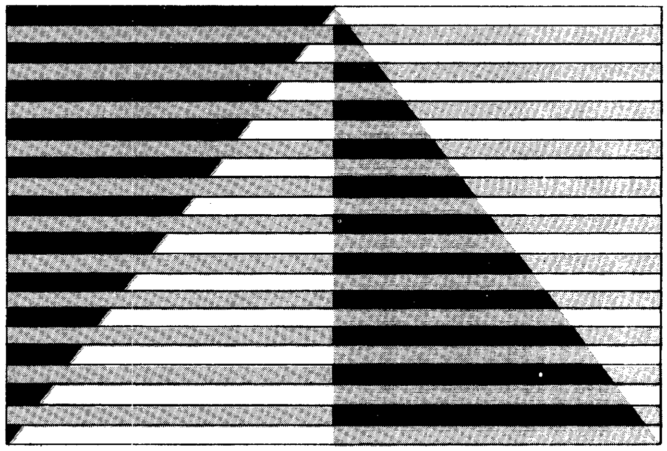
25 YEARS OF SERVICE

IEEE COMPUTER SOCIETY

CONFERENCE PROCEEDINGS

The 3rd Annual Symposium on COMPUTER ARCHITECTURE



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

JANUARY 19-21, 1976

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

IEEE Service Center
445 Hoes Lane, Piscataway, N.J. 08865

75CH1043-5C

PREFACE

GENERAL CHAIRMEN'S REMARKS

The success of the past two Symposia on Computer Architecture and the interest that they have generated are the greatest rewards that their organizers could possibly hope for. The increase in attendance and the quantity and quality of the papers received indicate that growth and creativity in computer architecture continue on the upswing. At the same time the broad circulation (over 3,000 copies) of these Proceedings provides recognition to the authors and coverage for the interested audience not in attendance at the conference.

An objective of the symposium is to present the methodologies and languages for representing architectural design within the pragmatics of system evaluation and implementation. While these topics have received attention in the abstract a greater participation of users and designers from government and industry would aid greatly in developing these areas. Symposium attendance has been historically divided equally between two groups, university and industry, but it has been difficult to obtain submissions from practitioners of the art. It is a credit to the Program Committee and in particular to its chairman that at least a third of the final program consists of industrial contributions. This is perhaps a good place to remind the non-academic readers of these Proceedings of their obligation to give, in the measure that they receive, from their own valuable store of practical experience and knowledge.

An exacting Program Committee has been responsible for the delicate task of program selection and organization. Their acute perception in the choice of contributions is evidenced in the pages that follow. We thank them all but in particular Dan Siewiorek for his ability to multiprocess a broad variety of tasks in a competent and efficient way. One new and valuable feature added to this Third Symposium is the Session on Recent Results. It included reports on fresh developments of significance not yet sufficiently ripe at the deadline of manuscript submission. The archival value of the presentations in the program should be complemented by these reports of work in progress, and also by a one day Tutorial on Microprogramming by Mike Galey and Richard Kleir.

An evening panel discussion organized by Yoahan Chu will concentrate on the unifying and changing aspects of software and architecture design. This session and the informal atmosphere encouraged during the coffee breaks and other social functions should provide the opportunity for personal interaction among the participants of the conference.

As in the past, the presence of papers from nine different countries brings out the cosmopolitan flavor of this conference. Responsible in no small part for this participation are Rodnay Zaks from France and Reiner Hartenstein from Germany ghrough the support of the Euromicro association. The "Best Paper Award", instituted last year to encourage excellence in the written and oral presentations of a paper at the Symposium will be continued. The winners for 1975, Harold W. Lawson, Jr. and Bengt Magnhagen from Linköping Högsköla in Sweden, will receive their award of $100.00 and a certificate during the opening ceremonies of the Third Symposium for their paper, "Advantages of Structured Hardware."

Our appreciation should also go to the unsung heroes of an effort like this one: to those who enriched the selection process by submitting papers of quality that were not, for special reasons, accepted at this time and to the many who aided in behind-the-scenes arrangements. Among the latter we single out Harvey Glass and Joe Deeds who worked diligently to make it more pleasant for those attending our 1976 meeting on the shores of the Gulf of Mexico.

Michael J. Flynn
Oscar N. Garcia

PROGRAM CHAIRMAN'S REMARKS

A survey of the symposium sessions indicates some current trends in computer architecture research. The most popular topics (by number of papers submitted) were computer networks and multimicroprocessor systems. A heightened awareness of the software/hardware interface is exhibited by three sessions and an evening panel discussion. The sessions cover topics of hardware/software system considerations, resource sharing and process coordination, and architectures to support software concepts.

Performance of computer systems continues to be an important topic. One session covers the theoretical concepts of performance evaluation and modeling while another session is devoted to architectural features for performance enhancement. A historical perspective of the art of computer design is the theme of the architecture evolution session. This session focuses on the design decisions in two computer families and promises to be one of the highlights of the symposium.

Interest continues in hardware descriptive languages and computer architecture education. Finally, two new sessions round out the program: the effects of special applications on computer architecture, secondary storage.

The large response to the symposium's call for papers provided a wealth of material from which to assemble a technical program. I would like to thank all the authors of submitted papers for their interest and assistance in putting together a quality program.

The fact that over 80 papers were reviewed in less than two months is a tribute to the efforts of the members of the program committee and the referees. Their assistance is deeply appreciated. I would also like to acknowledge the support provided by Oscar Garcia in the many phases of program planning. Finally, a special note of thanks is due Dorothy Josephson for keeping the manuscripts and letters rolling. My informal calculations indicate that program correspondence was in excess of 800 letters.

Daniel P. Siewiorek

PROGRAM COMMITTEE

Daniel P. Siewiorek, Chairman	W. H. Huen	Leon Presser
Daniel Atkins III	E. Douglas Jensen	John F. Wakerly
Harvey Cragon	Harold Lorin	John H. Wensley
Edward Davidson	Craig Mudge	Neil C. Wilhelm
Reiner W. Hartenstein	Michael D. Mulder	William A. Wulf
John P. Hayes	Marshal C. Pease	Rodnay Zaks

REFEREES

A. M. Abd-Alla
Guy Almes
George A. Anderson
Judith A. Anderson
James B. Angell
Daniel Atkins III
J. L. Baer
Mario Barbacci
Forest Baskett
A. P. Batson
Dileep Bhandarkar
Barry Borgerson
William Brantley
R. E. Brundage
Don Chamberlin
Herbert Chang
Lih Chang
Yaohan Chu
Douglas Clark
Harvey Cragon
Edward Davidson
Peter Denning
D. Dennis
Jack B. Dennis
Lloyd Dickman
Donald Dietmeyer
Linda Dodge
Richard Eckhouse
Robert A. Ellis
Lee Erman
T. Feng
Eduardo Fernandez
Edward Feustal
Lawrence Flon
W. S. Ford
Warren Franz
Samuel Fuller
Oscar Garcia
M. Z. Ghanem
H. M. Gladney
R. H. Glorioso
John Grason
A. N. Habermann
V. C. Hamacher
A. Hassitt
John P. Hayes
Leonard Haynes
Leonard D. Healy
Fredrick J. Hill
Terry T. Hsu
Wing Hing Huen
Ashok Ingle
Portia Isaacson
E. Douglas Jensen
Richard Johnsson
Anita K. Jones
Angel Jordan
J. Egil Juliussen
Robert Jump
Olaf Kaestner
Theodore Kehl
Willis King
Leonard Kleinrock
Michael Knudsen
Uno Kodres
R. Krishnan
H. T. Kung
John A. N. Lee
Victor Lesser
Roy Levin
G. J. Lipovski
Ming T. Liu
Harold Lorin
Harold Livings
David Misunas
Thomas Mitchell
Craig Mudge
Michael Mulder
Dave Nelson
Peter Neumann
Peter Oleinick
Severo Ornstein
E. W. Page
Alice Parker
Janak H. Patel
Marshall Pease
Karla Martin Perdue
Udo W. Pooch
Leon Presser
Tom Price
H. R. Ramanvjam
S. S. Reddi
David C. Rine
Larry Robinson
Brian Rosen
Steven Saunders
Michael Schlansker
N. F. Schneidewind
Mark Sebern
Daniel Serain
Mary Shaw
Howard Jay Siegel
Shankar Singh
Basil Smith III
Edward Snow
Harold Stone
S. Y. W. Su
Richard Swan
Daniel T. W. Sze
A. Thomasian
Kenneth Thurber
Judy A. Townley
Rollins Turner
J. D. Ullman
Christopher Vickery
Maniel Vineberg
Z. G. Vranesic
John F. Wakerly
Jerry Waxman
Terry Welch
John Wensley
Neil C. Wilhelm
Wayne T. Wilner
Larry Wittie
Y. S. Wu
William A. Wulf
S. G. Zaky

CONTENTS

ABSTRACT

Gordon Bell, William D. Strecker
November 8, 1975

COMPUTER STRUCTURES: WHAT HAVE WE LEARNED FROM THE PDP-11?

Over the PDP-11'S six year life about 20,000 specimens have been built based on 10 species (models). Although range was a design goal, it was unquantified; the actual range has exceeded expectations (500:1 in memory size and system price). The range has stressed the basic mini(mal) computer architecture along all dimensions. The main PMS structure, i.e. the UNIBUS, has been adopted as a de facto standard of interconnection for many micro and minicomputer systems. The architectural experience gained in the design and use of the PDP-11 will be described in terms of its environment (initial goals and constraints, technology, and the organization that designs, builds and distributes the machine).

1.0 INTRODUCTION

Although one might think that computer architecture is the sole determinant of a machine, it is merely the focal point for a specification. A computer is a product of its total environment. Thus to fully understand the PDP-11, it is necessary to understand its environment.

Figure Org. shows the various groups (factors) affecting a computer. The lines indicate the primary flow of information for product functional behavior and for product specifications. The physical flow of goods is nearly along the same lines, but more direct: starting with applied technology (e.g., semiconductor manufacturers), going through computer manufacturing, and finally to the service personnel before being turned over to the final user.

The relevant parts, as they affect the design are:

1. The basic technology--it is important to understand the components that are available to build from, as they directly affect the resultant designs.

2. The development organization--what is the fundamental nature of the organization that makes it behave in a particular way? Where does it get inputs? How does it formulate and solve problems?

3. The rest of the DEC organization--this includes applications groups associated with market groups, sales, service and manufacturing.

4. The user, who receives the final output.

Note, that if we assume that a product is done sequentially, and each stage has a gestation time of about two years, it takes roughly eight years for an idea from basic research to finally appear at the user's site. Other organizations also affect the design: competitors (they establish a design level and determine the product life); and government(s) and standards.

There are an ever increasing number of groups who feel compelled to control all products bringing them all to a common norm: the government(s), testing groups such as Underwriters Laboratory, and the voluntary standards groups such as ANSI and CBEMA. Nearly all these groups affect the design in some way or another (e.g. by requiring time).

2.0 BACKGROUND

It is the nature of engineering projects to be goal oriented--the 11 is no exception, with much pressure on deliverable products. Hence, it is difficult to plan for a long and extensive lifetime. Nevertheless, the 11 evolved more rapidly and over a wider range than we expected, placing unusual stress on even a carefully planned system. The 11 family has evolved under market and implementation group pressure to build new machines. In this way the planning has been asynchronous and diffuse, with distributed development. A decentralized organization provides checks and balances since it is not all under a single control point, often at the expense of compatibility. Usually, the hardware has been designed, and the software is modified to provide compatibility.

Independent of the planning, the machine has been very successful in the marketplace, and with the systems programs written for it. In the paper (Bell et al, 1970) we are first concerned with market acceptance and use. Features carried to other designs are also a measure of how it contributes to computer structures and are of secondary importance.

The PDP-11 has been successful in the marketplace with over 20,000 computers in use (1970-1975). It is unclear how rigid a test (aside from the marketplace) we have given the design since a large and aggressive marketing and sales organization, coupled with software to cover architectural inconsistencies and omissions, can save almost any design. There was difficulty in teaching the machine to new users; this required a large sales effort. On the other hand, various machine and operating systems capabilities still are to be used.

2.1 GOALS AND CONSTRAINTS - 1970

The paper (Bell et al, 1970) described the design, beginning with weaknesses of minicomputers to remedy other goals and constraints. These will be described briefly in this section, to provide a framework, but most discussion of the individual aspects of the machine will be described later.

Weakness 1, that of limited address capability, was solved for its immediate future, but not with the finesse it might have been. Indeed, this has been a costly oversight in redundant development and sales.

There is only one mistake that can be made in a computer design that is difficult to recover from--not providing enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer. Of course, there is a fundamental rule of computer (and perhaps other) designs which helps to alleviate this problem: any well-designed machine can be evolved through at least one major change. It is extremely embarrassing that the 11 had to be evolved with memory management only two years after the paper was written outlining the goal of providing increased address space. All predecessor DEC designs have suffered the same problem, and only the PDP-10 evolved over a ten year period before a change was made to increase its address space. In retrospect, it is clear that since memory prices decline at 26% to 41% per year, and many users tend to buy constant dollar systems, then every two or three years another bit is required for the physical address space.

Weakness 2 of not enough registers was solved by providing eight 16-bit registers; subsequently six more 32-bit registers were added for floating point arithmetic. The number of registers has proven adequate. More registers would just increase the context switching time, and also perhaps the programming time by posing the allocation dilemma for a compiler or a programmer.

Lack of stacks (weakness 3) has been solved, uniquely, with the auto-increment/auto-decrement addressing mechanism. Stacks are used extensively in some languages, and generally by most programs.

Weakness 4, limited interrupts and slow context switching has been generally solved by the 11 UNIBUS vectors which direct interrupts when a request occurs from a given I/O device.

Byte handling (weakness 5) was provided by direct byte addressing.

Read-only memory (weakness 6) can be used directly without special programming since all procedures tend to be pure (and reentrant) and can be programmed to be recursive (or multiply reentrant). Read-only memories are used extensively for bootstrap loaders, debugging programs, and now provide normal console functions (in program) using a standard terminal.

Very elementary I/O processing (weakness 7) is partially provided by a better interrupt structure, but so far, I/O processors per se have not been implemented.

Weakness 8 suggested that we must have a family. Users would like to move about over a range of models.

High programming costs (weakness 9) should be addressed because users program in machine language. Here we have no data to suggest improvement. A reasonable comparison would be programming costs on an 11 versus other machines. We built more complex systems (e.g., operating systems, computers) with the 11 than with simpler structures (e.g. PDP-8 or 15). Also, some systems programming is done using higher level languages.

Another constraint was the word length had to be in multiples of eight bits. While this has been expensive within DEC because of our investment in 12, 18 and 36 bit systems, the net effect has probably been worthwhile. The notion of word length is quite meaningless in machines like the 11 and the IBM 360 because data-types are of varying lengths, and instructions tend to be in multiples of 16 bits. However, the addressing of memory for floating point is inconsistent.

Structural flexibility (modularity) was an important goal. This succeeded beyond expectations, and is discussed extensively in the part on PMS, in particular the UNIBUS section.

There was not an explicit goal of microprogrammed implementation. Since large read-only memories were not available at the time of the Model 20 implementation, microprogramming was not used. Unfortunately, all subsequent machines have been microprogrammed but with some additional difficulty and expense because the initial design had poorly allocated opcodes, but more important the condition codes behavior was over specified.

Understandability was also stated to be a goal, that seems to have been missed. The initial handbook was terse and as such the machine was only saleable to those who really understood computers. It is not clear what the distribution of first users was, but probably all had previous computing experience. A large number of machines were sold to extremely knowledgeable users in the universities and laboratories. The second handbook came out in 1972 and helped the learning problem somewhat, but it is still not clear whether a user with no previous computer experience can determine how to use a machine from the information in the handbooks. Fortunately, two computer science textbooks (Eckhouse, 75; and Stone and Siewiorek, 75) have been written based on the 11 to assist in the learning problem.

2.2 FEATURES THAT HAVE MIGRATED TO OTHER COMPUTERS AND OFFSPRINGS

A suggested test (Bell et al 1970) was the features that have migrated into competitive designs. We have not fully permitted this test because some basic features are patented; hence, non-DEC designers are reluctant to use various ideas.

At least two organizations have made machines with similar bus and ISP structures (use of address modes, behavior of registers as program counter and stack); and a third organization has offered a plug-replacement system for sale.

The UNIBUS structure has been accepted by many designers as the PMS structure. This interconnection scheme is especially used in microprocessor designs. Also, as part of the UNIBUS design, the notion of mapping I/O data and/or control registers into the memory address space has been used often in the microprocessor designs since it eliminates instructions in the ISP and requires no extra control to the I/O section.

Finally, we were concerned in 1970 that there would be offsprings--clearly no problem; there have been about ten implementations. In fact, the family is large enough to suggest need of family planning.

3.0 TECHNOLOGY

The computers we build are strongly influenced by the basic electronic technology. In the case of computers, electronic information processing technology evolution has been used to mark the four generations.

3.1 Effects Of Semiconductor Memory On The PDP-11 Model Designs

The PDP-11 computer series design began in 1969 with the Model 20. Subsequently, 3 models were introduced as minimum cost, best cost/performance, and maximum performance machines. The memory technology in 1969 formed several constraints:

1. Core memory for the primary (program) memory with an eventual trend toward semiconductor memory.

2. A comparatively small number of high speed registers for processor state (i.e. general registers), with a trend toward larger, higher speed register files at lower cost. Note, only 16 word read-write memories were availableat design time.

3. Unavailability of large, high speed read-only memories,

permitting a microprogrammed approach to the design of the control part. Note, not for ca paper, read-only memory was unavailable although slow, read-only MOS was available for character generators.

These constraints established the following design principles and attitudes:

1. It should be asynchronous and capable of accepting various configurations of memories in size and speed.

2. It should be expandable, to take advantage of an eventually larger number of registers for more data-types and improve context switching time. Also, more registers would permit eventually mapping memory to provide a virtual machine and protected multiprogramming.

3. It could be relatively complex, so that an eventual microcode approach could be used to advantage. New data-types could be added to the instruction set to increase performance even though they added complexity.

4. The UNIBUS width would be relatively wide, to get as much performance as possible, since LSI was not yet available to encode functions.

3.2 Variations In PDP-11 Models Through Technology

Semiconductor memory (read-only and read-write) were used to tradeoff cost performance across a reasonably wide range of models. Various techniques based on semiconductors are used in the tradeoff to provide the range. These include:

1. Improve performance through brute force with faster memories. The 11/45 and 11/70 uses bipolar and fast MOS memory.

2. Microprogramming (see below) to improve performance through a more complex ISP (i.e., floating point).

3. Multiple copies of processor state (context) to improve time to switch context among various running programs.

4. Additional registers for additional data-types--i.e., floating point arithmetic.

5. Improve the reliability by isolating (protecting) one program from another.

6. Improve performance by mapping multiple programs into the same physical memory, giving each program a virtual machine. Providing the last two points requires a significant increase in the number of registers (i.e. at least 64 word fast memory arrays).

4.0 THE ORGANIZATION OF PEOPLE

Three types of design are based both on the technology and the cost and performance considerations. The nature of this tradeoff is shown in Figure DS. Note, that one starts at 0 cost and performance, proceeds to add cost, to achieve a base (minimum level of functionality). At this point, certain minimum goals are met: for the computer, it is simply that there is program counter, and the simplest arithmetic operations can be carried out. It is easy to show (based on the Turing machine) that only a few instructions are required, and from these, any program can be written. From this minimal point, performance increases very rapidly in a step fashion (to be described later) for quite sometime (due to fixed overhead of memories, cabinets, power, etc.) to a point of inflection where the cost-effective solution is found. At this point, performance continues to increase until another point where the performance is maximized. Increasing the size implies physical constraints are exceeded, and the machine becomes unbuildable, and the performance can go to 0. There is a general tendency of all designers to "n+1" (i.e., incrementally add to the design forever). No design is so complete, that a redesign can't improve it.

The two usual problems of design are: inexperience and "second-systemitis". The first problem is simply a resources problem. Are there people available? What are their backgrounds? Can a small group work effectively on architectural definitions? Perhaps most important is the principle, that no matter who is the architect, the design must be clearly understood by at least one person.

Second-systemitis is the phenomenon of defining a system on the basis of past system history.

Invariably, the system solves all past problems...bordering on the unbuildable.

4.1 PDP-11 Experience

Some of the PDP-11 architecture was initially carried out by at Carnegie-Mellon University (HM with GB). Two of the useful ideas: the UNIBUS, and the use of general registers in a substantially more general fashion (e.g. as stack pointers) came out of earlier work (GB) at CMU and was described in COMPUTER STRUCTURES (Bell and Newell, 1971). During the detailed design amelioration, 2 persons (HM, and RC) were responsible for the specification.

Although the architectural activity of the 11/20 proceeded in parallel with the implementation, there was less interaction than in previous DEC designs where the first implementation and architecture were carried out by the same person. As a result, a slight penalty was paid to build subsequent designs, especially vis a vis microprogramming.

As the various models began to be built outside the original PDP-11/20 group, nearly all architectural control (RC) disappeared, and the architecture was managed by more people, and design resided with no one person! A similar loss of control occurred in the design of the peripherals after the basic design.

The first designs for 16-bit computers came from a group placed under the PDP-15 management (a marketing person, with engineering background). It was called PDP-X, and did include a range. As a range architecture, it was better thought out than the later PDP-11, but didn't have the innovative aspects. Unfortunately, this group was intimidating, and some members lacked credibility. The group also managed to convince management that the machine was potentially as complex as the PDP-10 (which it wasn't); since no one wanted another large computer disconnected from the main business, it was a sure suicide. The (marketing) management had little understanding of the machine. Since the people involved in the design were apparently simultaneously designing Data General, the PDP-X was not of foremost importance.

As the PDP-X project folded and the DCM (for Desk Calculator Machine for security) project started up, design and planning were in disarray, since Data General had been formed and was competing with the PDP-8 using a very small 16-bit computer. Although the Product Line Manager, a former engineer (NM) for the PDP-8, had the responsibility this time, the new project manager was a mathematician/programmer followed by another manager (RC) who had managed the PDP-8. Work proceeded for several months based on the DCM and with a design review at Carnegie-Mellon University in late 1969. The DCM review took only a few minutes. Aside from a general dullness and a feeling that it was too little too late to compete. It was difficult to program (especially by compilers). However, it's benchmark results were good. (We believe it had been tuned to the benchmarks, hence couldn't do other problems very well.) One of the designers (HM) brought along the kernel of an alternative, which turned out to be the PDP-11. We worked on the design all weekend, recommending a switch to the basic 11 design.

At this point, there were reviews to ameliorate the design, and each suggestion, in effect, amounted to an n+1; the implementation was proceeding in parallel (JO) and since the logic design was conventional , it was difficult to tradeoff extensions. Also, the design was constrained with boards and ideas held over from the DCM. (The only safe way to design a range is simultaneously do both high and low end designs.) During the summer at DEC, we tried to free op code space, and increased (n+1'ed) the UNIBUS bandwidth (with an extra set of address lines), and outlined alternative models.

The advent of large, read-only memories, made possible the various follow-on designs to the 11/20. Figure "Models" sketches the cost of various models versus time, with lines of consistent performance. This very clearly shows the design styles (ideologies). The 11/40 design was started right after the 11/20, although it was the last to come on the market (the low and high ends had higher priority to get into production as they extended the market). Both the 11/04 and 11/45 design groups went through extensive buy in processes, as they came into the 11 by first proposing alternative designs. In the case of the 11/45, a larger, 11-like 18-bit machine was proposed by the 15 group; and later, the LINC engineering group proposed an alternative design which was subset compatible at the symbolic program level. As the groups considered the software ramifications, buy-in was rapid. Figure Models shows the minimum cost-oriented group has two successors providing lower cost (yet higher performance) and the same cost with the ability to have larger memories and perform better. Note, both of these came from a backup strategy to the LSI-11. These come from larger read-only memories, and increased understanding of how to implement the 11.

The 11/70 is, of course, a natural follow on to extend the performance of the 11/45.

5.0 PMS STRUCTURE

In this section, we give an overview of the evolution of the PDP-11 in terms of its PMS structure, and compare it with expectations (Bell et al, 1970). The aspects include: the UNIBUS structure; UNIBUS performance; use for diagnostics; architectural control required; and multi-computer and multi-processor computer structures.

5.1 The UNIBUS - The Center Of The PMS Structure

In general, the UNIBUS has behaved beyond expectations, acting as a standard for intercommunication of peripherals. Several hundred types of memories and peripherals have been attached to it. It has been the principle PMS interconnection media of Mp-Pc and peripherals for systems in the range 3K dollars to 100K dollars (1975). For larger systems supplementary buses for Pc-Mp and Mp-Ms traffic have been added. For very small systems, like the LSI-11, a narrower bus (Q-bus) has been designed.

The UNIBUS by being a standard has provided us with a PMS architecture for easily configuring systems; any other organization can also build components which interface the bus...clearly ideal for buyers. Good busses (standards) make good neighbors (in terms of engineering), since people can concentrate on design in a structured fashion. Indeed, the UNIBUS has created a complete secondary industry dealing in alternative sources of supply for memories and peripherals. Outside of the IBM 360 I/O Multiplexor/Selector bus, the UNIBUS is the most widely used computer interconnection standard. Although it has been difficult to fully specify the UNIBUS such that one can be certain that a given system will work electrically and without missed data, specification is the key to the UNIBUS. The bus behavior specification is a yet unsolved problem in dealing with complexity--the best descriptions are based on behavior (i.e., timing diagrams).

There are also problems with the design of the UNIBUS. Although parity was assigned as two of the bits on the bus (parity and parity is available), it has not been widely used. Memory parity was implemented directly in the memory, since checking required additional time. Memory and UNIBUS parity is a good example of nature of engineering optimization. The tradeoff is one of cost and decreased performance versus decreased service cost and more data integrity for the user. The engineer is usually measured on production cost goals, thus parity transmission and checking are clearly a capability to be omitted from design...especially in view of lost performance. The internal Field Service organization has been unable to quantify the increase in service cost savings due to shorter MTTR by better fault isolation. Similarly, many of the transient errors which parity detects can be detected and corrected by software device drivers and backup procedures without parity. With lower cost for logic and increased responsibility (scope) to include warranty costs as part of the product design cost forces much more checking into the design.

The interlocked nature of the transfers is such that there is a deadlock when two computers are joined together using the UNIBUS window. With the window a computer can map another computer's address space into its own address space in a true multiprocessor fashion. Deadlock occurs when the two computers simultaneously attempt to access the other's addresses through each window. A request to the window is in progress on one UNIBUS, and at the same time a request to the other UNIBUS is in progress on the requestee's UNIBUS, hence neither request can be answered, causing a deadlock. One or both requests are aborted and the deadlock is broken by having the UNIBUS time out since this is equivalent to a non-existent address (e.g., a memory). In this way the system recovers and requests can be reissued (which may cause deadlock). The UNIBUS window is confined to applications where there is likely to be a low deadlock rate.

5.2 UNIBUS and Performance Optimality

Although we always want more performance on one hand, there is an equal pressure to have lower cost. Since cost and peformance are almost totally correlated the two goals perfectly conflict. The UNIBUS has turned out to be optimum over a wide dynamic range of products, (argued below). However, at the lowest size system, the Q-bus has been introduced, which contains about 1/2 the number of conductors; and at the largest systems, the data path width for the processor and memory has been increased to 32-bits for added performance although the UNIBUS is still used for communication with most I/O controllers.

Since all interconnection schemes are highly constrained, it is clear that future lower and higher systems cannot be accomplished from a single design unless a very low cost, high performance communication media (e.g. optical) is found.

The optimality of the UNIBUS comes about because memory size (number of address bits) and I/O traffic are correlated with the processor speed. Amdahl's rule-of-thumb for IBM computers (including the 360) is: one byte of memory is required per instruction/sec and one bit of I/O is required for each instruction executed. For our applications, we believe there is more computation required for each memory word, because of the bias toward control and scientific applications. Also, there has been less use of complex instructions typical of the IBM computers. Hence, we assume one byte of memory is required for each two instructions executed, and assume one byte of I/O is an upper bound (for real time applications) for each instruction executed. In the PDP-11, an average instruction accesses three to five bytes of memory, and with one byte of io, up to six bytes of memory are accessed for each instruction/sec. Therefore, a bus which can support two megabyte/sec traffic permits instruction execution rates of .33 to .5 mega instruction/sec. This imputes to meory sizes of .16 to .25 megabytes; the maximum allowable memory is .3 to .256 megabytes. By using a cache memory with a processor, the effective memory processor rate can be increased to further balance the processor. Alternatively, faster floating point operations will bring the balance to be more like the IBM data, requiring more memory.

5.3 Evolution Of Models: Predicted Versus Actual

The original prediction (Bell et al, 1970) was that models with increased performance would evolve using: increased path width for data; multi-processors; and separated bus structures for control and data transfers to secondary and tertiary memory. Nearly all of these forms have been used, though not exactly as predicted. (Again, this points to lack of overall architectural planning versus our willingness and belief that the suggestions and plans for the evolution must come from the implementation groups.)

In the earlier 11/45, a separate bus was added for direct access of either bipolar (300ns) or fast MOS (400ns) memory. In general, it was assumed that these memories would be small, and the user would move the important part of his algorithm to the fast memory for direct execution. The 11/45 provided a second UNIBUS for direct transmission of information to the fast memory without Pc interference. The 11/45 also increased performance by adding a second autonomous data operation unit called the Floating Point Processor (actually not a processor). In this way, both integer and floating point computation could proceed concurrently.

The 11/70, a cache based processor, is a logical extension of using fast, local memories, but without need for expert movement of data. It has a memory path width of 32-bits, and the control portion and data portion of I/O transfers have been separated as originally suggested. The performance limitation of the UNIBUS are removed, since the second Mp system permits data transfers of up to five megabytes/sec (2.5 times that of the UNIBUS). Note, that a peripheral memory map control is needed since Mp address space (two megabytes) exceeds the UNIBUS. In this way, direct memory access devices on the UNIBUS transfer data into a mapped portion of the larger address space.

5.4 Multi-processor Computer Structures

Although it is not surprising that multi-processors have not been used except on a highly specialized basis, it is depressing. In Computer Structures (Bell and Newell, 71) we carried out an analysis of the IBM 360, predicating a multi-processor design. The range of performance covered by the PDP-11 models is substantially worse than with the 360, although the competitive environment of the two companies is substantially different. For the 360, smaller models appear to perform worse than the technology would predict. The reasons why multiprocessors have not materialized may be:

1. The basic nature of engineering is to be conservative. this is a classical deadlock situation: we cannot learn how to program multiprocessors until such systems exist; a system canot be built before programs are ready.

2. The market doesn't demand them. Another deadlock: how can the market demand them, since the market doesn't even know that such a structure could exist? IBM has not yet blessed the concept.

3. We can always build a better single, special processor. This design philosophy stems from local optimization of the designed object, and ignores global costs of spares, training, reliability and the ability of the user to dynamically adjust a configuration to his load.

4. There are more available designs for new processors than we can build already.

5. Planning and technology are asynchronous. Within DEC, not all products are planned and built at a particular time, hence, it is difficult to get the one right time when a multiprocessor would be better than an existing Uniprocessor together with one or two additional new processors.

6. Incremental market demands require specific new machines. By having more products, a company can better track competitors by specific uniprocessors.

5.4.1 Existent Multiprocessors -

Figure MP gives some of the multiprocessor systems that have been built on the 11 base. The top most structure has been built using 11/05 processors, but because of improper arbitration in the processor, the performance expected based on memory contention didn't materialize. We would expect the following results for multiple 11/05 processors sharing a single UNIBUS:

#Pc	Mp	Pc. PERF	Pc. PRICE	PRICE/ PERF*	SYS Price	Price/ PERF**
1	.6	1	1	1	3	1
2	1.15	1.85	1.23	.66	3.23	.58
3	1.42	2.4	1.47	.61	3.47	.48
40		2.25	1.35	.6	3.35	.49

*Pc cost only
** Total system, assuming 1/3 of system is Pc.cost

From these results we would expect to use up to three processors, to give the performance of a model 40. More processors, while increasing the performance, are less cost-effective. This basic structure has been applied on a production basis in the GT4X series of graphics processors. In this scheme, a second P.display is added to the UNIBUS for display picture maintenance.

The second type of structure given in Figure MP is a conventional multiprocessor using multiple port memories. A number of these systems have been installed and operate quite effectively, however, they have only been used for specialized applications.

The most extensive multiprocessor structure, C.mmp, has been described elsewhere. Hopefully, convincing arguments will be forthcoming about the effectiveness of multiprocessors from this work in order to establish these structures on an applied basis.

6.0 THE ISP

Determining an ISP is a design problem. The initial 11 design was based substantially on benchmarks, and as previously indicated this approach yielded a predecessor (not built) that though performing best on the six benchmarks, was difficult to program for other applications.

6.1 General ISP Design Problems

The guiding principles for ISP design in general, have been especially difficult because:

1. The range of machines argues for different encoding over the range. At the smallest systems, a byte-oriented approach with small addresses

is optimum, whereas larger implementations require more operations, larger addresses and encoding efficiency can be traded off to gain performance.

The 11 has turned out to be applied (and hopefully effective) over a range of 500 in system price ($500 to $250,000) and memory size (8k bytes to 4 megabytes). The 360 by comparison varied over a similar range: from 4k bytes to 4 megabytes.

2. At a given time, a certain style of machine ISP is used because of the rapidly varying technology. For example, three address machines were initially used to minimize processor state (at the expense of encoding efficiency), and stack machines have never been used extensively due to memory access time and control complexity. In fact, we can observe that machines have evolved over time to include virtually all important operations on useful data-types.

3. The machine use varies over time. In the case of DEC, the initial users were sophisticated and could utilize the power at the machine language level. The 11 provided more fully general registers and was unique in the minicomputer marketplace, which at the time consisted largely of 1 or 2 accumulator machines with 0 or 1 index registers. Also, the typical minicomputer operation codes were small. the 11 extended data-typing to the byte and to reals. by the extension of the auto-indexing mode, the string was conveniently programmed, and the same mechanism provided for stack data-structures.

4. The machine is applied into widely different markets. Initially the 11 was used at the machine language level. The user base broadened by applications with substantially higher level languages. These languages initially were the scientific based register transfer languages such as BASIC, FORTRAN, DEC'S FOCAL, but the machine eventually began to be applied in the commercial marketplace for the RPG, COBOL, DIBOL, and BASIC-PLUS languages which provided string and decimal data-types.

5. The criteria for a capability in an instruction set is highly variable, and borders on the artistic. Ideal goals are thus to have a complete set of operations for a given basic data-type (e.g. integers)--completeness, and operations would be the same for varying length data-types--orthogonality. Selection of the data-types is totally a function of the application. That is, the 11 considers both bytes and full words to be integers, yet doesn't have a full set of operations for the byte; nor are the byte and word ops the same. By adhering to this principle, the compiler and human code generators are greatly aided.

   We would therefore ask that the machine appear elegant, where elegance is a combined quality of instruction formats relating to mnemonic significance, operator/data-type completeness and orthogonality, and addressing consistency. By having completely general facilities (e.g., registers) and which are not context dependent assists in minimizing the number of instruction types, and greatly aids in increasing the understandability (and usefulness).

6. Techniques for generating code by the human and compiler vary widely. With the 11, more addressing modes are provided than any other computer. The 8 modes for source and destination with dyadic operators provide what amounts to 64 possible instructions; and by associating the Program Counter and Stack Pointer registers with the modes, even more data accessing methods are provided. For example, 18 forms of the MOVE instruction can be seen (Bell et al, 1971) as the machine is used as a two-address, general registers and stack machine program forms. (The price for this generality is extra bits). In general, the machine has been used mostly as a general register machine.

7. Basic design can take the very general form or be highly specific, and design decisions can be bound in some combination of microcode or macrocode with no good criteria for tradeoff.

6.2 Problems In Extending The Machine Range

Several problems have arisen as the basic machine has been extended:

1. The operation-code extension problem--the initial design did not leave enough free opcode space for extending the machine to increase the data-types.

   At the time the 11/45 was designed (FPP was added), several extension schemes were examined: an escape mode to add the floating point operations; bringing the 11 back to a more conventional general register machine by reducing the modes and finally, typing the data by adding a global mode which could be switched to select floating point (instead of byte operations).

2. Extending the addressing range--the UNIBUS limits the physical memory to 262,144 bytes (18-bits). In the implementation of the 11/70, the physical address was extended to 4 megabytes by providing a UNIBUS map so that devices in a 262K UNIBUS space could transfer into the 4 megabyte space by mapping registers.

   While the physical address limits are acceptable for both the UNIBUS and larger systems, the address for a single program is still confined to an instantaneous space of 16 bits, the user virtual address.

   The main method of dealing with relatively small addresses is via process-oriented operating systems that handle large numbers of smaller tasks. This is a trend in operating systems, especially for process control and transaction processing. It also enforces a structuring discipline in the (user) program organization. The RSX series operating systems are organized this way, and the need for large addresses except for problems where large arrays are accessed is minimized.

   The initial memory management proposal to extend the virtual memory was predicated on dynamic, rather than static assignment of memory segment registers. In the current memory management scheme, the address registers are usually considered to be static for a task (although some operating systems provide functions to get additional segments).

7.0 SUMMARY

This paper has re-examined the PDP-11 and compared it with the initial goals and constraints. With hindsight, we now clearly see what the problems with the initial design were. Design faults occurred not through ignorance, but because the design was started too late. As we continue to evolve and improve the PDP-11 over the next five years, it will indeed be interesting to observe, however, the ultimate test is use.

BIBLIOGRAPHY

Ames, G.T., Drongowski, P.J. and Fuller, S.H. Emulating the Nova on the PDP-11/40: a case study. Proc. COMPCON (1975).

Bell, G., Cady, R., McFarland, H., Delagi, B., O'Loughlin, J., Noonan, R., and Wulf, W. A new architecture of minicomputers-- the DEC PDP-11. Proc. SJCC (1970) Vol36, pp.657-675.

Bell, C.G. and Newell, A. Computer Structures. McGraw Hill (1971)

Bell, J.R. Threaded code. COMM ACM (June 1973) Vol 16, No. 6, pp 370-372.

Eckhouse, R.H. Minicomputer Systems: organization and programming (PDP-11). Prentice-Hall,(1975)

Fusfeld, A. R. The technological progress function. Technology Review (Feb. 1973) pp.29-38

McWilliams, T., Sherwood, W., Fuller, S., PDP-11 Implementation using the Intel 3000 microprocessor chips. Submitted to NCC (May 1976)

O'Loughlin, J.F. Microprogramming a fixed architecture machine. Microprogramming and Systems Architecture Infotech State of the Art Report 23. pp205-224

Ornstein, 1972? (page 28)

Stone, H.S. and Siewiorek, D.P. Introduction to computer organization and data structures: PDP-11 Edition. McGraw-Hill, (1975)

Turn, R. Computers in the 1980's. Columbia University Press 1974.

Wulf, W.A., Bell, C.G., C.mmp: A multi-mini-processor. FJCC (1972)

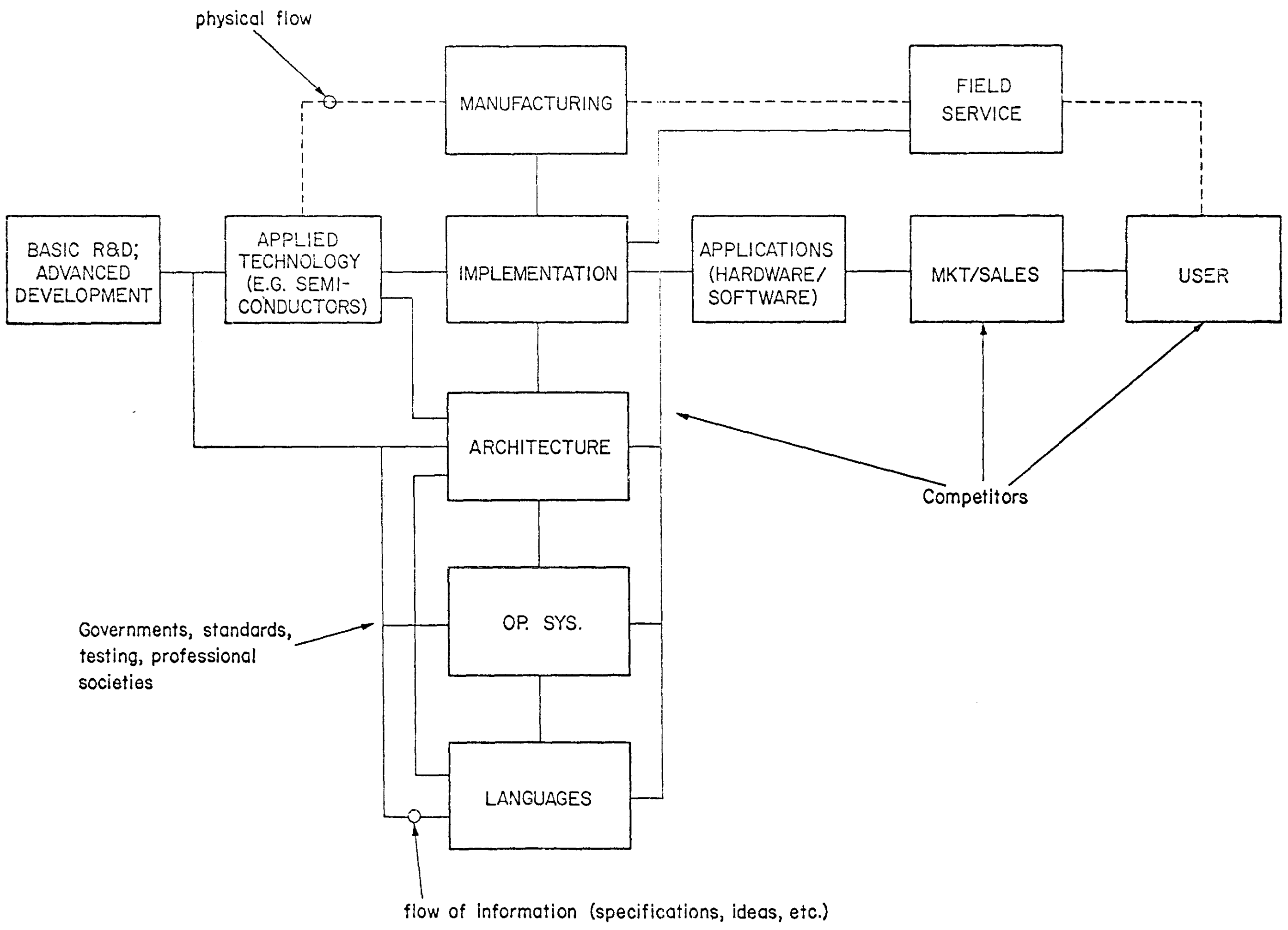
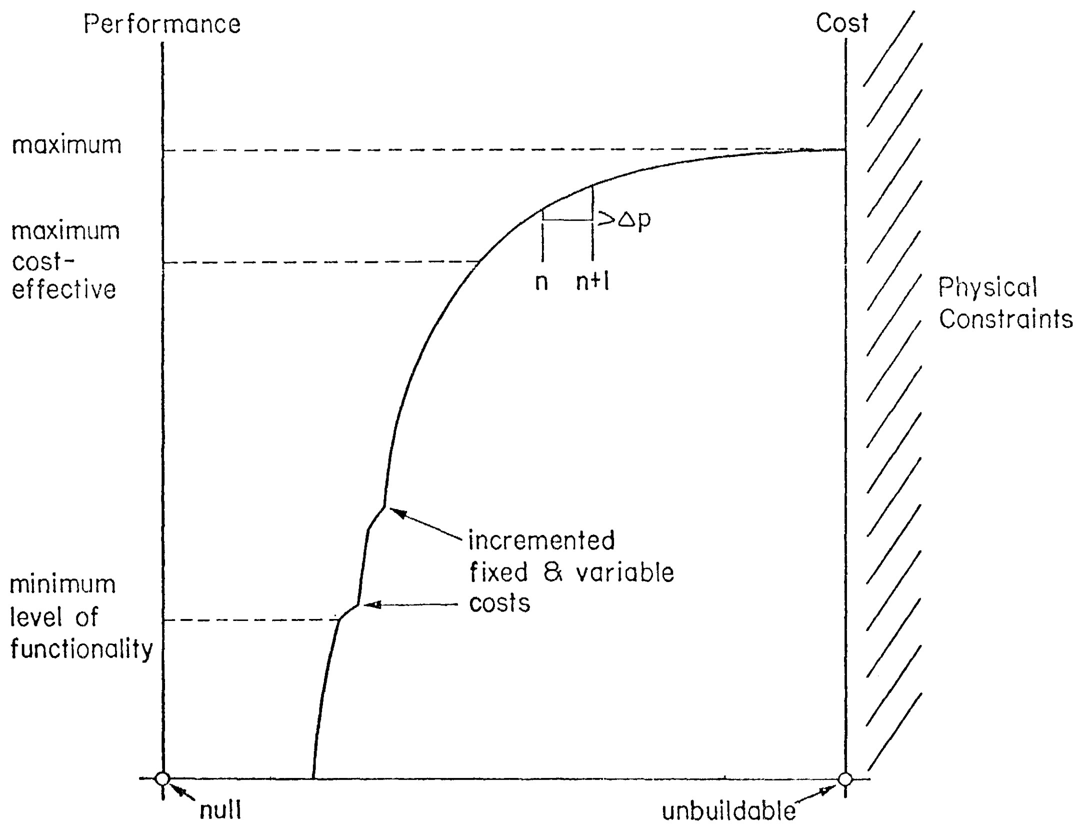

physical flow
MANUFACTURING
FIELD SERVICE
BASIC R&D; ADVANCED DEVELOPMENT
APPLIED TECHNOLOGY (E.G. SEMI-CONDUCTORS)
IMPLEMENTATION
APPLICATIONS (HARDWARE/ SOFTWARE)
MKT/SALES
USER
ARCHITECTURE
Competitors
OP. SYS.
Governments, standards, testing, professional societies
LANGUAGES
flow of information (specifications, ideas, etc.)

FIGURE ORG. STRUCTURE OF ORGANIZATION AFFECTING A COMPUTER DESIGN

Fig. DS DESIGN STYLES (IDEOLOGIES) IN TERMS OF COST AND PERFORMANCE



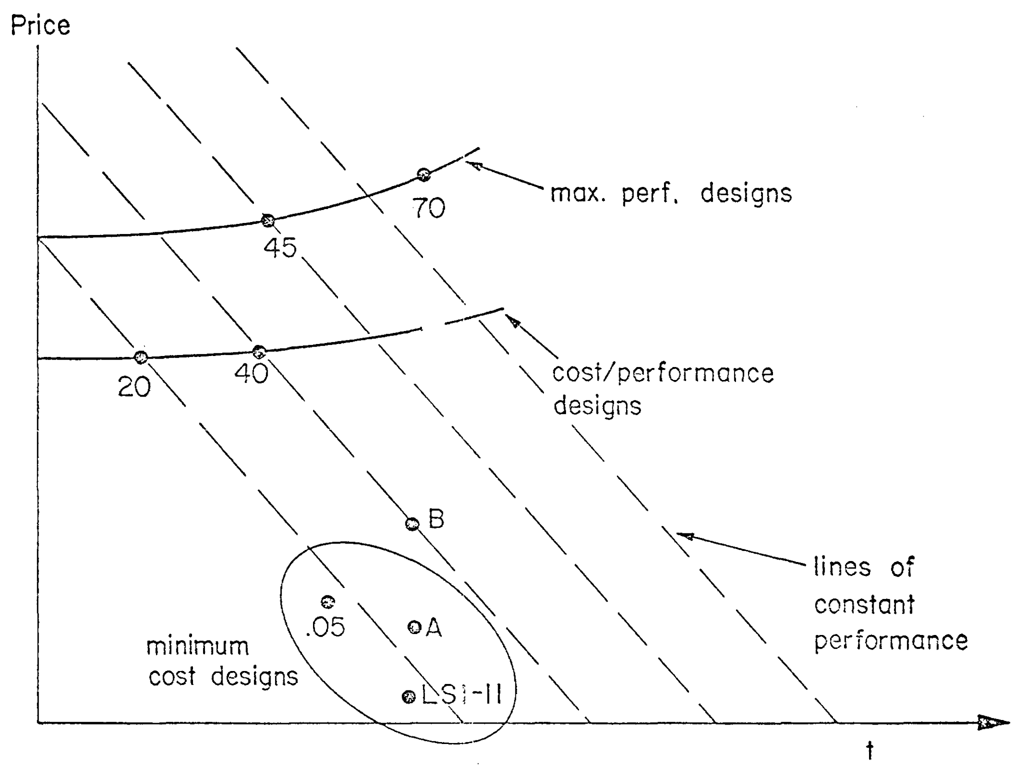
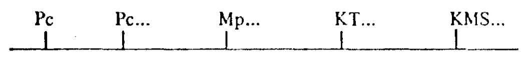
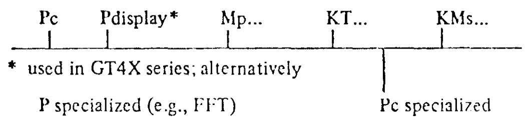


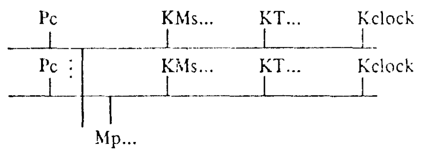
Fig. MODELS PDP-11 MODELS PRICE VERSUS TIME WITH LINES OF CONSTANT PERFORMANCE



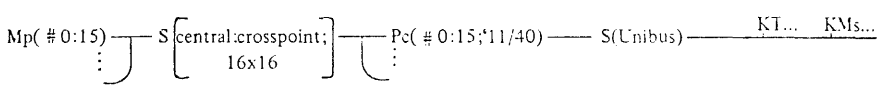
a. Multi-Pc structure using a single Unibus.


Pc
Pdisplay*
Mp...
KT...
KMs...
* used in GT4X series; alternatively
P specialized (e.g., FFT)
Pc specialized

b. Pc with P.display using a single Unibus.



c. Multiprocessor using multiport Mp.



d. C.mmp CMU multi-mini-processor computer structure.

Figure MP Multi-Processor Computer Structures Implemented using PDP-11

A PMS LEVEL LANGUAGE FOR PERFORMANCE EVALUATION MODELLING (V-PMS)

Helmut Kerner, Werner Beyerle
Institut fuer Digitale Anlagen
Technical University, Vienna

Summary

A comparison of Register Transfer level modelling and V-PMS is quite indicative. While RT-modelling approaches a restricted goal, viz. a hardware structure capable of performing a few hundred algorithms, a V-PMS level model has a complete computer system as its target, i.e. a composite of hardware structures of the RT-level complexity cooperating under the control of an operating system in the execution of a load.

In order to construct a language suitable for describing the hardware as part of a total PMS-level model for performance evaluation, the original form of PMS was substantially changed by providing an expanded set of building blocks with corresponding definitions. This language, with its clearly defined functions and performance data, its unambiguous communication blocks and rules for interconnections, provides a human reader with a clear understanding of the performance of components and of their internal communication within the computer system, and links the hardware part to a model of an operating system (to be supplied at a later time). For computer readable system specification a syntax for connecting the above symbolic components is proposed. A description of the CDC Cyber 74/CDC 6600 system examplifies the use of the proposed language and its merits for building performance evaluation models.

1. Introduction

The bulk of papers on Computer Descriptive Languages[1] treats the design of digital systems on the Register Transfer level of detail. Some provide macro-facilities within an RT-level description language as an upward extension of their language into the PMS region[4]. Others regard the formal descriptive facilities of their language as universally suited for any level envisioned.[5] Only few regard PMS as a second autonomous level of abstraction necessary to satisfy the needs for documentation of a complete computer system in unambiguous, concise and standardised form and for a well structured data base of technology and configuration data, both oriented toward man in his role as a designer or analyser of a system for a given purpose[2,3]. PMS in its present state is directed toward this important goal. Performance evaluation of computer systems is another, different purpose to be supported by computer system descriptions on the PMS level. A model of performance evaluation consists of three parts (Fig. 1).

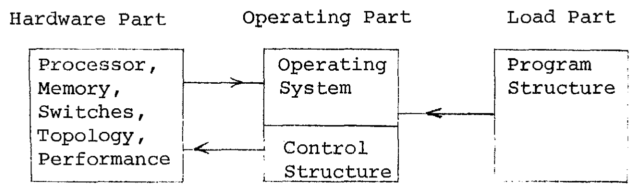


Fig. 1 PMS-level Model

(1) A model of the Hardware Part, describing the behavior (functions, responses, performance) of the hardware components, but not their internal construction (registers, instruction set, etc.).

A topological description of their interconnections is a necessary part of this model.

(2) An Operating Part, consisting of the operating system software, and the hardware (control structure) used for its execution (which may be part of the general hardware complex).

(3) A Load Part, i.e. the model of user programs.

This paper presents a language constructed for the description of the Hardware Part of an Evaluation Model and of the interfaces to the Operating Part. In its computer readable form it should be suitable as an input to a performance evaluation model (or simulator) by which its Hardware Part will be specified.

Obviously PMS seems to be a candidate for a hardware modelling language, provided it is redesigned from a man-understandable short notation into a machine readable language. More precisely, the following requirements must be met for performance evaluation:

(1) The sysmbolic components of the hardware system, (P,M,S,L), must be defined by a standard set of well defined performance parameters.

(2) Explicit and unambiguous rules must exist specifying the information transfers permissible between components. These must be formally stated.

(3) All detail not pertinent to performance evaluation (such as the number of internal registers, of subprocessors within a CPU, technology data, etc.) should not be part of the symbolic component definitions. Many important performance parameters must be introduced on a higher level (e.g. kernel times in lieu of instruction sets or times). Others must be added (e.g. concurrency within switches, etc.).

(4) Interfaces to the Operating Part (function selection, responses, etc.) are to be added.

These reqirements imply such deviations from the PMS notation that we decided to coin a new (but similar) acronym for the proposed language, namely "V-PMS" (Viennese-PMS) in order to avoid confusion with the original PMS notation, which is similar but not a subset of V-PMS.

2. V-PMS Language Definitions

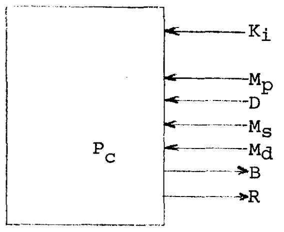
Hardware structures can be defined using four categories of V-PMS symbolic components: Processors, Memories, Internal Communication Components, and Peripherals. Each component is defined in the format:

<component symbol> (<attribute symbol>,<attribute value>,.....)
(<Action order name>,...<response name>,...)

The interface signals to the Operating Part (second parenthesis) will be omitted in graphical representations. Semantic definitions of components, performance attributes, action orders and responses are presented in list form and verbally explained.

2.1 Processors P

2.1.1 Central Processor P_c (T, T_{ki}, W) $(Ki, Mp, D, Ms, Md, B, R)$



K_i	Selected kernel functions from Operating Part
M_p	Program locations
D	Data magnitude
M_s	Source locations
M_d	Destination locations
B	Busy with K_i
R	Ready

Performance Attributes	Symbol	Unit
Cycle time	T	µs
Word Width	W	bit
Kernel time i	T_{ki}	s

A CPU ("P_c") could e.g. be defined by its instruction set. In staying with the purpose of the language, we chose, however, to characterise its performance by the time required to execute a set of kernels $K_1 \ldots\ldots K_i$, representing typical tasks such as matrix inversion, sorting, etc. Each one of these kernels is associated with a basic execution time per unit of data (e.g. sorting in memory of a block of n records of fixed structure). The "action-order" specifies the kernel type K_i, the memory block M_p holding the program, the number of records D, as well as the memory blocks M_s and M_d containing the source data and the result data, respectively. The basic execution time is modified depending on the type of memory and buffers used for data and program.

2.1.2 I/O-Processor $P_{i/o}$ (same as P_c) (similar to P_c) An I/O Processor "$P_{i/o}$" differs from the general purpose processor "P_c" only with regard to the "kernel" functions and the program location. Its kernels are a few distinct I/O programs controlling the transfer of data from a class of devices such as discs, tapes, or other low speed devices via controllers; its program is assumed to be located in the $P_{i/o}$. These processors are often called "channels". The action-order to an I/O processor will select an I/O kernel and identify an attached controller and peripheral.

2.1.3 Controller K (U,B) (B,R)

Performance Attributes	Symbol	Unit
Usage (Blockmultiplexer blx, Bytemultiplexer byx, Selector controller sel)	U	-
Blocksize (if blx was specified)	B	words

Controllers are trivial processors needed only for the activation of various peripheral functions. Their attributes comprise their usage time and the blocksize to be transmitted by one request on their action order line. They don't have any performance parameters of their own, because their operation is determined by the peripherals attached to the controllers.

On this level of detail the Operating Part is forced to provide a very detailed model containing a full sequence of action orders and responses. Their is also an alternative, global description of the I/O Processor with kernel functions implicitely defined by the file structure definitions contained in the Workload Model.

2.2 Memories M

2.2.1 Central Memory M_p (T,W,C,I)

Performance Attributes	Symbol	Unit
Access Time	T	µs
Word Width	W	bit
Capacity	C	words
Interleaving	I	-

2.2.2 Secondary Memory M_s (T,W,C)

Performance Attributes	Symbol	Unit
Access Time	T	µs
Word Width	W	bit
Capacity	C	words

Memories are devices which are random-adressable on a word or multiple-word basis with location-independent access times. Their block representation contains the attributes: The capacity referring to each unit depicted, and the attribute "I" for interleaving, stating how many such units can be accessed simultaneously within one memory cycle. An interleaving factor of I = n makes therefore a group of n memories appear maximally n times faster than each individual memory unit. Since it is intended to characterize only the performance of a system and not its detailed structure, no differentiation is made between different means of access such as parallel lines, synchronous or asynchronous time multiplexing of a single line, etc.

2.2.3 Buffer B (T_{tot}, T_{part}, W_{tot}, W_{part})

Performance Atributes	Symbol	Unit
Access Time for complete word	T_{tot}	µs
Access Time for partial word	T_{part}	µs
Word Width of complete word	W_{tot}	bit
Word Width of partial word	W_{part}	bit

Buffer memories are defined as small memories capable of performing a format transformation of their content such as a serial/parallel conversion, FIFO or FILO (stack) organisation, etc.

2.3 Intercommunication Elements IC

2.3.1 Switches S (I,P,CC,E)

Number of incoming lines	I
Number of exiting lines	E
Path-concurrency	CC
Priority model (if necessary)	P

There are two types of switches, one of them connects two groups of components by a one-to-one correspondence (Fig. 2). The parameters I and E give the number of components in each of the groups, while CC states the number of concurrent paths through the switch. For this type of switch the relationship $CC \leq \min(I,E)$ holds true. The path in the switch assumes the "usage" parameter of the two connected lines. The general case of this switch S (I,CC,E) is the crossbar switch. The switch with a path concurrency CC = 1 is known as "bus". A further special case is a multiplexer or a demultiplexer, i.e. a switch with only one incoming or outgoing line (I = 1 or E = 1) respectively. In order to simplify the graphical representation of a system, multiplexers and demultiplexers can be omitted. Should there be a possibility for conflict situations to arise (e.g. "I" incoming lines competing for a smaller number "CC" of internal connections), a standard form of conflict resolution uses a "first come first serve" switch discipline. Otherwise the form of priority will be explicitely stated. The most common priority, which provides service in the numerical order of the connection label, will be symbolized by the p-value P = n.

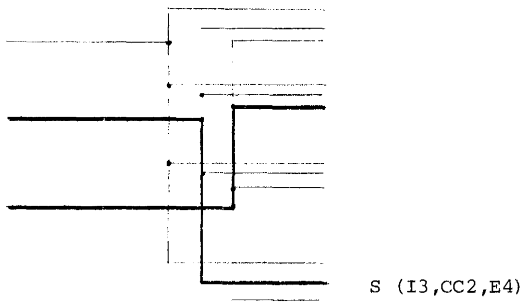


Fig. 2 Switch Schematic

The second type of switch is similar to a multiplexer, however, with the difference that it can simultaneously transmit from one input line into several (viz. CC) of the E outputlines.

The relationship CC$\leqslant$min (i,E) is true only in the direction from several outputlines (E>1) to a single inputline, i.e. CC = 1 in this direction. In the other direction the relationship 1<CC$\leqslant$E is true. Fig. 3 shows a diagram explaining the corresponding symbol.

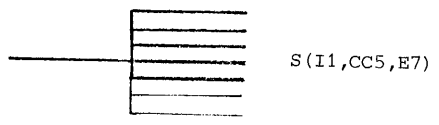


Fig. 3 Direction Dependant Switch, Information Replicating

2.3.2 Links L (R,W,CC,U)

Performance Attributes	Symbol	Unit
Transmission Rate	R	words/s
Word Width	W	bit
Concurrency	CC	-
(default value CC=1		
Usage	U	-
Simplex s, halfduplex h; default value h)		

For graphical representation the symbol L can be replaced by a line with the attributes written above the line. The following two representations are equivalent:

L (R10^4, W12, CC2, Ud)   2d, 10^4 x 12

A Link "L" indicates primarily which components are connected, i.e. it is normally assumed that the performance of the line is equal or better than that required by the adjoining components. Performance parameters (as listed) will only be shown if this assumption is not true and the line constrains the performance of the system. In graphical form the attributes can be stated above a line as shown above. The concurrency parameter CC states the number of physical lines of a given word width, rate and usage, which can be simultaneously used. For any other means of communication (such as time multiplexing) the transmission performance will be expressed by an equivalent number of physical lines. The default value of CC means a single line (CC = 1). The usage of the line is defined as halfduplex or simplex by the corresponding symbols shown above. In order to keep the determination of line concurrencies simple, duplex lines are prohibited and will be replaced by a pair of simplex lines. A missing usage parameter indicates halfduplex (h).

2.4 Peripherals T (.....) (B,R)
(every peripheral is able to give a ready and a busy signal)

	Performance Attributes	Symbol	Unit
Console CO	-		
Teletype TTY	Transmission Rate	R	characters/s
CRT- Terminal CRT	Transmission Rate	R	characters/s
	Capacity	C	character
	Drawing Speed	V	cm/s
	(for graphic Terminals only)		
Lineprinter LP	Print Rate	R	lines/min
	Line Width	W	characters/ line
Card Reader CR	Rate	R	cards/min
	Card Width	W	columns/card
Card Punch CP	Rate	R	cards/min
	Card Width	W	columns/card
Paper Tape	Rate	R	characters/ min
Reader PTR	Tracks	S	-
Paper Tape	Rate	R	characters/min
Punch PTP	Tracks	S	-
Magnetic Tape MT	Transmission Rate	R	characters/s
	Tracks	S	-
	Density	D	characters/cm
Rotating Mass	Transmission Rate	R	characters/s
Storage RMS	Capacity	C	characters
	Number of Cylinders	NZ	-
	Access Time	T	ms
	(one track/average/all tracks)		
	Rotational Time	TR	ms
	Positioning	POS	-
	(default value or 0 means "none", else POS1)		

The definitions of peripheral devices and their attributes are self-explanatory.

The Structure Part of the V-PMS Part is completely defined by its links and switches, whether explicitely or implicitely defined. A further simplification of the system description can be achieved by a short hand notation in form of a bracket symbol for replication. Any components, with the exception of links and switches, as well as any substructure composed of such legal components can be iterated. The following convention must be obeyed for the sake of clarity: switches within a replicated substructure must not be omitted even if such would normally be permitted (as we stated for multiplexers and demultiplexers). The number of replications is indicated by the number preceeding the bracket. The actual system can be distinguished from maximally allowed configuration by setting the number of replications for the latter in parenthesis (Fig. 4). Of course, the graphical repetition of components is a second form of replication. A distinction between the actual system and the maximal configuration is made by connecting the actual components with full lines and the additionally allowed components with dashed lines.

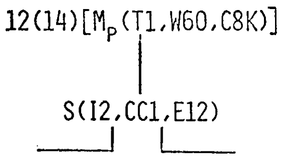


Fig. 4 Component Replication

3. Application Description of the CDC Cyber 74/CDC 6600

In order to demonstrate the features of the proposed V-PMS language and to evaluate its merits we present a description of the Cyber 74 (CDC 6600) system as an example:

A discussion of a few selected details of the system diagram (Fig. 5) may underscore these general remarks. At first we concentrate on the area labeled "A" in Fig. 5. According to the definitions, we recognize in the brackets a memory-switch combination. Each memory (with a 1 µs cycle time and 4 K words of 12 bit width) is connected to a switch capable of transmitting one word (12 bits) at a time (CC = 1) to one of ten output lines. This combination is ten times replicated. A group of 10 lines, one from each M-S combination, reaches the I/O processor "Pi/o". In the same way, another group of 10 output lines, indicated as simplex lines, connects through a switch to a group of 5 buffers. We recognize a similar group of 10 input lines reaching the M-S combination. Seventy halfduplex lines connect each of the 10 M_P's to each of seven controllers of peripheral equipment. The representation in V-PMS enables the observer (human or machine) to recognize the maximal possible information flow between the 10 (peripheral) memories and the connected devices. The

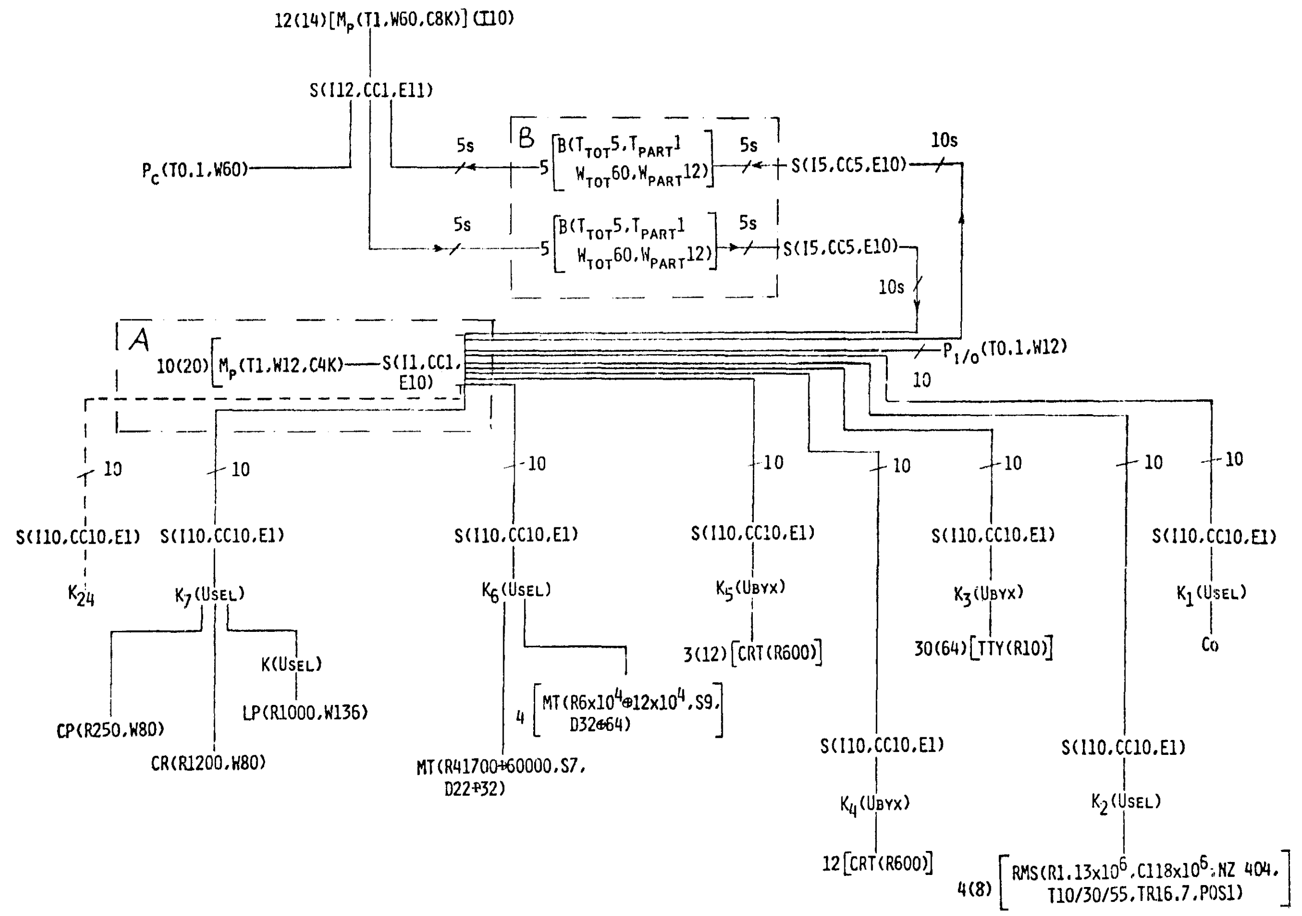


Fig. 5 Cyber 74 Diagram, Revised & Expanded PMS Language

original PMS representation does not reveal these for performance evaluation important facts.

The same configuration can be represented in a slightly different fashion (Fig. 6). Here each memory is connected to one I/O processor within the bracket.

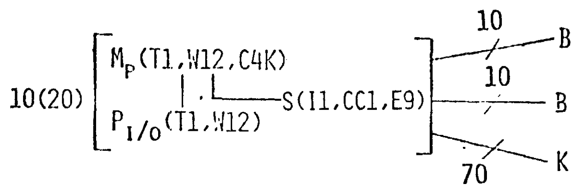


Fig. 6 Peripheral Memory - Processor Complex, Version II

Compared to Detail A, we indicate 10 I/O processors in lieu of one, however, with a ten times slower cycle time (and kernel times). This second representation shows another feature of the machine, namely the fact that each I/O program is restricted to 4 K of memory, while according to Detail A, I/O programs could exceed 4 K. With respect to performance the two representations are equivalent. The second Detail "B" of Fig. 5 shows the path between the previously discussed peripheral memories 10 $[M_P]$ of 12 bit word width and the central memory 12(14) $[M_P]$ with 60 bit words. The function and performance of the component group in the path can immediately be read from the V-PMS representation: five peripheral memories can simultaneously transmit via 5 simplex lines into a group of 5 buffers. Each one of these 5 buffers transforms 5 twelve bit words into one 60 bit word within 5 μs. A second block of 5 buffers disassembles 60 bit words into 12 bit words. Five assemblies and five disassemblies may maximally occur simultaneously.

A glance over the total system (Fig. 5) allows assessing the merits of the proposed PMS-level language. Despite the host of detail supplied in the graph, the general architectural features are immediately visible, yet the graph supplies all the information pertinent for performance evaluation, which is missing in the original PMS representation.

Appendix: Syntactical Definition

<configuration>::=<CA>{< continuation >}
<CA> ::=<CB>| integer(integer) [<content>]
<CB> ::=<P>|<M>| <T>
<content> ::=< CB>{ <continuation A>}
<P> ::=< P_c >|< $P_{i/o}$ > |<K>
<M> ::=< M_p> |< M_s >|<B>
<T> ::=< TTY> |<CRT>| <LP> |<CR> |<CP> |<PTR> |<PTP>|
        <MT >|<RMS>| <Co>
<continuation A> ::= < connection A >|<parallel> <CD>
                  {<continuation A>} |< parallel>
                  <parallel> <CD> {<continuation A>}
<CD >::= <CB> |<S>
<connection A> ::= <L> <CD>
<continuation> ::= <connection>|<parallel>< C>{<conti-
                                   nuation>}|
                 <parallel>< parallel>< C>{ < conti-
                                   nuation>}
<C> ::= <CB> |<integer(integer) [<content>] |<S>
<connection> ::=< L>< C>
<parallel> ::=<$\overbrace{\underbrace{L}}$>

Semantic Definition: The meta language symbol "$\overbrace{\underbrace{\quad}}$ " means continuation through parallel pathes.

References

/1/ Su, S.Y.H., A Survey of Digital Hardware Descriptive Languages; Proc. of the Workshop on Hardware Descriptive Languages. pp 144, 1974

/2/ Bell, G. and Newell, A. Computer Structures : Readings and Examples; McGraw Hill, 1971.

/3/ Knudsen, M.J., PMSL, An Interactive Language for Systemlevel Description and Analysis of Computer Structures; Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittscurgh, Pa. April 73

/4/ Piloty, R., RTS I (Registertransfersprache); 3. Aufl., Institut für Nachrichtenverarbeitung, TH Darmstadt, 1969

/5/ Lee, J.A.N., VDL - A Definitional System for all Levels; Proc. 1st Annual Symp. on Computer Architecture, Computer Arch. News, Dec.1973, Vol.2, no. 4, pp 41-48.

A DESIGN TOOL FOR THE MULTILEVEL DESCRIPTION AND SIMULATION OF SYSTEMS OF INTERCONNECTED MODULES

M. MOALLA - G. SAUCIER - J. SIFAKIS - M. ZACHARIADES
ENSIMAG - B.P. 53 - 38041 GRENOBLE-FRANCE

ABSTRACT : We suggest a methodology and a language to permit the study of a system's behavior (functional validation, evaluation of global performances, critical situations). Every system is regarded as an interconnection of communicating modules functionning in a synchronous or asynchronous manner. The control section and the data section of each module are described separetely in terms of respectively non-procedural and procedural sublanguages.
Key words : Data and Control Section of a system, Petri nets, procedural and non-procedural languages, Register Transfer Languages, High Level Languages.

INTRODUCTION

The object of this work is the elaboration of a design tool for complex systems regarded as the interconnection of communicating modules functioning in a synchronous or asynchronous manner. Modules are functional subsets performing particular functions for the whole system such as memory units, processors, channels, peripheral devices, etc...
This tool must permit, given a high level initial description (architectural definition), the study of the system's behavior, of its performances and the detection of critical situations such as conflicts, deadlocks and thrashing.
In part I, we give the characteristics which a tool responding to those objectives must possess. In particular, the necessity of two distinct types of descriptions appears for the control and data sections of a module. Moreover, in order to permit performance evaluation, time has been introduced as well as the possibility to create systems by interconnecting standard predefined modules.
In part II, a rough description of the language is given. Part III gives an example illustrating the application of the language to the description of interleaved memory banks with multiple entry points and two domains of application actually under study.

PART I

COMPUTER HARDWARE DESIGN LANGUAGES : A CRITICAL REVIEW

I.1. Description Levels of a System : We can generally distinguish 3 description levels for a system [1] :

- behavioral description in which properties of the system are specified in terms of the input/output relations. These descriptions are closer to conventional programs and they are not in the scope of this study.

- functional description in which the system is described as an algorithm in terms of its memory elements (variables). Operators used in the description may not be hardware primitives.

- structural description represents the system in terms of its hardware components, and requires a complete knowledge of logic design.

I.2. System decomposition : Given a system described by an algorithm, one can easily decompose it into two sections (sub-systems) connected as in figure 1. [2].

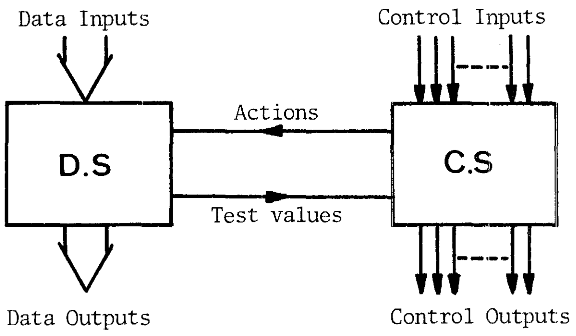


Fig.1.

- The Data Section (D.S) contains the set of a data registers and operators which are used either to calculate test values on the data or to transform them.

- The Control Section (C.S) ensures the sequencing of the operations to be executed in the D.S. It may be represented as an automaton receiving among its inputs test values depending on the D.S state. To each state of the C.S is associated a set of actions executable simultaneously (compatible) in the D.S. Setting the C.S at a state is interpreted by the D.S as the order to execute the actions associated to this state. The C.S goes from a state s to a state s' when the D.S has accomplished the execution of the actions associated with s and if the condition corresponding to the transition ss' is verified.

I.3. Computer Hardware Design Language With Respect to the Description Levels

It the case of a structural description of a system, the language must permit the description of the dialog between the two sections of the hardwired system. This is generally done in the following way : to every state of the C.S is associated a label of the program. This label names a set of instructions describing elementary actions and/or conditional actions. The elementary actions are those corresponding to the state referred to by the label. Conditional actions determine the successor state in the C.S. Such a description is facilitated by the use of non-procedural languages such as CASSANDRE [3] DDL [4] CDL[5]. Languages of this type are very convenient for the description of control structures.
In the case of a functional description, one can ignore how data transformations are performed and be interested only in the values of certain memory elements at precise instants. Thus, we can associate to the states of the C.S not only simple actions but procedures, as long as the evolution of the C.S does not depend on the values of the internal variables during the execution of the procedure. This approach permits a more global description of the system's behavior since it avoids non significant details of the C.S.
Among Computer Hardware Design Languages (CHDL), those often called procedural [1] are more adequate for descriptions of this type, for example APL [6], APDL [7], SIMULA [8].

I.4. A critic of the existing CHDL's for the proposed application

We do not intend to review, here, all the existing CHDL's. A complete classification of these languages, as well as interesting critics concerning their use, can be found in [1], [9], [10]. This study is limited only to the possibilities of using such languages in order to satisfy the objectives mentioned in the introduction. A language responding to these objectives has to possess the 3 following characteristics :

a) Be a non-procedural language in order to allow the representation of simultaneous actions in the description of control mechanisms whatever the level of description may be.

b) Provide the facilities of an algorithmic language permitting a powerful and concise description of data handling and computation.

c) Provide the possibility to manipulate software entities representing modules of the system, as far as their description, duplication and synchronized execution are concerned.

For the existing CHDL's, properties a) and b) seem to be contradictory. In fact, non-procedural languages are generally R.T.L.'s imposing a description very close to the harwired realisation. It is evident that such languages do not satisfy characteristic c).

Conversely, procedural languages more or less satisfy characteristics b) and c). Among those satisfying b), APL seems to be the most adequate to the problem,given the richness of its data manipulation operators and the facility to handle arrays. However one of its major drawbacks lies in the difficulty to create configurations from standard modules and to describe parallelism. The characteristic c) is partially satisfied by languages with synchronization primitives and module duplication. In SIMULA [8], the primitives "class" "sub-class", "detach", "resume" and "simulation class" are used essentially for this purpose. Nevertheless, all the definitions that are necessary for the arrangement of links between modules (represented by classes), the synchronization and the control of their execution are left to the programmer; thus requiring a good knowledge of programming techniques. And anyway, SIMULA being a high level language, it does not allow structural descriptions (R.T. level for instance).

Furthermore, the distinction between C.S and D.S is adopted by the majority of the existing CHDL's as far as the analysis and the description of the system are concerned. However, this distinction appears much less clearly once the program is written. Particularly when faced with a complex system with many levels of parallelism, the imbrication of the control and data structures makes the program hardly readable and modifiable.

PART II

PRESENTATION OF THE LANGUAGE

II.1. Methodological Aspects

After the critics formulated in the last paragraph, the approach adopted in the conception of this language becomes clearer. We shall use two sublanguages, respectively for the description of the two sections C.S and D.S. The sublanguage used to describe the C.S is non-procedural and permits, by methods exposed later, the description of synchronous or asynchronous control structures [11].
The manipulation of variables in the D.S is performed by a set of procedures; these procedures are described by a procedural sublanguage. The C.S of a module is initially represented by a graphic model derived from Petri-nets [12] given in the following paragraph . The translation of the graphic model into the language is done in a simple and direct way.

In order to permit the modular structuration of a program, three primitives have been introduced.One for the declaration of standard modules, one for the creation of a system by calling standard modules and one for their interconnection. Two modules are connected by confounding their respective interface variables. Finally, an execution time is associated with each procedure in order to have the possibility to study system performances and critical situations. Execution time is either fixed in advance or calculated dynamically as soon as a procedure is activated.

II.2. Mathematical Model for the Description of the Control Section

We give here a model for the description of control structures with parallel asynchronous evolutions. This model, actually under study [13], is as general as Petri-nets [12] but it allows a less constraining and more concise description of a system. Its drawback is that critical situations such as deadlocks, conflicts, determinacy are easier to detect in a Petri-net than in this model.

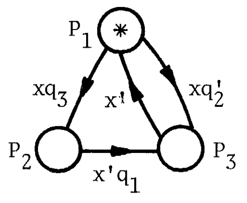
Definition 1 : A Parallel Process Control Network (P.P.C.N.) [13] is a quintuple $R = (X,P,Q,f,P^0)$ where :

- $X = \{x_1,x_2,\dots,x_m\}$ is a finite set of input variables
- $P = \{p_1,p_2,\dots,p_n\}$ is a finite set of objects called places
- $Q = \{q_1,q_2,\dots,q_n\}$ is a set of boolean variables in bijection with places
- f is a mapping, $f : P \times P \to \mathcal{F}(Q,X)$ where $\mathcal{F}(Q,X)$ is the set of boolean functions on Q and X; in addition f is such that $f(p_i,p_i) = 0, \forall p_i \in P$.
- P^0 is a set of initial places ($P^0 \subseteq P$).

With a P.P.C.N , one can associate a labelled digraph having as vertices the set of places and such that for every couple (p_i,p_j), $(f(p_i,p_j) \neq 0$, there exists an edge from p_i to p_j labelled by $f(p_i,p_j)$.

Example : $R = (\{x\}, \{p_1,p_2,p_3\}, \{q_1,q_2,q_3\}, f, \{p_1\})$
where f is defined by the following table :

f	P_1	P_2	P_3
P_1	0	xq_3	xq_2'
P_2	0	0	$x'q_1$
P_3	x'	0	0



Definition 2 : We define a token as the object having the following properties :

a) A place can contain one token at most. A token exists at the place p_i at time $t \Longleftrightarrow q_i(t) = 1$.
b) Every initial place contains a token.
c) A transition occurs from a place p_i containing a token to every place p_j such that $f(p_i,p_j) = 1$. When transitions occur from a place p_i to places $p_j,p_k,\dots$, the token is removed from p_i and a token is put in each of the places $p_j,p_k,\dots$

A token contained in a place is represented by a point within the circle representing the place.

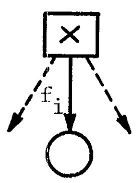


Fig.2a.

Définition 3 : A source place is a place always containing a token. We represent such a place by a square (figure 2.a). Generally, a system possesses an initialization procedure permitting it to be set to a particular initial configuration This initialization corresponds to a set of initial places of the graphic representation. In case several sets of initial places are possible depending upon input conditions, it is convenient to use this type of place (see following example).

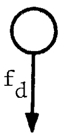


Fig.2b.

Also, it is often useful to have the possibility to express, for a system, deactivation conditions which can be expressed by a combinatorial function. For this reason, we permit, in the graphical representation, edges attaining no place (Fig. 2b). The firing of a transition of this type implies the disappearance of the token of its input place.

Note : A Petri network can be represented by a P.P.C.N. as suggests fig. 3. In fact, a Petri network is a P.P. C.N. having as labels functions of the form $\alpha(X) \prod_{j \in J} q_j$ where $\alpha(X)$ is a boolean function of the input variables and J is a subset of $\{1,2,\ldots,n\}$. For this reason, a P.P.C.N permits a less constrained and more concise description than Petri-nets.

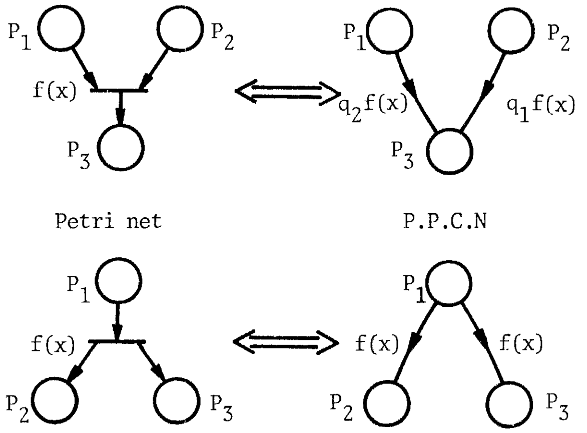


Fig.3.

Let us, for example, represent the evolution : "A token reaches p_1 if a token was at the preceeding instant at p_2 and p_3 but not at p_1".

In order to describe such an evolution by a Petri-net, one has to create one place p_1^* "complementary" of the place p_1; that is, P_1^* is a place containing a token if and only if, p_1 does not have one. We are then obliged to increase the number of places with respect to the number of places of the P.P.C.N representing the same evolution (Fig. 4).

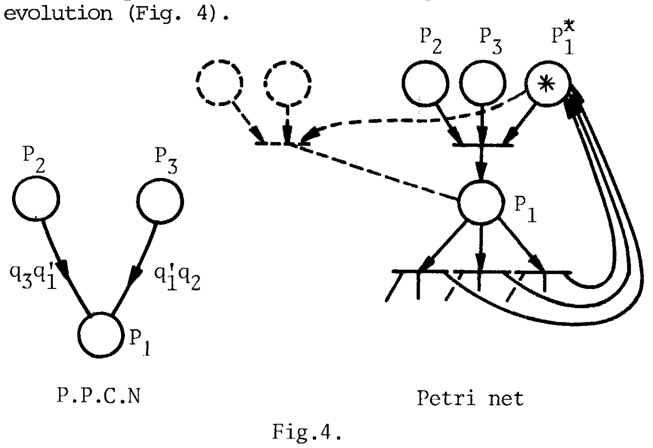


Fig.4.

The following example illustrates the application of the P.P.C.N's.

Example : We want to describe a system controlling the traffic through a one-track railroad tunnel which may be used by trains arriving in opposite directions (Fig. 5). We dispose of :

a) Pulse signals
- x_1 and x_2 indicating that a train approaches the tunnel in the directions 1 and 2 respectively.
- x_1^* and x_2^* indicating that a train has just entered the tunnel in the directions 1 and 2 respectively.
- x_3 and x_4 indicating that a train has passed through the tunnel in the directions 1 and 2 respectively.

b) Two lights (v_1,v_1'), (v_2,v_2') at the tunnel entrances indicating whether a train is allowed to pass through.

Let the control system be such that :
- $v_1 = 1$ and $v_2 = 1$ if there is no train passing through the tunnel or waiting to pass.
- $v_1 = 0$ and $v_2 = 0$ if there is a train passing through the tunnel.
- A train waiting in the direction 1 has priority over a train waiting in the direction 2. We suppose that there may be only one train waiting in each direction.

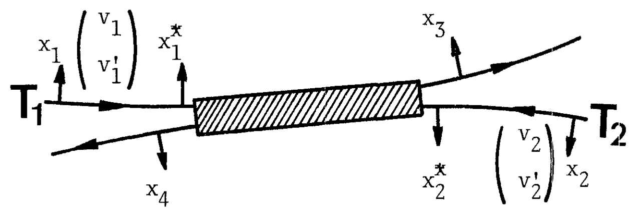


Fig.5.

The P.P.C.N. and the Petri network representing the system are given in Fig.6.The variables associated to the places are given the following interpretation :
- a_i : a train is waiting in direction i
- p_i : a train is allowed to go through the tunnel in direction i, but it has not entered yet
- t_i : a train is passing through the tunnel in direction i
- v_i : green light allowing the train to traverse in direction i.

Representation of the C.S by a P.P.C.N

In order, to represent the C.S of a system by a P.P.C.N, we generally associate to a place a list of actions described by a procedure to be executed by the D.S. During the execution of those actions, all the transitions emanating from the corresponding place are inhibited; they become enabled at the end of the procedure's execution. The evolution from a place depends on test values returned from the D.S and on the state of the C.S. The method described above does not correctly work if we associate procedures with the input places of a "join" transition. With P.P.C.N's, join transitions are not as explicit as with Petri nets, but it is easy to detect them by a very simple method [14].

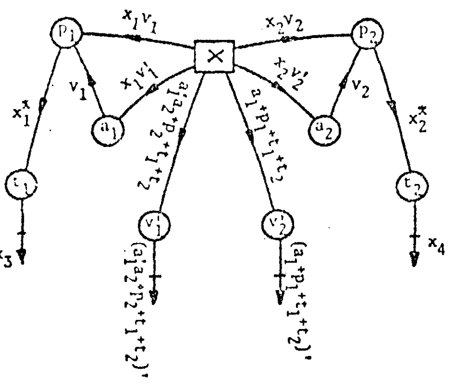
II-3. Presentation of the language

II.3.1. General structure of the program :
A program is composed of 3 parts.

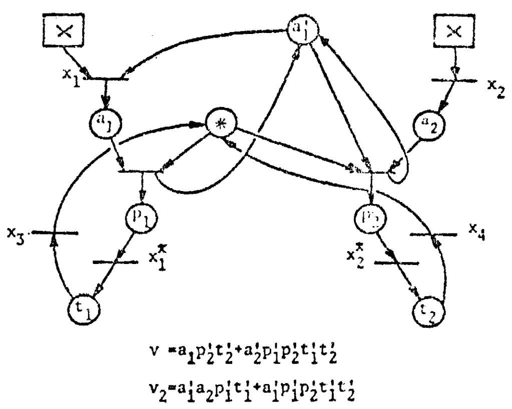
α) Definition of standard modules (library of modules used to build up the system)

β) Description of the interconnection of such modules in order to implement a system

γ) Initialization of the system and specification of the input sequence for which the system is studied.



R.C.P.P



Petri Net

Fig.6.

We shall give explanations for each of those parts delimited as follows :

```
     ┌ DCLMODTYPE
α)   │    :
     └ ENDMOND

     ┌ DCLCONF
β)   │    :
     └ END

     ┌ INIT
γ)   │    :
     └ END
```

II-3.1.1. Description of a Standard Module

It contains the D.S and C.S description as shown below :

```
DCLMODTYPE  MODEL (INTV1,......)
  DCLPROC  P1                 ⎫
     TEMP 5                   ⎪
     INTV1 ← 3                ⎪
       :                      ⎪
  END                         ⎬ Data Section
  DCLPROC                     ⎪
     t ← ?9                   ⎪
     TEMP t                   ⎪
       :                      ⎪
  END                         ⎭
       :
  DCLCTRL  SYN                ⎫
       :                      ⎬ Control Section
  ENDMOD                      ⎭
```

a) Data section : Instructions used in the description of a procedure are a subset of APL's instructions. To these instructions is added the primitive TEMP whose parameter indicates the execution time of the procedure. This parameter may be either a constant on an expression on the variables of the procedure.

b) Control section : The primitive DCLCTRL indicates the beginning of the description of the C.S. The keywords SYN and ASYN following DCLCTRL specify the type of evolution in the control structure, respectively synchronous and asynchronous. The basic instruction used for the description of the control section has the following format : P_i IF F_{ij} THEN P_j, where F_{ij} is the function associated to the transition (P_i, P_j). The functions F_{ij} can contain :

- predicates on the time variables or on the D.S variables,
- boolean expressions on input or internal variables of the C.S.

Special temporal predicates may be used. For example, the instruction :

P_i IF $!t_c$ THEN P_j

means that a transition takes place from P_i to P_j as soon as a time t_c has elapsed after the end of the procedure associated to P_i.

The language's syntax permits the description of many transitions in the same instruction by using multiple tests and branchings. For example, the instruction :

P_i IF F_{i1}; F_{i2}; F_{i3}; ... THEN P_{j1}; P_{j2}; P_{j3}; ...

has the interpretation : a transition takes place from P_i to P_{jk} if F_{ik} is true.

The instruction :

P_i IF F_{i1}, F_{i2}, F_{i3},... THEN P_{j1}, P_{j2}, P_{j3},...

can be used to give priority to an action. Priority is defined by the order of the conditions F_{ik}. That is, the transition labelled by F_{ik} is fired if F_{ik} is verified and if all the conditions F_{is} such that $s < k$ are faulty. The two preceeding instructions can also be put into the parametrized forms :

P_i IF F_{i1}; F_{i2}; F_{i3};... THEN P(J)

P_i IF F_{i1}, F_{i2}, F_{i3},... THEN P(J)

Two primitives, put at the head of the C.S description, allow the definition of input and output control variables. The primitive INPUT defines input variables of the C.S external to the module (example : INPUT $X_1, X_2, \ldots$).

The primitive OUTPUT defines output variables of the C.S. (example : OUTPUT $Y_1, Y_2, \ldots$).

The representation of a transition having as input place a source place is given by an instruction of the form :

1 IF F THEN P

For such an instruction, the condition F is tested permamently, and one token is put into the place P every time this condition is verified. Finally, the possibility is given to express directly the deactivation of a place P_i without token transfer to any other place, by using the instruction :

P_i IF F THEN $P_i^!$

II.3.1.2. Declaration of a configuration of the system

The primitive CREATE permits the creation of a new module by using the description of a predefined standard module. Example :

CREATE MODULE1 = MODEL

results in the creation of a new module called MODULE1 as a copy of the standard module MODEL.

- The primitive LINK realizes the interconnection of modules. The genitive notation (with point) is used to distinguish the interface variables of several identical modules. So, we can write :

LINK MODULE1.INTV1 = MODULE2.INTV1

to express the fact that the interface variables INTV1 of the two modules MODULE1 and MODULE2 are confounded. When there is no ambiguity, the genitive notation may be avoided.

II.3.1.3. Definition of the initial state and of the input values

The initialization of a variable may be done by using the primitive INIT. Example :

INIT MODULE1.INTV1 = 2

means that the variable INTV1 of the module MODULE1 is initialized to the value 2. The primitive INIT can recover many initializations and the variables concerned may be internal or interface variables as well as input variables of the C.S. The following examples show some possibilities of the use of the primitive INIT :
INIT MODULE1.VAR1 = value1 ; MODULE2.VAR2 = value2
INIT MODULE1.VAR1, MODULE2.VAR1 = value3
INIT VARINX = 1
All the variables non initialized explicitly by the primitive INIT are initialized to the default value zero. Inputs used to experiment the simulated system are introduced by means of the primitive ENTRIES. These inputs are represented by input lists, each list having the following format :

t_i : X_1 = value1, MODULE1.Y = value2,...

This means that at time t_i the input variables X1 and Y of the module MODULE1 will take respectively the values value1 and value2. It is also possible to define periodic inputs. Example :
t_j : + t_p [X1 = value1, MODULE.Y = value2]
This means that the variables X1 and MODULE.Y take respectively the values value1 and value2 at all the moments $t_j + mt_p$ for m = 0,1,2,...

PART III
PRACTICAL EXAMPLE

We illustrate this language by a simple example taken from [15] ; it describes a modular architecture for a multi-access, multibank memory system (§ III.1).

In part III.2, we will explain how this language can help the design (performance evaluation, detection of critical configurations...)

III-1. Description of the system

The structure of the system is detailed in fig. 7. We distinguish 3 types of modules :
α) memory bank (MB)
β) memory access multiplexor
γ) entry point (EP)
These modules'interfaces are as follows :

α) memory bank interface (MB_i)

. MA_i : address register (16 bits)

. MD_i : data register (16 bits)

. ST_i : status register (2 bits)
ST_i : 00 = MB_i inactive
ST_i : 01 = read operation in progress } posted by the multi-
ST_i : 10 = write operation in progress } plexor
ST_i : 11 = reading accomplished } posted by MB_i

(notice : we have modified slightly the original example in [15]).

. DES_i : entry point's address register in case of a reading request.

β) entry point interface (EP_i)

r_i	: 1 = Read request (1 bit)
w_i	: 1 = write request (1 bit)
b_i	: 1 = busy entry point (1 bit)
CA_i	: address register (16 bits)
CD_i	: data register (16 bits)
RRP_i	: 1 = read request from the processor connected to EP_i
WRP_i	: 1 = write request from the processor connected to EP_i
$ADDP_i$	: address buffer of the processor connected to EP_i
$DATAP_i$	: data buffer of the processor connected to EP_i

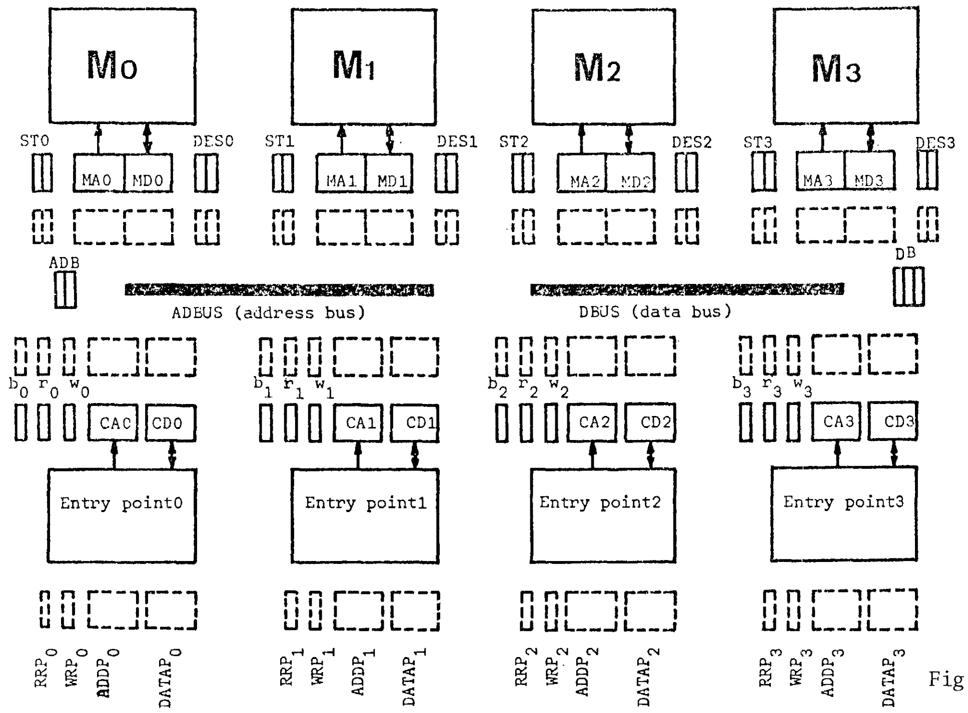


Fig.7 : Memory layout

The multiplexor uses two buses ADBUS and DBUS (address and data) to link an entry point to a memory bank. The internal register ADB(2 bits) indicates which entry point uses ADBUS; DB_0 and DB_1 indicate which entry point (write) or which memory bank (read) has posted a data on DBUS; DB_2 tells which way the exchange goes (respectively 0 and 1).
Figures 8a, 8b and 8c show the 3 module flowcharts and the corresponding control graphs.
Note : In what follows, we keep the same notations for places and for their associated boolean variables.

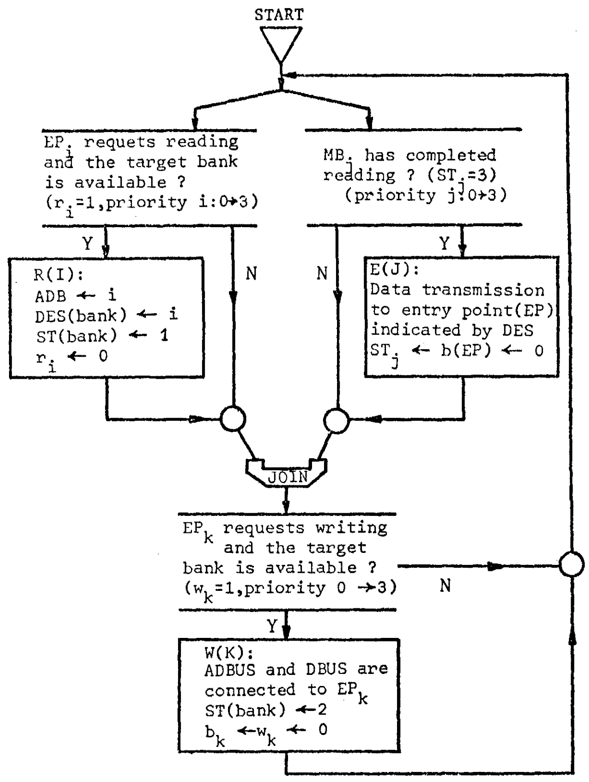

START
EP$_i$ requets reading and the target bank is available ? (r$_i$=1,priority i:0→3)
MB$_j$ has completed reading ? (ST$_j$=3) (priority j:0→3)
Y
N
N
Y
R(I):
ADB ← i
DES(bank) ← i
ST(bank) ← 1
r$_i$ ← 0
E(J):
Data transmission to entry point(EP) indicated by DES
ST$_j$ ← b(EP) ← 0
JOIN
EP$_k$ requests writing and the target bank is available ? (w$_k$=1,priority 0 →3)
N
Y
W(K):
ADBUS and DBUS are connected to EP$_k$
ST(bank) ←2
b$_k$ ←w$_k$ ← 0

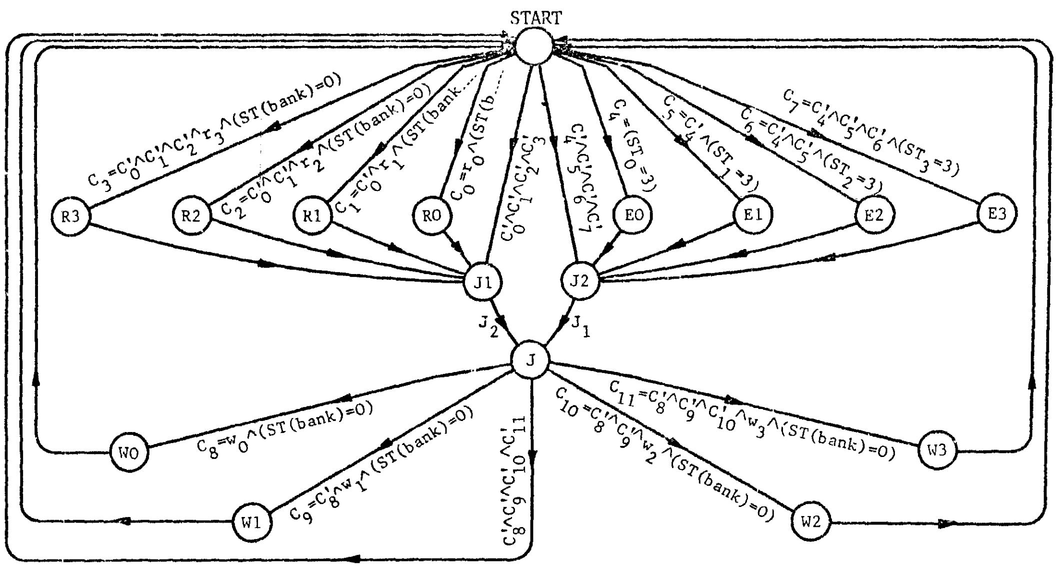
Fig.8a : Multiplexor's flowchart

III-2. System's description program
In order to simplify this example, we do not take into account the nature of the information exchanged between MB's and EP's; so registers MD and CD are not represented. Therefore, only the bank's number is retained as an address in CA.

```
DCLMODTYPE ENTRYP (b,r,w,CA,ADDP)
   DCLPROC RR
      TEMP t_rr (reception time of a reading request)
      CA ← ADDP
      r ← b ← 1
   END
   DCLPROC WR
      TEMP t_wr (reception time of a writing request)
      CA ← ADDP
      w ← b ← 1
   END
   DCLPROC AER
      TEMP t_aer (acknowledge time of the end of a rea-
                 ding operation)
   END
```

```
  DCLCTRL  ASYN
   INPUT RRP,WRP
   BEGIN
   IN IF WRP∧b'  THEN WR
   IN IF RRP∧b'  THEN RR
   WR IF b'      THEN IN
   RR IF b'      THEN AER
  AER IF 1       THEN IN
ENDMOD
DCLMODTYPE   MEMORYB(ST,DES)
  DCLPROC    RH
     TEMP    t_rh  (Read Handling time)
     ST    ← 3
  END
  DCLPROC    WH
     TEMP    t_wh (Whrite Handling time)
     ST    ← 0
  END
 DCLCTRL ASYN
  BEGIN
     READY IF ST = 1 THEN RH
     READY IF ST = 2 THEN WH
     RH    IF 1      THEN RA
     RA    IF ST = 0 THEN READY
     WH    IF 1      THEN READY
ENDMOD
DCLMODTYPE MULTIPLEXOR (ST(4), DES(4), b(4), r(4), w(4),
                        CA(4))
     DCLPROC R(I)
         TEMP  t_r
         DES[CA[I]]  ← I
         ST[CA[I]] ← 1
         r[I] ← 0
     END
     DCLPROC E(J)
         TEMP  t_er
         ST[J] ← b[DES[J]] ← 0
     END
     DCLPROC W(K)
         TEMP  t_w
         ST[CA[K]] ← 2
         w[K] ← b[K] ← 0
     END
 DCLCTRL  ASYN
  BEGIN
START IF C0,C1,C2,C3, 1 THEN R(0),R(1),R(2),R(3),J1 *)
START IF C4,C5,C6,C7, 1 THEN E(0),E(1),E(2),E(3),J2
R(I)  IF 1 THEN J1
E(J)  IF 1 THEN J2
J1    IF J2 THEN J
J2    IF J1 THEN J
J     IF C8,C9,C10,C11,1 THEN W(0),W(1),W(2),W(3),START
W(K)  IF 1  THEN START
ENDMOD
DCLCONF
  CREATE MEMORYB0,MEMORYB1,MEMORYB2,MEMORYB3=MEMORYB
  CREATE MULTIPLEXOR0 = MULTIPLEXOR ;
         ENTRYP0, ENTRYP1, ENTRYP2, ENTRYP3 = ENTRYP
  LINK MEMORYB0.ST=MULTIPLEXOR0.ST(0);
       MEMORYB0.DES=MULTIPLEXOR0.DES(0);
       MEMORYB1.ST = ...
              :
  LINK ENTRYP0.r = MULTIPLEXOR0.r(0)
              :
END
INIT  IN, READY,START=1;b,r,w, ST=0 (all places IN,
         READY,START,b,r,w,ST are initialized)
ENTRIES 0:RRP0,ADDP0=0;9:WRP1,ADDP1=1;...
         ..... STOP 20
END
```

*) In fact, conditions Ci must be entirely explicited in the program; they are given in Figure 8a.


START
R3
R2
R1
R0
E0
E1
E2
E3
J1
J2
J
W0
W1
W2
W3
$C_3=C_0'\wedge C_1'\wedge C_2'\wedge r_3\wedge(ST(bank)=0)$
$C_2=C_0'\wedge C_1'\wedge r_2\wedge(ST(bank)=0)$
$C_1=C_0'\wedge r_1\wedge(ST(bank...$
$C_0=r_0\wedge(ST(b...$
$C_0'\wedge C_1'\wedge C_2'\wedge C_3'$
$C_4'\wedge C_5'\wedge C_6'\wedge C_7'$
$C_4=(ST_0=3)$
$C_5=C_4'\wedge(ST_1=3)$
$C_6=C_4'\wedge C_5'\wedge(ST_2=3)$
$C_7=C_4'\wedge C_5'\wedge C_6'\wedge(ST_3=3)$
J_2
J_1
$C_8=w_0\wedge(ST(bank)=0)$
$C_9=C_8'\wedge w_1\wedge(ST(bank)=0)$
$C_8'\wedge C_9'\wedge C_{10}'\wedge C_{11}'$
$C_{10}=C_8'\wedge C_9'\wedge w_2\wedge(ST(bank)=0)$
$C_{11}=C_8'\wedge C_9'\wedge C_{10}'\wedge w_3\wedge(ST(bank)=0)$

(Fig.8a : Multiplexor's control graph)

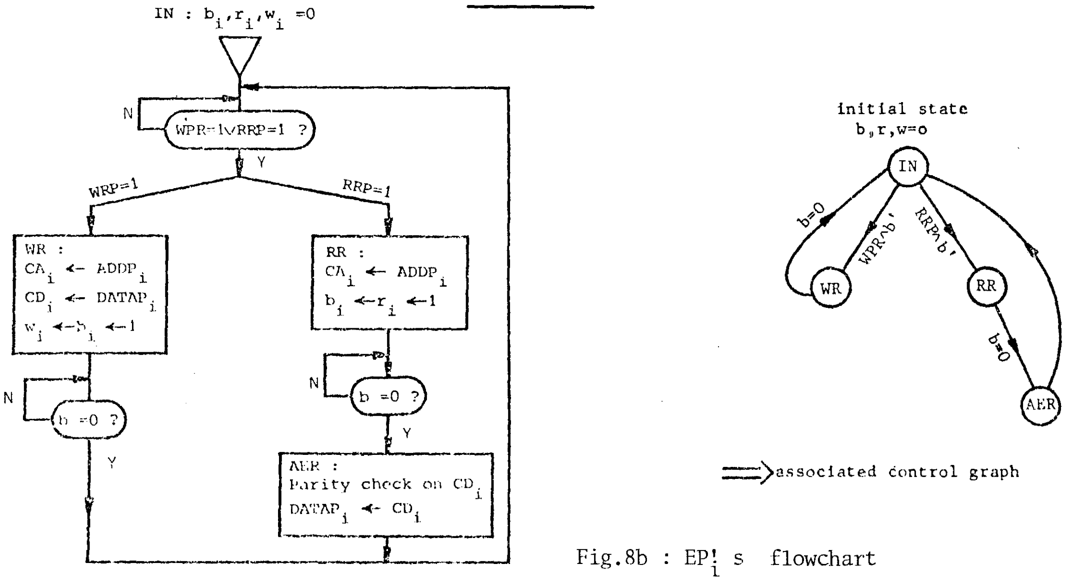


Fig.8b : EP_i' s flowchart

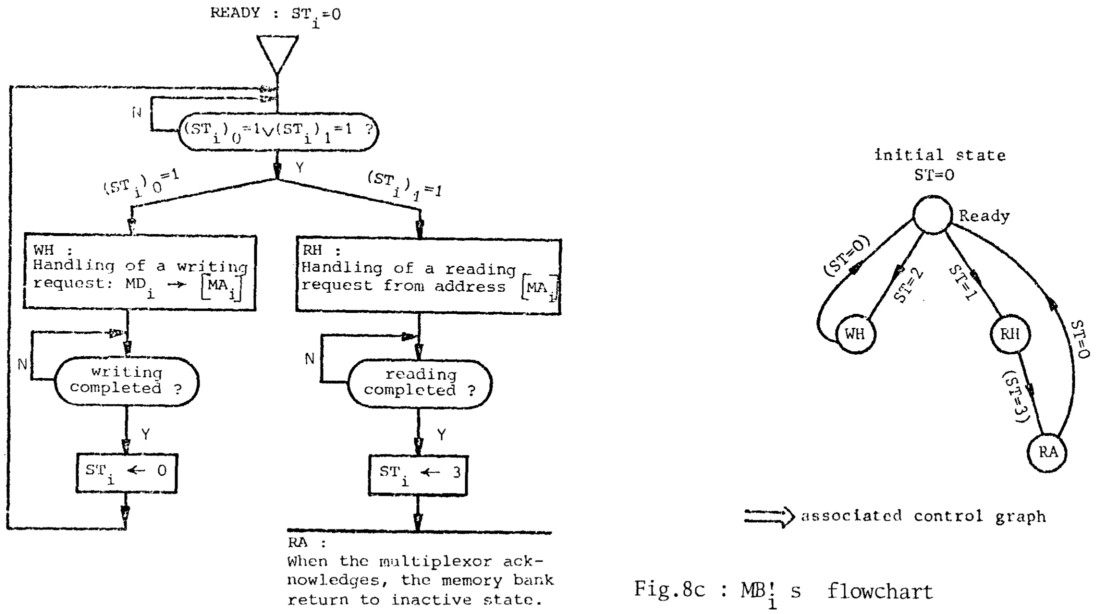

READY : ST_i=0
N
$(ST_i)_0$=1∨$(ST_i)_1$=1 ?
Y
$(ST_i)_0$=1
$(ST_i)_1$=1
WH :
Handling of a writing request: MD_i → $[MA_i]$
RH :
Handling of a reading request from address $[MA_i]$
N
writing completed ?
Y
N
reading completed ?
Y
ST_i ← 0
ST_i ← 3
RA :
When the multiplexor acknowledges, the memory bank return to inactive state.
initial state
ST=0
Ready
(ST=0)
ST=2
ST=1
ST=0
WH
RH
(ST=3)
RA
associated control graph

Fig.8c : MB_i' s flowchart

III-3. Comments on the design aid

ADBUS and DBUS allocation in the multiplexor is performed according to a decreasing priority $EP_0 \rightarrow EP_3$ and $MB_0 \rightarrow MB_3$. The question arises whether, for given execution times of the modules Multiplexor and MB, some request on EP_3 remains unsatisfied for some critical request frequencies on EP_0 and EP_1. We can find the critical values by observing the system for various execution times.

Likewise, a standard multiplexor module is defined to connect 4 MB's and EP's. But, without any essential modification in the description program, these numbers can be changed. Thus, we can study the correlation between structure, number of resources and performances. This kind of possibility is essential for the evaluation of the performances of I/0 architectures (through-put) according to the number of the channels and the repartition of the peripheral devices. Another idea could be to associate to every EP a critical time t_c such that : every request from an EP must be satisfied in time less tant t_c in order to avoid disturbances in the processor connected to EP. (A processor connected to EP could be an I/0 channel with response time bounded by a critical value). Notice that in this language, these critical time bounds can be expressed in the description of the processor by a timing function.

IV - CONCLUSION

The first interest of this tool is that it provides a methodology for the multilevel description of distributed systems. Certainly, its use is particularly interesting for the study of the behavior of a module (or set of modules), when it is integrated in a specific configuration rather than when it is considered separately. The separation between C.S and D.S corresponds implicitly to the distinction between K and (D,M) primitives in PMS [16]. Functional subsets corresponding to the primitives L and S are shared by the cooperating modules. It is nevertheless sometimes interesting, when emphasis is put on the study of the inter-module communication procedures, to extract these functional subsets in order to reconstruct the module performing the link. Conversely, it is also easy to deduce from the PMS description of a system (possibly completed by its ISP description [16]), the structure of the program describing this system in the proposed language.

We finally emphasize that our intention has been to provide a simple enough language to be user oriented. All monitoring functions concerning duplication, interactions and synchronization of the modules remain transparent to the user and are handled by the system supporting the programming in this language.

The elaboration of this tool has been motivated by two studies actually developed at the ENSIMAG [17], [18]:

a) Study and evaluation of a hierarchical memory structure for multiprocessor machines. Such a structure consists of :

. a hierarchy of physical memories

. algorithms of dynamic management at each level (address translation, communication between adjacent levels).

This tool must permit :

. validation and comparison of some choices of algorithms at each level

. verification of the global coherance of performances

b) Design of a complex I/0 system (UNIDATA 7740-50-70). This tool must permit :

. harmonization of module characteristics at each level (buffer dimensions, information adjustments, temporal characteristics).

. experimentation of the dialog procedure (priority logic, conflict resolution, resources dispatching) and evaluation of global performances.

REFERENCES

[1] M.R. BARBACCI : "A Comparison of Register Transfert Languages for Describing Computers and Digital Systems", IEEE Trans. on C., Feb. 1975

[2] V.M. GLUSHKOV, A.A. LETICHEVSKII : "Theory of Algorithms and Discrete Processors", Advances in Information Systems Science, Vol. 1, Chap. I, pp 1-58 Edited by Julius T.TOU.

[3] F. ANCEAU, P. LIDDELL, J. MERMET, C. PAYAN : "CASSANDRE : A Language to Describe Digital Systems Applications to Logic Design", 3rd International Symposium, Comp. and Inf. Science, Miami, Fla., Dec. 1969.

[4] J.R. DULEY : "DDL, A Digital Language", Ph. D. Dissertation, dep. El. Eng. Univ. Wisconin, Madison, June 1970

[5] Y. CHU : "Introducing Computer Design Languages", Digest of Papers, Compcon 72, San Francisco, Sept. 72, pp. 215-218.

[6] K.E. IVERSON : "A Programming Language", J. WILLEY 1972

[7] J.A. DARRINGER : "The Description, Simulation, and Automatic Implementation of Digital Comp. Processors", Ph. D. Thesis, EE department CMU, May 69

[8] O.J. DAHL, K NYGAARD : "SIMULA, an Algol Based Simulation Language", Comm. ACM, vol. 9, sept. 66

[9] S.Y.H. SU : "A Survey of Computer Hardware Description Languages in the U.S.A.", Computer, Dec. 74.

[10] J.J. CLANCY, M.S. FINEBERG : "Digital Simulation Languages : A critique and Guide" Proceedings - FJCC - 1965

[11] M. ZACHARIADES : "Langage d'aide à la conception des systèmes logiques", D.E.A. Report, ENSIMAG, Grenoble, France, june 1974

[12] C.A. PETRI : "Communication with Automata", Technical Rep. n° RADC-TR-65-377, vol. 1, Rome Air Development Center, Griffiss Air Force Base, New-York.

[13] J. SIFAKIS : "Modèles Temporels des Systèmes Logiques", Thèse Doc. Ing., Univ. of Grenoble, march 1974.

[14] M. MOALLA, J. SIFAKIS, M. ZACHARIADES : "Un langage d'aide à la conception d'un système de modules interconnectés". Int. Report, RR-16, ENSIMAG, Grenoble, France, oct. 1975

[15] F.J. HILL, G.R. PETERSON : "Digital Systems ; Hardware Organization and Design", pp. 395-403, John Wiley & Sons, 1973

[16] C.G. BELL, A. NEWELL : "Computer Structures : Readings and Examples", Ed. Mac Graw Hill, 1971, p. 15, Part. I, chap. 2.

[17] M. BENNETS, M. VERAN : Int. Report ENSIMAG, Grenoble, France, june 1975

[18] M. MOALLA : "Outil logiciel de test de simultanéité en intégration des systèmes", ENSIMAG, Grenoble, France, Seminar june 1975.

A Course in Computer Structures

by
Jonathan Allen
Department of Electrical Engineering and Computer Science
and
Research Laboratory of Electronics
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Cambridge, Massachusetts

Abstract

In this subject, we treat computer structure as an element of a group of interacting structures including the technology, algorithm, data, and programming language. In the belief that the best designs result when these structural factors "match" in a complementary manner, the influence of each of these domains is carefully studied at both the conceptual and descriptive levels. Thus a modular treatment of current technology is provided, as well as a thorough analysis of algorithmic structure as reflected in computational schemata. Single-sequence machine design is discussed, including advanced topics such as multiple functional unit conflict resolution. Following a presentation of microprogramming and input/output, virtual ideas centered around the notion of process are introduced, leading to the design of multiprocessing and multiprocessor systems. These ideas are applied and extended in the presentation of the Burroughs B6700 as a higher-level language machine designed to execute ALGOL efficiently. Finally, the subject concludes with an introduction to general interpretive structures for high-level languages.

1. Introduction

The content of this subject is determined by the belief that the best computer designs result from a consideration of all the structures that are involved in the solution of a problem on a digital computer. These structures are associated with:

1. Technology
2. Algorithm
3. Data
4. Programming Language
5. Architectural units

When these structural factors are mutually complementary, we believe that the best results follow, and we call the principle that requires this interaction "structural match." Thus, we believe that it is not possible to design a computer properly unless a unified view of all the structural constraints posed by the intended application area are well understood. The architectural structure of a computer is thus seen to be but one of a set of interacting structural factors which must be cohesively interwoven to provide an optimal design.

In addition to the structural match principle, we believe that it is important to achieve a proper balance between conceptual abstractions and the description of practice. Too often, the teaching of computer architecture is confined to a highly descriptive treatment of several machines such that it becomes very difficult to detect general principles. On the other hand, it is important to motivate conceptual constructs and results by showing their application to concrete designs. The course seeks to detect the underlying generality in computer design, but to continuously interrelate these results to classical designs which have stood the test of time. Experience has shown that it is possible to combine both bottom-up and top-down approaches by this means in a manner such that abstractions are motivated by concrete examples, and then these instances of practice are clarified by interpreting them in the light of more generalized structures.

The overall subject is split into three major parts:

I. A. Combinational and Sequential Logic; Computer building blocks
   B. Computation Schemata and Implementation

II. A. Single Sequence Computers
   B. Microprogramming
   C. Input/Output

III. A. Processes and Multiprocessors
   B. Algol and Block Structured Languages and the B6700
   C. Virtual machines and dynamic microprogramming

Each of these three groupings consumes about 1/3 of the subject time. We now discuss each of them in detail.

2. Logic, Technology, and Schemata:

In the first third of the subject, the background necessary for discussion of single-sequence computers is established. First, combinational and sequential logic is presented in order to establish a concrete basis on which to discuss computer design. This material is not presented from the point of view of a digital designer, since it is felt that topics such as gate minimization and state assignment are not needed to appreciate the design of computers. Nevertheless, basic combinational and sequential logic principles are presented, and there is a very complete discussion of flip-flops. Use is made of the Algorithmic State Machine[1] formalism which facilitates implementation-free definitions of sequntial circuits. Following the treatment of basic logic principles, modular computational elements are discussed, including registers, counters, encoders and decoders, ALU's, multiplexors, ROM's, and shift registers. The goal is to provide sufficient concrete understanding of logical modules so that students can appreciate how a block diagram description of a computer could be realized. This approach also provides a demystifying benefit. For example, many students have difficulty understanding how it is possible to write into and read from a register simultaneously until master-slave flip-flops are explained. Modern MSI and LSI practice is also discussed so that students can appreciate the construction of computers in terms of functional modules. This background is utilized later in the course when microprogrammed computers are discussed.

Once the student is well versed in the basic logical structures contained in computers, we abstract on these functional modules to obtain schemata[2] which concentrate on two aspects of computation. One aspect is the way in which data flows between modules, and the other is the control of this data flow sequence. Computation structures are simplified to schemata containing only memory cells and operators as nodes, so that the possibilities for concurrent operation can be clearly studied.

The first type of schemata studied is the data dependence graph[2] which naturally exhibits all of the possible concurrency in a computation by the simple means of only allowing each operator and memory cell to be used once in the course of evaluating the algorithm. Thus, if four additions are needed in an algorithm, then four add operators must be provided in the corresponding data dependence graph. In this way, the only thing that limits completely concurrent use of all the operators is the logical dependencies contained in the algorithm. Conflicting use of memory cells is impossible, since each cell has only one input. Figure 1 shows a typical data dependence graph (DDG).

The DDG provides a conflict-free, maximum-space, maximum concurrency representation of an algorithm. If the operators and memory cells are controlled appropriately, then a minimum-time maximally-parallel implementation results. One special case of interest is when the operators are all combinational. In this case, the memory cells can be removed, and no timing control is needed, since the entire schema is represented as one combinational circuit.

When the constituent operators of a DDG are sequential in nature, then it becomes important to control their time of operation. That is, we must provide a means to tell the operator when the input data are ready so that the operator may start. Similarly, the operator must acknowledge when it has completed its calculation, so that its output can be read at the appropriate time. The DDG contains all the information necessary to derive the constraints on operator sequence, and this is shown in the precedence graph of Figure 2. We can see from this graph that a given operator cannot be started until all operators which precede it in the graph have completed their operation. What is now needed is a modular control structure which enforces these precedence constraints, but no more. Although the case of synchronous control is treated completely, we put more emphasis on the general asynchronous case. From a general point of view, synchronous control timing is global in nature and forces a uniform lock-step time pattern over the entire system. Asynchronous control, however, is local in nature, and provides a timing structure which is sensitive to the individual device timing characteristics. We believe that it is important for the student to understand the relative virtues of both of these control systems, and examples of each are given throughout the subject.

The asynchronous control structure for a DDG can be constructed from sequence, fork, and join modules[2], the interconnection of which can be derived from the precedence graph associated with the DDG. The next addition to the schemata is a trigger module, which allows DDG's to be modified for pipelined operation. Basically, the trigger module for an operator checks both that new input data is ready, and that the operator's output has been latched up in its output register. Thus, by the simple expedient of adding one new type of control, the DDG structure is naturally extended to a pipeline schemata. Since piplined structures are of such practical importance, there is considerable discussion of these systems, both synchronously and asynchronously controlled.

Following the discussion of pipelined operators, the detailed specificity of operators and their individual operation times is abstracted away from the precedence graph and control module structures in order to introduce Petri nets.[3] Control examples such as FIFO buffers are discussed, and Petri net models for all of the asynchronous control modules are presented. Petri nets have proved to be a particularly clear and implementation-free way to concentrate on basic control issues, and although we emphasize their utility as abstract models for physical control structures, we also develop their theory including a discussion of liveness and safeness.

Building up from DDG's through pipelined systems, the next logical stop is to allow repetitive use of operators and cells within a schema and more than one input to memory cells. The resulting structures, called elementary schemata[2] comprise two parts: a data flow graph and a precedence graph. Since the data flow graph may contain directed loops, repeated use of operators and cells is possible, and the schemata represent algorithms in less space but more time than the corresponding DDG. This space-time trade-off is emphasized by showing that for each execution sequence of all the operators in an elementary schema precedence graph (which preserves the precedence relations), there is an equivalent DDG. In order to utilize operators and cells repeatedly, it is necessary to introduce a new control module, called union, which has one input link for each time the cell or operator must be used in the course of a computation. Perhaps the most interesting aspect of elementary schemata, however, is that they permit conflict at memory cells to arise, such as when two inputs to a cell are unordered in the precedence graph. We give a very careful treatment of conflict, and show that every execution sequence of a conflict-free elementary schema has the same equivalent data dependence graph. The fact that the sequence of memory cell loadings of every conflict-free schema is determined solely by the initial values of the cells is used to prove that every conflict-free elementary schema is determinate. Finally, we also prove that every conflict-free elementary schema is functional, in the sense that the final values in the output cells are functions of the initial values in the input cells. The notion of functionality also allows us to give a precise definition of equivalence of elementary schemata.

It has turned out that the careful treatment of conflict and its associated concepts has paid off substantially in the discussion of many practical topics including multi-ported memories, multiple-function-unit processors, process scheduling, and concurrent I/O. In all of these cases we have been able to give a rigorous conceptually-grounded discussion which avoids superficial description. A complete discussion of arbiters[4] is given, including implementation details, in order to illustrate the breakdown of theoretical models for synchronizers and arbiters when events are separated in time by arbitrarily small amounts.

The last type of schema to be discussed is the basic schema,[2] which adds iteration and conditional tests to the power of the elementary schemata. Two new control modules are introduced for this purpose, providing for control branching based on the truth value of test predicates. With the addition of these new features, "while P do G" and "until P do G" constructs can be easily modeled. There are no new conceptual difficulties due to addition of these structures, but the basic schemata are now adequate to serve as models for digital computers. Thus the treatment of schemata provides a clear foundation for the detailed study of computer structures, in which all of the difficult control problems are faced. Students are thus able to examine very complicated machines in the context of an appropriate conceptual framework.

3. Single-Sequence Machines, Microprogramming, and Input/Output:

The second third of the subject is devoted to single-sequence computers. After a brief review of computability results[5] which show that nearly every computable algorithm can be performed on a general purpose computer in finite time, a minimal-state machine model is introduced which has no state memory in the processor except for program sequence information. This machine allows us to concentrate on the basic control sequence within an instruction. From the minimal-state machine, it is natural to introduce the three-address architecture. A complete basic schema for a simple three-address machine is developed, and the time balance between memory and processor is discussed. Next, the classic one-address machine[6] is introduced, and historical perspective is used to motivate its structure. This architecture extends naturally to the modern general register machine, where we provide a basic schema for a subset of the DEC PDP-10.[7] The influence of technology on general register designs is pointed out, and the IBM 370[8] architecture is also discussed at this point.

Stack architecture is then developed, with an eye to later detailed discussion of the Burroughs B6700.[9] This is also an appropriate time to discuss program sequence control including subroutine access and recursion with their heavy reliance on stack mechanisms.

A complete treatment of memory designs is provided, including addressing by indexing and indirection. Multiport, overlapped, and interleaved memory modules are covered using the previously acquired background in asynchronous control and arbitration. A variety of cache systems is also treated, but virtual memory is saved for the last third of the subject.

A particularly interesting topic is that of multiple-functional-unit computers, such as the CDC 6600[10] and the IBM 360/91.[11] We point out how conflict can arise when such concurrency is allowed, and then show that the control structures which resolve the conflicts in both of these machines can be interpreted as transforming the multiple units into a dynamic DDG, which is of course conflict-free. This is a good example of how fundamental notions introduced in the first third of the subject can be applied to clarify difficult control designs and cut through superficial differences. We also discuss pipelined processors such as the Texas Instruments ASC[12] and the CDC Star-100,[15] utilizing the earlier treatment of pipelining.

After the basic structure of single-sequence machines has been treated, we consider microprogramming.[14] This is useful both to demystify the detailed construction of computers, and to prepare for a discussion of virtual machines later. The Wilkes model is presented as a systematic means to implement control, and then a simple but complete microprogrammed computer is discussed in detail. Register, ALU, buss (including three-state logic), memory timing, and main instruction subroutine designs are treated as well as condition testing and branch sets. Experience indicates that this concrete discussion of microprogramming is very useful to students as a basis for interpreting their conceptual understanding. Vertical and horizontal microcoding designs are compared, and the use of microcode to dynamically reconfigure the data flow structure (as in the SPS 41[15] arithmetic section) of a machine is presented. Then control memory design features are enumerated, leading to writeable control stores, dynamic microprogramming, and emulation. This seems to be the natural place to present the IBM 360/370 architecture,[14] with its provision for emulation (e.g. of the IBM 1401 and 7094) and family compatability achieved through various technological implementations.

The one remaining aspect of single sequence machines is input/output, the various features of which we unify by contrasting how processor state information is saved and switched. First, status driven I/O is presented via a teletype serial interface, complete with processor "busy-waiting." Then interrupts are introduced, and critically compared with status-driven I/O. The basic schema for a single-address machine is modified to accommodate this simple level of I/O. Next, direct memory access is developed as a special purpose wired-logic processor which avoids most interrupts in a block transfer. The direct memory access is then generalized to channel architecture, and a complete discussion of channel operation is given in the context of disk management.[8] Finally, channels are generalized to completely independent I/O processors. Virtual I/O processors, such as those used in the CDC 6600[10] and Texas Instruments ASC[12] are introduced using the "time-slot" sharing concept, and then completely separate I/O processors for high data-rate applications are used as an example of the use of microcomputers. As a practical application, we have recently added to this section of the subject a treatment of asynchronous bus design,[16] including the new instrument interface standard.[17] This material is particularly useful to non-computer-science students whose main concern is to acquire an in-depth understanding of the use of computers in complex instrumentation and data acquisition systems.

4. Processes, Block-Structured Languages, and Virtual Machines:

In the last third of the subject, we make a transition from single-sequence machines to the consideration of asynchronously interacting algorithms via the notion of the process,[18] and from this abstraction to virtual machine ideas. The previous discussion of input/output has provided concrete examples of interprocess communication, so that students are led naturally to the notion of process, or the virtual running of a program. In order to introduce virtual ideas, we first discuss virtual memories,[19] both those with paging and purely segmented types. These are then generalized to the notion of virtual device, as used in Dijkstra's THE system.[20] We are then led to the idea of process as the basic construct for virtual machines.

Processes are rigorously defined, as are processors, and the problems of asynchronous communication between processes via shared variables are discussed as examples of conflict at a higher level than was previously treated in terms of schema. Critical sections are defined, and process control primitives are introduced. The interactions of create, run, block, wakeup, and terminate are illustrated by many examples, and then we show how these primitives can be implemented.[21] The need for perfect arbitration leads to the notion of lock variables and the associated busy-waiting problems, and then Dijkstra's P and V semaphore operators are discussed. Detailed implementations of these operators are given, including introduction of test-and-set or equivalent instructions. Students enjoy the mental exercise of devising semaphore solutions to process communication problems, and we treat many classic examples, such as the bounded buffer problem, in class.

Following the consideration of processes we naturally turn to multiprocessor systems or several varieties.[22] Recently we have treated Illiac IV, C.mmp,[23] SPS-41,[15] and the BBN Pluribus[24] as varied and interesting examples of the many design options. These are particularly valuable illustrations since they can be strongly motivated by their intended areas of application.

Once a solid groundwork in processes has been established, we find it feasible to examine a very unique machine, the Burroughs B6700[9]. First, however, we review the semantics of ALGOL 60[25], and introduce Johnston's contour model[26] to make this explicit. We have found, incidentally, that the contour model provides an especially effective means for clarifying recursion, a topic which seems to require extensive explanation from many points of view. The contour model of ALGOL semantics can then be mapped one-to-one onto a stack model, where we develop completely the manipulation of static and dynamic links. Once the stack model has been presented, it is both natural and easy to introduce the B6700. We emphasize here that the B6700 is a very complicated machine, albeit of great interest. Nevertheless, given the requisite background in processes and stack implementation of ALGOL, students are able to insightfully absorb a vast amount of detail about the computer, since they have the appropriate conceptual background with which to appreciate and interpret the design. We have found this part of the subject, with its attendant treatment of procedure entry and exit, multiprocess contour model, and events and queues, to be extremely interesting and useful to students. Concepts which have been developed throughout the course up to this point, with the sole exception of microprogramming, all come together in the B6700, and students are quick to realize that the machine architecture would be unintelligible without a deep appreciation of the implementation needs of inter-process structure and ALGOL variable bindings. The careful treatment of these topics requires a lot of time, but we have found it to be extremely worthwhile.

Following the discussion of the B6700 as a higher-level language machine, it is natural to ask if machines can be built which can be effectively adapted to several higher-level languages. In effect, what is needed is an architecture that supports several virtual higher-level language machines. The best current example of such a computer is the Burroughs B1700,[27] which we discuss briefly. The use of variable length encoding, defined fields, and the provision of general interpretation facilities are illustrated with this machine. We find this to be a fertile area in the subject, but presently lack of time, and detailed knowledge of the B1700 implementation, have precluded greater emphasis.

5. Summary:

We have described an attempt to approach computer architecture from a broad perspective, and to treat machine architecture design as a system of cooperating processes, each of which represents the constraints imposed by a particular structural domain. In this way, the interaction between technological features, algorithmic structure, programming languages, and data structures is allowed to determine the resultant architecture. Our experience has shown that this approach provides a good balance between description of practice and conceptual analysis, so that the pedogogical experience is of immediate utility but also lasting importance. While it is difficult to isolate design principles for computer design, we feel that this combined structural approach provides a good basis on which to continue the search.

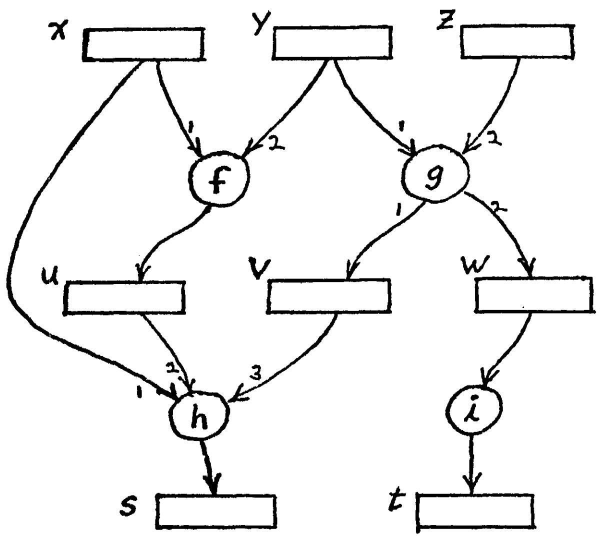


Fig. 1 Typical Data Dependence Graph

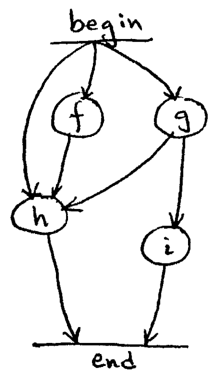


Fig. 2 Precedence Graph derived from Data Dependence Graph of Fig. 1

References

1. Clare, C. R. Designing Logic System Using State Machines, McGraw-Hill, 1973.

2. Allen, J. and Gallager, R. G. Notes for M.I.T. Subject 6.032; Computation Structures.

3. Petri, C. A. "Communication with Automata" Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1. Griffiss Air Force Base, 1966.

4. Patil, S. S. "Bounded and Unbounded Delay Synchronizers and Arbiters" Computation Structures Group Memo 103, M.I.T., June 1974

5. Minsky, M.L. Computation: Finite and Infinite Machines, Prentice-Hall, 1967.

6. Burks, A. W., Goldstine, H. H., and Von Neumann, J. "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" in Bell, C. G., and Newell, A.: Computer Structures, McGraw-Hill 1971.

7. Digital Equipment Corporation DECSYSTEM10 Assembly Language Handbook. 1972.

8. Katzan, H. Computer Organization and the System/370. van Nostrand Reinhold, 1971.

9. Organick, E. I. Computer System Organization; the B5700/B6700 Series, Academic Press, 1973.

10. Thorton, J. E. "Parallel Operation in the Control Data 6600 AFIPS Proc. FJCC, Part II, Vol. 26, 1964.

11. Entire issue, IBM System Journal, Jan. 1967

12. Texas Instruments, Inc. "A Description of the Advanced Scientific Computer System" Document M1001P, Dec. 1972.

13. Hintz, R. G., and Tate, D.P. "Control Data Star-100 Processor Design" Compcon '72 Digest of Papers.

14. Husson, S. S. Microprogramming Principles and Practices, Prentice-Hall, 1970.

15. Signal Processing Systems, Inc.: SPS-41 Users Manual, April 1973, Waltham, Mass.

16. Digital Equipment Corporation, PDP-11 Peripherals Handbook, 1975.

17. IEEE Standard Digital Interface for Programmable Instrumentation, IEEE Std 488-1975, April 1975.

18. Horning, J. J., and Randall, B. "Process Structuring" Computing Surveys, 5, No. 1, March 1973.

19. Denning, P.J. "Virtual Memory": Computing Surveys, 2, No. 3, Sept. 1970.

20. Dijkstra, E. W. "The Structure of THE-multiprogramming System" Comm. ACM, 11, No. 5, May 1968.

21. Dijkstra, E. W. "Cooperating Sequential Processes" in F. Genuys (ed.), Programming Languages, Academic Press, 1968.

22. Barnes, G. H., et al. "The Illiac IV Computer" IEEE Trans. on Computers, C-17, No. 8, Aug. 1968.

23. Wulf, W. A., and Bell, C. G. "C.mmp-A multi-mini-processor: Proc. 1972 FJCC.

24. Heart, F. E., et al. "A new minicomputer/multiprocessor for the ARPA network" Proc. 1973 National Computer Conference.

25. Randell, B., and Russell, L. J. Algol 60 Implementation. Academic Press, 1964.

26. Johnston, J.B. "The Contour Model of Block Structured Processes" SIGPLAN Notices, 6, No. 2, Feb. 1971.

27. Wilner, W. T. "Design of the Burroughs B1700" Proc. 1972 FJCC.

THE IEEE COMPUTER SOCIETY TASK FORCE ON COMPUTER ARCHITECTURE

George E. Rossmann
Palyn Associates, Inc.
4100 Moorpark
Suite 201
San Jose, California 95117

The subject of computer architecture as currently taught in most computer engineering and computer science programs is a mixture of architectural principles, organizational strategies, and implementation techniques. This blurring of the hierarchy of system levels that characterize the structure of a computer has made it very difficult for students (and often instructors as well) to determine what were the forces that led to the design decisions they have seen reflected in machines. Furthermore, current courses in computer architecture pay insufficient attention to the fact that to a user the essential part of any computer system is its visible facilities: language processors, operating system, and other software. Therefore, they do not support the integration of hardware and software design that is required to create computer systems which satisfy the user.

In view of these circumstances, a task force was established by the IEEE Computer Society to prepare a detailed specification for a course of study in computer architecture for students whose major interest is in computer engineering or computer science. The members of the task force were:

George E. Rossmann, Chairman: Palyn Associates, Inc.

C. Gordon Bell: Digital Equipment Corporation

Frederick P. Brooks, Jr.: University of North Carolina, Chapel Hill

Michael J. Flynn: Stanford University

Samuel H. Fuller: Carnegie-Mellon University

Herbert Hellerman: State University of New York at Binghampton

The task force defined computer architecture by deciding what professional architects are supposed to do. We determined that the computer architect's task is to define computer systems that use hardware and software technologies so as to best satisfy all the users' needs, including function, economy, reliability, simplicity and performance. In carrying out this task, the architect must develop an understanding of the potential applications of each system and then bring to bear extensive knowledge of hardware architecture, operating systems principles, implementation details, component technologies and many other things to accomplish its design.

The task force prepared a report on a course of study in computer hardware architecture. The material presented in this report was restricted to those topics which we felt every computer engineer and computer scientist ought to know and to those computer systems which have been in the mainstream of commercial equipment. The material was organized into 11 modules, each dealing with a fundamental aspect of computer hardware architecture. The modules are:

Module No.	Title
1	Introduction and Meta Representation
2	Data Representation
3	Instructions and Addressing
4	Interpretation and Control
5	Memory Hierarchies
6	Protection Mechanisms and Hardware Aids to Supervision
7	Specialized Processors
8	Multiple Computers
9	Performance Evaluation
10	Reliability
11	System Design Evaluation.

Not much was said in the report about software and operating systems directly, but their influence on hardware architecture permeated all the modules. The report was published in the December, 1975 issue of Computer magazine.

THE MINERVA MULTI-MICROPROCESSOR

Lawrence C. Widdoes, Jr.
Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Abstract

A multiprocessor system is described which is an experiment in low cost, extensible, multiprocessor architectures. Global issues such as inclusion of a central bus, design of the bus arbiter, and methods of interrupt handling are considered.

The system initially includes two processor types, based on microprocessors, and these are discussed. Methods for reducing processor demand for the central bus are described.

1. Introduction

At Stanford University we are in the process of constructing the Minerva Multi-Microprocessor, designed to allow continuing experimentation with a class of inexpensive, extensible multiprocessor architectures.

Given the dramatic progress of LSI technology, it has become important to find modules suitable for LSI implementation which fit together as natural parts of a much larger system. A multiple-instruction multiple-data stream (MIMD) [FLY72] architecture may be partitioned so that each processor corresponds to a single module, yielding these desirable attributes:

1. Extensibility. An MIMD architecture is extensible with a nearly constant cost/performance ratio, to some limit [BAU75].
2. Reliability. Parts of an MIMD machine may fail and be repaired without catastrophic effects [HEA73].

Other advantages of multiprocessors are discussed in [LEH66], [HWA74], [RAV73], and [REY74].

From the hardware point of view, the goal of the Stanford Minerva experiment is to make contributions to the following questions regarding a low cost, extensible multiprocessor:

1. What bus structure and hardware communication protocols are suitable?
2. What simple mechanisms can be devised for reducing the problem of limited bandwidth of the communication channels?
3. How can the components of a multiprocessor be partitioned for LSI packaging?

In addition, the experiment has numerous software goals, but these will be considered elsewhere.

The remainder of this paper discusses the structure of the Minerva Multi-Microprocessor, and presents our conclusions about these hardware issues.

2. Minerva Hardware

The Minerva Multi-Microprocessor system consists of a compatible set of asynchronous devices organized around a single demand-multiplexed bus IDBUS (Inter-Device Bus). The organization of the multiprocessor is shown in Figure 1, and the devices shown there are briefly described in the Appendix.

For reasons of extensibility, the shared bus IDBUS is the only communication path between devices, with the exception that each device which is capable of initiating a *conversation* over the IDBUS has at least one direct communication path to the IDBUS ARBITER, which arbitrates control of the IDBUS. This communication path consists of a request line and a grant line.

We have currently designed two processor devices, called the 8080 CPU and the 3000 CPU, based on the Intel 8080 and 3000-series microprocessors, respectively. Minerva will support at least eight CPUs of the 8080 class, and four CPUs of the 3000-series class simultaneously without severe problems of IDBUS bandwidth. We hope to increase these numbers by using software methods to reduce processors' IDBUS bandwidth requirements.

2.1 IDBUS Structure

To meet our aims of extensibility and low cost, we chose a single, demand-multiplexed bus with a data-path width of 32 bits. Because we are installing 32-bit processors (3000 CPU), this data width is necessary. We are able to draw conclusions about interfacing processors with narrow data words to a wide bus by using eight-bit processors (8080 CPU), so this data width is sufficient. We chose a demand-multiplexed design because we intend to acquire enough processors to severely overload the bus and to study means of reducing bus loading; time multiplexing would eliminate this area of investigation. Finally, because multiple buses provide at best no increase in cost-effectiveness, we chose the simpler single-bus structure. In general, the conclusions we draw can be applied to multi-bus structures.

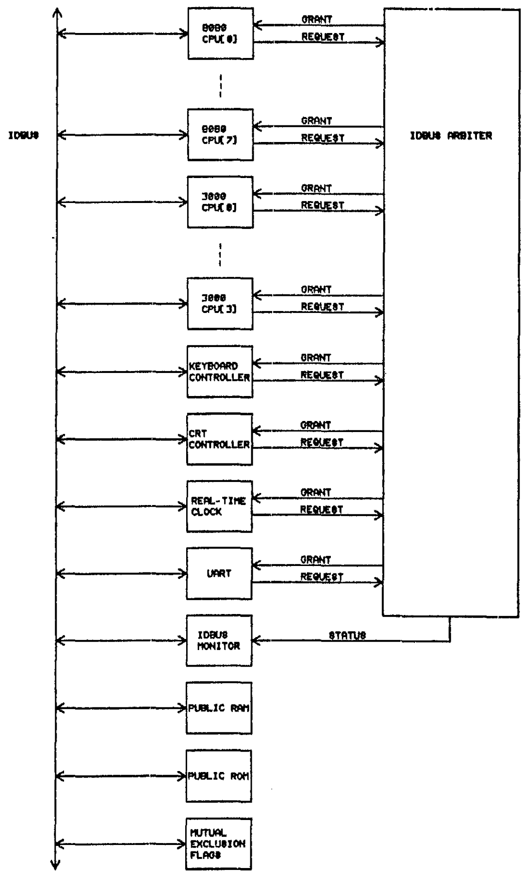


Figure 1
Minerva System Block Diagram

We use an IDBUS address width that allows access to 2^{30} bytes, large enough to allow the addition of virtual memory. Actually, the address consists of a 28 bit word address for read and an additional four byte-select signals for write. The byte-select signals enable writing of any combination of bytes within a four-byte word.

Any device which wishes to initiate a conversation over the IDBUS first reserves the IDBUS by means of a standard protocol: the device raises its request line and waits for its grant line to become asserted. For reasons of extensibility we do not daisy chain the requests and grants; daisy chaining leads to intolerable delays as the number of active devices grows. Although requests can be overlapped with bus operations, request turnaround time is important during periods of low bus utilization, for example, when a single processor is executing a largely sequential program segment; therefore a centralized arbiter is essential for a system containing many processors each with a widely variable bus bandwidth requirement.

Jordan and Baatz point out the deficiencies of a fixed-priority bus arbitration discipline [JOR74]. Although a fixed-priority discipline is tempting because of its simplicity, it is unsatisfactory. If the number of requestors is large enough to allow good bus utilization, then it is large enough to possibly lock-out the lowest priority requestor for an arbitrarily long time. If the maximum waiting time is not of interest for those processors which may be locked out, then this situation may be tolerable, but if processors are to be indistinguishable, and if a processor is to have the capability of meeting real time constraints, then lockout must be impossible; we must be able to calculate an upper bound on waiting time.

The IDBUS ARBITER is divided into two parts, a *priority* arbiter and a *FIFO* arbiter. Out of a total of 39 ports, 16 are FIFO ports, and the remainder are priority ports. The number of priority or FIFO ports can be easily increased at the cost of slightly longer arbitration time.

The priority ports are used for devices which have a predictable, non-saturating IDBUS bandwidth requirement. We include these ports primarily because it is often convenient for a device (eg., the CRT) which has a periodic, small IDBUS bandwidth requirement, to receive guaranteed fast service. Because the IDBUS demands of processors are generally dependent upon software, processors will ordinarily use FIFO ports. The 3000 CPUs actually have two request lines to the IDBUS ARBITER, one to a FIFO port, and one to a priority port. The priority line is used to make special IDBUS accesses under microprogram control. For example, one 3000 CPU acts as an arithmetic processor; arithmetic routines are stored in the local microprogram. Any processor requiring an arithmetic result places operands in main memory and requests service by sending an interrupt to the arithmetic processor. The operands are fetched from public RAM and the results are replaced in public RAM by requesting the IDBUS through a priority port. Since the original requestor was forced to wait in a FIFO queue for IDBUS access, this strategy does not greatly complicate the calculation of maximum waiting time for IDBUS service, but it significantly shortens maximum waiting time for arithmetic operations. The use of these priority lines will be carefully controlled during the construction of the microprograms, and therefore will not cause priority IDBUS demand to be unpredictable.

The disadvantage of FIFO arbitration is added hardware complexity, which may result in increased waiting time and increased cost. The increased waiting time is small, 100 ns waiting time for an eight-port FIFO arbiter versus 60 ns for an eight-port priority arbiter, both implemented with 7400-series SSI and MSI parts. The increased cost is not significant; the I/O pin count is obviously identical for all arbitration disciplines, and therefore the LSI implementation costs are essentially identical.

FIFO queueing is not the only discipline that will prevent IDBUS lockout; a rotating priority scheme would serve the same purpose and the costs are not significantly different. The arbitration time for a rotating priority discipline can be essentially identical to that of a fixed priority discipline.

2.2 Interrupt Structure

For reasons of extensibility, interrupts, like all other forms of communication, are constrained to be sent over the IDBUS. Interrupts are sent over the IDBUS from the interrupting device to the interrupted device by placing on the IDBUS an address which defines the IDBUS conversation as an interrupt and also defines the the interruptee processor, and data which defines the interrupting device and condition. Each device which has the capability to be interrupted listens to the IDBUS for its interruptee address.

Many I/O devices have the capability to send interrupts. Each I/O device which has this capability has, for each condition which may initiate an interrupt, two eight-bit internal registers, the *interruptee register*, and the *interrupt identity register*. These registers are accessible over the IDBUS. They are concatenated with appropriate constant signals in order to form the full address and data sent over the IDBUS during an interrupt conversation.

In conventional uniprocessor systems, the boolean variable which represents the state of an interrupt condition is stored at the interrupting device. Communication to a processor of the value of the variable is either by priority access to the central bus [DEC73], or by direct connection. In the first case the interrupt will be handled as soon as it is placed on the bus, and in the second case the interrupt may be handled at any time, since it does not tie up communications. In either case the interrupt is cleared when handled.

These conventional solutions are obviously unsatisfactory for an extensible multiprocessor. We cannot tie each interrupt flip-flop to each processor, and we cannot guarantee that an interrupt will be serviced as soon as it acquires the bus. We are therefore forced to store information about the state of an interrupt *at the interruptee*. The storage of such information causes a peculiar synchronization problem in disabling interrupts. That is, it is difficult to know, after a particular interrupt has been disabled, whether the interruptee will be interrupted by a previous event. The solution to this problem which avoids duplicating hardware in each processor is to include in each I/O device a status bit for each interrupt condition in that device. This status bit tells whether the interrupt condition has sent an interrupt over the IDBUS since the last time the bit was cleared; it is tested after the interrupt condition has been disabled.

The resettable interruptee feature provides much flexibility in interrupt handling mechanisms. In one degenerate case, all I/O device interrupts can be directed to a single, fast, interrupt handling processor. The 3000 CPU is well suited for this task, although the 8080 CPU is much too slow. The interrupt handling processor is responsible for scheduling all work for the other processors; they receive interrupts only from the interrupt handling processor. This idea has appeared elsewhere [LAM68]. Such an interrupt handling mechanism requires a fast processor in order to service the worst case accumulation of interrupts within desired time constraints.

We are able to implement interrupt handling mechanisms which do not require a fast processor, but are only slightly less general, by using the full power of the resettable interruptee. For example, whenever a processor receives an interrupt of type A, before servicing the interrupt it can examine the state of the other processors and assign the next interrupt of type A to the best processor. This mechanism is less effective than the centralized design because the assignment of work is made considerably before the work is to be done, and so this strategy will work only in an environment of periodic interrupts with predictable computing requirements. High processor speed is not required because task scheduling is distributed among all processors.

The function of the resettable interrupt identity feature is simply to allow reshuffling of priorities in cases where the interrupt identity is mapped onto priority levels at the processor.

Any processor which can access the entire IDBUS address space automatically has the capability to send interrupts; a processor can, in fact, simulate the occurrence of any type of interrupt by placing the appropriate data on the IDBUS. This feature facilitates experimentation with various interrupt handling strategies, and also allows processors to exchange signals without polling.

2.3 Mutual Exclusion

The software must deal with resources which need to be accessed sequentially, and it is therefore important to provide a basis for implementing software mutual exclusion primitives which is somewhat more convenient than IDBUS interlock [DIJ68].

Although the common solution is to allow all public RAM locations to be accessed with an indivisible read-modify-write operation, we chose to provide special mutual exclusion flags, for reasons of cost. Minerva has 256 mutual exclusion flags accessible over the IDBUS. A mutual exclusion flag becomes asserted whenever it is read, and in the same IDBUS cycle. It is cleared by a write. These flags thus implement the test-and-set primitive, from which higher-level mechanisms can be built.

In order to allow mutual exclusion on a lower level, a control signal is included in the IDBUS which retains IDBUS control for the grantee during the next bus cycle. This facility allows microprogrammed processors to implement a read-modify-write cycle, and also allows devices which are not processors to perform mutual exclusion if necessary without the extensive control structure required for using the mutual exclusion flags.

2.4 8080 CPU Structure

The 8080 CPU is a processor module based upon the Intel 8080 chip [INTA75]. It is shown in Figure 2, and consists of the 8080 chip and ancillary circuitry. The major functions of the ancillary circuitry are to receive interrupts, to provide private memory, to provide private I/O (eg., local status registers not accessible over the IDBUS), and to interface the 8080 chip's 8 data lines and 16 address lines to the 32 data lines and 32 address lines of the IDBUS.

All directly accessible IDBUS locations and all private I/O locations are accessed by memory reference instructions; we have converted the 8080 IN and OUT instructions to reference the lowest 256 bytes of private RAM. Addresses are translated before being placed on the IDBUS; commonly accessed IDBUS locations are part of the 16-bit 8080 address space, and all other locations can be accessed by indirection through an address register set up by a private I/O operation.

Although use of the 8080 chip as the basis for a processor module is not currently a cost effective way to obtain instruction executions per second in the Minerva system, we were interested in the problems involved in interfacing an eight-bit machine to a larger bus. The total gate count of the ancillary circuitry excluding private RAM is only about 700, and therefore it is easy to imagine that this hardware could be placed on bus interface chips organized in a bit-sliced architecture and controlled by an eight-bit CPU chip. Furthermore, the cost of the 8080 CPU device is very low, so that even with limited resources we have been able to begin construction.

2.4.1 Private Memory

Each 8080 CPU device includes 4K bytes of private RAM, organized as 4K by 1 byte in order to interface directly with the 8080 CPU chip. In order to reduce the cost of the 8080 CPU device, the private RAM is not directly accessible over the IDBUS. Software protocols have been established to allow one processor to access the private RAM of another, but private RAM is not normally accessed in that mode.

The primary purpose of the private RAM is to allow us to explore software methods of reducing IDBUS loading. Private RAM is located at the bottom of the 16-bit address space seen by the local 8080 CPU chip; private RAM and public RAM are addressed uniformly. Routines that are not location dependent can therefore be relocated from private RAM to public RAM or vice versa without modification, allowing dynamic optimization of the location of code.

Commonly used constants and reentrant routines are stored in private RAM, as well as some variables that are local to a process. In particular, the routine which copies from private to public RAM and vice versa is located in private RAM to minimize IDBUS loading. The storing of local variables in private RAM introduces a problem in task switching: If a process is switched from processor A to processor B, then B must either request the local variables from A, or the switching must occur at a time when the local variables are dead. It is possible to allow the top levels of a process stack to be stored in private RAM, and copying the stack is unnecessary if the process switches processors only upon exit from all blocks whose stack is privately stored. In order for the scheduler to make good use of this strategy, apriori knowledge of the probable execution times of process blocks is required.

Each 8080 CPU module has a unique, read-only *processor identity*. The processor identity is the ultimate basis for all differentiation between processors, and private RAM is used to accelerate access to private information by eliminating indirection. For example, interrupt vectors are in general different for each processor, and since they are stored in the lowest 4K of the 8080 chip's address space, it is not necessary to use software indirection to locate the proper vector at interrupt time.

The private RAM effectively makes the 8080 CPU into a processor which can be dynamically programmed to perform special macro functions without using IDBUS bandwidth. In this sense it is similar to a dynamically microprogrammed processor, except that its implementation is currently less expensive, and its "micro" language is the same as its "macro" language, resulting in the feature that code can be relocated from private RAM to public RAM and vice versa.

2.4.2 Instruction Stack

As an inexpensive means of reducing IDBUS loading caused by the 8080 CPU, we chose to employ instruction *prefetching*. Although instruction prefetching has been used elsewhere [BEL71] and a similar idea, the retention of previously executed sequential instructions, has also been used [THO64], previous uses have been for the purpose of decreasing average memory access time. The same issue can be considered from the point of view of decreasing bus loading.

The 8080 chip contains an eight-bit data bus, and executes

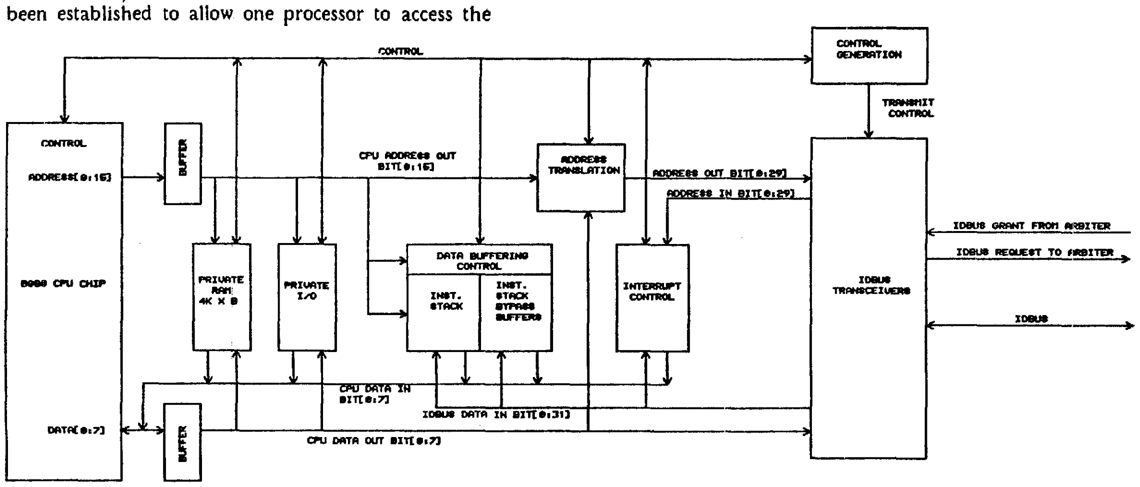


Figure 2
8080 CPU Block Diagram

instructions from one to three eight-bit bytes in length. Most instructions affect one byte of data, and in efficient programs the data byte is usually located in an internal register. As a consequence, most memory accesses are instruction fetches. This observation is supported by the data in Table 1, which were collected by simulating a mix of arithmetic and memory management routines. It should be noted that the programs sampled showed a wide range of values for the quantities listed; the range is indicated. Furthermore, the 8080 programs sampled have a strong tendency to execute sequential instructions. The sequentiality of programs has been asserted elsewhere [SHE68].

Fraction of Memory Byte Operations Represented by:	
Fetching All Instructions and Immediate Data	.80 to .99
Fetching First Byte of Non-Sequential Instruction	.01 to .05
Fetching Non-Immediate Data or Stack	.01 to .15
All Write Operations	.01 to .10

Table 1
8080 Instruction Stream Characteristics

These observations lead to the conclusion that using the entire 32-bit IDBUS data path to prefetch instructions would significantly reduce the 8080 CPU's IDBUS demand. We therefore implemented a four-byte *instruction stack* which operates as follows: When an instruction byte which is not in the instruction stack must be fetched over the IDBUS, the four bytes containing the desired byte and starting on a four-byte boundary are fetched and retained in the instruction stack. When an instruction byte which is already in the instruction stack is required, it is simply fetched from the instruction stack. Immediate operands are considered to be instruction bytes.

It is possible that an 8080 CPU may loop in its instruction stack, never fetching a new instruction over the IDBUS. A case in point is a jump to the current location; the instruction stack would not reflect any changes that might occur in that instruction in public RAM. This is a minor consideration; we do not intend that exit from four-byte loops should be dependent upon another device changing the looping instructions in public RAM. On the other hand, this problem should be considered for larger instruction stacks.

The case in which an 8080 CPU writes into its own instruction stream is more plausible than the case in which another processor writes into an 8080 CPU's instruction stream, and it is therefore important to prevent the anomaly in which a processor would write into its own instruction stream less than four bytes ahead of the current execution address, yet not execute what it had written because the instruction stack had not been updated by the write.

The instruction stack may be contrasted with a *cache*, which can retain non-sequential data and uses a more complicated replacement strategy. A cache is thus able to exploit the property of *program locality*, which includes most of *program sequentiality*, but also much more. Unfortunately, the cache is more complicated to implement, although it would involve the same number of I/O pins, and it suffers from the severe problem that if it is big enough to exploit locality, then it is big enough that out-of-date information becomes a problem. If the only information stored in the cache consists of reentrant procedures, then this problem is minimal. On the other hand, if the cache can contain only reentrant procedures, then much of the locality property is lost. We return to this issue in our discussion of the 3000 CPU cache in Section 2.5.1.

Unless the 8080 instruction stack were to be made large enough to allow an instruction in the stack to be used, on the average, significantly more than once, it would not pay to make the instruction stack larger than the data path, that is, four bytes. Even assuming a wider IDBUS data path width, the data path utilization decreases as the instruction stack size increases, as shown in Table 2. Here the data path utilization is defined to be the number of useful data bytes in an average bus operation divided by the width of the data path. The data in Table 2 were collected from the same programs as the data in Table 1, and show the same variability.

Stack Length (Bytes)	Average Data Path Utilization
1	1.0
2	.80 to .95
4	.60 to .90
8	.35 to .80
16	.20 to .60

Table 2
8080 Instruction Stack Size vs. Data Path Utilization

By setting a bit in a local status register, the 8080 CPU can cause every IDBUS fetch to be placed in the instruction cache. Executing in private RAM, the 8080 CPU can therefore access bytes in sequential IDBUS locations using minimal IDBUS bandwidth.

Our instruction stack requires four eight-bit, tri-state data latches, four four-bit address latches, and a small amount of control logic. This is in addition to the latches required for IDBUS data which is routed around the instruction cache.

Instruction prefetching may be contrasted with a multiple-bus design. An alternative IDBUS structure would consist of four eight-bit buses; each processor would have access to each bus, and public memory would be four-way interleaved. This design involves additional bus control at each processor, and additional bus arbiters. More importantly, it involves replication of bus address and control signals. This problem becomes relatively less important as the ratio of data width to address width increases, but an increased data width results in increased cost and also in less efficient use of the data word. A 32-bit data word is common; with 20 address signals, six control signals, and 32 data signals in each bus, a four-bus structure requires 232 signals, while a single bus with 32*4 data bits requires only 154 signals. Our instruction stack design requires only about 68 bus signals, and corresponds to the four-bus structure in which each bus has eight data bits, 30 address bits, and six control signals, a total of 176 signals.

On the other hand, multiple buses and interleaved memory allow more efficient use of the data path. With the instruction stack design, each write operation requires a full IDBUS cycle and therefore wastes three bytes of the data path. Other inefficiencies lie in data reads and wasted instruction bytes. Our instruction stack allows a data path utilization of from .6 to .9, (see Table 2), whereas the four-bus design would give a data path utilization of 1 (no waste). Assuming that looping in the instruction stack does not take place, in both cases data path utilization is proportional to maximum attainable total throughput.

2.5 3000 CPU Structure

The 3000 CPU is a 32-bit microprogrammed processor module based upon the Intel 3000 chip set [INTB75].

The 3000 CPU is largely conventional, executing microprograms from a RAM control store of 1K by 32 bits. The RAM control store is locally accessible, and is also accessible over the IDBUS. The design of the 3000 CPU is similar to the design of the 8080 CPU in interrupt reception, provision of private memory, and provision of private I/O. These aspects will not be discussed further. However, the problem of interfacing the high speed 3000 CPU to the IDBUS required a unique solution, and this is discussed in the next section.

2.5.1 Cache

Depending upon the macro-instruction set, one 3000 CPU is capable of almost fully loading the IDBUS. We intend to experiment with the design of instruction sets which reduce IDBUS loading, but microprogramming alone cannot efficiently take advantage of the dynamic locality of programs.

Fortunately, there has been much work done on the design of *caches* to decrease average memory access time [KAP73]. The dual role of a cache is that it can dramatically decrease the loading of a shared memory bus.

Our cache has a capacity of 256 *buckets*, each containing 32 data bits, 20 address bits, and two control bits: valid and modified. Addresses presented to the cache control are 28 bits, specifying a full IDBUS word address. Addresses, data, and control bits are presented to the cache under microprogram control; the cache is simply a RAM with special control logic to perform certain cache operations without microprogram intervention.

The microprogram uses the cache in three modes as follows:

1. Reading. If the address is a cache hit, then read from cache, otherwise create an empty cache bucket, read from IDBUS, store the address and data into cache, set modified to false, set valid to true, and read from cache.
2. Writing a *non-shared address*. If the address is a cache hit, then set modified to true, store the data, and exit. Otherwise, create an empty cache bucket, set valid to true, set modified to true, and store the address and data.
3. Writing a *shared address*. If the address is a cache hit, then set modified to false and store the data. In any case, perform a write over IDBUS.

In order to create an empty cache bucket, first the cache resident to be removed is chosen by direct addressing (hashing) using the address of the new cache resident, and if modified is true in the resident to be removed, then the resident to be removed is written over IDBUS.

Shared and non-shared locations are determined by software convention, and are differentiated by a special address bit. The 3000 CPU macro-instruction address space is thus folded; a real IDBUS location has one name as a shared location and another name as a non-shared location.

Our highest-level programming language will be modeled after Concurrent Pascal [BRI74], and will require explicit declaration of shared variables. The addition of a special address bit to differentiate between shared and non-shared variables causes problems for processors which have access to only a small address space and therefore cannot afford to fold that space, eg., the 8080 CPU, but such processors will treat all variables as shared and will not have caches.

The strategy outlined above for performing reads and writes uses minimal IDBUS bandwidth; besides reads of addresses which are not cache hits and writes of evicted cache residents, only writes to shared addresses require use of the IDBUS. This reduction in IDBUS loading is accomplished by maintaining redundant data in the various caches, and so provision must be made for keeping the redundant data consistent. Each 3000 CPU cache continually listens to the IDBUS for writes. Whenever the cache detects an IDBUS write, the IDBUS address being written is stored in a 28-bit buffer. The buffer is emptied by invalidating the cache resident having the address which was overwritten, if there is such a resident. Emptying the buffer occurs between processor requests to access the cache; this operation is implemented in hardware. There is no problem with buffer overflow, since the cache can be examined and a bucket invalidated significantly faster than the IDBUS write cycle time, and such buffer emptying operations have priority over microprogram requests for cache operations.

Writes over the IDBUS are not partitioned into writes to shared locations and writes to non-shared locations, therefore more cache operations are done than necessary, since optimally only writes to shared locations would affect every cache.

Incorrect results are obtained if two different 3000 CPUs attempt to write into the same address and use the non-shared name for the address. We count on a specially constructed high level language with concurrency primitives to prevent this type of error. Such a language is important for reliable programming with concurrency, regardless of the hardware. Brinch-Hansen discusses this issue [BRI73].

The 3000 CPU has the capability to perform an IDBUS read or write without using the cache. This capability is used by convention when accessing the mutual exclusion flags in order to prevent synchronization problems that would be caused by the existence of multiple copies of these flags.

Although our 3000 CPU design uses only random MSI and SSI in addition to the Intel 3000 chip set, the design can be partitioned into a small set of LSI chips. In the LSI implementation, the cache might operate without microprogram control, intercepting all IDBUS accesses except when explicitly inhibited, and also monitoring the IDBUS for writes to resident addresses.

3. Progress and Goals

We have completed the construction and debugging of a nucleus system containing one 8080 CPU, the IDBUS ARBITER, and several peripheral controllers, and have commenced construction of the remaining peripherals and a 3000 CPU. These devices will be completed by the middle of 1976. Several additional CPU's will then be added.

In parallel with the hardware construction, we are proceeding with the design of an operating system and a compiler for Concurrent Pascal. The philosophy in writing the operating system is to do task scheduling dynamically in order to create an environment in which the physical implementation does not have to be explicitly considered during programming.

4. Conclusion

Very low microprocessor costs allow us to consider connecting microprocessors together to form a low-cost extensible multiprocessor. To keep costs minimal, a central bus is used, and hardware in each processor exploits properties of programs in order to reduce IDBUS loading. Such hardware can be incorporated into LSI microprocessor chip sets with little additional cost.

The bus arbiter is centralized in order to reduce bus access time, which is important when bus utilization is low. The arbiter uses a FIFO scheduling discipline to bound bus lockout time.

Maximum flexibility of scheduling disciplines is provided by including a resettable interruptee register for each interrupt condition, and extensibility is retained by directing an interrupt to the appropriate interruptee over the central bus.

Acknowledgements

This research was performed while the author was a Fannie and John Hertz Foundation Fellow. The work was partially supported by the Joint Services Electronics Program under contract N00014-75-C-0601. The author wishes to express appreciation for the cooperation of Intel Corporation and of the Stanford Artificial Intelligence Laboratory.

Appendix

3000 CPU[0:3]
These devices are processors based on the Intel 3000 chip set. Control store is 1K 32-bit words of RAM. The 3000 CPU has 1K 32-bit words of private memory accessible by microprogram. A 512-bucket cache is used to reduce IDBUS loading.

8080 CPU[0:7]
These devices are processors based on the Intel 8080 CPU chip. Each contains private memory and has a 4-byte instruction cache used to reduce IDBUS loading. The 8080 CPU can access the entire IDBUS address space; less frequently referenced areas are accessed by indirection.

CRT CONTROLLER (CRT)
This device controls a standard raster-scan video monitor. It continuously reads lines from the PUBLIC RAM and displays them on the monitor, requiring less than 3% of the IDBUS bandwidth. It requires a data structure in RAM consisting of a linked list of lines of ASCII characters, with the head of the list at any location. Character fonts are stored in a special RAM accessible over IDBUS and are variable. The CRT never interrupts any processor.

IDBUS
The IDBUS is the only communication path between devices. It has a data path width of 32 bits, and an address width of 28 bits plus four byte-select signals.

IDBUS ARBITER
The IDBUS ARBITER arbitrates control of the shared bus IDBUS. It has two classes of request ports, priority ports and FIFO ports. The priority ports handle requests in fixed priority order, and the FIFO ports handle requests in FIFO order.

IDBUS MONITOR
The IDBUS MONITOR continually monitors the IDBUS and the IDBUS ARBITER. In a 256-bucket buffer, the IDBUS MONITOR records for each IDBUS cycle, or for a random sampling of IDBUS cycles, the grantee, the slave address, and a floating-point time interval since the last recorded IDBUS cycle. This buffer interrupts when full, and can be read over the IDBUS.

KEYBOARD CONTROLLER (KBD)
This device is the IDBUS interface for an ASCII keyboard.

MUTUAL EXCLUSION FLAGS (MUTEX)
MUTEX consists of 256 flags, addressed as memory and organized as one per 32-bit word. Two operations are defined for these flags: test-and-set and clear. These flags are used to easily implement mutual exclusion.

PUBLIC RAM (RAM)
The PUBLIC RAM consists of 16K 32-bit words of MOS dynamic memory having about 300 ns access time and 500 ns read cycle time. During a write, any bytes of the addressed word may be independently inhibited from being written.

PUBLIC ROM (ROM)
The ROM consists of 1K 32-bit words of reprogrammable MOS read-only memory having about 1.5 us access time. It is read over IDBUS and is used to bootstrap the system at power-on time.

REAL-TIME CLOCK (CLK)
The CLK is a 32-bit counter which increments at 10 us intervals. It will be used for time-of-day calculation and high-resolution interval timing. It delivers all 32 time bits when read. It has the capability to interrupt at the 32-bit alarm time which is stored in a single alarm register addressed as a memory location, and it has the capability to interrupt at timer overflow.

UART
This is the interface to a standard UART.

References

BAU75 Baum, A., and Senzig, D., "Hardware Considerations in a Microcomputer Multiprocessing System," *Proc. 1975 IEEE Computer Conference*, pp. 27-30

BEL71 Bell, C. G., and Newell, A.,"The IBM 7094 I, II," *Computer Structures: Readings and Examples*, McGraw-Hill, 1971, pp. 517-541

BRI73 Brinch-Hansen, P., *Operating Systems Principles*, Prentice-Hall, 1973

BRI74 Brinch-Hansen, P., "Concurrent Pascal: A Programming Language for Operating System Design," California Institute of Technology Information Science Technical Report 10, 1974

DEC73 Digital Equipment Corporation, *Peripherals Handbook*, Maynard, Massachusetts, 1973

DIJ68 Dijkstra, E. W., "Cooperating Sequential Processes," in *Programming Languages*, (F. Genuys, ed.), Academic Press, 1968, pp. 43-112

FLY72 Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol. C-21, No. 9, September, 1972, pp. 948-960

HEA73 Heart, F. E., et. al., "A New Minicomputer/Multiprocessor for the ARPA Network," *Proc. AFIPS 1973 National Computer Conference*, Vol. 42, pp. 529-537

HWA74 Hwang, K., et. al., "Multiprocessor System Design: Architecture, Control, and Hardware Organization," IEEE Computer Group Depository No. R74-84, 1974

INTA75 Intel Corporation, *Microcomputer Systems Manual*, Santa Clara, California, January, 1975

INTB75 Intel Corporation, *Intel Data Catalog*, Santa Clara, California, 1975

JOR74 Jordan, Bernard W., and Baatz, Eric L., "C.MUP -- Northwestern University's Multimicrocomputer Network," *Proceedings of the 1974 IEEE Symposium on Computer Networks: Trends and Applications*, pp. 51-56

KAP73 Kaplan, K. R., and Winder, R. O., "Cache Based Computer Systems," *Computer*, Vol. 6, No. 3, March, 1973, pp. 30-36

LAM68 Lampson, B. W., "A Scheduling Philosophy for Multiprocessing Systems," *Communications of the ACM*, Vol. 11, No. 5, May, 1968, pp. 347-360

LEH66 Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," *Proc. of the IEEE*, Vol. 54, No. 12, December, 1966, 1889-1901

RAV73 Ravindran, V. K., and Thomas, T., "Characterization of Multiple Microprocessor Networks," *Proc. 1973 IEEE Computer Conference*, pp. 133-135

REY74 Reyling, G., "Performance and Control of Multiple Microprocessor Systems," *Computer Design*, Vol. 13, No. 3, March, 1974, pp. 81-86

SHE68 Sherry, S. S., and Flynn, M. J., "Addressing Patterns and Memory Handling Algorithms," *Proc. AFIPS 1968 Fall Joint Computer Conference*, Vol. 33, pp. 957-967

THO64 Thornton, J. E., "Parallel Operation in the Control Data 6600," *Proc. AFIPS 1964 Fall Joint Computer Conference*, Vol. 26, pp. 33-40

A HIERARCHICAL, RESTRUCTURABLE MULTI-MICROPROCESSOR ARCHITECTURE

R. G. Arnold
Laboratory for Computer Science and Engineering
Department of Electrical Engineering
Rice University
Houston, Texas

E. W. Page
Department of Electrican and Computer Engineering
Clemson University
Clemson, S. C.

ABSTRACT

This paper introduces a system architecture which allows a high degree of restructuring so that system resources may be tailored to processing requirements. The proposed system organization consists of a large number of byte-slice processors interconnected through a system of busses. Each processor is capable of communicating with every other processor in the system and any number of adjacent processors may be strung together to create a wider arithmetic capability than is possible with a single processor. Processors may be organized into a number of independent teams while processor teams may, in turn, be organized in a hierarchical fashion to allow for concurrent processing. Processor teams may function either in cooperation with or completely independent of other processor teams.

All communication throughout the system consists of information packets containing the data to be transferred and a series of tags which indicate the destination address for the data and the action to be taken by the processor upon receipt of the information packet.

Two types of busses are employed: Conventional busses and the circulating loop (or Pierce loop). The circulating loop moves an information packet in a fixed direction a uniform distance in each unit of time and therefore allows independent data transfer operations to be carried out simultaneously. Several examples illustrate the utility of the proposed architecture.

INTRODUCTION

Computer technology appears to be reaching a point of diminishing returns in attempts to increase the basic speed of a large-scale processor. Regardless of the advances in hardware technology, there have traditionally been requirements for architectural innovations to gain increased speed and capabilities as well as added flexibility. Historically, the major concerns for parallel designs centered upon the efficient utilization of hardware resources. With the recent revolution in the capabilities and economics of large-scale integration technology, the cost of a basic central processing unit has decreased to the point where it is no longer a significant fraction of total system costs. At present, the cost of software development is of major concern even in conventional architectures and will likely be the limiting economic factor in architectures which have been developed to exploit program parallelism. This paper is directed towards the development of fundamentally different computer architectures for the efficient utilization of an aggregate of the newly available microprocessors operating concurrently to gain increased computational power.

In order to realize the potential advantages of the concurrency of operations possible in multiple-processor systems, an adequate system for communication and control among a multitude of processors must be developed. In the past, multiple-processor systems employed only a small number of complete processors or large numbers of slaved functional units and were structured accordingly. The communication between processors has often been achieved through the use of a dedicated set of channels, multi-port memories, a cross-bar switch, time-shared busses, or combinations of these methods; typically, the control arrangements have become less flexible as the number of processors increased. Thus, systems of the past are often unwieldy and impractical in terms of current desires.

T. C. Chen[2,3] has demonstrated the weaknesses in traditional concurrent systems and provided motivation for the development of multiple-processor systems that are loosely coupled with a high degree of local intelligence and autonomy. In his discussion on the efficiency of traditional, tightly coupled, concurrent systems, Chen shows that for small deviations from the ideal, perfectly parallel task to a real task with small amounts of serial or sequential requirements, the efficiency of a tightly coupled, concurrent system takes a precipitous drop. The efficiency falls initially at a rate of M-1 where M is the number of parallel elements in the system; the greater M is, the more significant the impact of less than perfectly structured, perfectly parallel problems. Since no two problems are ever quite the same, this also provides motivation to have the system adapt to fit the problem rather than distorting the problem to make it amenable to solution by the data processing system.

As a result of considerations such as those previously mentioned, a new system of processor should meet the following requirements:

1. A large number of processor modules should be possible.
2. Uniformity of modules from the point of view of the communication/control structure should exist.
3. Each processor module should be capable of communication with all (or most) other processor modules.
4. Blocks of processors should be able to function as a team, independently of other teams.
5. A hierarchy of control should be possible as shown in Fig. 1.
6. A dynamic ability to reconfigure the system (i.e., rearrange the hierarchy of control) to fit the system to the problem, thus allowing the system to appear as a Von Neuman machine, a parallel array processor, an associative parallel processor, etc., as required.
7. Considerations such as reliability, fault-tolerance, and graceful degradation demand the incorporation of redundancy and a capability for dynamic reconfiguration.

The system described here has been developed to satisfy the preceding requirements.

GENERAL SYSTEM OVERVIEW

As illustrated in Fig. 2, the proposed system consists of a number of modules containing microcomputers and ancillary circuits connected by a series of busses, loops, and SHORT or BLOCK/SHORT modules. All interprocessor communication takes place on the various busses. Each processor has its own independent memory and is capable of performing any of the system tasks assuming it has been suitably programmed. Along with the various elements of hardware in the system, a basic system philosophy and a set of communication protocols are required. It is intended that this system be restructurable and capable of being organized in a hierarchical fashion. As such, there will generally be one processor (any processor) responsible for overall system action. This processor designates subordinates, establishes the chain-of-command and directs its immediate subordinates in the tasks they are to perform. In order to implement this philosophy, the following basic characteristics/protocols will be incorporated into the design:

1. Each module will be named, both with a unique, permanent name, a "P-name" and with a name that is changeable, a "V-name." Each V-name consists of two parts: A Block or team name and an element name. All communication is carried out by tagging or addressing the information with the destination V- or P-name and placing it on a bus. Thus, data or commands may be passed to a module by specifying both the block and element names or to all modules in a block by specifying the block name and "XX" for the element name where "XX" specifies a "universal" name to which all modules respond. Likewise, information can be passed to all modules simultaneously by specifying "XX, XX" as the V-name.

2. All commands sent by a master or controlling module must be taken in by its subordinate and acknowledged. The subordinate queues the commands pending the arrival of the appropriate operands.

3. Task completion must be signaled.

4. Several adjacent processors may be strung together to form a wider arithmetic ability than would otherwise be available.

5. All communication throughout the system will consist of information packets containing the data to be transferred and a series of tags. Since each processor is identified by a name, all ambiguities associated with the transfer of information are resolved through the use of the processor names. In addition to the destination P- or V-name, each packet will contain tags uniquely associating the operands with the commands in a possible queue or other temporary storage medium. For data packets, a 1 bit tag will also indicate the order of the operands for non-commutative operations.

DESCRIPTION OF SYSTEM ELEMENTS

The heart of each system module is the microcomputer itself. Each microcomputer, the microprocessor with its memory, will be microprogrammed to provide all the basic functions of a standard processor and to respond appropriately to the actions of the system. It should automatically perform overhead type tasks. For example, it should maintain a queue of commands and automatically acknowledge the receipt of commands. Generally, the programs would consist of a series of subroutines whose call would be initiated by commands received from more superior elements of the hierarchy.

Each processor must have a priority interrupt capability such that interrupts occurring below the processor's priority level are masked. It must also have lines for the "carry out" generated by an arithmetic operation or left shift. Likewise, it should also have a "carry in" capability. Currently available processors organized on a bit slice basis provide these features.[9] There is no requirement as to word length, speed, etc., for the processor.

Communication between processors is provided by the system of busses. There are two basic types of busses employed in the system: Conventional time-shared busses and circulating busses. The circulating bus or C-Bus, often referred to as the Pierce Loop,[6,7,8] can be conceptually considered to be a circulating loop that moves a packet of data in a fixed direction a uniform distance in each unit of time. Any user can transmit by placing an information packet on the bus anytime a gap in the circulating traffic appears at its location. Each user must continually monitor the traffic passing its location. When a user recognizes that a packet passing its location contains its address (or name), the user removes the packet from the bus. The packet's former position in the traffic stream is now a gap, free to be filled with a new packet by any user. The C-bus thus provides a temporary memory or queue of information and is a means by which several independent data transfers can be carried out simultaneously. There are two classes of C-busses, the DONE busses and DATA bus. Their functions will be explained later. Conventional busses are of the typical "party line" arrangement having one transmitting user and many receiving users at a time; the CMD (Command) and ACK/NAK (Acknowledge/Negative-Acknowledge) busses are of this type. Each of the conventional busses is equipped with a bus controller to arbitrate conflicting requests for the bus for transmission. Any user desiring to transmit must be granted permission by the controller.

The functions of the busses can also be separated into two divisions: Data transfer and control. In order to control the system efficiently, several sets of busses providing command and control capabilities have been grouped together. Each set will collectively be termed a Control Group (C.G.). Each Control Group competes for attention from each processor on a priority basis much as in the case of a priority interrupt system. The Master Control Group (M.C.G.) is the highest, most significant priority or 0th level (C.G.0.). Each additional Control Group is on level 1, 2, etc. Each Control Group other than the Master can be blocked/terminated at the left edge of any processor by activation of the BLOCK/SHORT module. By this it is meant that conventional busses are blocked, circulating busses are "shorted" or the loop is closed. Each Control Group consists of a CMD bus, a DONE bus and an ACK/NAK bus. The CMD bus carries commands to processors. When a processor name matches the name attached to a command on a CMD bus at level "n," an interrupt to the processor is generated on interrupt priority "n." If the processor priority is lower than "n," the interrupt is accepted and the command is recognized as destined for this processor. A processor recognizing a command is obligated to reply positively on the ACK/NAK bus if the command can be accepted into its command queue. Otherwise, the processor replies with a negative acknowledge or NAK. The DONE bus provides the means by which the command processor can acknowledge the completion of the required task.

Data transfers are carried out on the DATA C-bus. A data item is placed on the DATA bus in the form of an information packet containing the data and the destination processor name. As the packet circulates around the bus, the destination name is compared to the name of each processor. When a match occurs between the name on a data item and a processor, that processor is signaled and it is obligated to remove the data item from the bus.

The basic bus formats for information packets with an explanation of the various fields is given below:

CMD BUS

P/V	NAME	OPERATION #	COMMAND

DATA BUS

P/V	NAME	OPERATION #	1/0	DATA

DONE BUS

P/V	Name	OPERATION #

ACK/NAK BUS

P/V	NAME	A/N

P/V	-- 1 bit that indicates that the content of the name field is to be interpreted as a module's permanent name (P), or its V-name (V).
Name	-- the name of a processor. When interpreted as a V-name, it consists of 2-parts, the block and the element name.
Operation #	-- Each command sent to a module is numbered and held in memory in numerical order by the receiving processor until its operands are present and there are no commands having a lower number in memory. The operands are uniquely identified as belonging with a particular command by a matching Operation #.
1/0	-- In the DATA bus format, this indicates the order of the two operands for non-commutative operations.
DATA	-- The actual operands, etc., transmitted on the DATA bus.
A/N	-- 1 bit that indicates the positive acknowledgement (A) of the receipt and acceptance of a command or a negative acknowledgement (N) indicating that the named module is unable to accept or perform the required operation.

The SYNC/CARRY LOOP also transfers data throughout the system. It is designed to transfer information shifted or "carried out" from the arithmetic section of a processor to the arithmetic section of another processor. This allows several processors to function as a single multiprecision arithmetic unit. The SYNC/CARRY LOOP passes through each processor module and has no storage of information. By activation of the appropriate SHORT modules, as shown in Fig. 3, the LOOP may be gated through the processor proper or past it. In a similar manner, it may also be "shorted" at the left edge of each processor module, i.e., it may be broken into two closed loops at the left end of the module.

In addition to the various busses, the items mentioned previously as BLOCK/SHORT modules and SHORT modules perform an important function in the implementation of a hierarchical structure. The BLOCK/SHORT modules are a part of every Control Group except the Master Control Group. Their function is to divide a Control Group into independent sections. This allows several teams of modules to operate independently in the same Control Group. Each BLOCK/SHORT module is controlled by the processor to its immediate left. Figure 4 illustrates the utility of the BLOCK/SHORT modules. Here C.G. 1 is broken into two independent parts with each section functioning just as if it were a complete C. G. Processor 2,1, for example, can then control 3,1 and 3,2 without any interaction with other processors on C.G. 1.

BASIC ILLUSTRATIONS

The system, when viewed in an unstructured, idle configuration, will appear as a collection of processors arranged in a cylindric fashion connected by a collection of busses. However, this structure, when viewed in an active state, will generally be divided into a collection of teams of processors in a hierarchy of responsibility and control. Structuring takes place in the following fashion:

1. Initially, the user will designate a processor as the master and load its memory with the appropriate programs. This processor then begins execution.

2. The master would decide which of the various processors will perform particular tasks.

3. The master commands each processor in turn to load the program being sent to it over the DATA bus.

4. Each processor then sets its V-name and priority to the values sent it on the DATA bus upon command of the master.

5. The appropriate modules are then commanded to activate their BLOCK/SHORT or SHORT modules, as required.

For example, the hierarchy shown in Fig. 5 may be defined in the system by activating the appropriate BLOCK/SHORT modules, naming the processor appropriately and specifying their priorities or the level on which the module expects commands. The 0th module has been established with the V-name of "1,1" and designated as the most superior element in this structure. Modules 1 and 5, assigned V-names of "2,2" and "2,1," respectively, are both directly controlled by "1,1" and expect commands at the 0th priority level, i.e., from the master control group. Module 1 (named "2,2") controls directly the three modules 2, 3 and 4 (named 3,1; 3,2; 3,3; respectively) through commands on the control group at the 1st priority level. Note that since the BLOCK/SHORT modules between 0 and 1 and between 4 and 5 have been activated, this group of processors is capable of completely independent action without interaction with other modules on the Control Group level 1. Assuming that the appropriate control modules in the SYNC/CARRY loop have been activated as shown, modules "3,XX" could be considered as an arithmetic functional unit of 3·n precision where n is the word size of a given module. Module "2,2" would be the controller for this arithmetic section.

As another example, consider a parallel array processor. This configuration, using an arithmetic capability of 2·n bits would appear as in Fig. 6. Again each level in the hierarchy is controlled on a different level control group. Module "1,1" is the system controller and actually contains the program to be executed. Each of the modules "3,1" through "M+2,2" contains the appropriate data elements as in any parallel array processor. Module "1,1" would control each of the functional groups A, B, ... by placing a command with the appropriate destination name on the Master Control Group CMD bus for the specific controlling module desired. Processor "1,1" can control all the functional groups simultaneously with one command addressed to "2,XX." Thus, as in the case of a parallel array processor, a single ADD, MULTIPLY, etc., command could cause all M functional groups to perform the required operation on the appropriate operands in each of the independent memories.

In the case that restructuring is required (due to problem changes or hardware failures), the master need only cause the system to pause while it proceeds through the structuring phase again, etc. It is assumed that the master can interrupt any processor by commands on C.G.0 which can never be blocked.

Although the preceding discussion and examples have only two C.G.'s and result in three levels of hierarchy, there could be several more C.G.'s. This would allow several more levels of hierarchy and, at each level, each processor would appear as a master to all those processors subordinate to it.

The following points should be noted:

1. All data transfers take place on the DATA bus. Therefore, this bus will be a bottleneck and its performance will seriously affect the total system throughput. The DATA bus must therefore be a very high speed bus.

2. In order that a group of m processors be connected to form an m·n bit arithmetic section, they must be adjacent or broken only by single modules operating on a different hierarchy level.

3. Although the master controller usually would communicate only with the modules one level below it in the hierarchy, it can send commands to any module through the master control group. It, therefore, can begin corrective action by reassigning names, etc., should a fault occur.

OBSERVATIONS AND CONCLUSIONS

The utility of the system proposed here depends upon the amount of hardware and software overhead required and the latency in the interprocessor communications. Based on the work of Hayes and Sherman, it can be shown that, on the average, in a light to moderately loaded system, the expected delay to place an information packet on a C-bus, and consequently the total message communication rate, is well within the practical limits for useful systems. Assuming a number of processors communicating with all other processors symetrically on a bus with a 1.5 x 10^6 word/sec rate with each processor transmitting at a rate of 50 x 10^3 word/sec, Hayes and Sherman show that each processor can expect a delay of less than 0.7 μs.[7] On the other had, Avi-Itzhak[1] has shown that, in the heavily loaded case, a deadlock situation can occur where competing groups of processors "see-saw" control of the bus, locking out all other processors. Therefore, it is important that the meaning of "heavy," "moderate," and "light" loading be determined quantitatively. Worst-case figures must be computed and the potential for deadlock eliminated.

As is evident from the examples, only a limited number of Control Groups are likely to be used. It will be necessary, however, to determine exactly how many levels of hierarchy and hence of Control Groups are required to provide a generally useful organization meeting the criteria mentioned earlier.

Since the system is to be constructed in a modular fashion with a uniform communications interface between modules, the direct physical configuration of a system to serve as a small real-time controller or other fixed-task system should be simple and straightforward. Following a "divide and conquer" philosophy, each module would be given a single fixed task and would be responsible for or to a small constant set of other modules thus reducing the problems inherent in handling multiple tasks or interrupts in real-time.

A point of major concern is the cost of the software required to support a system of the type proposed here and the difficulty of preparing user programs. While more research is necessary in this area before conclusions can be drawn, the complexity of the support software is reduced by the simplicity of the C-bus concept and by the intercommunications protocol which is largely hardware controlled.

An interesting point is that the proposed architecture can be configured for the execution of data-flow programs.[4,5] The difficulty of preparing data-flow programs is no more difficult than preparing programs for conventional machines since it is not necessary to explicitly detect parallelism. To execute data-flow programs, system processors, or perhaps processor teams, would be assigned as operators in the data-flow program. Each processor would be directed to distribute copies of its computational results to destinations indicated by the links of the program. The flow of data tokens is represented by the flow of operands on the DATA bus. The flow of control tokens in the form of packets transmitted on the control busses forces data-flow programs to enforce the firing rules.

In conclusion, an organization of microprocessors intercommunicating over a series of busses and having a restructurable, hierarchial control philosophy has been presented. Although the development of this architecture is by no means complete, it is hoped that the problems indicated and will yield a flexible multiprocessor architecture that allows restructuring of system resources to tailor them to processing requirements.

REFERENCES

1. Avi-Itzhak, B., "Some Heavy Traffic Characteristics of a Circular Data Network," Bell System Technical Journal, Vol. 50, No. 8, pp. 2521-2549, Oct. 1971.

2. Chen, T. C., "Distributed Intelligence for User Oriented Computing," AFIPS Conference Proceedings, Vol. 41, Part II, pp. 1049, 1972.

3. Chen, T. C., "Parallelism, Pipelining and Computer Efficiency," Computer Design, Vol. 10, pp. 69-74, 1971.

4. Dennis, J. B., "First Version of a Data-Flow Procedure Language," Symposium on Programming, Institut de Programmation, University of Paris, Paris France, pp. 241-271, April 1974.

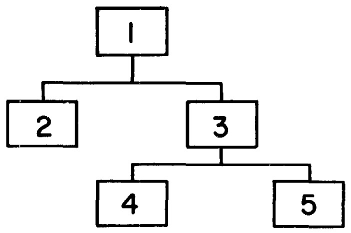
5. Dennis, J. B. and Misunas, D. P., "Preliminary Architecture for a Basic Data-Flow Processor," 2nd Annual Symposium on Computer Architecture, pp. 126-132, Jan. 1975.

6. Graham, R. L. and Pollak, H. O., "On the Addressing Problem for Loop Switching," Bell System Technical Journal, Vol. 50, No. 8, pp. 2495-2519, Oct. 1971.

7. Hayes, J. F. and Sherman, D. N., "Traffic Analysis of a Ring Switched Data Transmission System," Bell System Technical Journal, pp. 2947-2978, Nov. 1971.

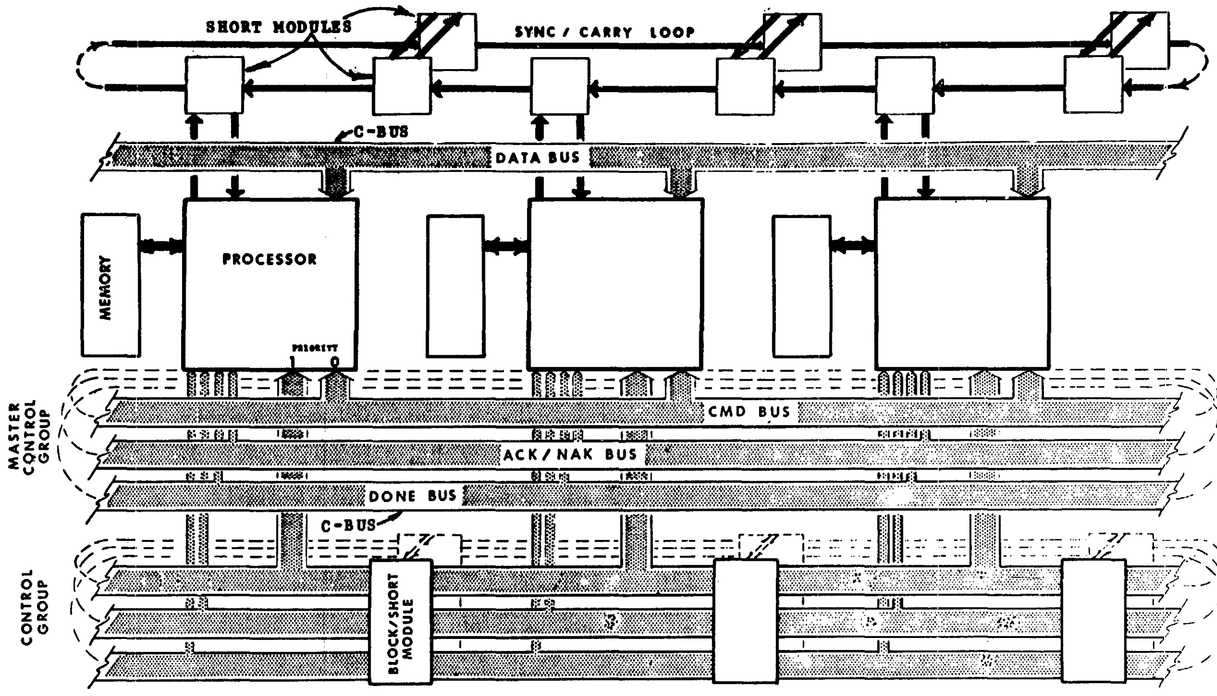
8. Pierce, J. R., "How Far Can Data Loops Go?" IEEE Trans on Communications, Vol. Com-20, pp. 527-530, 1972.

9. Rattner, et al., "Bipolar LSI Computing Elements Usher in New Era of Digital Design," Electronics, Vol. 47, No. 18, pp. 89-96, Sept. 1974



1 DIRECTLY CONTROLS 2,3
3 DIRECTLY CONTROLS 4,5

Fig. 1. Example of a Hierarchical Structure



NOTE: ONLY TWO (THE MINIMUM) CONTROL GROUPS HAVE BEEN SHOWN SEVERAL MORE WOULD BE DESIRED

Fig. 2. Hardware Organization.

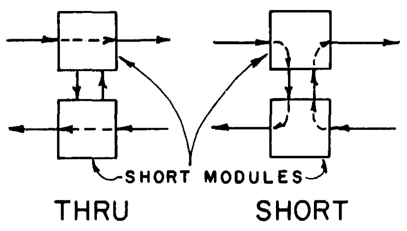


Fig. 3. SHORT Modules.

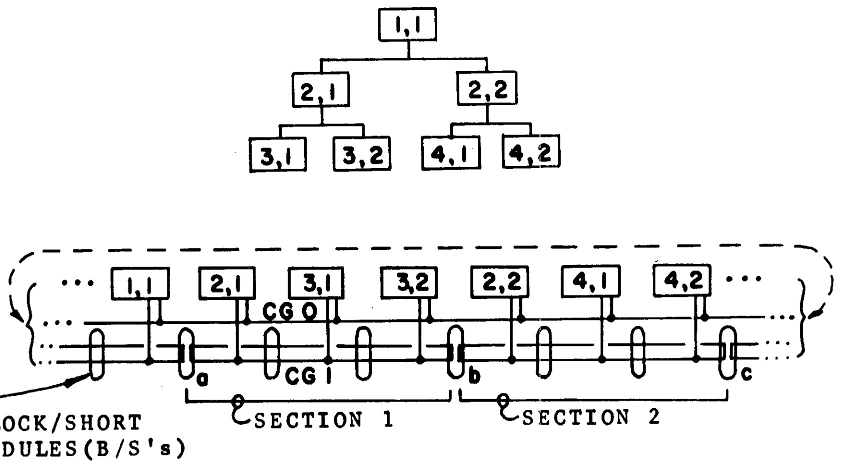


Fig. 4. A Hierarchical Organization.

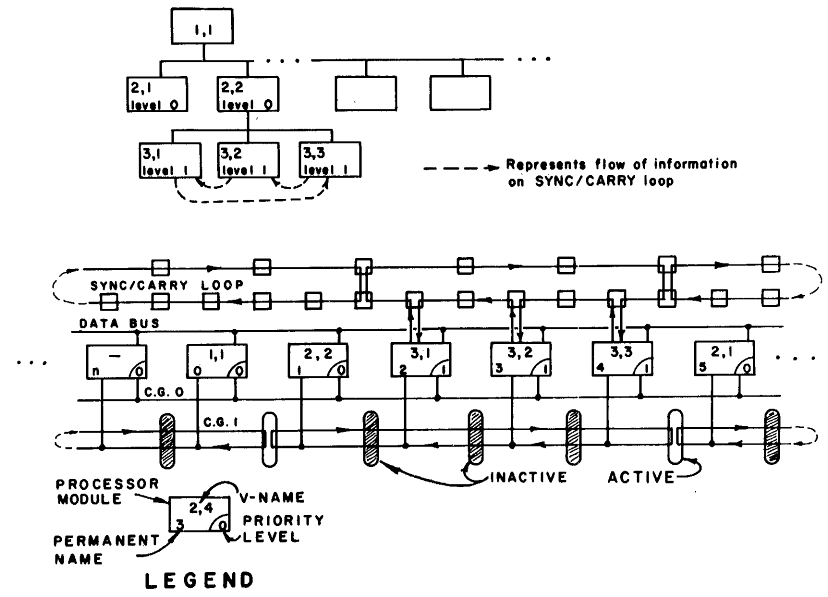


Fig. 5. A Hierarchical Organization Employing a Multiple Precision Arithmetic Unit.

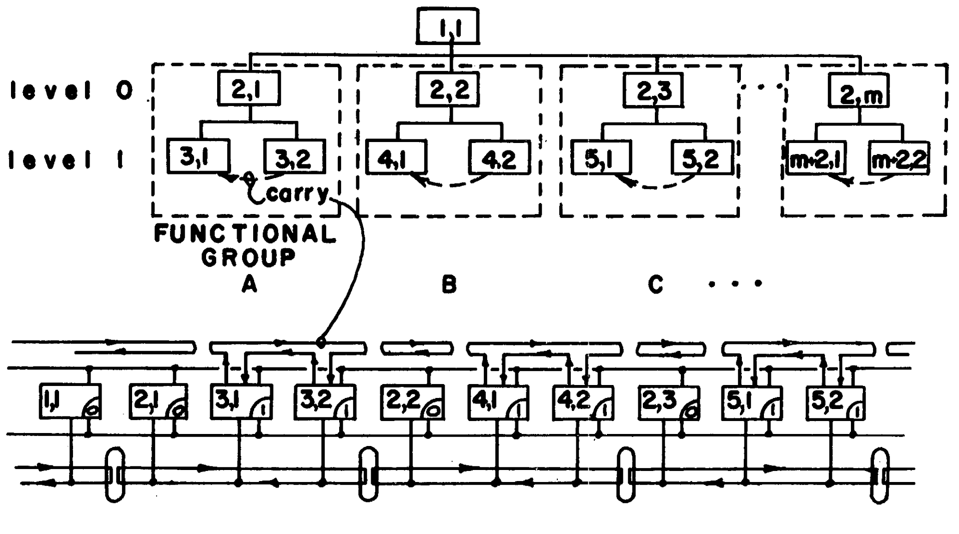


Fig. 6. An Array Processor.

A MULTIMICROPROCESSOR APPROACH TO NUMERICAL ANALYSIS: AN APPLICATION TO GAMING PROBLEMS

Robert McGill
and
John Steinhoff

Research Department
Grumman Aerospace Corporation
Bethpage, New York 11714

Abstract

A parallel processing system is described that consists of a minicomputer host and a set of bipolar microcomputer modules. It is argued that such a system in which the microcomputers operate with little mutual interaction should be effective for an important class of problems in numerical analysis. In particular, estimates are given for the operation of the system on a problem in gaming theory. In this problem, the extensive I/O and software capabilities of the minicomputer provide ease of use for a large part of the problem. The relatively simple part of the problem, which requires almost all of the computational time, is executed in parallel on the microcomputers. It is argued that the system, with 10 to 20 modules, would offer one to two orders of magnitude more speed at several orders of magnitude less cost than current large general-purpose machines. The potential for the development of new algorithms that exploit fully the characteristics of the new devices is discussed.

1. Introduction

The recent introduction of low cost bipolar microprocessors has made it possible to build very fast microcomputers for control and simple numerical processing. In this paper we consider the use of these microprocessors together with low-cost bipolar random access memories (RAM) for solving a certain type of large scientific numerical analysis problem. These devices are incorporated in simple, inexpensive computing modules that can perform a (limited) class of computations as fast as a large, general-purpose machine. A system comprising one or two dozen of these modules connected to a host minicomputer is proposed. This type of system should be very efficient for solving certain problems requiring a large number of simple computations, each with limited accuracy. Such problems may be wasteful of the resources of large, general purpose machines, and our approach should provide an effective alternative.

A study is made of a gaming problem whose solution requires the generation of a very large number of trajectories, each of which has to be tested for decision and win/lose conditions at many time points. The algorithm is similar to Monte Carlo techniques, for which the use of simple, special purpose machines has been considered[1].

The program divides naturally into a game simulation routine and an executive routine that initializes each game and makes decisions based on previous games. Although almost all of the computations are done by the game simulation routine, it is not very complicated and programming each module (which is microcoded) for this function is not too difficult. The rest of the program, which requires much of the software effort, is programmed in a high level language on the minicomputer, which also provides the I/O function. The gaming algorithm considered is easily adapted to the above scheme, and, since the entire game calculation is performed in each module, the only communication requirements are between the host and the modules. Also, the total time spent transferring data is small, and, for the number of processors considered, there should be little communications conflict.

The particular problem studied originated from an aerial combat analysis program being carried out at Grumman[2]. An estimate is made of the time and cost required to solve this problem on the proposed system and compared with estimates for an IBM 370/168. In addition to the gaming problem, some other problems for which the system should be effective are briefly considered.

The main point of this paper is to show that with currently available bipolar devices a system can be economically developed to solve certain numerical analysis problems that might be too costly or otherwise impractical to solve on large general purpose machines. Even without an automatic operating system, the time required to configure and microcode the modules should be considerably less than the analysis that went into the original problem formulation. This effort might lead to a numerical solution that otherwise could not be obtained.

2. The Multiprocessor System

The system as configured to solve the gaming problem is depicted in Fig. 1. A Data General minicomputer is used as host to the modules. Since the microcomputers should cost less than $2000 each, the entire system including one or two dozen modules represents about twice the cost of the original minicomputer.

For the gaming problem, as will be explained in Section 4, it is only necessary to transfer data between each module and the host. Also, each module executes the same program independently of the others, but with different data. Except for program loading, the modules and the host each operate under their own control. During operation, the decision to transfer data to or read data from the memory of the host is made by the modules, based on computed conditions. The data is then transferred via the direct memory access (DMA) channel of the minicomputer. For the number of modules considered and for the present problem, the amount of time each module spends transferring data will be small compared to the total computing time. Thus, access conflicts to the DMA channel should be infrequent. In this case it is only necessary to use a priority encoder and a decoder to resolve conflicts, as described, for example, in detail in Ref. 3. This system selects a device and then locks out all others until the transfer is completed. Since the modules execute identical programs the performance should be independent of the particular type of priority scheme (fixed, circulating, wait time dependent, etc.) and the fixed scheme, which is the simplest, is used. Each module enters a wait mode from the time it requests data until this data is received.

If the amount of data to be transferred were larger or if more modules were used, this simple scheme would be inefficient and a more sophisticated system would be necessary. The entire control and data flow scheme, however, is quite problem dependent and it seems best at this time to configure the system for the problem, rather than develop a general purpose scheme - especially since the cost of implementing these specific schemes is not too high. Some interesting possibilities for controlling complex systems of modules, based on graph models of computation, are given in Ref. 4.

Some other multiprocessor systems which use a Multiple Instruction, Multiple Data stream (MIMD) approach similar to ours are:

- Carnegie Mellon's C.mmp[5] - A set of 16 minicomputers communicating with memory modules through a crosspoint switch. Our processor modules are much cheaper and faster than these minicomputers, but do not have the flexibility and software support. Also, in our system, each module only has access to a single large memory in the host and its own small local memory

- The computer module set described by Fuller, Siewiorek and Swan[6] - These modules, like ours, are intended for special purpose application and also have individual local memories. There is no general purpose host, however, to coordinate activity and store infrequently used code. Instead, this code is distributed among the local memories, accessible to each module through a set of intermodule buses which would only directly connect nearby modules

- The multimicroprocessor system described by Senger[7] is similar to ours, but the microprocessors do not have individual control units. Instead, the processors are treated as resources and, when available, can be acquired by a controller through a hardware scheduler. There is a control unit for each instruction stream, but the number of processors need not equal the number of instruction streams.

The following are examples of modular systems that are configured specifically for a particular problem. Some idea as to the advantages of this approach, for some problems, over using a general purpose machine can be gotten from these references.

- A set of microprogrammable modules, described by Cooper[8] - Reflects the basic idea that complex arithmetic and logical functions can be implemented by a set of identical modules connected together specifically for the task. These modules are general building blocks and not constrained to operate in a particular configuration

- The CDC Modular Change Detection System[9] - This system, much larger than ours, consists of a set of 40 similar modules connected to each other and to a general purpose computer. Performing 320 million instructions per second, it implements a special algorithm to process image data at a cost reduction of three orders of magnitude, compared to standard, general purpose machines.

3. The Computing Module

The basic module, currently being constructed, consists of an Intel 3000 series bipolar microprocessor set, bipolar RAM and Schottky TTL MSI and SSI devices, assembled on a wire-wrap board. The module is arranged in four main sections (see Fig. 2):

- 16 bit Arithmetic Logic Unit (ALU)
- 256X16 bit Data Memory (DM)
- Microprogram Control Unit (MCU)
- 512X28 bit Microprogram Instruction Memory (MIM).

The full cycle time of the module is 145 ns. while the memory access time is 40 ns., so that the design possibilities are somewhat different from standard processors whose cycle time is memory limited. The operation of these units for one cycle is as follows:

- The ALU can decode an instruction and execute an arithmetic or logical operation
- The MCU can compute a new address for the MIM
- The MIM can output a microprogram instruction for the next cycle
- The DM can write a data word computed in the previous cycle and read a new word for the next cycle
- The MIM can be independently addressed by another module for a second read operation.

3.1 Microprogram Instruction Set

The MIM word has 4 fields:

- Jump - 7 bits used by MCU together with internal flags and carry from ALU to generate next MIM address
- Flag Control - 4 bits used by MCU to set internal flags and generate carry input for ALU
- ALU Control - 8 bits decoded by ALU for operation on data in one of 11 internal registers or on the DM bus
- DM Addressing - 9 bits used to form DM address, for write control and for loading of the Memory Address Register (MAR).

3.2 DM Addressing

There are 3 addressing modes:

- The (8 bit) address is taken from the MAR
- Considering the memory as a 16X16 word matrix, 4 bits from the MAR specify the column and 4 bits from the MIM word specify the row. This mode allows any one of 16 words in a column to be addressed directly by each microinstruction once the MAR is set up
- 4 bits from the MIM word specify one of 16 words in the first column, creating effectively 16 additional directly addressable registers.

Once addressed, a data word can be read from the DM in one cycle, operated on by the ALU in the next, and written back into the same location in the following cycle. This sequence of operations requires one microinstruction and a set of these operations can be executed at the rate of one per cycle. This capability requires little extra hardware (2 address registers) and a fast memory, and turns out to be very useful for the algorithms that we use, where variables are "updated" at each of many time points (see section 4).

The use of fast bipolar RAM, in addition to saving cycles normally required for some store operations, also results in shorter basic cycle times. Also, since the modules execute the same program the MIM can be shared (this sharing is completely automatic - the modules remain functionally independent). Thus, the cost of the bipolar RAM, which would otherwise be a dominant factor, is reduced to about that of the microprocessor chip set.

At present, we do not have a hardware multiply capability. Although a device could be made fairly cheaply using standard Schottky TTL parts which would perform a 16 bit multiply in several of our basic cycles, the trade-offs involved in including it are different here then in conventional computers: The cost of our modules is low and they are to be replicated as many times as is feasible so that the total cost of the multiply capability might become a significant fraction of the entire system. Also, the high data throughput of our module can make it possible to compute certain functions very rapidly without performing any multiplications (except for multiple shifts). It will be seen that the requirements of our particular gaming problem can be satisfied in this way. Future problems, however, will most likely require that at least some of the modules have a hardware multiply capability and this can be included at that time.

Some recently developed processors which are similar to ours are described in Refs. 10, 11 and 12. These machines are intended for dedicated signal and image processing, but have the following features in common with ours:

- Separate data and instruction memories
- Overlapped instruction fetch, execute and data fetch
- Microprogrammability
- Reliance on a general purpose computer (host) for mass memory and I/O
- Cycle time in the range of 50-200 ns. (the 50 ns. machines use ECL technology)
- Dedicated operation.

In addition, these processors have some features that ours does not, such as multiple, more sophisticated arithmetic units. The more general numerical analysis problems that we plan to attack result in algorithms having more branching possibilities and less regularity than signal and image processing algorithms and do not seem to be able to make efficient use of such micrcparallel structures. Instead, we approach our problems with a macroparallel structure consisting of a set of independent, simple modules.

4. The Gaming Problem

The gaming problem studied here is based on an algorithm being developed at Grumman for evaluating the effectiveness of an aircraft and its weapons systems in aerial combat[2]. A dogfight between two aircraft is simulated where the controls of each plane are adjusted according to the observed position of the opponent. To provide a simple but realistic simulation, information available to each pilot is assumed to be limited, and discrete visually discernable regions are defined so that controls can only be changed when an opponent moves into a new region.

The algorithm consists of two phases: In the first a sequence of runs are made (by the modules), each consisting of a simulated dogfight where the controls are chosen for each region according to weighted random variables. Each run continues until win, lose, or draw conditions are met. Based on the outcomes, the values used to weight the control choices are adjusted (by the host) and new runs are made. When the weighting values converge, a control strategy is available for the next phase. The second, or statistics, phase consists of a large number of the same aerial combat simulations, but the players use the best strategy (most heavily weighted control choices) developed in the first phase. The number of wins and losses for a sequence of initial conditions in the second phase determine the aerial combat effectiveness of the vehicles.

The program is meant to be used as a tool to determine the effectiveness of various changes in the characteristics of an aircraft in aerial combat situations. Thus, the program must be used many times with different values of parameters such as maximum velocity or turning radius. For this reason the computational time for an evaluation cannot be too long.

4.1 Implementation

The host is programmed in a higher level language for the executive function, which includes maintaining a stack of initial conditions, storing and updating the tables of weighting values and keeping track of the statistics and the convergence of the tables. Also, the memories of the modules are loaded from the host when it initiates the program.

Each module is programmed to independently compute an entire game. At the start of each game the module requests a set of initial conditions and initial controls from the host. It then computes and integrates the velocity profiles of the two vehicles. At each time point it tests the distance between the vehicles as well as relative position and heading angles to determine whether a new visual region has been entered, or a win, lose or draw achieved. If a new region has been entered, the relevant control variables are requested from the host and the game is resumed. If termination conditions are met, the outcome of the game as well as the regions entered and the corresponding control choices (in the "learning" phase) are transmitted to the host and a new game is started. The modules always use the DMA channel of the host when requesting or writing data, and compute the relevant memory locations.

4.2 Computational Requirements

Although the executive routines in the host, together with the weighting tables account for most of the memory requirements, almost all of the computations are done by the modules in generating and testing the trajectories. Therefore, we tried to use algorithms that would do these calculations in the least time.

The control choices for the vehicles, as well as the trajectory equations are quite simple and the accuracy requirements are minimal. The problem is that a very large number of games have to be played, in each of which these trajectories have to be computed at many time points.

We chose a scheme similar to a (sequential) digital differential analyzer for these computations: Complicated functions are written in terms of differential equations, which are then integrated. Also, products are written in differential form and, when one of the variables changes by a specified amount (a power of 2-signaled by the carry from an accumulator) the product is incremented by the other variable, shifted by the appropriate amount. Implicit equations can be solved by varying the unknown so that the equation is satisfied to within certain bounds.

An example of the technique used is the transformation from Cartesian coordinates (x, y) to polar coordinates (R, θ), where (x, y) are the relative coordinates of the vehicles and (R, θ) are the relative separation and angle, and

$x = R \cos \theta$

$y = R \sin \theta$.

At the start of the game the exact R and θ are computed by the host (this represents a negligible overhead). As the trajectories are computed x and y are changed. The algorithm is then used to compute new polar coordinates (R_c, θ_c), which are no longer exact but are close to the correct transformations. These approximate coordinates satisfy (to first order) the requirement that

$x_c = R_c \cos \theta_c$

and

$y_c = R_c \sin \theta_c$

be close to x and y. Each algorithm cycle R_c and θ_c are changed by a certain amount, a new x_c and y_c found and the new errors $|x-x_c|$, $|y-y_c|$ computed. The process continues until the errors are less than a specified amount. The property that makes this technique very fast is that by choosing certain increments for R_c and θ_c new values of x_c and y_c can be computed with only a shift and an add (see Fig. 4). The change in R_c, which we call a stretch is chosen to be $a_R 2^{-n} R_c$, and the new R_c is

$R_c' = R_c + a_R 2^{-n} R_c$.

The corresponding new values of x_c and y_c are

$x_c' = x_c + a_R 2^{-n} x_c$

$y_c' = y_c + a_R 2^{-n} y_c$,

where $a_R = 0, +1$ or -1 (see Fig. 3).

For the rotation,

$\theta_c' = \theta_c + a_\theta 2^{-m}$

and

$x_c' = x_c - a_\theta 2^{-m} y_c$

$y_c' = y_c + a_\theta 2^{-m} x_c$,

where $a_\theta = 0, +1$ or -1 (see Fig. 4).

For our algorithm, n and m are kept fixed and a_R and a_θ chosen to keep $|x-x_c|$ and $|y-y_c|$ within certain bounds. If n and m are chosen so that the changes in x_c and y_c for the above operations are not too different from the changes in x and y from time point to time point along the trajectory, only a few iterations will be needed at each step.

This technique is similar to that used in a fast curve generation algorithm[13] and in the Cordic algorithm[12,14]. These schemes can be made quite accurate, but in our case this does not seem to be necessary since the process we are modeling (a human pilot) has only crude perception of the relevant quantities.

4.3 Performance Predictions

At present, only a two dimensional case is being run (constant altitude) on a general purpose machine, and the control choices are restricted to left/right turn or straight flight, and accelerate/decelerate or steady speed. This initial study is being used to determine the feasibility of a full three dimensional analysis. The two dimensional program, coded in Fortran IV, takes about three hours on an IBM 370/168, and it is estimated that the three dimensional case would take between 100 and 1000 hours, depending on the amount of operator intervention.

About 80,000 games (both phases) are required for the two dimensional case. In each game there are about 2000 integration steps, and about 40 region changes where a new control choice must be made and stored. The large number of games required is due to the fact that over 300 regions were defined for each player with up to 6 control choices in each.

To determine the total computational time for our system to run the two dimensional case, we need to know the number of module cycles necessary to perform the trajectory computations at each integration step and the time required to transfer data to and from the host at each region boundary. From the micro-code already developed for the module and from the data channel characteristics of the minicomputer, we estimate that there are about 100 cycles required and 20 μs. data transfer time. Also, we have to take into account the time that the host spends updating the tables and computing initial conditions (about 20 μs. per region). Even though this time is insignificant when the problem is done sequentially it may be important here.

Based on the above numbers, and the 145 ns. basic cycle time we have:

- Module computing time per game;

  $t_M \approx 2000 \times 100 \times 145$ ns. = 29 ms.

- Data transfer time per game;

  $t_D \approx 40 \times 20$ μs. = .8 ms.

- Host computing time per game;

  $t_H \approx 40 \times 20$ μs. = .8 ms.

When the data channel to the host is being used, neither the module communicating to the host nor the host can do any computation. Thus, the time required per game for each module is

$t_M + t_D \approx 30$ ms.

and for the host

$t_H + t_D \approx 1.6$ ms.

If the number of modules N is much less than t_M/t_D there will be a negligible amount of communication conflict between modules and the total computing time will be

$T \approx 80{,}000 \times \max\,(t_H + t_D,\ (t_M + t_D)/N)$.

Using this approximation for $N \lesssim 10$, since $t_M/t_D \approx 36$, we have

$T \approx \max\,(2.1,\ 40/N)$ minutes.

Since the three dimensional case was estimated to take 30 to 300 times longer, the time required for our system with 10 modules to perform the full three dimensional analysis should be a manageable 2 to 20 hours.

5. Extensions of the System

There are two basic extensions that we can make to the system.

The first only involves software changes and perhaps minor modifications to the modules. The basic data transfer structure, however, would be kept fixed. This system should be applicable to algorithms similar to the one studied, such as Monte Carlo techniques[1] and (systematic) multidimensional global searches. As in the gaming problem, a large sequence of computations must be made, each computation depending only weakly on the others. Also, although requiring a large amount of computer time, these problems frequently involve only simple codes and modest storage and accuracy requirements and can be wasteful of the resources of a large general purpose machine.

A more involved extension of the system would be to include inter-module communication. This modification would most likely be necessary for the efficient solution of partial differential equations. However, if a certain class of very time consuming problems were chosen and the system configured specifically for them, this modification should not be too difficult. Certainly, the change would be very simple compared to developing a general inter-module communication scheme.

6. Conclusion

A multiprocessor system that solves a gaming problem has been discussed. The problem is one of a general class that requires the playing of large numbers of "games". In each of these the trajectories of two or more vehicles are computed and tested for decision and win/lose conditions at many time points.

The system uses a minicomputer (with extensive software and I/O capability) as an executive (host) and multiple microcomputers to implement the basic game computations. This system requires only one type of interconnection: host to microcomputer module. After microcoding the module part, via the host, existing software can be used for the complex executive part (where speed of executive is not critical, but ease of use and flexibility are). With this architecture, the computational speed should increase almost directly with the number of modules, up to 10 to 20 for our particular problem.

This approach, although relatively simple, only became practicable in the past year with the advent of the bipolar microprocessor. This device forms the basis of a complete microcomputer module with small memory but high computation speed - of the order of the IBM 370/168 (145 ns. cycle time) at a total cost of under $2000 (not including development costs). Thus, 10 to 20 of these modules and a host mini can now be assembled for about double the price of the minicomputer system ($80,000).

The particular gaming problem that we study arises in the evaluation of various systems in aerial combat maneuvers. Feasibility studies indicate that a realistic three dimensional problem would require several hundred hours on an IBM 370/168 to solve. Also, for a meaningful study, several solutions would have had to be obtained as various parameters are changed. Thus, the usefulness of the algorithm in determining the effectiveness of various systems in aerial combat requires that both the time and the cost of obtaining solutions be reduced by one to two orders of magnitude. Preliminary studies, based on a Data General minicomputer and a system of 10 modules, indicate that the solution time can be reduced by a factor of 50 to a manageable 2-20 hours.

Equally important in its affect on the performance of our system is the length of time it can be economically and practically run on the problem. This system could easily make a 10-hour run, with less expense and probably less turn-around time than a 1-hour run on a large machine. Thus, it seems reasonable to claim for the problem considered a factor of about 500 increase in efficiency for our system compared to an IBM 370/168.

In addition to the gaming case, our system should be directly applicable to other areas, such as global searches or global optimization. These problems, like ours, frequently do not have large memory or accuracy requirements and can be very wasteful of the capabilities of large computers. They can, however, require enormous amounts of computation.

Our main point has been to show that with currently available bipolar devices a system can be economically developed to solve certain numerical analysis problems that might be too costly or impractical to be solved on large general purpose machines. Even without an operating system, the time required to configure and microcode the modules should be considerably less than the analysis that went into the original problem formulation, and the additional effort might lead to a numerical solution that otherwise could not be obtained.

Acknowledgement

Many of the features of the microcomputer were suggested by Martin Kesselman, who is currently designing and building the first module.

References

1. Yu. A. Schreider, The Monte Carlo Method, Pergamom Press, New York, 1966, esp. Ch. 1.8. Also, E. Sadeh and M. Franklin, "Monte Carlo Solution of Partial Differential Equations by Special Purpose Computer," IEEE Trans. Comp., C-23, p. 389 (1974).
2. M. Falco, G. Carpenter and A. Kaercher, "The Analysis of Tactics and System Capability in Aerial Dogfight Game Models," Grumman Research Department Report No. RE-474, (1974).
3. D.P. Siewiorek, "Process Coordination in Multimicroprocessor Systems," Microarchitecture of Computer Systems, EUROMICRO, (1975), p. 1.
4. S.S. Patil, "Micro-Control for Parallel Asynchronous Computers," Microarchitecture of Computer Systems, EUROMICRO (1975), p. 17.
5. W.A. Wulf and C.G. Bell, "C.mmp - A Multi-Mini-Processor," AFIPS Conference Proceedings, Vol. 41 (FJCC 1972), p. 765.
6. S.H. Fuller, D.P. Siewiorek and R.J. Swan, "Computer Modules: An Architecture for Large Digital Modules," Proc. Symp. on Computer Architecture, (1973), p. 231.
7. D. Senger, "A Multiple Instruction Stream Processor," Microarchitecture of Computer Systems, EUROMICRO (1975), p. 71.
8. R.G. Cooper, "Micromodules: Microprogrammable Building Blocks for Hardware Development," Proc. Symp. on Computer Architecture (1973), p. 221.
9. P.J. Klass, "Analyzer Pinpoints Radar Changes," Aviation Week and Space Technology (May 26, 1975).
10. J. Allen, "Computer Architecture for Signal Processing," IEEE Proceedings (April 1975), p. 624.
11. G. Zimmermann, "SPDM - A Subprocessor With Dynamic Microprogramming," Microarchitecture of Computer Systems, EUROMICRO (1975), p. 149.
12. J. Staudhammer, "A Fast Display-Oriented Processor," Proc. Symp. on Computer Architecture, (1974), p. 17.
13. P.E. Danielsson, "Incremental Curve Generation," IEEE Trans. Comp. C-19, (1970), p. 783.
14. J.S. Walther, "A Unified Algorithm for Elementary Functions," Spring Joint Computer Conference Vol. 38 (1971), p. 379.

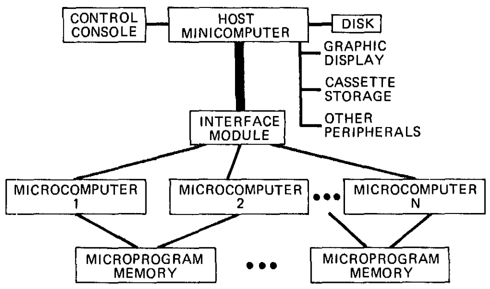


Figure 1: Multiprocessor System

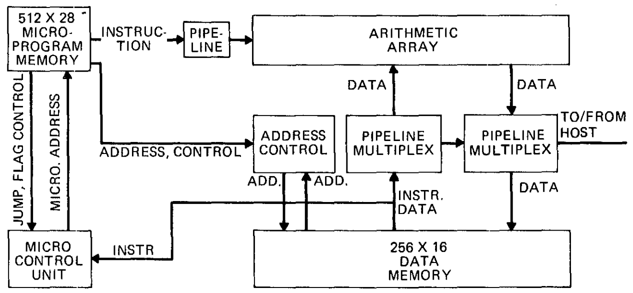


Figure 2: Microcomputer Module

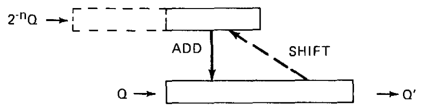


Figure 3: Stretch Operation; $Q = x_c, y_c, R_c$

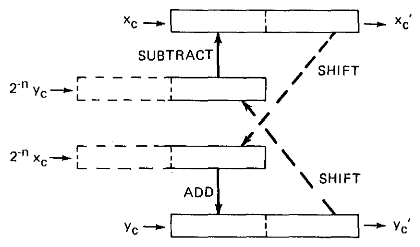


Figure 4: Rotate Operation

A MODEL OF INTERFERENCE IN A SHARED RESOURCE MULTIPROCESSOR*

John E. Jensen and Jean-Loup Baer
Department of Computer Science
University of Washington
Seattle, Washington

Abstract

This paper presents a generalized model of tightly-coupled multiprocessor systems which is then simplified to form a stochastic model for the study of interference. Analysis is performed on the resource contention which is characteristic of such systems in order to find a measure of system performance. After reviewing the problem of memory interference, the analysis is extended to contention in other individual resources, then combined to form a model for the interacting effects of contention in systems where processors contend for several shared resources.

1. Introduction

Recent design proposals and realizations [4,10,11] have included multiprocessors in attempts to meet the expanding demand for high-performance systems. A solution to the need for improved efficiency lies in the distribution, duplication and sharing of hardware resources. Unfortunately this leads to situations in which a given unit may receive several simultaneous requests for service (e.g. a memory module). The result is degraded performance, or interference, measured by comparing actual machine performance to the ideal case for which there is no contention. This paper presents a generalized model of tightly-coupled multiprocessors with highly shared computing resources. Analysis is then performed on the resource contention in order to find a measure of system performance.

The best known contention problem is when processors and IO controllers interfere in their access to main storage. Analytic models with exact solutions exist for two processor systems [8] via Markov chain methods, but the general case becomes too complex, precluding a precise solution. For a solution in closed-form one has to introduce simplifying assumptions in order to prevent the analysis from becoming unwieldy. A series of models have been introduced in which a prototype instruction is assumed and its execution rate (IER) analyzed for a variety of multiprocessor types [9]. Closed-form solutions are obtained for IER in terms of parameters which relate typical design characteristics of the memories and processors. In addition, cache memories may be introduced to the processor-memory interface [3]. In this paper, we extend previous formulas [9] to include cache memories, and then propose a more general one for systems in which processors contend for several resource classes as well as primary memory.

2. The Machine Model

2.1 A General Shared Resource Multiprocessor

Figure 1 shows a general model of a shared resource multiprocessor (SRM) in PMS notation [2]. The example was chosen purely for ease in description and conservation of space, with the design of more specific configurations being one of the objectives of the model. 8 central processors P.c share 16 modules of primary memory M.p through a central processor-memory switch S.mp. Each P.c possesses some local memory M.c and a set of mapping registers D.map which define its access to main memory.

The P.c's have no arithmetic power, performing only load, store and branch instructions. Other instructions are memely fetched and decoded, while operands are sent to a shared set of pipelined execution units D.e via a common request bus L.req. The P.c's are arranged into 2 time-shared rings by S.ring, which creates a maximum overlap of computing with a minimum bandwidth required in the request bus [4]. Input-output is initiated to IO controllers in the same way as a request for a D.e. When an "IO" instruction is executed, a request is sent to an appropriate K.io and the P.c is allowed to continue. IO-completion interrupts cause the appropriate P.c to be interrupted [5].

A special controller K.sched is provided for assigning new tasks to P.c's, with two options for flexibility in the scheduling mechanism. In the "floating control" scheme [5,7] P.c's perform their own scheduling under the control of K.sched. Under "fixed control", K.sched serves each request by returning the entry point of the new task in memory, while a dedicated processor P.sched (with associated M.a for the scheduling tables) constantly supplies K.sched with the next task to be assigned for execution.

2.2 Examples of the Model

The generality and versitility of the model may be illustrated by examining some current designs. C.mmp [11] is a set of 16 asynchronously executing PDP-11's (each with local memory) which access main memory through D.map's and S.mp. The ring structure and D.e's are missing since each P.c has its own complete processing capability. C.mmp's IO system is similar to its memory system in that the P.c's are connected to busses supporting the IO controllers by the S.kp switch. Scheduling is handled by the operating system without any additional hardware.

Figure 2 is a conception of Texas Instruments' ASC [10]. A single P.c feeds instructions to 4 high-speed pipelined D.e's which consume streams of vector operands under the control of registers found in M.c. The most interesting feature is the "peripheral processor" which performs the control and data-management functions for the ASC, and is actually a ring of "virtual processors" (P.v).

Figure 3 emphasizes the ring structure aspects by modeling Flynn's SRM [4]. It has 4 rings of 8 P.c's, and uses L.req and D.e's as in the model. The P.c's have no D.map or S.mp, but access memory through buffers. Cache memory M.c is associated with each ring. No mention is made of IO, and scheduling is done under program control through a standard fork-join construct.

2.3 The Simplified Machine Model

The model described thus far requires too much detail to be studied at the instruction level, hence we capture some of its generality into a more manageable form in Figure 4. Centrally located is S.mp which provides access by the P.c's and K.io's to the M.p modules. The specialized scheduling processor P.s (with memory M.a) makes all policy decisions regarding the activation of user and operating system tasks as well as allocating the system's resources. IO consists of three subsystems, representing the common IO speeds anticipated.

*This research was supported by NSF grant GJ-41164.

The multiprocessing resources consist of synchronized processor rings (3 in the figure) with a set of independent pipe-lined D.e's which are capable of performing all arithmetic functions (except divides) with the same latency. Each ring consists of skeleton P.c's and corresponding M.c's connected by a processor-ring switch S.p. The purpose of the time-multiplexed switch [4] is to select the P.c that is to be considered "active" during each time-slice of the ring, and to coordinate all communication between the P.c's and the D.e's and the remainder of the system.

The instruction units use an instruction set which is patterned after the SRM [4]. Each of the 8 skeleton P.c's begins its instruction-fetch sequence one minor cycle behind its predecessor on the ring. In one major cycle each P.c will prepare one instruction for execution to take place during the subsequent one. A 60ns minor cycle is assumed [1,10], resulting in a 480ns major cycle which provides ample time (120ns) for finding operands in an implicit cache. In the case where an access to main memory is required ("miss" on the cache), 600ns should be more than sufficient to perform the transfer (120ns plus one major-cycle delay) and still maintain the synchronous timing of the processor ring.

3. The Resource Contention Model

3.1 The Memory Interference Problem

In this section, we introduce an analytic model for general resource contention used to estimate the losses due to interference between processors requesting identical resources. We begin by examining memory interference (the request by more than one P.c for the same M.p module) using expected values for the number and types of instructions executed. The combined effects of the hardware speed and memory conflicts are characterized by a single entity, the instruction execution rate (IER), for which we calculate and estimate.

The P.c's and M.p's are viewed as a stochastic service system in which the M.p's represent m servers, each capable of serving one of k P.c's. Each server handles only those requests directed toward it, serving them in order of arrival and queuing those occurring when it is busy. The M.p's are characterized by a constant service time (access time) followed by an interval of unavailability (rewrite time) before subsequent requests can be serviced. P.c's are characterized by the amount of elapsed time between the completion of service on one memory request and the generation of the next one.

The problem is made more tractable with a few simplifying assumptions. Although processor behavior varies with different instruction types, the probability distribution of instructions, the average frequency of memory requests, and the average time required to execute one instruction can be determined. The access pattern of each processor is assumed to be random, and no distinction is made between read and write requests. We simplify further by considering each instruction to be a series of instances of a "unit instruction" consisting of one memory access followed by a fixed (mean) interval of processor activity.

3.2 An Analytic Model for Memory Contention

In Strecker's formulas for the "unit execution rate" [9], the execution sequence is considered as a Markov process, consisting of a series of "unit instructions", from which we may calculate the rate of memory service. (The principle parameters are defined in Table 1.) The unit instruction begins when an address is received by one of the m modules of M.p at S.mp. Ta is the time required for the memory control to set up the switch and for data to be delivered. Tw is the time required for the module to recover and become ready for the next request. Tp begins for each of the k active P.c's when it receives data from an M.p, extending through the computation until the P.c has a new data address. The "computation" done in this "unit instruction" may be an instruction decode, an (indirect) address computation, or the actual execution of a machine instruction. Several of these unit instructions comprise one complete machine instruction.

The unit execution rate (UER) is the number of unit instructions executed per unit time. In terms of the service times Sm and Sp [9]

$$\text{UER} = m * [1 - (1-Pm/m)^k] \;/\; Sm$$

such that

$$Pm = 1 - (m/k) * (Sp/Sm) * [1 - (1-Pm/m)^k].$$

The analysis is split into three cases (bases upon the relationship between Tp and Tw) which may be combined to form composite equations for the service times as

$$Sp = Tp\theta Tw \text{ and } Sm = Ta+Tw - (Tw\theta Tp) * (1-Pm/m)^k$$

(where $a\theta b = a-b$ if $a>b$, and $a\theta b = 0$ if $a\le b$). The complete equation for the unit execution rate is then

$$\text{UER} = \frac{m * [1 - (1-Pm/m)^k]}{Ta+Tw - (Tw\theta Tp) * (1-Pm/m)^k}$$

where

$$Pm = 1 - \frac{(m/k) * (Tp\theta Tw) * [1 - (1-Pm/m)^k]}{Ta+Tw - (Tw\theta Tp) * (1-Pm/m)^k}.$$

In order to solve the Pm equation, we examine the two cases $Tp\le Tw$ and $Tp>Tw$. In the first case $Pm=1$ and we are done. In the second case the denominator simplifies to $Ta+Tw$, resulting in a k-th order polynomial in Pm. Since the two sides of the equation are monotonic in opposite directions on the interval [0,1], for a given set of parameters we may solve for Pm in this interval and obtain the UER from the first equation above.

We extend this model by associating with each P.c a cache memory with access time Tb and "hit ratio" Pb. The effect of this addition is that with probability Pb, the memory request will be satisfied in the cache (hence no M.p service) while with probability 1-Pb, a normal memory cycle will be required. For the case where $Tp\ge Tw$ it has been shown [3] that

$$Sp = Pb*(Tp+Tb) + (1-Pb)*(Tp-Tw)$$

and

$$Sm = Pb*(\ 0\ ) + (1-Pb)*(Ta+Tw)$$

such that Pm equals

$$1 - \frac{m*[Pb*(Tp+Tb)+(1-Pb)*(Tp-Tw)] * [1-(1-Pm/m)^k]}{k * (1-Pb) * (Ta+Tw)}.$$

This new Pm equation has a single solution in the interval [0,1] as in the previous case. We may generalize this formulation to include the case where $Tp<Tw$ [6], but the memory being considered in this model is relatively fast, so the case $Tp\ge Tw$ is sufficient, yielding

$$\text{UER} = \frac{m * [1 - (1-Pm/m)^k]}{(1-Pb) * (Ta+Tw)}$$

where Pm is determined from the above formula.

3.3 Modeling Multiple-Resource Systems

Previously, a unit instruction was defined in terms of memory access frequency, with all other aspects of the instruction being considered as "processor activity", or Tp. Using the same analysis as above we can determine the effects of multiprocessor contention for other shared resources by extending the notion of a unit instruction to represent one "access" to a resource of <u>any</u> given class (e.g. pipelined D.e's) followed by the average processing time between requests for that resource class. The period of time comprising one unit instruction will, in all cases but for M.p, include several machine instructions. For example requests for floating-point multiplies occur in approximately

13% of the instructions for a scientific mix [6], such that one unit instruction for the multiply resource becomes 1/0.13 times the length of one machine instruction.

When main memory is considered as the sole contendable resource, the IER of a system is computed by first estimating the UER of memory, then dividing by the number of memory references per instruction. The UER of memory is computed using Strecker's approximation which assumes an otherwise constant P.c processing time. A similar set of assumptions will allow the UER to be calculated for the floating-point multiply units (or any other resource), given that some fixed value can be derived for the remaining "processor activity" between requests for the multiply units (cf. section 3.4). The IER can then be calculated by dividing by the frequency of multiply instructions.

In order to model the UER of other resources, the parameters used in the contention model must be generalized. Table 1 defines the set of resource contention parameters a-z which will be used in the remainder of this paper. The correspondence in parameter names for the memory interference example is given in the table and is illustrated here functionally.

$$\text{UER}(k,m,Tp,Ta,Tw,Tb,Pb) = h(k,m,t,a,w,b,p)$$

Table 2 illustrates typical figures for these parameters applied to a variety of harware resources.

With the introduction of pipelined D.e's, the number of stages v in the pipelines becomes of importance. So far we have assumed that all k P.c's actively contend for the system's resources at all times such that UER=h(k,...). In our machine model, however, the P.c's are intentionally arranged into time-phased rings of v P.c's each, so that they only contend with corresponding P.c's from other rings on the same time-slot, increasing the IER of the system. If the system contains k P.c's which are all active, then there are v separate contentions (one per time-slice on the processor ring) among goups of k/v P.c's. In this situation (for a single-resource system) UER=v*h(k/v,...) such that

$$\text{IER} = v * h(k/v,m,t,a,w,b,p) / f.$$

Suppose now that some P.c's are idle such that k is less than the total number of P.c's in the system. The approximation above is optimistic in that it assumes the k active P.c's to be optimally distributed over the v time-slots. In particular, if k<v, it computes the IER to be better than optimal! The invalidating factor is that not all v time-slots necessarily contain active processors. If we assume the k active P.c's to be randomly distributed, then c, the expected number of currently active time-slots, may be determined as was the expected number of busy memories:

$$c = v * [1 - (1-1/v)^k]$$

and hence

$$\text{IER} = c * h(k/c,m,t,a,w,b,p) /f.$$

3.4 The Model for Combined Resources

We have shown how the UER of each resource class may be determined, from which we calculate the performance measure IER=UER/f. In order to combine the analyses of the individual resources, we normalize this measure to the number of processors by the "processor execution rate" PER=IER/k. We also define the "effective execution rate" EER=IER(k)/IER(1) which measures the performance in terms of the number of effective processors, and the "multiprocessor efficiency" EFF=EER/k, which gives a direct measure of the degradation caused by contention in the system.

We now combine the analyses of the individual resources to form a model for the interacting effects of contention. Consider a system of k processors with n resource classes, each characterized by a set of parameters {m,v,a,w,b,p} (e.g. Table 2). We calculate the UER for each resource class i (assuming that we know t_i, the average time between the completion of service and the generation of the next request for resource i) by substituting the appropriate parameters into

$$h_i = h(z_i,m_i,t_i,a_i,w_i,b_i,p_i).$$

Allowing the unknowns z_i and t_i, an equation for L, the expected length of one complete machine instruction, may be obtained from L=1/PER in terms of the UER of the i-th resource:

$$1/L = (h_i/f_i) / z_i$$

with z_i, the average number of processors in contention for resource i, being computed as

$$z_i = k/c_i \text{ where } c_i = v_i * [1 - (1-1/v_i)^k].$$

The remaining unknown t_i was defined earlier (for systems with $t_i \geq w_i$ such that one unit instruction for class i has length t_i+a_i. (We have assumed for simplicity that p_i=0. Otherwise a_i may be replaced by the appropriate expression in a_i, b_i and p_i.) However, t_i is not a function solely of the i-th resource (as assumed earlier), but rather of the execution rates of the n-1 other resources. Thus the equation above contains two unknowns, L and t_i. In order to eleminate t_i, we repeat the above equation for the n resource classes and add a constraint to form a system of n+1 equations in n+1 unknowns $\{t_1,t_2,...,t_n,L\}$. The constraint is that L must be the sum of the access times per instruction of each of the n shared resources, plus the service time of the non-shared resources in the skeleton processor (time required to decode, index, and issue instructions).

To obtain an equation for this constraint, consider the example of Figure 5. Shown is a six-instruction sequence for a system with three resource classes: two M.p modules, an add and a multiply unit. (We assume an access time of 3 minor cycles and a rewrite time of 2 minor cycles for M.p, for a major cycle time of 8 minor cycles.) The time occupied by communication a_i between the processor and each resource is shown by solid lines in the figure, with dashed lines representing the other activities w_i. Occasional delays d_i, represented by dotted-lines, are caused when the requested resource is busy serving requests from another processor (e.g. the first multiply is delayed 1 major cycle). Requests to functional units are sent on the last minor cycle of the instruction, with the result available exactly one major cycle later (cf. a_2's and a_3's and their associated w_2's and w_3's). The skeleton processor looks only one instruction ahead and hence need not worry about potential register conflicts. This was also subsumed in our concept of a unit instruction.

The individual times may be summed in order to form a constraint on the length of each machine instruction, as demonstrated in Table 3. The total elapsed time for one unit instruction on resource i is

$$t_i + a_i + d_i,$$

where d_i is the average delay due to contention for the i-th resource. (Thus the table entries for t_i may be found by subtracting a_i and d_i from the total time elapsed). We use this expression to determine an expected value for the length of one complete machine instruction L in terms of the i-th resource

$$L = (t_i + a_i + d_i) * f_i.$$

The i-th resource occupies time $(a_i + d_i) *f_i$ out of each instruction, which may be solved from the equation above to yield

$$(a_i + d_i) * f_i = L - t_i * f_i.$$

If we let Lo be the time required per machine instruction by the skeleton processor, we have as our constraint equation

$$L = Lo + \sum_{i=1}^{n} \{ L - t_i*f_i \}.$$

This completes our system of equations, which has a unique solution that may be determined numerically. The knowledge of L implies that of PER as defined previously and hence that of IER. The analytical solutions thus achieved are in accordance with the results from simulation presented in Table 4. The example system in Table 2 was simulated, with resource request frequencies determined by random draws from four typical instruction mixes [6]. The resulting instruction lengths are compared with the contention-free instruction lengths computed by ignoring time lost waiting for resources.

4. Summary and Conclusions

A general model of a large, tightly-coupled multiprocessor system has been introduced and shown to be capable of representing several recent design proposals and realizations. It was then reduced to a more specific model of a shared-resource multiprocessor for use in an analytical study of resource contention. By examining first the problem of interference in main memory, we have been able to abstract previous results [3,9] to find closed-form formulas for the effects of contention in any individual resource, on the assumption that the behavior of the system with respect to all of its other resources is known. Furthermore, we have combined the analyses of the separate resources to form a more complete model when processors contend for several resource classes simultaneously.

Solving for this model yields a unique solution which allows a prediction of performance and degradation in multiple-resource systems. Several hypothetical systems have been parameterized through the model, and the iterative numerical solution has converged to the correct processor execution rate in each case. The performance estimates measured by this analysis have been shown to be reasonable by simulation at the instruction level, and it is anticipated that future simulations of systems will make use of this result to account for hardware resource contention while retaining a high-level view of the systems being modeled.

References

[1] Anderson, D. W., Sparacio, F. J. and Tomasulo, R. M. "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling" IBM J. of R. & D. 11:1 (Jan. 1967), pp. 8-24.

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

[3] Bhandarkar, D. P. "Analytic Models for Memory Interference in Multiprocessor Computer Systems" Ph.D. Dissertation, Carnegie-Mellon University, Sept. 1973.

[4] Flynn, M. J. and Podvin, A. "An Unconventional Computer Architecture: Shared Resource Multiprocessing" Computer 5:2 (March-Apr. 1972), pp. 20-28.

[5] Gountanis, R. J. and Viss, N. L. "A Method of Processor Selection for Interrupt Handling in a Multiprocessor System" Proc. IEEE 54:12 (Dec. 1966) pp. 1812-1819.

[6] Jensen, J. E. "Dynamic Task Scheduling in a Shared Resource Multiprocessor" Ph.D. Dissertation, University of Washington (in preparation).

[7] Pariser, J. J. "Multiprocessing With Floating Executive Control" IEEE Int. Conv. Record, 1965, pp. 266-275.

[8] Skinner, C. E. and Asher, J. R. "Effects of Storage Contention on System Performance" IBM Systems J. 8:4 (1969), pp. 319-333.

[9] Strecker, W. D. "Analysis of the Instruction Rate in Certain Computer Structures" Ph.D. Dissertation, Carnegie-Mellon University, June 1970.

[10] Watson, W. J. "The TI ASC -- A Highly Modular and Flexible Super Computer Architecture" Proc. AFIPS 1972 F.J.C.C., pp. 221-228.

[11] Wulf, W. A. and Bell, C. G. "C.mmp -- A Multi-Mini-Processor" Proc. AFIPS 1972 F.J.C.C., pp. 765-777.

Table 1 - Contention Model Terminology

Ta	effective access time of M.p (service time)
Tw	effective rewrite time of M.p (recovery time)
Tp	average time between the completion of service on one memory request and the generation of the next request by P.c
Tb	cycle time of fast buffer memory
Sm	time required by M.p to service one request
Sp	time beyond memory cycle required by P.c to prepare the next request
Pb	probability of finding the request in buffer
Pm	probability that a request is queued at an M.p
a	service (access) time of each resource (Ta)
b	buffer speed for each resource (Tb)
c	number of processor-ring time-slots containing requests for each resource
d	delay time caused by contention at each resource
f	frequency of use for each resource (ratio of requests per number of machine instructions)
h	the contention function (UER)
i	index to the various resource classes
k	number of active processors (those to which tasks are currently assigned)
L	length of one machine instruction (inverse of IER on one processor)
m	number of resource units in each resource class
n	number of resource classes
p	probability of using buffer for each resource (Pb)
t	time between completion of service on one request for each resource and the generation of the next request for that resource (Tp)
v	number of stages in the functional-unit pipelines for each resource (coincides with the number of time-slices in the processor rings)
w	recovery (rewrite) time for each resource (Tw)
z	average number of processors in contention for each resource

Table 2 - Example of Resource Parameters

Resource Class	m	v	a	w	b	p
main memory	16	1	600ns	120ns	60ns	0.9
integer add	1	8	0	480ns	-	0
floating add	2	8	0	480ns	-	0
multiply unit	2	8	0	480ns	-	0
divide unit	2	8	0	1920ns	-	0
logical unit	1	8	0	480ns	-	0
shift unit	1	8	0	480ns	-	0
IO controller	4	1	0	12ms	-	0
scheduler	1	1	480ns	6μs	480ns	0.9

Table 4 - Comparison With Simulation Results
(average instruction length in nanoseconds)

Instruction Mix	Analytic Result	Simulation	Contention-Free
Floating Point	483	477	401
FORTRAN I/O	650	659	369
XPL/S Compiler	439	443	371
SIMTRAN Simulation	532	535	377

Table 3 - Timing Summary for the Computation A*B-C*D+E

	Total Occurrences			Length of Occurrence			Total Time Consumed			
	a_i	w_i	d_i	a_i	w_i	d_i	a_i	w_i	d_i	t_i
memory units	11	11	5	3	2	-	33	22	21	26
add unit	2	2	1	1	8	8	2	16	8	70
multiply unit	2	2	1	1	8	8	2	16	8	70

Total Decode Time (Lo)	6
Total Access Time a_i	37
Total Delay Time d_i	37
Total Time Elapsed	80
Number of Instructions	6
Average Instruction Length	13.3

Figure 3 - The Model Applied to Flynn's SRM

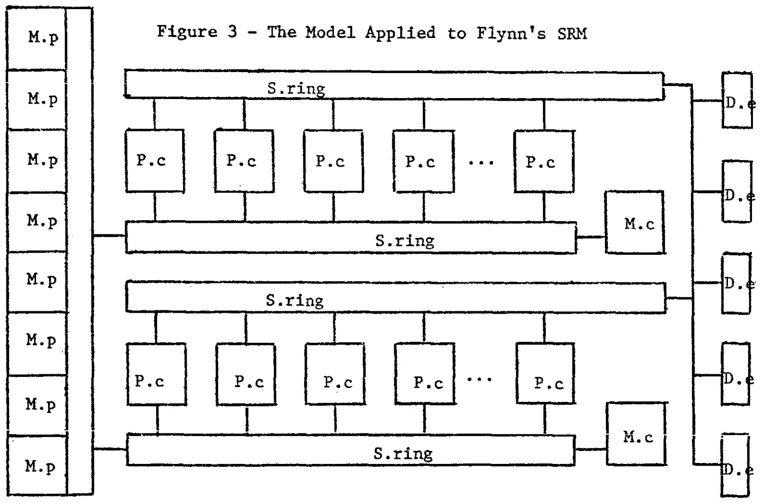


Figure 4 - The Simplified Machine Model

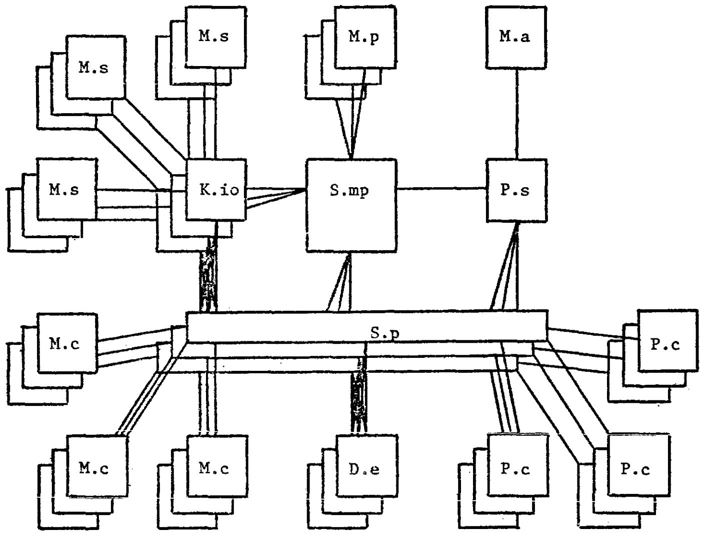


Figure 1 - A General Shared Resource Multiprocessor

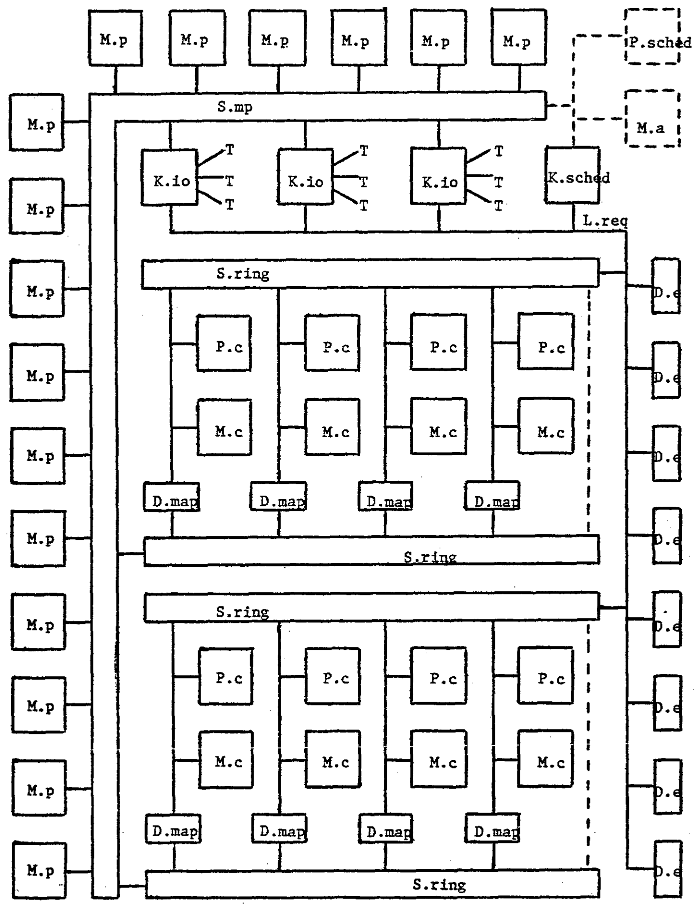


Figure 2 - The Model Applied to TI's ASC

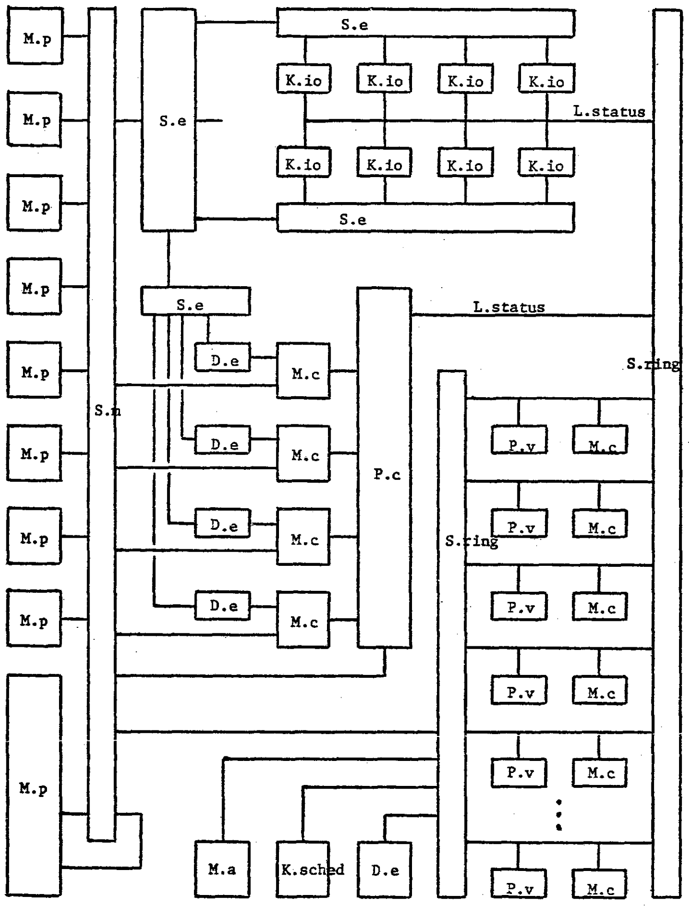
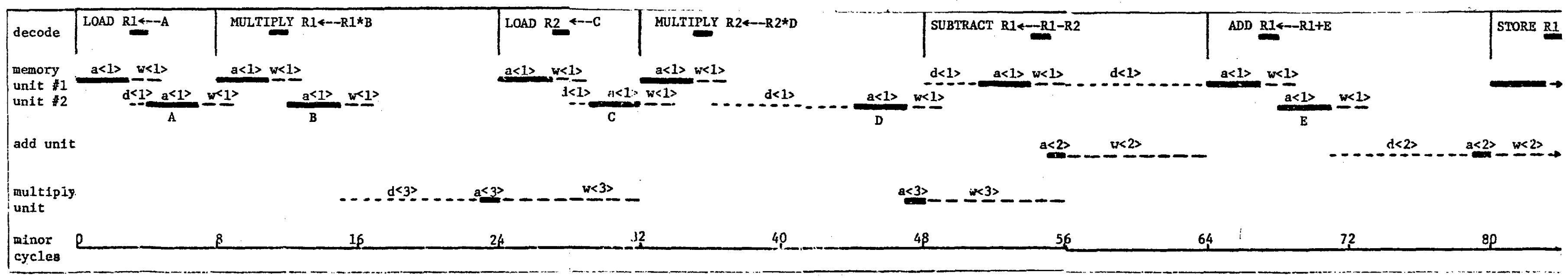


Figure 5 - Instruction Sequencing for the Computation A*B-C*D+E



A Computer Simulation Facility for Packet Communication Architecture *

by

Clement K.C. Leung
David P. Misunas
Andrij Neczwid
Jack B. Dennis

Project MAC
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract: Several proposals for computer data processing and memory systems that exploit the inherent parallelism in programs expressed in data flow form have been advanced recently. These systems have packet communication architecture -- each system consists of many units that interact only through the transmission of information packets through channels that link pairs of units.

A simulation facility for evaluating the programmability and potential performance of these proposed data processing and memory systems has been designed. The facility uses microprocessor modules to emulate the behavior of system units or groups of units. By conducting a simulation in accurate scale time a precise extrapolation of performance of a proposed system may be obtained.

The user of the facility will specify the system to be simulated in an architecture description language. A host computer translates the system description modules into microprocessor programs and controls the loading and monitors the operation of the microprocessors. Application of the facility is illustrated by consideration of a simple data flow processor.

Introduction

Recently, a number of proposals for computer data processing and memory systems organized to exploit the parallelism inherent in programs expressed in data flow form have been developed. These include a series of machines of increasing capability described by Dennis and Misunas [2, 3], two machines capable of supporting high level language including procedures as data [5, 6, 8, 9, 10] and memory systems organized for highly parallel operation [1].

Each of these systems consists of many units connected by channels, and is organized so the units operate asynchronously and interact only through transmission of information packets over the channels. Each unit of these systems is designed so it never has to wait for a response to a packet it has transmitted to another unit, if other packets are waiting for its attention; this design principle permits a high level of concurrent processing. The units themselves may be constructed of simpler units and channels that cooperate in the same manner, yielding a hierarchical structure in which interactions occur only at well-defined interfaces. Systems structured to operate according to this discipline are called packet communication systems and are said to have packet communication architecture.

The application of packet communication architecture to computer system design is now sufficiently advanced that careful evaluation of the performance potential of proposed systems is required. Since analytic techniques of sufficient power are not known, evaluations must be carried out by simulation. The simulation of a conventional computer architecture is readily carried out by programming a conventional Von Neuman-type computer, and the result of such simulation may be easily interpreted to predict performance of a proposed machine. However, simulation of a highly asynchronous system is not so easily accomplished using a conventional sequential computer -- much effort (in programming and in simulation runs) would be spent in the implementation of psuedo parallel processes and the coordination of their interactions.

With the advent of low-cost LSI processors there is an attractive alternative to programmed simulation on a conventional computer: a system having packet communication architecture is divided into parts and each part is emulated by a microprocessor. We have designed an architectural simulation facility based on this idea. The facility consists of a number of microprocessor modules arranged so they may easily communicate through a network for the simulation of any packet communication system. The system to be simulated is specified in an architecture description language designed expressly for packet communication systems. A host computer translates architecture descriptions into program modules executed by the microprocessors. The host computer also provides means for debugging and for measuring performance of the simulated system.

Our explanation of the simulation facility is aided by discussing its application to modeling the operation of a simple data flow processor. We start with a brief description of the data flow processor and show how the structure of this processor might appear when expressed in our architecture description language. Next comes a detailed discussion of the hardware portion of the facility and how it supports the modeling of packet communication systems. We conclude with a brief discussion of the software support to be implemented on the host computer.

An Example of Packet Communication Architecture

Throughout this paper we shall use a simple data flow processor as an example of a packet communication system. This data flow processor has been proposed for certain signal processing computations such as waveform generation and filtering in which a fixed constellation of operations is applied to a stream of data. The processor does not support data-dependent decisions, structured data, or procedures, though these features have been considered in generalized versions of this processor [3, 5, 6, 8].

The units and channels that comprise the top-level description of the data flow processor are shown in Figure 1. Instructions of a data flow program to be executed by the data flow processor are stored in Instruction Cells (Figure 2). Each Instruction Cell holds an instruction of the program, contains registers for holding one or two operands of the instruction, and is

* The work reported here was supported by the National Science Foundation under grant DCR75-04060.

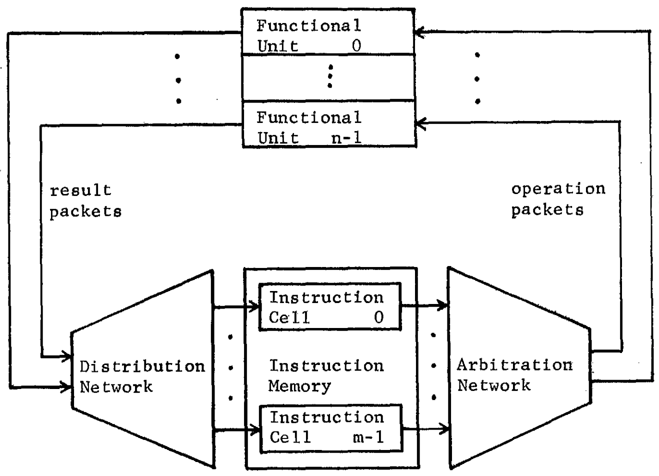


Figure 1. Structure of the elementary data flow processor.

designated by a unique cell identifier. An instruction specifies an operation to be performed on its operands and specifies each register (by a cell identifier and a register index 1 or 2) which is to receive a copy of the result. When all operands of an instruction are present in a Cell, the Cell is enabled and its content is transmitted as an operation packet to the Arbitration Network. Each operation packet is forwarded by the Arbitration Network to a Functional Unit capable of interpreting the operation packet. A Functional Unit performs the function specified by the instruction code of the operation packet it receives on the operands in the packet and, for each destination specified in the operation packet, generates a result packet consisting of a copy of the result and the cell identifier/register index of a destination cell register. The Distribution Network accepts result packets from the Functional Units, and delivers each result packet to the Cell addressed by the cell identifier in the packet. After the result packet is received by a Cell, the result in the packet is stored in the register addressed by the register index of the packet. If all of its operands are present, a Cell receiving a result packet is enabled and generates another operation packet to be processed. A more detailed description of the architecture and operation of the data flow processor is given in [2]. We note that depending on their construction, the Arbitration Network and the Distribution Network are capable of processing one or more packets simultaneously.

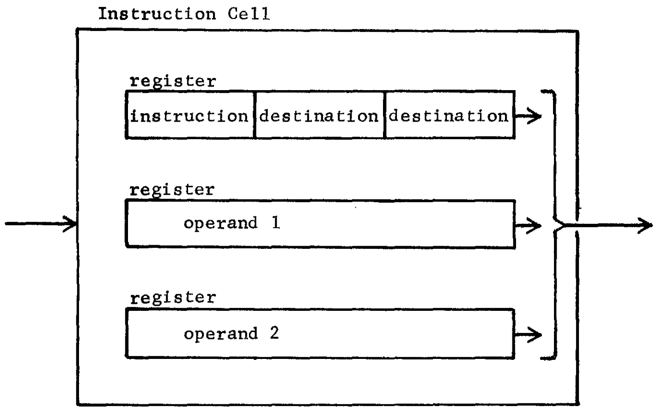


Figure 2. Structure of an Instruction Cell.

Architecture Description Language

Our architecture description language is a design notation for packet communication systems. The basic unit of description is a module with a number of input ports and output ports. A description module is either a structural description or a behavioral description. A structural description of a module specifies the decomposition of the module into simpler modules and the channels connecting ports of these simpler modules. A behavioral description specifies the module's behavior in the form of a sequentially executed program that:
(1) receives packets from a specified input port;
(2) transmits packets over a specified output port; or
(3) updates state variables of the module. In these respects our language is adapted from the notation used by Rumbaugh to formally describe his data flow multiprocessor [9].

In addition, our description language borrows much of its syntax, type structure and elementary control structure from PASCAL [11]. An information packet or a state variable is defined as a PASCAL record whose components are individually accessible. Packet type information is included in the specification for each channel connection, and for each input port and output port declation, permitting the support software for the simulation facility to enforce strong type checking throughout the hierarchical description of a system.

The overall architecture of the data flow processor is specified in the description language module Processor shown in Figure 3. Processor contains a list of submodules and a list of interconnections. The interface assumed for each submodule is given by the type of information packet which may be transmitted over its input and output ports. The relevant packet definitions

```
Processor: module (m: integer, n: integer);

structure:
Cell [1..m]: module
    distnet-in input port,
    arbnet-out output port;
Arbitration-Network: module (m, n)
    cell-in [1..m] input port,
    fcn-unit-out [1..n] output port;
Functional-Unit [1..n]: module
    arbnet-in input port
    distnet-out output port;
Distribution-Network: module (m, n)
    fcn-unit-in [1..n] input port,
    cell-out [1..m] output port;

Cell [1..m] . arbnet-out send operation-pkt
    to Arbitration-Network · cell-in [1..m];
Arbitration-Network · fcn-unit-out [1..n] send operation-pkt
    to Functional-Unit [1..n] · arbnet-in;
Functional-Unit [1..n] distnet-out send result-pkt
    to Distribution-Network·fcn-unit-in [1..n];
Distribution-Network · cell-out [1..m] send result-pkt
    to Cell [1..m] · distnet-in;

end Processor;
```

Figure 3. Top level description of the data flow processor.

```
address = record [cell-id: integer; register-id: integer];

operation-pkt = packet [opn: opcode;
                        destination: array [1..2] of address;
                        opd: array [1..2] of operand];

result-pkt    = packet [cell-id: integer;
                        register-id: integer;
                        opd: operand];
```

Figure 4. Packet definition.

for Processor are presented in Figure 4. The specification of the data types opcode and operand depends on the kind of computation to be implemented on the data flow processor and is not given in Figure 4. A complete specification of the data flow processor is obtained by supplying description modules for Cell, Arbitration-Network, Function-Unit and Distribution-Network. Each of these description modules must satisfy the interface requirements set forth in the definition of Processor and must implement the operation of the corresponding unit of the data flow processor as outlined above.

In illustration of the technique for specifying the behavior of a module, a specification of the operation of the module Cell is given in Figure 5. Cell communicates with the other submodules of Processor via its input port distnet-in and its output port arbnet-out. Packets of type result-pkt and operation-pkt are received and transmitted by Cell at distnet-in and arbnet-out respectively. The state variables of Cell provide storage for packets received and store state information for controlling the operation of Cell. The state variables are initialized and reset as necessary from one cycle of operation of Cell to the next. The when statement in Cell (Figure 5) is activated upon receipt, at distnet-in, of a result packet which delivers an operand to the instruction held in Cell. When all the required operands are available, an operation packet is formed and emitted at arbnet-out by the send statement (Figure 5). A when statement contains one or several blocks of statements, one block for processing the input packets arriving at each input port. The complete execution of a when statement embodies: (1) receiving and acknowledging an input packet from one of the input ports monitored by the when statement, and (2) executing the block of statements for processing input packets arriving at the input port.

The specifications of Processor and Cell illustrate the descriptive power of the architecture description language. Other submodules of Processor can be similarly defined. After presenting the hardware facilities in the next section, we will describe the implementation of the module Cell as a program executed on the processor modules.

Organization of the Simulation Facilities

The simulation facility shown in Figure 6 is composed of a host computer, a number of microcomputer modules each consisting of a microprocessor and a number of memory modules, a control bus for host-microcomputer communication, and a Routing Network for transmitting packets between microcomputer modules. The host computer loads simulation programs into microcomputer modules, monitors and controls the progress of a simulation, and collects statistical data for performance evaluation. The control bus transmits commands, addressing information and data from the host to the microcomputer modules, and transmits acknowledge signals and memory word contents from the microcomputer modules to the host. Under control of the host, microcomputer modules execute programs which simulate the operation of units of a simulated system. In addition to communicating with the host via the control bus, each microcomputer module is connected by an input port and an output port to the Routing Network, through which the module sends or receives packets from other modules.

The Routing Network provides a buffered path between every pair of microcomputer modules, permitting the transmission of packets without regard for whether the destination processor is ready to receive them. A packet transmitted to the Routing Network from a microcomputer consists of a destination address for the packet and the packet content. The destination address is used by the Routing Network to direct the packet to the input port of the appropriate microcomputer module. The Routing Network performs arbitration and distribution functions in a manner similar to that described for the Arbitration and Distribution Networks in [2].

Before we describe the structure of the commands issued by the host and the various schemes by which the

```
Cell: module
    distnet-in input port receives result-pkt,
    arbnet-out output port sends operation-pkt;

behavior

/* State Variables */
respkt : record result-pkt;
operation: opcode;
dest1, dest2: address;
operand1, operand2: operand;
opd1-expected, opd2-expected: boolean;
opd1-received, opd2-received: boolean;

repeat begin
    opd1-received := if opd1-expected then false else true;
    opd2-received := if opd2-expected then false else true;
    while ¬ opd1-received ∨ ¬ opd2-received do
        when distnet-in receives respkt do
            case respkt · register-id of
            1: begin
                if opd1-received then error;
                opd1-received := true;
                operand1 := respkt · value end;
            2: begin
                if opd2-received then error;
                opd2-received := true;
                operand2 := respkt · value end;
            endcase;
    send [opn: operation;
        destination[1]: dest1; destination[2]: dest2;
        opd[1]: operand1; opd[2]: operand2]
    at arbnet-out;
end repeat;

end Cell;
```

Figure 5. Specification of the operation of an Instruction Cell.

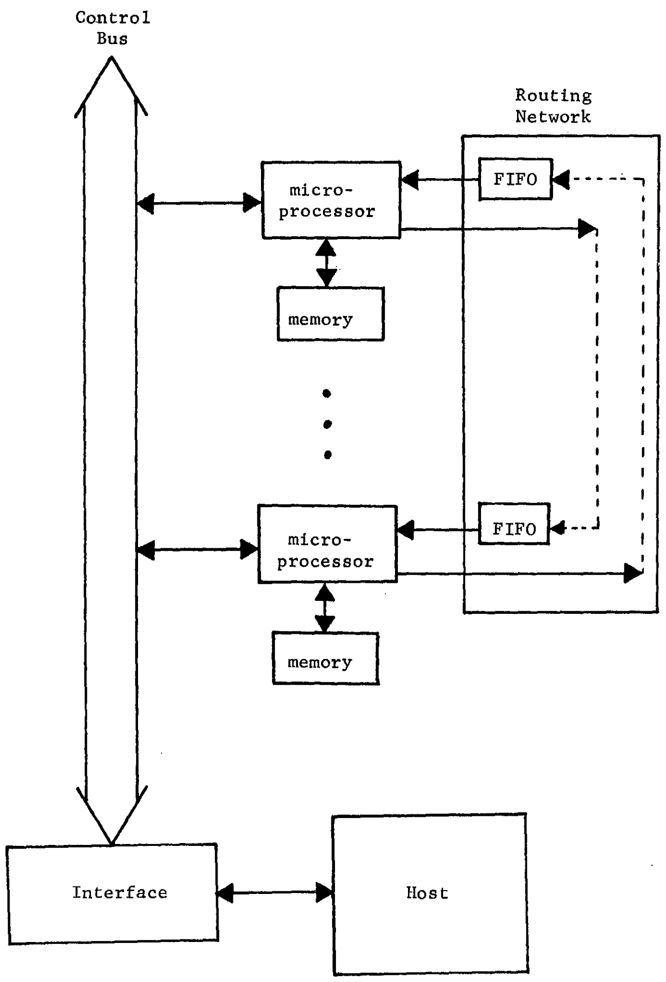


Figure 6. Organization of the simulation facility.

host controls a simulation, let us examine the mechanisms available for controlling a microcomputer module in more detail. The simulation program module contained in each microcomputer module is organized so program execution starts from a home state and returns to this home state after each transaction, that is, after the complete processing of a packet. Each microcomputer module also has a wait state in which no instructions are executed and control of the internal busses of the microcomputer module is relinquished to the host.

Two special registers in each microcomputer module, the run count and the wait flag, are set by the host and utilized to control the progress of a simulation. The run count of a microcomputer module is set by the host to the desired number of cycles of operation of the simulated unit for the current simulation. Each time a cycle of operation is completed, the run count is decremented. If the decremented run count is zero, the microcomputer module enters a wait state and signals to the host that it has entered that state. A negative run count enables a microcomputer module to process transactions until halted by the host. The wait flag is set by the host when it is desired that the designated microcomputer module(s) enter the wait state. The flag is checked by a microcomputer module when the module is in the home state. Hence, microcomputer modules placed in the wait state through setting of the wait flag have no partially completed transactions, and all state variables of the modules are in a consistent state.

The host performs its control functions by issuing commands to the microcomputer modules via the control bus. Commands issued by the host are either addressed or universal. A universal command is obeyed by all microcomputer modules, and such commands are used by the host to start, stop, and temporarily suspend the execution of a simulation. An addressed command is executed only by a designated microcomputer module. Each command transmitted over the bus consists of a selection code and a command name. The selection code specifies which of the microcomputer modules is to respond to the associated command. Each microcomputer module examines the selection code of each command to determine whether the module should respond to it.

The host can issue one of nine commands to a microcomputer module. The possible commands are Read, Write, Hold, Release, Halt, Enable, Clear, Start and Reset. The Read and Write commands provide the capability to examine or alter the contents of a memory module associated with a microcomputer module. The other commands are used in the selection of a microcomputer module for execution of a Read or Write command, or for controlling the progress of a simulation.

Often, it is desired that several, but not all, of the microcomputer modules respond to a Write command simultaneously, for example, when loading a simulation program into a number of microcomputers which are to simulate identical units. This function is accomplished by individually issuing Enable commands to the desired processors. Commands issued subsequently are executed by all enabled processors until a Clear command is received from the host. Note that the Clear command can be either addressed or universal.

The Start, Hold, Release, Halt and Reset commands are used to implement the various schemes by which the host controls a simulation. All microcomputer modules of the system are initially in the wait state. A simulation is initiated by a universal Start command which places all microcomputer modules in their home states. A simple scheme to halt a simulation is to issue a universal Hold command which halts program execution in all microcomputer modules immediately. The host is then free to read or write into the memory modules by issuing Read and Write commands. Program execution at each microcomputer module can be restarted at the point of interruption by issuing a universal Release command.

All microcomputer modules can be put into their wait states simultaneously and immediately by issuing a Reset command. However, when the microcomputer modules are to be stopped for the purpose of debugging and evaluation, all modules should be in consistent states. This is accomplished through the use of a universal Halt command. Execution of the Halt command sets the wait flag of each microcomputer module by generating a universal Hold command followed by a universal Write into the wait flags, and then a universal Release. Each microcomputer module, upon reaching its home state, then discovers that its wait flag is set, enters its wait state, and signals the host. When the host has received an acknowledge signal designating that each microcomputer module has entered its wait state, it can examine and alter the memory contents of any microcomputer module, and it can examine the status of each microcomputer input port to see if there are any packets present.

Once a simulation has been halted and the status of the facility has been determined, one or several microcomputer modules can be enabled for a specified number of transactions by properly setting their run counts, setting the wait flags of the other microcomputer modules and then issuing a universal Start command. Receipt of the Start command causes each microcomputer module to exit its wait state and reenter its home state. The microcomputer modules whose run counts were set will accept packets at their input ports. All others will immediately reenter their wait states.

An active microcomputer module will signal the host computer after completing the specified number of transactions. The acknowledge signals from the microcomputer modules are ANDed and ORed to produce a Universal Acknowledge and an Addressed Acknowledge, indicating that the appropriate processors have responded to a universal or addressed command.

The various control schemes and communication protocols presented provide a minimal capability for controlling and examining system operation during a simulation. The fact that the host can readily access the individual memory modules allows one to easily extend the control, analysis and debugging capabilities in software. Each microcomputer module can store any desired status information in its memory for the host to retrieve, even to the point of retaining all packets processed by the module.

An example of a software evaluation facility is the evaluation of performance of individual sections of a simulated processor through analysis of _event counts_. An event count is a count maintained by an individual microcomputer of the number of transactions which have taken place since initiation of a timing interval. The use of event counts allows the study of the relative efficiency of sections of the simulated processor and provides data necessary for determining such parameters as cache size and structure of the memory/processor interconnection networks.

Simulation of a Packet Communication System on the Hardware Facility

A packet communication system is simulated on the hardware facility through simulation of one or more units of the system on each microcomputer module. The constructs used in the simulation programs are implemented on a microcomputer module in a straightforward manner. The implementation of packet transmission and processing, the identification of microcomputer states during program execution and the coordination between packet processing and microcomputer state transitions are further illustrated in this section using the module _Cell_ (Figure 5) as an example.

In general, a unit simulated on a microcomputer module may have several input ports. A separate input buffer is allocated in memory for each input port of the simulated unit. Every packet transmitted through the Routing Network specifies a target port, which is an input port of a simulated unit. A microcomputer module, upon receipt of a packet, uses this target port designation to deposit the packet in one of its input buffers.

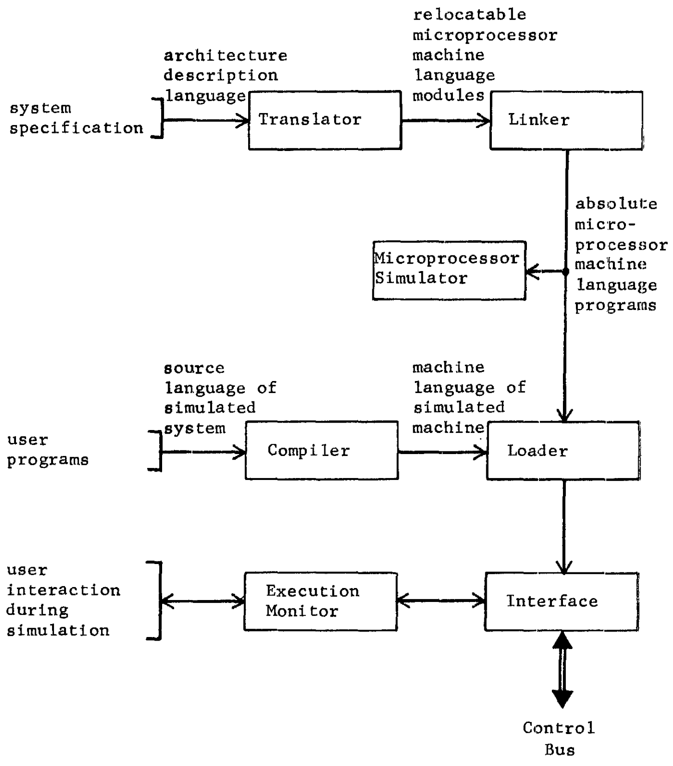
The program module _Cell_ has one input port _distnet-in_. If _Cell_ is the only unit simulated on a microcomputer module, every packet arriving at the input port of the microcomputer is automatically deposited in the buffer associated with _distnet-in_. Any output packet of the module _Cell_ is transmitted through the output port _arbnet-out_.

Each microcomputer module is in a wait state after the simulation programs have been loaded. A Start command transfers the microcomputer module from the wait state to the home state, and initiates execution of the simulation program. Unless temporarily halted by a Hold command, the execution of a simulation program on a microcomputer module proceeds until a _when_ statement is reached, at which point the microcomputer reenters its home state. Upon reentering its home state, the module examines its wait flag and enters the wait state if the wait flag has been set by the host. If the wait flag is not set, the microcomputer module queries its wait flag and the status of the input ports monitored by the _when_ statement in turn using a round-robin algorithm, until the wait flag is set or an input packet becomes available. If the wait flag is set, the microcomputer enters its wait state. If an input packet becomes available first, the _when_ statement is executed.

When the program module simulating _Cell_ is executed on a microcomputer, the microcomputer enters its home state each time the _when_ statement (Figure 5) that receives result packets at _distnet-in_ is reached. The run count of a microcomputer module is decremented at the end of each cycle of operation of the simulated unit, and the microcomputer module enters its wait state if the updated run count becomes zero. In the case of _Cell_, the run count is decremented and examined each time the body of the outermost _repeat_ statement is executed.

Software Support

The structure of the controlling software system for the simulation facility is presented in Figure 7. Operation of sections of the simulated system is specified by modules in the architecture description language in the manner described earlier. These modules are translated into relocatable microprocessor object



relocatable microprocessor machine language modules
architecture description language
system specification
Translator
Linker
absolute micro-processor machine language programs
Microprocessor Simulator
source language of simulated system
machine language of simulated machine
user programs
Compiler
Loader
user interaction during simulation
Execution Monitor
Interface
Control Bus

Figure 7. Structure of the simulation control system.

code and are stored in the file system of the host computer; the necessary programs from the file are linked together to form a non-relocatable microprocessor program. Either the individual procedures or a complete simulation program can be tested by use of a microprocessor simulator residing in the host computer. Once the simulation programs have been validated by use of the microprocessor simulator, the programs are loaded into the microprocessors, and the facility is ready to execute a program of the simulated machine.

A user program to be executed on the simulated architecture is compiled into the machine language of the simulated machine and sent to the microprocessor system for execution. The debugging and evaluation capabilities of the system are used to control execution of the program and evaluate feasibility of the proposed system architecture.

Conclusion

The architecture simulation facility appears to be a powerful tool for the evaluation of packet communication systems. Its capabilities permit the testing and evaluation of a broad range of architectural concepts. The facility is currently under construction using the Motorola M6800 microprocessor and a DEC PDP-11 host computer. Portions of the software system are being developed on a PDP-10 computer to allow use of the language CLU [4, 7]. The system is intended to be used primarily for an investigation of the design and capabilities of data-flow processors, and we expect it to be invaluable for this application.

Acknowledgements

The authors wish to thank Bob Jacobsen and Dave Isaman for many helpful comments and suggestions.

References

1. Dennis, J. B. Packet communication architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, 1975.
2. 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, 402-409.
3. Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, 1975, 126-132.
4. Liskov, B. H., and S. N. Zilles. Programming with abstract data types. Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices 9, 4 (April 1974), 50-59.
5. Misunas, D. P. A Computer Architecture for Data-Flow Computation. S.M. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., June 1975.
6. Misunas, D. P. Structure processing in a data-flow computer. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, 1975.
7. Project MAC Progress Report XI, July 1973-1974. Project MAC, M.I.T., Cambridge, Mass., 35-50.
8. Project MAC Progress Report XI, July 1973-1974. Project MAC, M.I.T., Cambridge, Mass., 84-90.
9. Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Report TR-150, Project MAC, M.I.T., Cambridge, Mass., May 1975.
10. Rumbaugh, J. E. A data flow multiprocessor. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York 1975.
11. Wirth, N. The programming language PASCAL. Acta Informatica 1 (1971), 35-63.

COST, PERFORMANCE AND SIZE TRADEOFFS FOR DIFFERENT LEVELS IN A MEMORY HIERARCHY

S. L. Rege
EMSO, Advanced Development
Burroughs Corporation
Piscataway, New Jersey 08854

Abstract

This paper evaluates the effect of cost and performance tradeoffs on memory system hierarchies achieved by varying the total amount of memory at any two adjacent levels. The hierarchy is analyzed in a multiprogramming mode by using a two server cyclic queuing model. As an example, a two level hierarchy of Bipolar, MOS and a three level hierarchy of Bipolar, MOS, and CCD for the primary memory are compared. A figure of merit that is a function of the number of instructions executed by a given processor is used to evaluate the different memory hierarchies. It is shown that up to 3:1 advantage in performance can be achieved by using a three level rather than the two level hierarchy at the same total cost. The effect on the performance of the memory hierarchy due to the change in the degree of multiprogramming, the speed and cost of CCD technology used, the speed of the CPU used and the amount of CCD and MOS memory used are then evaluated. The performance of two and three level hierarchies is also analyzed as a function of the primary memory requirements versus the CCD speed.

Introduction

Until recently, electronically addressable devices such as ferrite core, plated wire, semiconductor memories and the electromechanically addressable devices such as magnetic tapes, disks and drums were the few technologies from which a computer system designer could build a memory system. A number of different new technologies and devices have been developed that close the 'access gap'[9] between the two dissimilar technologies mentioned above. Some of these are the Charge Coupled Devices (CCD's)[2], Bubble Memories[4], Electron Beam Addressed Memories (EBAM)[11] and Domain Tip Propagation (DOT)[10]. Other technologies like CMOS[1], and Integrated Injection Logic (I^2L)[7], compete directly with the existing technologies. Table 1 shows the possibility of a six level hierarchy and some cost and performance projections for these technologies.

This paper uses a two server queuing network model to analyze the cost/performance tradeoffs achievable by using various sizes of the memories at different levels and various combinations of the memory technologies. Specifically CCD is used as an example in the 'access gap' and comparison is made between two and three level hierarchies for the primary memory. A multi-programmed mode of operation is assumed and a figure of merit is defined to analyze the hierarchies.

Multiprogramming and Memory Hierarchy

Multiprogramming is multiplexing of CPU over a number of different tasks residing in the primary memory. A task switch is made whenever a particular task has to wait for certain resource (e.g. secondary memory) long enough to justify the overhead involved in switching. Multi-programming, a psuedo parallel operation, has been used as a method to enhance the CPU utilization when memory technologies and I/O equipment with significant access gaps are used. Therefore, given a processor and a memory hierarchy using different technologies T_1, $T_2 \ldots T_n$ (Figure 1), a boundary exists such that an access across the boundary necessitates a task switch. There may be various reasons for the boundary to exist between any two particular technologies. One of the main reasons is the disparity between the task switching time required by the processor and software and the access time of the technology.

In the memory hierarchy (Figure 1) the technologies that are used on the processor side of the task switching boundary form a part of the primsry memory, while the others form the secondary memory. The degree of multi-programming is the average number of active tasks that reside in the primary memory and is usually a function of the primary memory size and working set size of the program.

Model and Assumptions

The behavior of a typical task executed in a multiprogramming environment is represented by four states: the task being serviced by the processor, the task waiting for the secondary memory or I/O service in a queue, the task being serviced by the secondary memory or I/O, and finally, the task waiting in a queue for processor service. Thus, in general, there are two queues and two service facilities and a task cycles through them until it is completed (Figure 2). This, then, can be modeled by a two server cyclic queuing model. Traiger[13] has referenced the use of this model, Fuller and Baskett[5] have used it in their analysis of scheduling philosophies of drum systems while Bhandarkar[3] has used it to compare magnetic bubbles, CCD's, Fixed and Moving Head disks, etc. Most previous researchers have used CPU utilization as a main criterion to evaluate the effect of multiprogramming. Some of the other criterions considered are the waiting time in queue and the memory utilization, which is the percentage of the time that a given memory spends its time transferring its data. The criterion used here will be the ratio of the actual number of instructions executed by the processor to the maximum number of instructions executed provided all the memory was substituted by the level having the fastest speed.

The assumption made in using the two server queuing model (Figure 2) is that both server one, consisting of the processor and the primary memory, and server two, consisting of secondary memory and I/O, have an exponential service time distribution. Even though this may not be the case in any particular computing system, most models make this assumption since most natural phenomenon can be modeled by a poisson process and a general feeling for the performance of the hierarchy can be determined. Later, simulations may be used to verify the results. A FIFO scheduling philosophy is assumed for all queues in the system.

Hit Ratio Characteristics

A typical hit ratio characteristic as shown in Figure 3 is used to determine the performance of the hierarchy. The statistics were taken from some representative programs for a large computer. Once the hit ratio characteristics are known, the miss ratio characteristics can be easily determined.

Processor Characteristics

A typical processor activity is characterized as an instruction fetch, instruction decode, data fetch and data operation (Figure 4). Thus, using this model, the average time interval between the issuance of successive memory accesses can be determined. For a more rigorous analysis of the processor behavior characteristics, see Strecker[12].

Performance of the Hierarchy

If λ is assumed to be the average service rate of the first server, then the mean execution interval $1/\lambda$ can be expressed as [Bhandarkar[3]]:

$$1/\lambda = \frac{\text{Hit Ratio}}{\text{Miss Ratio}} \, [t\,(M_p) + t\,(P_c)]$$

Where $t\,(M_p)$ = aggragate access time for the primary memory

$t\,(P_c)$ = average processing time between successive memory accesses.

Assuming μ as the service rate for the second server the probability of CPU being busy or CPU utilization is given by:

U = probability of CPU being busy

= 1 - probability (M jobs queued for second server)

$$= \frac{1 - \rho^M}{1 - \rho^{M+1}} \qquad \text{(Hiller[6])}$$

Where M = the degree of multiprogramming and

$\rho = \lambda/\mu$

Once the CPU utilization is found, then the figure of merit (f) can be derived as:

$$f = \frac{\text{No. of inst. executed with a given hierarchy}}{\text{No. of inst. executed with all memory substituted by fastest technology}}$$

$$= \frac{t\,(P_c) + t\,(\text{fastest memory})}{t\,(P_c) + t\,(M_p)} * U$$

Where t (fastest memory) = access time of the fastest memory, and U is determined by using the equation given above.

A Memory Hierarchy Design

The final outcome of a memory system design in which a user is interested is its cost and performance. Invariably, the requirements are to minimize the cost while maximizing the performance. The cost and performance of the memory system is a

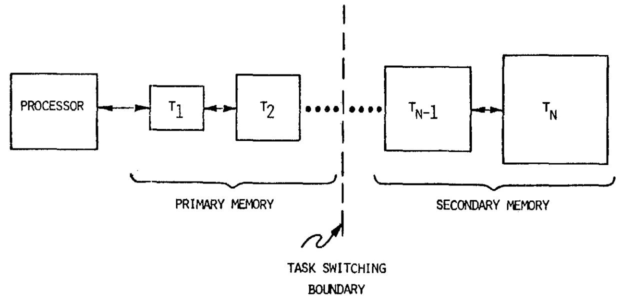
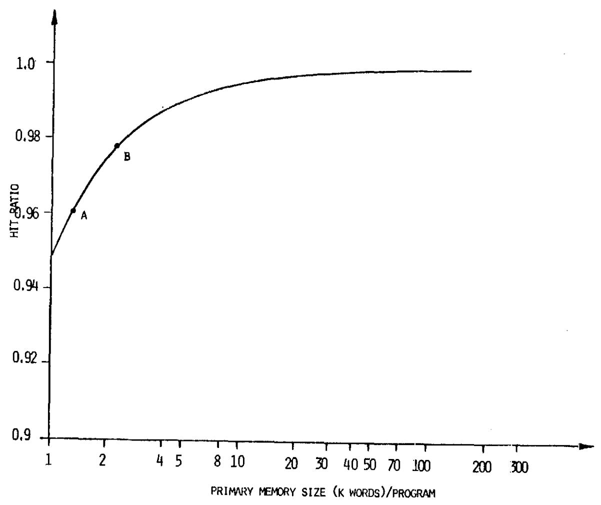


FIGURE 1: A MEMORY HIERARCHY WITH A TASK SWITCHING BOUNDARY

FIGURE 3: TYPICAL HIT RATIOS FOR A SET OF REPRESENTATIVE PROGRAMS



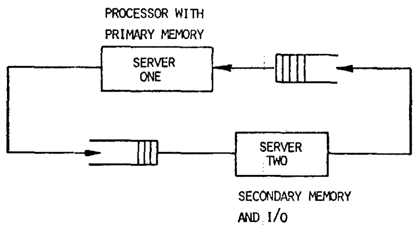


FIGURE 2: TWO SERVER QUEUING MODEL

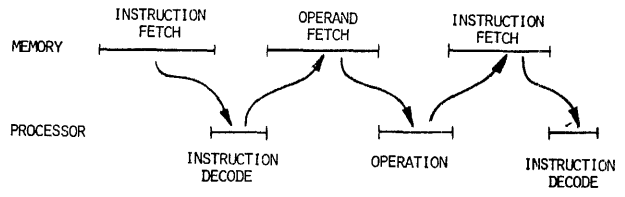


FIGURE 4: MODEL FOR PROCESSOR AND MEMORY OPERATION

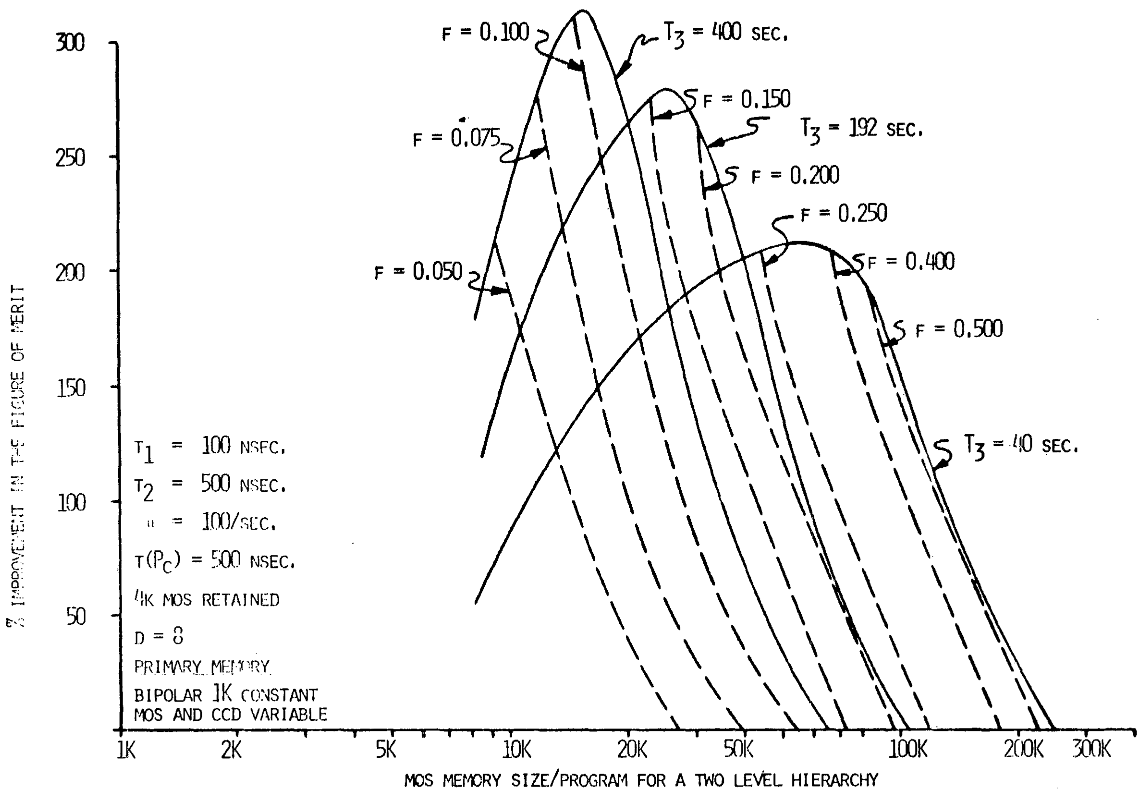


FIGURE 5: EFFECT OF CONSTANT COST AND CONSTANT PERFORMANCE TRANSFORMATIONS ON A MEMORY SYSTEM DESIGN

FIGURE 6: GRAPH SHOWING THE ADVANTAGEOUS REGIONS FOR TWO AND THREE LEVEL MEMORY HIERARCHIES FOR PRIMARY MEMORY.

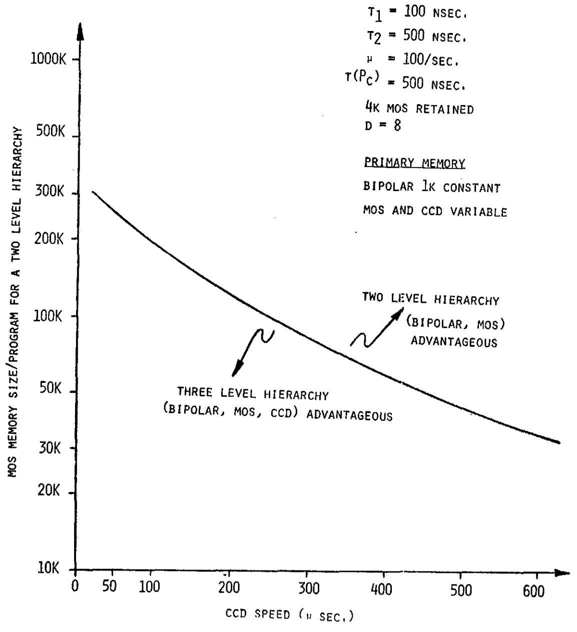
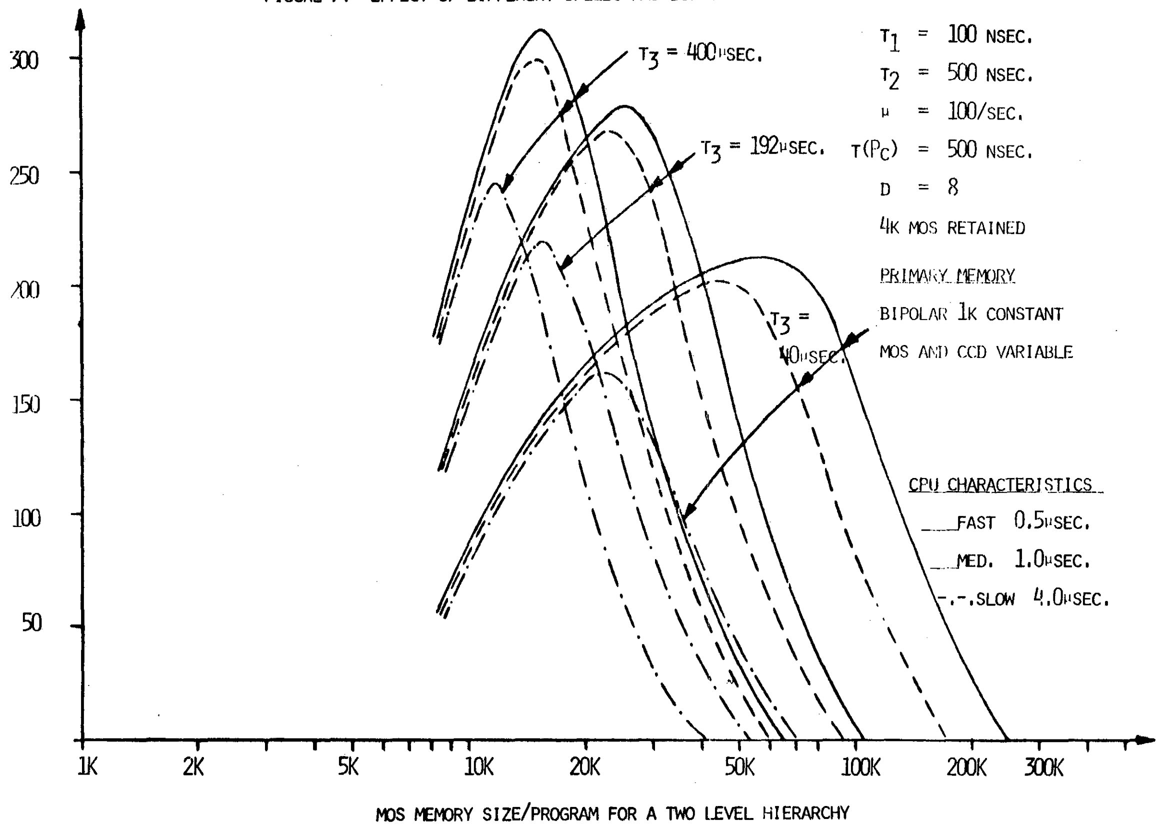


FIGURE 7: EFFECT OF DIFFERENT SPEEDS AND DIFFERENT CPU'S ON PERFORMANCE



MOS MEMORY SIZE/PROGRAM FOR A TWO LEVEL HIERARCHY

FIGURE 8: EFFECT OF DEGREE OF MULTIPROGRAMMING ON PERFORMANCE OF THE SYSTEM

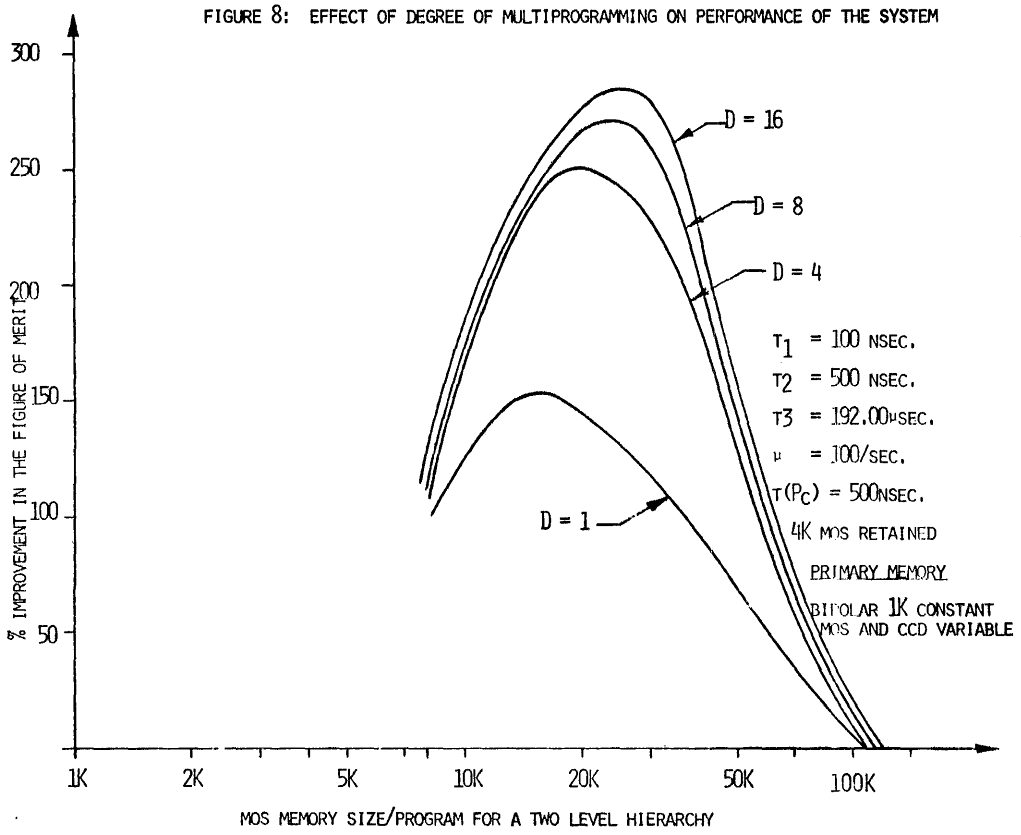
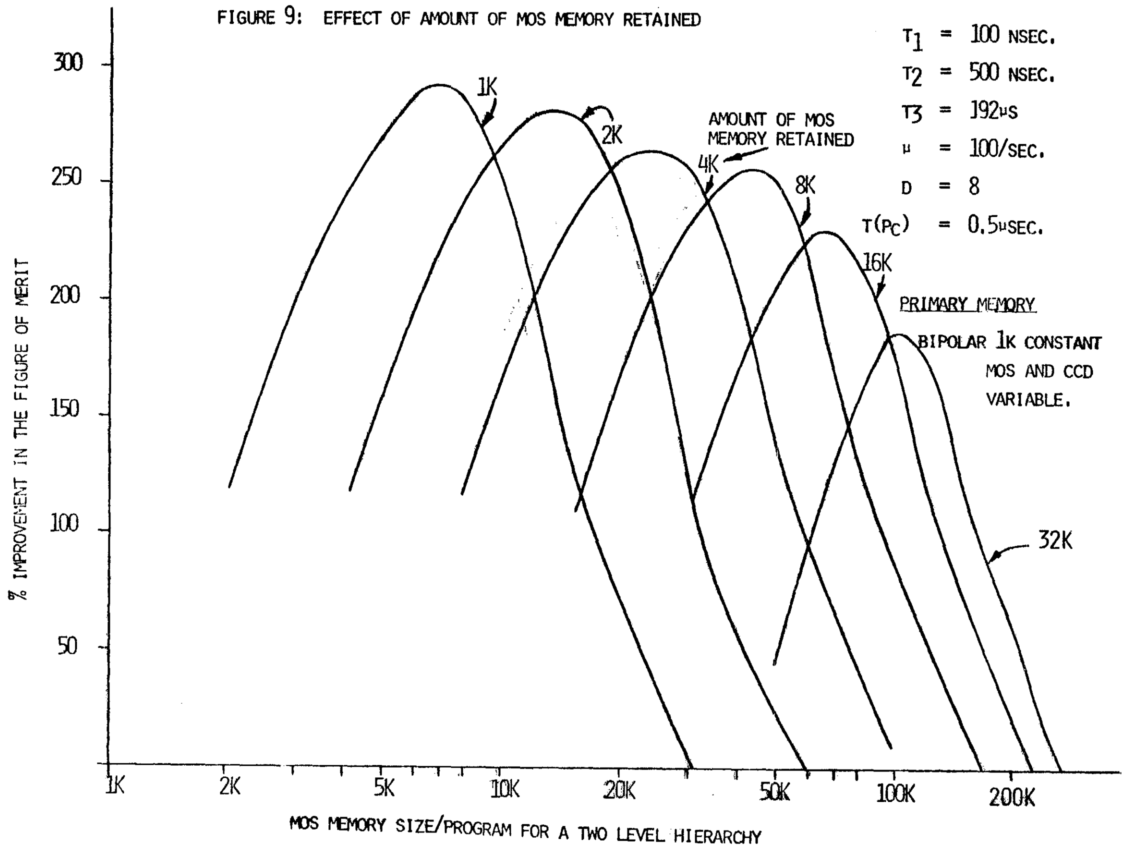


FIGURE 9: EFFECT OF AMOUNT OF MOS MEMORY RETAINED



STORAGE LEVEL	COMMONLY USED NAME	RANDOM ACCESS TIME (APPROX.)	COST/BIT	TECHNOLOGY	
1	Cache	50 nsec.*	1.0¢	Fast Semiconductor RAM	-----
2	Main	500 nsec.	0.1¢	Slow Semiconductor RAM	Ferrite CORE
3	Block or Swapping	50μsec.	40m¢	Fast CCD	EBAM
4	Backing Store	400μsec.	10m¢	Slow CCD Fast Bubbles	Fixed Head Disks & Drums
5	Secondary	50 msec.	1m¢	Slow Bubbles	Moving Head Disks
6	Mass	5 sec.	0.1m¢	Automated Tape Tape Handlers	Laser Devices, Etc.

TABLE 1

STORAGE TECHNOLOGY SPECTRUM, MEMORY HIERARCHY LEVELS AND COST PROJECTIONS

*(See Martin and Frankel [1975] for cost performance projections.)

NAME	SYMBOL	CHARACTERISTICS	REMARKS
PROCESSOR	$t(P_c)$	FAST 0.5 usec. MED. 1 usec. SLOW 4 usec.	The processor characteristics are for average time between issuance of memory requests.
BIPOLAR MEMORY	T_1	100 nsec.	
MOS MEMORY	T_2	500 nsec.	
CCD MEMORY	T_3	FAST 40 usec. MED. 192 usec. SLOW 400 usec.	Cost ratio with MOS 3 4 5
DISK AND I/O	-	10 msec.	
DEGREE OF MULTIPROGRAMMING	D	1 TO 8	-

TABLE 2: DIFFERENT PARAMETERS USED FOR THE EVALUATION OF THE PERFORMANCE OF THE HIERARCHY

AN INPUT INTERFACE FOR A REAL-TIME DIGITAL SOUND GENERATION SYSTEM

by

Paul E. Dworak
Department of Music
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Alice C. Parker
Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

A man/computer input interface is described in this paper. This interface allows human control of the frequency, amplitude, spectral content and envelope of real-time sound production.

The interface consists of a two-dimensional array of keys for entering data, latches and comparators to signal changes in key depression, hardware for scanning the keyboard and address generation logic. Both the fundamental frequency represented by each key column and the harmonics of each frequency -- represented by each element in the column -- are programmable. Amplitude and envelope are software controlled.

Applications of the interface instrument range from music composition to clinical use in auditory testing. The interface is part of an electronic sound generation system presently being designed and constructed.

Introduction

The designs of electronic sound-generating instruments during the past 25 years have demonstrated that it is both desirable and necessary for an instrument to vary, under human control, the frequency (pitch), amplitude, spectral content (tone color or timbre) and envelope (attack-decay) of a sound being produced. At present, analog synthesizers can provide real-time control of all these parameters, but at best, only a few different events may be programmed[1].

Hybrid analog/digital systems are limited in the number of parameters that can be varied simultaneously.[2] [3] On the other hand, completely generalized sound production can be obtained using a digital computer and a software approach.[4] [5] This has not been attempted previously on a real-time basis. However, with the introduction of high-speed, low-cost digital logic, it seems reasonable to hope for real-time digital sound production if an efficient man/computer input interface can be designed. Such an input interface should provide information about frequency, amplitude, spectral/content and envelope to a dedicated central processor.

The real problem that needs to be solved in interface construction is the meaningful representation of as many parameters as possible on an input device with limited dimensions and little hardware. In his own electronic music studio, Stockhausen[6] has rediscovered and demonstrated that the combinations pitch and tape speed (tempt), loudness and tape speed and even spectral content and speed are interdependent. By constructing similar relationships, it is not only desirable, but possible, to represent up to four musical parameters (pitch, amplitude, envelope and spectral content on a two-dimensional input device by carefully defining the relationships between the parameters.

The device described here has been developed as part of a system that will provide this degree of flexibility. The total system design consists of the following units (see Figure 1):

1. A keyboard-like interface to provide direct control of frequency, spectral content, amplitude and envelope
2. A microprocessor to service and interpret the binary information provided by the input interface, to allocate digital oscillators and to control the digital-to-analog conversion process
3. A minicomputer employed to provide auxiliary software input, mass storage for memory of recent events and increased computing power
4. A random access memory for storing waveforms, sample increment information used in frequency computations and amplitude curves

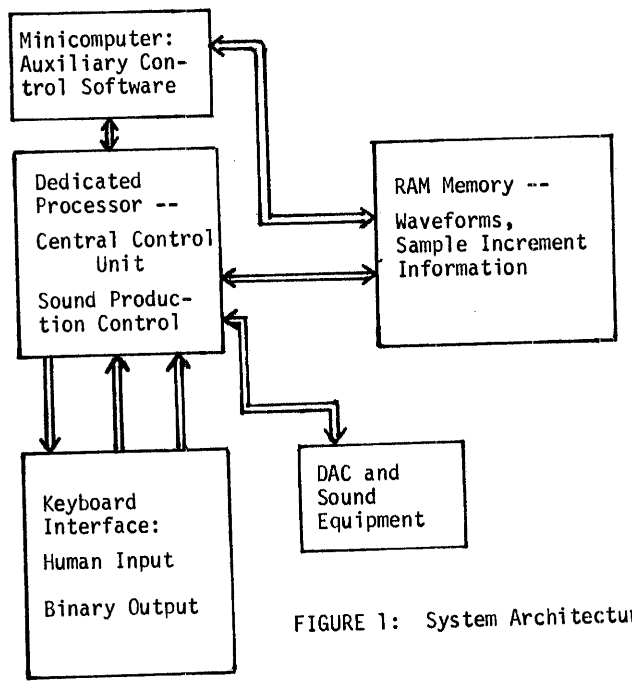

Minicomputer: Auxiliary Control Software
Dedicated Processor -- Central Control Unit Sound Production Control
RAM Memory -- Waveforms, Sample Increment Information
Keyboard Interface: Human Input Binary Output
DAC and Sound Equipment

FIGURE 1: System Architecture

Since the instrument being described is digital, it allows the use of any frequency values, any frequency increments and any waveforms. These parameters may be predefined or varied under minicomputer-program control.

Sampling theory and digital sound production are well documented in the literature. Furthermore, except for the interface, the other elements of the system outlined in Figure 1 are available commercially. The remainder of this paper will therefore deal with the details of the design of the interface.

A Multi-Input Interface

The digital interface, a two-dimensional prototype of which has been constructed and tested, is designed to convert information entered along its X and Y axes into binary information representing the fundamental frequency, spectral content and amplitude. Since the output data of the sound desired is binary and the system is programmable, the interpretation of entered data by the sound generation system is software dependent. The interface is a multi-input device, as opposed to many single-input analog keyboards. The number of inputs that can be entered simultaneously depends upon the cycle time of the central processor used and upon the memory capacity of the total system, but should never be less than ten.

The keyboard is a 256-by-8, X-by-Y matrix of siwtches. The prototype of the keyboard which has been constructed has an 8-by-3 array of switches (Figure 2).

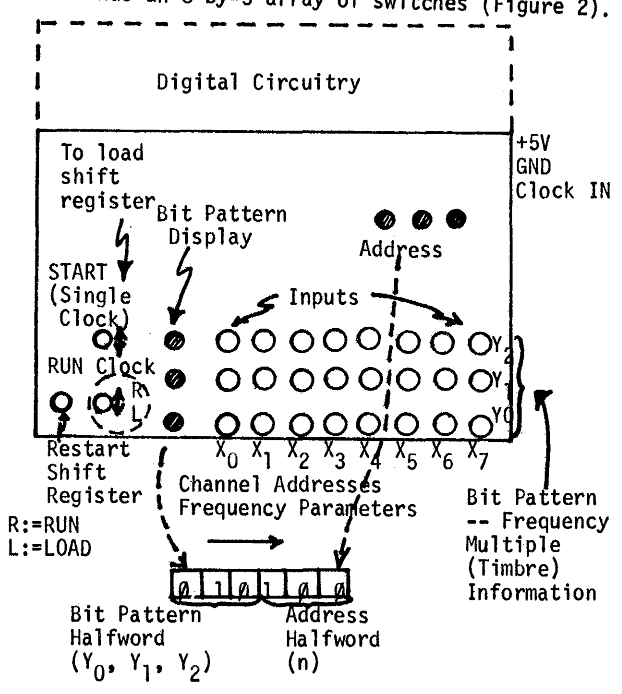


Figure 2: Physical Layout of the Prototype Keyboard Interface and Sample Data Word

The X addresses are quickly scanned to locate changes in key depression from the last scan. The binary address of the changed key status is stored in the right half of each binary data word.

The switches depressed at address X_n(any single X address) are represented by a bit pattern corresponding to $(X_nY_1, X_nY_2, \ldots, X_nY_m)$ $(X_{nON}Y_{mON} = 1$ and $X_{nON}Y_{mOFF} = \emptyset)$. It is apparent that the X addresses may be programmed to correspond to any frequency, as a result of which all pitch intervals and degrees of microtonal tuning are possible.

The bit pattern information at each address is envisioned as corresponding to harmonic or nonharmonic multiples of the fundamental frequency specified by each X address. If a sine wave is specified, if $Y_{\emptyset}$ = fundamental frequency, and if Y_1 and Y_2 are the first two integral multiples of $Y_{\emptyset}$, simple Fourier synthesis of a few waveforms is possible. If the inputs $Y_{\emptyset}$ through Y_2 reference square waves, sawtooth waves or user-supplied waveforms, more complex waveforms are possible. Finally, if Y_1 and Y_2 represent nonharmonic multiples of the frequency specified by $X_nY_{\emptyset}$, other complex tones can be derived. Of course, more interesting tones can be produced when eight "Y" inputs are available.

The interface will be scanned from $X_{\emptyset}$ to X_{255} in operate mode and whenever a new signal is found at any address, scanning will be stopped. An interrupt will be supplied to the central processor. The central processor will provide logic signal to restart the scan. In the prototype, a restart can be supplied manually.

The inputs will be interpreted as follows: the address X_n will be interpreted as the RAM location address, the contents of which contain the fixed-point increment to be used during the sampling process. The bit patterns for each X address will act as indirect references to addresses containing other such increment information. The bit pattern information will also instruct the processor how many summations are to be performed (how many overtones are present) before a sample is supplied to the digital-to-analog converter.

Scanning will stop only when a new input appears or when an old input disappears (onset of sound and end of sound). Signals that are unchanged will be ignored, as will blank addresses. Processing of signals in progress will be uninterrupted unless a change in some aprameter occurs (see Figure 3).

Memory locations will be allocated in RAM for each X-key column and a conversion from the key frequencies to the binary sample increment value needed by the central processor for sound production will be performed by a software program. Loading the binary values will be performed by the minicomputer. Waveforms for amplitude control will be loaded in allocated storage in a similar fashion, employing a program that allows the waveforms to be "drawn" in. The interface interpretation program will be stored in ROM. Both the keyboard and the RAM memory will be interfaced as peripherals to the central processor via input-output ports. Information represented by the keyboard interface states is dependent on the contents of the RAM memory.

Representation of Amplitude and of Envelope

It has already been explained that, for any address X_n, the Y switches represent programmable overtones. With no additional hardware, the bit pattern representation of the Y switches can be used to describe amplitude directly and envelope (amplitude over time) indirectly. The following conventions will be followed:

1. The greater the number of Y switches depressed at any address, the greater

(1)	(2)	(3)	System Action
Ø	Ø	Ø	Do nothing at all
1	Ø	Ø	Do nothing at all
Ø	1	Ø	Do nothing new; continue processing the tônes(s) represented
1	1	Ø	Do nothing new; continue processing the tones(s) represented*
Ø	Ø	1	Prepare to discontinue the last tone(s) of this address (channel)
1	Ø	1	Stop scan; dump 'ØØØ' into data word; stop tones(s) at this address*
Ø	1	1	Prepare to start (or change tone(s) of this address*
1	1	1	Stop scan; dump bit pattern into* data word; start pitch computations

(1) Address, High or Low; Is this address being scanned now? Yes = 1; No = Ø

(2) $A_0 + A_1 + A_2$; Are any of the keys of this address depressed? The OR of the input is represented here.

(3) $A \neq B$; Has the comparator detected a new input? (Is the past bit pattern of this address different from the present?)

* Principal Status: Start Tone(s)
Continue Tone(s)
Stop Tone(s)

FIGURE 3: Interface States and Functions

the amplitude will be.

2. Since a single finger can normally depress only adjacent switches, the first switch depressed will be interpreted as the center frequency of a band-pass filter which attenuates the frequencies represented by the other Y switches according to a programmable curve.

3. Adjacent switches depressed after the first switch will increase the total amplitude, but not move the center frequency of the filter, unless the lowest frequency switch depressed changes. That is, if $Y_Ø$ is depressed first, the dominant tone produced will be the frequency represented by $Y_Ø$, if $Y_Ø$ remains depressed while Y_1 is lowered, $Y_Ø$ will still predominate, but the amplitude will increase; if, as a next step, $Y_Ø$ is raised and Y_2 depressed -- Y_1 undisturbed -- the frequency represented by Y_1 will now predominate.

4. Amplitude levels will change smoothly, not in discrete steps. Adding or subtracting Y switches from the total number depressed will describe a new steady state reached after a system-specified time period.

It will be necessary for the processor to keep track of the order in which switches are depressed and released. General curves of the type shown in Figure 4 (all derived from the concept of a single dominant tone) will be stored in memory to govern amplitude control.

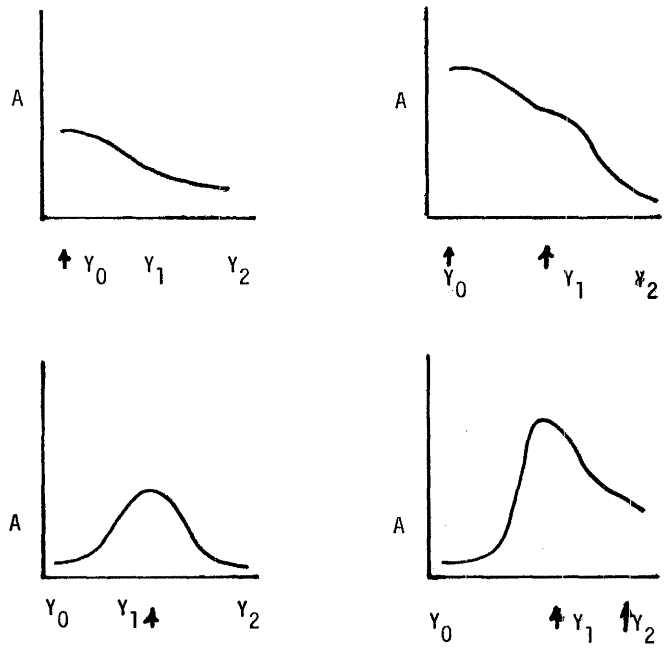


FIGURE 4: Relative Amplitudes (A) for Various Switch Settings

This interpretation of the bit pattern information will provide a wide variety of stable timbres, especially when eight Y switches are available. It will be possible, for example, to stress the fundamental or lowest frequency at any address, to stress the highest overtones (frequencies) at this address or to sweep through the overtones in any order.

Possible Functions

A few examples of data input and sound output will give the reader a better understanding of the keyboard's possible functions:

1. If at address X_1, a single Y key is depressed, e.g., Y_1, the pitch represented by that key will be looked up and converted to sound whose timbre follows the stored waveforms. The other Y pitches at this X address will also be converted, but they will be attenuated according to an amplitude curve similar to the lower left one in Figure 4 (waveforms and amplitude curves are stored in different locations in memory and function differently). Obviously, if three pitches are sounding at X_1, the listener will hear a single waveform containing the frequency components indicated by both the waveforms and the attenuation curve.
2. If Y_2 is also depressed at X_1, the previous pitches will remain as they were, except that the pitch specified by Y_2 (and from its waveform the associated harmonics) will be amplified. The resulting timbre will be brighter.
3. If all Y inputs are depressed, all frequencies referenced will be heard and the resulting sound may be noise-like. This result is considered desirable, since it enables the listener to approach noise-like sounds in an organized fashion by starting with more simple sounds. And the technique for making the transition is simple, since it is directly related to the number of keys depressed.
4. If the Y inputs are touched each in turn at X_1, the resulting sound will approximate that produced by sweeping a filter.
5. If at address X_1, Y_1 is depressed, and at address X_{10}, Y_1 is depressed, two distinct tones will be heard, since they will have common waveforms and equal amplitudes.
6. If at address X_1, all Y's are depressed, and at X_{10} only Y_1 is depressed, the louder sound at X_1 will absorb the higher, quieter sound at X_{10} into a single complex bright tone.

The types of sounds produced are limited only by the combinations of keys that the fingers can reach and in the present design, this represents no limitation. Any timbres that can be expressed as sums of frequency components of specified amplitudes can be at least approximated.

Circuit Theory

For any single channel of the interface (i.e., any X_n address; see Figure 5), an input, indicated by the closing of one or more of the three switches along the Y axis, is inverted and passed both to the D inputs of a Quad Latch (TTL-7475) and to the B inputs of a four-bit Magnitude Comparator (TTL-7485). That is, input $Y_\emptyset$ appears inverted on latch input D_1, delayed on latch output Q_1 and on comparator input B_1. Input Y_2 appears on latch input D_3 delayed on latch output Q_3 and on comparator input B_2.

The Q outputs of the latch are tied to the $A_\emptyset$ through A_2 inputs of the comparator. Since the lateh inputs are enabled only once during each scan and only when the channel to the right of the channel under consideration is being scanned (see Figure 6), any change in the inputs subsequent to the enabling will cause a change in the B inputs of the comparator, but not in the A inputs. This will cause the A = B output to go low (Figure 5).

When this condition occurs, the interface is prepared to indicate a change of state (see Figure 3). When the scan again reaches this channel, the address input will be high, as will the NOT of A = B. The AND of these two signals allows any signals present on the inputs to be passed as high outputs to the data word and on the prototype, to the LED display as well (Figure 5).

If both the address and the inverted comparator output are high, the high resulting from the AND of these tow inputs is inverted and is used to block the clock input. This stops the scan by inhibiting the shifting in the eight-bit Serial-In-Parallel-Out Shift Register (TTL-74164) to be discussed in the next section. As long as the clock is blocked, both the binary address of the channel being examined and the bit pattern on that channel will be displayed.

A RESTART pulse momentarily brings the blocking high to ground and permits counting and shifting to continue until a new input is found.

Once an input is latched, the inputs on that channel will be ignored if there is no further change in them, since the A inputs of the comparator will equal the B inputs. As long as A = B, any further display for this channel will be blocked and the clock will count normally. When an input is present for some time, the interface will recognize two state changes for any channel:

1. A change in the bit pattern for that channel in which case scanning will be stopped, the new bit pattern displayed and passed to the central processor.

2. A removal of all inputs for that channel in which case scanning will be stopped and zeros will be displayed and passed to the central processor.

RESTART will re-enable scanning in all cases.

Scanning of the Keyboard

Scanning is controlled by a TTL-74164 Shift Register whose "H" output is tied to its serial input. Information on the serial input then appears on the "A" output. Clocking the shift register causes the information on A to be shifted into B, from B to C, and so forth. Information on H is shifted into A and the process repeats (see Figure 6). Shift registers are cascaded to provide 256 address selection lines.

The initial value of the first shift register is '1ØØØØØØØ' (binary). As a result, during shifting only one channel (address) will be high at a time, making possible the logic described in the last section.

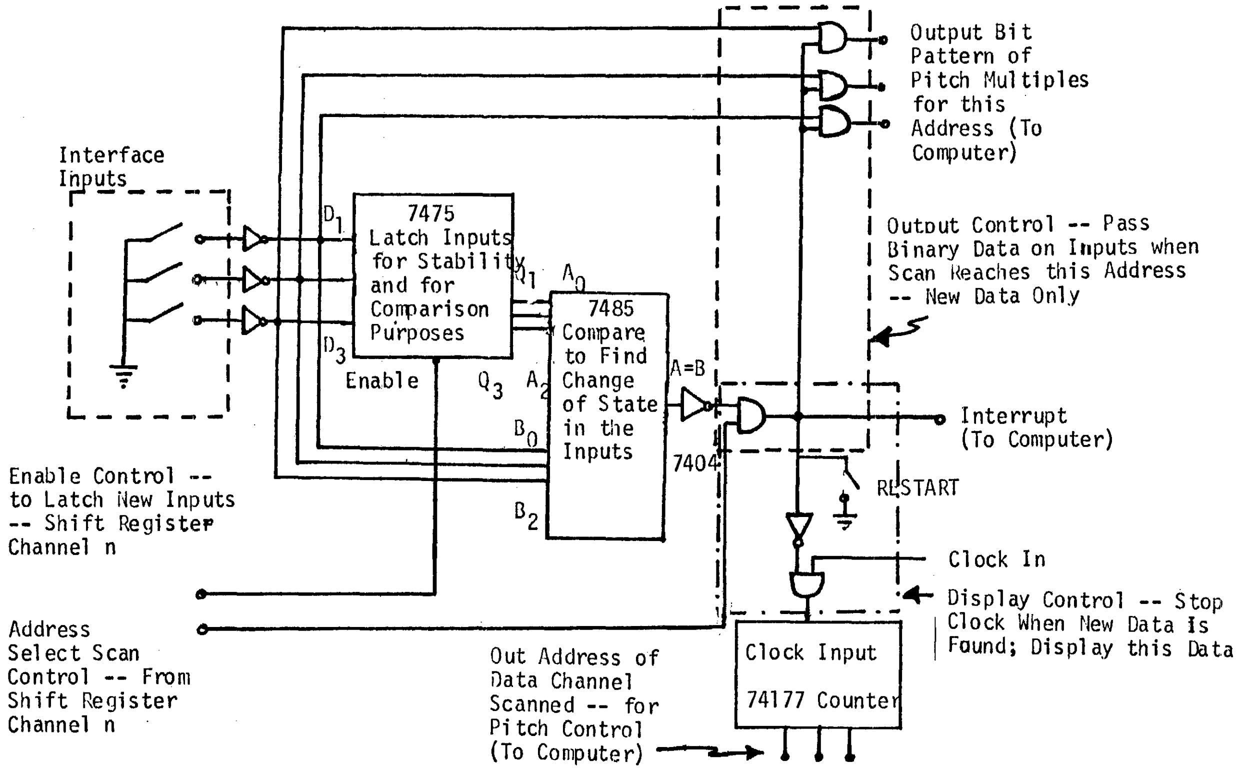

Output Bit Pattern of Pitch Multiples for this Address (To Computer)
Interface Inputs
7475 Latch Inputs for Stability and for Comparison Purposes
D_1
D_3
Q_1
Q_3
Enable
A_0
A_2
B_0
B_2
7485 Compare to Find Change of State in the Inputs
A=B
7404
Output Control -- Pass Binary Data on Inputs when Scan Reaches this Address -- New Data Only
Interrupt (To Computer)
RESTART
Clock In
Display Control -- Stop Clock When New Data Is Found; Display this Data
Enable Control -- to Latch New Inputs -- Shift Register Channel n
Address Select Scan Control -- From Shift Register Channel n
Out Address of Data Channel Scanned -- for Pitch Control (To Computer)
Clock Input
74177 Counter

Figure 5: Basic Logic for a Single Interface Channel (X_1 Address) of the Prototype

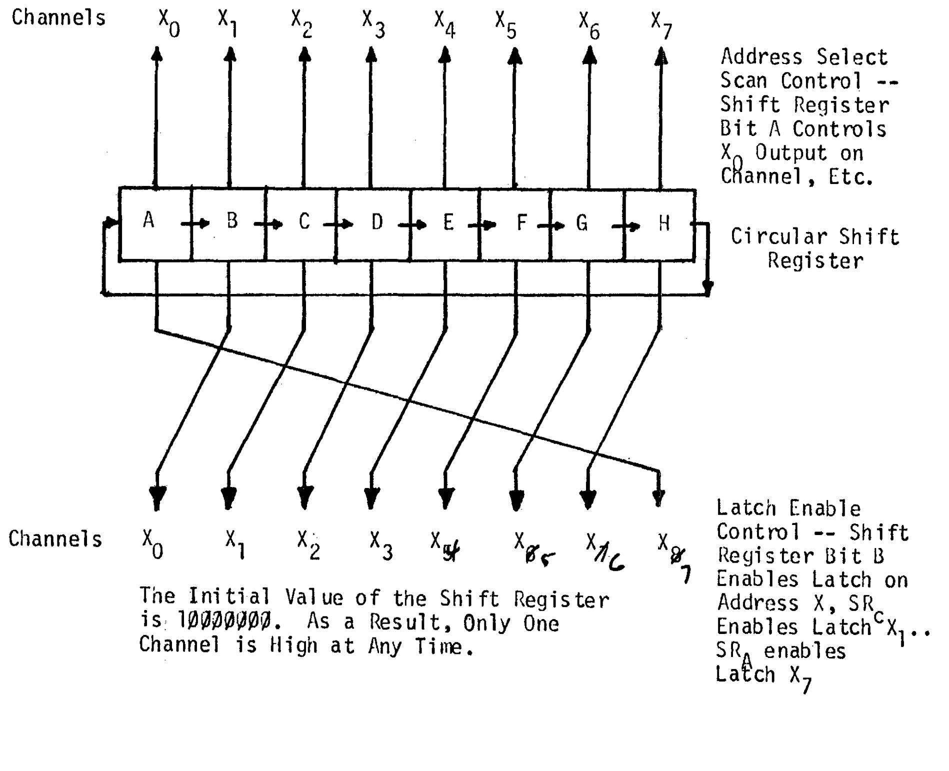

Channels
X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7
Address Select Scan Control -- Shift Register Bit A Controls X_0 Output on Channel, Etc.
A B C D E F G H
Circular Shift Register
Channels
X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7
Latch Enable Control -- Shift Register Bit B Enables Latch on Address X, SR_C Enables Latch X_1.. SR_A enables Latch X_7
The Initial Value of the Shift Register is 10000000. As a Result, Only One Channel is High at Any Time.

Figure 6: Prototype Shift Register Functions

The shift register outputs control two principal functions. Any output controls the display of data on its respective channel. New information at address X_0 is displayed when a 1 is shifted into A. THe same procedure is followed for the other addresses. The shift register outputs also enable the latches of the channel (address) preceding the one being scanned. The latch on channel X_0 is enabled by shift register output B, latch, latch X_1 by output C, and so forth. Latch X_8 is enabled by shift register output A.

Present Project Status and Future Research

At the present, a prototype of the keyboard exists. The D/A conversion hardware has been designed and constructed. Software for computation and output of waveforms has been tested. THe remaining tasks include acquisition and programming of a didicated processor, system integration and testing.

The keyboard is expected to be used for music composition and the investigation of waveforms not found in conventional musical instruments.

Applications

THe applications of a sound keyboard range from electronic music composition to auditory research. The concept of having fingertip control of sound production would allow design engineers to hear research waveforms without constructing circuitry. Psychophysical experiments on hearing would be enhanced by such a device. The interface will also make possible human control of electronic music events in live performance without resorting to magnetic tape and to sequences.

Acknowledgements

The authors would like to express their appreciation to Dr. A. G. Jordan for his encouragement in the research described here.

References

[1]Hubert S. Howe, Electronic Music Synthesis (New York: W. W. NOrton & Co., 1975).

[2]Max Mathews, "Computers and Future Music," Numus-West, No. 6-74.

[3]Sergio Franco, Hardware Design of a Real-Time Musical System, Doctoral Dissertation, University of Illinois, Urbana, Illinois, 1974.

[4]Max Mathews, The Technology of Computer Music (Cambridge, Mass.: M.I.T. Press, 1969).

[5]Heinz von Foerster, James W. Beauchamp, ed., Music by Computers (New York: Wiley & Sons, 1969).

[6]Jonathan Cott, Stockhausen: Conversations with the Computer (New York: Simon and Schuster, 1973).

A MICROPROCESSOR ORIENTED DATA ACQUISITION AND CONTROL SYSTEM FOR POWER SYSTEM CONTROL

by

Michael C. Mulder
Bonneville Power Administration

and

Patrick P. Fasang
University of Portland and
Bonneville Power Administration

Introduction

The advent of microcomputer systems with their inherent cost/performance advantages had precipitated a reassessment of known application areas that heretofore were not considered candidates for digital systems solution. In the power industry there are many applications for microcomputer subsystems for performing data acquisition functions, monitoring changes in the status of the high voltage transmission lines, issuing control commands to open or to close high voltage breakers via relays, and for acting as an intermediate device for storing data or system states. Microprocessor oriented subsystems provide cost/performance improvement over existing subsystems and are easily applied to applications that currently do not use digital systems.

Described in this paper is a MICRO processor oriented Data Acquisition and Control system referred to as MICRODAC which is capable of monitoring high voltage perturbations (via transducers), accepting and issuing control commands, performing format changes and error encoding/decoding, performing system self checks, and transmitting data to supervisory computers. This system performs a very necessary function in power system data acquisition and control at low cost, high reliability, and low power consumptions.

I. System Organization and Operation

Due to the severity of EMI/RFI fields present in most power system field sites, the use of digital systems has been minimal to date. However, with the availability of small, low cost and low power consuming microprocessing systems, combined with special shielding techniques and the improvements in fiber optics, it is now possible to blend these systems and techniques into a solution for such data acquisition and control applications. Figure 1 provides a glimpse of the overall functional system organization that appears effective. Note that two of the three interfaces to the external world are optically isolated and the third (i.e. power supply) is heavily filtered and isolated. The digital computing system is shielded with two forms of shielding; EMI/RFI shielding and capacitive shielding.

The digital system organization of MICRODAC, shown in Figure 2, utilizes the Motorola family of microprocessing modules and consists of an MC6800 microprocessor, a 1 megahertz crystal controlled clock, a 7040 Hz oscillator for asynchronous interfacing, two RAM's of 128 words x 8 bits each, two ACIA's, 14 PIA's, eight INTEL 1702A PROM's of 256 words x 8 bits each, PIA-input/output interface circuitry, A/D converters, and D/A converters. The control program is stored in the PROM's which are easily programmed, and the RAM's are used for temporary storage of data and for MPU stack operations.

MICRODAC responds to control commands and changes in the sensed input data. Upon detecting a change in the input data, the PIA-input/output interface circuit generates an interrupt signal which is sent to the MC6800, which in turn services the interrupt signal by updating tables located in RAM. MICRODAC then notifies the supervising computer of the change and sends the new data to a service center via a fiber-optic communication link. After notifying the supervising computer, MICRODAC returns to surveillance mode. When a control command is sent to MICRODAC from the service center (e.g., system query), MICRODAC executes the issued control function. For both data acquisition and control modes of operation, MICRODAC performs a software self check function.

To provide for operation in a hostile environment MICRODAC is housed in a double shielded containment cabinet. The outer cabinet is constructed with a material which will attenuate EMI and RFI interference. The inner cabinet provides for capacitive shielding.

All required power supplies are derived from an uninterruptible power supply system. This system uses three methods of power conversion. A primary D.C. voltage of +13.5 volts is obtained from a well regulated power supply in the uninterruptible power supply system. During normal conditions, the +13.5 volts is used to charge an external +12 volt battery and to power two DC/DC converters. One converter outputs a +15 V D.C. and a -15 V D.C. The other converter outputs a +5 V D.C. When there is an A.C. power outage, the converters operate from the external +12 V D.C. backup battery. The battery is continuously charged and is ready for service at any time. There is no switching of power source involved when there is an A.C. power outage because the input to the backup battery and the inputs to the two DC/DC converters are both tied to the +13.5 V D.C. line. The -9 V D.C. for operating the PROM's is derived from the -15 V D.C, The backup battery is good for about eight hours.

II. Hardware Considerations

The MC6800 microprocessor system operates solely with memory space. The MPU references all components connected to its bus as memory locations. The address and data bus operate at standard TTL levels. Selection of the various memory or input/output components is done by selectively enabling the appropriate address lines. The MC6871A discrete clock provides the MC6800 with two clock signals, namely Ø1, and Ø2. This clock is crystal controlled and has pulse stretching capabilities via the $\overline{\text{HOLD1}}$ and $\overline{\text{HOLD2}}$ inputs.

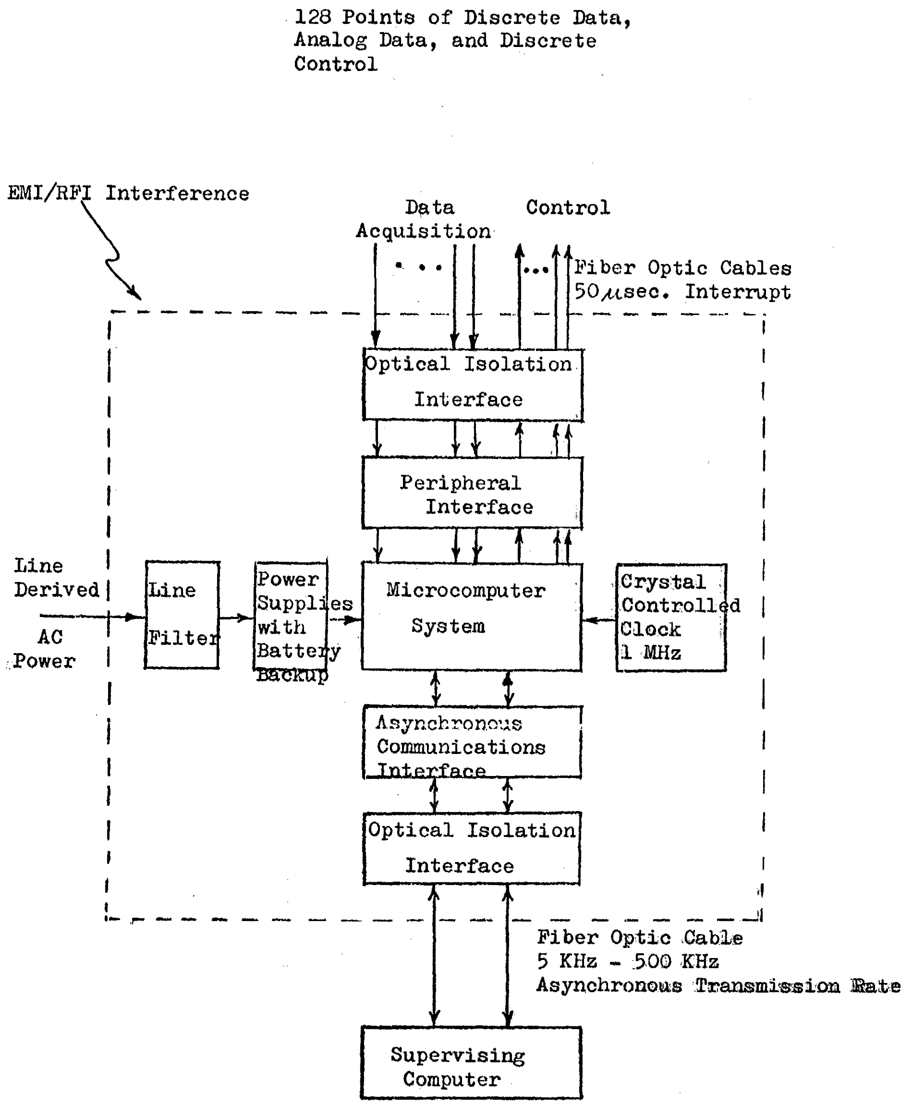

128 Points of Discrete Data, Analog Data, and Discrete Control
EMI/RFI Interference
Data Acquisition
Control
Fiber Optic Cables 50μsec. Interrupt
Optical Isolation Interface
Peripheral Interface
Line Derived AC Power
Line Filter
Power Supplies with Battery Backup
Microcomputer System
Crystal Controlled Clock 1 MHz
Asynchronous Communications Interface
Optical Isolation Interface
Fiber Optic Cable 5 KHz - 500 KHz Asynchronous Transmission Rate
Supervising Computer

Figure 1 - MICRODAC FUNCTIONAL ORGANIZATION

The single-step circuit allows execution of a program stored in PROM one instruction at a time. The address and data bits are displayed on the address and data line LED displays, respectively. Initiating a single instruction execution is done by depressing the single step switch on the front panel while in HALT mode.

MICRODAC has two types of memory, namely RAM's and PROM's. The RAM's are used for temporary storage of data and for MPU stack operation. The PROM's are used for program storage. The eight PROM chips can store up to 2048 words of eight bits each. Each chip holds 256 x 8 bits, and programs can be stored in each chip by selectively setting the appropriate bits after all of the bits have been set to the zero state by exposing the chip to ultra violet light. Once programmed and protected from ultra violet light, the programs reside in PROM with no decay time constant. The procedure for loading a program into a PROM is as follows: first, the assembly language program is translated into an octal listing; the octal listing is then translated into a BNPF listing: the BNPF listing is loaded into the PROM programmer, and the PROM programmer then programs the PROM. This process typically takes 2-4 hours.

Interfacing the MPU to the external world is provided by the Asynchronous Communications Interface Adapter (ACIA) and the Peripheral Interface Adapters (PIA's). The ACIA provides data formatting and control capabilities for interfacing parallel information from the MPU to serial asynchronous data communication devices. The PIA's are used as interfacing buffers between the MPU and external peripheral equipment which supplies data in parallel.

The drive capability of all control signals from the MPU is one standard TTL load and 30 pf. A number of control signals must drive more than one standard TTL load and therefore necessitate the use of signal drivers. All drivers, except the data line drivers, are unidirectional.

The power requirements for MICRODAC are as follows: one +5 volt 3 amp d-c power supply and one -9 volt 1 amp d-c power supply. All Motorola chips require only the +5 volt power supply. The INTEL 1702A PROM chips require both power supplies.

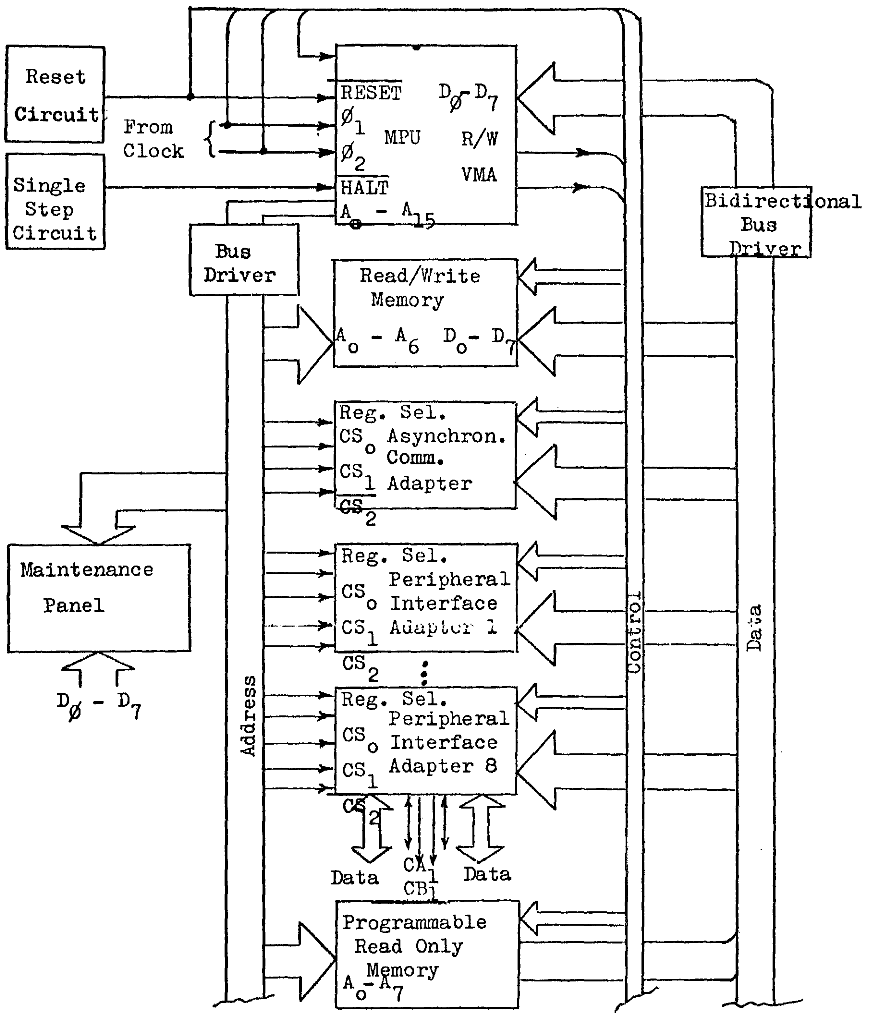


Figure 2 - MICROCOMPUTER SYSTEM

III. System Software Organization

System software for MICRODAC is arranged as shown in Figure 3. When system RESET is initiated, the software branches to a set of initialization routines which set the input/output configuration and issue control commands to all units tied onto the common busses. Table formats and initial values are also set. The software then initiates a software system self check. This is accomplished by exercising the MPU and I/O interfaces with known data and with the MPU inspecting the results. The MPU is checked first, then the PIA interfaces, and lastly the ACIA interface to the supervising computing unit. If all results are as anticipated, the system clears the interrupt bit of the MPU and the system moves to a WAIT state; waiting for a data change to occur or for a command to be issued by the supervising computer. When either occurs, the system polls the PIA's and the ACIA to identify the source of the interrupt. The polling software is organized on priority, with high priority elements serviced first. Multiple interrupts are easily and swiftly serviced. Once the element generating the interrupt is identified, the software branches to the servicing routine to complete the required function. When all interrupts have been serviced, the MPU returns to the self check software; and if the self check routines are successfully executed, the MICRODAC system returns to the WAIT state. Typical times for servicing interrupts are 48 microseconds. The MICRODAC utilizes 1000 bytes of PROM storage. The software organization is modular and is easily extended to allow additional PIA's and ACIA's to be added to the system. A portion of the software is shown in Figure 4.

IV. Fiber-Optic Communication Link

Serving as a "smart" subsystem in a power system, MICRODAC will be working in an electromagnetically hostile environment. As mentioned earlier, MICRODAC is housed in a double shielded containment cabinet to shield against EMI and RFI fields. For the purpose of isolation, low loss fiber optic cables are used as the communication link between MICRODAC and the supervising computer. The length of the fiber optic cable varies with practical limits of 200 meters. The maximum bit rate at which the fiber optic cable will operate is 50 KHz. Operating in the optical domain the fiber-optic link offers such advantages as high bandwidth, no cross-talks, no electromagnetic interference, and high electrical isolation from potential noise source.

The fiber-optic communication link consists of a fiber-optics bundle of 6 fibers, Ga As LED's and driver circuits, and avalanche photodiodes and receiver circuits. The LED's are

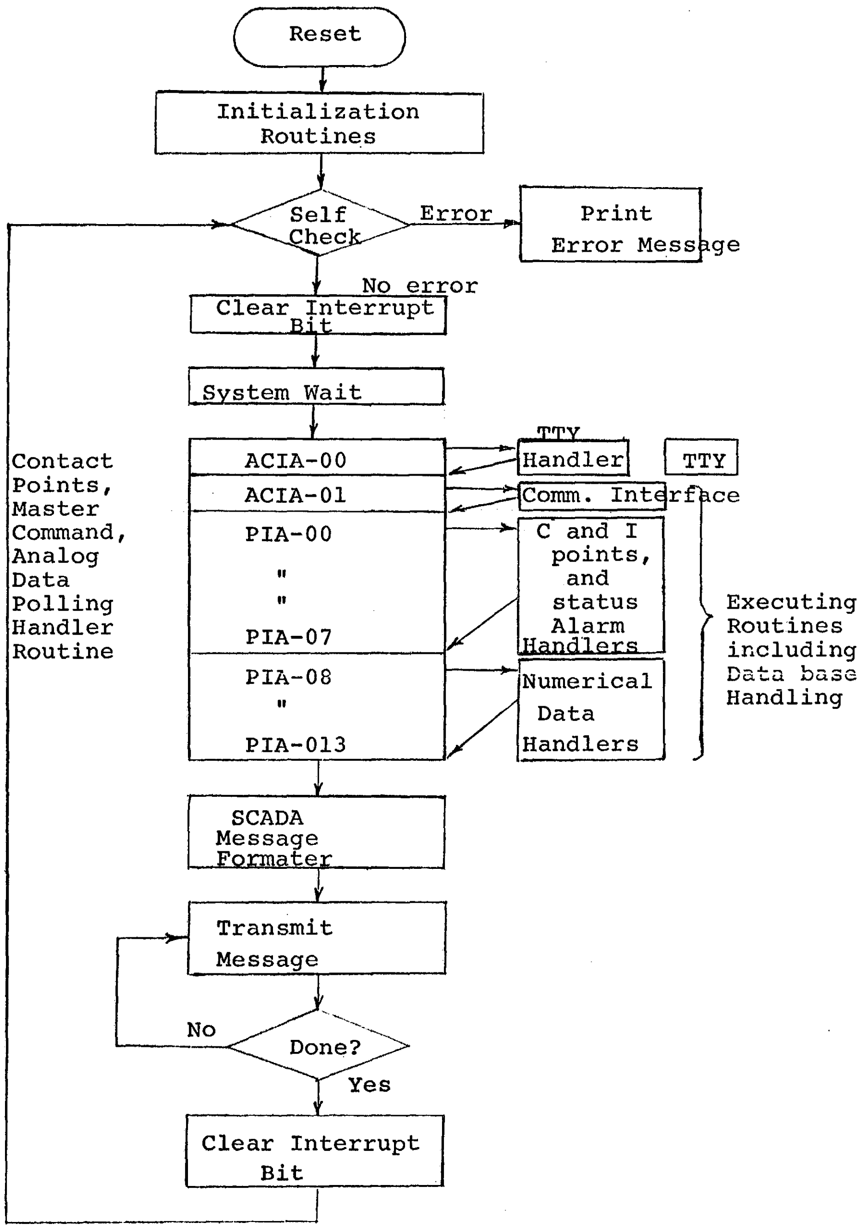


Figure 3 - PROGRAM FLOW CHART FOR MICRODAC

modulated by varying the junction current. The outputs of the LED's are transmitted over the fiber-optic bundle with the photodiode serving as detectors on the receiver end. The system can transmit and receive data rates up to 30 megahertz with an S/N of 500 to 1. Input and output electrical signal connections are made with BNC connectors. The electrical-to-optical signal converter uses the input data to modulate the light emitting diode source. The reconverter at the output uses photodetectors and amplifiers to produce a replica of the input electrical data.

V. Conclusion

MICRODAC has been constructed as a prototype unit to demonstrate the capabilities of microprocessors as applied to power system data acquisition and control problems. The system has performed successfully and has exhibited excellent cost/performance benefits. Software for MICRODAC is currently being written for several applications in which large numbers of units will be used.

Reference

"MC 6800 Microcomputer Reference Manual," Motorola, Inc., Semiconductor Products Div., 5005 E. McDowell Road, Phoenix, Arizona 85008.

```
                  *             ACIA INTERRUPT HANDLING
                  *
4100                     ORG     $4100
4100 B6 2004      POL    LDA A   $2004     LOAD STATUS REG.
4103 2A 31               BPL     POL1      INTPT. PENDING?
4105 B6 2005             LDA A   $2005     LOAD 1ST CHAR.
4108 97 10               STA A   $0010     STORE 1ST CHAR.
410A B6 2004             LDA A   $2004     LOAD STATUS REG.
410D 2A FB               BPL     *-3
410F B6 2005             LDA A   $2005     LOAD 2ND CHAR.
4112 97 11               STA A   $0011     STORE 2ND CHAR.
                  *
                  *  EXECUTE COMMAND 00000010 = ISSUE CONTROL PACKET
                  *  LOCATED IN $0011.  COMMAND 00000100 CAUSES A
                  *  TABLE READ AND SENDING OF DATA PACKET IN $0050,
                  *  $0051 TO THE ACIA.
                  *
4114 96 10               LDA A   $0010     LOAD COMMAND
4116 91 60               CMP A   $0060     COMPARE TO CMD 1
4118 27 04               BEQ     CMD1      IS THIS CMD1?
411A 91 61               CMP A   $0061     COMPARE TO CMD 2
411C 27 07               BEQ     CMD2      IS THIS CMD2?
                  *
                  *  CMD1 ISSUES CONTROL PACKET TO PB.
                  *
411E 96 11        CMD1   LDA A   $0011     LOAD CONTROL PACKET
4120 B7 200A             STA A   $200A     ISSUE PB COMMAND
4123 20 11               BRA     POL1      CONTINUE POLL
4125 96 50        CMD2   LDA A   $0050     LOAD CMD
4127 B7 2005             STA A   $2005     TRANSMIT COMMAND
412A B6 2004             LDA A   $2004     MAKE SURE TDRE IS SET
412D 85 02               BIT A   #$02
412F 27 F9               BEQ     *-5
4131 96 51               LDA A   $0051     LOAD DATA PACKET
4133 B7 2005             STA A   $2005     TRANSMIT COMMAND
                  *
                  *  PIA INTERRUPT HANDLING.  INTERRUPT IS PROCESSED
                  *  BY MPU WITH INPUT DATA PACKET STORED IN $0051.
                  *
4136 B6 2009      POL1   LDA A   $2009     LOAD PA CONTROL REG.
4139 2A 16               BPL     POL2      INTERRPT PENDING?
413B B6 2008             LDA A   $2008     LOAD DATA PACKET
413E 97 51               STA A   $0051     STORE IN TABLE
4140 96 50               LDA A   $0050     LOAD COMMAND
4142 B7 2005             STA A   $2005     TRANSMIT COMMAND
4145 B6 2004             LDA A   $2004     MAKE SURE TDRE IS SET
4148 85 02               BIT A   #$02
414A 27 F9               BEQ     *-5
414C 96 51               LDA A   $0051     LOAD DATA PACKET
414E B7 2005             STA A   $2005     TRANSMIT DATA
4151 3B           POL2   RTI               RETURN FROM INTERRUPT
                  *
                  *  INTERRUPT VECTORS
                  *
47F8                     ORG     $47F8
47F8 4100                FDB     $4100     INTERRUPT REQUEST
```

Figure 4 - A PORTION OF THE SOFTWARE FOR MICRODAC

MULTIPROGRAMMING FOR REAL-TIME APPLICATIONS

H. M. Gladney
and
G. Hochweller[†]

IBM Research Laboratory
San Jose, California 95193

Abstract

A software system supporting multiple event-driven processes concurrently on a small process control computer is described. Each application can be programmed and tested independently. An implementation of the system, called LABS/7, has been productive for over three years.

The limits to which real-time multiprogramming may be pushed are explored.

Introduction

In recent years, dramatic reduction in the cost of logic and memory has made it reasonable to consider applying dedicated, and largely idle, minicomputers to real-time control and data acquisition applications. The conventional wisdom seems to have been that an independent minicomputer should be assigned to each application, and that, for applications generating large amounts of data, this minicomputer should be supported by a large flexible central processor. However, cost reduction of electronic circuitry has not been paralleled by similar reductions in the cost of programming, of computer-computer communication links or of peripheral hardware. For example, King and Carbonaro[1] show that sharing a printer between several processors can be cost-effective and imply that similar arguments applied to other peripheral devices. Wann and Ellis[2] describe a linked system of minicomputers which has as one of its objectives sharing of peripherals. While this paper demonstrates how simple the basic support can be for loosely coupled minicomputers, it is not clear that the extra effort to segment applications is repaid. Jensen's description[3] of a somewhat differently linked system shows that partitioning an application can be a non-trivial exercise. The arguments supporting assignment of a single processor to each application have been that, because of system-wide supervisor overhead, it is very difficult to provide real-time applications adequate response if they compete for a single processor; that errors in one application will impact another; and that distinct applications cannot be independently programmed. This paper describes one way in which these difficulties can be overcome.

Recently we described a system, called LABS/7, which operated on a hierarchy of computers in which each of several satellite processors supported multiple independent applications.[4] The emphasis in that paper was on distribution of function between a centralized host and multiple satellites, and on system features that make application programming easy for inexperienced programmers. In the present paper, we focus on aspects of satellite processor multiprogramming that make a great deal of sharing feasible, and attempt to communicate the limits. To a large extent, the external characteristics of the supervisor and application programs are independent of the hardware architecture and of how function is distributed between the satellite processor and other attached processors, so these factors will be ignored in this paper as much as possible.

Most of the structure of LABS/7 is a straightforward application of methods used previously in process control computers such as the IBM/1800 and general purpose machines such as System/360. The central idea behind the method for real-time multiprogramming is simple. Most of the processing required by many real-time applications has very modest response requirements. This may be exploited by running most of each application at low priority if the system provides for rapid response to critical short segments of code.

System Objectives

The overall objective of LABS/7 was to provide a system which manages the entire set of resources of a set of coupled computers and defines to the user simply and precisely how an application can be designed, including options how to assign the function among the involved processors.

If we assume it possible to reduce overall costs by having several applications share a process-control computer, a general requirement follows--*it is necessary to support multiple independent real-time processes concurrently in such a manner that each application can be designed, programmed, and tested with minimal reference to other applications.*

Programming development is an increasing cost factor. This has been quantitatively explored in recent study of the cost-effectiveness of laboratory automation.[5] In addition, significant improvements can result if programming assistants are unnecessary. To maximize the usefulness of a real-time system to engineers, *each user must be able to write his own application programs, with minimal consultation with support programmers, in a language which provides detailed control of timing, synchronization and input-output to the instrument. This command language should be very easy to learn.*

Control of any single real-time application usually involves several asynchronously running tasks. A control program to manage the system's serially-reusable resources is desirable whether or not the system is multiprogrammed. Such a control program becomes only slightly more complex if independent applications are to be supported. If each application is to be able to access the full capabilities of the hardware, it is impossible to eliminate the need for coordination between users. It must be the function of the control program to minimize the coordination necessary, and to provide methods of resolving contention for each shared resource. A summary statement of the objective is: *The system must include a multitasking supervisor, which permits each application programmer to control the required resources and to synchronize related events. Performance specifications for the system components should be explicit.*

Below there are itemized specifications[6] for software which address these general objectives. This

[†]Present address: Deutsches Elektronen-Synchrotron, Notkestieg 1, 2 Hamburg 52, W. Germany.

list of specifications is not intended to be complete, but to focus on those requirements that are specifically sensor-based, or for which the implementation is strongly influenced by real-time requirements, such as responsive communication between the real-time processor and a large central machine. Real-time control and data acquisition are to be supported by a control program which manages the machine resources and by independently prepared application programs. The requirements include:

(i) A command language for real-time application programs;
(ii) The ability to initiate any application program either from a terminal or from another application program and to pass a short parameter list to the new program;
(iii) Multiple tasks within each application program, and mechanisms for task synchronization;
(iv) Pre-emptive task switching;
(v) The ability to include sections of machine code within an application program;
(vi) The ability to include in an application program a machine language subroutine for asynchronous service of an interrupt signal;
(vii) The ability to record the interval between two sensor inputs to high precision;
(viii) Compactness of the permanently resident portion of the control program and of real-time application programs;
(ix) A relocating program loader;
(x) A program-accessible timer for each task.

Data reduction, storage and reporting functions can be supported on either the real-time processor or an attached central processor. The balance between what is best done on the satellite and what is best on the host will vary for different installations and for different applications. Much can be accomplished with:

(xi) The ability to transmit data to and from the host, to initiate program execution at the host from an application program on the satellite; and to synchronize program events on the host and satellite;
(xii) The ability of a real-time program to call FORTRAN subroutines;
(xiii) Support for several satellite computers on a single host.

To give the user as much flexibility and simplicity as possible in program preparation, the requirements include:

(xiv) Symbolic addresses in application source code, including those of hardware devices and data files;
(xv) The ability to load new programs and to test them concurrently with active applications.

Architecture of Labs/7

The objectives listed above have been realized in a system implemented on the IBM System/7 as the controller and with either IBM System/360 or System/370 as the host installation. Since the features of interest are not particularly hardware dependent, hardware descriptions which are available elsewhere[7] will not be repeated. It is only pertinent that the System/7 has conventional process control computer architecture, with four interrupt levels, registers for each hardware level, and that wide range of peripheral devices, including sensor input-output and teleprocessing ports, disks and printers is available.

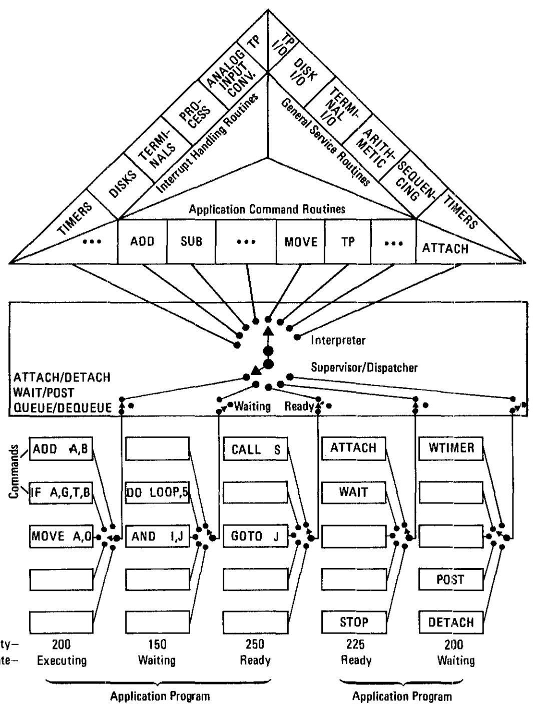
What is LABS/7? It is a combination of functions including:

- a supervisory program for the satellite, with service routines for the sensor interfaces, communication hardware, disks, terminals, and a printer;
- a set of commands, with which the user creates programs which are executed interpretively by the supervisor;
- a host communication program supporting several satellite controllers;
- predefined datasets and command procedures on the host to facilitate program preparation;
- a set of utility programs to initialize the satellite and to transfer programs and data between the satellite and the host.

We will describe the key features of the supervisor and the real-time command language. The program preparation facilities used with LABS/7 are the macro assembler, FORTRAN IV compiler, and link-editor provided as IBM products for the System/7.[8,9] The system is oriented towards program and data storage on a System/7 attached disk, although this is not essential.

LABS/7 Supervisor and Emulator*

The user's view of the satellite supervisor and the emulator is sketched in Figure 1. The elementary unit of work for the supervisor is a command. Commands are combined to form tasks, each of which is assigned a service priority which is used by the supervisor to allocate execution time. An application program consists of one or more related tasks which can share variables.



*We use the word "emulator" for lack of a more accurate word. Normally an emulator is a set of microprograms which create a target machine on a different vehicle machine. An interpreter normally denotes a set of programs which translate high level language to machine language immediately prior to execution. The device used in LABS/7 is a combination of these methods and somewhat closer to the first.

Application program execution is assigned to the lower two hardware priority levels. Although the supervisor does not enforce it, it is intended that a task for which timing precision is irrelevant be given a priority corresponding to the lowest hardware level. The upper two hardware levels are used for servicing completion signals generated by the I/O hardware, and for process interrupt exit routines (see below).

The supervisor manages storage in the System/7 dynamically. The resident supervisor code occupies between 3000 and 6000 words of storage, depending on which optionals such as disk support, printer support and telecommunication support are included and on the size of the sensor I/O interface and the number of terminals supported. The remaining storage is allocated in continguous blocks to application programs. An application program can be assigned any available storage. There is no software limit to the number of tasks or the number of programs executing concurrently.

The supervisor includes an emulator which executes each application program command by analyzing its assembled form and linking to a system-resident routine. Following completion of each command execution, the supervisor processes the next sequential command in the highest priority task that is ready. Since each emulator subroutine is designed to run within 250 machine cycles, the top priority task will be served within about 400 machine cycles of being posted ready. (The latter estimate includes an extremely conservative allowance for supervisor overhead originating from unrelated tasks.) If a task becoming ready has higher hardware priority than the active task, switching occurs without waiting for completion of the current command.

Included within the supervisor is support for bidirectional transmission of data with several options of transmission hardware. This software isolates the user from details of transmission protocols and manages the transmission line on a first-in, first-out basis. Initial program load can occur from disk, across the host communication link or from paper tape. Both direct access and communication support are optional; the system will operate with one or both omitted. Support is included to provide each application with an independent terminal.

LABS/7 Command Language

The command language for application program development is intended to make it easy to write an application program with fairly detailed control of the hardware. The burden of program translation is borne by a standard macro assembler[9] to avoid interpretation overhead when the program is executed. The resulting relocatable load module consists mainly of pointers to emulator routines and operand addresses. This structure minimizes the application program storage requirements. Operands in the source code are either symbolic or explicit, and may be indexed. In particular, data files are referred to by symbolic names which are bound to disk data sets by the program loader. Sensor-based input/output commands refer to specific hardware addresses by symbolic name. This feature makes application programs independent of the machine configuration. Source programs may have source subroutines, which may be nested.

An application program may have more than one task. Each task runs independently, subject to the availability of resources requested from the system and the completion of events for which it explicitly waits, although tasks may communicate with each other by using common storage locations. For synchronization of task execution with other tasks, with external events, or with the completion of some of the slower I/O, the command language supports the definition of symbolic events; it provides for either explicit posting of an event completion or for connecting an event name with an emulator routine that must post completion.

Commands have been designed to be similar in appearance to FORTRAN statements wherever possible, while maintaining the flexibility inherent in assembler language. Many instructions have vector operands with automatic indexing provided. For example, one may add, subtract, multiply, or divide two vectors with a single command. The overhead associated with emulation of each LABS/7 instruction is 25 to 40 machine cycles depending on the type of instruction and the concurrent system activity.

In the current implementation, there are about 70 commands, which may be grouped in 11 categories; some examples are given in Table I. A simple example of an application program is included in the appendix. Because all application program requirements could not be satisfied by emulator routines, the command set has a mechanism to include a "user exit routine", written in System/7 assembler language. This is convenient for including functions which are not used frequently enough to be permanently resident, and for testing new functions.

FORTRAN subroutines may be called by real-time programs, and may themselves call subroutines written in the real-time command set. Most of the facilities of the FORTRAN IV are available.[8]

Timing and Responsiveness

Engineers and scientists frequently over-estimate their data-rate and timing precision requirements when they initially consider automating an application. In this section, we present observations which can be applied to reduce timing constraints.[10] The timing precision and data-rate capabilities of LABS/7 are then summarized followed by an abstract discussion of the mechanisms that permit real-time multiprogramming with good responsiveness.

1) For most sensor-based applications, the precision required of timing control provided by the computing system is less than the intrinsic speed of the computer. In many data acquisition applications, time is not even an explicit variable, so that as long as reasonable throughput is maintained, data input can be controlled by a low priority program. For data acquisition applications which are time-dependent, it is most often necessary to know when the data was taken, not to control data acquisition to precise intervals.

2) A process control computer is capable of collecting a million numbers a minute or 10^9 data per day. This data rate is far in excess of what is usually required, even for a demanding set of applications. Scheduling of the work, which can be done by the system itself, can alleviate peak loads.

3) Inexperienced users tend to want to collect too much data by an order-of-magnitude or more. Every data point collected must be either processed or wasted.

4) Missing the collection of a few points in a file of physical measurements may be unimportant if it can be recognized that this has occurred. Only if a critical measurement cannot be redone in time does loss of data become serious.

In most applications, a very high data rate needs to be sustained for only for a very short interval. Since the start of a high-speed process can be synchronized by the computer, it is very easy to devise strategies which avoid conflict between incompatible high-speed runs. For the few applications which require high rates for sustained periods (or extremely accurate timing) inexpensive solutions can be designed with integrated circuit chips built into a processor interface. Two examples, one from a laboratory and one from a quality control test illustrate the point. In mass spectroscopy of a gas chromatographic effluent, scans of 5000 points every half-second may be indicated; however, only scans corresponding to a chromatographic peak need to be gated into the controller, and only the data significantly greater than the noise level are meaningful; in a typical 15-minute run of 10^7 points, interface electronics can reject as uninteresting all but 10^5 points. In fatigue testing of jet aircraft wings, it may be necessary to process 50,000 strain measurements per second for hundreds of hours in order to record the details of a fracture when it finally occurs; one could use a microprocessor which passes data to the control processor only when the strain is changing rapidly, and at that time usurps control of the processor.

In LABS/7, since the objective was to give the user all possible freedom, some responsibility for sensible use of the resources was also transferred by providing guidelines which have been successful at avoiding timing conflicts between applications. The responsiveness of LABS/7 running on a System/7 can be summarized as follows. The next command of the top priority task will be executed within 200 μsec. of the time the task is ready to execute. For situations in which the interval between two occurences must be either very short or controlled to a close tolerance, a user exit routine mechanism can be programmed. For applications in which the response to an external signal must occur within 20 μsec., there is a process interrupt exit routine mechanism. Sustained data rates of 1000 points per second for multiple concurrent applications are possible with the normal capabilities of the LABS/7 emulator, as is an aggregate data rate of 20,000 points per second. Bursts of digital input or output data at over 100,000 points per second may be achieved using a process interrupt exit routine. As far as we know, these response times are adequate for the several hundred applications which are currently supported by LABS/7 in different locations. The mechanisms permitting application programs to include segments of machine code are seldom used.

Discussion

It is clearly not possible to arrange that a single processor serves multiple independent applications with guaranteed response times for each application irrespective of competing activity. Complete independence of competing applications is not generally possible unless all applications (except perhaps one) are slow relative to the processor. However, it is easy to arrange that the bulk of each application can be executed at relatively low priority and that task-switching is available very frequently and with low overhead. If the application programming system is such that critical portions of code are clearly identified small segments, and if the performance of elementary service mechanisms is precisely specified, it is possible to adjust the priorities of application tasks and interrupt servicing routines so that most requirements are met within acceptable engineering tolerances, and so that those portions not satisfied are clearly identified simple segments which might be candidates for service by an attached microprocessor. In our experience, adjusting priorities for a set of applications has always been very easy to do.

In the LABS/7 implementation, attainment of the achievable limits of performance was not emphasized, partly because none of the users of the system have requested performance improvements. However, it is worthwhile to estimate what limits are possible and what mechanisms might be made available so that application programs can access most of the speed of the hardware in a high level multitasked-multiprogrammed system. To a considerable extent, this discussion can be independent of details of the hardware architecture.

The supervisory program for a real-time system can be quite conventional--it should provide for enqueuing on serially reusable resources, creating and synchronizing tasks, waiting for events and responding to interruption requests. (In this discussion it is assumed that the processor has three or more hardware priority interrupt levels; a key parameter is the time required to switch from a lower to a higher priority level and to detect which of several possible interrupt signals was received.) If the processor includes time-sensitive devices, such as disks or telecommunication lines, they should be assigned to the second highest hardware interrupt level so that they cannot interfere with servicing the highest level. Over-runs on such devices can be handled with normal error mechanisms.

The majority of each application program is to be executed on the lowest processor level. If the system provides a mechanism by which control will be regained by a supervisory program periodically (e.g., every 250 memory cycles), it is possible to provide a large number of software priorities for multi-tasking. One very effective way to do this is with an emulator, which is executed without the supervisor ever giving up control. The LABS/7 supervisor switches control to the ready task of highest priority at the end of every emulator subroutine, whenever an I/O wait is necessary and in each pass through the loop of a data-dependent emulator subroutine. Such an emulator has other very desirable features: application programs are extremely compact, the application programming language is easily extended either with new emulator subroutines or with assembler macro commands whose individual instructions are emulator commands or other assembler macro commands; it is possible to enhance the emulator functions without reassembling application programs; adverse interactions between application programs are largely avoided by the isolation the emulator provides; and nearly all of the overhead of language translation occurs when an application is assembled. (Another alternative is the type of interpreter described by Freeman,[11] which should be further examined.)

Such an emulator is best exploited by running those program segments which do not have important timing requirements (see above) as the lowest priority tasks. Generally the sensitive program segments can be written as quite short medium priority tasks (10 to 30 emulator instructions corresponding to 2000 to 6000 memory cycles), with the effect that when one of these is encountered, there is very low probability of unacceptable delays because of competing processes. How to segment a program into separate tasks is often suggested by the application. Since data are shared by tasks within a program, segmentation does not materially complicate programming. If this type segmentation into tasks is not desired, a similar result can be achieved with a "change priority" instruction included in the emulator set.

For data collection tasks in which timing precision is important, the emulator can include a sensor input subroutine that includes in its results a time stamp. If the implementation is--mask interrupts; read internal clock; start sensor I/O read; unmask interrupts; wait for sensor I/O completion--the time interval between successive data reads will be precise to about 10 memory cycles (depending on functional details of the hardware). This interrupt masking will introduce only negligible disturbance to other functions of the system. With well-known interpolation methods, it is trivial to reconstruct data tables on equal time intervals if this is desired.[10]

All function described above is available without including machine code in the application program. If it is necessary that two external events of a single application occur within a small number of machine cycles of each other, a user exit routine, as described for the LABS/7 implementation, may be employed. If the machine language code of the subroutine were constructed similar to that described in the preceding paragraph, the minimal interval between the events would be determined by hardware limitations.

It is possible to provide for very fast response to interruption requests with an emulator subroutine which inserts into the interrupt decoder a branch to a process interrupt exit subroutine which is a part of an application program (there are many ways to do this). If no competition is active, the application subroutine can be entered within a few machine cycles of the time the hardware recognizes which of several interrupts occurred. If this device is used at the highest interrupt priority, the only interference to an application using it will be from other use of the same device, from interrupt masking as described immediately above and from interrupt masking required in the multitasking supervisor. In practice, the last source of interference is negligible compared to the former two. To understand these interferences, an example is helpful. Suppose there is a system restriction that process interrupt exit subroutines be limited to 25 machine cycles and that no user exit subroutine should mask interrupts for longer than 25 cycles. Suppose further that there is an average interval of 2500 machine cycles between such events and that their occurrences are uncorrelated, then the probability that entry of a process interrupt exit subroutine be delayed by 25 machine cycles is less than 1%, and the probability that the delay is 50 machine cycles or longer is less than .01%. If such analysis reveals that the possible peak interference is unacceptable, the system can include a mechanism to enqueue for exclusive service on the process interrupt exit routine function. Perhaps because of the application characteristics described above in the section on timing and responsiveness, we have not felt it necessary to include such enqueuing in the LABS/7 implementation.

Conclusions

It has been frequently assumed that multiprogramming a small computer for real-time applications is not viable for reasons of operating system complexity, supervisor overhead, or timing interferences between applications. In this paper we have described a counterexample and abstracted the mechanisms on which it is based. All of these mechanisms are individually well-known, and such a system could be implemented within almost any hardware architecture.

An implementation, called LABS/7, exists and has the following characteristics:

- The entire operating system, including disk, telecommunications and terminal support and an emulator requires about 6000 6-bit words of memory.
- Supervisor overhead, relative to applications coded in machine language, is less than 20%.
- Applications can largely be programmed independently. The areas of timing contention are well-defined and therefore easily resolved.
- Modifications and additions to the application language are easy to make.
- And support exists for attachment of several real-time processors to a host processor.

LABS/7 has been in productive service since 1973 so that the basic ideas presented have been tested and found to be useful.

Appendix - An Application Example

A short example is given below to demonstrate the simplicity of the language for the data acquisition portion of an application.

When a start signal triggers process interrupt PI1, 100 digital readings are to be taken from a scanning device, DI1. Each reading must be preceeded by the setting of digital latch DO1 to initiate a digital readout. Because a single reading is subject to noise, it is necessary to repeat the scan of 100 readings 50 times and average the results. The following LABS/7 statements illustrate how this may be accomplished:

```
     .
     .
     WAIT     PI1               WAIT FOR START SIGNAL
     DO       AVG,50            BEGIN AVERAGING LOOP
     DO       SCAN,100          BEGIN DIGITAL SCAN LOOP
     SBIO     DO1               SET DIGITAL LATCH
     SBIO     DI1,BUFR,INDEX    READ INTO INDEXED BUFFER
SCAN CONTINUE                   END OF SCANNING LOOP
*
* ADD 100 READINGS INTO DOUBLE PRECISION BUFFER
     ADD      AVG,BUFR,100,PREC=D
     MOVE     I,0               RESET READ BUFFER INDEX
AVG  CONTINUE                   END OF AVERAGING LOOP
*
* DIVIDE DATA FOR 100 PTS. BY 50. STORE RESULT IN BUFR
     DIVIDE   AVG,50,100,BUFR,PREC=D
     .
     .
BUFR BUFFER   100,INDEX=I
AVG  BUFFER   200
     .
     .
```

These data might be processed on the System/7 to produce a report or be sent to a host computer. In order to send the data to a System/370 host, the following simple addition opens System/370 data set named 'SYS7.TESTDATA' for output, writes data stored in BUFR to the host, and closes the data set.

```
        TP     OPENOUT,DSNAME   OPEN HOST DATA SET
        TP     WRITE,BUFR       WRITE DATA TO HOST
        TP     CLOSE            CLOSE HOST DATA SET
        .
        .
DSNAME  TEXT   'SYS7.TESTDATA'  NAME OF HOST DATA SET
```

Acknowledgements

We are deeply indebted to R. W. Martin and D. L. Raimondi for permission to use the material based on LABS/7 and for detailed debates of many items raised in this paper. Ms. J. Valdillez' help with the manuscript is gratefully acknowledged.

References

1. W. F. King and F. Carbonaro, "Output Device Sharing by Minicomputers," Proceedings of 2nd Annual Symposium on Computer Architecture, IEEE, p.141, December 1974.

2. D. F. Wann and R. A. Ellis, "Conjoined Computer Systems: An Architecture for Laboratory Data Processing and Instrument Control," Proceedings of 2nd Annual Symposium on Computer Architecture, IEEE, p.170, December 1974.

3. E. D. Jensen, "A Distributed Function Computer for Real-Time Computer," Proceedings of 2nd Annual Symposium on Computer Architecture, IEEE, p.176, December 1975.

4. G. Hochweller, H. M. Gladney, R. W. Martin, D. L. Raimondi, and L. L. Spencer, "LABS/7 - A Distributed Real-Time Operating System," IBM Systems Journal. Vol. 15 (in press). R. W. Martin, D. L. Raimondi, and L. L. Spencer, "LABS/7 - Laboratory Automation Basic Supervisor for the IBM System/7 - Application User's Guide," IBM Research Report RJ1501, January 1975.

5. R. H. Kay, H. D. Plotzeneder, R. J. Gritter, "Cost Effectiveness of Computerized Laboratory Automation," Proc. IEEE 63, 1495 (1975).

6. See also, H. F. Pike, "Future Trends in Software Development for Real-Time Industrial Automation," SJCC (1972).

7. "IBM System/7, System Summary; System Hardware and Software Overview," IBM Systems Reference Library GA34-002.

8. "IBM System/7 FORTRAN IV Language," IBM Systems Reference Library GC28-6876; "Combining FORTRAN IV and MSP/7: A Programming Guide," IBM Systems Reference Library SC34-0025.

9. "IBM System/7 Host Program Preparation Facilities on System/360 or System/370," IBM Systems Reference Library GC34-0018.

10. A specific example is given by H. M. Gladney, B. F. Dowden, and J. D. Swalen, Anal. Chem. 41, 883 (1969).

11. M. Freeman, "An Instruction Class for an Extensible Interpreter," Proceedings 2nd Annual Symposium on Computer Architecture, IEEE, p.195, December 1974.

TABLE I

EXAMPLES OF REAL-TIME COMMANDS

Category	Command Name	Function
System Configuration	SYSTEM7	Defines the size and I/O configuration of the S/7 hardware
	IODEF	Relates a symbolic sensor I/O address to S/7 hardware addresses
Task Control	ATTACH	Defines and starts a new task within a program
	WAIT	Waits for completion of a named event
	QUEUE	Enqueues on a named (serially reusable) resource, such as the printer
Program Flow	CALLFORT	Calls a FORTRAN subroutine, and passes parameters
	GOTO	Unconditional or calculated branch
Timing Control	INTIME	Returns the time elapsed since the last execution of INTIME, precise to 1 millisecond
	WTIMER	Waits until a previously set time interval expires
Data Definition	BUFFER	Defines a buffer and a pointer which is automatically indexed when certain I/O commands transmit to or from the buffer
Data Manipulation	ADD	Adds a single precision constant or vector to a single or double precision constant or vector
	CONVERT	Translates ASCII characters into EBCDIC characters or vice versa
	ADDINEX	Increments the contents of an index
Teleprocessing	TP OPENOUT	Initializes communication into a host processor dataset
	TP SUBMIT	Submits a job into the host processor jobstream
Terminal Support	YESNO	Transmits a query to a terminal, and branches if the answer is not yes
	WRITE	Enqueues for service a table of terminal output commands
Sensor I/O	SBIO AOx	Sets the voltage on analog output converter "x"
	SBIO DIy	Reads the condition of digital input word "y"

BASIL ARCHITECTURE - AN HLL MINICOMPUTER

Theodore H. Kehl
Departments of Computer Science and Physiology/Biophysics
University of Washington
Seattle, Washington 98195

Introduction

During the past several years a computer-aided design system (the Logic Machine) has been under development in our laboratory. Briefly, the Logic Machine consists of a microprogrammable control processor, one or more functional units, one or more bidirectional buses, and a microprogram all arranged to perform a specific digital algorithm. Our major goal has been to be able to construct any digital device with this system. We have been able to build a graphics display terminal (1), a floating point processor (2), string/array auxiliary processor (3), and a minicomputer (4). It has amazed us to see how simple and fruitful it has been to construct these devices. In this paper we describe the use of the Logic Machine design system to build still another digital device, a high-level language minicomputer.

The motivation for this effort is probably obvious to all hardware designers; software is the most expensive part of a computer system and a high-level language computer will significantly reduce software costs. Not so obvious is the task simplification at the systems programming level, enabling a programmer to quickly review his work, decide on additions or corrections, and expand a program or system. These benefits reduce the layers of logic and, we are convinced, will enable much more sophisticated software systems to be built. That is, machine languages are too primitive and, eventually, a programmer cannot keep track of all the facets of a system.

Another way to reduce system complexity is to reduce the size of the operating system. Clearly, a multiuser system requires a large operating system; the construction of a "private, personal" computer for single users would considerably reduce operating system size. LSI memories and bit-sliced microprocessors make this feasible. Of course personal, private computers are not a panacea; on the other hand, there are situations in which such systems are preferable (even mandatory) to multiuser systems. This is so in our laboratory, where the emphasis is on physiological data acquisition, experimental control, and simulation. With LSI microprocessors and semiconductor memories used in conjunction with our own design and construction facilities, it has been feasible to construct our own computers and the necessary operating systems.

Still another way to simplify a system is to limit the scope of intended applications. Admittedly our orientation is "warped" by the physiological research setting in which we are located. Later, for example, we will show how our need for vector arithmetic hardware has influenced our compilation techniques, that is, in character string processing. Vector/array machines were motivated by large scientific problems whereas byte machines were the result of the business requirements. The former are of little use in business applications, just as the latter are of little use in scientific applications.

Supported by a Grant from the National Institutes of Health, Division of Research Resources, Biotechnology Research Branch, RR00374.

These considerations led us to design (1) a high-level language minicomputer, (2) to be used in a "private, personal" fashion by, (3) physiologists/biophysicists. We call this system BASIL (BASIC - SIL, for System Implementation Language). The BASIL language is a version of BASIC enriched for systems programming. Although intended as a single-language computer the possibility of additional languages is kept open. Whether or not BASIL is useful to other scientists in their research disciplines can only be answered by those scientists; the complexity of modern science prevents us from making sweeping statements. We shall restrict our comments to the computer needs of physiologists/biophysicists.

These are as follows:

1) An aggregate data acquisition rate of 100 KHz A/D samples is usually adequate if the system has fast response time.

2) Digital output control and display feedback from the computer to the experiment (and experimenter) are required.

3) Floating-point arithmetic - vector and scalar - much "number-crunching" with a strong vector emphasis are required.

4) Peripheral high-speed and low-speed storage are needed. Often a single day's experiment will fill most of a reel of magnetic tape. For storage of intermediate results a modest-sized (4M words) disk is sufficient.

5) A main memory of 32 K words (16-bits/word) is quite adequate for physiological/biophysical research.

Perhaps a more useful list is one that describes nonessential features.

1) Multiuser operating system. Item 1 above prevents, in any practical way, sharing of a computer by experimenters. Although it may be technically possible to share, the operating system cost (to construct and maintain the software and provide memory) is so large as to make the single-user uniprocessor system a preferred alternative.

2) Multiple languages. An interactive algebraic language is sufficient for most physiological/biophysical research. All programming, including system programming, is to be done in this language. A single-language computer would be of significant benefit in an enviroment in which we expect to have no professional programmers. Even with professionals system modifications and developments are improved when a high-level language is used.

3) Line printers and card readers. With a single-user, single-language system a highly interactive, source-oriented text editor is quite easy to implement. Program development and listings on the scope are preferred over the card reader/line printer. A 60 char/sec printer/plotter is sufficient for both programming listings, manuscript text, and hard-copy graphical output.

As far as we know there are no minicomputers that satisfy all of these requirements. Nor are there likely to be any. Jordan Baruch (5) describes how the specialization in biomedical research has "disaggregated" the marketplace, making it unprofitable for vendors to pursue highly specialized subsegments. This is easily seen by the above comparison; the list of things not needed by the physiologist/biophysicist are precisely those required by the physician/hospital records system.

A global issue is how to "reaggregate" the biomedical community. Baruch recommends that scientists in subspecialities "standardize and specify out to some operating boundary" computer systems for that subspeciality. A high-level language minicomputer would be of great value in meeting Baruch's recommendation.

Microprogrammed Sequenced Functional Units

BASIL belongs to the class of digital devices we call Logic Machines. That is, several function units are connected together over one or more bidirectional data buses and are sequenced by microcommands emanating from a vertically encoded microprogram engine (control processor). The choice of vertical encoding and bus-access discipline are explained in a paper describing the LM^2, another logic machine minicomputer previously constructed in our laboratory. (4) BASIL, block-diagrammed in Fig. 1, uses the LM^2 functional units for main memory, peripheral I/O, cooperative processing. Here our attention will be restricted to the ALU-addressing function unit, where most of operations associated with a CPU are performed.

Second-generation microprocessor LSI integrated circuits make up nearly all of the data path of BASIL. Four Monolithic Memories (6701) Schottky bipolar bit slices, each a 4-bit slice (see Fig. 2) capable of 205 ns microinstructions, form the data part. A-source and B-source registers select 2 of the 16 general registers for input to the ALU. A μ-instruction register selects one of eight functions to be performed on a variety of sources and destinations.

The macroinstruction format is shown in Fig. 3 and is composed of three fields: ALU control field, microroutine branch field, and two flag bits. The flag bits are used as general-purpose modifiers and do not have a rigid definition.

Macroinstructions require a 16-bit word. Operand addresses follow the instruction and each operand address requires a 16-bit word. Thus, for example, $A + B \longrightarrow C$ requires four words of storage:

"ADD"

ADDRESS OF A

ADDRESS OF B

ADDRESS OF C

Execution of the macroinstruction proceeds as follows (refer to Fig. 4, a listing of the BINOP macroroutine). A macroinstruction program counter (MacroP) is bussed to the memory address register (MAR). If the memory is busy an interlock holds up processing until not busy, at which time a read cycle is started. Just after the read cycle starts, MacroP is incremented. These actions are performed by a single microinstruction - MicroPRDINC.

MDRMACRO moves the contents of the memory data register to the macroinstruction register. This microinstruction is also interlocked to memory access complete to hold up processing until the data are valid. Also the μ-instruction register in initialized to transfer inputs to A-ram. A-source and B-source are set to 0 and 1, respectively.

"DECODE," a control processor microinstruction, allows peripheral device functional units to gain control of the control processor. See [4] for a complete description of cooperative processing wherein the control processor executes microinstructions for peripheral devices.

The six-bit control processor microroutine address portion of the macro is applied to the address inputs of the macro branch table to produce a 12-bit branch address. This branch table has lowest priority relative to other branch tables located in the peripheral device functional units. Consequently, if none of the other functional units require service, DECODE gates an address from the macro branch table to the control processor. This address causes the control processor to start execution of the appropriate microroutine required for the macroinstruction.

MDRDIS transmits the operands from the memory data register to the 6701 bit slices. Steering for the first operand to Ao was initialized during macro fetch by micro MDRMACRO. That steering is changed for the second operand by incrementation of the 6701 ALU microinstruction register which is, in fact, a counter.

With the two operands in register 0 and 1, the microinstruction MacroREGLD transfers the ALU field from the macro register to the 6701 microinstruction register.

Finally, the address of the destination is acquired. CPMACRO changes the ALU field of the macro register to steer from 6701 to output; the result is transmitted to the memory, and the microroutine closes by jumping back to macro fetch.

Given in Fig.4 is the timing of macro fetch and BINOP. The timing assumes TI's TMS 4030 with 300 ns access, 500 ns full cycle). BINOP would execute in 2 μsec if all operands were stored in a 100-ns memory (such as cache) but require more than twice as long (5.1 microseconds) when stored in this memory. BASIL is actually faster than the worst case described in Fig.4 because main memory is 4-way interleaved.

Performance

A conventional computer would require a minimum of three instructions to perform $A + B \longrightarrow C$:

```
LD    A
ADD   B
ST    C
```

If we assume a 500-ns memory, the conventional machine would require 3 μsec: 3 instruction fetches, 2 operand fetches (for A and B) and a store (C). BASIL's worst case timing is 5.1 for the same operation (BINOP). We are willing to accept this two-fold speed reduction considering the ease with which the macrocode is produced. Realistically, the over-all machine performance of BASIL is much better for the following reasons.

On the one hand, if we assumed a 5-bit operation code, a 16-bit macro can only address 2048 words directly. This addressing space is unacceptably small and most minicomputers have page registers to increase the addressing space. Control of the page register adds to the compiler complexity and consequently is often not used. Alternatively, double word instructions are used and, while simplifying compiler construction, such a machine is only about as fast as BASIL - two memory cycles are required for each macro fetch, and three for operand manipulation. (In point of fact, high-level languages for minicomputers often produce far less efficient code. This is because a compiler writer will often first design a group of macros and the object time code consists of calls to these macros. Significant overhead is introduced in the calling operation; RT-11 Fortran for the PDP-11/45, a macro-based compiler, does $A + B \longrightarrow C$ in 12 μsec. More sophisticated compilers, such as Fortran-4 PLUS, produce better object time code but with much slower compilation and much larger memory requirements.)

Furthermore, BINOP is a scalar operation: the worst case for BASIL and the best case for a conventional machine. Shown in Fig. 5 are all of the macros necessary for the compilation of BASIL. (Notably missing are the floating point arithmetic operations, which are done in a separate auxiliary string/array processor [3]). Many of these macros manipulate strings of characters, usually at memory bandwidth speeds. Since much of compilation involves character string operations, BASIL compiles at quite high speed. Unfortunately, it is nearly impossible to give an estimate of BASIL compilation speed relative to that of a conventional machine. For the most part BASIL seems to be at least twice as fast.

But speed is not really an important issue because, in the environment for which BASIL is intended (i.e., a physiologist/biophysicist's laboratory), compilation speed does not have highest priority. Of much greater importance is the ease with which external command-and-control functions can be incorporated in the software. Our goal is to encourage the basic scientist to develop his own command-and-control systems. With conventional computers this is an onerous task involving command language decoding, that is to say, lexical analysis.

BASIL's lexical analysis commands

Lexical analysis consists of separating the parts of a line of source code into its constituent parts. Because a conventional computer can only manipulate a single character at a time, lexical analysis tends to be slow and cumbersome. Our approach is to perform firmware character string operations. This has the advantage of increasing lexical analysis speed and decreasing the complexity of the software. In addition, character string commands are useful as fundamental operators in all software systems using text: text editors, information retrieval systems, type setting systems, etc., and thus has utility for run-time systems as well.

To illustrate the use of the firmware commands consider the lexical analysis of:

```
BETA = ALPHA .NOT. (GAMMA .OR. DELTA)
```

This line must be searched with a series of substrings taken from a table of operators. For example, the substring ".AND." would be passed over the line of source code and, in this example, a match would not be found. The command to do this operation would be:

```
CO=INDEX (LINE, OPER, 1)
```

where "LINE" is the source line, "OPER" is current operator currently being considered, "L"indicates the search to begin at the first character in "LINE." If the search is successful the index of "LINE" matching the first "." of ".AND." will be stored in "CO"; otherwise, CO = 0. In memory the sequential words for this instruction would be:

```
"INDEX"

ADDRESS OF LINE

ADDRESS OF OPER

ADDRESS OF CO
```

and execution proceeds at nearly memory speed.

As each operator is located it is moved (by instruction MOV) into an array of strings, and the operator substring of "LINE" is set to blanks. Thus, at the end of the operator scan procedure, the decimated line of the example is:

```
BETA   ALPHA   GAMMA   DELTA
```

The remaining substrings of characters, i.e., the symbols, are then transferred to the string array. Lexical analysis is thus completed.

Vector operations, I/O service and micro-interrupts

BASIL depends on vector-like operations, embedded in firmware, to speed up compilation. Microcode for vector-like operations is written to utilize as much of memory band width as possible; nevertheless, BASIL is memory band width limited. However, I/O must be supported at some minimum word rate. BASIL's disk drive, for example, must handle a word every 6.4 μsecs. BASIL's vector hardware, for maximum efficiency, performs microsteps in tight sequences such as transferring the contents of the MDR directly to the MAR (Fig.4). If the microcode were interrupted and the MAR altered, as it would certainly be to satisfy I/O, just after the MDR to MAR transfer, the indirect address would be lost. Hence, BASIL microcode must control the I/O requests and permit them to occur only at "safe" times. A conventional interrupt scheme will not suffice because it cannot recognize "safe" times.

A DECODE/PASS microinstruction permits I/O activity when no volatile information can be lost. Execution of DECODE/PASS is as follows: 1. μPC is stored in a DECODE/PASS register, 2. Peripheral device request priority network is checked, 3. If no peripheral devices are requesting service, the control processor passes and the next mainline microinstruction is executed, 4. However, if a peripheral device requires service the decode option is performed (the control processor accepts a peripheral device functional unit generated microroutine branch address resulting in a branch to a service microroutine), 5. The DECODE option arms the DECODE/PASS register so that, when the I/O service microroutine (which always ends with a DECODE) is completed a branch back to the location following the DECODE/PASS micro is executed. DECODE/PASS micros are sprinkled, by the microprogrammer, in his code so that the maximum worst case time is less than the fastest peripheral device. Shown in Fig. 4 are DECODE/PASS instructions with a maximum of .9 μsec worst case -- easily within the 6.4 μsec disk response time. Indeed a 500 K word/sec I/O transfer rate can be sustained.

Most DECODE/PASS micros occur just after a memory access while the memory is busy; at a time which would otherwise be wasted. Operating in the "shadow" time (as it is usually called) in this way slows BASIL little and usually not at all.

Discussion

It was necessary in the preceding sections to provide sufficently fine detail as background for what we believe are critical issues of high-level language minicomputer architecture. In our opinion, the main concern in HLL design is that too much language will be embedded in hardware/firmware. (See [6], [7]). If too much language is embedded in the hardware/firmware, the system is frozen into a specific rigid processing framework. The future of such a system is strictly limited; it will only execute the language that was built in. Care must be taken during the design phase to ensure that new, improved high-level languages can be developed.

Furthermore, it is difficult to justify the cost of an appreciable fraction of a system's hardware if that hardware is limited to source language translation/compilation. Rather, in our opinion, it is far better to try to design multipurpose system functions, those that are useful both at run-time and at compilation.

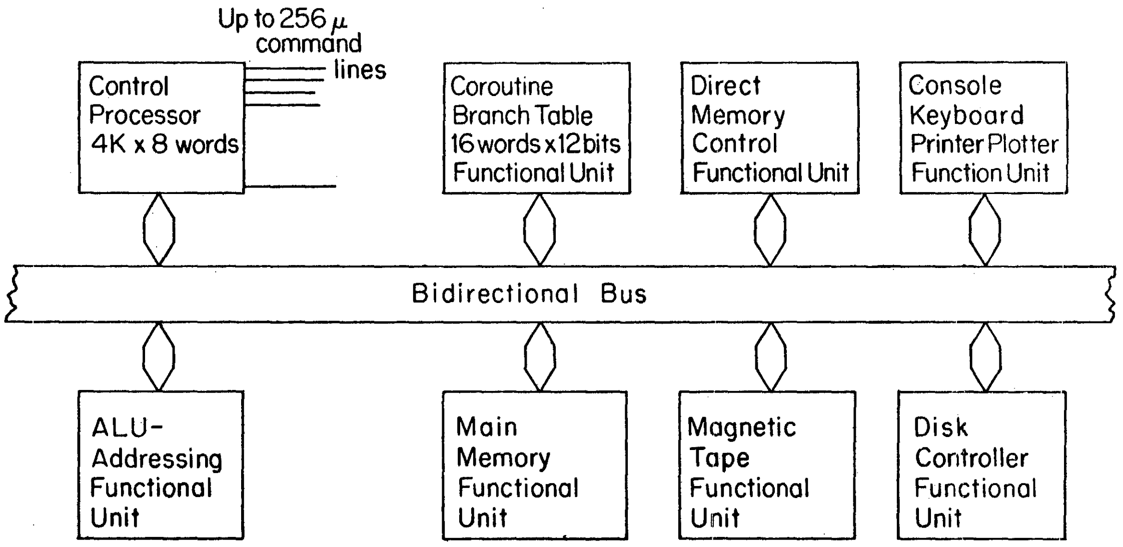
We believe BASIL's character string firmware represents such multipurpose system functions. Compilation is speeded and simplified because the firmware, executing at close to hard-wired speeds, manipulates source code more "naturally." Important system functions involving text manipulation, such as text editing, are executed faster and are easier to program.

Probably most important of all, however, is that new languages can easily be added and old languages improved because the tables that drive the lexical, precedence, and syntactical aspects of compilation are available to the software engineer for modification. Furthermore, if these tables are not enough, the software engineer can even build his own system functions; BASIL's control processor is user microprogrammable.

These are important issues for BASIL's end-user clientele -- physiologists/biophysicists. A great deal of work on unique high-level systems is needed for this important biomedical science. Current computers and operating systems are simply too primitive for these systems to be attempted.

By Chu's definition BASIL is an indirect HLL because it produces intermediate code, whereas a direct HLL executes source without translation [6]. Although there have been many proposals for HLL computers only a few attempts have been made. By far the most notable is SYMBOL [6,7,9], which has, among other features, hardware data structures, a special language (SYMBOL), all built into a hardwired machine. Only one SYMBOL machine has been built thus far, and we speculate that potential users have found the system too rigid. Weber [10] microprogrammed an IBM 360/30 to execute a high-level language called EULER. These authors report a significantincrease in execution speed.

While these projects have been milestones in the development of HLL, they have made discouragingly little impact on computer science. We feel that BASIL has better prospects because (1) the entire system is programmed in a single language and (2) it is designed for a specific clientele who traditionally have not been concerned with nor benefitted from nor have been interested in massive computer systems developments.



Up to 256 μ command lines
Control Processor 4K x 8 words
Coroutine Branch Table 16 words x 12 bits Functional Unit
Direct Memory Control Functional Unit
Console Keyboard Printer Plotter Function Unit
Bidirectional Bus
ALU-Addressing Functional Unit
Main Memory Functional Unit
Magnetic Tape Functional Unit
Disk Controller Functional Unit

Fig. 1 Block Diagram of BASIL at Functional Unit Level
(Not all functional units shown)

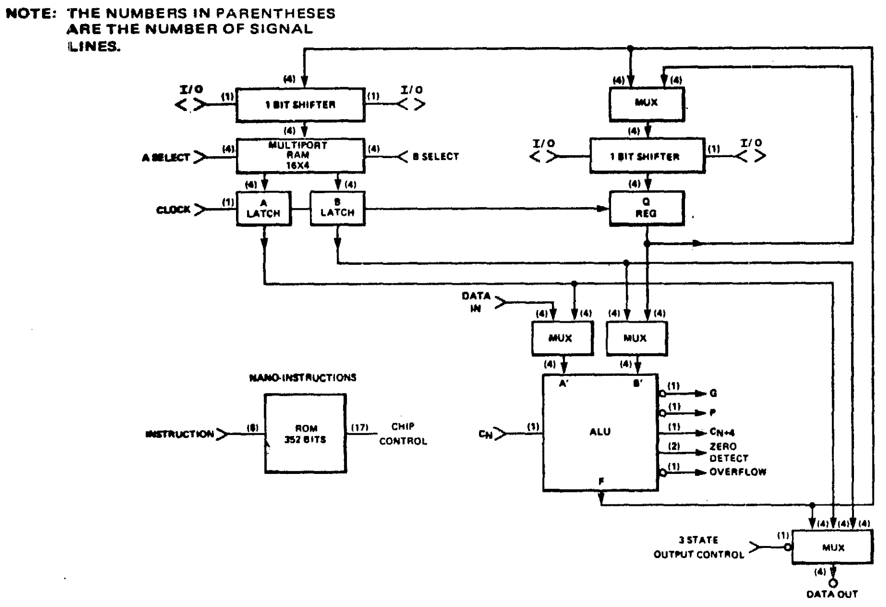


Fig. 2 Origanization of Monolithic Memories
Shottky Bipilar 4-bit Slice (6701)
(Reproduced with permission of Monolithic Memories, Inc.)

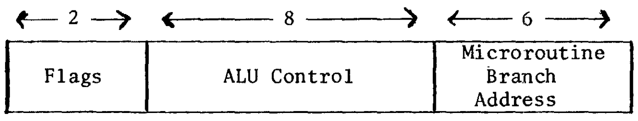


Fig. 3 Macroinstruction Format

Location	Microinstruction	Comment	Time (ns)
Macrofetch	MacroPRDINC	MacroP to memory address register (MAR); read cycle initiate; increment MacroP	100-300*
	MDRMacro	Interlock until data ready; memory data register (MDR) to ALU instruction register (AIR); initialize A- and B-source	100-400*
	Decode	Branch to address specified by highest priority F.U. (Macros have lowest, peripherals higher); allows peripherals to cycle steal	200
		Total macrofetch	400-900
BINOP: ADD, SUB, OR XOR, INC, DEC	MacroPRDINC	Acquire address of 1st operand	100-300*
	MDRMARRD	Interlock until data ready; MDR to MAR; READ cycle initiate	100-500*
	MDR DIS	MDR (1st operand) to 6701 ALU	100-500*
	DECODE/PASS	No volatile data; permit peripherals to get control processor	100-200
	MacroPRDINC	Acquire address of 2nd operand	100-300*
	BLD 1	Move the constant "1" to B-source	200
	MDRMARRD	Get 2nd operand	100-300*
	MDRDIS	2nd operand to 6701 ALU	100
	DECODE/PASS	Permit peripherals to cycle steal	100-200
	Macro REGLD	Macro instruction register to ALU instruction register	100
	MacroPRDINC	Acquire address for result	100-100*
	MDRMAR	Set up to store result	100-500*
	CPMacro	ALU to MDR, wait till memory not busy; start write	100-500*
	DECODE/PASS	Permit cycle stealing	100-200
	JMPFETCH	Jump to Macrofetch	200
		Total BINOP	1700-4200
		Total Macrofetch + BINOP	2300-5100
	Equivilant LM^2-BASIL execution for BINOP		20,000
	Equivilant PDP11/45 (RT11 FORTRAN)		12,000

Fig. 4 BINOP Listing

*Assumes (worst case) 500 ns memory without normal 4-way interleave

```
Arithmetic:
  BINOP (ADD, SUB, XOR, OR, AND)
  MONOP (Negate)
  MUL
  DIV
  Raise to power

Relationals:
  .NE.
  .LT.
  .LE.
  .EQ.   For both strings
  .GE.    and integers
  .GT.

Subscript Addressing:
  Loads:
    One dimension array          For both strings
    Two dimension array           and integers
    Multiple dimension array
  Store:
    One dimension array          For both strings
    Two dimension array           and integers
    Multiple dimension array

Control:
  Call subroutine
  GoTo
  Return
  If (false) GoTo
  For-loop test
  Move                  (strings and integers)

Functions:
  Substring
  Index
  Length
  Chrfcn
  Outstr
  Output (mag tape)
  Outeof    "
  Outrem    "

Read:
  Integer
  String
  String temp

Input:
  Integer
  String
  String - no quotes
  String temporary
  String temp-no quotes

Print:
  Print integer
  Print string
  Print control
  Print tab

Pseudo-ops:
  Forward reference
  Integer constant
  String constant
  Integer data pointer
  String data pointer
  Begin dimension statement
  Begin string declaration
  Where is JLOC for line X
  Where is integer symbol      Symbol table
  Where is string symbol        commands
  Where is label
  Where is unknown
  Assign integer
  Assign string
  Append string
```

Fig. 5 - Listing of macroinstructions required for BASIL compilier.

References

1. Torode, J. Q. and Kehl, T. H., "A Graphics Display Terminal Logic Machine," COMPCON '75, p. 313.

2. Torode, J. Q., "A Microprogrammable Logic Machine," Ph.D. dissertation, University of Washington, 1972.

3. Burkhardt, K. and Kehl, T. H., "A Logic Machine Auxiliary Processor," COMPCON '75, p. 309.

4. Kehl, T. H., Dunkel, L., and Moss, C., "LM^2 - A Logic Machine Mini-Computer," IEEE Computer, November, 1975.

5. Baruch, J. J., "Industrial View of Computer Applications in the Life Sciences," FASB Proceedings, Vol. 33, No. 12, Dec. 1974, p. 2412.

6. Mullery, A. P., Shaur, R. F., and Rice, R., "ADAM, A Problem Oriented Symbol Processor," Proc. SJCC, 1963, p. 367.

7. Rice, R. and Smith, W. P., "SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems," Proc. SJCC, 1971, Vol. 38, p. 575.

8. Chu, Yaohan, "Introducing the High-Level-Language Computer Architecture," Technical Report TR-227, University of Maryland, Feb. 1973.

9. Chesley, G. D. and Smith, W. R., "The Hardware-Implemented High-Level Machine Language for SYMBOL," Proc. SJCC, 1971, p. 563.

10. Weber, H., "A Microprogrammed Implementation of EULER on the IBM System/360-30," CACM, Sept. 1967, p. 549.

Function Distribution in Computer System Architectures

Harold W. Lawson, Jr.

Universidad Politechnica de Barcelona *

ABSTRACT

The levelwise structuring and complexity of a computer system is presented informally and as a general model based upon the notion of abstract machines (processors), processes and interpreters. The important domains of the computer architect are considered along with historical perspectives of certain stimulae and decisions that have affected the distribution of functions amongst the various levels of computer system implementations.

Keywords: Computer Architecture, Computer System Complexity, Computer History.

1. Introduction

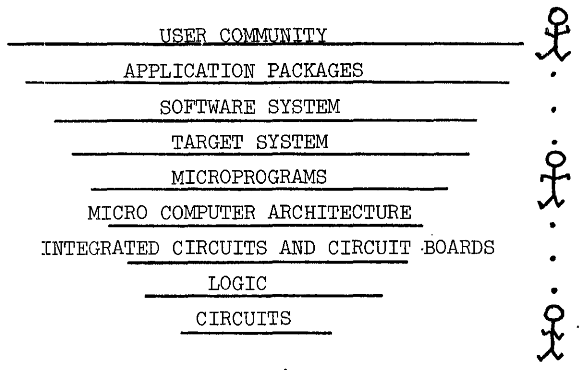
In the early days of digital computers, the stratification of computer systems was, on the surface, quite simple. Two main levels were apparent, namely, hardware and programs (e.g. software). Growth in the sophistication of the application of computers to new areas, changing physical technologies, the man-machine interface, the economics of computer usage, production and investment, inherent and created complexities and finally better understanding of the structuring of hardware and software have all influenced the levelwise structuring of the functions within computer system architectures as we view it in the mid 1970's. It would be difficult to get an agreement on precisely how many levels exist (or should be described) in a modern computer system. It would even be difficult to have agreement on the question: What is a modern computer sytem? In any event for purposes of this paper we shall begin with the leveling structure as introduced by Lawson and Magnhagen (1). This leveling is at least representative for a supporting implementation of the "Third Generation Computer System" environment as presented by Denning (2).

2. Informal View of Function Distribution

The above mentioned leveling structure appears in Figure 1. The lowest level is purely physical whereas the higher levels are all organizational, realized by hardware algorithms or program algorithms (or combinations). While the various levels may vary in content, one thing is clear: each level (1) uses level(1-1) as a "tool" for level (1) composition. In the next section we formalize, as a general model, the composition of levels and the inter-level relationships. Presently, we shall consider the implications of Figure 1 in an informal manner.

First let us consider the question of <u>complexities</u>. In the paper by Lawson and Magnhagen (1), the notions of horizontal (intra-level) complexity and vertical (nter- evel) complexity were introduced. That is, there is an <u>inherent</u> complexity within each level and <u>created</u> complexities due to the mapping of level upon level. Many examples of created complexities can be sighted. Some will be presented later in the paper.

The levels of Figure 1 are shown as an inverted pyramid to illustrate that "in general", the lower the level, the fewer people involved in designing, and producing the tools of the levels, whereas, as we go toward higher levels the greater the number of people involved in using lower levels as tools. That is to say, for example, that more people use integrated circuits than design and produce them, or, hopefully, more people use computers then design and produce computer hardwares and softwares. This relationship is an important factor in developing an informal notion of the <u>cost function</u> of vertical complexities to be specified shortly.



Computer System Architecture.

Figure 1.

As a rather practical matter, there appears to be a few unwritten principles that one can extract from the short history of designing and producing computers that have been related to many project developments and individuals making decisions.

Principle 1:
"If you cannot solve the problem, give it to someone else".

Frequently the passing of problem goes upwards in the leveling structure, thus increasing the number of people affected by the decision. <u>Complexities</u> become magnified.

Principle 2:
"If you cannot select an alternative, provide all possibilities".

That is, give your "users" general purposeness so that they can do anything they wish. While this may be useful at certain levels of the distribution, it can be disastrous at other levels and contribute heavily to magnified complexities. A simple measure of the application of this principle is the quantity of shelf-space for documentation of all possibilities (assuming the level is fully documented).

Of course these two principles are based upon the fact that the designers and/or implementors first realize that they have a problem or are aware of the alternatives. A lack in these directions can cause even greater magnified complexities in the eventual system.

Principle 3:
"If a tool <u>exists</u> that can be adapted to perform a function; use it".

* Former address: Mathematics Institute, Linköping University, Linköping, Sweden.

This decision, normally by those responsible for the economics of the system implementation, has frequently been catastrophic. The wrong tools are used to implement a level (l+1), thus forcing many complications upon the implementors and propogating complexities upwards. A professional plumber does not use carpenters tools to fix leaks in the plumbing, nor does he use general purpose tools to rectify a specific problem. Amateur plumbers due to the usage of the wrong tools or tools of too high a degree of general purposeness can create floods. Several analogies applied to computer system architectures may come to the readers mind in which floods were created.

Principle 4:
"If a design mistake is discovered during implementation, try to accomodate the mistake instead of fixing it".

This is, of course, an economic question of project investment that must be made by responsible project management in relationship to schedule slippage, penalties for late deliveries, etc. It is rare that the implementation procedure is reset to a point where the mistake can be corrected. Frequently, the end cost has been higher than the cost would have been for mistake correction. Many design mistakes have wound up being presented as "system features".

All four of the above principles have resulted in intra and inter level complexities. Now for the informal cost function. It should be obvious that if complexities are passed upwards, towards the users, the cost of complexity increases since the cost must be repaid for each usage. Whereas if complexities are passed downward, it is probable, but not always guaranteed, that total costs can be decreased.

To illustrate some concrete examples of the passing of complexity let us consider the following;

Passing complexities downwards;

- simplified job control language
- vertical (highly encoded) microinstruction formats
- tagged data and program object types
- virtual storage management by lower levels

Passing complexities upwards:

- user selection of a multiplicity of file accessing techniques
- complicated code compilation decisions left to compilers
- using unsuitable microarchitectures for emulating foreign target systems

The author does not offer any rule-of-thumb for deciding upon the correct structuring of the levels of a computer system architecture. A quote from Horning and Randell (3), with which this author is in complete agreement, explains why:

"The appropriate use of structure is still a creative task, and is, in our opinion, a central factor of any system designers responsibility".

The author does venture to say that some of the key factors are the selection of an understandable amount of semantic content at each level and the appropriate balance of special purposeness vs general purposeness. As for the number of levels; that is system dependent. A process control system does not require as many levels as a multi-user access system where the users are performing different types of data processing.

3. A General Model of Function Distribution

Horning and Randell (3) have pointed out that processes can be used to model parts of a computer system. In this section, we build on this notion. The main extension is upon the specification of program parts, which we consider as being composed of the execution sequence of programs utilizing one or more "abstract machines".

Two types of abstract machines form the notion of what we shall call "processors". One type of processor can service sequential processes, the other type of processor exists to take care of process interactions. The latter of these may be viewed as the controller of asynchronous concurrent events and for this type of processor we assume the notion of "monitors" as presented by Hansen (4) and further developed by Hoare (5) and Hansen (6). The important part is that the processors perform algorithms leaving out the notion of whether they are hardware, software or combinations thereof. A processor can, for example, be an arithmetical and logical unit as well as a resource allocation monitor.

A processor which we shall refer to as (pr) responds to a program (p) and an instance of the execution of (p) upon (pr) yields a process (ps).

We may state formally:

$$ps = f(pr,p) \qquad (3.1)$$

A process is a function of a processor and a program.

If we think then that each level other than the lowest level in Figure 1 is a processor, we can construct a computer system model generation formula

$$pr_i = ps_i = f(pr_{i-1},p_{i-1})\ i = 1,2,\ldots,n \qquad (3.2)$$

where n = number of levels above the physical circuit level of Figure 1 (i.e. the number of organizational levels). That is, a processor is defined as a process which is developed as a function of a lower level processor and its "program" where program does not only mean stored program, it can mean simple sequencing. Note that this formula also serves as a formal definition of the notion of an interpreter or interpreter hierarchy.

Given this notion, we can now state what the role and responsibility of the computer architect is in terms of producing a computer system design.

To find conventient mappings between each (pr_i,p) pair that permit convenient realizations of all levels and to distribute functions according to some goals amongst the various levels. Furthermore, to seek to minimize both intra and inter level complexities in all parts of the system.

In respect to system complexity, we can consider the following formalization as the measure of complexity(c).

$$c = \sum_{i=1}^{n} i^k \cdot \sum_{j=1}^{m} s_j$$

where: n = number of levels in the system
k = exponential growth of complexity between levels
m = number of potential state transitions within a level
s = a vector containing a measure of complexity of each potential state transition.

We note that the complexity is weighted by the level, thus reflecting the increasing cost of complexity discussed in the previous section. It is obviously difficult to measure the complexity of each transition in a uniform way, however, it is indeed related to the semantic content of discrete activities and the potential interactions of the activities at each level whether they be timed sequences through a logic chain, microinstructions, target instructions, procedures or a run in a multi-phase application package.

The programming language conrruent Pascal as presented by Hansen (6) contains some interesting features for controlling the complexity of interrelationships of processes and monitors. The process and monitor functions to be performed are declared as abstract types rather than as absolute objects. Further, a concise statement of real instances of processes and monitors giving each instance only specific "access rights" to other processes and monitors is made by a global declaration. The interconnections within an access graph are made quite explicit. Other interactions are automatically excluded by the programming language translator. This type of thinking would be extremely useful in constructing processors at all levels including microcode, Computer Aided Design, LSI layout, etc.

Having now considered the levelwise structuring and implication in a general model form, we shall finish by considering some historical events that affected the distribution of functions and various (pr,p) mappings.

4. Historical Perspective

We shall consider some, but certainly far from all of the events that have caused changes in the distribution of functions in computer system architectures.

Let us first briefly consider the physical component technology changes since we shall later concentrate on the organizational aspects. That is, what we have built and how we have structured the systems we have built.

The first major step above the early use of relay and vacuum tube technologies for logic realization, was the invention and use of the transistor. On the memory side, the first major step was the invention of magnetic core storage. We have gone through several generations of transistors realized in different types of technologies with astounding success in miniaturization, packing densities and speed increases to the point where the transistor currently forms the basis for most memories and logic.

We shall not belabor these obvious physical changes, however, it is worth noting that the physical side has had an important impact upon what we have built. Up until 1970's when mini and later LSI microcomputers became increasingly important, we viewed the central processing unit as well as the memories as expensive items.

Due to the early high costs, it is not hard to see why the first stored program* type architecture was so readily accepted. That is, uniform program - data stores, and a simple accumulator oriented processor logic.

* This concept is usually referred to as the von Neumann (7) type of architecture, but Professor Maurice Wilkes has informed the author that several people including the group at the Moore School of Electrical Engineering, University of Pennsylvania, contributed to the concept. Professor Wilkes proposes EDVAC type computer.

Early attempts to move away from the first stored program type architectures generally resulted in very complex, expensive hardware structures for processors such as the Burroughs B5000 (8,9). In any event, it was clear that certain architects such as Barton (10) did not consider the first stored program structure as a best solution that was cast in a Bible of stone. It is interesting to note that these early departing architectures would with todays integrated circuit technologies (and computer aided design techniques) be many orders or magnitude simpler to realize.

As the functional requirements for processors grew from very simple functions to include features such as floating point arithmetic, decimal arithmetic and input/output control, it became obvious to Wilkes (11) that this increase in complexity required a better organizational hardware implementation technique. Wilkes thus proposed new levels in the function distribution, namely, microprogram architecture and microprograms.

On the software organizational side, it became obvious in the late 1940's and early 1950's that constructing programs directly in machine language was a nuisance, therefore, assembly languages were conceived. Further, the notion of utility programs and subroutines for computations as well as computer management functions like input/output codes evolved to give economic and psychological advantages to the field of programming. These were the humble beginnings of system softwares.

The scope of the system software level increased with the proposal by Hopper (12) to use higher level programming languages. It is interesting to note that in the early developments of programming languages that some implementors realized that the machine for implementation, usually following the early stored program computer concept, was not a convenient machine for the mapping of programming languages programs. Therefore, the idea of inventing a pseudo-machine for the language and constructing an interpreter program of the pseudo-machine became popular.

During the mid 1950's many arguments occurred concerning the pro's and con's of using higher level languages vs assembly code on the first hand and, on the other hand, pseudo machines and interpreters as an implementation techniques as opposed to compiling code. Unfortunately, compiling won out on the efficiency of object program arguments. Thus, since the mid 1950's we have spent large sums of money reconstructing compilers to generate codes for new hardwares. Many times it is difficult and sometimes impoosible to decide upon the "best code" to generate for particular programming language features. With pseudo machines, there is usually only one best mapping, But at that time, it would have been difficult to propose constructing more hardware-like pseudo machines. Machine architectures and microprogramming, of course, have now evolved to the point where this is not only possible but economical, for example, see Wilner (13) and Lawson and Malm (14).

In the early 1950's a divergence in computer architecture occurred based upon the end use of systems. That is, processors, their related memories and input/output systems were made more special purpose and oriented towards scientific or commercial markets. Using this design strategy, certain ps = (pr,p) mappings were better for certain classes of applications. However, it was frequently required to supply compilers of scientific languages for commercial oriented processors and viceversa. These compiler mappings in many instances were extremely difficult. Like using carpenters tools to fix leaks in the plumbing.

One of the main arguments for moving to the System/360 type architecture in the early 1960's was to create a more general purpose architecture, thus cutting down of the proliferation of different system softwares that arose from having several special purpose architectures. One system for all users and, as was attempted, a uniform programming language for all, namely, PL/I. One of the main problems was that by adding general purposeness, the mappings ps = (pr,p) for most all areas, Fortran, Cobol, PL/I etc. to System/360 became more difficult. This resulted in very complicated schemes of compiler code generation and optimizations of System/360 codes as well as providing the economic need for several levels of support. Did this reduce the software proliferation?

While one may question the soundness of the target system architecture of System/360, it is important to note that this was the first wide scale use of Wilkes microprogramming concept. Various System/360 processors were designed and programmed to be compatable interpreters of the System/360 architecture. The microprocessors were not general purpose microprocessors, but designed for the special purpose of implementing System/360. In any event, since they were programmable they were used to produce "emulators" of IBM second generation equipment. Many of these ps = (pr,p) mappings, where pr was the microprocessor, were extremely painful and represent prime examples of the plumber using carpenters tools and creating floods.

On the operational (access) side of computers, it became evident in the mid 1950's that one user at a time exploiting such an expensive resource was uneconomical. Therefore, the ideas of spooling programs, supervisors, schedulers etc. eventually led to the notion of operating systems. As the reader is well aware, these properties led to profound changes in the way people use computers and led to special requirements upon many levels in the computer system architecture. Denning (2) does an excellent job of extracting the principles of operating systems as we know them in third generation computer systems.

In the mid 1960's several people were speculating that microprogramming would become an important media for aiding in programming language and operating system requirements. See, for example, Opler (15), Lawson (16), Wilkes (17) and Rosin (18). However, the necessary step to accomplish this was to have a more general purpose microprocessor. Such microprocessors started to be developed in the late 1960's, see Lawson and Smith (19) and have resulted in several interesting designs including those mentioned earlier, namely Wilner (13) and Lawson and Malm (14).

While redistribution of functions to a more general purpose microprocessor may have certain appeal in reducing complexity, it is a realistic fact of life that the huge investment in computer products (hardwares and softwares) of typical third generation products has slowed down the marketing of products based on different concepts for target system architectures and instruction sets. At least one system architecture by the Amdahl Corporation has used redistribution of functions to better use Large Scale Integration parts in perpetuating System/360 and 370 type architectures.

One prime example of the redistribution of functions from software to hardware has been the wide spread use of the concept of virtual memory. This concept has helped solve many complex problems with program overlays and file management by utilizing levels lower than the system software level. It is interesting to note that the idea and implementation existed for many years before it was widely implemented. See Kilburn, Edwards, Lanigan and Sumner (20).

On the hardware side, many manufacturers during the 1960's after strong resistance from "hardware artists" have accepted the use of Computer Aided Design techniques for accomplishing printed circuit board layout. This type of resistance is very similar to the argument of "programming artists" utilizing assembly languages vs the utilization of higher level languages. That is, I can always do a better job without automation tools. But we may ask; at what global cost? In any event, just as with higher level languages, CAD can be extremely effective and certainly permits us to more quickly and clearly realize more sophisticated architectures at the lower organizational end of the function distribution.

In the 1970's we have experienced an awareness (certainly not too soon) of the complexities of using computer systems. Much is now written on structured programming and programming style, see for example Dahl, Dijkstra and Hoare (21), Weinburg (22) and Denning (23). These writings have influenced application and system software construction and should in the future hopefully start to influence lower levels of the organizational end of the function distribution.

Hansens Concurrent Pascal (6) as mentioned earlier contains some interesting features for structuring and controlling "compoment" interrelationships. This work builds on that of Wirths Pascal (24). It is clear in these two cases that there is an attempt to redistribute functions in the distribution of Figure 1. By having the programming language translator "guarantee" that invalid interrelationships cannot exist, we can avoid constructing lower level hardware and/or microcode solutions to checking for invalid operations.

It is interesting to note that the concepts of structured programming and improved programming style require a restrictive type of programming, thus reducing semantics. Could this be a good general principle? Special purpose tools, with restricted semantics, can perhaps be used to solve, in a better manner, special problems!!

While this historical treatment has certainly not been exhaustive, it is hoped that the reader can see in terms of the earlier informal and general model presentations of structuring and complexity, some of the implications of the events that have happended in the brief history of digital computer design, implementation and utilization.

Conclusions

It would be difficult to terminate this paper without saying a few words about possible directions for function distribution in the future. One thing is clear, the range of the distribution that a computer architect must cover has increased. He must be intimately familiar with all levels with the exception of the details of the physics of logic realization by particular technologies. Further, he must be prepared to specify and appropriate number of levels to solve the problem at hand and assure that the ps = (pr,p) mappings are realizable, economic and convenient to utilize.

Current trends towards LSI logic parts at low costs due to high production volume will undoubtedly have a great impact upon the organizational levels, see Fuller and Siewiorek (25). Just make sure that the parts selected represent the correct tools to do the job, otherwise, look out for floods. The mass production of large scale integration parts may be insensitive to the complexity of the logic, but users of the components are not insensitive.

Due to the fact that processor physical structures have reduced in cost, we can expect more dedicated systems on the one hand and an attempt to utilize many processors in a distributed manner on the other.

As a practical matter for many areas of computer usage, we are saddled with our history due to large program investments. There will certainly be an impetus to seek solutions to new architectures that can accomodate this software investment. Most likely, these solutions will be combinations of several levels in the distribution of functions.

Acknowledgment

The author wishes to thank Professor Maurice Wilkes for reviewing this material, particularly to verify the historical aspects.

References

(1) H.W. Lawson, Jr. and B. Magnhagen. Advantages of Structured Hardware, Proceedings of the Second Annual Symposium on Computer Architecture, Houston Texas, January 1975.

(2) P.J. Denning. Third Generation Computer Systems. Computing Surveys 3, 4 (1971)

(3) J.J. Horning and B. Randell. Process Structuring. Computing Surveys 5, 1 (1973).

(4) P.B. Hansen. Operating System Principles. Prentice Hall, Englewood Cliffs, N.J., 1973.

(5) C.A.R. Hoare. Monitors: An Operating System Structuring Concept. Communications of the ACM 17, 10 October 1974.

(6) P.B. Hansen. The Programming Language Concurrent Pascal. IEEE Trans. on Software Engineering, Vol SE-1, No. 2, June 1975.

(7) - . von Neumann's Collected Works. A.H. Taub, Ed. Pergamon, London 1963.

(8) Burroughs Corporation. B5000 Reference Manual. Form 200-21014, Detroit 1961.

(9) W. Lonergan and P. King. Design of the B5000 System. Datamation, 7, 5, May 1971.

(10) R.S. Barton. Ideas for Computer Systems Organization: A Personal Survey. Software Engineering vol 1, Academic Press, New York, 1970.

(11) M.V. Wilkes. The Best Way to Design an Automatic Calculating Machine. Manchester U Computer Innaugural Conference, 1951.

(12) G.M. Hopper. Compiling Routine A-Ø. Unpublished documentation. May, 1952.

(13) W.T. Wilner. Design of the B1700. Proceedings of the FJCC, Anaheim, California, 1972.

(14) H.W. Lawson, Jr. and B. Malm. A Flexible Asynchronous Microprocessor. BIT 13, 2 June, 1973.

(15) A. Opler. Fourth Generation Software. Datamation 13, 1 (1967).

(16) H.W. Lawson, Jr. Programming-Language-Oriented Instruction Streams. IEEE Trans C-17 (1968).

(17) M.V. Wilkes. The Growth of Interest in Microprogramming: A Literature Survey. Computer Surveys 1, 3 (1969).

(18) R.F. Rosin. Contemporary Concepts in Microprogramming and Emulation. Computing Surveys 1, 4 (1969).

(19) H.W. Lawson, Jr. and B.K. Smith. Functional Characteristics of a Multi-Lingual Processor. IEEE Trans. on Computers vol C-20, July 1971.

(20) T. Kilburn, D.B.G. Edwards, M.J. Lanigan and F.H. Sumner. One-Level Storage System. IEEE Trans. EC-11, April 1962.

(21) O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare. Structured Programming. Academic Press, London, 1972.

(22) G.M. Weinburg. The Psychology of Computer Programming. Von Nostrand Reinhold, 1971.

(23) P.J. Denning, ed: Special Issue: Programming. Computing Surveys 6, 4, 1971.

(24) N. Wirth. The Programming Language Pascal. Acta Informatica, vol 1, no. 1, 1971.

(25) S.H. Fuller and D.P. Siewiorek. Some Observations on Semiconductor Technology and the Architectures of Large Digital Modules. Computer, October, 1973.

INTERFACE, A DISPERSED ARCHITECTURE

Chris A. Vissers
Twente University of Technology
Enschede, The Netherlands

0. Abstract

Past and current specification techniques use timing diagrams and written text to describe the phenomenology of an interface.

This paper treats an interface as the architecture of a number of processes, which are dispersed over the related system parts and the message path. This approach yields a precise definition of an interface. With this definition as starting point, the inherent structure of an interface is developed. A horizontal and vertical partitioning strategy,based on one functional entity per partition and described by a state description, is used to specify the structure. This method allows unambiguous specification, interpretation, and implementation, and allows a much easier judgement of the quality of an interface. The method has been applied to a number of widely used interfaces.

1. Introduction

Many committees, charged with standardizing an interface struggle many years (8 to 10 years makes no exception) to get the job done. What are their problems ? We can find at least three. First, an interface is always much more complex than a first estimate suggests. Qualification and quantification of the needs of the users is a difficult task, application dependent, and subject to different opinions. The definition of the functional contents of an interface that satisfies these needs introduces an extra choice, and consequently makes agreement one level more difficult. Second, the disclosure of an interface allows the linkage of products of different companies into one system, which requires the political will to make this happen. The third problem is the available methodology and language for the specification of an interface and its preliminary versions. Conventional methodology uses timing diagrams and written text, often illustrated with tables and drawings. This methodology has a number of serious disadvantages. The most important of these are discussed in this paper. Bad specification methodology makes an interface difficult to master and document, and enhances the risk of errors, incompleteness, inefficiency and vagueness. It also opens the door to obstruction of progress through vague reasoning. An interface with such characteristics contributes to nonuniform and unintended interpretations. And faithful to Murphy's law this has led to system malfunctioning, even though the interface was scrupulously interpreted.

Therefore, the availability of an efficient tool that allows unambiguous and clear specification and interpretation of an interface would be of great profit to both its designers and users. For the designers it facilitates a clear discussion and expedites a correct, complete, efficient and clear specification. For the users it will help to avoid system malfunctioning, caused by misinterpretation or unauthorized extension of a given interface.

This paper presents a specification method that tries to incorporate the desired characteristics. It has been applied to a number of existing and proposed standard interfaces [1,2,3] with satisfactory results. The state description technique, which is closely related with the method, is reflected in an interface which is now becoming an international standard. It proved to be of extreme value both in the development and use of this interface [4,5]. The method is based on the definition of an interface, which will be discussed first. Next, technique and language for the specification of an interface are discussed. This is followed by the development of a structuring discipline for an interface, and a discussion of the character of a standard interface. Finally some conclusions are drawn.

2. Definition

Few attempts seem to have been undertaken to state a manageable definition for the concept of interface. It may be that the term interface itself, and its translations to various languages(cutting place, tangent plane), pretends to be clear enough. But the term interface is currently used with many divergent interpretations. (The term 'connection' is used in [6] to indicate the same kind of relation definition as discussed in this paper). Therefore, if we want to discuss a specification methodology for it, an adequate definition is demanded.

What we clearly want, is to be able to bring system parts that can be considered and designed as separate functional entities, into relationship to form a system with a higher level of functional performance. The possibility that some system parts can be brought into relationship makes us say that these system parts can interface. Therefore an interface can intuitively, but still informally, be called a 'relation definition'. In order to interface, the system parts must be given certain properties which are attuned to each other.(e.g. two system parts know the same variable, one as its producer, the other as its consumer). All properties of a system part are defined by the complete specification of its functional behaviour, its architecture. Therefore we are able to define an interface by specifying the architectures of the related system parts. Through the definition of the architecture of each part, the interfaces of these parts are concurrently established. However, in some cases we may desire, or be forced, to define an interface first, and the complete architectures later (e.g. for the definition of a standard Channel-to-I/O interface). In these cases it is undesirable, unfeasible, and unnecessary to define the complete architecture of the related system parts in order to be able to define their interface.

System parts have a relation if they can affect each other's functional behaviour. Without mutual effect they ignore each other and the system parts are independent and unrelated. The mutual effect is through variables (messages) with a defined behaviour and exchanged via a message path. Suppose we want to define the effect of system part A on system part B through variable V. The behaviour of V can be defined through the definition of the generative mechanism of V, which belongs to A and forms a portion of A's architecture (the influence of A). The effect on B through V can be defined through the definition of that portion of B's functional behaviour expressing that effect (the effect on B). Conversely we can define the effect of B on A through W. This yields, another portion of A's and B's architecture. The two portions of A's architecture can be specified either as separate portions if there is no correlation between them, or integrated if there is. A similar statement can be made for B.

An interface of two or more systems parts defines for each system part that portion of its architecture that allows a relation between those system parts to form a system providing a desired function.

The previous reasoning assumed a functional passive message path for the exchange of the variables V and W. Though this is often the case, an interface may contain a message path that performs logic operations on the variables it exchanges as explained in section 4.3: Central message path. Consequently the definition of an interface has to be extended to contain the architecture of the message path, as shown in figure 1. As with the definition of an architecture, the definition of an interface is a specification problem. This specification problem concerns not one architecture, but a portion of each of the related architectures and the message path in between. In this sense the concepts of architecture and interface are equivalent. The term relational function is given to that portion of an architecture that is part of the considered interface. The remaining portion of the architecture is called local function. It forms the complement of the relational function in the architecture's total relation with its environment.

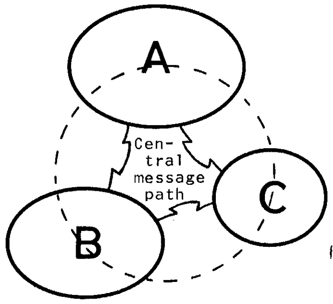


Figure 1a: Three related architectures A, B, and C.

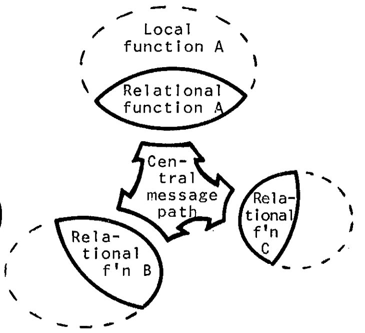


Figure 1b: The A-B-C Interface.

This definition provides the basis for a sound specification method: the interface is specified by the separate specification of each relational function and of the message path. Since these items are portions of an architecture, their description may use the same techniques and languages as applied to architectures in general.

3. Description techniques and languages

Three basically different description techniques are conceivable and used to specify an architecture: the phenomenological, the assertive, and the generative description.

3.1 Phenomenological description.

The phenomenological description gives an observation of the behaviour of the input and output variables. As known from automata theory, that bases its definition of an automaton on it, this is a valid specification method. But in order to be complete, the observation must include all input and output variables, and all possible sequences of their values. Though this requirement is not important for the development of (automata)theory, it is impractical for any architecture of some complexity, because of the sheer monotony and inordinate length of the sequences. It is not surprising that this method is not used in practice for the specification of architectures. Therefore it is a real surprise to observe that the conventional specification method for interfaces is still based on the phenomenological description, since the timing diagrams are literally an observation of the signal lines that exchange the messages among the system parts. Those diagrams are furthermore by definition incomplete, as long as they do not contain all input and output variables of the relation functions. This incompleteness however, is normal in conventional specification methodology, since the variables exchanged across the local function/relational function boarder, are normally missing in the timing diagrams. (In 4.1 Sources and sinks we come back to this point). To make things even worse, most specifications only contain the most significant sequences. Although this cuts down on the monotony and length of the sequences, it makes the specification even more incomplete. Hence, the written text, which usually goes together with the diagrams, becomes essential to fill up the gaps in the specification with timing diagrams. The text, however introduces several new problems. The use of another language will inevitably tempt the writer of the specification to 'explain' the diagrams. And so the reader must carefully distinguish between text that contains additional specification and text that contains redundant written specification of the diagrams. Furthermore it is in most cases not clear whether the text is meant to be assertive, generative, or phenomenological. The observation of the message path as basis for the description, results in an intermixed description of the contributing relational functions and the message path itself. This burdens the implementer of an architecture to untangle the relational function, that is part of this architecture, from the total specification. The phenomenological description is maximally unstructured, opposite to the nature of human thinking. Therefore the implementer has to bridge the 'maximum distance' from the phenomenological specification to his product, a realizable generative specification.

The previous observations suggest a specification method for an interface in which each relational function and the message path is specified individually. Each individual specification uses one description technique, preferably not the phenomenological, and one description language.

3.2 Assertive description.

The assertive description method, originally introduced to prove program correctness [7,8], specifies an architecture by specifying assertions on the behaviour of the input and output variables. In so doing, the assertions form in fact a shorthand notation for the phenomenological description of the input and output variables, and allow the latter's drawbacks to be avoided . The assertive specification can not be simulated, since this requires a model of the generative mechanism between inputs and outputs. Simulation can be highly desirable if we want to check the assertions against samples of the phenomenological description. The specification of the assertions themselves is the biggest problem in using this technique, in particular when the architecture is complex. This often requires that the architecture is specified as a collection of related partitions, and each partition is specified assertively. Through this partitioned specification, internal variables are defined, and the assertive approach comes close to the generative approach which is followed in this paper.

3.3 Generative description.

Associated with the phenomenological specification of a finite automaton, the type of system to which we are restricted when we start an implementation, is a (minimum) state machine. This state machine can be considered as the generative mechanism that maps the input onto the output. A description of it can replace the phenomenological description. For complex systems, as frequently encountered for interfaces, it will generally be difficult to establish this minimum state description. Since the generative description is used to replace a desired input/output behaviour (phenomenology) the latter is not available to deduce a minimum state description from it. Associated with the minimum state machine are many equivalent machines with the same phenomenology which do not contain the minimum set of internal states. Therefore, although the minimum state description often appears to be the most attractive one [2], we often have to be satisfied with a reasonably good equivalent description. Most of our experience is based on the use of this type of description for interfaces, though an assertive description might equally well have been chosen.

3.4 Language.

The most primitive language in which the generative mechanism for a finite automaton can be expressed is the state transition diagram or table, as used in sequential circuit theory. Though this language has succesful been used in a number of applications [1,3,4], and remains quite suitable for specific ('logic') functions, it appears to work inefficiently for more complex interfaces. In these cases an algorithmic language, that contains primitives for many (numerical and logical) operations is much more powerful [2]. Some examples of this are shown in figures 8 and 9. It remains essential though, that the algorithmic description is interpreted as a state machine description. The representation of the states is free, since they are internal to the automaton. The optimum choice is a representation that provides maximum clarity of specification. This can often be achieved by adapting state representation and formulation of transition conditions to each other [2]. The algorithmic language allows also many different representations for the state transition diagrams or tables. So these descriptions can be mixed with algorithmic statements, and simulated on a computer [9]. The possibility of simulating the interface is an important advantage of the generative description.

4. Structure.

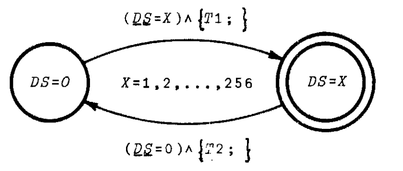
Human nature does not favour the specification of a complex system by one single homogeneous function, such as a large diagram, table or algorithmic expression. We no longer have confidence that it represents what we want. Instead we start with smaller individual parts of specification. For each part we have confidence that its specification is, or can made to be, what we want. And we link up (interface) those parts, often by extension of their specification, into a larger part of specification. Such a partioned specification method raises problems of its own: where to start, and how to link up the parts. A general structuring discipline, providing a structure in which partitions of specification can be embedded, can greatly help in reducing these problems. Such a structuring discipline for the specification of an interface is discussed in the following sections.

4.1 Sources and sinks.

The first class of partitions is called the source and sink functions layer. To operate as one functional entity -e.g. a Channel- information is exchanged between the local and relational function of the channel. Conventional methodology often hides the variables carrying this information in vague statements, such as
' if the Channel is able to communicate with a device, it places an address on the bus. . .'
Such a statement gives rise to numerous questions:
- Where and how is the ability to communicate generated ?
- When the channel is able, will it actually communicate ?
- What happens when the channel is able, but other activities require the channel's attention ?
- When, where and how are addresses generated, how and where are they coded ? Etc.

It is therefore necessary to make a clear distinction between the variables generated by the local function and by the relational function, and to decide which of these cross the boundary between local and relational. The interface is only interested in those locally generated variables that are inputs to the relational function. Since their generation is a part of the local function, which is per definition unknown, we cannot define their behaviour. But we can define their required behaviour via a finite automaton, called a source function:

A source function defines the existence, set of values, and required behaviour of a local variable, which is made available as input to the relational function [3].



```
      ∇ DECSOURCE
[1]    W1:→(DS=0)/W1
[2]     DS←DS
[3]     DELAY T2
[4]    W2:→(DS≠0)/W2
[5]     DS←DS
[6]     DELAY T1
[7]     →W1
      ∇
```

(b)

Figure 2: Decoded source.

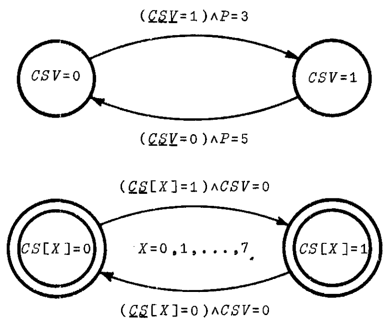
Graphic (a) and algorithmic (b) description of a decoded source. The local variable DS, whose behaviour is free, is used to determine the behaviour of the relational variable DS. The behaviour of DS is as follows: DS=0 remains at least T1 seconds valid, DS≠0 remains at least T2 seconds valid. Each value of DS≠0 is enclosed by the 'separation message' DS=0. The variable DS is used in the relational function. If DS behaves as DS (i.e. as required), the implementation of the source is trivial: a short-circuiting from DS to DS.

The local function represents the complement of the relational function in the architecture's total relation with its environment. This leads to the introduction of the sink function, the counterpart of the source function:

A sink function defines the existence, set of values, and behaviour, of a relational variable, which is made available as input to the local function [3].

A sink function has the same appearance as a source, therefore no examples are shown. Practical applications often require the combination of a source and sink function into one function. Such a function is called a conversational source or sink function, dependent upon its main task. The conversational character is required when the validity time (the time that the value of the variable remains unchanged) of a relational variable is defined in a logical way (figure 3) instead of by the use of time (figure 2).

(a)



(b)

```
      ∇ CODEDSOURCE
[1]    WAIT:→(CSV←(((CSV=1)∧P=3)∨CSV)∧~(CSV=0)∧P=5)/WAIT
[2]     CS←CS
[3]     →WAIT
      ∇
```

Figure 3: Coded converational source.

Graphic (a) and algorithmic (b) description of a coded conversational source. The local variables CSV and CS are used, together with the relational variable P, to determine the behaviour of the relational variables CSV and CS. As long as CSV=0, the code of CS may change. Analogous to figure 2, CSV=0 may be interpreted as the separation message. CSV is used both in the relational function and in the local function. The code of CS remains valid, as long as CSV=1.

When all sources and sinks for each relational function are identified, they form a layer that shields the local functions from the remaining part of the interface. This remaining part is called in figure 4a basic interface function. The source and sink layer defines the behaviour of all inputs and outputs of the interface and is therefore a suitable place to start the definition It shows that an interface can be considered as an architecture that is dispersed over several other architectures and the message path.

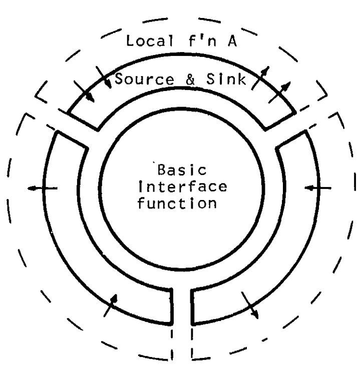


Figure 4a: Source and Sink layer and Basic interface function.

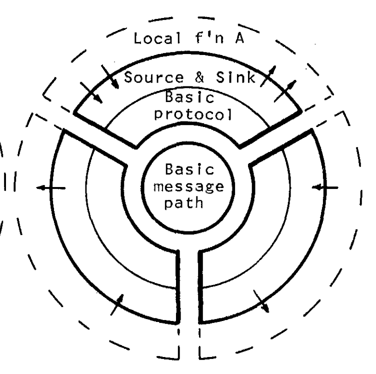


Figure 4b: Partitioning of the Basic interface function into the Basic protocol functions and the Basic message path.

When an interface is considered as a dispersed architecture, one can view a system either as a collection of related architectures (figure 5a), or as a collection of interfaces (figure 5b), The latter viewpoint is used when a system makes use of one or more predefined interfaces.

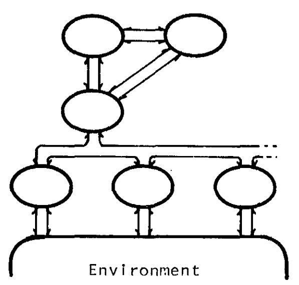


Figure 5a: System, specified by a collection of architectures.

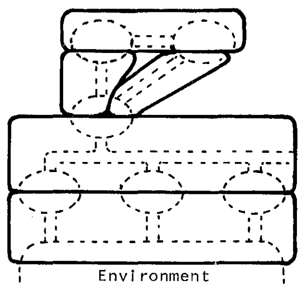


Figure 5b: System, specified by a collection of interfaces.

4.2 Basic protocol.

When the source and sink layer is defined, the remaining part of the interface represents its basic function. This basic interface function contains the flow of the data from sources to sinks and the operations performed on the data. If the basic interface function is defined as one automaton and subsequently partitioned into the basic protocol functions (the parts that are accomodated in the related architectures), and the basic message path (figure 4b), it will usually result in a voluminous, inflexible and costly basic message path. Most interfaces require a reduction of this cost through a reduction in the space and time allowed for the exchanged variables. At the same time increased flexibility is desired and obtained through a general purpose message path. Consequently the basic protocol functions have to be adapted to this reduced and generalized message path to form part of a definite specification. Starting with the basic protocol, however, is very useful in formulating the interface's basic task and in deciding how the elements of this task are allocated to the related architectures.

4.3 Central message path.

The next step is the definition of the central message path. The basic message path indicates what variables are to be exchanged. As a first step it can be decided how much space and time will be assigned to these variables. Three most important and competing parameters influence this choice. The first one is the possibility of physical separation of the architectures, in particular their maximum physical distance. The second is the desired time performance of the interface, and the third is the desired reliability of the interface. Another important parameter is the level of independence of the message path from the related architectures. The weight of those parameters is highly dependent on the application of the interface (e.g. industrial plant control versus laboratory experiments), which makes a universal central message path for all applications unlikely.

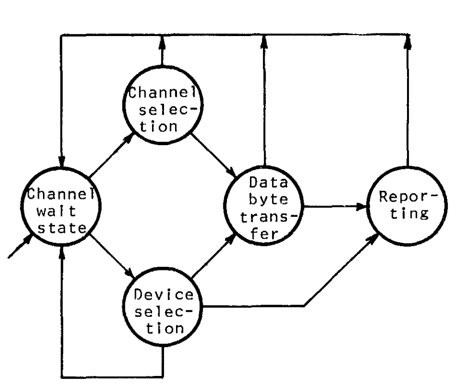


Figure 6: Basic protocol function.

A simplified basic protocol function of a complex Channel-to-I/o interface [1]. The function is located in the Channel. The diagram shows how a data transfer sequence can be build up. The elementary steps in the sequence are represented by the states in the diagram. The transitions in the diagram indicate how these steps may be sequenced. It follows that data transfers from different devices may be mixed. This allows the multiplexing of the message path.

As a second step, the function of the central message path is defined. In some message path configurations, such as in string or loop configurations, this function is trivial: just one or more connections. A less trivial function is represented by the so called bus or party line configuration that can be found in most Channel-to-I/O interfaces. Such a bus structure gives the 'or' of the coded variables presented to it by the relational functions. This 'or' is a simple, but essential function that allows multiplexing of data from various destinations. The implementation of the 'or' function is generally distributed over the relational functions.

```
     ∇ CMP
[1]   L:SRL←PSL
[2]     SRLV←PSLV
[3]     PRL←∨/SSL
[4]     PRLV←∨/SSLV
[5]     →L
     ∇
```

Figure 7: Central message path function of SDLC [1].

The central message path function connects one primary station to multiple secondary stations. The S(econdary) R(eceiving) L(ine) is connected to the P(rimary) S(ending) L(ine). Idem for the S(econdary) R(eceiving) L(ine) V(alid), which carries the clock.
The value of the P(rimary) R(eceiving) L(ine) is the 'or' function of the sending inputs to the line, represented by the vector S(econdary) S(ending) L(ine). Idem for P(rimary) R(eceiving) L(ine) V(alid). PSL and PSLV are generated in the function FRAMETRANSMITTER of figure 8. PRL and PRLV are used in the function FRAMERECEIVER of figure 8.

More sophisticated message path functioning can be found in many CPU-Channel interfaces. Here the message path contains store operations, priority assigments to regulate concurrent acces to the same storage locations, and the like. These functions are performed by main storage.

The CPU-Channel interface is a prominent example of the use of memory in the message path. Exchange of variables via memory (indirect transfer) allows either parallel or sequential operation of the relational functions. A freedom of choice which is left to the implementer. When no memory is used, the transfer is direct, which requires parallel operation of the relational functions. The determination of the space, configuration and function of the message path of the interface plays a definite role in the total performance and applicability of the interface. The message path is therefore central to the relational functions as illustrated in figure 10. It is not surprising that some names of interfaces are based on it (e.g. the unibus). But this does not justify the identification of the message path, or its momentary condition (e.g. the state of main storage as the interface between the program modules), as the entire interface. The message path should be derived from the basic protocol functions and not vice versa.

4.4 Transfer.

The introduction of the central message path requires an adaption of the basic protocol to the space, time, and function of the central message path. This requires a number of functions to adapt the format (multiplex, serialize) of the variables supplied by the basic protocol to the message path and vice versa. The sequencing of different variables over the same transmission path requires a mechanism to indicate the type and (in)validity of these variables. The introduction of the coded representation of the variables on the message path, discussed in the next paragraph, also makes these mechanisms necessary. Dependent on the choices made for the message path, such a mechanism can make use of handshaking, strobing, or enveloping techniques. The chance of message mutilation caused by the message path, often requires the introduction of message protection mechanisms These mechanisms can range from a simple parity check to complex methods such as cyclic redundancy check, buffering, numbering and retransmission.

The functions charged with these types of tasks form a layer, shielding the basic protocol from the message path. This layer is called Transfer in figure 10.

```
    ∇ FRAMETRANSMITTER;SPTR;SFLAG
 W1:→(~TRANS)/W1
  SPTR←FLAGPTR,(CHECKBITS SFRAME),SFRAME,FLAGPTR
 W2:→PSLV/W2
  SFLAG←((ρSPTR)>24+ρSRFAME)∨(ρSPTR)≤8
  PSL←(0,¯1↑SPTR)[SFLAG∨SCNT≠5]
  SPTR←(-SFLAG∨SCNT≠5)↓SPTR
  FRSEND←0=ρSPTR
  SCNT←PSL×SCNT+1
 W3:→(FRSEND,PSLV,~PSLV)/W1,W2,W3
    ∇
```

```
    ∇ X←CHECKBITS Y;N
  N←ρY
  X←16ρ1
 L:N←N-1
  X←1↓(X,0)≠POL∧X[0]≠Y[N]
  →(N≠0)/L
  X←⌽~X
    ∇
```

```
    ∇ FRAMERECEIVER;FINB;SUPR
 W1:→(~PRLV)/W1
  FINB←RFLAG
  SUPR←(~PRL)∧RCNT=5
  RFLAG←(~PRL)∧RCNT=6
  RCNT←PRL×RCNT+1
  RFRAME←(8×RFLAG)↓SUPR↓PRL,(~FINB)/RFRAME
  FRDY←RFLAG∧ρRFRAME≥32
 W2:→(FRDY,PRLV,~PRLV)/L,W2,W1
 L:RFOKE←∧/(16↑RFRAME)=CHECKBITS 16↓RFRAME
 W3:→(PRLV,~PRLV)/W3,W1
    ∇
```

Figure 8: Transfer functions of SDLC [2].

The FRAMETRANSMITTER generates PSL and PSLV (see figure 7) from the variable S(ending)FRAME, that it receives from the ENCODER function, shown in figure 9. It indicates when the frame is transmitted (by FRSEND) etc. The FRAMERECEIVER assembles a variable R(eceived)FRAME from PRL and PRLV (see figure 7), and presents this to the DECODER function of figure 9. The subfunction CHECKBITS generates the cyclic redundancy checkbits, and is part of both the transmitter and receiver function. It is not an individual automaton.

4.5 Coding and decoding.

As mentioned under 3.4 Language, the representation of the state of the automaton is free as long as we are in the architectural phase, and can be chosen to provide maximum clarity of specification. When the automaton is implemented, the implementer is free to represent the state of the variable by any suitable set of code elements according to his criteria. Source and sink variables are internal to the individual architecture's as are most of the variables contained in the basic protocol and transfer functions. Their final representation is the implementer's decision. The situation is different however, for the variables that are exchanged via the message path. Usually an interface is designed to allow the implementation of each architecture by an independent group without requiring them to communicate with all other groups. When this is the case, the representation of the variables crossing the message path is public and must be settled by the interface designer. The variables are principally provided or accepted by or via the basic protocol, and are subjected to the transfer operations. Hence the functions performing the coding and decoding form a layer in between the (basic) protocol and the transfer, as shown in figure 10. The message path provides the code elements for the representation of the exchanged variables.

```
    ∇ ENCODER;SAF;SCF;SIF
 W1:→(~TRANS)/W1
  SAF←(8ρ2)⊤IADD
  SCF←(8ρ2)⊤(16×FB)+(SNSI,SRQI,SROL,SNSA,SCMDR,SRR,SRNR,SREJ,SI)/
   3 7 15 99 135 ,(1 5 9 ,2×SNS)+32×SNR
  SIF←(SCMDR/HELPFIELD),(SNSI/OUTNSI),SI/OUTI
  SFRAME←SIF,SCF,SAF
 W2:→((ENCODERDY←TRANS),~TRANS)/W2,W1
    ∇
```

```
    ∇ DECODER;RAF;RCF
 W1:→(~FRDY∧RFOKE)/W1
  RAF←¯8↑RFRAME
  RCF←¯8↓¯16↑RFRAME
  RIF←¯16↓RFRAME
  MA←(2⊥RAF)∊(IADD,CADD)
  PB←RCF[3]
  INR←2⊥RCF[0 1 2]
  INS←2⊥RCF[4 5 6]
  RVIF←~RCF[7]
  RNSI←3=2⊥RCF[0 1 2|4 5 6 7]
  RSIM←7=2⊥RCF[0 1 2|4 5 6 7]
  RORP←19=2⊥RCF[0 1 2|4 5 6 7]
  RDISC←35=2⊥RCF[0 1 2 4 5 6 7]
  RSNRM←67=2⊥RCF[0 1 2 4 5 6 7]
  RRR←1=2⊥RCF[4 5 6 7]
  RRNR←5=2⊥RCF[4 5 6 7]
  RREJ←9=2⊥RCF[4 5 6 7]
 W2:→((DECODERDY←FRDY),~FRDY)/W2,W1
    ∇
```

Figure 9: Encoding and Decoding functions of SDLC [2].

The ENCODER function generates the variable SFRAME from the variables provided by the protocol functions of SDLC. The DECODER function performs the opposite operation. Both functions are linked to the transfer functions of figure 8.

4.6 Protocol.

The introduction of the central message path, the transfer layer, and the coding layer may affect the basic protocol functions. If so, these functions must be adapted to the communication facilities provided by this central part of the interface. This adaption predominantly involves the introduction of sequencing functions, due to the time limitations of the message path. The adapted basic protocol functions thus form a layer, in figure 10 called Protocol, in between the source and sink layer and the coding and decoding layer. The protocol layer which is defined through this procedure contains the highest level of functions representing the substance of the relation of the architectures. In a standard Channel-to-I/O interface, the interface is primarily concerned with the exchange of different classes of messages, such as commands, data, and status. The protocol layer in this type of interface controls such things as the setting up, maintenance and closing down of message transfers, as well as the interleaving of message transfers from different origins and their priorities in case of concurrency. In general few arithmetic and other data manipulation functions, are found in this type of interface. Other interfaces such as the Channel-Main Storage interface, or CPU-Main Storage interface, may contain in high degree of data manipulation and buffering functions. The CPU-Main Storage interface can be considered as including the definition of almost the entire CPU instruction set.

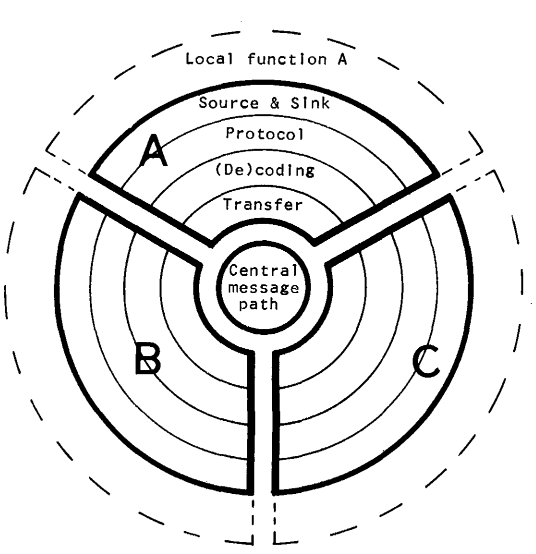


Figure 10: Layered structure for an interface

4.7 Further development.

The previous discussion of the structure of an interface suggests a sequence in the development of the layers, according to the sequence of the sections 4.1 through 4.6. This development is based on a strategy of successive definition. First the architecture of the total interface is determined, and its partitioning and dispersion over the related architectures. Next the architecture of the central message path is determined, and finally the architectures of the individual relational functions. Though this procedure is a useful guideline, a practical application often requires a substantial number of iterations through this sequence, due to the high dependency among the layers.

A further substructuring per layer may result in either the development of sublayers per layer (extended horizontal partitioning) or a partitioning of a layer into functions which are not or only slightly related (vertical partioning). The previous discussions have already used the vertical partitioning by interpreting each layer as a class of functions, and showing examples of such functions. Much is dependent on the possibility of defining a function first as an independent entity, and next of establishing the linkages to and from other functions. As is true for the vertical partitioning, the extended horizontal partitioning may also provide more clarity in the specification of the interface. The protocol function of figure 6 shows what type of operations may be sequenced. The way these operations are organized in detail can be specified in a lower protocol layer. Complex data transmission interfaces may build up their transfer layer as a stack of sublayers. Such a sublayer, and all that it encloses, may be interpreted as the central message path of the transfer layer that is just one level higher. An opposite development also occurs frequently: variables pass a layer unchanged.

The structure so far developed for the interface is shown in figure 11 from the perspective of an individual architecture. Each box in the figure represents a function, that exists in parallel with the other functions and is related with them via the exchange of variables. This horizontal and vertical structuring is different from the structuring in which functions on a lower layer are used to implement an abstract machine on a higher layer [10].

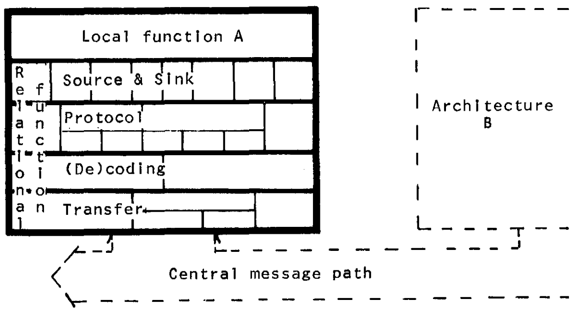


Figure 11: The interface from the perspective of architecture A.

5. What is a standard interface ?

As stated, a system can be understood as a collection of interfaces (figure 5b) as well as a collection of architectures (figure 5a). This viewpoint is significant when an interface is defined first, and the associated architectures later. This happens with a so called standard interface. A standard interface, such as a Channel-to-I/O interface, is always defined to meet many different architectures, e.g. printers, tape units, disc units, display devices, architectures that still have to be invented, etc. in different quantities and configurations. At the time of the definition of the standard, the current application area is known, and there is a rough estimate of the characteristics of future applications. Definition of a standard to include all current and future applications is not only impossible, it is also highly inappropriate since it loads any particular application with the overhead of a multiplicity of unused application functions. Instead the standard is defined to suit all requirements of current and future applications without containing the specific functions of individual applications. The standard is by definition incomplete. Consequently, when the standard is used in a particular application, each relational function has to be extended with application dependent functions. Those application dependent functions form yet another layer around the source and sink layer of the standard interface, and are designated 'Application' in figure 12a.

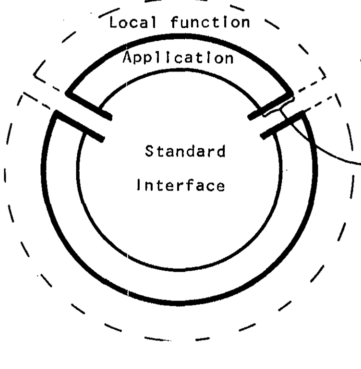


Figure 12a: Application of a Standard Interface

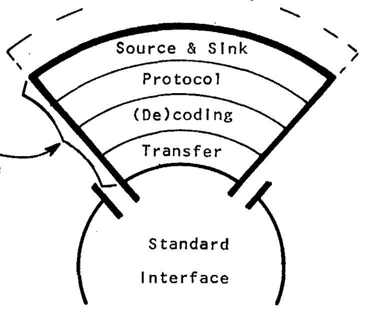


Figure 12b: Partitioning of the Application layer.

The consequence of this structure is that the variables exchanged among the application functions are unknown, i.e. transparent to the standard interface, and yet pass all layers and the message path. Since we want the function of the standard te remain invariant with each application, it implies that the standard has to provide for the space and time for the exchange of those variables. If on the level of the central message path the available space is to be defined in terms of available code elements, the definition of the available space at the level of the source and sink functions has to be in terms of the same number of code elements, since the of main memory contents, such that computations can proceed in an uninterrupted manner.

As discussed in [14], a space-time tradeoff exists regarding temporary results. Since each array operation consists of a startup time and an execution time, some time is wasted due to additional startups, when a large array has to be operated upon in parts. Additionally, there exists a memory management overhead in allocating space for temporary results.

A straightforward solution to the above problem is the "elimination" of intermediate array results, with the consequent saving of memory accesses and space [16]. This scheme, which has been implemented at the scalar level in the IBM 360/91 [17], is considered in the context of the SCR design for arrays of data, as the following two schemes:

(a) The storing and fetching of temporary results is avoided by transmitting them directly among the respective arithmetic units. This scheme can be extended to sequences of assignment statements having common subexpressions and to the case where the final result of an array expression is the input to another one.

To weigh the attractiveness of this approach, we evaluate the relative saving in memory accesses when an array assignment statement, involving n binary operators is evaluated. Denoting the number of array elements by ℓ, customarily $3n\ell$ memory accesses would be required, while the proposed scheme requires $(n+2)\ell$ accesses; hence $2(n-1)\ell$ accesses are saved. Given that the probability of the occurrence of an arithmetic assignment statement with n $(n>0)$ binary operators is p_n and postulating a fixed mean array size $(\bar{\ell})$ for array expressions of varying complexity, then the relative saving in memory accesses is $(2\bar{n}-2)/3\bar{n}$, where $\bar{n}$ is the mean value of n.

(b) Memory accesses are saved by concurrently executing operations involving the same input operands. An example of the relative saving in memory accesses using this scheme is given in Section 2.4.

The use of variables in a sequence of array assignment statements of a program can be represented as a directed acyclic graph, which will be called the data digraph. Each node in the data digraph corresponds to an input variable or the generation of a result (permanent or temporary). The links determine variables or temporary results, which are utilized in generating a new result. The data digraph can then be manipulated (see Section 2.5) to determine sets of operations whose simultaneous execution minimizes memory accesses. Such sets of operations, which have to be executed in a single step by the SCR, constitute a task.

To illustrate the previous discussion, we consider the multiplication of two vectors with complex data types:

$$A.B = (a+a'i) \cdot (b+b'i) = (ab-a'b') + (ab'+a'b)i$$

Figure 1 gives the data digraph corresponding to this computation. In this case the relative saving in memory accesses, when all operations are performed in one step is 66.7%.

2. The SCR: Functional Description

2.1 Operating Environment of the SCR

The SCR is intended to operate in conjunction with a multiprogramming/multiprocessing computing system, whose interfaces with the SCR are discussed here.

The computing system consists of several Program Processors (PP's), which execute user programs and perform OS functions. The PP's and the SCR share a high-bandwidth main memory by means of a main memory controller. The main memory is large enough to allow multiprogramming. The PP's are equipped with local memories, thus relieving the main memory from excessive PP accesses. Programs executed by the PP's have special provisions for specifying array operations and while executing user programs, the PP's relegate array operations to the SCR. However, scalar operations and also array operations that cannot be vectorized (see [15] for examples) are handled directly by the PP's. The SCR has local autonomy and requests for computation or tasks, which the SCR receives from various PP's are enqueued in the SCR and assigned to execution based on local self-optimization considerations.

2.2 Functional Organization of the SCR

The SCR design is aimed toward the major goals of achieving fault-tolerance ("graceful degradation") and of making efficient use of main memory bandwidth.

The approach employed to preserve main memory bandwidth is to allocate several Arithmetic Processors (AP's) to the execution of a task such that temporary results are transmitted directly from one AP to another*. Since rather high bandwidths of data transmission are involved, an Interconnection Network (IN) is used to transmit intermediate results among the AP's. Additionally, due to the high data transfer rates at which array operands are to be transmitted between the main memory and the SCR, dedicated Address Generators (AG's) are assigned to each array operand.

In order to achieve fault-tolerance and high availability, a "pooling" concept is used for the various subsystems of the SCR. In the case of AP's, the mean AP requirement for a single task (as generated by a program translator) is smaller than the total number of AP's. During program execution, a subset of the available AP's (under some constraints due to the IN) is assigned to the execution of a task. Several tasks can be executed concurrently in the SCR. The binding of program requests to the SCR elements is deferred until the time of execution. At that time it is performed dynamically taking into account the inventory of available elements. Consequently, system operation can continue with fewer elements (in "degraded mode") after failures of system elements occur.

Figure 2 gives a block diagram of the SCR and its interfaces with the computing system. The SCR consists of the following subsystems:

(a) A pool of m AP's (Arithmetic Processors) which access the main memory controller by means of a Memory Interface Unit (MIU). The AP's are high bandwidth, pipelined arithmetic units capable of performing basic arithmetic operations generating elementary results (sums, products, etc.), as well as some common matrix operations such as the inner product (it is considered to be a nonelementary result). The internal structure of the AP's will not be discussed here, but we postulate that once an AP is set up by an external command, it proceeds autonomously with the assigned operation. An Input Switching Unit (ISU) whose function is described in (c) below is associated with each AP.

(b) The MIU contains a pool of k AG's (address generators) which generate the addresses of data elements to be transmitted to or from main memory. Each AG is associated with a buffer memory to mask the variation in main memory response time. High bandwidth buses are used to transmit data and addresses between AG's and the main memory controller. The operation of AP and AG units is overlapped, such that the AG's fetch input operands in lookahead mode into the buffers, before the AP's operate upon them.

(c) The IN (interconnection network) provides data communication links among the AP's according to the pattern described in Section 2.5. The ISU associated with each AP selects the specified inputs from the set of buses originating from the AG's and other AP's (the IN) according to task requirements under external control.

(d) A Switching Network (SN) is used to dynamically assign AG's to AP's. The motivation and certain

*A variation of this approach is discussed in Section 2.4.

advantages of adopting this scheme are discussed in Section 2.4.

(e) The Scheduler in addition to task scheduling, controls PP-SCR communication. The requests for computation ("tasks"), which the SCR receives from various PP's are enqueued by the Scheduler and slated for execution based on the availability of the SCR elements requested by the task. Upon the completion of a task, the Scheduler notifies this event to the program which originated the task. Task scheduling is discussed in more detail in Section 3.1.

(f) The Control Unit (CU) performs the actual set-up of a task, which is defined by the Scheduler. The CU generates a control vector, which determines the hardware configuration of the AP's and the AG's, in addition to initializing certain registers internal to these units, for the duration of a task. In other words, this is a virtual processor-memory-switch system [18], where the desirable configuration can be achieved by means of static (residual) microprogramming. After task setup a computation proceeds independently from the CU. The CU attention is required when an arithmetic exception or hardware fault is indicated. The CU communicates the status of failed units to the Scheduler, which updates the SCR configuration tables accordingly.

In the following sections we discuss tradeoffs that arise in implementing such a design.

2.3 Issues in Utilization Arithmetic Processors in Tandem

Although we consider homogeneous AP's, their rate of execution may differ in performing basic arithmetic operations of varying complexity. Here, we address the issue of matching the bandwidth of pipelined units, which have to be connected in series to execute a task.

When we define the delay in one pipeline segment as one cycle, then for some basic operations, such as addition and multiplication, the AP's generate one elementary result per cycle. The execution rate for more complex basic arithmetic operations, such as division, is a multiple of the cycle time. We also assume that the delay due to the IN is fixed and equal to one cycle (see also Section 2.5).

Since we are dealing with deterministic queueing systems operating in tandem, the provision of buffers between AP's to hold unprocessed intermediate results doesn't improve performance and throughput is determined by the execution rate of the "slowest" AP*. The following alternatives are to be considered in operating several AP's in tandem:

(a) Control the inflow rate of array data, such that it matches the bandwidth of the slowest AP. This approach results in a potential waste of memory bandwidth, as well as the underutilization of the non-bottleneck AP's.

(b) Only arithmetic operations which are executable at the same rate should constitute a task. A disadvantage of this scheme is the fact that temporary results generated due to this constraint have to be saved in main memory. An advantage of this scheme is the potential simplification of control.

(c) Design the AP's to generate one elementary result per cycle, regardless of the complexity of the basic arithmetic operations to be performed. The drawback of this approach is that the additional hardware is not justifiable for infrequent and complicated operations.

(d) Use multiple AP's in parallel, in order to compensate for the low execution rate of complicated arithmetic operations. A disadvantage of this approach is the additional control requirement, as well as complications in the interconnection scheme.

*Such buffers, when introduced in the ISU, can be used to delay the inputting of operands into an AP.

To simplify this discussion, it is assumed in the remainder of this paper, that the set of basic arithmetic operations are executed at the same rate. However, a more careful study of the above-mentioned tradeoffs is required for a design decision.

2.4 The Memory Interface Unit

In this discussion we consider an AP which has two operand inputs and one output. Then the permanent (static) association of three AG's with each AP reduces the AG utilization due to the following:

(a) The capability of directly passing intermediate results among AP's obviates the need to access the main memory for them.

(b) Scalar variables occurring in an array expression are treated as "immediate" operands and can be replicated by the AP's.

In considering the dynamic sharing of the AG's among the AP's, we face the following tradeoffs:

(a) The number of AG's can be reduced substantially. At this point we estimate the saving in the number of AG's when array expressions involving binary operators are considered. Given that $\bar{r}$ AP's are required on the average by each task, then there are $\lfloor m/\bar{r} \rfloor$ tasks in the system, each requiring $\bar{r}+2$ AG's for their execution, and the number of AG's saved is: $3m - (\lfloor m/\bar{r} \rfloor)(\bar{r}+2) \simeq 2m(1-1/\bar{r})$. For $\bar{r} = 2$, m AG's are saved, which is a considerable reduction in system hardware. A more accurate approach to determine the number of AG's is given in Section 3.2.4.

(b) The AG's can be used independently for memory remapping functions (e.g., transposing a matrix) or initialization of arrays (e.g., setting the elements of a matrix to zero). To remap a data structure, two AG's are used, one for fetching the data and the other one for storing. This approach, of course, requires a means to interconnect the AG's. Memory remapping operations usually precede certain arithmetic operations, for example, in matrix multiplication, we may need to transpose the second matrix in order to realize efficient columnwise fetching of the second matrix during matrix multiplication.

(c) There is a potential gain in the reliability of the system due to functional modularization. On the other hand, the unreliability of the SN associating the AP's and the AG's, its cost, and the overhead required for control require attention. The generality of the dynamic association provided among the AP's and the AG's is affected by hardware technology (cost, reliability, speed), as well as the issue of scheduling and resource utilization.

(d) It was suggested in Section 1.4 to save memory accesses by concurrently executing operations involving the same input operands. This scheme can only be realized when an AG in charge of accessing an array can be associated with multiple AP's simultaneously. In the case of matrix multiplication, this approach results in considerable savings in memory fetches. We consider the case when the multiplication of two nxn matrices is performed using p AP's; then one row of the first matrix can be multiplied simultaneously by p columns of the second matrix†. In this case, a single AG is allocated to fetch the row vector and the number of memory fetches is $n^3(1+1/p)$ versus $2n^3$ in the conventional scheme (p = 1). The saving in memory accesses is 37.5% for p = 4 and approaches 50% for p = n.

While a more detailed study is under way to determine the SN design, for the purpose of this discussion the SN is assumed to be a crossbar switch, the implication being that there are no restrictions in associating AP's and AG's. The crossbar switch consists of a single bus per AG against three buses per AP, two of which correspond to ISU inputs, while the third one is the AP output.

†The inner product operation is used.

2.5 The Interconnection Network and Task Characteristics

The function of the IN is to provide data communication links among the AP's. The IN is a passive subsystem and the actual line switching is performed in the ISU's under CU control, as part of task setup. The following criteria are used in synthesizing the IN:

(a) The available interconnection patterns should be suited to the most common instances of computation that arise. Less common cases are transformed to directly executable form before execution.

(b) The utilization of AP's should not be drastically reduced due to limitations imposed by the IN. This underutilization, which manifests itself in the form of increased response time in executing tasks, is due to the fact that the multiple AP's required by a task should be able to transmit intermediate results directly to each other.

(c) The AP's should be interconnected in a manner that facilitates scheduling. A homogeneous interconnection is then superior to a nonhomogeneous one.

(d) The IN should accomodate undisrupted operation when failures in the system occur. Due to the relative hardware complexity of the AP's with respect to the IN, we primarily consider AP failures. The protection of the IN is to be considered separately. The degradation in system performance upon AP failures then should correspond to the actual loss in computational capacity.

(e) The overall operation of the SCR is simplified when the delay introduced by the IN in transmitting data is fixed.

Assuming that dedicated buses are used in the IN to satisfy criterion (e), we initially use criterion (a), task characteristics, to design the IN. The design thus obtained will be evaluated and modified (if necessary) to satisfy the other criteria.

If the input nodes of the data digraph (as defined in Section 1.4) are omitted, we obtain a digraph which reflects the interconnection requirements for a sequence of assignment statements. Generally, due to an insufficient number of AP's and/or limitations in the IN, it will not be possible to perform the sequence of computations in one step. Hence, we face the issue of partitioning the data digraph.

In the data digraph, we differentiate between links corresponding to the transmittal of intermediate and permanent results. Since permanent results have to be stored in main memory, the cost in memory accesses of cutting a link corresponding to the transmittal of a temporary result is twice that of a permanent result or input variable. Additionally, when we limit the maximum number of operations allowable in a task, then under certain restrictions, an optimal partitioning procedure exists, which minimizes the cost of links joining the various subsets of the data digraph [19].

Based on the above discussion, the following scheme is adopted to design the IN and to determine the optimal task size. First a static analysis of a set of "typical" programs is performed by partitioning their digraph for different maximum task sizes. Then we determine the task size, which, while maintaining memory accesses at a low level, requires an IN of moderate complexity.

To illustrate the concept, we consider the design of an IN, when the interconnection digraph corresponding to tasks has at most four nodes and five links. This means that a task consists of at most four arithmetic operations and not more than five results are transmitted among AP's. For m AP's, an IN where the output of AP[i] is connected to the ISU of AP[(i+1)<u>mod</u>m] and AP[(i+2)<u>mod</u>m] then satisfies the interconnection requirement. Figure 3 illustrates the AP's and the IN in this case. This illustrative IN will be further discussed in the remainder of this paper and its properties will be further investigated.

An example (taken from [15] and translated into APL) is given in Figure 4(a) to illustrate the mapping of task (program requirements into the SCR hardware. All variables are arrays of the same size (ℓ). The data and the interconnection digraph for this task are given in Figures 4(b) and 4(c) and Figure 4(d) is a possible setup for the task. Note that "Y" is fetched by a single AG and channeled simultaneously to three AP's.

2.6 The Coordination Issue in the SCR

In Section 2.2 it was noted that after the CU sets up a task for execution, the AP's and the AG's proceed autonomously with its execution. This section describes the coordination of the operation of a set of AP's and AG's which are assigned to the execution of a task.

During the execution of a task, the next set of input operands must be placed into the buffer before an AP may start to operate upon them. To denote the presence of operands in the buffer, indicators are associated with each location in the buffer. The AP's assigned to a task, start operating on the next set of input operands, when all respective indicators indicate readiness. However, this approach is expensive in terms of hardware and provisions are needed to handle the interspersed vacuous results that would be generated by the AP's.

A reduction in hardware and control complexity is achieved by using multiple buffers associated with each AG and assigning a single indicator to each buffer. In this case, after operating on the contents of a set of buffers in one burst, the set of AP's will immediately switch to the next set of buffers, if they are ready; otherwise the AP's are idle. However, the situation will arise infrequently if the total computational bandwidth is matched to the main memory bandwidth and input operands are mapped in the memory in a manner which facilitates their access. The first scheme outperforms the second one with respect to overall execution time when idling occurs. Here we face a tradeoff between complexity of control and delay in execution.

A similar problem arises when a computation is halted because an output buffer is full. In general, a set of AP's executing a single task proceed with their operation, when the next set of buffers associated with their inputs are full and those associated with their outputs empty. This scheme can then be implemented by distributed control, where the AP's sense the state of certain indicators (as determined by the task), before proceeding to operate on the next set of operands.

3. Task Scheduling in the SCR

3.1 Task Characteristics

As discussed in Section 2.1, tasks originating at the PP's are enqueued in the SCR and then executed according to a predetermined scheduling discipline. Tasks sent by a PP may be independent, in which case they can be executed concurrently; or there may be precedence relationships among tasks, which are observed by the Scheduler. In this discussion only independent tasks are considered.

Tasks sent by the PP's contain enough information for their scheduling as well as setup. A computational task specifies the following: arithmetic operations to be performed, variables or "pseudo-variables" [6] involved in the operation, the interconnection among AP's for transmitting intermediate results, and the delay which should be introduced in the ISU in inputting certain operands to the AP.

The AP requirement of a task is determined by the number of arithmetic operations. The AG requirement is similarly determined by the number of array operands to be transmitted to or from main memory. The interconnection requirement then poses a problem. In the general case, we consider a processor digraph, where the nodes correspond to AP's available for task allocation and the links in the graph correspond to the IN lines among those AP's. In order to schedule a task, we must find a subgraph of the above graph in which the interconnection digraph of the task can be imbedded. On the other hand, due to the regularity of interconnections in the illustrative IN, the above approach (which is similar to finding isomorphic graphs) can be avoided by transforming the interconnection requirement to an AP proximity requirement. Due to the large number of cases involved and space limitations, we treat this issue by considering two examples:

(a) F ← ((((A+B)+C)+D)+E) can be evaluated by assigning AP's which are at most two apart. If the expression is rewritten as: F ← (((A+B)+(C+D))+E), then at least three contiguous AP's are required to execute the task. Note that the total latency in the pipeline has been reduced from four to three levels.

(b) X ← (A+B) x (A-B); y ← (A+B) ÷ (A-B) can be executed when four contiguous AP's are allocated to the task.

Due to the unavailability of statistical data characterizing tasks as discussed above, the queueing characteristics of the system are studied in the next section under postulated task characteristics.

3.2 Evaluation of Queueing Characteristics

3.2.1 The Single Resource System

We consider a queueing model of the SCR system, where tasks require multiple AP's for their execution and constraints due to other resources are not considered [20]. In the open queueing system, the task arrival process is Poisson with rate λ. The input stream consists of tasks with unequal processor requirements (given by the vector R). Tasks in different task classes (a task class is determined by the AP requirement) occur with fixed probabilities (given by the vector F). All tasks are enqueued in a single queue in the order of their arrival. The processing time distribution is exponential with rate μ.

For some realistic task characteristics, increasing the number of AP's over $\underset{\sim}{\max}(R)$ under a fixed total processing capacity (to be denoted by C which is set to unity for normalization purposes) results in improvements in the mean response time characteristic for high utilization factors. This is due to the reduction in unutilized capacity associated with idle processors, an inevitable occurrence under nonpreemptive scheduling disciplines. Additionally, a set of unbiased scheduling disciplines (with the exception of FCFS) considered in [20] were observed to have a close mean response time characteristic once m is increased over $\bar{r}$ (mean AP requirement of tasks), under fixed total processing capacity. The first-bit (FF) scheduling discipline, which inspects the queue (waitlist) from head to tail and allocates the first executable task, is employed in this discussion due to the simplicity of its implementation.

A simulation program was developed to determine the queueing characteristics of the SCR (see [21] for details). The normalized mean response time graph ($\mu C\bar{T}/m$) versus the normalized arrival rate ($\lambda_n = \lambda\bar{r}/\mu$) as obtained by the simulation program is given by graph (a) in Figure 5, for the following set of system and task characteristics: m = 8; R = (1,2,3,4) and F = (0.5,0.25, 0.125,0.125). This graph will serve as a benchmark in further sections.

3.2.2 The Effect of the IN on Performance

Here we use the simulation program to determine the effect of the IN on the mean response time characteristic.

Instead of generating the interconnection requirement of tasks individually, the following approach was taken in order to keep the simulation cost down. The system was subjected to the same workload twice, with the following restrictions on task assignment:

(a) Only contiguous AP's can be allocated to the execution of a task.

(b) No two neighboring AP's allocated to the execution of a task can be more than two AP's apart.

The mean response time graphs which are obtained from the simulation program under restrictions (a) and (b) provide upper and lower bounds to the mean response time characteristic and are given by graphs (b1) and (b2) in Figure 5. Note that for a high utilization factor ($\rho = \lambda_n = .9$) the increase in mean response time is about 10%, which indicates that the IN design satisfies criterion (b) in Section 2.5 for the assumed task characteristics.

3.2.3 Operation in Degraded Mode

In this section we study the performance of the system when AP failures occur. First we note that the Scheduler need not distinguish between a busy and a failed AP. However, since a failed AP remains "busy" indefinitely, provisions are needed in the IN to bypass a failed AP.

Graphs (c1) and (c2) in Figure 5 give the upper and lower bounds to the mean response time characteristic for a single AP failure. Note that while the computational capacity of the system has been reduced to C = .875, the mean response time increase (the average of c1 and c2 values is considered) is about 35% when λ_n = .6. On the other hand, if we consider a seven AP system with C = .875, then the mean response time increase is only 24% over the eight AP system for λ_n = .6. This discrepancy in mean response time degradation indicates that criterion (d) in Section 2.5 is not satisfied and performance can be improved by providing additional (redundant) links in the IN.

Consideration of double AP failures in the system makes the inadequacy of the IN further evident. For example, if the two AP failures are four apart (which occurs in 4 out of 28 cases when m = 8), no task requiring four contiguous AP's can be allocated. However, system operation is possible in other cases and graph (d2) in Figure 5 gives the lower bound to the mean response time characteristic when two contiguous AP's fail.

Based on the above discussion, we add additional links to the system. Provided that ($i \pm 3$) $\underline{\text{mod}}$m links are added to the system, then for the task characteristics under consideration, a single failed AP is completely masked. The failure of two adjacent AP's would then correspond to a single AP failure in the original IN design.

Another method, which avoids a large degradation in the mean response time characteristic upon AP failures, is to reduce the effect of AP failures on processing capacity by increasing the number of AP's under fixed total processing capacity. There are two disadvantages to this scheme: (1) increasing the number of AP's beyond a certain point results in increases in mean response time; (2) additional hardware is required in related subsystems when the number of AP's is increased.

Finally, heuristic approaches can be used to improve the performance of the system for given task characteristics. For example, a nine AP system with the original IN will tolerate all two AP failures and a considerable improvement is then achieved at the cost of one extra AP.

The task during whose execution a hardware failure occurs, is reenqueued for execution, given that its input operands were left intact.

3.2.4 The System with Two Resources

In the single resource system (Section 3.2.1), it was postulated that the number of AG's is always adequate, such that no task has to wait for this resource.

Here we consider the case where the AG's, in addition to the AP's are considered for task scheduling and there are no restrictions in associating AP's and AG's for task execution (see Section 2.4).

In order to determine the number of AG's, when the number of AP's and task characteristics are fixed, such that the performance of the system doesn't degrade when the second resource is introduced, the following two approaches are considered:

(a) Determine the mean response time of the system under a given scheduling discipline and load level (arrival rate), while increasing the number of AG's (a parameter in the simulation program) until the relative improvement in mean response time with respect to the single resource system is sufficiently small. A disadvantage of this approach is the high cost of simulation as well as the fact that the result depends on the load level and the scheduling discipline.

(b) This approach is independent of the scheduling rule and is based on the argument that the throughput bound of the single resource system shouldn't be reduced, when the constraint due to the second resource is introduced. The throughput bound is defined as the smallest input rate for which the system is guaranteed to saturate regardless of the scheduling discipline, which may be preemptive. The throughput bound for the two resource system for given system resources and task characteristics can then be obtained by a simple extension of the linear programming formulation given in [20].

For the task characteristics considered in Section 3.2.1, and assuming that each task requires two more AG's than AP's (this is true for any assignment statement involving binary operators), the second approach was used to determine that at least eighteen AG's are required for eight AP's to maintain the throughput bound at the same level.

In general, the two approaches complement each other as follows. The second approach (perhaps enhanced by a utilization factor argument) is used to narrow down the search space and determine a tentative number of processors for the "second" resource. This number is then used to determine if the performance of the two resource system is satisfactory for the scheduling discipline under consideration.

In the above case, the performance of the system was determined to be satisfactory under the first-fit scheduling discipline (modified for two resources), when eighteen AG's were provided.

3.3 Dynamic Versus Static Scheduling of Tasks

In this section we compare the SCR system with the four-pipeline version of the ASC [5]. These two systems have the following differences:

(a) In the ASC, the pipelines are available only to the program executing in the central processor. In the SCR system, any program executing in the PP's can generate tasks for the SCR. While it can be projected that due to resource sharing, the AP utilization will be higher in the SCR, this advantage needs to be weighed against the overhead in time and hardware to implement the SCR scheme.

(b) Since there are no provisions in the ASC to transmit intermediate results, only independent operations can be executed simultaneously, while the SCR performs operations at the task level. To illustrate this point, we consider the execution of the sequence of assignment statements given in Figure 4(a). The FORTRAN optimizing compiler developed for the ASC will minimize the schedule (the maximum finishing time for the set of operations) by breaking down a computation into several parallel segments if required [15]. The schedules for the ASC and the SCR systems are given by Figures 6(a) and 6(b), respectively*. Given that all arrays have ℓ elements and denoting the startup time by ϵ (the startup time is not shown explicitly in the diagrams) then the execution time in the "modified" ASC is $\ell+12\epsilon$ versus $\ell+4\epsilon$ in the SCR.

*It is assumed that all basic arithmetic operations take one cycle, which is not true in the case of ASC.

(c) As was noted in (b), the objective of the ASC's FORTRAN compiler is to speed up the execution of the program executing in the central processor, while the objective in the SCR is to speed up the execution of the set of programs executing in the PP's. Hence, it is undesirable to speed up the operation of a single program at the expense of other programs in the system. However, based on more careful analysis, it may prove advantageous to break down computations of arrays whose size exceeds a certain threshold value.

(d) In the case of a pipeline failure in the ASC, although operation continues with the pipe disabled, the schedule generated by the FORTRAN compiler loses its optimality and there is a need to recompile the program if optimal execution is desired. As described earlier, the SCR system is less sensitive to AP failures.

These potential advantages of the SCR over the ASC system provide further motivation for an in-depth study of the suitability of the SCR as a special purpose array processor.

4. Conclusion

This paper presents our current results on the SCR, in a design study concerned with the evolution of a fault-tolerant hierarchical computing system. While additional work is required to evaluate the quality and the completeness of the proposed SCR design, the design can be considered as a contribution to the series of novel architectures addressing themselves explicitly to efficient solutions for extensive numerical computing requirements.

Acknowledgments

The authors are pleased to acknowledge suggestions by Professors Gerald Estrin and Tomás Lang of UCLA and by the conference referees, which helped to improve the quality of this paper.

References

[1] Avizienis, A., "Architecture of Fault-Tolerant Computing Systems," Proceedings International Symposium on Fault-Tolerant Computing, Paris, France, June 1975, pp. 3-16.

[2] Flynn, M. J., "Trends and Problems in Computer Organizations," Proceedings of IFIP Congress 74, Stockholm, Sweden, August 5-10, 1974, pp. 3-10.

[3] Iverson, K. E., A Programming Language, J. Wiley and Sons, New York, 1962.

[4] Hintz, R. G. and D. P. Tate, "Control Data STAR-100 Processor Design," Proceedings Sixth Annual IEEE Computer Society International Conference, San Francisco, California, September 1972, pp. 1-4.

[5] Texas Instruments, Inc., The ASC System - Central Processor, Austin, Texas, 1973.

[6] Giloi, W. K. and H. Berg, "STARLET - an Unorthodox Concept of a STRING/ARRAY Computer," Proceedings of IFIP Congress 74, Stockholm, Sweden, August 5-10, 1974, pp. 103-107.

[7] Ruggiero, J. F. and D. A. Caryell, "An Auxiliary Processing System for Array Calculations," IBM Systems Journal, Vol. 8, No. 2, 1967, pp. 118-135.

[8] Barnes, G. H. et al., "The Illiac IV Computer," IEEE Transactions on Computers, Vol. C-17, No. 8, August 1968, pp. 746-757.

[9] Crane, A. B. et al., "PEPE Computer Architecture," Proceedings Sixth Annual IEEE Computer Society International Conference, San Francisco, California, September 1972, pp. 57-60.

[10] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21, No. 9, September 1972, pp. 948-960.

[11] Thurber, K. J. and P. C. Patton, "The Future of Parallel Processing," IEEE Transactions on Computers, (Correspondence), Vol. C-22, No. 12, December 1973, pp. 1140-1143.

[12] Chen, T. C., "Parallelism, Pipelining and Computer Efficiency," Computer Design, January 1971, pp. 365-372.

[13] Kleinrock, L., "Resource Allocation in Computer Systems and Computer Communication Networks," Proceedings of IFIP Congress 74, Vol. 1, pp. 11-18.

[14] Owen, J. E., "The Influence of Machine Organization on Algorithms," Complexity of Sequential and Parallel Numerical Algorithms, Ed. J. F. Traub, Academic Press, 1973, pp. 111-130.

[15] Wedel, D., "FORTRAN for Texas Instruments ASC System," SIGPLAN Notices, Vol. 10, No. 3, March 1975, pp. 119-132.

[16] Abrams, P. S., "An APL Machine," SLAC Report No. 114, Stanford University, Stanford, California, October 1970.

[17] Tomasula, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, Vol. 11, No. 1, January, 1967, pp. 25-33.

[18] Lesser, V. R., "Dynamic Control Structures and Their Use in Emulation," SLAC Report No. 157, Stanford University, Stanford, California, October 1972.

[19] Kernighan, B. W., "Optimal Sequential Partitions of Graphs," Journal of the ACM, Vol. 18, No. 1, January 1971, pp. 34-40.

[20] Thomasian, A. and A. Avizienis, "Dynamic Scheduling of Tasks Requiring Multiple Processors," Proceedings Eleventh Annual IEEE Computer Society International Conference, Washington, D.C., September 9-11, 1975, pp. 77-80.

[21] Thomasian, A., "The Design Study of a Shared-Resource Parallel Processing System," Ph.D. Dissertation, in progress, Computer Science Department, University of California, (to be available in June 1976).

List of Abbreviations

AG:	Address Generator
AP:	Arithmetic Processor
CU:	Control Unit
IN:	Interconnection Network
ISU:	Input Switching Unit
MIU:	Memory Interface Unit
PP:	Program Processor
SCR:	Shared Computing Resource
SN:	Switching Network

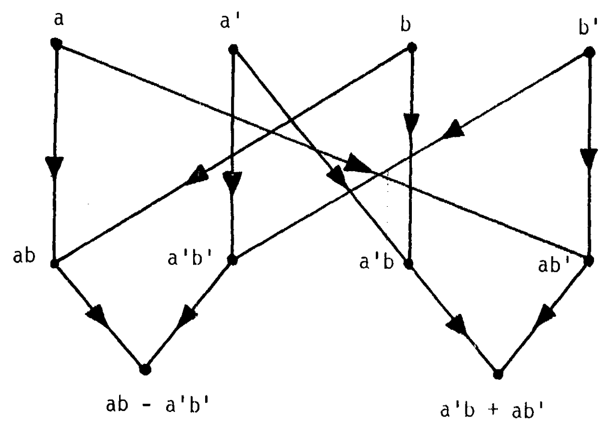
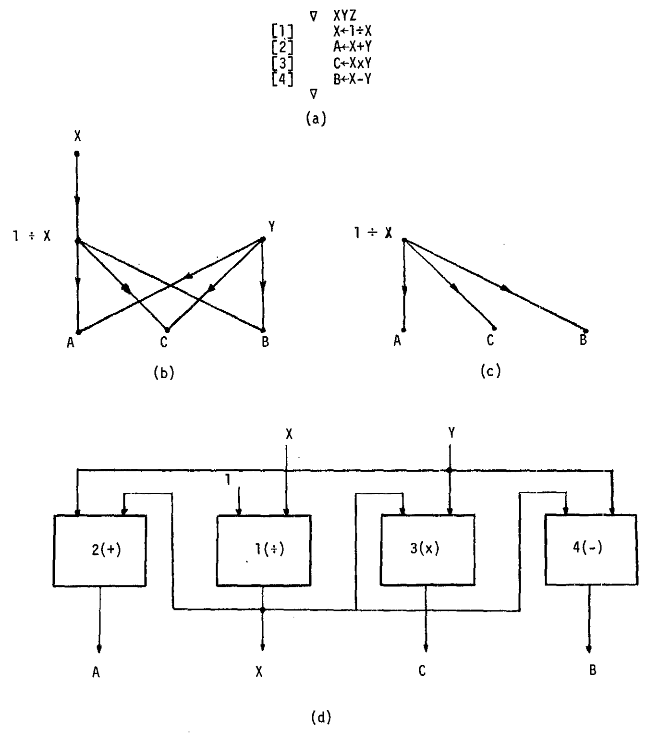


Figure 1



```
        ∇ XYZ
[1]       X←1÷X
[2]       A←X+Y
[3]       C←X×Y
[4]       B←X-Y
        ∇
```

Figure 4

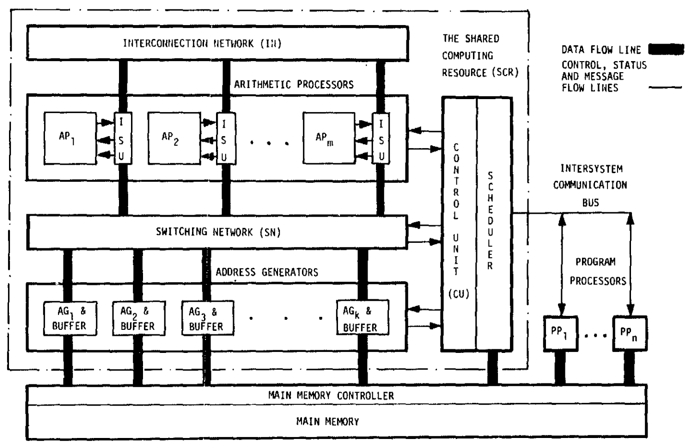


Figure 2

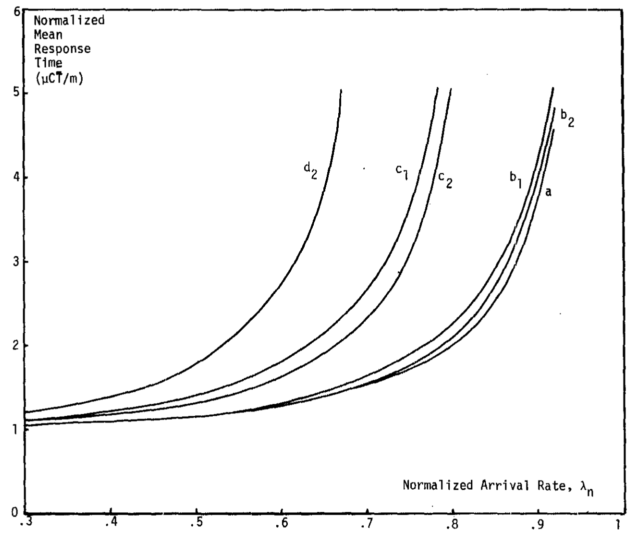


Figure 5

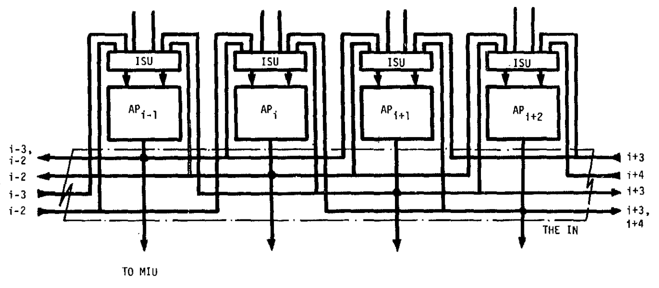


Figure 3

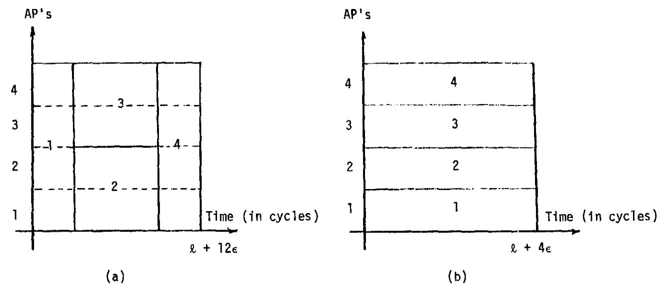


Figure 6

HARDWARE SUPPORT FOR INTER-PROCESS COMMUNICATION AND PROCESSOR SHARING*

W.S. Ford
and
V.C. Hamacher

Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Ontario, Canada

Abstract

The abstraction of a computer system as a set of asynchronous communicating processes is an important system concept. This paper indicates how the concept could be supported at a low hardware level. A new inter-process communication mechanism called a mailbox is introduced. Examples of its use as a programming tool are given. This is followed by a description of hardware features that use this mechanism as the basis of communication between the components of a complete system. These features include processor-sharing hardware capable of handling process selection and switching with high efficiency. It is also indicated how these features can take the place of conventional input/output structures.

1. Introduction

The abstraction of a computer system as a set of asynchronous communicating processes [1] is a concept which is widely accepted and used by present-day programmers. In fact, this abstraction is now recognized to the extent that most modern systems support it at either a low-level software (e.g., [2,3]) or firmware (e.g., [4,5]) level. Because it is becoming increasingly economical to implement widely-used programming features in hardware, consideration should be given to low-level hardware support for this important abstraction. Much can be gained by going to the basic hardware level, as this removes the restriction of simply implementing sequential algorithms, as is the case when firmware or software features are superimposed upon a conventional hardware structure. We shall therefore propose new low-level features which would exist, for example, at the machine instruction level of a minicomputer, or, in a larger machine, at a level below firmware which implements more complex functions. These features consist of the following:

a) Mechanisms for controlling inter-process communication and synchronization. These facilities should provide a programmer with operators capable of efficiently emulating familiar programming language synchronization primitives (e.g., semaphores [6], monitors [7]) and solving common synchronization problems. They should not need to differentiate between software processes running internal to a CPU and external processes based on peripheral devices or other CPU's. This generality will allow input/output handling and multiprocessing to be incorporated in a natural way.

b) Facilities for handling the sharing of a CPU among a number of logical processes. Process selection and switching should be automatic and fast; however, there should be sufficient flexibility for scheduling policies to be determined under program control.

* This research was supported in part by grant A-5192 from the National Research Council of Canada.

The combination of these features should relieve the programmer from the well-known problems associated with the conventional interrupt mechanism. It is essential that time overheads, especially in process switching, be kept low in order to compete with more conventional input/output structures. An attempt will be made to restrict these proposals to features whose implementation in current or foreseeable technology would be cost-effective, even for application in machines down to the minicomputer scale.

This paper will concentrate on the hardware aspects of these features, and only brief examples of their role in programming will be given. The programming aspects will be discussed further in a future publication. For clarity and consistency, Pascal notation [8] is used throughout for the description of all hardware and software concepts and algorithms.

2. The Mailbox Mechanism

The concept of a low-level mailbox mechanism has been discussed by Spier [9]. He defined the basic characteristics of any interprocess communication mechanism, then introduced a single bit "mailbox" as the "most elementary communication mechanism which would satisfy all of the requirements". His mailbox was capable of transmitting, after initialization, one one-bit message from a sender process to a receiver process. We add practicality to this mechanism with the following two extensions:

a) It is made reusable so that it can pass a stream of messages.

b) A data-carrying capability is added so that a message can convey a word of information.

A mailbox can now be considered as possessing both a state (*full* or *empty*) and contents. In Pascal notation, *mailbox* can be defined as a structured type:

```
type mailbox = record
                    state:  (empty, full);
                    contents:  word
               end
```

The sending and receiving operations on mailbox *m* can be described respectively as *PUT value AT m* interpreted as:

```
repeat until m.state = empty;
m.state := full;
m.contents := value
```

and *GET value AT m* interpreted as:

```
repeat until m.state = full;
m.state := empty;
value := m.contents
```

It must be stipulated that, following a successful until test, the mailbox should be inaccessable to other processes until the assignment operations are complete. This mechanism guarantees that every message sent by a *PUT* operation will be received by exactly one *GET* operation.

This mailbox has considerable value as a low-level programming tool for general synchronization problems. We present here two brief examples of the use of the mechanism in programming-- the implementation of semaphores and queues.

2.1 Semaphores

The binary semaphore is an important tool as it has been used widely in published solutions to many interesting synchronization problems. It has also been shown [7] to be a suitable mechanism for implementing monitors. Wirth [10] pointed out that a general semaphore corresponds to a message queue which passes only null messages. A similar concept is used here to implement a binary semaphore using a single mailbox. We consider *semaphore* as a type:

```
type semaphore = mailbox
```

and define the operations on semaphore *s* as *P(s)*, implemented as *PUT AT s*, and *V(s)*, implemented as *GET AT s*. The null *value* arguments of the *PUT* and *GET* instructions indicate that the mailbox *contents* field is unused.

With this implementation, the mailbox *full* state corresponds to a semaphore value <1, while the mailbox *empty* state (the natural initial state) corresponds to a semaphore value of 1 (the natural initial state for a mutual exclusion semaphore). A small inconsistency in this implementation is that, strictly, it should be illegal to attempt to execute a *V*-operation on a binary semaphore with value 1. Rather than detect such an attempt as an error, the mailbox mechanism will cause the violating process to be delayed until the next *P*-operation. If the semaphores are used correctly (as in compiler-generated code), the inconsistency will not arise.

2.2 Queues

The mailbox mechanism also provides for a simple implementation of FIFO queues of known maximum length, as required for buffering message streams between processes. This queueing problem has also been referred to as a bounded buffer producer/consumer problem [6,7].

A queue of maximum length *k* words is implemented as an array of *k* mailboxes, all initially *empty*. An in-pointer and out-pointer initially point to one of these locations. A queue of maximum length *k* can therefore be described as the structured type:

```
type queue [k] = record
                     store:  array [1..k] of
                                 mailbox;
                     in, out:  1..k
                 end
```

For a queue *q* two operations are defined. The operation for a producer process to append a word to the tail of the queue is *APPEND word TO q* which can be implemented as:

```
with q do begin
      PUT word AT store [in];
      in := if in < k then in + 1
                else 1
end
```

The corresponding operation for a consumer process to remove a word from the head of the queue is *REMOVE word FROM q* implemented as

```
with q do begin
      GET word AT store [out];
      out := if out < k then out + 1
                 else 1
end
```

If there is only one producer process and one consumer process operating on the same queue, the above will be correct without any need for semaphores or indivisible operations. Whenever the consumer process attempts to remove a word from an empty queue it will automatically be blocked until the queue is no longer empty. Conversely, if the producer process attempts to append a word when all *k* locations are full, it will be blocked until the consumer removes a word. In the case of more than one producer process for the same queue, it is necessary to enclose the *APPEND* code in a critical region guaranteeing mutual exclusion between producers. A similar modification applies to the case of more than one consumer.

3. Mailbox Memory

We now present a proposal for a hardware implementation of mailboxes which enables them to be used as the basis of communication between the components of a complete system. For this purpose, a physical processor is considered to be any CPU, or any hardware device which logically communicates directly with a CPU process. This includes input/output channels and some peripheral devices. In general, a physical processor may be capable of supporting more than one logical process. The principal path of communication between physical processors is mailbox memory (an array of mailboxes). A possible system configuration is illustrated in Figure 1. This shows all physical processors connected to a common bus which is managed by a mailbox memory controller. Firstly, the mailbox memory and the controller will be described, assuming that each physical processor supports only one logical process. The following section will discuss compatible hardware features for efficiently handling the sharing of a physical processor.

Mailbox memory consists of a number of addressable locations, with word size typically one or two bytes. Any mailbox location may be in either a *full* condition, in which case its contents represent some meaningful value, or an *empty* condition in which the contents are undefined and inaccessable. An additional bit for each word indicates a *full* or *empty* state. Since these state bits are accessed more frequently than the complete words, they can profitably be retained in separate higher-speed memory devices.

The mailbox memory controller receives *PUT* and *GET* requests on the bus, each request specifying a single mailbox memory address. These requests originate from either explicit CPU instructions, or from device interfaces. All are handled identically. The *PUT* operation applied to an *empty* mailbox causes a value to be passed from the processor and stored in the location, the state of that location then becoming *full*. Conversely, a *GET* operation on a *full* mailbox causes the contents of that location to be passed to the processor and the location assumes the *empty* state. The read operation on mailbox contents is allowed to be destructive.

Any attempt to execute a *PUT* operation on a *full* mailbox, or a *GET* operation on an *empty* mailbox causes a BLOCK signal to be sent back to that processor. That processor then enters a mode where it monitors the bus waiting for a WAKEUP signal for that particular mailbox. Whenever a *PUT* or *GET* operation is successfully executed, a WAKEUP signal is broadcast on the bus together with the address of the mailbox involved. When a blocked processor eventually detects the appropriate WAKEUP signal, it then reissues the original *PUT* or *GET* request.

The activity of the mailbox memory controller in processing *PUT* and *GET* bus requests can be described as the following indivisible sequence:

```
while true do
begin
   receive op for mailbox m from processor p;
   state := m.state {Retrieve the state bit};
   if (op = GET) ∧ (state = full) then
```

```
    begin {Successful GET}
        generate WAKEUP (m);
        m.state := empty;
        value := m.contents;
        transfer value to processor p
    end
    else if (op = PUT) ∧ (state = empty) then
    begin {Successful PUT}
        generate WAKEUP (m);
        m.state := full;
        transfer value from processor p;
        m.contents := value
    end
    else generate BLOCK (p)  {Unsuccessful
                              PUT or GET}
end
```

An essential feature of the mailbox memory controller is that it can be processing only one *PUT* or *GET* request at any time. Hence there may be times when more than one of the processors are competing for access to the controller. It will be assumed that such competition is resolved on a priority basis, as with conventional bus conflict resolution.

4. Processor Sharing

When the physical processor is a CPU, special provision must often be made for sharing it among a number of internal processes, each being an instance of execution of a machine program. These internal processes must be able to communicate with each other, as well as with external processes, via mailbox memory.

4.1 Process Status Table

Assume that a physical processor may support a maximum of N internal processes. Each such process is assigned a unique identifying number in the range [0, N-1], this number being an index into a hardware process status table. The organization of this table is shown in Figure 2. Naturally, only one process is executing machine instructions at any time, and the identifier of that process is held in a hardware register denoted the current process register. For every process, the status table contains a bit to indicate ready/blocked status, and a register containing a priority value. These entries are used by a despatching mechanism which, when enabled, loads into the current process register the identifier of a ready process selected according to priorities. This despatcher will be discussed in more detail later.

Each internal process is assumed to have its own partition of memory for storage of local data. Each process also has its own set of CPU registers, including program counter and memory bounds registers. The register sets for all processes can be implemented in a high-speed random access memory configuration. When accessing this memory, the most significant portion of the address is obtained from the current process register, so context switching between processes simply involves changing the contents of that register. Therefore, the only time overhead in process switching can be the despatcher delay, which will be discussed later. To briefly support the feasibility of this feature, it should be pointed out that the value of N for a 16-bit minicomputer could reasonably be of the order of 16. With 8 CPU registers this would require a high-speed random access memory of 128 16-bit words. This is not an unreasonably expensive item in current high-speed logic technology.

The execution of a *PUT* or *GET* machine instruction causes the processor to issue an appropriate *PUT* or *GET* request to the mailbox memory controller. If the request can be satisfied directly, the appropriate data transfer is made and execution of the same program continues. If, however, the request cannot be satisfied and the BLOCK signal is returned, the status of the current process is set to blocked and the program counter is not incremented. In each process status table entry there is an additional word of sufficient length to hold a mailbox address. When a process is blocked, the address of the mailbox at which it is blocked is entered in that word. After blocking of a process the despatcher is invoked, causing the processor to switch to a new process.

To handle WAKEUP signals, the processor has an asynchronous mechanism which continually monitors the bus for any WAKEUP signal. On every such signal, this mechanism searches the process status table for processes which are blocked at that particular mailbox, and if any are found their status bits are set to ready. Also, a processor flag is set, indicating that the despatcher should be invoked at the first opportunity. For fast execution of the wakeup phase, it is apparent that the "blocking mailbox" words of the process status table could be configured as an associative memory. For a system with N = 16 and a maximum of 1024 mailbox words, this would call for a relatively inexpensive 16-bit by 10-bit associative memory.

The "wakeup" time can actually be as long as one mailbox contents access time with zero effective time cost. This is because every WAKEUP signal is followed by a mailbox contents access. If the *PUT* or *GET* operation involved was initiated by this processor, then the processor will be delayed anyway until the completion of that access. If the operation was initiated by some other processor, then the "wakeup" can completely overlap program execution. It is impossible for the processor to commence executing a *PUT* or *GET* instruction until the previous mailbox operation is completed, so there is no possibility of "block" and "wakeup" modifications of the process status table clashing, provided:

a) the time for the table adjustment following a WAKEUP signal is less than a mailbox contents access time, and

b) the time for the table adjustment following a BLOCK signal is less than a mailbox state bit access time.

It is clear that a CPU must have special process management instructions for initiating, terminating, and supervising internal processes. There must at least be instructions to enable a process to modify the register contents of another process (for initializing the program counter and memory limits), to set or clear ready/blocked flags, to suspend another process by forcing it to block at a dummy mailbox, and to initialize mailbox states to *empty*. Access to these instructions should be restricted to nominated supervisory processes. There is also a need for instructions to change process priorities. To maintain flexibility in scheduling, it appears that access to these instructions should be relatively unrestricted.

4.2 Despatcher

The purpose of the despatcher is to examine all priority registers and ready/blocked flags of a processor, and load into the current process register the identifier of some ready process whose priority is no lower than that of any other ready process. If no process is ready, it must generate the signal CPUIDLE which inhibits processing. The despatching activity need only be carried out after a change has been made to the process status table, i.e., after a recognized WAKEUP signal, a BLOCK signal, a change priority instruction, or a process management instruction. A synchronization problem arises as the recognition of WAKEUP signals is not synchronized to the basic processor cycle, as the other activities are. In resolving this problem, it should be stipulated that, when the mailbox memory controller broadcasts a WAKEUP signal, it should not have to wait for responses from processors (i.e., a processor simply "absorbs" a WAKEUP signal). Nor should the speed of the mailbox memory be severely restricted by possible excessive delays on the part of processors in reacting to wakeup signals. For this reason, it appears sensible to synchronize despatching to the processor and provide the despatcher with storage to take a "snapshot" of the ready/blocked flag status whenever it is invoked. On receipt of any WAKEUP signal, the processor wakeup mechanism then has only to execute an associative search and set any required ready flags. It is then capable of immediately accepting another WAKEUP signal. When the despatcher is about to commence its cycle, it latches in the current values of the flags and uses the latched values. Further WAKEUP signals can then be accepted while the despatcher is operating, although any process made ready by such a signal cannot run until after the subsequent despatcher cycle.

The next consideration is the possibility of having despatching overlapping normal processing. Assume that the despatcher is invoked at the end of any instruction cycle in which the process status table was modified, either by the instruction itself (which may have caused a BLOCK, changed priorities, set or cleared ready/blocked flags, etc.) or by the recognition of WAKEUP signals during that period. In the simplest implementation (no overlap), a new instruction cycle is not commenced until the despatcher cycle completes. All despatching time therefore becomes processing overhead. Another approach is to permit processing of new instructions to continue while the despatcher is still operating. For obvious correctness reasons this cannot be done following a BLOCK signal or some process management instructions, but it may be possible to delay process switching if the only table changes that have been made since the last despatching cycle have been caused by priority changes or WAKEUP signals. If this is done, the effects of priority changes and wakeups will be delayed for a limited number of instructions. This will not normally affect correctness.

There exist a variety of possible methods for implementing the despatcher, e.g., sorting networks [11], associative memory [12], or microcoded sequential algorithms. To estimate the achievable speed of a despatcher, consider a simple combinatorial network implementing the required function. This is basically an N-way p-bit digital comparator, where p is the number of bits of each priority register. A fast practical configuration is a tree structure of two-way comparator elements as illustrated in Figure 3. The function of each comparator element is to compare two input priority values, and output both the higher of the values and the identifier of the process having that higher priority. At the lowest level, the priority lines must be gated with the corresponding ready flags to ensure that only processes with ready status are considered (assume priority 0 is equivalent to blocked status).

Assuming N processes, the delay time for this circuit will clearly be $\lceil \log_2 N \rceil \tau$, where τ is the delay of a p-bit comparator. This comparator is equivalent to a p-bit subtractor, and it can be shown that an achievable delay for such a circuit is $2 + 2\lceil \log_F p \rceil$, where F is the maximum permissible fan-in. A large range of priority values may be desirable for implementing, for example, several of Hoare's algorithms [7]. Assuming p up to 16, F = 4, and a gate delay of 20 nsec., this would give $\tau = 120$ nsec. For a processor supporting 16 processes, the total despatcher delay would be 480 nsec., i.e., of the same order as a memory cycle time.

4.3 Process Modules

An important system parameter is N, the maximum number of processes supported by a CPU. It is essential that N be sufficiently large for any given application, but not be excessively large because of the high cost in wasted register sets, associative memory, etc. It should therefore be a highly flexible parameter. One way of achieving this is the use of a modular hardware structure, where each of m modules contains the registers, status table, associative memory, and section of the despatcher for n processes, where N = mn. Any particular machine can then be built with as many modules as required for the particular application, and machine capability can be expanded as required by adding modules.

With this modular approach, the tree structure model of the despatcher (Figure 3) is no longer acceptable. To estimate achievable despatcher speed, consider, therefore, an array structure as shown in Figure 4. In this array, each A element is a 2-way p-bit comparator, and each B element is an n-way p-bit comparator. The time delay in an A element will be τ, and in a B element will be $\lceil \log_2 n \rceil \tau$. The total despatcher delay will therefore be the basic delay in B elements plus the time for the result to propagate through the A elements, i.e., $(\lceil \log_2 n \rceil + m)\tau$. Hence, for speed reasons, n should be large relative to m; however, for flexibility n should be small. In a realistic situation n might be 4 or 8. (It is, in fact, possible to combine modules of different n in the same system.)

Assuming a value of n = 4 and all other parameters the same as for the despatcher discussed previously, the total delay for a modular despatcher would be 720 nsec. which is still an acceptable value.

5. Input/Output

In conclusion, we shall demonstrate the role of the proposed hardware features in the driving of conventional input/output devices.

Consider, firstly, the class of devices whose basic unit of data transfer is no more than a few bytes. The class includes keyboards, teleprinters, paper tape equipment, real-time clocks, process control interfaces, etc. As was shown in Figure 1, these devices can be configured so as to communicate directly with the mailbox memory controller by *PUT* and *GET* bus requests. From a system point of view each device can therefore be considered to be executing an internal program containing *PUT* and/or *GET* statements.

For example, an output device such as a teleprinter may be considered to be executing the program:

```
while true do
begin
   GET characater AT teleprintout;
   print character
end
```

where *teleprintout* is a mailbox dedicated to that device. A CPU process can then send a character to the teleprinter by executing the single instruction *PUT character AT teleprintout*. Output to the teleprinter could be buffered using the FIFO queue mechanism as follows. Assume a queue *printqueue* of sufficient maximum length, then the main process code for emitting a character is *APPEND character TO printqueue*. An additional CPU buffer process executes the following program:

```
while true do
begin
   REMOVE nextchar FROM printqueue;
   PUT nextchar AT teleprintout
end
```

This process effectively takes the place of a conventional device interrupt service routine. For efficient device operation it should, of course, have a relatively high priority.

Input devices can be handled similarly. For example, a keyboard may be considered as executing the following program:

```
while true do
begin
   receive character from operator;
   PUT character AT keyboardin
end
```

A CPU process then receives a character from the keyboard by executing the instruction *GET character AT keyboardin*. It is assumed that if the device is in its blocked internal state it is incapable of accepting another character from the operator, e.g., the keyboard is locked. Again it is possible to buffer the device using the FIFO queue mechanism and a dedicated CPU buffering process.

Special consideration must be given to devices such as disks which require high-speed transfers of large blocks of data to or from conventional memory. With these transfers it would be unrealistic to pass all data through mailbox memory, so we assume the existence of some form of channel which controls the direct transfer of blocks of data between the device and conventional memory. However, certain communications between the channel and CPU processes will be passed via mailbox memory. These include requests for transfers, notification of completion of transfers, and notification to the CPU of any error conditions. A CPU process initiates a transfer by depositing an appropriate message in a dedicated mailbox known to the channel. Several processes can thereby share the device with conflicts being automatically resolved on a priority basis at that mailbox. To wait for the completion of its transfer, a process waits for a response from the channel at another dedicated mailbox. To handle error conditions, a convenient approach is to have channels and devices report all error conditions to special CPU processes dedicated to handling such conditions, rather than report them to the process requesting the transfer. Each special process waits at a mailbox for notification of an error and can take any required action (e.g., notify the operator). It then responds to the channel that it should either repeat or abort the transfer. It is possible to share the same error-handling process among a number of channels and/or devices. This provides a very convenient way for handling similar error conditions at different devices; for example, all console display messages regarding device states can now originate in the one process.

6. References

1. Horning, J.J. and Randell, B. (1973), Process Structuring, Computing Surveys, Vol. 5, No. 1, pp. 5-30.
2. Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. (1974), HYDRA: The Kernel of a Multiprocessor Operating System, Comm. ACM, Vol. 17, No. 6, pp. 337-345.
3. Ritchie, D.M. and Thompson, K. (1974), The UNIX Time-sharing System, Comm. ACM, Vol. 17, No. 7, pp. 365-375.
4. Liskov, B.H. (1972), The Design of the Venus Operating System, Comm. ACM, Vol. 15, No. 3, pp. 144-149.
5. Atkinson, T. (1974), Architecture of Series 60/ Level 64, Honeywell Computer Journal, Vol. 8, No. 2, pp. 94-106.
6. Dijkstra, E.W. (1968), Cooperating Sequential Processes, in Programming Languages, F. Genuys (ed.), Academic Press, New York, pp. 43-112.
7. Hoare, C.A.R. (1974), Monitors: An Operating System Structuring Concept, Comm. ACM, Vol. 17, No. 10, pp. 549-557.
8. Wirth, N. (1971), The Programming Language Pascal, Acta Informatica 1, pp. 35-63.
9. Spier, M.J. (1973), Process Communication Prerequisites or the IPC-Setup Revisited, 1973 Sagamore Conference on Parallel Processing, Syracuse University, pp. 79-88.
10. Wirth, N. (1969), On Multiprogramming, Machine Coding, and Computer Organization, Comm. ACM, Vol. 12, No. 9, pp. 489-498.
11. Thurber, K.J. (1974), Interconnection Networks - A Survey and Assessment, National Computer Conference, Vol. 43, pp. 909-919.
12. Berg, R.O. and Johnson, M.D. (1970), An Associative Memory for Executive Control Functions in an Advanced Avionics Computer System, Proc. 1970 IEEE International Computer Group Conference, pp. 336-342.

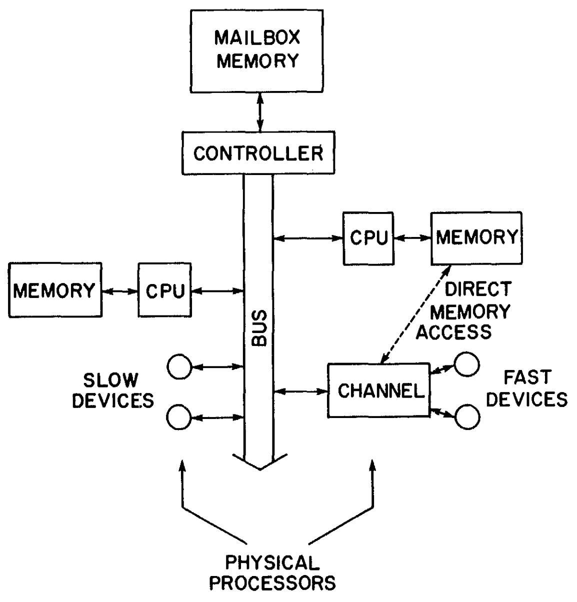


Figure 1: Possible System Configuration

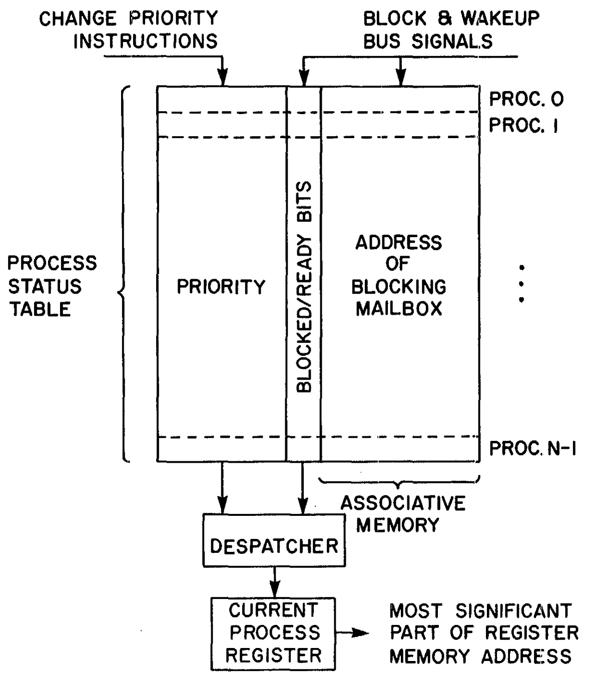

CHANGE PRIORITY INSTRUCTIONS
BLOCK & WAKEUP BUS SIGNALS
PROC. 0
PROC. 1
PROCESS STATUS TABLE
PRIORITY
BLOCKED/READY BITS
ADDRESS OF BLOCKING MAILBOX
PROC. N-1
ASSOCIATIVE MEMORY
DESPATCHER
CURRENT PROCESS REGISTER
MOST SIGNIFICANT PART OF REGISTER MEMORY ADDRESS

Figure 2: Processor-Sharing Hardware

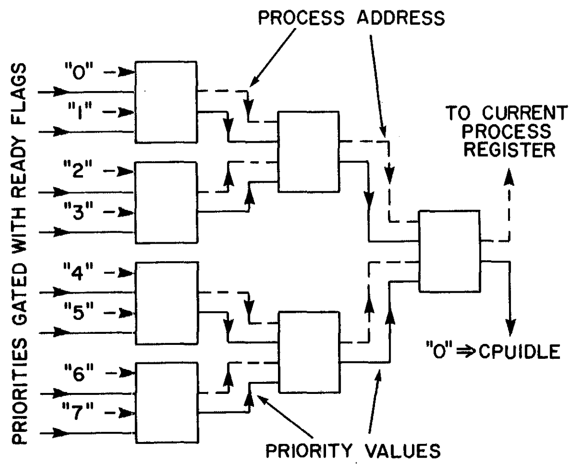


Figure 3: Combinatorial Despatcher for 8 Processes

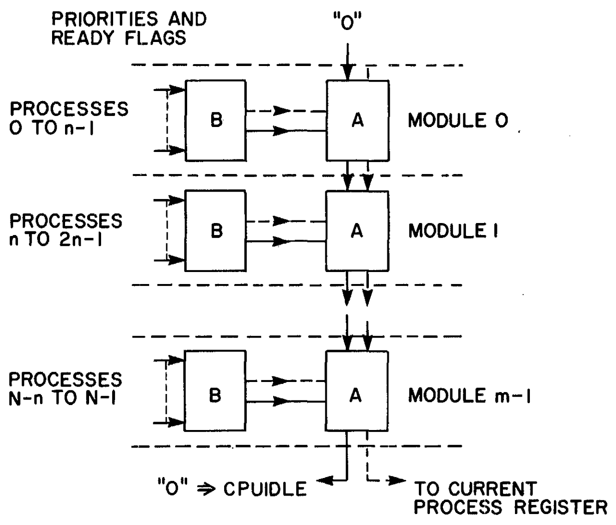


Figure 4: Structure of a Modular Despatcher

A Taxonomy of Display Processors

Ulrich Trambacz and Georg Hyla

Technical University of Berlin

Einsteinufer 35-37

D-1000 Berlin 10, West-Germany

- Abstract -

A potential customer examining computer graphics systems including a random positioning and refreshed CRT needs a lot of time and effort to form an opinion of the various marketed systems. To him not only systems appear very different, but also manuals are often cryptic and equivocal. In addition, graphic packages supplied by manufactures naturally take adventage of specific capabilities of the hardware and try to bypass their deficiencies. As a consequence, it is nearly impossible to run on a display system application programs which are written for another system.

Questions that drive from this situation are: what, of what nature, and where are these differences among display systems.

A prerequisite to investigating the differences among display systems is a uniform description. A notational system covering both the physical structure and the program level was given by BELL and NEWELL in form of Processor-Memory-Switch (PMS)- and Instruction-Set-Processor (ISP)-notation. These notations have been applied to a number of historic (ESL Console, DEC 338, IBM 2250, Evans and Sutherland LDS 1) and present (Adage AGT 400,IDIIOM/II, IMLAC PDS-4, LUNDY System 32, The Picture System, Vector General) display processors.

The historic evolution of display systems is characterized by MYER and SUTHERLAND using the term "wheel of reincarnation". A full rotation of this wheel is passed when another level of computer peripheral is added to the system, further removing the display CRT itself from the central processor. This evolution will be shown at presentation time with idealized systems at each stage described in PMS-notation.

In the fifties display devices were tied directly to the central registers of the host computer. In the early sixties a data channel was included as a link between the display device and the host or central computer. This channel soon was developed into a display processor which in turn became a full-fledged mini-computer with some graphic features. In the late sixties and early seventies this potentially never-ending cyclical process of nesting levels of graphics computer power stabilized somewhere around two full rotations of the "wheel" by the design of "stand-alone" and "intelligent" satellites.
On the first plance the ISP-description of a display processor does not disclose a taxonomical scheme. Because of the great number of graphical functions implemented in the various - and sometimes sophisticated - processors, the ISP-description certainly becomes rather bulky. But it is exactly this volume which necessitates that an unequivocal language like ISP be applied to the display hardware.

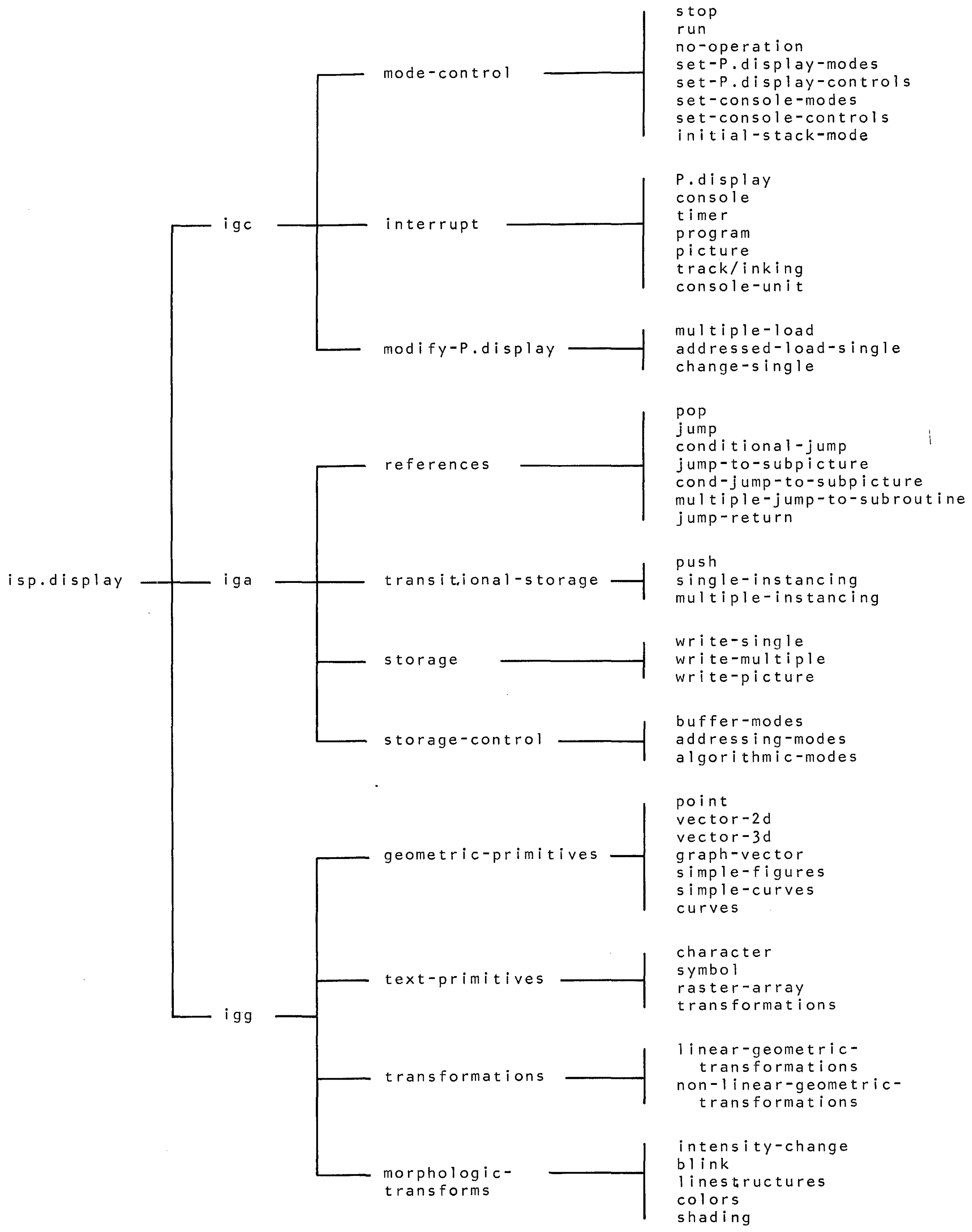
A look at the functional level of a display processor shows that certain characteristics shared by a group of instructions exist so that a classification throughout all descriptions is possible. There is one set of instructions controlling the display processor, another set handling references, and a third set driving graphic generators:

1) instruction group computing (igc)
2) instruction group addressing (iga)
3) instruction group graphic (igg)

An instruction group covers a number of functions each including a number of instructions.
This hierarchy forms an instruction tree which clarity is further enhanced by unifying the diverse instruction identifications provided by manufactures. The attached figure shows an instruction tree derived from the ten examined display systems. The major differences between the examined display systems concern discipline, which is the specification of a function, and significant achievements in computational power, especially with respect to transformation and clipping hardware.

On the functional level the systems appear more alike than on the discipline level. Essentially, no basic changes occured in terms of geometric primitives beside circle and reflection generation available in a few commercial systems.
The resemblance on the functional level might be the key to achieve standards for display systems.

Instruction Tree of Display Processors



TRAVERSING BINARY TREE STRUCTURES WITH SHIFT REGISTER MEMORIES

W. E. Kluge
Gesellschaft für Mathematik
und Datenverarbeitung mbH Bonn
5205 St. Augustin 1
Germany

The paper proposes a tree-structured shift register memory in which traversals of binary data tress in pre- or end-order are performed as sequences of two non-cyclic data permutations which move the data tree relative to a unique access port that is located in the root node of the memory tree. These two permutations, denoted A and B, emulate elementary traversal steps. Permutations A correspond to traversals between nodes of an even tree level and the next higher odd level, permutations B correspond to traversals between nodes of an odd tree level and the next higher even level. Traveling from a node to its left successor requires one permutation, traveling to its right successor requires two identical permutations in succession. Three identical permutations in succession perform a counter-clockwise cyclic traversal within a subtree which comprises a (root) node, its left successor node and its right successor node; i.e. subsequences AAA and BBB yield identity. Accordingly, if the conventional address assignment of trees applies, then the permutation (traversal) sequence that is effective on the memory corresponds to the address of the node that is actually being visited. Starting to the right of the most significant '1' bit in the address code and proceeding in the order from left to right, a '0' corresponds to one permutation and a '1' corresponds to two identical permutations; the permutation is A if the bit position is even, and B if the bit position is odd. Thus, the state of permutation may be identified by the address code. The permutation sequence that traverses the data tree is generated from single control bits, associated with every node of the data tree, which distinguish between branch nodes and leaf nodes. Whenever a branch node is being visited, then the permutation to be executed next changes with respect to the preceding permutation. Whenever a leaf node is being visited, the next permutation remains the same as the previous one.

ARCHITECTURAL SUPPORT FOR SYSTEM PROTECTION

Eduardo B. Fernandez, Rita C. Summers, and Charles D. Coleman
IBM Los Angeles Scientific Center
Los Angeles, California 90067

A set of architectural extensions, involving hardware/software interaction, is proposed to constrain the execution-time behavior of application and higher authority programs, running in a CPU of the type of IBM System 370. The extensions consist of the addition of a new state to the previous supervisor and problem states, enforcement of disciplined transition between states, hardware distinction of four data types, and a set of rules that enforce the structure of processes operating in this environment. Application of the extensions to a shared data base shows that the protection of the operating system under which it runs can be enhanced significantly, with respect to errors or attacks from the users of this data base.

THE DESIGN OF A USER-PROGRAMMABLE DIGITAL INTERFACE

James W. Gault
Electrical Engineering Department
North Carolina State University
Raleigh, North Carolina

Alice C. Parker
Electrical Engineering Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

An interface for digital computers and peripherals is described in this paper. The design process is traced, beginning with the definition of the problem environment, and the derivation of primitive interfacing functions. The functions are associated with four functional classes; data input/output, data storage, data manipulation, and control. Interface capabilities range from control over the synchronization of input and output pulse data to control over the data word widths acceptable. System limitations include technical, timing, and synchronization problems. The interface is modular, generalized, and user programmable. The control is contained in two levels: a user microprogram, and a read only nanoprogram.

SELECTION SCHEMES FOR DYNAMICALLY MICROCODING FORTRAN PROGRAMS

Philip S. Liu
Department of Electrical Engineering
University of Miami
Coral Gables, Florida 33124

Frederic J. Mowle
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

The objective of the present study has been to investigate possible methods to reduce a program's execution time by detecting and converting automatically the more frequently executed program parts, mostly inner loops, into microcode. The methods proposed were static loading of inner loops, selective loading of inner loops, overlay of inner loops, and user-aided scheme. Using Fortran programs as the test programs, a simulation program was written to measure the gain achieved by each method. A final gain between 1.587 and 4.76 was achieved by the proposed methods for memory speed ratios between 3 and 8. It was found that 90% of the final gain of the test programs could be obtained with writable control memory requirements that were less than 40% of the final requirement.

SYSTEM DESIGN OF A GRAMMAR-PROGRAMMABLE HIGH-LEVEL LANGUAGE MACHINE

Serge Fournier and Ming T. Liu
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue
Columbus, Ohio 43210

An architectural concept called Grammar-Programming is introduced which allows computers to be constructed that can directly execute a variety of high-level languages. Representing an intermediate level between the basic hardware/firmware functions of ordinary computers and the software operations of language translators, it is shown how grammar-programs can be constructed which specify the syntax and semantics of various programming alnguages. The Grammar-Programmable Machine (GPM) then uses these specifications to process directly the users' high-level language programs. In the Ph.D. dissertation* upon which this abstract is based, a model is first developed for representing the syntactic and semantic characteristics of context-free language generators, and an automaton called a Syntax Network (SN) is constructed. Next a simple, statement-directed language is introduced to express the states of the syntax network and to define the actual grammar-programming language. A simulator is then implemented which is used to test the grammar-programs written for ALGOL and SNOBOL. Finally, the architectural organization for the Grammar-Programmable Machine is described at the register-transfer level. By taking advantage of its intermediate position between software compilation and hardware interpretation of high-level languages, the Grammar-Programmable Machine is able to emphasize the best features of both techniques and to achieve a potential that neither can reach individually.

*Serge Fournier, The Architecture of a Grammar-Programmable High-Level Language Machine, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, June 1975.

SMS 101 - A STRUCTURED MULTIMICROPROCESSOR SYSTEM WITH DEADLOCK-FREE OPERATION SCHEME

Ch. Kuznia, R. Kober, H. Kopp
SIEMENS AG
D 8000 München 70
Hofmannstr. 51
Germany

The presented multimicroprocessor system has been designed to treat certain problem classes such as large systems of differential equations or online process control. It consists of a main processor and an arbitrary number of modules, each with a microprocessor, a private memory and a communication memory. The most characteristic features of the SMS 101 organization are:

- a phase structured interaction scheme (PSI-Scheme) which simplifies organizational problems
- a data communication concept, that replaces common memory by simultaneously storing data in distributed memories.

A first realization of the system comprises eight modules, but the organization allows an extension up to several hundreds of modules. Thus with present day technology a computing capacity can be achieved which is 2 or 3 magnitudes higher than that of conventional computers.

THE DESIGN OF A MULTI-MICRO-COMPUTER SYSTEM

S. H. Fuller, D. P. Siewiorek and R. J. Swan
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Continuing advances in semiconductor technology now makes practical the construction of multi-micro-processor systems with tens to hundreds of processors. We are currently involved in the design and construction of a multi-micro-processor system to experimentally investigate the problems of building and programming systems with a large number of processors. The LSI-11 microcomputer is the basic "computer module" that provides processing power and primary memory. The interconnection scheme between the computer modules allows the processors to cooperate in a true multiprocessor fashion: they can share and efficiently access all of primary memory. A number of working groups are now investigating the central problems facing the design and successful application of reliable multi-micro-processor systems and these problems will also be discussed.

DESIGN AND SIMULATION OF THE DISTRIBUTED LOOP COMPUTER NETWORK (DLCN)

Cecil C. Reames and Ming T. Liu
Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, Ohio 43210

Summary

The primary goals of this paper are two-fold: 1) to present the design and hardware implementation of the interface transmitter for the Distributed Loop Computer Network (DLCN), using a novel shift-register insertion message transmission mechanism, and 2) to discuss simulation results comparing DLCN with Pierce and Newhall loops, which verify earlier claims as to DLCN's superior performance.

I. Introduction to DLCN

The Distributed Loop Computer Network (DLCN) is envisioned as an integrated hardware/software/communication system that is to be designed and operated with distributed control. The goal for the network is to provide efficient, inexpensive, reliable, and flexible service to a localized community of semi-autonomous users in an environment of constantly changing user demands and requirements. Previous research concerning DLCN concentrated attention on the communication network, as it was felt that existing loop networks made rather inefficient usage of the loop communication channel. Accordingly, the authors designed a novel message transmission mechanism for DLCN which is much more efficient and sophisticated than those in current use [12,13]. Implemented in the loop interface hardware, the new shift-register insertion mechanism has the following important characteristics: 1) concurrent and direct transmission of variable-length messages onto the loop is possible; 2) hardware buffering of incoming messages by the interface permits nearly immediate access to the loop for locally generated messages, thus greatly reducing queueing and total transmission times; and 3) automatic regulation of loop message traffic is provided, all accomplished in a completely distributed loop network.

II. DLCN Interface Transmitter Design

A loop network is composed of a high-speed digital communication channel (1 to 10 megabits per second), arranged as a closed loop to which computers, terminals, and other peripheral devices are attached through loop interfaces (see Figure 1). Messages from a sender are put onto the loop by their interface, then travel around the loop from interface to interface until removed by the interface for the addressed receiver. Thus, the design of the loop interface and the transmission mechanism it incorporates are of extreme importance in the operation of a distributed loop network.

The loop interface can be logically partitioned into message receiving and transmitting sections (see Figure 2). The design of the interface receiver is fairly simple. It accepts incoming messages from the loop, checks their destination address fields, and either delivers them to the input buffer of the attached component if they are addressed to it or passes them on to the interface transmitter for relaying to the next interface. The function of the transmitter is to place messages onto the loop, both incoming messages relayed from the receiver and newly generated messages from the local attached component. The mechanism incorporated in the interface transmitter must be capable of merging these two message streams onto the loop without interference and without the use of centralized control.

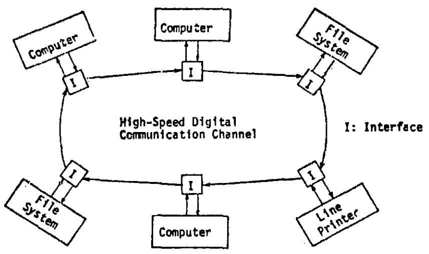


Figure 1. A Distributed Loop Computer Network

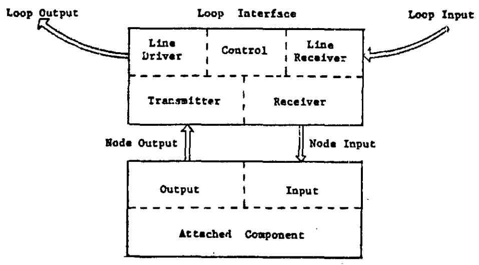


Figure 2. Functional Organization of Loop Interface

Newhall and Pierce Transmission Mechanisms

Two transmission mechanisms are in common use today for loop systems. In the Newhall loop [2,3, 14], a round-robin control passing token circulates around the loop and allows only one interface at a time the opportunity of transmitting. The selected interface may place one or more arbitrary length messages onto the loop, or may simply pass the control token on to the next interface downstream. In the Pierce loop [10,11], communication space on the loop is divided into one or more fixed-size slots. Messages are also divided into frames or packets, so that each packet will occupy one slot on the loop. The transmission mechanism is as simple as waiting for the beginning of an empty slot and filling it with a packet.

Both of these transmission mechanisms are simple to implement but suffer from certain inherent shortcomings. The control passing mechanism limits message transmission to just one interface at a time and thus results in very inefficient loop channel utilization and long message delays. Dividing messages into packets introduces other problems. Not only is there delay in waiting for an empty packet to arrive, but considerable communication space is wasted when dividing variable-length messages into fixed-size packets. In addition, all the facilities required for converting messages into packets and back again - disassembly, sequencing, buffering, and reassembly - must be provided by the loop interface or attached component. Thus, neither mechanism makes very efficient usage of the loop.

DLCN Transmission Mechanism

With the elimination of these faults in mind, a third transmission mechanism was developed by the authors for use in DLCN [8,12,13]; a somewhat related concept, although not nearly as sophisticated, was independently proposed by Hafner [6]. Called the shift-register insertion technique for the transmission of variable-length messages, it combines the best features of the two aforementioned mechanisms. This new mechanism makes possible the concurrent generation and direct transmission onto the loop of arbitrary length messages in a completely distributed network. A model of the mechanism and a detailed explanation of how it operates have been published before [12], so the explanation given here will be somewhat sketchy.

The loop transmitter must accept the two streams of incoming relayed and locally generated messages and must transmit both streams onto the loop without mutual interference. Conflicts, which might otherwise occur because of the simultaneous arrival of messages from both streams, are resolved by delaying the incoming relayed messages in a variable-length shift register located in the loop interface. Thus, as long as delay buffer space is available in the interface, the transmission of locally generated messages can have priority over the relaying of incoming messages, as the latter can be delayed if necessary. Of course, the amount of delay (worst case upper limit) and thus also the length of locally generated messages cannot exceed the size of the delay buffer.

The interface transmitter (see implementation in Figure 3) operates in one of two modes: relay and transmit. In relay mode (M=0), incoming messages are passed through the delay buffer (DB) and back onto the loop. The amount of delayed data in the buffer (indicated by counter DC) does not change while incoming messages arrive but decreases when no traffic is incoming. In transmit mode (M=1), locally generated messages (from DB) are put onto the loop. In this mode the amount of data in the delay buffer increases when incoming messages arrive and remains constant otherwise. The switch from relay to transmit mode (requested by RDY being set) can occur when both the following conditions are met: 1) the relaying of an incoming message is not in progress, and 2) the available space in the delay buffer is at least as large as the message to be transmitted.

Properties of DLCN Mechanism

The superior performance of DLCN's transmission mechanism can be largely explained by the existence of the interface delay buffer and by the ease with which the two above conditions for message transmission can be met. DLCN does not have to wait for a control token or an empty packet to arrive before sending a message. Assuming buffer space at least as large as the message to be transmitted is available, DLCN can insert a locally generated message onto the loop at the end of relaying any incoming message. Subsequent incoming messages (if any) can be temporarily delayed in the interface buffer until transmission of the local message is completed.

Thus the DLCN transmission mechanism minimizes the time a message must remain queued waiting to get onto the loop, at the possible expense of transmission time once on the loop. As the simulation results to be presented in the next section will conclusively verify, total message transmission time and queueing time are both substantially reduced by this method, together with average and maximum queue lengths. These latter facts mean that attached components can get rid of generated messages quickly and do not need large output buffers for queueing many messages.

Having partially filled delay buffers at each interface which can absorb small fluctuations tends to have a stabilizing effect on performance. Furthermore, the finite **size** of each delay buffer automatically regulates the amount of traffic which can be put onto the loop by any interface, as delay buffer space must be available before transmission can occur. However, whenever delay buffer space is available, nearly immediate access to the loop is guaranteed, regardless of other message traffic already on the loop. All these factors taken together mean that message delays are smaller and that more efficient utilization of the loop is achieved.

III. Simulation Results

Mathematical analysis of DLCN as an open queueing system with cyclic feedback will be attempted in future research (see Figure 4), but the difficulty of this task suggested that a simulation study would be more appropriate for preliminary verification of performance claims. Accordingly, simulation models were written in the GPSS/360 language for all three networks - DLCN, Pierce, and Newhall - so that relative performance could be more easily judged. The primary quantities of interest in this study were total message time and queueing time, although many other quantities were measured during the simulation. It is probably best to list all times which will be discussed and give their precise definitions:

1) queueing time - time elapsed from message generation until placement on the loop by the transmitter;
2) transmission time - time elapsed from message placement on the loop until the last character is received and removed from the loop;
3) acknowledgement time (DLCN only) - time elapsed from generation of the acknowledgement message at the receiver until the last character is received at the transmitter;
4) total message transmission time - sum of 1) and 2) only for Newhall and Pierce loops; sum of 1), 2), and 3) for DLCN.

Characteristics of All Simulation Models

The general characteristics of all three networks modeled were the same. Each consisted of 6 nodes, with each message source being an identical independent Poisson process. Messages produced at each node were uniformly addressed among the other five nodes, so that message traffic was entirely symmetric and random. Message data lengths were exponentially distributed with a mean of 50 characters; 9 additional characters of header information were added to each message or packet produced. All timing was in arbitrary character-time units, so that no particular line rate was assumed. Propagation delay on the communication channel itself was ignored, while each interface contributed 2 units of delay: 1 unit in the receiver for address checking and 1 unit in the transmitter. While these assumptions are somewhat unrealistic, most are fairly standard, and it is hoped that their simplicity will aid in later mathematical analysis of DLCN.

Special Features of the DLCN Model

The simulation models for the Pierce and Newhall loops were kept simple and unsophisticated. However, several special features were modeled for DLCN in order to correspond more closely to the situation expected in a real network. For example, a 6-character acknowledgement message which can be embedded in each data message and returned to the transmitter to indicate acceptance, error, or receiver

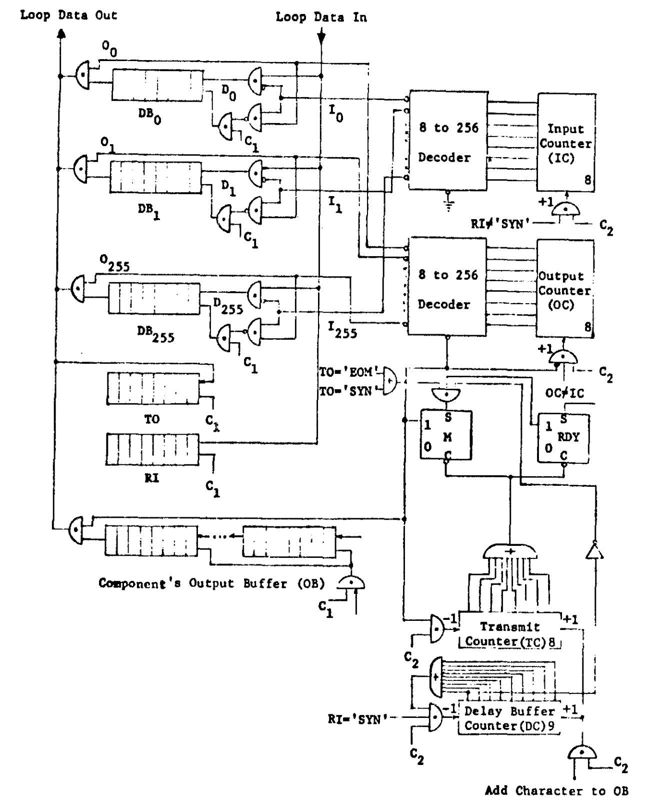


Fig. 3. Hardware Implementation of Shift-Register Insertion Transmission Mechanism

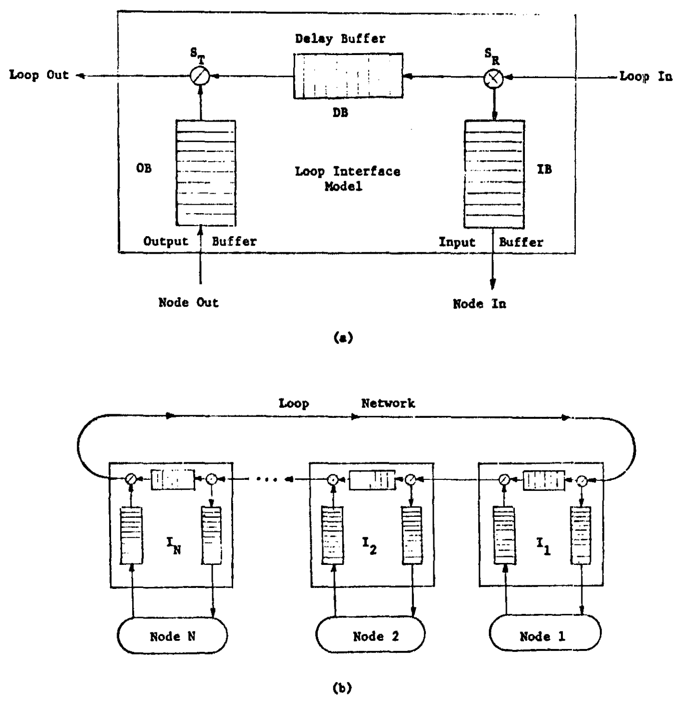


Figure 4. Loop Interface Model for Analytical Study

TABLE I

DLCN SIMULATION TIMES

IA	line	queueing time		transmit time		acknowledgement		total time		delay time*	
time	usage	mean	dev	mean	dev	mean	dev	mean	dev	mean	dev
3600	.056	2.1	14.3	58.6	47.8	13.7	12.8	74.5	51.1	10.9	29.8
1500	.138	6.4	25.8	61.3	50.1	19.1	26.5	86.7	61.8	12.5	32.3
900	.235	12.2	38.3	67.8	64.2	24.0	35.6	103.9	83.3	14.7	37.3
600	.365	19.4	45.9	79.6	78.3	37.1	59.2	136.1	113.8	19.8	42.6
480	.474	30.1	65.2	102.1	124.6	42.2	57.5	175.2	163.4	25.2	55.9
420	.543	39.9	84.2	115.1	145.3	55.1	68.7	210.2	193.4	30.4	63.0
342	.677	64.2	145.9	150.8	206.4	81.8	97.4	297.7	279.0	42.9	86.6
300	.759	101.6	222.1	210.3	334.0	91.4	93.1	404.0	450.0	56.6	130.9
270	.844	181.5	504.0	332.7	659.0	132.4	122.4	648.4	903.0	89.6	242.8
240	.937	303.1	606.0	468.9	805.0	176.8	139.9	900.6	1061.	131.2	353.0

*delay time is for each interface visited

TABLE II

PIERCE AND NEWHALL SIMULATION TIMES

Pierce Loop

IA	line	queueing time*		total time	
time	usage	mean	dev	mean	dev
2700	.098	10.9	38.3	115.2	79.3
1800	.147	18.7	53.3	124.4	94.7
1200	.200	27.9	71.5	133.7	109.6
900	.293	47.1	103.3	152.1	133.4
720	.367	69.1	111.3	174.1	142.9
600	.430	74.9	123.3	180.9	152.2
540	.479	119.1	194.3	215.3	212.9
480	.513	148.4	259.0	251.8	263.0
420	.633	215.6	271.0	326.1	299.0
360	.717	257.7	317.0	365.3	342.0
330	.762	360.9	455.0	463.7	457.0
300	.801	587.2	661.0	690.7	686.0
270	.935	1412.	1329.	1511.	1357.

Newhall Loop

IA	line	queueing time		total time	
time	usage	mean	dev	mean	dev
2100	.153	15.3	32.9	77.8	55.9
1500	.183	21.1	45.1	84.4	66.4
900	.242	38.6	65.0	101.0	78.3
600	.328	75.5	111.3	137.7	120.3
480	.378	135.2	204.9	198.5	210.1
420	.424	283.6	343.0	346.5	343.0
360	.487	611.6	558.0	675.4	559.0
330	.518	3210.	2230.	3269.	2231.
300	.511	6564.	4384.	6632.	4384.

*queueing time per packet, not per message busy has been proposed for DLCN [8] and was included in the simulation model. A receiver error rate of 1 character in 10,000 was then modeled, with messages received in error being retransmitted until accepted. A receiver busy period of 5 time units after accepting each message was also modeled (to correspond roughly to component processing time), with messages received during that busy interval being rejected and retransmitted. Finally, messages were randomly assigned priorities of 0 (lowest) to 7 (highest), with acknowledgement messages always having priority 7; these priorities were used in determining if the relayed incoming or the locally generated message should be transmitted first, rather than always giving priority to the local message.

Details of Each Simulation Model

In the DLCN simulation model, the size of each interface's delay buffer was changed from 256 to 512 characters. This change was necessary because the truncated exponential distribution used allowed message lengths of up to 500 characters, and the delay buffer must be at least as large as the longest message to be transmitted. The amount of data in each delay buffer was tabulated every 20 time units in order to obtain the distribution of buffer contents and of delay time.

For the Pierce model, a packet size of 72 characters (including the 9 characters of header information) was initially tried. Further simulation, however, showed that the optimal packet size (that which minimized total message transmission time) was only 36 characters. Since the number of packets in a message is geometrically distributed [1], the optimal packet size can also be calculated by minimizing the product of mean number of packets per message and packet size; so doing gives an optimal packet size of 36.28 characters, which agrees nicely with the simulation result. It was decided to place just one complete packet on the loop, as has been done in prior simulation studies [1,7], for this minimizes packet transmission time. Since the 6 nodes together introduce only 12 units of delay, a delay box of 24 time units was placed between the last and first interfaces so as to form an entire packet interval of 36 time units.

Two possible schemes for the control passing mechanism were investigated in the simulation modeling of the Newhall loop. In the first method, all messages in the queue of the selected interface were transmitted one by one, the control token being passed only when that queue was empty. In the second method, the control token was passed after only one message from the queue was transmitted, whether other messages remained in the queue or not. Method one led to longer queue lengths, but in all cases gave shorter total message transmission times, and thus it was adopted for the Newhall simulation model.

Comparison and Evaluation of Simulation Results

Tables I and II present the simulation quantities of primary interest. The message interarrival time for each source (node) is given, together with the average utilization level of the interface transmitter (which is effectively the same as the communication channel or line load level). All other entries are mean times as labeled and their standard deviations.

Figure 5 shows a graph of mean total message transmission time (both including and excluding acknowledgement time for DLCN) versus mean source arrival rate for all three networks. Figure 6 shows the percentile distribution of total transmission time for all three networks (emphasizing DLCN's performance) and gives an indication of worst-case behavior. Notice that at low levels of loop channel utilization, the performance of the Newhall loop closely approaches that of DLCN. As the traffic level increases, however, the Newhall loop soon falls far behind. A little thought as to the operation of the control passing mechanism in the Newhall loop explains why. If all message queues are empty (or nearly so), the control token will circulate around the loop every 12 time units (thus the minimum line utilization is .083, not 0), and the mean queueing time will be 5-1/2 time units. Compare this result with DLCN, which does not have to wait at all unless an incoming message is being relayed (and then only to the end of that particular message). For DLCN, the situation does not change as the traffic load increases, but for the Newhall loop, the control token is delayed more and more and takes longer and longer to make a complete circuit. Yet even if one transmitter is active at all times, the mean line utilization can only be about 50% (since on the average, messages only travel halfway around the loop before being received).

At low levels of line utilization, the Pierce loop does not fare as well as either DLCN or the Newhall loop. The reason for this fact is that a message always has a mean wait of half the packet interval (17-1/2 time units) and must then be transmitted in several packets (the mean measured was 2.36 packets per message, which agrees closely with analytic calculations [1,7]). At higher traffic levels, however, the performance of the Pierce loop is better than that of the Newhall loop, for the packet mechanism can allow two or more transmitters to be active concurrently (even with a single packet), as long as each transmitter finds the packet empty when it arrives. DLCN, of course, is better than either network, for it does not have to divide a message into packets and does not have to wait for a control token or an empty packet to arrive. It is interesting to note that even if the time required for an acknowledgement message to return to the transmitter is included in DLCN's total time, it still performs better than the Pierce loop at all traffic loads and better than the Newhall loop except at very low utilization levels.

For the Pierce and Newhall loops, the average transmission time on the loop is the same for any traffic load (measured by simulation as 46.7 time units per packet for the Pierce loop, 63.0 time units per message for the Newhall loop). For DLCN, however, since messages may be delayed during transmission, the mean transmission time does increase significantly with higher traffic loads (as shown in Table 1). So why does DLCN give better overall performance than either of these other networks? The answer lies in an examination of the queueing time spent by a message waiting to be transmitted onto the loop.

Figure 7 is a graph of the mean queueing times for each of the three networks. In the case of the Pierce loop, the times are for packets, not messages, and reflect the fact that when a message is being divided into packets, one packet is formed and added to the transmission queue at the start of each packet interval, until the proper number of packets have been generated. Notice the extremely small queueing times for DLCN as compared with the other two networks. Maximum and average queue lengths are similarly smaller for DLCN, since messages can get onto the loop so quickly. This very small queueing delay for DLCN more than offsets the increased transmission time and leads to its superior overall performance.

The gain in performance for DLCN is primarily due to the hardware delay buffer in each interface and to the fact that it only delays incoming messages by

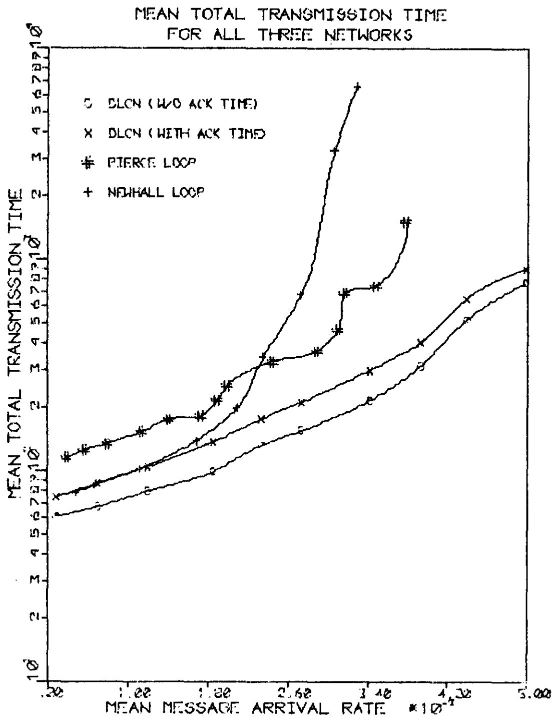


FIGURE 5.

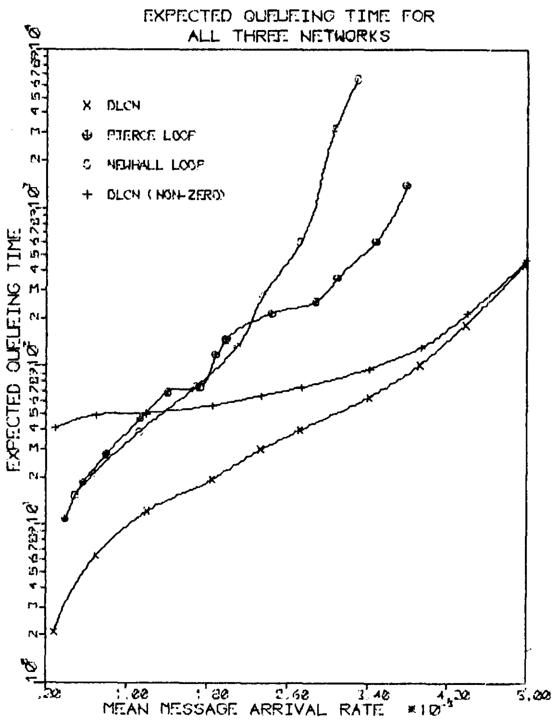


FIGURE 7

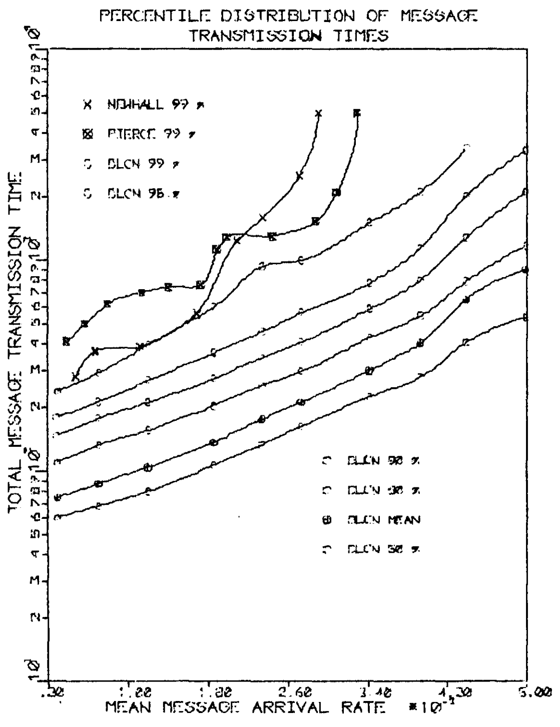


FIGURE 6.

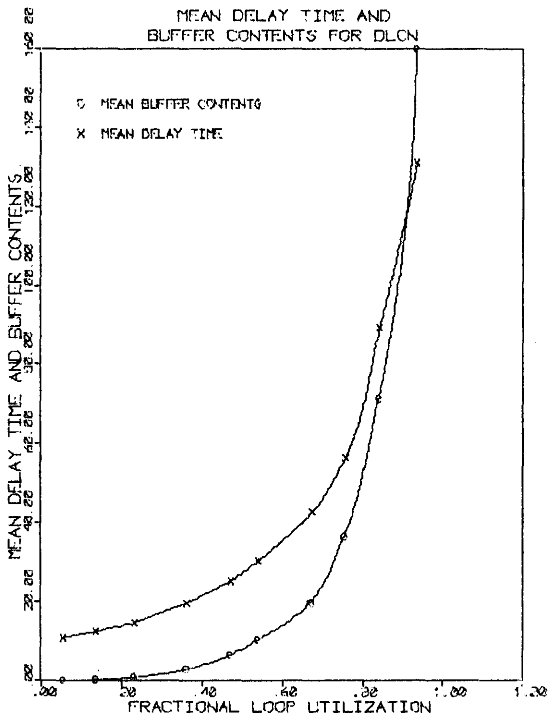


FIGURE 8.

the minimum amount necessary. Figure 8 shows graphs of the average contents of a delay buffer and the average message delay experienced, both plotted versus line utilization. Surprisingly, both graphs are rather flat and increase only slowly until very high load levels are reached. In fact, the simulation showed that a message almost never had to wait for available delay buffer space until a line utilization of .75 was reached. Even at a line load of .85, only 5% of the messages had to wait for delay buffer space to become available. Thus the delay buffer accomplishes very nicely the task for which it was designed.

Much additional simulation work has been done in studying the effects of variation of parameters, such as buffer size, mean message length, message length distribution, message priority, message arrival distribution, etc. Because of limited space, the results of all these studies cannot be presented here. However, the DLCN simulation model has proved to be fairly insensitive to most parameter variations, and no unexpected changes in behavior have been observed.

IV. Conclusions

An implicit goal in the design of DLCN and its new transmission mechanism was the desire to make more efficient utilization of the loop communication channel. At the same time, it was felt that control of the network should be completely decentralized and distributed. The hardware implementation given for the interface transmitter and the simulation results presented for its performance have shown that both of these goals have been successfully accomplished.

These two goals of efficiency and distributed control have been extended and adopted as the design philosophy of the Distributed Loop Computer Network. DLCN is intended to be an integrated hardware/software/communication system which uses distributed control to provide efficient, inexpensive, reliable and flexible service to its users. As the results presented in this paper and elsewhere indicate, much of the work needed in the hardware and communication areas has now been performed. Attention is therefore turning to the software and the design of the Distributed Loop Operating System (DLOS). A preliminary investigation of the design requirements for the low-level network software has already been carried out [8]; the areas of distributed process control and data base management are now being studied. Further research in these areas, ultimately leading to complete specification of the structure of DLOS and to implementation of a prototype version of DLCN, will be the subject of future papers.

Acknowledgement

The authors wish to express their appreciation to Dr. Marshall C. Yovits for his encouragement and constant support during the period of this research.

References

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

[2] Farber, D. J., *et al.*, "The Distributed Computer System", *Proc. COMPCON '73*, February 1973, pp. 31-34.

[3] 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, Georgia, October 1969, pp. 1-33.

[4] Fraser, A. G., "Loops for Data Communication", Computing Science Technical Report #24, Bell Laboratories, Murray Hill, New Jersey, December 1974.

[5] Fraser, A. G., "Spider - A Data Communication Experiment", Computing Science Technical Report #23, Bell Laboratories, Murray Hill, New Jersey.

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

[7] Hayes, J. F., "Modeling an Experimental Computer Communication Network", *Proc. DATACOMM '73*, St. Petersburg, Florida, November 1973, pp. 4-11.

[8] Liu, M. T., and C. C. Reames, "The Design of the Distributed Loop Computer Network", Vol. I, *Proc. 1975 International Computer Symp.*, Taipei, Taiwan, August 1975, pp. 273-283.

[9] Peebles, R. W., J. Labetoulle and E. G. Manning, "Analysis and Simulation of a Homogeneous Computer Network", Technical Report CCNG-E-30, Computer Communications Network Group, University of Waterloo, Waterloo, Ontario, January 1975.

[10] Pierce, J. R., "How Far Can Data Loops Go?", *IEEE Trans. on Comm.*, COM-20, 3 (June 1972), 527-530.

[11] Pierce, J. R., "Network for Block Switching of Data", *Bell Syst. Tech. Journ.*, LI, 3 (July/August 1972), 1133-1143.

[12] Reames, C. C. and M. T. Liu, "A Loop Network for Simultaneous Transmission of Variable-Length Messages", *Proc. 2nd Annual Symp. on Computer Architecture*, Houston, Texas, January 1975, pp. 7-12.

[13] Reames, C. C. and M. T. Liu, "Variable-Length Message Transmission for Distributed Loop Computer Networks", Technical Report OSU-CISRC-74-2, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, June 1974.

[14] 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, April 1972, pp. 29-46.

DISTRIBUTION OF FUNCTIONS AND CONTROL IN RPCNET

Paolo Franchi
IBM Scientific Center, Pisa, Italy

Summary

This paper describes the general characteristics and architecture of RPCNET, a distributed computer network for use in the Education and Research area which is being developed in Italy. This project is a joint undertaking of the National Research Council, IBM Scientific Center and a number of Universities and Research Institutes.

In order to match the variety of needs and contraints inherent in the environment in which the network must be developed and made operational, a functional rather than a system approach has been followed in the designing of the Network. The present architecture includes some modifications to the original design which were necessary in order to establish a more definite boundary between applications, interface and communication subnetwork.

The mapping of these functions into physical components is also presented as well as some types of applications.

1. - Introduction

The REEL Project was formally established in June 1974 as a cooperative effort among the IBM Scientific Center of Pisa, the Computing Center of the University of Padova and CNUCE, the Computing Center of the National Research Council. Other partners, such as the University of Torino, CSATA (the Center for Advanced Technology Applications of Bari) and CNEN, the National Council for Nuclear Energy (Bologna), joined the Project later.

The objective of this cooperation is to study a networking solution for the Italian scientific community. More specifically, purpose of the Project is to provide Computing Centers in the Education and Research area with a sensible way of sharing their computational resources, such as application programs, data sets, compilers and programming subsystems.

This objective should be attained without causing unnecessary intereference with the normal activity of the Centers and at the same time minimizing additional hardware and software requirements. For this reasons the basic features of RPCNET (REEL Project Computer NETwork) are: distributed control topology, dynamically variable configuration and nonhomogeneous nodes.

In order to satisfy this requirements the initial design of the Network [1] was based on a classical structure where each node was composed of one Front End Processor (FEP) and one or more Hosts. The communication functions could also be performed by a Host subsystem ("logical FEP") at the partners' sites where a physical FEP was not available [2]. On the other hand, the physical FEP, rather than the Host, appeared to be a suitable residence for some types of applications (for instances, terminal supports).

This approach allows a maximum flexibility in the logical and physical configuration of the Network, but tends to vitiate the original concept of FEP and Host. Moreover, using this approach, it is much more difficult to define and design a neat boundary between the communication subsystem and the application interface.

Therefore, when these problems became evident during the development of the Project, a critical revision of some points became desirable. After this analysis, the architecture of RPCNET was partially revised, concentrating on the distribution of functions and control.

This revised design is the subject of this paper. For the sake of convenience in this presentation we have borrowed some concepts and terms from the IBM Systems Network Architecture [3].

2. - Basic Elements

The general structure of RPCNET is defined in terms of two types of physical entities:

- "Network Node",
- "Network Connection",

and one type of logical entity:

- "Network Table".

A Network Node (or simply "Node") is any data processing system, including one or more physical processors, which is able to perform a minimum subset of the functions defined as "Node Functions", and which is described by a state vector in the Network Table.

Network Connection is any duplex communication channel which connetcs two Nodes and whose activity state is reported to the Network Table.

The Network Table is a representation of the static and dynamic topology of the Network in terms of Network Connections and Nodes. A subset of information contained in the Network Table is stored in each Node and is continuosly updated by the system.

The Node Functions are composed af three functional sets (Fig. 1):

- "Communication Functions"
- "Interface Functions"
- "Applications"

The Communication Functions is the minimum subset of these functions which is necessary to qualify a data processing system as a Node of RPCNET. The Communication Functions of any one Node in cooperation with the Communication Functions of the other Nodes will result in a single functional unit which is called the "Common Network" (Fig. 1).

Applications can access the Common Network only by means of a set of functions called Interface Functions. The Interface Functions of any one Node cooperate with the Interface Functions of the other Nodes and with the Common Network. The resulting functional unit is called the "Communication System" (Fig. 1).

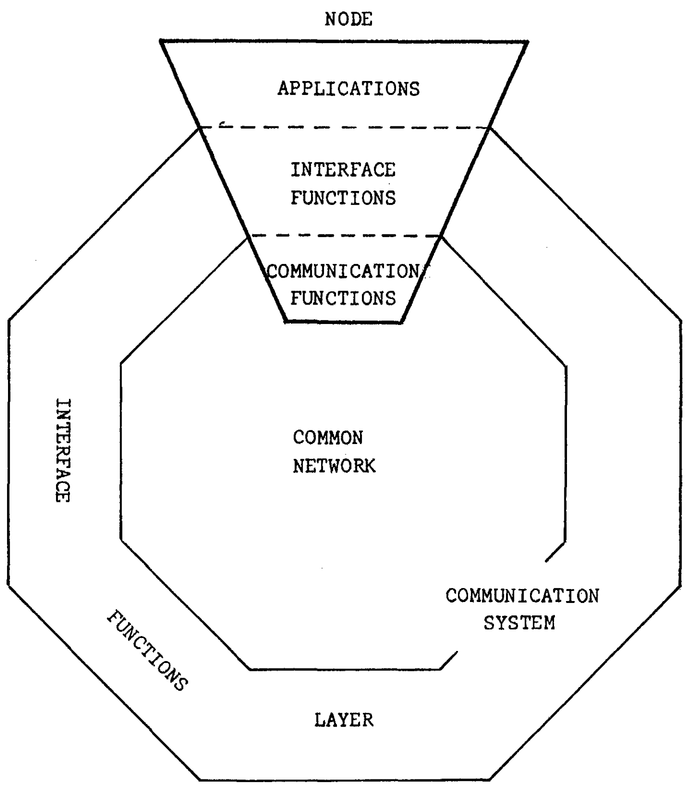


Fig. 1. Node, Communication System and Common Network relationship.

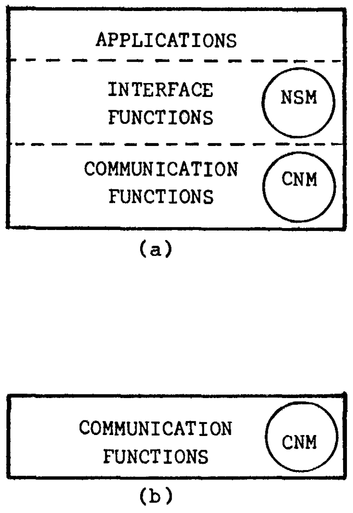


Fig. 2. One processor Nodes: (a) "Full Node"; (b) "Switching Node".

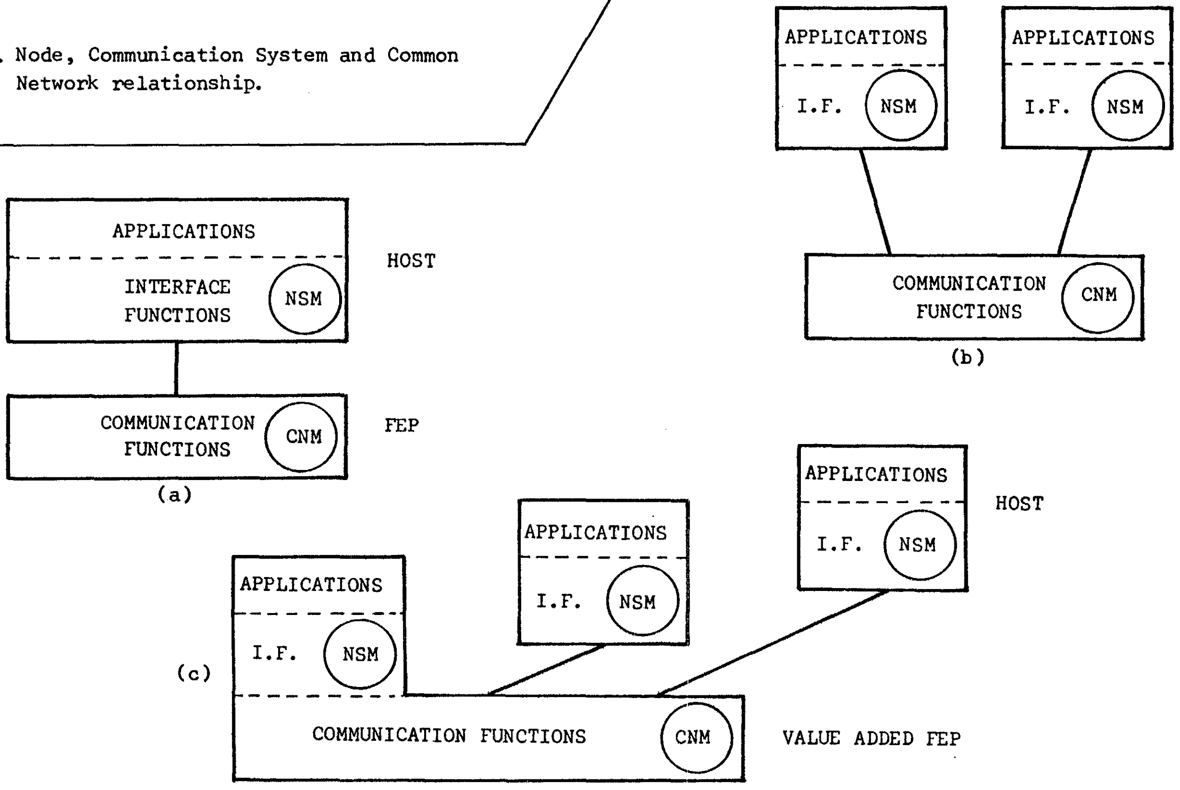


Fig. 3. Multiprocessor Nodes: (a) "Host and FEP Node"; (b) "Multiple Host Node"; (c) "Value Added FEP Node".

Applications of the Network are users of the Communication System. The Applications functional set can be empty without altering the Communication System. Similarly, no Interface Function is necessary to make the Common Network operational.

The Common Network is a general purpose packet switching system for moving unrelated data units called PIUs (Path Information Unit) from one location to another one.

3. - Communication System Control

As far as the activity of the Communication System is concerned, the control functions in each Node are performed by two network addressable components:

- "Network Services Manager" (NSM),
- "Common Network Manager (CNM).

In each Node there is one and only one CNM and one, several or, in certain cases, no NSMs.

CNMs share the control of the Common Network in a equihierarchical way. That is, the Common Network works as a symmetrically distributed control machine whose control elements are the Common Network Managers.

NSM supervises the network activities of the Applications, namely, providing them with a "Network Environment". The capability to contact other Applications is a specific service offered by this environment. NSMs share the control of the "Interface Functions Layer" of the Communication System (Fig. 1).

The way in which CNMs define the logical configuration of RPCNET is largely independent of both the physical layout of the processors to be included in the Network and of their interconnections.

In other words, the creation of CNMs should be considered as a sysgen option of the Communication System. In Fig. 2 the two cases of a single processor Node are shown. Figure 3 represents some examples of multiple processor Node. In this latter case, the connection between processors belonging to the same Node is considered as an "Internal Connection" and is not included in the architecture of RPCNET. Conversely, all the communication channels which link processors supporting CNMs are considered as Network Connections.

Each Application is supervised by one and only one NSM, which must be the NSM residing on the same physical processor which supports the Application.

4. - Information Exchange

In addition to the NSM and CNM a third type of network addressable unit exists, called the "Logical Channel Termination" (LCT). The "Logical Channel" is the basic facility by which two Applications can trade information across the Communication System. Two LCTs exist for each Logical Channel.

The port through which an Application can access the network facilities (network services and logical channels) is called "Logical Unit" (LU). Several LUs can be requested by and dedicated to a single Application. Each LU is controlled by one Application and by one NSM. LUs belong to the Interface Functions Layer.

Two LUs can communicate with each other either by means of the corresponding NSMs or through a Logical Channel, that is a pair of LCTs. This latter type of activity is called a "Session". A Session involves a pair of LUs and a pair of LCTs for a certain period of time. Multiple Sessions between LUs are not allowed.

In Fig. 4 the possible exchange of information between network addressable units are shown. The unit of information which flows from a network addressable unit to another one through the Common Network is called Basic Information Unit (BIU). BIUs are subdivided when necessary, and mapped into PIUs before entering the Common Network. The maximum length of a PIU is a constant of the Common Network.

The size of the BIUs exchanged between two NSMs or between two CNMs is predetermined so that these BIUs do not necessitate segmentation. The upper bound on the size of BIUs exchanged either in-node or exchanged between remote LCTs is the same.

A network addressability exception (monosegment BIU) is generated by the NSM or CNM of the destination Node or by a CNM of an intermediate Node when the destination LCT, NSM or CNM is not reachable (Fig. 4).

5. - Transmission Header

The Transmission Header (TH) is that part of a PIU which provides addressability, identification and sequential order of the BIU or BIU segment carried by the PIU as its text part. The network addressed of CNM, NSM and LCT is shown in Fig. 5. These addresses are carried as Destination Addressed Fields (DAFs) and Origin Address Field (OAFs) by the PIUs.

Two main types of PIUs are defined, each type being identified by a Format IDentifier (FID):

- "In-session" (FID=1), carrying a segment of a multisegment BIU;
- "Out-of-session" (FID=2), carrying an entire or non-segmented BIU.

In both types a Data Count Field (DCF) contains the binary count of the bytes in the BIU or BIU segment carried by the PIU.

In a FID=1 type PIU a Sequence Number Field (SNF) contains the sequence number of the BIU within the Session, and a SeGment Number (SGN) contains the order number of the segment within the BIU.

In a FID=2 type PIU the corresponding fields contain a Request IDentifier (RID) which identifies any Request and an Action Code (AC) which indicates to the receiving CNM or NSM the meaning and use of the associated BIU.

6. - Access to the Communication System

The application can access the Communication System by using a defined set of functions and services which are provided by the Interface Functions Layer. These functions and services can be invoked following the specifications of a Macro Language (RNAM) defined for RPCNET.

In the first step (OPENLU) an Application asks the Communication System for one or more LUs. Contact with

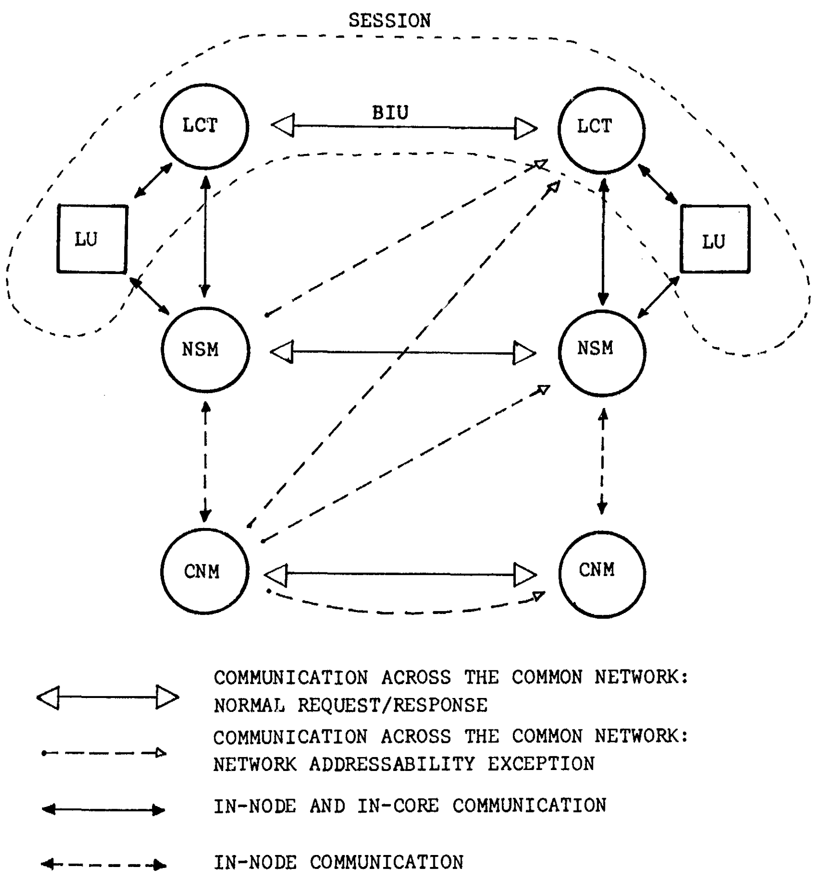

SESSION
BIU
LCT
LCT
LU
LU
NSM
NSM
CNM
CNM
COMMUNICATION ACROSS THE COMMON NETWORK: NORMAL REQUEST/RESPONSE
COMMUNICATION ACROSS THE COMMON NETWORK: NETWORK ADDRESSABILITY EXCEPTION
IN-NODE AND IN-CORE COMMUNICATION
IN-NODE COMMUNICATION

Fig. 4. Information exchange between network addressable units.

NODEID	0 0	0 0

(a)

NODEID	HOSTID	0 0

(b)

NODEID	HOSTID	LCTID

(c)

Fig. 5. Network address of: (a) CNM, (b) NSM, (c) LCT. The NODEID identifies the Node and the HOSTID identifies the subarea controlled by one NSM.

the local NSM is then established and the Application becomes addressable, through its LUs and NSM, from the Communication System.

At this point the Application can send a message to (MESSAGE) and receive a message from other Applications, can make enquiries (ENQUIRE) about the state of availability of other Applications, and can establish a Session (BIND) between one of its LUs and a remote LU. This Session can be established only if the remote Application has issued the INVITE macro.

All these services are provided by the local NSM, which has permanent contact with the remote NSMs.

Once in Session, two Applications can exchange information through LCTs, without the intervention of their NSMs. An Application can receive (RECEIVE) and send (SEND) BIUs, following preestablished rules of data flow or breaking (BREAK) these rules.

In-session BIUs are composed of two parts: the Request/Response Unit (RU) which is transparent to the Communication System, and the Request/Response Header (RH), which is built by the Communication System on the basis of a parameter list provided by the Application. On the receiving side, the RH is checked by the Communication System and passed to the Application as an end status condition of RECEIVE macro.

At the end of a Session, NSM is again invoked to close the Session (UNBIND) and release the LU (CLOSELU).

A specific set of logical rules for exchanging data in the framework of a Session, implemented on an RNAM language program, is called a "User Protocol". User Protocols are not included in the architecture of RPCNET. Each Application develops its own User Protocol in order to communicate with other Applications.

7. - Node Structure and Data Flow

Figure 6 shows the internal structure of a Node which corresponds to the case of the one processor Node illustrated in Fig. 2a.

The modification necessary to account for other cases described in Fig. 2 and Fig. 3 are obvious. In Fig. 6 the dashed arrow between NSM and CNM represents an in-core communication. This connection does not exist when the upper and lower part of the Node communicate through an Internal Connection. In this case the

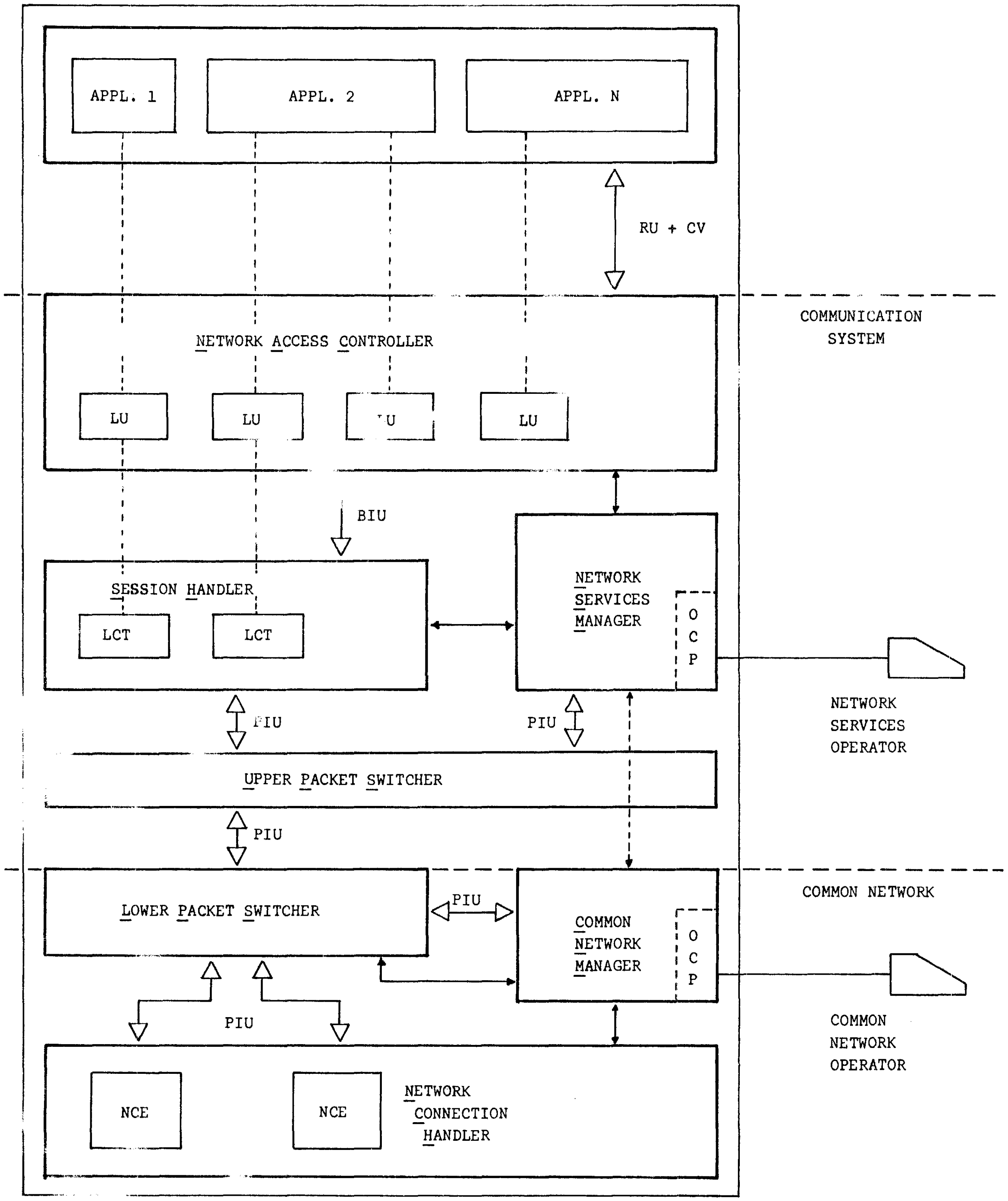


Fig. 6. Node structure and data flow.

communication task is taken over by the Upper and Lower components of the Packet Switcher.

The Network Connection Handler controls and handles Network Connections such as BSC or SDLC Data Links and Channel Attachments. Each Network Connection is represented by a Network Connection Element (NCE) to which outbound PIUs are enqueued.

The Network Connection Handler keeps idle Data Links in an active state by sending special empty frames called Hello message.

Each change in the Network Connection state is reported to CNM which is responsible for the reconfiguration of the Common Network and the updating of routing tables. The reconfiguration mechanism [1,4] is based on the flooding technique, which allows broadcasting of vital information in the most reliable way.

The Common Network Operator is conceived of as a non-automatic extension of the CNM. In the testing stage of the Network, the role of the Common Network Operator should be gradually reduced and its functions partially taken over by the CNM.

In the meantime an Operator Command Processor (OCP) allows the operator to issue commands such as: start and stop of Network Connections, display and modify Network Tables, message sending to other operators, displaying Packet Switcher queues and so on.

The Packet Switcher (Lower and Upper component) routes outbound PIUs to NCEs and inbound PIUs to the CNM, NSM or Session Handler.

The Session Handler takes care of the segmentation of out-bound, In-session BIUs and the reassembling of inbound, FID=1 PIUs. It also performs some functions of In-session data flow control. Each session is represented by a Logical Channel Element (LCT); LCTs have a one-to-one relationship to the local LUs as well as remote LCTs and LUs.

The Network Access Controller not only manages the contact point of the Applications with the Communication System, but also allocates and deallocates LUs, translates Communication Vectors (CV) into RH and viceversa. This component performs also the cross flow control of BIUs to/from Session Handler, to/from Application and to/from NSM.

The Network Services Manager supports the Out-of-session activities of the LUs and provides them with a Network Environment. The functions of this environment are the already mentioned MESSAGE, ENQUIRE, BIND, INVITE and UNBIND functions.

The NSM receives from the CNM all the information concerning changes in Common Network states which may affect the normal operation of In-session activities. This information is namely reachability of remote Nodes and availability of their NSMs.

The Network Services Operator can issue commands such as: start and stop Applications, enquire about number and level of activity of LUs and LCTs, etc. Also the functions of the Network Services operator will be gradually reduced and taken over by the NSM.

8. - Implementation and Applications

The systems which have been considered in the REEL Project for the implementation of Nodes are: VM370, OS/VS systems and System/7.

Three types of Network Connections have been considered: the System/7-370 Channel Attachment (RPQ D08112), which is also used for Internal Connections; BSC Data Links between System/7 (equipped with TPMM, RPQ D08011) and OS/VS, VM370 systems equipped with the Transmission Control Unit (TCU) of the 370x series in emulation mode. Between two Systems/7s BSC and SDLC Data Links have been considered.

Besides the implementation of the Communication System, the project plan includes four Applications, which should allow testing of the entire system. These Applications are: the emulation of the 2702 TCU and the support of 2741 Terminal (System/7); the access to the OS/VS and VM370 Spool System (System 370).

Due to the fact that User Protocols are not included in the architecture of RPCNET, an Application can communicate only with another Application which uses the same User Protocol. In this project, two different User Protocols have been defined: the "Interactive Session Protocol" and the "Spool-to-spool Session Protocol".

An alternative approach was considered where a very general User Protocol would match the variety of possible network Applications. However this has been judged too expensive.

Acknowledgments

The content of this paper is the result of a cooperative effort on behalf of several individuals. The author would like to acknowledge in particular the valuable discussions with and helpful suggestions of L. Lazzeri, L. Lenzini, C. Menchi, F. Tarini, A. Fusi, G. Gori, M. Maier, C. Paoli, R. Porinelli and G. Sommi.

References

[1]- P. Franchi, "Internode Communication Functions: Initial Design". IBM Scientific Center Technical Report 513-3527, April 1974.

[2]- P. Franchi and G. Sommi, "RPCNET Features and Components". Proceeding of the European Computer Conference on Communications Networks, London, September 1975.

[3]- IBM System Reference Library, "System Network Architecture". Form No. GA27-3102.

[4]- William D. Tajibnapis, "The Design of a Topology Information Maintenance Scheme for a Distributed Computer Network". Proceedings of the ACM Annual Conference, November 1974.

EFFICIENT MESSAGE ROUTING IN MEGA-MICRO-COMPUTER NETWORKS

Larry D. Wittie
Computer Science Department
State University of New York at Buffalo
Amherst, New York 14226

Abstract

With rapidly developing microprocessor technology, we can anticipate an entire microcomputer being contained on a single, low-cost LSI chip. It will be technically and economically feasible to interconnect thousands or millions of these microcomputers to form a very large and powerful machine -- a Mega-Micro-Computer (MMC). This paper defines a system of interlocking buses allowing dense message flow within an MMC. Each microcomputer shares two buses; each bus is shared by sixteen computers. There is a simple algorithm for optimal routing of messages. Data activity on each bus is analytically determined as a function of network size and the spatial distribution of messages between nodes. MMCs are about equally efficient whether connected by buses or by Pierce rings. For equal line costs, an MMC can allow 200 times denser message flow than a million computer network structured like Illiac IV.

1. Introduction

Single LSI chips already can contain either 16K bits of memory or sophisticated microprocessors. Within a few years, both the memory and logic of a complete microcomputer will be combined on a single LSI chip. Microcomputers of at least the power of a PDP-8 should cost only a few dollars.

There already is much interest in high speed computers built as networks of microcomputers.[1,2,3] Commercial production of an array of 512 Intel 8080 microprocessors has recently been announced.[4] Fast network computers are needed for weather modeling and satellite data reduction.

This paper presents a communication structure for efficiently linking thousands or millions of microcomputers to form a Mega-Micro-Computer (MMC). The most difficult problem in MMC design is allowing for frequent messages among processors. By presuming messages are generated at the same rate throughout an MMC, this paper analyses message delays and local differences in communication line activity inherent to network topology.

2. Characteristics of MMC Components

This paper assumes that each microcomputer in an MMC has a 16-bit CPU, 16K to 64K words of 16-bit RAM, and two bus port controllers, all on a single chip. Each port is capable of both input and output access to its shared, external communications line. Each microcomputer node can process its CPU task and two port messages concurrently. Most messages received at a node are just relayed from one bus to another on the way to a different node.

The shared communication lines linking microcomputers will be called buses in this paper. Each message is presumed to contain destination information, to have exclusive use of each bus on its route for activation time Ta, and to be relayed to another bus only after delay time Td. The bus times Ta and Td are constant throughout the network. All connections within an MMC are local enough that propagation delays are almost insignificant. This paper assumes that activation time Ta is short enough that bus access delays can be ignored. A bus is said to be overcrowded if messages try to access it more rapidly than once every Ta. One measure of network activity tells the minimum global intermessage interval that avoids local overcrowding.

This paper does not specify exact message protocols nor distributed line control methods, such as discussed by Nisnevich.[5] Digital communications techniques such as the buffer and forward loops of Pierce[6] (multiple messages, fixed length), Newhall[7] (one message, variable length), or Reames[8] (multiple messages, variable length) or such as the fast time division multiplexing Collins-system[9] may be used instead of buses. The times Ta and Td can be adjusted by factors of 1/N and B, where N is the number of simultaneous messages and B is the number of buffering nodes on each line.

3. Nested Groups of Computers Sharing Buses

Microcomputer nodes in an MMC network are assumed to lie regularly spaced in a toroidally continuous, plane square of side=(2S+1). The x or y distance between adjacent nodes is 1. Buses connecting the nodes logically subdivide the physical space into nested square groups of 16, 16^2, 16^3,..., $16^{(J+1)}$ computers, as shown in Figure 1. Spanning but not leaving each $16^{(i+1)}$-group are level-i buses ("i-buses"), each connected once in each inner 16^i-group. Each complete bus in the network, regardless of level, is shared by exactly 16 regularly spaced computers. The higher the bus level, the wider the physical spacing between connections. The smallest are the level-0 buses, one local to each 16-group in the network. Level-J buses are the longest, each spanning the entire network. Buses near the edges of the MMC square may be incomplete, with fewer than 16 connections.

In Figure 1, the single, double, and triple lines demarcate 16^1-, 16^2-, and 16^3-groups respectively. Nodes appearing in Figure 1 are connected to 24 different 0-buses, 16 1-buses, 64 2-buses, 48 3-buses, and 24 4-buses. However, only one 1-bus, one 0-bus and parts of a 2-bus and another 1-bus are explicitly shown.

Each node is connected to two buses, one local and the other higher level. The numbers in Figure 1 give the level of the non-local bus shared by each node. Since the highest bus level(J) is 4, the MMC network in Figure 2 contains 1 million (16^5) computers. The nodes marked with an X would be wired to level-5 and higher buses in larger MMCs.

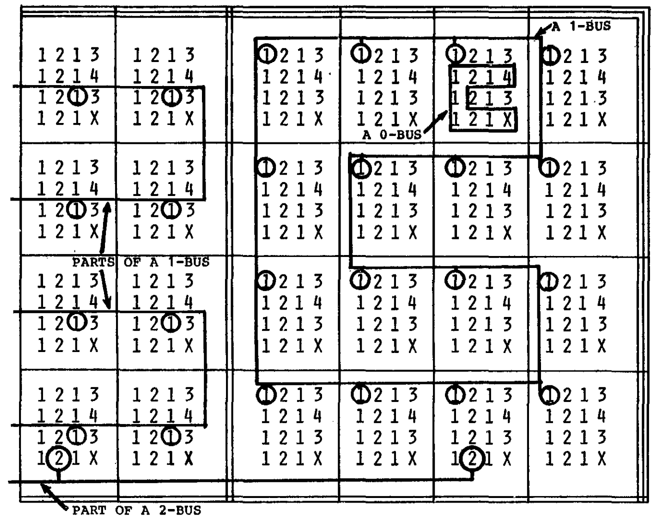


Figure 1 Logical Groups of Computers Showing Levels of High Buses Shared By Some Nodes of a $16^5 (=10^6)$ Micro-Computer Network for Highest Level (J)=4.

The hierarchy of overlapping buses re-organizes the physical plane of MMC nodes into a (J+1)-dimensional hypercube of side=16. Each node has a (J+1)-place hexidecimal index, or hyperspace address. Level-i buses run parallel to the i-th hyperdimension axis; the indices of nodes on the same i-bus differ only in the i-th place. The 3-place indices in Figure 2 allow addressing of 4096 ($=16^3$) computers in a network with buses of levels 0 through 2. Each node index in Figure 2 is subscripted by its high bus level.

The allocation function ALLH(n) determines the level of the higher bus connected to node n. ALLH(n) is the least i for which $n \bmod 2^i = 2^{i-1}-1$. It gives the position of the rightmost 0 in the binary representation of n. ALLH(01F)=ALLH(011111)=6 and ALLH(00A)=ALLH(1010)=1. For the fraction $1/2^J$ of the nodes whose ALLH value exceeds J, either the higher bus port is left disconnected or it is connected to some extra J-bus.

4. Message Routing In an MMC Network

Messages between computers not sharing a bus must be relayed through intermediate nodes. There is a simple algorithm to select a route using the fewest buses. It is equivalent to that of Pierce[6] for multiple ring systems. Unlike Brandenburg's[10] scheme for Pierce rings, no directory is needed. Only the current and final node addresses are needed. The path length varies as log(Dn), where Dn is the absolute difference between the two node indices.

The MMC routing algorithm is as follows:

1. The final destination is the immediate destination.
2. If the current and final addresses are identical, the message has arrived. Otherwise if the current and immediate addresses differ leftmost in the i-th hex position, an i-bus is needed.
3. If the current node is on an i-bus, then relay the message to the node with the same i-th address place as the immediate destination. Repeat from step 1.
4. Otherwise, overwrite the lower i bits of the current address with the high i bits of the pattern 011...1 to find the nearest node on an i-bus. Make this i-bus node the immediate destination. Repeat from step 2.

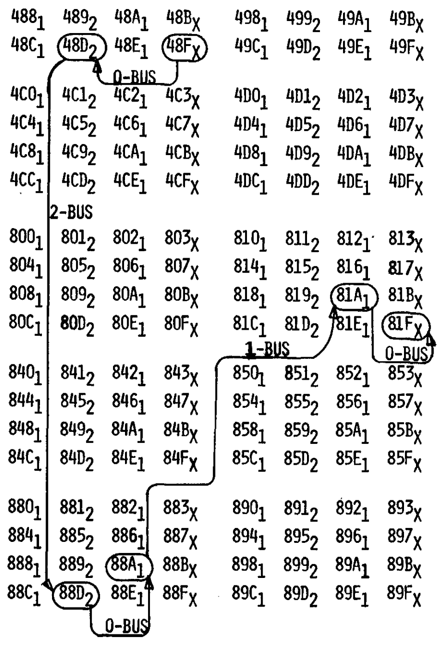


Figure 2 Bus Path From Node 48F to Node 81F in Part of a Network of 4096 Computers With Bus Ports Higher Than J=2 Unattached(X).

LOCATION(L)	IMMEDIATE GOAL(G)	DECISION
4 8 F 0100 1000 1111	8 1 F 1000 0001 1111	2-BUS NEEDED 2-BUS NOT AVAILABLE
0-BUS	4 8 D 0100 1000 1101	NEW SUB-GOAL TRANSFER ON 0-BUS
4 8 D 0100 1000 1101 2-BUS	8 1 F 1000 0001 1111	2-BUS NEEDED TRANSFER ON 2-BUS
8 8 D 1000 1000 1101	8 1 F 1000 0001 1111	1-BUS NEEDED 1-BUS NOT AVAILABLE
0-BUS	8 8 A 1000 1000 1010	NEW SUB-GOAL TRANSFER ON 0-BUS
8 8 A 1000 1000 1010 1-BUS	8 1 F 1000 0001 1111	1-BUS NEEDED TRANSFER ON 1-BUS
8 1 A 1000 0001 1010 0-BUS	8 1 F 1000 0001 1111	0-BUS NEEDED TRANSFER ON 0-BUS
8 1 F 1000 0001 1111	8 1 F 1000 0001 1111	ALL DONE

Table A Routing From Chip(48F) To Chip(81F)

This algorithm uses primary bus moves to transform the message origin address into the destination address, one hex place at a time, higher places first. Interposed before each primary move are secondary bus moves to reach a node on the primary bus. Since each node is on only one high bus, at least a 0-bus secondary move is needed after each.

Figure 2 shows the buses used by a message from node 48F to node 81F. The nodes subscripted by X are not connected to any high bus. Table A lists the algorithm steps needed to send the message as shown in Figure 2.

5. Message Activity Rates On Buses

One can analyse message activity on MMC buses if all nodes generate messages at the same rate M and distribute them over the plane with the same annularly symmetric function MDEN(r), where r is the maximum X or Y distance between the sending and receiving computers. Because of these homogeneity assumptions, all MMC buses at the same level are equally active. Bus activity differs from level to level because of secondary moves on low buses and because of high bus moves for long distance messages. A detailed analysis of MMC bus activity has been developed elsewhere[11] and is summarized below.

The calculation of a precise expression for activity on each MMC bus requires summations over all messages from one computer and over all computers in the network. The space around the central computer (0,0) of a (2S+1)-square MMC can be covered by concentric square annuli of distance r from the center, for $1 \le r \le S$. Because MDEN symmetry is assumed, messages from the central node to nodes on the same r-annulus are equally frequent.

The probability that a message from (0,0) to (x,0) exits from a 16^i-group is $\min(1, x/4^i)$. The probability than an i-bus is the highest needed for a message from (0,0) to (x,y) on the r-annulus

$$\begin{aligned} &= (x+y)/4^i - xy/16^i, && \text{if } r < 4^i; \\ &= 1 && \text{, if } 4^i \le r. \end{aligned}$$

Define:

$$\begin{aligned} \mathrm{POLY}(r,i) &\equiv (r-4^i)(r/2-4^i)/16^i \\ &\qquad \text{for } 0 \le i < J,\ 1 \le r \le S; \\ &\equiv 1, \text{ for } i=J,\ 1 \le r \le S; \\ &\equiv 0, \text{ otherwise.} \end{aligned}$$

The expected number of high i-bus uses because of one central message to some node on the r-annulus

$$\begin{aligned} &= \mathrm{POLY}(r,i+1) - \mathrm{POLY}(r,i), && \text{if } r \le B^i; \\ &= \mathrm{POLY}(r,i+1) && \text{, if } B^i < r. \end{aligned}$$

Summing over all r-annuli, the expected rate of high i-bus uses from all central messages is:

$$\begin{aligned} \mathrm{MHIX}(i) &= 8M \sum_{r=1}^{4^{i+1}} r \cdot \mathrm{MDEN}(r)\mathrm{POLY}(r,i+1) \\ &\quad - 8M \sum_{r=1}^{4^{i}} r \cdot \mathrm{MDEN}(r)\mathrm{POLY}(r,i), \\ &\qquad\qquad \text{for } 0 \le i \le J; \\ &= 0 \qquad\qquad \text{, otherwise.} \end{aligned}$$

Two hex indices differing in their I-th place, have a 15/16 probability of differing in any lower place. A message requiring a high I-bus use has a (1-1/16) probability of requiring a primary i-bus use, for $0 \le i \le I$. For the ALLH defined in this paper, every use of a bus on levels (4i+1) to (4i+4) must be set up by a secondary use of an i-bus.

Denoting a move on an i-bus by the vector MV_i and the vector sum of moves resulting from a high i-bus move by $HIMV_i$, the scalar count BUSE(i) of all i-bus uses is implicitly defined by

$$\sum_{i=0}^{J} \mathrm{BUSE}(i) * MV_i \equiv \sum_{i=0}^{J} \mathrm{MHIX}(i) * HIMV_i,$$

with $\mathrm{BUSE}(i) \equiv 0$, for all $i > J$.

By equating the coefficients of each MV_i, it can be shown that

$$\mathrm{BUSE}(i) = \mathrm{MHIX}(i) + \mathrm{MBAC}(i+1) + \sum_{k=4i+1}^{4i+4} \mathrm{MBAC}(k),$$

with $\mathrm{MBAC}(i) \equiv \mathrm{BUSE}(i) - \mathrm{MHIX}(i)/16$, for all i.

Since each bus is shared by 16 computers and all buses on the same level are equally active, the expected rate of activity of each i-bus from all messages is

$$\mathrm{BACT}(i) = 16 \cdot \mathrm{BUSE}(i)/\mathrm{BCON}(i),$$

where BCON(i) is defined as the fraction of nodes connected to some i-bus. For the ALLH defined previously,

$$\begin{aligned} \mathrm{BCON}(i) &= 2^{-i}, \text{ if } 0 \le i < J; \\ &= 2^{-J}, \text{ if } i=J \text{ and higher ports unused;} \\ &= 2^{-(J-1)}, \text{ for } i=J \text{ and higher ports used for J-buses.} \end{aligned}$$

The only factor needed for a closed form expression for BACT(i) is the exact form of MDEN(r). Defining MDEN(r) for a given network depends on knowing the computation tasks being performed. However, we can select a class of MDEN distribution functions which cover the range of likely message distributions:

$$\mathrm{MDEN}_P(r) = C/P^r, \text{ where } C = 1/\sum_{r=1}^{S} (8r/P^r).$$

$\mathrm{MDEN}_1(r)$ is uniform, modeling a random access memory with no locality. $\mathrm{MDEN}_2(r)$ decays so rapidly with distance that 99% of all messages reach the nearest 360 neighbors of any node; for P=8, 99% reach the nearest 48.

6. Message Transmission Efficiency

Two measures, MDLY and MINT, of MMC message transmission efficiency are derived from BUSE(i). MDLY, the average message delay, is proportional to the average number of buses used to relay a message:

$$MDLY=Td \sum_{i=0}^{J} BUSE(i)/M.$$

MINT is the minimum interval allowed between messages from the same computer so as not to overcrowd the busiest bus:

$$MINT=Ta \cdot \max_{0 \le i \le J} BACT(i)/M$$

The larger MINT is, the less frequently computers may communicate.

In general, message density functions such as $MDEN_2$ and $MDEN_8$ greatly localize messages, causing most to use only low level buses. All destinations are equally likely for $MDEN_1$. Since most nodes are distant, the average message for $MDEN_1$ uses high level buses nearly as often as low level buses.

Table B shows values of the measures MDLY and MINT for MMC networks containing from 25 to one billion (10^9) computers. Delay and interval measures for networks of more than 1000 computers are nearly constant for local distributions. For problems with only local messages, a unit message is delayed only 2 Td times and millions of computers can each send messages as often as once every 24 Ta times.

NETSIZE	J TO SPAN (B=4)	$MDEN_8=C/8^R$		$MDEN_2=C/2^R$		$MDEN_1=C/1^R$	
		MDLY IN TD	MINT IN TA	MDLY IN TD	MINT IN TA	MDLY IN TD	MINT IN TA
25	1	1.8	21.6	1.9	22.8	2.0	23.4
1089	2	2.0	23.6	2.8	29.8	4.3	41.3
4225	3	2.1	24.1	2.9	30.8	5.5	58.4
25921	3	2.1	24.1	2.9	30.8	6.3	60.0
1002001	4	2.1	24.2	3.0	31.1	8.4	120.0
25010001	6	2.1	24.3	3.0	31.3	14.0	474.0
1024064001	7	2.1	24.3	3.0	31.3	18.7	960.0

Table B Delay and Interval Measures For Messages Distributed Locally ($MDEN_{8,2}$) And Widely ($MDEN_1$) if $BCON(J)=2^{-(J-1)}$

For uniform distributions, maximum activity occurs on the next to highest (J-1) bus and requires MINT about equal to $15*2^{J-1}$. Average message delay increases roughly linearly with J. Almost all problems require no worse than uniform message distribution over a network. In even a randomly accessed network of one million (10^6) computers, average message delay is only 8 Td and each computer can send a message every 120 Ta.

7. Way(s) To Connect One Million Computers

As a specific example, consider an MMC of one million (10^6) computers spaced 5 cm apart in a square array. The highest level (J=4) buses are less than 300 meters (6*1000*.05) long for a maximum propagation delay of 2 microseconds (micsec) (at 15cm/nanosec.) Presume the average message packet has 3 words (48 bits) of header addresses and 13 words of information for a total of 16 words (256 bits).

7.1 MMC Networks With Buses

First presume each shared communication line is a bus with 16 data lines and 4 address lines, each 1 Megabit/second (Mbs). Presume each message takes 2 micsec for propagation delay, 1 micsec for the addressed port to buffer each word and to pass it to its companion port, and a final 2 micsec for starting the next relay. For a 16-word message, the bus times are Ta=18 and Td=20 micsec.

For locally distributed messages ($MDEN_8$), MDLY=2.1 and MINT=24.1. Once every 440 micsec, each computer can originate a 16-word message (208 bits of data) that will take about 42 micsec to reach its destination.

For widely distributed messages ($MDEN_1$), MDLY=8.4 and MINT=120.0. Each node must wait 2200 micsec between 16-word message, each relayed for 170 microseconds. Because of the factor of ten difference in MDLY and MINT for million computer networks, the intermessage interval is about ten times the individual message delay if buses link the nodes.

7.2 MMC Networks With Rings

Alternately, presume each shared line is a 20 Mbs Pierce ring allowing up to 16 simultaneous messages. Let each message be a 264 bit block: 8 ring control bits, 48 address bits, and 208 data bits. Each port must have a fast (say, 50 nanosecond per bit) shift register through which all messages pass. The path delay for a message passing through an average of 8 buffers is 106 micsec (264*.050*8). Allowing 4 micsec extra for relaying to another ring, the message delay Td is 110 micsec per ring. At saturation the ring can carry 16 simultaneous messages, yielding an effective line activation time Ta of 6.6 micsec (106/16) per message.

For local spread ($MDEN_8$) the minimum intermessage interval is 160 micsec and average message delay is 230 micsec. For wide spread ($MDEN_1$) the interval is 800 micsec and the delay is 925 micsec for a 16-word message. In a MMC network 256-bit messages can be 3 times as frequent but each are delayed 5 times as long on rings as on buses. The ring delay can be decreased by using faster shift registers. However, the delay and interval are much more nearly equal for rings.

Ring linked structures require fast buffering in each node and ring synchronization hardware. Short messages are as expensive as long. Because line messages pass through all nodes in a ring, open failure of a single node will break a ring but not a bus. In a bus structure, neighbors may route messages around a failed node.

7.3 Solomon Networks

For contrast, consider a SOLOMON-type network of one million computers each linked to its four neighbors as in the Illiac IV[12]. Since there are the equivalent of 4/2 links per SOLOMON node versus only 2/16 links per MMC node, each link must be 1.25 Mbs to have an average of 2.5 Mbs per node. To shift a 256-bit message over one node takes 205 micsec=Td=Ta. The average path length for $MDEN_1$ is 1000 nodes and for $MDEN_8$ is 2 nodes. For local messages, the message delay and interval are each 410 micsec; for widespread messages, they are 205,000 micsec.

TYPE OF NETWORK OF ONE MILLION (10^6) COMPUTERS		MMC NETWORK GROUPED BY 16 WITH BUSES	MMC NETWORK GROUPED BY 16 WITH PIERCE RINGS	SOLOMON NEAR-4 NETWORK WITH LINES
BUSES/COMPUTER IN NETWORK		0.125	0.125	2.000
LOCAL DISTRIBUTION $MDEN_8 = C/8^R$	INTERVAL= MINT IN MICSEC	440	160	410
	DELAY= MDLY IN MICSEC	42	230	410
GLOBAL DISTRIBUTION $MDEN_1$ = CONSTANT	INTERVAL= MINT IN MICSEC	2200	800	205000
	DELAY= MDLY IN MICSEC	170	925	205000

Table C Minimum Intermessage Interval and Average Message Delay For 256-bit Messages in Networks Linked With An Average of 2.5 Mbs of Input/Output Line Capacity Per Computer.

The SOLOMON machine is about 2 times slower even for local messages and 200 times slower for messages distributed uniformly over one million nodes. Table C summarizes the differences in minimum intermessage interval and average message delay for the three network structures just described.

8. Conclusions

A hierarchically linked network of thousands or millions of microcomputers can form a very powerful, very flexible computing system. Groups of individual computers may dynamically partition or unite themselves to solve a changing mixture of many small or a few large problems. Over a range of network sizes from 10^2 to 10^9 computers, grouping computers 16 to a bus provides an efficient tradeoff between minimum intermessage interval and average message path delay time. Messages are transmitted much more rapidly and at lower cost in MMC networks than in SOLOMON-like networks. Because of myriad alternate paths, grouped networks can bypass failed components. The advent of truly parallel mega-micro-computer networks will introduce new fields of basic algorithms, operating systems, and machine architecture.

References

1. V.M. Glushkov, et al, "Recursive Machines and Computing Technology", Information Processing 74, Vol. 1, North Holland, 1974, pp. 65-70.

2. C.H. Radoy, and G.J. Lipovski, "Switched Multiple Instruction, Multiple Data Stream Processing", Proc. 2nd Symp. on Computer Architecture, Jan. 1975, pp. 183-187.

3. E.D. Jensen, "A Distributed Function Computer for Real-time Control", Proc. 2nd Symp. on Computer Architecture, Jan. 1975, pp. 176-182.

4. R.A. Frank, "Imsai Arrays Micros for Low-Cost Power", Computer World, Vol. IX, No. 44, October 29, 1975, p. 1.

5. L. Nisnevich, and E. Strasbourger, "Decentralized Priority Control in Data Communication", Proc. 2nd Symp. on Computer Architecture, January 1975, pp. 1-6.

6. J.R. Pierce, "Network for Block Switching of Data", Bell Sys. Tech. J., Vol. 51, No. 6, July-August 1972, pp. 1133-1145.

7. E.E. Newhall, and A.N. Venetsanopoulos, "Computer Communications - Representative Systems", Information Processing 71, North Holland, 1972, pp. 545-552.

8. C. C. Reames, and M.T. Liu, "A Loop Network for Simultaneous Transmission of Variable-Length Messages", Proc. 2nd Symp. on Computer Architecture, January 1975, pp. 7-12.

9. R.L. Sharma, et al, "C-System: Multiprocessor Network Architecture", Information Processing 74, North Holland, 1974, pp. 19-23

10. L.H. Brandenburg, et al. "On the Addressing Problem of Loop Switching", Bell Sys. Tech. J., Vol. 51, No. 7, September 1972, pp. 1445-1469.

11. L.D. Wittie, "Communications in Hierarchical Mega-Micro-Computer Networks", SUNY/ Buffalo, Computer Science Dept., Technical Report No. 102, December 1975.

12. G.H. Barnes, et al, The "Illiac IV Computer", IEEE Trans., C-17, Vol. 8, August 1968, pp. 746-757.

Acknowledgement

Preparation of this paper was supported in part by SUNY Research Foundation Grant 050-7358A.

AN INVESTIGATION OF DESCRIPTOR ORIENTED ARCHITECTURE

Terry A. Welch
Department of Electrical Engineering
The University of Texas at Austin
Austin, Texas 78712

Abstract

A computer architecture is proposed which uses a hardware implemented descriptor system to provide facilities for explicit data typing, memory relocation, and access protection at the data element level. The problem of having to fetch descriptors for every operand access is overcome by storing descriptors in small fast memories of various types. The resulting machine runs simple languages (such as FORTRAN) as fast as conventional architectures, and offers significant speed improvement for languages using complex data types (such as data management systems). The cost of the descriptor storage hardware is shown to be modest, so this architecture would be suitable for machines as small as a large minicomputer.

Introduction

A descriptor architecture is a computer organization in which data descriptions are explicitly stored and interpreted by hardware. That is, for each data field, there is a word or two of memory, called the descriptor, which defines the mode, precision, access protection, location, and structure of the data. The descriptor is read each time the data field is accessed, so the hardware has significant information with which to better carry out the operation to be performed. The use of descriptors potentially provides many important improvements in computer design in two categories: (1) decreases software costs due to better diagnostic tools and simplified programs, and (2) more efficient memory management in terms of mechanisms for memory protection and for data relocation.

Descriptors have been used in some form on several machines, most notably on Burroughs systems [3,4], have been used in many software systems, and have been investigated in the literature with regard to a number of applications. Comprehensive discussions of uses and potential uses appears in [1] and [2]. A characteristic of previous descriptor designs, however, is that the descriptor was used for only one or two of its several possible uses, and thus the overhead of storing and accessing the descriptors was not always justified by the benefits of their use. The investigation described here focuses on the possibility that a machine designed specifically to utilize descriptors can minimize their overhead memory costs through good hardware design and still retain the ability to use descriptors for enough different functions to achieve significant performance gains. A research computer is now under construction at The University of Texas at Austin for the purpose of testing those possibilities.

Descriptor Utilization

The various uses for descriptors are summarized here, to indicate the type of facilities needed.

Data Type Information

Descriptors provide explicit identification of operand data type, which permits operators to adapt to the operands supplied. Descriptors are particularly useful when working with a wide range of data types, and/or dynamic data types, especially as appear in data base management. It is desirable to have the ability to build hierarchies of descriptors to describe complex data structures. A criterion of a good descriptor system would be its ability to run type-tolerant languages such as APL in compiled rather than interpreted form.

Addressing

Memory addresses appear in the descriptors, so only descriptor identifiers appear in instructions. This serves both to reduce instruction lengths, and to put memory addresses into a compact fixed format so that hardware aid for memory allocation and relocation is more easily achieved. A good descriptor implementation should be able to support a dynamic segment relocation system with substantially less addressing overhead than occurs in a paged virtual memory system.

Protection

Descriptors can serve to restrict the type of access and the range of access for individual variables. They have many of the properties associated with "capabilities," which are receiving increasing interest for protection purposes at the operating system level[5]. Previous implementations[6,7] have used capabilities at the segment level, but this is too coarse of a resolution for many applications such as data management. An objective of a descriptor implementation is to provide efficient hardware support for a capability based protection system extended down to the level of protecting access to individual variables.

This above set of objectives, to be achieved simultaneously, requires an elaborate descriptor system which contains information describing variables at the segment (relocation) level, at the protection field level, and at the level of components in data structures. Further, every operand access must be carried out through descriptors. The implementation problem is that some variable accesses may require several descriptor accesses, giving unacceptably high execution time overhead. This paper investigates hardware solutions to this problem, utilizing small fast memories to store descriptors.

Descriptor Addressing Requirements

The objective in building a descriptor-oriented machine is to provide a mixture of hardware and microcoded firmware to manipulate complex data structures at the machine language level. This requires the facility to carry out rapidly a sequence of descriptor interpretatons for each machine language instruction. A principal source of possible inefficiency is in address computation, both for accessing data in memory and for identifying descriptors in a high-speed descriptor storage unit.

For these purposes, we presume that memory space is utilized in segments, so that a segment is defined to be a continuous memory space allocated as one unit. No restrictions are placed on segment contents, which may be mixtures of data types, code, and descriptors. Segment addressing is hoped to be of sufficient flexibility and efficiency so that a computer system could use segments as a basic memory allocation unit, or could use the segmentation system on top of a paging system.

For purposes of this discussion, assume that descriptors are segregated within segments (e.g., all descriptors for a segment appear in the first locations of the segment). Descriptors must contain at least the following set of abilities, to provide a flexible addressing system:

To access data: Segment ID, relative address displacement

To access descriptors: Segment ID, descriptor number.

Segment ID is an identifier which selects a unique segment, but whose exact coding will vary substantially from system to system. Segments addressed will typically be: (1) the segment in which the descriptor resides, (2) the segment in which an actual parameter list (from a routine which called the present routine) resides, and (3) arbitrary segments in which global variables, other procedures, and large data structures might reside.

Descriptors are viewed as occurring in two general usage categories:

(1) Directly accessed, which include descriptors for data elements which have explicit variable names (with proper data organization these can be few in number; perhaps about ten for a FORTRAN subroutine when variables of common type are listed as a vector under one descriptor).

(2) Indirectly accessed, which define the details of data structures, parameter lists, etc. (these hardly exist in FORTRAN, but might number in the hundreds for a procedure in a data-structure oriented language).

These two types are distinguished for purposes of efficiency in access.

The descriptor-based addressing system is presumed to support very efficient memory relocation capabilities. A segment must be able to be loaded in an arbitrary memory location, and perhaps be able to be moved dynamically while a program is running. This requires that absolute memory addresses be treated carefully. In particular, each segment's absolute address should appear in a small fixed number of places in the system, so that when the segment moves few addresses need to be changed.

An important restriction on the system addressing is that no program be given access to unauthorized addresses, which means to unauthorized descriptors. Thus, when switching from one program to another, care must be taken to block access to any stray descriptors remaining in the descriptor storage unit. The memory protection features potentially offered by a descriptor system can be preserved if a program can access only those descriptors specifically allocated to it.

The above paragraphs describe a basic set of properties characterizing a descriptor-oriented system. The remaining sections of this paper will attempt to demonstrate that these properties can be efficiently implemented at the machine language level.

Hardware Facilities

The strategy proposed for descriptor handling is to keep the most frequently accessed descriptors in small fast buffer memories, so that average descriptor access times can be kept small. It is conceivalbe that this descriptor memory could be organized as a single content-addressable memory (CAM), but speed and cost are improved if other accessing methods are used as well. The descriptor buffer memory would best consist of the following three component memory types.

RAM

A random access memory is used to store the directly accessed descriptors for the presently active procedure. The RAM can be addressed directly by descriptor number, since these numbers can be assigned consecutively by a compiler. The RAM could be block-loaded upon program entry, since the descriptors would be in a contiguous main memory area. Block loading would utilize the rapid block transfer capability (provided by interleaving), which is increasingly common even in small computer memories, to load the RAM much faster than if descriptors were loaded piecemeal. The RAM can be small, not more than 64 words, so it can be quite fast.

LIFO

A last-in-first-out stack is used for procedure linkage. The top stack element contains the descriptor for the actual parameter list, which in general will point to another segment. The top element is used for most parameter references, but other stack elements must be kept accessible in case a parameter is passed from procedure to procedure. Implementation of a stack is very simple with a few RAM memory chips and an up-down counter to indicate stack top. The overflow problem is best handled, in small machines, by providing an adequate stack size (say 64 locations) and issuing an error message if the stack ever fills up.

CAM

A content addressable memory is used for all other descriptors. The CAM is addressed by the two-tuple {segment number, descriptor number}, which typically might be 20 bits of address, so a RAM is not feasible. The necessary size of the CAM varies with application: FORTRAN-like languages will rarely use it; languages using extensive data structures or global variables will need large quantities. An initial experimental value of 256 locations has been selected for evaluation. That size is too big to build a real associative memory within a reasonable budget, so a RAM with hash-coded addressing is used instead. The result is an average read time of around 300 nsec., which might not be enough faster than an access to main memory to justify its costs (about $600.00 product cost) except in larger systems. The nature of the use of the CAM is such that its omission would not fatally cripple a descriptor machine (as discussed in the conclusions section).

In addition to the descriptor storage mechanism described above, a fourth memory, to store segment base addresses, is used. This would be a RAM, addressed by segment number, which contains the absolute machine address at which each active segment is stored. This approach keeps the absolute addresses out of the descriptors, thus reducing their length and simplifying relocation procedures. The number of base registers thus provided could be 64 or 256 with little difficulty. If the segment identification numbers exceed eight bits, then probably a segment name mapping system would be added as discussed below.

The important point to notice from the above discussion of hardware facilities is their moderate cost. Without the CAM, the system costs roughly 100 integrated circuits, which is equivalent in cost to about 4K of 32-bit semiconductor memory. Adding the CAM would double that. This then would be not too expensive to add to a medium-sized minicomputer (or larger) system.

Operand Addressing

The normal operation of operand address interpretation follows this sequence:

(1) When an instruction is fetched, it specifies an operand by descriptor number.

(2) The descriptor is fetched from the descriptor RAM, and it specifies segment number and displacement for the variable.

(3) The segment number is used to fetch a base address from the base register memory, which is added to the displacement to give the full memory address of the operand.

This sequence requires perhaps 300 nsec. in conventional TTL logic, which would decrease instruction execution rates by 10% to 40% in typical minicomputers designs (hopefully compensated for by requiring fewer instruction executions).

In more complex data structures, including parameter lists, a descriptor may point to another set of descriptors, not necessarily in the original segment. These secondary descriptors would be accessed from the CAM. The first access to each secondary descriptor will not find it in the CAM, so it will have to be written there after fetching it from main memory. The unsuccessful CAM access may take about two main memory cycles, but should happen only once per descriptor, unless the CAM is fairly full of active descriptors, or unless that routine is swapped out in a multi-user system.

One type of descriptor can be given special treatment. This is the indirect pointer descriptor which serves only to point to another single descriptor. It is useful for crossing segment boundaries, mapping non-contiguous descriptor sets into contiguous descriptors, and providing a data element with several access paths having different access restrictions. Whenever such a descriptor is found, its target descriptor is substitued for it in the descriptor storage RAM. That is, for a given descriptor number, there will exist different contents in the RAM than appear in main memory, but functionally the two forms are equivalent. This provides the benefit that each time that variable is accessed after its first access, a level of indirection is bypassed (and a CAM location is saved).

Of course care must be taken that this dynamic binding of variables does not produce unauthorized access paths for a program.. For example, if an indirect descriptor is pointing to a procedure's parameter, and if the parameter descriptor is substituted for the original indirect descriptor, that binding must be broken if the procedure is exited and re-entered with a new parameter; if the old substitued descriptor is still existing in descriptor storage, the procedure may pick up the wrong parameter. There are three such problems to consider here:

Relocation. If absolute memory addresses appeared in descriptors, then descriptor substitutions would have to be unbound whenever a segment was moved. The design proposed here segregates memory addresses from the descriptors at all times, so no such problem arises.

Parameters. As shown in the example above, care must be taken when exiting procedures to unbind all parameter linkages. This is done by erasing all descriptor RAM contents upon exit. It is not efficient to erase the CAM each time a procedure is exited, so descriptor substitution is not permitted in the CAM.

Protection. When a sequence of descriptors is traversed to arrive at a variable, the resulting type of access permitted may not be less restrictive than that specified by any descriptor on the path. Therefore, when a target descriptor is substituted in RAM for an indirect descriptor, the resultant access restriction must be as restrictive as the more restrictive of the original two descriptors.

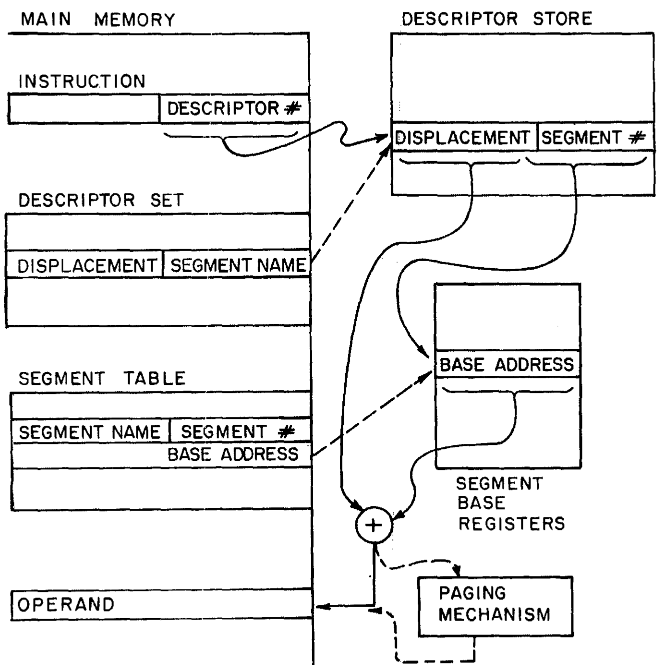
It appears feasible to permit modification by substitution of descriptors in the dynamic stored form, provided the above restrictions are followed.

Segment Addressing

The base régister storage system has the objective of providing low cost dynamic memory relocation capabilities, while removing some troublesome addressing problems from the descriptor storage process. This is a segment address mapping system intended for systems where memory is allocated by segments. It would also be useful in paged systems, to translate segment names into a more convenient address range so that virtual addresses within the computer are kept at a manageable length. The following discussion described two systems, a full virtual memory system and a very simple mapping system, which both use the same hardware facilities.

The virtual memory system presumes long segment names, and a software controlled segment table to map these names into memory addresses. This table is very slow in access time, so hardware is provided to minimize the number of accesses. Specifically, the base register memory is used to hold addresses of all active segments, using the following strategies (summarized in Fig. 1):

Register Assignment. Base registers are assigned sequentially, using a hardware counter, whenever a new segment is activated. The assigned base register holds the segment's memory address or an indication that the segment is not available in memory.



OPERAND ADDRESSING

Figure 1

Segment Table. This table in main memory must contain entires for all segments, indicating file or memory location. When a segment is moved into main memory, a base reigster is assigned, and its number is stored in the segment table. If the segment is later relocated or swapped out, the base register is modified accordingly.

Descriptor Contents. Descriptors in main memory will contain segment names as part of their address information. When a descriptor is first accessed, the segment table will be accessed to find the assigned base register number for the segment referenced. This number becomes part of the descriptor in the descriptor storage mechanisms (RAM, CAM, and LIFO), so that each subsequent use of the variable requires only direct access to its base register. Thus, the segment table is accessed only once for each first use of a descriptor.

Register Reassignment. Since the register set is fixed and smaller than the number of possible segments, eventually the end of the set is reached and previously assigned registers must be reassigned. Care must be taken that entries in the segment table and descriptor store do not still reference previous assignments of reassigned registers. The simplest algorithm is, upon reaching the end of the base registers, just to empty all base registers and erase all assigned register numbers from the segment table and descriptor store. This requires that descriptors be stored with segment names intact, so that they can be relinked at any time.

Thus, there exists a mechanism whereby a relatively short base register number is substituted for each active segment name. This permits use of a random access memory to store base registers (in place of the associative memory used in most virtual systems), which can therefore be made large enough in size to efficiently handle all active segments and yet be moderate in cost. Such a system is possible principally because descriptors are stored in volatile form in a separate storage mechanism, so operand segment ID's can be modified dynamically as segment base register numbers change.

An alternate usage of the same hardware apparatus produces a poor man's segmentation system of reasonable simplicity. If segment names are restricted in range to the size of the base register set, then the base registers can be set up directly by the system loader, and no run time access to the segment table is needed. That is, the loader must worry about assigning and reassigning base registers to segments, but if the number of base registers is adequate this is not a complex job. The loader must identify segments and modify all inter-segment descriptors (except those for parameters) when a routine is loaded. Thereafter, the segment can be reallocated or removed with the only modification being to base registers.

When this approach is compared to the base register system of conventional machines, two distinct differences are noted:

(1) The number of base registers must be restricted in conventional machines because of the number of address bits which would be required in each instruction. By moving the base register designation into the descriptor, it may be given more bits since it is not retrieved from main memory so frequently. Consequently, the number of base registers may be comfortably large in a descriptor machine.

(2) The programmer need not assign base registers since the loader can perform this action. This is made easy by the large number of registers available, and because segment names appear only in a relatively few places in the code due to the descriptor structure.

This level of relocation system appears quite feasible for small time-sharing systems.

Procedure Entry

Procedure entry and exit is frequently a time-expensive operation, so this process is worth special attention. In particular, any system of hardware support for dynamic address binding must be shown to be both efficient and accurate in handling parameters.

State Switching

When a new procedure is entered, the descriptor storage RAM must be loaded with the new descriptor set. To gain efficiency in state restoring, this RAM is duplicated so that the descriptor set of the calling routine is not lost. This permits rapid returns from the last issued procedure call, but other returns will require the RAM to be reloaded again. Available statistics[9] indicate that 50% to 80% of procedure exits will not require RAM reloading. The other descriptor storage facilities pose less trouble. The LIFO stack automatically stores its state information so no special action is required. The descriptor CAM and base register set have contents which may be used by any routine which can address them, so they do not constitute part of a procedure's state.

Parameters

Actual parameters are organized as a contiguous list of descriptors in the calling routine. Since normally the correct set of data descriptors will not be contiguous, the parameter list descriptors will be indirect pointers to the true descriptors. The parameter list is pointed to by a parameter descriptor held in the LIFO descriptor memory. Thus, a parameter reference will require three descriptor fetches: one from the LIFO store to find the parameter list, a second (from the CAM on later accesses) to find the location of the actual descriptor, and a third (usually from the alternate RAM) which actually describes the data. While this is a long sequence, it will normally be fast. Note that this mechanism has no difficulties in the case where an actual parameter is itself a formal parameter. The first such reference is slow, but on subsequent references the substitution for indirect descriptors permits many descriptor accesses to be bypassed.

Protection

The descriptors in the parameter list can contain access restrictions, to distinguish call-by-name from call-by-value parameters. This eliminates copying of parameters in all cases, except called-by-value parameters in which might be modified by side effects of later procedure calls.

LIFO Store

The stack will contain the conventional linkage information to the calling routine, namely the return address and pointer (descriptor) to the parameter list. This information does not get stored in main memory, so the stack must be protected against erasure. Interior elements in the stack can be accessed, so the passage of parameters from one routine to another can be handled easily. The choice of a stack mechanism, rather than storing linkage information in main memory, is based on cost. It reduces control costs, and is itself not particularly expensive using existing memory circuits.

In summary, then, the procedure mechanism is this:

(1) Upon entry, the parameter descriptor and return address from the call are pushed onto the stack; the descriptors for the new procedure are loaded into RAM storage.

(2) Parameter references are indirect through the top stack element, and thence indirect through a parameter list descriptor to the data descriptor; the last descriptor may itself be a formal parameter, which lengthens the initial access time but causes no control problems.

(3) Procedure exit consists of popping off the top stack element, but no special actions are required on other storage elements.

This procedure mechanism appears to be sufficient to eliminate run-time software except in special instances. The apparatus established to reduce the overhead of descriptors access serves also to reduce the overhead of indirect parameter access, so parameters need not be copied into the called routine. In addition, dynamic relocation is not obstructed, so that the calling routine can be relocated during procedure execution without affecting parameter access.

Conclusions

The above described system is presented as a feasibility demonstration that the use of dedicated small fast local memories can reduce the overhead of descriptor processing sufficiently to make it as fast as more conventional instruction processing. Instruction execution times are increased perhaps 25% due to longer operand interpretation processing. Procedure entry requires a block transfer of descriptors, which takes perhaps the equivalent time of two to four instruction executions, in a interleaved memory system. Each parameter accessed for the first time requires two extra memory cycles. These time costs are offset by the saving in run-time procedure software execution, by reduced operating system code for relocation, and by reduced instruction size. The last effect is significant since typically eight bits can be eliminated from 32-bit memory reference instructions, and this represents a 25% reduction in memory bandwidth requirements (e.g., the 16-bit base plus displacement of a IBM 370 instruction could be replaced by a 8-bit descriptor number).

It appears that FORTRAN level languages will break even or run slightly faster on the architecture proposed, but extensive testing will be required to verify the exact savings. This would indicate that the addressing flexibility inherent in a descriptor system does not need to penalize execution of simple languages.

On the other hand, operand fetches would be slower for type-tolerant languages such as APL and data management systems. Systems with richer type structures and with dynamic variable sizes will use more descriptors, most of which are used infrequently (e.g., a descriptor giving the length of a field in a particular file record). These secondary descriptors will cause frequent fetches from the CAM or main memory, and thus will cause longer instruction interpretation times. It should be observed, however, that these secondary descriptors perform functions which are not directly available in conventional machines, so accessing and interpreting a secondary descriptor replaces a software action in present language implementations. Thus, even if no CAM is provided and every secondary descriptor reference requires a main memory access, the proposed architecture will run these high-level languages significantly faster than a conventional machine of equivalent technology.

The implementation proposed here is an example of a necessary trend in computer architecture today, namely extension of the memory hierarchy downward through use of small fast memories. This system uses dedicated functional memories rather than or in addition to general usage cache-type buffers. Use of a dedicated memory is made more valuable by the descriptor format, because it segregates frequently-used address information into a compact fixed-format area (the descriptor set). This strategy of using descriptors to isolate critical machine language data appears to be a promising way to achieve increased use of hardware to replace low-level software.

Acknowledgement

This research was support under National Science Foundation Grant GJ-42514 and Joint Services Electronics Program Contract F 44620-71-C-0091.

Bibliography

[1] Feustal, E. A., "On the Advantages of Tagged Architecture," IEEE Trans. on Computers, C-22 (July 1973), pp. 644-656.

[2] Iliffe, J. K., Basic Machine Principles, 2nd ed., Macdonald/American Elsevier, 1972.

[3] Burroughs B6500 Information Processing Systems Reference Manual, Burroughs Corp., Detroit, Mich., 1969.

[4] Wilner, W. T., "Design of the Burrough B1700" AFIPS Conference Proceedings, Vol. 41, FJCC 1972, p. 489.

[5] Proceedings of the International Workshop on Protection in Operating Systems, IRIA, Paris, France, August 1974.

[6] Shepherd, J., "Principal Design Features of the Multi-Computer (The Chicago Magic Number Computer)," ICR Quarterly Report, No. 19, Institute for Computer Research, University of Chicago, November 1968, sec. 1B.

[7] England, D. M., "Architectural Features of System 250," Plessey Telecommunications Research, Ltd., Taplow Court, Taplow, Maidenhead, Berkshire, England, 1972.

[8] Linden, T. A., "Capability-Based Addressing to Support Software Engineering and System Security," Proc. Third Texas Conference on Computing Systems, November 1974, p. 8/5.

[9] Batson, A. P., Brundage, R. E., and Kearns, J. P., "Design Data for Algol-60 Machines," Third Symposium on Computer Architecture, Clearwater, Florida, January 1976.

TAGGED ARCHITECTURE AND THE SEMANTICS OF PROGRAMMING LANGUAGES: EXTENSIBLE TYPES

by E.A. Feustel†
Rice University
Houston,Tx.

Keywords: Programming languages, computer architecture, step-wise refinement

Abstract

This research note suggests that before we design hardware or software for the task of problem solving, we re-evaluate the task of problem solving in terms of the linguistic constituents which will be required and the manner in which these linguistic constituents will be combined. Utilizing the principles of composition and abstraction-to-specifics, we conclude that all data and programs might be realized in a structured format called messages. We conclude with a few preliminary thoughts and questions as to how an architecture designed for such structured operands and operators might be designed.

Introduction

A premise of these preliminary thoughts is that there are many problems which we want to solve by computer whose solutions are currently beyond the state of the art. The principal reason for this situation appears to be that the complexity associated with preparing a computer solution combined with the complexity of the problem makes the complexity of the task beyond our ability to solve the problem [1].

Numerous techniques have been advocated for reducing the complexity of the problem of producing a computer solution. Software techniques include debugging [2,3,4], structured programming, step-wise refinement [5,6], proving programs, data abstraction [7], and very high level languages [8,9]. Hardware techniques include high level language machines [10] of which SYMBOL [11] is best known, and tagged architecture [12,13].

We will pursue the assumption that we should re-evaluate the entire hardware-software interface as seen by the problem solver to establish a gestalt of problem solution by computer [14]. It would seem appropriate to attempt to retain as many features of a natural language solution of the original problem as is possible rather than to limit the solver in an artificial manner as is done by so many hardware-software-interfaces to solvers in today's computational environments. We thus pursue the solver's artificial language and the eventual language of the machine from the linguistic viewpoint, attempting to find a representation for his problem which will be closely matching the problem in structure and language. Only after we have done this will we evolve a language for programming and a mechanism for the evaluation of programs.

The Requirement for an Abstraction-to-Specifics Methodology

The principal requirement for a programming tool for the solution of complex problems is an abstraction-to-specifics methodology, a step-wise refinement method. This methodology requires that each succeeding refinement of an abstraction (lower level) be functionally isolated from higher levels and dependent only on constituents selected from lower levels of abstraction or from a level parallel to itself.

Although it might appear that at any level we should only be able to refer directly to the next level without recourse to constituents at even lower levels (as found in the T.H.E. operating system [15]), if we are to reflect the linguistic processes in which we analyze the problem in a direct manner, we must permit a downward directed graph model of abstraction of connected constituents instead of an onion model of layered connections.

The methodology which we wish to select is to give us the necessary freedom to deal with the real issues of solving the complex probelm. It should allow us to reflect the solution to this problem in a transparent manner--that is, the methodology used to solve the problem should not add unduly to the complexity of the total solution. The use of the abstraction-to-specifics methodology permits us to ignore detail (temporarily) while we pursue the space of possible solutions utilizing higher level abstractions.

After we have found abstractions representative of the class of solutions which we desire, the methodology should provide a structure wherein the detailed solution may be generated in successive specification. It is desirable that the structure bear the weight of detail at each level, permitting us freedom in dealing with the implementation of each first level abstraction in a group of second level abstractions, etc.

One methodology of great utility is the treatment of all computational entities as messages. Messages may represent names of objects, functions on objects, procedures to be carried out, the objects themselves, and homogeneous and non-homogeneous groupings of objects. In the sequel we will make some preliminary comments on their use, flexibility, and implementation.

The Constituents of Computation

A most important prerequisite to our use of an abstraction is the ability to name it and to refer to it by the name or names chosen for its use. The notion of name permits us to refer to an object such as an operator, an environment, a function, and a datum in an abstract way without being concerned with the representation or the details of its implementation or access. We will next discuss several desired features of names.

At the minimum we must be able to directly associate a name with the object to which it is permenently bound. In addition however we wish to be able to reference an object by naming the name of an object (indirect reference) or alternatively an alias for a direct reference or an indirect reference. Although a permanent binding is sufficient, we would prefer to permit a dynamic binding so that we might manipulate

† This work was supported in part by National Science Foundation Grants: GJ3647 and DCR72-03609A01.

the structure of our universe of discourse and ease the task of the composer of each abstraction. Finally we wish to be able to specify attributes and properties associated with each name in a manner similar to the property or attribute lists in LISP. Such properties might include type information, copy rights, access, sharing, use, and environmental information.

Names refer to objects of different types which may be manipulated in manners consistent with their property lists. We prefer the notion that data may be obtained functionally [16]. That is, there should be no distinction between information which is retrieved as the result of a function call or from accessing the object associated with a name. Functions on the other hand may be manipulated, created, destroyed, and acessed as data--although not in the same manner as done on the von Neumann machine. Further we feel that data should derive its properties from the chain of attributes taken from the chain of names which access it and not from the representation of the data itself. Thus atomic data and functions are but packets of information to be interpreted by the viewpoint provided by the accessing symbol. Names may also be used to refer to packets of information clustered together, collectively, individually, or by some subset property. We observe that the notions which we have outlined above are little different from the way in which language permits the use of abstraction.

As in conventional mathematical notation, we would choose to have polymorphic operators and functions whose meaning is determined by the attributes (inherited or direct) of the names with which they are associatedvariadically in prefix, suffix, or infix notation. Thus the symbol for an operator is merely a name for an abstract procedure whose function is semantically determined by the tupe of its arguments and which may have a dynamic meaning dependent on the state of computation. It must be possible to specify the binding of operators to their parameters, formal and actual, and to create closures representing Curied functions. We assume that any operator may be recursive, may create new operators and environments and may be self destructive.

Extended classes of data and operators may be composed by encapsulation [18] through indirect or direct reference on previously defined objects and operators. Since data and operators are treated identically, no distinction is to be made between abstractions containing procedures, data, or mixtures of both. Such compositions will normally be copies of original constituents unless sharing is specified as an option [19]. Since names are compound objects themselves, we choose to eliminate the distinction between name and object and represent the closed metalanguage thus defined as typed packets which we will call messages.

The mechanism which we have chosen for the representation of problem solutions appears to be quite general. Yet it so resembles the necessities of the language which we will use to solve the complex problems, that the difficulties of specifying a solution may be minimized. Thus we are led to a consideration of how such a message computer might be organized.

The Message

One informal view of a message is that of a memorandum as utilized in an office environment. The message contains information about who is to receive it, who sent it, and the date that it was sent, the topic to be discussed, and the body of the message. It may contain other information including a routing list, access information, and a memorandum reference number. The latter identifies it in case of the need for a reply or comment.

In the same manner, a message in a computational situation consists of separate parts, each of which may be messages. A header is followed by a sequence of messages which is followed by a trailer. The header may indicate the process which is to receive the message as well as indicating the number of messages in the sequence and their relative position within the body of the message or it may indicate the purpose for the inclusion of some or each of the messages in the sequence.

Several points are worth noting about the definition of messages. First a message is defined recursively in that a message may contain a sequence of messages. This sequence must be finite and may be empty. The latter sequence indicates a null object of a certain type. Second, the definition of a message implies the possibility of a nested set of potentially nested messages. This nesting property corresponds naturally to the conventional notation for block structured languages, to the structuring methods in most languages for data, and to the structure of most file systems. Third because of the fact that indirect references may be imbedded within a message objects may share each others components or refer recursively to themselves. In conventional programming languages this notion corresponds to procedure calls in the procedural part of the language, to pointers in the data structure part of the language, or to cross-references within a dictionary system.

The use of the message as defined above may be shown in an informal manner by comparison to a memorandum and its effect on the employee receiving it. The memorandum may contain orders for the immediate execution of a certain procedure at an assigned priority. It may indicate that a procedure parameterized by one or more other memoranda are to be executed. Or it may be a procedure whose execution is conditioned on some other external event (a deferred procedure). Alternatively it may be regarded as a conveyance of data from the sender to the receiver in which case it may be put in a file of messages under its reference heading for further use.

Let us assume that the actor process [20] receives a message. After examining the header, it may execute the process described in the message immediately. It may evaluate parameters of the message from one or more separate environments and then execute the procedure. It may file the message for future reference and/or execution under a name it generates or one it obtains from the message itself in an environment which is specified internally or externally.

The Implementation

In this research note, we do not propose a concrete realization of the architecture which implements the message concept. (A design is in progress.) We do speculate on some of its characteristics and invite discussion from the research community.

It seems likely that such a system architecture could consist of several specialized computers in the same manner as SYMBOL does. One computer might manage the space of names for a given environment and would be responsible for the process which map a name to an object and which determine the inherited attributes from the name chain. A second computer might manage the mapping of the physical representation of the object into the virtual message space and would be responsiblefor dictionary maintenance. A third computer might be responsible for marking messages to be deleted and doing garbage collection in virtual message space. A fourth computer might be responsible for message transport to external devices and a fifth for internal transport and pipeline management. A sixth computer might perform type checking on arguments of procedures. A seventh might be involved in the evaluation of function, perhaps using several functional units in a dataflow scheme. An eighth might be used for copying messages. A ninth might be used for I/O and dynamic changes in representation from internal form to external form. And so on.

One question which arises is what method should be used to store message information. Our preference would be for a bit serial store organization because of the implicit assumption that messages may be of variable length. BORAM, magnetic bubble memory, or charge coupled device memory might be used, organized in a parallel manner so that different messages might be accessed simultaneously. After selection of numeric components such as numbers, vectors, and arrays from a larger message, the resulting representations would probably be stored in a local operand cache, organized in word format for use by parallel arithmetic units.

A major advantage of the associative mapping scheme for names to objects is that the names now convey type information and representation information which facilitate easy manipulation of and scheduling for the flow of operand streams. Names and their associated structural information and procedure references may be held in a smaller, faster storage unit. Name lookup may be overlapped with computation.

It seems quite likely that a message machine will consist of many specialized components designed to optimize handling of specific types of messages such as procedures, queues, arrays, etc. and that the number of such components would increase for high performance machines and be smaller on intermediate performance machines. The design of the components, their interconnection, and their interaction promises a new era in development of computing machinery.

References

1. Dijkstra, E.W.,"The Humble Programmer", CACM 15, No. 10, Oct. 1972, pp 859-866.
2. Gaines, R.S., The Debugging of Computer Programs, Ph.D. Thesis, Princeton University, Princeton, N.J. August 1969.
3. Grishman, R.,AIDS: All-Purpose Interactive Debugging System User's Manual, Courant Institute of Mathematical Science, New York,N.Y.,1968.
4. Kulsrud,H.E., HELPER: An Interactive Extensible Debugging System, Working Paper No. 258, Institute for Defense Analyses, Princeton N.J., May 1969.
5. Dijkstra, E.W., "Notes on Structured Programming", Structured Programming, Academic Press, New York, N.Y., 1972, pp. 1-82.
6. Wirth, N.,Systematic Programming; An Introduction, Prentice Hall, Englewood Cliffs, N.J., 1973.
7. Liskov,B.H., and Zilles, S.N.,"Programming with Abstract Data Types", SIGPLAN Notices 9, 4, April 1974, pp. 50-59.
8. Schwartz, J.T., On Programming -- An Interim Report on the SETL Project, Instalment 1: Generalities, Computer Science Department, Courant Institute of Mathematical Sciences, NYU, New York, N.Y. February 1973.
9. -----,"Proceedings of a Symposium on Very High Level Languages", SIGPLAN Notices 9,4, ACM, New York, N.Y. April 1974.
10. -----,"Proceedings of a Symposium on High-Level-Language Computer Architecture",SIGPLAN Notices 8, 11,ACM, New York, N.Y.,Nov. 1973.
11. Rice,R.,Smith, W.R., "SYMBOL: A Major Departure from Classic Software-Dominated von Neumann Computing Systems", AFIPS Conf Proc 38, pp.575-587.
12. Iliffe, J.K., Basic Machine Principles, 2nd Ed., American Elsevier, New York,N.Y., 1972.
13. Feustel, E.A.,"On the Advantages of Tagged Architecture", I.E.E.E. Transactions on Computers, C-22,7,July 1973, pp. 646-656.
14. McMahan, L.N., Language Directed Computer Architecture, Ph. D. Thesis, Rice University, May 1975.
15. Mutschler, E.O.,III, An Integrated Model for Computational Processes, Ph.D. Thesis, Rice University, May 1973.
16. Dijkstra, E.W., "The Structure of the T.H.E. Multiprogramming System," CACM 11, 1968, pp. 341-346.
17. Reynolds, J.C., "Gedanken - A Typless Language Based on the Principle of Completeness and the Reference Concept", CACM 13, 5, May 1970, pp 308-319.
18. Redell, D. D., Naming and Protection in Extendible Operating Systems, Technical Report 140, Project MAC, MIT, Cambridge Mass., Nov. 1974.
19. Wozencraft, J.M., Evans,A.Jr., Notes on Programming Linguistics, Department of Electrical Engineering, MIT, Cambridge, Mass., Feb. 1971.
20. Hewitt, C., "Planner," Project MAC Progress Report XI, MIT, Cambridge, Mass., July 1973 - July 1974, pp. 221-282.

DESIGN DATA FOR ALGOL-60 MACHINES[†]

by

A. P. Batson, R. E. Brundage,[**] and J. P. Kearns
Department of Applied Mathematics and Computer Science
University of Virginia, Charlottesville, Virginia 22901

Key Words: program behavior, machine design, high-level language machines, Algol-60, virtual memory, processor utilization

CR Categories: 6.22, 4.22

Abstract:

The performance of a high-level language machine will depend upon the characteristics of its workload. Equally, the design for such a machine should be guided by some understanding of the work which it will be called upon to perform. We present here some behavioral properties of a large Algol-60 program in terms of the requests they represent for the various processing resources of an Algol-60 machine. The data provide insight into the behavior of such machines, particularly with respect to dynamic memory requirements and the procedure activation rates associated with direct Algol-60 execution. The data can thus be of value to designers of high-level language machines in view of their implications for the performance of such systems.

1. Introduction

The syntactic rules of a high-level language together with their semantic interpretations serve as functional specifications for a high-level language machine. That is, these rules prescribe the ultimate results which must be achieved as a result of the interpretation of source-language programs by the machine. However, these rules do not provide guidance to the machine designer as to how to best implement such a language interpreter. Many linguistic constructs are used only rarely, whereas others may be frequently encountered and would merit extra attention in the design process so as to assure acceptable performance levels for the complete system. Knuth's[1] studies of a large sample of Fortran programs, for example, shows that certain statement types are much more common than others. Moreover, the 'static' characteristics of the syntactic rules cannot provide a designer with the information he needs to visualize the time-dependent behavior of a machine as it is exercised by actual source language programs. For example - if the majority of procedure calls in practice involve only a limited amount of computation before procedure exit, then it would be advantageous to provide a fast implementation of the procedure call and exit mechanism, whereas if the opposite characteristic were typical of real programs (i.e., procedures had long lifetimes), then the details of the design of the calling mechanism would have only a limited effect upon system performance. Another example can be found in the memory requirements of programs, such as the depth of the pushdown stack required for a stack-based Algol-60 machine. The syntax of Algol-60 gives no clue towards a maximum stack depth to build into the machine, save infinity of course! Since any sensible and practical design must satisfy the maxim "don't make all the users pay for what only 1% of them need", then clearly the designer of a high-level language machine should start with some idea of the characteristics of the workload which is to be interpreted by his design.

The workload for a high-level language machine is, of course, source-language programs and their data. The literature contains little experimental information on the characteristics of symbolic high-level language programs. As mentioned earlier, Knuth[1] collected information on the prevalence of certain statement types in Fortran programs, and also a limited amount of data on some dynamic properties of a few programs. Two of us[2,3] have described in detail the dynamic behavioral properties, at the source-language level, of a sample of Algol-60 programs. These programs were small-to-medium in size, with fairly short execution times, and were selected from a collection of production jobs for scientific/engineering applications. In contrast, we present here the results of a study of a rather large Algol-60 program. In fact, the Algol program whose execution characteristics we describe here is the Algol-60 compiler for the Burroughs B5500, with a fairly large Algol program as its input data.

2. Conceptual Resource Model

The results we shall present are discussed in terms of a simple resource model of the Algol-60 machine. We identify four separate and distinct resources - an arithmetic/logical processor, a string processor (i.e., a processor which performs character manipulation functions), a virtual memory processor, and an input/output processor. Any executing Algol-60 program can be represented as a sequence of requests (which we term an execution trace) for the services of the various processors. This execution trace is the input to the abstract representation of the Algol-60 machine, shown in Figure 1. A detailed description of the syntatic and semantic characteristics of the execution trace can be found in Brundage's thesis[4]. The only new addition to the model described in our earlier work[2,3] is the addition of the string processor, which has been included to facilitate description of the significant amount of string manipulation involved in our current example, when the program executing on the Algol machine is a language translator. The operation of the abstract virtual machine in Figure 1 can be visualized as follows: The execution trace, representing the stream of resource demands made during process execution, is input to the interpreter, which directs the resource requests to the appropriate processor.

We next specify the measurement units for the resource model. We have chosen to define time in units of work performed by the computational and string processors. In the model, requests for

[†]This research was supported by NSF Grant GJ 1005

[**]Now at the Department of Mathematics, Florida State University.

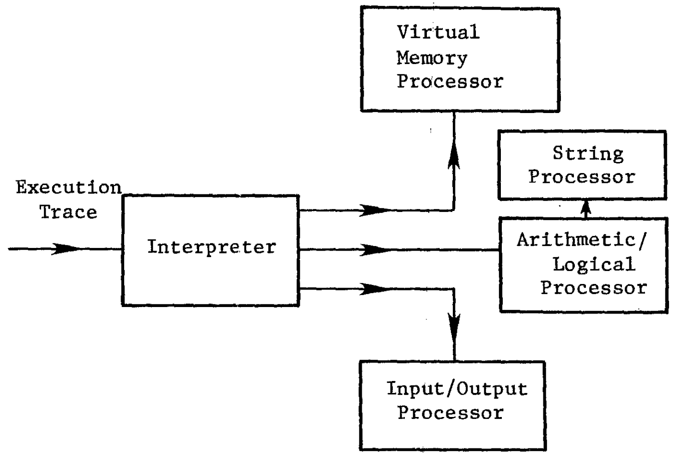


Figure 1 Algol Process Machine

service by the input/output processor or the virtual memory processor in the execution trace do not "consume" time but they do signify that the arithmetic/ logical processor (or string processor) has been halted because of a request for service by a different processor. Thus the process is blocked from using the arithmetic/logical processor until this request has been satisfied. Memory requests are specified in "words", which correspond in general to the cells of Johnston's contour model[5].

3. Measurement Technique

The hardware and software of a Burroughs B5500 were modified to permit the acquisition of a magnetic tape of execution events with each event precisely time-stamped. A software-controlled hardware counter was constructed and the 1 MHz clock pulses of the B5500 processor were input to this counter, which could be started, stopped, read, and reset under program control. The B5500 Algol compiler and the operating system were modified such that each event of interest during execution of an Algol program caused a time-stamped event record to be written on magnetic tape. This trace tape was used, in conjunction with an inverse symbol table, to generate symbolic trace tapes which were processed to obtain the results presented here. More complete details of the technique for data collection and reduction are described elsewhere[2,4]. All times given in the results are for the B5500 equivalent of our computational processors and do not include time normally spent on that real processor for virtual memory allocation or for I/O processing. These activities were deleted during processing of the raw trace data to permit presentation of the results in terms of the abstract machine of Figure 1.

4. Results

The B5500 Algol compiler, the source language program executing on the Algol machine in this experiment, is a large and complex program. It contains around 12,000 lines of Algol code, and has over 250 Algol blocks. When this program was executing in the experiment described here, there were 37,261 block/procedure entries, compared with 28,911 of these in the 34-program sample described earlier[2,3] by two of the authors. In addition, there were 22,875 distinct activations of the string processor. This effect was not included in the earlier results since the programs in that sample performed little character manipulation.

The results presented earlier for the 34-program sample consisted of 23 distinct distributions, and it is clearly impossible to present all of these here. Rather, we will direct our attention to those execution characteristics which are most evidently germane to the designer of an Algol-60 machine, and the characteristic statistics for the distributions of these values are presented in Table 1.

Two important machine resources are processors and memories. We first discuss some of the data on the memory demands made on the Algol machine. The first line of Table 1 shows the statistics for the distribution of sizes of program segments called during execution. The most interesting characteristic of these code segment size distributions is the fairly small size of most code segments. The experimental results for the sample of user programs[2] are not markedly different. The user sample had mean and median of 38.7 and 23 words, respectively, as compared to 90.8 and 14 words found in this experiment. Since a program segment must be brought into executable memory before it can become the site of execution, the relatively small sizes of the segments have definite implications for the design of the virtual memory subsystem for an Algol-60 machine. For a page-based design, for example, it would seem prudent to keep page sizes down to something rather smaller than the current fashion of 1K words to avoid excessive internal fragmentation of executable memory.

Another consequence of a procedure call is that space must be allocated for its contour data segment. Most Algol-60 implementations store this in a push-down stack. Line two of Table 1 shows the statistics for contour data segment sizes for the 134 distinct blocks/procedures activated during this experiment. The distribution has 37,261 events (more than the 33,184 program segment activations because of recursive procedure calls, which generate a new contour data segment for each recursion), and the mean and median are 6.96 and 2.5 words respectively. These figures indicate the amount of stack space associated with procedure calls, and do not include storage for the elements of arrays declared locally to the block. Recursive procedure calls accounted for 11% of the sample.

Once a procedure has been entered it has a certain active lifetime before it is finally exited, and line three of Table 1 shows the statistics of the lifetime distribution for all contour data segments. The mean of around 8 ms., and median of 0.75 ms. are those measured on our B5500 1 MHz equivalent of the arithmetic and string processors of Figure 1, and should be modified appropriately for a processor with a different execution speed. These block lifetimes are very short, and in fact 50% of the lifetimes of all procedure activations are less than 0.35×10^{-4} of the total processor time of the experiment. Block lifetimes are not in general equal to the processor time consumed while control resides in that block (the outer block, for example, has a lifetime equal to the total processor time) and we shall later provide much stronger evidence of the preponderance of short-lived, short execution-time procedures in the workload of the Algol machine.

The dynamic distribution for the block execution times - i.e. the processor time consumed while control resides in each activated block, is described by the statistics of line four of Table 1. The mean of this distribution at 0.57 ms. is considerably smaller than that of the block lifetime distribution, as would be expected, but the median of 0.34 ms. is

Variable	Sample Size	Median	Mean	Std. dev.
Program Segment Size (words)	33,184	14	90.8	185
Contour Data Segment Size (words)	37,261	2.5	6.96	9.75
Contour Lifetime (milliseconds)	37,261	0.75	7.94	250
Block Execution Time (microseconds)	37,261	344	566	1322
Contour Transition Interval (microseconds)	74,521	140	282	509
Block Execution Time/Block Lifetime	37,261	1.00	0.64	0.33
String Processor Burst Time (Microseconds)	22,875	235	330	246

Table 1 Summary Statistics for Measured Variables

not greatly different from the median of the lifetime distribution. This suggests that a large number of the procedures activated are simple, in the sense that they make no calls on other procedures, and we explore this hypothesis explicitly below. The most outstanding feature of the block execution times is their short duration. The mean of 566 μs represents the execution of only around 50-100 B5500 instructions per procedure, and thus it is clear that for this particular workload the Algol machine needs an efficient implementation for the procedure activation mechanism. The results from the 34-program sample reported earlier[3] are not in conflict with this observation - although the mean found there was 17.7 ms. with a standard deviation of 1260 (the distribution was highly skewed), the median was found to be 0.63 ms. One is tempted to ascribe the smaller mean for the Algol compiler to the fact that it was written by very expert programmers who used structured programming techniques (in 1966 or so) long before that term arrived in the world of computing. Equally, we point out that extensive modularization of programs into small procedures can impose severe execution speed penalties if the procedure-calling mechanism is inefficiently implemented. Taking the present results for the Algol compiler we see that a procedure activation time of 50 μs represents 10% of the total execution time. The median procedure execution time was 0.16×10^{-4} of the total processor time for the sample.

Another indication of the importance of the procedure entry and exit mechanism is seen in the distribution of contour-transition intervals[3,5]. A contour-crossing event is either the entry to or exit from a procedure or block. Each such event, in terms of the abstract machine of Figure 1, involves a call on the virtual memory processor for allocation or deallocation of a contour data segment. In more practical, current systems it corresponds to a call on the procedure entry or exit mechanism, since there are only a very few non-procedure blocks in the sample. The mean of around 280 μs, and median of 140 μs, (line five of Table 1) illustrate even more forcefully the important effect on performance of the design of this feature of a high-level language machine.

The values obtained when each block execution time is divided by its lifetime will be unity only for "simple" procedures, i.e. those which make no calls on other procedures. We see in line six, Table 1 that around 50% of all procedure activations were of that type. The standard "intrinsic" procedures of the Algol compiler (such as ABS, SQRT, etc.) were treated as if they were in-line code and calls on them are not included as procedure activation events. The relatively large number of such simple procedures indicates that some performance improvement might be obtained by designing a special simple mechanism for their implementation. The corresponding figure for the sample of 34 user programs reported earlier was that 80% of all procedure calls were of this simple type.

In the final line of Table 1 we present the statistics for the durations of bursts of activity on the string processor. Our execution traces were such that the string processor is always called from the arithmetic/logical processor, and the string processor times are included in the block execution times described above. The 37,261 block activations contained 22,875 calls on the string processor for string manipulation activities (corresponding to entries to "character mode" on the Burroughs B5500 processor), and these string manipulation bursts had a mean duration of 330 μs, with a median of 235 μs. The large number of calls for such services, and the fact that they account for over a third of all the processing time, illustrates the utility of sophisticated string manipulation facilities in language translation.

5. Concluding Remarks

The results presented here can serve as useful information to the designer of an Algol machine, whether this be in the form of a complete hardware system or through the more conventional software-hardware combination in use today. The general characteristics of the data are not in conflict with the results presented earlier[2,3] for a sample of 34 user-written production programs, though there are some interesting differences which we shall comment upon later. Perhaps the most outstanding result, for the designer of a high-level language machine, is the crucial importance of the procedure entry and exit mechanism. It seems clear that procedure activation occurs at such frequent intervals that close attention should be given to the design of this feature of an Algol machine. Moreover, the frequency of activation of "simple" procedures is so high that it may well be worthwhile to provide a special mechanism for such procedure calls. In the user program sample, where 80% of all procedure activations were of this type, there were in addition an unknown large number of calls on standard procedures such as SIN, SQRT, etc. which would considerably increase this effect. It may be inexpedient to implement all procedure calls with a generalized Algol-60 facility.

The block execution times for the Algol compiler execution presented here have a distribution which is much less skewed than that found for the sample of user programs. While this is partially due, no doubt, to the fact that only one program is involved here (though written by several programmers), we feel that programming style is a major contributing factor. The Algol compiler is a well-modularized, sophisticated program written by very experienced programmers, and one is tempted to call it "structured". It is, therefore, probably no accident that the rate of procedure entry and exit events is significantly higher for this program than was found for the sample of user programs. If programmers of the future are going to be using structured programming techniques, at least in the sense of writing highly modularized programs, then our results indicate that designers of new systems must give especially careful attention to the implementation of procedure entry, parameter transfer, and procedure exit mechanisms to ensure high performance.

6. References

1. Knuth, D. E., "An Empirical Study of FORTRAN Programs" Software Practice and Experience, 1 (1971) pp. 105-133.

2. Batson, A. P. and R. E. Brundage, "Measurements of the Virtual Memory Demands of Algol-60 Programs" (Extended Abstract) Proc. Second Annual ACM-SIGMETRICS Symposium, Montreal, 1974, pp. 121-126.

3. Brundage, R. E. and A. P. Batson, "Computational Processor Demands of Algol-60 Programs" Proc. 5th ACM-SIGOPS Symposium on Operating Systems Principles, Austin, Texas, 1975.

4. Brundage, R. E., Ph.D. thesis, University of Virginia, 1974.

5. Johnston, J. B., "The Contour Model of Block-Structured Processes" ACM-SIGPLAN Notices 6 (2) (Feb. 1971) pp. 55-82.

CACHE MEMORIES FOR PDP-11 FAMILY COMPUTERS

WILLIAM D. STRECKER
RESEARCH AND DEVELOPMENT
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

ABSTRACT

This paper gives a summary of the research which led to the design of the cache memory in the DEC PDP-11/70. The concept of cache memory is introduced together with its major organizational parameters: size, associativity, block size, replacement algorithm, and write strategy. Simulation results are given showing how the performance of the cache varies with changes in these parameters. Based on these simulation results the design of the 11/70 cache is justified.

Introduction

One of the most important concepts in computer systems is that of a memory hierarchy. A memory hierarchy is simply a memory system built of two (or more[1]) memory technologies. The first technology is selected for fast access time and necessarily has a high per bit cost. Relatively little of the memory system consists of this technology. The second technology is selected for low per bit cost and necessarily has a slow access time. The bulk of the memory system consists of this technology. The use of the hierarchy is coordinated by user software, system software, or hardware so that the overall characteristics of the memory system approximate the fast access of the fast technology and the low per bit cost of the low cost technology. An example of a user software managed hierarchy is core/disk overlaying; and of a system software managed hierarchy is core/disk demand paging. The prime example of a hardware managed hierarchy is a bipolar cache/core memory system.

Until recently the concept of cache memory appeared only in very large scale, performance oriented computer systems such as the IBM 360/85 [1,2] and 370 models 155 and larger. Recently a small cache was announced as an option for the DG Eclipse [3] computer system. A larger, internal cache memory is part of a recently announced DEC PDP-11 family computer system: the PDP-11/70 [4]. The content of this paper is a summary of the research done on the feasibility of using a bipolar cache/core hierarchy in PDP-11 family computer systems.

Cache Memory

A cache memory is a small, fast, associative memory located between the central processor (Pc) and the primary memory (Mp). Typically the cache is implemented in bipolar technology while Mp is implemented in MOS or magnetic core technology. Stored in the cache are address-data (AD) pairs consisting of an Mp address and a copy of the contents of the Mp location corresponding to that address.

The operation of the cache is as follows. When the Pc accesses Mp the address is first compared against the addresses stored in the cache. If there is a match the access is performed on the data portion of the matched AD pair. This is called a *hit* and is performed at the fast access time of the cache. If there is no match -- called a *miss* -- Mp is accessed as usual. Generally, however, an AD pair corresponding to the latest access is stored in the cache -- usually displacing some other AD pair. It is the latter procedure which tends to keep the contents of the cache corresponding to the Mp locations most commonly accessed by the Pc. Because programs typically have the property of *locality*, that is, over short periods of time most accesses are to a small group of Mp locations, even relatively small caches have a majority of Pc accesses resulting in hits. The performance of a cache is described by its *miss ratio* -- the fraction of all Pc references which result in misses.

Cache Organization

There are a number of possible cache organizational parameters. These include:

1. The size of the cache in terms of data storage.
2. The amount of data corresponding to each address in the AD pair.
3. The amount of data moved between Mp and the cache on a miss.
4. The form of address comparison used.
5. The replacement algorithm which decides which AD pair to replace after a miss.
6. The time at which Mp is updated on write accesses.

The most obvious form of cache organization is fully associative with the data portion of the AD pair corresponding to basic addressable unit of memory (typically a byte or word) as indicated by the system architecture. On a miss this basic unit is brought into the cache from Mp. However, for several reasons, this is not always the most attractive organization. First, because procedures and data structures tend to be sequential, it is often desirable to bring into the cache on a miss a block of adjacent Mp words. This effectively gives instruction and data prefetching. Second, because of associating a larger amount of data with an address, the relative amount of the cache storage which is used to store data is increased. The number of words moved between Mp and the cache is termed the *block size*. The block size is also typically the size of the data in the AD pair[2] and is assumed to be that for this discussion.

In a *fully associative* cache any AD pair can be stored in any cache location. This implies that for a single hardware address comparator the Mp address must be compared serially against the address portions of the AD pairs (which is too slow). Alternatively there must be a hardware comparator for each cache location (which is too expensive). An alternative form of cache organization which allows for an intermediate number of comparators is termed *set associative*.

A set associative cache consists of a number of sets which are accessed by indexing rather than by association. Each of the sets contains one or more AD pairs (of which the data portion is a block). There are as many hardware comparators as there are AD pairs in a set. The understanding of the operation of a set associative cache is aided by Figure 1. The n bit Mp address is divided into three fields of ℓ, i, and b bits. Assume that there are 2^i sets. The i bit index field selects one of these sets. The A portion of each AD pair is compared against the ℓ bit label field[3] of the Mp address. If there is a match, the b bit byte field selects the byte (or other sub unit) in the D portion of the matched AD pair.

If there is no match Mp is accessed and (generally) a new AD pair is moved into the cache. Which of the AD pairs to be replaced in the set is selected by the *replacement algorithm*. Typical replacement algorithms are first in, first out (FIFO); least recently used (LRU), or random (RAND).

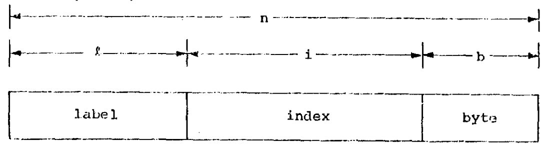


Figure 1 Address fields for a Set Associative Cache

There are two limiting cases of the set associative organization. When the number of sets is the cache size in blocks, only a single hardware comparator is needed and the resulting organization is called *direct mapped*. It is the simplest form of cache organization. When there is only one set, clearly a fully associative cache results.

So far in the discussion there has been no distinction made between read and write accesses. When the Pc makes a write access, ultimately Mp must be updated. There are two obvious times when this can be done. First is at the time the write access is made. Both Mp and the cache (if there is a hit) are updated simultaneously. This strategy is termed *write-through*. Alternatively, onlv the cache can be updated on a write hit and only when the updated AD pair is replaced on some future miss is Mp updated. This strategy is termed *write-back*. The choice between these two strategies involves systems considerations which are beyond the scope of this paper.[4]

There are other possible asymetries in the handling of reads and writes. One possibility is that after a write miss an AD pair corresponding to that access is *not* stored in the cache. This is termed *no-write-allocate*. The alternative is of course termed *write-allocate*.

Cache Memory Simulation

The understanding of memory hierarchies (and programs) has not reached the point where cache performance can be predicted analytically as a function of cache organizational parameters. As a consequence the studying of cache memory behavior is done through simulation. (Some cache simulation results for other computer architectures are reported in [2, 5, 6, 7]). For the purposes of this study a two part simulator was constructed.

The first part was a *PDP-11 simulator*. This is a PDP-11 program which runs other PDP-11 programs interpretively. A variety of properties of the interpreted programs can be collected including the sequence of Mp addresses generated. The latter is termed an *address trace*. The address trace is processed by the second part, the *cache simulator*. This is parameterized by cache organization and determines the miss ratio for a given address trace.

Cache Simulation Results

Since the performance of cache memory is a function not only of cache organization parameters but also of the program run, it is desirable to run cache simulations with a wide variety of programs. Multiplying these by a wide variety of a cache organizational parameters to be simulated resulted in a considerable amount of simulation data of which only the highlights are reported here.

The first experiment was to determine the approximate overall size of the cache memory. Plots of the miss ratio against cache size for several programs[5] are given in Figure 2. (All sizes in both the figures and the discussion are 16 bit PDP-11 words.) A block size of two and a set size of one were held constant. In general the miss ratio falls rapidly for caches up to 1024 words and falls less rapidly thereafter.

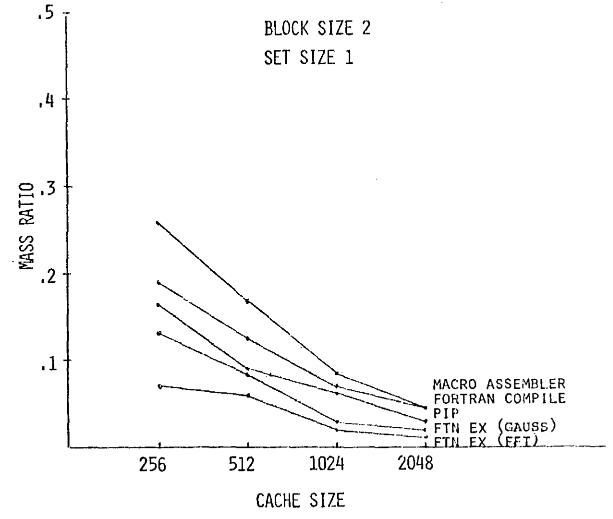


Figure 2 Effect of Cache Size on Miss Ratio

Figure 3 depicts the effect of set size (associativity) on cache performance. In order to clarify the results, Figures 3 through 6 only contain simulation data for a single program (the Macro assembler) which had the highest miss ratio in Figure 2. As expected a larger set size reduces the miss ratio. The largest improvement occurs in going from set size one to set size two. Although not shown, even going to fully associative has little further effect on the miss ratio.

In Figure 4 the impact of block size is shown. Especially in smaller caches, going to a larger block significantly reduces the miss ratio. This is a result of a smaller cache depending more on the prefetching effect for its performance.

The effect of write allocation and replacement algorithm is given in Figure 5. For the program considered there is negligible performance difference across the different strategies.

In Figure 6 the effect of periodically clearing the cache is depicted. This approximates the effect on the cache of rapid context switching in that when a new program is brought in the cache appears "clear" to it. Even completely clearing the cache every 300 Pc accesses only degrades the miss ratio to 0.3. This represents a worst case condition that would be unrealized in practice. For example the "new program" brought in every 300 Pc references might be an interrupt handler. Any program running that often would typically find that the cache always contained information relevant to it. Indeed for the cache organization given it is impossible in 300 accesses to significantly clear a 1024 word cache.

Conclusions

The performance goals of the PDP-11/70 computer system required the typical miss ratio to be 0.1 or less. Analysis of the preceding data with emphasis on the breaks in the curves suggested that the optimal organization was a cache size of 1024 words, block size of two words, and a set size of two. Since the data suggests that replacement algorithm and write allocation strategies have negligible effect, a no-write-allocate strategy and a random replacement algorithm were selected.

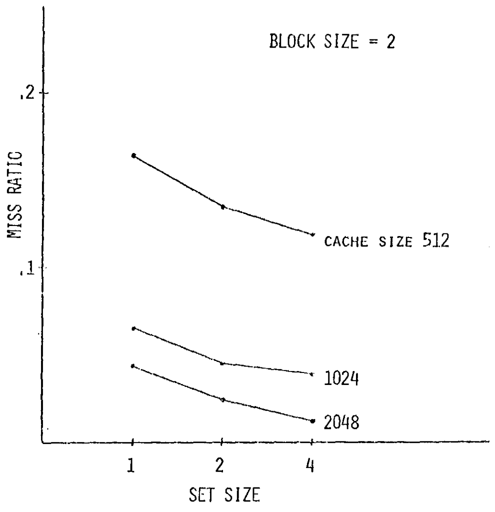


Figure 3 Effect of Set Size on Miss Ratio

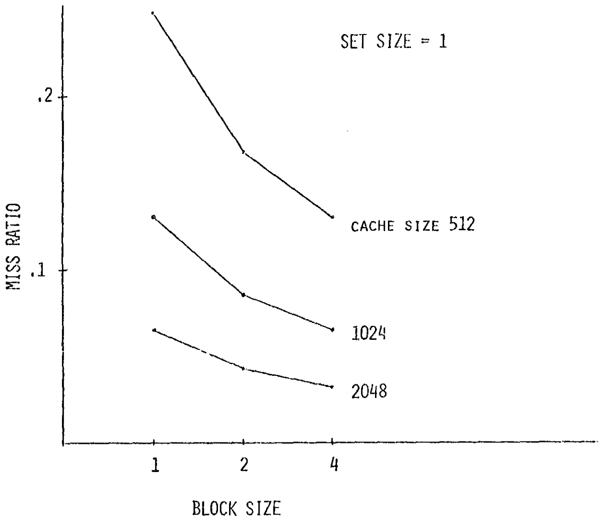


Figure 4 Effect of Block Size on Miss Ratio

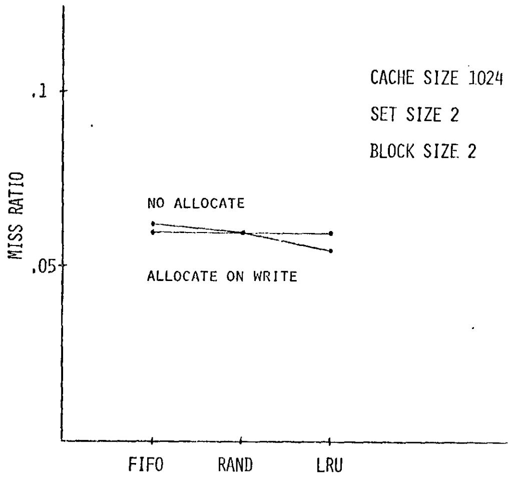


Figure 5 Effect of Replacement Algorithm and Write Allocation on Miss Ratio

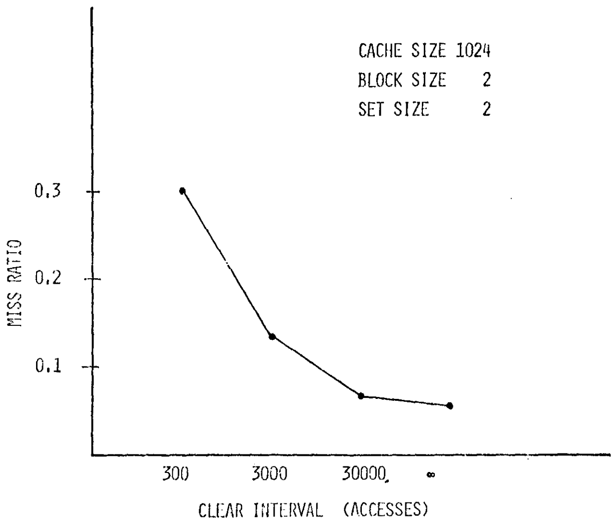


Figure 6 Effect of Clear Interval on Miss Ratio

REFERENCES

1. Conti, C. J., "Concepts for Buffer Storage", *Computer Group News*, Vol. 2, No. 8, March 1969.

2. Conti, C. J., Gibson, D. H., and Pitkowsky, S. H., "Structural Aspects on the System / 360 Model 85, I. General Organization", *IBM Systems Journal*, Vol. 7, No. 1, 1968.

3. *Eclipse Computer Systems*, Data General Corp., 1974.

4. *PDP-11/70 Processor Handbook*, Digital Equipment Corp., 1975.

5. Meade, R. M., "On Memory System Design", Proceedings of the *Fall Joint Computer Conference*, 1970.

6. Bell, J., Casasent, D., Bell, G. G., "An Investigation of Alternative Cache Organizations", *IEEE*

Transactions on Computers, Vol. C-23, No. 4, April 1974.

7. Gibson, D. H., "Considerations in Block Oriented Systems Design", *Proceedings of the Spring Joint Computer Conference*, 1967.

NOTES

1. Memory hierarchies can of course consist of three or more technologies. Discussion and analysis of these multilevel hierarchies is a fairly obvious generalization of the discussion and analysis given here.

2. In a few complex cache organizations such as that in the IBM 360/85 the size of the D portion of the AD pair (called a *sector* in the 360/85) is larger than the block size. That potential level of complexity will be ignored in this discussion.

3. Note that in a set associative cache only the label field must be stored in the cache AD pair -- not the entire Mp address.

4. For the PDP-11/70 system,write-through was chosen. The main impact of this is that each write access as well as each read miss results in an Mp access. Data suggests that in PDP-11's about 10% of Pc accesses are writes.

5. These programs are system and user programs running under the PDP-11 DOS operating system. They include the Macro assembler, FORTRAN compiler, PIP (a file utility program), and FORTRAN executions of numerical applications. The range of miss ratios is typical for the much wider group of programs actually simulated. Indeed the miss ratio for the Macro assembler for a given cache size was the *worst* of any program simulated.

IMPROVING THE THROUGHPUT OF A PIPELINE BY INSERTION OF DELAYS [†]

Janak H. Patel and Edward S. Davidson
Coordinated Science Lab
University of Illinois
Urbana, Illinois 61801

Summary

A pipeline is defined to be a collection of resources, called segments which can be kept busy simultaneously. A task once initiated, flows from segment to segment for its execution. A collision occurs if two or more tasks attempt to use the same segment at the same time.

The collision characteristics of a pipeline with respect to a schedule of task initiations are investigated. A methodology is presented for modifying the collision characteristics with the insertion of delays so as to increase the utilization of segments and hence the throughput under appropriate scheduling.

I. Introduction

Pipelines are becoming increasingly common in many computers, sometimes for achieving high speed computation at a lower cost than would result from simply using higher speec electronic components. However, in most cases it is used because of a better performance per unit cost over other architectures. A pipeline as defined here is a collection of resources called segments which can be kept busy simultaneously. A task once initiated, flows from segment to segment for its execution, in a predetermined manner. The effectiveness of the pipeline lies in the fact that a task can be initiated before the completion of some previously initiated tasks resulting in high performance and segments can be special rather than general purpose resulting in low cost. We term a pipeline in which all the tasks have identical flow patterns, a single function pipeline. In a multifunction pipeline there are two or more distinct possible flow patterns and each task uses one of these flow patterns. Each flow pattern is identified by a function name and it can be displayed in a reservation table, such as Figure 1 and 6. Rows correspond to segments and columns to units of time. A function name, denoted by a single capital letter, is placed in row i and column j (cell (i,j)) if after j units of execution a task with that function name requires segment i. We shall consistently use X as a function name in single function pipelines. Fig.6 is a reservation table of a multifunction pipeline with two distinct flow patterns for two functions A and B.

In our model we assume that a task once initiated must flow synchronously without preemption or wait. There is no restruction on the flow patters, however. In Fig. 1, multiple X's in a row may indicate either a slow segment or segment reusage (feedback). Multiple X's in a column indicate parallel computation. It is the reusage of a segment which poses a problem, namely, two or more tasks may attempt to use the same segment at the same time, resulting in a collision. However, in multifunction pipelines even without any reusage, a collision may occur because of two or more independent and distinct flows of tasks.

In previous work, the central problem treated is to schedule the tasks in a given pipeline so as to schieve high throughput without causing any collision. This problem was first investigated in [1]. Subsequent work on this problem is reported in two doctoral theses [2,3]. An overview of some related results and a more comprehensive bibliography can be found in [4]. Our investigation is from a different perspective and seeks a methodology for modifying the reservation table of a given pipeline so as to increase the utilization of segments and hence the throughput under appropriate scheduling.

The pipeline utilization is limited by its collision characteristics which are a result of the usage patterns of the segments. One way of modifying usage pattern is by segment replication. Another way is to remove our assumption regarding the waiting of a task between two steps and provide internal storage buffers which allow variable delay between segments [4]. Still another way of changing a usage pattern is by inserting noncompute segments, which simply provide a fixed delay between some computation steps. It is the modification of a pipeline by the use of noncompute segments which is the concern of this paper. It is assumed that any computation step can be delayed by inserting usage of a noncompute segment, where each X in the reservation table is considered to be a computation step.

We shall first consider single function pipelines for ease of understanding, since the notational complexity of multifunction pipelines is considerable.

II. Single Function Pipelines

We start by investigating some collision characteristics of a single function pipeline (referred to simply as pipelines in this and the following section). A usage interval of a segment is defined to be a time interval between two reservations (X's) of that segment by a single task. For example in Fig. 1, all usage intervals of S_0 are 2, 3 and 5. Let $\underline{F}$ be the set of all usage intervals of a pipeline; e.g., $\underline{F}=\{1,2,3,5\}$ for Fig. 1. Clearly any two tasks will cause a collision if and only if they have the same initiation time interval as one of the usage intervals.

A sequence of task initiations can be completely described by a sequence of initiation intervals between successive tasks (also known as latency). For example, task initiations at time instants 0, 3, 5, 9 and 12 can be described by the latency sequence $\langle 3,2,4,3\rangle$. An initiation interval of 0 is not permissible. Let $\underline{G}$ be the set of all initiation intervals (not just the intervals between successive initiations) of a latency sequence. Thus $\underline{G}$ for the latency sequence $\langle 3,2,4,3\rangle$ is $\{2,3,4,5,6,7,9,12\}$.

If a subsequence of latencies appear periodically in an infinite sequence, it is termed an initiation cycle. Thus a cycle (2,3,2,5) implies an infinite initiation sequence $\langle 2,3,2,5,2,3,2,5,2,3,2,5,2,\ldots\rangle$. A constant latency cycle is a cycle with only one latency latency; e.g., cycle (4). Let the period, p, of a cycle be defined as the sum of the latencies in the cycle. Thus the period p of cycle (2,3,2,5) is 12 and p of cycle (4) is 4. The average latency, ℓ_a of a cycle is the average of the latencies of the cycle. For example, ℓ_a for cycle (2,3,2,5) is 12/4=3. This implies an average initiation rate of one task every 3 time units.

The initiation interval set $\underline{G}$ of a cycle is simply the set $\underline{G}$ of the infinite initiation sequence implied by the cycle. Thus $\underline{G}=\{4,8,12,16,20\ldots\}$ for cycle (4) and for cycle (2,3,2,5) $\underline{G}$ is $\{2,3,5,7,9,10,12,14,15,17,19,21,22,24,26,\ldots\}$. Let $\underline{G}$ mod p be the set formed

[†] This research was supported in part by the National Science Foundation under Grants GJ-35584X and GJ-40584 and in part by the Joint Services Electronics Program (U.S.Army,U.S.Navy, and U.S. Air Force) under Contract DAAB-07-72-0259.

by taking modulo p equivalents between 0 and (p-1) of the elements of $\underline{G}$. For cycle (2,3,2,5) with p=12, $\underline{G}$ mod 12={0,2,3,5,7,9,10} and for constant cycle (4) with p=4 $\underline{G}$ mod 4={0}. It can be shown that $\underline{G}$ and $\underline{G}$ mod p of a cycle have the following simple properties, remembering that 0 is not a permissible initiation interval.

P1.a. if $g \neq 0$ then $g \in \underline{G}$ mod p $\Rightarrow g+ip \in \underline{G}$ $\forall i \geq 0$

b. $0 \in \underline{G}$ mod p and $ip \in \underline{G}$ $\forall i \geq 1$ always.

P2.if $g \neq 0$ then $g \in \underline{G}$ mod p $\Leftrightarrow (p-g) \in \underline{G}$ mod p.

It is useful to introduce the set $\underline{H}$, the complement set of $\underline{G}$ in $\underline{Z}_+$, the set of positive integers. Clearly $\underline{H}$ mod p = $\underline{Z}_p$-$\underline{G}$ mod p. Where $\underline{Z}_p$ is the set of integers modulo p. Then the following is a direct consequence of P2.

P3. $h \in \underline{H}$ mod p $\Leftrightarrow (p-h) \in \underline{H}$ mod p.

An initiation interval between two tasks is said to be <u>allowable</u> with respect to a pipeline if these tasks do not collide in the pipeline. A cycle is allowable with respect to a pipeline if all its initiation intervals are allowable. Conversely, we also say that a usage interval or a pipeline is allowable with respect to a cycle if no collision occurs. A collision occurs in a pipeline when a cycle is followed iff (if and only if) some initiation interval of the cycle equals a usage interval of the pipeline. Thus a cycle is allowed by a pipeline iff there are no elements common between the usage interval set, $\underline{F}$, of the pipeline and the initiation interval set, $\underline{G}$, of the cycle; i.e., iff $\underline{F} \cap \underline{G} = \Phi$, or equivalently, iff $\underline{F} \subseteq \underline{H}$. Thus $\underline{H}$, the complement set of $\underline{G}$ can be described as the set of allowable usage intervals with respect to the given cycle. By using the property P1 of $\underline{G}$, the allowability condition can be reduced to the following theorem.

<u>Theorem 1</u>: A cycle with period p and initiation interval set $\underline{G}$ is allowed by a pipeline with usage interval set $\underline{F}$, iff ($\underline{F}$ mod p) $\cap$ ($\underline{G}$ mod p) = Φ. □

A constant latency cycle (ℓ) has p = ℓ. Its $\underline{G}$ mod p is always {0} and hence the following.

<u>Corollary 1.1</u>: A constant cycle (ℓ) is allowed by a pipeline iff no usage interval is an integral multiple of ℓ.

It is helpful to look at the allowable usage interval set $\underline{H}$ to see what allowable pipelines can be constructed for a given cycle. Let a row which has an X in each of columns $t_1,t_2,\ldots t_k$ be denoted as row $\{t_1,t_2,\ldots t_k\}$; e.g., the 2nd row of Fig. 1 is row {1, 2,4}. A pipeline is allowed by a cycle if all its rows are allowed. To construct an allowable row we can start by placing an X in some column i. We can place another X in some column j, only if the usage interval $|i-j| \in \underline{H}$; a third X in some column k if $|i-k|$ and $|j-k| \in \underline{H}$, and so on.

However, it is convenient to restrict the column numbers to be between 0 and (p-1), and still retain all the useful information. For this, let us define two elements $i, j \in \underline{Z}_p$, to be <u>compatible</u> if $|i-j| \in \underline{H}$ mod p. The use of the absolute quantity can be avoided by using property P3 of $\underline{H}$ mod p. Thus we have the following lemma.

<u>Lemma 2.1</u>: Two integers $i,j \in \underline{Z}_p$ are compatible iff (i-j) mod p $\in \underline{H}$ mod p. □

Using the definition of compatibility or the above lemma we can form all the compatibility classes on the elements of $\underline{Z}_p$, given a cycle. A <u>compatibility class</u> is one in which each element is compatible with every other element in the class. We need to form only the maximal compatibility classes. A <u>maximal compatibility class</u> is not a subset of any other compatibility class.

If $\{z_1,z_2,\ldots z_k\}$ is a compatibility class with respect to some cycle then the row $\{z_1,z_2,\ldots z_k\}$ is allowed by that cycle. This is because by the definition of compatibility all usage intervals $|z_i-z_j|$ are allowable. In this way we can produce only a limited number of allowable rows. However, with the use of' property P3 and Lemma 2.1 it is possible to construct other allowable rows as follows.

<u>Theorem 2</u>: Given a cycle with period p, the following rows, and only those rows, are allowed by the cycle:

row $\{z_1+i_1p, z_2+i_2p,\ldots\}$ $\forall$ integers $i_1,i_2,\ldots$ and $\forall$ compatibility classes $\{z_1,z_2,\ldots\}$ of the cycle. □

Consider a problem in which a pipeline, characterized by its usage interval set, is given and one has complete freedom in choosing an allowable initiation sequence. Bounds on the minimum average latency of such sequences and a branch-and-bound algorithm to discover a minimum average latency allowable cycle are reported in [1] and [4]. Minimum average latency cycles maximize segment utilization, where utilization is measured as the percent of time the segment remains busy.

Here we consider the reverse problem. Namely, a cycle is given and one has complete freedom in choosing any allowable usage pattern. While the solution to the former problem is useful for scheduling a given pipeline, the solution to this problem is useful for designing a pipeline for a given schedule. Theorem 2 completely characterizes the entire class of allowable pipelines. We shall soon see that it is possible to put an upper bound on segment utilization with the help of the compatibility classes. To achieve maximum utilization of a segment for a given cycle, we must increase the number of usages per task; i.e., increase the number of X's in a row. Theorem 2 gives all possible allowable rows and it implies that the maximum number of X's in any allowable row is equal to the size of the largest compatibility class. Thus the maximum achievable utilization of a segment with respect to a given cycle is the ratio of the size of the largest compatibility class to the average latency of the cycle.

<u>Example 1</u>: For cycle (1,9), p=10, average latency ℓ_a=5, $\underline{G}$ mod 10 = {0,1,9} and hence $\underline{H}$ mod 10 = {2,3,4, 5,6,7,8}. The maximal compatibility classes containing 0 are {0,2,4,6,8}, {0,2,4,7}, {0,2,5,7}, {0,3,5,7}, {0,2,5,8}, {0,3,5,8}, and {0,3,6,8} of which the largest has size equal to 5. Note that classes containing 0 are sufficient to characterize all classes since a constant may be added modulo p to all elements of a compatibility class to produce another compatibility class. Thus by Theorem 2, no allowable row has more than 5 X's. This implies that the maximum possible segment utilization with cycle (1,9) is 5/5=100%. □

<u>Example 2</u>: For cycle (2,3,7), p=12, ℓ_a=12/3=4, $\underline{G}$ mod 12={0,2,3,5,7,9,10} and hence $\underline{H}$ mod 12={1,4,6,8, 11}. The maximal compatibility classes containing 0 are {0,1}, {0,4,8}, {0,6}, and {0,11} of which the largest has 3 elements. Thus the maximum number of X's in any allowable row is 3 which in turn implies a maximum segment utilization of 3/4=75%. In other words no allowable pipeline for cycle (2,3,7) has a segment which is busy more than 75% of the time. □

Among cycles with same ℓ_a, those which allow a high utilization and hence more economical realization are clearly preferable. Furthermore they offer more flexibility in pipeline design. Let us define a cycle to be perfect, if it allows a 100% segment utilization; e.g., cycle (1,9) of Example 1. Unfortunately we cannot test the perfectness of a cycle without forming the compatibility classes. However, we know a special class of perfect cycles which are of considerable interest in single function pipelines.

Theorem 3: All constant latency cycles are perfect.

Proof: For constant cycle (ℓ), $\underline{G}$ mod p=$\{0\}$ and thus $\underline{H}$ mod p=$\{1,2...(\ell-1)\}$. One can verify that $\{0,1,2,...,(\ell-1)\}$ is a compatibility class with ℓ elements. Hence the upperbound on the segment utilization is ℓ/ℓ = 100%. □

III. Noncompute Segments

In this section we consider the addition of noncompute segments to a pipeline to make it allowable for a given cycle. The effect of delaying some computation step can be displayed in a reservation table by writing a 'd' before the X which is being delayed. Each d indicates one unit of delay called an elemental delay. In the absence of any other information on precedence, we must assume that all the steps in a column must be completed before any steps in the next column are executed. Therefore, if the steps in column 2 of Fig. 1 are unevenly delayed, we must store the output of some steps so that all the outputs are simultaneously available to the steps in column 3 of Fig. 1. The effect of delaying the step in row 0, column 2 (X_{02}) of Fig. 1 by 2 units and X_{22} by 1 unit is shown in Fig. 2. The elemental input delays d_1, d_2, and d_3 require the elemental output delays d_4, d_5, and d_6. Now given some integer i between 0 and (p-1), we are in a position to delay any step arbitrarily such that the step occurs in a column number equivalent to i modulo p. Thus given a cycle, we can make any row of a given reservation table to look like one of the rows of Theorem 2; provided of course, the row does not have more X's than the size of the largest compatibility class of the cycle. Hence we have the following theorem.

Theorem 4: For a given cycle, a pipeline can be made allowable by delaying some of the steps, iff the number of X's in each row of the reservation table is less than or equal to the size of the largest compatibility class of the cycle. □

Corollary 4.1: For a given constant latency cycle (ℓ), a pipeline can be made allowable by delaying some steps, iff there are no more than ℓ X's in each row of the table. □

An important implication of Corollary 4.1 is that by adding elemental delays to a pipeline one can always fully utilize a single function pipeline with the use of a cycle with constant latency equal to the maximum number of X's occurring in any single row of the reservation table. Full utilization of a pipeline here, means that at least one segment is busy all the time. Thus the maximum achievable throughput of that pipeline is attained. Of course complete redesign or replication of selected segments to reduce the number of X's in a row may allow higher throughput.

Example 3: The reservation table of Fig. 1 is to be made allowable with respect to cycle (1,5). The resulting table appears in Fig. 3. For cycle (1,5), p=6, $\underline{G}$ mod 6=$\{0,1,5\}$ and hence $\underline{H}$ mod 6=$\{2,3,4\}$. The maximal compatibility classes containing 0 are: $\{0,2,4\}$ and $\{0,3\}$. The first row of Fig. 3 is row $\{0,2,10\}$, which resulted from the class $\{0,2,4\}$ by constructing row $\{0,2,4+p\}$ as per Theorem 2. The second row, $\{1,3,5\}$ results from class $\{0,2,4\}$ and the third row, $\{2,4\}$ results from class $\{2,4\} \subset \{0,2,4\}$. Thus all the rows are allowable. □

Once we have a modified table, we need to assign the elemental delays to noncompute segments. Noncompute segments are physical resources like any other segment and may be shared by various elemental delays for their efficient utilization. Two elemental delays d_i and d_j are defined to be compatible if $|t_i-t_j|$ mod p $\in \underline{H}$ mod p. Where t_i and t_j are labels of the columns in which d_i and d_j appear. Clearly, if d_i and d_j are compatible, they can share one noncompute segment because the usage interval $|t_i-t_j|$ is allowable. Using the above definition we can form the maximal compatibility classes of all the elemental delays present in the solution. All the elements of a compatibility class can share a single noncompute segment. Now the problem reduces to the standard covering problem; i.e., finding the minimum number of compatibility classes which cover all the elemental delays.

Example 4: The set of elemental delays of Fig. 3 is $<d_1,d_2,d_3,d_4,d_5,d_6,d_7>$. Their corresponding column numbers are <3,6,7,8,9,2,3>. For cycle (1,5), $\underline{H}$ mod 6 is $\{2,3,4\}$ (from Ex. 3). Thus $\{d_1,d_2\}$, $\{d_1,d_3\}$, $\{d_2,d_4\}$, $\{d_2,d_5\}$, $\{d_2,d_6\}$, $\{d_2,d_7\}$, $\{d_3,d_5\}$, $\{d_3,d_7\}$ are the maximal compatibility classes. Noting that the subsets of maximal compatibility classes are compatibility classes, one of many possible minimal covers is $\{d_1,d_2\}$, $\{d_4\}$, $\{d_5\}$, $\{d_6\}$, $\{d_3,d_7\}$. Thus 5 noncompute segments are required. The assignement above is shown in Fig. 4, where S_3 through S_7 are noncompute segments. □

Besides reducing the number of noncompute segments in a solution, it is also important to reduce the added execution delay. The execution delay of a task in Fig. 1 is 6 units while in the modified table of Fig. 4 it is 11 units. In situations where it often becomes necessary to empty the pipeline; e.g., due to logical dependancies among tasks, the execution delay of a task can become an important parameter in determining the overall throughput. Therefore, we shall take the added execution delay as the objective function to be minimized. Now the problem of making a pipeline allowable can be formulated as follows.

Let I and J be the number of rows and columns in the given reservation table. Let d_{ij} and d'_{ij} be the number of elemental delays to be inserted respectively at the input and output of a step X_{ij} of the reservation table. If no X occurs in cell (i,j) of the table then d_{ij} and d'_{ij} are defined to be zero. Some other d_{ij} can be set to zero if it occurs between two consecutive computation steps which are indivisible. Let D be the added execution delay. Then the problem can be formally stated as:

$$\text{Minimize } D = \sum_{0 \le j < J} \left(\max_{0 \le i < I} (d_{ij}) \right)$$

subject to the constraints,

integer $d_{ij} \geq 0$.

$$[(c-b)+d'_{ab}+d_{ac}+\sum_{b<j<c} \left(\max_{0 \le i < I} (d_{ij}) \right)] \bmod p$$

$$\in \underline{H} \bmod p.$$

for each pair $<X_{ab},X_{ac}>$ with c > b.

where, $\underline{H}$ is the set of allowable usage intervals with

	0	1	2	3	4	5	6	7	8	9	10	
S_0	X		X									
S_1		X		X		X						
S_2			X		X							
S_3				X			X					$\{d_1,d_2\}$
S_4									X			$\{d_4\}$
S_5										X		$\{d_5\}$
S_6			X									$\{d_6\}$
S_7				X				X				$\{d_3,d_7\}$

FP-4687

Figure 4. Assignment of elemental delays to noncompute segments

	0	1	2	3	4
S_0	A		B		B
S_1		AB		B	
S_2	B		A	AB	

FP-4688

Figure 6. Reservation table for a multifunction pipeline

	0	1	2
S_0	X		X
S_1	X	X	

FP-4689

cycle (2). $\underline{H}$ mod 2 = {1}

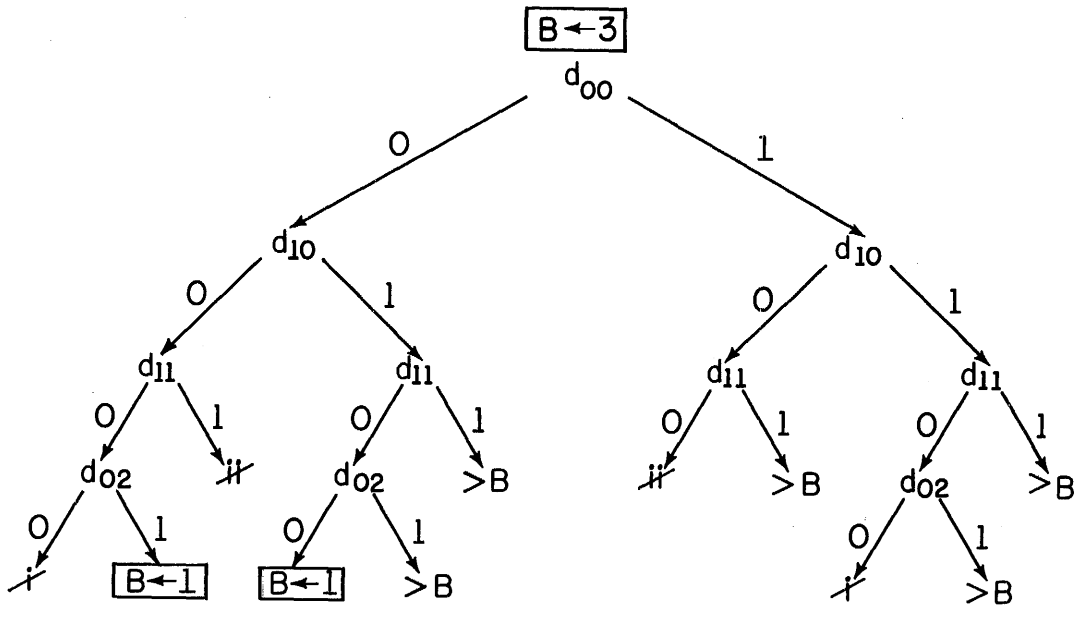
Added delay:

$$D = \max\{d_{00},d_{10}\} + d_{11} + d_{02}$$

Constraints:

(i) $[2 + \max\{d_{00},d_{10}\} - d_{00} + d_{02} + d_{11}\}$ mod $2 \in \{1\}$.

(ii) $[1 + \max\{d_{00},d_{10}\} - d_{10} + d_{11}\}$ mod $2 \in \{1\}$.



Optimum solutions are: 1. $d_{00} = d_{10} = d_{11} = 0$. $d_{02} = 1$.

2. $d_{00} = d_{11} = d_{02} = 0$. $d_{10} = 1$.

Figure 5. Making the pipeline allowable for cycle (2): A branch-and-bound search for optimum solutions.

ON-LINE ARCHITECTURE TUNING USING MICROCAPTURE

A. M. Abd-Alla and Laird H. Moffett
The George Washington University and
the Naval Research Laboratory
Washington, D.C.

ABSTRACT

The modification or tuning of the microcode in a computer that utilizes a writable control store is one method whereby a program's execution time can be improved. A method for automatically performing a microcode tuning or synthesis has been developed by Drs. Karlgaard and Abd-Alla and is discussed in detail in [1]. Presented is an extension of this effort which allows the microcode tuning to be performed on-line on program loops. This is accomplished by gathering data on characteristics of the program during its execution, utilizing this data to generate the information required to tune the microprogram, initiating the on-line tuning procedure, and transferring to the tuned routine to complete the execution.

INTRODUCTION

The primary effort in utilizing user microprogrammed machines is usually to write a microprogram that performs in a manner similar to an assembly language macro. This is usually done because the user requires greater throughput for his program than the standard microcode allows. This remicroprogramming to increase throughput (or tuning) is performed manually. The general steps required to perform this tuning are:

(1) Identify segments of the microcode which permit possible optimization

(2) Create a new micro-routine which performs the same function of the chosen microcode segment so as to improve machine performance

(3) Load the new micro-routine into the machine and communicate these architecture changes to the system programs so that the new architecture can be utilized.

A method for automatically performing microcode tuning or synthesis has been developed by Abd-Alla and Karlgaard [1]. Although this method is general enough to be applied to many user applications, it requires considerable overhead which prohibits its use for real time applications. This paper is an extension of the microcode tuning effort to allow the tuning of program loops to be accomplished "on the fly" during run time. The technique itself is entitled "Microcapture Timing" where microcapture refers to the capturing of the program loop and tuning refers to the remicroprogramming of the machine. To date there is no other method known to the authors that performs architecture tuning on-the-fly.

The following sections present briefly: the tuning algorithm developed in [1], the drawbacks of this method for real-time applications, a method for performing the trace to generate the required statistics, an approach to performing the actual synthesis, and future efforts to be performed by the authors in examining the utility of this tuning method.

Heuristic Synthesis Algorithm

A synthesis procedure was presented [1] whereby the time required to perform operations in a program loop could be reduced. Basically this method required a trace of the program in order to gather data concerning the program operation. This data would enable one to detect the presence of loops, the number of times a specific memory location has been addressed within the loop, and whether that address was an instruction or an operand location. Then, as shown in Fig. 1, from statistics generated from the collected data the loop boundaries were determined and the most often used memory locations holding data referenced within the loop were determined. Those were placed in the GP microregisters and a new set of microinstructions were created which utilized a microoperation stream equivalent to a register-to-register stream. After the loop was completed, a restore operation was performed. The synthesized microcode, along with the preload and restore operations, would then be called by a macro developed by the assembler or compiler. When this internal macro was called during program execution, the GP micro registers would be preloaded, the tuned microcode for the loop executed and the system restored for a continuation of the rest of the execution. This method has shown loop execution to be increased by a factor of 8 for a data movement program. Several drawbacks to utilizing this procedure in this manner are that: (a) A trace program to generate usage data increases the amount of overhead for the program, (b) A program to generate the statistics must be run prior to selection of the data to be placed in the GP microregisters, (c) The synthesis is performed in software rather than by a microprogram.

The Trace and Statistics Generation

If one can perform a trace which does not increase the program execution overhead and can generate the statistics as the program is operating, then two major hurdles have been removed. This would allow the algorithm to be truly automatic. An approach to accomplishing this is to perform the trace and statistics calculation in hardware. This may seem very difficult at first but a relatively simple scheme for accomplishing this is described below.

Based on a study of programs performed by IBM on the IBM 360 the average number of assembly language instructions in a program loop is 8 [2]. Therefore, if we examine every location as it is accessed and maintain a file of the last 16 or 24 locations used, we will encompass most program loops. These 16 or 24 file locations will contain the address of the instruction plus the address of any operands the instruction may utilize during its execution. The statistics that are required for each location accessed during the execution of the loop are the number of times the location is accessed, determination of whether that location contained an instruction or an operand, and the determination of whether or not it is jump instruction.

This can be accomplished by the use of a content addressable memory (CAM) and a high speed random access memory (RAM) used in conjunction with the microstore. See Fig. 2. The basic approach is as follows: as the computer reads an instruction from memory, the Content Addressable Memory is simultaneously searched using the location being addressed, i.e., contents of program counter as a target.

If there is a match, a flag in the CAM would be set corresponding to a repeated location. For each CAM word there is a corresponding word in the random access memory. Contained in that word is: (a) the count or whether that location has been recently addressed, (b) whether or not it is a jump instruction, (c) whether it is an instruction or an operand, (d) whether the instruction contains an indirect address, and (e) computer status information. The accessed RAM data is compared to 110 (count, jump and no operand) and if it is equal, the synthesis phase may be initiated. If it is not equal to 110, the count field is set to 1 and the word is stored back in the RAM location.

If there is no match, then the next available space in the CAM is loaded with that addressed location. (This space is determined by the CAM/RAM address counter which is modulo the number of words in the CAM.) The corresponding location in the RAM has its count set to 0 and its jump, operand, indirect and status locations set accordingly.

Whether the accessed location is an instruction or operand location can be determined by the computer phasing. To determine if it is a jump instruction or if the instruction contains an indirect address either the instruction decoder has to set a flag or it can be determined in the microcode. The status information can be determined by examining the pertinent flip-flops and registers. This hardware will then generate the loop statistics required for synthesis. Notice that the above hardware will require the loop to be computed twice with the standard microcode before the synthesis phase of the tuning process begins.

Performing The Tuning

The next phase of the tuning process is the actual synthesis of the new microcode. In selecting the synthesis method, one must remember that the execution of the synthesis program is pure overhead. This "wasted" time will not be made up until several executions of a given loop in the application program have been performed. Because of this fact, the authors found it necessary to perform the synthesis algorithm by microprogram rather than by software.

There are several basic approaches to the tuning which may be utilized. One is the synthesis of a new microprogram to execute the loop as performed in [1]. The primary difficulty with this method for run time utilization is the large amount of time that would be required to perform the deletion and creation of new microinstructions. Also, new microinstructions would have to be added that would allow manipulation of its own memory contents and specific bit generation and deletion facilities.

As a basis for determining the time it would take to perform the synthesis routine described above, a modified HP 2100A was chosen. This machine is a modified version of the HP 2100 Computer. One of the modifications which is essential is the ability to lead the microstore with microcode commands. It was assumed in the modification that the procedure and the tuning for loading the microstore from a microprogram was different than loading the microstore from software. The technique used would load the load buffers directly using special yet simple additional hardware. It would then require approximately 2 microcycles to load the microstore under microprogram control. To write a new microprogram for each instruction requires either a microprogram that has a file of microprograms that would utilize general purpose microregisters or an analysis microprogram that would modify each microinstruction as it was executed. This latter approach is prohibitive due to the amount of micro accessing and storing required and the difficulty (new microinstructions required) in modifying the subfields that require modification. The former approach would require a rather large microprogram and would still have the difficulty of modifying the subfields to place the proper microregister in the instruction. Either of these methods would force the synthesis routine to take too long. The number of times through the loop would have to be large in order to benefit from the tuning.

A modification of this method which would reduce the synthesis time significantly is not to create or delete microinstructions but to leave the instructions as they are, link them together and remove the instruction access. This can be effectively performed by using a pointer list approach. This approach would have a microprogram that would create a list of pointers. These pointers would contain the address of the initial microinstruction for each machine instruction in the loop and the address of the operand to be fetched from memory as the actual instruction would. This pointer list would be a series of jump instructions that perform two specific tasks: (1) load the address of the operand into the appropriate microregister as the initial accessing of the instruction would have done and, (2) jump to the proper address in microstore to initiate the execution of the microinstructions. The formation of these jump instructions would be performed after the instruction fetch and before the execution of each instruction. As one can observe, the microinstructions in the pointer list routine would be similar (a jump instruction); only the data would be changed. The improvement using this synthesis method is the difference between accessing an instruction in main memory versus accessing an instruction in the microstore.

The formation of the pointer list then is the synthesis routine. The synthesis routine is initiated after having received the "go ahead" signal from the microcapture hardware. It performs the following functions: (i) permits normal access of the machine instruction, (ii) transfers the operand address from the microregister that it was loaded into the proper segment of the microstore register (general purpose register in most machines with writable control store), (iii) transfers the address as determined by the instruction decoder or mapper to the proper segment of the microstore load register, (iv) places the proper instruction bit pattern into the microstore load register, (this bit pattern is the same for every instruction in the pointer list), (v) load the microinstruction into proper location in the microstore, and (vi) execute the instruction as usual. The synthesis procedure terminates when the synthesis of the returning jump is performed. Some of these functions can be performed in hardware under microcode command and others can be performed by microcode. As can be seen, the savings here is in the time to perform the synthesis.

During execution of the loop under pointer list control the program counter is incremented when the pointer list is reaccessed and reset to the initial loop address when the returning jump is performed. Any conditionals interior to the loop increment the program counter naturally. The loop execution is terminated when the program counter does not match an instruction address in the CAM. The contents of the program counter is the address of the next instruction that is accessed from main memory.

The pointer list algorithm can be modified one step further with only a small increase in overhead. By using the additional microregisters that are not used by the standard microinstruction set, or by including special registers in the microcapture hardware, further improvement can be made to the performance of the pointer list approach. During the last step of the synthesis procedure (when the instructions are being executed), if the count in the microcapture hardware of that memory access location is 1, then load the operand into a special register. Now each time that location is accessed, its data are fetched from the special register rather than main memory. This will require additional provisions in the hardware to keep track of the mapping between the memory locations of the operands as referenced by the instructions and the corresponding special registers. Prior to each main memory access, during loop execution, a CAM search is performed to determine if the data being accessed is in a special register or in main memory and the memory access microinstruction is altered if necessary. The method used in the simulation was to expand the number of bits in the RAM in order to place the operand in the RAM.

Performance Analysis and Simulation

The cost of the microcapture architecture tuning is the time required to actually perform the synthesis, the restore for continuation and the additional hardware required for statistics generation. The question immediately arises as to the trade-offs in the implementation. If it can be shown to be throughput effective, then the additional hardware is justifiable. To determine this requires some analysis, a detailed simulation of the scheme and an investigation into typical loop profiles.

To begin with a determination of the crossover point between performing the algorithms and not performing the algorithm in an operating situation is required.

A simple analysis to determine the crossover point is presented below. Let us assume for simplicity that the time to perform the synthesis is directly proportional to the number of instructions in the loop.

Let t_n = time required to perform the synthesis; the loop is executed once as the synthesis is being performed.

Let t_1 = time to execute the loop once using unsynthesized instructions.

Let t_2 = time to execute the loop once using the synthesized instructions.

Also assume $t_n = bt_1$ where $b > 1$ and $t_1 = at_2$ where $a > 1$.

Let y = number of cycles through the loop.

Now let us examine two specific cases:

Case 1 $y = 2$

Since two cycles are required before the synthesis begins, then no time is gained or lost.

Case 2 $y > 2$

The time to execute the loop y times with no tuning is t_1y and the time to execute the loop y times using the algorithm is

$$2t_1 + t_n + t_2 (y - 3) \qquad (1)$$

To determine the break-even point:

$$t_1 y = 2t_1 + t_n + t_2 (y - 3) \qquad (2)$$

$$\text{or} \quad y = 2 + \frac{ab - 1}{a - 1} \qquad (3)$$

To determine the values a and b will require a simulation. Let us take an example. Let the execution improvement be two hence a = 2 and the time to synthesize relative to regular loop operation be a factor of 5 hence b = 5, then

$$y = 2 + \frac{2 \times 5 - 1}{2 - 1} = 11 \text{ times} \qquad (4)$$

through a loop before improvement occurs. So if the average number of times through a loop (which references a number of locations in the CAM) is greater than 11 the method is useful.

The simulation of both of these synthesis techniques were performed on the HP 2100A for two programs: a data move program in which data is moved from one area of core memory to the other, and a linear search program where the target is sequentially stepped through the memory locations containing the data being searched. The basic timing results are shown in Figures 3 and 4.

As one can observe the crossover between time on the standard HP 2100A and the HP 2100A with the pointer list technique employed is 4.0 times through the loop for the data move program and 4.3 for the linear search. Further modification of the pointer list technique to incorporate the special registers for repeated operands gives a crossover of 4.0 for the data move and 4.1 for the linear search. The percent improvement in per loop performance is given in the table below.

	Data Move	Sequential Search
Pointer List	55	86
Modified Pointer List	98	150

Notice that in the linear search simulation the entire synthesis phase had not been complete when the target had been located. Likewise, although to a lesser extent, with the data move because the return jump was not executed. Making a linear approximation to the graphs in Figures 3 and 4 at the synthesis point and substituting these approximations into the analytical equations yields a crossover of 3.98 and 4.3 for the data move and sequential search respectively for the pointer list technique. Similarly for the modified pointer list technique we obtain 3.99 and 4.2 as the crossover points.

Cache Versus or in Combination with Microcapture

To compare the performance of these two architectural techniques in the execution of small loops the data move and the linear search programs were analyzed. This analysis was based on the HP 2100A simulation using a faster memory. The assumptions made for this analysis are: the HP 2100A operation remains the same, the cache memory cycle time is equal to two (2) microcycles, and the percentage of bits on the cache is 100. Figures 5 and 6 show the difference between the HP 2100A using microcapture techniques and the HP 2100A using a cache memory.

It is also interesting to investigate using the two techniques in combination with one another. Figures 7 and 8 show the performance improvement that can be gleaned by using these techniques together.

There are three primary areas for tradeoffs between the two techniques. These areas are performance when executing loops, types of systems each can be used with, and the cost of implementing each technique.

A performance comparison can be made by examining Figures 5 and 6. The crossover point between the cache and the modified pointer list is 10.0 times through the loop in one case and 15.7 times through the loop in the other. For the microcapture tuning to be better than the cache requires the average number of times through captured loops to be greater than 10 or 15.

In order for a computer system that utilizes a cache memory to be easily implemented and produce a relatively high hit/miss ratio, it has to be able to move the data into its cache memory in blocks rather than as single instructions. This requires the computer to have a paged memory management system. Typically computers that have paged memory systems are relatively expensive ($75,000 and up). This limits the usage of a cache to larger systems. The microcapture tuning technique, on the other hand, can be utilized in any level of computer system.

The third area, cost of implementation, is important due to the order of magnitude difference between the cost of the two techniques. A 1024 word 16 bit/word cache memory costs approximately $2,500 while the cost of implementing the microcapture hardware is about $300.

From an examination of Figures 7 and 8 it is evident that a performance improvement can be gleaned if the two techniques are used in conjunction with one another. Since the performance improvement is present and the cost of including microcapture in a system which uses cache memory is very small, it is reasonable to use the two techniques in combination.

SUMMARY AND FUTURE EFFORTS

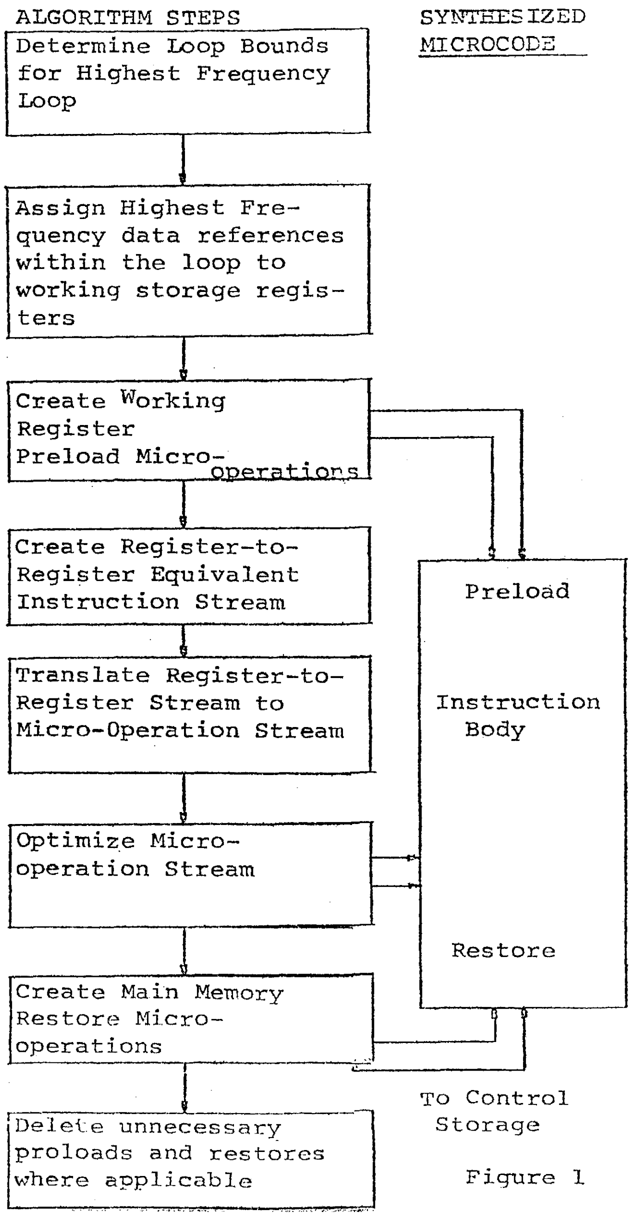
To determine the usefulness of these tuning methods, typical loop profiles in programs should be analyzed. This would include the determination of the "average" number of instructions per loop and the "average" number of times a single loop is executed in a given environment. If this "average" number of times through a small loop is larger than the crossover, then the synthesis procedure is a useful technique because of the definite throughput improvement.

REFERENCES

1. Abd-Alla, A.M. and Karlgaard, D.C., "Heuristic Synthesis of Microprogrammed Computer Architecture", IEEE, Transactions on Computers, Vol C-23, No. 8, August 1974, pp. 802-807

2. Meltzer, A.C., Private Communication, Chairman of the Electrical Engineering and Computer Science Department at George Washington University

3. Hewlett-Packard Company, Microprogramming Guide for the Hewlett-Packard Model 2100 Computer, Hewlett-Packard Company Document 5951-3028, February 1972



ALGORITHM STEPS
SYNTHESIZED MICROCODE
Determine Loop Bounds for Highest Frequency Loop
Assign Highest Frequency data references within the loop to working storage registers
Create Working Register Preload Micro-operations
Create Register-to-Register Equivalent Instruction Stream
Translate Register-to-Register Stream to Micro-Operation Stream
Optimize Micro-operation Stream
Create Main Memory Restore Micro-operations
Delete unnecessary proloads and restores where applicable
Preload
Instruction Body
Restore
To Control Storage

Figure 1

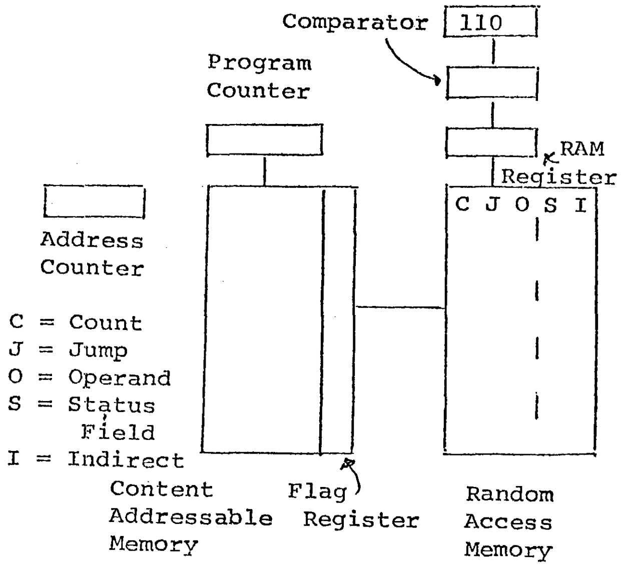


Figure 2a TRACE HARDWARE

RAM FIELD	FUNCTIONAL DESCRIPTION
C	If Flag set, set count to 1
J	Set to 1 if a jump instruction
O	Set to 1 if an operand
*	If comparison true adjust mapper to execute from synthesizer routine

FIGURE 2b Random Access Memory

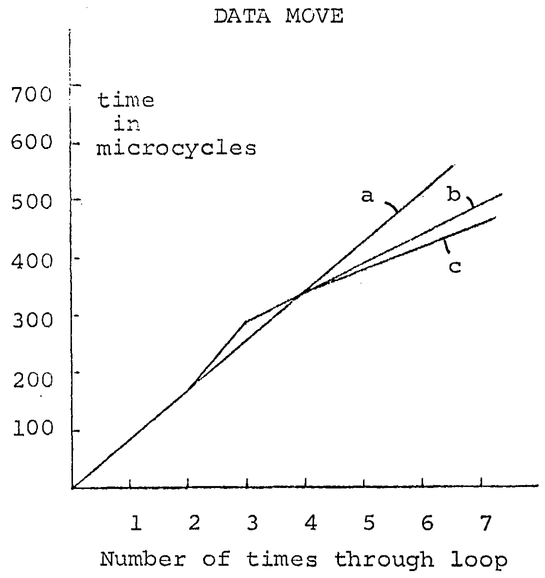


Figure 3

a-standard
b-pointer list
c-modified pointer list

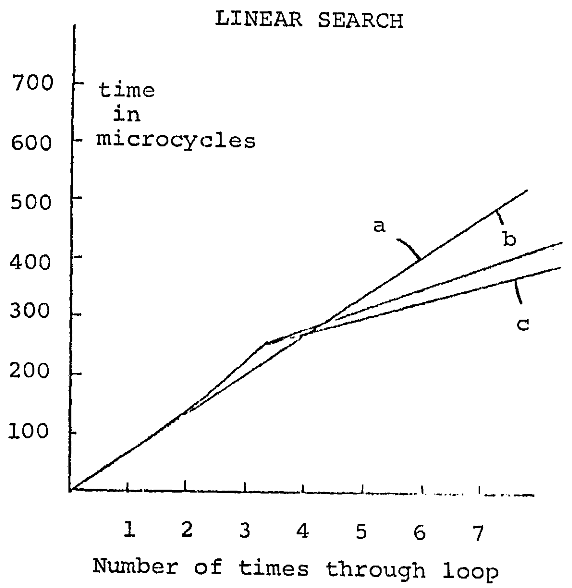


Figure 4

a-standard
b-pointer list
c-modified pointer list

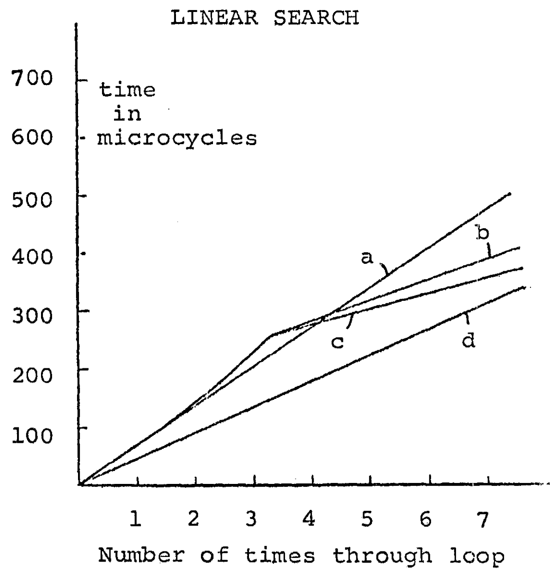


Figure 6

a-standard
b-pointer list
c-modified pointer list
d-cache

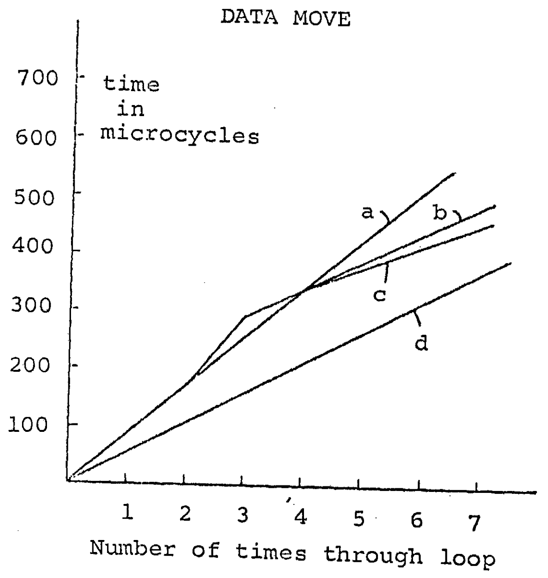


Figure 5

a-standard
b-pointer list
c-modified pointer list
d-cache

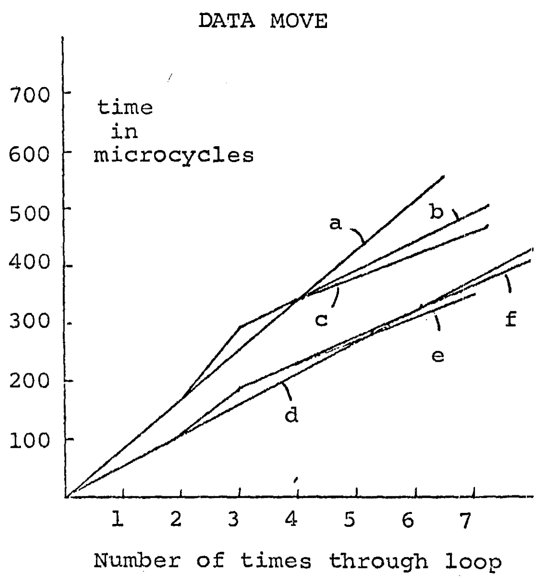


Figure 7

a-standard
b-pointer list
c-modified pointer list
d-cache
e-cache/pointer list
f-cache/modified pointer list

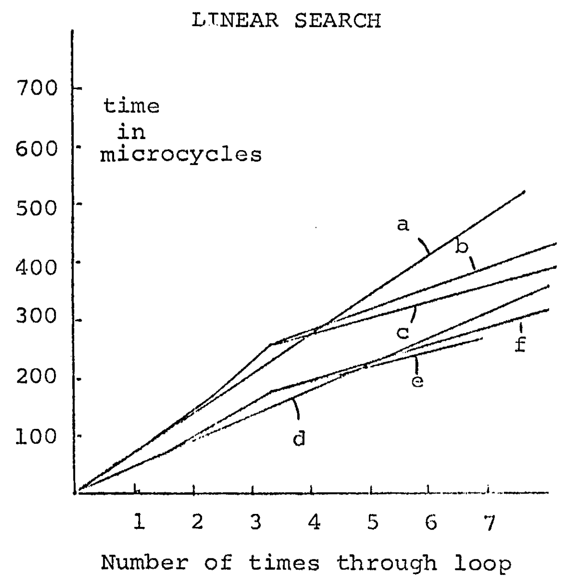


Figure 8

a-standard
b-pointer list
c-modified pointer list
d-cache
e-cache/pointer list
f-cache/modified pointer list

A CHARACTER-ORIENTED CONTEXT-ADDRESSED SEGMENT-SEQUENTIAL STORAGE

LEONARD D. HEALY
Computer Laboratory
U. S. Naval Training Equipment Center
Orlando, Florida 32813

The Context-Addressed Segment-Sequential Storage (CASSS) described in this paper provides a solution to the problem of data retrieval from a large, nonpreorganized file. It provides this capability entirely by hardware, eliminating the need for special data structuring solely for the purpose of reducing search time. The major features of the architecture of a character-oriented CASSS system are described, including the basic hardware configuration selected to implement such a system and the set of search instructions chosen to provide a wide variety of search operations useful in information retrieval. Of particular importance in this application is the method of quasi-parallel instruction execution, which allows a full string search of the entire data base in a single cycle of the sequential storage device used.

Introduction

Conventional information retrieval systems limit the user to a search capability restricted in either the flexibility of the search that can be conducted or in convenience in access to the system. The direct search--comparison of the contents of an entire file to a search criterion--provides the most comprehensive capability. However, the time required for transfer of the entire data base to core storage for search limits this technique to applications where a number of queries can be accumulated and processed in a single batch operation. Conventional on-line data retrieval systems obtain the shorter search time needed by augmenting or restructuring the file (e.g., indexes and inverted files). This limits the possible searches to those supported by the particular file arrangement chosen and compounds the problem of file maintenance.

Earlier special-purpose hardware systems for information retrieval relied upon keywords[1,2] or summary records.[3] More recently proposed systems[4,5,6] provide a sophisticated search capability that justifies the term context-addressing rather than content-addressing. Since the storage structure of these systems consists of sequentially accessed storage (e.g., disc tracks), the system is referred to as a Context-Addressed Segment-Sequential Storage (CASSS).

CASSS Organization

The data structure used in the CASSS system is a one-dimensional array of words called a file. From the software viewpoint, collections of words related in some way are stored together in a contiguous section of the file called a record. Figure 1 shows how a file of mixed-size records is mapped into a linear list and then divided into segments to match the storage structure. The function of search and retrieval operations is to examine all records to determine which ones satisfy a search criterion and to transfer those selected to the core storage of the host computer.

The basic architectural element in the system is the cell, consisting of a storage segment and its associated processor. Figure 2 shows a block diagram of the system. Each cellular processor, under command of the common controller, can perform a search of its entire storage in a single rotation of that storage. The controller is used to broadcast instructions to the cells and to provide the other functions needed to interface the CASSS system to its host computer.

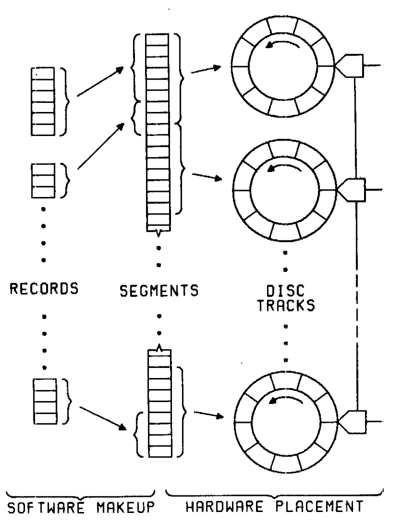


Figure 1. Storage of Records as Segments

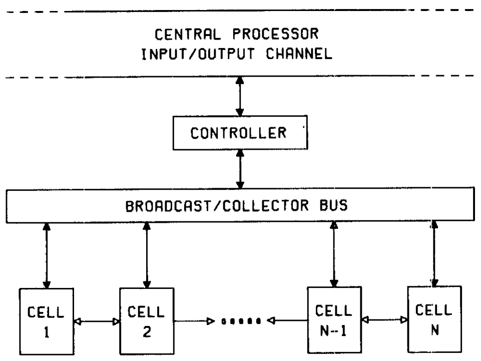


Figure 2. System Block Diagram

This organization offers advantages other than its rapid search capability. It can simplify the software support needed by eliminating the need for multi-level mappings both from high-level retrieval languages to machine language and from user-oriented view of data to machine-dependent storage structures.[7] The similarity between the procedure used to specify a query using a high-level language and the execution of the hardware instruction set in a CASSS system has been demonstrated.[8]

Character-Oriented CASSS

This paper describes a CASSS architecture for the handling of data stored in their natural form (the character strings familiar to the user) rather than encoded in some manner to enhance retrieval operations.

It allows retrieval of information from a file that is not necessarily preorganized by retrieval operations that are not necessarily planned before the file is created. This does not preclude special file organizations or search methods, but it does dictate a set of choices in the architecture that are different from systems that are limited to more organized data structures or search methods. The major features of this design are the organization of the cellular processor, the instruction set and a means of executing instructions in a quasi-parallel manner, and the I/O subsystem that provides autonomous retrieval of selected records.

Processor Design

The design of a cellular processor to perform string search operations is a compromise between the full string search capability of SNOBOL4 and what is both useful in searching large data bases and practical to implement by a sequential search. The design steps consist of selection of a data representation method and a hardware configuration to perform the search.

Data Representation

The choice of the character as the atom of information to be stored requires a suitable alphabet for data representation and to provide extra code combinations not in the data set for control functions needed in implementing the search algorithms. The Extended Binary-Coded-Decimal Interchange Code (EBCDIC) suggests itself for this purpose, but any character code with unassigned combinations could be used. The extra codes are needed to replace control functions that were handled in previous CASSS systems[4,5,6] by flag bits appended to each word. Flag bits were appropriate where the word being stored was relatively long and the flag bits occupied very little of the total storage. However, flag bits appended to each code in the 8-bit character representation of data result in a 12.5 per cent increase in the hardware required.

Character Marking

The most restrictive change caused by the use of control codes instead of flag bits is the marking of characters during search. The process of searching all records in storage and marking those that satisfy a query is performed in steps by marking individual stored characters that satisfy some criteria. Characters are marked by substituting a mark symbol, designated M_1, for the character code. The companion operation of unmarking a character previously marked is accomplished by reversing the substitution process. The comparand character for each search conducted is placed in a register for temporary storage at the end of the search cycle. During the next cycle, while the search and mark operation is being performed, the characters marked on the previous cycle are unmarked by substituting the character held in the temporary storage. This substitution method limits each individual search instruction executed to the marking of one specific character, but the saving in storage cost by avoiding an extra bit in each character far outweighs the effect of this restriction.

Processor Organization

Figure 3 shows a block diagram of the search portion of the cellular processor. Details of the search instructions, presented in the next section, do not alter the basic configuration. The dotted boxes in the block diagram represent connections to other parts of the cell, and the solid boxes represent the functional elements required to execute the search instructions. Table I indicates the purpose of each block. Instruction execution consists of circulating the data and rewriting them in the sequential storage. When marking or unmarking of a character is called for, the appropriate symbol is substituted into the recirculation loop.

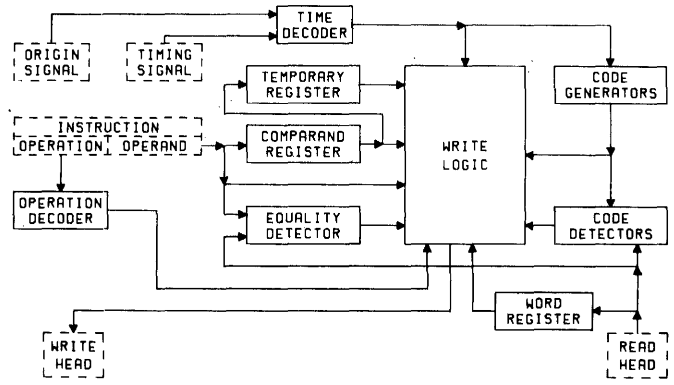


Figure 3. Instruction Execution Portion of Cellular Processor

Table I
Processor Functional Elements

Timing Signal. The timing signal provides a series of clock pulses for the entire processor. These clock pulses are synchronized with the data from external sources.

Origin Signal. The origin signal provides an output that is true ("1") only during the first data bit available from the sequential storage.

Read Head. The read head provides the serial data stream representing the information in the sequential storage.

Write Head. The write head enters the serial data stream provided at its input into the sequential storage. The positioning of the heads is such that the data entered into storage replace the data currently available in the word register.

Instruction Input. The instruction to be executed consists of an operation code that specifies what is to be done and an operand that indicates the character being searched for.

Time Decoder. The time decoder provides eight separate pulses, defining the eight bit intervals during the serial processing of a single character.

Code Generators. The code generators produce a serial 8-bit code corresponding to each special symbol that must be recognized by the processor.

Code Detectors. Code detectors provide recognition of each of the special codes.

Word Register. The word register provides a one-character delay to allow the character being processed to be examined before it must be rewritten into storage.

Comparand Register. The comparand register is used to hold the character to be searched for in sequential storage. It receives its input from the instruction operand during the first character time and recirculates its contents during other character intervals.

Temporary Register. The temporary register is used to hold the comparand from the previous instruction. It receives its input from the comparand register during the first character time and recirculates its contents during other character intervals.

Equality Detector. The equality detector compares the input to the comparand register to the input to the word register.

Write Logic. The write logic connects the appropriate code generators, the temporary register, or the word register to the write head.

Control Logic. The control logic consists of an instruction decoder and the logic necessary to provide control signals to the other elements in the cellular processor.

Search Instructions

The format of the search instructions used in the character-oriented CASSS system is a character string. The first character denotes the instruction code, and succeeding characters specify the operand.

The elementary search instructions have a single-character operand; the more complex instructions have a variable-length operand. Table II gives a description of the search instructions selected for search of non-preorganized data bases. A more formal instruction definition in APL notation is available in a report.[9]

Table II
Description of Search Instructions

The following instructions mark each stored character that satisfies the conditions given and unmark any previously marked characters that do not satisfy these conditions.

● Mark Character MC C	Matches the comparand C.
● String Search SS C	Matches the comparand C and follows immediately a character marked by the previous search.
● Ordered Search OS C	Matches the comparand C and follows a character in the same record marked by the previous search.
● Ordered Field Search OF Z C	Matches the comparand C and follows a character in the same record marked by the previous search with no intervening end-of-field code Z.
● Move Mark MM D_a D_b C	Matches the comparand C and follows a character in the same record marked by the previous search by exactly D characters (where D is specified by D_a D_b treated as a binary number).
● Inequality Search IS Z C_1 C_2 ... C_N Z	Matches the delimiter specified by Z and follows a numeric string that: (1) follows immediately a character marked by the previous search, and (2) satisfies the comparison $(<, \leq, =, \neq, \geq, >)$

The following instructions add the quantity C_1 C_2 ... C_N treated as a binary to the contents of the first threshold accumulator if the conditions shown are met. They unconditionally unmark all previously marked characters.

● Threshold Addition TA N C_1 C_2 ... C_N	The record was marked by the previous search.
● Threshold Prime TA N C_1 C_2 ... C_N	The record was not marked by the previous search.

The following instructions compare the contents of the threshold accumulator designated to the comparand C_1 C_2 ... C_N according to the comparison option $(<, \leq, =, \neq, \geq, >)$ specified and perform the additional functions indicated.

● Threshold Test TT N C_1 C_2... C_N	If the comparison between the comparand and the first threshold accumulator is successful, increment the second threshold accumulator by one. Reset the first accumulator to zero.
● Threshold Compare TC N C_1 C_2 ... C_N	If the comparison between the comparand and the second threshold accumulator is successful, mark the $\underline{B}_1$ symbol following the second accumulator. Reset the second accumulator to zero.

Elementary string searches are performed using the first three instructions in the list. For example, the search for the string \$ABC\$DE\$, where \$ represents an arbitrary string including the null string, is done by the instruction sequence: MC A, SS B, SS C, OS D, SS E. Execution of this program leaves the E in each string in storage that matches the input string marked.

The next three instructions in the list are designed for use where the user knows the data format. These instructions allow the programmer to locate fields within a record that are identified by a header or are a fixed number of characters from some other field that can be located. The Move Mark (MM) and Inequality Search (IS) instructions illustrate the effect of the limitation imposed by the substitution method of marking characters. It is not possible to move the mark ahead by a fixed distance--the marking must be limited to a specific character. Similarly, a numeric string to be marked if it meets some arithmetic comparison $(<, \leq, =, \neq, \geq, >)$ must be followed by a known delimiter, allowing the marking to be restricted to a specific character. Any non-numeric symbol in the data alphabet may be used for the delimiter. The characters within the comparand field for the IS instruction must be numbers.

The threshold instructions provide the capability to evaluate retrieval criteria based upon some Boolean or threshold function of individual string searches. This feature requires the addition of two accumulators, designated A_1 and A_2, respectively, at the end of each record. A control symbol, designated $\underline{B}_1$, is placed before, after, and between these accumulators to allow the processor logic to locate them. The first accumulator, with its associated threshold instructions, allows evaluation of any linearly separable threshold function of individual string searches. The second accumulator, with its associated instructions, allows evaluation of any m-out-out-n function of the functions evaluated using the first accumulator. This capability allows efficient evaluation of a wide range of search functions of interest in information retrieval.

Search Example

Use of the instructions is best illustrated by an example. Assume a file made up of personnel records, with the arrangement of the first nine fields as shown in Table III. Let the problem be retrieval of all records that satisfy the following criterion: ((Name: Julia Smith) OR (Maiden Name: Julia Jones) OR ((Name: ..... Smith) AND (City: Gainesville, Florida)) OR ((Maiden Name: ..... Jones) AND (City: .... Florida))) AND ((No. Dependents: $\leq$ 4) OR (Salary: >150.00)).

Table III
Field Allocation within Each Record

Delimiter	Field Use	Field Length
$\underline{D}_1$	Social Security No.	9 characters
	Name	Variable
/1/	Street Address	Variable
/2/	City, State	Variable
/3/	Zip Code	5 characters
	Date of Birth	6 characters
	Maiden Name	Variable
/4/	Salary	Variable
/5/	No. Dependents	Variable
/6/	....	....

Let a set of X_i's, where $1 \leq i \leq 6$, represent the truth value of the individual string searches. The search function is $(X_1 \vee X_2 \vee X_3 \vee X_4) \wedge (X_5 \vee X_6)$. It can be implemented by the appropriate set of string searches followed by threshold operations. Table IV shows how the threshold instructions are used in the search program. Execution of this program causes all records that satisfy the query to be marked in the delimiter following the second threshold accumulator.

Table V shows a section of the complete program. It illustrates the method of locating the field delimiters and moving the mark over fixed-length fields to reach the field to be searched.

Quasi-Parallel Instruction Execution

Since each instruction performed by the CASSS system requires a search of each record in storage, the execution time for a single search instruction is at least the time required to traverse all records.

Table IV
Use of Threshold Instructions

	INST.	COMPARAND	REMARKS
			..... (String Search for X_1) .....
14.	TA	1,1	Add one to first threshold accumulator in those records where search is succsssful.
			..... (String Search for X_2) .....
30.	TA	1,1	Add one to first threshold accumulator in those records where search is successful.
			..... (String Search for X_3) .....
58.	TA	1,1	Add one to first threshold accumulator in those records where search is successful.
			..... (String Search for X_4) .....
77.	TA	1,1	Add one to first threshold accumulator in those records where search is successful.
78.	TT	1,1	Add one to second threshold accumulator in those records where the first accumulator holds at least one.
			..... (String Search for X_5) .....
83.	TA	1,1	Add one to first threshold accumulator in those records where search is successful.
			..... (String Search for X_6) .....
88.	TA	1,1	Add one to first threshold accumulator in those records where search is successful.
89.	TT	1,1	Add one to second threshold accumulator in those records where the first accumulator holds at least one.
90.	TC(IT)	1,2	Transfer to core storage the identifier for each record for which the second threshold accumulator holds at least two.

Table V
Search Program Example

	INST.	COMPARAND	REMARKS
1.	MC	$\underline{D}_1$	Mark start of each record.
2.	MM	10,0,"J"	Locate and search first character in name field.
3.	SS	"U"	Continue name search.
4.	SS	"L"	
5.	SS	"I"	
6.	SS	"A"	
7.	SS	" "	
8.	SS	"S"	
9.	SS	"M"	
10.	SS	"I"	
11.	SS	"T"	
12.	SS	"H"	
13.	SS	"/"	Complete name search.
14.	TA	1,1	Add one to first threshold accumulator in those records where search is successful.
15.	MC	"/"	Begin second search sequence.
16.	SS	"3"	
17.	SS	"/"	Mark beginning of zip code field.
18.	MM	12,0,"J"	Locate and search first character in maiden name field.
19.	SS	"U"	Continue maiden name search.
20.	SS	"L"	
21.	SS	"I"	
.	.	.	
.	.	.	
.	.	.	

However, it is possible in most circumstances to execute several search instructions during one sequential storage cycle. Though this adds to the cost of the cellular processor, it results in an "effective" implementation. The quasi-parallel execution of k instructions results in a k-fold decrease in search time with only a small increase in processor hardware. Without this feature, the character-oriented CASSS is not practical. A search rate limit of one character per sequential storage cycle is too slow to be useful. For example, the search program for the sample problem above would take almost 100 storage cycles it if were executed one instruction per cycle. Quasi-parallel instruction execution reduces the time required to six storage cycles.

The string matching that is characteristic of a typical search program makes quasi-parallel execution possible. Each string search program begins with an MC instruction and executes a sequence of comparison operations. The search ends by marking the character at the end of the desired string or by using the result of the string search to increment the threshold accumulator at the end of the record. An example of such a program is the task of incrementing the first threshold accumulator in those records that contain the string $AB$CD$.

The program to perform this search is: MC A, SS B, OS C, SS D, TA 1,1. Figure 4 shows the diagram of a sequential machine in which the states indicate the instruction to be executed next. At each step, the machine considers only three comparands: (1) symbol $\underline{B}_1$ to determine whether the TA instruction must be executed, (2) the comparand of the instruction being executed, and (3) the comparand for the previous MC or OS instruction. In other words, the only possible options during the search of the data are: (1) to end the string search because the end of the record has been reached, (2) to continue the search of a contiguous string of characters satisfying the search, or (3) to reinitiate the string search from the beginning of the comparand string to be matched.

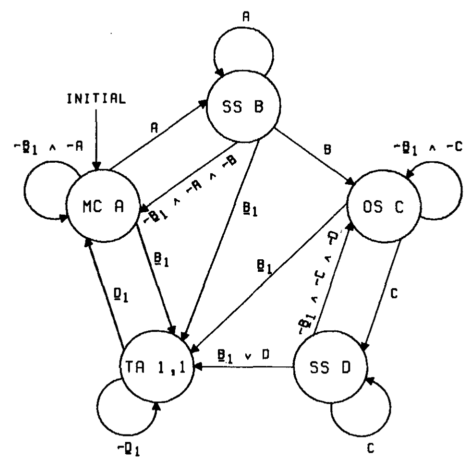


Figure 4. Instruction Selection

There are two limitations upon the quasi-parallel execution of instructions. The first is that parts of two different string searches cannot be executed together. This restriction is made part of the logic of the cellular processor. The second restriction is data dependent and cannot be solved by hardware. If a string used as comparand is embedded in a stored string such that its occurrences overlap, only the first of these stored strings is marked by the string search. For example, the string ABABAB contains the string ABAB twice, but the search for the string ABAB using quasi-parallel execution of instructions will detect only the first occurrence of that string.

Fortunately, the problem of embedded strings does not occur in the major applications of the CASSS system. In those cases where it is a problem it can be corrected by programming. A dummy instruction is added for this purpose. Its only function is to signal the hardware to perform instructions before and after the dummy instruction on separate storage cycle. In the case of the string search ABAB, a dummy instruction inserted in the middle of the search instructions causes the search for the first AB to be separated from the search for the second AB. The result is that the string ABABAB in storage is marked after each occurrence of the string ABAB.

Correct results in searching for embedded strings can be guaranteed by breaking the string to be searched into substrings such that no substring repeats its initial character. This algorithm is not implemented in hardware because it is not an "effective" solution to the problem for all applications. For example, the search program for the sample problem in the previous section can be performed correctly in six storage cycles (one cycle for each separate string search to be evaluated), but implementation of the above algorithm divides the search so that it takes 15 storage cycles. The choice of how to break the strings is left to the programmer, because efficient programming depends upon a knowledge of the data base. For applications other than text editing, the programmer can probably do a more efficient search than that obtained by direct application of the algorithm that guarantees correct search of any possible storage contents.

Data Transfers

The character-oriented CASSS system uses the I/O techniques common to third-generation computers. The interconnection to the host computer consists of a low-speed I/O channel for transfer of search instructions and a direct port to core storage for high-speed data transfer. The I/O subsystem provides the capability to read, replace, or modify either entire records or portions of records. Data transfers, once initiated by a command from the host computer to the CASSS system, are executed without further intervention by either the computer or the search execution portion of the CASSS system.

Marking Records for Transfer

The semi-autonomous transfer of data requires the addition of several character positions in each record immediately after the record delimiter code. The first two positions hold a binary number representing the record length, and the next one or more character positions provide one-bit control flags for each high-speed data path provided. Once a record is marked, the transfer is performed and the flag is reset by the I/O controller. The transfer paths share the common entry port to core storage and thus do not operate in parallel. However, each transfer path has the characteristics of a channel and will be referred to as such. The number of channels, and thus the number of control positions in the record header, is made optional to fit the particular use.

The major problem in record retrieval is moving the mark that is entered in the record contents by the search operation to the beginning of the record. This is a backward-marking operation that cannot be done by sequential storage. The simplest way to accomplish it is to provide a bit-per-record random access storage, but this is inefficient use of storage in a system that holds a large number of records. Instead, this design takes advantage of a characteristic of the search problem--the number of records to be marked is much less than the number of records stored. Therefore, it is less costly to store complete identifiers for the few records than to store one bit for each record.

Three alternatives for providing this storage suggest themselves: (1) adding several words of storage within each cell, (2) providing a single storage shared by all cells, and (3) using an assigned buffer area in the central computer's core storage. The last method is selected because of its flexibility. The number of records retrieved by a search is likely to vary widely for different classes of retrieval problems. The first two methods require that storage hardware be sufficient to satisfy the type of search that retrieves the most records. A buffer area in core storage can be altered in size according to the class of problems being solved.

A record can be identified by either its position within the file (e.g., the 29th record) or by its physical location (e.g., the 10th record in the 3rd cell). The former method is the one selected, because it uses a file characteristic rather than a storage characteristic. The record number is useful in references between records, even in a dynamic situation, whereas a physical location is not.

Use of the record number imposes the burden of translating between record number and physical location upon the cellular processor. Three registers are added to each cellular processor for this purpose. These registers hold: (1) the first record in the cell, (2) the last record in the cell, and (3) the record most recently processed by the cell. The last register is used for encoding record location to record number, and the firt two allow the cell to recognize those identifiers that refer to its contents.

The marking of records for transfer is done in two steps. The first step is performed by the last search instruction in a sequence. The Identifier Transfer (IT) option is added to each of the search instructions to cause the identifier for each record marked by the instruction to be transferred to core storage. When this option is selected, the normal I/O transfers are interrupted while the instruction is being executed. As each cellular processor performs its search operation, it sends the identifier for each record it marks to the I/O controller.

The transfer of record identifiers is much faster than the transfer of the records themselves for two reasons. First, the amount of data to be transferred is much less, so that fewer overlaps occur. Secondly, each cell provides temporary storage of the identifier it is trying to transmit for the length of time it takes to process the next record. Only in the case of sustained occurrence of overlaps does the identifier transfer take more than one cycle of the sequential storage.

Record Retrieval

The Input-Output (IO) instruction selects a channel for transfer of records indicated by the references in core storage and initiates the transfer by marking the selected records at their begining.

Only the part of the operation concerned with retrieving the identifiers from core storage and routing them to the proper cells is considered here. The other aspects of the IO instruction are essentially like those used with a conventional disc controller.

The transfer of record identifiers from core storage begins by temporarily halting any normal I/O transfer that might be in progress. The identifier block in core storage is then broadcast to all cells, starting with the first word in the block. The broadcasting is done one word at a time, with the controller waiting for a reply before sending the next word in the sequence. The cell that holds the indicated record (readily determined since each cell has a register to hold the upper and lower record numbers stored in its segment) accepts the information and sends a reply signal. The cell uses this information to mark the record indicated in the appropriate I/O flag bit the next time it becomes available from sequential storage. The execution of the IO instruction is complete when the channel for transfer is established and all records to be transferred are marked in the flag bit for this channel in the record header. The transfer itself then proceeds in parallel with other operations being performed by the CASSS system.

Summary

The CASSS architecture that has been developed provides an effective solution to the problem of retrieval of information from large files that are not necessarily preorganized. It allows implementation of sophisticated search strategies based upon file content--searches that a conventional system can perform only by time-consuming direct search--in a time comparable to that obtained in a conventional system only by using a highly structured file organization. It simplifies the problem of file maintenance by removing the need for highly structured file organization solely to enhance retrieval.

The ability to search unformatted data is obtained by storing data as a character string and searching it by string matching operations. The search is performed rapidly by dividing the total data base into a number of segments that are searched in parallel. The implementation of complex searches is made possible by an instruction set tailored to perform searches for character strings and by a scheme for quasi-parallel execution of search instructions. This latter feature is the most important attribute of the architecture. It makes the string search technique feasible by searching for an entire string rather than a single character at a time.

The input-output subsystem provided is an adaptation of the method used on third-generation computers. This gives the same magnitude of improvement over previously proposed search and retrieval systems that the change to autonomous input-output transfers gives over second-generation computers. The use of the core storage of the host computer as the random access storage needed in marking records for output provides a means of handling cross-references and multi-linked data structures.

Acknowledgements

The author wishes to acknowledge the advice, help, and assistance in conducting this research given him by Drs. Keith L. Doty, Gerald J. Lipovski, and Stanley Y.W. Su of the Electrical Engineering Department of the University of Florida.

References

1. R. H. Fuller, R. M. Bird and R. M. Worthy, "Study of Associative Processing Techniques," Rome Air Development Center, Griffiss AFB, N. Y., TR No. RADC-TR-65-210, Sept. 1965.

2. J. L. Parker, A Logic per Track Information Retrieval System, Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, Illinois, Feb. 1971.

3. G. F. Coulouris, J. M. Evans and R. W. Mitchell, "Toward Content-Addressing in Data Bases," Comput.J., vol.15, pp. 95-98, May 1972.

4. B. Parhami, "A Highly Parallel Computing System for Information Retrieval," in 1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.,vol. 41 pt. 2. Montvale, N. J.: AFIPS Press, 1972, pp. 681-690.

5. L. D. Healy, G. J. Lipovski and K. L. Doty, "The Architecture of a Context Addressed Segment-Sequential Storage," in 1972 Fall Joint Comput. Conf., AFIPS Conf. Proc., vol. 41 pt.2. Montvale, N. J.: AFIPS Press, 1972, pp. 691-701.

6. G. P. Copeland, Jr., G. J. Lipovski and S. Y. W. Su, "The Architecture of CASSM: A Cellular System for Non-numeric Processing," Proc. First Annual Symposium on Computer Architecture, Gainesville, Fla, Dec. 1973, pp. 121-128.

7. S. Y. W. Su, G. P. Copeland, Jr. and G. J. Lipovski, "Retrieval Operatións and Data Representation in a Context-addressed Disc System," Proc. ACM SIGPLAN/SIGIR Interface Meeting on Programming Languages and Information Retrieval, Gaithersburg, Md, Nov. 1973.

8. G. P. Copeland, Jr. and S. Y. W. Su, "A High Level Data Sublanguage for a Context-addressed Segment-sequential Memory," submitted for publication.

9. L. D. Healy, The Architecture of a Context-Addressed Segment-Sequential Storage, Ph.D. dissertation, Department of Electrical Engineering, University of Florida, Gainesville, Fla, June 1974.

SOME IMPLEMENTATIONS OF SEGMENT SEQUENTIAL FUNCTIONS[†]

J.A. Bush G.J. Lipovski
S.Y.W. Su J.K. Watson
S.J. Ackerman
University of Florida
Gainesville, Florida 32611

Abstract

Since conventional computers are straining to handle the increased size and sophistication of non-numeric processing (data management, information retrieval, artificial intelligence), a new class of non-numeric architectures is evolving. The segment sequential architecture is one of these. Further development of this architecture requires new techniques for multiple cell operation and intercell communication to handle control and search operations. This paper describes such techniques for instruction fetching, operand recall, string, set and tree context searching, and pointer transfer. It is expected that combinations of these techniques will appear in future architectures that are needed for non-numeric processing.

1. Introduction

Recognizing a need for non-numeric processors to support large data bases, several investigators have been striving to develop suitable architectures: Fuller, et al., 1965,[2] Parham, 1972,[6] and Healy et al., 1972.[3] Several systems have been proposed,[1,3,4,5] and two of them - CASSM at the University of Florida and RAP at the University of Toronto[8] - are presently being developed. These systems all make use of sequential memory organized as segments associated with simple processors.

In the segment sequential architecture,[3] fixed length words are organized into variable length records, which are packed into a single file. See Fig. 1. The programmer can consider this data to be in a single file. However, the file is broken into equal length segments of words, and each segment is stored on a separate disc track, CCD shift register, or magnetic bubble memory. In the following discussion, the terminology appropriate to discs will be used. A "microprocessor" and associated segment of memory are here called a cell. A one-dimensional array of cells can search the file in parallel in one cycle of the disc, as each cell operates on the first word in its segment, then the next word in the segment, and so on. See Fig. 2.

The segment sequential architecture is claimed to be particularly suitable for non-numerical processing because the software can ignore the location (unit, surface, track, sector, etc.) of words on the disc, and can process the information where it is on the disc. In addition the hardware can simultaneously process words on different disc tracks so that arbitrarily large data bases can be searched in the same time it takes to search one segment. These two features make the architecture very attractive for data management systems, which account for a very large share of day-to-day work done on computers. For a discussion of further advantages of such disc systems, see [9,10].

It should be noted that a segment can contain several records, and a record can span several segments. The latter problem requires some intercell communication so that records can be searched as a whole, for example to find all sets that contain words a and b, even if a and b are on different segments.

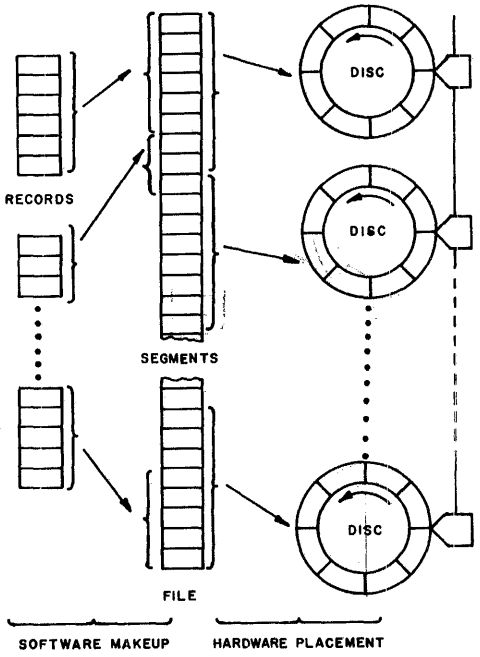


Figure 1—Storage of records as segments

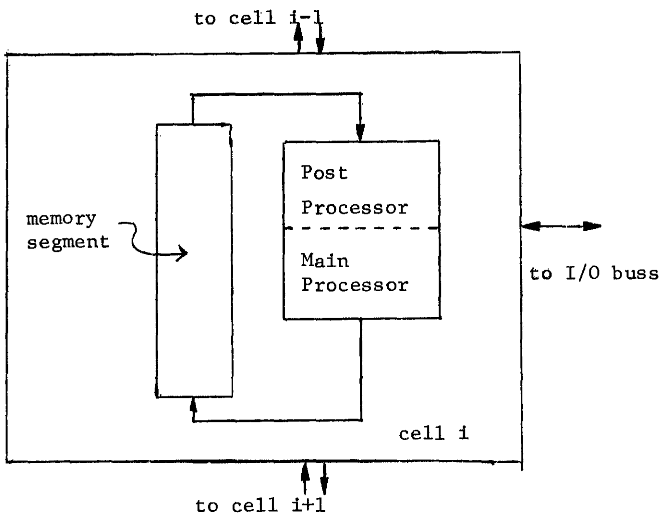


Figure 2. Cell processor and Memory Segment

We suggest that the non-numeric processor has evolved at this point to about where the numeric processor had evolved in the early forties. The development of techniques for random access memories in 1943 had to predate the development of the von Neumann architecture. This paper presents a number of techniques for intercell communication. It is intended to provide the kind of background that the random access memory provided to the von Neumann architecture. Based on some of these

[†]This paper was supported by NSF Grant GJ-43225.

techniques and others yet to be developed, a general purpose architecture for non-numeric processing should emerge, to be widely used in data management systems.

In this paper, implementations for five techniques: instruction fetching, operand recall, string and tree searching and pointer transfer are presented for multiple segment systems. Further techniques for input and output in multiple segment systems are presented by De Martinez.[1] Here, each technique is presented as independently as possible from the others so that a subset of the techniques can be used in any future architecture. However, they are so ordered that simple concepts presented first will help explain more subtle concepts given later. Generally, the techniques are first presented for a single segment where they are easy to describe. Then the more complex multiple segment case is considered.

For purposes of discussion, the following conventions and definitions are used in the rest of this paper. Fixed length words are divided into data and tag[7] fields. The tag field is used not only to identify word types as data, instructions, operands or erased words, but also tag bits are used to mark successful searches, and so on. Data words are organized into records, whose first word is especially tagged to be a delimiter word. These records are organized, later in the paper, as nodes in a tree. The file is divided into segments, as depicted in Fig. 1, such that the topmost segment contains the first record(s). As Fig. 2 shows, the cell that contains each segment communicates to its next upper and next lower neighbor, and to a common buss for I/O. The ordering of records is thus retained by the order of segments and the order within each segment. The searching and rewriting of words from top to bottom on the segment is called the scan, and the time after the bottom word is processed, before the top word is again processed, is called the gap. A cycle is a gap plus a scan.

Generally, an instruction, such as "Search for A" is conducted in one or two cycles. The first cycle is executed in the main processor (Fig. 2) while the second cycle is executed in the post-processor if necessary. These are concurrent so that the post-processor completes execution of instruction i while the main processor executes instruction i+1. This permits the processor effectively to execute one instruction per cycle. The post-processor operation is logically complete just before the operation done by the main processor. (In practical systems using garbage collection, delays are necessary between post-processor and main processor.)

2. Fetch-Cycle-Equivalent Operations

Analogous to the fetch cycle operations of fetching an instruction and recalling an operand in a standard computer, the segment sequential architecture utilizes techniques described below. During one cycle of the disk, one instruction in the common register is executed on the data in all tracks simultaneously. A second word may or may not be required as an operand in the O register for the instruction. The instruction and operand, if needed for the current cycle, are fetched in the previous cycle.

2.1 Instruction Fetch

Some words on the disc are especially tagged to be instructions, in distinction to data or operands, and some of these are further tagged from time to time as active. It is possible to append active instruction words onto the bottom of the disc from an external computer, or to activate instructions already on the disc as a result of searching data on the disc. Among all active instructions, the topmost is fetched to be the instruction for the next cycle, and is deactivated. By other means, several consecutive instruction words can be activated during one cycle, to be fetched one at a time later. More generally, active instructions scattered throughout the disc will be fetched one at a time. Note that this is an instance of a general case of first-in-first-out order (FIFO order) because words can be appended at the bottom and used or removed from the top of the memory (disc).

2.1.1 Single Track. For single-track case, the problem is to take instructions from the track in FIFO order. This is done by saving the first active instruction word encountered in a scan and then marking that word as inactive so the next active word will be taken on the next scan.

This technique requires a buffer register F.B and a flip-flop F.FULL. F.FULL is cleared at the beginning of each cycle. As long as F.FULL is clear, words from the disc are loaded into F.B. Each word that enters the main processor is checked to see if its tag indicates an active instruction and F.FULL is clear. If they are, the instruction is marked inactive and F.FULL is set. Thereafter, this prevents any succeeding words from being chosen. At the end of the scan the copy of the instruction word retained in F.B is sent to the I-Register, where it will be decoded as the instruction for the next scan. During that cycle, another instruction is to be fetched. F.FULL is cleared during the gap time, to prepare for the next scan.

2.1.2 Multiple Track. The multiple track system must fetch the topmost active instruction in the entire disc. Consider a two-track disc in which the upper track (higher priority) has an active instruction A near its bottom and the lower order track has an active instruction B near its top. Word A should be fetched and deactivated even though the two processing elements will meet B first as they scan the tracks. Means to save at least word A, and to deactivate only word A are required. The proposed strategy is to save the topmost instruction within each segment as before, but not to deactivate it. The location within the segment of the "fetched" word is also saved as the word is "fetched" on each track separately. During the gap, a priority circuit determines the top cell that fetched an instruction. That instruction is sent to all cells, and in the following cycle, that cell deactivates its "fetched" instruction.

This technique requires F.B, F.FULL as before, a word counter WCT, two word-count-save buffers F.WCT.B and F.WCT, in each cell and a conventional hardware priority circuit (like an I/O priority circuit) between cells. (See Fig. 3.) In each cell, F.B and WCT are initially cleared. WCT is incremented as each word is scanned. The topmost instruction in each cell is found as in the single-track case by saving the first active instruction word in F.B and the word count WCT in F.WCT.B, then setting F.FULL. During the gap, F.FULL is used as an indication of 'instruction found' for input to a priority circuit which decides which cell with F.FULL set is the highest. This topmost cell sends its instruction from its F.B to the I-Registers in all cells and its F.WCT.B to its own Post-Processor's F.WCT (so that the instruction on that track will be deactivated in the next cycle). Other cells load a very large number (11...1 >> max(WCT)) into F.WCT (so that no work will be deactivated). Deactivation will be done when F.WCT is equal to WCT in the Post-Processor. All lower order cells which found instructions are prevented from transferring their F.B to the I register in all cells, and their F.WCT will never equal WCT, so those lower instructions remain active for the next cycle. Finally, all F.FULL's are cleared, and the segment is ready for the next scan.

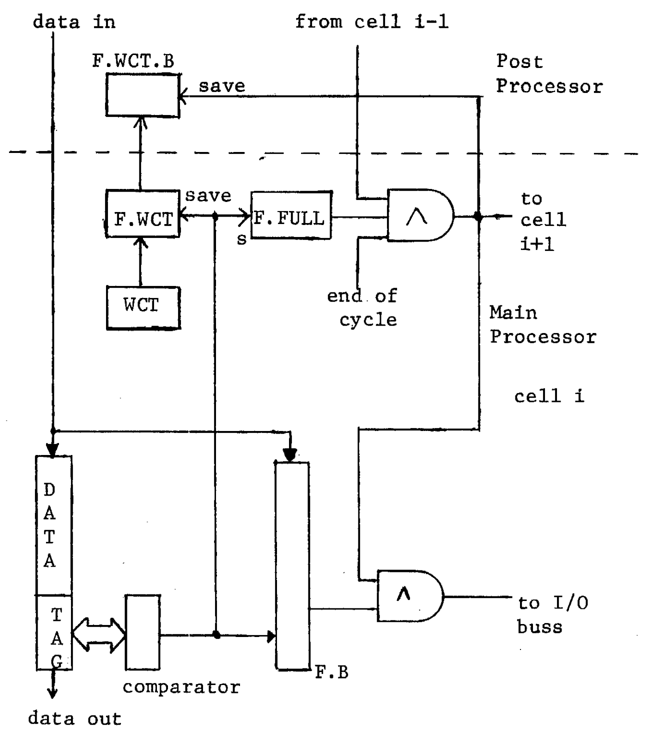


Figure 3 -- Instruction Fetch Logic

2.2 Operand Recall

A collection of words on the disc are especially tagged to be a stack of operands in distinction to data or instruction words. New operands are generally pushed onto the stack on the bottom. If an operand is required by an instruction, one word is popped from either the top (FIFO order) or bottom (LIFO order) of this stack. Popping an operand from the top of the stack can be similar to fetching an instruction. Identical but separate hardware is used, except that the operand word is actually erased from the disc rather than merely deactivated. However, popping the bottom word on a stack requires different techniques. In the following section, techniques are presented for popping either the top or bottom word of a stack, using the same hardware.

2.2.1 Single Track. While the instruction is being fetched, it is not yet known whether an operand will be needed. Nevertheless, an operand is saved but not erased. When the instruction is decoded and an operand is indeed required, the last-found operand is actually erased from the disc. If the operand is recalled from the top, one must save only the first stack word. If the operand is recalled from the bottom of the stack, one keeps saving all stack words. In the end, the bottom stack word is the word that was last saved, and it will be used as the operand in the next cycle if it is needed.

These techniques require the word counter WCT and save registers O.B, O.WCT and O.WCT.B, as well as flip-flop O.FULL. Initially, O.FULL is cleared, and is set when a stack word is found. Similar to multiple track instruction fetching, when an operand word is found it is loaded into O.B and WCT is put in O.WCT.B unconditionally if LIFO order is used, or conditionally on O.FULL being clear if FIFO order is used. The recalled operand is now in O.B and its location is in O.WCT.B. During the gap if the operand is found to be needed in the next instruction, it is transferred to the operand register O, while O.WCT.B is transferred to O.WCT in the Post-Processor, otherwise a very large number is put in O.WCT. In the next cycle, the word image on disc is erased by the Post-Processor if O.WCT equals WCT.

2.2.2 Multiple Track. By simple extension to the previous techniques, the multiple track case can be handled. A priority network identical to that used by the instruction fetch is used to locate the topmost cell with an operand for the FIFO case. A mirror-image priority network is used to locate the bottom-most cell with an operand for the LIFO case. If an operand is required, the operand in O.B in the cell chosen by the priority network is loaded into O in each cell, and O.WCT.B is loaded into O.WCT. In all other cells, or if no operand is required, a very large number is loaded into O.WCT.

3. Content and String-Context Searches

A search instruction may look for a word which compares with the operand of the instruction. This is the basic content addressing mode of most associative memories. As a result of such a search, all words that satisfy the search are marked by setting a tag bit, and other words are unmarked by clearing this tag bit. It would be possible later to find the marked words and output them or rewrite part of them.

Alternatively, this mark can be passed, like a token in a relay race, from one word to lower words to find a string of consecutive words. Suppose one wishes to rewrite any word on the disc that follows a pair of words, "a" and "b". One first searches for every occurrence of an "a" in a word, marking that word. In the next instruction, one marks only all words that have a "b", and that are preceded on the disc by a previously marked word. In the final instruction, one rewrites all words that are preceded by a previously marked word. Note that searching is conducted in the context of a data structure, a string of words. In one variation of this operation (ordered set search), the string on the disc can have extra words, i.e., the word sequence axyzb would satisfy this search. In another variation (inverse ordered set search) the string on the disk can have fewer words, i.e., the word sequence "a" alone would satisfy the search. Various combinations of these string searches can be used to recognize some misspelled words, recognize patters, and so on. These operations are easily carried out in a segment sequential architecture as we describe below.

3.1 Content Searches

As indicated previously in Fig. 1, all words are fixed length and stored serially. Under no condition is a word stored partly on one track and partly on another. Numbers are stored least significant bit first. The tag bit M that identifies marked words can be the last bit of each word on the disc.

An equality comparison can be made using a JK flip-flop S.T, which is initially set for each word. As each word is scanned serially, the corresponding operand bits and data bits from the disc are compared through an exclusive-OR gate to the K input, to clear S.T if they differ. At the end of the word, S.T is loaded into M and is then set for the next word. An arithmetic inequality test can be done with a serial subtracter. Bits can be ignored (masked out) by disabling the clock to S.T or the carry flip-flop of the subtractor when such bits enter the comparator or by ORing mask bits into both inputs to the comparator. Considerable flexibility is obtained when the word is divided into fields (e.g., tag, name, value fields) and separate comparator checks each field. At the end of the word, combinational logic is used to load the tag bit M, based on the results of the various comparators, and on the instruction bits that select various search options. Some of the search options are discussed below.

3.2 Search Next

A search next is a search for a word satisfying some basic criteria which immediately follows a word satisfying some previous criteria (indicated by the mark on that word). The string search indicated in Section 3 is performed by first searching for and marking (setting M-Bit) of all words containing "a" as described in Section 3.1; then performing, on the next cycle, a search for all words containing "b" which immediately follow a marked word (i.e., one with "a") as we describe below.

3.2.1 Single Track. Herein, a dual rank flip-flop S.LAST is used to indicate the state of the mark bit of the previous word encountered. The data word itself is searched as in 3.1. S.LAST is cleared at the start of a cycle. At the end of each word, S.LAST is cleared if the comparator outputs a zero at the end of the word; then this value is exchanged with the tag bit M of that word.

3.2.2 Multiple Tracks. The only difference between this and the single track case involves the value of S.LAST at the beginning of each cycle. Previously, the topmost word of the track had no predecessor, but now it is preceded in the segment by the last work on the track above this one. Clearly, only the topmost cell will initialize S.LAST to zero. All others will initialize their S.LAST to the final value of tag bit M, which has been put in S.LAST, in the next lower cell. That is, the S.LAST values of all the cells will shift down one cell with a zero filling on the top.

This is a simple instance of a technique here called precomputation. Until the instruction has actually been selected at the end of a cycle, it is not known whether it will, in fact, be a SEARCH NEXT instruction. However, if it is, one must already have the last M bit in S.LAST ready to pass to the next cell. That is, the value in S.LAST must be precomputed and available for a SEARCH NEXT instruction; it must be computed in every cycle whether it is needed or not.

3.3 Search and Hold

An inverted ordered set search can be implemented as a variation on the string search using Search Next instructions. It is only required to leave the tag bit M set to 1 if M became 1 (i.e., not to clear M until the end of the query). In order to mix this type of search with conventional string search operations, it is useful to select only some words in file, by means of a tag bit H, so they retain their value of M if H is 1, as in the search and hold instruction, while other words load M each time as in the search next instruction if H is 0. By this means, it is possible for a record in the data base to recognize whenever a sequence of arguments of string search instructions has a pattern, such as "a" followed immediately by "b", followed somewhere in the record by "c". The word storing "b" has tag bit H set to 1.

3.4 Search Lower in Record

This search consists of searching for and marking each word satisfying some condition which occurs after (lower in the record) a word which satisfies some previous condition. For example: search for all occurrences of "b" after "a". This is clearly similar to a "search next" except that the desired word need not immediately follow the previously marked word. This search is usually modified by taking records into account: the word "b" must be after "a" but in the same record as "a", not in a lower record. Records are separated by data words that are especially tagged as delimiters. We must find "b" after "a" with no delimiter words in between. This type of instruction can be used in ordered set searches discussed in Section 3.1. It is also the basis for "forward marking" since the word "b" is located forward on the disc as the head scans the disc. It will be contrasted to "backward marking" in Section 4.

3.4.1 Single Track. The problem is simply one of recording for each cycle the fact that a word with its match bit set has been encountered after a record delimiter has been passed.

Herein, a flip-flop S.ABOVE is initially cleared and is cleared whenever a delimiter is found. It is set when a word is encountered that had tag bit M=1. M is simultaneously loaded with 1 if the content search comparator gives a 1 and the value of S.ABOVE was a 1 just before the word was encountered; otherwise it is cleared. Next, each word is examined to see whether it is a record delimiter; if it is, then S.ABOVE is cleared.

3.4.2 Multiple Tracks. Operation within the track can be similar to the single track case if S.ABOVE is properly initialized. However, whereas the forward motion in a single disc track automatically propagates the M bit lower in the record, with multiple tracks it is necessary to propagate this bit to an indefinite number of lower cells up to and including the first one that has any delimiters in it in order to initialize S.ABOVE. Thus, it is necessary to know if any M bits between a delimiter and the end of a track are set, recording this in a flip-flop S.FOUND, and also to know if any delimiters had been found on a track, recording this in a flip-flop S.DELIM. Propagation is easily accomplished by a carry lookahead circuit (using, say, 74182's) propagating "carries" from higher to lower cells. S.FOUND and the complement of S.DELIM are input to the generate and propagate of the carry lookahead, and the "carry" is loaded into S.ABOVE. This initializes S.ABOVE so that the operating continues as in 3.3.1.

It should be noted that this same carry lookahead circuit is capable of being time-shared by many functions, some of which have just been mentioned. By setting all propagates to zero, the generate in each cell goes to the next lower cell, and only that cell. This was used in the search next operation. This circuit can also be used as the priority circuit to find the lowest cell with an operand-LIFO order. A carry lookahead circuit in the opposite direction, discussed later, can also be used to fetch instructions and obtain FIFO operands.

3.5 Burst String Searches

The string searches discussed in the previous sections require one cycle per word in the query. This creates a serious bottleneck. The following technique is capable of searching for strings of arbitrary length n in two cycles in most cases. However, an n word random access memory is required to store the words of the query. This algorithm is analogous to the string search algorithm commonly used in software: first search for just the first word of the string; when it is found search for the rest of the string.

3.5.1 Single Track Implementation. This algorithm requires an n memory B.M, a memory address register B.A, flip-flops B.LEFT and B.BUSY and a tag bit F on each word. In the first cycle, the first word of the string B.M.[0] is compared against each word on the disc, setting F if a match is found and clearing F otherwise. At the beginning of the second cycle, B.LEFT and B.BUSY are cleared. In the second cycle, when a word is found with F=1 and a burst search is not in progress B.BUSY is 0, F is cleared, B.BUSY is set, and a burst search is initiated as follows. As successive words appear after the first word, words B.M[1], B.M[2], ... are read by means of B.A and compared one at a time against the successive words read from the disc. The burst search is terminated and B.BUSY is cleared if either a mismatch is found or all the words match. In the latter case, the string is found and the algorithm is successfully completed. In the former case, a burst search is initiated whenever the next word is found with tag bit F=1.

Note that a word with F=1 may be encountered in the middle of a burst search. If this occurs, B.LEFT is set. If B.LEFT is set at the end of a scan and no match for the complete string has been found an extra scan is requested. Extra scans are requested until a match has been found or B.LEFT is clear at the end of a scan. The search terminates unsuccesully if B.LEFT is clear and no match for the complete string is found.

3.5.2 Multiple Track Implementation. A priority circuit and intercell communication to initialize B.BUSY are required for multiple track operation. Only one random access memory B.M is needed. Cells get access to B.M by means of the priority circuit which sets B.BUSY. The first cycle, comparing B.M[0] against all words and setting tag F, is done as in the single track case. B.BUSY is cleared at the beginning of the second cycle only, and B.LEFT is cleared at the beginning of the second and later cycles. In the second (and successive cycles), it is possible that a word with F=1 will be met in more than one track at the same time. A priority circuit is used to initiate a burst search (set B.BUSY and clear F) in only the prior track. The other track will set B.LEFT to request a burst search in a later cycle. It is also possible that a string will overlap tracks. This is indicated when B.BUSY is 1 at the end of a scan. The flip-flops B.BUSY are simply shifted down one cell at the end of each scan to continue the search. If B.LEFT or B.BUSY is 1 at the end of a scan, another scan is requested.

4. Set and Tree Context Searches

The content and string context searches described in the last section provide basic information retrieval functions. They are not sufficiently powerful for most information retrieval applications, however. As before, we are assuming that a collection of words is organized as a record, and that the first word in the record is tagged as a delimiter. It is at least necessary to organize the records as unordered sets having an indefinite number of words. It should be possible, for instance, if two words "a" and "b" are in such a set, to search for "a" and then change "b", whether "a" is higher than "b" or "b" is higher than "a" in the file. Note that the ordered set search operations could only do this if "a" was higher than "b" on the disc. The operation of forward marking used in ordered set searches must be mated with some techniques for backward marking.

It is also quite useful to search these unordered sets for a subset of words given by the operands of some search instructions. A mark bit is needed for the set as a whole to indicate if the set is a successful candidate for continuing a string search. This bit can be maintained in the delimiter word at the beginning of the record. (Actually, a stack of bits in the delimiter is maintained to permit logical operations on the results of searches.) Then, the results of searches are uniformly sent to the delimiter word, using backward marking which will be discussed below, and this result is available to affect further searches or modify instructions in the record by means of forward marking techniques similar to those used for search lower (Section 3.4). This technique can be used for unordered set searches and for modifying words in the set without regard to their order.

The solution to this problem can be generalized to handle tree data structures. Tree structures naturally handle such data management problems as are found in corporations, armed forces, or libraries. We shall consider the extension to backward marking to handle searches in trees as well.

4.1 Backward Marking

The general problems is that for every record, the successful result of a comparison between an operand and any word in it should be stored by setting a bit in the delimiter. If one or more words in a record satisfy the comparison, then the result stored in the delimiter should be 1, but if no words satisfy the comparison the result should be 0.

4.1.1 Single Track. For a record contained entirely within a segment, the problem is simply one of getting the result of a search in a word in the record stored into the delimiter word that has already been passed in the segment.

Implementation of this function uses a delimiter counter M.DCT and a 1 x n random access memory RAM where n is the maximum number of records possible on the disc segment. M.DCT is able to be used as an address to read or write in RAM. Each bit in the RAM is a carrier for each record whereby it is set in one revolution in the main processor by the result of a comparison on data words, and is unloaded into the delimiter word and cleared in the next revolution in the post-processor. Initially, M.DCT is cleared, and it is incremented each time a word with a delimiter tag is encountered. Thus, the i^{th} bit of RAM is addressed for reading or writing as words in the i^{th} record are encountered. During a backward marking instruction, if a comparison between the comparand and a word met on the disc is satisfied, the bit RAM [M.DCT] is written with a 1. In the next revolution in the post-processor, the bit RAM [M.DCT] is copied into the mark bit in the delimiter word and is cleared. Note that if two successive backward marking instructions occur, the reading and clearing of the RAM in the post-processor due to the first instruction occurs simultaneously with the writing of the RAM in the main processor; however, the reading and clearing is logically performed immediately before the writing.

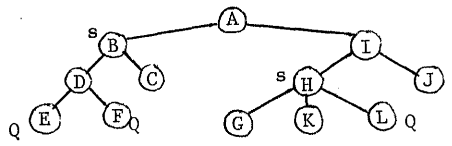
4.1.2 Multiple Tracks. As with the search lower operation, this operation should permit a record to overlap multiple tracks. The operation takes place over two cycles as in the single track case except that S.DELIM (see 3.4.2) is set in the first cycle if any delimiters are found. Note that after the first cycle, if one or more tracks contain a continuation of a record begun on a track to the left, the result of the comparison is automatically put in bit zero of the RAM, and note that at this time, M.DCT of the cell to the left of this continuation tracks (the cell containing the delimiter for this record) still points to the RAM bit which is associated with the record spread over these tracks. It is only necessary to put the OR of these bits into RAM[M.DCT] of the cell to the left. From there, it can be distributed to the rest of the record by forward marking.

A carry lookahead circuit can transfer these bits. It propagates from lower to higher cells. The complement of the delimiter indicator S.DELIM is the "propagate", the bit RAM[0] is the "generate", and the "carry" is put into RAM[M.DCT] at the end of the first cycle of a backward mark operation, before M.DCT is cleared for the next cycle.

4.2 S-Q Tree Search

By simple extension to forward and backward marking, an S-Q search of a tree can be conducted. The records of the previous searches are here considered to be nodes of a tree. The S-Q search refers to searching particular nodes of a subtree of some larger tree. Typically, a tree will have several subtrees defined within it. The subtree consists of some node (marked S) and all of its descendants. Within the subtrees, some nodes are marked as being qualified for a search (Q-nodes). A well-formed tree has no Q above S and no S above S. The S-Q search is a search of all of the qualified (Q) nodes of a subtree; the results are stored in a bit of the S-node of each subtree--rather like a backward marking of nodes.

In what follows, we will assume that the tree is stored in left list matrix order, somewhat like putting a tree structure into an outline form like that used in writing this paper. For example, the tree below would be stored as:



A	B	D	E	F	C	I	H	G	K	L	J	node
	S		Q	Q	Q	Q	S			Q		mark

Fig. 4

From the way the tree is stored, if the S and Q nodes have been marked (as we discuss later) and are well-formed, it is clear that this search will be analogous to the backward marking in that each S-node is marked with results of a search of words between it and the next S-node. The major difference will be that the search is confined solely to qualified nodes (Q-nodes). A bit will not be set in the RAM when a comparison between the operand and a word unless the word is in a Q node. As before, the search results will be ORed into a RAM, and the Post-Processor will mark the appropriate word. Now, however, the RAM is addressed by a count of S-node delimiter words. That is, M.DCT is incremented only when a delimiter of an S node record is encountered. Also, the multiple track case requires an identical carry lookahead circuit. However, the propagate bit from each cell is from a flip-flop which is set whenever the delimiter of an S record is met.

The implementation of this technique requires tag bits S and Q in delimiter words which can be set or cleared; S=1 indicates that the node which this delimiter word begins is an S node. It required, for the multiple track case, the flip-flop S.DELIM, which is set if a delimiter of an S record is met, and its complement is input to the propagate of the carry lookahead. Also, a flip-flop M.Q is needed to indicate that the record we are in is a qualified record (Q record). M.Q and S.DELIM are initialized and run as in the search lower operation, and backward marking is done as in set searches, using delimiters of S nodes to control S.DELIM and set searches.

4.3 Establishing S and Q Bits

In order to utilize the general S-Q tree search, one must set the S and Q bits. It is possible that each delimiter word will have a unique code word, so that content addressing can be used to find the delimiters and rewrite the S and Q tag bits. However, herein, we describe techniques whereby S and Q bits can be "walked" through the tree structure to their final location for an S-Q search, in a manner similar to the CDR-CAR operations in LISP. This adds another dimension to the searching capability of the machine, particularly with regard to artificial intelligence.

Herein, it is assumed that each node has a level number (equal to the number of nodes in a chain between it and the root of the tree) and a name. These can be stored in the delimiter of each node. In the following discussion, we need only look at the level and name in the delimiter words. We shall now ignore the data words within the records. For any node, its ancestors are nodes between it and the root of the tree, its descendants are nodes in the subtree below it and its sons are nodes immediately below it in its subtree, following terminology from the "family tree". It is necessary to be able to identify ancestors, sons and descendants of any node, particularly an S node and a Q node. Then, one can search for a node, using content searching for the name of the node, which is the son of an S node, erasing the old S bit and setting S in the son or sons that satisfy the search. A sequence of such searches is analogous to CDR-CAR operations in LISP. However, unlike LISP, trees other than binary trees can be used, more than one son can be marked, and any ancestor or any descendant that satisfies the content search can be marked.

In the following discussion, S marks will be moved about in the tree. Q marks can be moved about in similar fashion. We will assume that before the operation, the tree is well formed. The son and descendant searches are shown first. The problem is to identify nodes that are in the subtree of nodes that were marked with S=1. The ancestor search is the inverse of the descendant search; the potential ancestor must be uniquely identifiable by content addressing and the subtree of that node is then checked to see if it has an S node. If so, the potential ancestor is finally marked.

4.3.1 Single Track. The implementation of son and descendant searches requires a save register M.L to save the level of an S node, and a flip-flop M.D to indicate that the delimiters being examined are in a subtree of an S node. M.D is initially zero. It will be used, together with the comparator, to find new delimiters to set S=1. The node delimiters will each contain a level number and a unique name in specific fields and some will be marked in a particular bit (S-bit) as being S-nodes. If we assume that the hardware is told only whether it should look for son or descendant of an S node, then it will first have to search for any S-node and when it is found, set M.D, record its level number (LEVEL in M.L). Then it will have to compare level numbers LEVEL of a succeeding node to M.L: for son, the comparator must check for LEVEL = M.L + 1, for descendant, LEVEL > M.L and to indicate when the subtree is left, M.D is then cleared when LEVEL < M.L.

The ancestor search requires the use of the RAM and M.DCT counter because it is a backward marking operation. M.L and M.D are used as in the previous section. It is necessary to be able to identify potential ancestors (PA's) by content addressing alone, and such PA's must be well formed (i.e., no PA is in the subtree below another PA). Two cycles are required for this operation. M.DCT and M.D are initially cleared. When a PA is found, its level is put in M.L and M.D is set. M.D is cleared when a delimiter with level LEVEL $\leq$ M.L is found. If an S node is found while M.D is 1, then RAM[M.DCT] is set. In the next cycle, M.DCT is cleared and incremented as before; S bits are first cleared, and RAM[M.DCT] is put into each S bit of delimiter word.

4.3.2 Multiple Tracks. If a tree is stored over several tracks (Fig. 5), then the immediate hardware problem is one of intercell communication. The subtree may cross cell boundaries, and each cell needs to know at the start of the cycle if it is starting in the middle of the proper subtree with S=1 to determine if it is a descendant, or a subtree of a PA to determine if the PA is an ancestor and it needs to know if it is a son. In terms of the hardware outlined for the single track case, this means properly initializing M.L and M.D.

Because trees are well formed and stored in left-list order, it is possible to acquire information to initialize M.L and M.D on a single cycle immediately before the actual operation cycle. It can be shown that the cell which contains the end of the subtree can be found by obtaining the minimum level number LL within each cell. If a subtree consists of a given node at level M.L and all of its descendants, then clearly any cell which contains the end of the subtree must have a node of level L $\leq$ M.L. Thus, LL $\leq$ M.L.

The hardware problem is one of detecting in one cycle the lowest level numbers of each track. At the end of this cycle and before the start of the operation cycle, this information can be used to compute the initial values for M.D and M.L for each cell.

This technique requires the register M.L and flip-flop M.D discussed before a flip-flop S.DEL and a register M.LL to hold the least level and a propagate circuit. The operation takes two cycles. In the first cycle, M.LL is first set to a large number, then loaded with min (M.LL,LEVEL) where LEVEL is the level of each delimiter encountered. Meanwhile, S.DEL is initially cleared, and is set if an S node is found for son or descendant searches, or if a PA node is found for ancestor searches. M.L and M.D are computed as in the single track case but no S bits are changed yet. At the end of this cycle, assuming the tree is well formed, it is only necessary to send M.L of any cell with M.D = 1 to all lower cells down to and including one with S.DEL = 1.

This can be done using a carry lookahead circuit, sending one-bit of M.L at a time through the generate to the carry and to M.L, where the complement of S.DEL is the propagate signal. M.L is correctly initialized this way. To initialize M.D, M.L is compared to M.LL, and the signal which is one if M.L $\geq$ M.LL is the propagate, M.D is put into the generate and the carry is put into M.D. Now that M.L and M.D are initialized, the second cycle is executed as in the single cycle case and S bits are now changed as in that case.

5. Pointer Transfers

A means to store pointers and to efficiently transfer signals via these pointers is invaluable for associative nets and other data structures. Each record in the data file is assigned a logical address LA equal to the number of records above it. A pointer from record "A" to record "B" is implemented by storing the LA of "A" in a word in "B". In its simplest form transferring of pointers means that, as several records are marked, all records pointed to from such marked records will become marked. A second form is where pointers are reflected. This means that, as several records are marked, all records that point to a record which is also pointed to by such marked records will become marked. It should be noted that pointer reflection is tantamount to a massive parallel content search. It is capable of higher performance than any other technique described in this paper. These forms are modified by content and string or tree context addressing to make them useful. However, in the ensuing discussion attention will be focused on the transferring of pointers and the reflection of pointers.

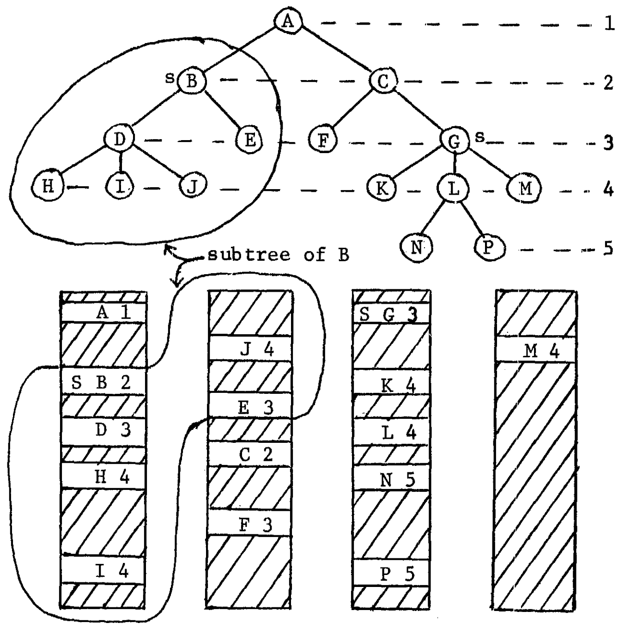


Figure 5 -- A TREE STORED OVER SEVERAL TRACKS

Pointer transfer means herein that several words in the data base storing pointers have been marked as source pointers, and all records which have logical addresses equal to the value of any of these marked words are to have their delimiters marked. Pointer reflections means herein that several pointer words have been marked as source pointers, several words have been marked as sink pointers, and wherever a sink pointer is equal to any source pointer, the sink pointer should be further marked.

5.1 Single Track

These techniques utilize the association of each record with a random access memory bit. A RAM and delimiter counter M.DCT are used. Initially the RAM is clear. In the first cycle, as a source pointer with bit value POINT is encountered, RAM[POINT] is set. For pointer transfers, in the second cycle, M.DCT is initialized to zero and is incremented as each delimiter word is met. RAM[M.DCT] is read out and marks the delimiter word. For pointer reflections, as each sink pointer with bit value SINK is encountered, RAM[SINK] is read out and marks the sink pointer word. After each such operation, RAM must be cleared.

5.2 Multiple Tracks

In multiple track systems, the RAM bit in the appropriate cell has to be set or read, and memory access conflicts have to be resolved. We posit an overall virtual random access memory RAM' which is stored in the RAMs in each cell. Each record is associated with a logical address LA' equal to the number of records above it in the file. RAM'[LA'] is associated with the LA'th record. Yet on each track the logical address LA is defined for each record as before, as the number of records above it on the track. As before, RAM[LA] is associated with the LAth record. Note that this permits the second part of a pointer transfer operation to be done exactly as it was for the single track case. The conversion from LA' to LA is quite simple: if TD_i is the total count of delimiters on cell i then on the ith cell LA' is LA + $\sum_{i=1}^{j-1} TD_i$. Then to read or write RAM'[LA' it is necessary to locate the cell it is in and to determine LA. To do this, one feeds LA' into the left input of the leftmost cell, and each cell subtracts TD_i from what is input on the left, outputting it on the right to be input on the left of the next lower cell. The leftmost cell that outputs a negative number uses the number put into it, a positive number, as LA to read or write RAM[LA].

Note that only one bit at a time can be read or written in RAM'. It may happen that two cells may encounter a source pointer at the same time. Although this should occur very rarely, it creates a memory access conflict because it may require setting a bit in two different locations of RAM' using two different values of LA' at the same time. A buffer register can store LA' in the two cells so that both can be used as addresses at successive time slots. Nevertheless, these buffers may be full when a source pointer is encountered. Thus, it is necessary to mark source pointers, deleting the mark when the pointer is actually transferred. If any pointers are still marked at the end of a cycle, an extra cycle is required to transfer these remaining pointers. It is expected that all source pointers can be recorded in the RAM in a few cycles for either pointer transfer or reflection, and that all pointers can be read from the RAM in a few cycles for pointer reflection.

The implementation of these techniques requires the RAM in each cell, register M.TD; counter M.DCT flip-flop M.LEFT, tag bits SR and SK on pointer words to indicate source and sink pointers, or priority circuit, a one bit buss B and a special adder between cells. For an arbitrary number of cells, a serial tree adder provides a practical adder. See Fig. 6. The adder cell is shown in Fig. 6a. Numbers are fed serially, least significant bit first, through a, d and f. c is just a, e is a+d and b is d+f. Note that in a tree constructed from such cells as shown in Fig. 6b, any downward-directed link generally has the sum of all values to the left of it, and any upward-directed link has the sum of all values below it. The bottom nodes connect to the processing cells that contain the RAM and disc tracks discussed earlier. Note that sums are accumulated from left to right in these cells. For instance, D is A+B+C. If A is the logical address LA', and B and C are the negatives of their delimiter counts ($-TD_i$) then D is LA.

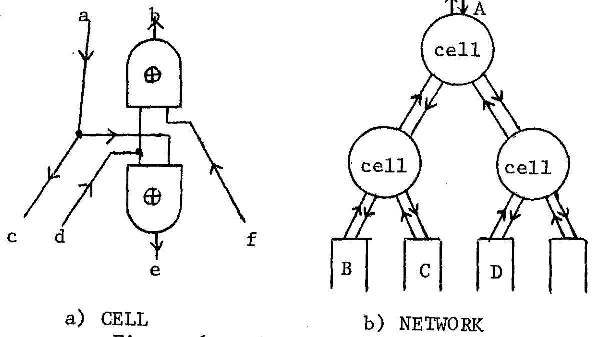


Figure 6 -- A TREE ADDER

Pointer transfer is accomplished as follows. SR is set on source pointers and M.DCT has the number of delimiters in each track, as in the backward marking operation. The negative of M.DCT is put in M.TD. RAM and M.LEFT are cleared. If one of the cells meets a source pointer and the priority circuit finds that it is the prior one (leftmost) having a source pointer, then tag bit SR is cleared, the value of this word LA' is put into the top of the tree (at location A in Fig. 6b), and M.TD is put into each bottom input to the tree (locations B, C, etc. in Fig. 6b). The bottom outputs (location D, etc.) are collected. If the next right cell has a negative value and this cell has a positive value, then that value is used as LA to set RAM[LA]. If another cell finds a source pointer and is unable to send it out because the priority circuit prevents it, tag bit SA is not cleared but M.LEFT is set. At the end of the cycle, if any cell has M.LEFT = 1, then another cycle is used to send out remaining pointers, and M.LEFT is initially cleared for this cycle. This continues until all source pointers are sent out (M.LEFT is zero in all cells). In the final cycle, the bits in RAM are put in the delimiters using M.DCT as in the single track case.

Pointer reflection is similarly handled. First, source pointers are transferred as above. When all source pointers are transferred, sink pointers are selected just as source pointers, their LA's are translated, and the bit RAM[LA] read from the selected cell is output on a buss B, to the cell containing the sink pointer, where it is written.

6. Summary

We have developed implementations for some basic segment sequential functions. As mentioned at the beginning, however, these functions do not constitute a complete set of primitive functions for data base storage and management. The concepts of word insertion and deletion, collection (of words which satisfy search criteria), and garbage collection (packing of unused words to eliminate fragmentation) have not been mentioned here, but are discussed by De Martinis, et al.[1]

All of these concepts were developed for use in the CASSM system at the University of Florida, but this system is intended to be a research vehicle for determining just which of these concepts will prove to be useful for specific applications. The basic functions and their implementations are presented independently here in the belief that they will prove useful in the design and implementation of further systems with specific applications.

References

1. De Martinis, M., "A Self-Managing Secondary Memory System," to be published.
2. Fuller, R.H., Bird, R.M., and Worthy, R.M., "Study of Associative Processing Techniques," AD-621516, August 1965.
3. Healy, L.D., Doty, K.L., and Lipovski, G.J., "The Architecture of a Context Addressed, Segment Sequential Storage," Proceedings of FJCC, Vol. 41, Part I, 1972, pp. 691-702.
4. Hollander, G.L., "Quasi-Random Access Memory Systems," Proceedings of EJCC, 1956, pp. 128-135.
5. Minsky, N., "Rotating Storage Devices as Partially Associative Memories," Proceedings of FJCC, Vol. 41, Part I, 1972, pp. 587-596.
6. Parhami, B., "A Highly Parallel Computer System for Information Retrieval," Proceedings of FJCC, Vol. 41, Part I, 1972, pp. 681-690.
7. E.A. Feustel, "On the Advantages of Tagged Architecture," IEEE Trans. Computers (July 1973) pp. 644-656.
8. Ozkarahan, E.A., Schuster, S.A., Smith, K.C., "A Data Base Processor," Technical Report CSRG-43, University of Toronto, Nov. 1974.
9. Lipovski, G.J., Su, S.Y.W., "On Non-numerical Architecture," Computer Architecture News, Vol. 4, No. 1, March 1975, pp. 14-29.
10. Su, S.Y.W., Lipovski, G.J., "CASSM: A Cellular System for Large Data Bases," Proc. International Conference on Very Large Data Bases, Farmingham, Mass., Sept. 1975, pp. 466-472.

A SELF MANAGING SECONDARY MEMORY SYSTEM*

Manlio DeMartinis
Departamento de Circuitos y Medidas
Universidad de Carabobo, Venezuela
G. Jack Lipovski
Stanley Y.W. Su
J. K. Watson
Department of Electrical Engineering
University of Florida
Gainesville, Florida 32611

Abstract

A Self Managing Secondary Memory (SMSM) organization is proposed herein, in which hardware directly assists the storage, retrieval and management of arbitrary length records on such devices as fixed head discs or charge coupled devices (CCD's). This paper emphasizes some of the techniques used to implement an SMSM system.

In an SMSM, fixed length words are organized into variable length records, and these records are packed into a file. The first word of the record, a label, can be associatively addressed to mark the record. Marked records can be output, erased, or a word or a collection of words can be inserted after the label of such records. Erased words are shifted to the bottom of memory as data words are packed upward, so that new records or extensions of old records can be inserted at the bottom of the file. In this system, although the file appears to be a single one dimensional array of words, it is actually stored on a number m of n word circular access memories, such as CCD's or tracks of a fixed head disc. Larger systems are implemented by increasing m. The access time for the entire system depends only on n.

This architecture is self-managing in that no directories are kept, nor is software garbage collection or allocation necessary. The hardware replaces these functions. This appears to be a desirable direction for secondary memory architectures to develop, with special application to their use in computer networks.

This paper discusses techniques for implementing an SMSM. These techniques were developed as part of the Context Addressed Segment Sequential Memory (CASSM) system. This paper therefore also describes that part of CASSM that, by itself, forms a useful SMSM. It is hoped that these techniques will be useful in the development of a new class of intelligent secondary memories to meet present and future needs of computing systems.

I. Introduction

A traditional secondary memory system represents a very large part of the global investment in a computer system. The smaller the computer, the bigger the cost of the secondary memory is with respect to the global cost. Since the secondary memory is a "passive" element with respect to the data processing, part of its cost is in the software needed for the managing system. Traditionally, a sequential secondary memory, such as a fixed head disc, is treated as a random addressed memory. Fixed length words are organized into fixed size tracks and sectors, and are located by track, sector, surface and unit addresses. The software maintains a directory to map the name of a variable length record into the address(es) of sector(s) where it is stored. Garbage collection software also collects unused space from time to time. However, as hardware costs drop, it is feasible and attractive to put some processing logic on each head of a fixed head disc, or on each charge couple device (CCD) memory to assist these software functions. By replacing random addressing by content addressing of a label in each record, the records do not have to be tied down to tracks. Several short records can be put on one track, or a long record may be put on several tracks. Each record will have a label word, and the record will be located by its label. Since several records with the same label will be retrieved together, there is no need for linking such records in the directory. In effect, the "directory" is stored with the data by means of the labels. Moreover, since the records are not tied down to tracks, words in records can be moved from one track to another as hardware automatically collects garbage words into one area where large records can be input. The memory is self managed to efficiently store variable length records.

A self-managing memory has three attractive features. Whereas, in a conventional system, loss of the directory will usually cause loss of all the data, since the "directory" is stored with the data in this system, the directory is only lost when the data is lost. This system can be more fail-soft. Whereas, in a conventional system, the directory may be so large that it is stored on the disc and two accesses are required to get the directory first and then the data, in this system, it is not necessary to retrieve the directory. This feature saves the time required for one access to the disc. Moreover, it does not require the computer to store and search the (large) directory in its primary memory. This is a very attractive feature for small computers that do not have much memory. Finally, whereas, in conventional systems used with computer networks, long protocols are required to gain access to the directory, in this system the only requirement is the establishment of unique labels and the passing of these labels between cooperating processes. For example, it would be reasonable to initially apportion groups of labels to each of the processes, which individually assign them upon execution of the UNIQUE function (6) to records created by the process. Each processor gains control of the disc in a simple way, such as a hardware priority circuit. It uses the label to access the record. Cooperating processes need only have the label to access the same record.

In the design of the Context Addressed Segment Sequential Memory (CASSM) system (1, 2, 3), several techniques have been developed that allow automatic managing of variable length records stored on a segment-sequential memory system such as a fixed head disc. These operations include: garbage collection, insertion of single words or blocks of words in a record input of records at the end of the file, and several modes of outputting data. While these techniques were developed for a more powerful machine capable of searching relational, hierarchical and network data bases, these techniques in themselves make up a useful self-managing secondary memory system. The purpose of this paper is, then, twofold: 1) show those techniques of the CASSM system that relate to input, output and garbage collection as part of the series of papers on CASSM; and 2) cast these techniques in a self-contained paper that

*This paper was supported by NSF grant GJ-43225

shows how a relatively simple self-managing secondary memory can be constructed with the desirable features given above. It is hoped that this paper will show such techniques as will be useful in the development of a new class of intelligent secondary memories to meet present and future needs of computing systems.

2. System Description

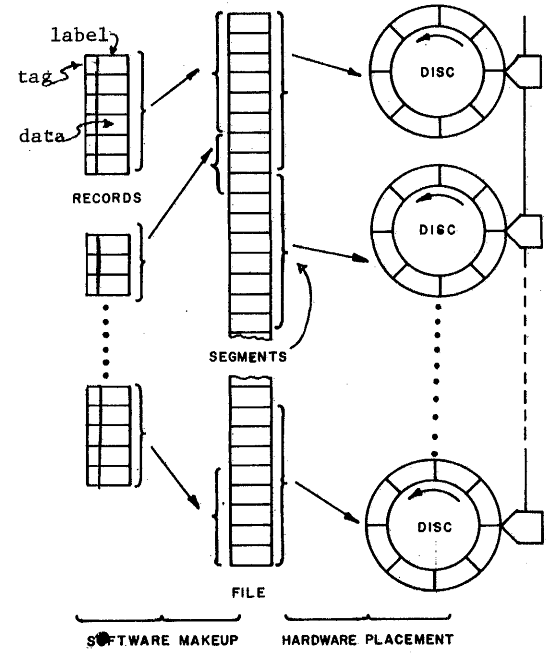
In this section, the software view of the file structure is presented first. Then operations on the file are presented. Finally, the block diagram of the system and construction of the cell are outlined. While this section describes the architecture of an SMSM, it must be emphasized that this architecture is presented only as a vehicle to describe the techniques, developed in later sections, in a more cohesive form.

2.1 Word and File Structure

Figure 1 shows the basic word structure. Each word is fixed length, and consists of two fields: TAG and DATA. The TAG field (7) is used to distinguish the several types of words that may exist simultaneously in the system memory and mark words for processing. The DATA field stores the actual data of the word. Figure 2 shows, in its left half, the basic software file structure. RECORDS consist of a variable number of words, the first of which is a label for the record. All the records are packed together to form the FILE. Note that only one FILE exists in memory. Unused words "below" the FILE are available for storing new records.

	TAG			
DL	0	X	C	LABEL
DT	1	0	C	DATA
GB	1	1	0	Don't care
EOF	1	1	1	Don't care

Figure 1. WORD STRUCTURE



gure 2. STORAGE OF DATA

For the purpose of searching operations and garbage collection, the TAG field must differentiate, at least, between the following types of words:

a) Data (DT): This word identifies an actual data word. For the purpose of this paper it will be considered that data words consists only of a fixed length string of ASCII characters.

b) Delimiter (DL): This word marks the start of a logical record within the file. A record is defined as the block of contiguous delimiter and data words, starting with a delimiter, up to but not including the next delimiter.

c) Garbage (GB): This word is to be deleted by garbage collection hardware.

d) End of File (EOF): All words below the file are EOF words. The memory is empty if filled with EOF words.

The tag bits, X and C, are used to mark delimiter and data words for input or output. These are discussed in the next section.

2.2 Operations on the File

The general operations on the file are discussed first. Then, an "instruction set" of input/output commands is presented that implements these operations. The general operations are input, output, and deletion of records.

The normal technique for inputting new records is to put them at the end of the file. An input command exists for this type of input. There is a command to initialize the memory by filling it with end-of-file (EOF) words. The memory can then be filled by inputting at the end of file. It is sometimes possible to add new words to an existing record with label L by inputting a new record with label L at the end of file. However, it is occasionally necessary to input a single word or a block of words at the beginning of an existing record. Two input commands are provided for such cases. There is a command to mark delimiters with label L to prepare for inputting single words or blocks of words below such marked words.

Output is generally accomplished by a command that sets the collection bit C in all delimiter or data words in a record or records that are to be output and selecting three options for the mode of output. This can be done in one revolution. The words are output later. As collection words are output, the first mode of the output instruction can cause the SMSM to merely clear the C bit in such words, to leave the word in place but prevent it from being output again until the C bit is set again (SAVE) or to delete the word (DELETE). The SAVE mode can be used when the memory is storing permanent files. The DELETE mode can be used when the memory is being used as a spooling device. Words can be deleted as they are used. Unfortunately, there is no way in a multiple segment system to quickly output words from many records in the same order that they are stored in the file. The output is controlled by a second mode which may be RANDOM, DELIMITED, ORDERED, or UNIQUE. The RANDOM mode outputs words quickly (essentially in one disc revolution) but they appear in a different order than they are in the file. The DELIMITED mode outputs words in a record in the same order as they are stored in a record, but the records may be output in a different order than they appear in the file. Output in this mode takes somewhat longer time than in the RANDOM mode. The ORDERED mode outputs words in the same order as they appear in the file. For such multiple track systems, this output mode takes as many revolutions as there are tracks that contain words to be output. (Note that for outputting single records, this mode can be quite fast.) Finally, the UNIQUE mode, which is an important output mode in sophisticated systems like CASSM, outputs words in ordered mode but does not output a word if that same codeword were output already.

There is a possibility that two output commands can be executed sequentially so that collection words from both commands may be outputting together. This may be deliberate or it might be unacceptable. A third mode of the output command (MGC, WTC) can halt the processing of the command until all collection words are output (wait for collection, WTC) or execute the command regardless of whether collected words will be mixed together upon output (merge collection, MGC).

In addition to the output command, there is a command to cancel all output by clearing the C bits. It is useful, especially when output is done in the DELETE mode, so that the computer receiving words from this memory can stop output when its input buffer is full.

Garbage collection is accomplished by a command that changes data or delimiter words into garbage words. Such words migrate to the bottom of the file, where they become end-of-file words.

The user sees only a one dimensional file of fixed length words, arranged into variable length contiguous word records that have a delimiter word (DL) which has a label word stored at its beginning. For the simplicity of presentation, we propose the following commands, although more complex commands would undoubtably improve the performance of the machine. Herein, L is a code word to be compared against labels, and B is a full word, including tag bits. These commands are supplied by means of a direct memory access (DMA) input channel from a small or large machine, or a computer network. An output DMA channel is implicitly controlled through the input DMA channel.

<u>Erase Memory</u> EM

The entire memory is filled with words with tag EOF (end of file).

<u>Input at End of File</u> IE B

The word B replaces the first EOF word, thus being appended to the end of the last record. A rapid succession of such commands can append many words to the bottom of the file in essentially one revolution of the disc.

<u>Output record</u> OR L,m_1, m_2 · m_3

The record with label L is output using modes m_1, m_2, m_3. This sets the C bit on all words in the records with label L and sets the output mode m_1, m_2, m_3. C marked words will be output later under mode m_1 (SAVE, DELETE)-determines whether words that are output are deleted or not, m_2 (RANDOM, DELIMITED, ORDERED, UNIQUE)-determines the order of output from multiple cell systems, and m_3 (MGC, WTC)-determines whether this command must be delayed until all previous collection words are output.

<u>Kill output</u> KO

All C bits in memory are cleared, thereby preventing any more output.

<u>Delete record</u> DR L

The words in the record(s) with label L are rewritten with tag GB(garbage).

<u>Mark label</u> ML L,n

The X bit is set to 1 in any delimiter word(s) having label L and is cleared in other delimiter words. The number n needs to be saved for block insertion to indicate the number of words in the block.

<u>Write label</u> WL L

The label L is written into the label field of any delimiter word that has X=1.

<u>Input word</u> IW B

The word B is inserted immediately below any label marked with X = 1, and X is cleared. All words below the label with X=1 are moved down one word in the file to make room for B.

<u>Input block</u> IB B

The word B is inserted into the block after the label marked with X = 1, and X is cleared. Only one delimiter word in memory may have X = 1 when this instruction is executed. A rapid succession of such commands can insert a block of words into the disc in (essentially) one revolution of the disc.

Clearly, an I/O program can be written using the I/O commands given above. For instance, to add a new record consisting of delimiter word B_1, and data words B_2 and B_3 to the file, one executes

IE B_1

IE B_2

IE B_3

In the next section, the implementation and timing of these commands in a cellular system is considered.

<u>2.3 System Implementation</u>

In a content addressed system, a high degree of parallelism for data processing and overall operations is possible and desirable. Systems with such architectural characteristics have been presented in [1, 2, 5]. Based on these ideas an outline of the basic architecture for the SMSM follows:

a) The system will consist of a linear array of identical cells in which each cell communicates directly with its two neighbors and with a common I/O bus (see Fig. 3). The array will be considered vertical, using such terminology as top cell, next lower cell, etc.

b) Each cell consists of a segment of memory, to be implemented, for instance, with CCD's, and a logic section which is able to search, modify and rewrite data, and perform input/output and managing operations (see Fig. 4).

c) Last, a <u>C</u>ontrol <u>M</u>odule (COM), will control all the common operations in all the cells, communicate to a computer or network via input and output DMA channels commands received from the computer to all the cells, where they are executed in parallel by all the cells. See figure 3.

Within each segment of memory, the data is written in a bit-serial mode, and all the cells run on a common clock. In that way the data is shifted synchronously in all the memory segments. The data shifted out from memory is fed into the logic section, where it is processed, and then rewritten into the memory segment.

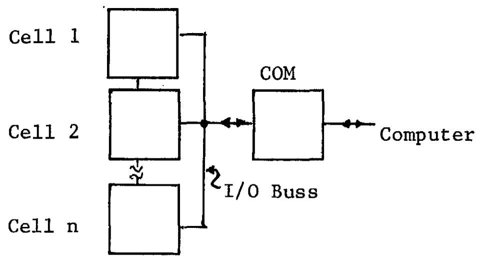


FIGURE 3. SYSTEM STRUCTURE

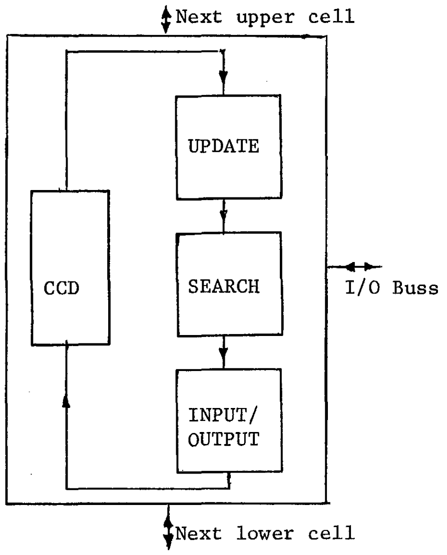


FIGURE 4. CELL STRUCTURE

As was noted in section 2.1, the words will be organized in records, and each record may contain a variable number of words. The different records of a file will be stored in the system in sequential order, starting with the top cell, and continuing with the next lower ones. In that way, variable length files are divided into equal segments, each one containing a number of words equal to the amount of valid data that can be stored in one cell memory. This is shown in the right half of Figure 2. Each cell memory may contain one or more whole records, or only part of a record, and a record may start in one cell and continue in the next one.

Within each cell, the memory segment is scanned. Thus, in all cells, the top word in each segment is read by each cell concurrently, then the next word in each cell is read, and so on. This process is here called a scan. After the last word has been read, a short gap occurs, and the first word of each segment is again read concurrently in each cell. The scan and gap together form a memory cycle. Note that the cycle time is dependent upon the number of words in a segment, but that the size of the memory can be extended indefinitely by adding more cells without increasing the size of the segment.

The following commands wait until the beginning of each scan and last throughout a scan: EM, OR, KO, DR, ML, WL and IW. However, the command OR will also have to wait until the last output is complete (all C bits are zero) if mode m_3 is WTC. The IE and IB commands wait until some segment finds the first EOF word, or X=1 delimiter, respectively, but a rapid succession of such commands can be executed to write words one after another into the memory after this.

Note that the OR and KO commands control the system output direct memory access channel, and that the DR command controls the garbage collection system. These systems operate concurrently with the input channel to achieve a degree of speedup in the SMSM architecture.

The logic section within each cell contains several different modules that perform specific functions on the data as it is shifted through them. The functions performed in these modules, and a simplified description of the logic in each module, will be discussed now.

The search for a given word, and the modification of the matched ones takes place in the search processor module. Therefore this module must, at least, contain a one word length shift register where the data is held while it is being examined and possibly changed after it has been examined, an input to receive the operand from the COM module and a serial comparator. Several "flags" (or flip-flops) must be implemented in that module in order to store the results of matches when the operation to be performed must be extended to subsequent words of a record, or when that information is needed later on for other operations. In particular, delimiter words are searched by the ML command to set X=1 in preparation for word or block insertion in that module. A flag is set to continue the IB command in a segment, and intercell links are provided to resume such IB commands on the next lower cell when block insertion overlap several segments. A flag is set for the OR command to set the C bits in the delimiter and following data words to collect them for output. Similarly a flag is set for the DL command to change the delimiter and following data words to garbage words to prepare them for garbage collection. Intercell links and logic is provided to initialize these flags whenever a record overlaps two or more segments so that the entire record is eventually marked for collection or made into garbage words.

The I/O module will handle input at the end of file, when the IE command is executed and all the modes of output. Other modes of insertion take place in the update module as explained shortly.

To handle input at the end-of-file, the data to be input is provided from the input DMA (direct memory access) via the COM module and buss as an operand of an IE command. A tag comparator for EOF words is implemented. A flag is set to mark the cell to determine the highest cell having EOF words. The topmost cell having EOF words will replace that word with an input word provided by an IE command. As long as IE commands continue to be given, their operands are written in successive EOF words.

The output hardware will detect words pre-marked for output, (C=1) and set a flag to mark the cell for priority purposes. Priority will be determined by the mode of output, as will be explained in section 5. The module of the cell that has priority will communicate via the COM module with the computer by means of the output DMA (Direct Memory Access) channel.

Within the update module, garbage collection hardware deletes garbage words and shifts the valid data toward the top of the overall system memory, increasing the availability of space at the end of the system memory. Word and block insertion also take place within this module. The update module will consist of a tag comparator for the detection of garbage words and words with X=1 that define the place for insertion of words or blocks.

As explained later, the garbage collection circuitry will need a two word shift register having taps on the input, middle and output, that provide the necessary slack to expand and contract the data stored in each cell memory. An up-down counter and associated control circuitry is needed to select the taps on the shift register.

The COM module will have all the hardware needed to interface with the computer, a counter register which is loaded by the ML command, a subtractor to handle the block insertion and a shift register to provide comparands to all modules for the OR, DL and ML commands. The words for word insertion or block insertion are provided by the input DMA by means of the COM module and buss as operands of IW or IB commands.

Besides all the above mentioned hardware, each cell must be provided two extra registers that will hold the first and the last word stored in the CCD memory. These registers are used for the word shifting between cells in the garbage collection and word insertion operation. The implementation of some functions (e.g., serial comparison) is well understood and need not be further explained. However, the implementation of several processes deserve greater exposition. The remainder of this paper will focus on garbage collection, input and output, which are at the heart of a self-managing secondary memory. Garbage collection and word insertion, as proposed by Copeland et al [2] is explained for multiple track systems, and input and output techniques are introduced. These will be treated in detail in the following sections. Further techniques for searching data in segment sequential memories can be found in the paper by Bush et. al. [4].

3. Garbage Collection and Word Insertion

Garbage is created when records are erased; or as we see later, when a block of words is inserted that is not equal to the size of a memory segment. Such garbage words need to be collected towards the bottom of the memory where they become EOF words to provide room for large records to be written.

One of the ways of inserting limited amounts of data into existing records is word insertion. It is the inverse of garbage collection. It is discussed along with garbage collection because it interacts with that process. In the following sections, garbage collection, word insertion, and then multiple word garbage collection and insertion are considered.

3.1 Garbage Collection

As the result of processing, words are erased from time to time by changing their TAG field to "garbage" type words. Garbage words are automatically moved toward the bottom of the system memory, as good words are packed toward the top.

In order to explain the basic mechanism of how the garbage collection words, a system with only one cell will be considered first. Figure 5 is a simplified schematic diagram of the update sub-module. The circuit consists basically of two one-word shift registers SR_1 and SR_2, a selector switch S and an up-down counter TC (tap counter) having values +1, 0 or -1, that controls the position of the selector. During the scan period, words are fed into the update module, and sent to the next module via the selector switch S. In the figure, the selector S is shown in its 'normal' position (0). The positions +1 and W1 are used mainly for word insertion that is explained in the next section.

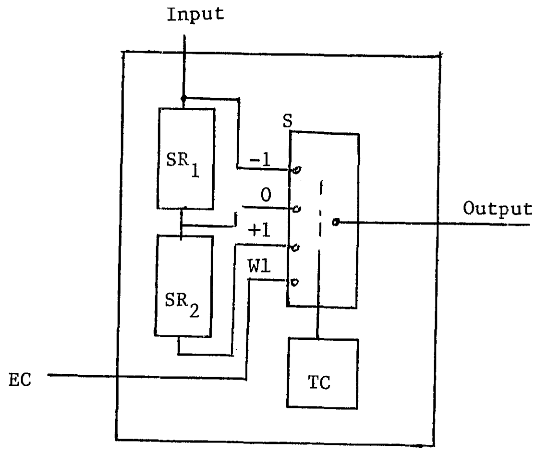


FIGURE 5. UPDATE MODULE

Fig. 6 shows the data stored on the segment memory, where W_1 to W_n is the fixed number of words processed during the scan period and contains the valid data stored on the cell memory, and b_{i-1} along with t_{i+1} are the extra words used for word insertion and garbage collection respectively. Since b_{i-1} is used for word insertion, it will not be considered in the discussion that follows. For a single cell-system, the word t_{i+1} will always contain an end of file word (EOF). In a multiple cell system, the word, t_{i+1} of the bottom cell will be an EOF word.

At the end of the gap, the selector S is always reset to the normal position (0) by initializing TC to 0, so that there is one word delay between the output of the search processor and the output of the garbage collection. If during the scan period, a garbage word is detected, TC is decremented by one, and S moved to the position '-1'; in that way the garbage word is deleted. However, at the end of the scan, the segment will be short of one word, and t_{i+1} will be considered as the last word of the scan. Since t_{i+1} contains an EOF word, the result of deleting the garbage word will effectively shift it to the bottom of the memory, and convert it to an EOF word.

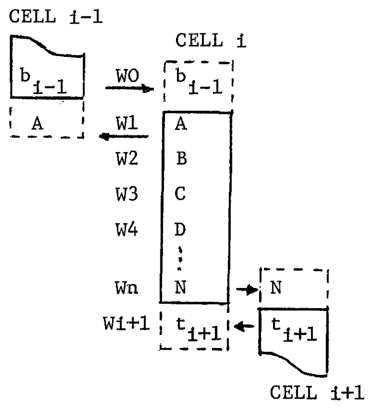


FIGURE 6 DATA IN A CELL

For a multiple segment system, the memory space W_{n+1} in segment i contains a copy of the top valid word (W_1) of the next lower segment i+1. To perform this, the first word W_1 of every segment is stored into a one word length register R at the beginning of each scan, and then the content of each register R_i is written on the previous segment at the end of the scan. The only exception is for the last segment, which always contains an EOF word in W_{n+1}, and the 'next lower cell' to it always loads an EOF word in the register. Figure 7a shows a stable situation for the data in W_{n+1} and the one word registers R_i for the ith cell and the bottom cell of a multiple cell system.

In order to explain how the garbage collection words, consider first a case only one garbage word GB in only one segment, say segment i, and no word input (or other operations) is taking place. Before the garbage word appears, the data in the cells will be stable, and will look as shown in Figure 7a. Later on, a garbage word is created, and Figure 7b is a 'snap-shot' at the beginning of the scan when the garbage word is going to be detected (Scan 1). During this scan, GB will be deleted from the segment i as explained for the case of one segment; the difference now is that the topmost word of segment i+1 has been inserted as the last word of segment i. Figure 7c shows the situation at the end of Scan 1. We see the word W_{n+1} of segment i is now 'empty', and this condition is reflected by TC=-1 for that segment.

The fact that TC=-1 in segment i, will signal when word W1 is SR1 in all cells, all the cells below cell i, to shift one word up during the next memory cycle. To perform this, TC is set to -1 in all the cells below cell i, and is set to 0 in all cells above cell i, and in cell i at the after Scan 1. At the beginning of the next scan (Scan 2), all the cells below cell i will not write word W1 but will write W2 in its place in the CCD memory. Being short a word, they will write

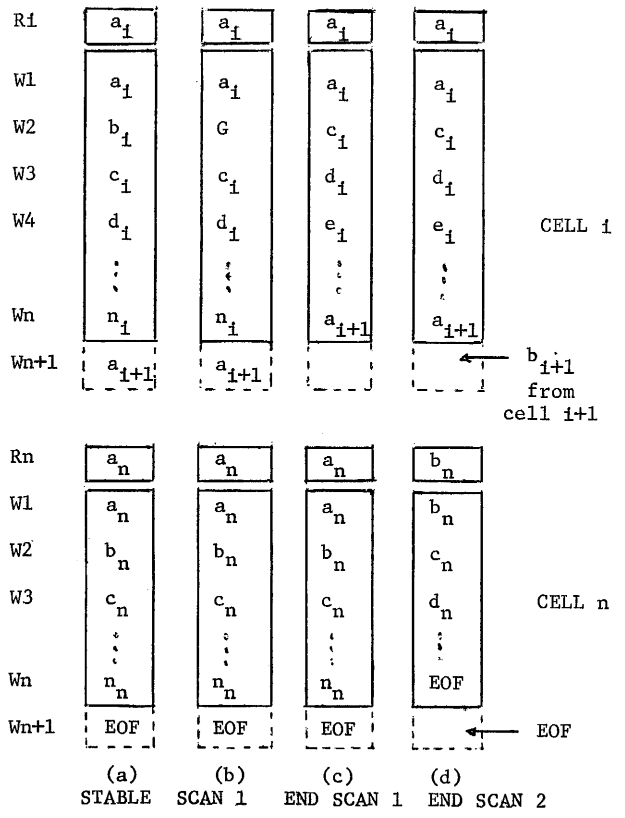


FIGURE 7 GARBAGE WORD DELETION

the word t_{i+1} from register R_{i+1} at the end of the scan. At the end of Scan 2, all the cells below cell i have moved one word upward, and an EOF word is inserted at the bottom of the last segment, see Figure 7d.

Finally, we consider the anomalies encountered when more than one garbage word is found. Two garbage words can be encountered on the same cell. However, when the second garbage word is met, the tap is already in the -1 position, which prevents such a word from being picked up. It will be collected in a later scan. Two garbage words can be encountered in different cells, say cells i,j. This requires the lower cell, j, having a garbage word to send up a garbage word to cell j-1 in lieu of its word W_1, and initialize its TC to 0 instead of -1. The garbage word sent to cell j-1 will be collected in a later scan. Finally, if a garbage word is met when a cell is shifting words up (scan 2 in Figure 7), the TC having been initialized to -1 at the beginning of the scan and not having been changed, the cell will not collect this garbage word until a later scan.

If word insertion is taking place together with garbage collection, it will be possible to delete more than one word per memory cycle. The general case is explained after word insertion.

3.2 Word Insertion

In word insertion, the word B sent by the computer will be inserted right after any word that has X=1 in an IW command. This is carried out in the garbage collection module, within the same circuit used for the deletion of garbage words (see Figure 5).

A given cell will be able to insert a word if a word with X=1 has just been passed and the selector S is at the -1 or 0 position. It should be noted that if the word with X=1 is found on the bottom of one segment, a signal is sent to the next lower segment to enable insertion in its first word.

For the description that follows, it will be considered that only word insertion is taking place (there are no garbage words), and the computer has a word ready to send. The mechanism for insertion within a cell is very similar to that of word deletion, but making use of the +1 and W1 position of the selector.

At the beginning of the scan when the word is to be inserted, the selector S will be at its normal position. When the word with X=1 is detected at SR1, the X bit is erased and a signal is sent to the computer which will feed the new word into the EC line, and the selector S is temporarily set to the W1 position to receive that word. In the meantime, the word in SR1 is shifted into SR2. After the insertion the selector S will be in the +1 position, and no further word insertion may be performed. This mechanism allows the insertion of one word per memory cycle in a cell.

In a multiple cell system, if at the end of a scan period a cell has the condition TC=+1, the segment will have an extra word on it. During the next scan the shifting of one word toward the end of the memory must take place. The mechanism to perform the shifting is very similar to the one used for garbage collection, only that in this case, a word b_i is shifted downward instead of upward.

To handle the word shifting in a multiple cell system, an exact copy of the last valid word (W_n) of the previous segment is stored in register P at the beginning of each segment memory in W_0, as shown in Figure 6 (the only exception is the first segment in which, the data in W_o is irrelevant). This is done by writing the last valid word W_n of each segment into register P_i at the end of each scan period if S=0 for that segment, and then writing the content of the register into W_o, before the beginning of the scan, on each subsequent segment i+1.

If at the end of a scan, cell i has the condition TC=+1, and all the cells below i have the condition TC=0, TC will be initialized to 0 in all segments above i and in segment i, while TC will be intialized to -1 in all cells below cell i at the end of the scan. This causes the word originally in W_o to become the word now in W_1. Note that at the beginning of the scan, W_o is in SR1 while W_1 is in both SR2 and the search processor. Changing TC from -1 to 0, causes the content of W_o to become first word of the scan. The overall effect of the above procedure is the shifting of one word toward the end of the memory in all the cells below cell i. At least one EOF word is needed at the end of the last cell in order not to lose data. If the EOF word does not exist, the system memory is full and the word insertion (or any other input) should be inhibited.

Finally, some anomalies can occur where two words are inserted at the same time. Like the garbage collection anomalies, if both words are in the same cell, or the lower cell is moving a word down, TC will be in the +1 position so that the second word cannot be inserted. It must be inserted in a subsequent scan. The IW command may take several cycles to insert all words in this case. If more than one cell, say cells i and j, have TC=+1 at the end of a scan, the lower cell, u, will still have one too many words in it at the end of the next scan. The extra word not able to be put on the CCD memory is stored in register P_i instead of the last word W_N, so that it will be passed down to the next lower cell in the next cycle.

3.3 Insertion and Deletion of More Than One Word

It was assumed that only one operation was taking place in the description of the garbage collection and word insertion. However, the garbage collection circuit is active during the word insertion operation; which means that both operations may take place during the same memory cycle. The fact that the word insertion and garbage collection are operations that complement each other will allow, when occuring together, to delete or insert more than one word per memory cycle.

On a single track system, insertion can always take place if there is enough room on the shift registers SR1 and SR2 to take up the slack; and conversely, garbage can be collected if deleting a word by moving the switch S does not move it beyond the shift register. This permits more than one word to be inserted or deleted per scan. For instance, if every other word was a delimiter word that has X=1 enabling it for insertion and every other word was tagged as garbage, then all the erased words will be deleted and the word B will be inserted after every delimiter word, all in one scan. Note that the counter TC would oscillate between 0 and +1, for instance, as words are deleted and inserted.

On a multiple track system, the operation during the scan is the same as for the single track system. Note that each cell autonomously can determine whether insertion or deletion is possible, looking only at the counter TC.

During the gap, words W_1 and W_n are exchanged between cells. This requires two identical adder-like carry circuits in place of the priority circuits. The two carries correspond to control signals that cause the word to be shifted upwards in each cell, or downwords. Note that if a cell has too many words, it should shift words downward cell by cell until a cell that is short of a word is encountered. Such a cell terminates the downward shift. Conversely, if a cell has not enough words then it should cause words below it to be shifted up. This should cause a word to be shifted up one cell, cell by cell, until a cell is encountered that has too many words. Such a cell terminates the upward shift.

The logic for multicell operation can be implemented by two chains of AND-OR gates through the chain of cells, or equivalently by 74182 carry lookahead generator chips, that implement two carry chains. "Carries" are propagated from higher to lower cells in both cases. In terms of carry lookahead logic, if SW is 1 if a cell is short a word and EW is 1 if a cell has an extra word, then <u>one</u> carry lookahead has SW input to its generate and $\overline{EW}$ to its propagate and the second has EW input to its generate and $\overline{SW}$ to its propagate. Both propagate carries to lower cells. The carry out of the first one indicates a word is to be shifted upward and a carry out of the second indicates a word is to be shifted down. Except for the anomalies indicated earlier for garbage collection and word insertion. These carries simply initialize the counter TC to -1 if the first carry is 1, or to +1 if the second carry is 1, or to 0 if neither carry is 1. The anomalies can be handled as indicated in the previous sections.

This garbage collection technique can be extended to always either collect n garbage words or insert m words by using a counter TC that can go from +m to -n, coupled to an (m+n) tap shift register in a similar manner. A suitable scheme for transferring m or n words between cells in one step is not known; however it is possible to transfer one word at a time using the scheme shown above. Also, it will be necessary to save the top n words and bottom m words on each track for possible transfer, and some care is required to locate the next word below one with X=1 if this word appears on the top of a segment.

4. Block and End Insertion

Fast block insertion is possible in a multitrack system. However, it may create garbage words. Insertion at the end of the file is the fastest and cleanest method for inserting words. These techniques are described below.

4.1 Block Insertion

Block insertion input allows the insertion of a large amount of words, equal to an integer multiple of the number of words in a cell memory. The new segment is inserted below a word with the X bit set and no more than one word with X=1 is allowed, in the whole system memory, at the beginning of this input mode.

To explain the mechanism used for the segment insertion refer to Figure 8. For simplicity, only a three segment system, with four words per segment is shown, and exactly one segment is going to be inserted (segment H I J K).

At the beginning, before the insertion, the system memory will look like shown in Figure 8a. Only one word exists with X=1 (word b), and there is enough space for the insertion of the new segment (the last half of segment two, and all segment three is filled with EOF words). During the scan for the ML command, the word with X=1 is detected, and at the end of the scan, the cell containing the X=1 word is marked by setting a flip-flop. (Note that at the end of the scan, only one segment is marked since no more than one X=1 word is allowed). At this point, if an IB command is taken, during the scan the cells below, and including the X=1 cell will pass all their data to the cell immediately below (in the example of Figure 8, cell 2 passes its data to cell 3, and cell 1 to cell 2). The cell with X=1, will rewrite its data up to the word with X=1, at which point the X bit is erased and a signal is sent to the computer to synchronously start the transmission of the new data. This data is sent to the X=1 cell, and written on the memory, starting with the word immediately below the X=1 word. This is shown in Figure 8b, which is a 'snap-shot' at the end of that scan. Note that at the end of the scan, the X=1 word has moved to the next cell. In the next scan, the input is continued at the next segment (segment 2 in the example), and when the last word of the inserted segment is written the X=1 word will disappear. This is shown in Figure 8c.

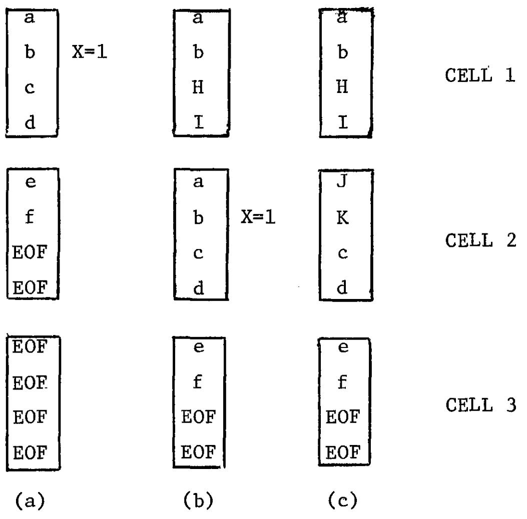


Figure 8. Insertion of Block HIJK.

If the amount of data to be inserted is greater than one segment, the shifting of the data in each cell to the one below, as explained above, should continue on each next memory cycle, and concurrently with the input of data, until there is enough space for all the new data. To handle this, a counter register is initially loaded with n, the amount of words that the computer will send for insertion, as provided by the ML instruction. The counter is decremented as words are input. The general algorithm to decide if the data has to be shifted, starting on the next cycle is: each time a scan has been completed in block insertion, the data will be shifted on the next scan if the content of the counter is greater than the number of words on a segment. The shifting of all data from each cell to the next lower cell will be done in all cells below the one that, at the end of the scan, had just inserted a segment, and the next lower cell will insert words as explained for the one segment insertion.

If a block is to be input which is not a multiple of the number of words in a segment, the remaining words needed to fill out a segment are input as garbage. Garbage collection hardware must shift these garbage words to the end of the file.

Finally, we note some abnormal conditions that have to be considered. Before starting the data shifting, the availability of space is tested by looking if the first to last cell has at least one end of file word, and if this is not the case, the system memory will be considered full for block insertion, and the input should stop. Also, garbage collection is always inhibited during block insertion since the deletion and shifting of words may disturb the segment insertion.

4.2 Input at the End of the File

The input at the end of the file is handled by the input/output module and it is the normal input mode of the system. In this mode the words will be transferred from the computer and written at the end of the existing file, starting with the first EOF (end of file) word found. To handle this input, the system keeps track of the cells with EOF words on it, and at the end of each scan the cells are marked by setting a flip-flop.

A priority circuit determines the topmost cell with EOF words on it, and an input will start at this track and with the first EOF word in the track. Files are first created in the system using this mode of input.

5. Output

The output of words to the computer is handled by the input/output module. This module may be implemented in such a way that it is always active (like the garbage collection) since there is no conflict with other operations in the system. In that way, words marked for output (words with the C bit set) will be sent to the computer as soon as they are detected. Output is controlled by output modes provided by the OR command. If mode m_1 indicates SAVE, when a word is output, the C bit is cleared. If mode m_1 indicates DELETE, when a word is output, it is changed to a garbage word.

The mode m_2 is used to assign the output priority to the different cells that have words with the C bit set. In the ordered mode, only one cell can output data in each cycle, starting with the topmost one with a word with C=1 in the first scan, and continuing with the lower cells in an ordered fashion on each subsequent scan. In that way the data is sent to the computer in the same order as it is on the file. A flag stores the fact that words with C=1 are still in a cell. The priority circuit gives access to the topmost cell having this flag set. In each cycle, the memory is evaluated to find the topmost cell with collection words in it; this cell outputs its collection words in the next cycle. If a cell has the output access, and the computer is not ready by the time a word with C=1 is found, the output will be held until the end of that scan (even if the computer signals ready before the end), and resumed on the same cell on the next scan in order to preserve the order of the wors.

In the delimited mode, the order of the words is kept within each record, but the records are sent to the computer in random. In this mode, all the cells scan in parallel, searching for a delimiter with C=1. As soon as one or more cells find the cells that this condition the priority network locates the highest such cell. It sets a flag and gains access to the computer until the next delimiter word with C clear is found, at which time the scan for another delimiter with C=1 is resumed in parallel in all the cells. However, if at the end of a scan, a given cell has the output control, meaning that a record starting in one cell and continuing to the next one is being sent out, then the output access will be passed to the next cell for the next cycle in order to keep the order within the record.

In the random mode, all the cells are searched in parallel. As soon as one or more cells find a collection word, the priority circuit gives access, temporarily, to the topmost one, which outputs the word at this time. After the word is output, the search is resumed in parallel in all the cells. The same-procedure is repeated on each scan. This mode of output is the fastest, but the output is scrambled.

In the unique mode, the output of data of duplicate words is prevented. In this mode, the cells will search in the ordered mode for words with C=1, but each time the cell with access to the computer finds just one word with C=1, it outputs only that one word. During the gap, this word is broadcast to all the cells in order to search for duplicate words within the ones with C=1. During the subsequent scan, the C bit erased if a match is found. During the same scan, after the C bit is possibly cleared, the cells search in the ordered mode for just one more word C=1, continuing as above. This mode allows the output of only one word per cycle, sent to the computer and broadcasted to all the cells at the end of the scan.

6. Conclusions

A collection of techniques for hardware management of a secondary memory have been described. Garbage collection and input and output techniques have been shown to be simple, even on a multiple track (segment sequential) memory.

The techniques for managing secondary memory, the main content of this paper, were presented in the context of a very simple architecture. They can be embedded in more complex architectures, such as the CASSM architecture [3]. However, this simple architecture itself offers a secondary memory capable of storing arbitrary length records, accessing them without need for a directory, linking separate records automatically, spooling records so that they can be queued and output on demand, and collecting garbage words. The absence of a directory often saves one access time. In small systems, the directory need not be brought into the small primary memory. In networks, no protocols are needed to gain access to the directory. Finally, essentially no software is needed to manage the directory or collect garbage. A self-managing secondary memory that uses techniques such as those suggested in this paper could be a new and useful type of secondary memory.

References

[1] Healy, L. D., Lipovski, G. J. and Doty, K. L., "The Architecture of a Context Addressed Segment-Sequential Storage", Proc. FJCC, pp. 691-701, 1972.

[2] Copeland, G. P., Lipovski, G. J. and Su, S. Y. W., "The Architecture of CASSM: A Cellular System for Non-Numeric Processing," Proc. of the First Annual Symposium on Computer Architecture, pp. 121-128, Dec., 1973.

[3] Su, S. Y. W., Lipovski, G. J., "CASSM: A Cellular System for Large Data Bases," Proceedings of the International Conference on Very Large Data Bases, Farmingham, Mass., September, 1975, pp. 456-472.

[4] Bush, J. A., Lipovski, G. J., Watson, J. K., and Su, S. Y. W., "Some Techniques for Implementation of Segment Sequential Functions." These proceedings

[5] Ozkarahan, E. A., Schuster, S. A., Smithe, K. C., "A Data Base Processor." Technical Report CSRG-43, November, 1974.

[6] Walden, David C., "A system for Interprocess Communication in a Resource Sharing Computer Network", Comm. ACM., April, 1972, Vol. 15, No. 4, pp. 221-230.

[7] Feustel, E. A., "On the Advantages of Tagged Architecture," IEEE Trans. Comp., July, 1973, pp. 644-656.

PRICE/PERFORMANCE COMPARISON OF C.MMP AND THE PDP-10*

Samuel H. Fuller
Departments of Computer Science and Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

The analysis in this paper shows a multiprocessor like C.mmp to have a factor of three to four cost/performance advantage over uniprocessor systems such as the PDP-10 when implementations using similar technologies are considered. This comparison is shown to be very sensitive to memory prices and considerable attention is given to normalizing memory costs between C.mmp and the PDP-10.

An important part of this analysis is a comparison of the PDP-10 architecture with the PDP-11 architecture (i.e. the architecture of the processors of C.mmp). When the limited address space of the PDP-11 is not a problem, we see that to a close approximation it takes the same number of PDP-11 instructions (average length 25 bits) as PDP-10 instructions (length 36 bits) to represent a program.

While the comparison in this paper explicitly considers multiprocessor degradation factors such as memory interference, it does not address the problem of writing software systems capable of taking full advantage of the multiprocessor structures. The comparisons in this paper are primarily ofcused on comparing the hardware structures of uniprocessors and multiprocessors. Work is now in progress at CMU that is attempting to evaluate the effectiveness of both individual multiprocessor structures application programs and multiprogrammed systems operating on C.mmp.

CONTENTS

1. INTRODUCTION

During the first half of the 1970's a surprising number of computer systems designed as an interconnected set of smaller computers have been proposed and a nontrivial number of these systems have been built. Figure 1.1 [Baskett, 1975] helps to explain this spurt of interest in systems built with several small processors rather than a single larger processor. This figure shows the cost effectiveness of all the computers listed in Computer Review [1975], measured in Processor-Memory Bandwidth/Dollar, as a function of the price of the smallest configuration. Note that the $10,000 systems (i.e. minicomputers) appear to be at least an

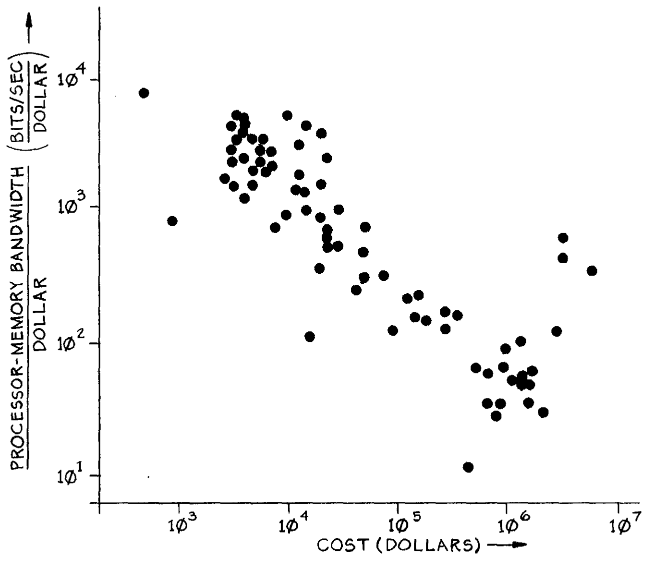


Figure 1.1. Cost/Performance as a Function of System Cost

order of magnitude more cost-effective than the larger machines. There a number of apparent reasons for this phenomenon:

1. Continuing advances in semiconductor technology favor the small processor. LSI (Large-Scale-Integration) memory and ALU chips have been able to dramatically cut the cost of producing minicomputers. Recent LSI advances such as the Intel 3000 bit-slice processing element [Intel, 1974] and the DEC LSI-11 will continue to drive down the price of minicomputers. The larger processors that rely on specialized logic to speed up ALU functions, prefetch and buffer instructions, overlap instruction execution, etc. are currently less able to exploit the present LSI technology.

2. Economies of scale. A production line that produces on the order of 10,000 minicomputers a year (or 10^5 to 10^6 microcomputers a year) will not have the overhead per computer that a production line has that produces 50 to 100 (large uniprocessor) computer systems a year.

3. Pricing policies that bury the cost of software development for the large computer systems in the price of the hardware.

See [Bell et al., 1971] for another view of the reasons for the emergence of multi-mini-processors.

The purpose of this paper is to try to take a more detailed, concrete look at the cost effectiveness of computer systems built from multiple, mini-processors.

*This work was supported in part by the Advanced Research Projects Agency under contract F44620-73-C-0074 which is monitored by the Air Force Office of Scientific Research and in part by National Science Foundation Grant GK-41070.

Specifically, we compare C.mmp*, the multi-mini-processor computer system that has been developed at CMU, with a standard, uniprocessor computer system: the PDP-10. Discussion of the details of C.mmp [Wulf, et al., 1975, Wulf and Bell, 1971] and the PDP-10 [DEC, 1971] are not within the scope of this paper. The PDP-10 is a conventional uni-Pc computer system and C.mmp is structured as a canonical multiprocessor computer system. It consists of up to 16 equal, asynchronous Pc's that share a large Mp.

Manufacturer's specification sheets are vague at best and often (intentionally?) ambiguous. For this reason, whenever feasible we are collecting data rather than rely on published information. The uniprocessor that is used in this comparison in the PDP-10 (KA10 processor). We are using the PDP-10 not because it is necessarily the best example of a uniprocessor system but because we have two at CMU that are readily available for us to measure and several problems exist that have been programmed for both the PDP-10 and C.mmp. Ideally, we will be able to expand this work in the near future to include processors of other manufacturers.

The purpose of this report is limited to a comparison of the price/performance of multi-Pc systems to uni-Pc systems. For this reason we are limiting the scope of this work to the central processors (Pc's), primary memory (Mp), and I/O channels (Kio's) or I/O processors (Pio's) of the computer system. Secondary storage units, terminals, and communication subsystems are excluded not because they are insignificant in cost or performance but because they are common to multi-Pc and uni-Pc systems and their structure is not directly affected by the fact there are one or many Pc's. On the other hand, both Mp and Kio's (or Pio's) often need to be structured differently to interface with a uni-Pc or a multi-Pc processing element.

A recent budget survey [McLaughlin, 1974] helps to put the scope of this report in perspective. Of the 194 computation centers polled, only 39% of the budget was used for hardware purchases or leases; the bulk of the budget went to salaries and overhead. Of the money spent for hardware, about half was spent for central processors and main memory. The other half of the hardware money was spent on secondary storage, communication costs, terminals, and unit record devices.

In spite of the fact that the Pc/Pio/Mp subsystem of a conventional computing facility only comprises about one fourth the total operating costs, the assertion that a multi-mini-Pc system is more "cost-effective" than a single, larger mini-Pc system has been argued for several years now. The most ovvious approach to addressing this issue is to set aside costs common to both multi- and mini-Pc systems and examine the cost-effectiveness of changes in the computer's structures introduced by adding multiple Pc's.

2. MEASURES OF PRICE AND PERFORMANCE

2.1 Simple Performance Parameters

The initial measures that we use in this paper for performance are:

- Instructions per second (units: 10^6 instructions/sec. Denoted as MIPS, Millions of Instructions Per Second)
- Processor-Memory Bandwidth (units: 10^6 bits/sec.)

*We use the PMS (Processor-Memory-Switch) notation of Bell and Newell [1971] to describe computer systems at the "block diagram" level. C.mmp is a PMS acronym for a multi-mini-processor Computer system. Other frequently used PMS names include Pc for central Processor and Mp for primary Memory.

The above two measures of Pc performance are less than ideal. However, they both have obvious intuitive meanings, and they can be directly measured on operational systems. The following comments may help in our interpretation of these measures:

2.1.1 The MIPS measure unfairly favors the small, primitive machine. Clearly a simple 16 bits/word minicomputer executing 1 MIPS is not as powerful as a 1 MIPS 36 bits/word computer. The individual operations being evoked on the minicomputer operate on less than half the number of bits than the larger word computer. ← In addition, Pc's that incorporate such features as multiple general purpose registers, vector (or block move) instructions, or a rich set of data types (e.g. the IBM 360 and 370 architecture) will have lower MIPS rates because of these features, but, in fact, are more powerful because of these features.

2.1.2 The Processor-Memory Bandwidth measure tends to give an unfair advantage to large word Pc's relative to the smaller word minicomputers. A Pc that gets 64 bits/fetch (e.g. the IBM 370/168) will often not use all 64 bits. In an extreme case it only wants a byte of information and the remaining seven bytes are simply discarded. On the other hand, a minicomputer that only accesses 16 bits/fetch will not be as inefficient in its use of memory bandwidth. In fact, any low cost implementation of a processor (e.g. the IBM 360/30) will make more efficient use of its Processor Memory Bandwidth than a larger Pc.

The above problems with MIPS and Memory Bandwidth suggest that we might do well to use both in any comparison of minicomputers to larger computers and at least as a first order approximation use MIPS and Processor Memory Bandwidth as upper and lower bounds (respectively) of the power of the mini-Pc relative to the larger Pc.

2.2 Benchmarks

Because of these problems with MIPS and Processor-Memory Bandwidth, we will also use one other measure of Pc performance: benchmark (or kernel) programs. In other words, measure the execution time of the same problems on the various machines of interest. The main problems with this approach are that the results are often very problem specific -- and hence the need for many benchmarks -- and the fact it is very time-consuming to recode a set of given problems on the different machines of interest. However, there is sufficient question with the accuracy of the MIPS and Processor Memory Bandwidth measures that several benchmarks have been developed for use in the comparison of C.mmp to uniprocessors.

2.3 Mp Capacity

The price of memory is a significant factor in the prices we will discuss below and yet none of the measures of Pc performance are influenced by Mp size. Clearly a 10 MIPS Pc with 4K words of memory will have a better price/performance figure than a 5 MIPS Pc with 3,000K words of Mp, but it is also just as clear the 5 MIPS-3 Megaword system is the more powerful system. As the MIPS rate of a Pc is increased, it will need more Mp in order to have sufficient data to keep it busy. This phenonema is captured by a piece of computer science folklore known as "Amdahl's Constant". It states that a balanced computer system needs 1 Megabyte of Mp per Pc MIPS. Rather than argue the accuracy of Amdahl's constant, in the following analysis we will indicate both the performance of the Pc and the size of Mp in the systems we compare. When comparing a uni-Pc to a multi-Pc system we will vary the size of Mp over the range of practical configurations.

2.4 I/O Bandwidth

Like Mp capacity, I/O bandwidth is not included in any of our performance measures, yet a system is in danger of being starved for data if the I/O channels have insufficient capacity. As with Mp capacity, we will simply state the I/O transfer rate capacity of the various systems and not try to develop formulas to integrate I/O bandwidth into measures of Pc performance. (Aside: Another, lesser known "Amdahl constant" is that 1 bit of I/O is needed per instruction executed [Amdahl, 1970]. However, we won't pursue this idea any further in this report.)

2.5 Measures of Price

Our basic measure of price will be March, 1975 retail prices. This measure has the attractive property that it spans different Pc architectures, Mp configurations, and I/O channel structures rather cleanly. However, we still must contend with the following troublesome details that conspire to blur our comparison.

2.5.1 The year a computer system is implemented influences its cost-effectiveness as expressed in current retail prices. For example, there is no question that you receive more power (MIPS, Processor Memory Bandwidth, ...) per dollar today from a KL10 -- a PDP-10 implemented in 1974 -- than from a KA10 -- a PDP-10 implemented in 1966. In fact, the primary reason for reimplementing a given instruction set every two to five years is to get the improved price/performance available with the newer technology.

2.5.2 Marketing strategies will distort prices somewhat in order to hide software costs, encourage user acceptance, etc. Hence manufacturers' prices cannot be assumed to give too accurate a measure of the fundamental cost of implementing the system.

2.5.3 While we use retail prices whenever possible, some of the components of C.mmp are not commercially available and the only solid dollar figures we have are construction costs at CMU. As with the performance measures, rather than attempt to construct and justify an appropriate CMU cost to manufacturers' retail price coefficient, we will simply indicate which figures are CMU cost figures and which are retail prices.

3. COMPARISON OF THE PDP-10 AND PDP-11 PROCESSORS

C.mmp is implemented with PDP-11 Pc's as the central processing elements. Hence if we are to make any meaningful comparison between the PDP-10 and C.mmp, we need some measure of the relative power of these two instruction sets. In addition, we need both absolute and relative measures of the execution rates of the various implementation models of the PDP-10 and PDP-11 architectures.

To assist in this comparison, four benchmarks were used:

PDE. This is a classic partial differential equation solver that uses Liebmann's iteration method (i.e. simple relaxation). Two's complement integer arithmetic is used and 16 bits of precision is assumed to be sufficient. PDE has been implemented in BLISS on both the PDP-10 and PDP-11. On both machines we have an unoptimized and an optimized version. The inner loops of the optimized versions are written in assembly language.

L*. L* is an interpretative list processing system and the L* benchmark consists of a set of small programs that exercises the stack, list, and arithmetic facilities of L*. It is written in assembly language on both the PDP-10 and PDP-11.

TECH. This is a chess playing program and is intended to represent a typical application in artificial intelligence. To a first approximation, TECH is simply a tree-searching program that derives most of its power from alpha-beta pruning. Both PDP-10 and PDP-11 versions are written in BLISS.

Integer Programming. A modified branch-and-bound procedure for linear integer programming problems. Both the PDP-10 and PDP-11 versions are written in BLISS.

Table 3.1 gives the static comparison between the PDP-10 and PDP-11 for these benchmarks. A number of interesting observations can be made from Table 3.1, but the most significant are:

1. The ratio of PDP-11 instructions to PDP-10 instructions needed to implement this set of benchmarks is nearly unity.

2. The PDP-11 is able to represent these programs with 0.665 (about 2/3 the number of bits required by the PDP-10.

3. The average number of 16 bit words needed per PDP-11 instruction for this set of benchmarks is 1.62

	PDP-10	PDP-11			$\frac{\text{PDP-11}}{\text{PDP-10}}$ ratios		
	instr	instr	words	$\frac{\text{words}}{\text{instr}}$	instr	words	bits
PDE	267	269	437	1.62	1.01	1.64	0.728
L*	120	109	186	1.71	0.910	1.55	0.689
TECH	2378	2429	3829	1.58	1.03	1.61	0.716
Integer Prog.	744	560	882	1.57	0.64	1.18	0.527
Average	-	-	-	1.62	0.90	1.50	0.665

Table 3.1. Benchmark Program Sizes

Tables 3.2, 3.3, and 3.4 give information needed for a dynamic comparison of the PDP-10, specifically the KA10, and the various PDP-11 models. Table 3.2 gives the MIPS and Processor Memory Bandwidth for the KA10. The most significant figure in Table 3.2 is the MIPS rate observed when the KA10 was compute bound running a general mix of programs (over $5*10^7$ instructions were counted). The fascinating aspect of this MIPS measurement for the KA10 is the observed degree of variability. For example, for the MIPS reading averaged over 1 sec., the mean is 0.342, but the standard deviation of 0.190; for the 10 sec. readings the standard was still 0.075. Therefore, a reasonable confidence interval (95%) surrounding our mean reading of 0.342 MIPS is (0.327, 0.357).

Reading averaged over 10 second intervals (14 observations)

Max: 0.480 MIPS
Min: 0.224 MIPS

Reading averaged over 1 second intervals (20 observations)

Max: 0.515 MIPS
Min: 0.138 MIPS

The MIPS and Processor Memory Bandwidth is given for the benchmarks to indicate how closely they match the actual averages seen by the general purpose programs.

Benchmark	inst. x 10^3/sec.	words x 10^3/sec
General Use	342	(data not available)
PDE (optimized)	332	484
L*	289	491
[Lunde, 1974]	312	(data not available)

Table 3.2. PDP-10 (KA10) Execution Rates

Table 3.3 gives the execution rates for the PDP-11/20 and PDP-11/40. The principle conclusions are:

3.4. MIPS: 11/20 = 0.186; 11/40 = 0.34 (estimate)
3.5. Pc-Mp Bandwidth (10^6 words/sec.): 11/20 = 0.470; 11/40 = .870 (estimate)

	C.mmp Pc's[1]			
	PDP-11/20		PDP-11/40	
Benchmark	inst. x 10^3/sec	words x 10^3/sec		
Job monitor	201	446	(2)	(2)
PDE (Optimized)	210	455		
L*	155	508	(2)	(2)
C.mmp Avg.	186	470	(2)	(2)
Relative Speed [O'Loughlin, 1975]	-	1.0	-	1.85

[1]PDP-11/20's and PDP-11/40's are slightly slower on C.mmp than in standalone configurations because of delays in the crosspoint switch.

[2]At the time these measurements were conducted no PDP-11/40 was operational on C.mmp and these figures for the PDP-11/40 are estimates based on PDP-11/20 to PDP-11/40 measurements reported by O'Loughlin [1975].

Table 3.3

Table 3.4 completes this dynamic comparison of the PDP-10 to the PDP-11 by showing the total execution times for the benchmark programs. In fact, it is possible to estimate the PDP-11/20 to KA10 execution time ratio from the already discussed ratios of PDP-11 to PDP-10 instructions and PDP-11 to PDP-10 MIPS. Whichever way it is computed, directly from Table 3.4 or indirectly from program sizes and MIPS rates, we find a ratio of throughput (benchmarks/second) of KA10 to PDP-11/20 throughput of 2.16 and a KA10 to PDP-11/40 throughput ratio of 1.17 (estimate) -- not far from unity.

Benchmark	KA10	PDP-11/20	$\frac{11/20}{KA10}$
PDE	124.7	186.4	1.49
L*	66	195	2.95
Integer Programming	30.0	61.4	2.04
Average	-	-	2.16

Table 3.4. Benchmark Timings

Before continuing, a few comments are needed on this comparison of the PDP-10 to PDP-11 instruction sets. Most importantly, the comparisons in this section have not adequately accounted for the differences in data-types between the PDP-10 and PDP-11. The PDP-11 only has 16 bit integers and when we consider solving problems often encountered on the PDP-10 (and hence C.mmp) 32 or 36 bit integers will be required. The current PDP-11 instruction set will force us to emulate the manipulation of large integers via software routines. Another factor missing from this comparison is floating point numbers. Although the PDP-11/40 has a floating point option, it has floating point add (and subtract) times of 20 μsec., a floating multiply time of 20 μsec., and a floating divide time of 47 μsec. In contrast, the KA10 has much faster floating point operations: add (subtract): 5 μsec.; multiply: 11 μsec.; and divide: 14 μsec. Hence, the PDP-11/40 has an execution rate very close to the KA10 for the basic operations, but is a factor of 3 to 4 slower for floating point operations. This shortcoming of the PDP-11/40 needs to be studied in further evaluations of C.mmp. A factor that tends to downplay the significance of this floating point comparison is that instruction mixes of large, conventional Pc's show that floating point instructions rarely exceed 10% of the instruction mix, even for scientific computations [Stone, 1975, p. 540]. The PDP-11/45 executes floating point operations at about the same rate as the KA10. In addition, the PDP-11/45 has both 32 and 64 bit floating point data-types; comparable to the 36 and 72 bit floating point data-types of the PDP-10.

Another important factor in comparing the PDP-11 to the PDP-10 is that the PDP-11 has a much smaller address space than the PDP-10; in fact, the PDP-11's 64K byte address space is less than 1/16th the address space of the PDP-10 (and 1/356th the 16M byte address space of the IBM 360-370). When we use the PDP-11 processor in a multi-mini-configuration we can expect a larger overhead to establish and maintain addressability than we have historically experienced with the PDP-10 or other conventional processors such as the IBM 360 and 370 Pc's. The PDP-11's poor suitability for applications heavily oriented toward large integers or large address spaces suggests that the above measurements comparing the 11 to the 10's instruction set be viewed with caution.

This addressing space problem for the PDP-11 (and minicomputers in general) has not been solved with the memory mapping units of either the PDP-11/40, 11/45, 11/70, or C.mmp. In these cases the physical memory can be substantially larger than 64K bytes, but the immediately accessible address space remains 64K bytes: explicit loading of the memory mapping registers is required to give the Pc addressability outside its "immediate virtual address space" of 64K bytes.

4. C.mmp MULTIPROCESSOR OVERHEADS

In the last section we attempted to establish a quantitative relation between the processing power of the KA10 and the PDP-11/20 and 11/40 (the Pc's of C.mmp).

In this section we examine factors that affect performance as we connect the PDP-11 Pc's into a multiprocessor configuration.

4.1 Memory Interference

A number of studies have been conducted to determine the amount of performance degradation that results when the Pc's of a multiprocessor contend for primary memory [Strecker, 1970; Bhandarkar, 1074; Bhandarkar and Fuller, 1974; Baskett and Smith, 1975]. Although the mathematical techniques used have differed among the studies, they give remarkably consistent results for the set of configurations that include actual and proposed C.mmp configurations: they all show degradation factors of less than 10%. The fundamental reason is that the Mp ports of C.mmp have a much higher bandwidth (2.5 Megawords/sec.) than either the PDP-11/20 (0.47 Megawords/sec.) or the PDP-11/40 (0.87 Megawords/sec.). Figures 4.1-4.2 show the Processor Memory Bandwidth for C.mmp from a number of aspects. Figure 4.1 shows the bandwidth as the number of Mp ports is varied from 1 to 16. The five PDP-11/20's, variable number of 11/40's, is included since the actual C.mmp system at CMU now has five 11/20's and two 11/40's and will soon grow to a full 16 Pc system by adding nine more PDP-11/40's. The dotted lines show performance with no memory interference and no cache memories, the solid lines show the expected interference, and the lines with circles are the expected performance with 1024 word cache memories that are being designed for the PDP-11/40's. These caches will only buffer read-only pages and the cache bit ratio is estimated to be 0.5.

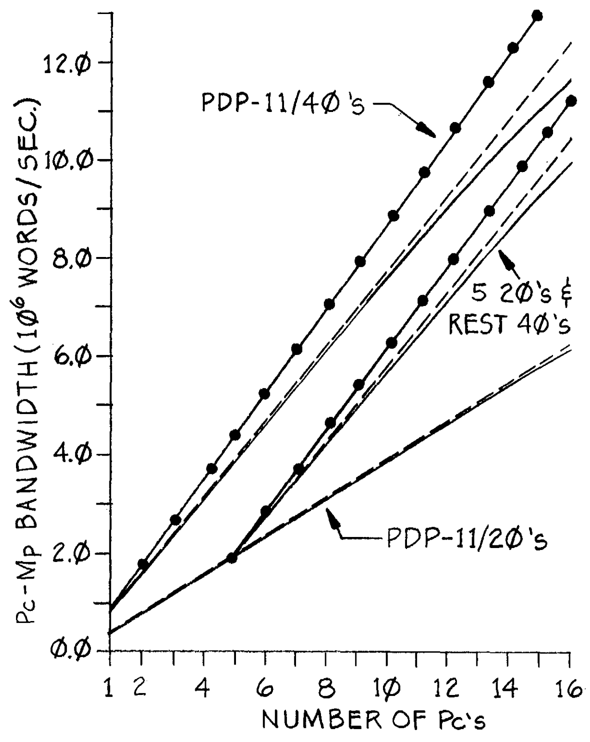


Figure 4.2. Effective Pc to Mp Bandwidth
Dotline: Maximum Bandwidth of Pc's

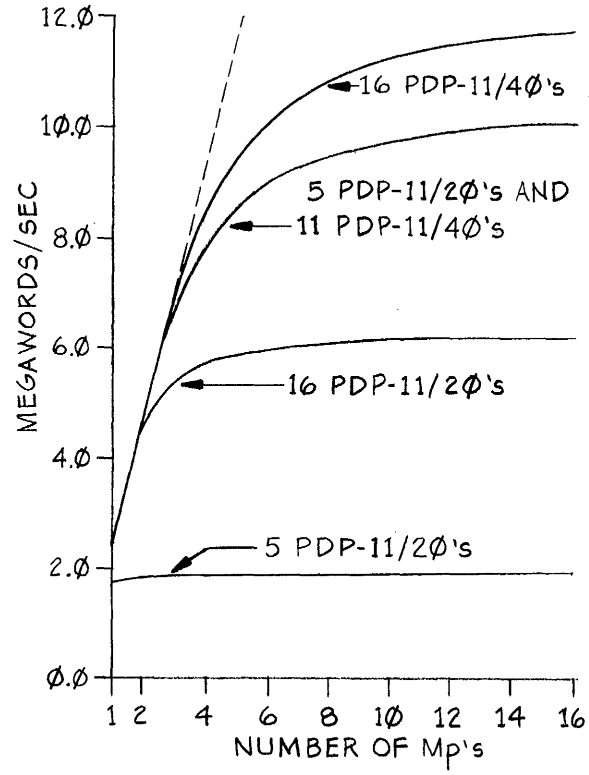


Figure 4.1. Effective Pc to Mp Bandwidth (no caches)

Dotline: Maximum Bandwidth of Mp's

4.2 Software Overheads

At this point in the development of C.mmp and its operating system, Hydra, we have very little information on the extent of software overheads that must be incurred because we must coordinate the execution of cooperating, asynchronous, parallel processes. What little hard data we do have is given in Figures 4.3 and 4.4. Figure 4.3 shows the execution rate of the PDE benchmark as a function of the number of available Pc's on C.mmp. This figure exhibits the promising property that we get very nearly linear speed up as the number of Pc's is increased. In fact, the PDE benchmark is so easily decomposed it would have been a bit surprising, given we know memory interference is negligible for five 11/20's, if we had seen less than linear speed up.

Figure 4.4 is probably a bit more useful. It shows the execution time of the PDE benchmark as a function of the number of processes the work has been divided among. Also on the graph is the time the PDE benchmark takes on a standalone PDP-11/20. We can see that for this benchmark C.mmp switching overheads and Hydra have induced an overhead of 25% when compared with the PDE benchmark run on a standalone PDP-11/20. Figure 4.4 also illustrates the speed up in the PDE benchmark you would expect as the benchmark is decomposed into successively parallel processes in order to take advantage of the multi-Pc's. Another useful observation from Figure 4.4 is the slight upturn in execution time as the number of processes becomes very large. This is simply a measure of the overhead incurred as Hydra is required to manage more processes than can be usefully dispatched. The encouraging fact is that the incremental overhead in adding these processes is as small as it is.

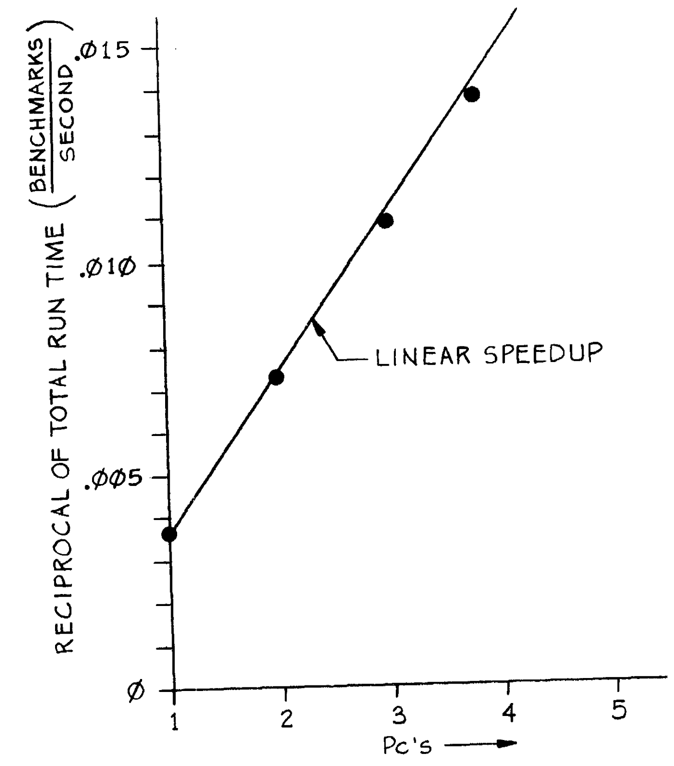


Figure 4.3. Execution Rates for PDE Benchmark (18 process versions of PDE used) (Unoptimized PDE version)

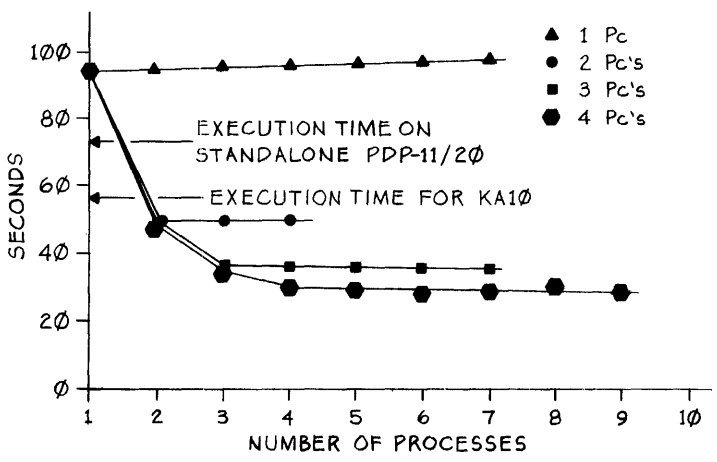


Figure 4.4. Running Times for PDE Benchmark
20x10 data array
10^4 iterations
16-bit fixed point data types
inner loop coded for speed
supervisory program p'ed on a semaphore

5. PRICES

5.1 PDP-10's

Central Processors

KA10 Pc (first delivery: 9/67):	$130K
KI10 Pc (first delivery: 5/72):	$200K
KL10 Pc (first delivery: 6/75 (est.))	$250K

(Measurements reported in this report are only for the KA10; this is the only PDP-10 Pc available at CMU. For measures of KI10 and KL10 performance we must depart from our objective of using actual measurements rather than manufacturers' published data and use DEC's published ranking of a KI10 being 2.0 times a KA10 and a KL10 being approximately two times a KI10.)

Primary Memory

128K word, 4 port Mp module for either KA10 or KI10:	$110K
256K word, 8 port Mp module for KL10:	$180K

Data Channels (DF10)

DF10 (max. transfer rate: 10^6 words/sec.):	$ 14K

5.2 PDP-11's

Central Processors

PDP-11/20 (first delivery: 4/70): (This is a 7/72 price. PDP-11/20 no longer offered by DEC. This price includes 4K words of Mp.)	$ 9.95K
PDP-11/40 (first delivery: ?/72): (This price includes 8K words of Mp.)	$ 12K

Primary Memory

16K word Mp module (with parity):	$ 5.95K

5.3 C.mmp Specific Hardware (These are CMU costs rather than DEC prices.)

16 Pc by 16 Mp Crosspoint Switch: (First, and at present only, 16x16 switch cost $100K.)	$ 50K
8K word Mp module (from Ampex):	$ 1.3K

Processor Modifications to adapt PDP-11 Pc to C.mmp

PDP-11/20:	$ 4K
PDP-11/40: (Includes 1024 word cache.)	$ 5K

5.4 Primary Memory Prices

In reasonably large quantities, it should be expected that the price per bit of Mp should be independent of whether the memory is attached to a uniprocessor or a multiprocessor and whether it is in 16 bits/word or 36 bits/word configurations. However, the prices given above are a bit at variance with this expectation:

KA10, KI10 Mp:	$110K/128K words = 2.38¢/bit
KL10 Mp:	$180K/256K words = 1.95¢/bit
PDP-11 Mp:	$5.95K/16K words = 2.32¢/bit
C.mmp Mp:	$1.3K/8K words = 1.02¢/bit

Note that the C.mmp memory comes out significantly cheaper than either the PDP-10 or PDP-11 memory from Digital. There are two reasons for this: (1) the C.mmp memory does not include the cost of the switch while the KL10 memory has an 8 port switch included and (2) add-on Mp manufacturers sell Mp modules at a lower price/bit than a mainframe manufacturer such as Digital. If we include the $50K cost of the C.mmp switch in the cost of a 1 Megaword Mp constructed from the Ampex memory modules, we get:

[128 ($1.3K) + $50K]/ Megawords = 1.35¢/bit.

The difference between 1.35¢/bit for C.mmp and 1.95¢/bit for the KL10 can be accounted for by the facts we are using the cost of the C.mmp switch rather than attempting to estimate its fair market price and the core memory itself is coming from Ampex rather than the mainframe manufacturer. In order to better understand the effect Mp prices have on our latter cost/performance evaluation, let us postulate a 1.35¢/bit Mp for the KL10. This gives us a price of $124K rather than $180K for 256K words of memory.

6. COST/PERFORMANCE COMPARISONS AND SUMMARY

It should be clear from the previous sections that while C.mmp is able to utilize most of the computing power from the PDP-11 Pc's, it cannot be justified solely on the absolute computing power it provides. A 16 PDP-11/40 C.mmp has a Pc/Mp Bandwidth of 241 x 10^6 bits/sec. while the CDC 7600 has a Pc/Mp Bandwidth of 2180 x 10^6 bits/sec. and the IBM 360/195 has a bandwidth of 1185 x 10^6 bits/sec. [Computer Review, 1975]. The primary justification for constructing multi-mini-processors such as C.mmp stems from their cost/performance advantages over conventional uniprocessor systems.

Table 6.1 shows the performance, price, and several price/performance measures for various representative C.mmp and PDP-10 configurations. These figures are directly based on our discussions in the previous sections. A prominent factor in the price of the Pc/Mp/Kio computer subsystem is the cost of the Mp. However, none of the measures of performance we have discussed here includes the amount of Mp in the system. Hence column 2 in Table 6.1 shows the amount of Mp assumed and we include a Mp-Pc "balance index" to give a rough guide as to the amount of Mp relative to the computing power that is provided.

The cost of a bit of Mp should not be a function of whether it is used in a multi-mini-processor or whether it is used in a uniprocessor system. However, for non-technical reasons current PDP-10 Mp is priced at 1.96¢/bit while C.mmp Mp costs 1.35¢/bit. Therefore, two hypothetical PDP-10's are included in Table 6.1 that assume Mp is available for the PDP-10 at 1.35¢ per bit.

Since the price/performance comparisons of Table 6.1 are most sensitive to Mp configurations and prices, Figure 6.1 shows the price/performance of C.mmp and the various PDP-10 systems as a function of Mp size and price. Note that as the price of Mp is increased, the fact that there is a single Pc or a multi-Pc becomes irrelevant; the price of the system is determined by the size of Mp and the performance by the Pc MIPs. As Mp prices decrease, the cost of the Pc dominates and now we see that C.mmp becomes a factor of 4 more cost effective than the most cost-effective PDP-10.

CONFIGURATIONS			PRICES	PERFORMANCE				PRICE/PERFORMANCE	
Pc	Mp	Kio	$ x 10^3	inst./sec. (MIPS)	Avg. Pc-Mp Bandwidth (Megabits / sec.)	Max. I/O Bandwidth (Megabits / sec.)	Mp/Pc Balance Index (Megabits Mp / MIPS)	instructions/sec. / dollar	Pc-Mp bits/sec / dollar
Standard PDP-10's									
KA10	128Kw	2DF10's	130+110+28=268	.342	.498x36=17.9	72	13.5	1.27	66.8
KI10	256Kw	2DF10's	200+220+28=448	.684	35.9	72	13.5	1.52	80.1
KL10	256Kw	2Df10's	250+180+28=458	1.37	71.8	72	6.74	2.99	157
PDP-10's with 1.35¢/bit Mp									
KI10	256Kw	2DF10's	200+125+28=353	.684	35.9	72	13.5	1.94	102
KL10	256Kw	2DF10's	250+125+28=403	1.37	71.8	72	6.74	3.40	178
C.mmp Configurations									
5 20's	512Kw	-	70+83+50=203	.927	37.6	70	8.34	4.57	185
5 20's 11 40's	1Mw	-	257+166+50=473	5.02	203	225	3.18	10.6	429
16 40's	1Mw	-	272+166+50=488	5.95	241	225	2.69	12.2	494
16 40's	2M	-	272+333+50=655	5.95	241	225	5.38	9.08	367

Table 6.1. Price/Performance Figures for C.mmp and PDP-10's

The dotted lines in Figure 6.1 are for a hypothetical C.mmp in which the Pc's are not standard PDP-11 processors but are Pc's constructed from Intel's I3000 microcomputer chip set [Intel, 1974]. We estimate that an Intel 3000 Pc, and its associated relocation registers, could be priced at $4000 rather than the $17,000 now needed for a PDP-11/40 and its relocation registers.

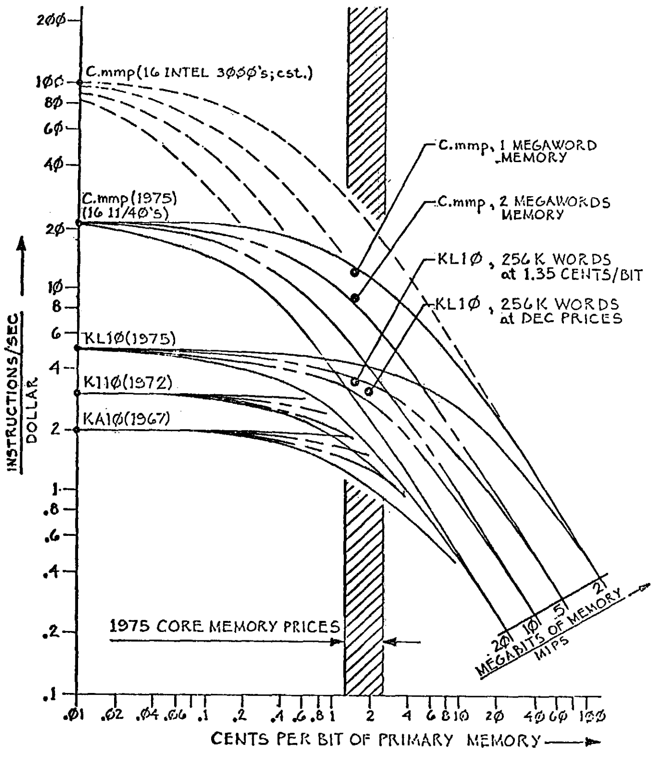


Figure 6.1. Cost Performance of C.mmp

REFERENCES

Amdahl, G., Lecture in course on cache memories and computer architecture (EE 392C), Stanford Universi- Winter quarter, 1970.

Baskett, F. and A. J. Smith, "Interference in Multiprocessor Computer Systems with Interleaved Memory." To appear in Comm. ACM (1975).

Baskett, F., Figure 1.1 is a revision of an unpublished graph developed by Forest Baskett (1975).

Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, New York, 1971.

Bell, C. G. et al., C.mmp: The CMU Multiminiprocessor Computer: Requirements and Overview of the Initial Design, Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa. (1971).

Bhandarkar, D. P. and S. H. Fuller, A Survey of Techniques for Analyzing Memory Interference in Multiprocessor Systems, Carnegie-Mellon University Technical Report, Pittsburgh, Pa. (April, 1973).

Computer Review, GML Corporation, Lexington, Mass., 1974.

(DEC, 1975), PDP-11/05/10/35/40 Processor Handbook, Digital Equipment Corporation, Maynard, Mass. (1973).

(DEC, 1975), LSI-11 Microcomputer, Digital Equipment Corporation, Maynard, Mass. (1975).

Intel 3002 Control Processing Element; Shottky Bipolar LSI Microcomputer Set, Intel Corporation, 1974.

McLaughlin, R. A., "A Survey of 1974 dp Budgets," Datamation, February 1974, 52-56.

Newell, A., P. Freeman, D. McCracken, G. Robertson, "The Kernel Approach to Building Software Systems," CMU 1970-71 Computer Science Research Review.

Newell,A., and G. Robertson, Some Issues in Programming Multi-Mini-Processors, Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., January 1975.

O'Loughlin,J. F., "Microprogramming a Fixed Architecture Machine," Infotech State of the Art Report 23, Infotech Limited, Maidenhead, England, 1975.

Stone, H. S. (ed.), Introduction to Computer Architecture, SRA, Chicago, Illinois, 1975.

Strecker, W. D., Analysis of the Instruction Execution Rate in Certain Computer Structures, Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, Pa., 1970.

Wulf, W. A. and C. G. Bell, "C.mmp -- A Multi-Mini-Processor," AFIPS Conference Proc., Vol. 41, Part II, FJCC 1972, 765-777.

Wulf et al., "The Hydra Operating System," submitted to the Fifth ACM SIGOPS Symposium on Operating System Principles (November 1975).

flexibility in pipeline design. Let us define a cycle to be perfect, if it allows a 100% segment utilization; e.g., cycle (1,9) of Example 1. Unfortunately we cannot test the perfectness of a cycle without forming the compatibility classes. However, we know a special class of perfect cycles which are of considerable interest in single function pipelines.

Theorem 3: All constant latency cycles are perfect.

Proof: For constant cycle (ℓ), $\underline{G}$ mod p=$\{0\}$ and thus $\underline{H}$ mod p=$\{1,2...(\ell-1)\}$. One can verify that $\{0,1,2,...,(\ell-1)\}$ is a compatibility class with ℓ elements. Hence the upperbound on the segment utilization is ℓ/ℓ = 100%. □

III. Noncompute Segments

In this section we consider the addition of noncompute segments to a pipeline to make it allowable for a given cycle. The effect of delaying some computation step can be displayed in a reservation table by writing a 'd' before the X which is being delayed. Each d indicates one unit of delay called an elemental delay. In the absence of any other information on precedence, we must assume that all the steps in a column must be completed before any steps in the next column are executed. Therefore, if the steps in column 2 of Fig. 1 are unevenly delayed, we must store the output of some steps so that all the outputs are simultaneously available to the steps in column 3 of Fig. 1. The effect of delaying the step in row 0, column 2 (X_{02}) of Fig. 1 by 2 units and X_{22} by 1 unit is shown in Fig. 2. The elemental input delays d_1, d_2, and d_3 require the elemental output delays d_4, d_5, and d_6. Now given some integer i between 0 and (p-1), we are in a position to delay any step arbitrarily such that the step occurs in a column number equivalent to i modulo p. Thus given a cycle, we can make any row of a given reservation table to look like one of the rows of Theorem 2; provided of course, the row does not have more X's than the size of the largest compatibility class of the cycle. Hence we have the following theorem.

Theorem 4: For a given cycle, a pipeline can be made allowable by delaying some of the steps, iff the number of X's in each row of the reservation table is less than or equal to the size of the largest compatibility class of the cycle. □

Corollary 4.1: For a given constant latency cycle (ℓ), a pipeline can be made allowable by delaying some steps, iff there are no more than ℓ X's in each row of the table. □

An important implication of Corollary 4.1 is that by adding elemental delays to a pipeline one can always fully utilize a single function pipeline with the use of a cycle with constant latency equal to the maximum number of X's occurring in any single row of the reservation table. Full utilization of a pipeline here, means that at least one segment is busy all the time. Thus the maximum achievable throughput of that pipeline is attained. Of course complete redesign or replication of selected segments to reduce the number of X's in a row may allow higher throughput.

Example 3: The reservation table of Fig. 1 is to be made allowable with respect to cycle (1,5). The resulting table appears in Fig. 3. For cycle (1,5), p=6, $\underline{G}$ mod 6=$\{0,1,5\}$ and hence $\underline{H}$ mod 6=$\{2,3,4\}$. The maximal compatibility classes containing 0 are: $\{0,2,4\}$ and $\{0,3\}$. The first row of Fig. 3 is row $\{0,2,10\}$, which resulted from the class $\{0,2,4\}$ by constructing row $\{0,2,4+p\}$ as per Theorem 2. The second row, $\{1,3,5\}$ results from class $\{0,2,4\}$ and the third row, $\{2,4\}$ results from class $\{2,4\} \subset \{0,2,4\}$. Thus all the rows are allowable. □

Once we have a modified table, we need to assign the elemental delays to noncompute segments. Noncompute segments are physical resources like any other segment and may be shared by various elemental delays for their efficient utilization. Two elemental delays d_i and d_j are defined to be compatible if $|t_i-t_j|$ mod p $\in \underline{H}$ mod p. Where t_i and t_j are labels of the columns in which d_i and d_j appear. Clearly, if d_i and d_j are compatible, they can share one noncompute segment because the usage interval $|t_i-t_j|$ is allowable. Using the above definition we can form the maximal compatibility classes of all the elemental delays present in the solution. All the elements of a compatibility class can share a single noncompute segment. Now the problem reduces to the standard covering problem; i.e., finding the minimum number of compatibility classes which cover all the elemental delays.

Example 4: The set of elemental delays of Fig. 3 is $<d_1,d_2,d_3,d_4,d_5,d_6,d_7>$. Their corresponding column numbers are <3,6,7,8,9,2,3>. For cycle (1,5), $\underline{H}$ mod 6 is $\{2,3,4\}$ (from Ex. 3). Thus $\{d_1,d_2\}$, $\{d_1,d_3\}$, $\{d_2,d_4\}$, $\{d_2,d_5\}$, $\{d_2,d_6\}$, $\{d_2,d_7\}$, $\{d_3,d_5\}$, $\{d_3,d_7\}$ are the maximal compatibility classes. Noting that the subsets of maximal compatibility classes are compatibility classes, one of many possible minimal covers is $\{d_1,d_2\}$, $\{d_4\}$, $\{d_5\}$, $\{d_6\}$, $\{d_3,d_7\}$. Thus 5 noncompute segments are required. The assignement above is shown in Fig. 4, where S_3 through S_7 are noncompute segments□

Besides reducing the number of noncompute segments in a solution, it is also important to reduce the added execution delay. The execution delay of a task in Fig. 1 is 6 units while in the modified table of Fig. 4 it is 11 units. In situations where it often becomes necessary to empty the pipeline; e.g., due to logical dependancies among tasks, the execution delay of a task can become an important parameter in determining the overall throughput. Therefore, we shall take the added execution delay as the objective function to be minimized. Now the problem of making a pipeline allowable can be formulated as follows.

Let I and J be the number of rows and columns in the given reservation table. Let d_{ij} and d'_{ij} be the number of elemental delays to be inserted respectively at the input and output of a step X_{ij} of the reservation table. If no X occurs in cell (i,j) of the table then d_{ij} and d'_{ij} are defined to be zero. Some other d_{ij} can be set to zero if it occurs between two consecutive computation steps which are indivisible. Let D be the added execution delay. Then the problem can be formally stated as:

$$\text{Minimize } D = \sum_{0\leq j<J} \left(\max_{0\leq i<I} (d_{ij}) \right)$$

subject to the constraints,

integer $d_{ij} \geq 0$.

$$[(c-b)+d'_{ab}+d_{ac}+ \sum_{b<j<c} \left(\max_{0\leq i<I} (d_{ij}) \right)] \text{ mod p} \in \underline{H} \text{ mod p}.$$

for each pair $<X_{ab},X_{ac}>$ with c > b.

where, $\underline{H}$ is the set of allowable usage intervals with respect to the given cycle with period p, and

$$d'_{ab} = \max_{0 \le i < I} (d_{ib}) - d_{ab}$$

The constraints result directly from Theorem 1. The term (c-b) is the usage interval which existed between X_{ab} and X_{ac} before the insertion of any delays. The variable d'_{ab} is the number of elemental delays at the output of step X_{ab}; d_{ac} is the number at the input of step X_{ac}. The summation term in each constraint is the effect of inserted delays in the intervening columns between X_{ab} and X_{ac}.

Since all the constraints are in modulo p arithmetic, d_{ij} need only take integer values between 0 and (p-1). Thus the solution space of the above problem is finite. This places an upper bound on the added execution time equal to (p-1)·J, where J is the number of columns in the reservation table. Moreover, the objective function D is nondecreasing in d_{ij}. These properties suggest the following branch-and-bound algorithm to find all minimum added delay solutions.

Let the number of X's in the reservation table be n and let the n variables, d_{ij}, be stored in any arbitrary order in a one dimensional array V. Let D(i) represent the value of the objective function for given values of V(1) through V(i), with V(i+1) through V(n) taken to be 0.

Algorithm B:

```
B1.  [Initialize] $i \leftarrow 0$; BOUND $\leftarrow (p-1)\cdot J$;
B2.  [Advance] $i \leftarrow i+1$; $V(i) \leftarrow 0$;
B3.  [Check bounds and constraints] if (V(i)=p) or
     (D(i)>BOUND) then go to B6; if a completely as-
     signed constraint is violated then go to B5;
B4.  [Solution found?] if i<n then go to B2 else out-
     put the solution V(1) through V(n) and D(n);
     BOUND $\leftarrow$ D(n);
B5.  [Try another value] $V(i) \leftarrow V(i)+1$; go to B3;
B6.  [Backtrack] $i \leftarrow i-1$; if i>0 then go to B5 else
     terminate the algorithm.
```

The last value of BOUND is the minimum value of the objective function over all possible solutions and therefore the output solutions meeting this bound are all the minimum added delay solutions. If only one optimum solution is desired, the condition D(i)>BOUND in step B3 should be changed to D(i)≥BOUND.

A complete example with the constraints and the backtrack tree are given in Fig. 5. The variables, d_{ij}, have been retained in the figure for simplicity. B is the variable BOUND, and '>B' indicates that the bound has been exceeded, and '$\not a$' indicates that the constraint (a) has been violated.

This algorithm is remarkably efficient in our limited experience. For example for one 20 variable problem with a potential 10^{14} nodes only 10^4 nodes were expanded and an optimum solution was obtained in 40 seconds on an IBM 360/67. For a particular class of problems, the technique of [5] may be applicable to estimate the complexity of the algorithm.

IV. Multifunction Pipelines

Here we present the generalizations of most of the results and definitions of the previous two sections. The variables X and Y will be used in most of these results, where X and Y take function names as their values; the values need not be distinct. Let $\underline{F}_{XY}$ be the set of usage intervals of all <X,Y> pairs in the reservation table. This set can be formed by taking all pairwise distances between an X and a Y which appears to the right of the X in the same row. For example, for the reservation table of Fig. 6, the sets of usage intervals are: $\underline{F}_{AA}=\{1\}$, $\underline{F}_{AB}=\{0,1,2,4\}$, $\underline{F}_{BA}=\{0,2,3\}$, $\underline{F}_{BB}=\{2,3\}$.

Similarly we define $\underline{G}_{XY}$, the set of initiation intervals of all <X,Y> pairs of a cycle, to be the set which contains all intervals of a task of type X from a previously initiated task of type Y. A cycle is described with latencies suffixed with the function name of the task being initiated with that latency; e.g., cycle $(1_A,1_B,2_A)$. The period p is the sum of the latencies. The initiation interval sets for cycle $(1_A,1_B,2_A)$ are: $\underline{G}_{AA} \bmod 4=\{0,1,3\}$; $\underline{G}_{AB} \bmod 4=\{2,3\}$; $\underline{G}_{BA} \bmod 4=\{1,2\}$; $\underline{G}_{BB} \bmod 4=\{0\}$. The properties P1, P2 and P3 can be generalized as follows.

P4.a. if $g\neq 0$ then $g \in \underline{G}_{XX} \bmod p \Rightarrow g+ip \in \underline{G}_{XX}$ $\forall i\geq 0$.

b. $0 \in \underline{G}_{XX} \bmod p$ and $ip \in \underline{G}_{XX}$ $\forall i\geq 1$, always.

c. if $X\neq Y$ then $g \in \underline{G}_{XY} \bmod p \Rightarrow g+ip \in \underline{G}_{XY}$ $\forall i\geq 0$.

P5.a. if $g\neq 0$ then $g \in \underline{G}_{XY} \bmod p \Leftrightarrow (p-g) \in \underline{G}_{YX} \bmod p$.

b. $0 \in \underline{G}_{XY} \bmod p \Leftrightarrow 0 \in \underline{G}_{YX} \bmod p$.

P6. if $h\neq 0$ then $h \in \underline{H}_{XY} \bmod p \Leftrightarrow (p-h) \in \underline{H}_{YX} \bmod p$, where $\underline{H}_{XY} \bmod p$ is the complement of $\underline{G}_{XY} \bmod p$, in $\underline{Z}_p$.

Theorem 5: A cycle is allowed by a multifunction pipeline iff $(\underline{F}_{XY} \bmod p) \cap (\underline{G}_{XY} \bmod p) = \underline{\Phi}$, or equivalently iff $(\underline{F}_{XY} \bmod p) \subseteq \underline{H}_{XY} \bmod p$, $\forall$ X,Y in the set of function names present in the cycle. □

The generalization of the definition of compatibility is straightforward, except that each integer must be suffixed with an appropriate function name. Thus two elements i_X and j_Y, such that $i,j \in \underline{Z}_p$ and $j>i$, are said to be compatible if $(j-i) \in \underline{H}_{XY} \bmod p$. The following are generalizations of Lemma 2.1 and Theorem 2.

Lemma 6.1: Two elements i_X and j_Y such that $i,j \in \underline{Z}_p$ are compatible iff $(j-i) \bmod p \in \underline{H}_{XY} \bmod p$. □

Theorem 6: Given a cycle with period p, all possible rows which are allowed by the cycle are:

row $\{(i+i_1p)_X,(j+j_1p)_Y,\ldots\}$ $\forall$ nonnegative integers $i_1,i_2,j_1,j_2\ldots$ and $\forall$ compatibility classes $\{i_X,j_Y,\ldots\}$. □

The maximal compatibility classes can be formed in a manner similar to the one for single function pipelines. As an example take again the cycle $(1_A,1_B,2_A)$ whose $\underline{G}$ sets were formed earlier. The allowable usage interval sets are: $\underline{H}_{AA} \bmod 4=\{2\}$; $\underline{H}_{AB} \bmod 4=\{0,1\}$; $\underline{H}_{BA} \bmod 4=\{0,3\}$; $\underline{H}_{BB} \bmod 4=\{1,2,3\}$. The maximal compatibility classes containing 0_X are: $\{0_A,2_A\}$, $\{0_B,1_B,2_B,3_B\}$, $\{0_A,0_B,1_B\}$, $\{0_B,3_A,3_B\}$.

A compatibility class $\underline{C}_1$ is said to cover another class $\underline{C}_2$ if for each function, the number of elements of that function type in class $\underline{C}_1$ is greater than or equal to the number of elements of the same function type in class $\underline{C}_2$. In the above example, $\{0_A,0_B,1_B\}$ and $\{0_B,3_A,3_B\}$ cover each other. The same definition for cover applies among rows and also between a row and a compatibility class. Now we have the generalization of Theorem 4.

Theorem 7: For a cycle, a multifunction pipeline can be made allowable by delaying some computation steps iff each row of the reservation table is covered by at least one compatibility class of the cycle. □

Now it is a simple matter to formulate the problem of making a pipeline allowable. In a multifunction pipeline different functions have different execution times. Let D(X) be the added execution delay to function X. The objective function can be any function of the D(X)'s, which is nondecreasing in each D(X); e.g., some linear combination of D(X)'s with positive coefficients. Let $d_{ij}(X)$ be the number of elemental delays to be inserted at the input of a step of function name X in cell (i,j). Let I and J be the number of tows and columns in the reservation table. The added execution delay for a function X can be expressed as

$$D(X) = \sum_{0\le j<J} \left(\max_{0\le i<I} d_{ij}(X), \right)$$

While the constraints can be written from Theorem 5, the usage interval between X_{ab} and Y_{ac} can be expressed as: [(the distance of Y_{ac} from column 0) - (distance of X_{ac} from column 0)]. Thus we have the following set of constraints.

Integer $d_{ij}(X) \ge 0$

$$\text{and } [(\sum_{0\le j<c} \max_{0\le i<I} d_{ij}(Y)+d_{ac}(Y)+c)$$

$$-(\sum_{0\le j<b} \max_{0\le i<I} d_{ij}(X)+d_{ab}(X)+b)] \bmod p$$

$$\in \underline{H}_{XY} \bmod p. \text{ for each pair } <X_{ab},Y_{ac}>.$$

From property 6 we can see that we need construct only one constraint per pair without regard to the magnitudes of b and c. The algorithm to obtain an optimum soltuion is the same as Algorithm B.

V. Concluding Remarks

We have presented the allowability characteristics of pipelines and cycles. We know the structure of all allowable pipelines for a given cycle. It is seen that one can utilize a pipeline fully by adding noncompute segments to make it allowable with respect to a perfect cycle. For nonperfect cycles, the pipeline can still be made allowable if every row of the reservation table is covered by at least one compatibility class of the cycle.

For single function pipelines, constant latency cycles were shown to be perfect. Thus a single function pipeline can always be utilized fully with the use of an appropriate constant latency cycle.

For multifunction pipelines, there is no straightforward procedure to construct a perfect cycle, given a mix of functions to be executed. However, if a cycle is given, it can always be tested for its perfectness with the use of compatibility classes. Cycles which are most likely to be perfect are those having evenly spaced task initiations, as well as a fairly regular pattern of functions. These cycles have a small set of initiation intervals and hence one has more freedom in choosing an allowable usage interval. For the same reason, these cycles are also most likely to require a small number of noncompute segments in making a pipeline allowable.

For increasing the throughput beyond what would result due to the full utilization of a pipeline, segment replication must be done. Segment replication is also a viable alternative to noncompute segments if the costs are comparable. For a cost effective design, segment replication and addition of delays should be considered simultaneously.

References

1. E.S. Davidson, "The design and control of pipelined function generators," Proc. 1971 Int. IEEE Conf. on Systems, Networks and Computers, Oaxtepec, Mexico, January 1971.
2. L.E. Shar, "Design and Scheduling of Statically Configured Pipelines," Tech. Report No. 42, Digital Systems Lab., Stanford University, Sept. 1972.
3. A.K. Winslow, "Task scheduling in a class of pipelined systems," Report R-633, Coordinated Science Lab, Univ. of Illinois-Urbana, Nov. 1973.
4. E.S. Davidson, L.E. Shar, A.T. Thomas, and J.H. Patel, "Effective Control for pipelined computers," Proc. Compcon Spring 1975, pp. 181-184, Feb.1975.
5. D.E. Knuth, "Estimating the efficiency of Backtrack programs," Mathematics of Computation, Vol.29, no. 129, pp. 121-136, Jan. 1975.

	0	1	2	3	4	5
S_0	X		X			X
S_1		X	X		X	
S_2			X	X		

FP-4684

Figure 1. Reservation table

	0	1	2	3	4	5	6	7
S_0	X		d_1	d_2	X			X
S_1		X	X	d_4	d_5		X	
S_2			d_3	X	d_6	X		

FP-4685

Figure 2. Delaying parallel computation steps

	0	1	2	3	4	5	6	7	8	9	10
S_0	X		X	d_1			d_2	d_3	d_4	d_5	X
S_1		X	d_6	X		X					
S_2			X	d_7	X						

FP-4686

Figure 3. Making a pipeline allowable for cycle (1,5)

	0	1	2	3	4	5	6	7	8	9	10	
S_0	X		X									
S_1		X		X		X						
S_2			X		X							
S_3				X			X					$\{d_1,d_2\}$
S_4									X			$\{d_4\}$
S_5										X		$\{d_5\}$
S_6			X									$\{d_6\}$
S_7				X				X				$\{d_3,d_7\}$

FP-4687

Figure 4. Assignment of elemental delays to noncompute segments

	0	1	2	3	4
S_0	A		B		B
S_1		AB		B	
S_2	B		A	AB	

FP-4688

Figure 6. Reservation table for a multifunction pipeline

	0	1	2
S_0	X		X
S_1	X	X	

FP-4689

cycle (2). $\underline{H} \bmod 2 = \{1\}$

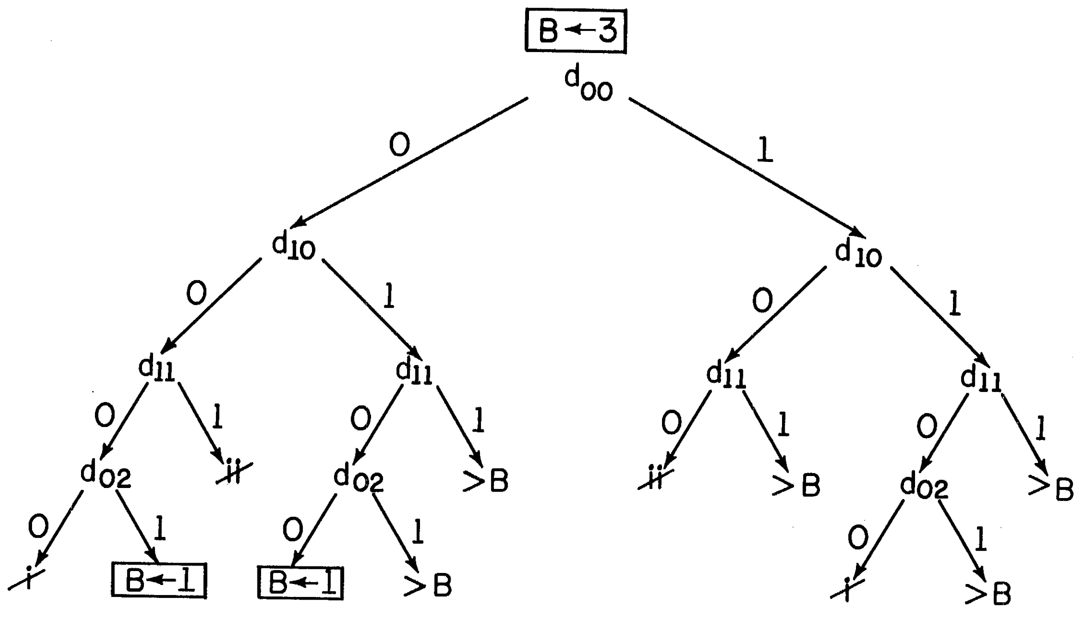
Added delay:

$$D = \max\{d_{00},d_{10}\} + d_{11} + d_{02}$$

Constraints:

(i) $[2 + \max\{d_{00},d_{10}\} - d_{00} + d_{02} + d_{11}\} \bmod 2 \in \{1\}$.

(ii) $[1 + \max\{d_{00},d_{10}\} - d_{10} + d_{11}\} \bmod 2 \in \{1\}$.



Optimum solutions are: 1. $d_{00} = d_{10} = d_{11} = 0.\ d_{02} = 1.$

2. $d_{00} = d_{11} = d_{02} = 0.\ d_{10} = 1.$

Figure 5. Making the pipeline allowable for cycle (2): A branch-and-bound search for optimum solutions.

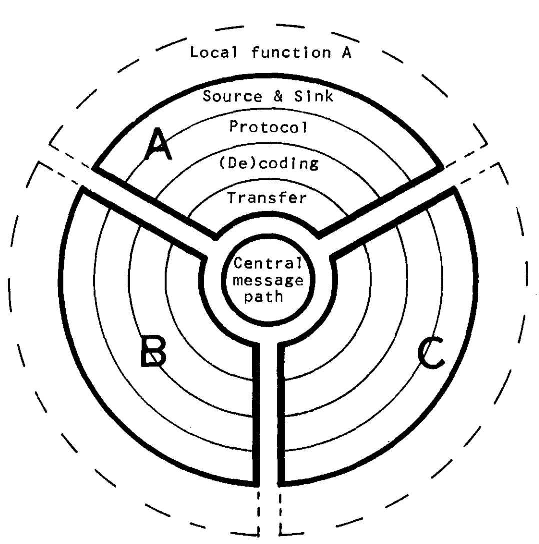


Figure 10: Layered structure for an interface

4.7 Further development.

The previous discussion of the structure of an interface suggests a sequence in the development of the layers, according to the sequence of the sections 4.1 through 4.6. This development is based on a strategy of successive definition. First the architecture of the total interface is determined, and its partitioning and dispersion over the related architectures. Next the architecture of the central message path is determined, and finally the architectures of the individual relational functions. Though this procedure is a useful guideline, a practical application often requires a substantial number of iterations through this sequence, due to the high dependency among the layers.

A further substructuring per layer may result in either the development of sublayers per layer (extended horizontal partitioning) or a partitioning of a layer into functions which are not or only slightly related (vertical partioning). The previous discussions have already used the vertical partitioning by interpreting each layer as a class of functions, and showing examples of such functions. Much is dependent on the possibility of defining a function first as an independent entity, and next of establishing the linkages to and from other functions. As is true for the vertical partitioning, the extended horizontal partitioning may also provide more clarity in the specification of the interface. The protocol function of figure 6 shows what type of operations may be sequenced. The way these operations are organized in detail can be specified in a lower protocol layer. Complex data transmission interfaces may build up their transfer layer as a stack of sublayers. Such a sublayer, and all that it encloses, may be interpreted as the central message path of the transfer layer that is just one level higher. An opposite development also occurs frequently: variables pass a layer unchanged.

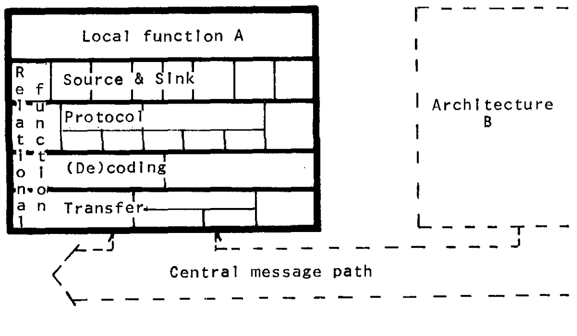
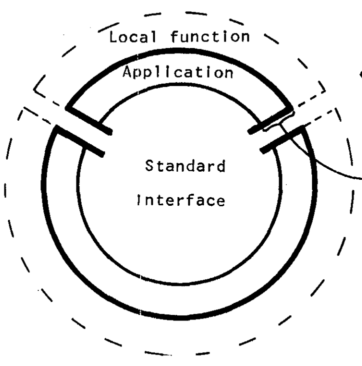


Figure 11: The interface from the perspective of architecture A.

The structure so far developed for the interface is shown in figure 11 from the perspective of an individual architecture. Each box in the figure represents a function, that exists in parallel with the other functions and is related with them via the exchange of variables. This horizontal and vertical structuring is different from the structuring in which functions on a lower layer are used to implement an abstract machine on a higher layer [10].

5. What is a standard interface ?

As stated, a system can be understood as a collection of interfaces (figure 5b) as well as a collection of architectures (figure 5a). This viewpoint is significant when an interface is defined first, and the associated architectures later. This happens with a so called standard interface. A standard interface, such as a Channel-to-I/O interface, is always defined to meet many different architectures, e.g. printers, tape units, disc units, display devices, architectures that still have to be invented, etc. in different quantities and configurations. At the time of the definition of the standard, the current application area is known, and there is a rough estimate of the characteristics of future applications. Definition of a standard to include all current and future applications is not only impossible, it is also highly inappropriate since it loads any particular application with the overhead of a multiplicity of unused application functions. Instead the standard is defined to suit all requirements of current and future applications without containing the specific functions of individual applications. The standard is by definition incomplete. Consequently, when the standard is used in a particular application, each relational function has to be extended with application dependent functions. Those application dependent functions form yet another layer around the source and sink layer of the standard interface, and are designated 'Application' in figure 12a.



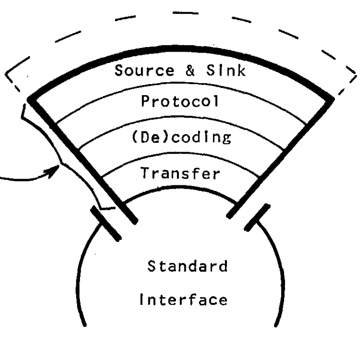


Figure 12a: Application of a Standard Interface

Figure 12b: Partitioning of the Application layer.

The consequence of this structure is that the variables exchanged among the application functions are unknown, i.e. transparent to the standard interface, and yet pass all layers and the message path. Since we want the function of the standard te remain invariant with each application, it implies that the standard has to provide for the space and time for the exchange of those variables. If on the level of the central message path the available space is to be defined in terms of available code elements, the definition of the available space at the level of the source and sink functions has to be in terms of the same number of code elements, since the coding of the variables is transparent with respect to the standard. The coded source in figure 3 is an example. Therefore, in using a standard interface in a particular application, the application dependent interface can be defined according to the procedure explained above. The standard interface is now embedded as a central message path with a high level of complexity. (See figure 12b).

6. Application

The structuring and description discipline has been succesful applied to a number of existing and proposed standard interfaces. Among these are a complex data transmission interface [2], two I/O interfaces, one complex Channel-to-I/O interface [1], and an instrumentation interface [4]. The relational function of the secondary station of SDLC [2] was for example described by 25 functions, each of an average complexity as shown in the figures 8 and 9. It contains 4 sources, 2 sinks, 8 protocol, 2 decoding, 3 encoding, and 6 transfer functions. A formal specification was developed as far as the intentions of the interface architects were stated unambiguously. This specification was generally a fraction of the length of the original document. As part of the description process ambiguities and omissions in the original documents were systematically uncovered.

The state description technique was introduced in an IEC (TC 66/WG 3) standardization activity in june 1973 [4], and eventually accepted as the method to define the considered interface. In the opinion of the committee it has contributed much to the fact that the definition work was practically completed within 9 months, that is May 1974 [5]. A structured and complete description of this interface can be found in [3].

7. Conclusion.

The proposed design discipline facilities fast, correct, efficient and clear specification, interpretation, and judgement of an interface through the definition and its evaluation into a structured specification methodology. As such it can be profit for both interface designers and users:

- The definition provides a better understanding, of what an interface substantially is: a specification of a portion of each of the related architectures (relational functions) and the architecture of the message path, defined to provide cooperation of the related architectures. It is not the story of the reporter, who is sitting on a grandstand, viewing the communication between the related architectures, observing, interpreting and logging what happens. It is the rules of the game according to which the teams play.
- The architecture of each relational function and the message path is specified individually. For all these architectures one specification methodology and language should be used. Poor interface specification mixes relational functions and message path, as well as specification methodologies and languages.
- The horizontal and vertical partitioning strategy for the specification of the relational functions facilitates the recognition of the nature of functions of a particular application and their embedding in such a structure. It facilitates easier specification and recognition of quality and correctnes of the individual and compound functions.
- A standard interface is by definition incomplete. It can be interpreted as a complex central message path, that can be extended to a complete interface in a particular application.
- The method has proven to be applicable to a number of widely used interfaces.

8. Acknowledgement.

The project Interface was suggested by G.A. Blaauw. It is one of the projects of the digital techniques group at Twente University of Technology that focusses on the control of the design process of digital systems. The author would like to thank G.A.Blaauw, B.v.d.Dolder, and R.Davey for their constructive criticism during the preparation of this paper.

9. References.

1. M.J. Heg - Formal description and evaluation of a proposal for an international standard for an input/output interface for electronic data processing systems (ISO TC97/SC13) - M. Sc. Thesis (Dutch/English). June 1975 - Twente Univ. of Techn.
2. B.v.d. Dolder - Algorithmic description of a data transmission interface - M. Sc. Thesis (Dutch) - June 1975 - Twente Univ. of Techn.
3. C.A. Vissers - Digital Techniques IV: Interface- Lecture notes - Spring 1975 - Twente Univ. of Techn.
4. Byte-serial bit-parallel standard interface for programmable measuring apparatus - Drafts of July 1973 and June 1974 - IEC TC66/WG3.
5. D.E. Knoblock, D.C. Loughry, C.A. Vissers - Insight into Interfacing - IEEE Spectrum - May 1975 - pp. 50-57.
6. D.L. Parnas - Information distribution aspects of design methodology - Proc. IFIP, 1971 - North-Holland Publishing Company (1972).
7. R.W. Floyd - Assigning meanings to programs - Proceedings of symposia in Applied mathematics, Vol. 19, Mathematical aspects of computer science, pp. 19-32, American Mathematical Society, 1967.
8. C.A.R. Hoare - An axiomatic basis for computer programming - Comm. ACM. 12, 576-580, 583 (1969).
9. F. Wijnstra - A conversational system for representation and verification in APL of interfaces - M. Sc. Thesis (Dutch) - sept. 1975 - Twente Univ. of Techn.
10. E.W. Dijkstra - The structure of the THE multiprogramming system - CACM, Vol. 11, No.5, May 1968. pp. 341-346.

A DESIGN STUDY OF A SHARED RESOURCE COMPUTING SYSTEM*

A. Thomasian and A. Avizienis
Computer Science Department
University of California
Los Angeles, California 90024

Summary. The motivations for the design study of a modular, shared resource computing system are given by discussing fault-tolerance and resource utilization issues in parallel processing architectures. A design is presented which employs an array of pipelined arithmetic processors to perform array operations. The design provides for fault-tolerance ("graceful degradation") capability and is efficient in using main memory bandwidth. Various architectural tradeoffs of the design are discussed. Some results of simulations used for the verification of design decisions are also reported.

1. Some Current Issues in the Design of Numeric Processors

1.1 Introduction

We wish to report several aspects of a design study of a fault-tolerant (highly available) [1] Shared Computing Resource (SCR) for parallel processing, which is intended for use in a multiaccess, scientific ("number crunching") computing environment. In fact, the SCR corresponds to a high capacity node in a hierarchical network of computing nodes [2]. Our purpose is to present the evolving architecture of the SCR system, identifying the constraints that apply and the tradeoffs that have to be considered.

Many scientific computations involve array processing and hence need a mathematical programming language with array capabilities, such as APL [3]. The large computing requirements of such programs call for a system which is tailored to carry out array computations very efficiently. To this end, some computers such as the CDC STAR-100 [4], the TI ASC [5] and the STARLET computer [6] implement array operations directly in their hardware. Alternatively, special purpose systems are used in conjunction with general purpose computers to attain cost-effective operation in array processing. The IBM 2938 Array Processor [7] and, at a larger scale, the Illiac IV [8], the PEPE [9], and the SCR belong to this category.

To put the SCR design in the proper perspective, we initially discuss fault-tolerance and resource utilization issues in parallel processing architectures. This is followed by the functional organization of SCR, its operation in the context of a hierarchical multiprogramming/multiprocessing system, various tradeoffs considered in designing the SCR and the scheduling issue in the SCR system. We conclude our discussion with a comparison of the scheduling of array operations in the SCR and the ASC systems.

1.2 Fault-Tolerance Issues in SIMD Computer Architectures

SIMD computer architectures [10], which are of interest here have been classified to [11]:

(a) Parallel in space and structured array machines with a high level of interconnectivity among the processing elements, such as Illiac IV.

(b) Unstructured linear array and associative processors, such as the PEPE system.

(c) Primarily parallel in time or pipelined processors, such as the STAR-100 and the ASC.

*This research was supported by the National Science Foundation, Grant No. DCR72-03633 A03.

Obviously, a wide range of systems with various degrees of interconnectivity exist in the spectrum between "structured" and "unstructured". Also, pipelining can be incorporated in the first two categories.

The hardware complexity of SIMD systems impairs their reliability and complicates the implementation of fault-tolerance or, at least, partial fault-tolerance. This is especially evident in structured systems. For example, we consider dynamic reconfiguration in the Illiac IV. In order to switch out a failed processing element (PE) and activate a spare PE, we would need additional, high bandwidth interconnections among the PE's. Furthermore, operation in a "degraded" mode (with fewer PE's) is not practical, since most programs written for the Illiac IV take its structure into account during computation. On the other hand, in systems with limited interconnections, "graceful degradation" and even complete fault-tolerance can be achieved at little extra cost, once provisions for on-line fault detection have been incorporated.

It is evident that fault-tolerance or partial fault-tolerance ("graceful degradation") is a very desirable attribute for parallel processing systems. The SCR system described in this paper is an attempt to explore the problems of introducing fault-tolerance into parallel processing. The motivations for some design decisions of the SCR are given in the next two sections.

1.3 Processor Utilization Issues in SIMD Architectures

A frequent resource underutilization associated with parallel-in-space SIMD computer architectures is due to the following facts: (1) often task requirements cannot be matched to the available processors, and (2) processors are used sporadically during their assignment to a program.

Resource sharing in space with self-optimized scheduling has been proposed to increase resource utilization in parallel processing systems [12]. For example, resource sharing was proposed for the Illiac IV computer [8] consisting of four quadrants, such that tasks requiring one or two, but not all quadrants for execution would be able to share the system.

Theoretical justifications for resource sharing are provided in [13]. It is shown that an important performance measure, the mean response time, improves significantly when the system load and processing capacity is increased simultaneously. The space sharing approach is further discussed in conjunction with the SCR system.

1.4 Memory Utilization Issues in High-Performance Computers

The largest cost component in high-speed pipelined computers is the main memory; hence strong emphasis must be placed on effective utilization of memory bandwidth and space.

Since we frequently deal with large arrays of data, the efficient handling of temporary results has major importance in such computers. Due to limited memory space, the programmer might be constrained to use a given vector length for all of his computations [14]; alternatively, the space reserved for temporary results might be specified via a compiler run time parameter [15]. Another major issue, which underscores the importance of memory bandwidth and space, is the refurbishing of main memory contents, such that computations can proceed in an uninterrupted manner.

As discussed in [14], a space-time tradeoff exists regarding temporary results. Since each array operation consists of a startup time and an execution time, some time is wasted due to additional startups, when a large array has to be operated upon in parts. Additionally, there exists a memory management overhead in allocating space for temporary results.

A straightforward solution to the above problem is the "elimination" of intermediate array results, with the consequent saving of memory accesses and space [16]. This scheme, which has been implemented at the scalar level in the IBM 360/91 [17], is considered in the context of the SCR design for arrays of data, as the following two schemes:

(a) The storing and fetching of temporary results is avoided by transmitting them directly among the respective arithmetic units. This scheme can be extended to sequences of assignment statements having common subexpressions and to the case where the final result of an array expression is the input to another one.

To weigh the attractiveness of this approach, we evaluate the relative saving in memory accesses when an array assignment statement, involving n binary operators is evaluated. Denoting the number of array elements by ℓ, customarily $3n\ell$ memory accesses would be required, while the proposed scheme requires $(n+2)\ell$ accesses; hence $2(n-1)\ell$ accesses are saved. Given that the probability of the occurrence of an arithmetic assignment statement with n ($n>0$) binary operators is p_n and postulating a fixed mean array size ($\bar{\ell}$) for array expressions of varying complexity, then the relative saving in memory accesses is $(2\bar{n}-2)/3\bar{n}$, where $\bar{n}$ is the mean value of n.

(b) Memory accesses are saved by concurrently executing operations involving the same input operands. An example of the relative saving in memory accesses using this scheme is given in Section 2.4.

The use of variables in a sequence of array assignment statements of a program can be represented as a directed acyclic graph, which will be called the data digraph. Each node in the data digraph corresponds to an input variable or the generation of a result (permanent or temporary). The links determine variables or temporary results, which are utilized in generating a new result. The data digraph can then be manipulated (see Section 2.5) to determine sets of operations whose simultaneous execution minimizes memory accesses. Such sets of operations, which have to be executed in a single step by the SCR, constitute a task.

To illustrate the previous discussion, we consider the multiplication of two vectors with complex data types:

$$A.B = (a+a'i) \cdot (b+b'i) = (ab-a'b') + (ab'+a'b)i$$

Figure 1 gives the data digraph corresponding to this computation. In this case the relative saving in memory accesses, when all operations are performed in one step is 66.7%.

2. The SCR: Functional Description

2.1 Operating Environment of the SCR

The SCR is intended to operate in conjunction with a multiprogramming/multiprocessing computing system, whose interfaces with the SCR are discussed here.

The computing system consists of several Program Processors (PP's), which execute user programs and perform OS functions. The PP's and the SCR share a high-bandwidth main memory by means of a main memory controller. The main memory is large enough to allow multiprogramming. The PP's are equipped with local memories, thus relieving the main memory from excessive PP accesses. Programs executed by the PP's have special provisions for specifying array operations and while executing user programs, the PP's relegate array operations to the SCR. However, scalar operations and also array operations that cannot be vectorized (see [15] for examples) are handled directly by the PP's. The SCR has local autonomy and requests for computation or tasks, which the SCR receives from various PP's are enqueued in the SCR and assigned to execution based on local self-optimization considerations.

2.2 Functional Organization of the SCR

The SCR design is aimed toward the major goals of achieving fault-tolerance ("graceful degradation") and of making efficient use of main memory bandwidth.

The approach employed to preserve main memory bandwidth is to allocate several Arithmetic Processors (AP's) to the execution of a task such that temporary results are transmitted directly from one AP to another*. Since rather high bandwidths of data transmission are involved, an Interconnection Network (IN) is used to transmit intermediate results among the AP's. Additionally, due to the high data transfer rates at which array operands are to be transmitted between the main memory and the SCR, dedicated Address Generators (AG's) are assigned to each array operand.

In order to achieve fault-tolerance and high availability, a "pooling" concept is used for the various subsystems of the SCR. In the case of AP's, the mean AP requirement for a single task (as generated by a program translator) is smaller than the total number of AP's. During program execution, a subset of the available AP's (under some constraints due to the IN) is assigned to the execution of a task. Several tasks can be executed concurrently in the SCR. The binding of program requests to the SCR elements is deferred until the time of execution. At that time it is performed dynamically taking into account the inventory of available elements. Consequently, system operation can continue with fewer elements (in "degraded mode") after failures of system elements occur.

Figure 2 gives a block diagram of the SCR and its interfaces with the computing system. The SCR consists of the following subsystems:

(a) A pool of m AP's (Arithmetic Processors) which access the main memory controller by means of a Memory Interface Unit (MIU). The AP's are high bandwidth, pipelined arithmetic units capable of performing basic arithmetic operations generating elementary results (sums, products, etc.), as well as some common matrix operations such as the inner product (it is considered to be a nonelementary result). The internal structure of the AP's will not be discussed here, but we postulate that once an AP is set up by an external command, it proceeds autonomously with the assigned operation. An Input Switching Unit (ISU) whose function is described in (c) below is associated with each AP.

(b) The MIU contains a pool of k AG's (address generators) which generate the addresses of data elements to be transmitted to or from main memory. Each AG is associated with a buffer memory to mask the variation in main memory response time. High bandwidth buses are used to transmit data and addresses between AG's and the main memory controller. The operation of AP and AG units is overlapped, such that the AG's fetch input operands in lookahead mode into the buffers, before the AP's operate upon them.

(c) The IN (interconnection network) provides data communication links among the AP's according to the pattern described in Section 2.5. The ISU associated with each AP selects the specified inputs from the set of buses originating from the AG's and other AP's (the IN) according to task requirements under external control.

(d) A Switching Network (SN) is used to dynamically assign AG's to AP's. The motivation and certain

*A variation of this approach is discussed in Section 2.4

Effect of CPU Speed

The percentage improvement in the figure of merit (f) versus the primary memory cost/program for CPU's with different speeds is drawn in Figure 7. For low primary memory cost/program the combination of a slow processor with a slow CCD has a better performance improvement than a fast processor with a fast CCD. Also in certain regions of the graphs the three different speed CCD's and a given CPU track each other showing no significant advantage in using a fast processor over a slow processor. This indicates that in this region the instruction execution is memory speed limited rather than processor speed limited.

Effect of Degree of Multiprogramming

The effect of degree of multiprogramming D on f is shown in Figure 8. A large improvement occurs in going from D=1 to D=4, a small amount in going from D=4 to D=8 and very little in going from D=8 to D=16. This is due to the high probability of at least one task waiting for service at the processor queue when the degree of multiprogramming is increased. A similar behavior can be expected if the CPU's or the type of CCD are changed.

Effect of Amount of MOS Memory Retained

Figure 9 shows the effect of the amount of MOS memory retained on the percentage improvement in f. As higher amount is spent on the primary memory per program, it is seen that the optimum percentage improvement in f occurs at higher and higher values of the MOS memory being retained.

Conclusions

A two server queuing model is used to analyze the performance of a memory hierarchy in a multiprogramming mode. For the primary memory a two level hierarchy of Bipolar, MOS is compared with a three level hierarchy of Bipolar, MOS and CCD by keeping the cost of the primary memory constant. A figure of merit that is a function of number of instructions executed is used to evaluate the hierarchies. It is shown that a hierarchy using CCD's has 2 to 3 times higher figure of merit over that using just MOS. Effect of varying the speed of the CCD's used, effect of different CPU's, effect of degree of multiprogramming and the effect of the amount of memory retained is then evaluated. An interesting result seen is that for small values of primary memory a slow CCD with slow CPU has better figure of merit than a fast CCD with fast CPU. Also, for certain regions of primary memory requirements, it is seen that no advantage is gained by going to a faster CPU.

Acknowledgement

I would like to thank Marvin E. Steiner, Kent R. McCune and G. Panigrahi for pertinent comments and discussions.

References

1. Altman, L.: "Special Report - CMOS Enlarges its Territory," Electronics, Vol. 48, No. 10, May 15, 1975, pp. 77-88.
2. Amelio, G. F.: "Charge-Coupled Devices for Memory Applications," NCC, AFIPS Conference Proceedings, Vol. 44, May 1975, pp. 515-522.
3. Bhandarkar, D. P.: "Computer System Advantages of Magnetic Bubble Memories," IEEE Computer Society Repository, No. R-75-114, June 1975.
4. Bobeck, A. H. and H. E. D. Scovil: "Magnetic Bubbles," Scientific American, June 1971.
5. Fuller, S. H. and F. Baskett: "An Analysis of Drum Storage Units," JACM, Vol. 22, No. 1, January 1975, pp. 83-105.
6. Hiller, F. S. and G. L. Lieberman: Introduction to Operation Research, Holden-Day, San Francisco, 1967.
7. Harton, R. L., J. Englade and G. McGee: "I^2L Takes Bipolar Integration A Significant Step Forward," Electronics, Vol. 48, No. 3, February 6, 1975, pp. 83-90.
8. Martin, R. R. and H. D. Frankel: "Computer Memories of the Future," IEEE Intercon, April 1975, Session K.
9. Pugh, E. W.: "Storage Hierarchies: Gaps, Cliffs, and Trends," IEEE Transactions on Magnetics, Vol. Mag. -7, No. 4, December 1971, pp. 810-814.
10. Spain, R. and M. Marino: "Magnetic Film Domain-Wall Motion Devices," IEEE Transaction on Magnetics, Vol. Mag.-6, No. 3, September 1970, pp. 451-463.
11. Speliotis, D. E.: "Bridging the Memory Access Gap," NCC, AFIPS Conference Proceedings, Vol. 44, May 1975, pp. 501-508.
12. Strecker, W. D.: "An Analysis of Instruction Execution Rate in Certain Computer Structures," Ph.D. Thesis, EE Dept., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania, 15213, 1970. (AD711408).
13. Traiger, I. L. and R. L. Mattson: "The Evaluation and Selection of Technologies for Computer Storage Systems," AIP Conference Proceedings, No. 5, Part I, Magnetism and Magnetic Materials, 1971, pp. 1-12.

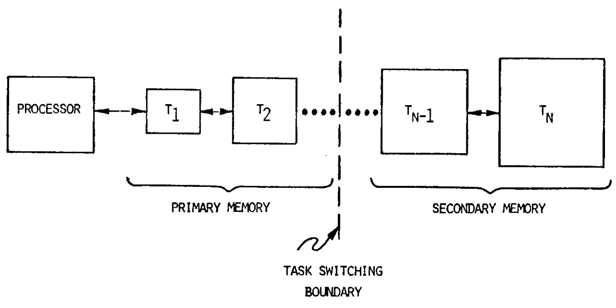
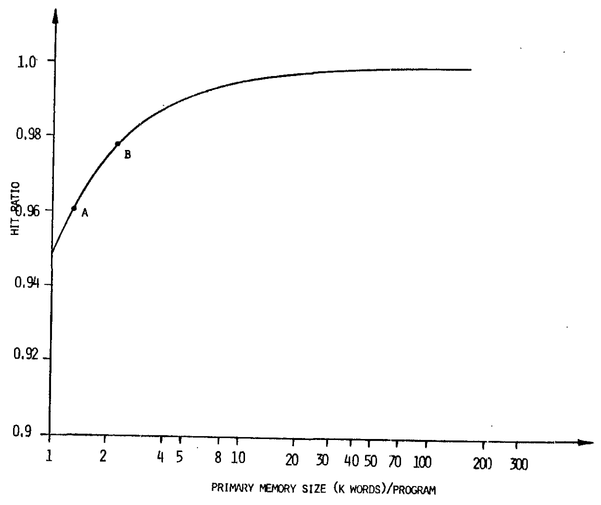


FIGURE 1: A MEMORY HIERARCHY WITH A TASK SWITCHING BOUNDARY

FIGURE 3: TYPICAL HIT RATIOS FOR A SET OF REPRESENTATIVE PROGRAMS



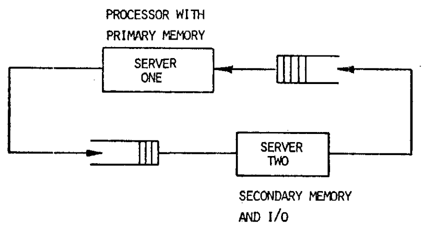


FIGURE 2: TWO SERVER QUEUING MODEL

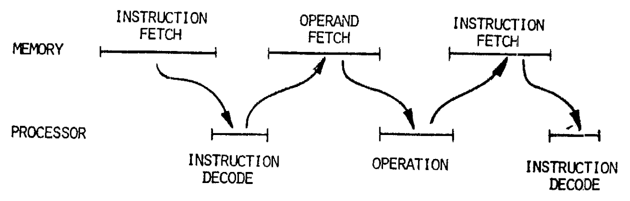


FIGURE 4: MODEL FOR PROCESSOR AND MEMORY OPERATION

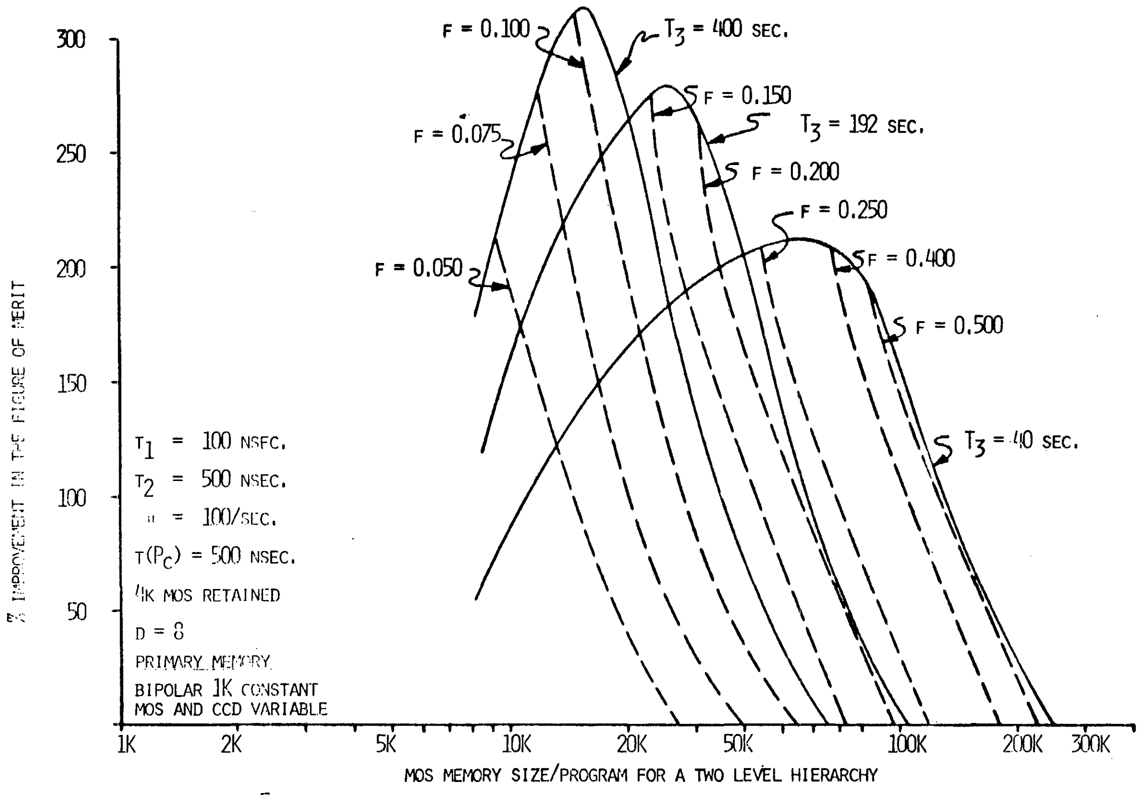


FIGURE 5: EFFECT OF CONSTANT COST AND CONSTANT PERFORMANCE TRANSFORMATIONS ON A MEMORY SYSTEM DESIGN

Model and Assumptions

The behavior of a typical task executed in a multiprogramming environment is represented by four states: the task being serviced by the processor, the task waiting for the secondary memory or I/O service in a queue, the task being serviced by the secondary memory or I/O, and finally, the task waiting in a queue for processor service. Thus, in general, there are two queues and two service facilities and a task cycles through them until it is completed (Figure 2). This, then, can be modeled by a two server cyclic queuing model. Traiger[13] has referenced the use of this model, Fuller and Baskett[5] have used it in their analysis of scheduling philosophies of drum systems while Bhandarkar[3] has used it to compare magnetic bubbles, CCD's, Fixed and Moving Head disks, etc. Most previous researchers have used CPU utilization as a main criterion to evaluate the effect of multiprogramming. Some of the other criterions considered are the waiting time in queue and the memory utilization, which is the percentage of the time that a given memory spends its time transferring its data. The criterion used here will be the ratio of the actual number of instructions executed by the processor to the maximum number of instructions executed provided all the memory was substituted by the level having the fastest speed.

The assumption made in using the two server queuing model (Figure 2) is that both server one, consisting of the processor and the primary memory, and server two, consisting of secondary memory and I/O, have an exponential service time distribution. Even though this may not be the case in any particular computing system, most models make this assumption since most natural phenomenon can be modeled by a poisson process and a general feeling for the performance of the hierarchy can be determined. Later, simulations may be used to verify the results. A FIFO scheduling philosophy is assumed for all queues in the system.

Hit Ratio Characteristics

A typical hit ratio characteristic as shown in Figure 3 is used to determine the performance of the hierarchy. The statistics were taken from some representative programs for a large computer. Once the hit ratio characteristics are known, the miss ratio characteristics can be easily determined.

Processor Characteristics

A typical processor activity is characterized as an instruction fetch, instruction decode, data fetch and data operation (Figure 4). Thus, using this model, the average time interval between the issuance of successive memory accesses can be determined. For a more rigorous analysis of the processor behavior characteristics, see Strecker[12].

Performance of the Hierarchy

If λ is assumed to be the average service rate of the first server, then the mean execution interval $1/\lambda$ can be expressed as [Bhandarkar[3]]:

$$1/\lambda = \frac{\text{Hit Ratio}}{\text{Miss Ratio}} \left[t\ (M_p) + t\ (P_c) \right]$$

Where $t\ (M_p)$ = aggregate access time for the primary memory

$t\ (P_c)$ = average processing time between successive memory accesses.

Assuming μ as the service rate for the second server the probability of CPU being busy or CPU utilization is given by:

U = probability of CPU being busy

= 1 - probability (M jobs queued for second server)

$$= \frac{1 - \rho^{M}}{1 - \rho^{M+1}} \qquad \text{(Hiller[6])}$$

Where M = the degree of multiprogramming and

$\rho = \lambda/\mu$

Once the CPU utilization is found, then the figure of merit (f) can be derived as:

$$f = \frac{\text{No. of inst. executed with a given hierarchy}}{\text{No. of inst. executed with all memory substituted by fastest technology}}$$

$$= \frac{t\ (P_c) + t\ (\text{fastest memory})}{t\ (P_c) + t\ (M_p)} *U$$

Where t (fastest memory) = access time of the fastest memory, and U is determined by using the equation given above.

A Memory Hierarchy Design

The final outcome of a memory system design in which a user is interested is its cost and performance. Invariably, the requirements are to minimize the cost while maximizing the performance. The cost and performance of the memory system is a function of the technologies T_1, $T_2 \ldots T_n$ and their sizes S_1, $S_2 \ldots S_n$ used at any level.

Reduction in the cost of the memory system necessitates a small amount whereas increase in performance necessitates a large amount of memory at the lower levels (levels nearer the processor). Therefore a problem encountered in the design of the memory hierarchies is that of finding a mix of memories for different levels in the hierarchy that would give an optimum performance for a given cost.

Assume that a certain cost constraint exists for the design of the primary memory. Also assume that a two level hierarchy of Bipolar and MOS, with sizes S_1 and S_2 respectively, satisfies the cost constraint and places the hierarchy at point A on the hit ratio characteristics (Figure 3). A three level hierarchy of Bipolar, MOS and CCD having the same cost as above is one that has memory sizes of S_1, S_2', and S_3 respectively such that $S_3 = X^* (S_2 - S_2')$, and $X > 1$ is the cost/bit ratio between the MOS and CCD memories. Let these sizes place the memory hierarchy design at Point B on the hit ratio curve (Figure 3). Since $S_3 > S_2$ the primary memory hit ratio is improved. Then the performance of the hierarchy can be determined by finding the hits and the access times for each level.

The following sections will evaluate the effects on different parameters due to the constant cost conversion described above. The various aspects investigated will be the effect on the performance due to:

a) CCD's of different speed and, hence cost used,
b) CPU's of different speed,
c) the change in the degree of multiprogramming,
d) the amount of MOS memory replaced by CCD.

The different parameters for the Bipolar, MOS and CCD memories, the service rate μ of the secondary memory and I/O and characteristics of different CPU's used in the calcualtions are shown in Table 2.

Effect of CCD Speed

The effect of replacing a partial amount of MOS memory by cost equivalent CCD is shown in Figure 5. The percentage improvement in the figure of merit (f) in replacing a two level by a three level hierarchy is shown against the cost of the primary memory per program. The percentage improvement in the figure of merit is defined as:

$$\text{Percentage Improvement in the figure of merit} = \frac{f_2 - f_1}{f_1} * 100$$

Where f_2 = figure of merit for a three level hierarchy

And f_1 = figure of merit for a two level hierarchy

Since the two and three level hierarchies have same amount of Bipolar memory, the cost is represented in terms of the size of the MOS memory/program for a two level hierarchy.

The graph shows that the replacement of MOS by CCD improves f over a wide range. It is advantageous to use slow speed low cost CCD's to replace MOS when the total dollars to be spent on primary memory is low. When the amount of money to be spent increases, then the medium cost medium speed CCD's give a better improvement in f than the low cost low speed, and high cost high speed CCD's. Finally, in the high cost region, high cost high speed CCD's become the most advantageous choice. A little thought will show that intuitively this makes sense. Also, the highest improvement in f is obtained for the slowest CCD's (about 300%), when the MOS primary memory per program is about 16K.

The graph shown in Figure 5 can also be used for constant performance transformation rather than constant cost transformation. The dashed lines on the graph show lines of constant figure of merit, which is an indication of the performance of the processor together with the memory system. Thus, following the same dashed line one can determine the cost savings that are incurred in switching from a two level Bipolar, MOS hierarchy to a three level Bipolar, MOS, CCD hierarchy for various CCD devices. With the particular assumption made in drawing the graph and a figure of merit of $f = 0.100$, if the cost of the MOS memory required for a two level hierarchy is 52K units then the cost of the three level hierarchy for the slowest CCD device ($T_3 = 400\mu sec.$) is 15K units. Thus a cost advantage of about 3.5 times is realized for a constant performance transformation.

Figure 6 shows the advantageous regions for Bipolar, MOS, CCD and Bipolar, MOS combinations as a function of CCD speed and primary memory requirement.