical models of parallel computation," IEEE Transactions on Computers C-22 (8), pp. 718-727.

8. Lee, J.A.N., Computer Semantics - Studies of Algorithms, Processors and Languages Von Nostrand Reinhold, New York, 1972.

9. Hold, A., and Commoner, F., "Events and Conditions," Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Association for Computing Machinery, New York, pp. 3-52.

10. Noe, J. D., and Nutt, G. J., "Macro E-nets for representation of parallel systems," IEEE Transactions on Computers C-22 (8), pp. 718-727.

11. Johnston, J. B., "The contour model of block structured processes," Technical report 70-C-366, Research and Development Center, General Electric, Schenectady, New York, 1970.

12. Berry, D. M., "Block-structure: retention or deletion," Proceedings of the Third ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1971.

13. Organick, E. I., Computer System Organization: The Burroughs B5700/6700 Series, Academic Press, New York, 1973.

14. Freeman, H., "On the encoding of arbitrary geometric configurations," IRE Transactions on Electronic Computers, pp. 260-268.

15. Earnest, C. P., Balke, K. G., and Anderson, J., "Analysis of graphis by ordering of nodes," Journal of the ACM, January 1972, pp. 23-42.

REFERENCE CONCEPTS IN A TREE STRUCTURED ADDRESS SPACE

Lennart Löfgren
SAAB-SCANIA AB, Datasaab Division
Linköping, Sweden

Summary. We start by summarizing our goals with an emphasis on our requirements on references. A dynamically varying space of virtual addresses is postulated. This space has the structure of a rooted tree. All references considered are essentially relative: to a target from a context. A few reference concepts are defined and explored, together with a unified operation of qualification and offsetting and one operation for forming references from new contexts.

1. Introduction

1.1 General

For a long time we have recognized the fact that it wastes core space to carry the full address of the computer in the address part of each addressing instruction. S/360 works with a base address and a 12-bit displacement, the Datasaab D5 mini computer uses an 8-bit displacement and an elaborate scheme to select a base address.[6]

There is one more reason to avoid introducing full absolute addresses in the program code. Generally speaking, this is flexibility: self-relocatability of program code, and capability for dynamic linkage while retaining pure procedure code.

There is reason to believe that these benefits could be achieved on a more universal scale if a generalization of this technique could be found, extending it to all occurrences of addresses within a computer memory.

Before we proceed, however, a number of concepts of this issue should be clarified, such as:

(i) Where do addresses occur outside of machine instructions?
(ii) What precisely do we mean by addresses?
(iii) What can a generalized technique roughly be like?

About (ii). We have become accustomed to the idea that the address information in machine instructions is hardware modified in various ways to produce a memory address capable of controlling the storage accessing circuitry. For this modification further information is supplied, such as page start addresses, base register and index register contents, all of which we could lump together under the term supplementary address information. The word "address" has often been used to refer to both the displacement in instructions and the mentioned supplementary information. In programming "address" is taken even more generally to mean almost any operand (or parameter) of an address (in the previous sense) computation, particularly if it can be regarded, in isolation or in context, to identify a data object.

Definition of reference. For the purpose of this paper, avoiding futile discussions of terminology, I replace this loose usage of "address" and take reference to mean a piece of information that permits accessing of a data object within the address-and-storage space under consideration. This space may in principle be virtual or real, although in sect. 2 a virtual space will be postulated.

About (i). References are frequently used wherever we need to represent connections between elements in data structures, except in cases where we are able to, and choose to, do so by storing pairs of connected elements adjacently with respect to storage addresses, or on addresses determined by an arithmetic rule. For references one quite commonly uses memory addresses today. These are efficiently handled by current hardware, e.g. when used as indirect addresses, or after they have been loaded into index or base registers etc. Other variants of references are also employed, as was noted About (ii), last paragraph.

The need for references is basic and ubiquitous. Could one unified, less problematic, form be provided they would flood our computers.

About (iii). We develop a form of reference that is both suitable for a majority of programming needs and at the same time efficiently hardware interpretable. The latter means in particular that its currently intended corresponding machine address should be automatically computed when the reference is engaged for accessing. Also the most commonly demanded manipulations on references should be available as programmable hardware operations.

How can this be done? We can analyze the usage of such programmer defined information that is used for references, and offer a unified concept that covers a majority of such usage. This concept has to be sufficiently simple and uniform to be hardware implementable. It must of course also cover the simple cases of instructions' address parts and today's use of plain memory addresses.

This paper investigates a proposed such concept. The idea is to construe references as partial, or incomplete, address information.

1.2 Preliminaries, definitions

Definition of qualifiers. A reference along these lines needs supplementation before it can be engaged in an access. The pieces of information needed for this supplementation also require careful attention. I call them qualifiers. They may be capable of completely supplementing a given reference into a unique identifier within the storage space under consideration, full qualifiers, or they may contain only part of what would be needed for this, partial qualifiers. Some of them might constitute objects for manipulation by application programs others might not. They can frequently be construed also as references, all by themselves. The reason why partial address information will be more advantageous to use for reference than a full address is that we can exploit locality.

Definition of locality, data objects, memory regions. To define locality more generally we need in the background the concepts of data objects and data structures and also of addressspace regions. It would carry too far to define them very rigorously.

An abstract data object is an elementary object, or a data structure; a data structure is formed by connecting data objects, one way or the other. A(n) (abstract) data object represents something, and should be called information object. A concrete data object is a representation of an abstract one, on a form that allows it to be stored in a memory space (a virtual or physical memory). The stored concrete object must represent the elementary objects as well as all object connections.

An address space region is a subset of the space of addressable storage cells, subject only to the restriction that there must exist some part P of the address and some interval, such that all cells in the subset have addresses whose parts P lie inside the interval.

Data object locality means that from within the given object there are more frequent connections with certain objects (possibly - and commonly! - including itself) than with others.

To draw on locality, we try to store concrete objects (structures) in memory regions. As many sub-objects should thereby be assigned to sub-regions as possible. Object-to-region correspondence should first be satisfied for objects with a high data object locality. To the degree that we succeed, we would get reference locality, in the following sense:

References in a given region refer into certain regions more frequently than into others.

Ideas; address reduction. If we have reference locality in a region R with respect to some region S, let us represent references from within R to within S by a (virtual) memory address where a certain part P(S) is reduced or omitted. P(S) is identical to, or part of, that part of addresses into S which, by the definition of region, are confined to some interval. We reduce P(S) by subtracting from it some number in the interval, e.g. the lower bound, thereupon deleting all leading zeros in the representation of P(S). The latter is the explicit manifestation of the information loss that we want to bring about. If the interval consists of a single number, the result will be zero and we omit P(S) altogether from the reference. We term this process reference formation through address reduction. The supplementary information needed to reconstitute a full virtual address, the qualifier, is seen to be the subtracted number (together with some indication as to where in the address P(S) is, or should be, situated).

We will propose one unifying abstract data type to apply both to references and qualifiers, partial or full. One single operation will be devised for recombining a reference with its full qualifier into a full address, for combining a reference with a partial qualifier into a reference presupposing a smaller qualifier and for combining full or partial qualifiers into more comprehensive qualifiers. - In fact we devise a construct that the designer can employ for either of these purposes according to his choice, just as a binary word can be used to represent many kinds of numbers. We also devise a single operation for address reduction, which takes for operands the address and the desired qualifier and produces a reduced address, suitable as a reference under that qualifier. Because of the unification of concepts, qualifiers can also be reduced, making them "more partial". We want to develop these ideas in the context of a comprehensive semantics for a virtual addressing system, as has been requested by Dennis.[2]

1.3 Advantages - goals

The advantages we set out to buy with our complication of reference representation are twofold, viz.:

(i) Conservation of space
(ii) Flexibility by avoiding unnecessary commitment

With both of these objectives reached we have torn down the worst obstacles against catering for a long desired programming generality, formulated e.g. by Dennis in his well-known requirements:

Ability for a program module to:

1) create information structures of arbitrary extent
2) call on procedures with unknown requirements for storage and information structures and
3) transmit information structures of arbitrary complexity to a called procedure.

Let me qualify this statement. To cater for requirement 1) and 2) we would need a very large space indeed. Of course a practical limit could be set, but this would mean that uncomfortably large fields would have to be reserved for references were these to be represented as addresses. Furthermore, requirement 1) and 2) are reasonably implemented by the well-known technique of sparse addressing in a virtual memory.[8] In our terminology this means finding for each data structure to be created a region, which the data structure is unlikely to overflow while it grows up. Doing so for our several structures will aggravate the problem of large address fields. We solve this problem by simply never storing any full addresses but only partial ones, having limited size. Advantage (ii) also derives from partialness: We identify data objects only as within regions. Identification of regions is done by the qualifiers, which need to be stored only once for each context.

If we read 1) and 2) literally, it would require us to extend our entire virtual address space occasionally. Any occurring full addresses would then have to be extended - an impossible task, were they scattered about as references. The advantage (ii) of using partial addresses for references admits such extensions. Only the full qualifiers as defined, need to be extended. Their number will be held to a minimum; wherever an application program needs a qualifier, a partial one will do - and this is to be enforced. Furthermore advantage (ii) will help when a data structure overflows its region. We can extend the region by a virtual move (to be defined in sect. 4) of any hindering objects to another region, or do such a move of the growing object itself. A virtual move is achieved by changing a qualifier - the references themselves are uncommitted (or unbound) with respect to the global location within the space, of the regions referred into. Advantage (ii) could possibly also be helpful in catering for requirement 3). For one parameter passing method is to do moves, physical or virtual.

Turning to more nearby aspects of what we want to achieve, we note about today's minicomputers that there is a demand for much more memory, virtual or real, than is consistent with the wordlength one likes to have. A consistent use of a reference system based on the present method allows any amount of storage space to be addressed from a minicomputer's program without resorting to ad hoc features.

Quite generally, in systems architecture, we will be able to rely on large virtual memories, and on references, to an unprecedented degree. We have e.g. hitherto, with few exceptions [4], avoided to have large data objects resident in virtual memory spaces over time periods, comparable to the lifetimes of software systems. Considered over such periods data objects grow and shrink, undergo reorganizations and proliferate generations, all things leading to the Dennis requirements. References in our sense can be used freely for most identification needs, thus lessening the importance of the current technique to frequently reassign the names of some available name space in the system (such as a virtual memory) to different, externally named objects. [5]

We can sum up the goals of this work like this:

We want to devise a method that allows that

(a) the space of addresses can be extended
(b) that space extensions can be made by way of insertion of new regions between previously adjacent address regions
(c) that intra-regional references are unaffected by extensions, thus afortiori not lengthened
(d) that intra-regional references are self-relocatable w.r. to simultaneous relocations of the whole region content.

1.4 Applications, and scope of present work

The work accounted for here is intended as a conceptual basis for a new architectural feature, based on a multilevel virtual memory and includes no actual architecture design. This feature has been intended both for a new, extended, model of an existing minicomputer (the Datasaab D5/30 [6]) and for an entirely new computer architecture based on an Intermediate High Level machine language. It is also being applied to an interpreter (with both a soft and a microprogrammed version) for a language SILL (System Implementation Language, Linköping; implemented at Linköping University [7]).

All intended architectures, and indeed any architecture, reasonable as a candidate for application, must protect all full virtual addresses and full qualifiers from access from non-privileged programs. Programs refer to data objects via a fixed, small set of current root contexts, only generically identified as one of a few standard contexts, such as "current procedure code region", "current working data region" or "current link segment region". This identification is built into the machine languages used, even in the case of the mini (D5). The latter already has a rudiment of this feature.

Supervisory software, executing in privileged mode, or in case of SILL, the program and microprogram code of the interpreter, maintains one full qualifier for each of the standard contexts in each running task. Hardware or interpreter (micro) program automatically supplements a reference with the qualifier of the identified standard context as soon as the program submits that reference to be used for access. Access to regions outside of current standard contexts is provided either by the simultaneous addressing of a qualifier and provision of a reference to be qualified (something similar to indirect addressing with post indexing), or by loading a qualifier into a "temporary context indicator", which in its turn can be identified in instructions, alternatively to one of the standard contexts.

All of the user maintained qualifiers are partial ones. Which further qualifier is understood in these cases is determined by special rules, which vary between architectures.

A completely different approach to automate the handling of referencing devices is found in reference 5. [5]

2. Structure of Address Space

Our method is based on a multilevel virtual address-and-storage space that has structural properties as described in this section.

The address space has the structure of a rooted tree, i.e. a directed graph that is connected and mesh free. The tree depicts the space in the following way: Its leaves are the true virtual storage cells. Such a cell can be selected by a virtual address; here denoted Absolute Virtual Address, AVA, for reasons that will appear later. An AVA should be considered a string of subfields that represent non-negative integers: $a_0, \ldots, a_{n-1}$ $(n \geqslant 1)$. (So that if field i is b_i bits long $0 \leqslant a_i < 2^{b_i}$, $0 \leqslant i < n$). Naming the AVA A, we write $A = (a_0, a_1, \ldots, a_{n-1})$. Looking at the tree we number the outgoing edges from any node N: $0,1, \ldots, f_N-1$, where f_N is the branching factor of node N.

To select a leaf/cell, given the AVA $A = (a_0, a_1, \ldots, a_{n-1})$ we proceed as follows: Consider the root node. Pick the leftmost field of A, representing a_0, and select edge no. a_0 from the root node. Then consider the node entered by the selected edge. Pick next field of A, a_1 and select edge no. a_1 out of those emanating from that node. Proceed similarly field by field until and edge is selected by a_{n-1}. The node where this edge ends is the node selected by A. If this happens to be a leaf, i.e. a terminal node, we have identified a true storage cell. If not - we have identified something in the machinery: a thing that corresponds to a non-terminal node of the tree. Peeping into possible implementations, we dare guess that that thing is a table of the outgoing edges, or rather things that these depict. To make edge selection easy, the table is linearly stored, so that an edge can be found by indexing. An edge stands for an address or other identifier of the node where it ends. Terminal nodes, leaves, are no tables but fixed length cells, storage places for words. There is no need to store their incoming edges - their addresses - separately if the set of leaves under the same next higher level node - the prune - is allocated linearly. The edges/addresses are computed simply by indexing

from its start address. Thus the next-to-terminal nodes need have no table counterpart. Instead the incoming edge to such a node corresponds to the start address of the prune, and the node might stand for the prune as opposed to its elements.

We will put no restrictions on the tree such as uniform depth or branching factor - these are things up to the designer. We also note that a meaningful AVA must not contain a subfield representing an edge number greater than the branching factor minus one of the node it applies to. Reacting to violations also belongs outside of this account. The physical location of the tree root is supposed to be known to the system through means devised by the designer. We also note that, over time, the space is allowed to change in size and structure.

3. Reference concept I: Generalized offsets and RVA's

3.1 Preliminaries

Given an address space as of sect. 2 we start looking for workable reference concepts on the basis of the considerations in sect. 1. We note that any subtree is a region, in the sense defined in 1.2. We make the decision that the candidate regions that we are primarily going to consider for locality based address reduction will be the subtree regions. - As we have seen in 2, any node of the tree is identifiable by an AVA. We borrow from graph theory the simple truth that a subtree has a unique top node which is its root. Consider all the AVA's to cells/leaves in a subtree. They have a common part in their addresses (AVA's) and this part is identical with the whole AVA of the subtree root. So to form partial addresses to them, we can simply remove the root AVA from their respective AVA's! The reason we can do so is that the interval to which the designated part P(S) of their addresses (AVA's) is confined, can be considered a single number, viz. the root AVA (Cf. sect. 1.1). Thus this one is also the full qualifier.

We also have the need to form partial qualifiers. And we have noted the desirability of a concept unifying the notions of qualifiers and references. Furthermore we know that the interval used in the definition of a region is not unique - we should try to leave the choice of interval with the systems designer. In the back of our minds we also observe that a most common form of programmer defined address reduction is the use of relative addresses, i.e. address offsets relative to an address called base address. In linear address spaces such an offset is simply the target address minus the base address.

As a first approximation we define an analog of address difference, commonly called offset, and we use this term in a generalized definition:

Let $A_M = (a_0, \ldots, a_{m-1})$ and
$A_N = (b_0, \ldots, b_{n-1})$ be
AVA's of nodes M and N respectively ($m, n > 0$).

Make $k = \max(m, n)$.

If $m > n$ define $b_n = b_{n+1} = \ldots = b_{k-1} = 0$
If $n > m$ define $a_m = a_{m+1} = \ldots = a_{k-1} = 0$ } (1)

We define offset of M with respect to ("from") N:

$$\text{offset}(M, N) = (a_0-b_0, \ldots, a_i-b_i, \ldots, a_{k-1}-b_{k-1}) \quad (2)$$

for $0 \leq i \leq k-1$.

This notion of offset has the serious flaw that it is not a one-to-one function in its first argument when holding the second one fixed (nor vice versa). Because consider what happens with offset(NT, N) where NT is some non-terminal node and N is any node that has a greater distance, say 1 levels further down from the root than NT. Then if AVA of NT has m components, offset(NT, N) must have k = m+1 ones. Consider offset(X, N) to any node X with AVA's like NT's except for having one up to l-1 further fields suffixed that are zero. These offsets are all equal, and equal to offset(NT, N). Nodes with this property are situated along an "all-zero-edge path" starting in NT.

Because the offset is to be used as an identifier of its first argument, it has to be a bi-unique function (precisely: in its first argument holding the second one fixed). So this situation has to be remedied. We do this by adding the requirement that we know in advance either that the target node is terminal or which level it has in the tree. We term this information the s t a t u s of the node.

3.2 Properties of first approximation

Some of the most important properties are listed here:

(i) Use for addressing. The offset can be used for addressing (selection) in the presence of information on which node it is an offset from, called the base node, and on the status of the target node. This can obviously be done, as follows: The base node B is supposed to be known by its AVA. We can now find the AVA of the target node T, given the value of offset(T, B) by adding this componentwise to the path designation of B:

Let
AVA of $B = (b_0, .., b_{n-1})$
and $\text{offset}(T, B) = (d_0, \ldots, d_{k-1})$

Necessarily, by (1), $k \geq n$. (3)

Then
AVA of $T = (b_0+d_0, .., b_i+d_i, .., b_{k-1}+d_{k-1})$ (4)

with $0 \leq i < k$ and $b_n = b_{n+1} = \ldots = b_{k-1} = 0$.

If one or more of the trailing components of the k-tuple of (4) are zeros the status information assumed to be known for the target node T is used to determine which node with this offset is the one addressed.

(ii) Offset addition. The next important property of offsets to be considered is the elimination of an intermediate base by addition. We define addition of offsets as componentwise addition, considering any missing trailing components in one of the offsets to be zero. Assume:

$$P_1 = \text{offset}(B_1, B_0) = (e_0, \ldots, e_{m-1}) \quad m \geq 1 \quad (5)$$
$$P_2 = \text{offset}(T, B_1) = (f_0, \ldots, f_{n-1}) \quad n \geq 1 \quad (6)$$

Then
$P_3 = P_1 \underline{add} P_2 = (e_0+f_0,\ldots,e_i+f_i,\ldots,e_{k-1}+f_{k-1})$ (7)

where $k = \max(m, n)$ and $e_m, e_{m+1},\ldots = 0$ or $f_n, f_{n+1},\ldots = 0$ in the cases $n > m$ or $m > n$ respectively. $0 \leq i < k$.

Up to a possible presence or absence of one or more trailing zero components P_3 is the same as offset(T, B_0).

(iii) Offset subtraction. One may want to find the offset from a new base when offsets to the new base as well as to the target are given. A condition for meaningful application of this operation is that the given offsets should be relative to the same base. Taking

$(e_0,\ldots,e_{m-1})$ to denote offset(T, B_0) = P_1 (8)
$(f_0,\ldots,f_{n-1})$ to denote offset(B_1, B_0) = P_2 (9)

we define

$P_3 = P_1 \underline{sub} P_2 = (e_0-f_0,\ldots,e_i-f_i,\ldots,e_{k-1}-f_{k-1})$ (10)
with $0 \leq i < k$, $k = \max(m, n)$, $e_m = e_{m+1} = \ldots 0$ or $f_m = f_{m+1} = \ldots 0$ as under (ii).

P_3 is the same as offset(T, B_1), again except for possibly trailing zeros.

3.3 Revised definitions

Two offsets are equivalent if and only if they are componentwise equal, when possibly missing trailing components in one of them are considered zero.

Definition. For the next approximation to our unified reference concept we pick equivalence classes of offsets with respect to the stated equivalence relation. We term these Relative Virtual Addresses, RVA. We can represent RVA's by offsets, and thereby drop or add trailing zeros freely without changing denotation. (Cf. fractional notation with decimal point.) As with offsets we still need the status of the target - and of course a base node - to uniquely identify a node by an RVA. An RVA represented by an offset(T, B) we write RVA(T, B).

The operations add and sub are trivially carried over to RVA's by prescribing that they be performed on arbitrarily chosen offsets representing the respective operands. That this works can be trivially shown, which will not be done here. We note that, using (5), (6), (7) and (8), (9), (10), respectively, we get what we want, viz.:

RVA(T, B_0) = RVA(B_1, B_0) add RVA(T, B_1) (11)
RVA(T, B_1) = RVA(T, B_0) sub TVA(B_1, B_0) (12)

For convenience and uniformity we include also RVA's standing for the same things as AVA's; call them Full Virtual Addresses, FVA's. An FVA is simply an RVA with respect to the root node of the space. We can now use add and sub when operands include absolute virtual addresses, and generally apply all the algebraic properties of RVA's also to FVA's. It is easily verified that an FVA has the same components as its corresponding AVA (disregarding irrelevant trailing zeros).

4. Applicability of RVA's

We contend that RVA's almost serve as a basis for the unifying data type that we promised at the end of 1.2. Regard RVA's tentatively as reduced addresses: P = RVA(T, B) is the address of a node T, reduced by a qualifier, which coincides with the AVA of node B, A_B. Recombination of P into a full address is done by performing A_B add RVA(T, B) regarding A_B as an FVA. - To actually find RVA(T, B) we simply do A_T sub A_B, regarding also A_T, i.e. the AVA of T, as an FVA. Obviously we can find RVA(T, B) even if we do not have the AVA's A_T and A_B; it is sufficient to have RVA's with respect to some common node, "base node", thus not necessarily the root node. The final RVA(T, B) to be recombined will commonly be the result of a number of steps of "de-reduction" (= qualification) by partial qualifiers. Some of these steps might be performed in advance by combining partial qualifiers. All of the flexibility suggested in 1.2 with regard to these manipulations is clearly there: making qualifiers more or less comprehensive by doing add or sub with desired qualifiers; making references that are reduced addresses more or less reduced by doing sub or add respectively with a desired qualifier. Add and sub are associative. Reduction of an RVA with respect to a given region can be done through sub by any qualifier RVA, which is at the same time a reference RVA to a node, terminal and within the region, or non-terminal and within a subtree whose leaves are all within the region. (For subtree regions the root node of these is a natural candidate, see sect. 6.)

RVA's have two flaws, however, when considered for usage as references. The first has been mentioned: It is their dependence on "status" for unique identifier capability (3.1,end). The second is that they would, as references, after all have to carry implications on matters wholly outside the region within which they refer, no matter how they are reduced with respect to that region. This is contrary to objective (ii) of 1.3, as it represents an unnecessary and, as we shall see, detrimental commitment.

The careful reader may have noticed the second flaw in that we have done nothing like "deleting leading zeros", as is required for address reduction (1.2). In fact, consider two nodes M and N both belonging to a proper subtree of the space, and the form this sample RVA(M, N):

RVA(M, N) = (0, 0, ..., 0, d_i, d_{i+1}, ..., d_{k-1}) $(1 \leq i < k)$. There are at least as many leading zeros ($\geq i$ zeros) as there are levels 'above' the subtree, i.e. edges along the path between the subtree root node and the space root node. These zeros do nothing but indicate which levels in the tree the fields d_i, d_{i+1},..., d_{k-1} apply to. Doing so they presuppose that above the subtree there exist indeed that many levels in the address space. This is a matter global to the subtree region and therefore such zeros are a bad commitment. We do indeed want to allow global scale reorganizations of space structure without having to go in and modify references local within a region that is not internally affected. This is the point of 1.3 (ii) and (c) and the meaning of self-relocatability, 1.3 (d). We must, e.g., be able to insert or delete nodes dynamically on the levels above the subtree, to extend or shrink the space where needed. This is one kind of virtual move of a subtree. Giving all the nodes of the subtree new addresses, AVA's, is the meaning of the phrase Virtual Move.

The subtree nodes must not have to be identified by new references after this operation - as long as the references are considered as identifiers only within the subtree region.

To conclude, we note that RVA's might violate also our objective (i): If all the zeros are represented explicitly as zero fields, space is wasted and RVA's would have unbounded lengths. However, this could be alleviated by a smarter representation, simply by leaving out all leading zero fields and representing the number of these in a separate new main component. This removes none of the flaws, but it will serve us as a useful construct to aid in the construction of a better concept.

5. Reference concept II: Local Virtual Address

5.1 Constructing LVA's

Consider an RVA representation, called FRVA, abstracted from specific formats, where possible leading zeros may (but need not) be left out and the number of these thus dropped is given in a main component Z of the representing data object. Clearly $Z \geqslant 0$. So take an FRVA to be a pair:

$$\left.\begin{aligned} &FRVA(T, B) = (Z, D) \\ &D = (d_Z, d_{Z+1}, \ldots, d_{k-1}) \end{aligned}\right\} \quad (13)$$

where d_i are integers, $Z \leqslant i \leqslant k-1$, $k \geqslant 1$, T and B are arbitrary nodes and D is the (k-Z)tuple of remaining fields of RVA(T, B) after Z leading zeros have been left out.

We will construct another abstract data object lacking the flaws of the RVA (and the FRVA). We assume an FRVA is given as in (13). First we pick one of the new object's components to be D. Assuming (13) we further define:

$$C = \text{level of } B - Z \quad (14)$$

(In graph theory the level of a node N in a tree is the number of edges on the path between the tree root node and N.) Are these components sufficient? In order to do add and sub we need to know the complete string of RVA components, which can be found from the (k-Z)tuple $(d_Z, \ldots d_{k-1})$ and Z using (13). To find Z from C, by (14) we need the level of B. With RVA's we did not have to know the status of any of the nodes involved when we did addition or subtraction. We are not going to give up this valuable property. So we consider a further component that could help us, the level distance E to the target node T from the base node B, of (13).

$$E = \text{level of } T - \text{level of } B \quad (15)$$

We are now in a position to define a Local Virtual Address, LVA, of a node T with respect to a node B:

$$LVA(T, B) = (C, D, E) \quad (16)$$

where C and D must come from (13) and (14), using nodes T and B.

5.2 Demonstrating that LVA's work

Let us examine how LVA's can be made to work in accordance with the suggestions in sect. 1.2 and 1.3. In sect. 4 we have noted how RVA's could be used in this way were it not for their noted flaws. The close connection of LVA with RVA, rooted in LVA definition, allows us to regard LVA's as representations of RVA's. Our strategy will be to show that all wanted properties, as of sect. 4, are inherited by LVA's while the unwanted ones are not.

LVA's as representations of RVA's. As long as we regard the address space structure as given, or fixed, LVA's can be considered to represent RVA's. Sect. 5.1 shows how for all RVA's we can make LVA's. We also will define operations, LVA-add and LVA-sub applicable to LVA's and with the property (called distributivity) that if R_1 can represent P_1 and R_2 can represent P_2 then R_1 LVA-add R_2 can represent P_1 add P_2 if the latter operation is meaningful as of 3.2 (ii), and similarly for LVA-sub and sub. (R_1 and R_2 stand for LVA's, P_1 and P_2 for RVA's.) With distributivity given, it is clear that from (11) and (12) we would get the analogs:

$$LVA(T, B_0) = LVA(B_1, B_0) \text{ LVA-add } LVA(T, B_1) \quad (17)$$

$$LVA(T, B_1) = LVA(T, B_0) \text{ LVA-sub } LVA(B_1, B_0) \quad (18)$$

From here on we will call the LVA operations add and sub, as this can cause no ambiguity. The LVA operations will be defined by rules to compute their results from the operand LVA information alone, i.e. needing no auxiliary data such as e.g. status of nodes involved. We will observe while we proceed how the noted flaws of RVA's are not present in LVA's.

Effectiveness for accessing. Finally we will review the conditions for engaging a reference reduced-address in the accessing of the object it identifies. We do so while noting that LVA's are a kind of such addresses, as well as qualifiers and full addresses. We show that effective accessing will be achieved.

Properties of LVA's. An LVA is constructed from an RVA by purposely dropping information on which depth its components are supposed to apply to. (The depth means the number of levels from the root. - Components of an RVA are said to apply to the level that the corresponding components are associated with in a pair of AVA's defining the AVA, cf. 2 and 3.1.) Otherwise LVA's and RVA's are the same thing. This lack of information in an LVA, however, implies that it does not represent a unique RVA: All RVA's gotten by prefixing zeros arbitrarily to the D component of an LVA (zero or more), can be represented by that LVA. When used, however, an LVA is associated with a particular RVA. This comes about when it gets interpreted as from a specific base node, as does the level of the target node, something we also will need: The E component defines this by (15). - Thereby, also, the first flaw of RVA's (end of 3.1) is removed in LVA's. So we will be allowed to talk about "the number of zeros understood to be prefixed", i.e. Z, and of the levels of the two nodes. These things are references to absolute depth. They are parameters in the construction, however, and they are all eliminated in the computing rules.

The lack in an LVA of information on depth, or, which is the same, the number of nodes between application region and root, is precisely what allows insertion or deletion of such nodes in the space, and indeed also self-relocation under move, real or virtual, of the whole contents of a region to any different, but similarly structured, address space region.

LVA's lack the second flaw of RVA's that was noted in sect. 4. This means an extension of a similar flexibility achieved already in the RVA's: These allow self-relocating moves but only within the same, constant, level in the space.

Devising LVA add and sub. The operations will be in essence the same things as add and sub of RVA's or offsets, as defined in sect. 3.2 The only distinguishing thing is that in adding (subtracting) offsets we add (subtract) components that have same index in both operands. In other words, in offsets and RVA's the component strings (n-tuples) are aligned, and we can start directly with the first component in each operand and proceed componentwise. With LVA's we must first align.

The theoretical identity of LVA and RVA operations warrants the promised distributivity. With it the truth of (17) and (18) would be established. That alignment can indeed be computed will soon be shown.

Establishing computation rules for add. Only add will be treated, the case of sub being similar. Given are the LVA's R_1 and R_2:

$$R_1 = (C_1, D_1, E_1) \tag{19a}$$

$$R_2 = (C_2, D_2, E_2) \tag{19b}$$

We devise and validate rules by which to compute

$$R_3 = R_1 \text{ add } R_2 \tag{20}$$

$$R_3 = (C_3, D_3, E_3) \quad \text{where} \tag{21}$$

C_3, D_3 and E_3 are to be computed. In use, as we have noted R_1 and R_2 both identify target nodes with respect to intended base nodes. For addition to be meaningfully defined the target of the first operand is required to be the base of the second - it is the intermediate base to be eliminated (3.2 (ii)):

$$R_1 = LVA(B_1, B_0) \tag{22}$$

$$R_2 = LVA(T, B_1) \tag{23}$$

Thus assumed to refer to given base nodes, R_1 and R_2 become associated with two RVA's, which they are taken to represent. We represent these RVA's in turn by certain FRVA's (see (13)), viz. those that arise if their respective D components are chosen respectively as D_1 and D_2 of (19). The FRVA Z components then get determined uniquely. Thus associate

$$\text{with } R_1: \quad Q_1 = FRVA(B_1, B_0) = (Z_1, D_1) \tag{24a}$$

$$\text{with } R_2: \quad Q_2 = FRVA(T, B_1) = (Z_2, D_2) \tag{24b}$$

For the level in the space tree of an arbitrary node N we write L(N). From the definition (14) of C components, applied to (19), we get, in view of (22), (23) and (24):

$$Z_1 = L(B_0) - C_1 \quad Z_2 = L(B_1) - C_2 \tag{25}$$

We now consider alignment. How many component places, "steps", must one of D_1 or D_2 be shifted before a componentwise addition may be performed? Think of right shift and consider this as prefixing zeros. Consider the meaning of Z's in (24) as given by definition under (13). The number of shift steps must be given by $|S|$, where

$$S = Z_1 - Z_2 \tag{26}$$

and which D to shift by the sign of S. Now

$$S = Z_1 - Z_2 = L(B_0) - C_1 - (L(B_1) - C_2) \text{ by (25)}$$

$$S = L(B_0) - L(B_1) + C_2 - C_1$$

So by (15), (24a) we give the rule to compute S

$$S = C_2 - C_1 - E_1$$

Case I: S = 0, i.e. $Z_1 = Z_2$. Neither D_1 nor D_2 need be shifted. Rule for D, Case I: Form D_3 by componentwise addition of D_1 and D_2 (possibly supplementing trailing zeros in one of them, as in 3.2 (ii)).

If the number of zero components understood to be left out in an imagined FRVA: $Q_3 = (Z_3, D_3)$, is taken to be Z_1 (so that $Z_3 = Z_1$) then indeed we have done no more than performed a regular RVA add, treating the $Z_1 = Z_2$ leading zeros in both operands separately from the rest of the components dealt with: As regards the said zeros a count of them is just copied into Q_3, while the remaining components have been added regularly. Calling the RVA's represented by Q_1, Q_2 and Q_3: P_1, P_2 and P_3 respectively, we have achieved:

$$P_3 = P_1 \text{ add } P_2 \tag{27}$$

Case II: $S > 0$, $Z_1 > Z_2$. D_1 applies deeper (farther from the root) in the space tree than D_2. Rule for D, Case II: Prefix S zero components to n-tuple D_1 getting (n+S)tuple D_1^* ("right shift"); then add corresponding components of D_1^* and D_2 (possibly supplementing trailing zeros) to obtain D_3. Consider a transformation of the FRVA Q_1 into $Q_1^* = ((Z_1 - S), D_1^*)$. Q_1^* obviously represents the same RVA as Q_1 - we have just chosen to leave out a smaller number of zeros from the D component, thus representing them explicitly in D_1^*. As under case I we use D_3 to imagine a FRVA: $Q_3 = (Z_3, D_3)$ with $Z_3 = Z_2$, by choice. Q_1^*, Q_2 and Q_3 have the same count of dropped zeros, viz. $Z_1 - S = Z_2 = Z_3$ and for the same reasons as under I we have for the RVA's represented (with similar notation):

$P_3 = P_1^*$ add P_2 But Q_1^* represents the same RVA as Q_1, i.e. $P_1^* = P_1$ so

$P_3 = P_1$ add P_2 (like (27))

Case III: This is like case II with roles of D_1 and D_2 reversed. Rule for D, Case III: Prefix (-S) zeros to D_2 getting D_2^*. Perform componentwise addition (as above) of D_1 and D_2^* to get D_3. Imagine $Q_3 = (Z_3, D_3)$ and choose $Z_3 = Z_1$.

Analogous to case II we get: $P_3 = P_1$ add P_2^* and as $P_2^* = P_2$ because Q_2 and Q_2^* represent the same RVA we get

$P_3 = P_1$ add P_2 (like (27))

Because $P_1 = RVA(B_1, B_0)$ and $P_2 = RVA(T, B_1)$ by their definitions and (24), we can apply (11) to (27), now that (27) is shown to hold in all three cases. We get

$$P_3 = RVA(T, B_0) \qquad (28)$$

In all three cases we also have found a preferred FRVA representation of P_3, viz.

$$Q_3 = (Z_3, D_3) \qquad (29)$$

$$\left.\begin{array}{l}\text{where } Z_3 = Z_1 \text{ in Case I and III} \\ \text{and } Z_3 = Z_2 \text{ in Case II}\end{array}\right\} \qquad (30)$$

Also because Q_3 represents P_3, (28) means that

$$Q_3 = FRVA(T, B_0) = (Z_3, D_3) \qquad (31)$$

Now that we can compute D_3 we turn to how to compute the rest of R_3, viz. C_3 and E_3. We have (31) so we can use the definition of C that helps us to derive an LVA from an FRVA, (14):

$C_3 = L(B_0) - Z_3 =$

$$= \begin{cases} L(B_0) - Z_1 \text{ in Case I and III, by (30)} \\ L(B_0) - Z_2 = L(B_0) - Z_1 + S \text{ in Case II, by (30) and (26).} \end{cases}$$

From this we get the computation rule for C:

$$\left.\begin{array}{l}\text{In Case I and III make } C_3 = C_1 \\ \text{In Case II make } C_3 = C_1 + S\end{array}\right\} \text{ by (14) and (24a).}$$

To compute E_3, use def. (15) on nodes T and B_0, in view of (31), to get

$E_3 = L(T) - L(B_0) = E_2 + L(B_1) - L(B_0)$ using (15), (16), (19b) and (23).

We have by (15), (16), (19a) and (22) that $L(B_1) = E_1 + L(B_0)$ so we state the rule for E:

$E_3 = E_1 + E_2$

From the consideration of Case I thru III it is clear that, following the rules, we will have done no more than add on whatever two RVA's P_1 and P_2 that R_1 and R_2 can represent. This establishes distributivity.

Conclusion. Local Virtual Address, LVA, is the promised unifying data type (1.2 end). An LVA will be a reduced address or a qualifier in the same manner as the RVA it is taken to represent. Thus the assertions of sect. 4, first paragraph, will carry over literally if "RVA" is replaced by "LVA" and all reservations removed. Under Properties of LVA's we have established that the RVA flaws are not inherited. - While reviewing sect. 4, we will make more concrete for LVA's the process of recombination into full addresses, in preparation for engaging them the accessing of the data objects they identify.

Recombination and Access. (Cf. sect. 4) The full qualifier representing the AVA A_B of some base node B is to be an LVA, call it Q, now interpreted as an FVA. (A proposed term for LVA so interpreted would be Global Virtual Address, GVA.) Referring to 3.3, last paragraph, we have: Q = LVA(B, Root) when Root is the root of the space tree. We are to recombine R = LVA(T, B) into a full address. We do

S = Q add R = LVA(B, Root) add LVA(T, B)

S = LVA(T, Root), by (17).

S is a GVA. $S = (C_S, D_S, E_S)$. The base node of S is the root, so (14) becomes:

$C_S = L(Root) - Z_S$. $L(Root) = 0$, however. So $Z_S = (-C_S)$ where $(Z_S, D_S) = FRVA(T, Root)$, (13), and $RVA(T, Root)(= A_T)$ is immediately reconstructed by prefixing the Z_S zeros to D_S (shifting). A_T is an FVA having the components of the Absolute Virtual Address of T (3.3 end) and can thus be used directly for access in virtual storage, N.B. if we know the level of T - as A_T is indefinite with respect to trailing zeros so its number of components does not tell this level. But $E_S = L(T) - L(Root) = L(T) - 0 = L(T)$ so this level is also contained in S.-LVA's and their associated add operation can be used exclusively (but for a final shift), in the process of preparing reference LVA's for accessing.

5.3 LVA normalization

Consider an LVA

R = (C, D, E) = (3, (0,0,2,-1,5), 2).

Obviously R'= (1, (2,-1,5), 2) can represent the same RVA's as R; the choice of 3 for C was arbitrary. The arbitrariness comes from (13), where it was introduced for the sake of generality. R' is shorter, and presupposes only one level to exist above the base node whereas R requires three such levels - an unnecessary commitment. The remedy is simple: We define normalization of an LVA as the process of dropping all leading zeros in the second main component ("D"), while subtracting their number from the first main component ("C"). An LVA with no such leading zeros is said to be normalized. In general, LVA's in use should be kept on normalized form. A further benefit of a normalized LVA is that it is a unique representative of the class of LVA's meaning the same thing - i.e. capable of representing the same set of RVA's. The LVA to any node N with respect to this node itself, LVA(N, N), is not normalizable, nor LVA's to nodes M along the path from the root node to N or along a path of solely zero numbered edges in a subtree whose root is N. In these cases we conventionally pick a standard LVA to be (0, (0), E), E being the level distance as usual (15).

6. Remarks on use of LVA's in systems design

Choice of qualifier. The choice of qualifier is equivalent to the choice of common base node for a set of reference LVA's to objects in a region. This choice can have an important effect on LVA length for the set and thus for storage economy.

For a subtree region the natural choice appears to be the subtree root. Then, a qualifier LVA that identifies the subtree from outside is an LVA to the subtree root, and such LVA's will always only carry components for levels above this root. The reference LVA's to objects in the subtree, presupposing such a qualifier, will have at most so many components as the identified node has levels above itself in the subtree. This appears reasonable.

Multiple interpretation. There is nothing intrinsic in an LVA making it a full qualifier GVA, any particular region's qualifier or a reference LVA with respect to a particular base node. As an LVA it can represent many RVA's (any number of zeros supposed in front of the D) and these RVA's in turn can represent many references, depending on base node supposed. As soon as a definite base node is understood, however, both these aspects of indefiniteness are removed: The interpretation is then fixed. Once implemented, LVA's constitute a general purpose facility offered to all algorithms realized or realizable on the computer, whether these are implemented in hardware or software, whether system software or application programs. The algorithms using it, or the algorithm designer, must keep track of interpretation - just as with any other data type supported, such as binary words.

Protection. There are simple and efficient ways for system algorithms, hardware or supervisory, to confine user algorithms to any desired set of interpretations. The moment to do so is at recombination with a system known full qualifier. E.g. if a range of confinement is to be a certain subtree, then arrange that recombination is to be done with a qualifier Q that is the GVA to the subtree root. If in the recombining add $\neg S < E_Q$, see (26) and below (we must have to do with Case II), then a violation is caught. - Space does not allow further expansion on this theme.

Efficiency. There are evident similarities between LVA add and sub and current, well-understood, widely implemented operations, such as floating point arithmetic or string and decimal number manipulation. Therefore, no doubt efficient hardware implementations can be designed. Of course much attention must be paid to design parameters, such as size and uniformity of field lengths in the D component. - For the underlying virtual space, variability, size and other points of design are of course crucial parameters in hitting tradeoffs between functional capability, flexibility, cost and efficiency. But these are matters independent of the LVA concept.

7. Conclusion

Local Virtual Addresses meet their objectives as stated and summarized in subsection 1.3. - The important problems of how and when names of a System Name Space are allocated and deallocated to/from externally identified data objects may be greatly alleviated by clever application. But the essence of these problems is outside the present work. As I see it, we have here a next major challenge. I would propose to chose for Name Space the set of LVA's. I hope that when these things will be tackled it will appear that the concepts presented will provide a good and profitable basis.

References.

1. J.B. Dennis, Programming Generality, Parallelism and Computer Architecture, Proceedings of IFIP Congress 1968, Vol. 1, North-Holland Publishing Company, 1969, A.J.H. Morrel Ed., pp. 484-492.

2. J.B. Dennis, The Current Challenge to Computer System Architects, Paper given at the International Workshop on Computer Architecture, Grenoble, France, June 26-28, 1973.

3. D.C. Evans and J.Y. Leclerc, Address mapping and the control of access in an interactive computer, AFIPS Conference Proceedings, Vol. 30, 1967, SJCC, pp. 23-30.

4. D. Morris and G.D. Detlefsen, A Virtual Processor for Real Time Operation (Ch. II, esp. p. 20), in Software Engineering, COINS 3, Proc. of the 3:rd Symposium on Computer and Information Sciences, 1969, 1970, pp. 17-28.

5. R.L. Gordon, The Impact of Automated Memory Management on Software Architecture, Computer (IEEE), Vol. 6, No. 11 (Nov. 1973), pp. 31-36.

6. L. Löfgren, D5/30 Reference Manual, Datasaab internal publication, ZND5-70:127.

7. H.W. Lawson, On implementing SILL at the Linköping University, Linköping, Sweden, personal communication.

8. J.B. Dennis, Segmentation and the Design of Multiprogrammed Computer Systems, Journal of the ACM, Vol. 12, No. 4, pp. 589-602, in particular p. 596, 597.

A VIRTUAL MEMORY FOR MICROPROCESSORS

Judith A. Anderson
National Aeronautics and Space Administration
Kennedy Space Center, Florida

G. J. Lipovski
Department of Electrical Engineering
University of Florida
Gainesville, Florida

ABSTRACT

A virtual memory system for microprocessors is described. The system is designed to be extensible, to minimize software and execution overhead and to minimize operating system requirements. Specific application of the virtual memory system with the INTEL 8080 microprocessor is given, describing the necessary software constraints and operating system requirements.

INTRODUCTION

The availability of inexpensive microprocessors has significantly changed the ground-rules for the design of computers that use them. Because input/output I/O devices are far more expensive than microprocessors, it becomes desirable to share them among microprocessors in a micronetwork. Also, because the required memory is now several times more expensive than microprocessors in many cases, it is also desirable to reduce the amount of local primary memory required in each microprocessor. However, a small memory raises the cost of software because overlays are difficult to program and debug. Herein, it is proposed that a small virtual memory system be included in each micro-computer, utilizing a micronetwork to share bulk memory. Structurally, then, the system is similar to a BBN multiprocessor [1]. However, it is shown that a microprocessor such as the Intel 8080 [2], with modified memory hardware, can be economically implemented to have the same desirable effect--to minimize the amount of local primary memory associated with each microprocessor.

A virtual memory for a given microprocessor resides somewhere in a micronetwork. It is partitioned into contiguous equal length pages. Some pages may be on one disc while other pages might be in another microprocessor's memory. The microprocessor itself contains copies of some of these pages, called active pages, in its own primary memory. Generally, the microprocessor will generate a virtual address A for a word in this virtual memory to read or write it. If A is on an active page, the word should be read or written as quickly as possible in the primary memory of the microprocessor. If A is not on an active page, (this is called a page fault) then the entire page that word A is on is brought into the primary memory and thus made active. Then it can be read or written. However, since the primary memory in a microcomputer is fixed, when a new page is brought in, one of the pages in that memory must be "banished" back to the disc or another microprocessor from which it was taken, to make room for the new page.

This paper shows a distributed virtual memory organization which is believed to be novel. The main point is that each page of primary memory, say a 256 x 8 bit page, should have a page comparator and a counter. This provides a cheap and an extensible virtual memory for microcomputers. However, beyond just presenting this system, this paper shows why each part of the hardware is included. It is written in the format of "mathematical proofs", but with the idea of breaking down the design into simple decisions so that the reader can identify those parts of the design which he disagrees with. In this way, we hope that architecture will become more of a science, less of an art. The premises, like axioms, are basic assumptions which we hope the reader will accept without further argument. This is similar to their use by Parhami [3]. Thereafter, each implication, like a theorem, makes one point about the design. However, unlike theorems which can be proven, implications can only be argued. The arguments are the reasons we had for making the implications. If the reader does not agree with the implication, the argument should be able to convince him, or he should be able to show us where we are wrong. This is similar to the use of implications by Szygenda and Hemphill [4]. We hope that this style will not be pretentious or formalistic, but will be suitable to clearly expose each decision that we made in the design of the cellular virtual memory.

This paper is divided into four sections. The hardware construction is considered first. Next, constraints on the software of the microprocessor imposed by this virtual memory are examined. Then implementation with the INTEL 8080 is considered as an example of the architecture. Finally, some general implications on the software operating system are considered.

VIRTUAL MEMORY HARDWARE

When implementing the virtual memory, the question arises as to which functions should be implemented in hardware and which in software. For the determination of this system, several premises are given.

P1: The system should be extensible--adding pages should require a minimum of hardware/software change.
P2: Minimum execution time and minimum storage overhead for the virtual system is desired.
P3: Additional hardware costs cannot exceed the cost of additional primary memory that it replaces.
P4: Multiple word instructions are used.
P5: Operating system requirements should be kept minimal.
P6: As much of the system as possible should be independent of the processor type.

The first question to be considered is how and where the virtual address is to be decoded to select an active page and how and where page faults are detected. Since these operations are done for every memory reference their execution time and software overhead are of particular significance.

I1: Address translation in the primary memory for active pages and page fault detection must be performed with hardware.
Argument: Software address translation results in large execution time overhead as well as storage overhead which violates P2. Hardware implementation requires little or no execution overhead if the page is available in the physical memory and no software overhead.

I2: Address translation and page fault detection hardware should be cellular, where each page of real memory has an associated cell for address translation and page fault detection.
Argument: In order to make the system extensible (P1), it must either be cellular, each cell containing the logic necessary to perform the required address translation for one page, or the hardware required for the maximum number of possible pages must be incorporated in the system. The latter would not be cost effective since it would require more hardware than would be needed except in the extreme case. If the address translation logic is page associated, this logic may be incorporated directly on a memory chip containing the page memory.

This leads to the following construction (figure 1). Each cell will have a register, PAGE. It is loaded by the software to be the virtual page number when the page is loaded. PAGE is compared with the high order bits of the address sent out by the microcomputer. If a match occurs, the memory associated with the cell is enabled so that the word chosen by the low order bits of the address can be read or written. Also, the output of the match from each cell is wire-ORed. If no match occurs, a page fault exists, and the microcomputer must bring in the page. This may be handled by an interrupt in a small microcomputer, since most microprocessors have an interrupt facility (P6). Note that most memory accesses take about as much time in this virtual memory implementation as they do in a real memory implementation.

When a page fault occurs, some page must be banished from real memory and the required page must be brought in. The requirements of the cell to support these operations are now considered. Generally, a page in real memory has to be selected to be removed. However, if it has not been written on (it is called a clean page), the copy in secondary memory is the same as the copy in real memory, and the page need not be written back to secondary memory. But if it has been written on (it is called dirty) then the page must be written back to secondary memory before a new page is brought in. Also, when the machine is first turned on, or when a new program is started, the first memory reference should cause a page fault to begin loading the new program. To prevent accidentally matching a page number in the memory, some mechanism is required.

I3: Determination of whether a page is dirty, or whether the page contains currently valid information, should be done in hardware.
Argument: Implementation of either function requires one bit in each cell. The dirty page bit is set when the page is written on by the program. The valid page bit is set when the page is loaded and is cleared when the machine is started or when a program is completed. Maintaining the dirty page bit with software would require execution of an additional subroutine each time a store instruction is performed, to save the fact that the page was written on, introducing significant execution overhead (P2). The valid page bit simplifies the loading of programs because only the starting address of the program is needed to load it.

The LRU (Least Recently Used) page replacement algorithm has been shown to be an effective method of choosing pages for replacement [5,6] and will be the method to be considered in this implementation of virtual memory. The LRU page replacement algorithm can be implemented in many different ways. Two will be considered here. The first is a total hardware implementation and consists of an associative queue. Each time a page is referenced, the page number is removed from the queue and is placed at the bottom. This causes the LRU page number to rise to the top of the queue. When a page is required, the page number is read from the top of the queue.

The second implementation of the LRU paging algorithm is a combination of hardware and software techniques. A counter is associated with each physical page. Every memory cycle, all the counters are incremented. When a page is accessed, its counter is reset. Overflow of the counters will be inhibited. When a page is required, the contents of the counter associated with each physical page is read and the maximum found. The maximum counter value corresponds to the LRU page.

I4: The counter implementation of the LRU page replacement algorithm is better than the associative queue.
Argument: The associative queue is not easily extensible. The counter implementation is extensible, the counter being associated with each page (P1). It provides flexibility--pages can be kept in primary memory by specifying which pages have their counters examined when a page fault occurs. The execution overhead, while larger than that of the queue implementation, is not significant since the operation is performed only when a new page is required.

An implementation of a cell incorporating the mechanisms described above consists of a status register with a "dirty bit" and a "valid page" bit and the counter for the LRU algorithm in addition to the PAGE register and comparator described earlier. The output of the comparator is tied into an OR-rail for determining a page fault and is used to enable the memory cycle if the page is being accessed. The PAGE register, the counter, the dirty indicator and the memory data lines are tied through a selector switch to the I/O bus. The page is treated as an I/O device so that they can be accessed by the software when a page fault interrupt is serviced.

The instruction that was being executed during a page fault must be re-executed using correct data. This requires saving the address of the instruction and also the page number of the missing page, as is now discussed.

When a page fault occurs, the memory data being accessed is not available and all zeroes is returned. We opt to use the virtual memory system with most microprocessors (P6). Most microprocessors currently available cannot respond to an interrupt until the instruction being executed is complete. This causes either the wrong instruction to be executed or an instruction executed with the wrong data. Correcting for the invalid instruction or data leads to the following implications.

I5: Either the instruction length or the address of the first byte of an instruction must be saved when a page fault occurs.
Argument: When a page fault occurs during an operand fetch, the instruction must be re-executed after the new page is available with the correct operand. Note that the instruction may be a multiple word (P4). In order to "backup" the program counter, it must be either decremented by the length of the instruction or reset to the address of the first byte of the instruction. The length of the instruction varies, but can be determined from its op code during the fetch cycle.

I6: The address of the first byte of the instruction should be saved when a page fault occurs, rather than determining the instruction length.
Argument: The address of the first byte of each instruction can be saved in a register. When a page fault occurs, loading of the register can then be disabled, saving the contents until the page fault is processed. To determine the instruction length requires decoding the op code of the instruction. This is processor dependent, violating P6, and could be very complicated depending upon the instruction set.

The virtual page number being accessed during a page fault must be determined. Two ways of doing this are considered. The first is to trap the page number being broadcast when the fault occurs. The second involves determining first whether the fault occurred during an instruction fetch or an operand fetch. If it was during an instruction fetch, the program counter contains the page number. If not, the instruction must be decoded to determine which register has the page number being accessed.

I7: Saving the page number being broadcast is better than decoding the instruction.
Argument: Saving the page number is easy to implement, requiring only a register which is loaded when enabled by the page fault signal. It would require a minimum of software--one I/O command to read the register. Implementation would not be processor dependent (P6). Decoding of the instruction would require either a software routine to examine the instruction which would be storage consuming (P2) or combinational circuitry to determine which register contained the page number. This instruction decoding would vary from processor to processor, violating P6, and can be very complex, as in the case of the INTEL 8080 [2].
The mechanisms discussed in I5 through I7 are shown in Figure 2. Note that this hardware is not needed in each cell associated with a page of real memory. Rather, one copy of it is required for each microprocessor.

PROGRAMMING CONSTRAINTS

When a page fault occurs, the effects of the instruction being executed at the time of the page fault must be correctable. This requirement leads to the following implications.

I8: Multiple word instructions cannot cross page boundaries.
Argument: Multiple word instructions are of two general classes: (a) the additional words are used as immediate data, and (b) the additional words are used as an address. If a page fault occurs during the fetch of the second or third word of the first type of instruction, the operation can result in invalid data in one of the registers with no way of recovering the original data. With the second type of instruction, a page fault during the fetch of the second or third byte will cause an invalid address to be used, resulting in the wrong memory location being referenced and possibly having data stored in the wrong location or transferring to the wrong location. The effects of page faults occurring during the fetching of multiple byte instructions for the Intel 8080 are listed in Table I. Since these two classes of instructions represent a large portion of the total capability of the microprocessor (38 of the Intel 8080 instruction set--see Table I), they cannot be prohibited from use without seriously degrading system capabilities. A simple, easy-to-implement alternative is to avoid overlapping page boundaries with an instruction.

I9: Any instruction which cannot be "backed out of" after a page fault must be prohibited.
Argument: If a page fault can occur during execution of an instruction, and the result of that instruction is such that data is lost and cannot be retrieved (such as the contents of the accumulator), that instruction must not be used. This will, in general, be restricted to the direct memory arithmetic and logical instructions. The effect of instructions causing a page fault on the Intel 8080 are shown in Table II. Only three instructions must be prohibited.

I10: The interrupt must be enabled at all times while operating in virtual areas.
Argument: If the interrupt is disabled and a page fault occurs, the program will try to continue with invalid results. If the page fault occurs on an instruction fetch (that is, if the instruction were on a new page), the CPU will continue to execute null instructions, incrementing the program counter until it reaches a page that is available. Either case appears to be disastrous.

IMPLEMENTATION WITH INTEL 8080

The software constraints discussed above are easily enforced--a compiler can be used to automatically insert NOP instructions if a page boundary is being crossed. Relocatable segments can all be started on page boundaries. Prohibited instructions may be avoided by substituting a macro.
The operating system can be given control over the interrupt with the user only allowed to inhibit (mask) interrupts other than the page fault.

Non-stack instructions were examined. Those which could cause page faults are listed in Table II with their effect and remedial action. The only action required was to re-execute the instruction, which minimizes software requirements.

When an interrupt occurs in an INTEL 8080, at the completion of the execution of the current instruction, an instruction is accepted from the data lines without affecting the program counter. In order to save the contents of the program counter and branch to an interrupt processing routine, the RST instructin must be used. This causes the program counter to be pushed onto the stack. If the stack pointer is at a page boundary at the time, a page fault will occur. This will cause another RST instruction to be executed, placing the system in an endless loop. Several methods of preventing this situation were considered. All involved insuring a page fault could not occur during a stack operation.

(a) The stack may be constrained to one page which is kept core-resident.
(b) All pages allocated to the stack may be kept core-resident.
(c) The current page and the next nearest page (relative to the stack pointer) may be kept in core.

I11: Keeping the current and the next nearest page of the stack in core is better than restricting the stack to one page or keeping all stack pages in core.
Argument: Method (a) places a tight, almost impossible to enforce, constraint on the programmer and the operating system. Method (b) requires a large amount of dedicated pages. Method (c) allows flexibility in programming, placing no constraints on the programmer. It requires only two pages dedicated to the stack. Detection of the need to bring in a new stack page is easily implemented with a simple four state sequential machine (Figures 3 and 4). The only software requirements are that when the stack pointer is changed by other than a PUSH or POP instruction, the stack associated logic must be initialized and the appropriate pages loaded in addition to a routine to process the stack page fault interrupt.

OPERATING SYSTEM REQUIREMENTS

A deterrant to using a virtual memory in a microcomputer may well be that the operating system requirements to support it consume a large amount of real memory. This problem is currently under investigation. However, we conjecture that the real memory requirements for the resident operating system are quite modest.

Consider a likely application of this type of virtual memory in a micronetwork. A terminal will consist of one microcomputer with its primary memory, a keyboard, and a display. A controller will consist of one microcomputer with its primary memory and a fast I/O device such as a disc. A manager consists of one microcomputer and its primary memory. A micronetwork would then consist of several terminals, several controllers and one manager connected by a buss. (For

reliability, it should be possible to make any unused controller or terminal into the manager.) We believe that those aspects of the operating system that service interrupts and errors for an I/O device can be resident in the controller while the service routines for the keyboard and display in the terminal can be paged in, and need not be resident. It also appears that spooling and scheduling should be handled by the manager. We now argue that mapping virtual page numbers to addresses in I/O devices in controllers, should also be handled by the manager.

I12: The manager should map virtual page numbers into addresses in I/O devices.
Argument: A given terminal will execute a program in a virtual memory. However, the storage location for the pages of this virtual memory will be stored in various controllers to share the resources of the system. A table will be stored, for each terminal, which associates each page number with routing information to locate the controllers and with the physical address of the page in the controller. If all these tables are in one manager, then storage allocation, scheduling, communication through sharing pages, and garbage collection are easier than if each terminal stores its own table. Such tables can be made small and can be accessed quickly enough if, for example, they are stored as binary trees where each node describes a collection of contiguously numbered pages that are stored in contituous locations in a controller. Many tables will consist of only one node. Thus, the storage requirement should be modest. Moreover, since these tables are read only when the terminal has a page fault, and the page fault will require moving perhaps one or two pages, the overhead time in searching this table, and waiting to search it if the manager is busy, should be quite tolerable.

Therefore we imply that the resident operating system in the terminal will be quite small.

I13: The resident operating system in a terminal will have an initial program load routine, a page fault, and stack page fault program that will only determine which virtual pages need to be banished or brought it; and a program for communicating on the buss. Finally, these programs should be in read-only memory.
Argument: We believe that all the remaining operating system functions can be best handled in the controllers or the manager, or can be paged in on demand. However, a program for initializing the registers is required to be resident and in a read-only memory, and a program for handling page faults and buss communication must be available for use at all times.

Flow charts for the page fault handlers are shown in figures 5 and 6. These programs, a general interrupt handler and the buss communication program appear to be modest in size. We believe they will be the only resident part of the operating systems.

CONCLUSION

A novel cellular virtual memory has been described. A hardware comparator is used with each page of memory to assist the virtual memory functions of obtaining the word from fast memory, if it is there, or determining if it is not there to generate an interrupt. A counter is used with each page to determine which page is to be banished when a new page is brought in. Hardware is used to determine which page is missing, and which instruction failed. Software is used to find the new page to be brought in, and to retry the instruction that had the page fault. A widely available microcomputer, the INTEL 8080, was studied to examine what is required in a practical system that would use such a virtual memory.

Moreover, in this paper, a new style of presentation is used to give the reasons behind each design decision. Herein, the reader can determine where he disagrees with our design. Also, extensions to this concept, for example to defining a microprocessor better suited to using virtual memory, can be more easily spelled out. We hope that this style, as well as the content of this paper will be a useful contribution to the science of computer architecture.

REFERENCES

1. Ornstein, S. M. et al., "The BBN Multiprocessor".
2. INTEL, 8080 Single Chip Eight-Bit Parallel Central Processor Unit, April, 1974
3. Parhami, Behrooz, "A Highly Parallel Computing System for Information Retrieval," FJCC, 1972, pp. 681-690.
4. Hemphill, John M., and S. A. Szygenda, "Deriving Design Guidelines for Diagnosable Computer Systems," Proceedings of the First Annual Symposium on Computer Architecture, December 1973, pp. 131-135.
5. Coffman, E. G. and L. C. Varian, "Further Experimental Data on Behavior of Programs in a Paging Environment," CACM, July, 1968, p. 471.
6. Madnick, S. E. and John J. Donovan, Operating Systems, McGraw-Hill, 1974.

INSTRUCTION	EFFECT DURING PAGE FAULT	ACTION TO BE TAKEN
ACI SBI	(A)← invalid result	No recovery
ANI	(A)←0	No recovery
ADI SUI XRI ORI	(A)← unchanged	Re-execute instruction
CPI	Condition FF set invalid	Re-execute instruction
LXI B LXI D LXI H	(r)← invalid data	Re-execute instruction
LXI SP	(SP) invalid address	No recovery
IN	Input requested from wrong I/O device	
OUT	Output sent to wrong I/O device	
JMP JC JNC JZ JNZ JP JM JPE JPO	(PC)← invalid address	
CALL CC CNC CZ CNZ CP CM CPE CPO	(PC) ← invalid address	
STA	(M)←(A) where M is not known	
SHLD	(M)←(L), (M+1) H where M is not known	
LDA	(A)←invalid data	Re-execute Instruction
LHLD	(L),(H)←invalid data	Re-execute instruction

Table I. Effects of page faults during the fetch of Intel 8080 multiple byte instructions

INSTRUCTION	EFFECT DURING PAGE FAULT	ACTION TO BE TAKEN
MOV r,M	(r)←0	Re-execute Instruction
MOV M,r	No store	
*MVI M	No store	
**STA	No store	
STAX B	No store	
STAX D	No store	
**LDA	(A)←0	
LDAX B	(A)←0	
LDAX D	(A)←0	
**SHLD	No store	
**LHLD	(L),(H)←0	
INR M	No store, condition FF set	
DCR M	No store, condition FF set	
ADD M	(A) not changed	
SUB M	(A) not changed	
XRA M	(A) not changed	
ORA M	(A) not changed	
CMP M	Condition FF set invalid	
ADC M	(A)←(A) + carry	(A) cannot be reconstructed.
SBB M	(A)←(A) - borrow	
ANA M	(A)←0	

*2 byte instruction **3 byte instruction

Table II. Instructions causing page faults

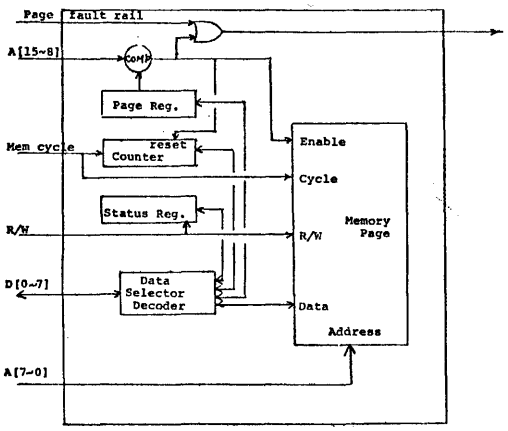


Figure 1. Physical page for virtual memory

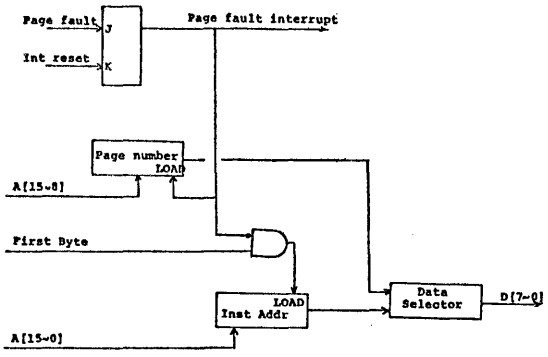
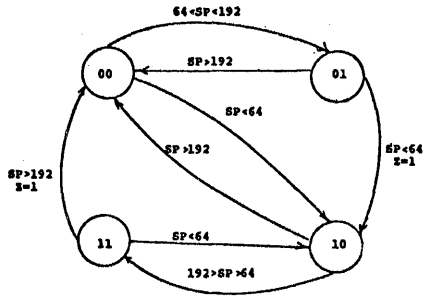


Figure 2. Logic to trap page number and instruction address at page fault



STATES S_1S_2	
0 0	SP points to top quarter of page
0 1	SP points to middle half of page and next higher page is available
1 0	SP points to bottom quarter of page
1 1	SP points to middle half of page and next lower page is available

Figure 3. Sequential machine to generate stack page fault

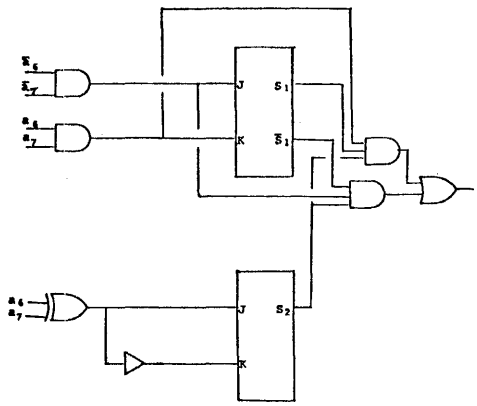


Figure 4. Realization of stack page fault generator

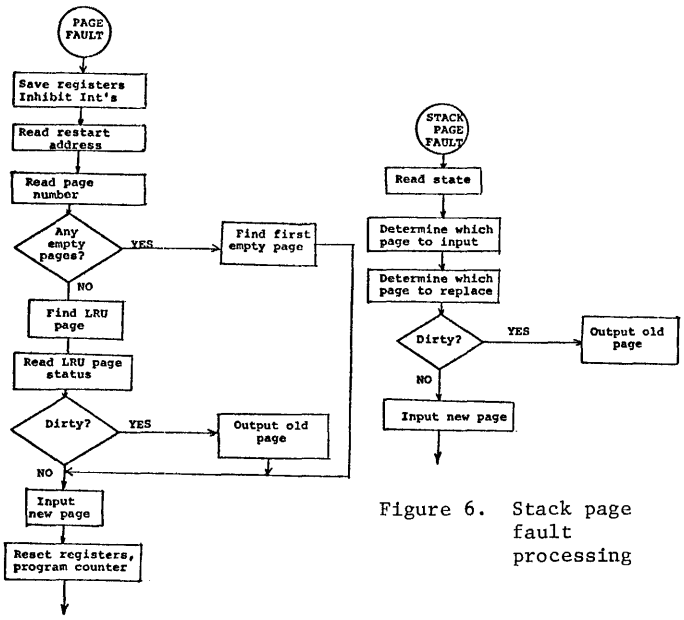


Figure 5. Page fault processing

Figure 6. Stack page fault processing

THE PERFORMANCE ENHANCEMENT OF DESCRIPTOR-BASED VIRTUAL MEMORY SYSTEMS THROUGH THE USE OF ASSOCIATIVE REGISTERS†

R. E. Brundage and A. P. Batson††
Department of Applied Mathematics and Computer Science
University of Virginia, Charlottesville, Virginia 22901

Summary

Contemporary paged virtual memory systems often use associative registers to reduce access time to frequently-referenced pages. Here we examine the analogous use of associative registers in descriptor-based, symbolically-segmented virtual memory systems, where each segment contains an entire data structure as defined in a high-level language. Symbolic trace data from production Algol 60 programs were used to determine performance improvement as a function of the number of associative registers in the system. Our results indicate that, even for reasonably large programs, a hit ratio of 0.9 is achieved with only 4 associative registers. Increasing the number of associative registers to 8 gives a hit ratio of 0.98, which with current technology gives a performance improvement of at least 80% in addressing speed over a similar system without associative registers.

We examine here the use of associative registers to reduce the overhead encountered in the addressing mechanisms of segmented virtual memory systems.[1] This addressing overhead is caused by the need for an additional memory reference for each access to an element of a symbolic information segment. We have studied the performance improvement, in terms of decreased virtual memory access time, which is achieved by using associative registers to hold the most recently-used segment descriptors. A novel feature of this performance analysis is the use of symbolic data-segment reference strings obtained from Algol 60 programs executing on a modified Burroughs B5500.

Symbolically-Addressed Virtual Memory Systems

Virtual memory[1] is the term coined for those computer memory subsystems in which programmer-defined symbols are at least one address-translation level removed from addresses in executable real memory. Programmars typically define information units and structures, such as simple variables and arrays, in a symbolic name space in which each information unit or structure receives a (contextually) unique name. The set of symbolic names is generally an unordered collection of symbol strings of some finite length, having a specified structure (e.g. Algol identifiers[2]). Symbolic names (identifiers) are objects in the so-called name space, which provides a means of naming and referencing information in the virtual address space. The virtual address space may take on a variety of forms, but here we restrict our attention to segmented address spaces, in which information structures, called segments, may be variable in size up to an implementation-imposed limit. In the present study, we further restrict our attention to non-partitioned information segments, i.e., each segment occupies a contiguous block of storage. In the virtual address space, each information segment is referenced via a special information unit called a descriptor, which is associated with the symbolic name (identifier) in the name space. The descriptor contains the information necessary to convert a symbolic address (segment name, displacement) into a physical address. Mechanisms for the implementation of this conversion are well-known and are described, for example, in Denning's survey article.[1] These implementations involve two separate references to main memory for each conversion of a symbolic address. The analogous overhead problem in paged virtual memory systems is frequently alleviated through the use of associative registers. The purpose of this work is to investigate the degree to which this addressing overhead in symbolically segmented systems can be reduced through the use of a small set of associative registers.

Generation and Collection of Symbolic Reference Strings

The symbolic reference strings used in this study were obtained from a collection of event-trace tapes generated by instrumented Algol programs serially executed on a modified Burroughs B5500. Algol programs were collected from the users of the University of Virginia B5500 and other sources and these were used to generate a data base for the study of program behavior. Included in the events collected are block (procedure) entry and exit, flow of control (global label branches), occurrence of I/O statements, processing of declarations by run-time procedures and other events necessary to reproduce the execution sequence. The event-types are primarily oriented toward aspects of program behavior related to resource allocation and utilization. One series of monitored runs also included the collection of references to symbolic data-segments (arrays) and a subset of this data was used in this study.

The method of obtaining symbolic references was relatively straightforward; in a descriptor-based machine such as the Burroughs B5500, it is possible to use the so-called presence bit in a manner somewhat different from that envisaged by the hardware designers.[3] To trap all references to data segments the presence bit, which normally indicates whether or not the segment is present in main memory, was simply turned off and stored elsewhere. Thus, at each reference to a data-segment, a non-present interrupt occurred, and the descriptors and associated information were passed to the event-trace monitor procedure and recorded on magnetic tape. Normal presence-bit handling was then performed by software to allow program execution to continue.

To process the data-segment reference strings, it was necessary to convert the references into symbolic form, using an event-trace processing program (currently implemented on a CDC 6400). The event-trac processing program simulates the execution of a trace run by emulating the run-time data structures on the Burroughs B5500. In order to relate the events to the symbolic level of the source program, the Algol compiler was modified to emit, along with the instrumente object code, a file containing the block structure and declarations (procedure, variable and label) occurring in the source text. This file is converted into a data structure analogous to a symbol table, but which is in

†This research was supported by NSF Grant GJ-1005.

††Part of this work was performed while this author was on leave at the Institut de Programmation, Université de Paris VI, France.

actuality an inverse symbol table as it is used to convert addresses into symbolic names. The symbolic reference strings were written on magnetic tape for subsequent processing by the associative memory analysis program.

Performance Analysis

The use of associative memory in an addressing mechanism for a segmented virtual memory system is illustrated in figure 1. The symbolic address pair is (S,ω), and in a system without associative memory the segment name, S, is used directly as an index into the segment name table, where the descriptor is stored. In the modified system the segment name is first presented to the associative memory (AM) register system, which responds with a match/no match condition. If a match occurs, then the real address can be computed using the output of the associative memory (base address of the symbolic segment) and the index of the symbolic address. Otherwise, the segment name table (in main memory) is referenced to obtain the base address. The advantage here is, of course, that for current technology, the associative register look-up is several times faster than the reference to main memory, so that the time to reference virtual memory will appear speeded up in direct proportion to the number of AM matches and the AM search time. This idea is hardly new, but it has only recently been incorporated into a symbolically addressed, variable-length, segmented system, the Burroughs B7700.

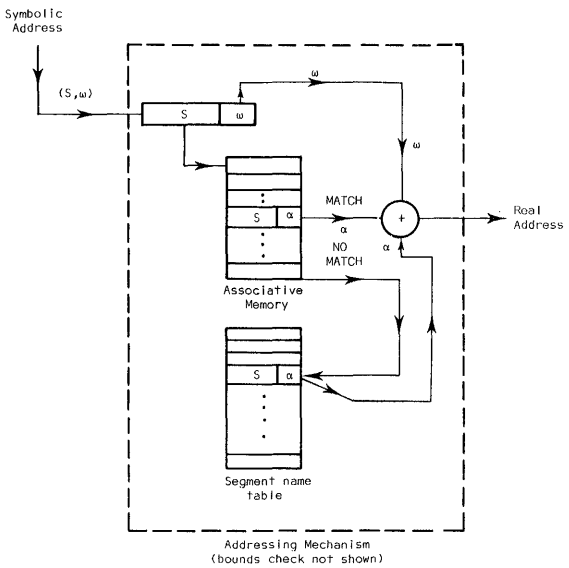


Figure 1 Operation of Associative Memory

The choice of the algorithm to manage entries maintained in the AM must be made in the light of several considerations, of which the primary one is that the AM is of a fixed (maximum) size. This, plus the fact that AM is a hardware-allocated, single-user resource, constrains the choices of replacement algorithms. The most likely choice, which is usually associated with paged virtual memory systems,[4,5] is the least-recently-used (LRU) replacement algorithm. This algorithm has been shown to be quite satisfactory in practice, and it has the added benefit of belonging to the class of so-called "stack algorithms" analyzed by Mattson et al.,[5] for which the analysis of storage hierarchy management is quite straightforward. In this work we have, therefore, assumed an LRU replacement policy for associative memory, and we now proceed to define measures for the performance of the virtual memory system. This development is based on the work of Mattson, et al.[8]

We first define the symbolic reference string. Given a set $N = \{n_1, n_2 \ldots n_N\}$ of symbolic segment names, a symbolic reference string is a sequence of references to segment names:

$$r_1, r_2, r_3 \ldots r_k, \ldots .$$

where r_k is the name of the segment referenced at the kth instant in virtual time.

A stack (of size N) is an ordered vector consisting of the set N arranged in order of the references to the elements of N,

$$S_t = (s_t(1), s_t(2), \ldots s_t(N)),$$

where $s_t(1)$ is the name referenced at time t - 1, and where S_t is updated as follows:

$$S_{t+1} = \begin{cases} (s_t(1), s_t(2), \ldots s_t(N)) \text{ if } r_t = s_t(1) \\ (s_t(j), s_t(1), s_t(2), \ldots s_t(j-1), \\ s_t(j+1), \ldots s_t(n)) \\ \text{if } r_t = s_t(j);\ j \neq 1. \end{cases}$$

Given an initial stack and a reference string, we can derive the stack distance string as follows:

at each time instant k, given S_{k-1} and r_k, then:

$$d_k = i, \text{ where } r_k = s_{k-1}(i)$$

Thus, for a given reference, r_k, the corresponding stack distance d_k is the distance "down" the LRU name stack at which the symbolic name is to be found. Thus, if the m top elements of this stack are held in associative memory, as will be the case for LRU management of an associative memory of capacity m, then a match in this memory will be found when $d_k \leq m$.

Following Mattson et al., we now present an expression for the success function, F(m), which is the proportion of references which will be matched in the top-of-stack memory of size m. Given a reference string of length L, the string is processed by maintaining the stack and a vector of stack distance counters $\sigma(d)$ (initially zero), where $\sigma(d)$ is incremented by one for each occurrence of stack distance d. The success function is then given by:

$$F(m) = \sum_{d=1}^{m} \sigma(d)/L$$

F(m) is often called the "hit ratio," and in the next section we present some values for this function for a collection of Algol programs.

If we assume that main memory has a read time of T_M units and that the associative memory has a search time of T_A units ($T_A < T_M$), then we can derive a measure of the improvement in performance obtained

with an m-element associative memory. Suppose that we are processing J different programs, then we can determine the success function $F_j(m)$ for the individual reference strings, where the jth string contains L_j symbolic references. Then, a(m), the overall fraction of references having an associative memory match (the hit-ratio) for an associative memory of size m is given by:

$$a(m) = \sum_{j=1}^{J} F_j(m)L_j/R, \text{ where } R = \sum_{j=1}^{J} L_j \quad .$$

We point out that a(m) is a value which is weighted by the different L_j's (running times) of the programs. The hit-ratio achieved in practice would be lower than that given by the expression above if the programs ran in a multiprogrammed manner, since the associative registers would be cleared at the time of program switching.

Using the value of a(m), we can easily derive the average segment reference time g(m):

$$g(m) = a(m)(T_A + T_M) + (1 - a(m))(T_A + 2T_M) \quad .$$

Here we have assumed that the reference time is $T_A + 2T_M$ when the descriptor is not in associative memory. This will not be true for all systems, because of factors such as interleaving and the overlap between reading and restoration times.

Finally, we define e(m) as the ratio of g(m) to the minimum possible reference time $(T_A + T_M)$, i.e.

$$e(m) = \frac{g(m)}{T_A + T_M} = a(m) + (1 - a(m)) \frac{(T_A + 2T_M)}{(T_A + T_M)}$$

This quantity e(m) is helpful for visualizing the way in which accessing efficiency varies with the size of the associative memory, since it is almost independent of the ratio T_A/T_M when this is much less than one. When T_A/T_M is zero, e(m) = 2 - a(m).

Results

The experimental data used in this study were 21 symbolic reference strings, obtained from 21 Algol 60 programs written in Algol 60 for the Burroughs B5500 computer. The only limitation of that version of Algol 60 of significance here is that any given dimension of an array may not exceed 1023 words in length, though the size of a symbolic data segment may greatly exceed that, of course, if it is presented as a multi-dimensional array. In this study, we consider only references to array elements - one can envisage that all simple variables will be stored in a separate, unique data segment. The 21 programs could be classified as small-to-medium in size and complexity, having from 3 to 28 symbolic data segments (arrays). Their total memory requirements on the B5500 (exclusive of system routines such as I/O modules) ranged from around 3.000 words to 35,000 words, with from 6 to 77 code segments (blocks). The reference strings themselves ranged in length from 470 to over 175,000 references to symbolic data segments. All of these programs were taken from the user group at the University of Virginia or from the program library. The applications themselves were in general of a scientific nature, including programs for differential equation solution, factor analysis, regression analysis, chemical engineering applications, pattern recognition, and the reduction of environmental data.

Since the use of associative registers as a means for improving system performance is so closely allied to the characteristics of program structure and execution behavior, we first discuss a measure which indicates the dependence of the size of the associative memory on program size (number of symbolic data segments) for a given performance. We postulate here that the contents of the AM will be almost entirely data segment descriptors, since code and data segments are logically and physically distinct, and control resides for a significant time in a single code segment (which can be referenced via normal registers). Furthermore, the segment of simple variables can have a fixed base for the duration of a burst of processor activity on a program, and hence its descriptor can be stored in a conventional register. Thus, we define a ratio p, given by

p = (number of AM registers/(number of data segments)

and show, in Figure 2.1, how the success function, or hit ratio, varies with p for all 21 programs.

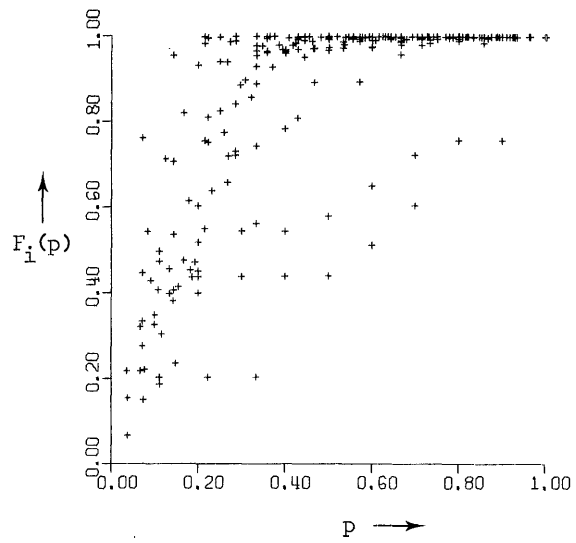


Figure 2.1 $F_i(p)$ versus p (all programs)

A high hit ratio with a small p value represents a program with a high degree of relative locality of reference, and it is clearly the case that some of the programs were more meritorious than others in this respect. It was postulated that somewhat better relative locality would be found in the larger programs, since most mortals cannot manipulate too many things at the same time. This supposition was confirmed by plotting the same data for programs with less than 11 data segments (Figure 2.2) and 11 or more segments (Figure 2.3). The data from the larger programs indicate that very high hit ratios are achieved when p is greater than about 0.4.

From the practical point of view, in terms of machine design, the more interesting question is concerned with the number of associative registers necessary for high performance on real program mixes. The shape of Figure 2.3, which in fact represents the range of relative locality for 85% of all programs measured, gives some hope that a given, fixed, AM size will produce acceptable performance improvement for all programs. This issue is explored in Figures 3, 4, and 5.

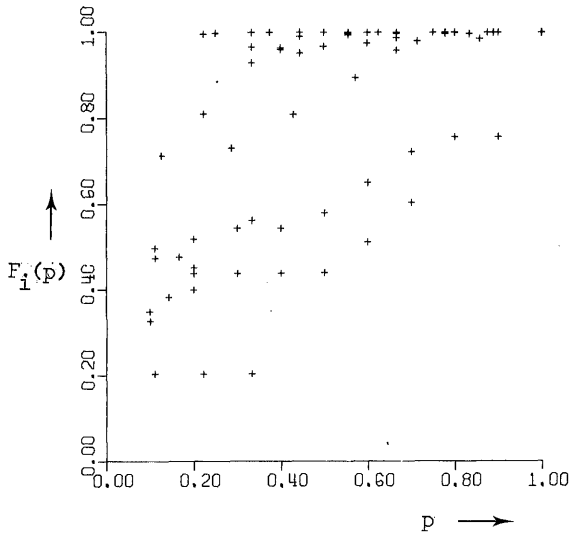


Figure 2.2 $F_i(p)$ versus p (programs with less than 11 segments)

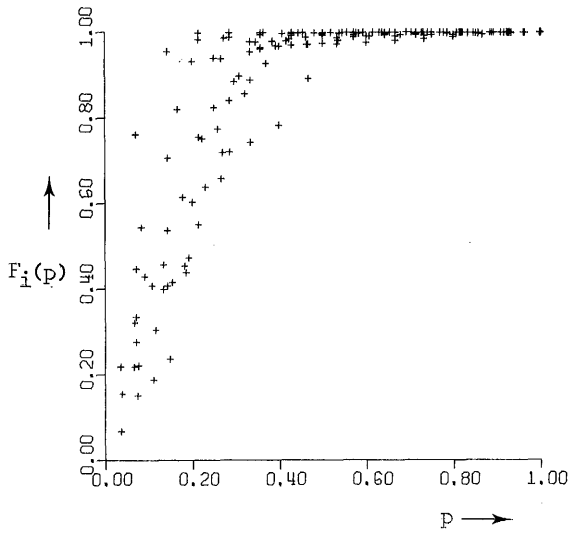


Figure 2.3 $F_i(p)$ versus p (programs with more than 10 segments)

Figure 3 displays the variation of the overall hit ratio, a(m) with m, the size of the associative memory. Two sets of points are given - one for all 21 programs, and a second set which does not contain data from one program (NONL) which was much larger then any other, making up about 30% of the sample. The data show that an AM size of 4 yields a hit ratio of greater than 0.9 for this sample. By doubling the size of the AM to 8 the hit ratio increases to 0.98, i.e. one can recover 80% of the overhead remaining at size 4. For larger programs than those measured here it could well be the case that the hit ratio would fall off somewhat, but the data presented in Figure 2.3 gives one real hope that performance would not fall to an unacceptable level with only 8 associative registers. For example, taking the most pessimistic viewpoint that relative locality is independent of

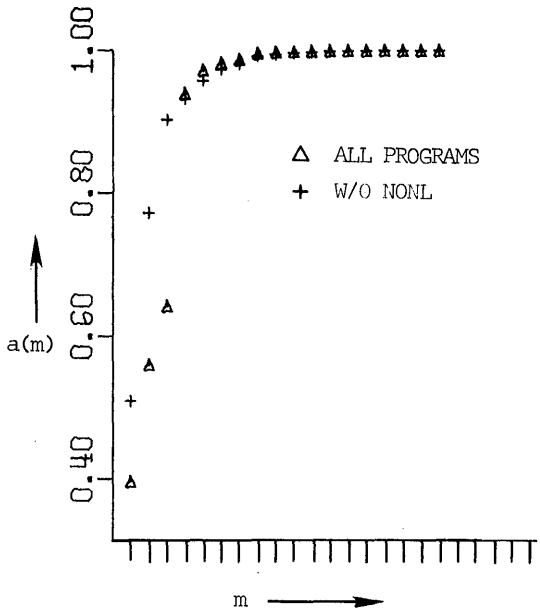


Figure 3 Hit Ratio as a function of Number of Registers

program size, and thus that Figure 2.3 is valid for very large programs, then an 80-data-segment program (which is a very large program indeed) would still achieve a hit ratio of around 0.4 with 8 associative registers. In practice, one would suspect that performance would be significantly better than this for the reasons described above.

To illustrate the increase in virtual memory performance as a function of AM we present in Figure 4 the variation of average reference time, g(m), with AM size. Here we have assumed a main memory reference time, T_M, of 1.0, and plotted three curves, for AM search times T_A of 0.01, 0.1, and 0.25. Finally, the data is shown in a somewhat more condensed form in Figure 5, where e(m) is plotted against m. The points shown are for $T_M = 1.0$, $T_A = 0.1$, but as was mentioned earlier, the graph is almost identical to those for other AM speeds if these are significantly less than main memory speed. Here e(m) is the ratio of actual reference time to the minimum possible reference time, and thus the graph gives a simple illustration of how performance changes with the size of the associative memory.

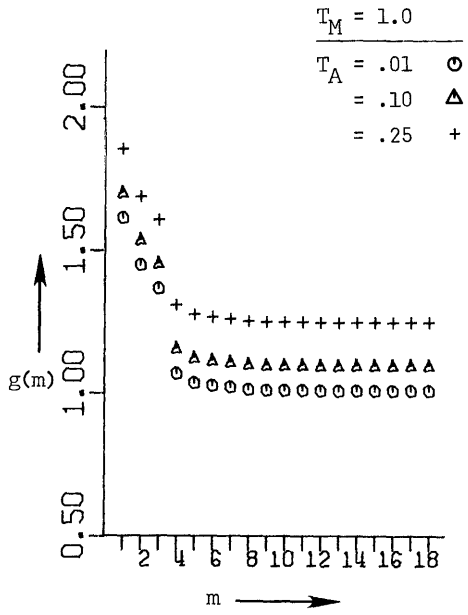


Figure 4 g(m) as a function of m

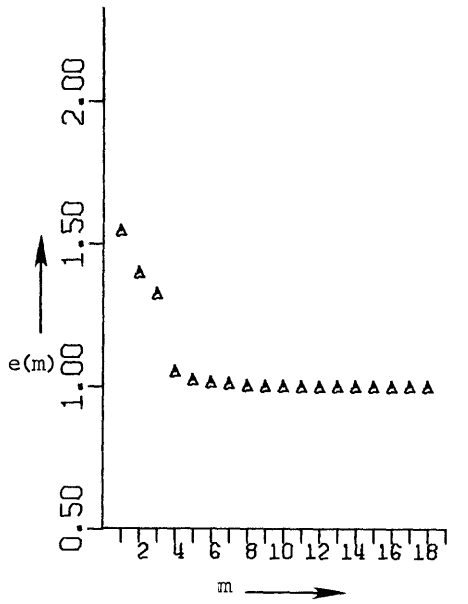


Figure 5 e(m) as a function of m

Discussion

The results presented here are essentially global measures of certain program behavioral characteristics, and we have shown how they can be utilized to evaluate a specific hardware design for a segmented virtual memory system. One important question concerns the extent to which this sample of programs is representative of the mix of real world computing problems. Clearly, an increase in the size of the sample would be an improvement, and an obvious bias is the lack of any really large problems. Although the current results seem to indicate that programs do not have very populous symbolic localities, whatever their size, it is clearly the case that this should be studied experimentally. In addition, studies of localized behavioral patterns, such as the way in which localities change with time, and the relationship of these localities to the lexicographic structure of the program, would be both informative and useful. We are in the process of investigating such behavior. It is difficult to compare these results with those found by Schroeder for the Multics system,[6] in that the two concepts of "segment" are quite different. Moreover, the GE-645 is a paged virtual memory system with a page size of 1024 words, and thus conceptually this is the minimum 'segment' size which can be referenced by a descriptor. Linguistically-distinct data structures, such as all arrays local to a procedure, may well be packed into a single such segment. Conversely, from our viewpoint, our typical program has a relatively large number of small, distinct segments.[7,8]

Another possible direction of study would be to consider the effects of different mechanisms for representing arrays in memory. On the Burroughs 5500-6700 machines, for example, multi-dimensional arrays are implemented in a tree-structured form. The elements of the array are contained in the leaves of the tree, and each array-row segment is allocated as a contiguous block of storage. Thus, to address an array element one must trace a path through the intermediate levels via dope-vector segments containing descriptors to the next lower level. Hence, an additional n - 1 memory references are incurred in referencing an element of an n-dimensional array on such systems. This would cause our analysis to be valid only as a lower bound on the average reference time to a symbolic segment, since these lower-level descriptors would obviously displace segment name table (root level) descriptors already in associative memory, causing a higher replacement rate and a corresponding degradation in average AM hit ratio.

The effects of environment change (block exit) have not been considered in this study. Since Algol 60 array segments are deallocated on exit from the block at which they were declared, then their descriptors become invalid at this time and may be delected from AM. Thus, an AM cell containing such a descriptor could be freed at block exit time, but this fact is not reflected in the analysis of the LRU algorithm by the stack processing technique. The effect of this omission is not likely to be great, since block exit will usually be associated with a change in the set of active data segments leading to prompt changes in the contents of associative memory.

Finally, we will remark that the use of symbolic trace data, as illustrated by the work reported here, is clearly important for the study of designs for high-level language machines. Although these results were obtained with Algol 60 programs, there is no reason to expect significantly different findings for programs written in other block-structured languages, such as PL/I and Pascal. Whilst considerable attention has been given to paged virtual memory systems, through the use of address traces and the like, it is evident that such data have little relationship to the original high-level program as written.

Hatfield and Gerald,[9] for example, have given some striking evidence of the behavioral changes in program execution on a paged machine which are obtained by varying the mapping between a program's symbolic name space and the linear virtual address space of such systems. It seems reasonably plausible to expect that the behavior of real programs, say in Algol, can only be discussed in terms of Algol-level behavioral data, such as symbolic traces, and equally that such data can be usefully employed for the design evaluation of high-level language machines.

References

1. Denning, P. J. "Virtual Memory," Computing Surveys, 2(3), September 1970, 153-89.

2. Naur, P. et al. "Revised Report on the Algorithmic Language Algol 60," Comm. ACM, 6(1), January 1963, 1-17.

3. Burroughs Corporation, Burroughs B5500 Information Processing Systems Reference Manual, Form No. 102326, September 1968.

4. Belady, L. A. "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM Systems J., 4, 1966, 78-101.

5. Mattson, R. L., et al. "Evaluation Techniques for Storage Hierarchies," IBM Systems J., 9, 1970, 78-117.

6. Schroeder, M. D. "Performance of the GE-645 Associative Memory While Multics is in Operation," ACM SIGOPS Workshop on System Performance Evaluation, Harvard University, April 1971, 227-45.

7. Batson, A., et al. "Measurements of Segment Size," Comm. ACM, 13(3), March 1970, 155-59.

8. Batson, A. P. and Brundage, R. E. "Measurements of the Virtual Memory Demands of Algol-60 Programs," SIGMETRICS 74 Conference Proceedings, Montreal, October 1974 (extended abstract).

9. Hatfield, D. J. and Gerald, J. "Program Restructuring for Virtual Memory," IBM Systems J., 10, 1971, 168-92.

SPEAC

SPECIAL PURPOSE ELECTRONIC AREA CORRELATOR

Dr. Orin E. Marvel

Honeywell Inc.

ABSTRACT

In the high data rate systems (> MHz) associated with signal processing, a standard stored program computer is not fast enough. At the present time, parallel (1, 2) and pipeline (3) architectures are being used to solve these computational problems.

This paper describes a preprocessor which is optimized to perform sum-of-product operations with a "pipeline ripple through" architecture. A breadboard built to perform the "absolute difference" and "product" correlation functions using this architecture is described.

Introduction and Summary

With the introduction of low cost microprocessors and "ROM/RAM" memory elements, the area of real time control is rapidly expanding. Many problems requiring a high computation rate can best be solved by adding a special purpose preprocessor to the system. This paper describes a pipeline sum-of-products preprocessor architecture and a breadboard built to demonstrate:

- Pipeline arithmetic units
- Microprogram control
- Hardwired address generation
- Minimum/absolute difference correlation
- Maximum/product correlation

This system is built around the following three assumptions:

(1) Preprocessors require high speed repetitious processing of a sum-of-products nature. Typical applications are in the three to four sum-of-products per microsecond range.

(2) Most sensor based data (especially human related) has a very low dynamic range and four bit accuracy is sufficient.

(3) MSI/LSI hardware (multipliers, arithmetic limits, comparators, and registers) is easily obtained in four bit slices.

The breadboard described in this paper demonstrates the concepts of pipeline operation, sum-of-product architecture, and correlation processing. This type of demonstrator breadboard acts as an excellent training tool for new architectural concepts.

Pipeline System

In this paper, the pipeline concept will be associated with a preprocessing system. The pipeline system consists of a number of hardware modules (see block diagram of SPEAC) that each perform an operation on the data stream independently and possibly at the same time as the other modules. Data enters the beginning of the pipe and results exit at the end. The throughput of the system is a measure of how fast data enters and results exit from the system. Throughput is independent of the propagation time through the system. Figure 1 shows an example of a three-unit pipe that performs $AX^2 + BX + C$.

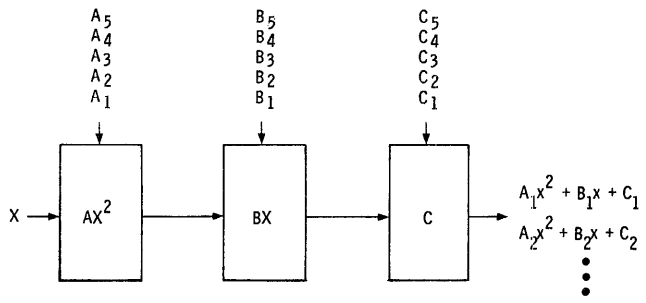


Figure 1. Pipeline Processor

In the system examples described in this paper, a microprogram control unit is the pipe controller or master executive. The only inputs, other than data, are clock signals and a master reset.

Pipeline Ripple Through Technique

The "Ripple Through" pipeline technique is a method of performing hard wired arithmetic and logical operations by implementing the algorithms directly in hardware. The one chip multipliers, adders, subtractors, and comparators that are readily

available allow high speed special purpose arithmetic operations to be performed with a small (< 50) parts count.

The performance that can be obtained with a sum of products unit depends on the component technology used. For example, the 16-bit sum of two 4-bit products takes:

(1) With CMOS implementation - 315 nanoseconds

(2) With TTL implementation - 98 nanoseconds

(3) With ECL implementation - 21 nanoseconds

However, one finds that the memory address computations and memory fetches associated with "feeding" the pipeline take more time than the arithmetic operations (this area is always overlooked in a special purpose processor design). The simplest system is obtained if the next address computation is performed in parallel with the arithmetic operation (for a TTL implementation, the address computation for a two dimensional correlation is 91 nanoseconds). Thus the total algorithm computation time depends heavily on the memory access time. A typical high speed TTL memory access time is in the 140 to 160 nanosecond region.

Preprocessor Architecture

The class of special purpose preprocessors described in this paper have the block diagram shown in Figure 2. The preprocessor after receiving a restart command generates all its own control and clock pulses. The unit produces memory addresses and requests; and receives the data to be processed. As soon as the computations are completed, an interrupt is raised so the result can be transferred to the main data processor.

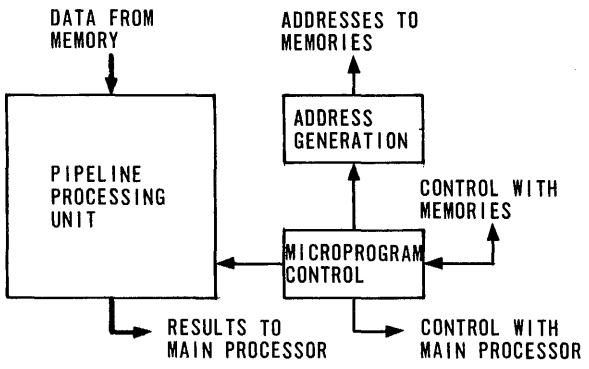


Figure 2. Preprocessor Block Diagram

The preprocessor contains the following three units:

(1) The microprogrammed control unit produces all internal control, data steering, and clock pulses. The control unit also handles all external interface control.

(2) The ripple through pipeline sum-of-products processing unit performs all the arithmetic computations. Ripple through pipeline means that the arithmetic functions of multiply, add, and compare are performed in different hardware subunits with no register or clock staging. The unit is designed by connecting all the arithmetic elements together and calculating the worst case delay between the memory port and the answer register. This becomes the basic computation period and defines the register gating clock.

(3) The hard wired address generation unit produces the addresses for the data required by the preprocessor. Since the addresses required are not in sequential order, the address computations are performed in parallel with the arithmetic computations. The complexity of most address generation algorithms cause this unit to be a limiting factor to the throughput of the system.

The preprocessor architectural trade-offs range from maximum throughput (arithmetic computation, address computation, and memory access in parallel) with highest parts count to the lowest parts count mechanization with the functions performed in sequence.

Applications

The preprocessor described in this paper is presently being applied to:

- Fourier Transforms
- Power Spectrum Density
- Correlation
- Weighted Averaging

Each of these problems requires the high throughput sum of a mathematical operation as described in the previous section. Simulations have shown that in the real world consistently high probabilities of success require sensor data digitized to 4 to 13 bit accuracy. The accuracy required depends on the signal to noise ratio, contrast in the sensor data, power spectrum, correlation aperture and array sizes. Where humans are involved with analyzing the sensor data, 4 bits has been found to be adequate.

The SPEAC demonstrator was built to perform two different correlation algorithms. SPEAC compares a sample array with a larger reference array and determines the position within the reference array where the best match is obtained. The output is a position and correlation result for that position.

The first algorithm is the Minimum Absolute Difference (MAD) algorithm. The MAD algorithm finds a correlation term such that:

$$P_{a,b} = \text{Min} \sum_{a,b} \left[\sum_{c,d} \left| R_{a+c,\, b+d} - S_{c,d} \right| \right] \qquad (1)$$

In other words, the sample array (Sij) is tried at all possible positions within the reference array (R_{kl}). At each position the sum of the absolute difference between the reference and sample arrays are calculated term by term. This gives an array of P_{kl} terms which are searched for the smallest term. (If a portion of the reference array contained a duplicate of the sample array, the P_{kl} for that position would be zero.) The MAD algorithm is very useful for correlations where the sensor data may contain a fixed bias. This same bias will appear in each P_{kl} term and will not affect the selection of the minimum term. However, the MAD algorithm is very susceptible to random noise upset, because of the absolute value computation.

The second algorithm is the Maximum Product (MP) algorithm. The MP algorithm finds a correlation term such that:

$$P_{a,b} = \text{Max} \sum_{a,b} \left[\sum_{c,d} \left(R_{a+c,\, b+d} \right) \left(S_{c,d} \right) \right] \qquad (2)$$

At each position where the sample (Sij) array is tried within the reference (R_{kl}) array the corresponding terms are multiplied and the sum of these products is obtained. This gives an array of P_{kl} terms which are searched for the largest term. (If the average value of the array terms is kept at 1/2 the maximum value, then the position where the sample array is duplicated within the reference array will have the maximum value.) The MP algorithm is very useful for correlations where the sensor data is noisy. Random noise will keep the average value over the array the same and not affect the selection of the maximum value. However, the MP algorithm is very suspectible to distributed bias, because of the non-linear variations of the product computation.

SPEAC Demonstrator

The Special Purpose Electronic Area Correlator breadboard shown in Figure 3:

(1) Performs Scene Matching using
- Mean Absolute Difference
- Product Correlation

(2) Operates on a
- 10 x 10 reference array
- 3 x 3 sample array

(3) Demonstrates
- Pipeline arithmetic processing
- Hardwired address generation
- Maximum and minimum detection

The operational speed of the demonstrator has been decreased by 1,000 so that the correlation computations can be observed. Of the total parts count, less than half are actively used to perform the correlation function.

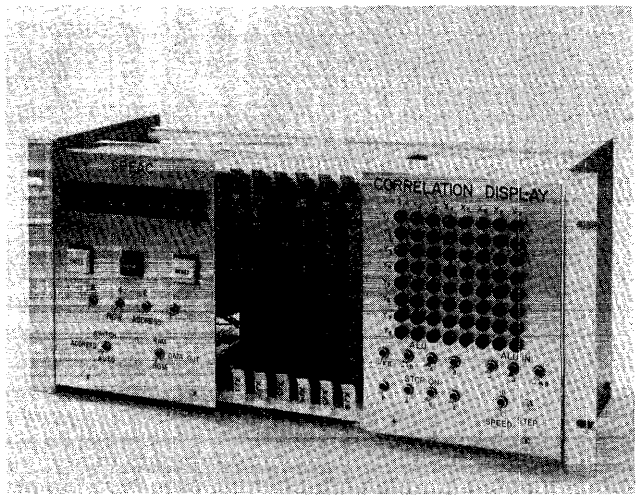


Figure 3. SPEAC Breadboard

Figure 4 shows an actual sample problem as performed on SPEAC. This example has a reference array with the samply array imbedded in it. The locations and correlation values of the exact matches are identified. One will notice in Figure 4 that a reference array of 10 x 10 and sample array of 3 x 3 produces a correlation array of 8 x 8. This can be verified by trying all possible positions within the reference array where the sample array can be placed.

Sample:

9	1	9
1	5	1
9	1	9

Reference:

5	1	7	2	4	7	1	7	1	8
5	9	1	9	8	8	7	5	8	0
0	1	5	1	0	4	2	8	2	3
6	9	1	9	2	2	8	4	8	4
0	3	2	8	4	2	2	8	2	6
8	6	2	2	2	7	1	4	1	1
5	7	1	2	2	2	0	1	3	2
0	0	6	7	2	7	5	2	9	2
4	2	0	7	7	7	4	5	9	5
5	8	3	1	0	4	1	2	9	2

Absolute Difference

27	61	32	44	54	24	54	19
48	0	49	24	22	43	17	49
40	48	37	37	43	23	57	22
36	23	45	27	37	37	29	45
45	26	40	36	33	38	37	29
32	38	35	20	43	26	31	38
44	27	41	40	37	32	39	39
37	39	40	33	48	36	25	56

Sum of Products

206	80	201	187	129	285	109	282
145	353	136	267	257	208	332	154
119	147	156	160	98	248	124	277
185	258	120	214	142	182	214	150
122	201	105	151	114	135	94	168
191	157	130	225	116	189	158	109
105	200	142	195	168	173	174	178
148	162	149	222	129	173	258	145

Figure 4. Sample Problems

Figure 5 shows a block diagram of the SPEAC processing section. The demonstrator consists of the following operational sections.

(1) The Auto address processor generates sample memory Addresses in a cyclic manner. At the same time the reference memory address is produced by adding the base (correlation array) address to the sample memory address.

(2) The sample array memory is mechanized with four 1 x 16 RAMs. A prearranged reference array is located in one 4 x 256 PROM. Also, four 1 x 256 RAMs can be loaded from the panel and used as the reference array.

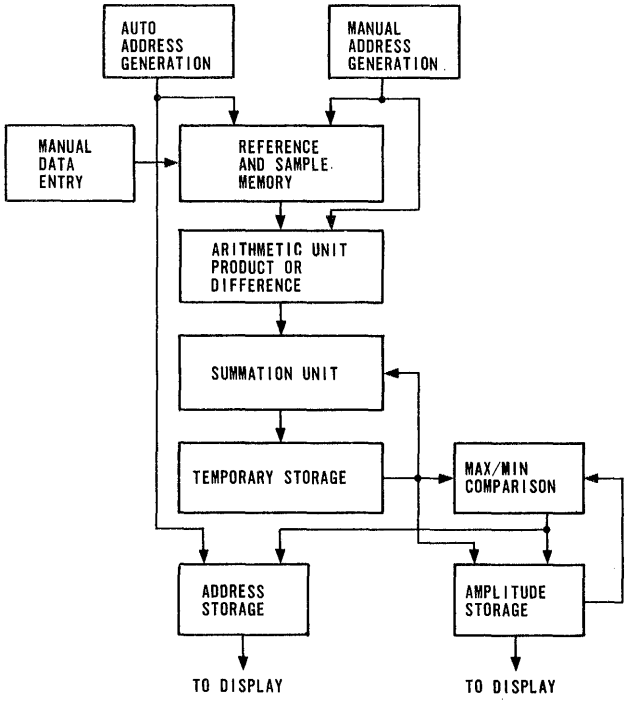


Figure 5. SPEAC Block Diagram

(3) The arithmetic unit contains a 4 x 4 multiplier chip for product calculations and a 4-bit subtractor chip for difference calculations. The absolute value subtraction is assured by a 4-bit comparator that selects the proper inputs to the subtractor. Thus, the arithmetic unit can perform $|R-S|$, RS, R^2 and S^2.

(4) The summation unit produces the sum of the term in the temporary storage register with one of the following:

- Multiplier Output
- Subtractor Output
- Reference Array
- Sample Array

(5) The temporary storage register which is edge triggered holds the partial summation term for nine sum iterations.

(6) The Max/Min comparison unit compares the last correlation term with the previously stored extremum correlation term. For product correlations, it keeps the larger value; while for the difference correlation, it keeps the smaller value.

(7) The Address and Amplitude storage register holds the best fit of the sample array within the reference array. The Amplitude portion is displayed on the correlation amplitude lights. The address portion is decoded to display the position of the optimum correlation.

This breadboard has been very helpful in understanding and solving operational and mechanization problems associated with a high speed pipeline preprocessor. It has shown that the "ripple through" pipeline architecture is a cost effective solution to real operational problems.

Bibliography

1. Hobbs, L. C., "Parallel Processor Systems, Technologies and Applications", Sparton Books, 1970.

2. Marvel, O. E., "HAPPE - Honeywell Associative Parallel Processing Ensemble", First Annual Symposium on Computer Architecture, 1973.

3. Wu, Y. S., "Architectural Considerations of a Signal Processor Under Microprogram Control", Spring Joint Computer Conference, 1972.

4. Rabiner, L. R. and Rader, C. M., "Digital Signal Processing", IEEE Press, 1972.

5. Webber, R. F. and Delashmit, W. H., "Product Correlation Performance for Gaussian Random Scenes", IEEE Transactions on AES-10, July 1974.

6. Pratt, W. K., "Correlation Techniques of Image Registration", IEEE Transactions on AES-10, May 1974.

ARCHITECTURAL ADVANCES OF THE SPACE SHUTTLE ORBITER AVIONICS COMPUTER SYSTEM

James M. Satterfield

National Aeronautics And Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Abstract

The Space Shuttle Orbiter manned reusable craft is being developed by NASA for applications in the 1980's and beyond. Navigation, guidance flight control, systems management and control, and payload checkout are but a few of the functions of the Orbiter which are being mechanized in the avionics computer complex.

The development of the system is traced through three distinct architectures to the current system. Factors affecting the architecture and system development are discussed. A system figure of merit for evaluating competing systems architecture is developed.

Introduction

The Space Shuttle Orbiter manned reusable craft is being developed by NASA for applications in the 1980's and beyond. Navigation, guidance flight control, systems management and control, and payload checkout are but a few of the functions of the Orbiter which are being mechanized in the avionics computer complex.

Mechanization of individual functions by computer control has been practiced ever since the 1940's, when specialized "computers" provided fire control and bombing and navigation aids to military pilots. More recently, navigation of both civil and military aircraft has been aided by highspeed digital computers. In the Apollo lunar flights, several digital computers provided flight control, rendezvous guidance and return to earth navigation functions.

The computer complex being developed for the Space Shuttle Orbiter will be called upon to provide not only navigation and guidance functions, but flight control over an extremely wide spread regime, systems monitoring and control, and payload checkout functions. This paper traces the early history of the Orbiter avionics computer complex and discusses some of the major architectural features of the system.

Evolution of the System

System architecture has undergone change and redefinition as program objectives were revised, requirements were better defined or understood, and subsystem design continued. The following traces the change in the system design.

Throughout most of the formal study phase (Phase B), the avionics system was characterized by a large centralized computer with an extensive, multiplexed data bus communications system linking various subsystems to the computer. Although a federated or distributed computer system was considered, it was not until nearly the end of the Phase B study that the centralized system was seriously questioned. Initial reaction drove the system design to a federated system using dedicated point-to-point wiring, but by the time of the RFP (request for proposal) release, the system design approached a centralized computer system once again, with several "central" computers instead of one and connected to subsystems with dedicated wires.

The first system architecture is illustrated in Figure 1 and was the system described in the proposal. Three large digital computers with 32,000 word memories (32 bit words) were specified for the guidance, navigation, and control functions. Separate Input-Output Boxes were specified to provide input-output channelization. Formatting of the displays, control of displays, and systems monitoring and payload related functions were to be provided by six smaller digital computers with 8000 word memories (16 bit words). CRT display devices with associated keyboards provided the system control and output displays. Two auxiliary tape memories were specified for storage of display formats and program copies.

Studies were then started to define a system wherein the number of computers would be reduced as well as the number of different types of computers. Figure 2 shows the baseline system which resulted from these studies. This architecture provided primary and standby computers for monitoring and control and payload related functions. These computers were identical machines with 32,000 word main memories. A fifth computer of a different type and with different memory size and organization, was provided for backup to the guidance and control functions. Table I compares the two architectures.

Continuing studies and better requirements definitions have led to the architecture of the current system, Figure 3. The "backup" computer was eliminated mainly on the assurance that catastrophic "generic" failures were preventable. It was replaced by a fifth computer identical to the man/standby GNC and system payload machines. Hardware interfaces were standized among the five computers so that any physical computer could be assigned either the GNC or the system-payload function, or both. Main memories were increased to 64,000 words for all machines.

Hardware-Software Considerations

The development of the avionics computer system was followed with great interest by those responsible for computer program development and verification. In fact, the avionics system was not completely specified and the system had already undergone two major architectural changes when the computer program designers began the software subsystem design.

Since the programming specifications provided to the software designers did not contain complete hardware subsystem descriptions, one of the early tasks was to assist in the hardware-software functional partitioning which had already begun. An objective was to help define the system architecture early so that software subsystem design could proceed aided by this influence on the system design.

The NASA plan for the Orbiter avionics software was patterned after Apollo and Skylab experience in terms of requirements development and contractor-government relationships. Because of the critical nature of the Orbiter avionics software performance in providing flight control and guidance functions for the Orbiter, reliable software was required. A better solution to program verification than "independent verification, wherein a separate contractor was engaged to retest the already tested program, was obviously needed.

There is an abundance of literature relating to the methodology for achieving reliable, quality software through structured program segments organized in a hierarchiacal order. In 1966, G. Jacopini[1] showed that any flow chartable program can be represented with three basic clauses or program figures connecting statements and the go to figure is not required at all. E. W. Dijkstra[2] further ruled out the use of the go to as detrimental to a visualization of progress of a process,and hence prevents an understanding of the program by the programmer.

H. D. Mills[3] and B. H. Liskov[4] have further developed the notions of segmented design using structured programming and introduced the ideal of the importance of organization structure. F. T. Baker describes in (5) the promising results achieved through application of these techniques to a commercial endeavor.

An R&D project to develop a higher order language for further space applications had been started in 1970 with a company well versed in Dijkstra's and others' work in reliable software and program correctness. An objective of the language and compiler development was enforement of structured programming rules. A dialect of the R&D language was specialized as "HAL/S" and was mandatory for use in all programming in the Orbiter avionics computers. This was necessary because the reliability demanded of the computer program could be satisfied only by adopting Harlan Mills' objective of never finding the first error, no matter how much it is read, tested, or used.[3]

Hardware-Software Integration

Finally, some comments are due on the integration, as currently practiced, between the hardware and software subsystems design. In the usual procedure, the hardware design is postulated and then software integrates the various hardware components and makes them function as a system. The program objective of use of "off-the-shelf" equipment and the realities of cost and weight budgets sometimes forced a selection of hardware based not on functional or performance requirements but based on lowest cost, weight, and availability. It is not hard to understand the difficulty in optimizing such a potpourrie of electronic components. The software designers were accustomed to preparing software designs for existing and supported (by operating systems) hardware and the evaluation of hardware options against software techniques was difficult. They were not, by and large, experienced in circuit design and hence lacked the capability to perform hardware software trades. Inexpensive programmable digital controllers and computers, which have replaced much discrete component logic circuitry, caused hardware designers some difficulty in applying the new devices in an optimum fashion and software designers some difficulty understanding them at a system level and then differentiating between general purpose digital computers and specialized controllers.

Apparently, a current need in terms of capability of technical people seems to be systems designers who can write and code programs as well as design logic circuits, who understand computer circuits as well as the set of instructions which are executed by the computer hardware. These people are required for hardware-software trade studies and partitioning of functions which must be the basis for effective system design out of which is produced an optimum system.

Appendix I

System Figure of Merit

Because so many configurations and designs are possible in the Orbiter avionics computer system, a System figure of Merit is developed to discriminate between the designs. The System Figure of Merit measures the system in terms of economy of redundancy management applications and in terms of the effectiveness of the system design. The System Figure of Merit is defined as follows:

$$\text{System Figure of Merit} = \frac{\text{Levels of redundancy req'd}}{\text{Levels of redundancy prov'd.}} \quad \frac{\text{Levels of redundancy req'd}}{\text{No. of computers used for all functions}}$$

Applying the System Figure of Merit to the systems discussed, one obtains:

System Figure of Merit (initial system)	= 3/4 x 3/9 x 1/4 = .250
System Figure of Merit (2nd system)	= 3/3 x 3/5 = 3/5 = .600
System Figure of Merit (current)	= 3/4 x 3/5 = 9/20= .450
System Figure of Merit (author preferred system)	= 3/4 x 3/4 = 9/16= .563

The second system design, although ranking highest (the upper bound on the System Figure of Merit is 1.0), required that the failure detection and redundancy management system be able to differneitate between a malfunctioning pair and pick the currently functioning member of the pair, to a certainty of 1.0. It is not clear how one would achieve such detection with techniques available today.

REFERENCES

1. Bohm,,C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Communications of Association for Computing Machines, Volume 9, (1966), pp. 366-371

2. Dijkstra, E. W., "Go to Statement Considered Harmful," Communications of the Association for Computing Machinery, Volume II, No. 3 (1968) pp. 147-148

3. Mills, H. D., "On the Development of Large Reliable Programs," IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 155-159

4. Kiskov, B.H., "A design Methodology for Reliable Software System," Proceedings of Fall Joint Computer conference, AFIPS, Volume 41, part 1 (1972), pp. 191-199

5. Baker, F. T., "System Quality Through Structured Programming," Proceedings of Fall Joint Computer Conference, AFIPS, Volume 41, Part 1, (1972) pp. 339-343

6. "Proposal for Space Shuttle Program, Technical Proposal, Volume III," Space Division, North American Rockwell, SD 72-SH-50-3, 12 May 1972

7. Satterfield, J. M., "Avionics Software Panel Minutes," Johnson Space Center Memorandum FS63-73-47, May 11, 1973

8. Satterfield, M. M., "Minutes of the Fourteenth Avionics Software Panel Meeting," Johnson Space Center, Memorandum FS-63-73-146, July 13, 1973

9. "Flight Software Memory Sizing and CPU Loading Estimates," IBM Federal Systems Center, Houston, TX, 74-SS-0119, April 1, 1974

10. Unpublished notes from M. D. Cassetti, "Orbiter Flight Computer Candidate Software Partitioning Operational Flight," April 2, 1974

RESERVE ACTIVE	RESERVE GROWTH MODULAR	WEIGHT	PARTITION RQD	VOTING AFTER 1 FAIL	
65K	96K	566	YES	NO	2-64K; 1-32K; 2-32K
141K	-	695	NO	YES	5-64K
146K	-	695	NO	YES	5-64K
66K	96K	630	MARGINAL	YES	5-48K
99K	-	556	NO	NO	4-64K
83K	16K	530	YES	NO	2-64K; 2-48K
51K	48K	504	YES	NO	4-48K
85K	-	573	YES	NO	3-88K; 1-32K

Table 1 - Comparison of Avionics Computer Systems

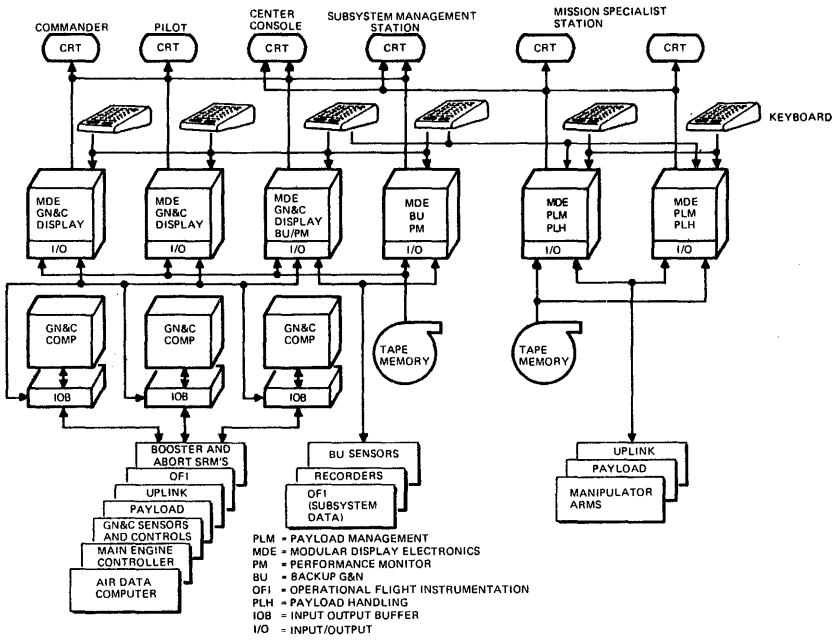


Figure 1 - Initial Space Shuttle Orbiter Avionics Computer System

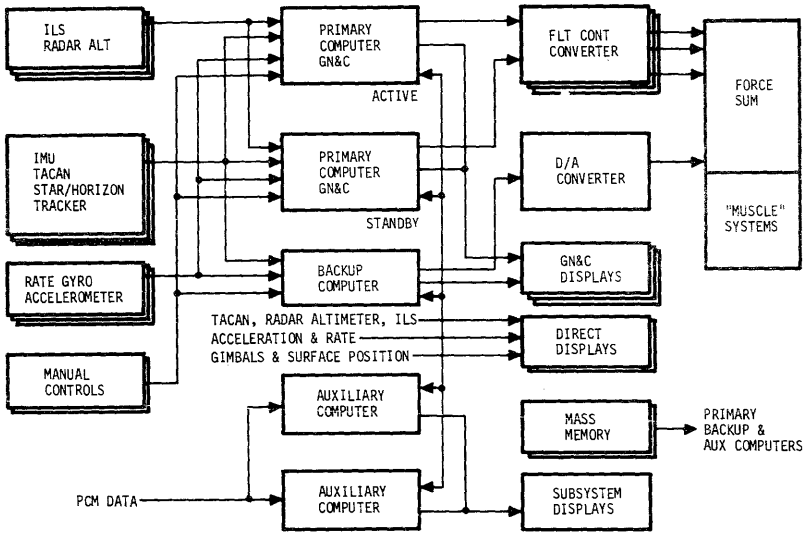


Figure 2 - Interim Space Shuttle Orbiter Avionics Computer System

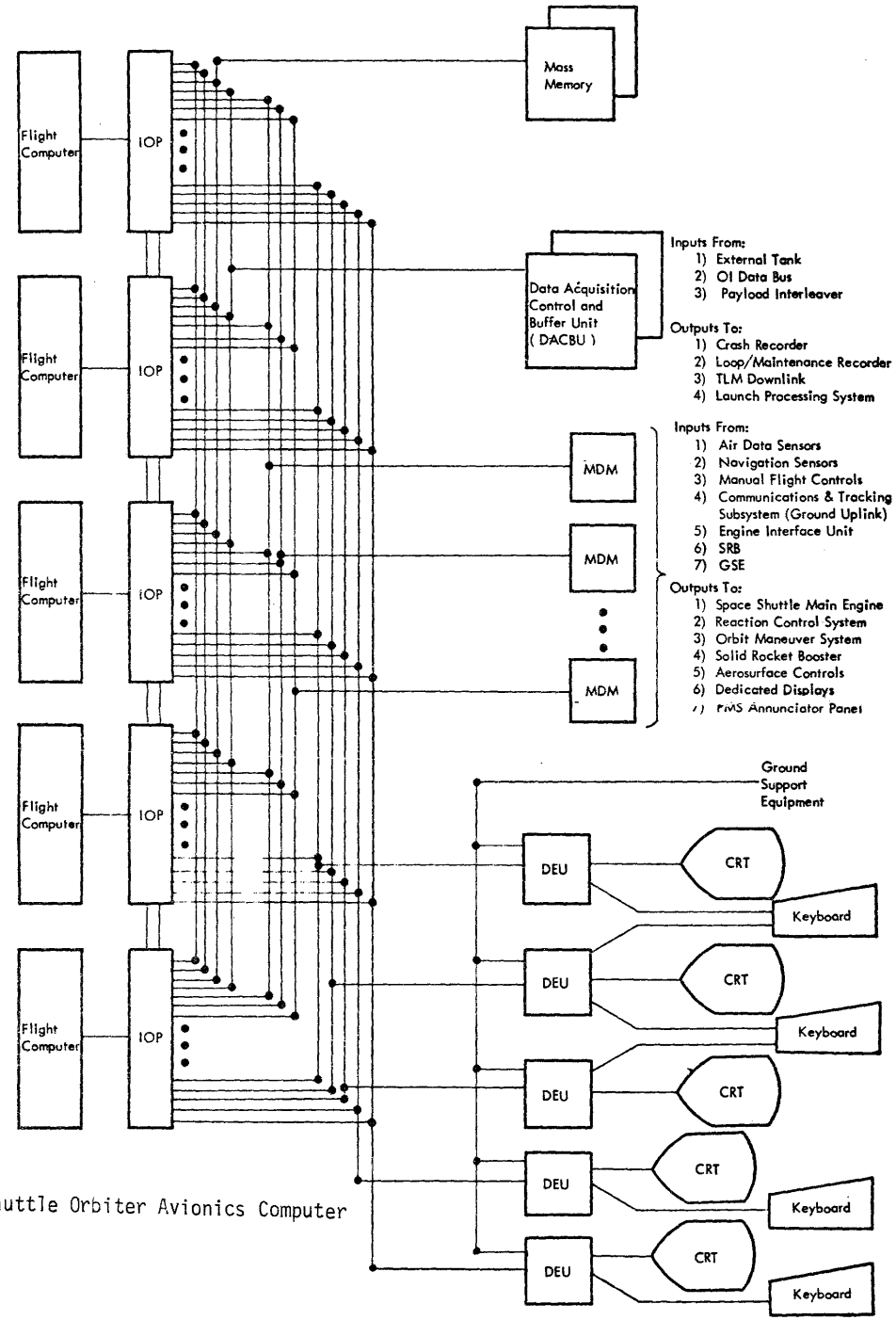


Figure 3 - Space Shuttle Orbiter Avionics Computer System

DESIGN STUDY OF AN AVIONICS NAVIGATION MICROCOMPUTER

Uno R. Kodres*
William L. McCracken**
Naval Postgraduate School
Monterey, California

Abstract

A microcomputer program to solve the complex task of airborne navigation was developed to demonstrate the practicality of replacing costly general-purpose digital computers with relatively inexpensive dedicated microcomputers on board naval aircraft. The microcomputer program showed that microcomputers have sufficient speed and accuracy to solve the navigation problem. In order to overcome the microcomputer's major deficiencies, speed and accuracy, special arithmetic subprograms based on table look-up were developed to trade inexpensive memory for more speed. An application of graph theory in the form of process graphs was made to facilitate the development and documentation of the navigation program.

Introduction

Naval aircraft are depending more and more on airborne digital computers. The digital computers currently used by naval aircraft are all large general-purpose computers. The large size, large power requirements, and great cost have limited each aircraft to one such computer. The complexity of these computers has made maintenance difficult. The large cost of each unit makes spares prohibitive; therefore, one computer going down results in the operational loss of one aircraft.

It is advantageous to utilize several small distributed computers to meet all of the systems requirements. Completely separate computers could be used for the various system requirements with back-up computers ready to fill in when there is a failure. In addition to reliability improvements, a distributed approach offers the possibility of matching equipment more closely to system requirements, and increasing standardization.

The creation of the microcomputer, using new developments in Large Scale Integration (LSI) technology, has made the distributed computer system possible. The microcomputer is a general purpose computer on a set of standard LSI chips and associated integrated circuits. The LSI chip measures 200 mils by 200 mils, requires less than one watt of power and costs about $30. The limitations of a microcomputer are a limited instruction set and slow speed.

This paper discusses the possibility of using the MCS-4 microcomputer as the Avionics Navigation Computer in a complex navigation system. Included in this paper are the programming aids developed, a program analysis of the requirements of this system, a discussion of the executive routine and subroutines used in this program, and the results of an error bound analysis of the navigation program.

The Intel MCS-4 was chosen as the microcomputer in this design study for two major reasons. The first reason is that the MCS-4 is the least powerful and hence serves as a lower bound of the microcomputers. To prove that the MCS-4 is capable of handling the required navigational computations, would in itself prove microcomputers capable of handling complex tasks. The second major reason for choosing the MCS-4 microcomputer is that it is available, has been tested, and has the required software aids to complete a design study.

The cost of developing a system that uses the MCS-4 microcomputer can be divided into two areas: Hardware and software. The hardware costs have been shown to be small for the microcomputer. This is a list of the MCS-4 hardware costs:

	(1)	(over 100)	(over 1000)
CPU	$60	$30 each	$15 each
ROM	$60	$15 each	$ 5 each
RAM	$30	$15 each	$ 5 each

The software cost of programming the microcomputer is not really known. The assembler-type language used to program the MCS-4 requires considerably more effort than a higher level language, such as FORTRAN. All indexing and transferring of data between the processor and the assigned locations in memory must be written into the program. The programming aids developed for this system were designed to decrease the cost of programming microcomputers.

Navigation System

Air Navigation is the process of directing the movement of an aircraft from an initial point to a desired final point. A Navigation System must provide timely coordinate measurement and computation of the aircraft's current position. The desired navigation system is one that gives a continuous, real-time indication of where the aircraft is located. The need for a navigation system to be reliable is a necessity. In case of partial equipment failure, the navigation system must automatically switch to an alternate source of information. Decision-making circuitry must be a part of the navigation system to insure a continuous flow of accurate navigation information.

The navigation system discussed in this paper is an Inertial/Doppler system integrated by an MCS-4 microcomputer. The microcomputer combines the short-term accuracy of the Inertial with the long-term accuracy of the Doppler to obtain the most probable position of the vehicle. The microcomputer is programmed to provide decision-making flexibility to insure the outputs remain accurate during partial system failure. The wind influencing the vehicle is continually computed and updated in the MCS-4 memory so that the computer can automatically switch to an air-data mode of operation in case of Inertial and Doppler failure.

The navigation equations used take the dead-reckoning outputs of the Inertial, Doppler, and Air-Mass Systems and extrapolate the present position of the vehicle from the last known position. Direct position data can be inputted into the algorithm and

* Associate Professor of Mathematics and Computer Science
** Lieutenant, U. S. Navy

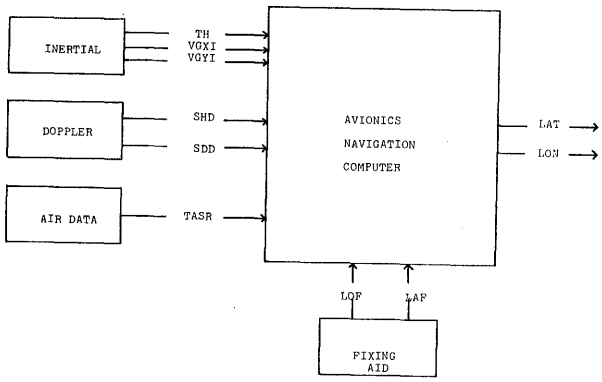
is used to update the position of the vehicle. The system drift error is computed by comparing the known position with the dead-reckoned computed position.

Navigation Process Graphs

Programming the navigation equations would be an easy task in a higher level language such as FORTRAN. To program the same equations in an assembly language proved to be tedious and complex. The requirements of keeping minute details in mind led to the development of a graphical means of representing the microcomputer program.

Graph theory provides a simple and powerful tool for constructing mathematical models of discrete arrangements of objects. The process graph consists of vertices which are pairwise connected by a directed line.

The MCS-4 microcomputer was first thought of as a black box taking inputs from the Inertial, Doppler, Air-Mass, and Position Fixing Systems and outputting the vehicle's current position in latitude and longitude. The graphical representation of the total system is shown in Figure 1.

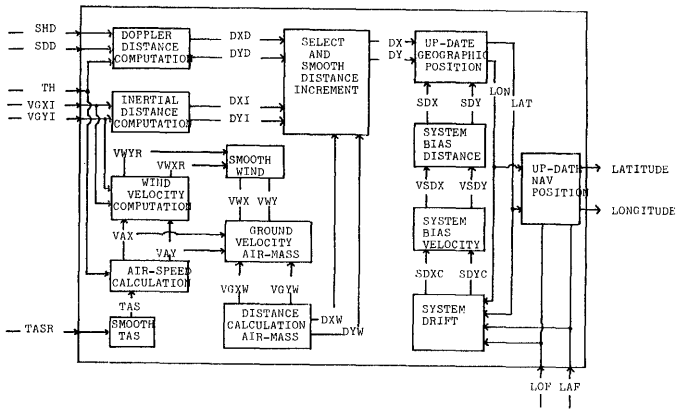


NAVIGATION SYSTEM BLOCK DIAGRAM

Figure 1.

Programming the navigation equations was accomplished by continually breaking down the functional operations into smaller and smaller parts until the operations were simple enough to be directly written in the MCS-4 machine language. The functional process graph, Figure 2, represents how the microcomputer program was initially broken down into the fundamental navigation equations.

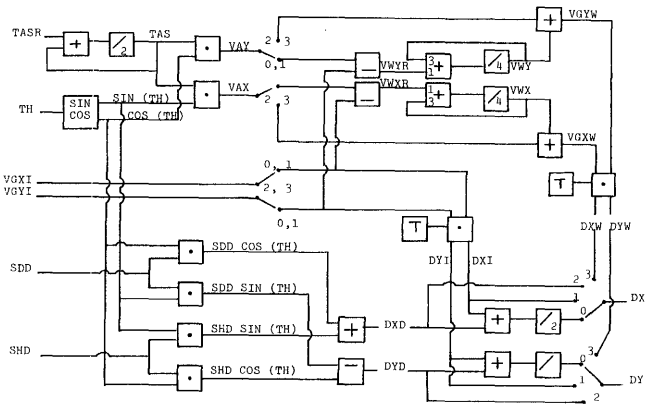
The functional process graph was analyzed to define the flow of the input variables and to determine the feasibility of a multiprocessor system in order to shorten the required computational time. It was noted that the program could be broken into two parts, the calculation of the distance increments, and the calculation of the latitude and longitude from the distance increments. The analysis demonstrated that the computation time could be nearly halved by having two microcomputers working simultaneously to produce the desired outputs. The system was designed to have one microcomputer receive the given input and compute the distance increments traveled, while at the same time, the second microcomputer computes the latitude and longitude from the previously calculated distance increments.



NAVIGATION FUNCTIONAL FLOW CHART

Figure 2.

The functional process graph was then broken down into the operational process graphs for each microcomputer. The operational process graph, Figure 3, represented the desired program for the first microcomputer. The program described by the operational process graph was written to investigate the time and effort required to develop the required software.



OPERATIONAL PROCESS GRAPH

Figure 3.

The program was written in a modular form. Each operation represented in the operational process graph was written as a separate subroutine. An executive routine was then developed to call each subroutine in the proper order.

Program Analysis

A program analysis was developed to define those problem areas that had to be solved before programming the microcomputer. The areas that were investigated were computational speed, memory space available, and accuracy required.

The operational process graph, Figure 3, represented the tasks to be accomplished. The type of operations and number of operations that were required are listed as follows:

Operations	Times Called
Multiply	12
Cosine	1
Sine	1
Addition	12
Subtraction	3
Division by two	7
TOTAL	36

The operational process graph was used to determine the speed limitations on each operation programmed. The total navigation cycle was limited to 200 milliseconds. Since the number of operations required to be performed differed depending on what sensors were operational, the critical path of the operational process graph was determined. The critical path occurred when the Inertial and Doppler were both operational, Figure 4.

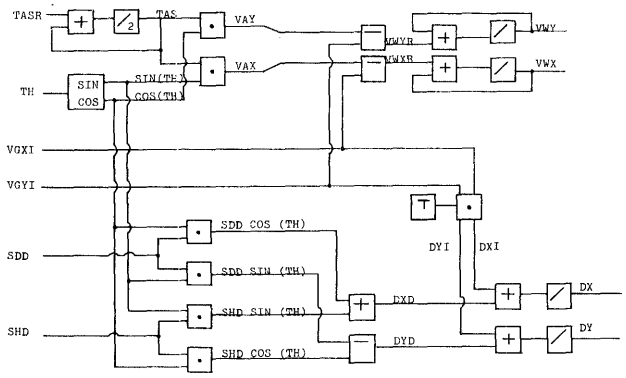


Figure 4. OPERATIONAL CRITICAL PATH

The operations involved in the critical path were ten multiplies, Cosine and Sine calculations, ten additions, three subtractions, and seven divisions by two. The total computational time for the critical path was limited to 200 milliseconds.

The critical operations which had to be developed were the Multiply, Cosine, and Sine subroutines. Investigating previous work on the MCS-4 indicated that previously programmed multiply, Cosine, and Sine routines were requiring 50 msec., 650 msec., and 750 msec. respectively. In order to program the microcomputer for navigation, the development of routines which required less than 200 msec. was essential.

The amount of memory available to program the navigation routine was a function of the 4004 CPU and the number of microprocessors used. One 4004 CPU can directly drive sixteen ROMs and sixteen RAMs. The number of instructions in the navigation routine was limited by the space available in the ROMs. Since each ROM could hold 256 instructions, the program was limited to 4,096 instructions per microprocessor.

The time required to sequentially execute every instruction in the sixteen ROMs would be 44 millisecond. Since the program was to be written with an executive routine and a set of subroutines that would repeat the same set of instructions several times, the limiting time constraint would be reached before using up the available ROM space in one microprocessor. It was determined from this analysis that ROM space would not be a limiting factor in writing the navigation program.

The amount of memory space available to store the values of each variable was determined by the space available in the RAMs. Since one 4004 CPU could drive 5120 bits of RAM, the navigation program was limited to 320 variables of 16 bits for each microprocessor. The number of variables required was determined from the operational process graph, Figure 3, where each line connecting a pair of vertices represents one variable. There were 54 lines indicating that a maximum of 54 variables were required plus those variables used in any single operation. By overlaying variables in the same RAM memory space, the memory space requirement was reduced. It was determined from this analysis that RAM space available would not be a limiting factor.

The required accuracy of the navigation program was a function of the accuracy of the input variables. The accuracies of Inertial, Doppler, and Air-Mass systems used on board the P3C aircraft were used in this analysis as representing the state-of-the-art systems in naval aircraft today. The specifications for these systems are as follows:

System	Designation	Accuracy
Inertial	ASN-84	$\pm$ 1.5 knots RMS
True Heading	ASN-84	$\pm$ 9 ARC-MIN RMS
Doppler	APN-187	$\pm$ 1.0 knots RMS
True Air Speed	Pitot-Static	$\pm$ 2.0 knots RMS

The accuracy of the navigation program was a function of the accuracy of input data as well as the bit size assigned to each variable. The limited accuracy of the input data permitted each variable to be no greater than 16 bits. This allowed each variable to be represented by four hexadecimal-digits with the first bit assigned as the sign bit. The hexadecimal point for speed measurements was fixed so that there is one hexadecimal digit to the right of the decimal point. This allowed the accuracy of the speed inputs to be within $\pm$ .0625 knots. The range on the inputs due to a 16 bit variable limitation was $\pm$ 2047.99 knots. The accuracy requirement was not considered a major limitation in the program analysis.

Navigation Program

A large number of the mathematical operations required in the navigation program are repeated many times. In order to decrease the total programming effort and also decrease the memory-capacity requirements, many of the operations required were written as subroutines. The subroutines developed for the navigation program were divided into two groups, those involving complex mathematical operations and those involving more common functional operations.

The major limitations of the MCS-4 microcomputer to be overcome were the limited instruction set and slow speed of calculation. The multiplication routine was especially written to enhance the capability of the MCS-4 microcomputer in order to satisfy the requirements of the navigation program.

A multiplication routine had been written involving multiplication by a series of additions. The program required only fifty instructions; however, the computational time was 40 milliseconds. Since the navigation program required a minimum of ten multiplications, this method was unsatisfactory.

One of the advantages the MCS-4 microcomputer has is its inexpensive memory. It was decided to investigate a different way of programming the microcomputer that would take advantage of available memory. It was discovered that memory space could be traded for speed by using a table look-up scheme. An example of this

method follows:

	$(X_3$	X_2	X_1	$X_0)$
TIMES	$(Y_3$	Y_2	Y_1	$Y_0)$
	X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0
X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1	
X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2	
X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3	

The table of values used in the multiplication routine consisted of a 16 by 16 matrix of product values. Each product value was an exact value of a multiplication of two single hex-digit numbers. The row that would normally contain the products of a zero multiplication was used for instructions within the ROM containing the table.

The method used in this procedure becomes very complex because of the large number of separate hex-digits involved and the small number of index registers available to store each digit. This problem becomes more complex since each addition of two hex-digits creates a possible carry. A solution to this problem was to make a process graph that simulated the multiplication process. The process graph for the multiplication routine is shown in Figure 5. The symbols used have the following meaning:

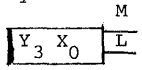
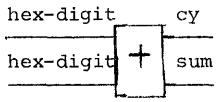
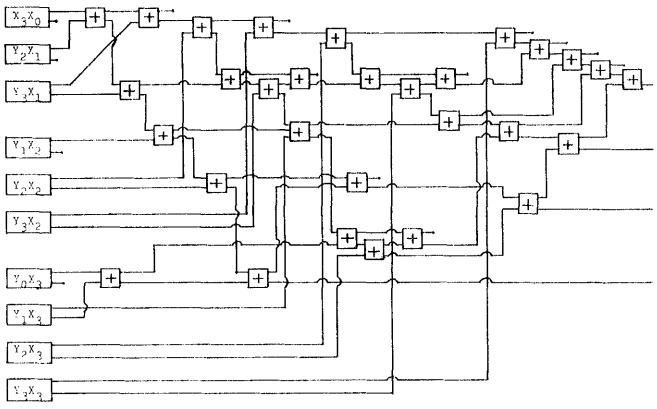


table value of product of the digits X_0 times Y_3



addition

The multiplication routine was written to give a truncated product of two four-digit hex numbers accurate to four significant hex digits.



PROCESS GRAPH FOR MULTIPLICATION ROUTINE

Figure 5.

The multiplication routine required the memory space of three ROMs and computed its result in five milliseconds.

The development of a Cosine routine for the microcomputer which could compute sufficiently fast was one of the major programming tasks. Cosine routines written for general purpose computers are usually written as series approximations in order to save memory space. The object of the routine written was to increase the speed of calculation.

Two cosine routines previously programmed on the MCS-4 Microcomputer were investigated. The first was a Chebyshev approximation routine which required 750 milliseconds to compute the Cosine. The second routine investigated was a Cordic approximation which required 350 milliseconds. Both methods were too time-consuming for this project.

The procedure developed in this project was a table look-up, linear interpolation routine. The Newton Divided-Difference Interpolating Polynomial was used because of its simplicity. The size of the table required and the accuracy of the results are both functions of the degree of the Polynomial used. For simplicity and speed, a first-order divided-difference table was used which resulted in a linear interpolation of the form:

$$F(X) = F(X_0) + (X - X_0)\, F[X_1 \cdot X_0]$$

where $X = \theta$, $F(X) = \cos\theta$

$$F[X_1 \cdot X_0] = \frac{F(X_1) - F(X_0)}{X_1 - X_0}$$

The table consisted of all values of $F(X_i)$ and $F[X_{i+1} \cdot X_i]$ for $F(X) = \cos\theta$, $0 \leq \theta \leq 1.88$ radians in hexadecimal. The table was constructed from a FORTRAN program which used a decimal increment of $\frac{1}{.03125}$, equivalent to .08 hexadecimal, and outputed the desired table values in hexadecimal.

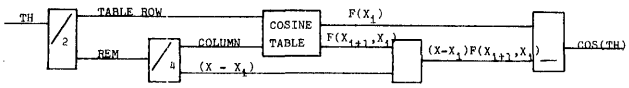
The size of the table loaded into the program was a function of the required accuracy of the Cosine routine. The data supplied to the program from the navigation devices was accurate to three significant figures.

Different-size tables were constructed and tested for accuracy. Since the interpolation was linear, the largest error occured at the midpoint between each table value.

The table size was adjusted until the maximum error was within three units in the fourth significant figure, thus guaranteeing three significant figure accuracy.

The table values were loaded sequentially into the ROM. The first entry was the value of $F(X_0)$ followed in order by $F(X_1 . X_0)$, $F(X_1, \ldots\ldots)$, $F(X_n)$, $F(X_{n+1} \cdot X_n)$. The remaining part of the ROM was used for the interpolating routine. The step by step procedure of the Cosine routine is shown in the Cosine process graph, Figure 6.

The table used in the Cosine Routine was also the table required by a Sine routine. It was noted that this routine could also be used to find the sin θ for $0 \leq \theta \leq 90°$ by subtracting the input θ from 90 degrees (.188 rads-θrads) and using the same routine since $\cos(90 - \theta) = \sin(\theta)$.



COSINE PROCESS GRAPH

Figure 6.

The memory space required by the Cosine routine was one ROM plus the space taken up by the subroutines called by the Cosine routine. The main part of the Cosine routine contains only 46 instructions. The time required to execute the Cosine routine was basically the time required to execute the multiplication. The Cosine routine required only a total of 5.17 milliseconds. This computational speed represents a 70-fold decrease in the computational time to compute the Cosine by previously available routines. By table look-up schemes, it was proven that the computational speed of the microcomputer could be competitive with that of a general-purpose computer.

The common subroutines were written to do the basic housekeeping operations such as storing data, simple arithmetic, shift operations, and transfer of data between RAM and IR. These subroutines were called by the executive routine and the multiply and Cosines routines to aid in the data handling. The functions handled by the common subroutines were those best suited for the MCS-4 and therefore could be written in a straight-forward way requiring little speed or memory space. The functions of the common routines were broken into three groups: Arithmetic, Shifting, and Data Handling.

The arithmetic routines handled the simple additions and subtractions required in the navigation program. These routines were handled well in the MCS-4 by the 4-bit ripple-through carry type adder incorporated in the 4004 CPU. This allowed direct addition or subtraction of two hexdigits in either the accumulator or RAM. There were two addition routines, two subtraction routines, and two special purpose arithmetic routines written.

Multiplication and division of hexdigits by a multiple of two was accomplished by shifting the variable either left or right the required number of bits. To take advantage of this capability, two subroutines were written for the navigation program that involved division by two.

The transfer of data within the navigation program was accomplished by the data handling routines. These routines saved much memory space in the Executive Routine and the Multiply and Cosine Routines by grouping these tasks into separate subroutine calls. There were five data handling routines written for the Navigation program. Three routines were written to transfer data between the IRs and RAM. Another routine was written to transfer data between different locations in RAM. A special routine was written to load the proper time interval of one navigation cycle into the IRs. All five routines were written to handle data of four hexdigit size.

Each operation defined in the operational process graph, Figure 3, was successfully programmed within the memory and time constraint of the program analysis. Each subroutine was written in a modular form allowing for easy addition or subtraction of new code. The memory size, computational time, and capability of each subroutine is listed in Table I.

Subroutine	Length (8-bit words)	Time (μsec)	Purpose
MULT	743	5000	[IR(8→B)]·[IR(C→F)]→IR(C→F)
COS	255	5170	COS[IR(C→F)] → IR(C→F)
ADDRAM	12	289	RAM(0,1) + RAM(2,3)→RAM(2,3)
ADDRAMIR	22	238	RAM(0,1) + IR(C→F)→IR(C→F)
SUBIR	17	183	IR(4-7)-IR(C→F)→IR(C→F)
SUBRAMIR	32	346	RAM(2,3)-IR(C→F)→RAM(2,3)
COMPLEMENT	28	302	-[RAM(2,3)]→RAM(2,3)
COMANGLE	18	194	(90°)-IR(C→F)→IR(C→F)
DIV2IR	24	259	[IR(C→F)]/2→RAM(0,1)
DIV2	33	356	[RAM(2,3)]/2→RAM(2,3)
RAMIRC	16	173	RAM(2,3) → IR(C→F)
RAMIR8	16	173	RAM(0,1) → IR(8→B)
IRRAMC	16	173	IR(C→F) → RAM(2,3)
TRANRAM	8	86	RAM(2,3) → RAM(0,1)
TIME	9	97	(Nav Cycle Time)→IR(8→B)

Table I NAVIGATION SUBROUTINES

The Executive Routine was written to call up the subroutines in the order described by the operational process graph, Figure 3. The Executive Routine established the priorities of each function and was designed to make all the decisions in the execution of the Navigation program. The variables used by the Executive Routine were all stored in RAM. The Executive Routine was loaded into two ROMs with space left for addition of new code.

Error Bound Analysis

Microcomputers have a limited arithmetic capability. It was therefore very important to avoid unnecessary precision throughout the calculations. Since the inputs into the Navigation program came from instruments whose precision is limited to three hexadecimal digits, the choice of four hexadecimal arithmetic was considered to be sufficiently accurate. An error bound analysis was performed to show that the input errors dominate the total error.

The starting point for the error analysis is the operational process graph, Figure 3. It was apparent from the operational process graph that the outputs DX and DY are symmetric, therefore an analysis of only the computations for DX were made. A process graph that involved only the operations which have an influence on DX was constructed, as shown in Figure 7 from the operational process graph, Figure 3.

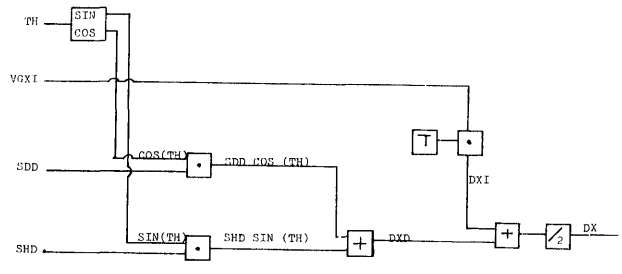


Figure 7. PROCESS GRAPH OF DX

The errors corresponding to each operation in Figure 7 were designated e_1, e_2, e_3,...,e_8. The initial errors of the inputted data were expressed as e(TH), e(VGXI), e(SDD), and e(SHD). Due to the smallness of the errors, the products of errors were considered negligible when compared to the linear terms.

There were two means by which each operation contributed to the error propagation.

(1) Transmitting the errors which were inputted into the operation.

(2) Adding an error of its own, which is due to the rounding or truncating process which limits the number of digits carried to the next operation.

The transmitted errors were calculated by calculating the differentials of the expression.

$$d(x+y) = dx+dy$$

$$d(xy) = ydx+xdy$$

$$d\left(\frac{x}{y}\right) = \frac{ydx-xdy}{y^2}$$

$$d(\sin x) = \cos x\, dx$$

$$d(\cos x) = -\sin x\, dx$$

The rounding or truncating errors were simply added to the transmitted error and thereafter propagated through the remainder of the calculation.

The error bound for each operation in Figure 7 due to roundoff and truncation was found to be:

Operation	Error Bound (Decimal)	Corres. Error
Cosine	3×10^{-5}	$\lvert e_1 \rvert$
Sine	3×10^{-5}	$\lvert e_2 \rvert$
Multiplication	3×10^{-5}	$\lvert e_3 \rvert \cdot \lvert e_4 \rvert \cdot \lvert e_5 \rvert$
Division	8×10^{-6}	$\lvert e_8 \rvert$
Subtraction	8×10^{-6}	$\lvert e_6 \rvert \cdot \lvert e_7 \rvert$
Addition	0	

The error bound for the inputs was obtained from published sources. Since an actual maximum error for each system could not be found, the 3σ value, 99.7% CEP, was used. The systems on board the P3C naval aircraft were used as representing the current "state-of-the-art" systems in operational use today.

System	Error (3σ)	Corres. Error
Inertial Navigation System	5.4 NM/HR	$\lvert e(VGXI) \rvert$
Inertial Navigation True Heading	.5 degrees	$\lvert e(TH) \rvert$
Doppler Along Heading	1.8	$\lvert e(SHD) \rvert$
Doppler Across Heading	3.6	$\lvert e(SDD) \rvert$

The results of the error bound analysis indicated that the maximum error created in the navigation microcomputers computations was only 1.8 per cent of the total maximum error. It was concluded from this analysis that the accuracy of the MCS-4 navigation program using a 16-bit fixed word data length was well within the limits required for the navigation problem.

The best way to check the results of this error bound analysis would be to fly the system in an actual aircraft. Since an aircraft was not available for this purpose, a detailed FORTRAN simulation program was written to test the functions of the navigation program.

The FORTRAN simulation program was developed as an exact simulation of the navigation program developed for the MCS-4 microcomputer. The program was written by utilizing the process graphs previously developed. The program included a parallel solution of the navigation equations utilizing the FORTRAN routines available on the IBM 360/67 computer. The results of these runs and the comparison of the errors developed are summarized in Table II.

Type of Path	Nav. Mode	Type Input	Computed X-Loc. FORTRAN	Computed X-Loc. MCS-4	Computed Y-Loc. FORTRAN	Computed Y-Loc. MCS-4	Maximum Error Due to NAV Program	Maximum Error Due to Input
Point to Point	Integrated	Exact	-0.9814	-0.9811	1.0427	1.0425	.03%	
	Inertial		-1.1041	-1.1038	0.9813	0.9811	.03%	
	Doppler		-0.8588	-0.8586	1.1041	1.1038	.03%	
	Air-Mass		-1.0912	-1.0909	0.9812	0.9808	.04%	
Point to Point and Back	Integrated	Exact	-0.0001	0.0	0.0	0.0	0.0	
	Inertial		0.0	0.0	0.0	0.0	0.0	
	Doppler		-0.0001	0.0	-0.0001	0.0	0.0	
	Air-Mass		0.0	0.0	-0.0001	-0.0001	.01%	
Point to Point	Integrated	Max Error	-1.0037	-1.0034	1.0428	1.0423	.05%	2.23%
	Inertial		-1.1228	-1.1227	0.9625	0.9622	.03%	1.88%
	Doppler		-0.8845	-0.8841	1.1230	1.1225	.05%	2.57%
	Air-Mass		-1.1088	-1.1081	0.9636	0.9639	.07%	1.76%

TABLE II RESULTS OF FORTRAN SIMULATION

The results confirmed the results of the error bound analysis. It was noted that the greatest computational error occurred when the vehicle traveled a direct path with constant inputs. This was due to the linear addition of the truncation error when the inputs remain constant. It was also noted that the computational error was zero when the vehicle returned to the departing position indicating that the truncation error cancelled in the opposite direction.

Summary

The total program consisted of 1768 instruction words on seven 4001 ROM chips broken into an executive routine and fifteen subroutines. Two 4002 RAM chips are required to store the data and variables used in this program. The total computational time required for one navigational cycle is between 36 and 80 milliseconds depending on the navigational mode used. The computational error developed by the navigation program from error bound analysis represents only .1 per cent of the total error. The program used a sixteen bit fixed point variable which allows it to accept inputs up to 2047.99 knots with an accuracy of $\pm$ .0625 knots. The total cost of the one CPU chip, seven ROMs, and two RAMs used by the system is $95.00.

The results of this design study indicated that a microcomputer is both fast enough and powerful enough to handle the complex task of navigation. It is concluded that many of the dedicated computational tasks being done by large general-purpose computers can be done by microcomputers.

References

1. Adams, J. R. and others, The Preliminary Design of a Long Range Navigation System for Tactical Aircraft, Group Thesis Project 0910, United States Naval Postgraduate School, Monterey, 1970.

2. Altman, L., "Single-Chip Microprocessors Open Up a New World of Applications," Electronics, V. 47, p. 81-87, 18 April 1974.

3. Busacker, R. G., and Saaty, T. L., Finite Graphs and Networks: An Introduction with Applications, McGraw-Hill, 1965.

4. Cushman, R. H., "Understanding the Microprocessor is No Trivial Task," EDN, V. 18, p. 42-49, 20 November 1973.

5. Cushman, R. H., "Understand the 8-bit P: You'll see a lot of it," EDN, V. 19, p. 48-54, 20 January 1974.

6. Cushman, R. H., "Don't overlook the 4-bit μP: They're here and they're cheap," EDN, V. 19, p. 44-50, 20 February 1974.

7. Cushman, R. H., "The Intel 8080: First of the Second-Generation Microprocessors," EDN, V. 19, p. 30-36, 5 May 1974.

8. Cushman, R. H., "Microprocessors are rapidly gaining on Minicomputers," EDN, V. 19, p. 16-20, 20 May 1974.

9. Holt, R. M., "Current Microcomputer Architecture," Computer Design, p. 65-73, February 1974.

10. Intel, MCS-4 Micro Computer Set, 1973.

11. Intel, MCS-8 Micro Computer Set, 1973.

12. Kayton, M., Fried, W. R., and others, Avionics Navigation Systems, Wiley, 1969.

13. Keele, R. V., Microprocessor Trade-Off Study for Project 2175, Tentative and Unpublished NELC Technical Note, San Diego, California, 6 March 1973.

14. McCracken, D. D., and Dorn, W. S., Numerical Methods and FORTRAN Programming, Wiley, 1964.

15. McCracken, W. L., Design Study of an Avionics Navigation Microcomputer, Thesis, United States Naval Postgraduate School, Monterey, 1974.

16. Naval Air Development Center, Navigation FP-106, 1973.

17. Naval Air Systems Command 01-85 ADF-2-10.1, Integrated Weapons System Theory for A6E Aircraft, 1971.

18. Ogdin, J. L., "Survey of Microprocessors Reveals Limitless Variety," EDN, V. 19, p. 38-43, 20 April 1974.

19. Patrol Squadron Thirty-One, P3C Electronic Systems Operational Training Study Guide, 1970.

20. Reyling, G., "Performance and Control of Multiple Microprocessor Systems," Computer Design, p. 81-86, March 1974.

21. Schultz, G. W., Holt, R. M., and McFarland, H. L., "A Guide to Using LSI Microprocessors," Computer, p. 13-19, June 1973.

22. Sperry Rand Corporation, P3C Operational Program, 1970.

23. Weissberger, A. J., "MOS/LSI Microprocessor Selection," Electronic Design, p. 100-104, 7 June 1974.

AN ITERATIVELY STRUCTURED INFORMATION PROCESSOR

Gerald R. Kane
University of Tulsa
Tulsa, Oklahoma

Summary

There exists a widely held belief that the full potential of LSI technology will be realized by arrays of logic circuits. A novel machine architecture is proposed that consists of a uniform array of identical cells with the property that the array executes a high-level programming language directly. Some of the more important features of such machines are their inherent abilities to sustain concurrent processes and to maintain efficient information storage. A detailed simulation of a language machine in its architectural style is presented.

1. Introduction

For some time a widely held belief has existed that integrated circuit technology will increase to the point that large iterative arrays of logic circuits will become practical. To this end, many cell specifications have been proposed [1], and synthesis and analysis techniques have been developed for specifying these arrays.

Cell complexity separates cellular logic into two not-so-distinct groups. The term 'micro-cellular' is applied to those logic arrays whose basic cell configurations are on the order of complexity of a J-K flip-flop. These cells are often combinatorial, and the arrays employ micro-program techniques to realize various functions. The 'macro-cellular' arrays have more complex cell structures. It is doubtful that micro-cellular logic arrays will achieve any degree of acceptance before macro-cellular representations become effective. This principle can best be stated--complex cells are complex and simple cells are complex, too. The macro-cellular representations have the advantages that cell functions are more readily assimilated on a large scale, and array customization through either manufacturing processes or stored program techniques is not an essential part of the design.

It would seem reasonable that cellular information processors could be constructed to execute higher-level languages directly. By combining the system hardware and software into one unit computing complexity and cost should be reduced; the macro-cellular array appears to be a most reasonable way of achieving this end. The central idea of this paper is that it is possible to exploit the geometry of an iterative machine architecture (its structure) to achieve some degree of efficiency when dealing with languages that are structure oriented.

2. Simple String Processor - A language for an iterative language machine.

The simple information processing language described here is a string rather than a list processing language. The syntax for the language S^{2}P (Simple String Processor) is given in Table 2.1. The semantics of the language are for the most part self-explanatory. The program consists of a set of strings identified uniquely by name. Strings may be decomposed in two ways by the use of the suffix operators head and tail:

label ABC:XYZ≠
ABC head = X
ABC tail = YZ

and larger strings may be formed by catenation:

ABC cat ABC = XYZXYX
ABC tail cat ABC head = YZX

The replacement or definition of a string may be effected by the use of the assign statement. The only relational operator provided is the equal sign. It is the function of this operator to compare two strings of arbitrary length. The if-then constuct causes the then clause to be executed only if the string comparison is equal. Control transfer to a named string is by way of the goto statement. These three simple statement types--assign, if and goto-- comprise the language. Input/output will be ignored.

Methods of implementing a simple language like this on a conventional machine are well known [2]; the reader is invited to recall such methods and contrast them with the iteratively structured approach. Consider now a linear array of cells each containing a single symbol. Strings are formed by storing the symbols sequentially. Unused strings will be conveniently left nameless and a dynamic "garbage collection" feature will be provided to keep the array tidy by removing unnamed strings. A good garbage collector is of such importance that its operation will be discussed first.

The garbage collection algorithm consists of locating all unnamed strings in the array and erasing them. Named strings are then moved to the left so that in a completely garbage collected arrangement only named strings appear on the left and empty cells appear on the right. It is important to realize that the garbage collector and string processor operate concurrently. The language is designed so that these two processes do not interact; there can be no conflict of interest so that both operate at the same level of priority.

Referring to the syntax of Table 2.1, a named string <NAMED STRING> consists of:

label <NAME> : <STRING> ≠.

A simple regular expression for a string is then

A [B] C [D]° E

where

A = label
B = alphanumeric
C = :
D = any symbol
E = the string terminator ≠.

The string is just those symbols next to the colon with the appropriate boundary symbols. From the right of the colon the regular expression [D]° E describes the string and looking to the left [B]A describes the string. Hennie [3] indicates that any regular expression can be recognized by a linear array of cells with information flow in only one direction. The garbage collector must recognize not only the end of a suitable expression but its extent as well; bilateral signal flow is required for identification of the garbage cells.

The operation of the processor is best understood after a brief description of the Lee intercommunicating cells [4]. Each of the Lee cells contains memory for a symbol plus an activity bit and sufficient logic to interpret commands given the memory by a control element. These commands include:

Reset - Set all activity bits to 0
Right - Propagate activity bits right
Match - All cells with matching activity/symbol bits set activity bits to 1 otherwise reset.

TABLE 2.1

Syntax for S^2P

<ALPHANUMERIC> :: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<SYMBOLS> :: = <ALPHANUMERIC> | cat | head | tail | label | if | equals | then | end | ;

<NAME> :: = [<ALPHANUMERIC>]

<LABEL> :: = [label <NAME> :]

<STATEMENT> :: = <GOTO> | <ASSIGN> | <IF>

<STATEMENT LIST> : = <STATEMENT> [;<STATEMENT>]°

<GOTO> :: = goto <NAME>

<ASSIGN> : : = set <NAME> to <EXPRESSION>

IF :: = if <EXPRESSION> equals <TERM> then <STATEMENT LIST> end

<TERM> :: = <NAME> | '<STRING>'

<EXPRESSION> :: = <TERM> | <EXPRESSION> head | <EXPRESSION> tail | <EXPRESSION> cat <EXPRESSION>

<NAMED STRING> : : = <LABEL> <STRING> ≠

<PROGRAM> : : = [<LABEL> STATEMENT LIST> ≠] [<NAMED STRING>]° ≠

Where [ ] is one or more occurrences of the bracketed item
[ ]° indicates an item that occurs zero or more times.

Figure 2.1 indicates the required actions for locating a string named A in the Lee memory. First the array is reset. Then all the labels are identified by the match request 0/label. The activity bits are passed right and a match request for 1/A is made. This cycle of the propagate right-match symbol is repeated until the name is exhausted. A propagate right followed by a match request 1/: will tag with one activity all those strings (there may be more than one) with the desired name. In figure 2.1 this is the string A. The proposed processor differs from the Lee memory in two important respects: (1) the cells are able to reflect different degrees of activity and (2) the sequential commands that direct the memory come from within the memory itself as opposed to an extemal director. A machine of this kind has been proposed by Sturman [5]. The improvement offered by the S^2P machine is its use of a high level language as its machine language.

Commands are transmitted to all cells via the complex signal bus that carries information in the form of a complex symbol from either the language alphabet or one of the augmented symbols ø, u, #, ‡ and □ which are used for special commands. The sequence of control is provided by an intra-cellular signal Z (which effectively modifies the state of the cell to the right).

Any cell containing an alphameric symbol will output its positive activity and its symbol to the complex symbol bus and then propagate its activity and the Z signal right. All cells interpret the signal on the complex bus in the following manner: (1) activity on the bus is zero then the condition is idle, i. e. a NOP command, (2) activity is less than zero (-1, -2, -3) then the cell symbol is compared to the bus symbol and a match occurring propagates the absolute value of the bus activity to the right, (3) for activity 1, 2, 3 both cell activity and symbol must match the complex bus symbol activity, (4) activity -0 use the bus symbol to direct further action.

The goto and set cells function identically. Both initiate a search for a name using 1 activities in the matching process. Encountering a ; cell while propagating 'one' activities causes a transfer of control. The to cell upon receipt of the Z signal outputs -0/: to the bus in order to create a pseudo-colon and then outputs -2/label to begin the search for the name in the replacement string.

The set sequence requires a copy operation. The easiest way to construct a new string for the set operation is to copy the new name ending it with a colon and then copy the appropriate expression following the colon. This simple scheme will fail when the string to be replaced is involved in the expression or the replace string already exists. Two things must happen when a name has been copied: (1) a pseudo-colon must be generated that can be converted to a real colon at the completion of the assign statement and (2) the 1/colon cell associated with the existing name must be activated so that it can be deleted at the end of the assign. A string without a colon will revert to free storage by action of the garbage collector. The copy operation is accomplished by the special cell □ which copies the symbol from the complex bus and propagates the □ to the cell on the right.

The expression evaluation sequence is straight-forward. Head and tail are suffix operators to simplify the implementation. Consider the case where the expression is simply a name. When the semicolon (statement terminator) is reached the name will have been located and the cell to the immediate right of the name will be +2/:. The semicolon must provide signals to copy the string.

The catenation operator works similarly. First the +2/* cell directs the copy of the named string and then begins a search for a new name. The quote operator necessitates modification of the above scheme. Z incident on the 2/quote causes the cell to retain the two activity, now when the Z signal reaches the 0/; cell, a copy operation is indicated but there is no colon to bypass so the ; (and similarly the cat) uses the 0,2 activity to determine the necessity of skipping a colon. Z incident on +0/quote is an idle condition and thus a way to imbed comments in the memory although such comments must contain a colon to avoid garbage collection. Quote cells cannot pass non-positive activities thus the second quote of a string will terminate the quote process.

The suffix head operator is handled by copying one symbol and then transmitting a +2/ø to the bus; the ø symbol matches nothing so this is a clear activities instruction. The tail operator outputs +2/u to the bus. +2/u is a propagate-2-activities-right as u matches anything. Activity 0 is sent to the right along with the Z signal upon completion of the

operation. Composition of operators is possible--the operators being applied in order closest to the name, e.g. ABC tail tail head will select the third character in the string ABC whereas ABC head tail tail will be ≠.

The conditional statement is a non-essential but useful feature of the language. Another special symbol # must be used to effect the comparison of two strings. The idea is to build the expression represented before the equals and then compare it character by character with the term following. The cell to the right of the # must match any 3 active signal on the bus. In this case the # is propagatedright; otherwise the # is deleted. The command -0/# forces the cell pair # □ to emit a +3/#. The -0/# also replaces # with ≠ thus returning any remaining comparant strings to free space. If the strings are not equal, the # will have been deleted or the # and □ will be separated by one or more characters. In this case the then cell driving the compare will not receive the affirmative +3/# and will propagate -0 and Z to the right. The -0 activity inhibits further communication with the bus until the end cell is reached.

3. Synthesis Techniques for Iterative Language Machines

The design of the S^2P machine based exclusively on the Lee memory is impractical for several reasons. The dependency on a central clock and the operation of the entire array in "lockstep" is particularly naive. In realizing an iterative machine the design engineer must be cognizant of the propagation of information through the array takes a non-zero interval of time. The description and simulation of an unclocked S^2P machine is provided in this section.

It is helpful to imagine the interconnecting structure of the asynchronous or unclocked S^2P machine to be a simple multi-rail transmission line where the propagation time is directly proportional to the inter-cell distance. The cells on the bus are at fixed constant spacing and the cells themselves make decisions in zero time; this is to say that all of the delays are lumped into the transmission characteristics of the bus. The dual approach of lumping all the delays in the cells with no propagation delay between cells more nearly represents the approach taken in the design (refer to the appendix) but is conceptually more difficult.

An APL program was written to simulate the machine described; the illustrations provided are taken from that simulation. Each cell has the capabilities: (1) of examining the bus at a "window" in the region of the cell and changing states according to the bus command, (2) of transmitting to the bus in a vacant window, and (3) of directing the cell to its right.

The branching operation requests an associative match for a particular label and then transfers control to that label. Spatial separation on the bus is used to order the constituents of a label. The bus is terminated at each end by its characteristic impedence so that signals reaching the end are absorbed. The process of locating strings in the set-to sequence is equivalent to the search for branch names.

In a clocked machine the copying of a string may be directed character by character until the string is exhausted: this is a reasonable approach when the amount of time required to retrieve a single character is a small constant. When the time required to access a character varies with the position of the string it is reasonable to attempt to find a burst mode of character transmission. This is accomplished by adding a command to the bus repertoire which directs a located string to transfer itself to the bus. The difficult thing with this burst transfer mechanism is to decide when the transfer is complete. The time required to make this decision is related to the size of the machine and the relative locations of the accessed string and the directing string. Another single rail bus with the same transmission characterics as the symbol bus is provided to sense the "size" of the machine. By placing a positive pulse on the bus and terminating the line appropriately the cell may sense completion of the transfer by awaiting the reflected pulse. The bus termination must be chosen in such a way that the pulses are reflected only once. In this way it is only necessary for the cell to count two incident, reflected pulses. It is not necessary for the control pulses to travel to the extremities of the array. The structure of the machine requires that the copy cell, □ in the examples that follow, be the extreme right cell in use. This means that it is not necessary to search further to the right for a name. Similarly the cell containing the found name is the furthest cell to the left that need be searched. The cells may be designed to reflect the sense pulses conditionally.

The following snapshots (Figure 3.1 A to F) are successive states of the S^2P machine. The APL symbols used include ⌊ for label, ∇ for set, → for goto, and ← for to. The example is a program string that reverses a string named XYZ. No attempt is made to illustrate the if-then mechanism as it does not represent any new ideas beyond the goto and set-to constructs. The term cycle refers to the length of time it takes a bus signal to move between adjacent cell sites. "Cycle" is taken from the familiar analog in the design of classical machines.

The first snapshot with Z in the ∇ cell initiates the search for the label. The information packet +1/Z +1/Y + 1/X -1/⌊ moving to the right is searching for the name and a similar packet is moving to the left. The differences in spacing of the left and right information trains are due to Doppler shift.

The ← (to) cell must direct two similar operations on the bus. The string that is to be replaced and a new string with the same name must both be tagged. This is done by the +0/:p and the +0/← cells respectively as set by the -0/: bus directive.

Once the -0/: directive has been placed on the bus the command string can begin expression evaluation. In this case the search is for the string with the same name. The search for the original set name has progressed sufficiently by snapshot 5. The +1: marks the search string; by the next frame (snapshot 6) the string has been tagged for ultimate deletion.

Before this matching operation has taken place the command string has provided signals to find the replacement string and to locate its tail. The frequency change (separation between the +2: and +2/u bus directives) is due to the desire to maintain some separation in left going commands. Even before the set string has been located the machine is paused, waiting the copy of the tail of XYZ as indicated by the +0/*P cell as seen in snapshot 5 and following. The graphic symbols ⊥ and ⊤ are used to represent originated (positive) and reflected (negative) pulses on the control bus. The -0/* directive will initiate the required copy. Meanwhile on the right side of the array, the new name XYZ is being copied as shown in snapshots 5 and 6. The 'M' state is used to inhibit the copy function for one cycle to avoid copying the bus contents which have already been copied into the cell on the left. This lockout technique with the bus caused by the cell change

directives is necessary to preserve timing relationships. This relationship may seem awkward in the bus configuration as described but when the cell delay dominates as in the realization presented in the appendix such an interlock is easily achieved. The cell must have the power not only to sense bus activity but the velocity of such a disturbance as well.

By snapshot 7 some 37 command cycles into the execution of the string, the new name XYZ+ has been formed at the right of the array and the locating of XYZ< is in progress at the left.

One frame later the -0/* directive has enabled the copy cell □to copy 2 active bus symbols (the copy cell is in the 'P' state). Notice also the negative reflection on the control bus caused by the copy cell. By this time the tail operation has marked the +2/Q cell to eventually begin the copy operation.

The 'C' state is shown directing the placement of the to-be-copied symbols on the bus. The operation terminates on the ≠. In the state configuration of snapshot 11 the array is quiescent; the +0/*P will next become active after receiving the two τ pulses. Some time later (cycle 85) the ;SRQ packet has been copied and the search for the head of XYZ is in progress. This condition persists without complication until the head operation takes place at the left of the array in snapshots 15 and 16. First the head of XYZ is marked by the +2/.P activity state. This permits two things to happen. If the program request is of the form of a single head followed by a copy request then the cell will place only its contents on the bus and revert to its formed quiescent state. Should an additional request such as tail follow the head request then the marked cell will retum immediately to the idle state as the tail of a single character is null.

Having placed the +2/P on the bus, the array waits for the copy operation to take place. The ; cell continues to count negative control bus pulses. The process of searching for a new name begins in snapshot 21. The name is copied while the goto proceeds and is returned to the array for garbage collection by terminating it with a ≠. The +0/:P cell that has waited patiently for annihilation is finally satisfied by the receipt of the -0/≠. The new string XYZ was formally inaugurated when the + cell received the -0/≠ bus directive between snapshots 19 and 20. The program forces a transfer to ABC and would remain in this loop indefinitely. Three garbage strings are created in each pass through the loop and are consumed by the garbage collector.

The action of the garbage collector in S^2P is indicated in snapshots 21and following. The isolation phase quickly locates the dummy label created by the branch instruction and the compactor phase is initiated almost as rapidly.

4. Summary and Conclusions

This paper has presented a novel architecture based upon the concept of direct execution of a higher level language. A simple string language was introduced and an example machine for that language was derived. It is important to realize that the resulting machine was itself a simple string of cells; the structure of this language is reflected in the hardware. It can be demonstrated that necessary features such as arithmetic processing and macro facilities can be added to the iterative language machine. An important result is the ability of such a machine to support parallel processing. This is demonstrated by the simultaneous processing and garbage collection in the S^2P machine but may be extended to multiple processing streams [6].

Current technology is leading toward the feasibility of individually owned computers. The continued development of Von Neumann based machine architectures and the software support for them in the form of compilers and single process operating systems may be misdirected. The need for processors with minimal software support is not in the distant future; it is in the present.

Appendix

An asynchronous network is required to simulate the actions of a transmission line as required in the development of the S^2P machine. The design goals include:

1. Facilities for the propagation of information to either left or right

2. At any cell site information from only one direction may be handled at a time; in a situation where signal flow is both left and right preference is given to those signals moving to the left although every directive reaches every cell on an interleaved basis.

3. The order of the symbols on the bus is preserved; the spacing between symbols is irrelevant

4. Cell directives and bus directives remain in lock-step

5. Direction of motion is preserved and such information is available to the cells

6. It is possible to selectively terminate the bus at any cell site. Selection features include--but are not limited to--bus contents, cell contents, and direction of motion.

Extensive use is made of the ready-acknowledge form of commands. A command produces a ready signal when it desires to initiate a sequence. A completion signal then responds with an acknowledge. The double bussed RDY-ACK signal uses a form of transition logic [7]. If the signals are different, then the line is active; if they are the same, the idle condition prevails. The generation of RDY merely changes the level of one of the lines while the generation of ACK changes the other. The XOR function will then sense the appearance of a ready signal.

The development of the bus network must pay proper attention to the nature of the asynchronous control circuitry. All unnecessary delays should be eliminated and the dependency on transition time minimized. The bus register will consist of a number of bits sufficient to include the width of the bus and some error checking circuitry. Since signals may be propagated in both directions, it will be necessary to load the bus from two sources. It is assumed that the data propagation rate equals the control signal rate so that by the time the actuating ready signal is received, data must be available at the register. A double D flip-flop has been designed such that it can be loaded from two different data paths by two separate clocks. The restriction placed on the clock lines is that at most one clock be high at any time; data transfer is on the leading edge of the clock line--this rising edge trigger operation is assumed for each variety of flip-flop in the design. The set and reset lines are normally high. At most only one of R, S, C_1, C_2

may be high for the flip-flop states to be deterministic. The use of this flip-flop is illustrated in figure A1. The RDAT and LDAT lines are data from the right and left cells respectively; RXFR and LXFR are clocking signals to the flip-flop. BXFR is a transfer signal to a hidden register that is used to allow the crossing of data streams from the left and right. BDAT and SETBUS are used within the cells to write data on the bus.

In order to handle bi-directional data flow a cell may not receive data from the right unless it is not busy and it has transferred its contents to the left--as indicated by the NXOR gate in the lower right corner of figure A2--or the buffer register to the right has been loaded--as indicated by the BUFFILD signal. The chain of a pair of ready-acknowledge signals at the upper left permits the interleaving of signals from the left and right. In fact, the chain forces such a condition if possible so that a long string of closely packed symbols from one direction or the other cannot hang the bus. The remaining RDY-ACK pair is used to transfer data from the left. The delay element is necessary to assure that the hidden buffer register has stabilized before the bus register is clocked from the left. The purist can be satisfied by using an additional RDY-ACK network in the place of the delay. The internal signal DONE is cleared on any clock of the bus register and set when the symbol has been interpreted. LXFR, RXFR also set intemal motion flip-flops so that the direction of propagation is maintained. Toggling Q_2 and Q_4 with PROPL and PROPR appropriately will cause propagation of the symbol; by changing the states of the motion flip-flops the bus may be effectively terminated anywhere throughout its length.

References

[1] Robert C. Minnick, "A Survey of Micro-Cellular Research" JACM XIV-2 APR'67.

[2] Stuart E. Madnick, "String Processing Techniques," CACM X-7 JUL '67.

[3] Frederick C. Hennie, III, Iterative Arrays of Logic Circuits, MIT Press (1961).

[4] C. Y. Lee, "Intercommunicating Cells, Basis for a Distributed Logic Computer," FJCC Proceedings 1962.

[5] J. N. Sturman, "An Iteratively Structured General Purpose Computer," IEEETC XVIII-1 JAN '68.

[6] Gerald R. Kane, "Iteratively Structured Information Processors," Ph.D. dissertation, Rice University (1972).

[7] Y. H. Chang, "Transition Logic and a Synthesis Method," IEEETC XVIII-2 FEB '69.

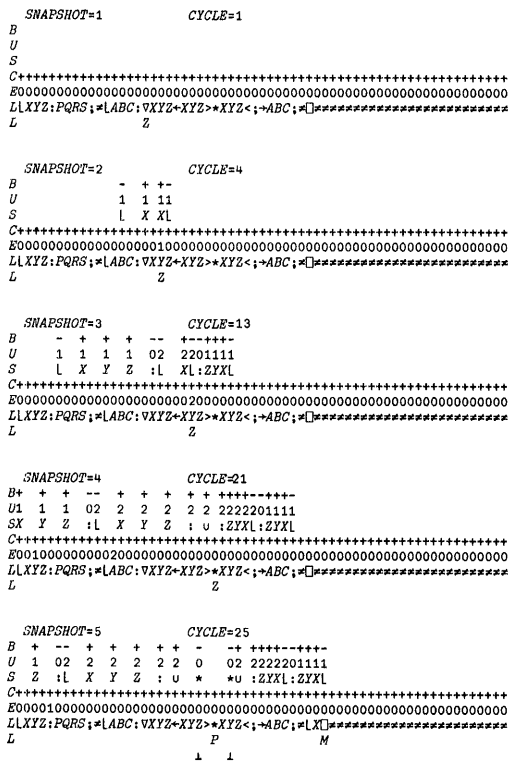


Figure 3. 1 A

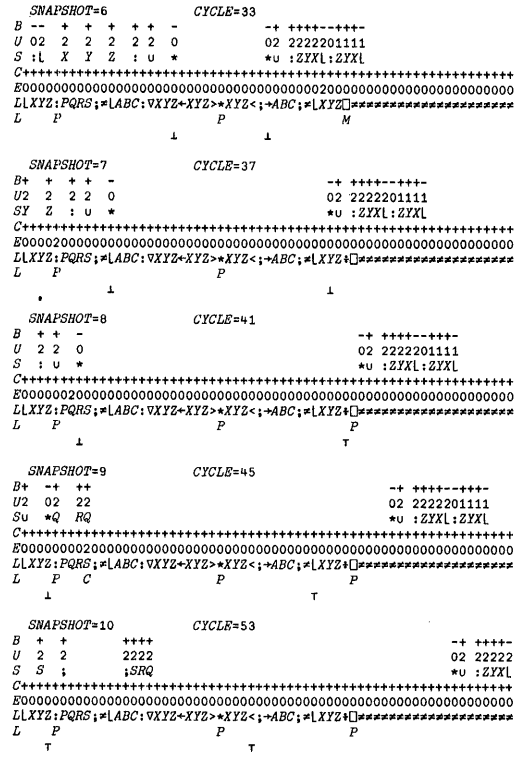


Figure 3. 1 B

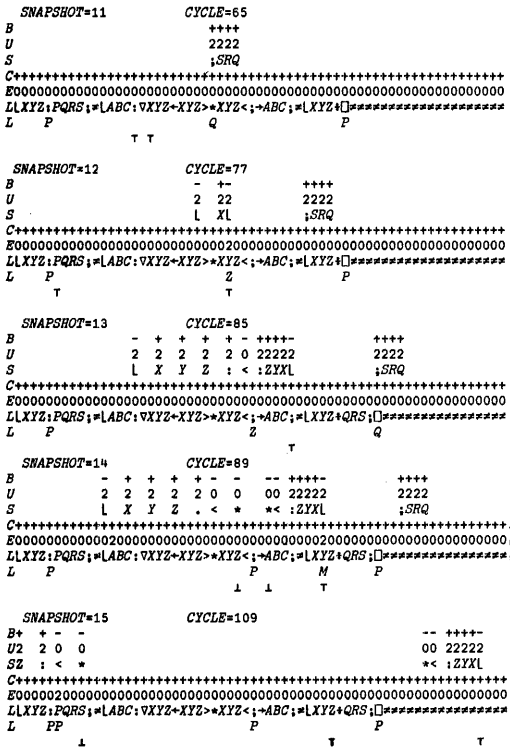


Figure 3. 1 C

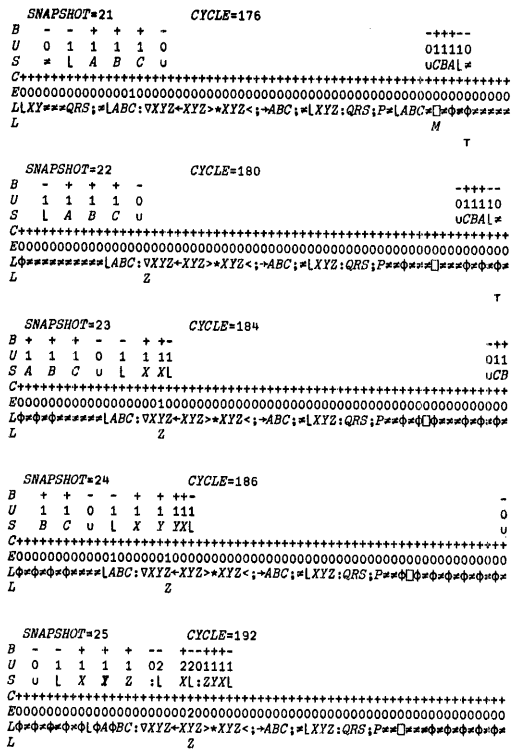


Figure 3. 1 E

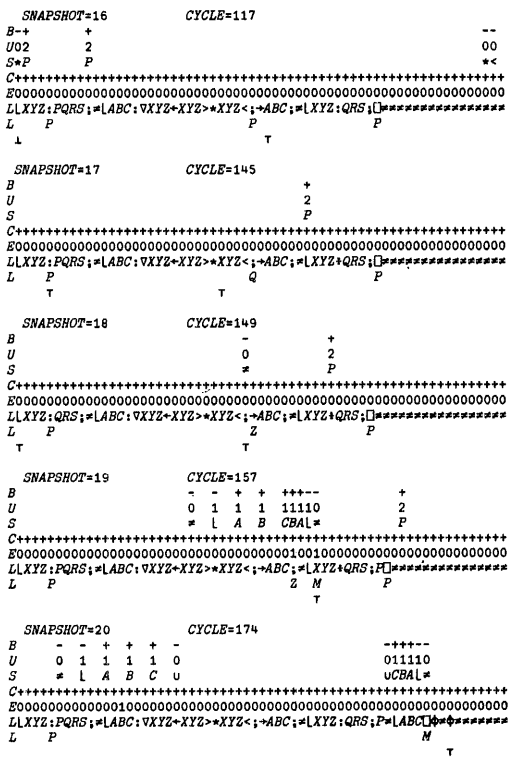


Figure 3. 1 D

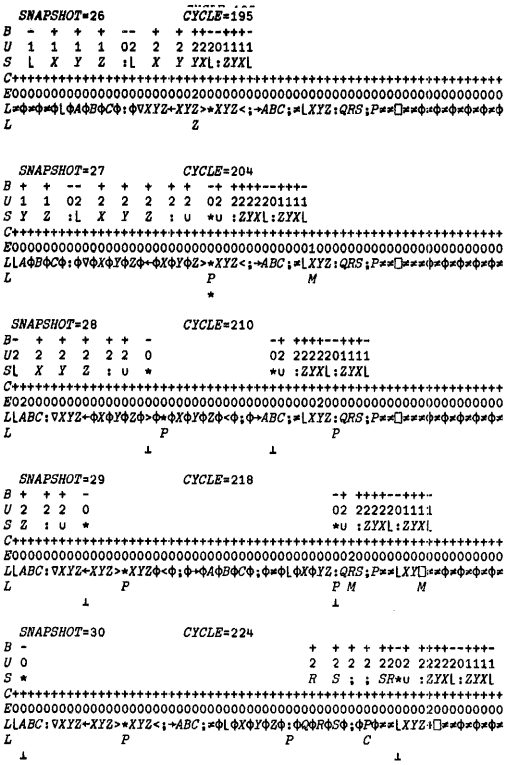


Figure 3. 1 F

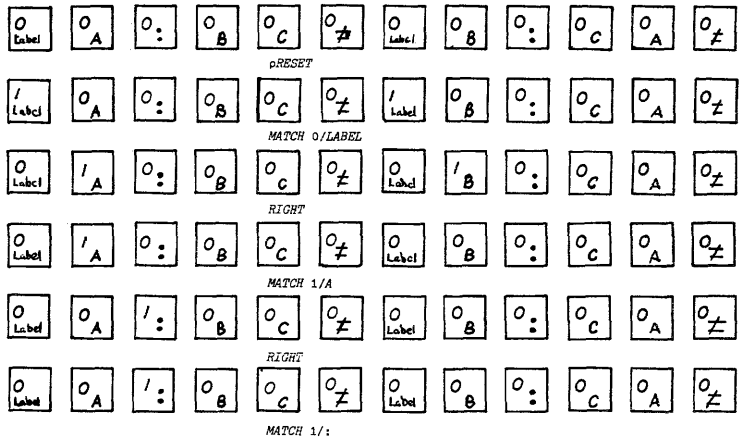


Figure 2.1
Locating a String Named A in a Lee Cell Memory

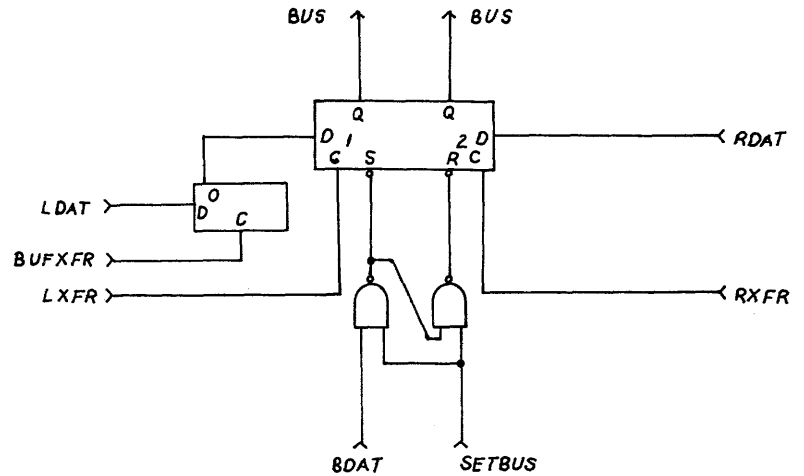


Figure A.1
Bus Register for an ISIP

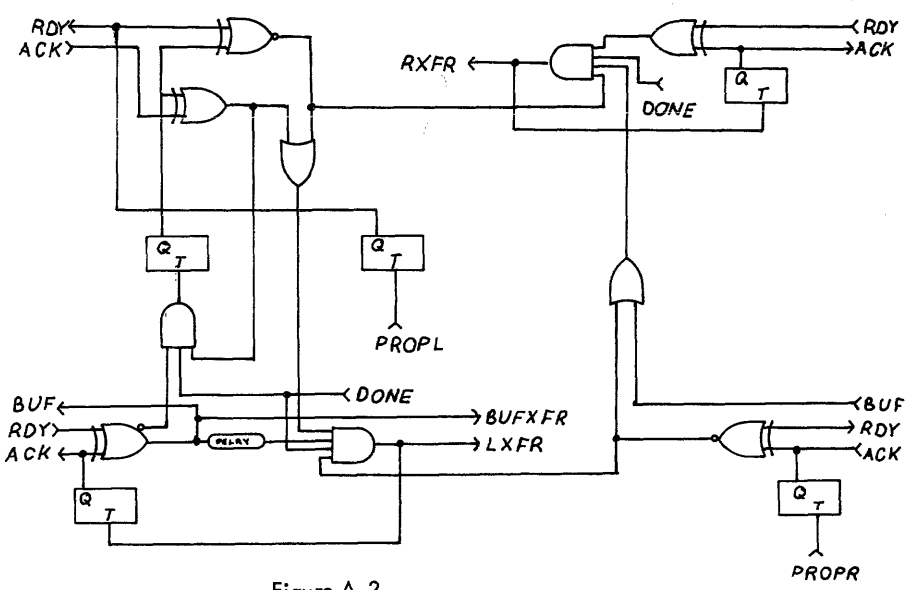


Figure A.2
Bilateral Asynchronous Bus Network

Hardware-Software Interactions in SYMBOL-2R's Operating System

H. Richards, Jr. and A. E. Oldehoeft*
Computer Science Department
Iowa State University
Ames, Iowa

Summary

This paper describes the chief attributes of the Virtual Processor placed at the disposal of each user of the SYMBOL-2R time-shared multiprocessor system, and the mechanisms by which SYMBOL's hardwired operating system manages processing-mode transitions for individual Virtual Processors and allocates hardware resources — processors and memory space — among competing Virtual Processors. It describes the provisions by which the unusually high-level capabilities of the hardware are augmented by software, and contrasts the structure of the software component of the operating system with that of the hardware component. Finally, it describes the hardware/software partition of resource-allocation functions, in which allocation policies are controlled by software and executed by hardware.

Introduction

The SYMBOL-2R computer is an experimental time-shared multiprocessor system in which many of the functions normally found in software have been implemented in hardware. These hardware functions include the compilation of a high-level language, management of a virtual memory to support dynamically varying data structures, system supervision, and various other functions common to third-generation systems. Without recourse to system software, SYMBOL offers time-shared processing of interactive jobs, from initiation through loading, editing, compilation, execution, I/O, and termination, including all the necessary paging and processor-multiplexing operations. Software is required only to handle certain exceptional conditions acceptably, and to adapt resource-allocation policies to varying operating conditions.

Overview of the SYMBOL System

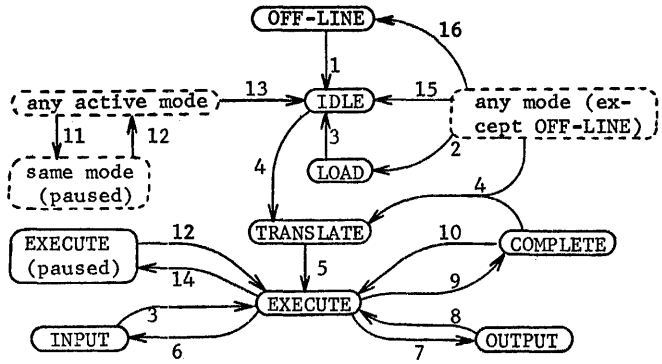
Since one of SYMBOL's novel properties is its ability to deliver an unusually high level of service with no system software, we first describe the appearance it presents to its users in this software-free configuration.

Virtual Machines

SYMBOL makes available to each user (up to 31 users at a time) a Virtual Machine, with which he communicates via a special SYMBOL terminal.** Each such terminal is equipped with at least a typewriter, a control-key console, and a set of processing-mode indicators.*** Using the typewriter and control keys, the user can load, edit, and cause to be executed a program expressed in the high-level SYMBOL Programming Language (SPL).[7,8] Figure 1 summarizes the Virtual Machines' processing modes, and the means by which the users control them. In this software-free configuration, the Virtual Machines operate in complete logical independence of one another.

Virtual and Actual Processors

We refer to the active data-transforming element of each Virtual Machine as a Virtual Processor (VP).



MODE TRANSITIONS

1. INITIATE key
2. LOAD key
3. end-of-record character read
4. RUN key
5. translation successfully completed
6. INPUT statement encountered in execution
7. OUTPUT statement encountered in execution
8. end-of-record character output, or CANCEL OUTPUT key pressed
9. execution successfully completed
10. RESTART key
11. PAUSE key
12. CONTINUE key
13. processing error encountered
14. PAUSE statement encountered
15. CLEAR key
16. TERMINATE key or communications failure

ACTIVE PROCESSING MODES

LOAD: source text is accepted from terminal's input device. Interactive editing provided includes insertion, deletion, and display of selected text (designated by characters or lines), and substring searching in both forward and backward directions.

TRANSLATE: source text is syntactically analyzed and object program is generated.

EXECUTE: object program (generated from source text in TRANSLATE) is executed.

INPUT: one record (character string) is accepted from the terminal's input device and assigned to a variable. Editing is provided as in LOAD mode.

OUTPUT: one or more records are transmitted to the terminal's output device.

Fig. 1. Virtual-machine Processing Modes and Mode Transitions

*The work reported on here was supported in part by the National Science Foundation under Grant GJ-33097X.

**SYMBOL can accomodate terminals of other types, but this requires software.

***Other I/O devices — card readers, line printers, tape drives, and the like — may be added. At any given moment, only one device of a terminal's complement can be actively transmitting or receiving data.

SYMBOL's hardware provides for 32 VP's, each of which receives intermittent processing service from one or another of five Actual Processors (AP's), according to its processing mode. The VP's are permanent entities, in the sense that each has hardware dedicated to its support (including a fixed portion of core memory for its Context Block). The AP's are specialized hardware modules[1,3,4,5,6] connected to one another and to two other modules, which provide supervisory and memory services, as shown in Fig. 2.

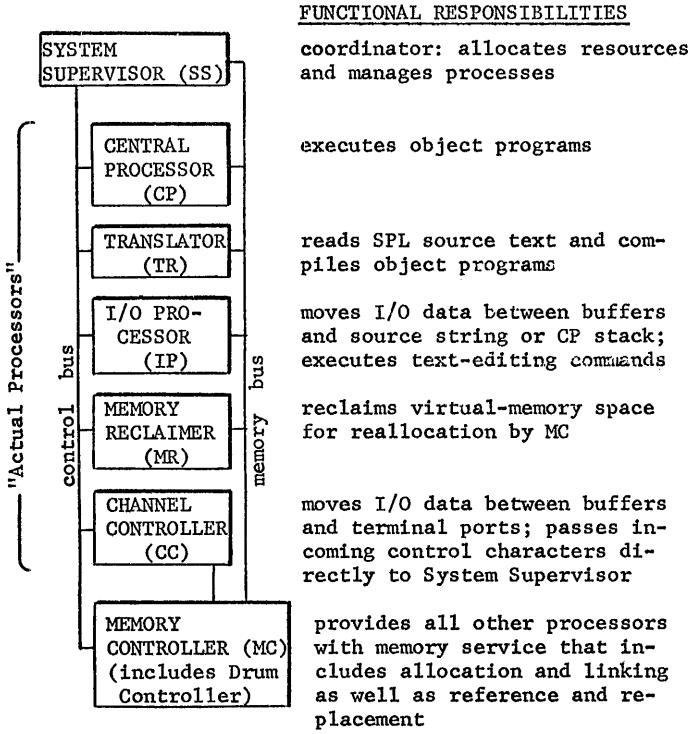


Fig. 2. The SYMBOL System's Hardware Components

Each terminal supported by SYMBOL is permanently connected to one of the VP's, up to a total of 31. The remaining VP has no I/O capability, and is used exclusively for system software.

Virtual Memory

The memory service provided to the other hardware modules by the Memory Controller (MC) is unusually elaborate, in that it includes allocation, linking, and garbage-collection features as well as conventional reference and replacement.[9] It is based on a virtual-memory address space of 16 million 64-bit words, partitioned into 64K pages. Pages are not shared; at any instant, each page in active use belongs to exactly one of the Virtual Processors. Pages do circulate over time, however; a page may be used by one VP, cleared, and returned by the Memory Reclaimer (MRP) to the system's pool of available pages, and later allocated to some other VP.*

A processor obtains access to memory by transmitting a request over the memory bus (witholding its request until the Memory Controller is free and no higher-priority processor is making a request). Specified in the request are such parameters as the type of operation to be performed (one of the 14 provided), VP number, address, and datum to be stored. The transaction is concluded by the Memory Controller's reply, in which it transmits to the requestor such items as succeeding address, datum fetched, and completion code (e.g., success, error, or page-fault).

*This circulation is not to be confused with the traffic of pages between core memory and drum, which is an altogether separate process (described below).

Actual Memory

The physical storage supporting SYMBOL's virtual memory consists of 26 pages ("frames") of magnetic core memory and 4K pages of magnetic drum (leaving 60K pages currently unimplemented). Each core frame is provided with an associative register that is loaded with the virtual-page number of its frame's contents and used by the MC in transforming the virtual addresses submitted by processors into the core addresses necessary for performing the required accesses.

When an AP requests access to a page not contained in some core frame, the MC returns a page-fault reply. The requesting processor then suspends its processing and reports to the System Supervisor (SS) for possible assignment to some other VP; it will later be reassigned to the original VP after the necessary page has been loaded into core memory.

In addition to the 26 frames used for buffering virtual-memory pages, SYMBOL's core memory contains 1.5K words used for other purposes. Part of this space is permanently allocated to 24-word Context Blocks (one for each VP) in which are stored such data as 1) processor-state information (during intervals when the VP is not being serviced by an AP), 2) memory-management data concerning the virtual-memory space allocated to the VP, and 3) process-control informations such as the VP's current processing mode, its blocked/unblocked status, and its I/O buffers' addresses. An additional region of 0.5K is allocated to I/O buffers, at the rate of 16 words per VP. The remainder of the 1.5K-word region contains information concerning the system as a whole, such as the Reserved Word Table of the Translator, the available-space information used by the Memory Controller, and the Software Entry-point Table and other data of use to the SS.

NOTE: The allocations described above are those of the original design. Currently, at Iowa State, SYMBOL is supporting only 16 VP's, of which a mere three are connected to physical terminals. The Context-block space of the unsupported VP's is used for I/O buffers for the three terminals, and the 0.5K originally intended for I/O buffers is used instead for buffering virtual memory, raising the total number of core frames to 28.

The SYMBOL Operating System

For the purposes of this paper, we consider the essential responsibilities of SYMBOL's operating system to be the following: 1) process management, which consists chiefly of effecting processing-mode transitions for each Virtual Processor as it passes from one processing stage to the next; 2) provision of auxiliary services, in addition to those provided by the hardware, that the user perceives as capabilities of his Virtual Machine; 3) resource allocation, i.e., allocation of Actual Processors to implement the system's Virtual Processors, and of core memory to implement virtual memory. This section deals with the first two of these topics; the third is deferred to the following section. We intend not to describe these functions in great detail, but simply to illustrate the partitioning of functions into, and the flow of control between, hardware and software.

A substantial part of the SYMBOL Operating System is implemented in the hardwired System Supervisor (SS). This part consists chiefly of functions characterized by high duty-factor or short response-time requirements, with certain additional functions included to give the system its unusual software-free capabilities. The software component of the operating system is implemented in programs expressed entirely in SPL and executed by the same Central Processor (CP) that executes

users' programs. It not only augments the capabilities of individual Virtual Machines, but also provides certain functions that enhance the performance of the entire system. Since the software-implemented functions are infrequently invoked, and are free from severe response-time requirements, the software is paged in only upon demand.

The Hardwired System Supervisor

The responsibilities of the SS (as distinguished from those of the Operating System as a whole) consist of 1) Context Block transformations effecting processing-mode transitions, 2) resource-allocation decisions, subject to software-controlled parameters (described in the following section), and 3) invocation of software for functions beyond those provided in the hardware. A fairly detailed description of the SS, published heretofore,[10] relieves us of any need to describe features of the SS other than those that bear directly on the hardware/software interface.

We view the SS as a cyclic process, as illustrated in Fig. 3, that waits in a stationary rest state until stimulated by some externally generated Service Request (SR). Upon receipt of an SR, the SS leaves its rest state and embarks upon a service routine appropriate to the request. Approximately 70 different SR's are serviced by the SS; some typical examples are 1) the Channel Controller reports that a user has pressed a control key on his terminal's control console; 2) some AP signals normal completion of its current task; 3) the Channel Controller reports that it has filled an input buffer; 4) the Drum Controller reports the passage of a drum sector boundary under its read/write heads; 5) some AP signals that it has experienced a page-fault interrupt. Most of the SS's service routines result in some alteration in the busy/idle state of some AP, or some change in the set of VP's awaiting service; on service-routine completion, therefore, the SS proceeds to a common Task Assignment (dispatching) routine before returning to its rest state.

For dispatching purposes, the SS maintains, for each AP, a queue containing an entry for each VP whose current processing mode calls for service by that AP. In Task Assignment, the SS scans each idle AP's queue and, on discovering a dispatchable VP, issues the AP a command to service it.

Most of the SS's service routines are undertaken on behalf of a specific VP, and entail such transformations on the VP's Context Block as 1) initializing the processor-state component of the Context Block, in the course of switching processing modes, 2) marking the VP blocked (after a page-fault interrupt) or unblocked (after completion of a paging task), and 3) swapping I/O-buffer pointers (after a buffer-full or buffer-empty signal). In most cases, the routine ends by deleting the VP in question from some queue and/or adding it to some other queue.

A crucial assumption underlying the design of the SS is that every service routine executes quickly enough that no serious delays result from processing service requests sequentially. This assumption permitted the SS to be designed as a non-interruptible and non-reentrant process (certain routines are segmented, with provisions for suspension and resumption in case of page faults).

An important distinction must be made between two classes of Service Requests. One class, whose members are called Normal Service Requests (NSR's), consists of SR's encountered in successfully processing an ordinary job, and for which the SS provides unique and complete service routines. It is the inclusion of these routines in the SS that allows software-free processing of error-free interactive jobs.

Service requests in the other class are called Exceptional Service Requests (ESR's). Some typical examples are processor-detected errors (e.g., syntax or execution errors in users' programs), processing-time

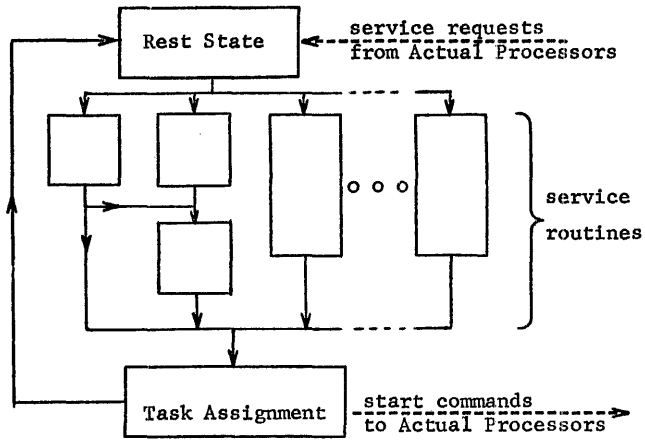


Fig. 3. General Flow of Control in System Supervisor

or memory-space limit violations, and certain terminal control-key signals. The software invoked in response to an ESR can provide such functions as diagnostic messages, login/logout and accounting operations, inter-terminal communications, and file-management facilities.

The Software Component of the Operating System

SYMBOL's operating-system software consists of a collection of programs, unique among which is "VPZ", so called because it permanently occupies Virtual Processor Zero. VPZ, the first program to go into action in response to an ESR, is invoked by the SS in a two-step operation which proceeds briefly as follows. The first step — the SS's immediate response to the ESR — places the VP generating the ESR at the head of the CP queue, with an ESR identification code in its Context Block indicating which of the 120 types of ESR's has actually occurred. There the VP waits for VPZ service along with, possibly, other VPZ requests, as well as VP's demanding ordinary VPZ service.

The second step of the VPZ invocation operation takes place when the SS, searching the CP queue for an executable CP task, encounters a VPZ request. As long as the binary semaphore "VPZACT" indicates that a VPZ task is in process, the SS ignores all new VPZ requests. The completion of a VPZ task, however, clears the semaphore, permitting the SS to assign VPZ to the first VPZ request in the queue. It selects VPZ's entry point from the Software Entry-point Table according to the ESR code, and signals the CP to begin executing. From that point until completion of the task, VPZ contends for CP service and memory space with the other CP tasks (unlike them, it is not subject to time-slicing preemption). Note: a software process can invoke the second step of the operation, and thus "call" VPZ, by inserting an ESR code into its own Context Block.

The semaphore VPZACT mentioned above ensures that VPZ is used only serially, and thus finishes processing each ESR before tackling its successor (a measure necessitated by the lack of code-sharing provisions in SYMBOL). In a timesharing system, such serial processing of independent ESR's would give rise to intolerable delays, especially since ESR processing can involve extensive dialogues between the system and the user/operator. In addition, SYMBOL's hardware supports I/O between each VP and the associated terminal, but does not readily handle I/O for the terminal-less VP Zero. Therefore the bulk of ESR processing is performed not by VPZ itself, but by the other major components of the operating-system software, namely, the Virtual Processor Monitor (VPM) programs. Each VP is provided with its own VPM, which is called into play by VPZ whenever the VP generates an ESR. As Fig. 4 suggests, the capa-

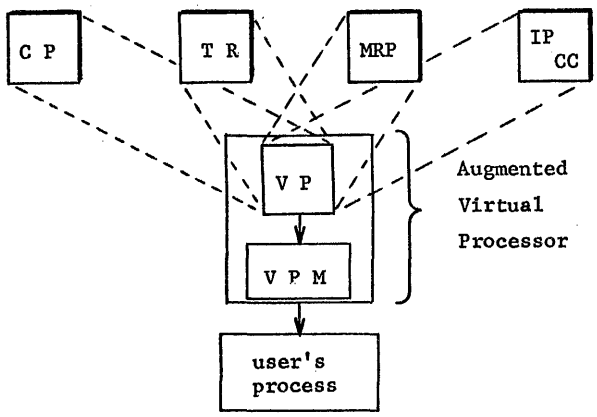


Fig. 4. Augmented Virtual Processor

bilities of each VPM merge, more or less imperceptibly from the user's point of view, with those of the hardware, to create an Augmented Virtual Processor possessing capabilities, such as program-error diagnostics, beyond those of the bare hardware.

The ESR-processing responsibility of VPZ is thus reduced to mere VPM initiation: saving the current Context-block data concerning the user process (in memory locations accessible to VPM for reference and modification) and replacing it with VPM Context-block data, thereby switching control of the VP from the user process to the VPM. At the conclusion of its ESR processing, the VPM "calls" VPZ to perform the reverse switch, restoring control of the VP to the user process. Because the VPM's all run on different VP's, their share of the ESR processing can proceed in (time-multiplexed) parallel, leaving only the relatively minor process-swapping VPZ portions to be performed serially.

Conditional ESR's. By setting certain bits in a given VP's Context Block, one can flag certain NSR's for ESR treatment, thereby substituting VPZ/VPM processing for the normal hardware algorithms on a VP-by-VP basis. For example, INITIATE-key, TERMINATE-key, and TERMINAL OFF-LINE NSR's can be flagged as ESR's in order to invoke such functions as log-in, log-out, and reverification routines. Central Processor I/O shutdowns, which the SS normally handles by switching the VP to the I/O processing mode for IP and CC service, can instead be handled by the VPM in a batch-processing mode. A third example is the CLEAR-key NSR, which is flagged for the duration of VPM processing to protect the pages containing the VPM from being cleared.

Allowing different NSR's to be flagged for different VP's accomodates VPM's that are tailored to the type of terminal equipment they are dealing with. For example, so-called Class-A VPM's are designed to handle the full SYMBOL terminal, with its control-key console and processing-mode indicators. For such terminals, the SS's normal response to a CP normal completion, which is to turn on the "Run Complete" indicator, is appropriate. Class-B VPM's, on the other hand, monitor Teletype-equipped terminals, which lack such indicators; accordingly, CP normal completions are flagged as ESR's for those VP's, which permits the VPM to output a printed message equivalent to the "Run Complete" indicator.

System-software Provisions in SPL

In many present-day systems, restriction of global system-intervention capabilities to authorized processes is imposed during execution by the provision of two mutually exclusive execution states, such as "supervisory state" and "problem state", and interrupting on attempts to execute certain instructions while in problem state. In SYMBOL, by contrast, there is but a single execution state; programs are designated "privileged" or "non-privileged" at translation time. SPL syntax[7] for non-privileged programs (which include all users' programs*) simply provides no means of accessing any part of memory other than that expressly allocated by the system for the program's variables. Restricted SPL syntax, which applies only to privileged programs and procedures, is a superset of the syntax that applies to users' programs; it allows statements that can explicitly access any location in core or virtual memory, using the memory-operation repertoire of the Memory Controller.[8] Another restricted statement allows the program to set up tasks, consisting of queue and core-word modification instructions, to be executed by the SS while the CP is momentarily idle (to forestall potential CP-SS race conditions).

Allocation Algorithms

In SYMBOL, the processor and memory allocation** algorithms are hardware-implemented functions of certain software-controllable parameters. This arrangement provides for fast execution of frequently invoked algorithms, while maintaining some flexibility in controlling the underlying policies. The purpose of this section is to describe the hardware algorithms and the roles of the associated parameters. The reader should bear in mind that SYMBOL's resource-allocation strategies date from 1966-67, and hence have been denied the benefit of more recent work. They are described here not because of any intrinsic merit or efficiency (matters which have yet to be resolved), but in order to illustrate a significant aspect of hardware/software interaction and functional partitioning.

Processor Allocation

The services provided by each of SYMBOL's Actual Processors are subject to demand by as many as 32 Virtual Processors simultaneously. One of the AP's — the Channel Controller — allocates itself by continuously polling all VP's. The others — CP, TR, IP, and MRP — depend on the SS to distribute their services among the contending VP's. For each of these AP's, the SS maintains a queue of VP's awaiting or receiving its services. Whenever a processor becomes available, the SS scans its queue and assigns it to the first VP that is not in page wait (a VP in page wait maintains its queue position). The number of queue positions scanned by the SS in each pass is bounded by one of the software-controllable parameters, namely, the Queue Search Limit (QSL), which thus bounds the size of the "multiprogramming set", i.e., the set of VP's in active contention for Actual Processor service.

The SS distributes AP service among competing VP's according to a preemptive service discipline. The time slice for each VP is determined by its Queue Run Time (QRT) parameter. This parameter is set by the SS to its software-controllable initial value when the VP is first added to a processor queue. At regular intervals, the SS decrements the QRT of each VP that is receiving service from some AP. Whenever some QRT reaches zero, the SS preempts the associated VP, demotes it to the bottom of its queue, and re-initializes its QRT. Since the SS scans processor queues from the top down, the topmost VP's are most likely to receive service, use up their allotments, and move to the bottom (the resulting circulation is illustrated in Fig. 5). A VP's probability of receiving service drops substantially when its time slice expires (see Fig. 6); if QSL

*Users' programs are allowed to call system-supplied privileged procedures.

**Throughout this section, the term "memory allocation" refers to the allocation of core frames by the SS as a part of its virtual-memory implementation responsibilities; it is not to be confused with the allocation of virtual-memory space to VP's by the MC.

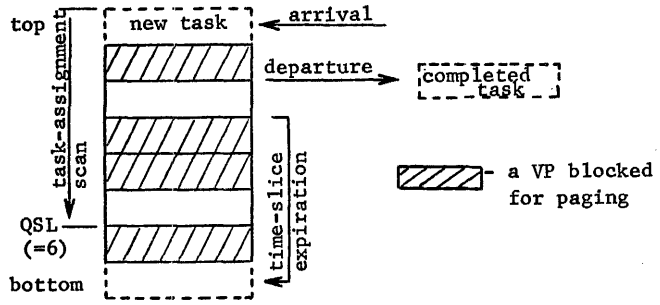


Fig.5. Circulation of VP's in an AP Queue

is less than the current queue length, the VP leaves the multiprogramming set entirely.

A VP's QRT parameter controls its time slice effectively only if the VP is operating on a processor-bound task. To keep a paging-bound VP (i.e., one whose working set exceeds the capacity of core memory) from blocking queue circulation by monopolizing core while accumulating very little processor time, each VP is provided with a backup time-slice parameter called CP-top Time (CPTT).* A VP's CPTT causes it to be preempted and demoted to the bottom of the CP queue after accumulating a software-specified quantity of "wall-clock" time at the top of the CP queue since its arrival or previous demotion.

In the interests of minimizing response time for short tasks, new arrivals to AP queues are added at the top. A task that can be completed in less than one quantum therefore receives immediate service, without being forced to wait while previous arrivals are served.

Memory Allocation

When the MC transmits a page-fault reply to an AP's memory-access request, the AP responds by saving its state in the Context Block of the VP which it is servicing and sending a page-fault signal to the SS. The SS generates a paging task for the faulting VP and adds it to the bottom of the paging queue. Each paging task requires up to three consecutive paging operations involving the paging drum: 1) write the replacement page to drum (if it has been modified), 2) check the validity of the previous write, and 3) read a new page into the core frame vacated by the replacement page. After generating a paging task, the SS removes the core frame containing the replacement page from the Core Frame List (CFL) and marks the VP blocked for page wait. It then returns to its rest state by way of the processor Task Assignment Routine. The remainder of this section describes the paging queue service and the algorithm for selecting the replacement page.

SYMBOL's paging drum is capable of one paging operation during each quarter revolution. Each passage of a quadrant boundary under the read/write heads is reported by the Drum Controller to the SS. While a short dead space between quadrants is being traversed, the SS performs post-processing on the completed operation and then assigns a paging operation for the upcoming quadrant. Post-processing a "write" or successful** "check" operation requires merely changing the operation to its successor. In post-processing a successful "read", the SS deletes the paging task from the queue, unblocks the VP responsible for it, and adds the core frame just filled to the bottom of the CFL. After post-processing the completed operation, the SS scans the paging queue, beginning at the top, until it finds an operation to assign for the upcoming quadrant.

*Only CP tasks can be paging-bound, because the TR, IP, and MRP command inherently small working sets.

**A "check" or "read" error is treated as an ESR.

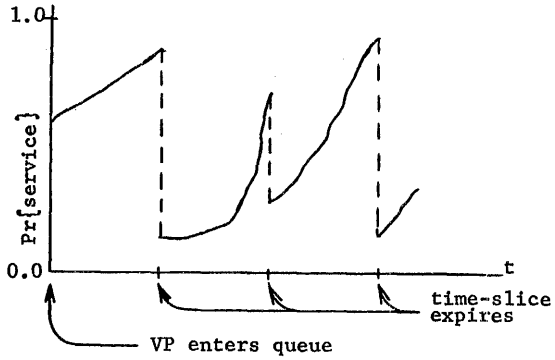


Fig. 6. Variations in Service Probability Experienced by a VP Circulating Around an AP Queue

The overall paging-queue service policy maximizes the throughput on the drum. Among operations eligible for a particular pass of a quadrant, fairness is maintained in that the selected operation is taken from the earliest-enqueued paging task. After post-processing and selection of the next operation, the SS passes through the Processor Task Assignment routine to its rest state.

The key phase of the paging process is the selection of the replacement page. Like all practical page-replacement policies, SYMBOL's attempts to select for replacement the page destined to go unreferenced the longest. Such well-known and accepted policies as Working Set and LRU [2] derive their expectations of future referencing patterns from those of the recent past SYMBOL attempts to extract the same sort of informatio from such page attributes as the processing mode and queue position of the VP owning it, as well as the typ of data contained within it.

The SS examines potential replacement pages by scanning the CFL, starting at the top. For each page under consideration, the SS looks up its Page Residenc Index (PRI), which amounts to a measure of the page's resistance to removal from core. The PRI assumes values according to the following table:

PRI	Page Characteristics
3	belongs to the VP occupying the top position in the CP queue
2	belongs to some other VP in the CP queue, or is a name-table page belonging to a VP in the TR queue
1	belongs to a VP in the IP queue, or is a non-name-table page belonging to a VP in the TR queue
0	an inactive page

The PRI of a page is assigned a positive value each time the page is transferred to core, and does not change thereafter until it is set "inactive" (PRI = 0) A page is set "inactive" by the SS whenever the VP to which it belongs undergoes a transition such that its in-core pages are unlikely to be referenced in the ne future. Examples of such transitions are PAUSE state ments or control-key signals, CP normal completions, and CP shutdowns for I/O. In addition, a VP's pages are inactivated whenever it leaves the multiprogrammi set because its time slice has expired.

The SS compares the PRI of each candidate for replacement with the Page Pushing Priority (PPP) of the faulting VP. A VP's PPP is computed as follows:

PPP	Virtual Processor Status
3	top of CP queue
max{2,p}	top of IP or TR queue (p is a software controllable parameter, $0 \leq p \leq 3$, for each VP)
p	other

The replacement page is the first page encountered in the CFL scan such that PPP > PRI. If none are found, the SS selects the first page such that PPP = PRI.* The page-replacement policy thus favors the VP at the top of the CP queue, for it can steal pages belonging to any other VP, whereas its own pages are theftproof (the extent of the CP-top VP's advantage depends on the values of "p" assigned to the other VP's by software).

Combined Allocation Policies

SYMBOL's resource-allocation policies focus on a single VP at a time (the one at the top of the CP queue), favoring it with top priority for both processor service and core-memory space. This priority is distributed among VP's in round-robin fashion at a rate, and in proportions, controlled by software through the time-slice parameters QRT and CPTT. Short-task responsiveness, moreover, is enhanced by granting every new arrival top priority for its first processing quantum. Finally, during intervals when the top-priority is in page wait, processor service is allocated to other VP's in the queue, to the extent permitted by the software-controllable parameter QSL.

The effectiveness of this design is currently the subject of empirical investigations at Iowa State University. Present plans call for a hardware-implemented software-accessible performance monitor, which will allow software not only to collect performance data for subsequent analysis, but also to exercise real-time control over SYMBOL's resource-allocation policies in response to variations in the system's operating environment.

Conclusions

Compared against conventional computing systems, SYMBOL-2R has implemented a substantially higher proportion of its operating-system functions in hardware. Since these functions include those that have short response-time requirements or high duty factors, the penalties incurred in implementing the remaining functions in non-core-resident software expressed in a high-level programming language are correspondingly diminished. The high "execution" speed of the hardware System Supervisor allows nearly all service requests to be handled sequentially, and thus leads to a rather straightforward design. The software part is more complex, since it must process multiple independent service requests concurrently in order to achieve satisfactory responsiveness. The resource-allocation mechanisms exhibit the same design philosophy: the typically high-duty-factor execution of allocation policies is performed by dedicated hardware, whereas the adjustment of these policies from time to time, to meet variations in the system's workload, is left to software.

*If no replacement can be found, the SS abandons the paging task and leaves the VP unblocked. Subsequently, the VP will again receive processor service and incur another page fault; this continues until the page fault can be serviced.

References

1. Anderberg, J.W. and Smith, C.L., "High-Level Language Translation in SYMBOL 2R," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, Association for Computing Machinery, New York, 1973.
2. Coffman, E.G., Jr. and Denning, P.J., "Operating Systems Theory," Prentice-Hall, Englewood Cliffs, N.J., 1973.
3. Dakins, M.C., "Nonnumeric Processing in the SYMBOL 2R Computer System," M.S. Thesis, Iowa State University, Ames, Iowa, 1974.
4. Hutchison, P.C. and Ethington, K., "Program Execution in the SYMBOL 2R Computer," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, Association for Computing Machinery, New York, 1973.
5. Jones, W.E., "The Role of the Interface Processor in the SYMBOL IIR Computer System," Special Report NSF-OCA-GJ33097-CL7304, Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, 1973.
6. Laliotis, T.A., "Implementation Aspects of the SYMBOL Hardware Compiler," Proceedings of the First Annual Symposium on Computer Architecture, Association for Computing Machinery, New York, 1973.
7. Richards, H., Jr., "SYMBOL IIR Programming Language Reference Manual," Cyclone Computer Laboratory, Iowa State University, Ames, Iowa, 1971.
8. Richards, H., Jr. and Wright, C.T., Jr., "Introduction to the SYMBOL 2R Programming Language," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, Association for Computing Machinery, New York, 1973.
9. Richards, H., Jr. and Zingg, R.J., "The Logical Structure of the Memory Resource in the SYMBOL-2R Computer," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, Association for Computing Machinery, New York, 1973.
10. Smith, W.R., "System Supervision Algorithms for the SYMBOL Computer," Digest of Papers, COMPCON72, Sixth Annual IEEE Computer Society International Conference, IEEE, New York, 1972.

THE DESIGN AND EVALUATION OF THE ARRAY MACHINE:

A HIGH-LEVEL LANGUAGE PROCESSOR*

Pierre Sylvain
Maniel Vineberg
Computer Science Department
University of California
Los Angeles, California

ABSTRACT

This paper describes work on a high-level language processor, the Array Machine System. The work described includes a description of the high-level language, a discussion of the two components of the system, the Net Compiler and the Array Machine, and recent measurements made on that system. The language described is based on postfix notation. The Net Compiler, which exists as a PL/1 program, prepares the control code for the Array Machine. The Array Machine, which exists as a PL/1 simulation program, executes the control code, using its primary processing element, an array of homogeneous building blocks. The measurements on the system feature a comparison to a third generation computer.

1. INTRODUCTION

The concept of direct execution of a high-level language (HLL) is not new. Cheaper electronic components and costlier software and support of large systems have encouraged considerable research in this field. Several designs have been proposed; several machines are in the development stage and others are already operational. Among the first reported efforts were the "Fortran Machine" by Bashkow et al[1], the "Cellular APL Computer" by Thurber and Myrna[2], and the "APL Machine" by Abrams[3]. All were intended to process a HLL directly although each required some pre-processing to produce an object code for execution on the machine. While these architectures were based on existing HLL's, the SYMBOL Processing Language and its associated SYMBOL-2R Computer[4,5] were designed concurrently. There is also the Aerospace Computer using the Space Programming Language SPL[6], the AADC Data Processing Element of the Navy using APL[7], and the Litton HOL Machine[8]. In addition to the product oriented projects just mentioned, there have been a number of research projects such as the Direct-High-Level Language Processor of Bloom[9], the HLL Machine GLOSS of Herriott[10], and the PL/EXUS Language and Virtual Machine of Sitton et al[11].

The processor which is discussed here was introduced in earlier publications by Vineberg and Avizienis [12,13]. The evaluation which will be described is part of a continuing effort to design and test a HLL processor. In this work, the Array Machine Language (AML) and its processor, the Array Machine System (AMS), are closely inter-related. The postfix structure of AML is related to the "AMS building blocks" and to the associated concept of a "net" of such building blocks. The AMS is composed of two separate units, the Net Compiler, a pre-processor which accepts a program written in AML and produces an internal "control code," and the Array Machine which executes this code using a number of AMS building blocks.

The present design of the AMS has been established with the help of a simulation package. Tests were conducted using a typical AML program to measure the performance of the simulated system under different design parameters, and to provide a basis for comparison of some aspects of the performance of the simulated syste to those of a "third generation" computer, the IBM S/36 Model 91.

*This research was supported by the National Science Foundation, Grant No. 33007X.

The AMS is described in three parts, 1) AML and the nets, 2) the Net Compiler, and 3) the Array Machine. The AMS simulation, the measurements and the results of the evaluation are then presented followed by conclusions indicating the present state of our research, our goals and objectives.

2. THE HIGH-LEVEL LANGUAGE AND THE NETS

2.1 The Array Machine Language (AML)

Programs written in AML represent the input data fo the net compiler. AML requires postfix notation for al arithmetic and logical expressions. This was done 1) to reflect the Array Machine architecture and 2) to ease compilation (at this stage of the research). Sinc any infix expression can be translated into postfix notation, the use of postfix notation is not limiting. Those operators currently available include fixed-point arithmetic operators +, -, *, and # (EXCLUSIVE OR) and the comparison operators <, ≤, >, ≥, =, and ≠ (results of comparison operations produce either 0 or 1, FALSE or TRUE). These are the projected machine primitives but, as will be explained, additional AML primitives, such as floating-point arithmetic operations are also available.

AML has three types of statements: the assignment statement, which can involve fixed-point or floating point arithmetic variables or Boolean variables; the transfer statement which is an unconditional branch to another part of the program; and the "transfer false" statement which is a conditional branch to another part of the program subject to a condition either in arithmetic or in logical form. Example 1 shows a) an arithmetic assignment statement which assigns the infix valu (A * (B+C))*(((D-A) + B) * C) to X, b) a transfer statement which transfers program control to label LAB and c) a transfer false statement which transfers control to label LAC if (X<Y) ∧ (Y<Z) is false.

```
a)  X = A B C + * D A - B + C * * ;
b)  T*  LAB                       ;
c)  T( X Y < Y Z < ∧ )F LAC       ;
```

Example 1: AML Statements.

AML programs are written in two sections, a declara tive part and a computational part. AML is formally defined by its grammar in Appendix A.

2.2 Arithmetic Building Blocks and Nets

Avizienis and Tung[14] have proposed an arithmetic micro-processor, the Arithmetic Building Element (ABE); it accepts two radix-16 digit operands and an operator and yields the result of the operation. The AMS building block, which accepts two word length operands and an operator, is postulated on the availability of the ABE implemented in LSI (Allen[15]) with such projected characteristics as high execution speed and low cost. The decision, yet to me made, of which functions to

implement on a building block, will be influenced both by the desire to provide as many important hardware primitives as possible in order to reduce compile time and execution time and by the costs and physical limitations of LSI implementation. Floating-point operations, to be discussed later illustrate this trade-off.

The concept of net is now introduced. A net is a connected collection of AMS building blocks where each output is either the input to other building blocks or is a final result (or both). The net generated by the statement a) of Example 1 is shown in Figure 1. Note that the operations to be performed by blocks 1 and 2 or 3 and 4 are independent; this independence will allow parallel execution on the Array Machine. As will be explained later, a net may represent more than one statement or part of a statement. A net may also be assembled to perform an AML primitive which is not an AMS building block primitive.

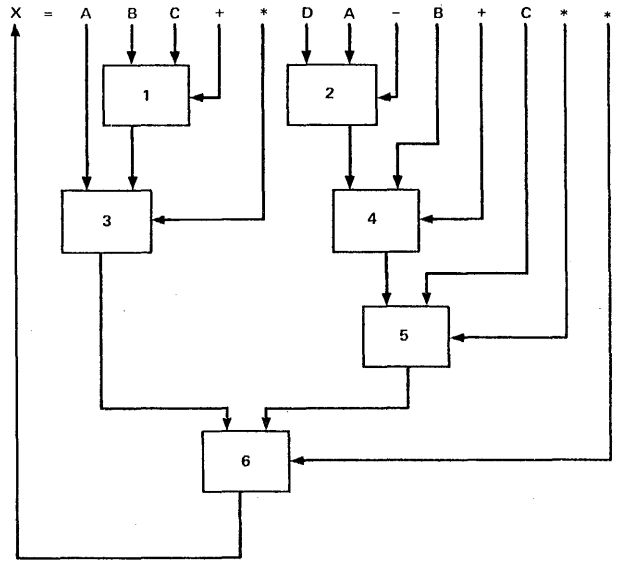


Figure 1. An AML Source Statement

3. THE NET COMPILER

The Net Compiler accepts an AML program and generates control code directly executable by the Array Machine. The control code is a sequence of net definitions representing the total mapping of the AML program statements into nets, (illustrated in Section 2.2). Any inherent parallelism that exists within a postfix expression is preserved by the Net Compiler. For example, in the net of Figure 1, the operations performed by blocks #1 and #2 are performed concurrently. Transmission of a result from one block to the input of another block within the Array Machine is done via a "processor bus" system. If two blocks wish to send their respective outputs at the same time and if there is only one processor bus available, a conflict will arise. At present, the Net Compiler takes care of such situations by specifying that the broadcast of one of the inputs be delayed; as an alternative this conflict resolution could be implemented in hardware. Figure 2 shows a simplified diagram of net compilation.

Net compilation involves three tasks, 1) syntax analysis, 2) semantic analysis, and 3) control code specification.

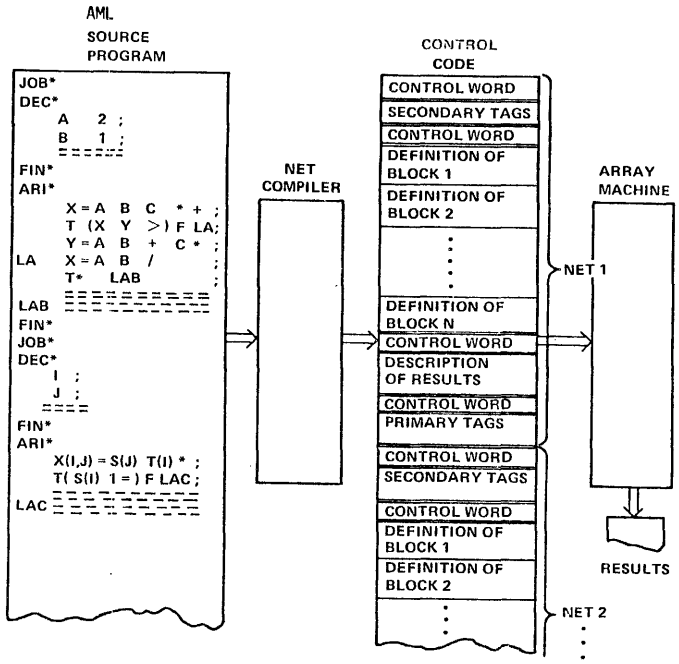


Figure 2. Program Flow in the Array Machine System

3.1 Syntax Analysis

The Net Compiler first records all variable declarations and then performs a lexical analysis on each element of the computational statements in the AML program. Each lexically correct statement is then checked syntactically to verify that it is a member of AML.

3.2 Semantic Analysis

Before semantic analysis is begun, an optimization preprocessor may be invoked to find those algebraically equivalent rewritings of arithmetic expressions which yield the best timings. These timings depend on the number of processor buses, on the number of building blocks, and on arithmetic operation times. The rules for generating these rewritings appear in Appendix B. The present version of the Net Compiler publishes the optimized rewritings but makes no actual substitutions.

The semantic analysis of a syntactically correct AMS program is performed in two steps, the static analysis and the dynamic analysis. In the static part, AML statements are mapped into nets regardless of the actual number of available AMS building blocks in the Array Machine and regardless of net boundaries. In the dynamic part nets are arranged according to the number of building blocks and the natural net boundaries. The "natural" net boundaries are as follows: a labelled statement must start a new net and hence forces the termination of the net from the preceding statement; a transfer statement must terminate a net; and an address dependency separates statements. An address dependency arises when a variable on the Left-Hand Side (LHS) of one statement appears as a subscript of a variable on the Right-Hand Side (RHS) of the next statement. This part of the Net Compiler also forms subnets if the net is oversized with respect to the number of available building blocks and it assigns the special standard nets for those AML primitives not implemented in hardware.

3.3 Control Code

Once the nets are formed, the control code is assembled and written in the instruction word format. The Net Compiler must include appropriate "secondary tags" to allow "direct input operand updating." This

need arises when one net will require a result from the preceding net as a consequence of either 1) a variable appearing on the LHS of a statement in the first net and the RHS of a statement in the second net or 2) the partitioning of a single statement.

At execution time, rather that waiting for the variable to be updated in memory, the Array Machine (which will be described in the following section) will use the secondary tag at the beginning of the description of the second net, to update the appropriate operand register directly. The Net Compiler also prepares temporary register tags for temporary results which must be saved for subsequent nets.

The control code specifies all operands, operators and operation times and result destinations (via "primary tags") for each block of each net. A second complete control code specification is prepared by repeating the dynamic semantic analysis for one less than the number of available AMS building blocks. This "fault tolerance" feature, along with a table of rollback points, allows testing of reconfiguration and recovery in the event of a single AMS building block failure.

4. THE ARRAY MACHINE

The Array Machine is designed to efficiently process the control code prepared by the Net Compiler. There are four main units in the machine: the instruction memory IM, the operand memory OM, the sequence and control unit SCU, and the array of building blocks ABB. The simplified machine block diagram is given in Figure 3.

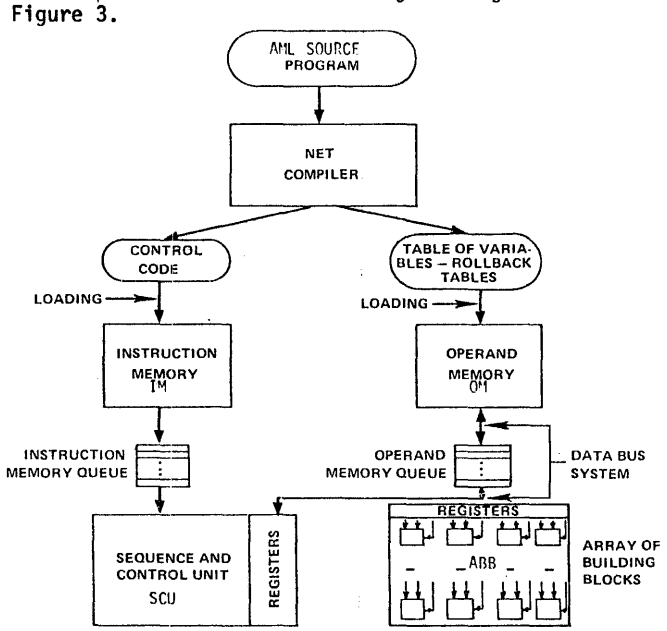


Figure 3. The Array Machine System Block Diagram

Control code execution requires three identical sets of internal registers; one set, the "primary registers," is used by the ABB and the other two sets, the "secondary registers" and the "tertiary registers," are used by the SCU. The control code is initially loaded into the IM and the operands and the rollback points table are stored in the OM prior to the beginning of execution. The machine runs in logical "net cycles" which are of variable length. The normal mode of operation of the machine proceeds as follows: during a net cycle, two separate and independent sequences of operations take place simultaneously; one net is executed from the primary registers by the ABB while the next net is fetched by the SCU from the IM and OM into the secondary registers and (in the case of a conditional transfer) into the set of tertiary registers. The net cycle is complete when both the execution and the fetch and decode operations are complete. At that time, the contents of the secondary (or tertiary) registers are passed to the primary registers and a new net cycle is started. When the sequence is broken by a normal transfer or a transfer false (with a FALSE condition), the normal mode of operations (which is called the "overlap mode") is modified (as described in Section 4.3). Functioning of the Array Machine will now be described in three parts, 1) the operand memory and the data bus system, 2) the instruction memory, and 3) the sequence and control unit.

4.1 The Operand Memory and the Data Bus System

The Array Machine has two independent memories, the operand memory OM and the instruction memory IM, and it is envisioned that they will have different organizations. In addition to the obvious advantage of allowing independent and simultaneous requests to each memory, the choice of two different units was dictated by the following:

i) The structure of the data stored in each system is different. Operands are more or less "randomly" distributed in the OM; access to these data from more than one source may cause conflicts. Instruction words on the other hand are stored sequentially within a net; the IM is interleaved and access to the IM is performed one word per module at a time such that no conflict can occur. Also, data are read out and written into the OM but instruction words, once loaded into the IM, are only fetched (there is no dynamic alteration of instruction words in the Array Machine).

ii) The OM and the IM are functionally different. Requests to the OM can originate from a large number of sources (all the registers) whereas requests to the IM are from at most two independent sources (the two SCU decoding units). The OM is linked to the SCU via a bus system with an elaborate memory manager including both input and output queues. The control of the IM is much simpler.

iii) Finally, measurements have shown that the rate at which instruction words are issued is a far more critical parameter for system efficiency than the average speed of operand fetching.

There are two distinct bus systems (shown in Figure 4) in the Array Machine, the processor bus system in the ABB for building block inter-communication and the data bus system for traffic of data between the OM and the SCU registers. Information on each bus system is tagged for sender-receiver identification. On the processor bus system, the tag contains the number of the sending building block and the data is received by any register whose tag field has been set (by an instruction word) to that number. On the data bus system the tag contains the receiving register number or address. Bus traffic on the data bus system is managed by the OM controller. Requests are delayed when there are not enough bus lines available.

4.2 The Instruction Memory and the Instruction Word Format

The IM contains the control code which is composed of a sequence of instruction words. Figure 5 shows the format of an instruction word. It has four fields, the link field, the status field, the tag field, and the address field. Each field is used in some special sense by each type of instruction word (cf. Figure 2). After the first control word, one finds the secondary

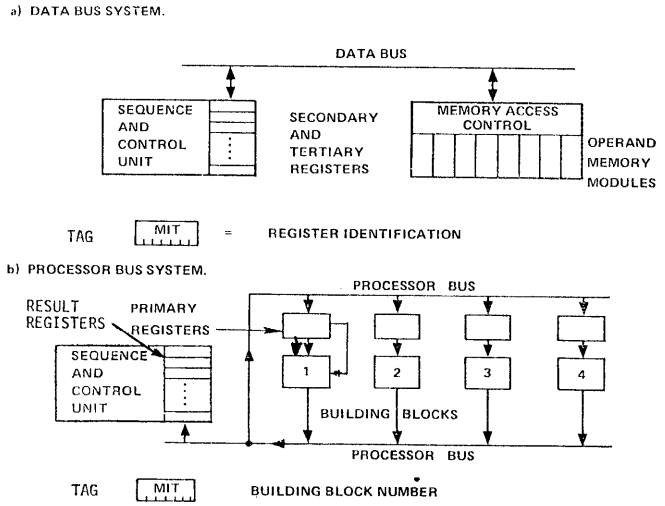


Figure 4. The Bus Systems

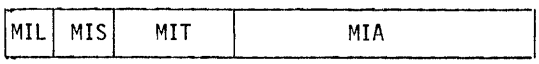


Figure 5. The Instruction Word Format

tags which identify the secondary registers to be updated directly from the net currently executing. Note that they are fetched first before execution starts in the ABB to prevent any loss of data (since the ABB could generate a result needed for updating at any time). When the ABB is executing a net terminated by a conditional transfer, the SCU fetches and decodes two different streams of control code simultaneously from the IM using two independent decoding units. The memory handler enqueues and initiates memory access requests, fills the buffers when accesses are complete and performs various housekeeping functions. The buffers are used in flip-flop; that is, a decoding unit and its registers work with the contents of one buffer while the other is being filled with the next instructions in sequence. Experimentation has shown that it is convenient to set the buffer length, a machine parameter, to a multiple of the IM interleaving, also a machine parameter.

4.3 The Sequence and Control Unit

The SCU fetches, decodes and sequences the flow of instruction words in a net in order to load the secondary or tertiary registers. In the overlap mode, the normal mode of machine operation, the SCU fetches and decodes the next net in sequence while the current net is being executed by the ABB. The control code description, although following a fixed pattern, contains a variable number of instruction words. There will always be four control words but there may or may not be secondary tags; the number of blocks is variable and each block definition may have from 3 to 9 words (depending on subscripting); there may or may not be results and primary tags. For these reasons, sequencing is complex but also strictly sequential; pipelining for instance, would be difficult. A means to improve SCU operations will be suggested later.

One of the most important problems in instruction decoding results from transfers. When a net terminates by an unconditional transfer, the SCU decodes a special instruction word which triggers a series of operations: the buffers are emptied and an IM request is made for instruction words at the branch location. A problem of variable integrity may arise if, in the target net, there are variables that are updated by the net currently executing. This problem was solved for two nets in sequence with the set-up of secondary tags. Such a scheme is not possible here since there is no way to know beforehand the dynamic path of control leading from one net to another (there may be many transfers to the same target net for instance). The SCU solves this problem by holding up all OM requests made by the target net until the net being executed has enqueued its write requests. If the transfer was a conditional branch then there are two possible target nets; the SCU then enters the "look-ahead" mode and the two independent decoding units fetch and decode the two possible target nets simultaneously. In this case, when one or both of the possible target nets are not the next net in sequence, the same problem of variable integrity arises and is solved the same way. As soon as the outcome of the condition is generated by the ABB, the SCU decides which net will continue to be fetched (in the event where fetching is not finished at that time) and consequently which net will be abandoned. Note that a branch means waiting for the correct instruction words to be delivered by the IM and can cause delay. Whenever possible, the SCU will issue IM requests in advance (i.e. before the net cycle is complete) as soon as it knows the target location(s).

At the end of a net cycle, the SCU passes the contents of the secondary (or tertiary) registers to the primary registers. Note here that due to the fact that in general nets are of different lengths, it is unavoidable that, at the end of each net cycle, either the ABB will wait for the SCU to finish the fetching of the next net or the SCU will wait for the ABB to finish execution. In the current AMS the time required to fetch and decode a net is, on the average, much longer than the time needed to execute that net. Therefore effort is being directed at making the SCU (in particular, buffer handling) more efficient.

One way to improve SCU processing speed would be to have a fixed pattern for net description (i.e. always reserve room for the maximum number of building blocks, consider that all operands have two subscripts etc...) and to have a fixed number of decoding units in the SCU that could fetch and decode several block definitions simultaneously. This involves a space/time tradeoff that is being considered.

5. THE SIMULATION AND THE EVALUATION OF THE ARRAY MACHINE SYSTEM

The AMS, incorporating the design ideas described in the previous sections, exists as a simulation package. The simulation of a preliminary detailed design allowed the viability of the system to be verified and its performance to be measured, both as a function of different system parameters, architectures, memory systems, etc.... It also helped to evaluate the AMS performance against that of the IBM S/360 Model 91.

5.1 The Simulation and the Results of the Measurement Tests

The simulation package consists of two distinct programs, the Net Compiler Simulation Program (NCSP) and the Array Machine Simulation Program (AMSP). Both are written in PL/1 and run on the IBM S/360 Model 91 of the UCLA Campus Computing Network Facility. An interactive CRT-display system available at this facility permits the user to use both programs in an interactive facion and to view the results. Runs are submitted with data consisting of those programs, written in AML, to be executed by the AMS. Extensive information concerning step-by-step operations of the various units of the system as well as program results and general timing information from the AMS are available for each run. The timing results of an execution

are detailed in three different categories: 1) fetch time, the number of machine cycles during which the SCU alone was operating (fetching), 2) exec time, the number of machine cycles during which the ABB alone was operating (executing a net), and 3) overlap time, the number of machine cycles during which the machine is in true overlap mode, that is, when both the SCU and the ABB are operating. The total running time is the sum of these three times. Both the NCSP and the AMSP are parameterized allowing for different system configurations. The same source program may be submitted with any number of lists of parameters. A parameter not specified in the list is given a default value by the NCSP.

The source program used for the measurement is MLSQ, a program taken from IBM's Scientific Subroutine Package. MLSQ solves a system of linear equations and represents a fairly large computation. It was translated into the source language for execution on the Array Machine System and is in all points similar to the PL/1 version. The correctness of the simulation was checked by running MLSQ on the AMS and in PL/1 with the same data and check for identical results. Among the many parameters tested in the simulation, the most important ones are:

- DECODE, the rate of instruction decoding and sequencing in the SCU and CYCLE, the rate at which instructions are issued to the SCU by the IM; these two important functional characteristics were taken as independent variables and were given a fixed set of values.

- The main other parameters tested were the number of ways the OM is interleaved, the OM cycle time, the number of ports per OM module, the number of available building blocks, and the number of data buses. These parameters were set to a value representing a "test case" and a first run was performed with these values and ten representative combinations of values of DECODE and CYCLE. Then for these ten combinations, several tests were run where only one parameter was changed from the test case at a time to determine the influence of this parameter.

Relative conclusions for each measurement were reached by considering two results: a) the total running time in machine cycles and b) the ratio of fetch time to total time. Ideally, the total time should be composed primarily of overlap time; the other two (that is, fetch time and execute time) are mainly due to the difference in length between any two nets and they should be small. Actually the measurements showed that fetch time, although substantially reduced by many refinements in the design, remained very important (about 65% of the total time). This constitutes the main result from the simulation; as foreseen above (cf. Section 4.3 about the SCU) it takes on the average more time to decode the instructions specifying one building block operation than to execute this operation. And in the present design the relative speed of the ABB which features parallel execution capabilities and the SCU performance are not yet matched. Standardizing the net description as suggested in the preceding section should improve SCU performance.

An extensive discussion of the results of the simulation tests can be found in [17]; the other main results are:

a) DECODE and CYCLE are critical parameters which have an important effect on performance. As illustrated by Figure 6, it is essential that they are matched, i.e., close to equality. The two curves are for two values of CYCLE. The topmost two curves represent measurements where all other parameters were set to their best values according to the simulation results (i.e., yielding the best performance). The lower two curves are for the test case. It is clear that the best improvement in performance is around the knee of the curve, that is where CYCLE-1<DECODE<CYCLE+1. Higher values of DECODE do not bring any improvement (the curves level off rapidly).

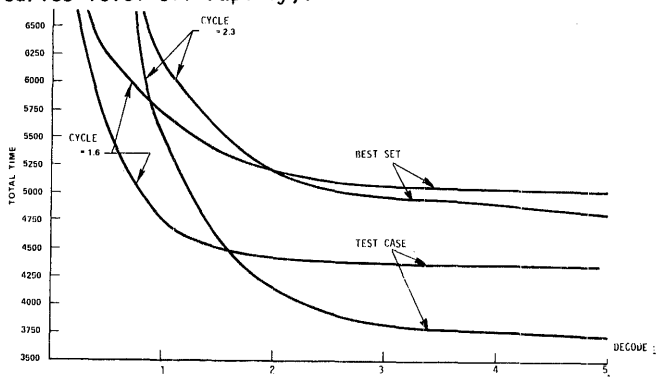


Figure 6. Timing Results of the AMS for Test Case and Best Set of Design Parameters

b) The operand memory characteristics, number of modules and access time showed a strong effect on performance. This is not surprising since for most of the net cycles operand memory requests are delayed until the moment they are all released at the same time, and a large bandwidth is critical. However, the number of ports per module does not seem to affect the performance.

c) For the number of building blocks, its upper limit is, as projected, equal to the largest possible net in the program. Simulation showed that regular programs exhibit a large number of natural net boundaries, therefore requiring a low maximum number of building blocks (typically, 10). Reducing this number degrades slightly the performance and significantly increases the ratio of fetch time to total time.

d) As an independent project[18] floating-point operations implemented in "software" with standard nets has been simulated in the Array Machine. The conclusion of this study states that execution time of these operations is significantly high and that it would be more advantageous to implement floating point operations in hardware in the Building Blocks. However, it is clear that the idea of "standardized" nets might be useful for other more complicated functions or operations like array operations.

5.2 The Evaluation of the System

In order to gain some insight into the viability of such a system, MLSQ was executed on both the Array Machine System (in the source language) and on the IBM S/360 Model 91 (in PL/1) and timing measurements were made during the execution phase of each. On the Model 91 the execution time was found by recording the machine clock before and after computation and then dividing the total time into Model 91 basic machine cycle (60 nanoseconds). In the AMSP, all parameters were set to values reflecting the functional characteristics of the Model 91: 1) the OM and IM were interleaved 16 ways, 2) the memory cycle times were set to 13, 3) DECODE was set to 1 instruction per cycle and 4) all operation times were equalized, 1 cycle for addition, 3 for multiplication, etc.

Clearly, a scientific program such as MLSQ tends to favor the Array Machine because of the inherent parallelism in the computations; but parallelism is also exploited in the Model 91 which employs both an

adder and a multiply/dive unit. The conclusions to be drawn from this comparison are merely indicative and in no way definite and abiding. Nevertheless, since, 1) we tried to make the comparison as fair as possible, 2) the result was that the Array Machine under the conditions specified above was 4 to 5 times faster than the IBM S/360 Model 91, 3) there are functional characteristics different from those of the Model 91 which are better suited to the Array Machine and, 4) a standardized block definition is expected to save substantial fetch time, the results of this experiment provide an early indication that design of the AMS is a viable one and that it should be studied further.

6. CONCLUSIONS

An Array Machine System based on a processor whose central feature is an array of homogeneous building blocks (whose construction will be feasible in the near future) has been detailed and designed in a simulation package. Although a series of measurements on the simulation indicated that the idea is feasible, particularly under certain conditions, it also showed that the system needs more work. Although the ABB represents a very powerful and fast resource, the SCU, the memory systems, and the Net Compiler must be carefully designed to efficiently exploit it. Space may have to be traded for time as in the standardization of the block definition.

The question of just what an array machine should or should not be remains open. The ABB may be considered more like a special resource which is efficient under certain conditions, for example, in the execution of long statements, of special functions which require a large net or of array operations which by definition imply a high degree of concurrency. Array processing is to be implemented, but it is not yet clear that the ABB should perform index incrementation, transfer false with a simple condition statements with one operation, etc., i.e. operations which do not exploit the full capacity of the ABB. It is clear from its description that even implemented totally in hardware, net compilation is slow. Our current research envisions two or more mini-processors sharing a single ABB. Some of these dedicated processors would be net compilers with local storage. In this scheme, the IM would become distributed in the system and the OM would be expanded into a hierarchy of storage devices shared by the Net Compiler and the ABB. Finally, the SCU would be expanded into a scheduler/supervisor to regulate the whole system, to keep the ABB busy, to supervise the memory transfers, etc.

As the design evolves, hardware as well as software fault-tolerant techniques, such as thosed used in the JPL-STAR computer[19] and other fault-tolerant systems will be incorporated into the design.

We have demonstrated the feasibility of direct execution of a simple HLL on an array machine which takes advantage of inter-statement and intra-statement parallelism. Work is continuing to use the ABB more efficiently and to add more advanced features such as fault-tolerance, self-optimization, distributed processing, and resource sharing to the original design. Such a computer system would be suitable for processing of any HLL that can be translated into the Array Machine Language with reasonable efficiency.

REFERENCES

[1] Bashkow, T. R., A. Sasson, and A. Kronfeld, "System Design of a FORTRAN Machine," IEEE Transactions on Electronic Computers, Vol. EC-16, No. 4, August 1967.

[2] Thurber, K. J., "System Design of a Cellular APL Computer," IEEE Transactions on Computers, Vol. EC-19, No. 4, April 1970.

[3] Abrams, P. S., "An APL Machine," Technical Report No. 3, Digital Systems Laboratory, Stanford Electronics Laboratory, February 1970.

[4] Proceedings of the Spring Joint Computer Conference, (collection of four papers on the SYMBOL computer), pp. 563-616, 1971.

[5] Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, (collection of four papers on the SYMBOL-2R computer), pp. 1-33, November 1971.

[6] Nielsen, W. C., "Design of an Aerospace Computer for Direct HOD Execution," Proceedings of the ACM-IEEE Symposium on High-Level Language Computer Architecture, November 1973.

[7] Nissen, S. M. and S. J. Wallach, "The All Applications Digital Computer," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, November 1973.

[8] Shroeder, S. C. and L. E. Vaughn, "A High Order Language Optimal Execution Processor," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, November 1973.

[9] Bloom, H. M., "Structure of a Direct High-Level Language Processor," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, November 1973.

[10] Herriot, R. G., "GLOSS: a High-Level Language Machine," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, November 1973.

[11] Sitton, G. A., T. A. Kendrick, and A. G. Carrick, Jr., "The PL/EXUS Language and Virtual Machine," Proceedings of the ACM-IEEE Symposium on High-Level-Language Computer Architecture, November 1973.

[12] Vineberg, M. and A. Avizienis, "Implementation of a Higher-Level Language on an Array Machine," Proceedings of Compcon '72, pp. 37-39, September 1972.

[13] Vineberg, M. and A. Avizienis, "Implementation of a Higher-Level Language on an Array Machine," International Workshop on Computer Architecture, Grenoble, France, June 1973.

[14] Avizienis, A. and C. Tung, "A Universal Arithmetic Building Element and Design Methods for Arithmetic Processors," IEEE Transactions on Computers, Vol. C-16, No. 8, August 1970.

[15] Allen, D., "The Development of Logic Design Procedures for Large Scale Integration Constraints," Ph.D. Dissertation, UCLA Computer Science Department, 1972.

[16] Tung, C. and A. Avizienis, "Combinational Arithmetic Systems for the Approximation of Functions," AFIPS Conference Proceedings, Vol. 36, SJCC 1970.

[17] Sylvain, P., "Evaluating the Array Machine," Technical Report No. UCLA-ENG-7462, School of Engineering and Applied Science, UCLA, August 1974.

[18] Thomasian, A., "The Implementation of Floating-Point Arithmetic on the Array Machine," Internal Memorandum No. 117, Digital Technology Research Group, Computer Science Department, UCLA, June 1973.

[19] Avizienis, A. et al, "The STAR (Self-Testing and Repairing Computer): An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Transactions on Computers, Vol. C-20, No. 11, November 1971.

APPENDIX A

Formal Definition of Source Language

The source language is defined by G, the source language grammar, where G = (V_N, V_T, S, P). V_N and V_T are the non-terminal and terminal alphabets where V_N = (S, S_a, S_z, S_t, A, Ø, B, Q, J, C, ℓ) and V_T = (v, ←, ;, a, +, -, *, /, b, ∨, ∧, ⊕, ¬, T, (,), F, =, ≠, <, >, ≤, ≥). The production set P follows:

Productions P	Comment
$S \rightarrow S_a, S_b, S_t$	There are three statement types.
$S_a \rightarrow v \leftarrow A$;	S_a is an arithmetic assignment statement.
A → A A Ø	A is a postfix arithmetic expression.
A → a	"a" is an arithmetic operand (variable or literal), fixed-point or floating-point.
Ø → +, -, *, /	There are four arithmetic operators.
$S_b \rightarrow v \leftarrow B$;	S_b is a Boolean assignment statement.
B → B B Q	B is a postfix Boolean expression.
B → B	(NOT) is a monadic operator.
B → b	"b" is a Boolean operand.
Q → v, ∧, ⊕	OR, AND, and EXCLUSIVE OR are logical operators.
$S_t \rightarrow J L$;	S_t is a transfer control.
J → T	T is an unconditional transfer.
J → T(C)F	This is a conditional transfer.
C → C C Q	C is a Boolean condition.
C → C ¬	
C → a a ℓ	
ℓ → =, ≠, <, >, ≤, ≥	ℓ is a comparison operator.

APPENDIX B

Optimization of Arithmetic Statements

Source code optimization considers arithmetic assignment statements having 3 or more operators. Each of these statements is rewritten to give every possible combination of operator placements. Certain rewritings are then selected as being optimum with respect to the criteria defined in the section on net compiling.

The rewritings are generated using identies 1 through 10 below:

(1) C + + = + C +
(2) C + - = - C -
(3) C - + = - C +
(4) C - - = - C +
(5) a b + = b a +
(6) C * * = * C *
(7) C * / = / C /
(8) C / * = * C /
(9) C / / = / C *
(10) a b * = b a *

In (5) and (10) above, "a" and "b" are well formed postfix expressions (e.g., C D -) As an example, A B - C - D - can be rewritten in the following ways:

(a) A B C + - D -
(b) A B - C D + -
(c) A B C + D + -
(d) A B C D + + -

Expression (a) above is also equivalent to A C B + - D - for example, but the operator placement is unchanged and the nets for these two rewritings are identical (except for an operand interchange). Such a rewriting is not considered.

In addition, a procedure is called which makes these substitutions.

(1) A B * A C * + = B C + A *
(2) A B * A C * - = B C - A *
(3) A B / C B / + = A C + B /
(4) A B / C B / - = A C - B /

A Preliminary Architecture for a Basic Data-Flow Processor*

Jack B. Dennis and David P. Misunas
Project MAC
Massachusetts Institute of Technology

Abstract: A processor is described which can achieve highly parallel execution of programs represented in data-flow form. The language implemented incorporates conditional and iteration mechanisms, and the processor is a step toward a practical data-flow processor for a Fortran-level data-flow language. The processor has a unique architecture which avoids the problems of processor switching and memory/processor interconnecion that usually limit the degree of realizable concurrent processing. The architecture offers an unusual solution to the problem of structuring and managing a two-level memory system.

Introduction

Studies of concurrent operation within a computer system and of the representation of parallelism in a programming language have yielded a new form of program representation, known as data flow. Execution of a data-flow program is data-driven; that is, each instruction is enabled for execution just when each required operand has been supplied by the execution of a predecessor instruction. Data-flow representations for programs have been described by Karp and Miller [8], Rodriguez [11], Adams [1], Dennis and Fosseen [5], Bährs [2], Kosinski [9, 10], and Dennis [4].

We have developed an attractive architecture for a processor that executes elementary data-flow programs [6, 7]. The class of programs implemented by this processor corresponds to the model of Karp and Miller [8]. These data-flow programs are well suited to representing signal processing computations such as waveform generation, modulation and filtering, in which a group of operations is to be performed once for each sample (in time) of the signals being processed. This elementary data-flow processor avoids the problems of processor switching and processor/memory interconnection present in attempts to adapt conventional Von Neuman type machines for parallel computation. Sections of the machine communicate by the transmission of fixed size information packets, and the machine is organized so that the sections can tolerate delays in packet transmission without compromising effective utilization of the hardware.

We wish to expand the capabilities of the data-flow architecture, with the ultimate goal of developing a general purpose processor using a generalized data-flow language such as described by Dennis [4], Kosinski [9, 10] and Bährs [2]. As an intermediate step, we have developed a preliminary design for a basic data-flow processor that executes programs expressed in a more powerful language than the elementary machine, but still not achieving a generalized capability. The language of the basic machine is that described by Dennis and Fosseen [5], and includes constructs for expressing conditional and iterative execution of program parts.

In this paper we present solutions to the major problems faced in the development of the basic machine. A straightforward solution to the incorporation of decision capabilities in the machine is described. In addition, the growth in program size and complexity with the addition of the decision capability requires utilization of a two-level memory system. A design is presented in which only active instructions are in the operational memory of the processor, and each instruction is brought to that memory only when necessary for program execution, and remains there only as long as it is being utilized.

*The work reported here was supported by the National Science Foundation under research grant GJ-34671.

The Elementary Processor

The Elementary Processor is designed to utilize the elementary data-flow language as its base language. A program in the elementary data-flow language is a directed graph in which the nodes are operators or links. These nodes are connected by arcs along which values (conveyed by tokens) may travel. An operator of the schema is enabled when tokens are present on all input arcs. The enabled operator may fire at any time, removing the tokens on its input arcs, computing a value from the operands associated with the input tokens, and associating that value with a result token placed on its output arc. A result may be sent to more than one destination by means of a link which removes a token on its input arc and places tokens on its output arcs bearing copies of the input value. An operator or a link cannot fire unless there is no token present on any output arc of that operator or link.

An example of a program in the elementary data-flow language is shown in Figure 1 and represents the following simple computation:

```
input a, b
   y := (a+b)/x
   x := (a*(a+b))+b
output y, x
```

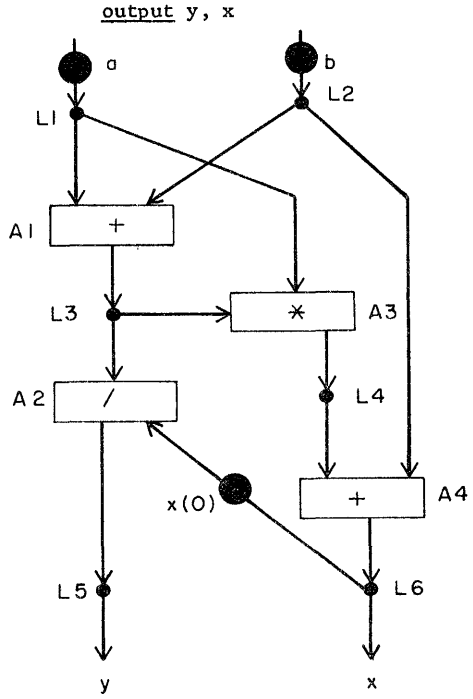


Figure 1. An elementary data-flow program.

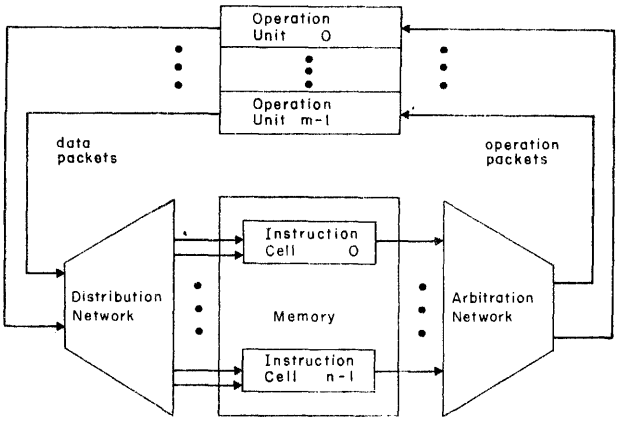


Figure 2. Organization of the elementary data-flow processor.

The rectangular boxes in Figure 1 are operators, and each arithmetic operator in the above computation is reflected in a corresponding operator in the program. The small dots are links. The large dots represent tokens holding values for the initial configuration of the program.

In the program of Figure 1, links L1 and L2 are initially enabled. The firing of L1 makes copies of the value a available to operators A1 and A3; firing L2 presents the value b to operators A1 and A4. Once L1 and L2 have fired (in any order), operator A1 is enabled since it will have a token on each of its input arcs. After A1 has fired (completing the computation of a+b), link L3 will become enabled. The firing of L3 will enable the concurrent firing of operators A2 and A3, and so on.

The computations represented by an elementary program are performed in a data-driven manner; the enabling of an operator is determined only by the arrival of values on all input links, and no separate control signals are utilized. Such a scheme prompted the design of a processor organized as in Figure 2.

A data-flow schema to be executed is stored in the Memory of the processor. The Memory is organized into Instruction Cells, each Cell corresponding to an operator of the data-flow program. Each Instruction Cell (Figure 3) is composed of three registers. The first register holds an instruction (Figure 4) which specifies the operation to be performed and the address(es) of the register(s) to which the result of the operation is to be directed. The second and third registers hold the operands for use in execution of the instruction.

When a Cell contains an instruction and the necessary operands, it is enabled and signals the Arbitration Network that it is ready to transmit its contents as an operation packet to an Operation Unit which can perform the desired function. The operation packet flows through the Arbi-

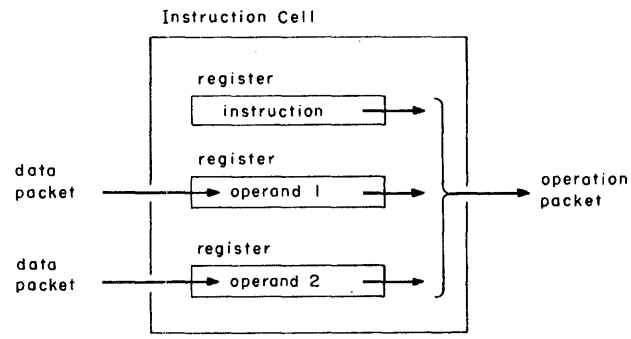


Figure 3. Operation of an Instruction Cell.

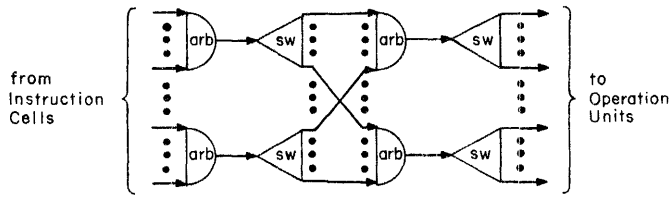


Figure 5. Structure of the Arbitration Network.

tration Network which directs it to an appropriate Oper tion Unit by decoding the instruction portion of the pa et.

The result of an operation leaves an Operation Unit as or more data packets, consisting of the computed value the address of a register in the Memory to which the va is to be delivered. The Distribution Network accepts d packets from the Operation Units and utilizes the addre of each to direct the data item through the network to correct register in the Memory. The Instruction Cell c taining that register may then be enabled if an instruc tion and all operands are present in the Cell.

Many Instruction Cells may be enabled simultaneously, a it is the task of the Arbitration Network to efficientl deliver operation packets to Operation Units and to que operation packets waiting for each Operation Unit. A structure for the Arbitration Network providing a path operation packets from each Instruction Cell to each Op eration Unit is presented in Figure 5. Each Arbitratic Unit passes packets arriving at its input ports one-at- time to its output port, using a round-robin discipline resolve any ambiguity about which packets should be ser next. A Switch Unit assigns a packet at its input to c of its output ports, according to some property of the packet, in this case the operation code.

The Distribution Network is similarly organized using Switch Units to route data packets from the Operation Units to the Memory Registers specified by the destina- tion addresses. A few Arbitration Units are required s data packets from different Operation Units can enter t network simultaneously.

Since the Arbitration Network has many input ports and only a few output ports, the rate of packet flow will b much greater at the output ports. Thus, a serial rep- resentation of packets is appropriate at the input port to minimize the number of connections to the Memory, b a more parallel representation is required at the outpu ports so a high throughput may be achieved. Hence, serial-to-parallel conversion is performed in stages within the Arbitration Network. Similarly, parallel-t serial conversion of the value portion of each result packet occurs within the Distribution Network.

The Operation Units of the processor are pipelined in

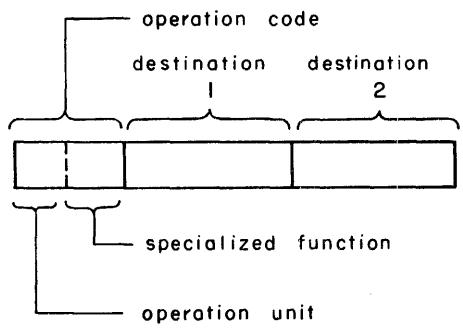


Figure 4. Instruction format.

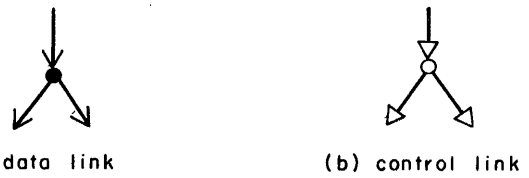


Figure 6. Links of the basic data-flow language.

order to allow maximum throughput. The destination address(es) of an instruction are entered into identity pipelines of the Operation Units and are utilized to form data packets with the result when it appears.

A more detailed explanation of the elementary processor and its operation is given in [6]. We have completed designs for all units of the elementary processor in the form of speed-independent interconnections of a small set of basic asynchronous module types. These designs are presented in [7].

The Basic Data-Flow Language

Our success in the architecture of the elementary data-flow processor led us to consider applying the concepts to the architecture of machines for more complete data-flow languages. For the first step in generalization, we have chosen a class of data-flow programs that correspond to a formal data-flow model studied by Dennis and Fosseen [5].

The representation of conditionals and iteration in data-flow form requires additional types of links and actors. The types of links and actors for the basic data-flow language are shown in Figures 6 and 7.

Data values pass through data links in the manner presented previously. The tokens transmitted by control links are known as control tokens, and each conveys a value of either true or false. A control token is generated at a decider which, upon receiving values from its input arcs, applies its associated predicate, and produces either a true or false control token at its output arc.

The control token produced at a decider can be combined with other control tokens by means of a Boolean operator (Figure 7f), allowing a decision to be built up from simpler decisions.

Control tokens direct the flow of data tokens by means of T-gates, F-gates, or merge actors (Figure 7c, d, e). A T-gate passes the data token on its input arc to its output arc when it receives a control token conveying

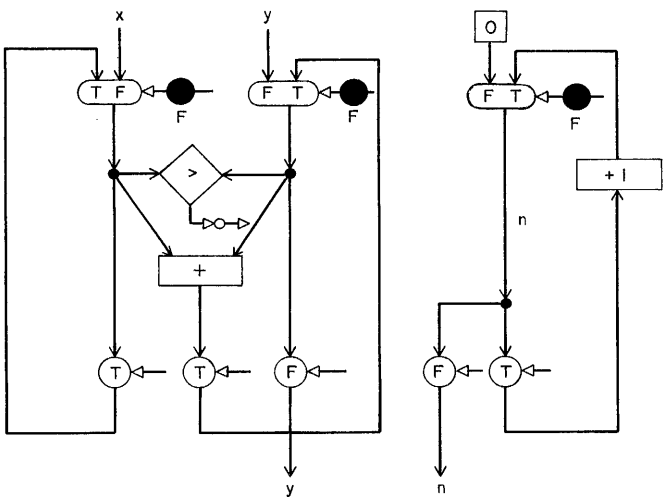


Figure 8. Data-flow representation of the basic program.

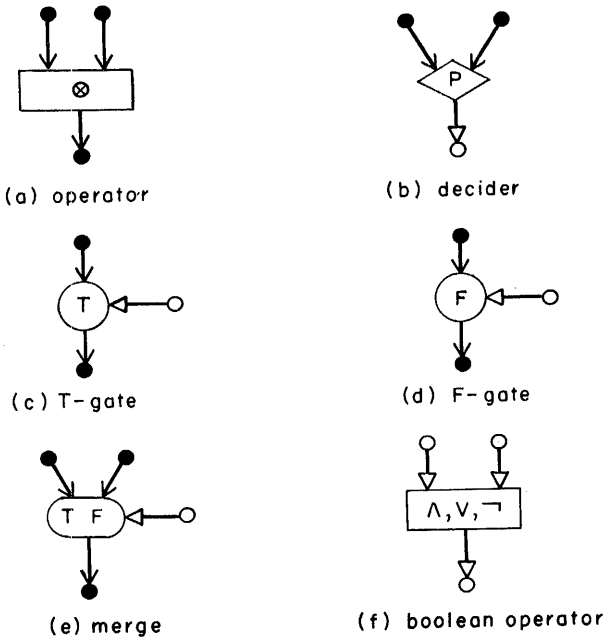


Figure 7. Actors of the basic data-flow language.

the value true at its control input. It will absorb the data token on its input arc and place nothing on its output arc if a false-valued control token is received. Similarly, the F-gate will pass its input data token to its output arc only on receipt of a false-valued token on the control input. Upon receipt of a true-valued token, it will absorb the data token.

A merge actor has a true input, a false input, and a control input. It passes to its output arc a data token from the input arc corresponding to the value of the control token received. Any tokens on the other input are not affected.

As with the elementary schemas, a link or actor is not enabled to fire unless there is no token on any of its output arcs.

Using the actors and links of the basic data-flow language, conditionals and iteration can be easily represented. In illustration, Figure 8 gives a basic data-flow program for the following computation:

```
input y, x
n := 0
while y < x do
   y := y + x
   n := n + 1
   end
output y, n
```

The control input arcs of the three merge actors carry false-valued tokens in the initial configuration so the input values of x and y and the constant 0 are admitted as initial values for the iteration. Once these values have been received, the predicate y < x is tested. If it is true, the value of x and the new value for y are cycled back into the body of the iteration through the T-gates and two merge nodes. Concurrently, the remaining T-gate and merge node return an incremented value of the iteration count n. When the output of the decider is false, the current values of y and n are delivered through the two F-gates, and the initial configuration is restored.

The Basic Data-Flow Processor

Two problems must be faced in adapting the design of the

elementary data-flow processor for basic data-flow programs. The first task is to expand the architecture of the elementary machine to incorporate decision capability by implementing deciders, gates and merges. A fairly straightforward solution to this problem will be presented.

However, in contrast to elementary data-flow programs, the nodes of a basic data-flow program do not fire equally often during execution. As computation proceeds, different parts of the program become active or quiescent as iterations are initiated and completed, and as decisions lead to selection of alternate parts of a program for activation. Thus it would be wasteful to assign a Cell to each instruction for the duration of program execution. The basic data-flow processor must have a multi-level memory system such that only the active instructions of a program occupy the Instruction Cells of the processor. In the following sections we first show how decision capability may be realized by augmenting the elementary processor; then we show how an auxiliary memory system may be added so the Instruction Cells act as a cache for the most active instructions.

Decision Capability

The organization of a basic data-flow processor without the two-level memory is shown in Fig. 9. As in the elementary processor, each Instruction Cell consists of three Registers and holds one instruction together with spaces for receiving its operands. Each instruction corresponds to an operator, a decider, or a Boolean operator of a basic data-flow program. The gate and merge actors of the data-flow program are not represented by separate instructions; rather, the function of the gates is incorporated into the instructions associated with operators and deciders in a manner that will be described shortly, and the function of the merge actors is implemented for free by the nature of the Distribution Network.

Instructions that represent operators are interpreted by the Operation Units to yield data packets as in the elementary processor. Instructions that represent deciders or Boolean operators are interpreted by the Decision Units to yield control packets having one of the two forms

$$\left\{\underline{\text{gate}},\ \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},\ \langle\text{address}\rangle\right\}$$

$$\left\{\underline{\text{value}},\ \left\{\begin{matrix}\underline{\text{true}}\\ \underline{\text{false}}\end{matrix}\right\},\ \langle\text{address}\rangle\right\}$$

A gate-type control packet performs a gating function at the addressed operand register. A value-type control packet provides a Boolean operand value to an Instruction Cell that represents a Boolean operator.

The six formats for the contents of Instruction Cells in the basic processor are given in Figure 10. The use of each Register is specified in its leftmost field:

- I instruction register
- D operand register for data values
- B operand register for Boolean values

Only Registers specified to be operand registers of consistent type may be addressed by instructions of a valid program.

The remaining fields in the Instruction Cell formats are: an instruction code, op, pr or bo, that identifies the class and variation of the instruction in the Cell; from one to three destination addresses d1, d2, d3 that specify target operand registers for the packets generated by instruction execution; in the case of deciders and Boolean operators, a result tag t1, t2, t3 for each destination that specifies whether the control packet is of gate-type (tag = gate) or of value type (tag = value); and, for each operand register, a gating code g1, g2 and either a data receiver v1, v2 or a control receiver c1, c2.

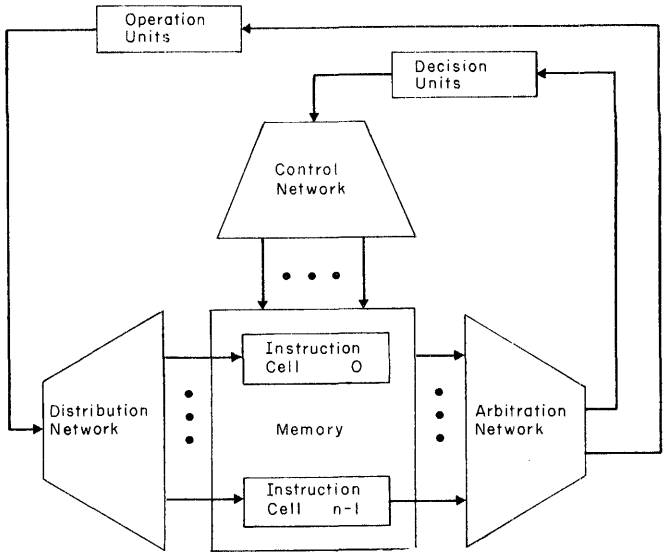


Figure 9. Organization of a basic data-flow processor without two-level memory.

The gating codes permit representation of gate actors that control the reception of operand values by the operator or decider represented by the Instruction Cell. The meanings of the code values are as follows:

code value	meaning
no	the associated operand is not gated.
true	an operand value is accepted by arrival of a true gate packet; discarded by arrival of a false gate packet.
false	an operand value is accepted by arrival of a false gate packet; discarded by arrival of a true gate packet.
cons	the operand is a constant value.

The structure of a data or control receiver (Fig. 11) provides space to receive a data or Boolean value, and two flag fields in which the arrival of data and contrc packets is recorded. The gate flag is changed from off to true or false by a true or false gate-type control packet; the value flag is changed from off to on by a data packet or value type control packet according to the type of receiver.

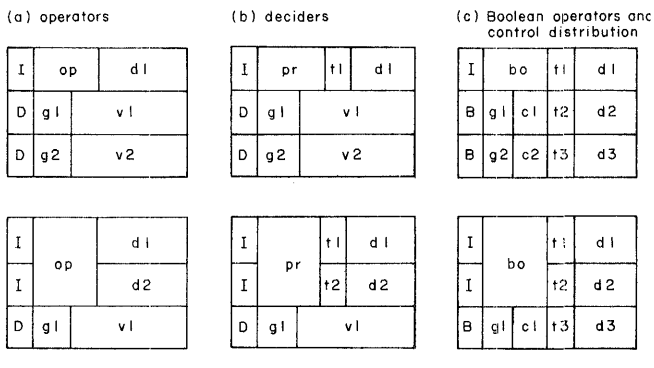


Figure 10. Instruction Cell formats for the basic processor.

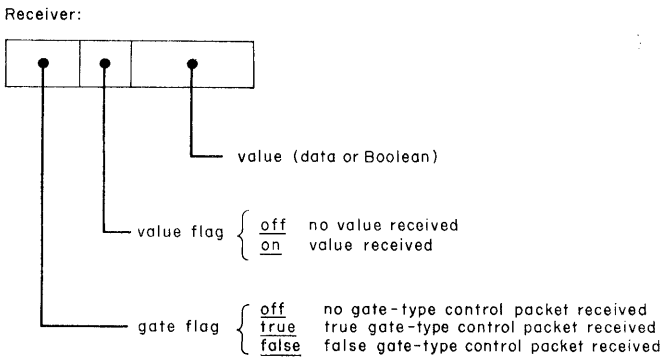
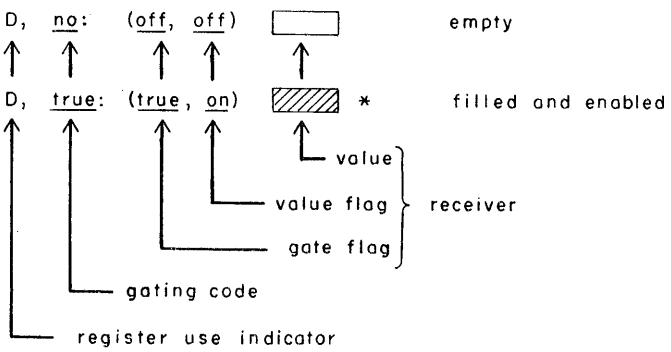


Figure 11. Structure and states of receivers.

Instruction Cell Operation

The function of each Instruction Cell is to receive data and control packets, and, when the Cell becomes enabled, to transmit an operation or decision packet through the Arbitration Network and reset the Instruction Cell to its initial status. An Instruction Cell becomes enabled just when all three of its registers are enabled. A register specified to act as an instruction register is always enabled. Registers specified to act as operand registers change state with the arrival of packets directed to them. The state transitions and enabling rules for data operand registers are defined in Fig. 12.

In Fig. 12 the contents of an operand register are represented as follows:



The asterisk indicates that the Register is enabled. Events denoting arrival of data and control packets are labelled thus:

- d data packet
- t true gate-type control packet
- f false gate-type control packet

With this explanation of notation, the state changes and enabling rules given in Fig. 12 should be clear. Similar rules apply to the state changes and enabling of Boolean operand registers. Note that arrival of a gate-type control packet that does not match the gating code of the Register causes the associated data packet to be discarded, and resets the Register to its starting condition.

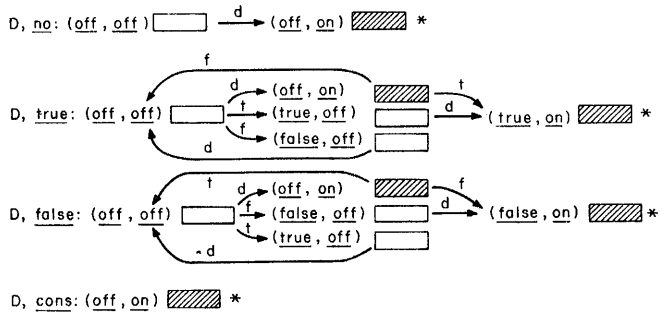


Figure 12. State transition and enabling rules for data operand registers.

The operation packets sent to Operation Units and decision packets sent to Decision Units consist of the entire contents of the Instruction Cell except for the gating codes and receiver status fields. Thus the packets sent through the Arbitration Network have the following formats:

To the Operation Units:

```
op, v1, v2, d1
op, v1, d1, d2
```

To the Decision Units:

```
pr, v1, v2, t1, d1
pr, v1, t1, d1, t2, d2
bo, c1, c2, t1, d1, t2, d2, t3, d3
bo, c1, t1, d1, t2, d2, t3, d3
```

An initial configuration of Instruction Cells corresponding to the basic data-flow program of Fig. 8 is given in Fig. 13. For simplicity, Cells containing control distribution and data forwarding instructions are not shown. Instead, we have taken the liberty of writing any number of addresses in the destination fields of instructions.

The initial values of x and y are placed in Registers 2 and 5. Cells 1 and 2, containing these values, are then enabled and present to the Arbitration Network the operation packets

$$\left\{\begin{matrix}\text{ident; } 8, 11, 14\\ \qquad\quad x\end{matrix}\right\}$$

and

$$\left\{\begin{matrix}\text{ident; } 7, 13, 20\\ \qquad\quad y\end{matrix}\right\}$$

These packets are directed to an identity Operation Unit which merely creates the desired data packets with the values of x and y and delivers the packets to the Distribution Network.

Upon receipt by the Memory of the data packets directed to Registers 7 and 8, cell 3 will be enabled and will transmit its decision packet to a Decision Unit to perform the less than function. The result of the decision will be returned through the Control Network as five control packets. If the result is true, Cells 4, 5 and 6 will be enabled and will send their contents through the

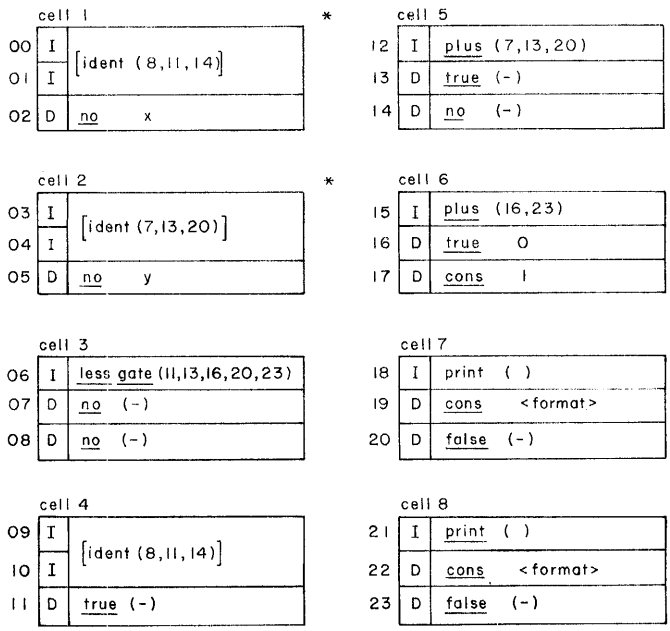


Figure 13. Instruction Cell initialization for the basic data-flow program in Figure 8.

Arbitration Network to Operation Units capable of performing the identity and addition operations. If the result of the decision is false, output cells 7 and 8 will be enabled, and cells 4, 5, and 6 will have their gated operands deleted.

Two-Level Memory Hierarchy

The high level of parallel activity achievable in data-flow processors makes a unique form of memory hierarchy feasible: the Instruction Cells are arranged to act as a cache for the most active instructions of the data-flow program. Individual instructions are retrieved from auxiliary memory (the Instruction Memory) as they become required by the progress of computation, and instructions are returned to the Instruction Memory when the Instruction Cells holding them are required for more active parts of the program.

The organization of a basic data-flow processor with Instruction Memory is given in Fig. 14.

Instruction Memory

The Instruction Memory has a storage location for each possible register address of the basic processor. These storage locations are organized into groups of three locations identified by the address of the first location of the group. Each group can hold the contents of one Instruction Cell in the formats already given in Fig. 10.

A memory command packet {a, retr} presented to the command port of the Instruction Memory, requests retrieval of an instruction packet {a, x} in which x is the Cell contents stored in the group of locations specified by address a. The instruction packet is delivered at the retrieve port of the Instruction Memory.

An instruction packet {a, x} presented at the store port of the Instruction Memory requests storage of Cell contents x in the three-location group specified by address a. However, the storage is not effective until a memory command packet {a, store} is received by the Instruction Memory at its command port, and any prior retrieval request has been honored. Similarly, retrieval requests are not honored until prior storage requests for the group have taken effect.

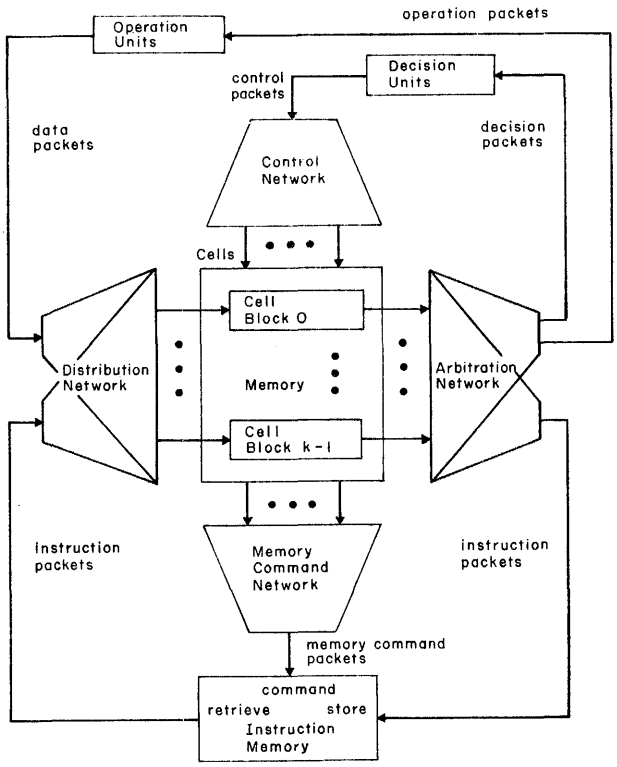


Figure 14. Organization of the basic data-flow processor with auxiliary memory.

We envision that the Instruction Memory would be designed to handle large numbers of storage and retrieval requests concurrently, much as the input/output facilities of contemporary computer systems operate under software control.

Cell Block Operation

For application of the cache principle to the basic data-flow processor, an Instruction Memory address is divided into a major address and a minor address, each containing a number of bits of the address. One Cell Block of the processor is associated with each possible major address. All instructions having the same major address are processed by the Instruction Cells of the corresponding Cell Block. Thus the Distribution and Control Networks use the major address to direct data packets, control packets and instruction packets to the appropriate Cell Block. The packets delivered to the Cell Block include the minor address, which is sufficient to determine how the packet should be treated by the Cell Block.

Operation and decision packets leaving a Cell Block have exactly the same format as before. Instruction packets leaving a Cell Block have the form {m, x} where m is a minor address and x is the contents of an Instruction Cell. The major address of the Cell Block is appended to each instruction packet as it travels through the Arbitration Network. In the same way, memory command packets leave the Cell Block with just a minor address, which is augmented by the major address of the Cell Block during its trip through the Memory Command Network.

Fig. 15 shows the structure of a Cell Block. Each Instruction Cell is able to hold any instruction whose major address is that of the Cell Block. Since many more instructions share a major address than there are Cells in a Cell Block, the Cell Block includes an Association Table which has an entry {m, i} for each Instruction Cell: m is the minor address of the instruction to which the Cell is assigned, and i is a Cell status indicator whose values have significance as follows:

status value	meaning
free	the Cell is not assigned to any instruction
engaged	the Cell has been engaged for the instruction having minor address m, by arrival of a data or control packet
occupied	the Cell is occupied by an instruction with minor address m

The Stack element of a Cell Block holds an ordering of the Instruction Cells as candidates for displacement of their contents by newly activated instructions. Only Cells in occupied status are candiates for displacement.

Operation of a Cell Block can be specified by giving two procedures -- one initiated by arrival of a data or control packet at the Cell Block, and the other activated by arrival of an instruction packet from the Instruction Memory.

Procedure 1: Arrival of a data or control packet {n, y} where n is a minor address and y is the packet content.

step 1. Does the Association Table have an entry with minor address n? If so, let p be the Cell corresponding to the entry, and go to step 5. Otherwise continue with step 2.

step 2. If the Association Table shows that no Instruction Cell has status free, go to step 3. Otherwise let p be a Cell with status free. Let the Associa-

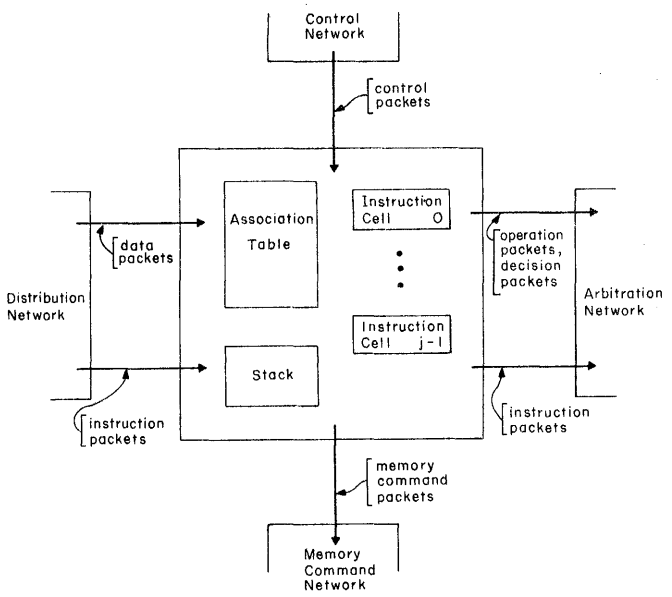


Figure 15. Structure of a Cell Block.

tion Table entry for p be {m, free}; go to step 4.

step 3. Use the Stack to choose a Cell p in occupied status for preemption; let the Association Table entry for p be {m, occupied}; transmit the contents z of Cell p as an instruction packet {m, z} to the Instruction Memory via the Arbitration Network; transmit the memory command packet {m, store} to the Instruction Memory through the Memory Command Network.

step 4. Make an entry {n, engaged} for Cell p in the Association Table; transmit the memory command packet {n, retr} to the Instruction Memory via the Memory Command Network.

step 5. Update the operand register of Cell p having minor address n according to the content y of the data or control packet (the rules for updating are those given in Fig. 12). If Cell p is occupied the state change of the register must be consistent with the instruction code or the program is invalid. If Cell p is engaged, the changes must be consistent with the register status left by preceding packet arrivals.

step 6. If Cell p is occupied and all three registers are enabled (according to the rules of Fig. 12), the Cell p is enabled: transmit an operation or decision packet to the Operation Units or Decision Units through the Arbitration Network; leave Cell p in occupied status holding the same instruction with its operand registers reset (receivers empty with the gate and value flags set to off). Change the order of Cells in the Stack to make Cell p the last candidate for displacement.

Procedure 2: Arrival of an instruction packet {n, x} with minor address n and content x.

step 1. Let p be the Instruction Cell with entry {n, engaged} in the Association Table.

step 2. The status of the operand registers of Cell p must be consistent with the content x of the instruction packet, or the program is invalid. Update the contents of Cell p to incorporate the instruction and operand status information in the instruction packet.

step 3. Change the Association Table entry for Cell p from {n, engaged} to {n, occupied}.

step 4. If all registers of Cell p are enabled, then Cell p is enabled: transmit an operation or decision packet to the Operation Units or Decision Units through the Arbitration Network; leave Cell p in occupied status holding the same instruction with its operand registers reset. Change the order of Cells in the Stack to make Cell p the last candidate for displacement.

Conclusion

The organization of a computer which allows the execution of programs represented in data-flow form offers a very promising solution to the problem of achieving highly parallel computation. Thus far, the design of two processors, the elementary and the basic data-flow processors, has been investigated. The elementary processor is attractive for stream-oriented signal processing applications. The basic processor described here is a first step toward a highly parallel processor for numerical algorithms expressed in a Fortran-like data-flow language. However, this goal requires further elaboration of the data-flow architecture to encompass arrays, concurrent activation of procedures, and some means of exploiting the sort of parallelism present in vector operations. We are optimistic that extensions of the architecture to provide these features can be devised, and we are hopeful that these concepts can be further extended to the design of computers for general-purpose computation based on more complete data-flow models such as presented by Dennis [4].

References

1. Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford, Calif., December 1968.
2. Bährs, A. Operation patterns (An extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972 (preprint).
3. Dennis, J. B. Programming generality, parallelism and computer architecture. Information Processing 68, North-Holland Publishing Co., Amsterdam 1969, 484-492.
4. Dennis, J. B. First version of a data flow procedure language. Symposium on Programming, Institut de Programmation, University of Paris, Paris, France, April 1974, 241-271.
5. Dennis, J. B., and J. B. Fosseen. Introduction to Data Flow Schemas. November 1973 (submitted for publication).
6. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974.
7. Dennis, J. B., and D. P. Misunas. The Design of a Highly Parallel Computer for Signal Processing Applications. Computation Structures Group Memo 101, Project MAC, M.I.T., Cambridge, Mass., July 1974.
8. Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. Appl. Math. 14 (November 1966), 1390-1411.
9. Kosinski, P. R. A Data Flow Programming Language. Report RC 4264, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., March 1973.
10. Kosinski, P. R. A data flow language for operating systems programming. Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September 1973), 89-94.
11. Rodriguez, J. E. A Graph Model for Parallel Computation. Report TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.

REDUCTION LANGUAGES FOR REDUCTION MACHINES

K.J. Berkling

Institut fuer Informationssystemforschung
Gesellschaft fuer Mathematik und Datenverarbeitung

St.Augustin, Schloss Birlinghoven, Germany

Summary

This paper describes a particular realization of a Lambda-Red language as a machine language. Parts of it resemble the Lambda Calculus or a transposed form of it. A constructor syntax is employed such that a linearized preorder representation of the syntax tree is the information structure on which the machine operates. Instances of reduction rules are recognized by combinations of constructors and atoms. Reduction rules with 1, 2 and 3 constructors and/or atoms have been described.

A recursive control structure forms the essential part of the implementation. There is a one to one correspondence between constructor syntax and control structure rather than a simulation of recursive structure by von Neumann type instruction sequencing. Thus the system is easily expandable by new constructors and atoms. Execution, that is the application of reduction rules, is subsumed under editing. Emphasis has been on the following pragmatic concepts: locality of action, directness, and security. There are no error messages, errors appear as irreducible expressions. There is no "RUN" instruction, instead the number of reductions to be performed is specified. Whatever happens - it is restricted to subexpressions. The user has the security that all his actions are limited and predictable.

The machine language resembles a higher level programming language, but it is "variable-free": expressions are named, not boxes which contain expressions. There seems to be a concept of late binding. However, this is not made a matter of principle, but a matter of degree. Between a general statement of a problem and the result are many intermediate representations - all in the same language - corresponding to various degrees of binding parameters. The user has complete freedom in the choice of variable names. There is a specially developped protection system which avoids confusion of variables.

The efficiency of the system is limited by the rate characters can be processed which is essentially determined by the control store cycletime. Measurements will be performed as soon as a simulation of the system is available. Three other important subjects, namely arithmetic, the definition of constants, and source sink input-output will be reported later.

1 Introduction

Programming has become the outstanding problem in computing. While technology has provided ever cheaper and faster hardware, the methodology to construct software is just emerging. Current hardware and software may be major obstacles towards a solution of the programming problem. However, there are several programming languages which are attractive because of their simplicity and expressive power: LISP[1], TRAC[2], and GPM[3]. Recently John Backus has added to this list his RED-languages. "RED" stands for reduction and denotes a special type of execution and evaluation.

The above mentioned programming languages have a common denominator which seems to be the reason for their attractiveness, namely, they are "substitutive". The concepts on which conventional hardware - and programming languages, too - are built contradict the concept substitution. Thus, implementations of substitutive systems on conventional hardware are bound to be inefficient.

It is the purpose of this paper to show

1) that it is possible to design hardware which is suitable for substitution systems,
2) that substitution is an important concept the implementation of which deserves further investigation, and
3) that the issue of higher level language machine architecture arises from the apparent inability of conventional hardware to support substitution economically.

2 Problem Identification

The following is an opiniated discussion of fundamental problem areas which are believed to be the cause of the difficulties in programming.

1) Long Range Effects

Writing a program generates a net of causes and effects which is not easily derivable from the visible representation of that program making it difficult to comprehend it. The effect caused by an instruction is long range: any part of the system may be involved.

2) Indirect Access

Operands and operators are seldom directly connected. Operands are rather referenced, or called by names and addresses. This separation permits the abstraction of programs from particular data. The construction of an algorithm therefore necessitates the construction of data access paths, algorithms to manipulate them, and algorithms to manage storage space. These latter tasks often overwhelm the construction of the algorithm itself.

3) Central Control

The control structure of a machine is that part which implements the next-state function of the system. This concept is based on the notion that a "present" state is defined at all. Thus, large systems with a central control resemble more a simulation than a "life" system. The many parts of such a system are merely enacted by a single agent which is mainly occupied with finding out what the "present" state of the system is.

4) Addressibility

As a consequence of points 1), 2) and 3) there is a need to generate, compute, and store addresses. Technology will provide larger and faster memories. Because of the limited address length of conventional hardware it is a problem to provide addressibility.

5) Machine Indenpendent Languages.

This notion is a misconception. Experience has shown that such a machine independence cannot be achieved mainly because of details and exceptions. Is machine independence a reasonable objective at all? A program consists of "instructions" directed to something real. To hide the real machine from the user may cause insecurity about what really happens.

3 Conceptual Approach to Problem Solution

We first formulate our attempts of solving the problems in rather general terms. Experience shows that theories employing short-range interactions have more success in explaining reality and are easier understood than long-range theories. The following ideas about a computing system seem therefore promising with respect to the problems listed in section 2:

Control is distributed to many parts cooperating over well defined local interfaces. These exist only to neighbour parts. The restrictions and boundary conditions which result from a realization of the system are an essential and determining factor. There are no unexplicable and opaque restrictions.

Programs and data are cohesive structures. They are modified and changed only locally. Substitution is always literal and not simulated. The system is interactive: program and data may be inspected and may be changed and modified at the discretion of the user. He will not be able to initialize changes he cannot predict or cannot oversee. There will be no addresses or variables.

4 Reduction Languages

J. Backus[4,5] has described a class of languages which seem to be prime candidates for our considerations. We disregard for the moment any particular realization of a reduction language. A reduction language L = (E,C,M) consists of a set E of expressions, a set C of expressions, and a partial function M from E onto C such that C is the set of fixed points of M. A simple example of a reduction language is the language of arithmetic expression without variables:

$$M((3+4)*7) = 49$$

Considering relizations we first have to explain M in terms of smaller steps.

A complete realization CR = (E,C,T), is a reduction language, where T is a total function from E into E, and C is the set of fixed points of T, the transition function. If $M_e = c$ then there is an integer n such that $T^n e = c$. With respect to our example we may write

$$T((3+4)*7) = (7*7)$$

$$T\ (7*7) = 49$$

Reduction languages constitute a shift of emphasis: the meaning is no longer in the states of an automaton taken on while reacting on input signales, the meaning is in the expressions and only there. But, if T is enacted by an automaton, this automaton will go through a sequence of states starting at and returning to an initial state. During that period e is not defined.

This should be contrasted with conventional instruction execution: to find out the meaning of one instruction execution the state of the whole system has to be inspected.

John Backus[5] lists six interrelated informal axiom which seem necessary and sufficient to characterize a reduction language. Property (5) needs special attention:

(5) The extended Church-Rosser property.

(a) Every terminating sequence of reductions of an expression yields the same meaning for it, and
(b) if an expression has a meaning, then every sequence of reductions on it terminates. (Property (b) is restricted to selected sequences in the usual form of the Church-Rosser Theorem.)

If anything equivalent to recursion, which cannot be relinquished, is implemented, there will always exist non-terminating reduction sequences, which a blind deterministic enactor of the transition function cannot detect. Without an enactor no expression does anything. So, we only need a human observer who always can force termination by switching off the enactor or setting a time limit. The advantage of a reduction system becomes particularly evident in this context. The result after each reduction-step is an expression e, which "means" something to the observer. He can usually decide if further reductions will terminate or not.

We have to distinguish further between "reduction" and "evaluation". On first sight these concept seem to denote the same thing. However, evaluation has the connotation that each and everything has a value. This is another pragmatic point supporting the case for a reduction system: it returns a "constant" expression c rather than an error message if it cannot apply reduction rules, and does not try to decide if this c is the result the user expected or the consequence of an error committed by the user. The expression c will be self-explanatory and exhibit why further reductions are impossible.

5 Syntax of a Reduction Machine Language

We talk about a "machine language" although there is nothing like a conventional machine language in a reduction machine. Instead, there are character strings which are transformed if certain characters or character combinations occur in these strings.

In particular, we employ a constructor syntax CS = (A,K) to define the set of expressions E. CS is a constructor syntax if

CS1) There is a subset $A \subset E$ the elements of which are called atoms.

CS2) There is a set K of constructors k which are functions from a subset S_k of E^n into E ($n \geq 0$).

CS3) For every expression $e \in E$, either $e \in A$ or there is a unique $k \in K$ and unique $e1,\ldots,en \in E$ such that $k[e1,\ldots,en] = e$.

These axioms regulate the construction of elements of the set E.

To be more specific, we only allow two-place (and one-place) constructors. So far a particular encoding or representation for constructor functions has not yet been given. We choose a direct, explicit one. Let k e1 e2 = e with juxta position as the only syntactic tool. The set of expressions becomes therefore the set of linear representations of binary trees in preorder form[6].

It is our intention to implement a Lambda-Red language, that is, a language which resembles the lambda calculus[7]. Lambda-Red languages are closed applicative languages. A reduction language L = (F,C,M) with a constructor syntax S = (A,K) is applicative iff:

AL1) A⊂C

AL2) There is a two-place constructor ap∈K such that for all e,f∈E:

M ap e f = M ap Me Mf

AL3) For all other constructors:

M k e f = k Me Mf

Using these three axioms we may reduce an expression e to a constant where the constructor ap occurs only with constant expressions: ap c d. The applicative language is closed iff there is a function

R∈ [C →[C → E]]

such that:

CAL1) R is total over C

CAL2) For every c∈C, Rc∈[C →E] is total over C.

CAL3) For all c,d∈C:

M ap c d = M ap Rc d

The function R is called representation function. A major part of a hardware implementation would consist of an implementation of this representation function R.

The function R will be undefined for many elements of C, numbers for example. The machine will take care of these cases in a manner consistent with the reduction concept. This means the machine will not stop with an incomplete reduction of the ap, on the contrary the machine will continue with other reductions after restoring the ap with its components. This has pragmatic significance as a direct in-place error indication.

Generally, only atomic elements of C will be associated to meaningful mappings C → E. In Lambda-Red languages, however, we have lambda expressions which are "applied" to arguments by substituting the argument at designated places of the lambda expressions.

5.1 Substitution

Lambda-Red languages resemble the lambda calculus and inherit all its problems caused by the introduction of variables.

We introduce a new class V of atoms called variables, and a constructor lambda represented by the character \. Variables are represented by strings of letters externally and their encodings internally. Variables are constants, thus if v ∈ V: Mv = v. This is in sharp contrast to conventional programming languages where variables are internally represented by a more or less complicated network of references to memory cells.

A lambda expression \ v e may be a component of an application

ap \ v e f

The reduction of this expression corresponds to the beta-conversion rule of the lambda calculus: f is substituted into e for free occurrences of v.

We have found by experimenting that it is convenient to have several ap-type constructors. Firstly, every constructor has a transposed form such that

ap' f \ v e

is an instance of beta-conversion and

M ap' f \ v e = M ap \ v e f

The use of transposed constructors may improve the readability of expressions.

The other differences of the ap-type constructors may be explained with reference to axiom AL2:

M ap e f = M ap Me Mf

This form of the axiom applies if e is not a lambda expression, since e has to be reduced to a constant before the representation function R can be employed. If, however, e is a lambda expression, various sequences of reductions are possible since substitution does not require constant components. The variety of ap-constructors arises if not both of their components are reduced to constants before substitution. Let g be a lambda expression in:

AL2L)
```
M ←  g f = M R2 R1g  f
M .  g f = M R2 R1g  Mf
M ∆  g f = M R2 R1 Mg f.
```

The function R1 prepares the lambda expression for substitution while the two-parameter function R2 performs it.

It is not necessary to introduce a fourth constructor for the case where the first component is not a lambda expression, since the machine is designed to detect this. So we have if e does not start with a \ constructor

AL2E)
```
M ←  e f = M ←  Me  f
M .  e f = M .  Me Mf
M ∆  e f = M ∆  Me  f
```

The constructor . corresponds to the usual ap while the constructors ← and ∆ yield still other cases. The same statements hold for the group of transposed ap-constructors, which are →, :, and ∇, respectively.

The contextual dependencies of constructors are rich enough to model all typical higher level language constructs. Moreover, the combinational properties of the constructors are paralleled by combinational properties of the machine components related to these constructors. This yields a very economic design for a rather rich structure.

So far we did not mention the problems with free and bound variables in substituton procedures. Our solution[8], which differs from all known ones, is not only effective but also efficient, because a substitution may be executed without any preparations. This method protects possible free occurrences of variables by a constructor lambda-bar from being bound erroneously while substituting. The lambda-bar constructor is represented by the character /. The user will find /v where he might have expected v alone. A lambda-bar expression can only appear as a component of a lambda expression with the same variable. The lambda-bar will automatically disappear if this lambda expression is applied to some argument.

We may now return to chapter 5.

There is also a constructor • corresponding to the DOT-operator of LISP[1] which serves to compose data structures.

A few other constructors are implemented for special purposes. The constructors # and ~ play the role of escape characters to denote numbers, the constructor ↑ is a special binding constructor and is dealt with later.

Expressions are linear representations of binary trees in preorder form. Constructors correspond to nodes, atoms to leaves, and subtrees to subexpressions. Constructors may be in a predecessor - left successor or predecessor - right successor relationship with respect to the tree structure. All

combinations are allowed with the following two exceptions:
1) A binding constructor (\') may only have variables as left-successor.
2) A number constructor may only have number parts as successors.

Primitive Atoms. The system is equipped with a set of primitive, one character atoms (leaves). Single letters serve as variables and may appear as leaves everywhere. The digits 0 to 9 are treated as integers and comprise the only numbers without the number constructors. If single digits appear in operator position, namely as that component of an application constructor which is subjected to the representation function, they are interpreted as constant functions which yield themselves if applied to arbitrary arguments.

Truthvalues are represented by single characters. They act in operator positions of the constructors (.,:,←, →) as constant functions, in operator position of the constructors (∧,∨), however as selectors in the conditional device.

We have the NIL atom represented by the right bracket] which acts as the identity function in operator position.

Members of the set of number-related operators and logical connectives are treated as constants if they appear in operand position. In operator position, however, the representation function is invoked according to their usual meaning.

There are three decomposition operators (○ ⊃ ⊂) which delete the constructor, the left subtree, or the right subtree, respectively.

Compound Atoms. Variables are formed by a sequence of letters terminated by any non-letter. (The set of non-letters contains the blank). Compound numbers start with the constructor #. Arithmetic will be handled as in the programming language TRAC².

Expressions. We are now in a position to form arbitrary binary tree structures from constructors as nodes and primitive and compound atoms as leaves. The machine accepts the preorder linear representation of such tree structures. The machine will reject any character string which is not a binary tree or can be augmented by the machine to such binary tree using NIL atoms.

6 Use of the Reduction Machine

Before we outline a possible implementation of the proposed reduction machine we will exhibit the representation of familiar higher level language constructs. We believe the proposed realization represents a rather close fit to the common usage of programming languages, such that the user does not really have to learn a new language. Although the rules are more stringent, we believe that a programm written in the proposed realization of a reduction language is more transparent and comprehendable, too.

Infix Expressions. The input of the machine accepts fully - parenthesized infix expressions. The printout appears also in this form. The internal representation, however, is a binary tree with two constructors corresponding to the left and right parenthesis and the two terms and the operator as leaves.

external	(e + f)
internal	( ) e + f . : e + f

The parenthesis ( ) and constructors . : , respectively, are made synonyms for the reduction processor. The printout processor, however, uses them to restore the external representation. The examplifying plus sign may be replaced by any expression.

The Polish prefix notation for arithmetic expression - which should not be confused with the preorder linear representation for program structures - has the following representation:

. . + e f

We are already in a position to implement the logical connectives without resorting to actual hardware. The concepts of constant and identity function are sufficient if we let the primitive constants (∧ ∨ ⊤ ⊥) play their various roles in operator and operand positions. In Figure 6.1 we have exhibited the evaluation of (A B) employing four reduction rules. Corresponding reduction rules for the OR function are M: ⊥ ∨ =] and M: ⊤ ∨ = ⊤. The transposed forms of all these reduction rules are provided, too.

Conditional. The constructor ∆ indicates a complete reduction of the operator component before the reduction process continues. This property allows us to construct a conditional

e	T^2_e
∆ ∆ f g	f
∆ ∆ f g	g

In general a predicate expression, that is an expression which reduces to a truth value, will be constructed as operator of the constructors (∆ ∇). The truth value acts then as selector. Finally, the reduction of f or g takes place. The conditional can be used correspondingly to model either the conditional expression or the conditional statement of conventional programming languages.

Lists. The constructor • corresponds to the DOT operator, . ⊂ and . ⊃ correspond to CAR (head) and CDR (tail), respectively. To facilitate listprocessing, the characters •, [, and , are made synonyms for the reduction processor such that a tree structure like

• e • f • g • h]

may be typed in and printed out as

[e, f, g, h]

The reason for encoding the NIL-atom by a closing bracket is now obvious. The list elements may be any permissible expressions including lists represented in this way.

Call by value. The constructors . and : correspond to a call by value since the operand is first reduced, then inserted, and finally the function is reduced.

Call by name. Similarly, the constructors ← and → perform call by name, since the operand is first inserted, and then the function is reduced.

It should be noted, however, that because of the lack of any side-effects whatsoever the result will be in either case the same, only the number of reductions might differ.

Functions. Functions will be constant expressions in general. Several parameters are simply listed with a lambda constructor each.

\v1\v2 -- \vn f v1,...,vn ∈ V

Arguments follow in the same sequence as the parameter variables, the corresponding ap's, however, in opposite direction:

apn -- ap2 ap1 \v1\v2 -- \vn f e1 -- en

The constructors ap1,...,apn may be selected from the set (. ← ∆). Constructor ap1 and its components are the first to be reduced, because it is the only one which has a lambda expression as operator component.

After all ap's have been reduced f has been changed such that all parameter variables are replaced by actual parameters. The reduction

of this expression results in a constant expression, that is, the value returned by the function.

Infix expressions are a special case of the application of two parameter functions. Constructors may be mixed with their transposed versions such that the same f

(e1 f e2) = ..f e1 e2

may be used in either context.

Since we do not have box-like variables in the reduction machine, it is not possible to fill a box the name of which has been transmitted to an expression by reducing this expression. However, the ap constructors offer a large variety in sequencing reductions. One sequencing method resembles procedural language.

PROCEDURAL Reduction. Assignment statements of conventional programming languages fill a box. The effect of this action extends up to the next filling. A reduction machine resembles this process by reducing one expression and substituting the result in an other expresseion. The constructor : allows as to write this down as an expression g

```
g =  : e1 \ v1
     : e2 \ v2
     : e3 \ v3
       ---
     : en \ vn
     f
```

This clearly resembles a sequence of assignment statements. The expression f, however, is not a GOTO !

The sequence of reductions is top-down because of the constructor :. The reductions affect only expression g. The results of the computations in g are either available as a modified f which may be a subexpression of another expression. Of, the expression f may be a function call. Using the constructor . a function call has a pattern like

. . . v e4 e5 e6

The reduction of g would then change f to a constant expression with results deposited in the arguments of the function call f.

The transposed form of this call

f = : e6 : e5 : e4 v

yields a pragmatically better pattern for g: there is a computation part, a memory part, and a connector part.

Further reductions become possible if a function gets substituted for v

: e6 : e5 : e4 \ v4 \ v5 \ v6 e

The memory part keeps e4, e5, e6 queued up for transmission to e.

To construct something which resembles a block or procedure we bind the connector v which gets the appropriate variable name END. The block can be inserted in a sequence of "assignment" reductions:

```
---
: e7  \ v7          "assignment"
                    reduction
A  \ END            equivalent to begin
    :  e1  \ v1  }
       ---       }  computation part
    :  en  \ vn  }
    :  e6        }
    :  e5        }  memory part
    :  e4        }
     END            connector
\  v4             }
\  v5             }  argument for block
\  v6             }
:  e8  \ v8       }
---
```

The sequencing properties of the constructor A effects the complete reduction of the computation and memory part before the block is applied to the argument (surrounding block) following the END variable. Note that the object in the reduction language which resembles a block must have an explicit interface for its output.

Recursion and Iteration. These constructs can be effected by using the binding constructor ↑, which is similar to the Lambda constructor \. There is a reduction rule associated with this constructor

↑ v e = ← \ v e ↑ v e

which reproduces an expression within itself. This is sufficient to construct recursion and iteration.

7 Implementation by Hardware

A reduction language machine may be realized in various different ways. In our first attempt we will choose the conventional separation of an active component operating on information stored in a memory component. In a next step one would try to combine these two components in one. The reduction concept seems particularly suited for a "processing in memory". But, for the time being, appropriate hardware concepts are not available.

The expressions in the foregoing sections are linear representations of binary trees. We store expressions as character strings in memory components which operate as stacks, that is, they act on push and pop signals and do not need addresses as input. An attempt[10] to directly represent trees was discarded.

We need an active component which scans expressions to find instances of reduction rules. It is not appropriate to perform this scan like an instruction counter scans an instruction stream, because the expressions are structured in subexpressions. The scan has to work forward and backward. An economic solution consists of generating in a stack called A the transpose of an expression in a stack called E.

before	after
F k e1 e2 A	E A k' e2' e1'

The transpose of an atomic expressions is usually the atom itself. Compound atoms may require special rules. The correspondence to a conventional von Neumann control structure is as follows: the constructors and atoms are the operation-codes, the expressions are the operands. There is a major difference, however. The control structure to be designed effects the transpose of an expression. This is asked for twice within this control structure. This cannot be transformed into a loop structure. The control structure is therefore fitted with a system stack called MU which serves to hold return points. The system stack has the same hardware properties as the other stacks in the system and is potentially infinite, too. We cannot tolerate a bound on the depth of expressions.

We will give a description of the control structure in the form of flowdiagrams.

Figure 7.1 shows the flow diagram for the transpose algorithm EA. Only one branch of each type is exhibited.

The basic pattern of control flow shows a character popped by stack E and "sorted" into its proper branch. This will be abbreviated by a horizontal line and solid triangles denoting the exit down for the annotated character. Pushing something into a stack is indicated by a solid triangle pointing upward. Stack markers are usually not annotated to the corresponding triangles. The correspondence is established by geometry. The particular choice of stack markers is a matter of the next lower design level.

The following concepts have proven useful in this context: A yes-exit consumates the tested character, there is no need to store the character, conveyed information is now in

the locus of control in the diagram. Once something is pushed into a stack, the control structure loses information and control is directed to a neutral point.

There is also a transpose algorithm AE. Execution of algorithm AE after algorithm EA restores all stacks to the original state.

The basic hardware to realize these algorithms consists of a processor component P and three memory components acting as stracks E, A and MU. Figure 7.2 shows a state chart of the processor component. There is one control store which translates stack marker encodings into control words. These control words contain in this simple basic model either a transposed constructor to be pushed into stack A and a signal to pop MU, or a stack marker to be pushed into stack MU and a signal to pop E. The other control store translates expression characters. Its control words contain either atoms to be pushed into stack A and a signal to pop MU, or a stackmarker to be pushed into stack MU and a signal to pop stack E. One-character atoms get simply moved from stack E to A, compound atoms will have special start and stop symbols around them. The transpose processor component may be realized as separate component interfacing with memory components, or as special state of the reduction processor.

We have seen in the preceding sections that an instance of a reduction rule can be characterized by constructor combinations with constructors or atoms. Although an expression is undefined while being transposed, its components are displayed during that process in the stacks E, A and MU in a way which is convenient for recognizing instances of reduction rules. We have represented in Figure 7.3 an arbitrary expression in a mixed form. Some parts of the expression are given as subexpressions e1 to e7, some parts as tree structure where we have to visualize the nodes as constructors. It also represents the state of the expression while being transposed. Stack A contains top-down the transposed subexpressions e4, e3, e2, e1, stack E contains top-down the expressions e5, e6, e7. Stack MU, however, contains stack markers instead of constructors. The stack markers contain also the information of being left- or right-successor. The position shown corresponds to the point * in Figure 7.1. Being at that position encodes the constructor k, which is not explicitly represented otherwise. Thus the expression is completely represented by the contents of stacks and the position of control. By inspecting the top elements of the stacks, we may determine the left- and rightsuccessor of k and its predecessor, from which we also learn whether k is a leftsuccessor or rightsuccessor. Instances of reduction rules are singled out in this manner while transposing an expression.

Execution of a reduction rule means a rearrangement of the top ends of the stacks. For example, the top element of stack MU is a marker indicating the left-successor of some constructor k2. The expression below k2 is now

k2 k e4 e5 e6

If we replace the marker by the corresponding one indicating the right-successor of k2, the expression below k2 has been changed to

k2 e4 k e5 e6

Another simple case is the identity function. Let the atom] be deleted from stack E and the locus of control in the corresponding branch of the control structure. Now we test the top element of MU for a leftsuccessor marker of the constructor . . This stack marker is automatically deleted if the test result is yes. We can see from Figure 7.3 that the resulting tree structure now contains e5 instead of .] e5.

Generally, one or more auxiliary stacks might be necessary to execute more complicated reductions. A subset of the lambda and ap-type constructors is exhibited in Figure 7.4. The only complicated operation is the transposition R1EB, which replaces free occurrences of a designated variable in an expression by a special one-character marker. The control flow brings into stack A the reduced or unreduced argument expression. The operation R2ABE is a simple transposition from B to E, except that the source is switched to A if the special one-character marker mentioned above appears in B. Actually BE is identical to R2ABE, since the special marker appears only in this context. The recursion constructor ' uses R1EB and R2ABE with the only difference that the recursion constructor together with its two components has been copied and transposed by EA to stack A. The copying process effects the duplication necessary for recursion.

The examples demonstrate the basic design methods for a reduction machine. New reduction rules can be added easily as long as there are enough codes for distinct stack markers available.

8 Interactive Computing

The most interesting property of a reduction language in the context of interactive computing is the return to the set E of expressions after each elementary reduction step. There is nothing to learn beyond the construction of elements of set E and the reduction rules. Editing is essentially designed to walk the tree represented by the input. Any editing operation can be applied to any subtree, including the order to perform reductions. Any finite number of reductions may be specified for a subtree. This solution insures that the machine always terminates at well defined points.

Acknowledgement

I am indebted to C.A. Petri who made this work possible by creating favorable conditions. Special thanks go to Mrs. E. Pless for preparing the paper on the BITS text editor.

References

[1] J.McCarthy et al
LISP 1.5 Programming Manual, Cambridge, Mass.: MIT Press, 1964.

[2] C.N. Mooers, L.P. Deutsch
TRAC, A Text Handling Language. Proc. ACM 20th Nat'l Conf. (1965) 229

[3] C. Strachey
A General Purpose Macro Generator. The Computer Journal 8(1965) 3

[4] J. Backus
"Reduction Languages and Variable-free Programming", San Jose, California, IBM Research Report RJ 1010, April 7, 1972.

[5] J. Backus
"Programming Language Semantics and Closed Applicative Languages", San Jose, California, IBM Research Report RJ 1245, July 5, 1973.

[6] D.E. Knuth
"The Art of Computer Programming, Fundamental Algorithsms", Vol. 1, Reading, Mass.: Addisom-Wesley, 1968, pp. 318-319.

[7] A. Church
"The calculi of lambda-conversion", Annals of Math. studies, No.6, Princeton Univ. Press, 1941.

[8] K.J. Berkling
"A Method to Rename Variables Using an Unbinding Operator",
(to be published)

[9] P.J. Landin
"The mechanical evaluation of expressions",
The Computer Journal, Vol. 6, No.4 (January 1964), pp. 308-320.

[10] K.J. Berkling
"A Computing Machine Based on Tree Structures"
IEEE Transactions on Computers, Vol. C-20, No.4 (April 1971), pp. 404-418.

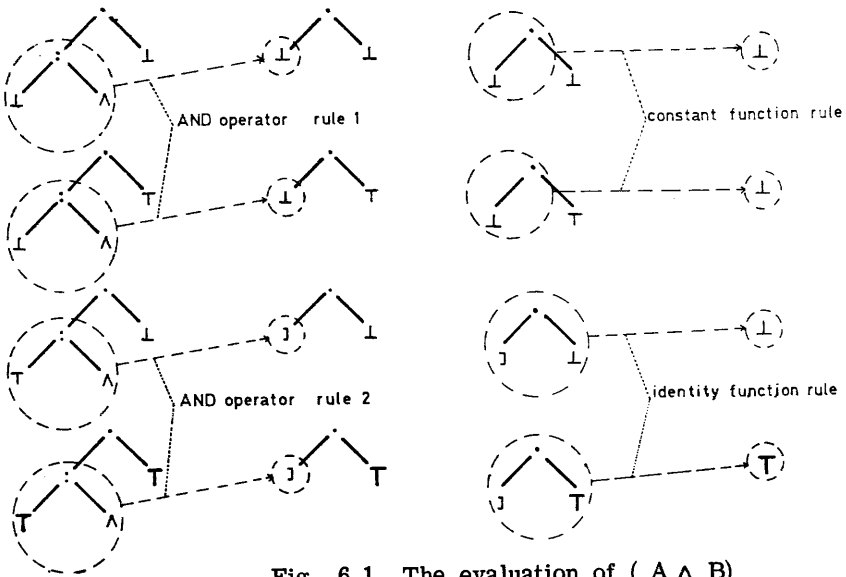


Fig. 6.1. The evaluation of (A ∧ B) = .: A ∧ B employing four reduction rules.

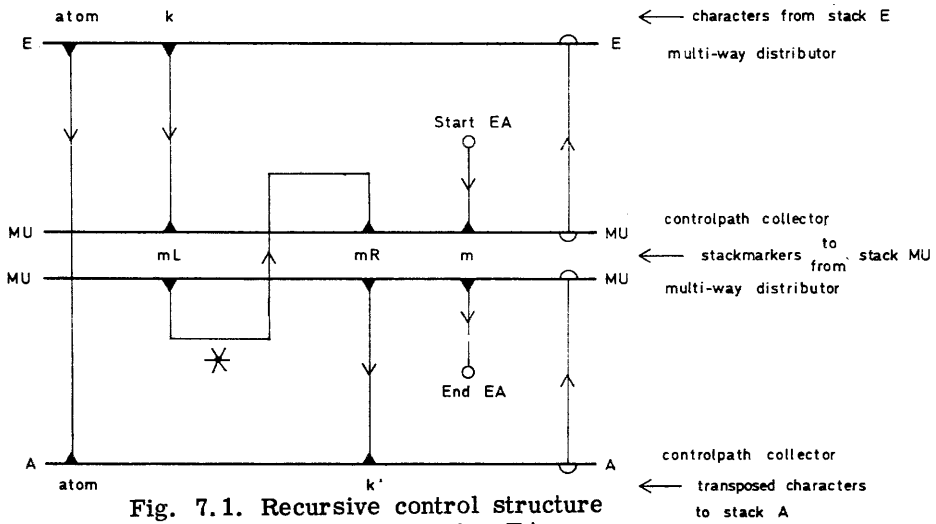


Fig. 7.1. Recursive control structure for the transposing algorithm EA using a system stack MU.

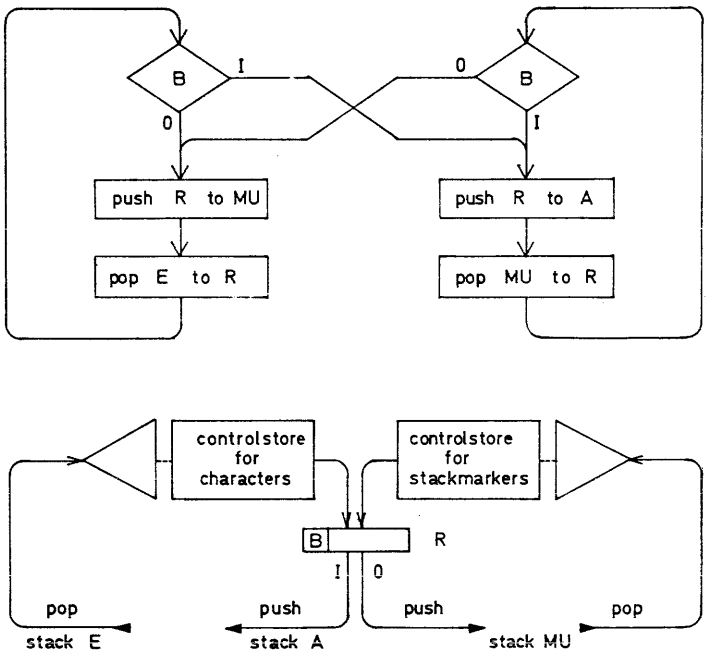


Fig. 7.2. State chart and dataflow graph for algorithm EA.

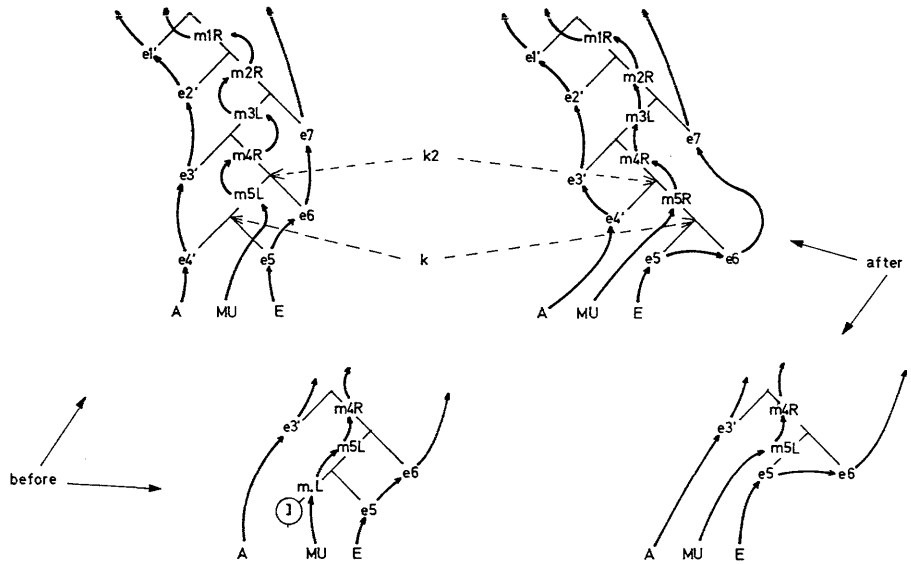


Fig. 7.3. Rearranging of tree structures by manipulating stack tops.

Fig. 7.4. Part of the reduction processor control structure for the constructors λ , ., and →.

OUTPUT DEVICES SHARING BY MINICOMPUTERS

Willis K. King
and
Fulvio Carbonaro

Department of Computer Science
University of Houston

Abstract

With the dramatic reduction in cost of central processors in recent years, the peripheral devices occupy an ever larger share in terms of cost of a minicomputer system. Sharing the expensive components among users is a time honored way of increasing utilization and hence reducing the cost of unit operation. This paper presents the results of a simulation of sharing an output device, such as the printer, by more than one processor under different workload conditions.

1. Introduction

With the coming of age of LSI technology, the cost of central processors and main memories has been reduced dramatically in recent years. Today in a typical minicomputer system the peripheral devices might represent up to 70-80% of the total cost of the system, the processor and the memory only 20-30%. This is the reverse of the cost distribution of computer systems with comparable computational power just a decade or so ago. With the advent of microprocessors, it is expected that this cost distribution will be even more lopsided. Under the circumstances, it is meaningful to reexamine the way the resources are used and develop new modes of operation to reduce unit operation cost.

Ever since the development of the first electronic computer, the trend in the industry has been to build larger and more powerful central processors. These processors, being very expensive, have to be kept productive all the time in order to justify their existence economically. Thus, large and complicated operating systems are built; multiprogramming and time sharing modes of operation are developed. Most are designed and used to maximize the utilization of the central processors. Now, with the center of gravity of cost moved towards the peripherals, it seems reasonable to use some of these methods to maximize the utilization of the more expensive peripheral devices. Recent publications [1, 2] indicate that people working with minicomputer systems are also gradually turning their attention to this problem.

There is a wide variety of applications of minicomputers, and the utilization of peripheral devices depend heavily on the particular applications. However, it is observed that there are many cases, especially in the university and research environment, where the utilization of some of the peripheral devices is rather low. It is also noted that among the peripherals, storage devices, such as disks and drums, have been used as shared devices for more than one processor with success, although the main purpose of such systems is to enable users to share the information on the device rather than sharing the device itself. The sharing of input devices such as card readers and paper tape readers is quite straightforward. The addition of a manually operated switch is sufficient to connect the device to different processors. For these reasons, we will concentrate our discussion in the paper to the output devices. moreover since in most minicomputer systems today the single most expensive component is probably the line printer, we will use the line printer as a specific example in our study. Although the following discussion will mention the line printers only, it is equally relavant to other output devices (e.g., plotters, punches, etc.) with minor modifications of certain parameters.

In the following paragraphs, we present an analysis of various printer sharing schemes. Using computer simulation the performance characteristics of each sharing method under different workloads is determined and its relative cost-effectiveness deduced.

2. Basic Sharing Methods

Since the local environment and the areas of application are different from computer system to computer system, one method of sharing might look perfectly reasonable in one case but entirely out of the question in another. We will analyse a few methods that are more general in nature, excluding those needing human intervention. However, before we discuss the individual methods, a few comments about the operation of the line printer is necessary. Unlike processor time sharing, it is unreasonable to switch the printer from one job to another at arbitrary time intervals. We considered the case of switching the printer from one job to anothe at the end of printing a page, but abandoned the idea because of the confusion it would cause in sorting out the different programs and the manual labor and time involved. If the paper can be separated immediately after a page is printed and the page can be deposited to different hoppers under program control, i.e., the paper transport works essentially like that of the card reader/punch, then this mode of printer-sharing may be viable. However, we do not know whether it is mechanically feasible or how costly it would be to build it. So we have to consider the output of a complete program as a unit and the printer can be switched from one processor to another only between such output units.

1) Direct connection - The printer will communicate directly with the processor whose output it is printing. There are two variations of this case.

a. Without buffer - This is the simplest case in which a number of processors are connected to the printer via an electronic switch. The first processor that requires service will seize the printer and the printer will not be accessible to other processors until it is released by the first one. We can use a first-in first-out queue with some priority scheme to resolve conflict of simulataneous requests, or more simply a round robin polling scheme can be built in hardware into the electronic switch.

This is, of course, not a very sophisticated method The cost of implementing it, however, is also low. Especially in the case of the round robin polling scheme A modulo counter driven by a clock can be used to scan the processors in turn for print requests. As soon as a request is detected, the counter will stop incrementing and the printer will be connected to the processor asking for service.

b. With buffer - For those systems, in which every processor has a mass storage device attached on line, a

buffer for the print output can be used to increase the performance. In this case, the processors will not ask for printer service until a job is completed and they do not have to stop and wait if the printer is busy when they request for service. A flag will be set and when the printer is free, it will recognize the flag and interrupt the processor to perform the printing. Alternatively it can be designed such that the processor can attempt to seize the printer only at the end of a job. If after the first job, the processor is unsuccessful in its attempt to seize the printer, it will wait until the completion of its second job before it tries again to ask for service. If it seizes the printer, it will then have output of both jobs printed before it releases the printer again. This method is similar to the spooling system [3] although in spooling the problem is mainly to maximize the utilization of a processor which might be driving several printers, while in our case here the job is to maximize the utilization of the printer by having it shared among several independent processors.

The cost of implementing this scheme is much higher than the previous one if mass storage devices have to be installed just for this purpose. However, if each of the processors already has a disk attached, it will probably not be too difficult to find the buffer space there. If the processor already has a tape cassette system installed and has room for expansion the cost of adding a drive is still relatively low as compared to the cost of a line printer. In addition to the hardware cost, one also has to include the software overhead of storing the output to the mass storage and later on fetching it out again for printing.

2) Single control - The printer will be under the control of a single processor. The print outputs of all the other processors will be communicated to the controlling processor which will store them in a mass storage and print them out later following some priority schedule. If the processors are sharing a mass storage device at the same time and each has direct access to the mass storage, then the only thing the controlling processor needs to do is to examine some flags that have been previously set by the other processors indicating a job in the buffer is waiting to be printed. If the controlling processor has sole control of the mass storage device as well, then each processor must have a communication link with the controlling processor built-in. In the extreme case when the controlling processor spends most of its time controlling the printing process, it may be more cost-effective if a dedicated processor (perhaps a microprocessor) is used just for that purpose. This method resembles the direct coupled system DCS [4,5] and the attached support processor system ASP) [6].

3. Simulation And Analysis of the Results

A software simulator has been built to study the behavior of the various ways of printer-sharing under different workloads. For this study, the workload is characterized by the time a job may spend in printing activities on the one hand and all other non-printing processing on the other. For every different workload, an eight-hour day is simulated for a particular system configuration. More specifically the following assumptions are made for the simulation:

1) The model is empty at the beginning of the simulation of every eight hour work day.

2) Interarrival times of jobs to the processor are made to follow a Poisson distribution with a mean interarrival time slightly smaller than the total throughput time such that the system will be slightly overloaded.

3) Service times for the processor, the printer and data transfer to and from buffers follow exponential distributions with the following mean service times:

$\frac{\mu_{pc}}{\mu_{pr}}$ has been set to vary from .2 to 5 for each configuration.

$\mu_{bf} \simeq 1/10 \mu_{pr}$ when applicable.

where μ_{pr} = mean printer service time

μ_{bf} = mean data transfer time to or from a buffer

μ_{pc} = mean time for all non printing processing

The time when the processor is waiting for the service of the printer and the printer idle time is recorded.

The result of the simulation is shown in Figures 1-5. Figures 1-3 show the cases when one printer is shared by 2 to 4 processors. In Figures 4 and 5 we show the more general processors. In all the graphs, the accumulated processor waiting time and printer idle time expressed as a fraction of the total available time are plotted against the ratio

$\frac{\mu_{pc}}{\mu_{pr}}$ We call this ratio, $r = \frac{\mu_{pc}}{\mu_{pr}}$, the workload ratio.

Now let T_t = the total processor time available for the system during a work period

T_w = The accumulated processor waiting time during the same period

T_o = the accumulated overhead

then $P = \frac{T_t - T_w - T_o}{T_t}$

is a measure of performance of the system.

Let C_s = the total cost of the printer-sharing system with n processors

C_I = the total cost of a system with n processors, each connected to a printer of its own.

then $C = C_I / C_s$ is a measure of cost advantage of the printer sharing system.

The cost-effectiveness E of the printer-sharing system can be expressed as

$$E = P.C = \left(\frac{T_t - T_w - T_o}{T_t}\right) \cdot \frac{C_I}{C_s}$$

The cost of the various components in a computer system is difficult to assess. According to a recent survey [7] the price of a minicomputer configured for the end user with 16 k byte of main memory ranges from just under $5000 to almost $20000. The cost of the printers varies even more widely [8]. However, it is expected that the cost of processors will come down by 15% a year in the next five years [7] while the cost of printers will decline at a much slower rate. It seems reasonable to consider the ratio of the cost of the components

$$C_{pr} : C_{pc} : C_d : C_t : C_i =$$

$$1 : 0.5 : 0.5 : 0.1 : 0.1$$

where C_{pr} = Cost of printer

C_{pc} = Cost of Central processor

C_d = Cost of disk

C_t = Cost of tape cassette drive

C_i = Cost of interface

The cost of n processors each connecting to a printer will be

$$n.C_{pr} + nC_{pc} = 1.5n$$

For direct connection without buffer

$$C_s = nC_{pc} + C_{pr} = 1 + 0.5n$$

For direct connection with tape cassette as buffer

$$C_s = nC_{pc} + C_{pr} + nC_t + nC_i$$

$$= 1 + 0.7n$$

For single control

$$C_s = nC_{pc} + C_{pr} + nC_i + C_d$$

$$= 1.5 + 0.6n$$

From these we can calculate the cost effectiveness of each case. for example, with the workload ratio = 1,3 we have in Figure 6 E versus n for different methods of printer sharing.

4. Conclusion

We have presented the result of the simulation of processors sharing printer(s) under different workload conditions and using different sharing methods. We used the result of our simulation to deduce the cost-effectiveness of each configuration and found that in most of the cases printer sharing will provide the users some advantages. To a large extent, the study confirms our intuitive idea of the system behavior. The simulation gives us the concrete evidence and the quantitative basis to decide whether and how we could share an output device such as the printer. If the cost of processors and mass storage devices actually comes down much more rapidly than that of the peripheral devices as it is generally predicted today, we believe output devices sharing by users of small computers will be a common practice in the future.

References

1. J. Vaucher, C. Rey, "A Hardware Laboratory for Computer Architecture Research," Proc. of the 1st Annual Symposium on Computer Architecture, Dec. 1973.
2. M. Tsuchiya, S.S. Yau, M.J. Gonzalez, "Use of a Computer Network as Peripheral Devices," Digest of Papers, CompCon 74.
3. R. F. Rosin, "Supervisory and Monitor Systems," Computing Surveys, vol. no. 1, March 69.
4. F.R. Baldwin, W.B. Gibson and C.B. Poland, "A Multiprocessing Approach to a Large Computer System," IBM System Journal, vol. 7, Sept 62.
5. E.C. Smith, Jr., "A Directly Coupled Multiprocessing System," IBM System Journal, vol. 2, Sept-Dec. 1963
6. IBM System/360 Attached Support Processor System, Version 2, System Manual, IBM Corporation, Data Processing Division, White Plains, N.Y.
7. L.C. Hobb, "Minicomputer Survey," Datamation, July 1974.
8. M.L. Stiefel, "Printers-A Technology Profile," Modern Data, vol. 4, no. 2,3, Feb. 1971.

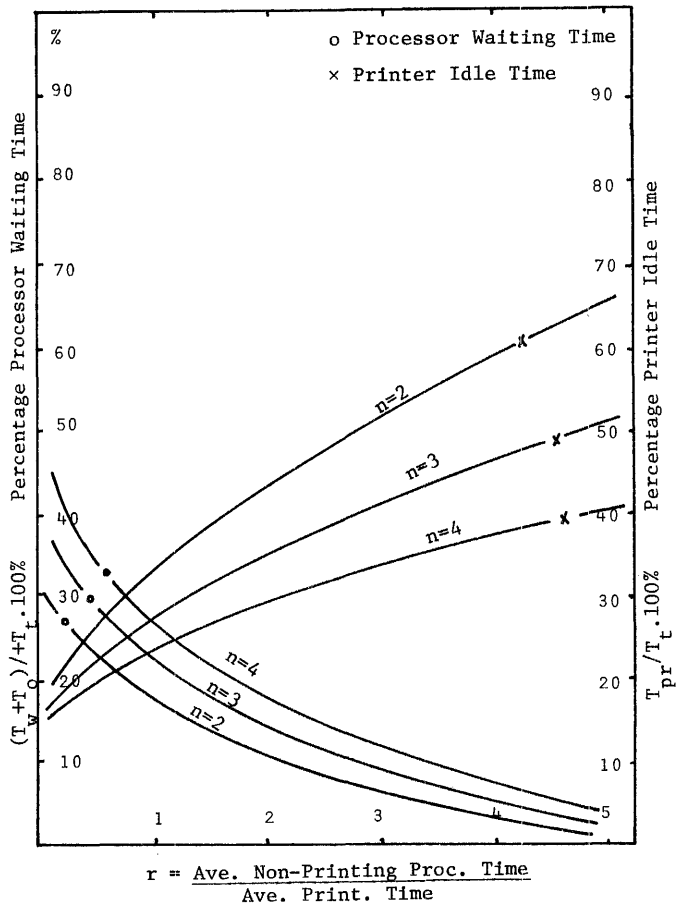


FIG. 1 DIRECT CONNECT WITHOUT BUFFER, 1 PRINTER

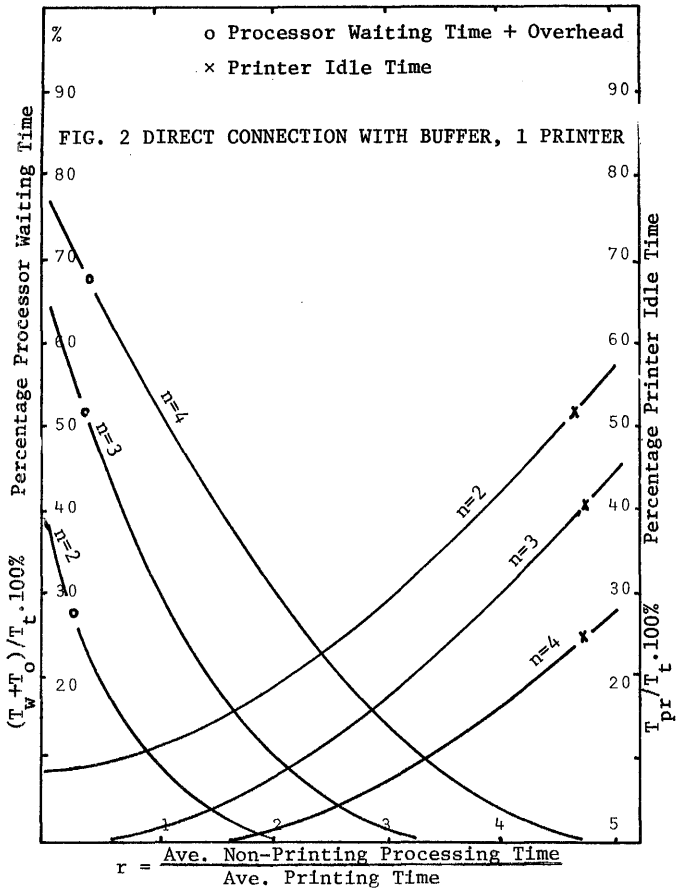


FIG. 2 DIRECT CONNECTION WITH BUFFER, 1 PRINTER

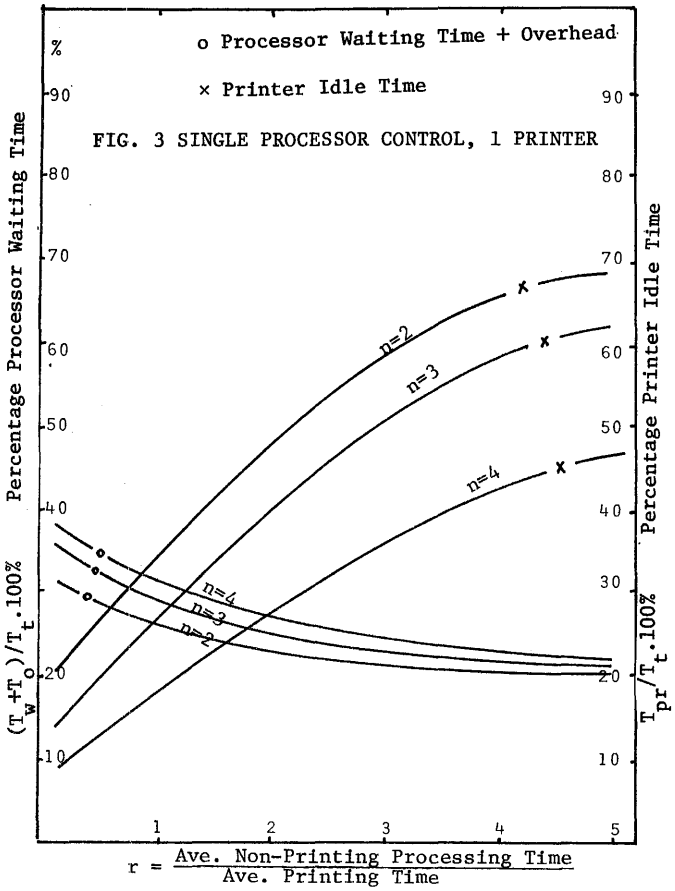


FIG. 3 SINGLE PROCESSOR CONTROL, 1 PRINTER

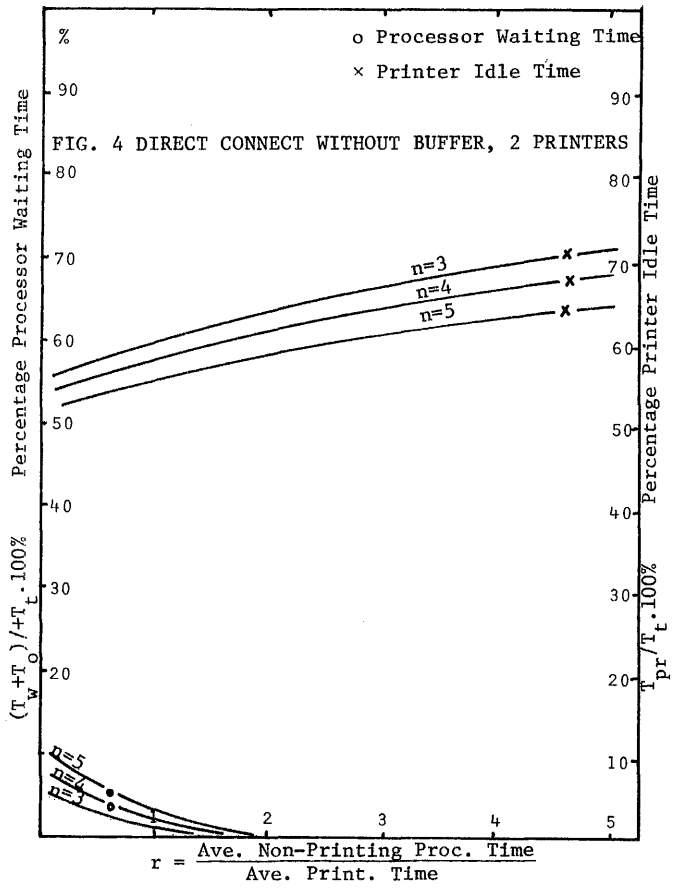


FIG. 4 DIRECT CONNECT WITHOUT BUFFER, 2 PRINTERS

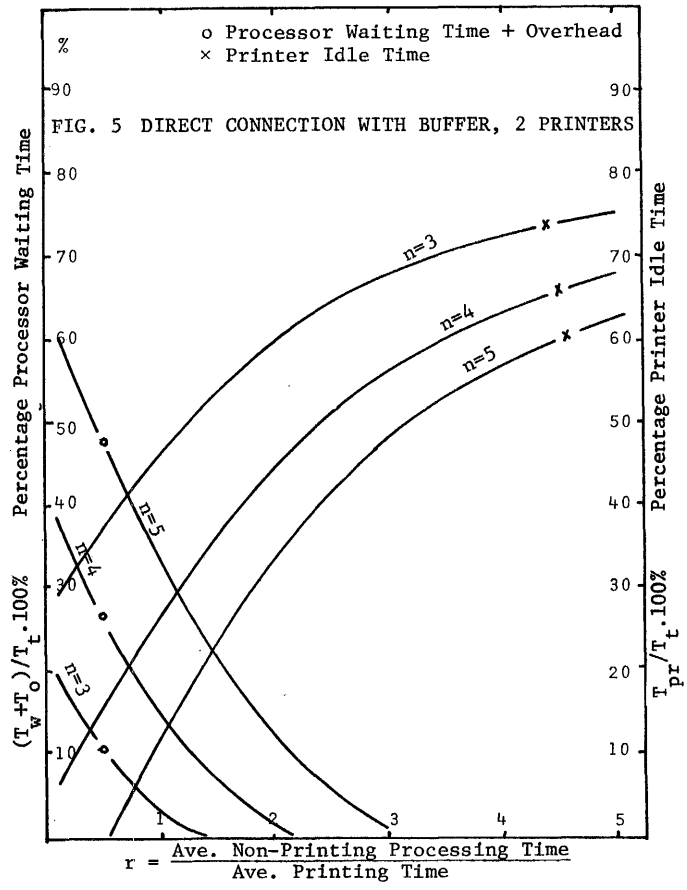


FIG. 5 DIRECT CONNECTION WITH BUFFER, 2 PRINTERS

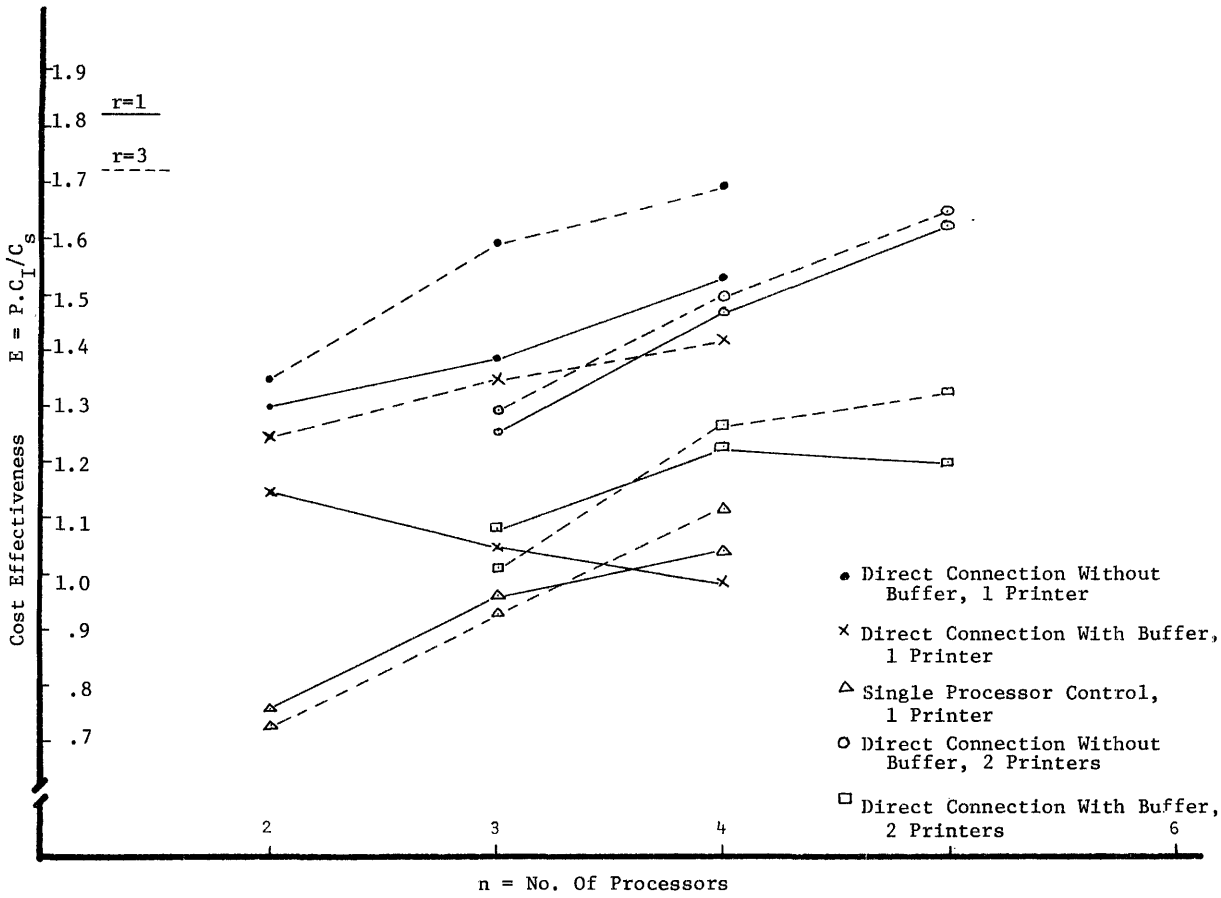


FIG. 6 COST EFFECTIVENESS VERSUS NO. OF PROCESSORS

ON RELATING SMALL COMPUTER PERFORMANCE TO DESIGN PARAMETERS

S. Rannem
V.C. Hamacher
S.G. Zaky
P. Connolly

Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Ontario, Canada

Abstract

This paper presents a method for correlating performance measures of small computers to design parameters. An experiment is described in which execution times and memory space requirements are gathered for three small benchmark kernels when run on fifteen small computers. The benchmarks are drawn from three different application areas. All of them exercise only the CPU ↔ memory area of the machines, and I/O operations are not involved.

Using standard regression analysis techniques, this data is then used to calculate coefficients in empirical equations which relate the performance measures, time and space, to easily quantifiable design parameters of the machines.

1. Introduction

The minicomputer designer, or user, continuously tries to answer the following question: For a given application area, which minicomputer design is the most appropriate from among some set whose members appear superficially to be about equally applicable? In this paper, we will describe the use of quantitative statistical techniques in determining the relative importance of various design parameters in the CPU ↔ memory area of minicomputers. The results are potentially useful as an aid in objective selection procedures; and, more fundamentally, as an indication of a rationale for design decisions. There appears to be no lack of new minicomputer designs. But it seems that there has not been enough effort in the area of quantitative comparative analysis that shows the effect of design alternatives on system operation.

The use of statistical techniques for quantitative computer architecture evaluation has not been very common. The book edited by Freiberger[1] is a collection of papers from a 1971 conference that did have statistical computer performance evaluation as its theme. On the other hand, many articles, (2)-(8), have dealt with a variety of techniques that have generally resulted in qualitative performance evaluations that are based on variables ranging from buyer confidence in the manufacturer through detailed subsystem and interconnection properties of a computing system. A recurring technique in this latter group is the use of well chosen benchmark programs, either actual or synthetic, that purport to characterize an existing or intended workload for a computer system. If the benchmarks are truly representative of the workload, they can be a practical tool for system evaluation from two standpoints; first, they are more accurate than the use of simplified analytic system models which might be used for performance prediction, and second, they are less expensive than reasonably detailed simulations of the actual systems under consideration.

Since most computer systems, even those based on minicomputers, are quite complex structures, it is not at all clear that there is a best way to evaluate performance. A compounding factor is the intended use of the performance results. The three main reasons for doing a performance study are: (a) to aid manufacturers in making design decisions in a new system, (b) to aid buyers in selecting a system from a number of alternatives, and (c) to facilitate improvements to an existing system.

The purpose of this paper is to investigate the feasibility of applying statistical regression analysis techniques to the objective determination of empirical relationships between performance measures and structurally determined variables that are normally considered as design parameters. Thus we are continuing the theme of the articles in (1). The study was performed over a range of fifteen computers, most of which are classified as minicomputers, but including some micro-computers and some medium-sized machines. Our results are potentially useful in indicating trends that might be significant in areas (a) and (b) mentioned above.

The remainder of the paper is organized into four sections. In section 2, the experimental method is specified and the data gathering technique is explained. Section 3 discusses the computer design factors that are felt to be the dominant ones in determining performance. They are used as the independent variables in the construction of empirical equations by regression techniques which give a best fit to the observed values of execution time and core space required. This section also includes a discussion of the limitations of the technique as well as its value in indicating the quantitative effect of various design changes on performance.

2. Experimental Method and Data Gathering

In performing regression analysis, two problems need to be considered. First, a measure for the performance of a given machine in a given application area should be obtained. Second, it is required to develop a quantitative representation for the various design parameters.

A survey of available evaluation methods is given in reference (4). For the purposes of this study, the kernel approach was adopted as the main evaluation scheme. A kernel is taken to mean a structurally identifiable subset of a benchmark program, chosen such that it accounts for most of the execution time of the benchmark. The adoption of the kernel technique is based on the following considerations.

1. It is required to evaluate the basic processor hardware architecture rather than the overall performance of a computer system.
2. Most minicomputer applications involve a dedicated processor programmed directly using assembly language. Therefore, compilers, operating systems and multi-programming considerations have only limited relevance to the results.

It is important to note, however, that the use of kernels for evaluation limits the validity of the

results to compute-bound applications. If it is required to include I/O or any other system considerations, the more general benchmark program approach can be used. Assuming that performance parameters have been obtained by one of these methods, the regression analysis described in the next section can be applied.

Memory requirement and execution time, rather than throughput, are often the relevant parameters in evaluating minicomputer performance. Therefore, total memory requirement in bits and execution time measured in memory cycles are used in this study as the performance parameters.

The problem of obtaining quantitative representations for machine design parameters will be presented in the next section. In the remainder of this section, the benchmark kernels used in this study and the performance data gathered will be described.

Three small synthetic benchmarks from different areas of computing were formulated. The flow-charts for the complete benchmarks were constructed without any particular machine in mind. Even though these flow-charts were reasonably detailed, there was enough flexibility at the coding phase to exploit the particular strengths of the instruction set and CPU facilities of a given machine. None of the benchmarks involved input/output operations. The benchmarks were actually run on only one machine. This aided the verification of the logical correctness and completeness of the flow-charted algorithms and also permitted the extraction of appropriate kernels. These kernels were programmed at the machine level by a single programmer (the first author) on fifteen different machines, mainly minicomputers. Because of the small size and easy understandability of the kernels, execution times and memory space requirements were relatively easy to compute for these kernels from their code on all of the fifteen machines. Memory space was recorded in bits and execution time was recorded in number of memory cycle times. The latter parameter allows a concentration on machine design features, and the actual speeds of the various technologies used in the machines have no effect on the results.

Brief descriptions of the full benchmarks and kernels extracted are as follows:

Benchmark No. 1 - High Precision Arithmetic

The wordlength of the machines ranged from 8 to 24 bits. As an indication of their ability to handle high precision arithmetic, the first benchmark performs 48-bit integer division by a standard technique. The flowchart for the main routine is shown in Figure 1.

It was found from an analysis of this benchmark that over 90% of the execution time was spent in the three subroutines MULAD, MULSB, and SHFTM. These routines thus constituted the kernel for benchmark no. 1. We will not give any detailed explanation of the full benchmark or the kernel. The inclusion of Figure 1 is meant to give an indication of the size and level of complexity of the types of kernels used in this study. Similar sizes and complexities apply to the following two benchmarks.

Benchmark No. 2 - Character Manipulation

The crucial problem in character manipulation applications is to use storage effectively, and at the same time facilitate fast processing. For example, in a machine where the smallest addressable data unit is 16 bits, two bytes or characters must be packed per data unit if good storage efficiency is to be maintained. However, this will impede the accessing of a single character. In the case of small computers, memory space is quite often limited, while execution times are not so critical. It was therefore decided that on all machines, maximum core packing of character strings would be used in this benchmark.

The actual processing problem that constituted this benchmark was the construction of a file of records to be printed. This print file was extracted from certain fields of a base file. A format list specified which fields to be selected. Linear searching of both the format list and the base file were involved in the process of constructing the print file. A structurally identifiable set of routines that accounted for about 65% of the execution time was declared as the kernel of this benchmark. This set was mainly concerned with string and substring accessing methods, and format list searching. Other standard character string operations were also included.

Benchmark No. 3 - List Processing

This benchmark exercises the ability of the machines to handle scattered data items. The particular algorithm that determined the benchmark was binary tree insertion and balancing. An 80% execution time kernel was identified and used on all machines.

3. Selection and Quantification of Machine Parameters

There is a great deal of judgement needed in specifying a complete enough set of suitable machine design parameters for purposes of characterizing performance. There are some obvious choices. Wordlength will clearly affect execution time performance on benchmark no. 1, and byte addressability should be a definite asset in benchmark no. 2. Beyond a few observations such as these, it is difficult to be confident about the real value of the more secondary design parameters, in terms of having consistent and significant effects on performance values. Quantifiable parameters such as number of general registers, address bits per memory operand reference, etc., should be included.

The following is a brief summary of the independent variables used in this study and their quantitative representation.

1. Memory wordlength: This is defined as the maximum number of bits per memory probe. This normally corresponds to the instruction length of the machine.
2. Minimum number of bytes per memory probe: This identifies whether or not the machine has byte addressability.
3. Add time: Because of variation of add time with addressing modes, etc., it is defined to be the execution time, in memory cycles, of the add instruction with one operand in a processor register, and the other operand in a directly addressable memory location. Note that the add time as defined here is representative of basic machine operations such as (i) fetch from memory to a register, or (ii) any operation involving a CPU held operand, the arithmetic unit, and a directly addressable operand in memory, where the result is retained in the CPU.
4. Register Strength: This is defined as the total number of registers which can be used for accumulator or pseudo-accumulator functions. Note that, to simplify the definition, no attempt was made to use a weighting scheme for different register functions, or to include such things as index registers. It is felt that such capabilities should be reflected in the strength of the instruction set.
5. Addressing Capability: Some of the parameters that can be used to represent the addressing capability of a machine are given below.
   a) Number of bits in direct address field.
   b) Maximum address reach per memory cycle of instruction fetching (measured in number of address bits).
   c) Maximum address reach per instruction word.
   d) Number of address modification bits per

instruction.

In the study reported here, machine performance is correlated with parameters (c) and (d) defined above.

6. Instruction Set Strength: Two parameters based on numerical scoring methods have been used:
   a) Arithmetic capability: One point is given for each of the following instructions: Add, Subtract, Multiply and Divide.
   b) Logical Capability: Scores are assigned as follows:

      0 points: no logical capability

      1 point: AND, conditional skip, and unconditional jump instructions

      2 points: AND, OR and simple conditional branch instructions

      3 points: All of the above plus Exclusive OR and complete conditional branch instructions

      4 points: All of the above plus bit test and bit manipulation instructions

The above definitions of arithmetic and logical capabilities are basically those given in reference (6).

Note that in the above choices an attempt has been made to keep the design parameters reasonably independent. The soundness of any particular choice, however, can only be judged by the goodness of the regression fit in the final results. Table I identifies the fifteen machines used in the study and lists their parameter values for the set of parameters that ultimately were found to account for most of the variation in the observed performance results.

4. Regression Fits to Experimental Data

Two forms of empirical equations that yield good statistical fits to the data of Table II are:

$$\text{(a)}\quad Y = a_0+a_1x_1+a_2x_1^2+a_3x_2+a_4x_2^2+\ldots+a_{2n}x_n^2 ,$$

$$\text{(b)}\quad Y = b_0x_1^{b_1}x_2^{b_2}\ldots x_n^{b_n} .$$

The dependent variable, Y, represents either execution time or memory space, and the independent variables, $x_1, x_2, \ldots, x_n$, represent machine design parameters. The a_i and b_i coefficients are routinely determined so that the equations represent a best fit (in a least squares sense) to the experimental data.

The computed execution times in memory cycles and the memory space requirements in bits for each of the benchmark kernels on each of the machines of Table I are shown in Table II. After a number of trials using different sets (usually of size 2 or 3) of the variables in Table I, the parameters that accounted for most of the variation in the performance measures were determined for each benchmark. For example, practically all the variation in execution time of benchmark kernel #1 can be explained by means of the CPU properties memory wordlength, register strength and add time. Adding additional CPU properties as independent variables, e.g. arithmetic capability or length of direct address field, did not significantly improve the fit. A summary of the significant parameters for each performance measure in the three benchmarks used is given in Table III. The best fit equations using these parameters as independent variables are given below.

Benchmark #1

$$\text{TIME: } Y = 2.29\times10^6x_1^{-2.26}x_4^{-0.276}x_3^{0.640}$$

$$\text{SPACE: } Y = 1.59\times10^4x_1^{-0.990}x_4^{-0.190}x_7^{0.400}$$

Benchmark #2

$$\text{TIME: } Y = 1.78\times10^3x_2^{1.48}x_3^{0.687}x_4^{-0.155}x_1^{-0.290}$$

$$\text{SPACE: } Y = 2.13\times10^3-964x_2+385x_2^2+2.97x_1+0.807x_1^2 +18.7x_4-0.296x_4^2$$

Benchmark #3

$$\text{TIME: } Y = 1.35\times10^5x_4^{-0.343}x_1^{-0.885}x_3^{0.497}$$

$$\text{SPACE: } Y = 6.14\times10^3-657x_5+33.3x_5^2-252x_6+41.7x_6^2 -149x_4+6.83x_4^2$$

In all of the above equations, the x_1 variables are those of Table I. An indication of the relative contributions to the fit by any of these variables can be found by calculating their respective indices of partial determination. These values are given in Table IV. While most of the results are as anticipated, some of them deserve a few comments. Table III shows that add time is one of the significant parameters in all three benchmarks. This can be attributed to the fact that add time as defined earlier is representative of the execution time of the average memory reference instruction.

It is useful to plot the empirical equations superimposed on the experimental data to provide a visual notion of the goodness of fit. Figure 2 shows the plot of the family of curves that fits the space requirement data from benchmark no. 3. For various constant values of address modification bits and number of registers, the curves plot memory space requirements versus the address reach property which is the important property for that benchmark and performance measure.

A more detailed description of the results of the regression analysis can be found in the first author's master's thesis.

5. Conclusions

A technique based on regression analysis has been presented, whereby small machine preformance can be correlated to its design parameters. The technique is based on quantitative representation of the various machine parameters, such as wordlength, number of registers, etc. The validity of this approach has been demonstrated by the fact that it was possible to account for almost all variations in the performance measures through appropriate selection of machine parameters with which these measures were correlated. For example, the standard error in the best regression fit of Figure 2 is 13.3%. Note that this figure has been adjusted to account for the small number of machines used. The resulting correlations are consistent with known qualitative trends. However, the method provides quantitative relationships, namely the regression equations in section 4, which are potentially useful in making rational design trade-off decisions.

The availability of a family of curves such as those of Figure 2 can also be useful in assessing the relative merits of two candidate designs in a specific application area. Since the machine parameters can be easily determined for any given machine, its performance measures can be obtained from the appropriate set of curves. Such a quick evaluation of performance should be very useful in machine selection.

It should be pointed out that use of the particular set of regression analysis results presented in this paper is subject to some limitations. The most important limitation is the small sample size (15 machines used). With this number of samples it is only possible to obtain reliable correlation results for the design parameters that have a major effect on performance. Less important parameters, that is those with a small index of determination, require a larger sample for their effect on performance to become measurable. This situation is reflected in the fact that many indices of determination in Table IV have been reduced to zero after adjustment for sample size.

6. References

1. Freiberger, W. (editor), Statistical Computer Performance Evaluation, Academic Press, New York and London, 1972.
2. Timmreck, E.M., "Computer Selection Methodology", ACM Computing Surveys, Vol. 5, No. 4, Dec. 1973.
3. Ferrari, D., "Workload Characterization and Selection in Computer Performance Measurement", Computer, July/August 1972.
4. Lucas, H.C. Jr., "Performance Evaluation and Monitoring", Computing Surveys, Vol. 3, No. 3, September 1971.
5. Olivier, R.T., "A Technique for Selecting Small Computers", Datamation, Vol. 16, January 1970.
6. Butler, J.L., "Comparative Criteria for Microcomputers", Instrumentation Technology, Vol. 17, October 1970.
7. Hillegass, J.R., "Standardized Benchmark Problems Measure Computer Performance", Computers and Automation, January 1966.
8. Joslin, E.O. and Aiken, J.J., "The Validity of Basing Computer Selections on Benchmark Results", Computers and Automation, January 1966.
9. Rannem, S., "A Study of Factors Contributing to Performance in Small Computer Central Processors", M.A.Sc. Thesis, Dept. of Electrical Engineering, University of Toronto, Toronto, Canada, 1973.

TABLE I

Machine Properties

		x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Machine	Name	Memory Wordlength (bits)	Minimum bytes/ memory probe	Add time (memory cycles)	Registers	Address Reach per memory word of instruction	Address modification (bits)	Arithmetic capability	Logical capability
1	PDP-8	12	2	2	1	8	1	1	1
2	Kongsberg 400	16	2	2	6	8	3	3	4
3	SAM	24	3	2	10	14	4	4	2
4	Datapoint 2200	8	1	7	5	3.2	0	2	3
5	H 112	12	2	4.5	1	8	1	1	1
6	PDP-11	16	1	4.2	6	8	3	2	3
7	DC 6024	24	1	2	5	15	3	4	3
8	Nova	16	2	3	4	8	3	2	1
9	Modcomp III	16	1	3	15	8	4	4	4
10	SEL 804A	24	3	2	5	15	3	4	3
11	SPC-12	8	1	5	5	6	0	2	2
12	Varian 520	8	1	3	7	5	3	2	2
13	Interdata 1	8	1	3	1	4	1	2	3
14	GE-PAC 4010	24	3	2	1	15	4	4	2.5
15	DataMate 16	16	2	2	2	8	3	4	2

TABLE II

Execution times (in memory cycles) and memory space requirements (in bits) for the 3 benchmarks

Machine	Benchmark No. 1		Benchmark No. 2		Benchmark No. 3	
	Time	Space	Time	Space	Time	Space
1	11448	1224	3627	1776	19305	2448
2	4809	1072	2515	1872	7805	2016
3	1535	864	4640	2904	7288	2640
4	49345	2104	3086	1664	41294	3584
5	23475	1452	7027	1880	29767	2532
6	6071	944	1597	1584	10796	2088
7	1365	768	936	1968	5778	2472
8	5460	1072	2818	2192	10000	2416
9	4616	1120	1012	1520	7256	1904
10	1576	936	4989	3024	8942	3048
11	33462	2064	2123	1752	23043	2928
12	23866	1576	1890	1352	19633	2192
13	41532	3016	2223	1456	50698	4248
14	2962	1248	6793	3456	12615	3144
15	6296	1328	2533	1872	9440	2240

TABLE III

Summary of Machine Properties showing definite and consistent relationships to Performance Measure

Performance Measure	Benchmark No.	Machine Properties in order of Significance
Execution time	1	Wordlength, Registers, Add time
	2	Bytes/Probe, Add time, Registers, Wordlength
	3	Registers, Wordlength, Add time
Memory Space Requirements	1	Wordlength, Registers, Arithmetic
	2	Bytes/Probe, Wordlength, Registers
	3	Address Reach, Registers, Address Modification

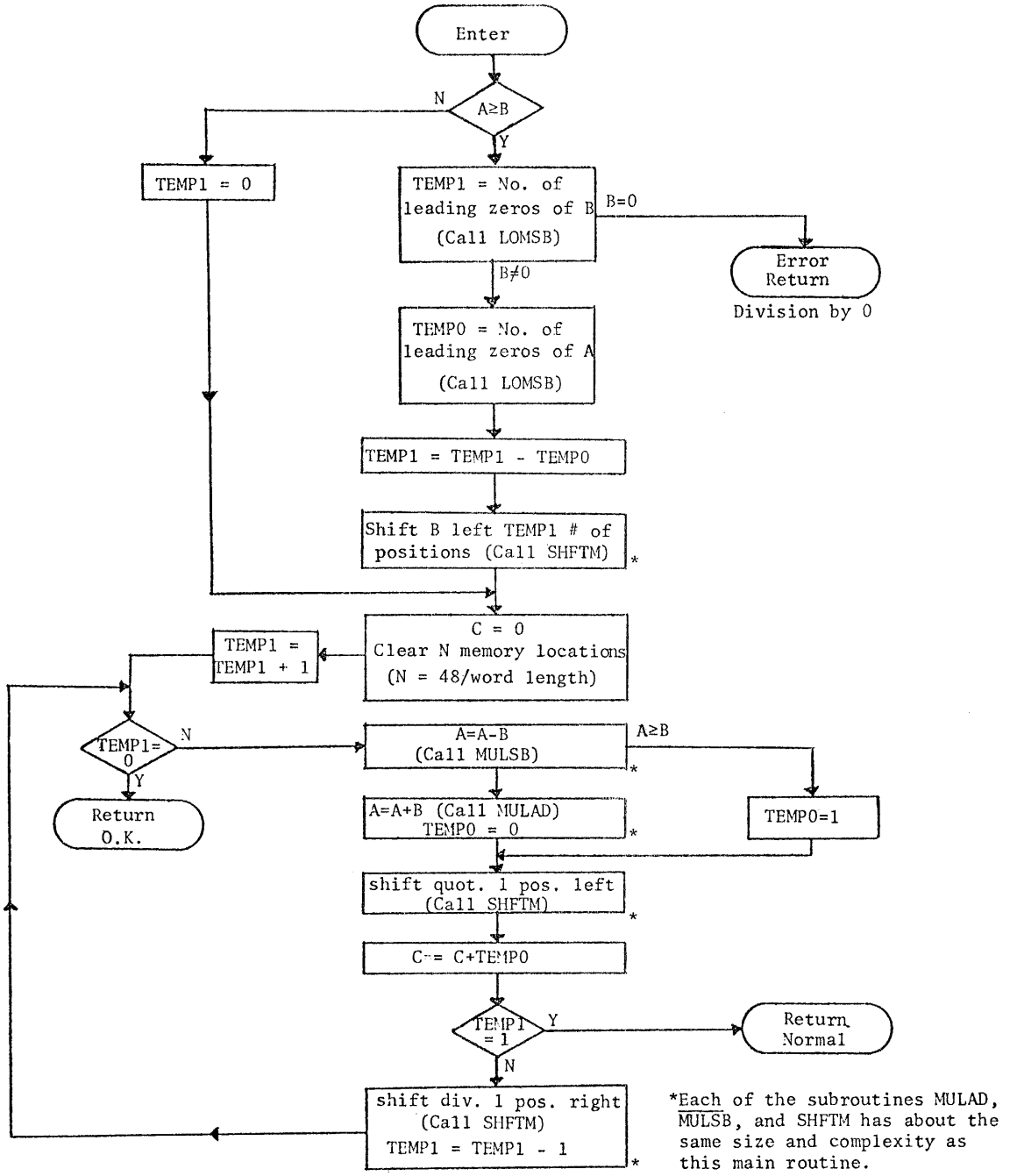


Fig. 1 Main Routine for Benchmark No. 1.

TABLE IV

Indexes of Partial Determination

Performance Measure	Central Processor Property	Index of Partial Determination (%)					
		Unadjusted			Adjusted		
		BM1	BM2	BM3	BM1	BM2	BM3
Execution Time	Wordlength	97.18	28.74	62.49	96.41	0.0	47.49
	No. of Bytes/ Memory Probe		91.57			86.39	
	Register Strength	81.60	48.39	66.93	76.58	19.72	53.70
	Add Time	72.10	69.30	34.25	64.49	52.24	7.95
	Logical		13.02	8.78		0.0	0.0
Memory Space Requirements	Wordlength	91.29	41.60		88.91	0.0	
	No. of Bytes/ Memory Probe		85.58			66.35	
	Register Strength	69.99	32.40	37.81	61.81	0.0	0.0
	Address Reach		44.21	67.97		0.0	43.95
	Address Mod. Capability			12.49			0.0
	Arithmetic	64.24			54.49		

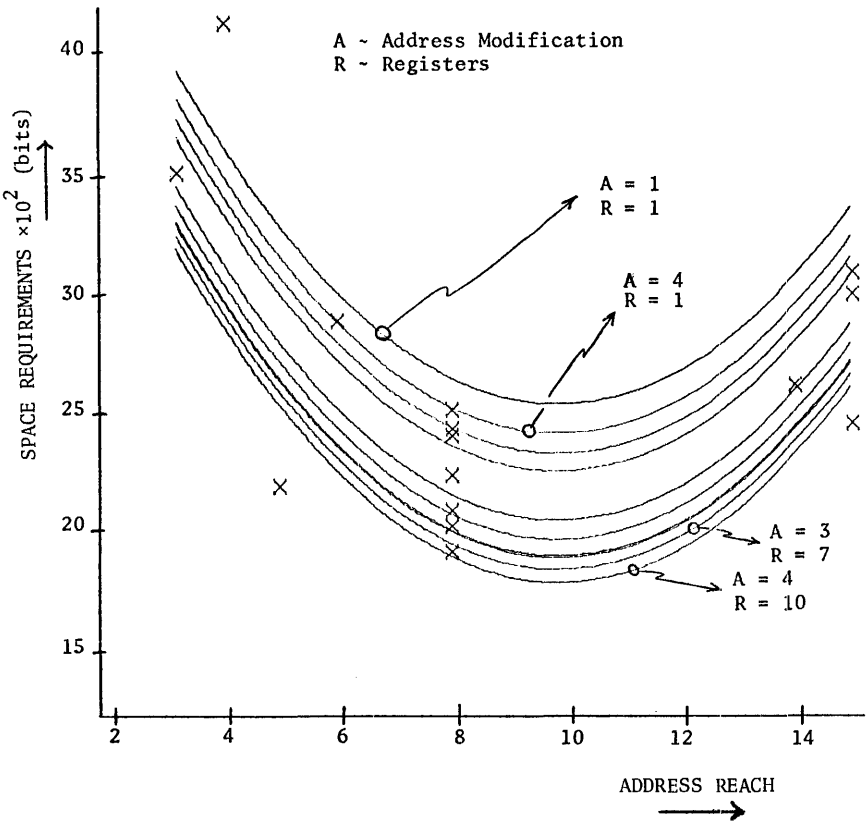


Figure 2: Regression Curves for Space Requirements on Benchmark No. 3

Statistical parameters indicating "goodness of fit":

Unadjusted Std. Error = 258
Adjusted Std. Error = 353
Max. deviation = 558
Unadjusted Std. Error (%) = 9.7
Adjusted Std. Error (%) = 13.3
Unadjusted Index of Determination (%) = 82.5
Adjusted Index of Determination (%) = 69.3

ADVANTAGES OF STRUCTURED HARDWARE

Harold W. Lawson Jr
Mathematics Institute
Linköpings Högskola
Linköping, Sweden

Bengt Magnhagen
Datasaab and
Systems Technique Institute
Linköpings Högskola
Linköping, Sweden

ABSTRACT

The proper structuring of the implementation levels in a digital computer system is an important attribute for all aspects of the product. That is to say, the design, development, testing, production and maintenance of the product are facilitated by a well-structured design of the software and hardware components. This paper discusses digital computer system structuring in general, followed by descriptions of the logical and physical architecture of the DATASAAB FCPU (Flexible Central Processing Unit). The implementation of a vector arithmetic instruction is presented to provide a more thorough insight into the operational aspects of the FCPU. Finally, the benefits derived from the structuring of this hardware product are presented.

Keywords: Computer Architecture, Microprogramming, Structured Hardware, Computer System Complexity.

1. INTRODUCTION

A significant number of suggestions have been made in the past several years about structured programming and the factors that affect programmers in designing, constructing, debugging and maintaining of programs [1, 2, 3]. We should not forget that programming is just one of the constituents in the implementation of a program solution that uses a digital computer system. There exist several levels of hardware structure as well as several programming levels (high-level language, assembly language, microprogramming language) in a complex digital computer system. The need for constraining complexity, through structuring, exists at all levels.

To illustrate the levels in the distribution of the computing system and to define the complexity problem, let us consider Figure 1. We must consider the complexity in two dimensions and strive to reach an appropriate structure for the "total system". From this figure, we define horizontal complexity and vertical complexity. A significant portion of computer science in the future must be directed toward studying both inter and intra level complexity [4, 5, 6]. From the hardware point of view we must develop "Reasonable Machines" [7].

In this paper we shall consider the logical and physical structure of a medium-scale CPU product. We shall discuss the advantages of the structuring of this "reasonable machine". Of course the term "reasonable" is a relative one. We consider this machine to be reasonable for the potential "users" of the machine. That is, those who microprogram the interpreter solution to various classical and new target systems and to those who must maintain the systems.

2. LOGICAL STRUCTURE OF THE DATASAAB FCPU

The logical structure of the FCPU has been discussed elsewhere in greater detail [8, 9]. However, we shall consider a brief overview of the logical structure in this section by discussing the following list of attributes of the FCPU.

1. Highly encoded (vertical type) microinstruction formats.
2. Asynchronous organization.
3. Data path width of 64 bits.
4. Writable control storage.
5. High-level hardware facilities.
6. Logical storage operations.
7. Hardware adaptability (for special requirements).
8. A high level oriented microprogramming language.

In order to provide a time overlapping facility in the utilization of FCPU hardware, and also to facilitate the use of highly encoded microinstructions, we decided upon the use of an asynchronous internal organization. That is, the FCPU is divided into several units each of which can be active simultaneously. The various units of the FCPU are illustrated in the block diagram of Figure 2. Synchronization registers are used to pass data path width values of 64 bits from one unit to another. These registers, which act as semaphore variables [10], are referred to as To-and-From Registers (TFR's).

Microinstructions which are 32 bits wide are read two at a time from the writable control storage and prepared for execution in the various units via the control unit in a microinstruction streaming manner. The decision as to whether a microinstruction can be executed and whether the stream can continue, can be determined from a number of predicates which include whether the addressed unit is busy, the state of a referenced TFR registers, whether the microinstruction is a branch, etc.

Due to the separation of CPU functions, there is no clock timing dependence upon the CPU as a whole. Therefore, we can easily introduce higher-level hardware facilities which have variable length execution times into the various CPU units. Examples include the use of bit manipulation, high speed multiplication, decimal arithmetic and editing functions. Logical storage operations are used so that the storage can be treated as a linear space without fixed word boundries.

The design of a target system implementation can include the use of hardware adaptability termed Variable Logic Sets (VLS) for those implementations where hardware assists for performance goals are required. VLS structures are also implemented in an asynchronous manner so that more sophisticated instruction decoding and addressing techniques can be accomodated without affecting global timing considerations.

The hardware is highly structured around the asynchronism. This structuring enables the microprogrammer to understand the FCPU structure in a much simpler manner than a complicated horizontal (minimally encoded) type of organization. However, the microprogrammer must still have a good knowledge of the FCPU architecture. Consequently in order to simplify microprogramming, we developed a machine dependent, high level oriented microprogramming language called ML [11].

In accordance with the complexity discussions in section 1, we have strived to make this a well-structured language. The instruction cycle, execution cycle organization of a target instruction interpreter is structured through the use of a START microinstruction for initiating target instruction execution and a DO CASE statement [12] for bounding the initial execution sequence for execution-cycles of individual target instructions or sets of target instructions. A skeleton of this microprogram organization appears in Figure 3. The DO CASE (n) is used as a declaration of a maximum of n microinstructions per case. In this manner, the VLS hardware, activated by the START microinstruction, can easily develop an entry address into this branch vector. If the interpretation is completed within the alloted number of microinstructions, a START microinstruction is used to start the next instruction cycle. Otherwise, a branch can be made from the branch vector to continuation microcode.

As you can determine from the previous discussions there exist several motivations for the structuring of the FCPU. Additional motivations and experiences are discussed in the following sections. The "reasonability" of the FCPU at least from a training standpoint has been proven by teaching people to construct productive microprograms in a surprisingly short period of time. This training has taken place in both academic and commercial environments. Many of the motivations for the flexibility aspects of the FCPU are similar to those discussed for other microprogrammed computer systems [13, 14].

3. PHYSICAL STRUCTURE OF THE DATASAAB FCPU

The first use of the FCPU is as the central processing unit of the medium-scale DATASAAB D23 target system. This general purpose computer system is a successor to the previous lines of medium-scale systems produced and mainly marketed in Scandinavia by Datasaab, namely the D21, D22, D220 and D223 systems. The resulting physical structure has been oriented toward the use of electronics to achieve sufficient micro-level performance in a flexible system. Other attributes which have been directives for the physical structure of the hardware are:

- Adaptability, modularization and "add-on" facilities.
- Target independent, microprogrammed, maintenance panel.
- Futuristic interface interconnecting fast units in an asynchronous environment.
- Easy failure diagnostic.
- High reliability.
- Short development and production time.

Figure 4 shows the disposition of the cabinets in the D23 system. As you can see, the physical structure looks like the logical.

The modularization and adaptability includes the facility of moving the boxes around in the cabinets and/or append cheaper or more powerful ones, due to the independence of interconnection interface cable lengths and standardized interface.

The FCPU is a separate processor that mainly communicates with the environment via the Main Storage Unit (MSU), but has positions in an "External Input Output" part for bit communication. The MSU interface consists of two identical interfaces, each with 32 bits data width due to the 32 bits MSU banks. The field manipulation facility is used in implementing logical storage addressing and works with the D23 target system which utilizes a basic word length of 24 bits.

The asynchronism means that each unit has its own control part, but most status and status manipulation are handled by the main Control Unit (CU). Because of the high internal speed and physical sizes there exist no clock pulse distribution, not even inside the separate units. Timing is constituted by delay lines. "Hand shaking" is used in the asynchronous interfaces throughout the system which results in high speed performance, greater control of activity and is helpful for "trouble shooting".

Microprogrammability in a read/write Control Storage (CS) makes it possible to give the FCPU different identities, that is, programming of hardware. For instance, during hardware test the identity is a microprocessor "TEST", not a D23 target system test. A specific facility is the microprogrammed maintenance panel. On the panel it is possible, through microprogram control, to display every register or whatever text is desired. The panel is designed to permit manual control on both the microprogram level and on the target program level. In addition, the panel has two standard IO-interfaces, so that the maintenance personnel can connect ordinary peripheral units, or telecommunication equipment, to initiate and analyze test program runs (locally or remote).

The control storage is $\leq$ 16K, 32 bits, and can be divided in two parts; one high-speed (and expensive) and one low-speed (and less expensive). The low-speed part uses the same storage boards as in the Main Storage Unit, and of course it is possible to use only low-speed or high-speed control storage.

4. DESIGN, DEVELOPMENT, TESTING, PRODUCTION AND MAINTENANCE OF THE FCPU

During all phases of the work with the FCPU we have exploited the structuring aspects of the product.

The logical design was made by a small group of highly experienced people, representing both hardware and software.

This mixture of personnel has helped to soften the hardware facilities and vocabulary. During the design phase the ML (microprogramming language) translator (MITRAN) and FCPU simulator (MISIM) were developed, and because of this close cooperation between hardware and software project members, the hardware specification was thoroughly penetrated from a microprogram consistency point of view. In addition, to gain a better understanding of the objects and relationships of the FCPU, a program was developed through use of the CADIS system [15], which is a tool for developing associative data structures.

The physical designers stated very early that due to short development time, easy and inexpensive production, high reliability and maintenance requirements, the design should include: 19" boxes, one standard board size with mixed wirewrapped/plated intraconnections, TTL and S-TTL, common power part plus one separate regulating module per box and extra high common mode for interconnections. Due to the asynchronism between the units it was possible to develop and test the different units rather independent of each other. One specific man then was responsible for the important intra- and interconnections (including the To-and-From Registers).

During the testing phase the engineer utilized debugged test programs since they already had run successfully in the simulator (MISIM). To shorten the testing time the engineer simulated his logic for logical and time "hazard" errors before the hardware was available for testing.

The testing of the FCPU, consisted of three phases:

1. Parallel test of the separate units (AU, CU, CS, FU, VLS).

2. Test of joined units (CS, CU), (CS, CU, AU), (CS, CU, AU, FU), (CS, CU, AU, FU, VLS23).

3. Test of microprogram for system identity and performance.

Phase 1 and 2 utilized the asynchronism so to run test sequences on separate "test controllers" delivering sequences of microinstructions to test the units off-line.

Phase 2 started with the joining of CS and CU with extra long interconnection cables and proceeded in the order indicated above. The FU had been tested against the MSU and I/O system separately before integration to the FCPU. The input/output media used during testing were a card reader (microprogram loading) and a line printer connected directly to the maintenance panel. A specific debugging facility of interest is the TRACE function microsubroutine that after an execution of a microinstruction stores variables of interest.

For phase 3 the panel was prepared to step microinstructions and/or to step target instructions. In addition, microprograms could be loaded via the MSU from disc, tape etc.

In the production line we exploit the asynchronism to produce and test units separately and then use the substitution method to verify system function, that is, to exchange modules an already working reference FCPU.

The documentation was well coordinated between the development and production departments because of the well structured hardware. From a database for the developed products documentation files, control information is directly fed to the production machines.

For maintenance, there exist several utility functions, which include the following:

- Input-Output interfaces on the FCPU panel prepared for local or remote control and analysis.

- A panel display for 256 characters, possible to display the content of all the FCPU registers and its CS.

- TRACE function.

- STEP function on microprogram level and target program level.

- Asynchronous units with well defined interconnection interfaces and highly encoded (vertical) microinstructions.

- Testprograms, for separate units or joined units, in ML and Target Language with well prepared test points.

5. APPLICATION EXAMPLE

As stated earlier, the first usage of the FCPU will be as an emulator of the DATASAAB D23. Special application oriented extensions to the D23 and new target systems will provide for powerful target level instructions with high semantic content. To illustrate this capability, and a high semantic content instruction, we shall consider the inner loop microcode and timing for the following simple vector operations.

ASUM

$C_i = A_i + B_i$, i = 1, 2,....., n

CASUM (Cumulative Sum)

$C_i = C_i + A_i + B_i$, i = 1, 2,....., n

The Field Access Unit (FU) as pictured in Figure 2, provides an address retention and stepping capability in four ACR (Address Control Registers). That is, a FU microinstruction can specify that the addresses used in reading or writing a logical storage cell (1-8 bytes) can be incremented by an element length. These FU operations are overlapped with operand fetching or storing and element processing in other FCPU asynchronous units. The inner loop microcode for the two operators is displayed in Figure 5. Note that the FU READ and WRITE instructions (1, 3 and 6) specify stepping (STEPR) and saving of the incremented address in ACR registers named ASTREAM, BSTREAM and CSTREAM.

Elements read are placed in AU TFR registers AVALUE and BVALUE whereas elements written are taken from the FU TFR called CVALUE. Counting of the number of elements processed (n) is done in the counting register CRO.1 via a CUP (Control Unit Processing) microinstruction (5), and testing for completion is accomplished through the name DONE which is equated to the Zero Sense of a counting register operation (8). The MIX (Microprogram Index Register) is used to hold the opcode specifying which operator is to be executed (2, 4). ASUM and

CASUM are two of several vector operator subroutines sharing the same address control code. The DO CASE statement indicates the branch vector structure (four microinstructions per case) of the operator subroutines. ENTER and RETURN statements control the FCPU subroutine stack. Overflow checking is performed for each element processed (7, 1.2).

The execution timing for ASUM and CASUM is displayed in Figure 6. The microinstructions are numbered with their corresponding numbers from Figure 5. Note that a microinstruction is readied for execution every 100 nanoseconds (unless a branch occurs). The notation

indicates a timing delay. That is, the microinstruction cannot proceed until some other event has been completed. The overlapping of time is interesting, particularly in case of instructions 4, 1.3 and 2.4 which, due to overlapping, are transfers of control with no time penalty. Subroutine overhead is eliminated in this microprogram organization.

The overlapping here provides an average microinstruction rate of approximately 170 nanoseconds. This time must be viewed as varying quite widely based upon the semantic content of the target operations involved and the organization of the microinstructions. Remember that microinstructions for the FCPU perform higher level functions and it normally requires fewer microinstructions to perform the execution of a sophisticated target instruction than on previous microprocessors. The incorporation of these vector operations into FCPU microprograms provides a performance factor of about 3.5 to 1 over D23 target language programs executing the same operations and makes the FCPU competitive with many much more expensive scientific processors in performing operations on these regular information structures.

6. SUMMARY AND CONCLUSIONS

Figure 7 provides a summary of the major logical and physical structure of the FCPU as used in the D23 system. Such diagrams are in aid in understanding complex systems. We have learned the importance of structuring in a formal way. A structure contains objects and directed relations in between them. For instance "benefit of A is B" or directly from figure 7 "implementation of physical structure is FCPU and Add-on".

Because of the formal way of structuring, it is possible to handle the objects and relations with computer programs such as CADIS [15]. In the development phase and also in the early production phase more detailed programs of structure developed using CADIS, were useful in handling construction changes.

Through the experience of the FCPU product development cycle at DATASAAB, we have learned that there is a strong motivation for well-structured hardware. Certainly, the larger the scope of the product, the more the motivation for structuring. It appears that structuring in both horizontal and vertical directions (as presented in the introduction) is an important means for controlling system complexity.

ACKNOWLEDGEMENT

We would like to thank our collegues at DATASAAB, who have contributed to and have learned to appreciate the benefits of the structuring of the FCPU.

REFERENCES

1. G. Weinberg, The Psychology of Computer Programming, van Nostrand Reinhold, New York, 1971.

2. Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, October 1969. Editors J.N. Buxton and B. Randell.

3. O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Structured Programming, Academic Press, 1972.

4. P.E. Danielsson, Microprogramming: A Hardware Point of View, Proceedings of the ACM International Computing Symposium, North Holland Publishing Co, Amsterdam, September 1973.

5. R. Hartenstein, Hierarchy of Interpreters for Modelling Complex Digital Systems, Geschellshaft für Informatik, Hamburg, October 1973.

6. E.W. Reigal and H.W. Lawson Jr, At the Programming Language - Microprogramming Interface, Proceedings of the SIGPLAN/SIGMICRO Meeting, May 1973.

7. R.F. Rosin, The Significance of Microprogramming, Proceedings of the ACM International Computing Symposium, North Holland Publishing Co, Amsterdam, September 1973.

8. H.W. Lawson Jr and B. Malm, A Flexible Asynchronous Microprocessor, BIT vol. 13, no 2, June 1973.

9. H.W. Lawson Jr and B. Malm, The DATASAAB Flexible Central Processing Unit (FCPU); Background, Concepts, Basic Design and Applications, Saab-Scania Report GM-72:295, November 1972. Also published in the Infotec State of the Art Report 17 on Computer Design, London 1974.

10. E.W. Dijkstra, Cooperating Sequential Processes, Programming Language, ed. F. Genuys, Academic Press, 1968.

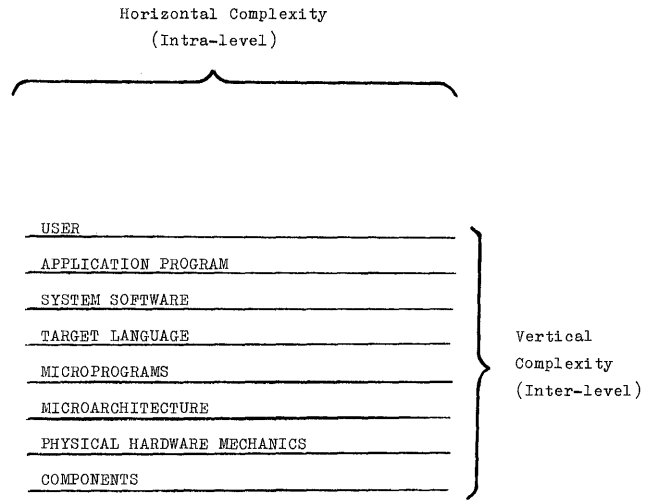
11. H.W. Lawson Jr and L. Blomberg, The DATASAAB FCPU Microprogramming Language, Proceedings of the SIGPLAN/SIGMICRO Interface Meeting, May 1973.

12. W. McKeeman, J. Horning and D. Wortman, A Compiler Generator, Englewood Cliffs, N.J., Prentice Hall, 1970.

13. H.W. Lawson Jr and B.K. Smith, Functional Characteristics of a Multi-Lingual Processor, IEEE Trans. Comput., vol. C-20, July 1971.

14. W.T. Wilner, Design of the B1700, Proceedings of the FJCC, Anaheim, California, 1972.

15. J. Bubenko, O. Källhammar and S. Berild, An Interactive Program System for the Manipulation of Associative Datastructures (in Swedish), CADIS working paper 32, IB-ADB70, Report no 25, 1970.

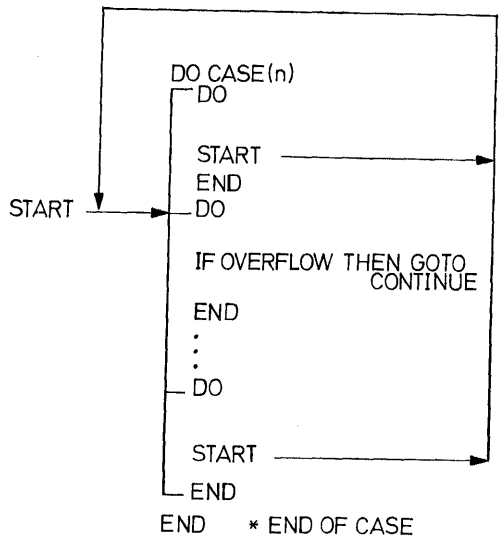


DISTRIBUTION OF DIGITAL COMPUTER SYSTEM COMPLEXITY

Figure 1

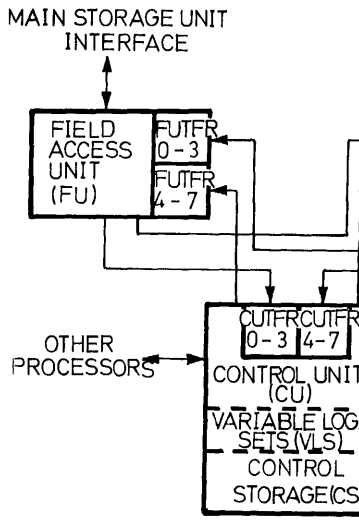
BLOCK DIAGRAM
FCPU

Figure 2



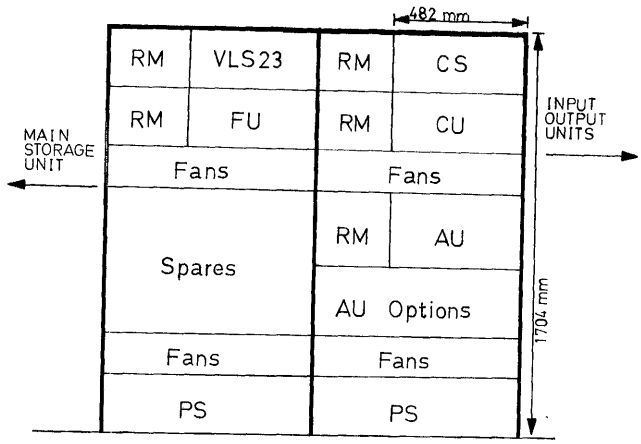
MICROPROGRAM ORGANIZATION

Figure 3



BLOCK DIAGRAM
FCPU

Figure 2



PS = Common Power Supply

RM = Regulation Module 60A, 5V

CU = Control Unit

CS = Control Storage (≤16K, 32 bits)

AU = Arithmetic and logical Unit

FU = Field Access Unit

VLS23 = Variable Logic Set for D23

Physical view of the FCPU
in the D23 system

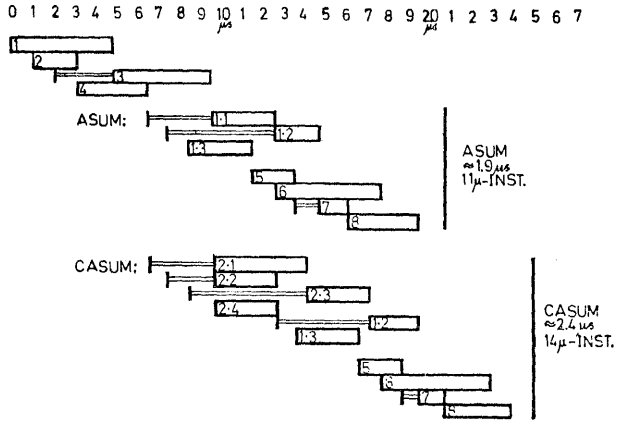
Figure 4

```
FCPU
UNIT

FU      1       LOOP:       READ (ASTREAM, AVALUE) STEPR SAVE (ASTREAM)
CUP     2                   MIX = TOPCODE
FU      3                   READ (BSTREAM, BVALUE) STEPR SAVE (BSTREAM)
CUB     4                   ENTER MIX ADDOP
CUP     5       RETUR:      CRO.1 = DECR (CRO.1)
FU      6                   WRITE (CSTREAM, CVALUE) STEPR SAVE (CSTREAM)
CUB     7                   IF OVERFLOW THEN GOTO OVERCODE
CUB     8                   IF NOT DONE THEN GOTO LOOP
CUB     9                   START ...               * INSTRUCTION FINISHED


                    ADDOP:  DO CASE (4)
                            DO
AU      1.1         ASUM:   CVALUE = ADD (AVALUE, BVALUE)
CUP     1.2         ISTORE: OVERFLOW = AU-OVERFLOW
CUB     1.3                 RETURN
                            END

                            DO
FU      2.1         CASUM:  READ (CSTREAM, TVALUE)
AU      2.2                 AVALUE = ADD (AVALUE, BVALUE)
AU      2.3                 CVALUE = ADD (AVALUE, TVALUE)
CUB     2.4                 GOTO ISTORE
                            END
                            END                     * END OF CASES
```

SAMPLE VECTOR OPERATIONS

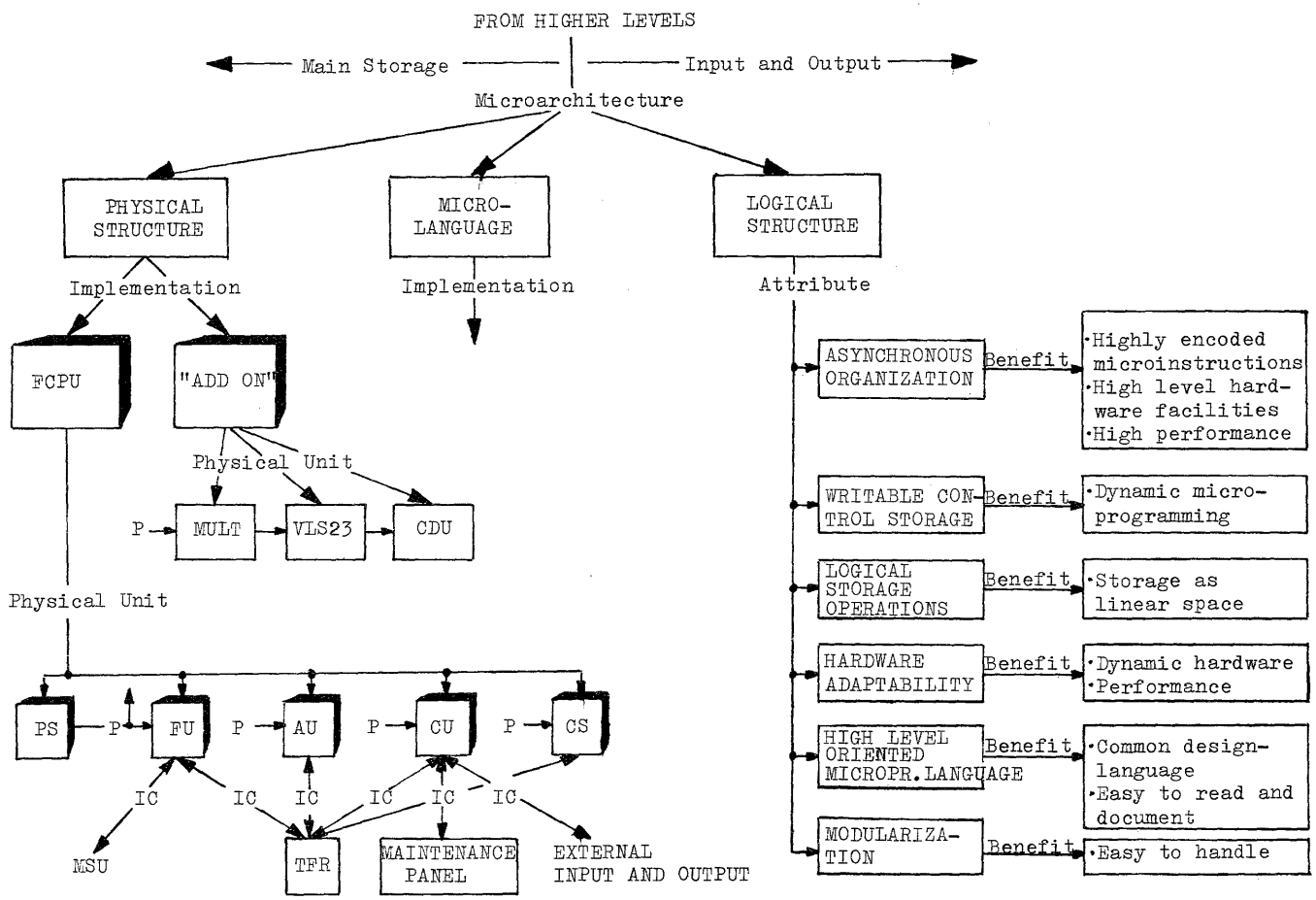
Figure 5

Figure 6



ASUM, CASUM EXECUTION TIMING

Figure 6



IC = Interconnection
CDU = Character Decimal Unit
MULT = Multiplication Unit
P = Power

STRUCTURAL SUMMARY OF THE LOGICAL AND PHYSICAL ORGANIZATION OF THE D23 MICRO-ARCHITECTURE

Figure 7

CONCEPTS OF THE MATHILDA SYSTEM *

by

Peter Kornerup, Department of Computer Science, University of Aarhus, Denmark

Abstract

A dynamically microprogrammable processor called MATHILDA is described. MATHILDA has been designed to be used as a tool in emulator and processor design research. It has a very general microinstruction sequencing scheme, sophisticated masking and shifting capability, high speed local storage, a 64-bit wide main data path, a horizontally encoded microinstruction, and other facilities which make it reasonably well suited for this purpose. This paper presents a brief overview of the MATHILDA system, and some of the concepts in more detail.

1. Introduction

1.0 Background

In 1971 the Department of Computer Science at the University of Aarhus started the design and construction of a microprogrammable minicomputer. The decision was based on the availability of small engineering staff and the evolution of a technology, which made it feasible to construct such experimental equipment, not as a technological experiment, but as a tool for work in emulation. The resulting machine, the RIKKE-0, was partially constructed, and started running in early 1972. In the meantime, a number of departmental projects were proposed, some of which were started while others were considered not to fit in with the present design. Various numerical analysis projects were put among the latter because RIKKE-0 has a short word size (16-bit) and in order to obtain an efficient implementation of even standard arithmetic operations a wider word was needed.

It was therefore suggested that a microprogrammed functional unit with a wider data path could be attached to RIKKE-0 as an I/O device, together with a wider memory. This organization would allow the problems of numerical analysis and those of the system-software to be more or less separated on the independent units. It was this functional unit which eventually became the MATHILDA machine (more detailed descriptions of this machine can be found in [1, 2]).

During the design phase of MATHILDA in mid 1972 it became apparent that those features which were felt to be necessary and those which came as side effects covered and extended those of the prototype RIKKE-0. As it had been decided earlier to construct more RIKKEs, the MATHILDA-design was adopted for those, with the only exception the data width being respectively 64 bit and 16 bit. Such a version of RIKKE has been running since early 74 as a prototype, and the full MATHILDA is due to be completed in late 74.

1.1 Motivations

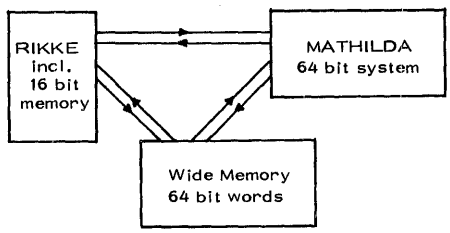
As mentioned earlier, the first motivation was to allow the implementation of arithmetics, (especially non-standard, like extended range, extended precision, significant digit, unnormalized, interval, rational or complex arithmetics). The general structure and microprogrammability of MATHILDA certainly will offer efficient implementation of the arithmetic primitives without the expense of special purpose hardware. The overall structure of the system will allow extensive experiments on various arithmetics, by changing underlying structure 3.

With respect to emulator or processor design in general, MATHILDA, or its 16-bit counterpart RIKKE, will allow the implementation of experimental virtual machines in various ways. The control of the system may be achieved both with various degrees of residual control and by immediate control, thus allowing various strategies to be investigated. Much of the bindir of particular decisions is left open for the implemento of a particular algorithm, also some of the features of the system (e.g. the bit-encoder) may provide the implementor with new alternatives.

In this way the MATHILDA-design will offer an experimental host for emulators; not in the way that i will provide standard solutions to specific problems, but by providing resources for alternative ways of implementation.

1.2 Global system design

The fact that MATHILDA is to be treated by its counterpart RIKKE as an I/O device, as shown in Figure 1, offers a great flexibility and suggests some interesting projects. It was decided that, except for a few control signals, every information interchange should be queued up so that asynchronous operation was possible. The same principle was also used with respect to a 64-bit Wide Memory, this being a commo resource to the system, and not dedicated to MATHILDA.



The MATHILDA-RIKKE-Wide Memory system
Figure 1

The Wide Memory is exclusively controlled by RIKKE i.e. it is the only unit that can deliver requests for memory access. The data transfer can take place on a number of memory ports. RIKKE itself will be attached to a set of I/O-ports of the memory system. RIKKE also has its own private memory (up to 64K 16-bit words). Standard I/O-devices are therefore tended to be coupled either directly to RIKKE or to other minicomputers communicating with it.

The idea of operation is to let RIKKE decode vi tual machine instructions, and to do all address calculations involved, while MATHILDA was to perforn the actual data transformation required. The philosophy of the design of MATHILDA was to give it capabilities for doing transformations upon a wide data word. Complicated "macro" routines could be implemented in MATHILDA microcode and thereby define

* This work is being supported by the Danish Research Council, grant no. 511-1546, and by NATO Grant no. 755.

"Nano-machine" which can be called upon from the microcode in RIKKE to define the complete virtual machine. In fact it is not a nano-machine in the QM-1 sense, but it may be operated in such a way that a similar effect is achieved. Among the differences in these approaches is that RIKKE, running in parallel with MATHILDA, normally will be ahead of it, preparing instructions and operands for it.

2. A brief description of MATHILDA

2.0 An overview

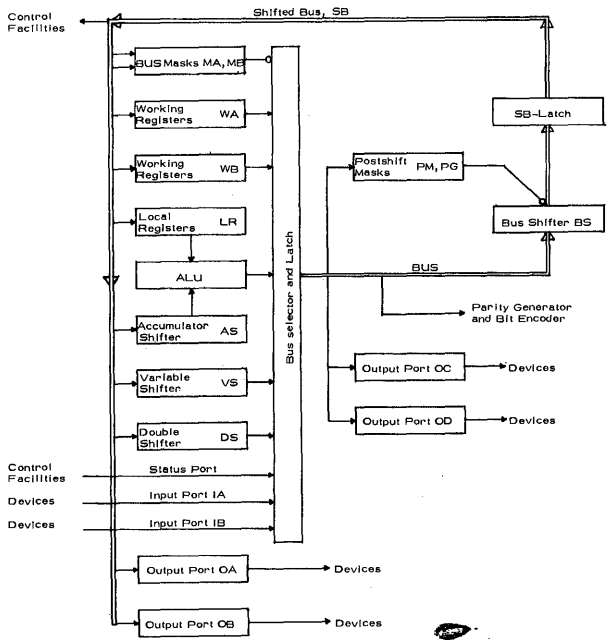
One may consider MATHILDA as composed of three layers, each of which is controlling the lower ones:

a) The decoding and sequencing unit
b) Control-facilities
c) The main data path

which now will be described briefly in the reverse order.

2.1 The Main Data Path, MDP

MATHILDA has a single 64-bit wide data path (called the Main Data Path, MDP) with 8 inputs consisting of various 64-bit wide sources. The information carried on the MDP is subject to transformations as described below.



MATHILDA Main Data Path

Figure 2

In every microinstruction it is possible to take the contents of a specified 64-bit wide source and mask it by use of a mask, BM, which is composed from two independently stored masks: BM = MA ∨ MB. Both MA and MB are read from a store each containing 16 such masks. The reason for the inclusion of double masks is that one group of masks (say MB) containing a no-mask and an all-mask can be used to enable/disable the other group of masks (say MA).

The masked data is buffered in a latch, the output of which is termed the BUS. The data on the BUS is continuously being encoded, yielding various types of control-information (parity and bit-encoding, see 2.3.2). Furthermore, the BUS-information may directly be used for loading into various special destinations, e.g. output buffers. The data on the BUS is passed through the bus shifter, BS, which when enabled shifts the data n bit positions right cyclic; where $0 \leq n \leq 63$.

The implementation of the BS is 64 parallel selectors: one selector for every position in the output which selects which one of the 64 input lines of the BUS is to be used (in a right cyclic connection) as the corresponding output bit. When no shift is required, the selectors all reside in a standard no shift position by disabling the selector via a dedicated bit in each microinstruction.

After shifting, the information is masked again, this time by a combined mask PM ∨ PG, where PM is similar to BM, and PG is an end-off mask generator, which allows the bus-shifter combined with PG to result in a logical shift. The PG is a PROM whose contents can be combined to yield the 128 masks which are required to make the BS appear as a logical left/right shifter as well as a cyclic left/right shifter. The enabling of the PG is determined by the PM, i.e. PM containing both a no-mask and an all-mask allows it to be thought of as a switch for the operation of PG.

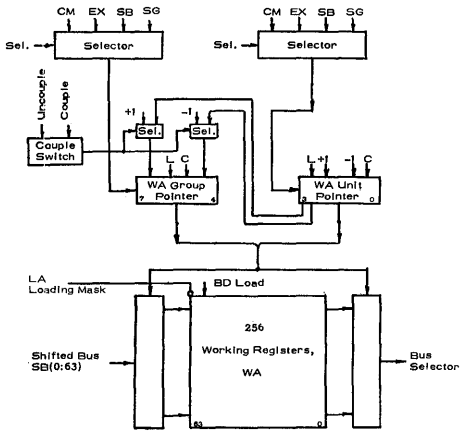
If the same source of control is used for BS and PG, then six bits will specify n, and the seventh whether a logical rightshift of n places, or a logical leftshift of 64-n bit positions, will be the result when PG is enabled. The information after shifting and postshift-masking is buffered in a latch, called the SB Latch. The output of the SB Latch is called the Shifted Bus, SB, and is finally loaded into selected destinations.

2.2 Additional MDP resources

The resources are either possible sources of a bus-transport, or destinations for a transport, or both. Except for the arithmetic-logic unit, ALU, they are all some sort of registers, some are pure storage elements (WA, WB, LR), some are shifters (AS, VS, DS), and the rest for I/O communication (IA, IB, OA, OB, OC, OD).

2.2.1 Working registers and their Loading Masks.

The system contains two local stores, WA and WB both containing 256 words. As they are identical we will only consider one of them, say WA, which is shown in Figure 3. Addressing of WA is made by a 8-bit pointer, WAP, which determines which location to read or write. WAP is, in fact, composed of two 4-bit pointers which may be coupled together (or be decoupled), which allows for considering WA as either 256 elements, or as 16 groups of 16 registers.



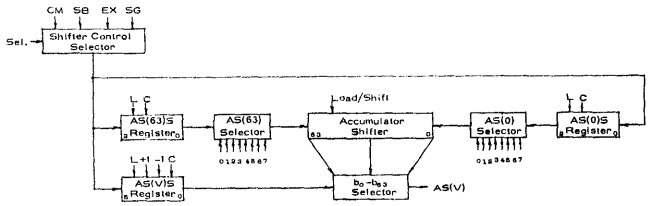
Working Registers A, WA

Figure 3

The writing of data into WA is also masked by means of a loading mask, LA. LA makes it possible to load only selected fields of the addressed WA-word, without affecting the remaining word. This permits the construction of say the result of a floating point operation, by loading the fields of the packed representation separately as they are being computed. LA (and LB on WB) is again a group of 16 masks with its own address-mechanism. (In fact LA is identical to a register group as shown in Figure 5.)

2.2.2 The Shifters, AS, VS and DS. The shifters are functionally identical, except for the fact that DS shifts two positions at a time, where AS and VS shift one position. It is therefore sufficient to consider one of them, say AS.

Inputs to the vacated bit in shift-operations may be chosen from any 1 out of 8 possible sources, one of which may be an arbitrary selected bit of AS itself, the AS(V) bit. This allows AS to act as a cyclic shifter of an arbitrary length n, $1 \leq n \leq 64$.



Accumulator Shifter, AS

Figure 4

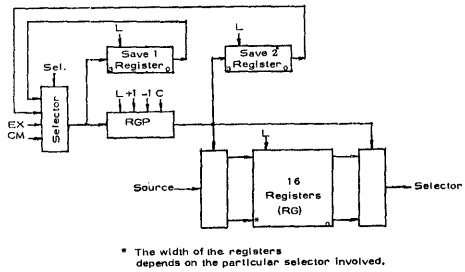
2.3 Control facilities

The control facilities consist of storage elements, data paths and functional units, thereby allowing a high degree of parallelism and permitting a variety of external, stored or computed control upon the units on the main data path. As it can be seen from the previous description of the MDP and its associated storage and functional elements, most of these resources require some sort of control information. This may be an address (e.g. for WA and WB), data for a functional unit (e.g. a shift specification for BS or mask specification for PC), a functional specification (e.g. control of ALU function) or a selector-specification (e.g. for vacated bit input to a shifter). The possible sources of control information are the following:

1) CM: the current microinstruction
2) EX: an external register (from controlling processor)
3) BE: the bit-encoder unit (see 2.3.2)
4) SB: rightmost bits of the shifted bus
5) SG: registers for residual or saved control (see 2.3.1)

It should be noted that item (1) above allows the user "immediate control" over the system's resources whereas items (2-5) offer varying degrees of "residual control" in the context of Flynn and Rosin [5]. Furthermore, among the control facilities are system counters, local storage groups, and condition save registers. There are also "status" and "snooper" facilities, which can be used to gather data useful in computing statistics concerning the system and thus enhance the experimental nature of this machine.

2.3.1 The Standard Groups. The amount of residual control facilities is large and could have been rather complex, because any of the selectors determining control sources has among its four inputs a "private" register group, providing stored (residual) control. However, a great deal of simplicity and modularity has been achieved both logically and physically by use of a uniform residual control concept, the standard group:

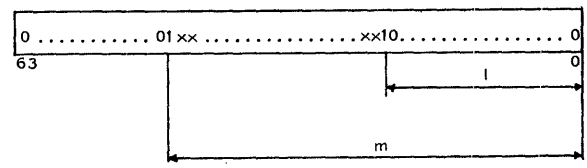


Typical Standard Group

Figure 5

A Standard Group is a storage element of 16 words with an address mechanism. The storage is used for residual control, and in these cases where the information residing there is part of a fixed environment, the loading of the storage takes place from the SB. However, in connection with some units, it was more natural to load the storage from the unit itself, so that the information there could be saved and later restored.

2.3.2 The Bit-encoder BE. One resource of the MATHILDA is to our knowledge a new invention. It is what we call the Bit-encoder, BE. In many algorithms it would be useful to have easy access to information about a given bit-pattern as to where is the first bit on, and where is the last:



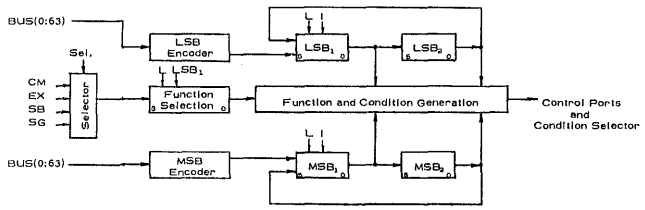
We call the quantities l and m respectively LSB (least significant bit) and MSB (most significant bit). The bit-encoder can provide the user with such numbers, and certain computations using LSB and MSB.

The Bit-encoder is continuously encoding the information on the BUS, yielding the two quantities l and m corresponding to the transported bit-pattern. By a microoperation these can be loaded into two registers LSB_1 and MSB_1. The computational network of the Bit-encoder further contains two additional registers LSB_2 and MSB_2, which can be loaded from LSB_1 and MSB_1 respectively, or be interchanged with these. Assuming LSB_2 and MSB_2 contain the encodings from a previous BUS-transport, the circuitry of the Bit-encoder continuously computes the following quantities among which any one may be selected as the BE-output:

BE Functions	
F	G
LSB_1	
$LSB_1 - 1$	
MSB_1	
$MSB_1 + 1$	$G = [F/2] + 1$
L_1	
$\Delta L = L_1 - L_2$	
$LSB_2 - LSB_1$	
$MSB_2 - MSB_1$	

$L_i = MSB_i - LSB_i \quad i = 1, 2$

$[x] ::=$ Integer part of x



The Bit Encoder

Figure 6

The output of the BE may be used in various control elements of the system.

Based on available conditions, it is possible to compare some of the characteristics of the two bit-patterns, to: a) direct decisions about the algorithm, b) choose BE-function, or c) interchange (LSB_1, MSB_1 and (LSB_2, MSB_2), before selecting the BE-function and use the selected information.

Since bit-patterns and bit-matrices play an intensive role in many non-numerical algorithms, fundamental operations providing the encodings l and m, may prove to be useful both on the virtual machine-level, and also in high-level languages.

2.3.3 The Snooper and Status Facility. The Snooper Facility consists of (a) a Snooper Control Store and (b) Snooper Resources (e.g. 2 groups of 16-registers, counters, and comparators). The Snooper unit works in the following way: when the address of the next microinstruction to be executed is sent to the MATHILDA Control Store address buffer, it is also gated into the Snooper Control Store address buffer. At the same time the microinstruction is fetched so that it can be executed, the contents of its associated Snooper Control Store location is fetched. In parallel with the microinstruction being executed, the contents of its associated Snooper Control Store just fetched is used to control the operation of the Snooper Resources. Snooper Control Store is 16-bit wide and has the same number of words as the MATHILDA Control Store. A snooper word can specify, for example, any two registers which can be counted up (or down). Snooper Control Store is writable so that different data gathering routines can be associated with the same segment of microcode without changing the microcode. The user is allowed to establish the correspondence between any particular snooper resource and the routine upon which it is snooping. Information gathered in the snoopers can be brought back into the system through the status port of the system (see Figure 2), just like other information from control facilities which have no direct connection to the MDP.

2.4 Microinstruction execution and sequencing

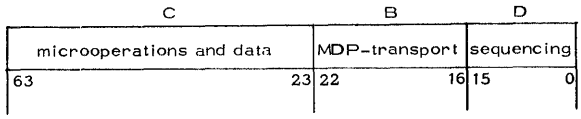
One microinstruction execution of the machine may be considered as consisting of four major sequentially executed steps:

A: Microinstruction fetch
B: Data transport on MDP
C: Execution of microoperations
D: Address calculation (for the next microinstruction)

Steps B, C, and D are controlled by fields within the 64 bit wide microinstruction. The execution of one microinstruction is to be considered as totally completed before the next microinstruction is executed, i.e., actions initiated by the execution of a particular microoperation do not span several microinstruction executions.

The control store may consist of up to 4096 words of 64-bits (80 nS). Initially, 512 words of control store has been implemented. It is writeable under program-control from an output port of the system itself, or from a 16-word deadstart PROM.

The microinstruction consists of three major fields, corresponding to the control of the previously mentioned steps B, C, and D:



The dealing of dynamic conditions has been made consistent. The machine can be run in two modes, under program control:

Long cycle: Any condition arising as an effect of the execution of steps A, B, C can be tested and used for sequencing in step D.

Short cycle: All conditions used in the instruction is of the state of the machine immediately prior to step A.

The B-field of an instruction specifies the source and destination of the MDP-transport, and the enabling of the Bus Shifter. The C-field contains selector specifications for the control facilities, and some fields which may be encoded as microoperations, or used as data depending on certain "mode-bits". These highly encoded fields (typ. 8-bits) are mostly used for exercising the control facilities associated with the arithmetic logical unit, the residual control standard groups, addressing of local storage elements, and so on. Up to 4 highly encoded microoperations can be specified in one microinstruction, plus 3 minimally encoded operations.

2.4.1 Sequencing. The determination of the address of the next microinstruction to be executed, is an implementation of the well known if-then-else construction. The selection is among two modes of address-calculations (rather than addresses themselves). Possible modes are:

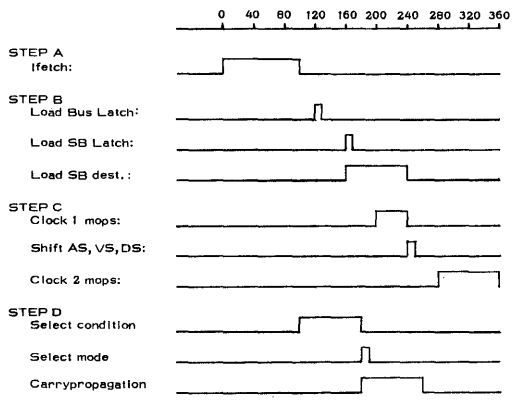
1) A - 1 2) A 3) A + 1	where A is the current address (location counter)
4) F(A,B)	where F is an ALU-function of inputs A and B
5) RA + B	top of return jump stack + B
6) RB + B	top of return jump stack + B
7) SA	buffer which can be loaded from the Shifted Bus
8) EX	external buffer

The B-input used is an address constant which is specified by the D-field, involving the selected condition and data from the C-field. Mode 4 allows for absolute and relative addressing.

The if <cond> then A_t else A_f clause is realized the way that the value of the specified condition selects among the two specified modes A_t or A_f.

2.4.2 Timing. One criteria was to have a clean and consistent way of dealing with the timing problems. We preferred a slower but more straightforward machine. Thus it is possible to consider a machine cycle (the execution of one instruction) as consisting of a number of logically sequential steps, corresponding to a sequential execution of the specified actions in the essentially horizontal instruction. Figure 7 shows actual timing and also the fact that microoperations are divided into two classes which are clocked differently.

This allows as an example within one microinstruction the loading of an element in a RG, and the changing of its pointer.



System Timing in short cycle

Figure 7

The resulting speed is at present 360 ns per short cycle, but may be improved (a pre-fetch of instructions may reduce it to 280 ns).

3. Conclusions and experience

3.1 Design and construction

As a department in a non-engineering university, our hardware- staff was very restricted (2 engineers + 1 technical assistant). This staff was mainly intended for minor construction and interfacings, together with service on all standard equipment in the department. The design was originally intended to be for a functional unit, very intimately connected to the controlling RIKKE-0. However, during the design process a number of complex facilities were added. Thus the functional unit grew into a selfcontained processor and, of course, our original time schedule for the construction was not satisfied. Besides the growth in design, unexpected other duties of the technicians, delivery problems, lack of project management experience, and the fact that the project was, in retrospect, a bit over-ambitious for the staff, delayed the project. It was furthermore decided to test the design on a 16-bit RIKKE version before the construction of the 64-bit MATHILDA.

The basic design, including the bus structure and the concepts of standard groups bit,encoder etc. was made in the period from February to August, 1972. The sequencing part and instruction format was designed in the fall, while the printboards for the bus-structure were laid out and actual mounting of RIKKE started. By August 1973, the control unit and control facilities (register groups) were ready for initial hardware testing. In late October, the 16-bit bus structure was added for testing, and in January 1974 initial testing of the RIKKE processor was considered completed. However, no main memory was added before March, due again to delivery problems. At the time of this writing (Oct. 74) the RIKKE has interfaced to it a high speed tape reader and punch and a console writer. The RIKKE machine is complete as the MATHILDA machine described in this document except that the following facilities are not yet implemented: Snoopers, Status and Bit-encoder.

The construction of MATHILDA was started in August, 1973, but since testing of the MDP (and the print boards) was not done before January 1974, the "go ahead" for the construction of the 64-bit bus structure could not be given before then. Here again delivery problems for the print boards caused delay. By now initial testing has started, but one bus module has yet to be mounted.

The cost of the construction is not easily calculated As estimated cost of $12,000 for components and $10,000 for mounting assistance was granted by the Danish Research Council. The support from the staff technicians cannot be computed that simple because of their other duties and projects. An estimate of 2 man-years of design and documentation from the technicians might be adequate. The experience gained in the department during this process cannot be underestimated, we learned a lot about what to do, and especially what not to do in such a project.

3.2 Experience with programming

Since the summer 73 an assembler, MARIA, and a simulator of the system have been in use [6]. Emulators for an O-code machine for BCPL [7], and a P-code machine for Pascal [8] on the RIKKE-MATHILDA-Wide Store system have been written. Furthermore, basic software (bootstrap loader, normalizer, etc.), an I/O-nucleus [9] and a large number of routines for the implementation of arithmetics have been implemented. Experience with programming of the processors shows that the design seems to be suitable for experimental purposes, although coding is not so straightforward because of the horizontal nature of the micro-instructions as on processors with more highly encoded (vertical) but more primitive instruction formats (e.g. the B1700). Certain facilities of the system have proved to be extremely useful, the easy and natural sequencing possibilities, the BS, BM, PG, and especially the BE.

Although the design of MATHILDA may seem somewhat complicated, experience from courses on computer architecture given at Aarhus and at the University of Southwestern Louisiana, indicates that the students learned the MATHILDA design with reasonable ease. We are at present only in the very beginning of the real use of the RIKKE and MATHILDA processors, and only in the future can real evaluations of the suitability be made.

3.3 Acknowledgements

The design was made in collaboration with Prof. Bruce D. Shriver (current address: University of Southwestern Louisiana, Lafayette, La.); without his enthusiasm and hard work during the design phase the goal was never achieved.

The author further wants to express his thanks to the Danish Research Council, who granted the construction costs and the stay of Bruce Shriver during the design phase. Thanks are also directed towards NATO Division of Scientific Affairs, which is supporting further collaboration between Aarhus and Lafayette. Finally thanks are expressed to colleagues for their comments and advice, to the students who helped with supportive work and software development, and to the technicians who patiently attended numerous discussions, accepted changes and additions from the designers during the process.

References

[1] Shriver, B.D., "A description of the MATHILDA System", Dept. of Computer Science Report PB-13, University of Aarhus, Denmark, April 197

[2] Kornerup, P. and Shriver, B.D., An overview of the MATHILDA System", Dept. of Computer Science Report PB-34, University of Aarhus, Denmark, August 1974.

[3] Shriver, B.D., "A small group of research projects in machine design for scientific computation", Dept. of Computer Science Report PB-14, University of Aarhus, Denmark, April 1973.

[4] Rosin, R.F., Frieder, G., and Eckhouse, R., "An environment for research in microprogramming and emulation", CACM, 15, No. 8, 197-212, August 1972.

[5] Flynn, M., and Rosin, R.F., "Microprogramming, an introduction and viewpoint", IEEE TC, C-20, No. 7, 727-731, July 1971.

[6] Lynning, E., Kressel, E., Andersen, H.O.S, Sørensen, I.H., "A users manual for the simulated RIKKE-MATHILDA system on the CDC 6400", Dept. of Computer Science Report, University of Aarhus, Denmark, 1974.

[7] Sørensen, O., "The emulated O-code machine for the support of BCPL", Dept. of Computer Science Document, University of Aarhus, Denmark, to appear.

[8] Kristensen, B.B., Madsen, O.L. and Jensen, B.B., "A PASCAL environment machine (P-code)", Dept. of Computer Science Report, PB-28, University of Aarhus, Denmark, April 1974.

[9] Rosin, R.F., "Proposal for a nucleus I/O system", Dept. of Computer Science Report, PB-23, University of Aarhus, Denmark, January 1974.

Appendix

Physical summary of MATHILDA

MATHILDA is mounted in 7 frames 8x45x60 cm, each of which is turnable around a vertical axis like pages in a book. Each frame is closed from both sides with removable boards of plexiglass, thus forming a closed box. Each "box" is equipped with three small ventilators blowing a stream of air up through the frame. Signal-interconnections between the frames are through standard cables containing 20 signal-ground wound pairs of wires. Plugs are mounted along the vertical sides of the frames.

The printboards are two-sided and are either special prints, or standard prints only containing power and ground where signal-interconnections are in the wiring. Special purpose prints exist only in three variants:

a) 8-bit of the whole MDP, 25x40 cm,
b) 4x(4 bit of a standard register group), 15x40 cm,
c) 256 words of control-store, 64-bit wide, 15x40 cm.

All of the special purpose printboards furthermore contain some room for additional circuitry.

No attempt has been made to carry signals to the edges of the boards for board-to-board and board-to-plug interconnections, all such connections are made with wires from the proper places on the prints.

All circuits are from the TI 74 series or equivalent. A large amount of signals (data buffers) have been made visible on the boards by light-emitting diodes or displays (for diagnosis in step-mode).

A survey of the content of the frames is given below:

Frame 0 : (Sequencing and Control Store)
1 pc 30x40 standard print containing sequencing, clock-generators and deadstart. Masterclock (40 ns steps) and its input into a shift register which pulses various clocks.
2 pc type c (above) prints control store, 128 pc TI 74200+amplifiers.

Frame 1 : (CA, CB, WAP, WBP, LA, LB)
4 pc type b prints (standard groups).

Frame 2 : (Standard groups for shifters, PG, BS, BM, PM, AL, etc.)
3 pc type b prints + 1 pc 15x40 standard print.

Frame 3 : (End- connections for bus modules, BE + various)
1 pc type b print + standard print.

Frame 4-7: (Each contains 16 bits of MDP with all registers and ALU)
2 pc type a prints + 1 pc 10x40 standard print on each frame.

Power supply: 8 pc, 5V, max 15 amp.

Console: (preliminary)
Buttons: Deadstart, Run, Stop after deadstart, Step-Stop (two pushes per cycle, first instruction fetch, second execution).
Switches: KA, KB conditions, 2 stop-switches (stop on execution of specific mops, i.e. not as testable condition).

Deadstart: 16 words of battery-driven CMOS-PROM are copied into control store repeatedly. Execution is forced to location zero.

SOCRATES

by

Caxton C. Foster
Computer and Information Science
University of Massachusetts/Amherst

Abstract

SOCRATES is a "Stack-Oriented Computer for Research and Teaching--an Exploratory System." It is designed around push down stacks as its main store. The structure of the machine is specified and several aspects of programming it are investigated.

Keywords: Push down stacks, computer architecture, LIFO stacks, machine design.

Introduction

Many computer architects spend their days reinventing the wheel; some with five spokes, some with four, and some with seventeen. Even the most cursory examination of computers extant today reveals at once the paucity of our imagination and the dreary sameness of our products. One begins to wonder if he would like his daughter to grow up and marry a computer architect. To state this another way, I ask the question, "Now that we have established the fact that we can produce untold variations on a theme by von Neumann, what do we do for an encore?

A couple of years ago, I was considering this problem when someone happened to mention that storage technologies, namely bubble memories and charge coupled devices, were showing every promise of becoming commercially feasible in the near future. On investigation, I discovered that indeed people were predicting that by 1976 one should be able to purchase a one inch cube with 32K words of 32 bits for $100. But, alas, the millenium would not arrive in '76, only the bi-centennial. There was a slight drawback to these burgeoning technologies. Access points were to be expensive. It was the interface with the external world that would cost the money, not the mechanism for holding the information. Thus, the only way of meeting these projected prices would be in memories organized as stacks, either FIFO or LIFO. That is, only as recirculating memories similar to drums or as push-down stacks could this dramatically low cost of .01¢/bit be expected to be realized.

Having spent several years programming an LGP-30 with a 17 millisecond drum as its main (and only) store I knew right away that push-down stacks would be a much more interesting area to investigate.

Given my overwhelming *weltanschmertz* vis-a-vis conventional architectures and the possibility, however remote, that truly inexpensive stacks would soon become available, I set out to design a computer that used push-down stacks as its primary memory. This paper is the result of that endeavor.

Now a caveat. Nobody, least of all the present author, is pretending that SOCRATES would be the machine-of-choice of your average, everyday programmer. The questions are rather, would such a machine be tolerable? Would it run ordinary type programs moderately fast? (Remember that to achieve a cost-performance equivalent to a conventional computer, it could be as much as 100 times slower.) Would the enforced push-down access make programming so clumsy that software costs would far outpace any hardware savings?

To attempt to answer these questions, we present below a possible architecture for a stack oriented computer and a few kernels of programs that tend to display the problems of the possibilities inherent in the design. No pretention of exhaustiveness is intended.

LIFO Stacks

LIFO or pushdown stacks are well known. They are characterized by having a single access port and a finite capacity. Items added to the stack are stored so as to preserve the order of their arrival and when read out is requested, the items are returned most recently arrived (last in) first. A shift register with right shift on push down and left shift on pop up is an example of such an organization provided that only the left-most bit is accessible.

We will distinguish three basic modes of addressing stacks. Consider first stacks from which information is to be obtained (operand sources).

- mode 1. The named stack contains the desired operand. Obtain a copy of the data but do not disturb the stack.
- mode 2. The named stack contains the desired operand. After obtaining a copy of the data "pop up" the stack removing the operand from that stack.
- mode 3. The named stack contains the "address" of the desired operand. Obtain the name of the stack which contains the operand and the new mode of address and reenter the address decoding process. Do not change the named stack.

For completeness, we will also allow "immediate addressing" in instructions by:

- mode 0. The name of the stack <u>is</u> the desired operand.

Mode 0 doesn't make sense when we are considering stacks as destinations for data but the other three modes carry over:

- mode 1. The named stack is the destination. Store the information in the top cell of that stack.
- mode 2. The named stack is the destination but before storing the information in the top cell of that stack, push down that stack, preserving its old contents under the new information.
- mode 3. As for sources above.

For orthographic purposes, we will use the following:

kind	mode	source	destination
immediate	0	#S	--
direct	1	S	D
stack	2	S↑	↓D
indirect	3	S	@D

To indicate which mode we desire will require two bits per address. If we wish to keep an address (mode plus name) to 8 bits out of respect for the National Standards Institute, then we will be restricted to having 64 stacks in our computer. Those who find this unduly restrictive, may dream of 9 or 10 bit addresses with consequently longer machine words.

For design purposes, we will imagine stacks of 32K words of 32 bits each with a 1/2 microsecond cycle time. This will provide a main store of $2^{15} \times 2^2 \times 2^6$ or 8 megabytes. We further imagine the cost of such a main store to be around $6500. (Estimates of 32K words by 32 bits for $100 were made as long ago as 1972.)

Even with stacks as large as these, they will occasionally become full or empty. These "exceptional" conditions will generate interrupts for our computer when an attempt is made to reference an empty stack or to preserve a full stack.

Brief Description of the Hardware

SOCRATES is a two's complement, binary, 3 address computer with 64 addressable registers of 32 bits each. Each register is backed up by a pushdown stack of 32K words. Registers are accessed via 3 busses, two read and one write, so that data fetch and store may be done in parallel.

One register and its associated stack are shown in Figure 1. References of modes 1 and 3 go directly to the TOP register. Mode 2, on reference to a source, causes the following sequence of operations to be carried out:

1. If (K) = 0, generate an irterrupt because the stack is empty.
2. When BUSY = 0, present (A) on read bus 1 or 2 as requested.
3. When data adsorbed by CPU and stack is released, start the popping operation on the stack. Make BUSY = 1, and decrement the counter K ← (K) - 1.
4. When pop is complete and new word is in register B, copy new word from B into A and make BUSY = 0.

When the stack is used as a destination in mode 2, this set of operations is executed:

1. If (K) = 77777_8 generate an interrupt because the stack is full.
2. When BUSY = 0, accept information from WRITE bus and give release to CPU. Make BUSY = 1.
3. Begin preserving stack writing old word from B into stack. Increment the counter K ← (K) + 1.
4. When preserve is complete, copy word in A to register B and make BUSY = 0.

This scheme of double buffering ensures that only when a single stack is referenced twice in a single instruction, the first time in mode 2, will there be a delay due to the stack cycle time.

Of the 64 registers in the CPU several are somewhat different from the above. Register 0 contains all zeros and attempts to load, preserve, or pop it are ignored, but are not considered to be faults.

Register 1 serves as the program counter and Program Status Word (PSW) for the machine. Several instructions reference this register implicitly and are described in the next section. Figure 2 shows the top cell of register 1.

The contents of the third byte (PS) specifies the stack which holds the next instruction to be executed, while the fourth byte (PD) names the stack holding "used" instructions. The instruction fetch cycle consists of moving one word from the PS stack to the PD stack and making a copy of the moved word in the instruction register. Mode 2 is always implied for both stacks involved in an instruction fetch.

Interrupts

There are two general classes of interrupts to be considered. External (I/O) interrupts will be honored

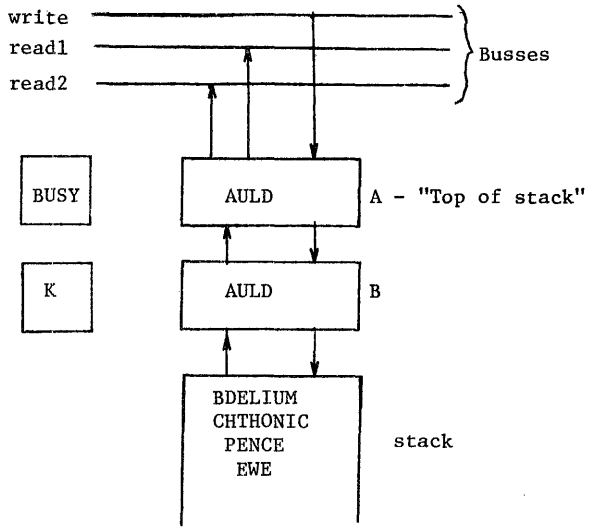


Figure 1. One register with its associated stack. Successive source references in mode 2 will present AULD, then BDELIUM, then CHTHONIC, etc.

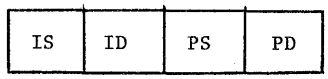


Figure 2. The top cell of register 1. IS and ID are the interrupt source and destination while PS and PD are the main program source and destination.

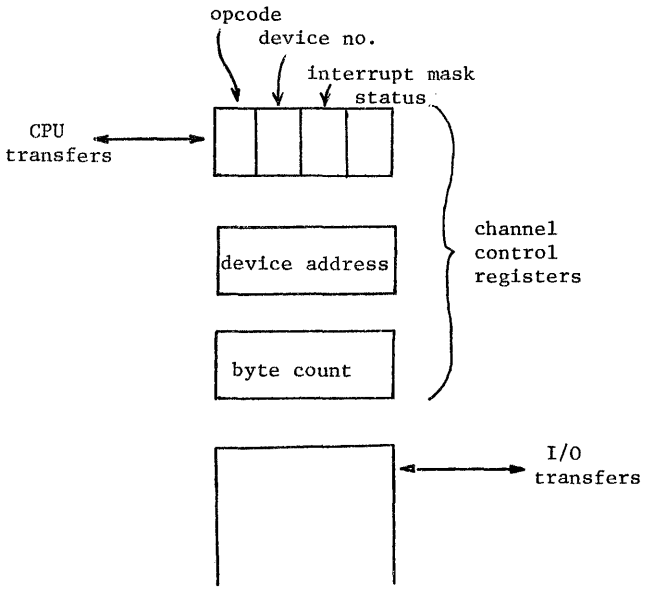


Figure 3. An I/O stack

between instructions only, the current instruction being allowed to complete its execution. Internal interrupts, however, may arise during the course of executing an instruction and must be honored at once. Four possible internal interrupt conditions can occur: First Source stack is empty, second source stack is empty, arithmetic fault, and destination stack is full. Suppose for example, the instruction to be executed is

A↑ + A↑ → ↓B

and the destination stack B is full or the source stack A has only one item in it or perhaps the addition results in an arithmetic overflow. In any of these cases, the items removed from the source stack in the course of partially executing this instruction must be replaced, lest they be lost. Then the instruction may be reinitiated after the interrupting condition is repaired--perhaps by scaling the data, or draining the destination stack.

We see therefore that the wired in interrupt routine must restore the source stacks to the condition they were in before the instruction commenced. We will leave it to the software to move the offending instruction back from the PD to the PS stack. Hence, upon detection of an interrupt condition, we must:

1. If internal, restore source stacks if they were accessed in mode 2.
2. Push down stack 1 preserving the current PSW.
3. Swap the two halves of the old PSW bringing the interrupt source and destination to the lower half and the main program source and destination to the upper half.

Suppose that the main program source and destination were stacks 2 and 3 and that the interrupt handling routine is stored in stack 36 and uses 37 as a destination. Then before an interrupt, stack 1 holds a PSW which looks like:

→ 36 37 2 3

and after an interrupt it looks like:

→ 2 3 36 37
36 37 2 3

This serves two functions. First, return from an interrupt is accomplished simply by popping stack 1. Second, the interrupt routine can find out (by examining the upper half of stack 1) where the main program resided which generated the interrupt (assuming it was an internal interrupt).

Condition Codes

Since instruction fetch is defined to be a stack operation, the mode bits of the two low order bytes of stack 1 are available for other uses. Bits 15 and 14 (the mode bits of the PS) and bits 7 and 6 (the mode bits of the PD) are called the "condition codes". They are preserved with the rest of the PSW upon interrupt or subroutine calls. They may be tested by various bit sensing instructions (see section on the set of instructions).

For arithmetic operations, the condition codes will be set as follows, depending on the value of the resultant:

bit 15 = 0 if result is zero, = 1 if non-zero.
bit 14 = 0 if result is positive,
= 1 if result is negative.
bit 7 = 0 if no carry out of bit 31,
= 1 if carry generated.
bit 6 = 0 if no overflow or underflow,
= 1 if overflow or underflow.

For search and move instructions, the condition bits tell what caused the operation to terminate:

bit 15 = 1 if comparand was found.
bit 14 = 1 if flag word was found before match.
bit 7 = 1 if source was empty before match.
bit 6 = 1 if destination was full before match.

After an interrupt is generated, the four condition bits taken together have the following meanings:

internal interrupts

15, 14, 7, 6	
0 0 0 0	reserved for service calls
0 0 0 1	source 1 empty
0 0 1 0	source 2 empty
0 0 1 1	destination full
0 1 0 0	arithmetic overflow
0 1 0 1	divide fault
0 1 1 0	exponent overflow
0 1 1 1	exponent underflow

external interrupts

15, 14, 7, 6	
1 M	stack 7M generated the interrupt. $0 \leq M \leq 7$

Input/Output

Depending upon the affluence of the purchaser from 1 to 8 I/O channels may be implemented. They are attached in place of stacks 77, 76, ..., 70. As far as the CPU is concerned, these stacks do not change by becoming I/O channels. They may still be used in a standard way. But upon issuing a Begin Output instruction, (BIO n) for stack n an autonomous transfer is begun in the fashion detailed in Figure 3. I/O takes place from the physical "top of the stack" in byte by byte format, left-to right. The physical top of the stack is the fourth word of the stack as seen by the CPU. The first word seen by the CPU forms the channel instruction register, device selector, interrupt mask and device status register (8 bits each). The next word is the device address register (for drums and disks) while the third word holds the number of bytes to be transferred.

Programming Considerations

A number of programming problems have been looked at in greater or lesser detail. In this section, we will present some results of these examinations. In general, our philosophy has been that time rather than space should be optimized. We would be most interested in hearing from any readers who can think of other program examples where our selection is found to be either helpful or inadequate.

Loops

LISP and other recursive languages notwithstanding, most computer programs rely heavily upon iterative loops. Figures 4 - 9 show a typical DO loop and its translation into assembly language code for SOCRATES and for a conventional type machine. The reader will

```
      DO 10  I = 1,50
      A(I) = B(I) + C(I)
10    CONTINUE
```

Figure 4. A typical FORTRAN DO loop

```
      ENI  49,1
.10   LDA  B,1
      ADD  C,1
      STA  A,1
      IJP  .10,1
```

Figure 5. An assembly language translation of the program of Figure 4 for a conventional machine. 5 words of storage. 351 memory references.

```
ADD  0,#49,I
NOP  .10
ADD  0,B,↓BTEMP
ADD  0,C,↓CTEMP
ADD  B↑,C↑,↓ATEMP
DZS  I,#1,I
LBK   .10
MOV  #50,BTEMP↑,↓B
MOV  #50,CTEMP↑,↓C
MOV  #50,ATEMP↑,↓A
```

Figure 6. The assembly language loop for SOCRATES assuming that B and C must be preserved and A must be left upright. 10 words of storage. 751 memory cycles. (NOP through LBK executed and then moved back to PS 50 times followed by 3 moves of 50 each. Assumes top of stacks "instantly available").

```
LDA  B
ADD  C
STA  A
LDA  B+1
ADD  C+1
STA  A+1
  :
  :
STA  A+49
```

Figure 7. The unfolded loop for maximum speed execution in a conventional one address machine. 150 words of storage. 300 memory references.

```
ADD  B,C,↓AINV      ⎫
MOV #1,B↑,↓BTEMP    ⎬ 3 × 49
MOV #1,C↑,↓CTEMP    ⎪
ADD  B,C,↓AINV      ⎭
  :
  :
ADD  B,C,↓AINV        1
MOV #49,BTEMP↑,↓B    49
MOV #49,CTEMP↑,↓B    49
MOV #50,AINV↑,↓A     50
```

Figure 8. An unfolded loop for SOCRATES which preserves B and C and re-erects A.

```
ADD   B↑,C↑,↓A
ADD   B↑,C↑,↓A
  :
  :
ADD   B↑,C↑,↓A
```

Figure 9. The unfolded loop assuming that B and C are not needed again and that A is left inverted. 50 words of storage. 50 memory cycles.

note that the program for a conventional machine coded as a loop (Figure 5) requires 351 memory cycles while the similar program for SOCRATES (Figure 6) requires 751 cycles. 300 of these extra cycles are caused by the necessity to "pour back" the 6 instructions of the loop. Another 150 can be accounted for by the necessity to restore B and C to re-erect A. Figures 7 and 8 compare an "unfolded" loop approach to the problem. The conventional machine requires 300 memory cycles and SOCRATES 296. Finally, in Figure 9, we show the SOCRATES program which might be written if we know that B and C were no longer needed and that it was permissible to leave the vector A upside down. It requires only 50 memory cycles.

Merge

A typical operation performed on a digital computer is to merge two sorted lists into a single master list. Figure 10 shows a conventional machine program for merging lists B and C into list D. The lists B and C are assumed to be in ascending order and each of length 100. Counting instruction fetches and data references, there are 2204 memory cycles required. Figure 11 is the comparable program for SOCRATES. It requires 2002 memory cycles. Figure 12 is an unfolded version of the same program requiring 600 memory cycles.

```
        ENI  199,4    put number of words into B4
        ENI   0,1     clear index register 1
        ENI   0,2     clear index register 2
        ENI   0,3     clear index register 3
LOOP    LDA   B,1
        SUB   C,2     minus (B < C)
        JMA   BFIRST  jump on A
        LDA   C,2
        STA   D,3
        INI   1,2
        INI   1,3
        IJP   LOOP,4  count down on index reg.4.jump
                        to loop
BFIRST  STA   D,3
        INI   1,1
        INI   1,3
        IJP   LOOP,4
```

Figure 10. A rather straightforward algorithm to merge lists B and C into D. Assume 100 items in each list. 2204 memory cycles required.

```
MVI #1,__,↓I
    #199          make I=199
GTS  B,C,__       skip next instruction
                    if B > C
MOV #1,C↑,↓B      put C on top of B if C ≤ B
MOV #1,B↑,↓D      move word from B to D
DZS  I,#1,I       I-1→I, if result is zero
                    skip
MOV #5,PD,PS      move 5 words from program
                    destination to program
                    source. Returns control
                    to GTS. instruction
```

Figure 11. Merge algorithm for SOCRATES. 2002 required. Leaves D upside down.

```
GTS   B,C,__      ⎫
MOV   #1,C↑,↓B    ⎪
MOV   #1,B↑,↓D    ⎪
GTS   B,C         ⎬ 200 × 3 instructions
MOV   #1,C↑,↓B    ⎪
  :               ⎪
  :               ⎪
MOV   #1,B↑,↓D    ⎭
```

Figure 12. Unfolded merge algorithm for SOCRATES. 600 memory cycles.

GTS compares the top of B and C. If C is smaller than B, we put the top item from stack C onto stack B. Then we move the top of B onto D and repeat.

Conclusions

We have presented one possible stack oriented machine and discussed briefly some aspects of programming it. In the introduction we posed some questions and it is now the time to discuss them in the light of what we have seen.

Is the machine tolerable? Yes. It has a fairly clean organization. Vector operations are natural. Iterative loops can be handled. Obvious problems would arise if random access to large arrays is required, but for many situations it appears to be adequate, even if far from ideal.

How big a penalty do we pay? For the kernels we have examined it seems that SOCRATES would require 2 - 3 times as many memory cycles as would a coordinate address based machine. Had the access ratio been 10 to 1 the point would have been debatable, but we observe about a 30:1 cost-performance improvement and that is enough to get excited by.

Can it be programmed by human beings? We have included some examples of how SOCRATES might handle some more or less common problems. To address the larger question consider that our quality software brethren urge us to, among other things:

- keep segements short - say 50 lines or so
- use one path in/one path out programming
- pass parameters between segments in a highly structured manner
- use local variables within a segment that evaporate when the segment becomes inactive.

It is my contention that all four of these things can be done simply and naturally on SOCRATES.

I would be less than truthful, however, if I left the impression that I would prefer SOCRATES to a conventional computer if cost were no object. I would not.

CONJOINED COMPUTER SYSTEMS: AN ARCHITECTURE FOR LABORATORY DATA PROCESSING AND INSTRUMENT CONTROL *

Donald F. Wann Robert A. Ellis
Computer Systems Laboratory
Washington University
St. Louis, Missouri

ABSTRACT

A new computer system organization and implementation is described and its application to problems in both clinical and research laboratory biomedical applications is outlined. This system, called a Conjoined Computer System to emphasize the idea of computer coordination, consists of a few stored-program computers, each with a fixed assigned task, and inter-connected with combined data and control paths which may be easily changed. Such systems appear to be useful in situations where a) one or more major laboratory peripheral devices are included, b) operational speed is required which is greater than that which can be obtained from a single processor implementation, and c) the algorithms utilized can be partitioned into two or more machines such that only moderate intercomputer communication rates are needed. This architecture offers the possibility of simplifying the peripheral interfacing, reducing total computation time, allowing rapid system reconfiguration, enforcing the discipline required for structured programming, and, by means of a centralized program development console, increasing the efficiency of program development.

I. INTRODUCTION

We have recently been examining the impact that "zero-cost" computers will have on the designer of scientific data processing systems for use in both clinical and research laboratory medical applications. The trend to the widespread use of minicomputers and the computer-on-a-chip will result in radical changes in the style of computer usage in these configurations. In particular, systems with multiple computers will become commonplace. By dedicating several computers to the solution of biomedical data processing problems the following advantages appear to accrue: 1) I/O communication is simplified, 2) peripheral interfacing is reduced in complexity, 3) total computation time is reduced, 4) system economies are realized, and 5) rapid reconfiguration is possible. However, such trends are not without their problems. The need for standardized and flexible intercomputer communication (both software and hardware), for simplified expansion and contraction, and for efficient programming and debugging, will become of prime importance in such environments.

Unlike other multicomputer and multiprocessor arrangements, we have developed a system organization consisting of a few stored-program computers, each with a fixed, assigned task, and loosely interconnected with fixed data and control paths which are different from system to system. We refer to this type of arrangement as a Conjoined** Computer System (CCS) to emphasize the notion of computer coordination. We make a distinction between the type of arrangement to be discussed and that of multiprocessors, array processors, and computer modules. Multiprocessor literature[1] deals principally with the application of a few rather large and powerful machines in a problem independent configuration for the solution of a given task. Reconfigurability, use of microcomputers, and interconnection schemes are not the central features of such systems. Likewise, array processors[1], although utilizing multiple computers, are normally not reconfigurable, the connections tend to be of an iterative nature, and the interface to peripheral devices is of minor concern. Workers at Carnegie-Mellon University have used the term "computer-modules"[2] to describe a modular computer system whose components are stored-program computers. The scheme seems to be intended for use in systems which consist of hundreds or even thousands of computer modules where the interconnections are not often changed.

In contrast we see immediate benefits in our system organization which consists of a few stored program computers engaged in specialized tasks and having flexible interconnection. The rationale behind this choice is predicated on the appropriate application of the CCS to the solution of laboratory computing problems. The tasks for which CCS are most useful are those which a) have one or more major peripheral devices, b) require improvements in processing speed or program efficiency, and c) utilize algorithms which can be partitioned into two or more machines such that the necessary communication between machines is at a very high programmatic level - thus requiring moderate intercomputer communication rates.

A key design principle of a CCS requires that the configuration be constructed such that a given processor module connects to a single major peripheral device. By committing a processor module to each peripheral, the interface design is extremely simple, since typically only one class of I/O is used. Furthermore, the peripheral-processor combination can be treated as an entity in programming or in configuring new systems. Thus, each peripheral can be viewed as a resource and can be easily shared among various systems.

In evaluating this approach we analyzed several existing laboratory computational tasks that had been implemented by us or our associates[3,4] and discovered that nearly all of them would benefit from this conjoined arrangement. It should be emphasized, however, that not every problem can be partitioned so as to make this an appropriate architecture. Obviously there are tasks which are best solved with single machines, with highly structured array processors, or special purpose configurations. The CCS approach merely complements these other architectures.

* This work has been supported by the Division of Research Resources of the National Institutes of Health under Grant RR-0096.

** conjoin: "to join together for a common purpose", Webster's Seventh New Collegiate Dictionary.

II. SYSTEM CONFIGURATION

A Conjoined Computer System consists of processor modules, an intercomputer communication network, peripheral devices, program development connections, and a program development console. These items are illustrated in Figure 1 which depicts a "representative" configuration and contains four processor modules, PM_1 through PM_4. Each of these modules is a complete processing unit, containing perhaps 4K or more of memory and has three types of connections to the remaining portion of the network: 1) the intercomputer communications illustrated by the heavy lines, 2) the program development connections (light lines), and 3) the peripheral device connections. The intercomputer communications lines are bi-directional paths that provide both data and control signals between machines. These paths are established by the user of the CCS and are determined exclusively by the algorithm requirements. The paths are completed by simple cable connections, thus can be easily reconfigured. For simplicity in software, daisy chain data communications are avoided if possible; observe that PM_4 communicates only with PM_3; if PM_4 needed access to PM_1 a separate cable would be installed. Each processor module has a number of available ports to this communication highway as indicated by the T-shaped connector, and the machine terminations are labelled using double subscript notation. A brief description of each of these major components is given below.

A. Processor Modules

The processor modules are self-contained, stored-program computers capable of being interconnected into networks. As such they generally must contain a central processing unit, memory, and ports for network interconnection. Our design goal was to implement a sufficiently general interconnection technique so that a given system could contain PM's of the microcomputer, minicomputer, and specialized high performance processor variety.

B. Processor Interconnections

Processor module interconnection is via processor ports. Both DMA and programmed I/O paths could be implemented; however, because of the distributed processing power of a CCS (which allows a PM to be dedicated to a single task), we have found that standard, single word programmed I/O transfers appear adequate for the majority of applications. To minimize the I/O programming complexity, a handshaking protocol path has been included in the port hardware and cabling and this establishes "port ready" levels. Generation of these protocol signals is automatic and the single word I/O commands have a test for successful data transfer. Thus, for the sending PM to complete a data transfer, a word will both have to be written by the sending PM and read by the receiving PM. This communicating protocol is identical to that used in the construction of macromodules[5,6] and will permit macromodules to be directly utilized in Conjoined Computer Systems. With the exception of a single mode bit, all other control information is encoded and transmitted in data words. Notice that the elimination of the DMA for an intercomputer communication medium does not preclude its use between peripheral devices and a processor - where it may often be the optium choice.

C. Peripheral Devices

As noted in Figure 1, processor modules 1, 2, and 4 have peripheral connections. Traditional design techniques are used for interfacing these devices to PM's via device ports and this usually permits standard manufacturer's interfaces to be employed. If a special interface must be designed, the interfacing task is simplified because at most a single major device is attached to each PM and control of the device by the system is through the PM. This means that each device can be addressed and controlled at a much highe level than is normally available. However, because of economic considerations, the one peripheral device - one processor plan may have to occasionally be modified, but continued reduction in microcomputer costs should alleviate this problem.

D. Program Development Network

The configuration just described - PM, PD, and intercomputer connections - is the basic computation network. Dr. Charles E. Molnar of the Washington University Computer Systems Laboratory has proposed two additional elements for efficient program development: the program development console (PDC), and connections from the PDC to each computer module. Prograr development consists of preparation, editing, filing, loading, and debugging. The PDC contains a CPU, system I/O devices (disk, magnetic tape, etc.) adequate fast memory to run system programs, keyboard and scope, and a hard copy device. Use of the PDC makes program development very convenient and in general the PDC can be quite elaborate in order to insure maximum programmer productivity. As a consequence, most of the manufacturer's software can be utilized with little or no modification. This equipment does not need to be permanently committed to a given system for the PDC is a self-contained, portable unit that can be shared among systems and users. The PDC can be removed from the CCS when program development is not taking place.

The network connects the program development console to the processor modules via ports which provide access to the machine state, e.g., address register, memory contents, program counter, control signals, etc. Because the PDC need not simultaneously communicate with more than one PM at a time, this network is formed by a single, daisy-chained path. This requires that the PDC select one PM and all signals which follow apply only to the selected PM until a new selection is made. Because of the sequential transmission of information on this network, the PDC port provides a number of features - such as address matching for synchronized program stopping - which require fast reaction to changing machine states.

Since the PDC is removable, the rest of the CCS should contain whatever devices are necessary (disks, tapes, scopes, etc.) for operational use; these are uniquely associated with the functioning system. Therefore, when the PDC is not connected to a specific CCS, some alternate equipment must be present for system start-up and program loading. Obviously, this equipment is much simpler (and less expensive) than the full PDC.

1. Program Preparation. Program preparation consists of the steps of text entry and editing, assembling, compiling, link-editing and filing. The PDC can use standard software products to perform these tasks. Because the PDC is self-contained, connection to the CCS is not required for program preparation. Programs can be prepared to run in PM's which may or may not support an operating system. The CPU in the PDC does

not have to be the same as the PM's; cross assemblers and compilers can prepare code for computers of different types.

2. Program Loading. The initialization and program loading of each PM can make use of standard bootstrapping procedures because the PDC has access to the PM machine states. It is also possible to use standard, or only slightly modified, loading programs and load module formats. As mentioned earlier, a method which does not rely on a PDC is also required for these steps.

3. Program Debugging. Programs in individual PM's typically are simpler than those written for conventional minicomputers because of the dedicated task organization of the system. However, the programs will be complicated by the intercomputer communication. It is for this reason that the PDC is physically connected to all of the PM's in a system. It is thus possible to use standard program debugging techniques, both control panel and software procedures. For example, the usual coresident debugging packages can be executed with little or no modification. Or new debugging systems may be developed which insert breakpoints and examine and modify memory locations, all controlled from the PDC. In addition, because the PDC has access to the PM machine states, "virtual consoles" can be implemented in the PDC by software. This permits active control-panel-style operation of more than one PM at a time. Of course, the PDC could also contain a special physical control panel which could be connected via software to any PM in the system.

III. IMPLEMENTATION

This section contains a description of the actual implementation of a Conjoined Computer System. For simplicity, the CCS to be discussed utilizes only a single type of computer, the Texas Instruments 980A[7]. These machines are sufficiently low in cost for several to be used in a single system, they possess characteristics that are similar to other minicomputers, and they were readily available in our Laboratory. As will be explained later, communication rates between machines of nearly 200,000 words per second can be achieved using the programmed I/O facilities and thus all PM interconnections were made via this type of channel. The direct memory access ports, which will yield rates of about one million words per second, were not used.

A. Communication Pathways

To establish a communication path between two machines it is necessary to insert a special CCS printed circuit card into an I/O slot of each of the machines and to connect these cards together using a communication cable. Each I/O card corresponds to a "port" and is assigned a port number, via a thumbwheel switch on the card. The present design permits 8 ports (numbered 0 to 7) and, since it is unlikely that a computer network would be maximally connected, should allow a quite elaborate configuration of machines. Provision has been made for future expansion to a total of 16 ports.

A port contains both a "send" and a "receive" channel which are basically identical in structure. Each channel has the following signals:

16 data lines
1 mode line
1 data delivery line
1 data delivery return line

The data delivery and data delivery return lines employ transition logic[8] and provide for handshaking protocol signaling between machines. The remaining lines use conventional two-state levels to indicate 0/1 values. The mode line is employed to identify the type of information that is being sent (received). This feature permits the sending of a 16 bit data word (mode = 0) or a 16 bit control word (mode = 1), the line being used at the receiving port to determine the mode of the received data. In the implementation described here, the sending of a control word always results in an interrupt request. When the interrupt state is entered the control word can then be read by the program in the receiving computer. For example, it might be desired to have the control word cause a disk to start seeking, etc.

B. Instructions

For each port there are three basic functions that can be performed, with each function being subdivided into a read or a write action. This results in a total of six instructions per port. These instructions are summarized below, where n is the port number:

Send Data Word to Port n
Receive Data Word from Port n
Send Control Word to Port n
Receive Control Word from Port n
Write Status Word to Port n
Read Status Word from Port n

The send and receive instructions (data or control) check the state of the channel (via the channel state hardware) and if the channel is "ready" they may transmit or receive a word. In this situation the next instruction is skipped. If a word cannot be transmitted or received, the next instruction, which is usually a branch, is executed and this can cause the transmitting or receiving instruction to be re-executed.

The status word contains bits to enable/disable the interrupt, to reset the channels, and to sense the channel status.

C. Data Transfer Rates Between Processors

It is useful to determine the rate at which words can be transferred from processor to processor via the intercommunication pathways. For simplicity, we shall only consider communication between two machines: the rates would be modified depending on the actual number of processors in a network. Let the transmitting processor be denoted by T, and the receiving processor by R. Since the machines will be executing individual non-synchronized instructions (each responding to its own system clock) it is not possible to compute exact rates; however, lower and upper bounds can be easily estimated. The form of the machine language program*** necessary for communication is listed below:

Machine T	Machine R
Write	Read
Branch	Branch
Test & Branch	Test & Branch

The Write instruction employes indirect addressing to obtain the data from the memory and a "busy" test is included in the instruction in order to check the channel status. If the channel is free (not busy) then the

*** This is the format for the TI980A. Other machines would have similar programs.

next word is placed on the channel, the Branch instruction is skipped and the Test & Branch instruction executed. If the channel is busy, the unconditional Branch is executed and this causes the Write to be tried again. The Test & Branch instruction determines if all words of the block have been sent; if not, a branch to the Write is effected - otherwise the transmission is finished and the machine proceeds to the next section of code. The program for processor R operates in a completely analogous manner except that the machine is reading data from the channel rather than writing data onto the channel.

The rate at which words can be transferred from T to R can be determined under two different conditions: 1) T always finds the channel available and R can read the data from the channel as rapidly as it arrives, and 2) T and/or R must occasionally wait while a word is being transmitted over the channel (or the protocol signals are being generated). The first case will yield the maximum theoretical transfer rate and is determined from the typical transfer paths diagrammed in Figure 2. The instruction execution times and transmission path delays are indicated and they result in a 4.5 microsecond delay per word, or a rate of nearly 220,000 words per second. In the second case we assume that the logic and transmission delays are sufficiently large that the sequence of Figure 2 cannot be obtained. Instead, the busy condition is encountered and the Branch instruction (and an additional Write and Read instruction) are executed. Computations show that the delay time is lengthened to about 8.75 microseconds yielding a transfer rate of about 114,000 words per second. To verify these values, tests were run with two TI980A computers. Depending on whether direct or indirect addressing was employed, rates of between 180,000 and 133,000 words per second were obtained.

D. Program Development Console

The implementation of the PDC including the communication to the various machines, the operating software, and the console interactions, is currently in progress. There are a number of very interesting techniques and programmatic aspects of this investigation, including problems of resetting, restarting of the multiple machines, and of high level debugging that are being considered. Details of the current work will be reported on in the future. It should be noted, however, that the PDC has broader application possibilities beyond the CCS architecture, and could be used as a general aid in minicomputer and microcomputer program development.

E. Programming

A set of subroutines has been written which may be called from FORTRAN or assembly language programs to perform CCS operations. Data routines poll ports and send or receive data. Control routines, in addition to sending or receiving data, cause interrupts on ports which are properly enabled and handle enabling, servicing, and disabling interrupts.

1. Data Routines

a. SEND (X,N,D). Send one data word, D, to the specified port N. If the data is not accepted into the buffer, the routine returns to X.

b. RCEIVE (X,N,D). Receive one data word from port N and put it at D. The routine goes to X if no data is present.

c. SENDM (N,M,D). Send M data words to port N from D. Return only after M words have been sent.

d. RCEIVM (N,M,D). Receive M data words from port N and store them at D. Return only after M words have been received.

e. TEST (X,D,N). Test for port requesting activity in the order defined in a table. Return with the port number in N and the data word in D. If all ports are tested and no requests are present, go to X.

f. NEXT (X,D,N). Continue testing for receipt of a data word in D. If all ports are tested and no requests are present, go to X.

2. Control Routines

a. SENDC (X,N,C). Send one control (interrupt) word C to port N. If the word is accepted and interrupts have been enabled, an interrupt occurs in the receiving processor. If the word is not accepted, the routine returns to X.

b. ENABLE (N,SUB). Enable interrupt on port N. SUB is the entry point of a user subroutine where control is transferred after an interrupt on N.

c. DISABL (N). Disable the interrupt on port N.

IV. APPLICATION

The CCS architecture is well suited for laboratory-instrumentation type tasks where a number of peripheral devices must be controlled. In the past this has usually been accomplished by employing one central processor. Since the peripheral devices are usually more costly than the CPU, the result is often a system in which sharing of the various peripherals from project to project is quite difficult. In addition, often there is a tendency for such single computer systems to grow in an uncontrolled manner. For example, a $100,000 configuration may be performing five $20,000 tasks. With the conjoined approach it is possible to easily create separate configurations and/or to share the peripheral resources, since both the hardware and often much of the software necessary to communicate with a device is associated with the processor module.

As an example of this technique, we will consider a system for counting of microscopic silver grains overlying biological tissue. A system is currently being employed that utilizes a single computer, a laboratory microscope, a computer controlled microscope stepping stage, a television camera for viewing of the tissue through the microscope, a CRT display for on-line viewing of the processed video, and a keyboard-scope for communication with the user[9]. The arrangement is depicted in Figure 3, which also indicates the peak data rates that can be expected along the various channels. The image processing algorithm involves the movement of the stage to focus the image, collection of video data, analysis of this data, display of the analyzed data, movement of the stage in the lateral direction to a new field, and occasional operator interaction. Because of the variety of peripheral devices, and because of the asynchronous nature of the data collection of the television camera, the programming requires the use of various interrupt service routines. The overhead involved in these routines, plus the time necessary to refresh the display,

move the stage, etc. severely limits the spped at which the individual grains can be counted. As a result, we are now in the process of implementing this in a conjoined format as is illustrated in Figure 4. Here four processor modules are used, each connected to an individual peripheral device. By this dedication, the interrupt mode is eliminated and the program development for each module is quite straightforward. The various data paths are dictated by the partitioning of the algorithm operations into functional blocks, many of which can be overlapped. A condensed version of the system operation is outlined in Table 1 which indicates the task of each processor module as well as the time period devoted to its execution. Our preliminary evaluation suggests that the CCS configuration (using processor modules with memory cycle times comparable to the existing PDP-12) would achieve a doubling of processing speed while also greatly simplifying both the hardware operation and software programming.

V. SUMMARY

The availability of conjoined computer system components offers several significant advantages to the designer of laboratory data processing and instrument control systems in which a variety of peripheral devices have an important role. In particular we see the following benefits:

1) The ability to solve certain real-time processing tasks should be improved since the use of several processors allows concurrent manipulation of input data, output data, and computations.

2) By assigning a single processor to each major peripheral device a significant reduction in interface complexity with the peripheral can be achieved.

3) Rapid reconfiguration of a system including both expansion and contraction, is possible.

4) Peripherals could be shared among users, since a high level, standardized communication (via the processor module) is available.

5) Partitioning the algorithm into hardware rather than use of memory address division should minimize (or virtually eliminate) the occurrence of adverse program interactions in which one program in memory alters another.

6) Since modularity is at a very high level and inter-computer communication is standardized, both rapid design and economical implementation can result. The employment of multiple copies of the same processor will yield cost economies since advantages can be taken of the obvious trend toward high volume, low-cost microcomputers.

7) The application of the program development console will simplify the software - even though multiple processors are utilized. This may well be one of the major achievements of the project.

8) The system organization enforces the discipline required for structured programming.

REFERENCES

1. Multiprocessors and Parallel Processing, Philip H. Enslow, Jr., Editor, New York, John Wiley, 1974.

2. Fuller, S. H., Siewiorek, D. P., and Swan, R. J., "Computer Modules: An Architecture for Large Digital Modules", Proc. of the First Annual Symposium on Computer Architecture, IEEE, Dec. 9-11, 1973, pp 231-237.

3. Wann, D. F., Woolsey, T. A., Dierker, M. L., and Cowan, W. M., "An On-Line Computer System for the Semi-Automatic Analysis of Golgi-Impregnated Neurons", IEEE Transactions on Biomedical Engineering, Vol. BME-20, pp 233-247, 1973.

4. Cowan, W. M., and Wann, D. F., "A Computer System for the Measurement of Cell and Nuclear Sizes," J. of Microscopy, pp 331-348, Dec. 1973.

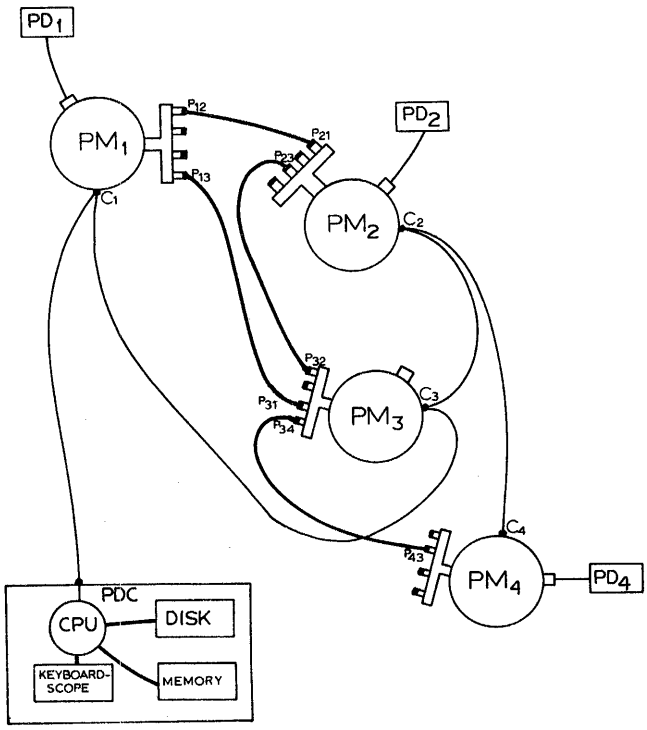
5. Clark, W. A., et.al. "Macromodule Computer Systems", Proc. of S.J.C.C., Thompson Book Co., Washington, D.C., pp 335-364, 1967.

6. Macromodular Computer Design, Final Report, February, 1974, Computer Systems Laboratory, Washington University, St. Louis, Mo.

7. Texas Instruments Incorporated, Digital Systems Division, Houston, Texas 77001.

8. Stucki, M. J., "An Approach for Synthesizing Transition Logic Circuits", Proc. 11th Annual Allerton Conference on Circuit and System Theory, October, 1973, pp 418-427.

9. Wann, D. F., Price, J. L., Cowan, W. M., and Agulnek, M. A., "An Automated System for Counting Silver Grains in Autoradiographs", Brain Research, 1974 (in press).



PM ⇒PROCESSOR MODULE
PDC ⇒PROGRAM DEVELOPMENT CONSOLE
PD ⇒PERIPHERAL DEVICE
C ⇒CONSOLE CONNECTION
P ⇒PROCESSOR PORT

FIGURE 1

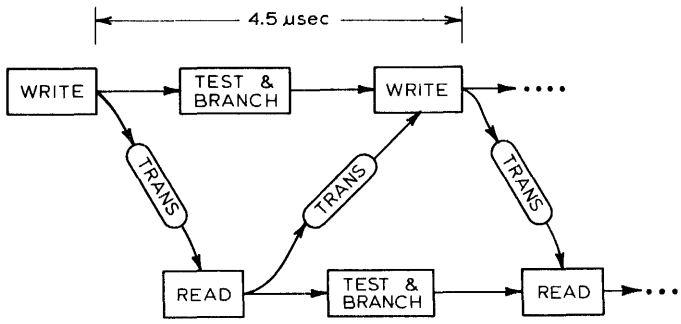


FIGURE 2

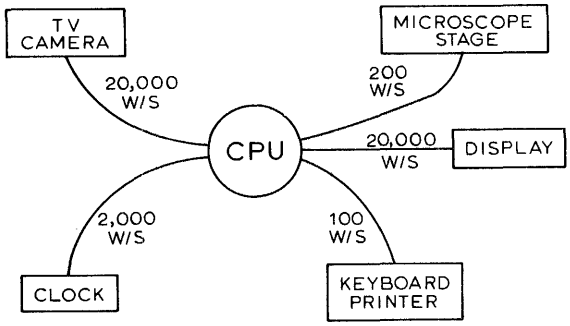


FIGURE 3

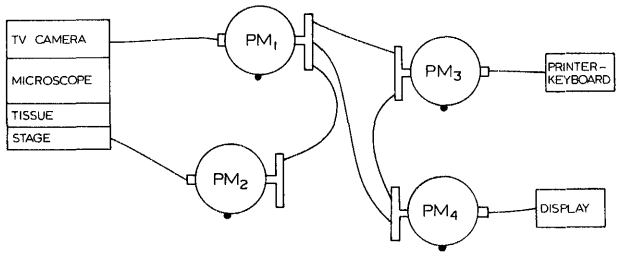


FIGURE 4

PM_1	PM_2	PM_3	PM_4
Sends data to PM_2 for focus	Receives data from PM_1 and performs automatic focus;	Executes counting algorithm on line of data	Displays data
Acquires data and performs preprocessing	Idle	Executes counting algorithm on line of data	Displays data
Transmit data to PM_3	Idle	Receives new line at data from PM_1	Clears Display File
Idle	Moves stage in X, Y	Executes counting algorithm	Displays data

Table 1

A DISTRIBUTED FUNCTION COMPUTER FOR REAL-TIME CONTROL

E. Douglas Jensen

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway NE, Minneapolis, Mn. 55413

Abstract

Due to advances in hardware technology, processors are no longer the limiting factor in system costs--the software and configuration portions are now dominant. Hardware can be effectively applied to these problems as well: low processor costs have revitalized multiprocessors and multicomputers; the executive functions entailed by such architectures can be facilitated with powerful yet flexible hardware mechanisms. This paper outlines the hardware aspects of an experimental distributed function computer intended to examine these issues in the context of real-time control systems.

Introduction

Modularity is one of the most critical aspects of a computer embedded in a real-time control system [1]. Each application is usually unique, with initially ill-defined computer requirements which continue to evolve throughout the lifetime of the system. Therefore, the computer must readily accomodate incremental adjustments in capacity without significantly impacting either the hardware or the software. This problem is particularly acute in aerospace and military applications, where changes can precipitate long and costly reverification and requalification.

Multisystems (multiprocessors and multicomputers) are one of the most promising methods of providing modularity [2]. Until recently, hardware costs have relegated multisystems to those relatively few applications which demanded (and could afford) the higher reliability and performance that such configurations (at least potentially) offered. Even in these select circumstances, however, the results have many times been less than desired. This has been primarily due to the economic necessity of maximizing processor utilization, which has led to incomprehensibly complicated multiprogramming/multiprocessing executives. Because of rapid advances in hardware technology, processors are no longer the limiting resource--creation and maintenance of these baroque software structures is now often the dominant portion of system cost. Thus, as the original constraint is disappearing, continued application of the remedy is itself becoming an even worse constraint.

While individual processor efficiency is of diminishing interest, system-wide supervisory overhead is still critical, especially in real-time environments. The development of new components as well as improved design and implementation techniques is making it increasingly feasible to augment executive software with hardware. Highly sophisticated functions can be consigned to hardware which exhibits much the same degree of malleability as software. This suggests that a careful re-evaluation of traditional hardware/software interfaces may be in order when designing a new computer.

The enhanced flexibility and precipitous drops in cost of contemporary and future hardware may form the basis for new multisystems which can effectively reduce the predominant software and configuration costs. Other benefits to be expected from such architectures include increased reliability, faster response times, and the possibility of physical dispersal. To help examine these issues, an experimental distributed function computer has been initiated. The effort is directed toward real-time control systems of the late 1970's, including aerospace and military applications. Therefore, many of the tradeoffs involving hardware are based on the projected technology of that period, even though the experiment must employ today's hardware. The computer has been designated the Modular Computer System (MCS).

System Philosophy and Overview

The geographic distribution requirements of real-time control systems typically vary from a few feet to a few thousand feet. This factor has a dramatic impact on the interconnection mechanism and thus the total architecture of a multisystem. The MCS is oriented toward the more physically centralized applications. For one reason, the more decentralized applications are the object of another concurrent design (to be reported on at a later date); in addition, the perspective of this project is the construction of one larger machine from a number of smaller ones, rather than the interconnection of separate computers.

Most recent multisystems are multiprocessors [3, 4, 5, 6, 7, 8], having a shared memory in the address space of all processors [9]. The MCS is a multicomputer, in which all interprocessor communication takes place over a set of global busses, as shown in Figure 1. Most earlier multicomputers (e.g., IBM DCS, ASP) were channel-interconnected; the MCS also differs from other modern multicomputers which have similar goals but in the context of greater geographic dispersion [10], and from networks which have entirely different objectives [11].

Each processor has a private memory containing all of its procedure and most of its data. A processor and its memory constitute a processing element. Partitioning the processing requirements is facilitated by the fact that dedicated control systems operate in a well-characterized environment--tasks are performed by some subset of a known set of programs, but perhaps in unknown sequences and with unknown timing. Even with this advance knowledge, the partitioning is still a non-trivial job. The aims and approaches of normal program modularization remain important: clean decomposition makes it possible to develop each module independently, to change one module with minimum impact on the others, and to understand the system by studying one module at a time [12]; a hierarchical structure simplifies higher level modules through the use of those on lower levels, and allows diverse systems to be built on common foundations [13]. The parallelism inherent in a multisystem adds another dimension to the problem--critical considerations here include the iteration rates, precedence relationships, and synchronization of modules. At the same time, the loose coupling between multicomputer processing elements constrains the intermodule communication. Protection is another issue which must be taken into account.

The final aspect of partitioning in the MCS constitutes a significant departure from the historical direction of computer system development. In recognition of the changing balance between hardware and software costs, program modules are assigned to processing elements effectively on a one-to-one basis. Multiprogramming is minimized by committing modules which must be unpredictably initiated to their own processing elements--processing elements are shared only by modules which are known to execute mutually exclusively. Preemption of modules is avoided where at all possible--interrupts and intermodule messages queue themselves for later handling, or activate an idle processing element. To prevent deadlocks, one module in a processing element may respond to a message while another module in that processing element is blocked (waiting for an answer to a message). With proper care, some modules may be replicated to reduce either multiprogramming of a processing element or traffic between processing elements. Certain infrequently referenced global data items are kept in the processing elements of associated modules. Other global items are accessed often enough that this data management task would interface with the primary function of the host--these items are therefore placed in processing elements dedicated to administering references to them. This philosophy substantially uncomplicates scheduling, resource allocation, and intermodule communication.

The processor capabilities could be chosen to exactly meet the needs of the partitioned modules for each application, but this would result in severe adaptability restrictions as well as logistical disadvantages. Instead, the MCS processors are homogeneous, and sized such that the demands of the largest most likely module can be handled by one processor. Also taken into account was the priority given to software simplicity over maximizing concurrency or minimizing processor cost. This means that in some cases it may be cost-effective to employ many processors of such a size that one could do the whole job. Consequently, while the processors may be overpowered for some module requirements, there is little if any economic incentive to fully utilize them. Processing element memories are typically on the order of 1K to 4K 16-bit words, and while there currently appears to be little need for secondary memory, it can easily be provided. Any of several commercially available minicomputers would be suitable as the MCS processing element; it is reasonable to expect that appropriate bipolar LSI microcomputers will appear in the foreseeable future. The intended applications appear to require configurations of between 10 and 100 processing elements.

To be successful, a distributed function computer must be founded on flexible, modular intercommunication mechanisms for both hardware and software. The MCS incorporates a pool of identical and autonomous global busses, the number of which depend on bandwidth and reliability requirements; the current implementation has up to four. For both functional and reliability reasons, most I/O devices must be accessible to more than one processing element, so they are attached to the global busses. Private I/O devices are also allowed. High speed I/O devices which require direct memory access are rare, but would be interfaced to one or more memory busses as well.

Because autonomy is an important partitioning criterion, intermodule coordination and communication are reduced--that which remains is highly disciplined. All intermodule contact occurs via messages on the global busses. The bus interface hardware automatically routes messages properly, regardless of static or dynamic processing element assignments --any co-residency is invisible to the software. Global data accesses and external I/O operations are also accomplished in the same well-defined way. Since hardware supports all forms of module interfacing, these standard protocols extend into the lowest levels of the system software.

A hardwired general purpose multiple semaphore mechanism is included as one of the devices on the global busses, to evaluate its utility in such areas as processing element, I/O, and module scheduling, resource allocation, intermodule communication, etc

All these powerful hardware augmentations are inexpensive, and yet implemented so as not to stifle exploration and adaptation.

Because the global busses are the key feature of the MCS, they are described in greater detail below.

Global Busses

The Global Busses (GBs) are paths used by every processing element for intermodule communication and low to medium speed external I/O. Some form of redundancy is necessary so that single-point GB failures do not disable the whole system. A bus which is physically redundant but functionally singular would be of no benefit to the system until an exceptional condition (bus failure) occurred. On the other hand, identical but independent busses not only provide this same protection, but also supply added band width under normal circumstances. If it is desired that the system be failsoft rather than fail-operational full advantage can be taken of this extra bus bandwidth otherwise, the system must be designed around the capabilities of the functionally minimum bus complement. Up to four autonomous GBs may be included to meet data rate and reliability requirements.

Because all devices are closely spaced, the cost penalties of parallel transmission over serial transmission are minimal, so each GB is one word (16 bits) wide (plus control). The higher bandwidth of a parallel bus reduces transfer latency while compensating for protocol and allocation overhead; it also allows for external I/O traffic, and insures ample margins for growth.

All devices employ identical interfaces which handle GB control and communications, as shown in Figure 2.

Bus Control

A GB is shared by all devices, so there must be a mechanism for guaranteeing that exactly one device is transmitting on it at any time. Devices are either masters or slaves: masters (processing elements and some I/O devices) may obtain control of the bus and initiate transfers on it; slaves (most I/O devices may only participate in such transfers. Each bus is independently allocated by bus controllers at every master, as seen in Figure 2. With either centralized or decentralized control there will always be some potential failures which would disable a bus, which is a principal motivation for multiple busses. But failures in a centralized controller are more likely to affect all of its busses and thus perhaps disable the entire system. The reliability of properly implemented decentralized control is superior because a failure will probably only disconnect that device from

that bus. Decentralized control can also be more modular: smaller configurations do not have to pay a penalty for capabilities needed only by larger machines --costs can vary smoothly with system size. Finally, decentralized control is faster in that each device makes its own bus allocation decisions instead of waiting to receive signals from a remote control controller.

Selection of an appropriate bus control procedure depends on such requirements as expected distribution of bus usage among devices, degree of responsiveness to device bus requests, flexibility of the allocation algorithm, variability in the number of devices, etc. As a consequence of the MCS design philosophy, the bus usage of each device can be forecast with reasonable accuracy. This implies the feasibility of a static bus control scheme. Whereas a dynamic mechanism assigns busses according to the priorities of real-time device demands, a static controller preschedules busses according to a priori knowledge or prediction of device needs. A device which requires a bus must wait until its next turn; the delay depends on the number of busses implemented, the number of devices earlier in the schedule which want a bus, and the length of time these prior users require. When a bus becomes available to a device which does not need it, control is passed immediately to the next device in the sequence. Since intermodule communication is one of the prime partitioning criteria, the full allocation flexibility of a dynamic bus controller is unnecessary--adequate responsiveness is further insured by the availability of multiple busses, and the higher speed of word-parallel transfers.

An important requirement for the GB controller is modularity, in that varying the number of devices should not have significant hardware impact. Dynamic algorithms (such as the method of independent requests [14]) tend to involve a direct linear function between the number of devices and the number of bus control lines. The cables, connectors, and control logic must either be initially sized to accomodate the maximum allowable configuration, or be capable of incremental adjustments in capacity. The former alternative is not only unmodular, both logically and financially, but also impractical for moderate to larger numbers of devices (e.g., independent requests require two control lines per device). The bus controller can be designed to allow for modular increments of logic, although generally the outweighing mechanical costs yield no savings over the first alternative. Furthermore, cables and connectors do not readily lend themselves to the modularity of the latter option.

A popular static bus control approach which is highly modular is daisy-chaining [14]; an arbitrary number of devices may be inserted or deleted at any point in the loop without affecting the control lines. On the other hand, daisy-chaining is slow (control signals must propagate through every device in the loop), unreliable (preventing device failures from incapacitating the entire system is difficult), and inflexible (the assignment algorithm is constrained to be round robin according to physical sequence on the bus).

The control procedure devised for the GBs exhibits the modularity of daisy-chaining without the attendant disadvantages. The availability of a GB is divided into time slots. One message of (limited) variable length may be transmitted in each time slot; a master may receive a reply from a slave in the same time slot. The time slots are preassigned to devices at system configuration time. Devices may be assigned multiple time slots in each complete allocation cycle to guarantee them various degrees of responsiveness or bandwidth. The schedules for each GB are independent, so that devices may have different priorities on each bus. Figure 3 shows that each bus controller contains a programmable read-only memory, having as many bits as the maximum number of time slots (currently 256), and a counter which addresses the memory. The address counters are all initialized at zero, and incremented together by a signal on the Bus Sync line at the end of each message. The length of the cycle is programmable; after the last time slot, the address counters reset to zero. A device receives control of the bus during every time slot corresponding to a "one" bit in his memory. If the device needs the bus, it transfers a message and generates Bus Sync; if the device does not need the bus, it generates Bus Sync immediately. The bus control hardware automatically detects and recovers from such problems as the next scheduled device failing to either use or pass control of the bus, or bus controllers somehow getting out of sync with the schedule. Because signals are not looped through all devices, control is fast and reliable. The assignment sequence may be whatever is desired. Changing the number of devices may involve substituting a different read-only memory chip and altering the counter reset point, but even this can be avoided at the expense of some wasted bus bandwidth.

Simultaneously available busses are resolved by an arbitor in the output controller, one bus being accepted and the others refused.

Bus Communication

GB communication is handled by communication units which connect each device to all busses, as shown in Figure 2. These units are functionally identical for all devices, although there are a number of options which allow them to be tailored for specific needs. The principal elements of a communication unit are the output controller, the input controller, and the communication controllers.

The GB message format for master devices is illustrated in Figure 4; messages from slaves do not include the destination name or bus control sync. In the case of a processing element, the bus control sync field and CRC word are supplied by hardware, and all else by software. The present design permits messages to contain up to 256 data words.

Output -- The output controller contains a FIFO queue, into which the device inserts messages to be transmitted. When a bus controller receives control of a GB, the output controller checks to see if its queue is empty; if not, it asynchronously removes one message and sends it via the appropriate communication controller; if so, it relinquishes control of the bus. Messages may vary in length up to 256 words; in the applications considered thus far, they usually range from one to 16 words. The current implementation permits the output queue to be one, four, 16, 32, 48, 256, or 1024 words long.

GB transfers occur in an asynchronous, ready/accept fashion. This provides automatic synchronization between devices of differing speeds, and a positive or negative response on the correctness of every word transmitted. The communication rate is limited by the fact that four bus propagation delays are required per word, but adequate bandwidth is assured by the availability of up to four word-parallel busses, the short distances between devices, and the inter-process communication partitioning criterion.

If a parity error is detected by the message destination, the source output controller will retry the word up to three times. Failing in these, it will terminate the transfer and notify its device; the destination input controller will invalidate the entire message. The output controller may be set either to skip this message and send the next one in the output queue when it next receives GB control, or to do no further transmitting until instructed to resume normal operation.

The output controller optionally appends a polynomial check word to the end of each message. If the destination input controller does not correctly decode this word, the entire message is repeated (in this same time slot). After three unsuccessful retries, the message is aborted and the source device notified.

If the destination device bus interface responds as busy to an attempted transfer, the source output controller retries the message the next time it receives control of a bus. After three unsuccessful retries, it may wait and then re-initiate the message, or it may notify the source device and (in the case of an I/O device) send an error message to another destination.

A variety of other errors are also detected and responded to by the input and output controllers, including optional record-keeping on errors.

The output queues are implemented with random access read/write bipolar semiconductor memories and address pointers. The output controller operates the pointers in a circular fashion; signals are available which indicate when the queue is non-empty and when it is full. So that message space in the queue is not released prematurely, the queue full signal is generated using the read pointer value for the last message successfully transferred. Attempts by the device to enter data in a full output queue are rejected.

For a processing element, the output queue appears as a single word in its memory address space. The processing element software loads the desired queue input address into a register in the output controller. Thenceforth, a store at that memory location will cause the data to be entered into the rear of the output queue. The most recent queue entry may be (nondestructively) read from that location.

When communicating with a slave device, a processing element may elect to retain control of the GB after a message is sent to possibly receive a reply. Such a reply will be returned only if the slave's input queue is empty (to avoid tying up the bus) and its output queue is not (data is available); otherwise (or if the selected destination is not a slave), a negative response will occur.

When a message has been correctly received, the output controller reallocates the GB by raising the Bus Sync line.

Input -- The conventional approach of addressing physical devices on the GB is replaced by the concept of associative routing to module names [15]. Every device is assigned one or more names corresponding to its software modules or global data items. Each input controller includes an associative memory which contains all the names for that device. GB messages (from masters) begin with a one-word destination name, which is compared with all names in all device input controllers. The input controller (if any) which finds a match then receives the message. Multiple devices matching will cause an error which is usually, but not always, detected. Failure of any device to respond to the destination name is detected by a time-out in the source output controller. The input controller resolves the simultaneous arrival of destination names on more than one GB--the output controller response time-out is long enough to accomodate the worst case delay. The associative memory in an input controller may presently have one register or multiples of four up to 256. The associative memories are read/write, so names may be assigned either permanently or dynamically. Names are automatically checked by the input controller when being entered to avoid (local) duplication. A default name is automatically supplied from a read-only memory at power-up.

The input controller relates each name to a unique input queue, as seen in Figure 5. Messages may be received on all GBs at once if they are directed to different destination names; simultaneously arriving names are handled in an arbitrary sequence. The device communication controller will respond busy to a message for a name which is already receiving a message on another GB; the source output controller will automatically retry later. The queues operate on a FIFO basis--the communication controllers insert messages at the rear, and the device asynchronously removes them from the front. The first word of a message stored in a queue is its word count--the destination name is omitted. The specific implementation of the input queues allows them to individually be one, 16, 32, 48, 256, or 1024 words in length. When a queue is full, any message for it is rejected and will be retried later. Even though the word count of an incoming message may be larger than the space available in the selected queue, the message words will be accepted until the queue is full. This allows for the possibility that during the message transfer period the destination device may read enough words from the queue to prevent overflow. If queue overflow does occur, the message will be rejected (to be retried), and whatever space it already occupies in the queue will be reused for the next message to that name. Incomplete messages in a queue can also result from transmission errors. Consequently, the input queues (like the output queues) are constructed from random access memories and pointers to facilitate garbage removal.

To a processing element, the input queues appear as one word each in its memory address space. The processing element software loads the desired memory addresses into another associative memory in the input controller. Thenceforth, a read at one of those addresses will access and remove the front word of the corresponding input queue; the processing element main memory will be inhibited from responding. Since input queue reading is destructive, the software must keep the message currently being processed in a main memory buffer. Writing at an input queue address writes into main memory. An attempt to write a duplicate address in the address mapping associative memory will cause an error response to the instruction.

The software currently has primary responsibility for protection: knowledge of its name is considered prima facie evidence of the right to communicate with a module; verification of the exact operation is then carried out by software.

There may be some question as to whether the input queueing hardware oversteps the bounds of mechanism into policy, at least in its prescription of FIFO

access. Thus far, FIFO appears to be most appropriate for the intended applications; however, associative queues [16] are being studied as a more general alternative.

Interrupts

In conventional real-time computers, an interrupt system is used to achieve rapid response to events in the environment. However, the asynchronous nature of interrupts implies a particularly intricate form of multiprogramming, which is contrary to the spirit of the MCS. The hardware minimizes interrupts by treating all external I/O in the same way as intermodule messages, queueing incoming messages, and handling many abnormal conditions without software intervention. There may sometimes be those interrupts which cannot be eliminated--these are considered during the partitioning. A hierarchical, vectored interrupt structure is also available on each processing element. Beyond this, a hardware augmented time-driven scheduling system is under investigation as a more suitable approach [17].

Conclusion

The MCS is a "software first" experimental computer based on the premise that in many applications the hardware constitutes a rapidly diminishing percentage of system lifetime cost. One result of this is that the global bus interface logic is approximately an order of magnitude more complex than the typical 16 bit minicomputer processing element which it serves. While there is ample evidence in support of the premise, there are also those real-time applications to which it (and therefore this architectural consequence) do not apply--small subsystems requiring only minimal software, for example.

This brief description of the MCS has focused on some of its hardware features, but there is also a simultaneous and interacting software task. An initial functional design has been completed, and the detailed implementation of a feasibility model has been initiated. Despite the formidable challenges remaining in such areas as partitioning, this experiment is already proving to be a successful technology development effort.

Acknowledgement

This work was sponsored in part by the Aerospace Division of Honeywell, St. Petersburg, Florida. It is one of several distributed computer projects in the SRC Computer Technology section which have mutually benefitted from the interchange of ideas. In particular, I am grateful to my colleagues George Anderson, Larry Jack, Mel Johnson, and Terry Saxton for their valuable contributions.

References

1. Marley, J.H., "Embedded Computers--Software Cost Considerations," Proc. National Computer Conference, May 1974.

2. Miller, J.S., D.J. Likly, A.L. Kosmala and J.A. Saponaro, "Survey of Multiprocessor and Multicomputer Systems," Multiprocessor Computer Systems Study (Chapter 2), NTIS N70-41238, March 1970.

3. Baskin, H.B., B.R. Borgenson and R. Roberts, "PRIME--A Modular Architecture for Terminal Oriented Systems," Proc. SJCC, May 1972.

4. Bell, C.G., and P. Freeman, "C.ai--A Computer Architecture for AI Research," Proc. FJCC, December 1972.

5. Wulf, W.A., and C.G. Bell, "C.mmp--A Multi-Mini-Processor," Proc. FJCC, December 1972.

6. Heart, F.E., S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," Proc. NCC, June 1973.

7. Syms, G.H., All Applications Digital Computer: Course Notes, NTIS AD-767327, March 1972.

8. Fuller, S.H., D.P. Siewiorek, and R.J. Swan, "Computer Modules: An Architecture for Large Digital Modules," Proc. Symposium on Computer Architecture, December 1973.

9. Comtre Corporation (P.H. Enslow, Editor), Multiprocessors and Parallel Processing, Wiley, 1974.

10. Anderson, G.A., "Interconnecting a Distributed Processor System for Avionics," Proc. Symp. on Computer Architecture, December 1973.

11. Rustin, R. (Editor), Computer Networks, Prentice-Hall, 1972.

12. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems Into Modules," CACM, December 1972.

13. Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press, 1972.

14. Thurber, K.J., E.D. Jensen, L.A. Jack, L.L. Kinney, P.C. Patton, and L.C. Anderson, "A Systematic Approach to the Design of Digital Bussing Structures," Proc. FJCC, December 1972.

15. Farber, D.J., J. Feldman, F.R. Heinrich, M.D. Hopwood, K.C. Larson, D.C. Loomis, and L.A. Rowe, "The Distributed Computing System," Proc. COMPCON, February 1973.

16. Jensen, E.D., "Mixed-Mode and Multidimensional Memories," Proc. COMPCON, September 1972.

17. Thurber, K.J., and L.A. Jack, "Time-Driven Scheduling," Proc. COMPCON, February 1973.

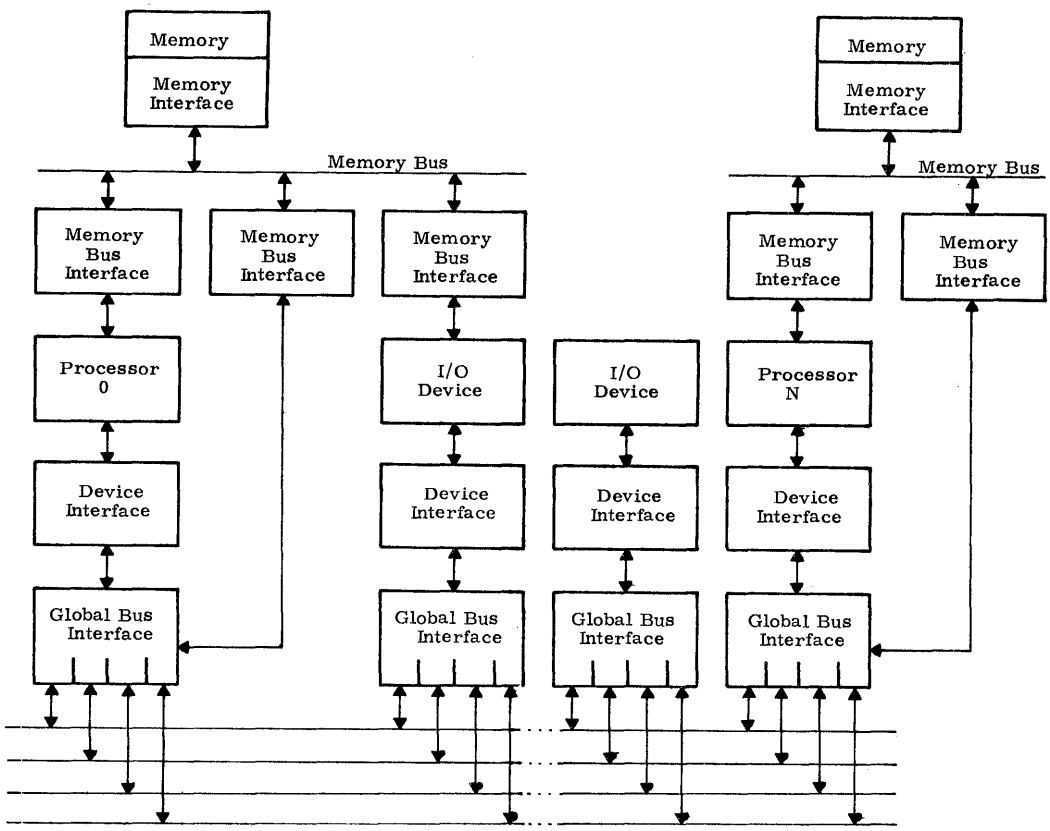


Figure 1. Modular Computer System General Structure

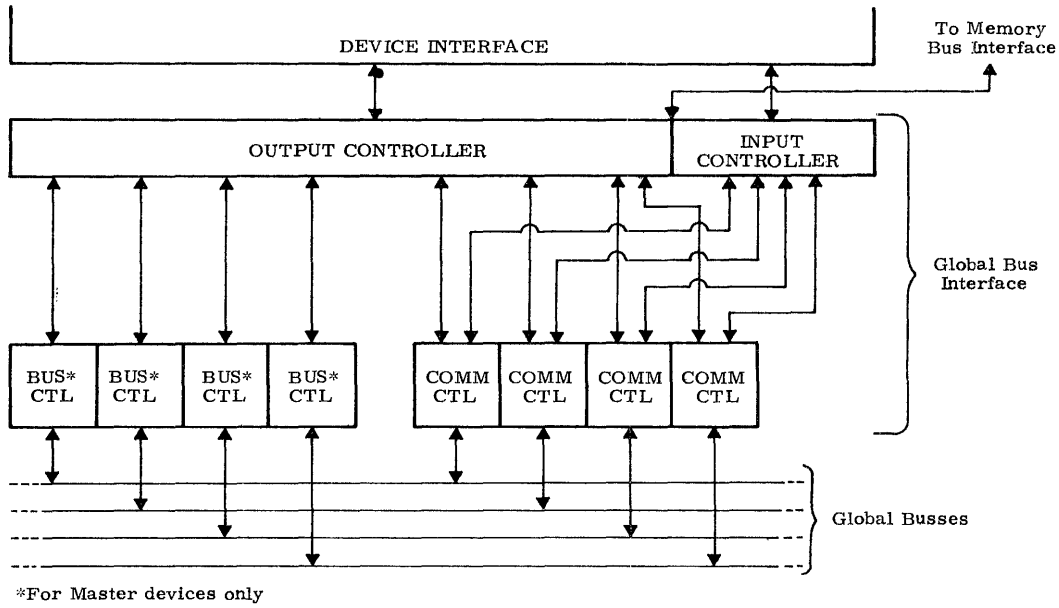


Figure 2. Global Bus Interface Units

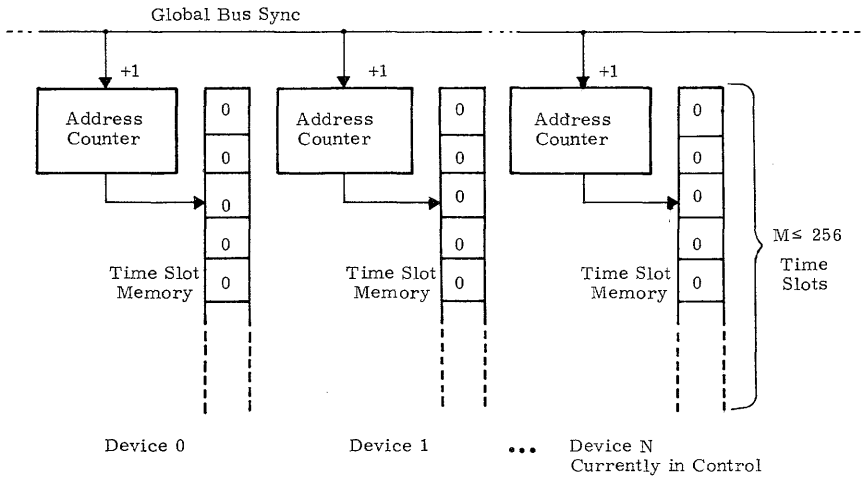
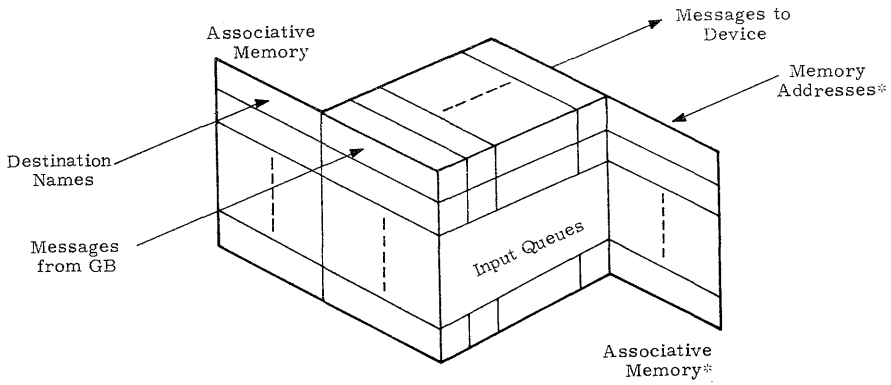
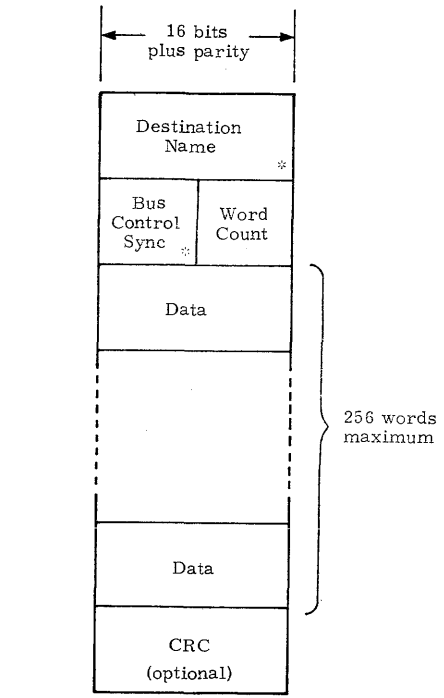


Figure 3. MCS Bus Control Technique



*For processing elements only

Figure 5. Input Controller Queuing Mechanism



*Master devices only

Figure 4. MCS Global Bus Message Format

SWITCHED MULTIPLE INSTRUCTION, MULTIPLE DATA STREAM PROCESSING

C. H. Radoy and G. J. Lipovski
Department of Electrical Engineering
University of Florida
Gainesville, Florida

ABSTRACT

A new architecture is proposed for the effective use of program memory in highly parallel applications. This architecture is particularly suited (but not limited) to being built with standard microprocessors. This architecture utilizes a combination of the state switching idea of the SIMD organization and the multiple data stream idea of MIMD organizations. Through scheduling, the program segments are broadcast at the times required to achieve efficient utilization of the parallel processing array. Relationships are developed to determine, for a given application, if this architecture is more cost effective than comparable SIMD and MIMD organizations.

INTRODUCTION

There is a set of interesting practical problems whose computational requirements demand "non-conventional" computer architectures. Example problems are: radar tracking and discrimination, air traffic control, and weather prediction. Each of these problems has a "real time constraint" which demands that the results of certain computations on data be available within some time interval after the data are presented to the system and a high degree of parallelism, wherein the same algorithm is applied to a large number of separate packages of data. This time constraint and the volume of data inherent in these tasks makes it impossible to satisfy their computational requirements with a single conventional CPU architecture.

For such well defined highly parallel algorithms, architectures have been studied that utilize parallelism, especially in the data stream, to get a suitable effective speed at the lowest cost, which is often strongly affected by the cost of memory. These generally fall into the single instruction stream-multiple data stream (SIMD) or multiple instruction stream-multiple data stream (MIMD) architectures defined by Flynn [3]. (See Figures 1a and 1b, using Bell and Newell's PMS notation [2] to show their structures.) In particular, the STARAN computer, a SIMD architecture utilizing an associative processing element, is widely touted as having significant advantages over conventional architectures for air traffic control [9]. Also, MIMD architectures like the BBN multiprocessor [4] can be used for such problems. However, herein a new, simple architecture, the switched multiple instruction, multiple data stream (SMIMD) architecture is proposed for real time computation that has a high degree of parallelism. See Figure 1c. This new architecture more effectively utilizes parallelism at lower memory cost than the other organizations in many cases.

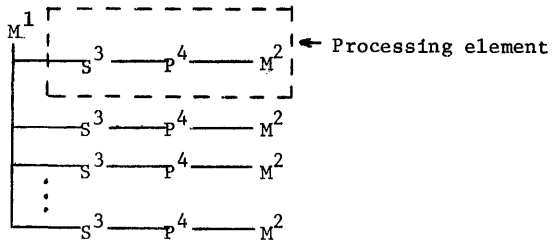
THE SMIMD ORGANIZATION AND OPERATION

The SMIMD architecture is very simple and is particularly suited to being built with standard microprocessors such as the 8080 [10]. It has evolved from such processors as the HAPPE [5] and is strongly related to the MIMD architectures. However, it does not have the properties of SIMD or MIMD architectures as described by Flynn [3]. It can be regarded as a special case of the MIMD architecture which gets around its greatest flaw - memory access conflicts or memory duplication - or as a new architecture midway in cost and speed between the SIMD and MIMD architectures which is more cost-effective than either in certain important cases. In this section, the basic mode of operation is defined. In the next section th interesting question, when to use the SMIMD organization, is addressed.

In the SMIMD organization, program segments are assigned to different program memory modules to be broadcast on different instruction streams. A program segment means here a part of the program that contains no data dependent branches. It may contain loops or subroutines, and so on, but no conditional branches wherein one processing element with some data may branch to a different instruction stream than another processing element with other data. Figure 2 shows an algorithm with three program segments. For an air traffic control algorithm to detect possible collision of pairs of aircraft, segment A might be a quick simple computation, for example to test if two aircraft are in the same sector. It is followed by a data dependent test. If the test fails, say when the two aircraft are not in the same sector, the segment A should be repeated with two new aircraft. If it passes, then a longer, more involved computation is required, say to evaluate the trajectories of the aircraft to see if they might collide. This is shown by two program segments, B and C. For simplicity here we assume that program segments A, B, C execute in the same amount of time.

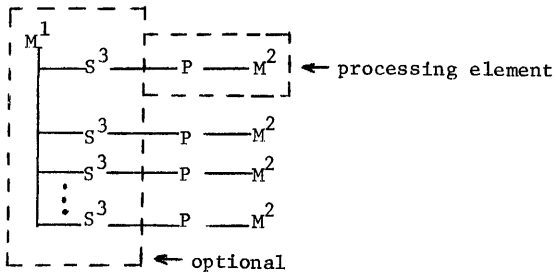
The SMIMD architecture utilizes this flow chart information to avoid memory conflicts while at the same time sharing the program memory among all the processors. Each segment is put in one of the program memories shown at the top of Figure 1c. The outputs of these memories are distributed on busses to all processing elements (PE). All PE's monitoring a given buss will execute, in lock step, the instructions broadcast over that buss. Each package of data is in a small local memory in each PE. The selector switch in the PE chooses one of the memory busses or the local memory according to this rule. When a data word is recalled or memorized, the bottom switch is closed so that local memory is accessed. When an instruction is obtained, it is obtained from one of the memory buss lines. The buss can be selected by a word α in a register of each PE whose output controls the selector. Normally the word α remains constant in a PE for some time so that a PE will execute the instructions broadcast from a given program memory and therefore follow the same program segment. Meanwhile, each memory will broadcast an instruction sequence down its buss so that instructions arrive at the times they are needed by the PE's. (This can be accomplished several ways, such as using a dummy PE to keep track of the program counter and subroutine return addresses, etc.)

When the machine was first started, all PE's would be loaded with data and initialized to monitor program segment A by setting the word α, say, to 0. The data dependent test would be conducted at the end of this segment. In all PE's where this test passes, the word α would be changed say to 1 to allow the PE to monitor program segment B, while the others would not change the word α, so that they repeat segment A. Presumably segment A would input some new data. At the end of segment B, all processors monitoring it would store a new value, say 2, in word α to cause them to monitor segment C, and when segment C is complete, all processors monitoring it would store 0 into α to cause them to return to monitor segment A.



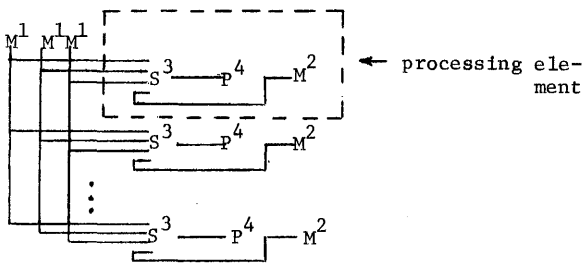
1) Instruction memory
2) Data memory
3) Switch - match bit in associative processor can be used to disable instruction in some cells
4) Processor - usually an associative memory word cell

a) SIMD



1) Backup memory
2) Normal instruction/data memory
3) Switch - arbiter required to resolve conflicts to backup memory

b) MIMD



1) Program memories, each memory holds different part of program
2) Data memory
3) Selector switch: when memorizing or recalling data, switch to local data memory, when fetching instructions, switch to one of the instruction memories
4) Off-the-shelf microcomputer (e.g., 8080)

c) SMIMD

Figure 1. Parallel Organizations

A Flow Chart

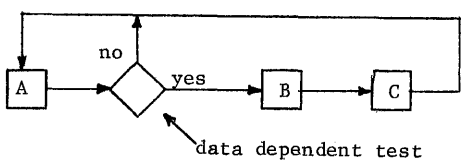


Figure 2

It can be seen in this simple example that all P E's are constantly executing instructions from one or another buss, and many processors are obtaining instructions from the same memory without generating conflicts, while the program is not duplicated in several memories. By comparison, in a MIMD architecture, the program is duplicated in several memories, increasing the cost of the processor, or the program is in one memory which suffers from conflicts, increasing the time to execute the algorithm. In a SIMD architecture, only one instruction stream is broadcast to all PE's, wherein some PE's (perhaps most of them) ignore the instructions that do not pertain to the data in them. Flynn shows that this PE idleness is the key problem in the SIMD architecture [3]. The purpose of the multiple instruction streams in the SMIMD architecture is to alleviate this problem, while requiring the storage of only one copy of the program.

At the outset, the SMIMD architecture is an attractive one. However, contrary to our simple example, program segments are rarely executed in integral multiple time units. This introduces a utilization factor. For example, if segment C were half as long as A and B, it would have to be filled with no-ops, or there would have to be a wait instruction to wake up a PE at the beginning of program segment A after it finishes segment C, so that the PE would synchronize with the other PE's executing segment A. If segment C were executed 10% of the time, the average utilization of the PE's would be 95%. In order to determine the best architecture - SIMD, SMIMD, or MIMD, we must consider the utilization of the processors as well as the relative costs of processors, switch and memory. This analysis is conducted in the next section.

CONDITIONS FOR UTILIZING THE SMIMD ARCHITECTURE

Some relationships are now developed which will be of use in determining which architecture (SIMD, MIMD or SMIMD) is most cost-effective for a given application. A data-parallel application and real time processing constraints that require at least D data entities be processed per unit time is assumed. The architecture that can achieve this required processing rate with the least hardware cost will be the most cost-effective for this application.

In determining the costs of these architectures, the total costs of program memory, execution unit array, and program memory-execution unit interface may vary. The per unit costs of these elements is assumed the same for any organization, which would be the case if the same technology is used in each architecture. But all other total system costs will be assumed to be the same for each architecture (e.g., a given application will require the same amount of data memory, regardless of which architecture is chosen.) Let "a" be the relative cost of one execution unit, "b" be the cost of one program memory-execution unit link, and "c" be the cost of storage of one copy of the program. If it is determined that the real-time processing constraint can be satisfied with a SMIMD architecture of M execution units and N instruction streams, then (assuming linearly increasing cost factors), the relative SMIMD cost is

$$C(SMIMD) = Ma + NMb + c \qquad (1)$$

It must now be determined if some other architecture could have satisfied the real-time constraint with less cost.

As was pointed out in the last section, SIMD and SMIMD execution units are often idle, even when operating under "peak load" conditions. (MIMD architectures are assumed to keep their execution units usefully active at all times.) The greater the execution unit idleness in an architecture, the greater the number of

execution units needed to perform a given amount of work per unit time. Specifically, the architecture utilization "U" is defined as the average fraction of time that its average execution unit is engaged in useful peak load work. Then, U times the number of execution units must be the same for all architectures if they are to have equal peak load processing rates.

Thus, if U is the SMIMD execution unit utilization, and V is the SIMD utilization, a SIMD architecture will require UM/V execution units to achieve the same processing speed as a M unit SMIMD processor. Consequently, an estimate of relative SIMD cost is,

$$C(SIMD) = (UM/V)\ (a + b) + c \qquad (2)$$

Since the peak load utilization of MIMD execution units will be equal to one, only UM execution units will be required in a MIMD architecture. If the MIMD instruction streams are generated by giving each processor its own copy of program memory, a configuration called here a "multicomputer" is obtained. This corresponds, in Figure 1b, to removing the left top memory and its switches. Here we have a relative cost of,

$$C(Multicomputer) = UM(a+b+c) \qquad (3)$$

A "multiprocessor" MIMD architecture in which there is only one common shared copy of program memory would be obtained in Figure 1b, by removing all memories to the right of the execution units. Performance degradation due to program memory access conflicts is introduced. For the purpose of this analysis, it is conservatively assumed that this degradation will not be significant if a mechanism is provided by which each execution unit can obtain one instruction per average instruction execution time. One way to do this would be through modularization and interleaving of the program memory. If one memory module can supply one instruction per execution cycle time, UM memory modules are needed to keep UM execution units busy. (And, of course, each execution unit will have to be able to obtain instructions from any memory module.) The relative cost of this approach is estimated by saying that this increased program memory modularization does not increase the cost (amount) of program memory, but that it does increase the cost of the memory-execution unit interface. In particular, a UM by UM interface is needed and the relative cost is,

$$C(Multiprocessor) = UMa + (UM)^2 b + c \qquad (4)$$

From these four relative cost estimates we can determine which architecture has the least cost for a given application and choice of implementation technologies. Comparing (1) with (2), (3) and (4), we find that the SMIMD cost is least if and only if the following inequalities are satisfied.

$$V < U\ (\tfrac{a}{b} + 1)/(\tfrac{a}{b} + N) \qquad (5)$$

$$M > [\tfrac{a}{b}\ (1 - U) + N]/U^2 \qquad (6)$$

$$\tfrac{c}{b} > [\tfrac{a}{b}\ (1 - U) + N - U]/[U - \tfrac{1}{M}] \qquad (7)$$

Thus, given estimates of U and the cost ratio a/b, we can determine (from 5) how large the SIMD utilization (V) would have to be to make SIMD more cost-effective than an N instruction stream SMIMD structure. Similarly, we can determine (from 6) a bound on the number of execution units (M) below which a multiprocessor structure would be more cost effective than an N stream SMIMD. Finally, with (7) and an estimate of M, we can find a lower bound for the cost ratio c/b. (Since we know b/a, a lower bound on c/b also establishes a lower bound on c/a.)

TABLE I

(Conditions for which SMIMD has least cost)

a/b	N	U	V ≤	M ≥	c/b ≥	c/a ≥
1	4	.8	.32			
2	4	.8	.40			
5	4	.8	.53	8	6.2	1.24
5	8	.7	.32	20	13.5	2.70
5	8	.8	.37	15	11.3	2.26
5	8	.9	.41	11	9.4	1.88
50	8	.7	.61	47	33	0.66
50	8	.8	.70	30	22.4	0.45
50	8	.9	.79	16	14.5	0.29
200	8	.8	.77	75	60	0.30
200	8	.9	.87	35	31.2	0.16

a/b: execution unit - link cost ratio

c/b: program memory - link cost ratio

c/a: program memory - execution unit cost ratio

N: number of SMIMD instruction streams

U: SMIMD execution unit average utilization

V: SIMD execution unit average utilization

Using this approach, the bounds in Table I were calculated. (The lower bound on M was used as the value of M in calculating the lower bound on c/b). The significance of the values in this table is the following. The lowest values of a/b represent extremely simple execution units; units so simple that they would not be capable of the independent operation required in a MIMD architecture. The a/b range 5 to 50 represents projected LSI microprocessor CPU costs, and the value 200 might be representative of a minicomputer CPU or a very fast special purpose execution unit. Thus, this table covers a very wide spectrum of potential implementations of highly parallel architectures. The calculated bounds on U, M, and c/a show that (if SMIMD utilizations in the neighborhood of 0.8 can be achieved) there are wide ranges of these parameters for which SMIMD architecture would be the most cost-effective

Thus, in many applications, the SMIMD architecture will make more effective use of the program memory resource than the MIMD or SIMD architectures, by obtaining good execution unit utilization from only one copy of the program. This effectiveness is a result of the fact that the program memory resources (the program segments) are broadcast on the SMIMD instruction streams at scheduled times chosen to achieve high execution unit utilization. The remainder of this report is concerned with the feasibility of obtaining program segment schedules that will result in efficient system operation.

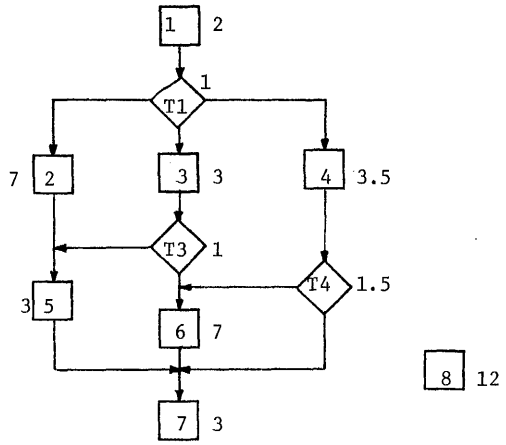
SMIMD INSTRUCTION STREAM SCHEDULING AND EXECUTION UNIT UTILIZATION

In discussing the utilization of execution units in a highly parallel architecture, it is necessary to make a clear distinction betweeen two types of program parallelism. When a program or algorithm can be broken into logically independent and different segments ("tasks"), these tasks can, in principle, be executed in parallel. This task parallelism differs from the situations where it is necessary to perform the same computation on more than one data set. A SIMD structure cannot exploit potential task parallelism; a multiple instruction stream architecture is required if different tasks are to be executed simultaneously.

Since the SMIMD architecture has multiple instruction streams, it can exploit both potential data and task parallelism. To illustrate how program segments can be scheduled on the SMIMD instruction streams to achieve high execution unit utilizations, we will first consider data parallelism, then task parallelism.

Consider the algorithm represented in Figure 3, and assume that this computation is to be performed on D data entities. If we use a conventional computer (or a set of conventional computers operating in parallel) each data entity will cause the computer to execute one of the possible operation sequence paths between segments 1 and 7. Thus, some of these segments (e.g., 5 and 6) are mutually exclusive in the sense that they will not both be performed on the same data entity. In this example, there are five possible paths between segments 1 and 7. If we use a SIMD or a SMIMD architecture, we must assume that for each of these paths there is at least one of the D data entities that will require that the path be executed. Thus, the SIMD and SMIMD computers will have to execute, at least once, each of the seven segments (and associated tests) in this algorithm.

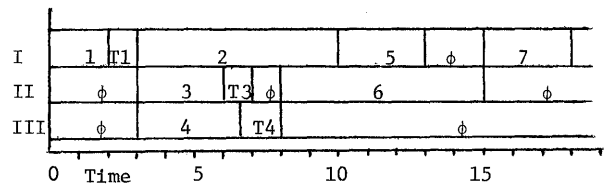
Algorithm Logical Structure



Possible Logic Path	Path Length (time units)
1, T1, 2, 5, 7	16
1, T1, 3, T3, 5, 7	13
1, T1, 3, T3, 6, 7	17
1, T1, 4, T4, 6, 7	18
1, T1, 4, T4, 7	11

Figure 3

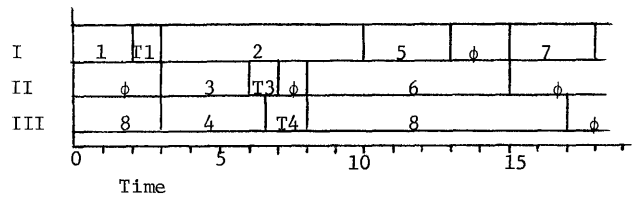
Instruction Streams



(ϕ denotes stream idle)

Figure 4

Instruction Streams



(ϕ denotes stream idle)

Figure 5

The SMIMD architecture allows us to reduce the total time required for one pass through this algorithm by executing some of these segments (the mutually exclusive ones) in parallel. A possible three instruction stream implementation of this program is shown in Figure 4. Comparing Figures 3 and 4, we see how the various conditional execution paths of Figure 3 can be followed by appropriate switching from one instruction stream to another at the end of the conditional test operations. The only apparent complication in using these streams for this example occurs at the end of T3 or T4 when some execution units will be switched to Stream I in order to execute segments 5 and 7 respectively. Clearly we do not want these units to execute the remainder of segment 2 which is being broadcast on Stream I when the switches occur. This can be prevented by setting these units' mode registers so that they will switch streams, but so they will also ignore commands on the new stream until they are "awakened". Also, suitable mechanisms can be provided so that only a specified subset of the idle units monitoring a stream will be awakened at a given point. (e.g., at the beginning of segment 5, we wish to awaken those units that were switched to Stream I by T3, while keeping idle those units that were switched there by T4.)

When microprocessor CPU's are used as SMIMD execution units, stream switching can easily be effected by using the program counter in each unit as the stream-switch control. (Since the CPU's will not be fetching their own instructions, the program counter need not contain the address of the instruction being executed.) By conditionally loading the program counter (i.e., by broadcasting a conditional jump instruction), conditional stream switching can be accomplished. Thus few "extra" instructions must be added to a program to cause appropriate stream switching. (A detailed implementation of this concept using the 8080 microprocessor can be found in reference 7).

To see if the SMIMD architecture is the best architecture for this example application, we need to evaluate expressions 5, 6 and 7. To illustrate this, assume that we are dealing with an implementation for which,

$$a=5,\ b=1,\ c=10,\ M=10.$$

To estimate V (the execution unit utilization in a SIMD architecture) it's assumed that each of the five possible logical paths are equally likely. Broadcasting all seven segments on one instruction stream would require 32 time units. Since the average path length is 15 time units, we see that V is less than .5. Using these estimates, we can see from the third line in Table 1 that if U is approximately 0.8 (or greater) the SMIMD architecture will be the most cost effective way of executing this example data-parallel algorithm.

If D (the number of data entities in this example) is equal to M, all the SMIMD execution units will be devoted to this algorithm. Since, according to the segment schedule in Figure 4, they all will complete the algorithm in 18 time units, we estimate U = 15/18 =.83. Thus, the data parallelism of this problem alone would be enough to maintain a high level of execution unit utilization. However, in general, we cannot expect that the data parallelism of an algorithm will exactly

match the number of execution units in the hardware. If they do not match, some execution units will be idle during the entire time that the data-parallel algorithm is being executed, and consequently the average unit utilization will be less.

In the SMIMD architecture (as in MIMD architectures), we can exploit task parallelism when we have idle execution units. As a simple illustration of this concept, consider segment 8 in Figure 3. This segment is logically independent of the other seven segments, in that it may be executed before, during, or after them. Figure 5 illustrates that, by using the idle time on Stream III in Figure 4, the 12 time-unit segment 8, can be broadcast within the 18 time units required by the data parallel algorithm. This concurrent execution of an independent segment is particularly important if that segment can utilize only one execution unit. In that event, if the segment were wholly broadcast either before or after the data-parallel algorithm, for 12 time-units only one of the M execution units would be active, and the average execution unit utilization would be substantially reduced. Thus, this simple example shows that task parallelism could and should be exploited to maintain high SMIMD execution unit utilization.

At this point, it is appropriate to briefly discuss how the sequence of operations illustrated in Figure 5 could be obtained. The generation of this operation sequence could consist of three steps: "program analysis", "segment allocation", and schedule execution. Program analysis and segment allocation would be performed at compile time. Program analysis obtains the algorithm's logical structure such as shown in Figure 3. The location of all data dependent branches and the corresponding possible logical flow paths is a simple procedure. Further, fairly accurate compile time estimates of relative segment execution times is straight-forward since, by definition, a segment contains no data-dependent branches.

Segment allocation assigns to each control unit (and hence to each instruction stream) a set of segments and the order in which they are to be executed. (Since segment allocation "binds" a segment to a stream, it can be followed by a code generator that will cause execution units to switch to specific streams when data dependent branches are executed.) Allocating segments to control units so that an optimum schedule, such as Figure 5, is obtained is not a trivial task. In fact, there are no efficient procedures that guarantee optimality. However, there are fairly simple scheduling heuristics which have been shown to be "nearly optimal" when used to schedule multiprocessors [8]. Similar efficient heuristics can be developed for the SMIMD architecture [7].

Once the segments have been allocated, schedule execution can take place with very little run-time overhead. Simple hardware mechanisms can be built into the control units to cause one unit to be idle until some other unit issues a special instruction. (e.g., control unit II would not begin broadcasting segment 3 until unit I had completed T1.) A similar mechanism could also cause a unit to interrupt the broadcasting of one segment and begin broadcasting another. (e.g., control unit III switches from segment 8 to segment 4 when unit I completes T1). (This preemption of segment 8 would not require significant run-time overhead as long as both segments 4 and 8 could fit into unit III's memory module. In that event, when the preemption occurred, the control unit would merely have to save the program counter contents corresponding to the preempted segment. All other process-state information, such as general register contents, would not be affected by the preemption; when preemption occurred, the execution units that were executing the preempted segment would be made idle until segment 8 was resumed. These units would not execute segment 4.) Thus, schedule execution can be effected without the overhead of a run-time resource scheduler or dispatcher. The possibility of a compile-time scheduler using preemptive scheduling techniques, increases the possible effective use of the SMIMD instruction streams [6]. Consequently, we are confident that efficient procedures can be developed which will enable us to generate operation sequences (for large complex algorithms) which will result in a high level of SMIMD execution unit utilization [7].

CONCLUSIONS

A new but simple architecture, SMIMD, is shown to more effectively utilize memory in highly parallel algorithms than the SIMD or MIMD organizations through the scheduling of program memory resources. This architecture utilizes a combination of the state switching idea of the SIMD organization and the multiple data stream idea of MIMD organizations. In the SMIMD architecture, only one copy of the program need be stored in random access memory. For given per-unit costs of memory, processor and switch-link inter-connection, and a known rate of processing, it can be determined whether the SMIMD organization is most cost-effective. This architecture can effectively utilize parallel processing arrays of minicomputer CPU's, microcomputer CPU's or simple comparator-based execution units. Compile-time scheduling techniques permit efficient execution of algorithms with very little run-time overhead.

REFERENCES

1. Barnes, G. et al., "The Illiac IV Computer," IEEE Trans. on Computers, Vol. C-17, Aug. 1968, pp. 746-757.

2. Bell and Newell, Computer Structures: Readings and Examples, McGraw-Hill, 1971.

3. Flynn, M., "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Computers, Vol. C-21, Sep. 1972, pp. 948-960.

4. Heart, F.E. et al., "A New Minicomputer/Multiprocessor for the ARPA Network," Proc. National Computer Conference 1973, pp. 529-537.

5. Marvel, O.E., "HAPPE - Honeywell Associative Parallel Processing Ensemble", Proc. Symp. on Computer Architecture, University of Florida, Dec. 1973, pp. 261-267.

6. Muntz and Coffman, "Optimal Preemptive Scheduling on Two-Processor Systems", IEEE Trans. on Computers, Vol. C-18, Nov. 1969, pp. 1014-1020.

7. Radoy, C.H., "Switched Multiple Instruction Stream Computer Architectures", Ph.D. Dissertation, University of Florida, Gainesville, 1975.

8. Ramamoorthy, C.V. et al., "Optimal Scheduling Strategies in a Multiprocessor System," IEEE Trans. on Computers, Vol. C-21, February 1972, pp. 137-146.

9. Rudolph, J.A., "A Production Implementation of an Associative Array Processor--STARAN," Proc. AFIPS FJCC 1972, pp. 229-241.

10. 8080 - Single Chip Eight-Bit Parallel Central Processor Unit, Intel Corporation, April 1974.

SEQUENTIALLY ENCODED DATA STRUCTURES THAT SUPPORT BIDIRECTIONAL SCANNING

ROBERT J. LECHNER
Honeywell Information Systems Incorporated
Billerica, Massachusetts

Introduction

This paper is concerned with the problem of identifying the type and/or length of unpredictable sequences of character-string data, or fields, in serial storage. A fundamental requirement is that the data sequence must be scannable in either direction.

There are two standard methods of encoding such sequences. One way is to reserve one or more "field separator" characters, which are not legal data characters.[1] This is inefficient since one or more intervening symbols must also be scanned to find the next separator in the sequence. Another is to store a separate index table with a field length or relative pointer entry for each data field in the sequence. This complicates I/O buffering since both index and value string queues must be maintained.

When each field has an imbedded binary symbol at one edge denoting its actual length, then it is possible to skip across fields merely by inspecting this length symbol. Unfortunately, this permits scanning only in one direction, unless the length symbol is imbedded at both edges of each field.

Section 1 describes a novel method of encoding field length or type data into one-byte separators that avoids the limitations of conventional methods. A "moving window" on the serial data buffer contains byte string data with imbedded field separators that require no reserved characters, yet permit scanning or skipping over fields in either the forward or backward direction. These field separators are called symmetric differences because they are a symmetric function of the two immediately adjacent data field types.

An implicit assumption of the first application was that field length was a fixed function of field type; once the type code was identified, field length could be determined by table lookup. Section 2 describes another major class of applications, in which the field type is either invarient or irrelevant, in which case the field separators can be used to determine field length directly. This class includes simple forms of text editing. Combinations of type and length variability can also be handled, by extending the length of the separator symbol to two or more bytes.

The symmetric difference method determines field position by accumulating incremental length data; consequently it is vulnerable to the so-called error extension problem, which means that an error in one field position can cause misalignment of many other field boundaries. A unique advantage of symmetric difference coding is an error detection and correction ability which applies only to the field separators. This insures correct field type identification and boundary alignment independent of errors in the field values themselves. Section 3 describes this feature, and also defines a null or zero-length field type which permits a long variable-length sequence to be split into fixed-length physical data blocks.

The problem of inserting or deleting fields is not specifically addressed by this paper. This problem is no more difficult with symmetric difference coding than with any other method of encoding data for contiguous storage. However, the introduction of the null field type described in Section 3 does permit local deletion of a field without immediate repacking (i.e., "garbage collection" can be deferred). Prior allocation of extra space by inserting a fixed percentage of null fields permits new field insertion with a minimum of data relocation.

More complex encoding and scanning algorithms are required when tree-like data structures are represented in one dimensional storage. Section 4 defines efficient packed sequential representations for tree-structured data. The subordinate data structures attached to each node are separated by a reserved field type called an "internode separator." Its non-null field value contains data related to the length (and possibly the type) of the adjacent subtrees; these can be entered or skipped in the forward or backward direction. Symmetric difference coding may also be used to encode the values of internode separator fields. Tree coding and scanning algorithms are illustrated by an example in Figures 4 and 5.

The space-saving advantage of symmetric difference coding is only applicable to data within a physically contiguous storage unit; in linked storage, forward and backward chain pointers cannot be merged at a single physical location. However, it is upward-compatible with standard methods by which noncontiguous physical data blocks are listed in index tables or logically chained together.

Representation of Variable Data Field Sequences

Sequential data entry and recording on serial media, such as tape cassettes, is conveniently accomplished by using a double-ended FIFO buffer or "deque" with top and bottom end markers, keyboard input (load) and tape output (write) pointers.[2] Logically, the buffer is cyclic, with no attention paid to the physical top and bottom end addresses; these ends are implicitly joined together by address arithmetic modulo the buffer size. In the most general case, tape block reading or writing (physical I/O) from one sector of the buffer can go on concurrently with data field insertion or extraction from another buffer sector (logical I/O).

The problem considered in this section is that of storing a sequence of data fields of varying type. We assume that all occurrences of one field type have the same length in bytes. Section 2 will consider the more general case when field lengths cannot be predicted from field types.

When the following two application requirements occur together, the FIFO buffer organization described above runs into difficulty:

(1) Optional or Varying Occurrences: Some field types may have optional or varying occurrences as attributes; in either case, the next field type is sometimes unpredictable, so that field type identifiers (names) must be recorded occasionally along with the data.

(2) Bidirectional Scanning: When backward as well as forward scanning of the data file is required, field by field, the queue becomes double-ended on the tape or physical I/O side.

Requirement (1) is motivated by a desire for flexible source data formating or even free-form (self-describing) data entry. Requirement (2) is implied by functions such as editing or on-line correction which make use of a "backspace one field" operator or function key, or a bidirectional scrolling facility using a CRT display.

Existing Approaches. One common approach to the file access problem is a cyclic FIFO buffer, in which a single pointer defines the current data entry point. Thus, field names and/or lengths must be stored adjacent to their value strings (see Figure 1). To scan the buffer from left to right, the next field's starting address is computed by adding the preceding field's length and starting address. Unfortunately we cannot get started on a right-to-left scan, because the value field is encountered first; we cannot locate its name without knowing its length, and vice versa.

Another approach avoids this last restriction (and permits random access to fields) by segregating field names from their values; the names (assumed fixed length) are kept in a separate ordered list, which can be indexed or scanned from either end. Before writing a tape block, the sequence of field names is appended to the value string sequence. Figure 2 shows this solution, with names stored in reverse order starting at the end of the data block. Names and values can be accumulated simultaneously and both share a common pool of unused byte-cells in the middle of the data block. This complicates physical concatenation or logical chaining.

Recommended Solution. An obvious way to overcome the one-directional scanning limitation of the format in Figure 1 is to imbed a field name before and after its value string (see Figure 3A). However, this doubles the space needed to store field names. The recommended approach is a simple extension of this format which avoids duplicate storage of field names, as illustrated in Figure 3B.

Figure 3A shows each field value with a name (one-byte symbol A, B, etc.) at both ends. In Figure 3B except for the initial and final field name, the two symbols separating each pair of field values have been combined into one symbol, the "symmetric difference" between the two field names it replaces.

The symmetric difference (or Boolean difference) ΔAB is defined as the carry-free exclusive-or (bit-wise modulo-two) sum of the binary codes for A and B: $\Delta AB = A \oplus B$. When A and B are k-bit symbols, the signed algebraic difference $\Delta AB = (B-A)$ or the ring sum $A+B \pmod{2^k}$ would work equally well, but the symmetric difference is faster on machines with an exclusive -or instruction.

There are two alternate interpretations for the codes A and B on which the symmetric difference is based. One interpretation, shown in Figure 3B, is an indirect or variable-field-type interpretation: A and B are field type codes; the length of each occurrence of a field type is assumed to be fixed and stored in a separate table indexed by the field type code. The second or variable-field-length interpretation, in which the type codes are identified directly with field length, will be discussed in the next section.

For example, suppose we have just begun a left to right scan in Figure 3B. Given the identity of the first field (A) we wish to identify the (unknown) second field as type B. We add the difference ΔAB to the first field name A, after locating ΔAB by looking up field A's length in a table. We locate ΔBC relative to ΔAB via the length of B, and then compute $C = B \oplus \Delta BC$ to identify the third field. Since $B = C \oplus \Delta BC$ and $A = B \oplus \Delta AB$, etc. this procedure works equally as well when beginning at the right side and scanning left, if the identity of the right-most field is known upon block entry.

Other Applications

The above solution assumed a relatively small number of field types, each of known constant length, repeated often enough to justify a stored table of field lengths. Other applications will now be considered.

Text Editing. For interactive text editing, a bidirectional scan has obvious advantages. A minor change to the semantics of the preceding scanning algorithm permits its application to a contiguous sequence of words or other "atomic" elements of text. Suppose we do not need to distinguish fields by "type" but regard them all as words, whose only distinguishable attribute is their length. Assume a maximum word length (e.g., 255 bytes) and let the field type code directly represent word length. (The reserved null field type of length zero is consistent with this convention.) Symmetric difference separators can be computed as before, with one additional advantage: the table which formerly defined the field length (when indexed by field type) now becomes the identity map and is no longer needed.

Introducing symmetric difference codes as field separators need not involve any

expansion of text volume. For example, suppose an "atomic" element of text is defined to be any contiguous string of non-space characters between two space characters. Without loss of generality, every pair of contiguous space characters may be regarded as bracketing a null or zero length atomic element. Then, every sequence of space characters may be replaced by an equal-length sequence of field separators, coded as symmetric differences.

For example, let the values of two fields A and B be "the" and "word", respectively. Adopt the convention than an underlined digit represents the 8-bit binary code for itself; e.g., 7 denotes the bitstring '00000111'. Then the type codes for fields A and B may be identified with their lengths and become 3 and 4 respectively. The symmetric difference $\Delta AB = 3 \oplus 4 = 7$. The text strings below are encoded without changing their lengths in bytes ("#" represents one space):

(a) "the#word" -------- "the7word"

(b) "the##word" ------- "the34word"

(c) "the##...##word" -- "the30...04word"

In (a), ΔAB replaces one space character; in (b), ΔAO ΔOB replaces two spaces; in (c), ΔAO ΔOO ... ΔOO ΔOB replaces a corresponding number of spaces.

Punctuation. One disadvantage of the above length-preserving text-encoding method is that it does not separate punctuation marks from alphanumeric strings within atomic elements of text. However, a punctuation symbol which is immediately preceded and/or followed by an alphanumeric string may be encoded as a separate one-byte field by artificially inserting a separator before and/or after the punctuation symbol. This expands the text by one or two extra bytes per punctuation symbol. Upon decoding, these separators should be replaced by null fields rather than space characters.* (In Section 4, we will adopt the convention that a null field exists between any two adjacent separators.)

Dictionary Look-up. All of the above text editing considerations apply equally well to dictionary (e.g., symbol table) look up. To expedite searches, a very long string of lexicographically ordered words can be partitioned into blocks, for which an index table can also be prepared. Index table entries point to block end points which contain field lengths, rather than symmetric differences, as in Figure 3B. This permits bidirectional scanning from multiple entry points.

*For full generality, in this case symmetric difference separators should include a one-bit tag to indicate whether the separator was artificially inserted or not. This bit does not have the even parity property described in the next section, but neither is it implicated in the error-extension problem mentioned there.

Variable-Length Fields with a Fixed Sequence of Types. Both applications above may be regarded as the special case (n=1) of a data stream with multiple field types in which the type sequence is predictable but field lengths are variable and must be encoded into symmetric differences. For example, card-formatted data with exactly one instance of each field type could use symmetric difference coding to suppress leading zeros and/or trailing blanks.

Fields of Unpredictable Type and Length. When neither field type nor length can be inferred from the other then symmetric difference coding of both length and type is possible. In general, this will involve more than 8 bits per separator, so a two-byte separator should be considered.

Error Protection

All encoding methods which build up a field address by incremental addition of preceding field lengths are vulnerable to a single erroneous field length indication. For example, suppose that the sample encoding of "the##word" in the preceding section contained a four-bit error pattern which changed the separator sequence (3,4) = 00000011, 00000100 into (15,8). If this error pattern is not detected, it will cause a left to right scan to erroneously interpret the null field length as $3 \Delta 15 = 12$ rather than 0. The scanner will jump 12 bytes to the right looking for the next separation. In general, this will force all succeeding field boundaries out of alignment and manual interpretation may be necessary to recover their data. This section describes a single-error-detection method using a one-way scan; correction requires a two-way scan.

Error extension can be avoided completely by recording a separate table of relative addresses or offsets to individual fields, as in Figure 2, but with other disadvantages discussed earlier. Another way is to take advantage of the unique parity checking feature of symmetric difference coding. This provides error detection and correction advantages equivalent to the use of two redundant pointer chains as shown below.

Padding. One requirement for effective error control is the ability to partition a sequence of variable-length fields into blocks of fixed length. In general, field boundaries will not coincide with block boundaries. To avoid splitting a field into two parts (which introduces a new coding problem) and to permit resynchronization of the field address pointer at inter-block boundaries, a block must be padded preferably by some method which does not complicate the logic of the scanning algorithm.

Padding can be accomplished by reserving a particular 8-bit symbol to indicate a "null" field type defined as one whose value string has zero length. To be consistent with those applications in Section 4 which replace a field's type code by its value string length in bytes, the zero 8-tuple is reserved as a null field type code.

Null field values take up no space, but their separators do. The scanning algorithm still works, and a small amount of logic to recognize and skip over nullfield separators will make them transparent to the scan algorithm.

Two non-null fields (say, field types A and B) with value strings denoted V_A, V_B and separator denoted $\underline{\Delta AB}$ appear as follows in Figure 3B:

$$V_A \ \underline{\Delta AB} \ V_B$$

To insert a null field (type $\emptyset$) between A and B, we merely replace $\underline{\Delta AB}$ by the two separators $\underline{\Delta A\emptyset}$, $\underline{\Delta\emptyset B}$; (this inserts one extra byte into the field stream). Two null fields between A and B would appear as $V_A \ \underline{\Delta A\emptyset} \ \underline{\Delta\emptyset\emptyset} \ \underline{\Delta\emptyset B} \ V_B$ (Each underlined triple represents a one-byte field separator.) Each additional null field introduces another $\underline{\Delta\emptyset\emptyset}$ separator, whose value is $\emptyset \oplus \emptyset = \emptyset$.

Self-identification. In order to begin a field scan at either block boundary it is necessary to know the identify of the first field to be scanned. For this purpose it is sufficient to adopt the convention that at least one null field will always be inserted at every block boundary.

Suppose a block boundary occurs between fields A and B of the preceding example. Within the field stream $V_A\underline{\Delta AB}V_B$ the separator $\underline{\Delta AB}$ must be assigned either to the block containing V_A or to the one containing V_B. Neither choice is satisfactory. For example, if $\underline{\Delta AB}$ is stored with V_B, then a backward scan into the block containing V_A requires prior reference to succeded blocks before field type A (and its length) can be identified.

This problem disappears if two null field separators straddle the block boundary. For example, with a boundary between $\underline{\Delta A\emptyset}$ and $\underline{\Delta\emptyset B}$ in the field sequence $V_A \ \underline{\Delta A\emptyset} \ \underline{\Delta\emptyset B} \ V_B$, the scanning algorithm will encounter a separator ($\underline{\Delta A\emptyset}$ or $\underline{\Delta\emptyset B}$) no matter which direction it begins to scan. Because zero represents the null field type, $\underline{\Delta A\emptyset} = A \oplus \emptyset = A$ and $\underline{\Delta\emptyset B} = B \oplus \emptyset = B$.

In other words, null fields have the nice property that adjacent separators directly identify the adjacent field type. In other words, the two fields at the edges of any block are self-identifying , if blocks are padded to avoid splitting fields and at least one null field is inserted at block boundaries.

Resynchronization. Null field insertion at block boundaries makes each block independent of adjacent blocks, as far as the field scan process is concerned. However, a correct field scan from one end of a block to the other still requires all intervening field separators to have correct values. What if an error occurs?

Consider first the error detection problem. If null fields are "inserted" at both block boundaries, a correct sequence of field separators will automatically have a zero-valued longitudinal parity check sum. For example, suppose a block contains four fields; A, B, B, C; the sequence of field separators and interspersed value strings will be

$$\underline{\Delta\emptyset A} \ V_A \ \underline{\Delta AB} \ V_B \ \underline{\Delta BB} \ V_B \ \underline{\Delta BC} \ V_C \ \underline{\Delta C\emptyset}$$

by definition, $\Delta\emptyset A = A$, $\Delta AB = A \oplus B$, etc; the parity check sum of these five separators is

$$\underline{\Delta\emptyset A} \oplus \underline{\Delta AB} \oplus \underline{\Delta BB} \oplus \underline{\Delta BC} \oplus \underline{\Delta C\emptyset} =$$

$$A \oplus (A \oplus B) \oplus (B \oplus B) \oplus (B \oplus C) \oplus C = \emptyset$$

That is, field type or length codes appear in pairs, causing pairwise cancellation in the overall checksum. In conclusion, this zero checksum property permits rapid detection of separator errors. It also discriminates between separator and value string errors if an overall error detection coding scheme is also used. Often, block lengths can be restricted to make the probability of more than one separator error per block negligible.

Assuming a single separator is in error, how can its location be established without ambiguity? A nonzero check sum over the correct location of an erroneous field separator sequence would yield the error pattern but not its location. However, this actual checksum is not computable because the computed sequence of separator addresses may diverge from the correct locations beyond the point of error. The solution is to begin checking from the opposite end as well.

It is easily verified that whenever the two sequences of separator addresses (starting from opposite ends of the block) have only one address in common, then their point of contact is the erroneous separator location. In this case, the correct separator value is the symmetric difference of the field type codes computed by the two scans just prior to their point of contact, since this gives an overall check sum of zero.

Furthermore, if the two address sequences have multiple contact points, then each contact point must be considered as a possible error location. This provides double error detection, although not all double byte errors are necessarily detected in this way. For example, if field type A and B have the same length, a double separator byte error pattern that interchanges $\underline{\Delta AB}$ with $\underline{\Delta AA} = \underline{\Delta BB} = \emptyset$ would recognize the field sequence $V_A \ \underline{\Delta AB} \ V_B \ \underline{\Delta BB} \ V_B$ as $V_A \ \underline{\Delta AA} \ V_A \ \underline{\Delta AB} \ V_B$.

Such double errors are not detectable by the symmetric difference coding (Note that they do not imply loss of correct pointer alignment).

Representation of Tree-Structured Data

The symmetric difference approach will now be extended to irredundant packed sequential representation of hierachial or tree-like data structures. This permits bidirectional scanning at any level of the tree, without the necessity of scanning intervening data.

As an example, consider the tree structure of Figure 4 in which A, B, etc. represent field types whose values V_A V_B, etc., are to be stored sequentially in the order (ABCDEFG). Nonterminal nodes R (for root) and T_k for subtree may have data fields A...G (leaves of the tree) or other subtrees attached to them. Branches leading to each subtree are surrounded by matched pairs of parenthesis. Although not shown on Fig. 4, the root node R also has an implicit pair of () brackets, which delimit (enable the scanner to begin at either end of) the entire tree. The corresponding parenthesized linear representation of the tree may be reconstructed from Fig. 4 by reading off all branch and leaf labels while traversing the tree counter-clockwise: A (B(CD))(E(FG)).

The value string corresponding to this representation is $V_A(V_B(V_CV_D))(V_EV_FV_G))$. This string is unambiguous as long as "(" and ")" are reserved characters that do not appear in V_A, ..., V_G, or if V_A through V_G are of known lengths. To avoid reserving the parenthesis characters, suppose we consider them as one-byte values of a special "punctuation" field type, σ. In our example, Vσ is either")" or "(" and must be imbedded within field separators just like any other field. Applying symmetric difference coding to this field sequence gives the following (inefficient) result, requiring two bytes per parenthesis character:

$\underline{\Delta\emptyset A}\ V_A\ \underline{\Delta A\sigma}\ V_\sigma\ \underline{\Delta\sigma B}\ V_B\ \underline{\Delta B\sigma}\ V_\sigma\ \underline{\Delta\sigma C}\ V_C$
$\underline{\Delta CD}\ V_D\ \underline{\Delta D\sigma}\ V_\sigma$

$\underline{\Delta\sigma\sigma}\ V_\sigma\ \underline{\Delta\sigma\sigma}\ V_\sigma\ \underline{\Delta\sigma E}\ V_E\ \underline{\Delta E\sigma}\ V_\sigma\ \underline{\Delta\sigma F}\ V_F\ \underline{\Delta FG}$
$V_G\ \underline{\Delta G\sigma}\ V_\sigma\ \underline{\Delta\sigma\sigma}\ V_J\ \underline{\Delta\sigma\emptyset}$

The ordinary fields Vσ containing reserved meta-bracket values "(" and ")" may be replaced by two distinct reserved field types, whose values may be null. For example, using ↓and ↑ to denote type codes for metabrackets of type "(" and ")" respectively, and ΔAB to represent the symmetric difference of type codes A and B, the tree of Figure 4 is representable as

$\underline{\Delta\emptyset A}\ V_A\ \underline{\Delta A\downarrow}\ \underline{\Delta\downarrow B}\ V_B\ \underline{\Delta BC}\ V_C\ \underline{\Delta CD}\ V_D\ \underline{\Delta D\uparrow}\ \underline{\Delta\uparrow\uparrow}$
$\underline{\Delta\uparrow\downarrow}\ \underline{\Delta\downarrow E}$
$V_E\ \underline{\Delta E\downarrow}\ \underline{\Delta\downarrow F}\ V_F\ \underline{\Delta FG}\ V_G\ \Delta G\uparrow\ \underline{\Delta\uparrow\uparrow}\ \underline{\Delta\uparrow\emptyset}$

This structure is efficient in storage because only one additional byte is needed for each imbedded open or close parenthesis symbol. However, every field must still be scanned to traverse the tree, and this is not efficient for many applications. For example, suppose A represents an "if" condition, B(CD) represents a "then" clause and E(FG) an "else" clause of a parsed source language statement to be interpreted. After run-time evaluation of condition A the interpreter would like to either scan B (CD) but skip E(FG), or skip B(CD) and then scan E(FG).

What is missing from the preceding tree representations? All of them are inefficient in the sense that a scan in either direction must still traverse every node and every leave of the tree to get from the one end to the other.

If the tree has many nested levels, and if we are only interested in selecting one higher-level node, then much time will be wasted in such a scan. What we need is a "subtree" or "internode" separator, which conveys information about the lengths of the complete substructures attached to its adjacent nodes. The rest of this section describes a tree structure representation which does permit skipping over subtrees, by reserving a single field type called an "internode separator" and using symmetric difference techniques to encode its contents.

Extension for Rapid Scanning. The preceding encoded representation of a tree structure is not efficient when a known path through the tree must be located. For example on Fig. 4, suppose field F is known in advance to be attached to the first leaf of the second branch of the third branch attached to the root node. The extension proposed below will permit the scanner to skip the first two branches attached to the root node, then descend one level and skip field E, arriving at field F in three jumps rather than the five jumps required to scan over A, B, C, D, and E.

A new reserved field type code (denoted σ) called an internode separator field, will be needed. Its non-null value depends only on the lengths of the subtrees which it borders or separates, and will be used to jump over a subtree in either direction. Symmetric difference coding will be used on internode separator field values, to minimize storage requirements. Field of type σ will be imbedded in the field stream just like data and null field types. While a field-sequential scan merely recognizes and skips over internode separators (as with padding fields), a tree scanning algorithm must contain logic to recognize them and use them appropriately.

A fixed length is assumed for internode separators; e.g. a length of 16 bits would limit the maximum subtree length to 65K bytes; if this is inadequate, a larger separator could be used. A stack will be used to save placemarkers (address pointers or offsets) to the beginning and end of each nested subtree whose content is being scanned (or one end and its length). The stack also permits a direct return from any subtree to its parent node without going through all other branches of this subtree. The stack is initialized to delimit both ends of the entire tree.

It is simple to construct the proposed encoded form of a tree by a sequential scan of its parenthesized field structure. Each closing parenthesis, and each opening parenthesis that does not have a closing parenthesis as its immediate predecessor, is replaced by an internode separator field with a 2-byte value. (A pair of brackets of the form ")(" is combined into one internode separator field.)

Within each internode separator field corresponding to a single open or close bracket will be placed the distance (in bytes) to the opposite matching bracket. Within the internode separator corresponding to a pair of "Close, open" or "), (" brackets is placed the symmetric difference of the distances to their matching brackets. (This construction is consistent with the previous encoding of field-lengths into field separators.)

Each pair of matching brackets plus all of its enclosed fields and nested bracket pairs corresponds to one subtree and the branch connecting to its parent node. For example, the tree of Fig. 4 contains 7 leaves and 4 subtrees. Its root node is labeled R and the root nodes of its subtrees are labeled T_1, T_2, T_3 and T_4.

The encoded representation of a subtree will be called a compound element, and represents a new byte-string-valued data type. That is, a compound element is any sequence of fields including internode separator fields, in which the latter obey certain constraints on their pairwise cocurrences and contain appropriate length-defining values.

Compund elements may be nested. Note that the internode separators corresponding to open or close type brackets are only distinguishable by tracking their positions relative to parent nodes; a separator position and value determines the location of its matching separator; the intervening internode separators specify inner structure at lower levels of the tree.

In figure 4, four compund elements are identified by parentheses around the branches leading to the root nodes of their corresponding subtrees. Only one pair of subtrees (T_1 and T_2) are adjacent to each other at the same tree level. The other two subtrees (T_3 and T_4) are isolated by fields or higher level brackets. Let V_A denote a value string for a field of type A. Let L_j denote the value of a σ-type field (instead of V_σ). Define L_j = length in bytes of (the encoded representation of) subtree T_j in Figure 4, and define $L_{ij} = L_i \oplus L_j$, the symmetric difference of L_i and L_j. Then the tree structure of Figure 4 requires the following sequence of field types to be encoded:

$$A\sigma B\sigma CD\sigma\sigma E\sigma F\sigma G\sigma\sigma$$

Substituting V_A for A etc., and L_j or L_{ij} for σ we obtain the field value sequence:

$$V_A L_1 V_B L_3 V_C V_D L_3 L_{12} V_E L_4 V_F V_G L_4 L_2$$

Appropriate field separators must now be inserted to punctuate this sequence of field values. Let ΔAB represent the single-byte field separator between two value strings or fields of type A and B. Using this notation, the fully encoded linear representation for the tree of Figure 4 is shown in Figure 5.

Figure 5 also illustrates the rules for computing compound element lengths. For each field which is a direct descendent of the compound element whose length is being computed, add the field's value string length plus one (separator) byte. For each nested compound element which is a direct descendent add its length plus three bytes (for a σ-type field and its separator). Finally, add three bytes for the σ-type field prefix to the compound elment itself. This is illustrated on Figure 5 for the subtree T_2 = (E(FG)); the lengths of V_E, V_F and V_G are denoted by x, y and z, respectively.

To illustrate the scanning process, the example of Figures 4 and 5 will be used. Suppose we wish to access the field F which is known a priori to be on the first branch of subtree T_4. T_4 is on the second branch of subtree T_2, which is on the third branch at the top level of the tree.

Our search for field F proceeds from the left edge of the encoded representation (Fig. 5) and follows the dotted lines:

(1) Read ΔØA, lookup the length of V_A and skip to ΔAσ.

(2) Read ΔAσ, advance and read L_1, and skip to L_{12}.

(3) Read L_{12}, compute $L_2 = L_1 \oplus L_{12}$, then stack the addresses of L_{12} and L_2. This will allow us to return from subtree T_4 to either the left or the right edge of the subtree T_2.

(4) Advance to ΔσE, lookup the length of V_E, and skip to ΔEσ. Advance and read L_4 and stack both of its addresses.

(5) Advance to ΔσF, which identifies F as the next field. This program skips over three subtrees and descends two levels down into the tree structure. By unstacking the return address we can return immediately up any branch to the next higher level of the tree. The stacked address of the current subtree's left or right boundary is used to resume the scan in the forward or reverse direction, as desired. Internode separators within a subtree must be located interior to these boundaries.

During the course of the scan, ambiguities will arise. The internode separator content does not specify whether it is an end of the current subtree, or an internode separator within it. This ambiguity may be resolved by comparing the separator location to the end of the current subtree, which is one of the two subtree boundary locations on the top of the stack. Another way to resolve this ambiguity is to assign three distinct type codes σ_1, σ_2, and σ_3 (first, last, and intermediate) internode separator field types, corresponding to parenthesis symbols "(", ")", and ")(" respectively.

Conclusion

A coding method that supports bi-directional scanning has been defined for fields of variable length and/or type. Parity checking techniques have been applied to resolve boundary alignment problems when a separator is in error. An extension of the concept to hierarchial parenthesized structures permits skipping over subtrees without scanning their contents. The method is economical in space and time and applies with minor differences to text editing or formatted data storage.

References

1. E. H. Heitz: "The Interpretation of Structured Stored Data Using Delimiters." Proc. 1970 ACM SICFIDET Workshop on Data Description and Access, November 1970, (pp 188-200)

2. D. E. Knuth: The Art of Computer Programming Vol. I (Fundamental Algorithms), Addison-Wesley, Reading, Massachusetts 1968

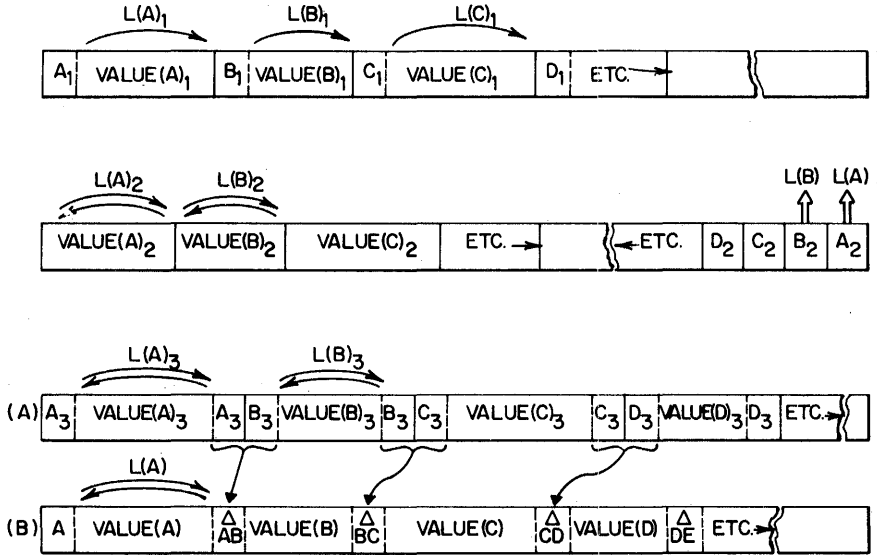


FIGURE 3. SYMMETRIC DIFFERENCE ENCODING

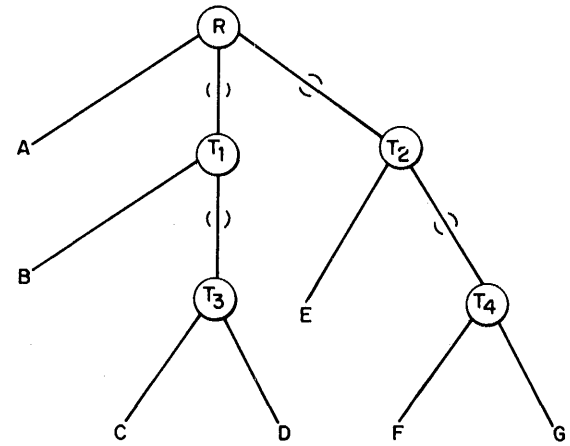


FIGURE 4. TREE STRUCTURE A(B(CD))(E(FG))

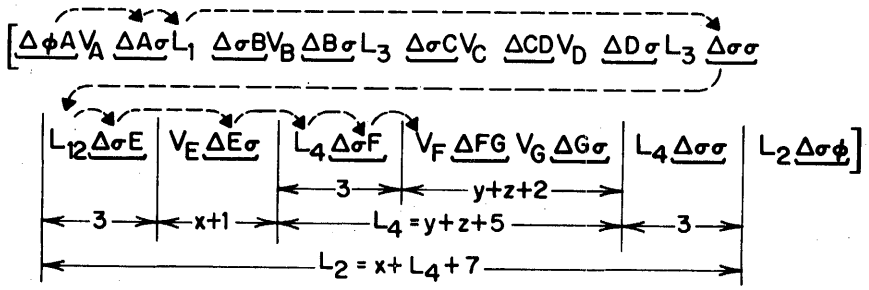


FIGURE 5. SCANNABLE LINEAR REPRESENTATION

AN INSTRUCTION CLASS FOR AN EXTENSIBLE INTERPRETER

Martin Freeman

Department of Mathematics and Statistics
The American University
Washington, D. C. 20016

INTRODUCTION

The interpretive process corresponds to creating a virtual machine to extend the capabilities of a hardware system. One of the drawbacks is that the interpreter and its instruction set are not implemented in the base hardware system; they are implemented as a series of routines written in the language of the base hardware system (machine code). This means, in many cases, that the interpreter is slower than the base system since it must be executed when a virtual instruction is fetched and executed. This paper presents the technique of interpretation through instruction execution which eliminates much of this overhead.

BACKGROUND

An interpreter can be characterized as a system that carries out the execution of a program by dynamically mapping each primitive instruction, at the time it is to be executed, into a sequence of target instructions which realize the semantics of the mapped instruction. An extensible interpreter is an interpreter which allows new instructions (routines) to be composed from primitive instructions. Extensible interpreters may be implemented by using a subroutine control structure; however, time is wasted in branching to and returning from subroutines of target instructions, especially when a primitive instruction maps to only a few target instructions. By implementing a new class of instructions which imbed the control structure of an extensible interpreter in the standard control structure, this overhead can be reduced and a valuable programming technique for constructing systems software introduced.

Typically the hardware interpreter cycle of a standard computer may be partitioned into instruction fetch, data-address computation, and instruction execution phases. Before presenting the new technique for implementing an interpreter control structure, two techniques which operate in the data-address and instruction execution phases respectively will be reviewed.

INTERPRETATION THROUGH DATA-ADDRESS COMPUTATION

In the data-address computation phase of a hardware interpreter cycle, use is made of the contents of the address field, special address field (e.g. index register field), and mode field (e.g. indirect addressing field) of an instruction to determine the memory address of data items. If the contents of the resulting location are required for the execution of the instruction, the data address is moved to an implicit register for processing in the instruction execution phase. If the address is required to store the result of the instruction execution, then the address itself is passed to the instruction execution phase. Relative addressing and indirect addressing (in combination or separately) are examples of data-address computation.

Recently Bell[1] has suggested a technique for imbedding a software interpreter in a machine whose hardware interpreter has a powerful data-address computation phase. The interpreter cycle models the following machine cycle:

1. S, the value of the PCth word of memory is fetched (where the PC is the program counter).
2a. The routine starting at location S of memory is executed.
2b. The value of the PC is incremented by one.
3. Go to 1.

Figure 1 shows the flow of control for this interpreter. Every instruction in the main code is an address pointing to the starting location of a routine; the last instruction in a routine is a jump to the next routine through the main code.

Realizing this technique on a PDP 11-45 can be done efficiently through the use of the following instruction:

JMP @ (R) +

Here R is a register which is dedicated to the imbedded interpreter. The instruction semantics are: jump to the location which is found in the location pointed to by the register R (a double indirect through R) and increment register R by one. If this instruction is placed at the end of each interpretive routine, much of the overhead of subroutine processing is eliminated. In this case the main code is a series of beginning addresses for the interpretive routines.

INTERPRETATION THROUGH THE EXECUTE INSTRUCTION

Several present-day computers have a minor interpreter cycle imbedded in their instruction execution phase through the use of an EXECUTE instruction. This instruction allows a one instruction interlude to occur where machine language instructions can be intermixed with interpretive instructions in the main program. As a consequence, an interpreter can execute machine language instructions without transplanting them into itself.

The semantics of this instruction are: the effective address of the instruction specifies, directly or indirectly, an object instruction (possibly another EXECUTE) to be

executed without changing the program counter to point to this instructions. The next instruction executed is the successor (i.e. PC←PC+1) of the EXECUTE instruction not the executed instruction, except when the object instruction is a branch, then the next instruction is the successor of the executed instruction.

The EXECUTE is useful for selecting specialized instructions for execution (e.g. I/O instructions). It is also useful for tracing a program during debugging (see Buchholz[2]).

INTERPRETATION THROUGH INSTRUCTION EXECUTION

The EXECUTE instruction imbeds a one target instruction interlude while the interpretation during data-address computation imbeds a many target instruction interlude. Suppose that a new instruction RETURN is created which combines and extends the property of interpretation through instruction execution embodied by the EXECUTE instruction and the property of the many target instruction interlude embodied by interpretation through data-address computation. This instruction can be used to realize an extensible interpreter having more than one level of definition. Routines name primitives of a higher level machine. The advantage of this control structure is that routines and primitives may be grouped together and treated as a single entity——a new routine. Figure 2 pictures this control structure as a tree while Figure 3 gives an example of a particular application.

Consider, for comparison sake, a standard memory reference instruction format as shown in Figure 4 (a); higher level instructions can be imbedded in this format. A primitive instruction may have the format as shown in Figure 4 (b), while a routine specification may have the format as shown in Figure 4 (c). Using this convention, as many locations as can be addressed using the standard format are able to be addressed.

To illustrate extensible interpretation through instruction execution, postulate a computer with a stack architecture, as shown in Figure 5, whose instruction set is given in Table 1. (A stack architecture is a convenience not a necessity.) The hardware for this architecture consists of:

1. A set of general purpose file registers that can be referenced by address.
2. A general purpose register A.
3. A register B which is the top element of a hardware control stack.
4. A register S which contains the address of the top element of a main memory operand stack.
5. A memory address register MAR.
6. A memory buffer register MBR.
7. A program counter PC.

Consider the micro-orders for a RETURN instruction (where the end of a routine is signalled by a word whose contents is zero):

```
START:   MAR ← B
         MBR ← MEM(MAR)
         B ← B+1
         IF MBR=0 THEN ( UNSTACK(B),
                         GOTO START )
         IF MBR_N=0 THEN   ( PC ← MBR,
                             GOTO FETCH )
                     ELSE ( STACK (B), B ← MBR,
                            GOTO START )
```

Here MEM is a quickverb for a memory read, STACK is a quickverb for pushing down the contents of the control stack, UNSTACK is a quickverb for popping up the control stack, and MBR_N is shorthand for the Nth bit of the MBR (i.e. the leftmost bit).

As an example of its use, consider the program segment of Figure 3 which computes the square of the number on the operand stack. It is assumed that the code for NOP is 0, that the RETURN for the primitive REPEAT is about to be executed, and that the register B is pointing to the MULT instruction in the main program.

Reflect upon the following snapshots of program execution:

1. A memory cycle is taken and the address of the MULT routine is placed in the MBR. Since it is a routine (i.e. MBR_N=1) the updated contents of register B are stacked. Register B is now changed to point to the MULT routine.
2. A memory cycle is taken and the address of the primitive TOP_IN_1 is placed in the MBR. Since it is a primitive (i.e. MBR_N=0) the address is placed in the PC so that the straightforward execution of the primitive can be accomplished.
3. After primitives TOP_IN_1, TOP_IN_2, and MLT have been executed, register B is pointing to the instruction after MLT. A memory cycle is taken and NOP is placed in the MBR. Since MBR=0, the hardware control stack is popped (into B) and processing continues in the main program.

The advantages of this implementation of the RETURN instruction are that it allows the imbedding of an extensible interpreter in a target machine, an implicit block control structure, and recursion. The disadvantages are that it requires an additional memory location for each NOP and an additional memory cycle for testing for the NOP.

Consider an alternate implementation of a RETURN instruction with the modified formats for higher level instructions as shown in Figure 6. Micro-orders for the alternate RETURN instruction are:

```
START:   IF FF=1 THEN (UNSTACK(B),FF←0)
         MAR←B
         MBR←MEM(MAR)
         B←B+1
         IF MBR_N=0 THEN (PC←MBR, FF←MBR_N-1,
                          GOTO FETCH)
                    ELSE IF MBR_N-1=0 THEN
                         (STACK(B),B←MBR,
                          GOTO START)
                    ELSE (B←MBR, GOTO START)
```

While it is true that the number of address bits is decreased by one over the previous implementation, it should be pointed out that the word length of an instruction is usually larger than the length necessary to access all memory locations (this may not be true of many minicomputers). While the hardware control stack plays more of a role in this implementation, it now contains less information as to the termination of a

routine. An example of the use of this implementation is given in Figure 7.

EXTENSIONS

The class of RETURN instructions may be augmented in several ways. The micro-order B←B+1 may be changed to

$$B \leftarrow B + MBR_{ADDRESS}$$

which allows relative addressing. Thus a command RETURN =2 would skip the next instruction in the calling routine on return. This provides security to register B while still allowing several return points. By convention, RETURN means RETURN =1.

The reader is now referred to Figure 8 where a recursive function FACT is defined which computes the factorial of the number on the top of the operand stack. Notice the primitive EQUAL, it contains a RETURN =2. This allows an instruction skip in the factorial program when the top two operand stack values are unequal and a continuation when these values are equal.

ARCHITECTURAL IMPLICATIONS

The imbedding technique described is quite efficient for routines that are memory limited; that is, those routines that contain a large percentage of memory reference instructions. This leads to a useful criterion for the selection of primitive instructions—select primitives that are memory limited.

Since recursion is allowed, and the control stack is of finite length, something must be done when the control stack is full. There are two alternatives——stop execution of the current program segment, or spill over into a protected area of main memory. Each has its associated tradeoffs.

Problems may also arise if interrupts are a consideration. It is conceivable that, if interrupts are disabled and a routine has a highly nested return structure, an interrupt may not be serviced in time. This can be averted by using either a fast control stack, or a counter which enables interrupts automatically after a fixed number of control stack levels are popped during the execution of a RETURN.

INTERCOMMUNICATION OF PRIMITIVES

In an extensible system, from time to time, new primitives may be added. It may also be necessary to imbed the control data structure (of a virtual machine) in main memory. For example, a target machine may be simulated by placing the memory and registers that are accessed by the control unit of the simulated machine in the main memory of the host machine. This means that primitives must be added to the system easily and must be able to communicate with one another through global data structures. A technique for this communication process that can be used draws upon a structure described in Wegner[5].

A primitive is composed of a communications region and a body. Addresses of global data structures (e.g. simulated memory and registers) are kept in the communications region. Target instructions in the body access the global data structure indirectly through the communications region. Thus primitives may be written and added without regard for their position or for the position of the global data structure (see Figure 9).

CONCLUSION

This paper presents a technique for imbedding an extensible interpreter in the instruction execution phase of the hardware interpreter cycle of a machine. This technique differs from the powerful interpreters of the Burroughs B5500[3] and the Iliffe Basic Machine[4] in that an interpreter is created with the use of a class of instructions, the RETURN instructions. The usefulness of this instruction class points out the need for further research into machine instruction sets.

REFERENCES

1. Bell, J. R. Threaded Code. Communications of the ACM, June 1973, pp. 370-372.
2. Buchholz, W. Planning a Computer System. McGraw-Hill, 1962, pp.146-149.
3. Burroughs B5500 Reference Manual, Burroughs Corporation, 1964.
4. Iliffe, J. K. Basic Machine Principles. American Elsevier, 1968.
5. Wegner, P. Programming Languages Information Structures and Machine Organization. McGraw-Hill, 1968, pp. 59-66.

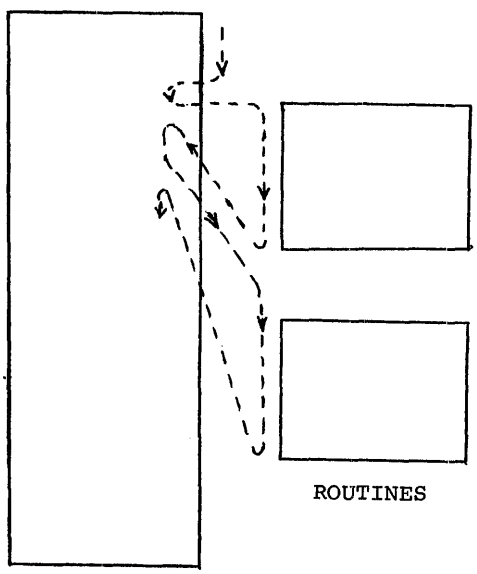


Figure 1 Threaded Code

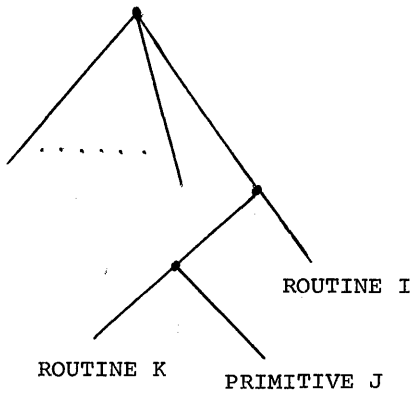
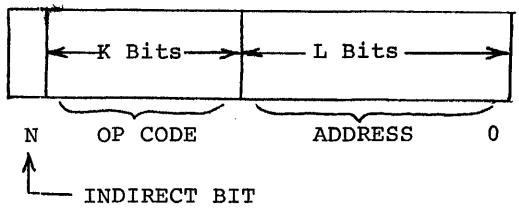
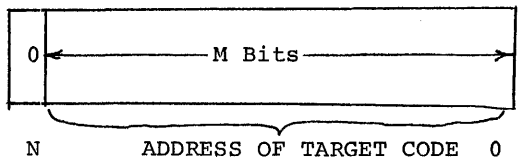


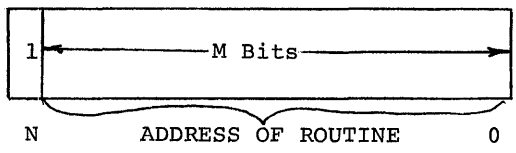
Figure 2 Tree Structure of Interpreter



(a)



(b)



(c)

Figure 4 Instruction Formats

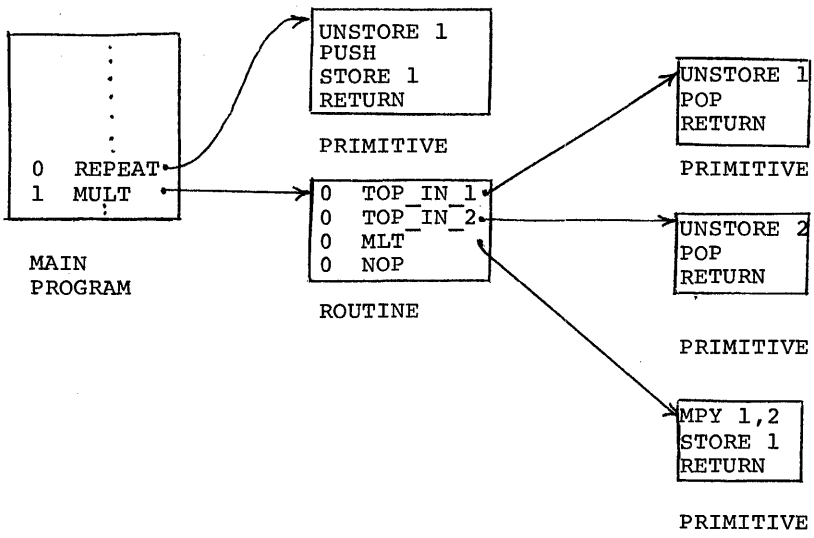


Figure 3 Square of Operand Stack for First Implementation

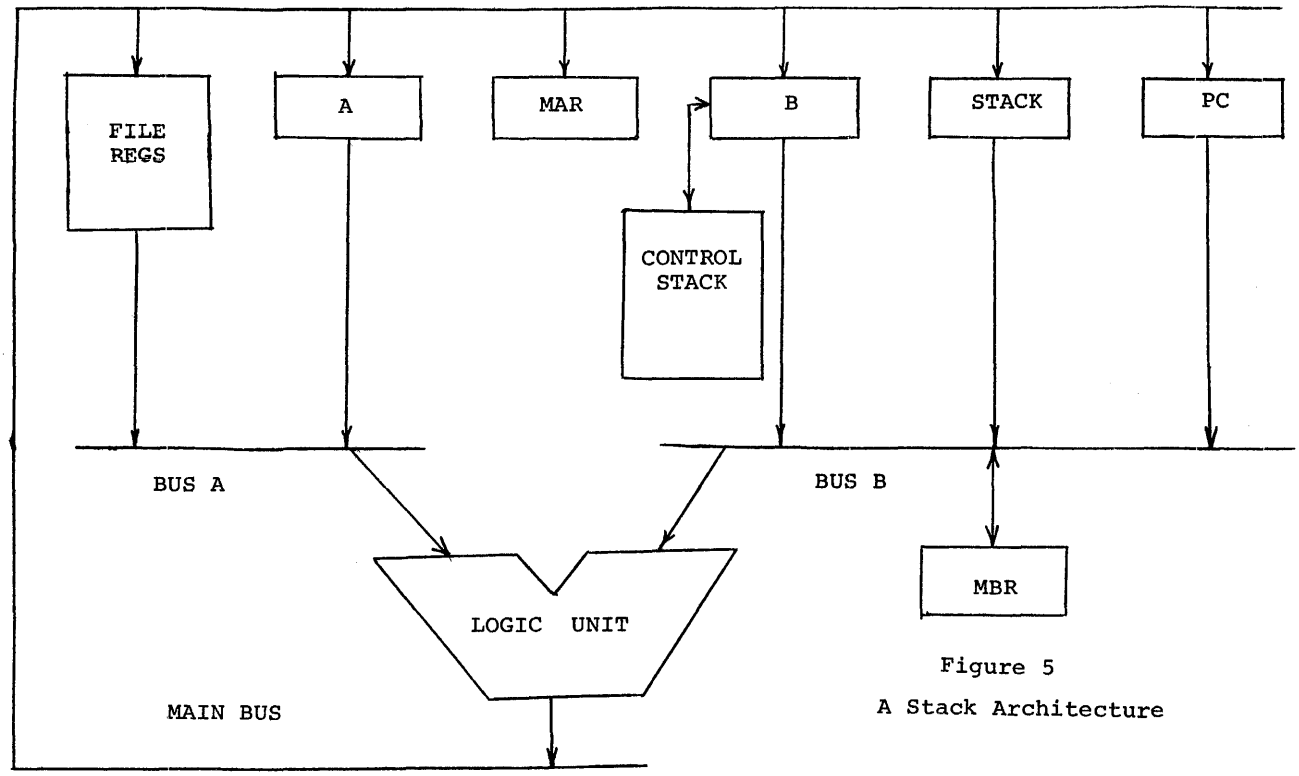
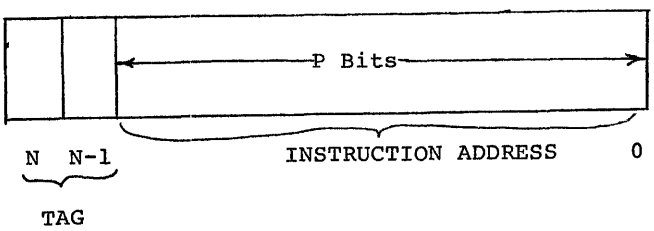


Figure 5

A Stack Architecture



TAG FIELD	SEMANTICS
0	Primitive Instruction
1	Primitive Instruction which is the last instruction in a routine
2	Routine
3	Routine which is the last instruction in a higher level routine

Figure 6 Instruction Format

TABLE I

Instruction Set of a Stack Machine

INSTRUCTION	SEMANTICS
LOAD I,=J	Load integer J into register I
STORE I	Copy the contents of register I into the top level of the operand stack
SUB I,J	Subtract the contents of register J from the contents of register I and place the result in register I
MPY I,J	Multiply the contents of register I by the contents of register J and place the result in register I
UNSTORE I	Copy the contents of the top level of the operand stack into register I
PUSH	Place a new uninitialized level on the top of the operand stack
POP	Remove the top level of the operand stack
NOP	Non-operation
SZE I	Skip the next instruction if the contents of register I is zero

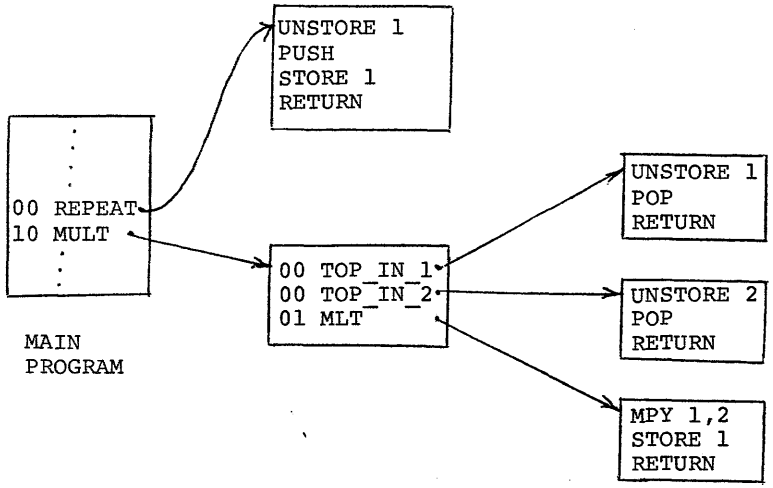


Figure 7 Square of Operand Stack for Alternate Impementation

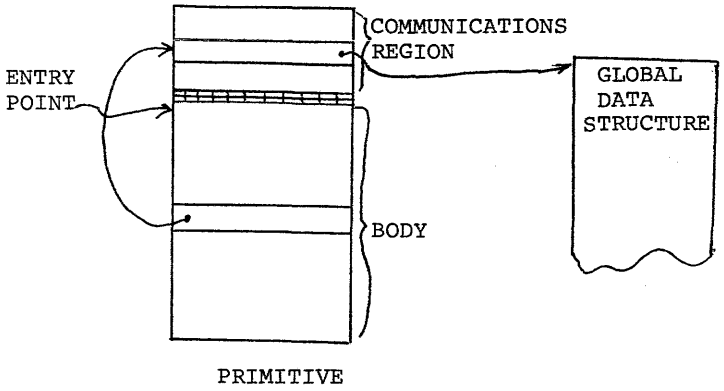


Figure 9 Intercommunication Structure of Primitives

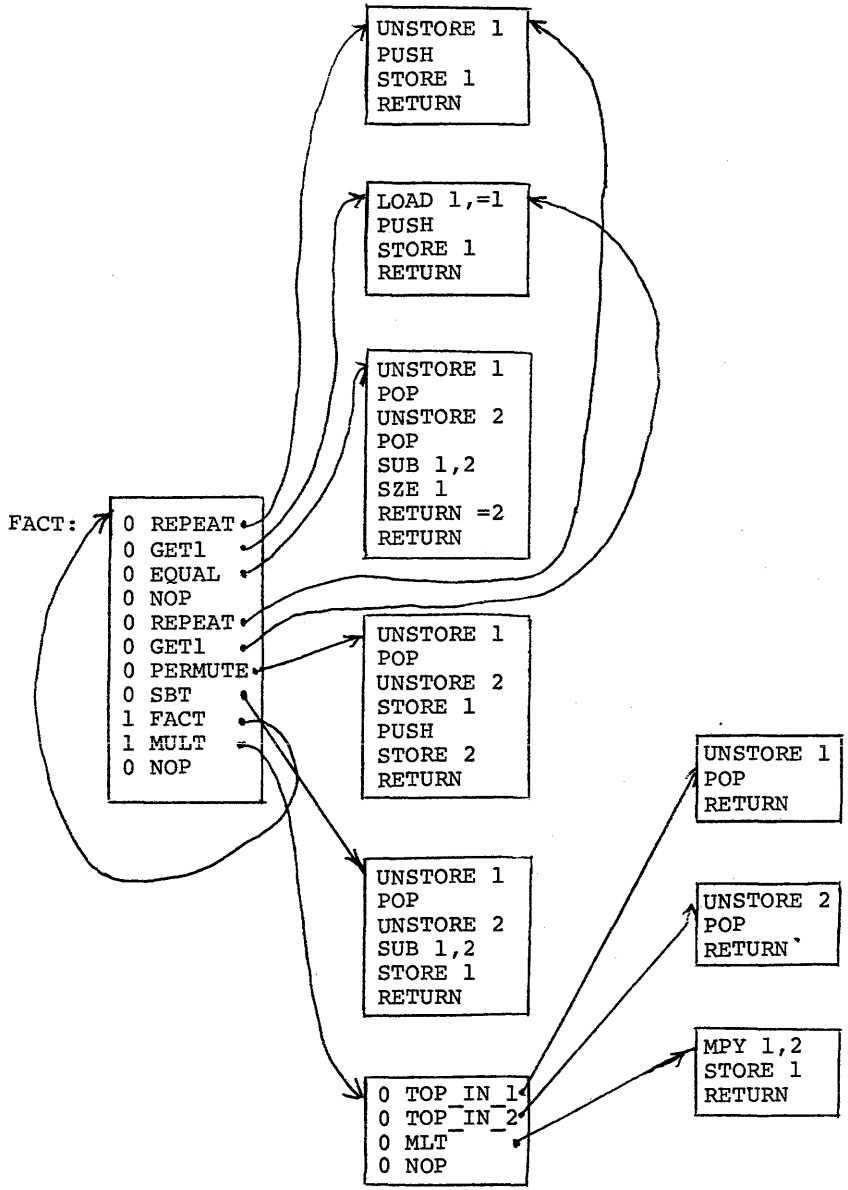


Figure 8 Factorial of Operand Stack

STARLET - A COMPUTER CONCEPT BASED ON ORDERED SETS AS PRIMITIVE DATA TYPES

W.K. Giloi and H. Berg

Department of Computer, Information, and Control Sciences
University of Minnesota
Minneapolis, Minnesota 55455

Summary

It is a well-acknowledged fact that the adequate structuring of data is as important as the adequate structuring of a program. The classical von Neumann - machine deals on the hardware level only with scalars and, therefore, provides no hardware support of any sort for the manipulation of more complex data structures. Some more recently developed concepts - all variants of Iliffe's Basic Language Machine[1] -- provide some hardware support for structuring data in the form of trees. Contrastingly, in the STARLET[2,3] concept to be discussed in this paper, the basic data structure is that of ordered sets (as given by strings and arrays). In the paper, the basic notions of this novel concept are introduced and discussed, especially the internal information structure which differs considerably from that of the von Neumann-machine or other concepts. This very idiosyncratic information structure has far-reaching consequences with respect to the hardware structure of the machine. Finally, a number of resulting features will be discussed and an attempt will be made to compare the computing power of STARLET with that of a von Neumann-machine with equally fast hardware.

Introduction

STARLET is an attempt to organize the hardware of a computer on a higher level than that of the classical von Neumann-machine. This is primarily accomplished by introducing ordered sets as the elementary data type of the machine -- in lieu of the scalars to be found in the von Neumann-machine. This has a number of far reaching consequences which shall be listed in the following.

Instruction Format

Unlike the instruction format of a von Neumann-machine, STARLET instructions do not refer to memory locations but to variables representing entities of data. All data of such an entity are of the same type, and they are ordered in the form of strings or matrices. Variables are identified by names. Because of the higher complexity of this basic data structure of STARLET, the processing of an instruction for a dyadic operation requires the generation of two source streams of data flowing into the data processing unit, and the generation of an object stream carrying the results of the operation. Therefore, the general format of a STARLET instruction consists of the operation code and the referencing of three variables, two source variables and one result variable. Of course, the interpretation of a STARLET instruction requires that the referenced variables must be somewhere described to the machine. For reasons which will immediately become obvious, such a variable-descriptor is not part of the STARLET instruction format but stored in a separate list called variable specification list (VSL).

Variable Specification List (VSL)

The VLS is the central part of the internal representation of a STARLET program. It is stored in a fast-access semiconductor memory and has one entry for each variable. These entries are called variable-descriptors and contain all necessary information about

- -the data type of the variable
- -the species the variable belongs to
- -the scheme according to which the data are ordered
- -the location of the data in a 'data list' (DL).

Since the variable-descriptors are stored in fixed memory locations, their addresses can be used as the internal variable identifier ("call by name" mechanism).

Species of Variables

The most important consequence of the separation of instructions which reference variable names and the descriptors of these reference variables is that all "restructuring operations", whose only effect is to change the ordering of the data in a set (a variable), can be executed purely on the variable-descriptors. New variables generated in such a way are termed "pseudo" variables as they have a descriptor but no data of their own. Conversely, variables which do have data of their own are called "true" variables. In other words, a pseudo variable shares its data with one of the existing true variables (only the ordering of these data has been modified or a subset has been selected). Hence, restructuring operations such as, for example, matrix transposition, indexing, and mapping, create only new variable-descriptors but leave the data untouched. Thus any unnecessary copying of data in the machine is avoided.

Hardware Structure

The decomposition of a program into a list of instructions (IL), a list of variable-descriptors (VSL), and a list of data (DL) suggests that each one of these lists is interpreted or processed, respectively, by its own processor. This leads to the rather orthogonal hardware structure of STARLET, as illustrated in fig. 1, which may be characterized as "an asymmetric four-processor system". The description of the detailed structure and the functions of such a system will be the major topic of this paper.

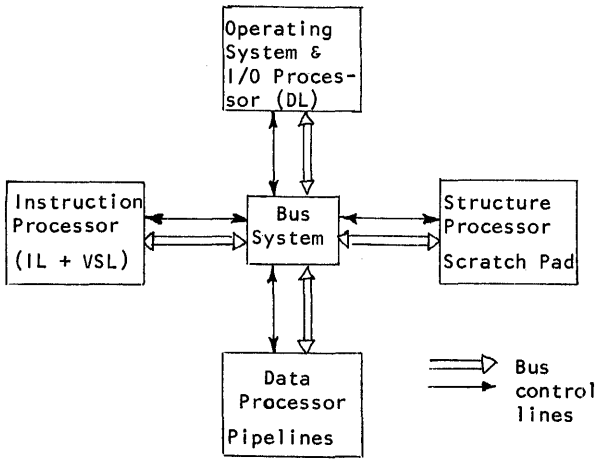


Fig. 1 Basic Structure of the STARLET System

The Information Structure of STARLET

The information structure of the classical von Neumann-machine consists of a program and its data.

An initial representation of such an information structure is read into the machine, and a computation is a sequence of transformations mapping this initial representation into a final representation. In STARLET, the initial representation of the information structure is a program with ordered sets of data. Hence, if L_p denotes the set of all syntactically correct STARLET programs and D_s the set of all structured data entities constituting the domain of L_p, we have the initial representation $R_0 \subseteq L_p \times D_s$. The first transformation is then given by the translation

$$\alpha : \quad R_0 \rightarrow I \times VS \times D \qquad (1)$$

where I is the set of instructions, VS is the set of variable-specifications ("descriptors"), and D is the set of all components of D_s. The variable-descriptors are derived from the declaration of the sets of data which are input to the program. The ordering relation on D is once and for all defined in the form of the row-major linear order. This transformation is performed by the I/O processor.

In the process of computation any such internal initial representation $R_i \subseteq I \times VS \times D$ is transformed into a final representation R_f according to one of the following functions

$$\beta : \quad I \times VS \times D \rightarrow VS \times D \qquad (2A)$$
(Data generating operations)

$$\gamma : \quad I \times VS \rightarrow VS \qquad (2B)$$
(Restructuring operations)

Eventually, we want to output results, i.e., to perform a mapping

$$\omega : \quad VS \times D \rightarrow D_S \quad . \qquad (3)$$

Generally, the transformations (2A) and (2B) are performed in collaboration of the structure processor and the data processor, whereas transformation (3) applies to the instruction processor as well as to the structure processor. Of course, the instruction processor is also employed for instruction interpretation which may, on the other hand, require the assistance of the data processor (for example, if a conditioned jump has to be executed).

Whereas input variables have to be declared as to their structure and their data type, the name-value binding of result variables (including all intermediate results) is automatically performed during program execution. Therefore, the STARLET assembly language is to a large extent declaration-free. Figure 2 depicts the information structure in STARLET as discussed here. The components of this structure shall be discussed in more detail in the following.

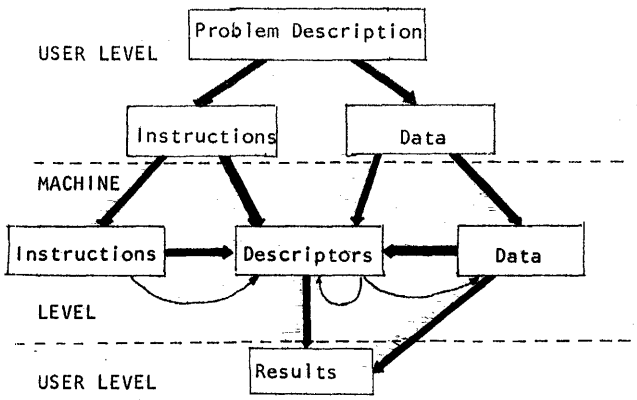


Fig. 2 Information Structure of STARLET

Instructions

As mentioned before, 3-address instructions are a prerequisite for dyadic operations performed on data streams in a pipeline mode. Since variables are defined by separate variable-descriptors (not being part of the instruction), the instruction format has only to accomodate the operation code and the names of three variables and can, therefore, be kept relatively small (e.g., 35 bits).

The set of STARLET operations is basically a subset of APL (with some modifications). The data generating operations are given by a subset of the APL primitive scalar and composite functions. Restructuring operations are: Transposition, rotation, indexing, mapping. The STARLET instruction repertoire contains also organizational instructions such as unconditioned and conditioned jumps, subroutine jumps, stack instructions, and I/O instructions. Table 1 gives the complete instruction list.

The STARLET instruction format permits the introduction of the transposition operator as a prefix to the variable name and, therefore, data generation instructions can be performed on matrices as well as on transposed matrices. Consequently, the transposition does not occur explicitly in the instruction set. Any extension of this scheme to restructuring operations other than the monadic, parameterless operation of transposition, however, would sacrifice the advantages of the descriptor concept such as the compact and redundancy-free instruction format and the flexibility of the machine language.

Data

The STARLET hardware recognizes the following four data types: REAL, INTEGER, BOOLEAN, CHARACTER. The pipeline mode of processing of ordered sets of data requires the generation of three streams of memory addresses in order to fetch the two operands and to store the result of a dyadic operation. These address streams have to be generated by a special hardware (called structure processor) in order to obtain the maximum rate as permitted by the memory cycle time. As mentioned before, data are always stored in the data list in a row-major linear order, whereas the order in which they are fetched can be arbitrary. The structure processor can easily be designed such that the generation of memory address streams includes data conversion functions. Hence, depending on the data type, streams of bits, bytes, or words are fetched and stored.

Variable-Descriptors

Variable-descriptors specify: CLASS, TYPE, SPECIES, DIMENSION, LOCATION OF DATA of variables. The meaning of these parameters shall be explained in reverse order.

LOCATION OF DATA is a pointer to the data list where the first data item of the set represented by the specified variable can be found. DIMENSION specifies the array dimension (number of rows and columns), or the number of characters in a string, etc. Since data are always stored in row-major order, the ordering of variables is thus defined by the dimension parameters (vectors are defined as matrices either with only one row or only one column).

SPECIES indicates whether a variable is a true variable or a pseudo variable. True variables are created by a data generating operation. Pseudo variables are created by a restructuring operation. In the machine, true variables are represented by a descriptor and a set of data, whereas pseudo variables are only represented by a descriptor. Figure 3a illustrates the two-stage process of addressing data through a descriptor. A pseudo variable descriptor does not reference data directly, since the variable

does not have data of its own. Instead, the descriptor references the true variable whose data are reordered or selected by the operation creating the pseudo variable. Furthermore, a pseudo variable descriptor has to contain all parameters specifying the generating operation (such as rotation parameters, or indexing (selection) parameters, or the name of a mapping vector, respectively). The scheme of addressing data through pseudo variables is illustrated in figure 3b. From the view point of machine hardware, a descriptor has a dual function: (1) it contains additional information necessary for the interpretation of an instruction and (2) it may be considered as being a micro-instruction to the structure processor causing the generation of data streams in a specified order.

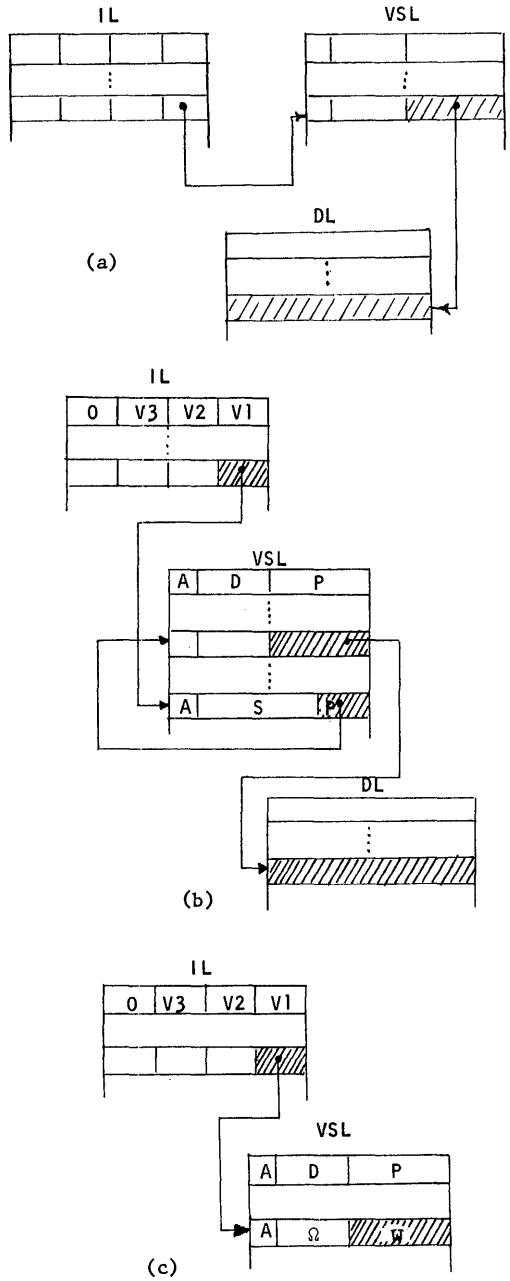


Fig. 3a-c Data Referencing Scheme

(a) for true array variables; (b) for pseudo array variables; (c) for scalars

In a pseudo variable descriptor, restructuring operations are, together with their parameters, independently specified for the two coordinates of a matrix. Hence, all possible combinations are permitted and, as a result, restructuring operation can be recursively performed on pseudo variables without limitation as to the depth of recursion. Each time, all that has to be done is to calculate new restructuring parameters as a function of the actual parameters in the descriptor and the instruction. This scheme is similar to the "beading" technique developed by Abrams,[4] but it has been refined to the point that a fixed descriptor format (e.g. of 35 bits) is never exceeded.

TYPE specifies the data type of a variable.

CLASS distinguishes between two possible classes of variables, namely ORDERED SETS and SCALARS. Such a distinction is an internal measure for improving the efficiency of operations performed on scalars, but it is not pertinent for the user. As depicted in figure 3, the access to data items is a two-stage process which requires first to access the respective descriptor. Whereas such a technique is the appropriate device for handling more complex data structures, it penalizes the use of single scalars. As scalars are true variables of dimension zero, all that has to be specified is its type, and the remaining part of the descriptor would be empty except for the pointer to the memory location where the scalar is stored. Therefore, it is a better method to store the scalar directly in the VSL word that is assigned to its name (in this particular case, of course, there is no difference to the von Neumann-machine). Figure 3c depicts the one-stage addressing scheme employed in the case of scalars.

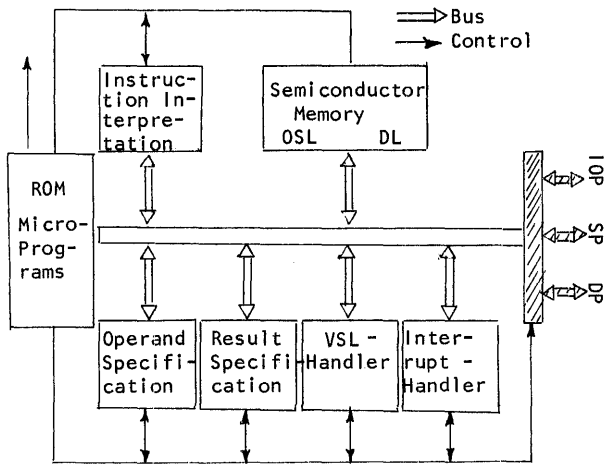


Fig. 4 Instruction Processor Block Diagram

At this point it may be useful to compare briefly the STARLET information structure with the one to be found in the BURROUGHS B5700/B6700 computers.[5] The main objective of the information structure in the BURROUGHS machines is to group data into "access regions" which correspond with the actual working set of a program. Therefore, data are grouped according to the scope of their identifiers or, in other words, according to the ALGOL-like block structure of the program. Descriptors in these machines identify program segments (blocks).

Whereas the B5700/B6700 computers can be characterized as "ALGOL-type" machines, STARLET is an "APL-type" machine in the sense that we find on the machine level almost all APL primitive functions, but

not the "defined function" mechanism. Hence, a STARLET machine program is a piecewise linear sequence, and notions like "block structure" are on this level meaningless (it has to be considered that, because of the higher power of APL mixed functions, most of the repetitive loops to be found in languages like ALGOL are simply not necessary). Hence, STARLET descriptors are not descriptors of program segments but of structured variables, acting as micro-instructions to the very unique "structure processor". The only common feature which we can see is that, in STARLET as well as in the B5700/B6700 computers, memory cells are accessed through variable-identifiers.

System Organization

In the following, we shall discuss the four major components of a STARLET system, I/O PROCESSOR, INSTRUCTION PROCESSOR, DATA PROCESSOR, and STRUCTURE PROCESSOR, with respect to their functions, interrelations, and mutual actions.

I/O Processor

The very core of the system, the I/O processor, will be a commerically available, general-purpose minicomputer that will come with all required channels, device controllers, and peripherals. Preferably, the minicomputer should be micro-programmable, since this feature provides a most elegant way of conditioning the computer as part of the system. Its memory must be of sufficiently large capacity as it has to accomodate the operating system with all its components (assembler, translator, etc.) as well as the data list.

Once the I/O processor has provided the internal information structure, as given in the form of the instruction list (IL), the variable specification list (VSL), and the data list (DL), the remaining tasks are to store the data list and to handle I/O. Therefore, it is advantageous to have a computer with dual port memory access. Requests to the operating system are transmitted via interrupts. Output requests have to be accompanied by the format specification as given in the instruction.

Instruction Processor

Tasks of the instruction processor are: (1) program control, (2) instruction interpretation, (3) initialization of data processor and structure processor, and (4) generation of requests to the I/O processor. The memory of the instruction processor is a fast-access semiconductor memory that has to accomodate the instruction list and the variable specification list. The instruction processor is initialized by a transfer of these two lists from the I/O processor.

As result variables are created during program execution, it is important to check automatically the conformity of the operands of an instruction before execution. Such conformity tests are partly built into the hardware and performed on the variable-descriptors. Variable-attributes to be tested by hardware are:

CLASS: certain operations are only defined on one of the two variable classes (e.g., restructuring operations can only be performed on variables of class array). Furthermore, the instruction execution may depend on the variable class (e.g., operations on scalars involve only the data processor, operations on arrays involve data processor and structure processor, etc.).

DIMENSION: In dyadic operations which are executed componentwise, the dimensions of the operands have to be equal. In the generalized inner product, the row-dimension of the first operands must equal the column-dimension of the second operand. In stack operations, the stack width must equal the respective variable dimension.

TYPE: In almost all operations, the operands must be of equal type.

Before an instruction can be executed, the instruction processor has to initialize the data processor and the structure processor by transferring all parameters specifying the structure of the operands and the operations to be performed. Furthermore, the instruction processor must calculate the parameters specifying the structure of result variables (as a function of the instruction and the operand structures) and to communicate them to the structure processor. If the prefix operator denoting transposition of a variable is encountered, that information has also to be communicated to the structure processor. Program control instructions (jumps) are immediately executed by the instruction processor; in the case of conditioned jumps, however, the assistance of the data processor is required in order to determine the truth value of the conditioning relation. Figure 4 depicts a block diagram of the instruction processor.

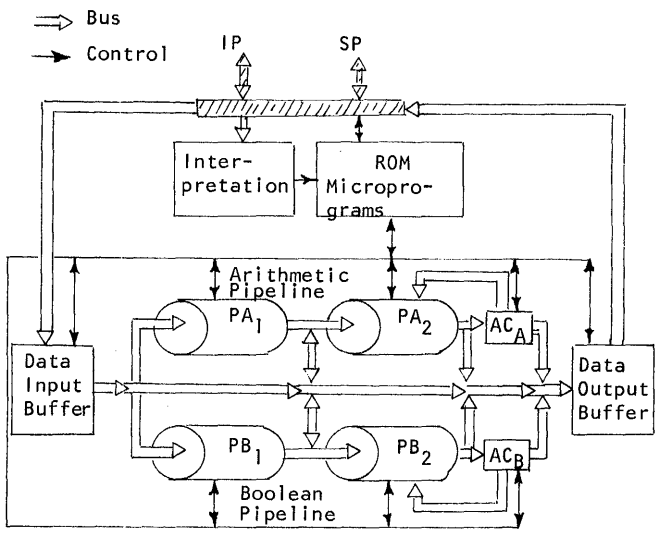


Fig. 5 Data Processor Block Diagram

Data Processor

The processing of data streams requires the repetitive execution of certain sequences of operations performed on scalars, each time generating a component of the result stream. The hardware structure of the data processing pipeline has to be designed for the most complex operations, namely the generalized inner products. Here, we have two dyadic operations, (f,g). f combines dyadically row-elements of the first operand with column-elements of the second operand, and g performs subsequently a reduction (in the APL definition) on the resulting components. Therefore, the pipeline has the structure P=(P_1,(P_2, AC)), that is, a processing unit P_1 is followed by a processing unit P_2 which is followed by an accumulator AC.

Since the instruction repertoire encompasses generalized inner products of boolean as well as of numerical data, the data processor contains two pipelines, one for boolean and the other for arithmetic operations (the "boolean" pipeline is very inexpensive as it processes only single bits). The order of P_1

and P_2 can be interchanged. With $P_1 = \times$, $P_2 = +$ in the arithmetic case and $P_1 = \wedge/\neq$, $P_2 = \vee/\equiv$ in the boolean case, respectively, we have the combinations $(\times,+)$ and $(+,\times)$ in the arithmetic case and $(\wedge,\vee)$, $(\vee,\wedge)$, $(\wedge,\neq)$, and $(\vee,\equiv)$ in the boolean case. It need hardly be mentioned that each processing unit of the pipeline can act autonomously.

Supply of Data

Scalar data are supplied by the instruction processor, whereas the components of arrays are fetched by the structure processor from the data list. The desired parallel operation of structure processor and data processor requires the buffering of the operand data streams. Such a buffer provides the additional advantage that, in operations combining a scalar and an array, the scalar can be stored in the buffer as long as it is needed. Likewise, a buffer is required for the result stream. Figure 5 shows a block diagram of the data processor.

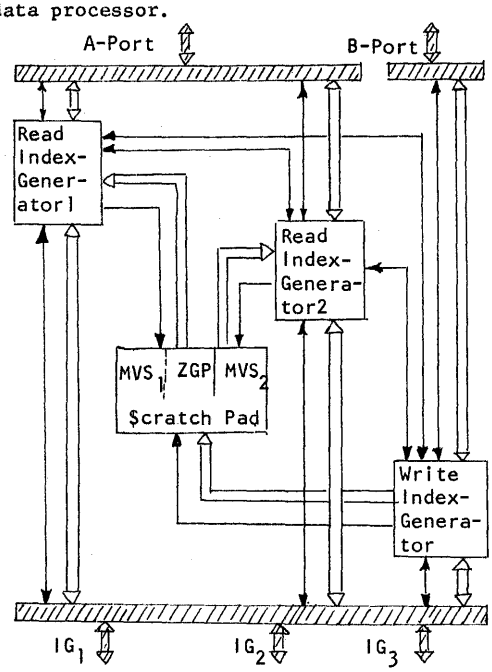


Fig. 6 Structure Processor Block Diagram

Structure Processor

The structure processor generates the address streams which are necessary in order to fetch operands from the data list, process them in the pipeline, and store the results back into the data list. As this has to be done at a high rate (only limited by the memory cycle time), the three address streams are generated by respective dedicated hardware units which we call index generators.

The two read-index generators need to have a certain computing power enabling them to execute the variety of index generation algorithms as required for the various restructuring operations. The most complex operation is in the case of mapping

$$MV1(k_1 \bmod i)i + MV_2(k_2 \bmod j) + B\ T\ ,$$

where MV1 and MV2 are the mapping vectors, and in all other cases

$$(k_1 \bmod i)i + k_2 \bmod j + B\ T\ .$$

B is a base address, T is a factor depending on the data type (addressing of bits, bytes, or words), and $i,j,k_1,k_2 \varepsilon [1 : n]$ are sequences of index numbers. Thus the required computing power is to count, to add, and to multiply.

The write index generator is much simpler as it has always to store data in a linear (row-major) order. This can be performed by a (modulo-n)-counter which controls a (modulo-m)-counter (m being the row dimension and n being the column dimension of an array). In the case of transposition, we have the same simple scheme, only that the role of the two counters is interchanged.

The execution of the generalized inner product requires the cyclic read-out of the rows or columns, respectively, of the two operands. In order to speed up this procedure, the structure processor is equipped with a fast-access scratch pad into which the respective row of the first operand is loaded (thus a multiple read-out from the data list is avoided). The same scratch pad is used to store the mapping vectors in the case of the mapping operation. The transfer of data from DL to the scratch pad is controlled by the write-index generator. Figure 6 shows a block diagram of the structure processor.

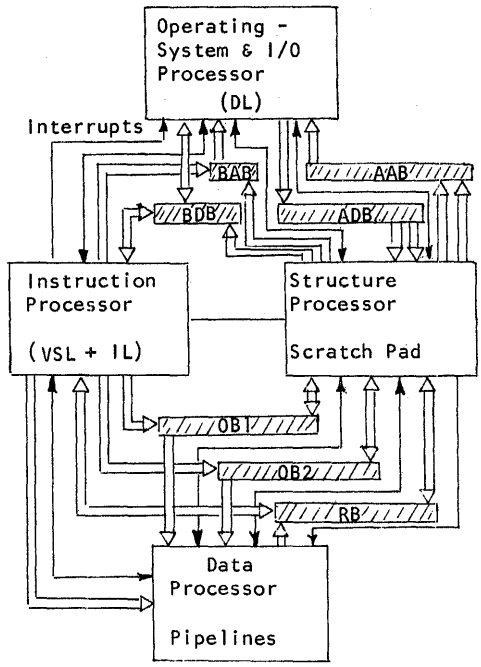


Fig. 7 STARLET Bus Structure

Bus System

The STARLET bus structure for interprocessor communication is illustrated in figure 7, assuming a dual port memory access in the I/O processor.

Resulting Features

Machine Language and High-Level Programming Language

STARLET provides on the hardware level the possibility to perform complex operations on sets of data ordered in the form of arrays. These operations are a powerful, slightly modified subset of the APL primitive functions.

The high-level programming languages to be implemented on a STARLET machine depend, of course, on the area of applications and the mode of operation (batch or conversational mode). Naturally, the machine is predestined for being an APL machine as it supports strongly the APL operations as well as the interactive mode of operation (delayed variable-value binding). However, STARLET will equally well support, for example, a discrete event simulation language designed for a batch mode of operation (supporting factors are the notion of ordered sets as the basis data structure and the stack instruction to be discussed later).

Instruction Look-Ahead Mechanism

Since the instruction processor is not involved in the processing of data, simultaneously to the execution of the current instruction it may already interpret the next instruction, calculate the result-descriptor, and prepare the initialization for the following instruction execution. JUMP instructions can also be executed in a look-ahead fashion. Of particular interest is that a restructuring operation may be executed by the instruction processor (as this requires only the creation of a new descriptor) while a data generating instruction is under execution. However, the calculation of a new array and its subsequent restructuring is a combination which may be encountered in a program many times.

Besides of the instruction look-ahead, a data look-ahead can be performed. This is important if the data list is stored in a virtual memory. Special STARLET properties which make such a data look-ahead especially efficient are: (1) STARLET programs will hardly contain DO-loops, as the basic data type is already a matrix (thus indexing is only required in the case of hyper-matrices). Therefore, jump instructions will usually occur in the only context of program branching. Hence, the average length of linear sequences in a STARLET program can be expected to be fairly large. (2) Each instruction look-ahead may already result in the swapping of two rather large blocks of data.

Stack Instructions

Stacks can be programmed simply by defining an internal array variable as a stack. Therefore, the stack width and depth can be arbitrarily declared (and is limited only by the maximum array dimension). The internal mechanism is as follows. As a function of the declared stack dimensions and the data type, a dummy matrix is created as an internal true variable. Furthermore, the variable referred in the stack instruction is created as a pseudo variable. This pseudo variable which is specified by a row-rotation, references the (internal) true matrix variable. Stack operations such as push-down and pop-up are executed by incrementing or decrementing, respectively, the rotation parameter. A matrix concatenation is a special kind of stack operation that requires only the addition of the row dimension of the additional matrix to the current rotation parameter. Once a stack has been completely filled up, it can be used like any other array variable.

Specifications

We decided to use a 32 bit data word for a real number and a 36 bit word (including a parity bit) in IL and VSL. Thus, the maximum matrix dimension is 127×127, and the maximum character string length is 4097. A semiconductor memory module of 4K×36 bits is considered to be sufficient for accomodating IL, VSL, and the scratch pads (for which 192 words are needed). The maximum time for a floating point operation in the data processor is 250 nanoseconds. The processing of one or two variables with n components, which requires n such operations for primitive monadic or dyadic functions and $2n^3$ operations for the inner product, takes the following number of memory cycles: Monadic function: n+3; dyadic function: 2n+4; inner product: n^3+n^2+3n+2. Hence, in the case of the inner product, the number of cycles required is smaller than the number of operations as soon as two matrices greater than (3×3) are multiplied.

Operation	Notation
NOT	B ← ~ A
ABSOLUTE VALUE	B ← |A
FLOOR	B ← ⌊A
CEILING	B ← ⌈A
ROW REDUCTION	B ← o/A
EQUAL	C ← B = A
UNEQUAL	C ← B ≠ A
GREATER	C ← B > A
LESS	C ← B < A
ADDITION	C ← B + A
SUBTRACTION	C ← B − A
MULTIPLICATION	C ← B × A
DIVISION	C ← B ÷ A
INNER PRODUCTS	C ← A+.×B
	C ← B×.+A
BOOLEAN MATRIX PRODUCTS	C ← B $o_1.o_2$ A
IDENTITY	B ← A
ROW ROTATION	B ← k ⌽ A
COLUMN ROTATION	B ← k ⊖ A
ROW MAPPING	B ← u ⍋ A
COLUMN MAPPING	B ← u ⍋⍋ A
ROW INDEXING	B ← i ⍳ A
COLUMN INDEXING	B ← i ⍳⍳ A
STACK DECLARATION	m,n,t ⍴ K
PUSH DOWN	K ↑ A
POP UP	A ↓ K
ROW DIMENSION	B ← ⍴A
COLUMN DIMENSION	B ← ⍴⍴A
VALUE	B ← (A)
ROW REPLACEMENT	B ← k : A
COLUMN REPLACEMENT	B ← k :: A
MEMBERSHIP	B ← a ∈ A
UNCONDITIONAL JUMP	→ (i)
CONDITIONAL JUMP	→ (AoB × i)
JUMP TO SUBROUTINE	→ (name)
INPUT	A ← □
OUTPUT	□ ← A
DIMENSION OUTPUT	□ ← ⍴A

TABLE 1

List of STARLET Operations

References

1 Iliffe, J.K., Basic Machine Principles, American Elsevier Publishing Co., New York, 1968.

2 Giloi, W.K., STARLET - Das Konzept eines interaktiven Kleinrechners fur die Array-Verarbeitung, Rechnerstrukturen, Proc. IBM Symposium on Computer Architecture in Wildbad, Germany, 1973, R. Oldenbourg-Verlag.

3 Giloi, W.K. and Berg, H., STARLET - An Unorthodox Concept of a String/Array Computer, Proc. IFIP Congress 1974, vol. 1, 103-107.

4 Abrams, P.S., An APL Machine, SLAC Report No. 114, Stanford University, 1970.

5 Organick, E.I., Computer System Organization: B5700/B6700 Series, Academic Press, New York, 1973.

A CELLULAR GENERAL PURPOSE COMPUTER

R. G. Cornell
Bell Laboratories
Naperville, Illinois

and

H. C. Torng
Cornell University
Ithaca, New York

Abstract

A 2-dimensional cellular general-purpose computer is specified. This particular cellular computer is distinguished from previously proposed, locally-controlled cellular computers in that the cellular structure is "hidden" from the user. At the ISP level, the machine is similar to a small-scale computer of the von Neumann type. However, the architecture of the computer does not feature physically isolated functional units to implement memory, processor, or control. As a result, we present a machine which may be programmed in the conventional manner, but which has the hardware advantages associated with the cellular structure. Additionally, the machine is controlled by a software microprogram, which lends itself to dynamic microprogramming and to such related applications as machine simulation.

Introduction

Continued interest in the investigation of cellular array implementation of computing devices has been inspired by the emergence of LSI as a state-of-the-art technology. Cellular realizations have been proposed which serve as universal logic modules,[1] functional units,[2,3] and complete computers,[4,5,6] with hardware complexity per cell ranging from several gates to tens of thousands of gates. Often, cellular computers proposed incorporate several cell types, and almost invariably the devices exhibit functional characteristics unique to cellular structures (these characteristics often make the programming of these devices nonintuitive or prohibitively complicated). In contrast, we report herein on the architecture of a cellular general purpose computer (CGPC) distinguished from those previously proposed in two respects. First, the cellular hardware structure is "hidden" from the programmer; the machine appears (at the instruction set processor [ISP] level) to be based upon the familiar von Neumann architecture. Second, the computer's machine language is determined by a software microprogram which may be altered dynamically. The computer's physical organization is presented, as well as a simple machine instruction set which is intended to serve an exemplary purpose during a discussion of CGPC operation.

CGPC Architecture

The computer consists entirely of a regular 2-dimensional array of identical circuits called cells. Each cell not assigned to an array boundary is connected to its four neighboring cells in an identical fashion (Figure 1). The cell contains no identifiable computing units such as full adders, counters, etc. Between any two cells defined as neighbors, the intercell communication which transpires at time t consists of a cell-edge output from each of these cells to the other. Cells are synchronized by an external clock such that cell internal state transitions for all network cells are enabled at integral time instants defined by that clock. We define a network cycle as the period between two adjacent clock pulses.

All aspects of program execution (storage, arithmetic and logical operations, and control) are performed within the array, independent of external control devices. All communication with peripheral devices transpires at an array boundary. There is no hard-wired specialization of any cell or group of cells through cutpoint[1] techniques. No region of the array is set aside to serve as a dedicated functional unit (e.g., control, ALU, etc.), hence all computer functions are fully distributed among the array's cells. Finally, the size of the array is not limited by the cell's particular structure, hence the control scheme employed accommodates a range of array sizes.

Functional Structure

The structure of the basic cell, as well as the cell intercommunication format and array boundary interconnection network, can best be explained if the functional organization of the CGPC is first presented. The M x N cell array is viewed as an M register device, each register being qN bits wide (where q is the number of bits of cell storage). The user's program, stored in the CGPC, is organized into words, each word occupying one network register. These words may be either instructions or operands. Any register of the array must be available for word storage, hence each register is logically identical to every other register in the CGPC. Each register is partitioned into six functional fields, as depicted in Figure 2A. An explanation of the bit lengths of each field will be temporarily deferred. Register fields are defined independent of the register's content.

Register Field Assignments - The two word types identified in association with the user's program are partitioned as shown in Figures 2B and 2C. The machine instruction features an operation code field and two fields for the identification of operands. An operand format field is incorporated for the purpose of expanding the operation code set where appropriate by providing a means whereby the function of the operand fields may be modified. Through the use of an associative addressing scheme, each word carries its own unique address, while each storage location (i.e., register) has no fixed address. It is to this end that an address field is included.

Finally, an activity state field is used to alert the cells that make up a register of the type of word stored in that register and the current functional activity of the register.

An operand stored in a register is again partitioned into six fields. The activity state field and address field serve identically as specified for the machine instruction. The three fields that are assigned to data storage are typically used as though they were a single field, although the fields may be used independently. A condition code field is incorporated to store the conditions detected during the last operation performed on that operand. This convention simplifies the interrupt handling routine as there is no localized condition register or flag in the CGPC.

The actual movement of data inside the array is minimized through the use of associative addressing and the activity state concept. Any register used for operand storage is available to the control unit as a general register. An operand is associatively located and tagged, and a logical or arithmetic operation is then performed on the datum in place (with a single data transfer being required in the case of a dyadic operation). Under this scheme, the CGPC may appear to the user to have an arbitrary number of general registers, a feature which is most attractive for machine emulation purposes.

The control unit itself consists of hardware uniformly distributed among all registers in the CGPC. The control sequence is provided by a microprogram stored in the CGPC registers in the same manner that the user's program is stored. The microinstruction shares the 6-field format provided by those registers (Figure 2D). The distinction between main storage and micro storage is made in the activity state field. Microinstructions are located by the content of their base address field, modified by the content of the branch code field; the machine is designed such that codes used as microinstruction base addresses may concurrently be used as main program addresses without causing ambiguity in addressing at either level. Microinstruction addressing will be discussed further by example in the section on operation. The microinstruction incorporates a microoperation code field used for array control and a peripheral order code field used for control of devices external to the array. The next address field denotes the base address of the next microinstruction to be executed in the control sequence.

Addressing - Microprogram sequencing is easily implemented by associative addressing. A microinstruction is located through the generation of an associative match between all microinstruction base addresses and the active microinstruction's next address symbol, accompanied by an associative match between all microinstruction branch codes and a condition symbol supplied by the main program. A microinstruction which experiences a match in both the address and branch code fields will be the next microinstruction to gain activity. The initialization of a microinstruction sequence by a machine instruction is implemented through an associative match between the machine instruction's operation code and the address of the first microinstruction in the corresponding microinstruction sequence.

The operand fetch is performed by comparing the machine instruction's source (or destination) operand field storage state against all main program word addresses. When a register containing an operand is located through a match, it is tagged with an activity state which reflects the significance of the register's contents.

A departure from the associative addressing process will be made to accommodate machine instruction sequencing. Linear sequences of machine instructions will occupy contiguous main program designated rows, with activity state propagation directed between adjacent words, thus replacing the function of a program counter. Any program branching instructions will employ associative addressing in the manner described for operand fetching.

Cell Structure

Each cell is endowed with enough storage such that a single cell may be assigned to represent the register's activity state. Six activity states are employed (inactive/active microinstruction, active machine instruction, active source operand, active destination operand, inactive main program word), requiring three binary storage devices to implement the cell internal state.

Array Structure and Boundary Cell Interconnection

The CGPC array, with its field-defined boundary cell interconnections and communication paths with a generalized peripheral device, is as shown in Figure 3. Register fields are established for the network by a set of fixed boundary inputs (Ψ's) applied to the bottom boundary cells. Fixed inputs are applied to other boundary cells for other special purposes (such as to denote the location of an operand's least significant digit). In general, these fixed signals ripple through the array and remain in steady state.

We define vertically-going signal propagation in the array such that for any particular column (of cells), signals propagating upward (downward) will have only one source (i.e., cell initiating the signals) during a particular network cycle. Accordingly, upward (downward)-going interregister signals are functionally defined on a field basis as follows. Upward-going signal propagation in field-B and downward-going signal propagation in field-D are to be associated with microprogram addressing. Downward-going signal propagation in field-B and upward-going signal propagation in fields-E and -F are to be associated with operand addressing; an exception being that simultaneously defined upward-going signal propagation in fields-D, -E, and -F is specified in association with 2-operand arithmetic or logical data operations. Downward-going signal propagation in field-E is associated with the micro operation code output from the active microinstruction, and downward-going signal propagation in field-F is associated with the peripheral order code output from the active microinstruction. Upward-going signal propagation in field-A is associated with linear instruction sequencing in the main program. Upward- and downward-going signal propagation in field-C is associated with main program-generated condition signals which provide branch code pointers for the microprogram.

Boundary cell interconnections are specified to support the interfield signaling. The array's $\hat{D}1$ output carries microprogram addressing information which is supplied to the microprogram base address field (field B) via an external connection to the array's B2 input. The $\hat{E}1$ output supplies micro order information to all registers via the external MO bus. Data transfers are conducted via upward-going signaling in fields D, E, and F. Loop-around is provided for these fields via the $\hat{D}2$ to D2, $\hat{E}2$ to E2, and $\hat{F}2$ to F2 connections. Communication with the peripheral device transpires via array outputs $\hat{D}2$, $\hat{E}2$, and $\hat{F}2$, and inputs A1, B1, C1, D1, E1, and F1. Operand addresses appear at array outputs $\hat{E}2$ and $\hat{F}2$, which are ORed and presented to input B1.

Microinstructions

A microinstruction set has been specified to implement machine instructions typical of small scale computers of the von Neumann type (single instruction stream, single data stream). A detailed listing of the microinstruction set has been omitted in the interest of brevity. The parallelism of the CGPC organization was not explicitly exploited, although to do so would require only simple modification to the basic cell. This decision was made to conform with the design goal of providing a machine which could be programmed in the conventional manner, but which is based on a purely cellular structure. The implementation of parallel (i.e., single instruction stream, multiple data stream) instructions is the subject of ongoing study with respect to this particular architecture.

A clear strength of this processor is that the microprogram is software defined, with all operation codes similarly software defined. The user could tailor such a system to his specific needs, as well as utilize dynamic microprogramming techniques.

Instruction Set Level Description

As an example, the ISP-level description of a typical microcoded implementation of the cellular computer is presented in the appendix. It should be noted that this ISP is typical of common small-scale, general-purpose computers of the von Neumann type. The programmer is unaware that there is no localized ALU, nor that a fetch does not cause data movement, but rather the assignment of an activity state. The ISP description includes instructions demanding the transfer of data to and from a peripheral device. Although the peripheral will not be described further, it is recognized that the peripheral must be able to initiate a program interrupt, as well as bootstrap the processor in order to load both the main program and the microprogram.

Operation

An example illustrating program execution given the ISP example is presented. Figure 4 represents a sample program segment featuring a series of instructions to be executed linearly (i.e., without program branching).

In the time frame shown, the machine instruction with address "MS4" is the active instruction, calling for the 2's complement sum of operands at locations "BETA" and "ALPHA" to be stored in location "BETA." Both operand locations have been fetched (i.e., located and tagged with the proper activity states.) Let us examine how the network might have reached such a configuration.

The program was loaded in queue fashion from the top-array boundary such that instructions which are to be executed linearly are physically adjacent. The propagation of activity among such instructions is upward (thus instruction with address "MS3" was executed previous to the current instruction).

Immediately after register "MS4" received 1_s-activity an operand-fetch microprogram subroutine was initiated. The contents of field-E were compared against the field-B contents of all CGPC registers. The register with address "BETA" was associatively located, and tagged with activity state 2_s (destination operand). The register with address "ALPHA" was similarly located and tagged with activity state 3_s (source operand). The control microprogram now examines register MS4's field-D, finds the "ADD" operation code, initiating the microroutine that accumulates the source operand data into the data field(s) of the destination operand.

The microprogram must then reset field-A of registers "BETA" and "ALPHA" to ϕ_s (inactive main program word) and initiate a "linear instruction fetch," that is, command that the 1_s-activity propagate "upward" one register.

This general description of CGPC operation applies to all nonbranching user's instructions. A microroutine to implement 0-, 1-, or 2-operand fetches is presented to illustrate microprogram sequencing.

At most, one microinstruction is active during any network cycle, and no microinstruction is active during two successive network cycles.

With reference to the sample microprogram segment presented in Figure 5, the microinstruction with address "OPFCH" is shown as 1_m-active (active microinstruction), indicating that the next address specified by the previous active microinstruction was "OPFCH," and that the null branch condition (ϕ) was presented to the microstore. This is the first instruction of the (multi-) operand fetch microroutine. The field-F symbol is ϕ, indicating that no order is directed to a peripheral device. Micro operation "TS2" directs the 1_s-active register to output its field-C symbol (operand format), which is made available to all field-C cells in the array. The 1_s-active field-C content is therefore available to the three microstore registers with address "OTST." As the next address is "OTST" (microstore register "OPFCH"), the microstore register with address "OTST" and branch code symbol (field-C) identical to the 1_s-active register's field-C content will become 1_m-active during the next network cycle (and the microstore register "OPFCH" will become ϕ_m-active [inactive microinstruction] during the next network cycle). Let us assume that the "ADD" operation is associated with the 1_s-active register, as is the case in Figure 4. The 1_s-active register field-C symbol is "3," thus the microinstruction with address

"OTST" and branch code "3" will become 1_m-active. Once it does, micro-operation "FH3" directs the 1_s-active register to output (upward only) its stored field-E symbol, which, through external interconnection will be made available to all CGPC field-B's. Simultaneously, micro-operation "FH3" directs all ϕ_s-active registers to attempt to match its field-B symbol with the 1_s-active register's field-E symbol, now being received as a downward-going field-B input. If a ϕ_s-active register detects a match, the activity of that register will be set to "2_s" at the beginning of the next network cycle. Meanwhile, the microstore register with symbolic address "OD" is selected by the next address symbol as the next 1_m-active register. During the following network cycle, micro-operation "FH2" directs the location and "3_s-active" tagging of the register with address matching the 1_s-active register's field-F contents.

Microstore register with address "OA" is selected as 1_m-active for the next network cycle. Micro-operation "FH1" commands the 1_s-active register to output (downward) its field-D state (corresponding to the operation code). The 1_m-active register has a null (symbol ϕ) next address, but is commanded by the micro-operation to pass its field-D input as its next address output. This has the effect of using the 1_s-active register's operation code as the next base address provided to the microstore. In our example, the microroutine with address "ADD" would be chosen, and thus the addition routine is initiated. The complete set of microroutines and descriptive examples of their operation may be found in Reference 7.

Implementation

Using synthesis techniques developed exclusively for use with cellular structures,[3,7] a gate-level realization of the basic cell has been determined. Additionally, the stable operation of the CGPC has been verified with maximum clock rates defined. An implication of the intercell communication scheme adopted is that many gate propagation delays are experienced by certain signals internal to the CGPC. Additionally, the minimum network cycle time which will support stable CGPC operation increases as a function of the number of cells in the array. Accordingly, two cell implementations have been proposed, as depicted in Figure 6. The first (Figure 6A) is as initially proposed, while the second (Figure 6B) features an intercell busing scheme to implement certain upward- and downward-going signaling. The advantage of the busing scheme is to greatly reduce the minimum network cycle time; given an M x N cell array, implemented with gates having a typical delay of Δg, the typical minimum network cycle is approximately

$$(18N + 10)\Delta g + \Delta p$$

where Δp represents the combined signal propagation delays on the vertical bus. Given the 4170 x 17 cell array used as an example and Δg = 30 ns, the minimum network cycle is approximately 9 μs.

It is envisioned that it is within the capabilities of near term LSI to implement a vertical "cell slice" chip version of the CGPC. The limiting factor is most likely to be leads, as the number of leads per cell slice is 8C + 34, where C is the number of cells per slice.

Conclusions

A 2-dimensional cellular general purpose stored program computer has been specified. This particular cellular computer is distinguished from previously proposed locally controlled cellular computers as its cellular structure is "hidden" from the user. Although the ISP description is similar to that of a small-scale computer of the von Neumann type, the architecture of the cellular computer does not feature physically isolated functional units to implement memory, ALU, or control. The hardware associated with each of the computer's functional units is uniformly distributed among all of the array's identical cells. The computer is software microprogrammed, which lends to dynamic microprogramming techniques, as well as establishing a strength as a machine simulator.

The CGPC structure does not favor a large associative memory space due primarily to the effect of intercell signaling on the maximum execution rate. Auxiliary random access memory must be interfaced to the CGPC at the peripheral interface.

Future applications under study include use of the CGPC as a small laboratory processor, as an intelligent memory interface, and as an imbedded microprocessor in a larger processor.

Appendix - ISP Description

This appendix provides the instruction set processor description of a 4166 x 17 cell, CGPC, given the microprogram presented in Table 1.

Memory - Pc/Mp State

Memory-processor\ $M[1:10106_8]<1:17>_8$	Each 8-state element represents a cell
Row-register\ $ROWi<1:17>_8 := M[i]<1:17>_8$	Each row may be considered as both storage and processor
Row-i activity state\ $AOi_8 := ROWi<1>_8$	Tag denoting the functional activity of Row-i
Row-i address\ $BOi\ <1:4>_8 := ROWi<2:5>_8$	Associative address
Inactive main storage row\ $\phi MS<1:17>_8 := ((AOi=0_8) \rightarrow (\phi MS := ROWi))$	Activity state tag 0_8
Active main storage instruction\ $1MS<1:17>_8 := ((AOi=1_8) \rightarrow (1MS := ROWi))$	Activity state tag 1_8
Active destination operand\ $2MS<1:17>_8 := ((AOi=2_8) \rightarrow (2MS := ROWi))$	Activity state tag 2_8

Active source operand\
$3MS<1:17>_8$:= ((AOi=3_8) → (3MS := ROWi))

Activity state tag 3_8

Destination operand data field\
$DDF<1:11>_8$:= $2MS<7:17>_8$

Source operand data field\
$SDF<1:11>_8$:= $3MS<7:17>_8$

Source operand condition code field\
SCF_8 := $3MS<6>_8$

Instruction Format

Operation code\
$OP<1:3>_8$:= $1MS<7:9>_8$

Destination operand address\
$DAD<1:4>_8$:= $1MS<10:13>_8$

Source operand address\
$SAD<1:4>_8$:= $1MS<14:17>_8$

Operand Format\
IT_8 := $1MS<6>_8$

Identifies the instruction as 0-, 1-, or 2-operand

Address Calculation Process

BOi := ((IT=2_8)→BOi := DAD; AOi←2_8); (IT=3_8)→(BOi := DAD; AOi←2_8; next BOi := SAD; AOi←3_8))

Instruction Interpretation Process

Interpreter := (Run → instruction fetch, next execute instruction, next interpreter)

Run is true when there is an active microinstruction in the microprogram. Interpretation cycle loop.

Instruction fetch := (AOi=1_8)→(AO(i+1)←1_8; AOi←0_8)

Instruction Set and Instruction Execution Process

Execute instruction := (

ADD(OP := 01)→(DDF←DDF + SDF); Add. OP code is arbitrarily assigned.

SUB(OP := 02)→(DDF←DDF - SDF); Subtract (negative values expressed in 2's complement.

AND(OP := 03)→(DDF←DDF ∧ SDF); Logical AND.

OR(OP := 04)→(DDF←DDF ∨ SDF); Logical OR.

XOR(OP := 05)→(DDF←DDF ⊕ SDF); Logical exclusive OR.

XFER(OP := 06)→(DDF←SDF); Move data from SDF to DDF.

INCR(OP := 07)→(DDF←↑DDF); Add 1 to stored value.

DECR(OP := 10)→(DDF←↓DDF); Subtract 1 from stored value.

COMP(OP := 11)→(DDF←-DDF); 2's complement of stored value.

CLR(OP := 12)→(DDF←0); Clear.

SHL(OP := 13)→(DDF←{DDF); Shift left logical.

SHR(OP := 14)→(DDF←}DDF); Shift right logical.

GOTO(OP := 15)→(AOi=2_8→AOi←1_8; AOj=1_8→AOj←0_8; next execute instruction); Unconditional program branch.

SKNZ(OP := 16)→(SCF←SDF; next SDF= 2_8→AOi=2_8→AOi←1_8; AOj=1_8→AOj←0_8; next execute instruction); Branch on not zero.

TSN(OP := 17)→(SCF←SDF; next SDF=1_8→AOi=2_8→AOi←1_8; AOj=1_8→AOj←0_8; next execute instruction); Branch on negative.

TSZ(OP := 20)→(SCF←SDF; next SDF=2_8→AOi=2_8→AOi←1_8; AOj=1_8→AOj←0_8; next execute instruction; Branch on zero.

TSP(OP := 21)→(SCF←SDF; next SDF=3_8→AOi=2_8→AOi←1_8; AOj=1_8→AOj←0_8; next execute instruction); Branch on positive.

READ(OP := 22)→(DDF←(data from peripheral));

WRITE(OP := 23)→((peripheral)←SDF);

JSR(OP := 24)→((2MS<10:13>$_8$←1MS<2:5>$_8$; next AOi=2_8→AO(i+1)←1_8; AOi←0_8, AOj=1_8→AOj←0_8; next execute instruction. Jump to subroutine.

RTS(OP := 25)→((ROWi<2:5>$_8$ = 2MS<10:13>$_8$)→AO(i+1)←1_8; AOj=1_8→AOj←0_8; next execute instruction); Return from subroutine.

START(OP := 26)→(no operation); Initial program instruction.

END(OP := 27)→(Run←0)) Execution termination.

References

1. R. C. Minnick, "Survey of Microcellular Research," J. ACM, Vol. 14, pp. 203-241, April, 1967.

2. A. Avizienis and C. Tung, "A Universal Arithmetic Building Element and Design Methods of Arithmetic Processors," IEEE Trans. Computers, Vol. C-19, pp. 733-745, August, 1970.

3. R. G. Cornell and H. C. Torng, "Cellular Arrays and Arithmetic Units: A Synthesis Procedure," Proc. PIB Symposium on Computers and Automata, pp. 527-538, 1971.

4. J. H. Holland, "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously," Proc. EJCC, pp. 108-113, 1959.

5. J. O. Campeau, "The Block-Oriented Computer," IEEE Trans. Computers, Vol. C-18, pp. 706-718, August, 1969.

6. J. N. Sturman, "An Iteratively Structured General-Purpose Digital Computer," IEEE Trans. Computers, Vol. C-17, pp. 2-9, January, 1968.

7. R. G. Cornell, "The Synthesis and Evaluation of Cellular Computers," Ph.D. Dissertation, Cornell University, Ithaca, New York, 1973.

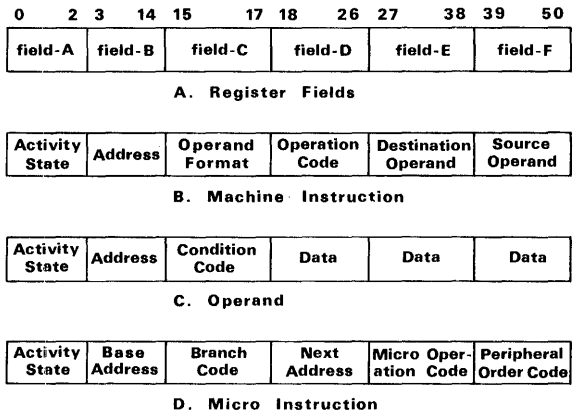


Fig. 2 **Register Fields and Word Partitions**

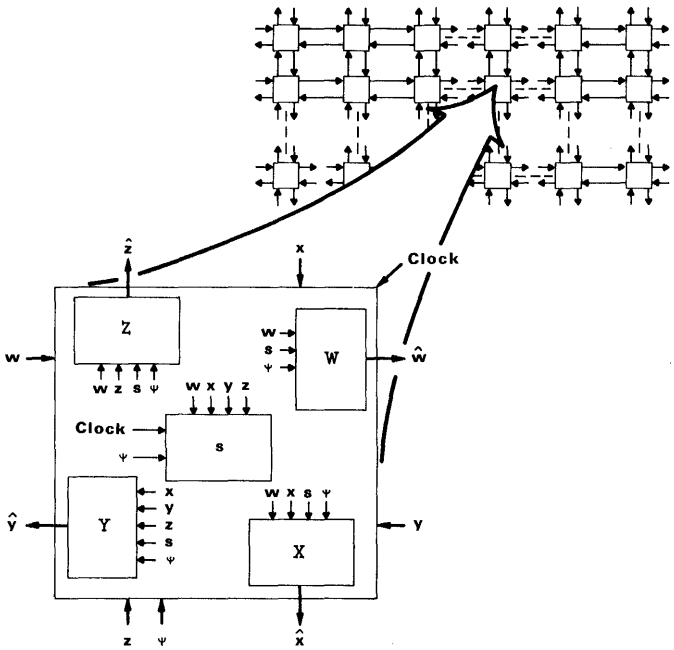


Fig. 1 **BASIC CELL**

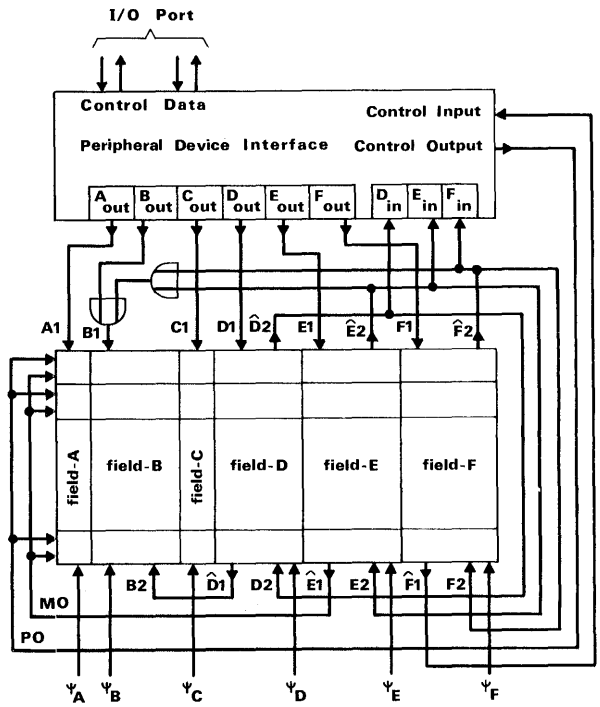


Fig. 3 **External Interconnections - Field Notation**

2_s	BETA	ϕ	5	4	2
ϕ_s	GAMMA	ϕ	7	6	5
3_s	ALPHA	ϕ	3	2	4
. . .	. . .	. . .	. . .	. . .	. . .
ϕ_s	MS5	2	INCR	BETA	ϕ
1_s	MS4	3	ADD	BETA	ALPHA
ϕ_s	MS3	3	OR	GAMMA	ALPHA
. . .	. . .	. . .	. . .	. . .	. . .
ϕ_s	MS1	1	START	ϕ	ϕ
Field-A	Field-B	Field-C	Field-D	Field-E	Field-F

Fig. 4 **Sample Program Segment -- Linear Instruction Sequence**

1_m	OPFCH	ø	OTST	TS2	ø
$ø_m$	OTST	1	ø	FH1	ø
$ø_m$	OTST	2	OA	FH3	ø
$ø_m$	OTST	3	OD	FH3	ø
$ø_m$	OD	ø	OA	FH2	ø
$ø_m$	OA	ø	ø	FH1	ø
Fld-A	Fld-B	Fld-C	Fld-D	Fld-E	Fld-F

Fig. 5 **Operand Fetch Microroutine**

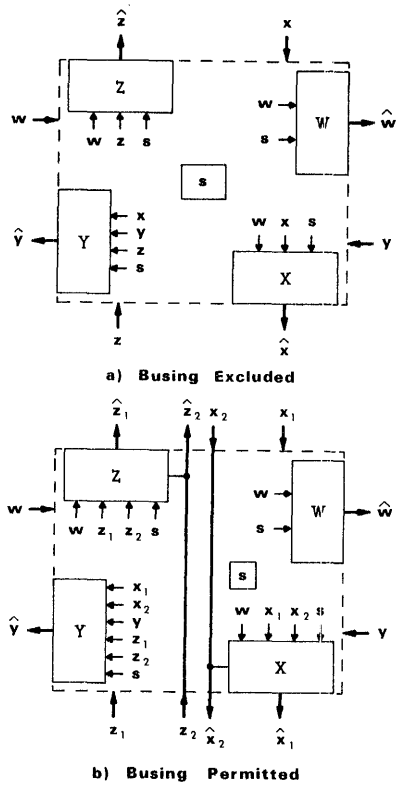


Fig. 6 **Cell Implementations**

A MACHINE-ORIENTED RESOURCE MANAGEMENT ARCHITECTURE

Barry C. Goldstein
Thomas W. Scrutchin

International Business Machines Corporation
Poughkeepsie, New York

SUMMARY

An architecture for resource management in a computer system is presented. The architecture is a subset of an APL-like higher level language machine architecture. The approach provides for a natural division of function between software and microcode/hardware. Any object in the system can be managed as a resource, and the fact that an object is managed can be transparent to the program using the object. Finally the resource management concepts are related to the current resource management problems of effective third-party control of resources, intelligent work scheduling, and deadlock resolutions.

Resources and Use Protocols

In defining an architecture for resource management in a computer system, a reasonable first question is what are the resources to be allocated? There are, of course, data files and records, and communication files, lines, and devices. But there are others. A casual look at the uses of OS/360 ENQ/DEQ[1] shows that programs make heavy use of synchronization facilities, which must be at the least a subset of our allocation facilities. While it is clear that those programs would not run, or at least would not run well, without such serialization, it is not clear what resources these ENQ names protect. Generally, there is some object, perhaps a conceptual or temporal object, for which the facility serves as a partial management facility. The inability to predict what the resources would be, even in a current operating system, makes unlikely any success in listing all the resources in the hypothetical architecture.

A new approach is therefore in order. First there is an object; then the use of the object by various entities in the system must be managed. At this point the previously ordinary object becomes a resource. Which objects are eligible to become resources? In particular, any object which can require some protocol for use, which means, in general, any object. Having determined that the existence of a use protocol makes an object a resource, we proceed to investigate more of the current resource management problems, trying at the same time to crystallize the objectives for a proposed architecture. The first problem to become apparent is that the management of resources in current systems is largely accomplished by convention. If the programmer of a function follows the allocation rules for the resource he is using, things generally work. If he fails to do so, things go awry. At the least, programs produced erroneous results. In many uses, however, the integrity of the resource is compromised. Often the violator of the resource is not responsible for the resource. So it is left to some slaving gnome to set right what the ignorant clod has destroyed. The conclusion reached is that those objects which are resources must be susceptible to effective third-party control, and not that we require smarter or better informed clods. The use protocols must not depend on programmer convention for effect.

Objectives

With this general concept of how resources should be managed, it was decided to constrain the design with a set of objectives. Some follow directly from what has been said, others merely appear to be good ideas.

- It should be possible to control or manage any object in the system with a use protocol.
- It should be possible to use an arbitrarily complex program as a use protocol.
- There should be machine-supported use protocols.
- The invocation of a use protocol should occur directly through the addressing structure of the machine, whether the use protocol is implemented directly in hardware/microcode or whether implemented in software.
- The programmer need not be aware if the objects are controlled or not. This transparency will allow effective third-party control of objects.
- The programmer must be able to request early invocation of a use protocol and have the effects endure. He should be able to explicitly declare his intention to use an object in a particular fashion.

Architectural Environment

For a variety of reasons--not the least of which is that the authors are devoted APL bigots--it was decided that the appropriate architecture for these concepts would be one in which the software/firmware interface of the machine is a higher-level

language. (The model was APL, but we will refrain from talking about that specific language and bring forth the particular aspects of the machine architecture which we feel are important to the proper application of our resource management concepts.)

We were specifically avoiding the kind of layered architecture where distinct lines were drawn between the programming language, the operating system, and the machine. It was felt that such an approach would force a description of resource management in at least three separate languages: the programming language, the operating system command language, and the machine language. With our machine, the programming language, command language, and machine language are one, leading to enormous economies of description.

Because we have a single-level architecture with the command language facilities directly available, the existence of the following features is assumed to make further discussion meaningful:

Multiprocessing. A particular instruction stream (i-stream) must have the ability to create asynchronous i-streams.[2] They must be structured to support implicit destruction of i-streams when a superior i-stream is destroyed, and to provide a capability to establish sets of i-streams to which resource usage can be constrained.

Objects. Another factor leading to the formation of the single-level architecture is the concept of objects. It is important that the system understand what the objects are.[3,4] All objects and subobjects, or elements of objects, are named in the machine language. For example, a vector called *A* is an object in the language. *A*[1] is an element of that object. The elements and objects can be controlled with separate use protocols. Considerable help from the machine is required in the area of control of objects with use protocols. This would be a near impossibility with an IBM System/360-like machine,[5] with a single continuum of addresses and, as far as the machine is concerned, almost no unique objects.

Namespaces. A namespace, similar to an APL workspace, is the externalization of the symbol resolution mechanism.[6] A local namespace is associated with each i-stream. Through the name space, the i-stream has access to other namespaces. Not every i-stream has access to every namespace. An i-stream can address directly any object in any namespace to which it has access by qualifying the name of the object. It is through the namespace that objects are controlled with use protocols. A dyadic operator, *CONTROL*, of the form:

A CONTROL B (1)

is introduced wherein the object, whose symbol is *A*, is controlled by the use protocol whose symbol is *B*. The symbol *A* may be resolved to any object in any namespace. Similarly, the symbol *B* may be resolved to any of the protocols in any namespace (such as an arbitrarily complex program or machine *LOCK*). More precisely, *B* represents not only the '*USING*' protocol but the '*RELEASE*' protocol as well, an indication of what should happen when the object is to be unallocated.

It is important to understand that the capability to address an object is given to an i-stream independently of the use protocols.[4,7] The use protocols specify additional constraints on the use of the object (such as synchronization constraints).

Locking. There is a machine lock facility for synchronization between i-streams, which will be the most primitive facility for controlling the use of resources. It is similar to the ENQ/DEQ facility of OS/360, which permits only two locked states, exclusive or shared. For complex objects such as arrays, structures, and "files," the simple exclusive shared approach is insufficient, so the locking facility will allow more locked states. For an individual atomic element of a complex object, a shared/exclusive lock is sufficient; but when the elements are being locked, there must be a control mechanism for the entire object. We have four locked states, or intents, against the object. The first is object exclusive. When the lock is held in an object exclusive state, it may not be obtained in any other state by another i-stream, and changes can be made to the elements of the object with the guarantee that every element of the object will be available while the lock is held. The object exclusive state is also used to prevent i-streams which are only reading the objects from seeing the object in an inconsistent state. The second state is object shared. While the lock is held in this state, no changes may be made by any i-stream. The third intent is element exclusive. The holder of the lock in this state intends to change individual elements, but has no stronger consistency requirement. This lock state is generally used with other locks on the elements within the object. The remaining state is element shared. The holder of the lock in this state does not intend to change the object and is looking at the elements in a manner which will allow other i-streams to change individual elements of the object other than the one(s) he is currently looking at.

When viewing the object, we find that only certain states or intents are compatible. Object exclusive is not compatible with any other intent. Object share is compatible only with object share and element share. Element exclusive is compatible with element exclusive and element share. Element share is compatible with object share, element exclusive, and element share.

Locking presents the simplest use protocol for an object. It also represents the highest potential performance, because it is a machine function. In the interpretation of a statement referencing a lock controlled

object (an object whose use protocol is the obtaining of a lock), the use protocol can be satisfied without trapping out to software.

Allocation Scopes

Up to this point locking has been discussed in terms of one i-stream holding a lock, thus preventing others from having access to the object. But if we look at the data set allocation facilities in OS/360, we find that they do not work in such a straightforward manner.[8,9] After a data set is allocated to a job, all the tasks of the job have access to the data set. While this is a rigid structure with many failings (for example, there are no facilities other than programmer convention to enforce compatible usage of the data set among the tasks of a job), it does indicate that there is a need for a more general facility.

Looking at the facility provided by OS in more abstract terms, we see that what is provided is the facility to declare a set of i-streams as a unit. Resources can be allocated to this set with a particular intent, which primarily limits the degree of competition to be allowed with i-streams (and sets of i-streams) outside the original set. There are also interactions and constraints placed on interactions among i-streams within the set. To see this we must look at the effect of allocating a resource to a set with a particular intent.

If a resource is allocated to a set with an object exclusive intent, the i-streams within the set can compete without concern for the i-streams outside the set, because it is guaranteed that no i-stream outside can hold the lock with any intent. This is the only simple case. In every other case, it is possible for i-streams outside the set to hold the resource with at least one intent. As long as the inner i-streams compete within the bounds of the original allocation, there are no problems. But consider a case where the resource is allocated to the set with an element exclusive intent and an operation is performed which requires that the object be held object exclusive. One might be inclined to disallow the case, declaring it to be an error. This would be a mistake.

Consider a program with parameters *A* and *B*. The program requires *A* with an element exclusive intent, and *B* with an object exclusive intent, while still holding *A*. We are in trouble if the program is called with the same object in parameters *A* and *B*. Hence, rather than disallow this, we will allow an i-stream within a set to strengthen the allocation to the set at any time. This may cause competition with i-streams outside the set which hold the resource with an intent incompatible with the strengthened intent. An attempt to strengthen an intent may have to wait.

While we allow sets of i-streams to be explicitly declared, entered, and left, we also have some implicit sets which closely approximate OS data set allocation. The implicit sets come from the order of creation of dependent i-streams. All of the dependent i-streams to any i-stream form a set to which resources may be allocated. There will be a standard facility in the language for naming this set and inquiring whether an i-stream is in the dependent i-stream set of another.

The notion of explicitly declaring a set of i-streams and allowing i-streams to enter and leave the set is straightforward, but creates some minor problems. An i-stream may enter a set to which an object is allocated object exclusive. It can then allocate the resource to itself with any intent, presuming of course that no other i-streams in the set have allocated the resource. Suppose that the i-stream then leaves the set. Without some action, we would be in a state where a set of i-streams and an i-stream not in the set hold a resource with incompatible intents. This must not occur, so an error is generated on the attempt to leave the set.

The following operators are introduced to define sets of i-streams:

$$B \ \ ENTER \ \ A \tag{2}$$

$$B \ \ LEAVE \ \ A \tag{3}$$

In (2) the dyadic function *ENTER* places the i-stream *B* into the set *A*. Likewise in (3), the function *LEAVE* removes the i-stream *B* from the set *A*. Each function has a monadic form:

$$ENTER \ \ A \tag{4}$$

$$LEAVE \ \ A \tag{5}$$

In these forms, the i-stream is not specified. The i-stream which enters or leaves the set *A* is the i-stream which invokes the function.

Resource Management Transparency

One objective is that a programmer should not have to deal with, or be aware of, the allocation of these objects. To do so, we must look into the structure of the language to see how these allocation or locking intents may be determined. Let *A* be a vector which is controlled by a lock, and in which each element is controlled by a lock. We will see that the lock intent can be determined by the context in which the symbol *A* is used.

1. $A \leftarrow B$ This statement implies an atomic modification to the entire object *A*. The interpreter indicates, in resolving the symbol *A*, that it is required in an object exclusive manner.

2. *A*[1]←*C* The intent is clearly that *A* must be locked with an element exclusive intent and that the element *A*[1] must be locked with an object exclusive intent.

3. *B*←*A* The required intent on *A* is object shared.

4. *C*←*A*[1] The required intents are element shared on *A* and object shared on *A*[1].

These are simple examples, but they form a base for determining the intents in complex cases. We propose that the duration for holding these locks be on a statement basis. All the required locks which are specified via the *CONTROL* dyadic are obtained as it becomes apparent that they are required. Upon statement completion, they are released. It would be useful, before beginning interpretation of the statement, to have some idea of the strongest intent to be expressed on an object. This facility would also be useful in dealing with deadlocks. There are cases where APL cannot be completely compiled.

One such case is where a single statement may generate the name of an object it intends to use. Without executing the statement, we do not know what resources are required. When it is apparent that we cannot "compile" the resource requirements for a statement, we will then revert to the interpretive scheme. We can do a partial compilation of those items which are resources. The degree to which we do this compilation and look-ahead will to a large measure be determined by the degree of parallelism in the machine. (Resource requirement extraction will also be treated in the section Job Scheduling and Deadlocks.)

Explicit Resource Management Functions

We have postulated a system in which resource allocation occurs implicitly over the granularity of a statement. There are times when the programmer needs the allocation over a set of statements. Consider the example of the *SORT* program in Figure 1 (written in APLGOL, see [10]). Here the programmer cannot afford to hold *A* with element exclusive at each statement and release *A* upon completion of each statement. *A* must be held object exclusive from the beginning of statement [2] through statement [11].

```
    ∇ SORT A;I;TEMP
[1]   I←1
[2]   DO WHILE I<⍴A
[3]   IF A[I]>A[I+1] THEN
[4]   DO
[5]   TEMP←A[I]
[6]   A[I]←A[I+1]
[7]   A[I+1]←TEMP
[8]   I←I-1
[9]   END
[10]  ELSE I←I+1
[11]  END
    ∇
```

Figure 1

In order to enlarge the extent over which the allocation will persist, the following APL operators are introduced:

A USING B (6)

A RELEASE B (7)

where in statements (6) and (7) *A* can be any set of objects, and *B* is the required intent. The *USING* dyadic informs the symbol resolver that it should invoke the use protocol associated with the object(s) *A*. If the use protocol is a software function, then the intent *B* is a parameter to that function. Furthermore, the release protocol (for example, the *UNLOCK*) will not be implicitly invoked upon completion of (6) but, rather, will not be released unless explicitly done so with the *RELEASE* operator. (See Figure 2.)

Now the integrity of the *SORT* program can be guaranteed, in Figure 2, so that, upon completion of the *USING*, it will hold *A* object exclusive until the *RELEASE* is issued.

```
    ∇ SORT A;I;TEMP
[1]   I←1
[2]   A USING OBJ_EXC
[3]   DO WHILE I<⍴A
[4]   IF A[I]>A[I+1] THEN
[5]   DO
[6]   TEMP←A[I]
[7]   A[I]←A[I+1]
[8]   A[I+1]←TEMP
[9]   I←I-1
[10]  END
[11]  ELSE I←I+1
[12]  END
[13]  RELEASE A
    ∇
```

Figure 2

Other Use Protocols

Inasmuch as we have shown how use protocols can, with the multiple intent strategy, be used to issue a machine *LOCK*, they can generally be used to invoke any arbitrarily complex program through a trap in the microcode/hardware (established with the *CONTROL* dyadic). In essence, the firmware escapes to the "software" routine upon resolution of the symbol. It should be possible that the object in question may not physically exist at the time of symbol resolution, but may be created by the invocation of the use protocol.

Programmed use protocols go beyond the simple synchronization mechanisms of locking. The administrator of a resource may wish to control many aspects concerning its use. The use protocol might check certain features of the execution environment before granting access to the object. A clear example of the usefulness of this concept is when we consider attaching security protocols to specific objects.[4,11] The use protocol may also have side effects on the environment in which the program is executing. This

allows a controller of a resource to actively insure that certain environmental conditions exist.

The inclusion of programmed use protocols reflects the designer's decision on an appropriate division between software and microcode. The locking protocols are useful when the objectives for controlling objects are bounded by the well understood problem of synchronization. There are considerably broader objectives in an installation, which should not be included in the machine. To accomplish these objectives, easily accessed traps into software should be provided.

Job Scheduling and Deadlock Resolution

We discussed earlier the problems involved in early determination of resource usage on a statement basis. Another environment in which these problems become apparent is the batch or background environment, when we wish to do intelligent scheduling of jobs. To do so, we must have an idea of the resources to be used. One approach to the determination of the requested resources would be to invent a job control language which explicitly declares the resources. This approach, however, violates two of our objectives. It would require more than a single language interface to the system, and it would require that the programmer be aware of precisely what the resources are.

The only approach consistent with our objectives is that we extract the resources from the programs to be run. As pointed out earlier, at best this process is incomplete, and it is often incorrect. But our intent in this facility is optimization. In those cases where we make mistakes, we should be able to correct them. In programs which hide resources, we will simply not do as good a job at scheduling.

An important point to be noted is the length to which we must go to extract resource requirements. In addition to considering the initial program and the called programs, we must also determine the resource requirements of the software use protocols for any complex resources. This would be a time-consuming task with tests for recursion and so on. It might well be that, in the implementation of a heuristic scheduler, we might choose a more limited approach to resource extraction in order to get the best overall performance.

The partial compilation can also be used in the prevention of the deadlocks; claim sets can be produced covering the resource requirements of the program.[8,12,13] These claim sets may be ordered sets. That is, the prevention algorithm could also be sensitive to the order, in disjoint subsets, in which the resources are needed as derived from the compilation. Operators such as *CLAIM* and *FREE* could be introduced into the language in order to enable the programmer to surface these claims to the system (thereby reducing the possibility of having the system overestimate his needs).

Conclusion

A major advantage of this resource management architecture over current systems is the single mechanism for the allocation of all system resources. Generality is accomplished by providing a single language interface to the machine, by allowing any object to be managed as a resource, and by building resource management into the basic addressing mechanism of the machine. A basic resource management facility is provided through objects called locks. The operations on these objects are primitive facilities of the machine. They control shared access to objects, where the degree of sharing reflects the intent of the program using the object. This primitive facility cannot support all the actual intents against an object. Thus the controlling mechanism is extensible through software. The addressing mechanism can invoke a program during symbol resolution to enforce the limitation of access to the object.

A second advantage is that resource management can remain transparent to the user of the resource. Shared resources can be controlled by a third party, an administrator for the resource, independent of the actions of the user of the resource. An important feature of this architecture is the automatic predetermination of resource usage which is instrumental in the scheduling of large background work requests, in the resolution of deadlocks, and in the efficient accomplishment of overall resource management transparency.

REFERENCES

1. IBM System/360 Operating System: Supervisor and Data Management Services, IBM Form No. C28-6646-2, IBM Corp. (Nov. 1968).

2. R.B.Hake and D.R.Page, "Tasking in APL," IBM Technical Report TR 12.102 (IBM United Kingdom) (May 1972).

3. J. Gray et al., "The Control Structure of an Operating System," IBM Technical Report RC 3949 (July 1972).

4. W. Wulf et al., "HYDRA, the Kernel of a Multiprocessor Operating System," Comm. ACM 17, 6 (July 1974), 337.

5. IBM System/370 Principles of Operation, IBM Form No. GA22-7000-3, IBM Corp. (Jan. 1973).

6. IBM APL/360-OS and APL/360-DOS System Manual, IBM Form No. LY20-0678-0, IBM Corp. (1971).

7. J.B.Dennis and E.C.Van Horn, "Programming Semantics for Multiprogrammed Computations," Comm. ACM 9, 3 (Mar. 1966), 143.

8. B.C.Goldstein, "On the Resolution of Deadlocks," IBM Technical Report TR 00. 2176-1, Poughkeepsie, N.Y. (1973).

9. J.W.Havender, "Avoiding Deadlock in Multitasking Systems," IBM Systems Journal, No. 2, 1968, 74-84.

10. R.A.Kelley, "APLGOL, an Experimental Structured Programmed Language," IBM Journal of Research and Development, 17, 1 (Jan. 1973), 69-73.

11. P.B.Hansen, Operating System Principles, Prentice-Hall (1973).

12. R.C.Holt, "On Deadlock in Computer Systems," Technical Report No. 71-91, Cornell Univ., Ithaca, N.Y. (Jan. 1971).

13. A.N.Habermann, "Prevention of System Deadlocks," Comm. ACM 12, 7 (July 1969), 373-377, 385.

A DESIGN-ORIENTED COMPUTER ENGINEERING PROGRAM

M. E. Sloan
Department of Electrical Engineering
Michigan Technological University
Houghton, Michigan 49931

Summary

Two national committees issued definitive computer curricula in the 1960s. Since then digital technology has changed, high school graduates have different preparation, computer science and computer engineering curricula have proliferated, and design has become more important. This paper proposes a computer engineering program that will respond to these changes. The program is a spiral one, based on an introductory computer engineering program, and emphasizing design through each of a number of streams that can reflect local interests and resources. This paper discusses the main courses of a stream and the basic introductory course.

I. Introduction

During the latter half of the 1960s, two national committees developed model curricula for computer science and computer engineering. The ACM Curriculum Committee on Computer Education prepared a preliminary report in 1965 and issued a final report in March 1968.[1] Their 1968 model curriculum for computer science was their last major work in this area although they have subsequently developed programs in other areas such as information systems.[2] The COSINE Committee published reports from 1968 through 1972. Their model curriculum for an undergraduate computer engineering option in electrical engineering was published in January 1970.[3] They disbanded in 1972.

More recent efforts to develop curricula for computer science and computer engineering, as opposed to information systems and data processing, have been sporadic and have been largely confined to development of guidelines for a single course. The effort sponsored by the IEEE Computer Society to suggest modules for a computer architecture course is a noteworthy example. The problems of funding and coordination of a major study of educational programs are, of course, immense. We probably can not expect a new major effort in computer science and computer engineering curricula in the near future.

Nonetheless technology has changed very rapidly since ACM and the COSINE Committee made their major recommendations. Large-scale integrated circuits have had a major impact. Logic design has involved increasingly larger scale components. The growth of minicomputers and microprocessors has caused major changes in the design of logic systems. Computer networks are on the verge of making major changes in the delivery of computer and communications services. Other improvements in the speed, size, and reliability of components have affected computer architecture.

Other changes have occurred at the same time. Students entering undergraduate computer science and computer engineering programs are increasingly better prepared for computer science although less prepared for analog-oriented courses. Mathematics courses in the high schools have been almost completely revised in the last decade with the result that students have a much different preparation than they had in the past.[4] They have a much greater knowledge of the concepts of discrete mathematics needed for work with computers. Many of them have studied programming while in high school. In fact some universities have stopped offering introductory computer programming as a credit course since they found that nearly all their entering freshmen could program and that those who didn't could pick it up relatively easily from a self-study course. In addition, the pervasive influence of computers throughout the culture has given students some basic concepts that they lacked in the past. For example, many college freshmen own calculators, such as the HP-35, that use reverse Polish notation. Consequently, the concepts of stack operation and Polish notation are ones that they already understand at least in an elementary way. Computer science and computer engineering programs, however, have been slow to build on the increased competence of entering students. A major weakness of many university professors is to regard students as tabula rasa, blank tablets coming to the university to be written on there.

As a consequence of increased student and faculty interest in computer science and computer engineering, course offerings have proliferated rapidly.[5] In many computer science and electrical engineering departments, it is no longer reasonably possible for a student to take all the undergraduate computer courses, let alone to take many of the first-year graduate computer courses for which he may have the prerequisites. A large number of course offerings necessitates a new approach to the design of a program if students are to be able to select courses intelligently. Since it is unreasonable to expect most entering students to know whether they would prefer to design integrated circuits, to design with integrated circuits, or to design operating systems, there is an increased need for introductory courses that provide systematic overviews of large areas. The student can then decide on the basis of an initial sample of several areas that he would like to pursue. Such an approach allows the student more flexibility and gives him more responsibility for his education than does the approach taken by ACM and COSINE that basically provided one main path through the curriculum with some electives. The

new approach recognized that the output of an undergraduate educational system will be graduates headed for distinctly different professional careers. Some breadth is sacrificed in return for greater specificity.

Another recent development affecting computer engineering curricula perhaps to a greater extent than computer science curricula is the realization of the importance of design. The 1950s and the 1960s saw engineering curricula oriented toward theoretical science rather than toward engineering design. Laboratory courses, drafting, and machine shop fell by the wayside as engineering schools increased the amount of basic science in their undergraduate programs. The 1968 Goals Report of the American Society for Engineering Education sought to reverse this trend by placing an increased emphasis on engineering as design. [6] This emphasis has continued with ECPD requirements for accreditation of M. S. programs that require a substantial part of the program to be design or synthesis instead of analysis. Students seem to appreciate a return to emphasis on design and are flocking to newly introduced freshman design courses and senior projects. Design courses usually require a greater depth of knowledge in one or a few closely related areas at the expense (if time is held constant) of a broad background. It is not possible to educate an engineer who is equally capable of designing in all the areas of an undergraduate engineering program or even all those in one department.

The influences that we have discussed so far seem to point toward the development of a new undergraduate computer engineering program. This program should be one that builds on the increased competence of entering students. It should present an overview of the areas that are available to him. Subsequent courses will build on the overview course in a way that should not be regarded as repetitive but instead as an opportunity to reinforce and expand on earlier learnings in the style of Bruner's spiral curriculum. The program should emphasize analysis procedures that are consistent with current and emerging technologies even when older approaches are more mathematically tractable. Design should pervade the curriculum. The goal should be the education of a graduate who can function as a competent beginning designer in some area whether it be design of digital subsystems or computer architecture. The remainder of this paper will discuss one possible approach for an undergraduate computer engineering program to accomplish these objectives.

II. A Design-Oriented Computer Engineering Program

The program discussed here is intended to prepare design engineers in several areas of computer engineering. The number and selection of the areas to be included would depend largely on the size and resources of the school. Figure 1 shows the main streams of the design programs for three different areas of computer engineering. One area, electronics systems architecture, may more accurately be regarded as an interface area between electronics and computer engineering. The figure is intended to show only the main flow through the design program. Students in each stream would take several other courses both within and without computer engineering

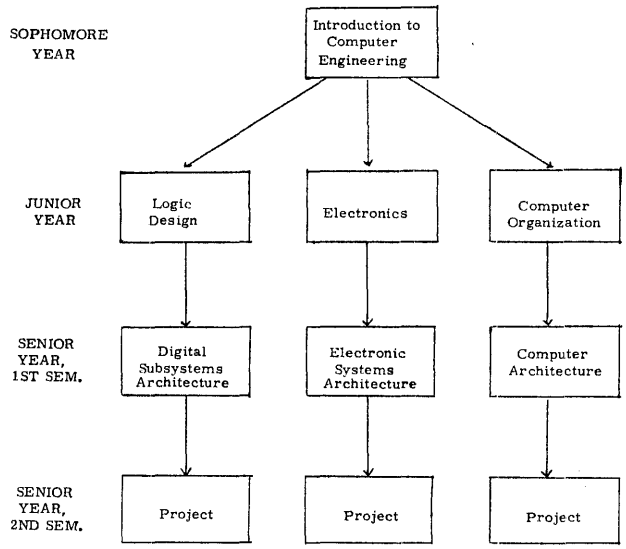


Fig. 1. Main Streams of a Design-Oriented Computer Engineering Program

In this program, all students begin at about the sophomore year with a course that introduces them to computer engineering. This course has a prerequisite only of introductory programming that the students may have satisfied in high school. Hence this course could be taught at the freshman level. Alternatively universities that have a high percentage of students transferring from community colleges may prefer to teach it in the first term of the junior year. The course is intended to provide an overview of all areas of computer engineering that do not require college-level prerequisites such as circuit analysis or electronics. The specific content of this course will be discussed in more detail in the next section. For the moment, the course can be considered to be based on some fairly broad introductory computer hardware and organization text such as Booth or Abrams and Stein. [7, 8] The exact content of this course would reflect the interests of the department. Some such introductory courses would have a strong computer science flavor and include discussion of operating systems, etc. while others would have a more traditionally electrical engineering flavor and include more logic and hardware material. The course should be accompanied by a laboratory in which students have an opportunity to design very simple projects, such as basic combinational and sequential logic circuits, and to write short assembly language programs.

After taking the introductory computer engineering course, students would branch into the stream most nearly following their interests. Each student would enter a year-long sequence that would strengthen the analysis and design tools in his area. Many students would choose two such sequences either to give them a more adequate basis for final choice of a stream

or to support interests that lie between two streams. The emphasis would be on constructing a strong theoretical framework for the area while at the same time providing some accompanying design experience. While design projects of the students in the introductory course would be mostly one-week projects, students in these intermediate courses would work on longer projects. By the end of the year, students should feel comfortable working on projects that would require a month or six weeks to complete. However, in most cases, students in one class would all be doing the same projects or perhaps choosing from a small set of well-defined projects.

The examples given for these year-long sequences are logic design, electronics, and computer organization. The logic design course would be the standard year-long logic sequence found in many electrical engineering departments. Many texts are available for this sequence. Because the introductory computer engineering course introduces logic concepts, the sequence could cover somewhat more material than is usually the case. Additional material might well cover modern logic technologies and design with integrated circuits. Current texts are still unfortunately weak in their coverage of IC design. It is much easier to cover standard minimization techniques that have nice algorithmic solutions than it is to treat minimization of logic circuits designed with integrated circuit chips subject to some cost criterion.

The electronics sequence would also be the standard year-long sequence in electronics offered by most electrical engineering departments. Again a wide variety of texts are available. Selection of a text that includes at least some material in digital electronics would be desirable to build on previous concepts and to prepare students for a course in electronic systems architecture.

The computer organization sequence would consist of at least one and preferably two courses. A model for the two-term sequence would be the two courses that COSINE proposed--Machine Structure and Machine Language Programming and Computer Organization.[3] Alternatively the courses could look more like the two ACM-recommended computer organization courses. Again, because some of the concepts had been presented in the first computer engineering course, more time could be spent on examining recent trends in organization. Several texts are available for this sequence also.

During the first term of the senior year, students in each stream would take an architecture course in their area. The aim would be to build on the fundamental concepts the student had obtained from previous courses with an emphasis on presenting examples of successful and unsuccessful designs. The student after completing the course should understand the design of several major projects, including the major choices made among alternatives.

The digital subsystems architecture course would look much like the course proposed by COSINE in November 1968.[9] The choice of text would probably reflect the interests of the instructor. Peatman and McCluskey and Gashwind are representatives of texts that cover some of the topics of the course.[10, 11] This course could easily be accompanied by student projects that correspond to parts of some of the larger projects studied.

The electronic systems architecture course would also reflect the instructor's interests. The balance between analog and digital designs studied could vary widely. No current text known to the author seems well suited for the course. Although there are a number of books on design with integrated circuits on the market, they tend to stress the principles of integrated circuit operation and the design of the chip instead of design with the chips. It is the latter that would be the basis of a course on electronics systems architecture. In the absence of a good text, the instructor will have to choose carefully the examples of designs for class study.

The computer architecture course would look much like many given now. A course that was based on analysis of a number of computer designs would probably use Bell and Newell possibly supplemented by computer manuals or descriptions of newer computer architectures, especially microprocessor architectures.[12] An alternative approach would be to use a more concept-oriented text and study design choices at the major component level, such as in Stone.[13]

The second half of the senior year for all streams would consist of the design, construction, and testing of a major project. Each student would work by himself or with at most one other student and would choose a design that would be in some way innovative. This at least would be the ideal but could be moderated as student capabilities and available resources would warrant.

The basic program outlined in this section could easily be modified to fit the needs of a particular department. Inclusion of more or fewer streams would be easy. The entire program could be moved one or more semesters earlier. The main advantage of the suggested level is the student's maturity. Ambitious students could take all courses in two streams or the intermediate courses in two or more streams. Students might wish to elect computer science courses relevant to their stream. However, the aim of the program should still be to present material in a spiral fashion. Each course or sequence would echo and build on earlier concepts. Each course or sequence would emphasize design as well as analysis. The student who completed a stream would have the ability and confidence to design projects in his area.

III. An Introductory Computer Engineering Course

Earlier the introductory computer engineering course that serves as a base for all streams in this program was described as a broad overview of those aspects of computer engineering that could be taught without a college-level prerequisite. This section discusses in more detail one introductory computer engineering course that has been taught at Michigan

Technological University to about 150 students in the last year. While the course is still being developed, its basic structure has taken shape.

The main purpose of the course is to serve as an overview of computer engineering. The course is taken by nearly all engineering students whether they originally intend to specialize in computer engineering or not. It is also taken as a computer hardware course by many computer science students.

The basic theme of the course is the hierarchy of levels of a computer as analyzed by Bell and Newell.[12] Because the course has no circuit analysis prerequisite and because it is not considered worthwhile to take enough time to present electronic circuits in any detail, the circuit level is omitted. The course involves learning modules for the five other levels and sublevels.

The combinational logic module introduces a few of the simpler theorems and postulates of switching algebra and the basic logic components. Classic minimization by Karnaugh maps and Quine-McCluskey methods is presented briefly but more emphasis is placed on semi-intuitive methods of minimization. Commonly available IC logic chips are discussed and choice of an appropriate logic technology is considered. Problems of interfacing with and minimization of IC circuits are also included. In the accompanying laboratory, students design a few simple combinational logic circuits, such as a decoder for a seven-segment display.

The sequential logic module is based on common flipflops. Analysis of flipflop circuits is treated fairly thoroughly. Synthesis of general sequential circuits is left to a later course. Instead, students learn to design simple standardized flipflop circuits such as several types of counters and registers. In the laboratory, they check their designs.

The register-transfer logic module pulls together the concepts from sequential and combinational logic sublevels. The general concept of register-transfer level analysis and design is presented and is illustrated by Digital Equipment Corporation's RTM modules. The basic circuit studied is an arithmetic unit. Students learn several types of algorithms for addition, multiplication, and division and their implementation for arithmetic in three radix-2 number systems. Again in the laboratory they can check one or more designs.

The next level studied is the programming level. Since all students have had programming experience in FORTRAN, this course contrasts higher level languages with assembly language. Students learn to write simple machine and assembly language programs for the department's PDP-8. The basic ideas of assemblers, compilers, and interpreters are presented.

The final level studied is the PMS or computer systems level. Memories, processors, and input/output devices are covered. The unit on memory introduces programmable ROMs and PLAs for comparison with random logic. The input/output unit stresses programmed vs. interrupt input/output handling. Basic organization types at this level are studied and are illustrated by the architecture of a HP-35, a PDP-8, and a PDP-11, and the IBM 360/370. Finally the desirability of computer networks is studied and examples of simple network configurations are examined.

IV. Conclusions

The changes in computer and digital systems technologies, the competences of entering undergraduate students, the proliferation of computer courses, and the need for increased emphasis on design all argue for a new look at computer engineering programs. A program that begins with a broad overview of computer engineering and spirals through a stream of design-oriented courses should adequately prepare computer engineering graduates for beginning design engineering careers. A program based on these principles can easily be adapted to meet the size, interests, and resources of any electrical engineering department.

References

1. ACM Curriculum Committee on Computer Education, "An Undergraduate Curriculum in Computer Science," Comm. ACM, March 1968, pp. 581-593.

2. J. D. Couger, ed., "Curriculum Recommendations for Undergraduate Programs in Information Systems," Comm. ACM, December 1973, pp. 727-749.

3. COSINE Committee, "An Undergraduate Computer Engineering Option for Electrical Engineering," Washington: National Academy of Engineering, January 1970.

4. M. E. Sloan, "Implications of Changes in Secondary School Mathematics for Computer Science and Computer Engineering Curricula," Proc. NCC, 43, May 1974.

5. M. E. Sloan, C. L. Coates, and E. J. Mc Cluskey, "COSINE Survey of Electrical Engineering Departments," Computer, June 1973, pp. 51-60.

6. Goals Committee, Final Report: Goals of Engineering Education, Washington: American Society for Engineering Education, 1968.

7. T. L. Booth, Digital Networks and Computer Systems, New York: Wiley, 1971.

8. M. L. Abrams and P. G. Stein, Computer Hardware and Software, Reading, Mass: Addison-Wesley, 1973.

9. COSINE Committee, Some Specifications for an Undergraduate Course in Digital Subsystems, Washington: National Academy of Engineering, November 1968.

10. J. B. Peatman, The Design of Digital Systems, New York: McGraw-Hill, 1972.

11. E. J. McCluskey and H. W. Gashwind, Design of Digital Computers, New York: Springer-Verlag, 1975, in press.

12. C. G. Bell and A. Newell, Computer Structures: Readings and Examples, New York: McGraw-Hill, 1971.

13. H. S. Stone, ed., Introduction to Computer Architecture, Palo Alto, California: SRA, 1975, in press.

AN EDUCATIONAL LABORATORY IN CONTEMPORARY DIGITAL DESIGN

by

Janis Beitch Baron

and

D. E. Atkins

Department of Electrical and Computer Engineering
and
Program in Computer, Information and Control Engineering
The University of Michigan
Ann Arbor, Michigan 48104

Summary

Formal education in computer architecture rests upon the integration of numerous specialized courses. One such course, a new laboratory concerning contemporary, register-transfer level digital design, is described. Design in the course proceeds at the level of interconnection of memories, arithmetic logic units, I/O interfaces, and buses, rather than at the individual gate level associated with traditional combinational and sequential logic design. The principle items of hardware used in the laboratory are DEC Register Transfer Modules, the DEC PDP/16M, and the INTEL SIM8-01 Microcomputer. Numerous software packages have been developed to support activities within the lab including an RTM simulator, a graphical simulation of an RTM control sequencer, a PDP/16M assembler, a microcode generator, and SIM8-01 utility routine, assembler, and simulator. During one fourteen-week term, students are expected to complete six assigned lab projects and one special project. Examples of projects are described. A critique based upon the students' reactions and the technician's experience with the equipment is given together with proposals for future additions to the course.

Introduction

Computer architecture is a system oriented design discipline requiring broad understanding within the hierarchy of computer system descriptions: the logic level, the instruction set level, the programming level, and the PMS level. (See reference [1] for a discussion of these description levels.) Formal education in computer architecture cannot be accomplished within a single course or two, but rather rests upon the integration of numerous specialized courses. This paper describes one such course in the computer engineering curriculum at The University of Michigan: a laboratory course concerning contemporary digital system design including, but not limited to, general purpose computers.

Laboratories in digital design ("logic labs") have existed within university curricula. several others exist within The University of Michigan. What justifies describing yet another? The answer to this question relies on our notion of "contemporary" digital system design which underlies the structure of this laboratory. Contemporary digital system design includes the following correlated properties:

- Uses medium and large scale integration (MSI, LSI) technology.
- Focuses on algorithms (not physical quantities such as voltages) and their alternate modes of implementation: hardware, software, microprogramming (firmware).
- Stresses register-transfer level primitives (registers, adders, memories, etc.) rather than gate level (gates, flip-flops, inverters).
- Employs a top-down approach (similar to structured programming) using flowchart specification of computation.

Laboratory projects are implemented using primarily two commercially available digital hardware products: the Digital Equipment Corporation (DEC) Register Transfer Modules (RTM)* and the INTEL SIM8-01 Microcomputer. A DEC PDP/8 minicomputer is also available for projects involving augmentation of general purpose machines with special purpose subsystems.

The course begins with several modest experiments to introduce the student to the Register Transfer Modules. Design proceeds at the level of interconnection of memories, arithmetic logic units, I/O interfaces, and buses rather than at the individual gate level associated with traditional combinational and sequential logic design. Design at this higher level permits the design and implementation of a moderately complex digital system, e.g. a mini-computer, within several laboratory periods. In later phases of the course, students select, design, implement, and analyze projects of their own chosing.

The course serves at least two broad purposes. It provides hands-on experience with digital hardware and immerses the student in the act of digital system design. Secondarily, the course develops skills in

*Register Transfer Module (RTM) is a registered trademark of Digital Equipment Corporation, Maynard, Massachusetts. The product is also known in a special form as the PDP-16.

the application of two relatively new commercial products which exhibit an increasing influence on the realities of digital design.

The course also serves as an introduction to techniques for implementing special-purpose digitial structures. Today there are many incentives to organize systems around a computer. However, with the present low cost of mini-computers, there would probably not be economic incentive to build a general purpose computer from modules used in this laboratory. The commercial value of register-transfer modules rests in the area of implementing special-purpose systems: either stand alone or augmented general-purpose computers.

This paper will include a description of the hardware and software facilities established for the laboratory, a description of the content of the course, and a critique based upon experience to date.

The intent of the paper is not to sell the goals of the laboratory, but rather to offer our experience to other educators who basically agree with the goals and who are also attempting to implement them.

Facilities

Hardware

The principle items of hardware used in the laboratory are DEC Register Transfer Modules, the DEC PDP 16/M, and the INTEL SIM8-01 Microcomputer. Other auxilary hardware required for interconnection will also be mentioned.

Register Transfer Modules (RTMs) - Register Transfer Modules represent a commercial approach to the implementation of modular computer systems which permit design and construction at the register transfer level. A history of modular computer systems, beginning with Estrin's "fixed-plus-variable" computer systems, is traced in reference [2].

Medium and large scale integration circuits in themselves provide a loosely constrained modular approach to the design of the data flow section of a digital computer. Modular systems, in the spirit described in reference [2] and exemplified in the DEC RTM's, go further by also standardizing and simplifying the control part of the system. They represent attempts to provide simple alternatives to the complexities and inefficiencies of programming general purpose computers for applications which lend themselves to specialized hardware organizations. The standardization implicit in the Register Transfer Modules enables students to design and construct several non-trivial digital systems within one term.

The RTM's are not projected as the way to implement digital systems. They do, however, provide a cost-effective solution to some design problems and are a pedagogical tool in studying solutions to many others.

The reader not familiar with a detailed description of Register Transfer Modules is referred to references [3]-[7]. Reference [3] contains the most correct description of the module pin assignments.

PDP-16/M Subminicomputer - In addition to the RTMs, the laboratory is equipped with one PDP-16/M, a specific configuration of RTM's sequenced by a microprogrammed controller. The unit is a functional computer with 256 words of read only control memory, 4 words of data read only memory, a general purpose register unit, 96 instructions, fully implemented 16 bit I/O enclosure, and power supplies. A detailed description of the unit is contained in reference [8].

The basic PDP-16/M sells for about $2000. The back plane is prewired to accept additional standard RTMs to extended memory and I/O capabilities.

The availability of the 16/M provides a specific contrast to the "chained evoke" control technique typically used with the RTMs. The economic advantages of a stored program approach to control become apparent for systems involving more than a few dozen control steps.

Both reprogrammable (PROM) and fusible link read only memories (ROMs) are available on the control memory module. DEC provides several options for programming ROMs: one using a PDP/8 and special interfacing hardware, one using the PDP-16/M itself. None of these are used in the U. M. laboratory. The PROMs on the PCS16-B control board have been removed and replaced by zero insertion force, dual-in-line sockets. With this modification the PROMs can be easily removed and replaced. They are programmed using the INTEL SIM8-01 prototype system described in the next section. An assembler to produce PDP-16/M code, including punched paper tape compatible with the PROM burning equipment, is described in the software portion of Section II.

INTEL SIM8-01 - An MCS-8 Microcomputer - Microprocessors are exerting an increasing influence on digital design, not only as the basis for microcomputer systems, but also as components within other systems. Evidence is accumulating that they are cost effective replacements for even small amounts of random logic. Exposure to microprocessor technology is an essential ingredient in computer engineering education. The INTEL SIM8 and associated hardware have proved to be a satisfactory vehicle for such exposure.

The SIM8-01 is a specific configuration of components from the INTEL Microcomputer Set built around the 8008-01 central processor chip. It is a prototyping board which forms an operational microcomputer. It includes a central processor (INTEL 8008), 1024 x 8 bit read/write memory, six I/O ports (two in and four out), a clock generator, and sockets for 2048 words of read only memory.

The SIM8-01, a single 10" x 12" board, is used in conjunction with the MCB8-10 interconnect and control module. With the addition of a MP7-03 PROM Programmer, an ASR teletype, and a utility program stored in PROMs, the hardware becomes the MCS-8 PROM Programming System. Two such configurations are included in the laboratory. The purchase price of each is approximately $1200. The organization and operation of this equipment is well documented in reference [9].

Interfacing Equipment - As discussed in Section III, the later part of the course is devoted to a free choice design project. Many of the projects require the interconnection of various combinations of the following: RTM systems, PDP-16/M, SIM8, and PDP 8 minicomputer. The logic levels of all of these are compatible. However, some additional interfacing logic may be required to satisfy timing constraints. Suitcase type logic kits, developed for a long standing introductory course in gate-level design,

have proved valuable in meeting the requirement. Ribbon cable terminated with 16 pin dual in line connectors (DIPs) are the primary means of interconnection.

A Microcomputer - RTM Student Project - Figure 1 shows a student built music synthesizer using RTM's, the INTEL equipment, and a suitcase logic kit.

Software

Several software packages have been developed to support activities within the laboratory. These packages, all but one of which are reasonably transportable, are briefly described in this section. References to more complete descriptions will be included. The exception cited above is a computer graphics package to simulate the operation of an RTM control sequence. A computer generated motion picture based upon the graphical simulation is scheduled for production within the coming year and will be available on a loan basis.

RTM Simulator - To reduce the amount of time that must be spent debugging once hardware assembly has begun, the RTM designer may choose to simulate his/her design on the central computing facility before building it. With SIM-16, a FORTRAN simulation of part of the RTM system, the digital hardware designer can create, test, and modify the logical structure of any single-bus RTM design. The program requires the user to describe the design as a sequence of FORTRAN CALL statements. These statements accomplish transfers of data between symbolic source and destination registers and perform control functions such as branching, merging, and decoding. The program allows close monitoring throughout the progress of the simulation. It is more fully described in Ref. [13].

Wiring Table Generation - To facilitate the assembly of an RTM device once its flowchart has been decided upon, a program called *WIREWRAP is used. *WIREWRAP, written in a combination of FORTRAN and IBM 360 Assembly Language, is similar to DEC's CHARTWARE, which runs only on the PDP-10. *WIREWRAP is a general-purpose logic design aid, extended to include RTM designs. Once a design is complete, the user need only describe it in a simple form that is derived directly from the flowchart. The program, then, decides which RTM's are needed, where each will be placed on the wire-wrap panel, and which pins should be connected together. It checks all gates and storage elements for overloading, calculates total cost and power requirements, minimizes wire length wherever possible, and generates complete wiring instructions and documentation. It is further described in Ref. [11].

Control Simulation on a CRT - As a pictorial aid for demonstrating the operation of the hard-wired control structure of the RTM system, The University of Michigan's PDP-9 Logic Simulation graphics system has been extended to provide an RTM control simulation. The simulation emphasizes the details of control timing rather than the trace of data flow provided by SIM-16. The control timing is graphically illustrated in slow motion on a DEC 339 display screen. The particular display, composed and controlled by the user by means of a light pen, simulates the operation of a specified CONTROL SEQUENCER. The CONTROL SEQUENCER may consist of EVOKE steps (mostly register transfers), two-way Boolean decisions, branches, merges, and subroutine calls. Figure 2 shows a student using the simulator.

PDP-16M Assembler - The PDP-16M Assembler, which runs on the central computing facility, translates symbolic programs for the PDP-16/M into binary object modules that are usually produced in the form of paper tapes. The contents of these paper tapes can be electronically burned into the Programmable Read Only Memories (PROMs) for use with the PDP-16M by means of the SIM8-01 microcomputer set and the Michigan INTEL Programmer and Loader Routines (described below). Ref. [16] is a user's guide for the Assembler.

Microcode Generator - As a special project, three students have developed a tool for implementing interpreters by means of microprogramming. The General Purpose Microcode Generator for Emulation on the PDP-16/M is an IBM 360 Assembly Language program which generates microcode for the control memory of the PDP-16/M. The program's input is a description of the instruction set of the emulated computer. This facility gives the students the opportunity to compare hardware and firmware implementations. It is further described in Ref. [8].

Utility Routines - To support software development for the INTEL Corporation SIM8-01 microcomputer, a set of utility programs has been developed. The Michigan INTEL Programmer and Loader Routines (MIM-9/PL) enable the user to load object code contained on paper tape into the writable memory of the SIM8-01, to debug the code interactively using a variety of breakpoint and status examination commands, and to electronically program the programmable read only memories (PROMs) once the code has been debugged. Although INTEL itself provides some utility programs, the MCS-8 PROM Programming System, the programs lack facilities for interactive debugging, are inflexible, and require an unnecessarily verbose format for object code. MIM-8/PL includes all of the facilities of the INTEL version plus others. The programs are available for general distribution. They are further described in Ref. [15].

MCS-8 Assembler - Object code for the SIM8-01 is typically produced by the Michigan INTEL MCS-8 Assembler (MIM-8/AL), an IBM 360 Assembly Language program. The assembler produces a binary-coded paper tape which is of the same format as the output of the PDP-16/M assembler. The contents of the paper tape can be electronically burned into a PROM by means of the SIM8-01 and MIM-8/PL (described above). MIM-8/AL can run under OS/VSI or the Michigan Terminal System and is available for general distribution. It is further described in Ref. [14].

MCS-8 Simulator - Students in courses without access to the MCS-8 hardware and students in the digital design lab who wish to debug their code before using the hardware make use of the MCS-8 Simulator. The simulator, written in IBM 360 Assembly Language, can execute about 300 simulated instructions per minute. It is designed to handle all of the MCS-8 instructions, including the input/output instructions. Its internal registers and I/O ports can be monitored throughout the simulation. In addition to providing a means for debugging code, the simulator generates a trace that is useful later when the code is being checked on the real machine. It is more completely described in Ref. [17].

Content of the Course

The content of the digital design laboratory course is built around class lectures, assigned

laboratory projects, and special student projects, with little emphasis on written examinations. Each week of the two-credit hour, fourteen-week course contains a one-hour lecture and a three-hour lab. The students spend additional time outside of class preparing for and documenting each lab project. Written examinations are not used since the projects themselves are a comprehensive test of what the students have learned.

The Class Lectures

In addition to covering the various hardware and software facilities and the assigned projects, the class lectures discuss the following topics:

1. Levels of description of digital design, in particular, the register transfer (RT) and processor-memory-switch (PMS) levels.
2. Design choices behind the production of the Register Transfer Modules for example, the logical design, the bus structure, the asynchronous timing.
3. Speed/cost, hardware/firmware/software, and special-purpose/general-purpose trade-offs.
4. Alternate forms of parallelism and synchronization of parallel activities.

Assigned Projects

During one fourteen-week term students are expected to complete six assigned digital design lab projects and one special project.

Table 1 lists the assigned lab projects and the number of lab periods allocated to complete each project.

PROJECT	ASSIGNED PROJECT	NUMBER OF 3-HOUR LAB PERIODS
1	1 to N Summer	1
2	Count the number of 1's or 0's in a word	1
3	Register I/O Utility	1-2
4	Single Precision, Integer Multiplication and Division	1-2
5	Simulated Serial Synchronous Receiver	2
6	MCS-8 Program	1-2

Table 1 - Assigned Lab Projects

Projects 1 and 2 enable the students to gain familiarity with the RTM's and the design process for using them.

Project 3 is the design and implementation of a "Register I/O Utility Routine." The routine uses the lights and switches for depositing in and examining the registers within the scratch pad memory. It also produces an I/O facility that is used in later projects, for example in lab 4.

Project 4 gives the students their first experience with the design and implementation of a fairly complex RTM system and with multiplication and division algorithms.

The fifth project is the design of a serial synchronous receiver. It is not built in hardware, but is implemented with the SIM-16 RTM simulation package. The students must design the receiver to satisfy a set of requirements including transmission speed, maximum message length, and specified character length.

The sixth project enables the students to gain familiarity with te CPU architecture and the instruction set of the MCS-8 micro-computer. They write and debug a subroutine to sort a group of records stored in a buffer in an external random access memory.

Special Design Projects

The final six lab periods of the course are devoted to free choice individual or small-group special projects. One suggested approach to the projects is intended to illustrate "fixed plus variable structure" architecture.

The students are told to:

1. Define an application problem.
2. Implement a software solution on the SIM8-01.
3. Identify a time-critical portion of the software realization and design a special purpose RTM or PDP-16M system to implement this portion.
4. Interface the special purpose processor to the SIM8-01 software.
5. Modify the program to use the hardware processor instead of the software routines.
6. Conduct a performance comparison of the two approaches.

Projects following this approach give the students experience with interfacing and with analysis of hardware/software trade-offs.

The following are examples of some of the more ambitious special projects:

1. The implementation of part of an assembler on the SIM8-01 with an RTM content-addressable memory for the symbol table.
2. A SIM8-01 based desk calculator with a PDP-16M floating point arithmetic unit and elementary function generator (sin,cos, etc.).
3. A SIM8-01-controlled signal processor with a PDP-16M Fast Fourier Transform (FFT) routine.
4. A CRT line drawer. The SIM8-01 receives line-drawing instructions from the teletype consisting of the endpoints of a set of lines. It then sends the data to an RTM device which controls the CRT display.
5. An infix to postfix translator for arithmetic expressions. The project includes a speed and cost comparison of an RTM vs. a PDP-8 assembly language implementation.
6. A "Digital Music Machine." The SIM8-01 receives the musical score via teletype, then translates and forwards it to an RTM music synthesizer. (See Figure 1.)

Critique of the Course

The following critique will discuss the studdents' reactions to the course, the technician's reactions to working with equipment, and some accumulated ideas for possible future additions to the course facilities.

The Student's Reactions

Aside from a few complaints, most students claim that they find the course challenging, interesting, and rewarding. The hands-on experience, they feel, is clearly superior to classroom lecture learning. The special project, most agree, is particularly interesting. Perhaps more time should be allotted for the final project by eliminating one of the first six projects.

The students' express only two significant complaints. The main complaint concerns the great amount of time that the course work requires. Many suggest that three hours of credit should be given instead of two. In addition, the students feel that the large amount of hand-wiring becomes tedious after about four projects. For this reason, work with the RTM simulator instead of the actual hardware by at least the fifth project is advisable.

The Technician's Reactions

The laboratory technician has mixed reactions about the desirability of working with the equipment used in the lab. His job consists of acquiring, assembling, and maintaining the equipment. These tasks, though not time-consuming by themselves, have been complicated by what the technician calls "growing pains." In other words, the equipment being used is new and still in the development stage. It is, therefore, insufficiently--and often incorrectly--documented. The problems of tracking down needed information often consume weeks. Fortunately, the availability of information from INTEL and DEC has subsequently improved.

Excluding all problems related to the "growing pains," the time requirements of the technician are not great. The assembly stage consists of a few weeks for the construction of four RTM assembly racks for the eight- to twelve-student labs, and a week or two for the remaining assembly details. Maintenance requirements are minimal. Burnt out PROMs constitute the only problem encountered with the MCS-8 system. In the past two years, we have found only two faulty RTM boards. No maintenance problems have arisen from use of the PDP-16M.

The technician has compiled lists of tips for use by others working with the same equipment. The lists, available from the authors, include tips on trouble-shooting, repair, facilities for ease of operation, purchase of the needed pieces of hardware, and proper care of the equipment.

Future Additions to the Course Facilities

Some possible future additions to the course facilities include:

1. Movies of the RTM control simulation - Students generally agree that the CRT display facility is a helpful, clear, pedagogic device. Hopefully, soon, movies of the display will be available for general distribution.
2. An interface between the SIM-16 RTM simulator and the *WIREWRAP program - After an RTM design has been logically debugged by means of SIM-16, the next step is to input the design to *WIREWRAP for implementation details. A program to convert the input for SIM-16 to a form compatible with the input for *WIREWRAP would greatly simplify the user's job.
3. An interface to enable the PDP-16M to substitute a minicomputer for the PROM during the micro-program development stage. Since PROMs cannot be re-programmed very easily or very often, debugging PDP-16M programs using the PROM to store the program is clumsy. A micro-program stored in a minicomputer, however, is easily altered. Perhaps, while the micro-program is being debugged, the minicomputer could feed it the micro-instruction that will, after the debugging is complete, be read from the PROM.

Acknowledgments

Contributions to this paper have been made by - the laboratory technician: Duane Haines: the instructors: K. B. Irani, S. Goldner, M. Bauer;

Those who developed the software facilities: S. Goldner, L. Uzcátegui, J. Gilbert, M. Ziegler, J. Blinn, K. D. Kanaby, D. R. Hanson, E. Berelian, J. Mulla, C. Zervos;

the students who designed and implemented the special projects described: T. Harkaway, P. Kostishak, D. Luther, J. J. Puttress, E. A. Berra, M. Shrader, R. Bryant.

This work was supported in part by the NSF Grant GY-8318.

References

[1] C. G. Bell and Allen Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 1971, Chapter 1.

[2] Robert A. Ellis, "Modular Computer Systems," Computer, Vol. 6, No. 10 (Oct. 1973), pp. 13.

[3] C. Gordon Bell, John Grason, and Allen Newell, Designing Computers and Digital Systems of Using PDP 16 Register Transfer Modules, Digital Press, 1972.

[4] PDP 16 Computer Designers Handbook, Digital Equipment Corporation, 1971.

[5] Register Transfer Modules..A Cost-effective, Easy-to-use, Method for Designing Logic Systems, Digital Equipment Corporation, 1973.

[6] C. G. Bell and J. Grason, "Register Transfer Modules (RTM) and their Designs," Computer Design, May 1971.

[7] C. G. Bell, J. L. Eggert, J. Grason, and P. Williams, "The Description and Use of Register Transfer Modules (RTMs)," IEEE Trans. Comput., Vol. C-21 (May 1972), pp. 495-500.

[8] PDP 16/M User's Guide, Digital Equipment Corporation, 1973.

[9] MCS-8 Micro Computer Set User's Manual, INTEL Corporation, March 1973.

[10] INTEL Data Catalog, INTEL Corporation, February 1973.

[11] Janis Beitch Baron, Using *WIREWRAP for PDP/16 Register Transfer Module Design, Department of Electrical and Computer Engineering, The University of Michigan, Ann Arbor, Michigan, July, 1973.

[12] Daryl E. Knobloch, *WIREWRAP User's Guide., Computer Center Memo #M181, The University of Michgian Computing Center, July 1971.

[13] James G. Gilbert, User's Manual for SIM-16, Electrical and Computer Engineering, The University of Michigan, Ann Arbor, Michigan, September 1973.

[14] S. M. Goldner, L. A. Uzcátegui, D. E. Atkins, K. B. Irani, User's Guide for the Michigan INTEL MCS-8 Assembly Language, Electrical and Computer Engineering, The Universlty of Michigan, Ann Arobr, Michigan, July 1973.

[15] S. M. Goldner. L. A. Uzcátegui, D. E. Atkins, and K. B. Irani, User's Guide for the Michigan INTEL MCS-8 Programmer and Loader System, Electrical and Computer Engineering, The University of Michigan, Ann Arbor, Michigan, July 1973.

[16] Leonardo A. Uzcátegui, PDP 16-M Assembler Description and User's Guide, Electrical and Computer Engineering, The University of Michigan, Ann Arbor, Michigan, July 1973.

[17] K. D. Kanaby and D. R. Hanson, MCS-8 Simulator, Electrical and Computer Engineering, The University of Michigan, Ann Arbor, Michigan, revised April 1973.

[18] Elias Berelian, Jamshed Mulla, and Christian Zervos, General Purpose Microcode Generator for Emulation on the PDP-16M, Electrical and Computer Engineering, The Univeristy of Michigan, Ann Arbor, Michigan, April 1974.

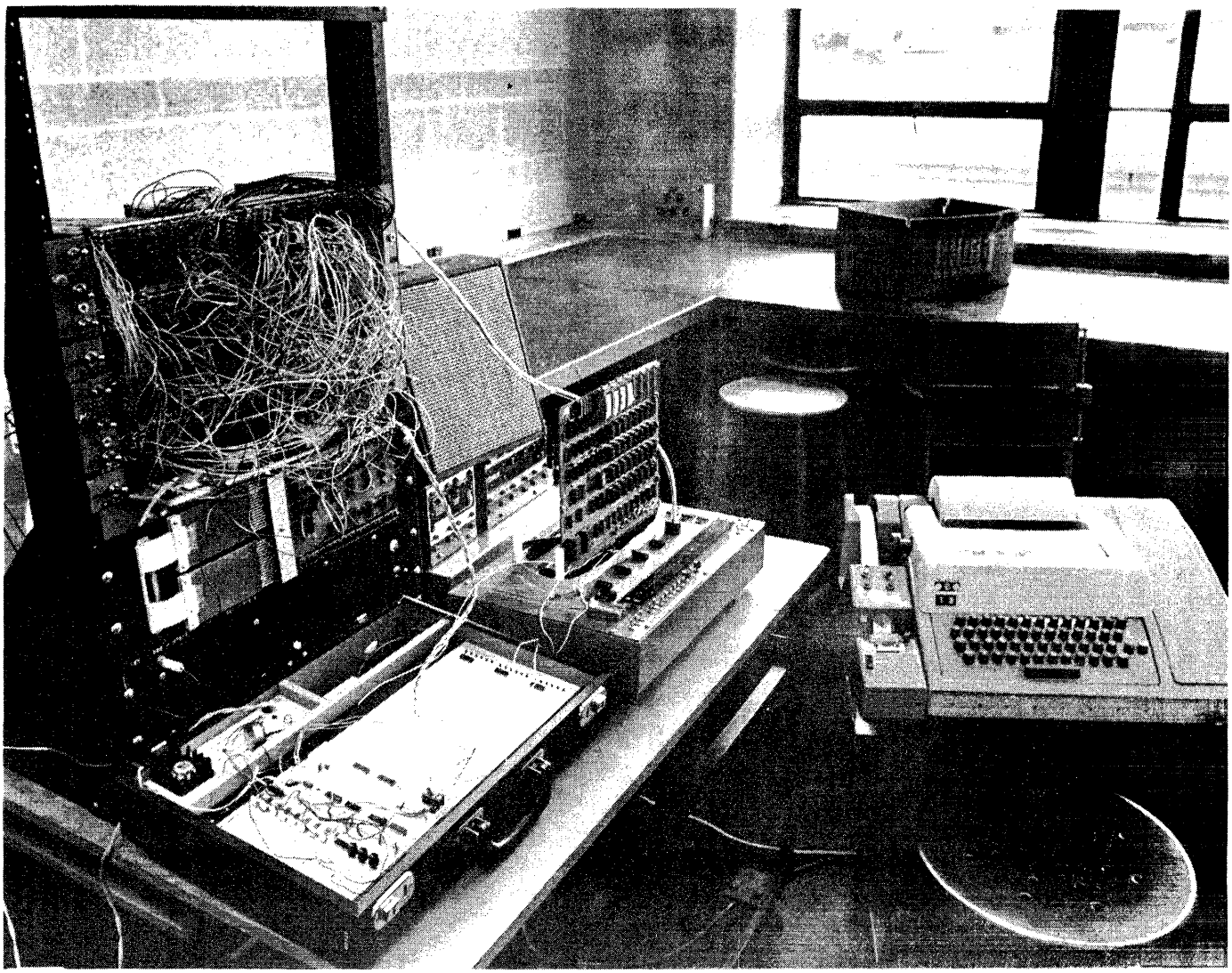


Fig. 1. A microcomputer - RTM student project (a music synthesizer).

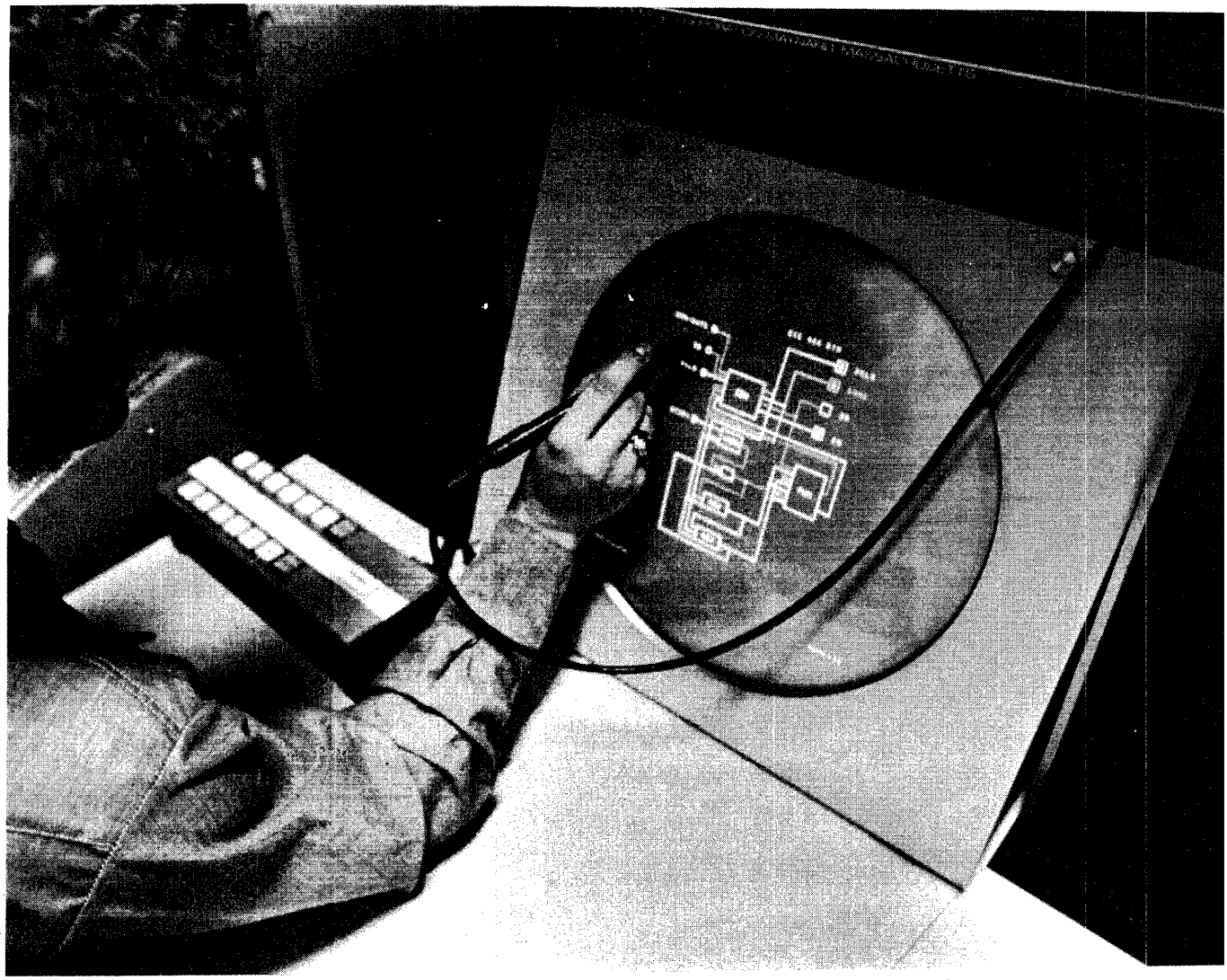


Fig. 2. Student using RTM control simulator.