RADC-TR-66-7, Volume I
Final Report

ADVANCED COMPUTER ORGANIZATION STUDY

Volume I - Basic Report

Donald L. Rohrbacher

TECHNICAL REPORT NO. RADC-TR- 66-7
April 1966

Information Processing Branch
Rome Air Development Center
Research and Technology Division
Air Force Systems Command
Griffiss Air Force Base, New York

ADVANCED COMPUTER ORGANIZATION STUDY

Volume I - Basic Report

Donald L. Rohrbacher

FOREWORD

This technical documentary report records the efforts and achievements on the advanced computer organization study conducted by Goodyear Aerospace Corporation, Akron, Ohio. The secondary report number assigned to this document by the company is GER-12314. This report is published in two volumes: Volume One, Advanced Computer Organization Study, Basic Report, and Volume Two, Advanced Computer Organization Study, Appendixes.

The study was conducted for the Rome Air Development Center (RADC) Air Force Systems Command, under Contract AF30(602)-3550, Project 4594, Task 459406. The RADC project monitor was Mr. Fred Dion, EMIIT. The report covers the 14-month period ending 30 November 1965.

Appreciation is extended to Dr. John Holland, University of Michigan, whose consulting services were extremely valuable in both the development and conception of many of the ideas presented. The major contributors to this study wereD. L. Rohrbacher (project engineer), Dr. K. E. Batcher, P. A. Gilmore, and G. W. Lahue. Substantial contributions also were made by G. P. Elliott, Dr. C. C. Foster, and D. C. Gilliland.

This technical report has been reviewed and is approved.

Approved: [illegible]
FRANK J. TOMAINI
Chief, Info Processing Branch

Approved: [illegible]
ROBERT J. QUINN, JR.
Colonel, USAF
Chief, Intel and Info Processing Div

FOR THE COMMANDER: [illegible]
IRVING J. GABELMAN
Chief, Advanced Studies Group

ABSTRACT

Advanced general-purpose computer organizations capable of parallel data processing were studied. To achieve maximum system performance from highly parallel computer organizations, new solution models and programming techniques must be developed. Hence, the following three areas were investigated simultaneously:

1. Applications - Study of problems and their inherent degree of parallelism, and development of theoretical solution models for use on a parallel processor
2. Programming - The programming of parallel solution models on the postulated computer organizations
3. Machine Organization - Development of machine implementations capable of parallel data processing

This study resulted in the design of two computer organizations (designated Machine I and Machine II) capable of parallel data processing and fast sorting and table searching in memory. These machine organizations were possible because of the development of a special memory that permits many processing and input-output units to access memory simultaneously without conflict.

The applications effort was focused on the development of solution models which exploited the maximum amount of parallelism resident within a problem. Two major problems were investigated: a dynamic programming problem, and parallel compilation.

Detailed programs were written for the dynamic programming problem on Machine I and a parallel compilation algorithm on Machine II. These same problems also were programmed on the IBM 7090 to provide a standard of comparison. In both cases, the parallel processing capability of the machines afforded significant increases in speed of program execution.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

SECTION I - INTRODUCTION

The next major advance in computer capability will result from radical changes in basic computer organizations rather than from increased computing speed. These new computers will be highly parallel machines capable of performing many different operations simultaneously. Such advanced computer organizations will necessitate basic changes in programming and structuring of solutions to problems. Under Contract AF30(602)-3550, Goodyear Aerospace conducted a 14-month study and investigation of such computer organizations. Advanced general-purpose computer organizations capable of parallel data processing were investigated. This report presents the approach, results, and conclusions encompassing the application, programming, and machine organization aspects of the study.

Section II summarizes the program including some of the more significant results and conclusions. The remainder of this volume presents the major efforts in greater depth.

Volume Two contains the appendixes, which detail the various study efforts. The appendixes are referenced at the appropriate points in Volume One.

SECTION II - SUMMARY

1. APPROACH

The primary objective of this program was the study and development of advanced computer organizations. The study resulted in the design of a general-purpose computer capable of parallel data processing. To achieve maximum understanding of parallel processing and the computer configurations required to achieve it, and also to avoid generalizations concerning parallel processing, the study effort was focused on the development of only two machine organizations (Machines I and II). Furthermore, only two problems (dynamic programming and parallel compilation) were analyzed in depth, with parallel solution models being developed and detailed computer programs written.

To achieve maximum system performance from highly parallel computer organizations, it is necessary to develop new solution models and programming techniques. Hence, the following three areas were investigated simultaneously:

1. Applications - Study of problems and their inherent degree of parallelism, and development of theoretical solution models for use on a parallel processor
2. Programming - The programming of parallel solution models on the postulated computer organizations
3. Machine Organization - Development of machine implementations capable of parallel data processing, with modifications made in accordance with the developments in the applications and programming areas

2. MACHINE ORGANIZATION

The largest problem in the design of a computer capable of parallel data processing was the design of the communication facility between the processors. Ideally, each processor should be allowed to communicate with any other processor. Machine organizations such as the SOLOMON approach the communication problem by allowing each processor to communicate with only a small number of the other processors. However, this technique requires the structuring of problems so that the required communication is along the paths built into the machine. Since some problems cannot be structured to fit the particular machine, the use of such machines is restricted.

Two sorting techniques developed at Goodyear Aerospace provided the basis for a practical solution to this communication problem. Special sorting and merging-separating networks based on these techniques allow any processor to communicate with any other. The basic element of sorting and merging networks is a comparison element. Such elements accept and compare the magnitude of two input words and order the words on the output. These networks made possible a memory organization having the following characteristics:

1. The contents of the memory are maintained in numerical order
2. It has a content-addressing capability
3. It permits many processors and I/O units to access memory simultaneously without conflict (for example, 1024 simultaneous accesses for a 32,000-word memory)

The Machine I organization utilizes this memory in conjunction with many processing units and I/O devices. The resulting computer has the following features:

1. Parallel data processing capability

2. Flexible communication between any processors
3. Parallel I/O channels
4. Freedom of processor assignment (no need to reprogram if processors are added, if a processing unit fails, or if other programs are run simultaneously)
5. Fast sorting and table searching in memory

Machine II was designed using the same basic sorting and merging-separating networks as Machine I. However, an additional sorting network, called the multiprocessor control (MPC), was added to assign tasks to the processing units. A block of instructions is read into the MPC and all instructions that can be executed simultaneously are automatically assigned to processing units for execution. Machine II has the same basic capability as Machine I plus the added advantage of the MPC. However, the processor assignment capability possessed by the MPC is restricted to those sets of instructions that can be fitted to a treelike structure.

The development of Machine I and Machine II comprised the major portion of the machine organization effort. However, some embryonic ideas were developed for an organization intended primarily for parallel nonnumeric processing. It was found that the characteristic distinguishing numeric from nonnumeric problems is the addressing of operands. In a numeric problem, most operands are addressed by their unique labels; in a nonnumeric problem, most operands are addressed by their properties (attributes). These two methods can be called "explicit addressing" and "implicit addressing," respectively. Sorting and merging-separating networks possess the kinds of properties needed for implicit addressing.

3. APPLICATIONS

Efforts in the applications area were directed toward the analysis and development of solution models suitable for implementation on a parallel

SECTION II - SUMMARY

processor. Problems of contemporary interest were examined to determine the degree of parallelism resident within them. Problems were restructured to the extent that greater parallelism could be achieved. Solution models were then specified to exploit the maximum amount of parallelism resident within a problem consistent with the parallel processor configurations developed under the machine organization area of the study.

Dynamic programming was the major numeric problem area examined. This is a mathematical technique devised by Richard Bellman for solving certain types of maximization problems. Analysis of the technique revealed the existence of extensive inherent parallelism. This was exploited in a solution model that specified an extensive restructuring of the technique. The restructured solution model made possible significant gains in speed of execution on a parallel processor.

Several other numerical problems were studied, including Jacobi's method of eigenvalue determination, relaxation solution for a system of linear algebraic equations, and numerical solutions to Laplace's equation. Each technique possessed sufficient inherent parallelism to make good use of the parallel processing capability in the machine organizations.

The parallel compilation of higher programming language statements was chosen as an example of a nonnumeric problem for major consideration. This choice was quite natural since investigation into the structure of parallel processor configurations and parallel execution of coded routines unavoidably led to the consideration of parallel compilation. Parallel compilation would permit all processing units not being used in execution of programs to be utilized in the compilation of the next programs to be executed. This would maximize the utilization of the parallel processor's hardware.

It was found that not only could many statements be compiled simultaneously, but parallelism also could be exploited in the compilation of each individual statement. Several parallel compilation algorithms were developed.

A short effort was directed toward the development of a programming language designed specifically for a parallel processor. The goal was to develop a language that could compactly denote the execution of parallel operations and that would allow, and indeed promote, ease of conceiving and expressing the structure of parallel solution models. The work could provide the basis for future expanded program effort.

4. PROGRAMMING

The final proof of the usefulness of any computer organization rests with the programmer. In this study, two parallel processor programs were written. The parallel solution model for the dynamic programming problem was programmed on Machine I and a parallel compilation algorithm was programmed on Machine II. In addition, sequential solution models for the same problems were programmed on the IBM 7090 computer to provide a comparison. Since the primary objective was to determine the needed machine capabilities and programming techniques, no attempt was made to optimize the programs and extract maximum parallelism from the problems.

Problem solution time for the dynamic programming problem on Machine I was 16 msec as opposed to 150 to 220 msec for the sequential machine. Therefore, solution time on Machine I was 9 to 14 times faster than the IBM 7090 computer. This can be attributed to the availability of the many processors capable of independent and simultaneous action on the contents of any word in the multiaccess sorting memory.

Processor loading reached a peak of 60 processing units out of a possible 512 proposed available processors. The average number of processors for the problem execution time was only 22.

Had the problem been sufficiently large to use the full 512 processors at the peak period, Machine I would have had a speed advantage of 76 to 119 to 1. In addition, many processors would have been available at nonpeak times for other uses, such as compiling.

The execution of the parallel compilation algorithm for Machine II required 53.6 msec. Considering the number of statements to be compiled as N, then the ratio of the compilation time of the IBM 7090 to that of Machine II is N to 27. The parallel compilation time is independent of the number of statements and will remain relatively constant at 54 msec. The time for sequential compilation increases linearly with the number of statements. When the number of statements exceeds 27, then the parallel processor time for compilation is less than that for the IBM 7090 computer. If 256 processors are available, then Machine II can average 219 statements every 54 msec. The speed advantage would then be 219/27 = 8 to 1 over the IBM 7090. It is believed that if time had permitted one of the other parallel compilation algorithms to be programmed, additional speed advantage would have been realized.

5. COMPARISON OF MACHINES I AND II

Both Machine I and Machine II are composed of sorting and merging-separating networks and processing units. Each has a large merging-separating memory; Machine II also has a smaller full-sorting memory (MPC). This additional Machine II hardware is offset by the fact that the Machine I processing units are much more complex. The type of program parallelism most economically implemented on Machine I consists of the parallelism existing between independent blocks of a program. Very short strings of independent instructions can be executed in parallel but usually the cost of setting up indices to maintain control is prohibitive.

In contrast, Machine II exhibits parallel execution capability on the same levels as Machine I and in addition recognizes instruction level parallelism independent of the programmer. Any independent strings of instructions are automatically executed as soon as the required operands are present. However, speed is sacrificed to obtain this additional capability provided for Machine II by the MPC. On a strictly sequential program, Machine I can obtain and execute two instructions every 30 μsec, while Machine II has an instruction execution time ranging between 13.8 and 96.6 μsec, depending on the instruction.

The basic processor of Machine I considered as an entity is not unlike the processors found in contemporary sequential machines. It has the normal arithmetic, quotient, and index registers and access capability to any word in memory. Hence, with some small degree of effort, an existing program could be, with modifications consistent with changing from one machine to another, run on Machine I with only one processor assigned. Machine II would require more extensive reprogramming of the existing program and, generally, it is expected that the program would have to be significantly revised to take advantage of the parallelism within the machine.

6. RECOMMENDATIONS FOR FOLLOW-ON EFFORT

Future work on this program should be carried out with the machine organization effort independent of the applications and programming effort. While many facets of the machine organization work need further study, the greatest knowledge can be obtained if the programming and applications efforts are not forced to be continually modified as a result of machine changes. Furthermore, Machines I and II as they are currently defined have many capabilities that have not been utilized and hence it is believed that the applications and programming effort should focus attention on how to exploit these existing capabilities.

Items suggested for further study include the following:

1. Machine organization
    a. Parallel nonnumeric processor
    b. Parallel I/O
    c. Interrupts
    d. Priorities
    e. Multiaccess systems
    f. Other applications of machine concepts
    g. Feasibility model
2. Applications and programming
    a. Macro instructions

b. Flow diagramming techniques

c. Program comparison problems

d. Compiler

e. Library subroutines

SECTION III - MACHINE ORGANIZATION

1. INTRODUCTION

Multiprocessors that possess a flexible communication structure can be built. The need for such structures is discussed and the networks affording this capability (sorting networks) are described. Three machine organizations using these networks are presented. They differ mainly in their machine language structure; these differences affect the utilization of the flexibility in communication afforded by the hardware.

2. FLEXIBLE INTERCOMMUNICATION

The largest problem in the design of any multiprocessor is the design of the communication facilities between the processors. If two or more processors are working on the same problem, they must be able to communicate with each other. One technique is to allow each process r to communicate with only a small number of the other processors (limited intercommunication). The common form employed is a two-dimensional array in which each processor can communicate with its neighbors to the right, left, up, and down. This technique gives rise to the question of structuring problems "space-wise" so that the communication required in the problems is along paths built into the machine. Some problems cannot be structured to fit the particular machine, and thus this technique is good only for limited-purpose machines.

Another technique is to allow each processor to communicate with any other processor (flexible intercommunication). The largest problem here is to do it without an inordinate amount of hardware. One solution is to build what looks like a telephone exchange and allow only a small number of the processors to converse at any one time. This gives rise to the question of structuring problems "time-wise" so that at no time will problems require more conversations than the machine will handle. Usually, this is so complex that no consideration is taken of it in the programming of problems; any tie-ups in the communications that occur are accepted as a fact of life.

These considerations lead to a study of networks to see if any networks exist which, without an inordinate amount of hardware, will allow n input lines to be connected to n output lines with any permutation (n may be in the hundreds or thousands). Such a network would allow flexible intercommunication without communication "tie-ups," as discussed below.

3. SORTING AND MERGING-SEPARATING NETWORKS

Any network capable of sorting (arranging in numerical order) a set of data can be used as a flexible communication network. The input lines send in items of data tagged with output-device addresses. The network sorts the data on the tags and the devices on the output lines read their respective data items. Two sorting techniques developed at Goodyear Aerospace, odd-even sorting[1,a] and bi-tonic sorting (see Volume Two, Appendix V), have the right kind of characteristics for this problem.

The basic element of a sorting network and a merging network is a comparison element. This element has two inputs and two outputs (see Figure 1). When two numbers are applied to the inputs, the element compares their values and presents the higher-valued number on its H output and the lower-valued number on its L output (if they have equal values, their common value is presented on both outputs). Many different realizations of a comparison element are possible, including the 13-NOR realization shown in Figure 2, which operates with data transferred serially, and the parallel transfer realization described in Volume Two, Appendix VII.

A sorting network is built up from merging networks by the well-known "sorting-by-merging" technique; for example, to sort eight items, first

[a]Superior numbers in the text refer to items in the List of References.

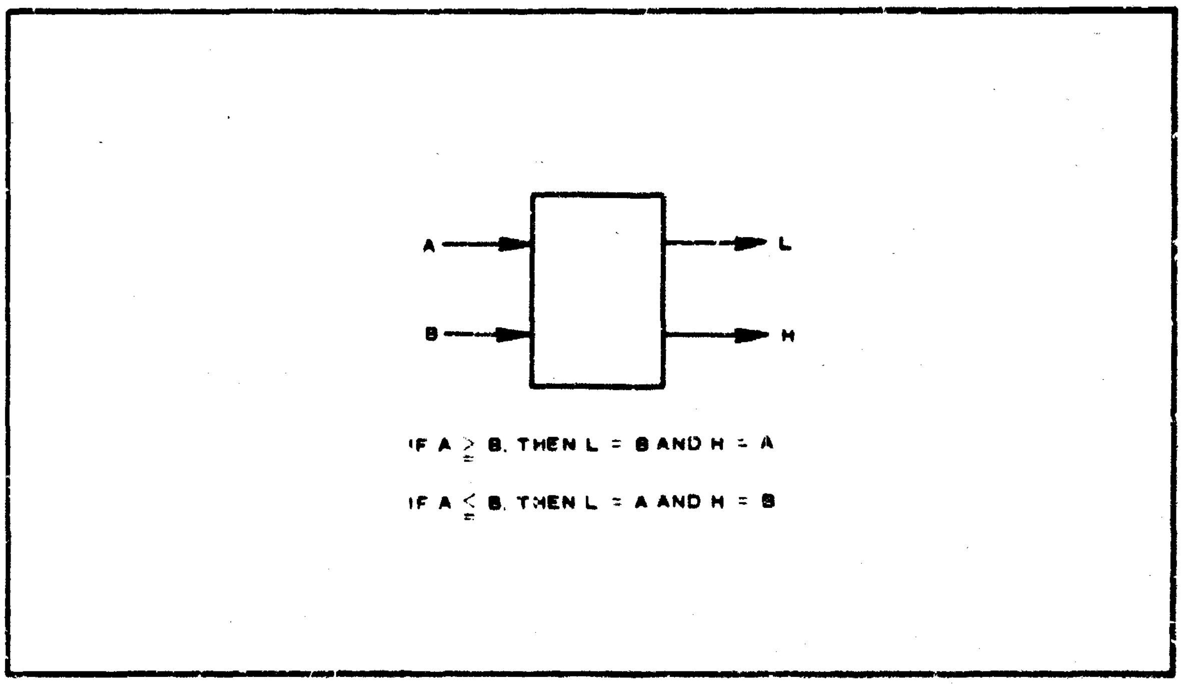
A
B
L
H
IF A ≥ B, THEN L = B AND H = A
IF A ≤ B, THEN L = A AND H = B

Figure 1 - Symbol for a Comparison Element

compare successive pairs to form four ordered lists of length two, then merge these lists two at a time to form two ordered lists of length four, then merge these to form one ordered list of length eight. A merging network can be constructed from comparison elements using the odd-even technique[1] or the bi-tonic technique of Appendix V. The construction of odd-even merging networks and the construction of bi-tonic networks are described in Item 3 of Appendix VI. Table I illustrates various characteristics of merging networks and sorting networks constructed by these two techniques.

As an example, a sorting network for 1024 items would have 55 levels and 24,063 elements if built using the odd-even technique. A comparison element such as that of Figure 1 should realize a delay time of less than 100 nsec so a 1024-item sorting network from these elements would sort in less than 5.5 usec. This illustrates the main characteristic of these sorting networks: large sets of items can be sorted quickly, which makes them practical for the communication networks of multiprocessors. The

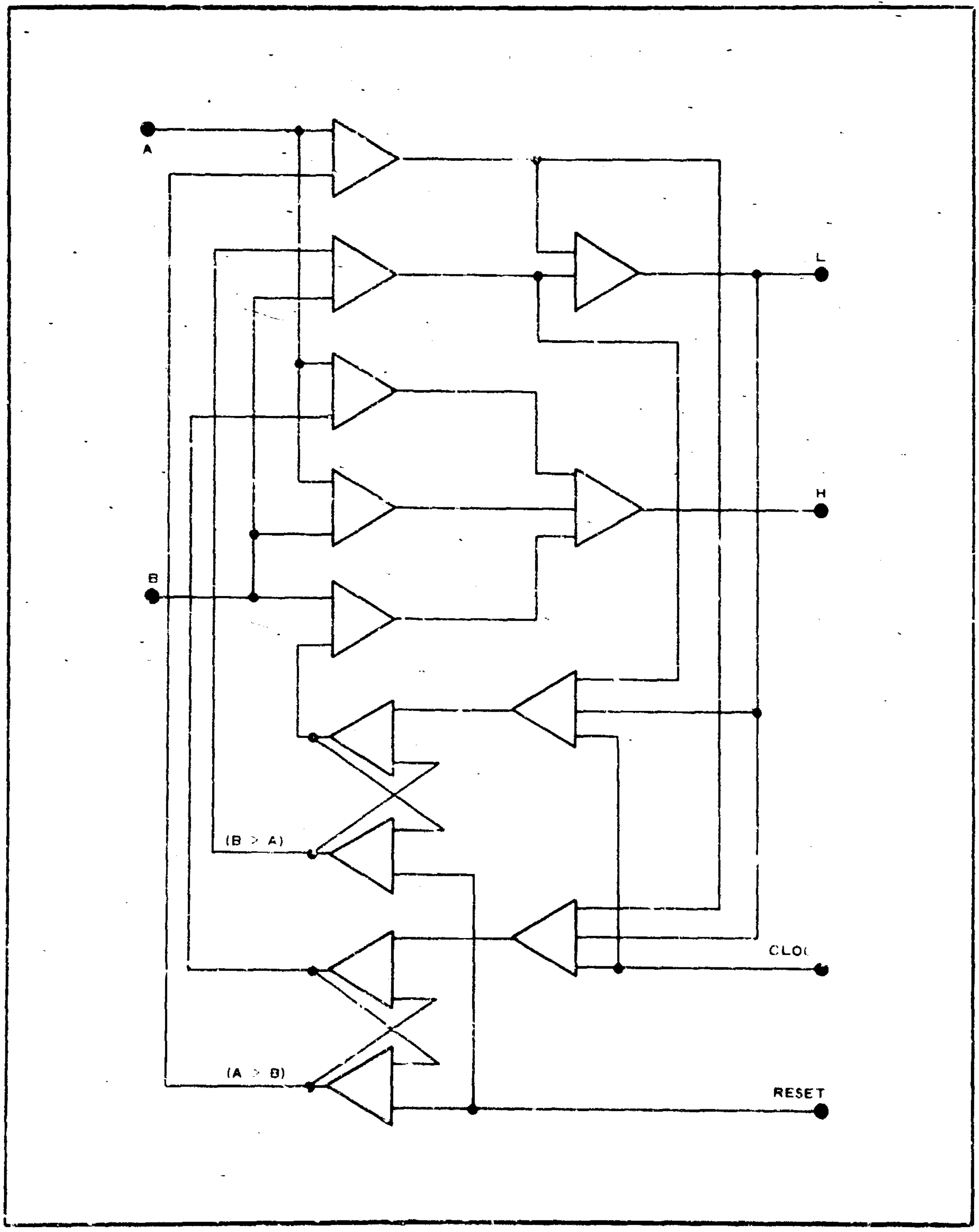

Figure 2 - A 13-NOR Comparison Element

TABLE I - CHARACTERISTICS OF MERGING AND SORTING NETWORKS

Characteristic	Odd-even	Bi-tonic
Levels required to merge 2^p items with 2^p items	$p + 1$	$p + 1$
Comparison elements required to merge 2^p items with 2^p items	$p2^p + 1$	$(p + 1)2^p$
Levels required to sort 2^p items	$p(p + 1)/2$	$p(p + 1)/2$
Comparison elements required to sort 2^p items	$(p^2 - p + 4)2^{p-2} - 1$	$(p^2 + p)2^{p-2}$

other characteristic is the amount of hardware involved. The only other well-known network that will flexibly connect 1024 inputs to 1024 outputs is a 1024 by 1024 crossbar with 1,048,576 cross points; the 24,063 elements of a sorting network compares quite favorably.

When a multiprocessor is considered, there is the additional requirement of buffering on the communication network. Without buffering, the programming of problems would be complicated by the need for planning each transfer carefully so no two processors would want to send data to a third processor simultaneously. This leads to consideration of using the communication network as the main machine memory. By this means, not only is buffering provided but also freedom in processor assignment. It would be no longer necessary to know which processor is doing what; to transfer a variable from one processor to another, one processor stores the variable in memory with a given label and the other processor reads it by referring to the label. Neither processor needs to know the location of the other. Freedom of processor assignment allows a program to be written without worrying which processors are unavailable because of other concurrently operating programs or because of processor malfunctions.

Discussed here is modification of a sorting network to form a multiaccess memory, and construction of a simpler device, a merging-separating memory. Storage capability can be added to a sorting network simply by adding it to the comparison elements. For instance, the 13-NOR comparison element of Figure 2 can be modified by including a shift register stage in each of its outputs (or inputs). When a sorting network is constructed out of these modified elements (an odd-even sorting network would require extra stages in certain places to equalize delays), the result is a set of shift registers interconnected so that their respective contents are arranged in order while being shifted.

A diagram of a multiaccess memory is shown in Figure 3. The output of the sorting network travels through some control logic elements into a set of shift registers and then to the input of the sorting network. A typical memory word has the following format.

ADDRESS	11	DATA

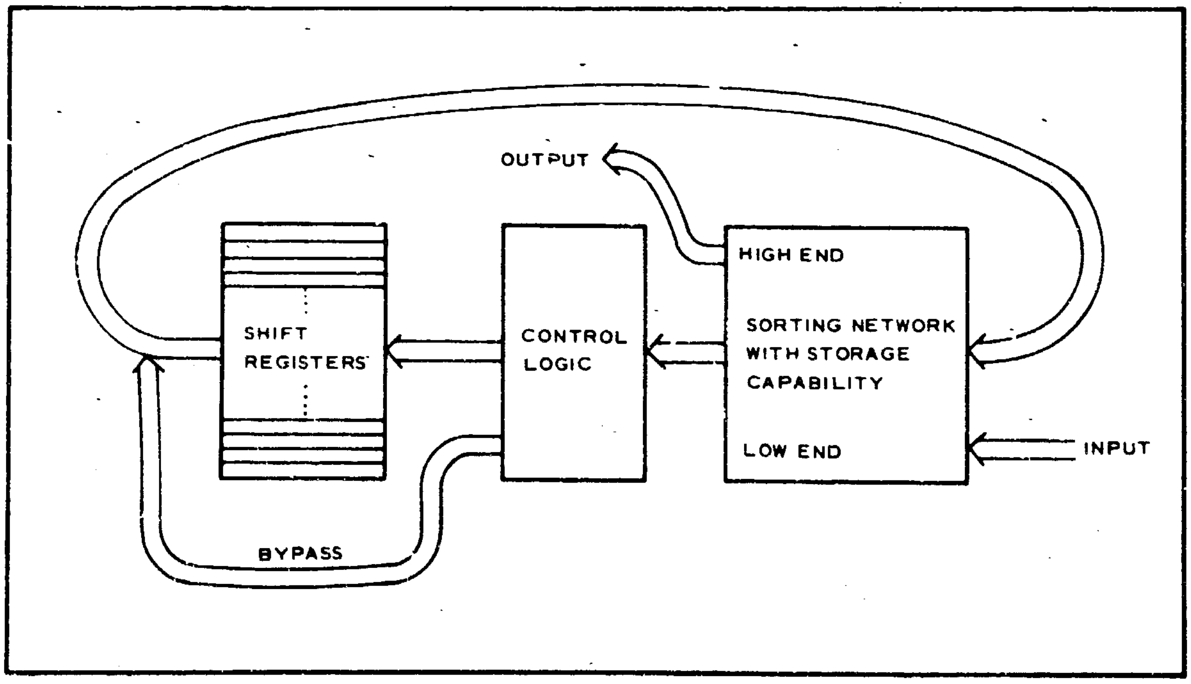

Figure 3 - Diagram of a Multiaccess Memory

Empty words are cleared to zeroes and no addresses are used above a certain limit (these addresses will be used in reading as explained below). The shift registers are long enough to handle the address field. The sorting action sorts th words by their addresses with all empty words accumulating at the low end of memory. To write a word, one of the input lines interrupts the recirculation of one of the empty words and substitutes an address and item of data in the format shown above. The sorting action will then place it in correct relation with the other words in memory. Many writes may take place simultaneously.

To read a word, one of the input lines sends in a read request or a read and erase request. These have the following formats.

ADDRESS TO BE READ	10	OUTPUT LINE	0 ——————— 0	*READ REQUEST*

ADDRESS TO BE READ	01	OUTPUT LINE	0 ——————— 0	*READ AND ERASE REQUEST*

The sorting action moves the request to a position just below the word requested. When the 10 or 01 pattern following the address field passes through the control block, the following action occurs:

1. The sorting network output of the word above the request is switched to the shift register of the request (this causes the data field of the word requested to replace the request's data field)

2. The sorting network output of the request bypasses the shift register (substituting the output line code for the address)

3. If the request is a read and erase and the addresses agree, the output of the shift register of the word above the request is gated off (substituting an empty word for the memory word)

4. If the addresses agree, a 11 is substituted for the original request code.

If more than one request is below a memory word, somewhat similar action occurs to transfer the memory word data field to all requests. As a result of this action, the request is modified to look as follows:

OUTPUT LINE	11	DATA FIELD OF MEMORY WORD

Output line codes have higher values than addresses so the request travels to the high end of memory on the succeeding movement through the sorting network, arriving at one of the output lines. To make sure it gets to the correct output line, a read request is entered each memory cycle for each output line whether the particular line wants to read or not; this guarantees that a request will appear on each output line every cycle and that the sorting action will arrange them in numerical order. Many reads may be taking place simultaneously along with many writes.

This memory has some characteristics that make it different from normal computer memories:

1. It has many input lines and many output lines, all reading and writing without queuing problems
2. More than one word may be stored at a given address. Instead of overwriting an old word, a new word overwrites an empty word. Words at the same address will be ordered by their data fields and a read request to the address will read the least-valued word.
3. Some addresses may be lacking words.

Characteristic (2) is valuable for sorting a set of items; they need only be stored with the same address. Characteristic (3) can be used to synchronize several processors working on the same problem; if processor Y expects an item of data with label Z from processor X, it need only look for address Z to see if processor X has stored it. Only new words, read requests, read and erase requests, and erased words are changing places in the memory; most of the sorting capacity of the network is not being used. This consideration leads to investigation of simpler networks.

If the read requests, the read and erase requests, and new words are ordered before entering memory, then only a merge is needed to combine them with the ordered-memory words rather than a complete sort. To read out, a means is required for moving all requests to one end of memory after they have read their respective memory words; this is called separating. A memory using these techniques is called a merging-separating memory.

Merging causes words at one end of memory to be "sprinkled" throughout memory and separating causes words "sprinkled" throughout memory to be collected at one end of memory; this suggests that the topology for a separating network should be the inverse of the topology for a merging network. Item 3 of Appendix VI describes separating networks based on the inverses of bi-tonic merging networks.

Item 3 of Appendix VI also describes a multiaccess merging-separating memory with serial data transfer. Generally speaking, new words and requests are sorted and then merged with the memory words. Words to be separated out (read requests and erased words) are flagged and then separated out by a separating network. The read requests and erased words are sent to another separating network, which splits these two sets. The read requests are then sorted with respect to output channel codes and sent out. Figure 4 shows the block diagram.

Items 2 and 3 of Appendix VII describe parallel merging-separating memories. Words are transferred in parallel rather than serially, leading to about a 2-to-1 speed advantage over serial memories. The amount of equipment in a parallel memory is greater than that of a serial memory

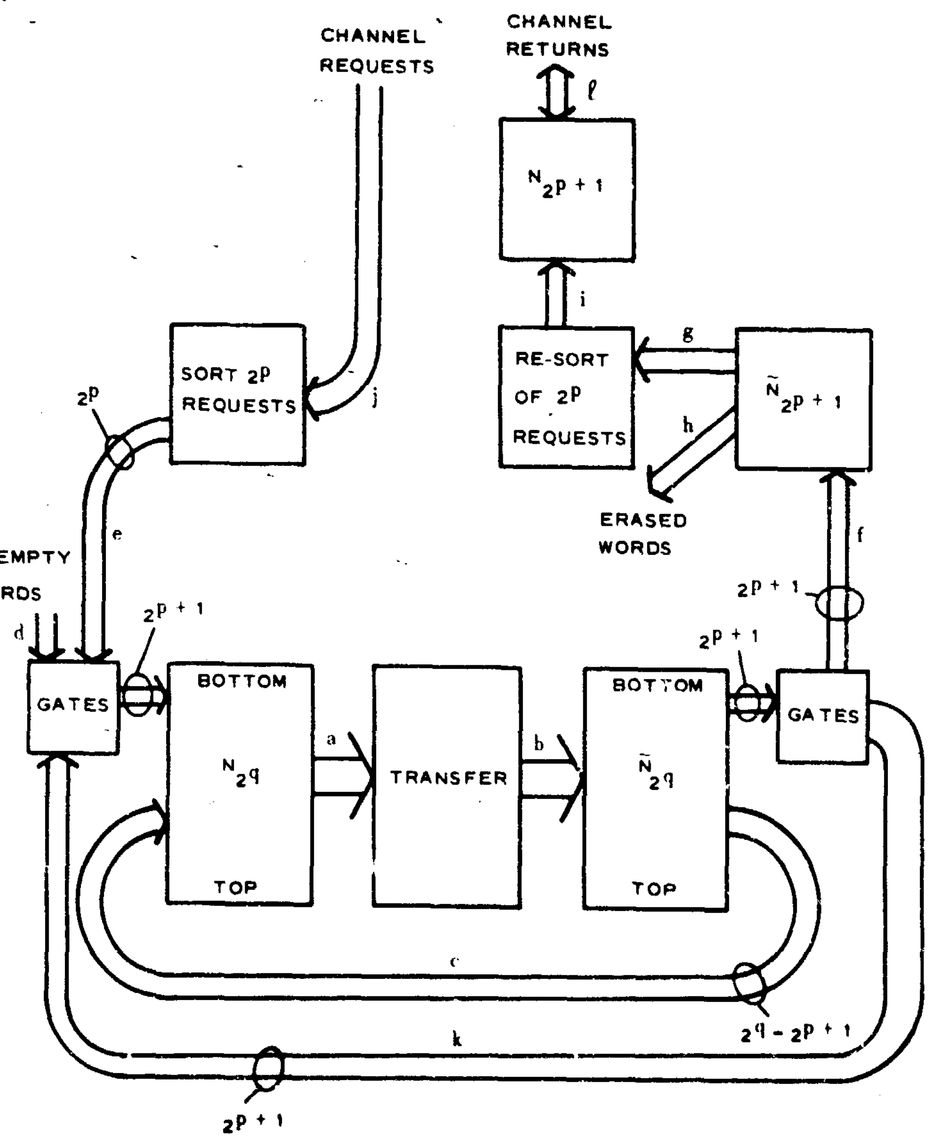

Figure 4 - A Multiaccess Memory with $2^q - 2^{p+1}$ Words and 2^p Requests

but not by a large factor. There are more kinds of elements than in the serial version. A comparison between serial and parallel memories should include wiring studies as this may be the determining factor in a choice between them.

Either version of a merging-separating memory is faster than a sorting memory of the same capacity because merging is faster than sorting. To illustrate this, consider a 32,768-word memory with 1024 access lines. A sorting memory would require 120 steps to sort. Two sort cycles are required to read a word: one to get the request to the word, one to get the request from the word to the output channel. Words could be entered each sort cycle so the memory cycle time is 120 steps and the access time is 240 steps. A parallel merging-separating memory requires 55 steps to sort the 1024 new items, 15 steps to merge, 15 steps to separate out erasures and read requests, 11 steps to separate erasures from read requests, and 55 steps to sort the 1024 read requests by channel number. This is a total of 151 steps for accessing. The cycle time of the parallel merging-separating memory includes 30 steps: 15 to merge, 15 to separate. If the step times were equal, the parallel merging-separating memory would have a cycle time 1/4 that of the sorting memory and an access time 151/240 that of the sorting memory. These figures are not quite true because separating steps are longer in the merging-separating memory; circuit studies using typical integrated-circuit modules indicate a 2-to-1 ratio between separating and other steps. Using this result, an advantage of 2.67 to 1 is obtained in cycle time and a 1.44-to-1 advantage in access time for the merging-separating memory versus the sorting memory.

Hardware requirements of the three memories (sorting, serial merging-separating and parallel merging-separating) are not very different. All three require a shift register stage for each bit and this seems to be a major cost item. Wiring studies probably will show the largest differences in cost between the three types. Machine organizations using these memories are discussed below.

4. MACHINE I ORGANIZATION

The block diagram of Machine I is shown in Figure 5. A multiple-access merging-separating memory as described previously forms the main machine communication network and memory. Memory storage may be backed up with disk storage, tape storage, core storage, etc., connected to the I/O channels along with other I/O devices. A number of processors (100 to 1000) are connected to other memory input and output channels, each using three channels (permitting a processor to get two operands and a next instruction each cycle time).

I/O control is obtained as follows. Each channel is given a unique address in memory into which processors store I/O control words tagged with priority fields. A nonactive I/O device interrogates its address periodically until it obtains a control word from its queue, at which time it performs the desired operation.

Control of the processors is obtained as follows. The first word of each

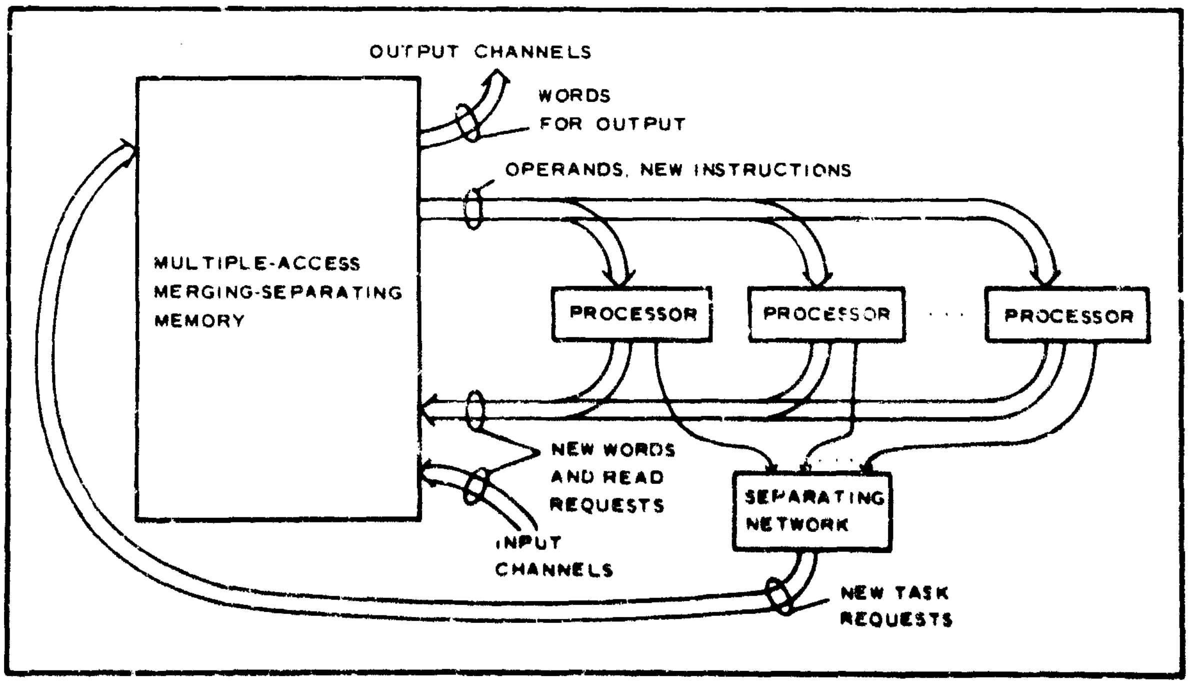

Figure 5 - Block Diagram of Machine I

task to be performed is stored in the highest memory address. Each word may have two instructions, the second of which is a jump to the program to be performed. Any inactive processor sends its code to the separating network (see Figure 5), which gathers all such codes at one end and transmits them to the high end of memory over the new task request channel. Each request interrogates a memory word and if its address is all ones it flags the word and inserts its code. The word will then fall out of memory in the separating phase and be sent to the processor. By this means, many processors can be started simultaneously on tasks waiting for execution.

More discussion of Machine I appears in Appendix VI, Items 4 and 5, and Appendix IV.

Basically, Machine I has the following features:

1. Parallel processing capability flexibly interconnected (any unit can read or write into any memory address so any structure between processors can be implemented)
2. Parallel I/O channels
3. Freedom of processor assignment (no need to reprogram if new modules are added, if modules fail, or if other programs are started)
4. Fast sorting and table searching in memory

5. MACHINE II ORGANIZATION

Machine II was designed to have improved facilities over Machine I for dividing tasks into subtasks and for communicating between subtasks. In Machine I, a task starts subtasks by storing "task control words" in the highest memory address to be fed to available processors. In many problems, it can be expected that several subtasks will be sharing a common program and working with different data.[a] The two instructions in a task control word are used for this purpose; one can specify a data index and the other the start of the program. When a subtask is running, any communication between it and other subtasks and any communication with its previous intermediate results takes place using memory addresses. If several subtasks share a common program, the unique index in each subtask is used to assign unique addresses to each subtask for this purpose. This leads to housekeeping problems, especially when the exact number of subtasks is unknown. The effect of this problem depends strongly on the amount of communication required in the subtask either with itself (previous intermediate results) or with other subtasks, and in some cases it may be severe enough to negate any time saved by "paralleling" the program.

An example showing this problem follows. Let there be a vector of length n containing a set of numbers, $x_1, x_2, \ldots, x_n$, in continuous memory addresses. It is desired to compute e^{x_i} for $1 \leq i \leq n$ and to store these values in another n-vector, $z_1, z_2, \ldots, z_n$. The value of n and the starting addresses are given to the program at execution time. A seventh-degree polynomial approximation to e^x is selected:

$$e^x \approx A + x(B + x(C + x(D + x(E + x(F + x(G + Hx)))))) .$$

One way to compute this is to use the familiar multiply-add iteration. This takes 14 steps. One program suffices with an index register taking care of the different x_i's and z_i's. Another way is to compute the polynomial rearranged as follows:

$$e^x \approx (((A + Bx) + C(x^2)) + x^3(D + Ex)) + x^5((F + Gx) + H(x^2)) .$$

[a]This situation arises in a multiprocessor program for the same reasons that a loop arises in a sequential processor program. When each member of a set of data has to be covered by a basic set of operations, the programmer finds it convenient to write one program and cover the set by changing indices. With a sequential processor, the program is applied to one member at a time; with a multiprocessor, the program may be applied to all members simultaneously.

Figure 6 shows how it can be computed in five steps using more than one processor. However, this will be difficult to apply in Machine I because of the amount of communication between the various parts; each variable to be communicated requires a unique address associated with the index, i. Housekeeping, store, and load steps would have to be added to the steps of Figure 6, negating much if not all the time savings. There is also the problem of reserving enough memory addresses for the intermediate results. The first way would be the only practical method on Machine I.

It can be expected that many arithmetic processes and other processes rising in practice will have "flow diagrams" similar to that of Figure 6; thus, a machine that could handle them without the difficulties in memory addresses of Machine I would have more utility. Machine II was designed with this in mind.

The basic machine language of Machine II was selected to allow easy programming of diagrams such as Figure 6. Note that every operation in Figure 6 receives two inputs while an output may be used in anywhere from one to six different places. This is an example of the fact that all normal computer operations involve one or two operands. The basic instruction format is:

NUMBER	OPERATION CODE	FIRST OPERAND	SECOND OPERAND
		(A)	(B)

The number field contains the label for the instruction, the operation code specifies what to do with the operands, and the two operand fields contain the number fields of the instructions that generated the operands (the result of any instruction is tagged with the number of the instruction so other instructions may refer to it). As an example, a program for the computation of Figure 6 is given in Table II. Note that there may be gaps in the

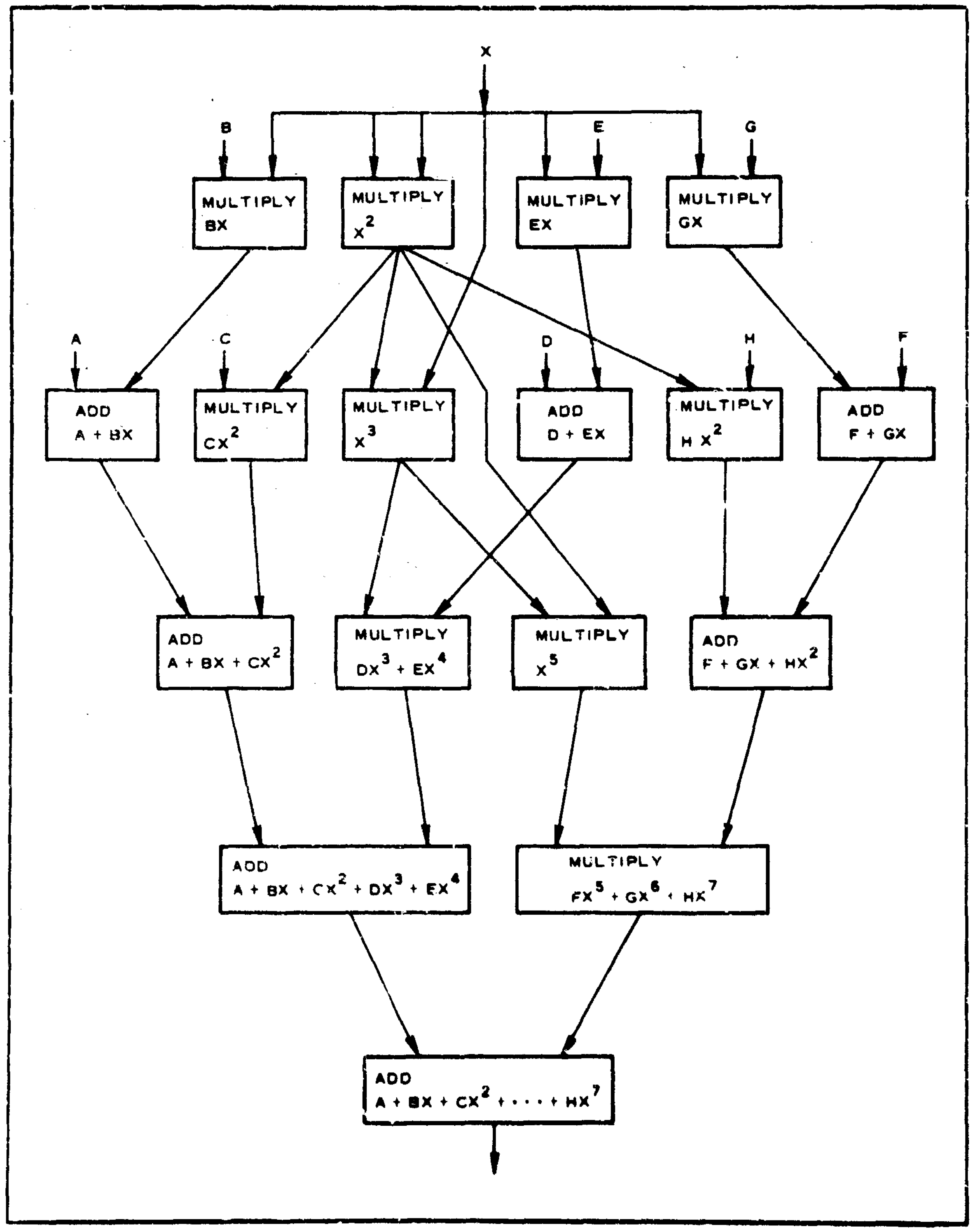

Figure 6 - Parallel Computation of a Seventh-Degree Polynomial

TABLE II - SAMPLE PROGRAM FOR PARALLEL COMPUTATION OF SEVENTH-DEGREE POLYNOMIAL

Number	Operation code	A	B	Results
1	. . .	. . .	. . .	x
2	ADD	37	38	$A + Bx + . . . + Hx^7$
5	MPY	1	40	Bx
6	MPY	1		x^2
7	MPY	1	43	Ex
10	MPY	1	45	Gx
11	ADD	39	5	A + Bx
13	MPY	41	6	Cx^2
17	MPY	1	6	x^3
20	ADD	42	7	D + Ex
23	MPY	46	6	Hx^2
25	ADD	44	10	F + Gx
27	ADD	11	13	$A + Bx + Cx^2$
31	MPY	17	20	$Dx^3 + Ex^4$
33	MPY	6	17	x^5
35	ADD	25	23	$F + Gx + Hx^2$
37	ADD	27	31	$A + Bx + . . . + Ex^4$
38	MPY	33	35	$Fx^5 + Gx^6 + Hx^7$
39	. . .	. . .	. . .	A
40	. . .	. . .	. . .	B
41	. . .	. . .	. . .	C
42	. . .	. . .	. . .	D
43	. . .	. . .	. . .	E
44	. . .	. . .	. . .	F
45	. . .	. . .	. . .	G
46	. . .	. . .	. . .	H

numbering and the numbering need not correspond to the order in which instructions are performed (Instruction 2 is performed last, for example).

Machine II is designed to look at a block of instructions such as that in the foregoing paragraph and to perform them in the correct order, executing many simultaneously if processors are available and the program admits it. Thus, a programmer can introduce parallelism in a task without the bother of task control words. To do this, it is necessary to interpose a multiprocessor control unit (MPC) between the processors and the memory (see Figure 7).

A program block is defined as a set of 1 to 256 instructions stored at one memory address (the instructions including the number fields are written in the data fields of the memory words). A program block is executed whenever it is read into the MPC. Parallel channels (1024) between memory and the MPC permit reading of several blocks simultaneously and at any one time the MPC may contain several blocks, each in various

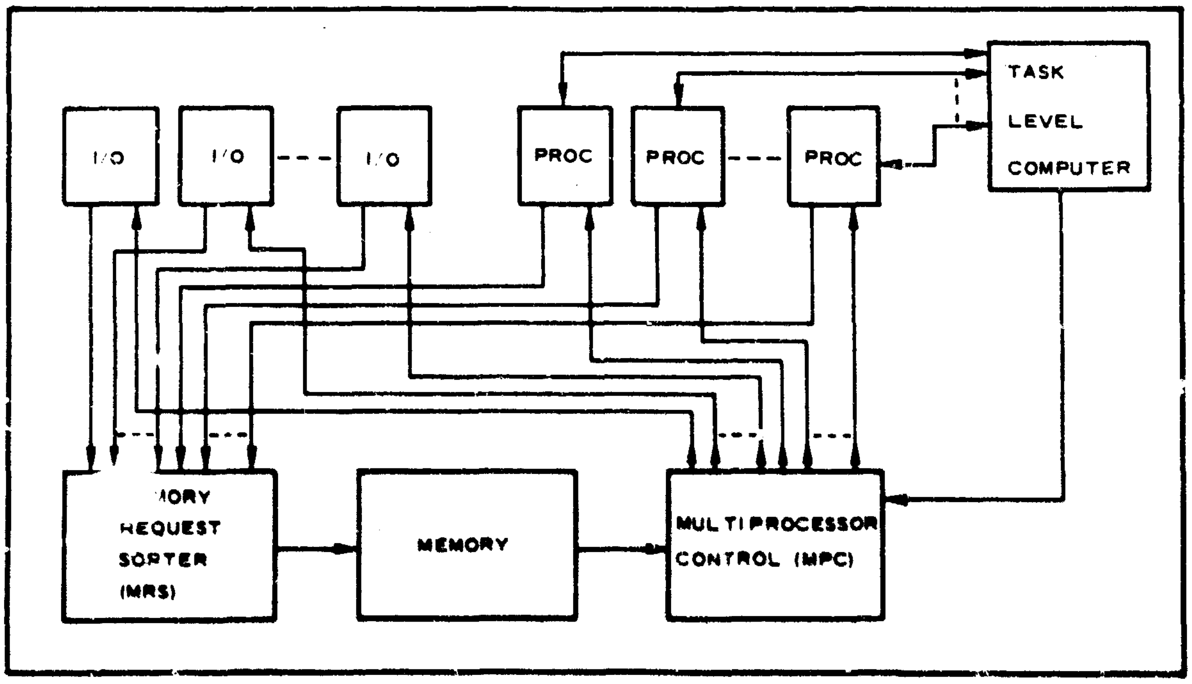

Figure 7 - Block Diagram of Machine II

stages of execution. When executed, the operand fields of the instructions refer to results within the same block (except for a few special operations).

A block is read into the MPC by a "start" instruction in a block already in the MPC which specifies its memory address. A block may have several copies of itself in the MPC at any time; each copy is treated as an independent unit so operand fields in each copy refer to results in the same copy.

When a Block A reads in a Block B through a start instruction, there may be some variables it wants to transmit to B. Operations SPB (Shift Previous Block) and SPR (Shift Previous Relative) in Block B can read results in A for this purpose. Return transfers of data can be obtained with BRG (Bring) operations in A to read results in B. These operations allow A and B to communicate with each other in the MPC without memory references, reducing the problem of unique memory addresses discussed previously. When Block A has several start instructions, each reads in a separate block. The blocks it reads in may contain more start instructions to read in other blocks, etc.

Thus, at any time, a program in Machine II will have the structure in the MPC of one or more trees (see Figure 8 for an example). The program started with one block (read in by the supervisory program), which started others, etc. A completed block (all instructions performed) is dropped from the MPC when all connecting blocks have completed all data transfers from it, leaving room in the MPC for new blocks. Interblock communication in the MPC is only allowed over links that still exist (of course, by memory references any block can communicate with any other unless it violates memory protection).

Conditional executions are obtained by conditional start operations; a block is started if and only if a condition (for example, that some result is positive) is met.

Memory protection in a multiprogramming environment is obtained by

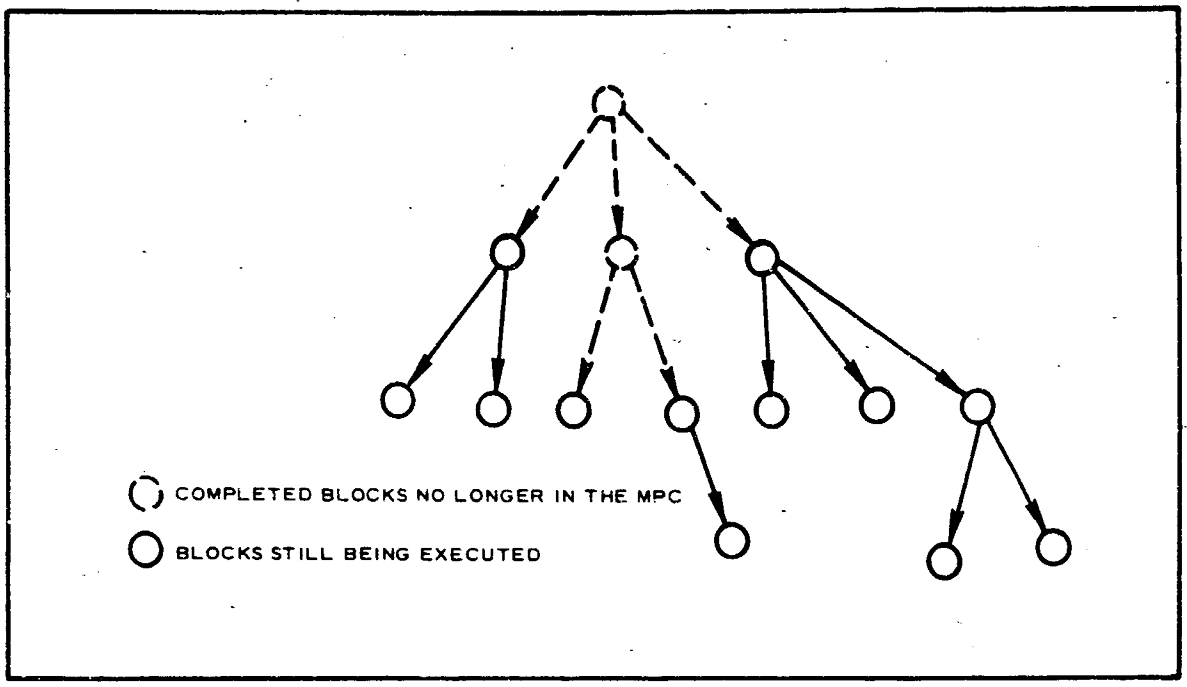
COMPLETED BLOCKS NO LONGER IN THE MPC
BLOCKS STILL BEING EXECUTED

Figure 8 - Example of a Program Structure in the MPC

giving each program a private code stored in the leftmost part of all its addresses. All memory references are specified with the program giving the rightmost part and its code giving the leftmost part. A special start instruction allows execution of supervisory routines for special purposes, such as I/O.

These are the basic characteristics of the machine language of Machine II. Appendix XIV gives a fuller discussion. The general implementation is described below with a fuller discussion in Appendix XV.

The implementation of the MPC (see Figure 7) uses a full-sorting memory. The MPC words are in seven different regions. The I/O region contains words being inputted or outputted by I/O channels. An I/O buffer region stores words waiting for transfer to the I/O region. New blocks enter the MPC through the memory region. Each instruction in the new block creates three MPC words: two operand requests (formed from the operand fields) and an operation word. The operation word is stored in the instruction region. The operand requests go to the result region where results of all blocks are kept. The requests read the operands and then join the corresponding operation words in the instruction region. When all three words are joined, the three are sent to the processor region to be executed by one of the processors. The seventh region is the pointer region, which stores the interblock links. A fuller description of the MPC is given in Appendix XV, Item 7. The memory request sorter (see Figure 7) is a sorting memory that arranges memory requests (read and write) from the I/O devices and processors into order and transmits them to memory. It is described in Appendix XV, Item 6.

The task level computer assigns priorities to tasks in a multitask situation. Priorities are dynamic, changing with machine usage and supervisory control. The task level computer measures machine usage by each task and changes priorities accordingly. Its presence simplifies the supervisor and makes possible a priority scheme wherein each task can be assigned a certain percentage of machine capacity and given execution time at regular intervals. A discussion of the task level computer is given in Appendix XV, Item 5.

Machine II has better communication facilities between subtasks (intrablock communication and interblock communication along links without using memory), but as shown in Figure 8 the program structure in the MPC is limited to one or more trees. Some problems may not fit this condition and memory transfers will be required. Thus, Machine II does not completely solve the problem stated at the beginning of this discussion (Item 5). A more flexible organization specifically designed for nonnumeric processing is described below.

6. PARALLEL NONNUMERIC PROCESSING

As discussed in the beginning of Item 5 above, there is a programming problem in flexible multiprocessors. The machine allows any subtask to communicate with any other subtask so data transfers must be specified in the programs; a fixed structure machine does not have the problem because the machine itself limits possible transfers to a small set. Data transfers between tasks in Machine I are specified using memory addresses; when one program controls several subtasks there is the problem of creating unique addresses for the several subtasks. Machine II allows data transfers within a program block and data transfers between linked blocks in the MPC without the use of memory addresses. If the program can be fitted with a tree-like structure, this technique can be used, otherwise memory addresses must be employed. An ideal machine should allow any subtask to be linked with any other subtask for data transfer without the worry of explicitly labeling the subtasks; this allows one program to control several subtasks concurrently since explicit labeling of operands is not required in the program.

As an example, an ideal machine should accept a set of data on which some structure or topology has been imposed (the structure is dictated by the problem and not by the machine). Figure 9 gives an example data structure. It should be possible to write a program for this machine that will treat one or more parts of this structure simultaneously without knowing the exact labels linked to certain items (the solid nodes) and specifies operands relative to the solid nodes by paths; for example, it may specify the cross-hatched items with the path A, B, F, or C', F (the prime means to travel backward over a link). It may also use "constants" linked to the program (the node linked to the program with a G link). The program should be capable of changing the structure, adding new items and new links, and deleting items and links. It should also be able to do pattern searching.

With this kind of machine, problems can be programmed without the worry of memory assignment and its attendant difficulties when one program controls several tasks simultaneously. Such a machine should be very versatile; nonnumeric as well as numeric problems will be easily treated. To show this, it first should be asked what is meant by nonnumeric as normally applied to problems. The words numeric and nonnumeric are misnomers since many nonnumeric problems contain numbers. The distinguishing

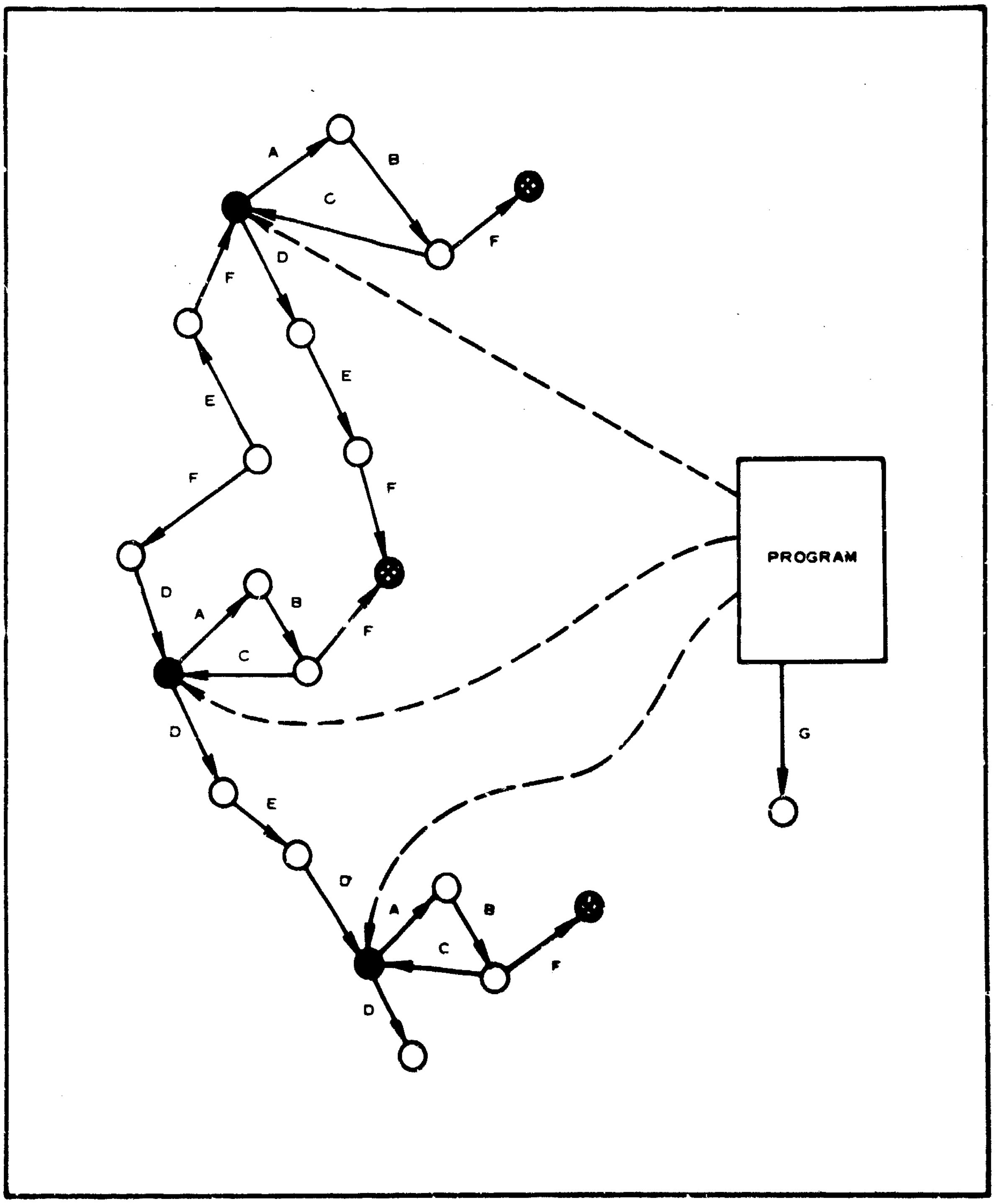

Figure 9 - Example Data Structure with a Program Working on Several Parts of It Concurrently

characteristic seems to be the addressing of operands. In a nonnumeric problem, most operands are addressed by their properties (attributes). The terms "explicit-addressing" and "implicit-addressing" come closer to describing the actual truth. Next, it should be asked what kinds of properties are asked for in implicit addressing. They seem to fall in three classes:

1. Properties dependent on the item per se; for example whether it is larger or smaller than some threshold, the pattern of bits it has in some field, etc. Content-addressing describes this class well.
2. Maximum or minimum properties such as being the largest or smallest item in some set. This might be called limit-addressing.
3. Structural or topological properties. Is the item related to its "neighbors" in a certain way, etc. This might be called structure-addressing.

With a sorting memory, a parallel nonnumeric processor could be built to handle all three classes. Appendix XVI discusses a suitable sorting memory. It stores two words for each link in the data structure, a forward word and a backward word. This permits use of a link in either direction. Three fields in a word indicate an initial node label, a link weight, and a terminal node label. Concurrent searches can be carried out to retrieve incident links on a set of nodes. This permits fast structure-addressing (a fast pattern search algorithm is shown in Appendix XVI). By allowing a node to carry more than one item and by storing node contents in the sorting memory, fast limit addressing is obtained (since the nodal contents are ordered).

Content addressing can be replaced by structure-addressing by the technique of linking an item to its attributes instead of storing the attributes in the item. Appendix XVI discusses this further. This gives the machine a multicomparand content-addressing facility (several different programs may be operating concurrently) so it may perform many content-addressing problems faster than a normal content-addressable memory (CAM) with its one comparand.

Time limitations in the study prevented the completion of a machine design based on the foregoing considerations. Appendix XVI discusses the sorting memory to be used along with a fuller discussion of nonnumeric processing in general and how this machine would compare with other nonnumeric processors (both hardware and software).

The flexible implicit addressing capability of this machine would cure most of the problems discussed in the beginning of Item 5.

7. CONCLUSIONS

It has been shown that a multiprocessor needs a flexible intercommunication structure to permit the processing load to be distributed among the processors. Sorting networks and memories based on sorting networks are a means of providing this structure without undue amounts of hardware.

Three different machine organizations are discussed using sorting networks. They differ in how operands are addressed Machine I requires assignment of memory addresses while Machine II permits some intercommunication without memory assignment. The third organization, oriented toward nonnumeric problems, should allow intercommunication without explicit addressing. A study of these organizations shows that the language in a flexible communication structure machine must be carefully designed to permit programs to use the flexibility effectively.

SECTION IV - APPLICATIONS EFFORT

1. GENERAL

Included in the original goal of the applications area effort of the advanced computer organization study was the selection and analysis of two problems suitable for implementation on a parallel processor. Each of the problems selected was to be subjected to a thorough analysis to determine its inherent parallelism. Restructuring of the problem was to be done to the extent that greater parallelism could be achieved. Solution models were then to be constructed that would allow the exploitation of the greatest degree of parallelism resident in the problem consistent with the capabilities of the parallel processor developed in the study. The first problem selected was numeric, the second was nonnumeric. A discussion of the results obtained in both the numeric and nonnumeric areas follows.

2. NUMERIC PROBLEM AREA

a. Introduction

Initial efforts in the numeric problem area were directed toward analysis of the dynamic programming technique. Preliminary analysis revealed the presence of potential parallelism in dynamic programming and the technique was selected as the basis for the first problem to be studied. Subsequent to the completion of the first problem study (dynamic programming), an investigation was initiated to determine a technique to be used as a basis for the second problem effort. Among the techniques investigated were Jacobi's method of eigenvalue determination, relaxation solution of a system of linear algebraic equations, and numerical solutions of Laplace's equation. Each was seen to possess sufficient parallelism to warrant its use as a basis for the second problem effort. However, it was decided to choose a nonnumeric second problem. A general discussion of results obtained in the numeric problem area is given below. A detailed discussion of the dynamic programming technique study is given in Appendix I; a detailed discussion of the numeric techniques investigated during the second problem selection effort is given in Appendix VIII.

b. Dynamic Programming Technique

(1) Discussion

Dynamic programming is a mathematical technique devised by Richard Bellman for the solution of maximization problems of the following type:

Let x be a resource that is to be divided among some n activities in amounts $x_1, x_2, \ldots, x_n$ such that

$$\sum_{i=1}^{n} x_i = x \tag{1}$$

and $x_i \geqq 0$ for all i's.

Let the return realized from allocating x_i to the ith activity be denoted by $g_i(x_i)$ where

$$\left.\begin{aligned} g_i(x_i) &\geqq 0 \\ g_i(0) &= 0 \,. \end{aligned}\right\} \tag{2}$$

Let the total return realized by allocating x in amounts $x_1, x_2, \ldots, x_n$ to the activity functions $g_1(x_1), g_2(x_2), \ldots, g_n(x_n)$ be denoted by

$$R_n(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} g_i(x_i) \,. \tag{3}$$

The problem, then, is to maximize the return function (3) over the space

$$S_n(x) = \left\{ (x_1, x_2, \ldots, x_n) \middle| \sum_{i=1}^{n} x_i = x,\ x_i \geq 0 \right\}. \tag{4}$$

The dynamic programming technique specifies a procedure for determining an optimal allocation of the resource x in amounts $x_1, x_2, \ldots, x_n$; that is, an allocation of the resource x for which the return function (3) is maximized. The specification of an optimal allocation rests on the construction of the sequences

$$f_1(x),\ f_2(x),\ \ldots,\ f_n(x) \tag{5}$$

and

$$x_1(x),\ x_2(x),\ \ldots,\ x_n(x)\ , \tag{6}$$

where $f_n(x)$ is defined by

$$f_k(x) = \max_{S_k(x)} \left[R_k(x_1, x_2, \ldots, x_k) \right]. \tag{7}$$

It is easily deduced that

$$f_1(x) = g_1(x) \tag{8}$$

and it may be shown that the following recursive relation holds

$$f_k(x) = \max_{0 \leq x_k \leq x} \left[g_k(x_k) + f_{k-1}(x - x_k) \right]. \tag{9}$$

The terms of the sequence (6) are then defined by identifying $x_k(x)$ as the allocation to $g_k(x_k)$ for which $f_k(x)$, defined by (9), is maximized. Equations (8) and (9) provide an inductive method for determining the sequences (5) and (6), and reduce the problem of maximizing one function of n variables to the problem of maximizing n functions of one variable. The value of $f_n(x)$, of course, gives the maximum return possible for the return function (3).

(2) Illustration of the Computational Procedure

As an illustration of the computational procedure, consider the maximization of a function

$$R_6(x_1, x_2, x_3, x_4, x_5, x_6) = \sum_{i=1}^{n} g_i(x_i) \tag{10}$$

under constants

$$\left.\begin{aligned} x_i &\geqq 0 \\ \sum_{i=1}^{6} x_i &= 2.0 \end{aligned}\right\}. \tag{11}$$

The problem then is to maximize (10) over the space

$$S_6(2.0) = \left\{ (x_1, x_2\ \ x_3, x_4, x_5, x_6) \,\middle|\, \sum_{i=1}^{6} x_i = 2.0,\ x_i \geqq 0 \right\}. \tag{12}$$

The maximization would proceed by construction of the sequences (5) and (6). The maximum return realizable by allocating the resource to the first activity only is given by

$$f_1(x) = g_1(x) .$$

The maximum return possible from the first two activities is then determined by computing

$$f_2(x) = \max_{0 \leqq x_2 \leqq x} \left[g_2(x_2) + f_1(x - x_2) \right] .$$

The maximum return possible from the first three activities is determined by computing

$$f_3(x) = \max_{0 \leq x_3 \leq x} \left[g_3(x_3) + f_2(x - x_3) \right] .$$

The inductive method of computing the sequence (5) continues until the maximum return possible from all activities is determined by

$$f_6(x) = \max_{0 \leq x_6 \leq x} \left[g_6(x_6) + f_5(x - x_6) \right] .$$

The sequence (6) is determined by recording the values for which the maximizations are effected. The actual calculation of the sequences (5) and (6) requires that the resource range $[0, x]$ be discretized by some partition,

$$0 = t_1 < t_2 < \ldots < t_n = x , \tag{13}$$

where $t_i = i\Delta$ for some fixed Δ. The partition (13) can then be compactly denoted by $0(\Delta)x$; that is, from zero through x in steps of Δ. In the case of the example problem, the partition might be $0(\Delta)x = 0(0.1)2.0$ for a Δ of 0.1. Given a partition such as (13), each activity function, $g_i(x)$, must be calculated at each point of the partition. Similarly, the construction of the sequence $f_1(x)$, $f_2(x)$, ... , $f_6(x)$ requires that $f_i(x)$ be calculated at each point of the partition since the construction of $f_{k+1}(x)$ requires the values of $f_k(t)$ for $t = 0(\Delta)x$.

(3) Parallel Features

Certain features of the dynamic programming technique are immediately seen to be amenable to parallel computation. Since the return functions involved in the maximization process are mutually independent, they can be evaluated in parallel. Further, the values of a return function for each point in a partition such as (13) may be computed in parallel. By restructuring the dynamic programming technique, additional parallelism may be realized. The heart of the dynamic programming technique is the construction of the sequences (5) and (6), namely

$$f_1(x),\ f_2(x),\ \ldots,\ f_n(x)$$

and

$$x_1(x),\ x_2(x),\ \ldots,\ x_n(x)\ .$$

Now the method for constructing the sequences (5) and (6), as outlined above, is sequential in nature. But it need not be. Instead of recursively calculating the functions $f_1(x)$, $f_2(x)$, . . . , $f_n(x)$, one may specify a concurrent pairwise maximization of the activity functions and thus inject additional parallelism into the dynamic programming technique. Consider the sample problem given above. The problem is to maximize the return function

$$R_6(x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6) = \sum_{i=1}^{6} g_i(x_i)$$

under the constraints

$$x_i \geqq 0$$

$$\sum_{i=1}^{6} x_i = 2.0\ .$$

Parallel maximization of the return function can be achieved by treeing the maximization process into three levels of parallel computation as follows.

Level 1 - for x = 0(0.1)2.0, compute:

$$\begin{cases} u_1(x) = \max\limits_{0 \leqq y \leqq x} \left[g_2(y) + g_1(x-y)\right], \\ y_1(x) = y \text{ at which the maximum occurs}. \end{cases}$$

$$\begin{cases} u_2(x) = \max\limits_{0 \leqq y \leqq x} \left[g_4(y) + g_3(x-y)\right], \\ y_2(x) = y \text{ at which the maximum occurs}. \end{cases}$$

Level 2 - for x = 0(0.1)2.0, compute:

$$\begin{cases} u_3(x) = \max\limits_{0 \leqq y \leqq x} \left[g_6(y) + g_5(x-y)\right], \\ y_3(x) = y \text{ at which the maximum occurs}. \end{cases}$$

$$\begin{cases} u_4(x) = \max\limits_{0 \leqq y \leqq x} \left[u_2(y) + u_1(x-y)\right], \\ y_4(x) = y \text{ at which the maximum occurs}. \end{cases}$$

Level 3 - for x = 0(0.1)2.0, compute:

$$\begin{cases} u_5(x) = \max\limits_{0 \leqq y \leqq x} \left[u_4(y) + u_3(x-y)\right], \\ y_5(x) = y \text{ at which the maximum occurs}. \end{cases}$$

Then $u_5(x)$ gives the maximum possible return for a resource x.

By treeing the dynamic programming maximization process in the fashion described above, the number of computational levels required can be reduced from the n required for sequential execution to approximately $\ln_2(n)$ for parallel execution. It is shown in Appendix I that for a small optimization problem such as (10), parallel methods can reduce the total number of computational levels required from 251 to 24. For larger problems, even greater advantages can be achieved.

c. Other Techniques

(1) Discussion

Prior to the selection of a nonnumeric second problem, several numerical techniques were examined for parallel characteristics. They include Jacobi's method of eigenvalue determination, relaxation methods, and a numerical solution to Laplace's equation. These techniques are reviewed in detail in Appendix VIII; a general discussion follows below.

(2) Jacobi's Method

Jacobi's method is a mathematical technique for finding the eigenvalues and eigenvectors of a real symmetric matrix. The method is based on the following well-known theorem from matrix algebra:

Let $A = (a_{ij})$ be an n-by-n real symmetric matrix. Then there exists an orthogonal matrix U such that

$$U'AU = D\left[\lambda_1, \lambda_2, \ldots, {}_n\right] = D, \qquad (14)$$

where U' denotes the transpose of U, $D = D\left[\lambda_1, \lambda_2, \ldots, \lambda_n\right]$ denotes a diagonal matrix, and λ_i, $i = 1, 2, \ldots, n$ are the eigenvalues of A.

Since in (14) U is orthogonal,

$$AU = UD \qquad (15)$$

and hence the columns of U are the eigenvectors of A.

Jacobi's method specifies the construction of a sequence of orthogonal matrices $T_1, T_2, \ldots, T_k$ such that

$$T_k' T_{k-1}' \ldots T_1 A T_1 T_2 \ldots T_k = C, \qquad (16)$$

where C is an n-by-n matrix whose off-diagonal elements are arbitrarily close to zero and whose diagonal elements are arbitrarily close to the eigenvalues of A. The columns of the matrix

$T_1 T_2 \ldots T_k$ are then arbitrarily close to the eigenvectors of A. It is shown in Appendix VIII that the construction of the sequence $T_1, T_2, \ldots, T_k$ involves operations that are readily adaptable to parallel execution. These operations include the searching of a set for the element of greatest magnitude, and extensive matrix operations. Searching a set for the element of greatest magnitude is an operation well suited to the sorting capabilities of Machines I and II (see Appendices VI and XV).

(3) The Relaxation Technique

Relaxation is a term originally applied by Southwell to a class of iterative methods for solving a system of linear equations. The term has since come to connote a broad class of methods for the approximate reformulation of physical problems in terms of systems of linear equations to be solved. An example of this expanded use of the term relaxation is offered below where a numerical solution to Laplace's equation is discussed. In the strict sense, the relaxation technique provides a method for solving a system of linear algebraic equations, expressed in matrix form as

$$AX = B , \tag{17}$$

where A is an n-by-n coefficient matrix of known constants, $X = (x_1, x_2, x_3, \ldots, x_n)$ is a column vector of unknowns, and $B = (b_1, b_2, \ldots, b_n)$ is a column vector of known constants.

The relaxation technique is an iterative procedure that specifies a sequence $X_1, X_2, \ldots, X_k (X_i = (x_1^i, x_2^i, \ldots, x_n^i))$ of approximations that converges to the solution vector X. Discussions of necessary and sufficient conditions for convergence may be found in Volume Two, Appendixes I, XIV, and XV, and in Reference 2. The technique assumes an initial guess X_1 and computes successively vectors $R_i = \left(r_1^i, r_2^i, \ldots, r_n^i\right)$ of "residuals" defined as

$$R_i = B - AX_i \tag{18}$$

for $i = 1, 2, \ldots, k$.

The residual vector R_i provides a measure of the closeness of the approximation X_i to X. Based on a residual vector R_i, the relaxation technique specifies a new approximation X_{i+1}. The process continues until the elements of the residual vector are sufficiently close to zero to satisfy a pre-established convergence criterion such as $R_i \cdot R_i < \varepsilon$ or

$$\max_k \left(\left| r_k^i \right| \right) < \varepsilon \ .$$

The relaxation method involves the repeated execution of the operations of matrix multiplication and addition, multiplication of a vector by a scalar, and searching a set for the element of largest magnitude. Each of these operations is well suited to parallel execution, and the operation of finding in a set the element of largest magnitude may be accomplished rapidly on a parallel processor having sorting capability.

(4) Numerical Solution to Laplace's Equation

Discussed herein is the numerical solution of Laplace's equation over a rectangular region, R. It is assumed that R is partitioned by an equally spaced rectangular mesh and that Dirichlet boundary conditions are specified. Given a function u(x, y) for which Laplace's equation obtains over R, one writes

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \ . \tag{19}$$

Letting the interval for the mesh over R be denoted by Δ, the partial derivatives for u(x, y) may be approximated by

$$\left.\begin{aligned}
\frac{\partial u}{\partial x} &= \frac{u(x+\Delta,\ y) - u(x,\ y)}{\Delta}, \\
\frac{\partial u}{\partial y} &= \frac{u(x,\ y+\Delta) - u(x,\ y)}{\Delta}, \\
\frac{\partial^2 u}{\partial x^2} &= \frac{u(x+\Delta,\ y) - 2u(x,\ y) + u(x-\Delta,\ y)}{\Delta^2}, \\
\frac{\partial^2 u}{\partial y^2} &= \frac{u(x,\ y+\Delta) - 2u(x,\ y) + u(x,\ y-\Delta)}{\Delta^2},
\end{aligned}\right\} \qquad (20)$$

ana the difference equation counterpart of (19) may be written as

$$u(x,\ y) = \frac{1}{4}\left[u(x+\Delta,\ y) + u(x-\Delta,\ y) + u(x,\ y+\Delta) + u(x,\ y-\Delta)\right]. \qquad (21)$$

Equation (21) approximates u(x, y) at each interior mesh point of R by the average of "north, south, east, west neighbors." Other such difference equation approximations to u(x, y) at interior points of R are available.

Iterative solutions to Laplace's equation based on approximations such as (21) converge (see Appendix XIV) and are often called "relaxation solutions." A sequential iterative solution would proceed by ordering the interior mesh points of a region R and cyclicly applying the approximation over the ordering until some specified convergence criterion is met. In a sequential pass over the ordered interior mesh points of R, two possibilities for updating the values for u(x, y) at each interior mesh point are available: (1) as each new approximation to u(x, y) is generated at a point, it is made available for subsequent calculations in the pass; (2) each pointwise approximation to u(x, y) made in a given pass uses only point values available at the end of the preceding pass. The former (latter) method of updating often is called the method of successive (simultaneous) displacements.

The numerical solution to Laplace's equation over a rectangular region partitioned by an equally spaced rectangular mesh is specified easily in terms of an approximation such as (21) and the methods of simultaneous or successive displacements. Iterative numerical solutions to Laplace's equation over a mesh begin by assuming some initial values for u(x, y) at interior points. Clearly, the greater the accuracy of the initial approximations, the more rapid should be convergence. Appendix VIII describes a method for rapidly computing initial approximations to u(x, y) over a mesh based on known boundary values. The methods of approximating u(x, y) is well suited for parallel execution and is called parallel fill-in (PFI).

The numerical solution to Laplace's equation over a mesh is well suited to parallel computation. For a parallel processor of sufficient size, a processing unit could be assigned to each of the interior mesh points. Each unit would then compute and store, in an iterative fashion, approximations to u(x, y) at its assigned point. In the event that the number of interior mesh points exceeded the number of processing units, each unit could be assigned a block of interior mesh points and the iteration would proceed "parallel by block and sequential by point within a block." A test for convergence based on maximum pointwise change in approximation values between successive iterations could be accomplished readily on Machines I and II due to their rapid sort capability.

3. NONNUMERIC PROBLEM AREA

a. Introduction

The second problem selected for the advanced computer organization study was the parallel compilation of higher language programming statements. The language selected for use was MAD.[2] The selection of parallel compilation was quite natural in that investigations of parallel processor configurations and parallel execution of coded routines lead to consideration of compiling source programs, written in a higher language, in parallel.

Prior to the second problem study, a short effort was directed toward the construction of a language designed specifically for a parallel processor. The goal was to construct a language that would allow, and indeed promote, ease of conceiving and expressing the structure of parallel solution models. Results of the effort are presented in detail in Appendix IX. A general discussion of parallel compilation follows. A detailed discussion will be found in Appendixes X and XI.

b. Parallel Compilation

In the process of compilation, a sequence of statements written in a higher language, such as MAD, is translated into a sequence of machine language statements. The compilation process usually will decompose higher language statements into a matrix form of triples and then from the matrix establish a set of machine language statements. Included in the compilation process is the handling of such considerations as dimension, mode, and storage allocation.

The compilation algorithm developed during the second problem effort deals only with the decomposition of higher language statements into triples. The statements themselves are restricted to replacements involving nonsubscripted variables. It is assumed that the statements are written in MAD and that the precedence hierarchy is that of Arden, Galler, and Graham.[3] The precedence hierarchy is limited to the set of operators given in Table III. It is further assumed that the replacement statements are stored, symbol by symbol, in an ordered list. For example, the MAD statement

$$F = A + B * .ABS. (C + D) \qquad (22)$$

is assumed to be stored in a list as follows.

β	S_β	
0	$\vdash$	
1	F	
2	=	
3	A	
4	+	
5	B	
6	*	(23)
7	. ABS.	
8	(	
9	C	
10	+	
11	D	
12	)	
13	$\dashv$	

TABLE III - PRECEDENCE HIERARCHY

Operator	Description	Precedence
. ABS.	Absolute value	Highest
. P.	Exponentiation	↓
$-_u$	Unary minus	
*, /	Multiplication, division	
+, -	Plus, minus	
=	Equals (substitution)	
$\vdash$, $\dashv$, (,)	Begin statement, end statement, open parenthesis, close parenthesis	Lowest

As shown in Appendix X, the triples corresponding to (23) are just

C	+	D
0	.ABS.	R1
B	*	R2
A	+	R3
F	=	R4

(24)

where Ri denotes the resultant from the ith triple (row). Then (24) would be read, row by row, as:

$$R1 = C + D$$

$$R2 = \text{.ABS. } (R1)$$

$$R3 = B + R2$$

$$R4 = A + R3$$

and finally

$$F = R4 = A + B^* \text{ .ABS. } (C + D)$$

which is just (22).

In parallel compilation, the aim is to examine simultaneously in successive passes many statements such as (22) stored in the fashion of (23), and to form on each pass all possible triples and statement simplifications for the entire set of statements. An algorithm for effecting parallel compilation was developed during the second problem effort and is summarized in Figure 10.

The tests (operations) indicated in Figure 10 are applied on each pass to a list such as (23). Sequences of items taken 3, 4, or 5 at a time (blanks are ignored) that meet certain conditions are sought. If the indicated conditions obtain, triples are formed and/or statements simplified as indicated. As the structure of the flow chart in Figure 10 indicates, the four operations may be executed concurrently and the algorithm will be capable of decomposing, in parallel, all the substitution statements of a source language (MAD)

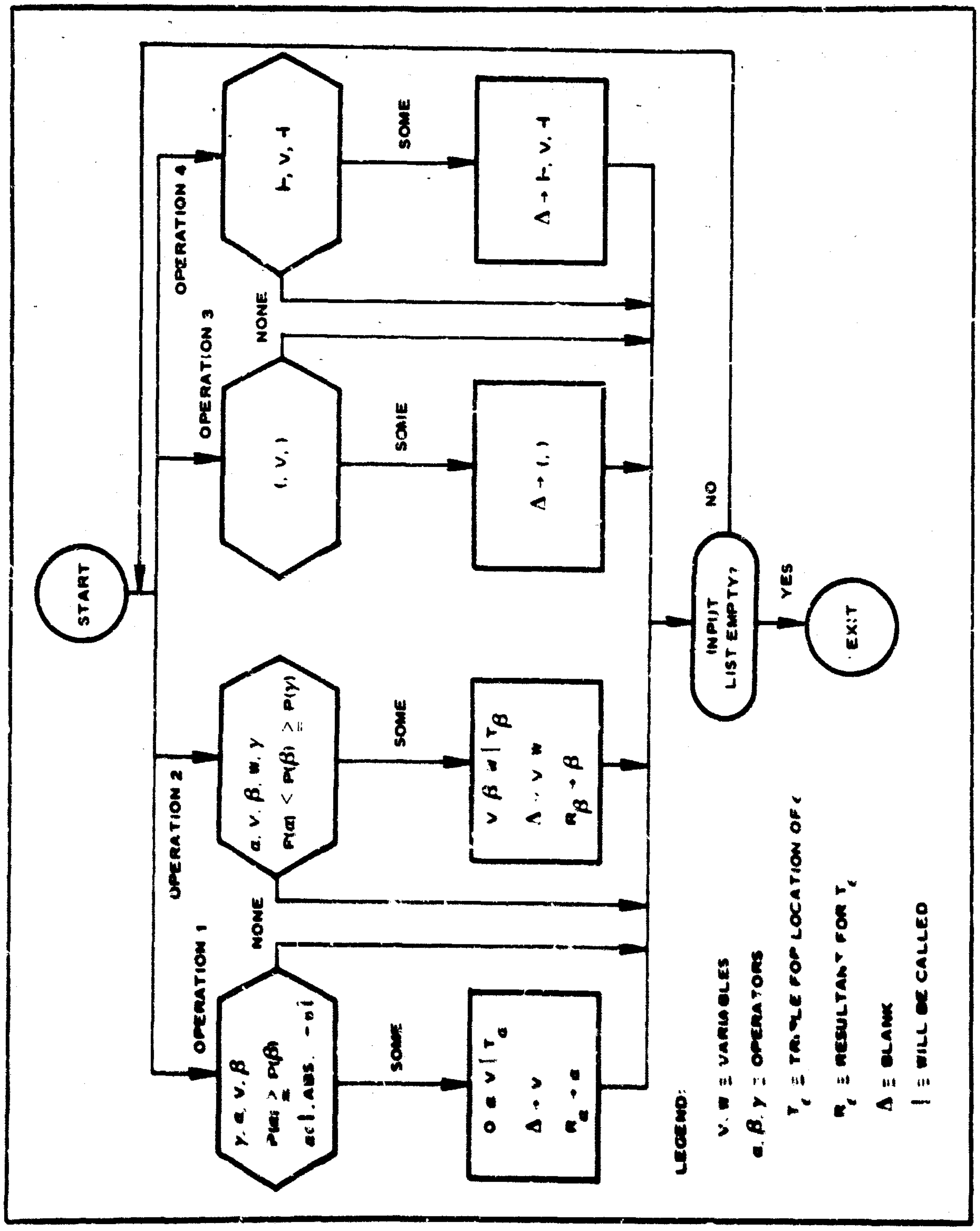

Figure 10 - Parallel Compilation Algorithm

program into a string of triples ready for final assignment (machine language). Several passes through the loop may be required; the number will depend on the size and complexity of the program to be compiled. The operations indicated in Figure 10 proceed as follows:

1. Operation 1 looks for quadruples ABCD where

   A is an operator

   B is either a "$-_u$" or an ".ABS."

   C is a variable

   D is an operator such that $P(D) \leq P(B)$ where $P(x)$ denotes the precedence of x as given in Table I

   It is assumed that B is the αth item on the input list. C is removed and B is replaced by the variable R_α. A triple is formed of 0, B, and C and its resultant is stored in R_α.

2. Operation 2 looks for all quintuples ABCDE such that

   A, C, and E are operators

   B and D are variables

   $P(A) < P(C) \geq P(E)$

   It is assumed that C is the βth item on the input list. B and D are removed. C is replaced by R_β, and a triple is formed of B, C, D with resultant R_β.

3. Operation 3 removes parentheses surrounding single variables

4. Operation 4 removes all sequences $\vdash A \dashv$ where A is a variable

A step-by-step example of the application of the compilation algorithm may be found in Appendix X.

It will be noted that the triples generated on each pass correspond to basic arithmetic operations that can be performed at the time of the pass. Hence, the compilation algorithm generates triples suitable for parallel execution and provides a first approach to the recognition of low level parallelism within a source program.

The parallel compilation algorithm was programmed for Machine II (see Appendix XIII for details). The programming proved to be difficult and the results suggested the desirability of modifying the algorithm. Modification seemed desirable because implementation of the algorithm in the form of Figure 10 required the initiation of an excessive number of parallel processor "tasks" (see Appendixes XV and XVI), led to extremely cumbersome control programs, and failed to yield anticipated levels of speed advantage for parallel over sequential compilation.

As a means of obviating the problems of implementation, the following modifications were considered: (1) preliminary translation of input statements to reverse Polish rotation, and (2) innovations in the utilization of the parallel processor programming language. Both modifications were investigated with fruitful results. An unexpected result of the investigation was the development of a completely new form of the compilation algorithm. The results of the modifications are detailed in Appendix XI.

Although insufficient time was available to investigate the modifications in detail, preliminary investigations indicate that the first two modifications are easily programmable but do not result in significant speed advantages. The restructured algorithm appears to provide maximal utilization of parallelism resident in the compilation process and should offer significantly increased compilation speeds.

SECTION V - PROGRAMMING

1. MACHINE I

Machine I is composed of many identical processors, each having the capability of simultaneously accessing by content any location in the self-sorting memory. There are 512 processors in Machine I, each having a program counter, instruction register, accumulator, quotient register, and six index registers. These registers are analogous to those in conventional computers.

The program counter generates instruction addresses. The instruction register is composed of an upper and lower half, each is capable of containing a 36-bit instruction and contains the current instruction to be executed. The accumulator is similarly composed and can be considered as two 36-bit registers, upper and lower, or as one 72-bit register. The quotient register is organized similarly to the accumulator. The index registers in each processor have a desirable capability - any three may be added together with the contents of the address field of the instruction to generate an address, an operand, or a shift count.

The inherent sorting capability of Machine I significantly reduced the execution time of a portion of the dynamic programming problem. The determination of the largest return out of a number of possible returns was completely resolved via the sort memory. The various possible combinations of returns were calculated and stored with the same address. The characteristics of the memory allowed the larger of two or more numbers always to return at the "top" of the sorted table after one machine cycle. In the dynamic programming problem, the routines were such that only two numbers were sorted per machine cycle, but the technique is not limited to only two. For large-scale sorting, any number of elements could be sorted in the same length of time - one machine cycle.

SECTION V - PROGRAMMING

The instruction set for Machine I is an extension of one that might be found on a contemporary machine. One class of instructions considers the contents of the address field as the operand which is modifiable by index register combinations. Class 2 instructions treat the contents of the address field as the operand address, also modifiable by the index register. Class 3 instructions treat the contents of the address field as a shift count that is modifiable by the index registers. Class 4 instructions allow inter-register transfers within a processor as detailed in Appendix IV.

Some instructions are peculiar to Machine 1 and were useful in synchronizing operations when multiple processors were operating on a common problem. The nonpresence jump instruction when executed causes a jump to some location when the address of the word requested from memory by the previous fetch type instruction is not the same as the operand address in the instruction. A typical application would be when two processors are operating in conjunction on a problem - one generating a piece of data that the other is looking for - clearly an indication is necessary that the correct data have been acquired. It should be mentioned that any request for memory data always returns a piece of data, either the correct word or the word whose value is next higher than the request word. This is a characteristic of the sorting memory.

There also is a set of instructions that allows searching for single words, or searching for the smallest word between limits. These instructions are desirable for searching internal areas of a list without requiring examination of the entire list. If the upper and lower bounds of the list are known, limit words lying within these bounds can be used to isolate portions of the list directly. With the erase options available with the search instructions, either single or multiple entries within a list can be isolated and erased using only one instruction as explained in Appendix IV.

The instruction execution time for Machine I for a two-instruction word is 30 usec.

Machine I is best suited for advantageous use of parallelism that exists between independent blocks of a program or between independent programs. Multiple branches within a program to start independent blocks of the program can be programmed easily. The ends of the branches are joined via the nonpresence jump instructions. Independent programs such as a compiler and numerical and nonnumerical problems can be executed in parallel. (Of course, this assumes some form of supervisory control incorporating processor loading, problem execution time, and memory loading information to eliminate or reduce conflicts between different problem programs.)

A major point of interest in Machine I is its content-addressing capability. In contemporary machines, the memory is addressed by the absolute address of the word in question. In Machine I, any word in memory may have any name (address) assigned to it. When the usefulness of this word has expired, it is erased and the word is then available to any processor for naming. An added advantage to the content-addressing capability is then the more efficient use of the available memory, since blocks of memory are not assigned permanently to a particular program but float around, so to speak, wherever needed, dependent, of course, upon the programmer's maintenance of a clean memory within his program; that is, the erasure of data no longer needed.

2. MACHINE II

Machine II is a parallel processor with the ability to access simultaneously by content many locations in a self-sorting memory. There are 256 processors in Machine II, each having access to program instructions in the multiprocessor control unit (MPC). There are no registers, such as instruction counter, accumulator or index, as such, accessible to the programmer Instructions that generate a result are, when executed, replaced by the result. These instructions, however, are not executed until the operands are available. Hence, the sequential nature of an instruction string is preserved where necessary by the sequential availability of the operands. On the other hand, independent strings of instructions are executed in parallel as soon as the requisite data are presented to the several strings. This type of parallelism is inherently available in Machine II in addition to the capability of parallel execution of independent blocks of a program or of independent programs (see Appendix XV).

The instruction set for Machine II is similar to that found in a two-address-per-instruction machine. The arithmetic instructions contain in their address fields the addresses of the two operands. The logical instructions are similar and quite extensive and comprise all possible functions of two Boolean variables in terms of the "and," "or," and "not" operators. Shift instructions enable left and right shifting of operands that may be located within the current program block or in the previous program block. Data to be carried along through a program are passed from block to block by means of the shift instructions. The "bring" instruction enables a program block to retrieve a piece of data generated by a block that is started by the block in which the bring instruction resides.

"Read memory" and "read memory indirect" instructions allow access to the main sorting memory. "Threshold search" allows retrieval of the word in memory whose contents at the given address a[illegible] just above a threshold.

"Start" instructions enable a block to order the execution of a subsequent block, and to establish the priority of the started block. Conditional starts have the same result as the start except that it is based on the condition or state of some word in the curre[illegible]t block.

Most instructions have an "erase" option attached whereby either or both operands may be erased after instruction execution. This is desirable to maintain a clean memory. There is an explicit erase instruction that permits erasure of all words in the previous block that lie between two limiting words. An "erase previous block relative" allows erasure between limits relative to the start instruction that started the current block. An "m wait" [illegible]struction permits starting of a block on the condition that all memory operations in the block have been completed.

Instruction execution time varies from one to seven MPC cycles or from 13.8 to 96.6 μsec. The particular processor executing an instruction, however, is active for no more than one cycle and the result, if a result is generated, is available the subsequent cycle (see Appendix XV).

The types of parallelism that can be executed most easily on Machine II are instruction level parallelism and independent program parallelism. The machine removes from the programmer the recognition of instruction level parallelism through its ability to execute instructions whenever the requisite operands are available. The recognition of independence in program blocks is still the programmer's responsibility, and is his problem to program so that the machine can execute the problem in minimum time. Independent programs can, of course, be executed simultaneously, limited only by the size of the memory and MPC (see Appendix XIV).

3. MACHINE I AND THE DYNAMIC PROGRAMMING PROBLEM

The dynamic programming problem was chosen because of wide interest in this area and techniques developed for parallel processors might be useful. The problem is characterized by a number of independent functions that define a return for a given resource assignment. Returns are calculated for all values of the available resource for all activity functions. In essence, all combinations of activity function returns are examined to determine which combinations of resource assignments will give the maximum return for each of the possible resource assignments.

In the problem chosen for Machine I, there were six independent activity functions. Some were relatively simple and exhibited a very high degree of sequential independence. Activity functions $g_1(x)$ and $g_2(x)$ were of this nature. At first glance, and independent from the problem as a whole, it would be natural to assign processors to the calculation of the various returns via a treeing program to minimize the computation time. However, this was not done because the simplicity of the functions involved made the the basic computation routine quite fast in relation the more complex activity functions. It so happened that there were pairs of activity functions whose routine execution times were similar. The outputs of these activity function pairs were used as the inputs to the maximization routine that would determine for a given resource the best allocation of that resource. In this manner, as soon as a pair of returns was generated, the maximization routine immediately calculated the best allocation. These maximized returns were in turn maximized with other maximized returns until a table was generated giving the final allocation of any resource for a maximum return (see Appendix III).

The more complex routines, $g_3(x)$ and $g_4(x)$, exhibited parallelism that could realistically be extracted to speed up problem solution. Portions of these routines were broken away and executed in parallel. Additionally, these routines were treed; that is, the number of processors assigned to the computation of returns doubled for each level of the tree.

The maximization routines took advantage of the inherent capability of the sorting memory to determine the maximum return from two functions for a given resource. The returns for various combinations of resource allocation were stored in memory with a common name, with the larger return automatically sorted to the top where it was picked off. The recommended resource allocation for the given resource was then stored in a table with similar recommendations for all other values of allowable resource. This table was the goal of the problem.

With a given input resource, the recommended assignment then could be found that would generate the largest return.

The problem for Machine I was numerical in nature, required the generation of tables, required table sorting, exhibited a high degree of parallelism, and resulted in the development of treeing and timing techniques to arrive at a reasonably optimum solution time.

Problem solution time for the dynamic programming problem on Machine I was 16 msec as opposed to 150 to 220 msec for the sequential machine

Solution time on Machine I was 9 to 14 times faster than the sequential machine; this can be attributed to the availability of the many processors capable of independent and simultaneous action on the contents of any word in the multiaccess sorting memory.

Processor loading reached a peak of 60 processing units out of a possible 512 proposed available processors. The average number of processors for the problem execution time was only 22 (see Figure 11).

If the computer had been fully utilized at the peak period on a larger problem, then the speed advantage would have ranged from 512/60(9-14) = (76 to 119):1. In addition, many processors would have been available at nonpeak times for other uses such as compiling. If the average loading of the processors is considered, then the speed advantage would be 512/22(9-14) = (207 to 322):1.

4. MACHINE II AND THE COMPILATION PROBLEM

The compilation problem for Machine II was chosen because it was

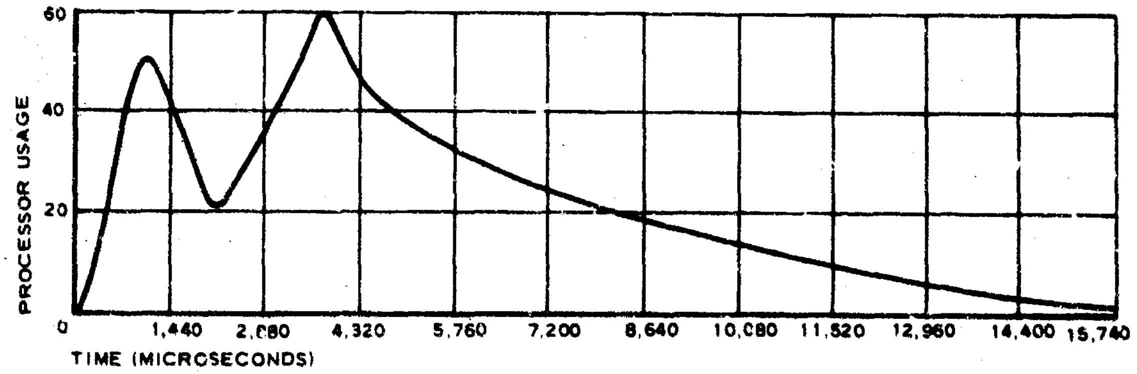

Figure 11 - Processor Usage

believed that there would be a significant advantage to be able to compile while simultaneously executing other programs. It was realized that the whole compiler could not be implemented in this study and hence a portion of the compiler was chosen to demonstrate feasibility. The portion chosen was the scanning of the substitution statement and generation of the corresponding object programs for a limited number of typical operations.

The problem basically consisted of scanning an input statement, distinguishing between variables and nonvariables, determining precedence relations of the operators, and forming a sequence of machine instructions that when executed would generate the desired result.

The scanner implemented was of the Polish type. Each element of the substitution statement was examined to determine if it was a variable or nonvariable.

Variables are immediately transferred to the output string. Nonvariables or operators, if their precedence is equal to or less than the precedence of the operator currently on top of the stack, are transferred to the output list. As soon as an operator is transferred to the output list, there is then a triple composed of two operands and one operator that are used to form a sequence of machine instructions. The triple removed from the output list is replaced by a variable in the form of the triple's resultant address. Depending upon the operator, one of a number of generator programs is started that generates the actual machine language instructions. The generator programs operate simultaneously with the scanner once the triple has been removed from the output list.

The operations considered in the substitution statement were floating point arithmetic, exponentiation, unary minus, absolute value, and equality (see Appendix XIII).

The execution of the parallel compilation algorithm for Machine II required 53.6 msec. Considering the number of statements to be compiled as N, then the rates of the compilation time of Machine II to that of the IBM 7090 is N:27. The parallel compilation time is independent of the number of statements and remains relatively constant at 54 msec. The time required for sequential compilation increases linearly with the number of statements. When the number of statements exceeds 27, the parallel processor time for compilation is less than that for the IBM 7090.

If 256 processors are available, then Machine II can average 256/1.17 = 219 statements every 54 msec. The speed advantage would then be 219/27 = 8:1 over the IBM 7090. If time had permitted one of the other parallel compilation algorithms to be programmed, additional speed advantage would have been realized.

SECTION VI - COMPARISON OF MACHINES I AND II

Both Machine I and Machine II are composed of sorting networks, merging-separating networks, and processing units. Both have a large merging-separating memory, but Machine II also has a smaller, full-sorting memory (MPC). This additional hardware for Machine II is offset by the fact that the processing units in Machine I are much more complex than those in Machine II.

A basic processor of Machine I considered as an entity is not unlike the processors found in contemporary sequential machines. It has the normal arithmetic, quotient, and index registers and access capability to any word in memory. Hence, with some small degree of effort, an existing program could, with modifications consistent with changing from one machine to another, be run on Machine I with only one processor assigned. Considering speed of execution, the average problem so assigned would be slower on Machine I because of the increased instruction execution time.

However, for the execution of an iterative program, the sequential execution time increases linearly with linear data output while the time required for parallel execution increases linearly with exponential increase of data output. The exponential increase is due to the binary treeing program used to assign processors to a problem. The number of processors active on a problem via the binary tree increases by a power of two for each level of the tree. The tree could conceivably be ternary, octal, or decimal where from each node or active processor, 3, 8, or 10 more processors are started. Some techniques to minimize the time required to start these processors and to prevent program monopoly of processors to the detriment of other programs are necessary for these options (see Appendixes III and XIV).

Machine II programs require data passing, index simulation, multiple testing for branches, subsequent block testing, and varying degrees of instruction layout within a block. Simple existing programs easily incorporated in a Machine II block might be executed as fast or faster than on a sequential machine. The amount of parallelism at the instruction level would be the determining factor. In this instance, the 96.6 usec required for data passing has been eliminated and the instructions most likely to be encountered have execution times of 41.4 to 55.2 usec (see Appendixes XIII and XV).

Treeing of processors is permissible in Machine II and the same advantages accrue as with Machine I. Requisite data for the consistent operation on and generation of problem data must be passed to each activated processor. The treeing process would be most useful for execution of programs operating on independent subsets of the problem. Experience has shown that control problems develop when treeing is used to start processors operating on data lists in which subsets of the data are not independent of each other.

Generally, it is expected that existing programs would have to be significantly revised to take advantage of the parallelism found in Machine II.

Machine I programs are stored in the main memory and regardless of how many processors are operating on a program, there is only one copy of the program. The same techniques used in writing an iterative loop for a sequential machine are used similarly in Machine I for writing programs that allow multiple processing. Pertinent data naming lists, base locations, and temporary storage addresses are passed to each activated processor. These data are stored in the same relative registers of all processors active on the problem and according to instruction are treated the same by all processors.

Index passing in the treeing operation requires temporary storage of the incremented indices and a processor start with the first executed instructions requesting the index values stored in the temporary locations.

Machine II programs are also stored in main memory; however, copies of this original are made in the MPC whenever it is to be executed. Multiple executions require multiple copies. An extensive treeing of programs should be accompanied by program disposal of unneeded data to prevent overloading of the MPC and subsequent rejection of incoming program blocks.

The two memories also require distinct instructions for accessing data stored in them. This distinction is a consideration to be kept in mind when programming; data generated by a program usually are found in the MPC unless stored explicitly in the main memory, original data normally are found in the main memory unless explicitly transferred to the MPC.

The processors in Machine I are controlled more easily by the programmer than in Machine II. Each processor called for in Machine I is made active by a specific instruction. Multiple processors usually are called for by making each activated processor execute a portion of the program, which in turn calls for additional processors. The limit is reached by incrementing a counter for each processor started. The count is passed to each processor started and compared with some limit beyond which no more processors are started.

Processor usage in Machine II is not directly controllable by the programmer in the sense of activating a processor explicitly by instruction. The programmer can segment the program into blocks, and squeeze instruction level and block level parallelism out of his problem, but the processors only become active when a program block is stored in the MPC. Only those instructions within a block whose operands are available are executed. The sequential nature of a string of instructions is preserved by the sequence of operand availability. The parallel nature of independent strings is recognized by the machine through the availability of the requisite operands.

The type of program parallelism most economically implemented on Machine I consists of the parallelism existing between independent blocks of a program. Very short strings of independent instructions can be executed in parallel but usually the cost of setting up indices to maintain control is prohibitive and the short strings are performed in sequence buried in larger independent strings.

In contrast, Machine II exhibits parallel execution capability on the same levels as Machine I and in addition recognizes instruction level parallelism independent of the programmer (see Appendix XV). Any independent strings of instructions are executed as soon as the required operands are present.

If such a string, for example, has as operands two constants that were passed to the block containing the string, and possibly a similar string, then the two strings, if of similar length, would generate their resultant at the same time. However, speed is sacrificed to obtain this additional capability provided for Machine II by the MPC. Machine I is capable of obtaining and executing two instructions every 30 µsec. Machine II has an instruction execution time ranging between 13.8 and 96.6 µsec, depending on the instruction.

For both Machine I and II, problem analysis and flow charting should reveal areas of potential parallelism. Flow charting should reveal areas of independence within a program that can easily be formulated for Machine I or Machine II. Completely independent blocks can be executed in any sequence. Where there is sequential dependence at some point in the program, then to prevent waiting for a result it is best to have executed the relatively independent block before the result is actually needed. This entails starting a processor at some point prior in time by an amount of time equal to the time required to generate the desired result. In Machine I, the result would be stored in memory with some name attached. This result would be available directly to any processor possessing the name. In Machine II, the result could be left either in the MPC or the memory. Leaving the result in the MPC is undesirable because in this case the only block that can easily reach the result is the subsequent block started.

Storing the result in main memory makes it available to any program block possessing the name. The name in this case would have to be passed along or generated at the point where the independent parallel paths joined.

Flow charting generally will not display instruction level parallelism that Machine II recognizes readily. Only the actual machine language program will show it.

The maintenance of a clean memory is a necessity in both Machines I and II. The demands are not nearly as stringent in Machine I as in Machine II. Data whose [illegible] has been outlived can either be erased promptly or accumulated [illegible] erased collectively. There is less chance of overloading the main memory in Machine I than there is in overloading the MPC in Machine II. Main memory in Machine I is essentially working memory while the MPC is essentially working memory in Machine II. All information generated in Machine II resides at least temporarily in the MPC and may reside in the main memory if explicitly stored there. So, it is apparent that the same demands for a clean memory in Machine I also exist for Machine II, but Machine II has the added requirements of absolute cleanliness in MPC at the risk of trash accumulation and possible program lockouts from the MPC.

Sorting operations in Machines I and II are accomplished almost independently of programming efforts. If a list of items is given or is being generated, a sorted list is available one machine cycle after the last item is placed on the list. The programming consists essentially in placing items on the list and naming each item the same. Since the names are all the same, sorting proceeds through a comparison of the magnitudes of the data in the data fields of the items. This process was used advantageously in the maximization routine of the dynamic programming programs for Machine I. Any item added to the list occupies a position ordered relative to all other items on the list after one machine cycle.

SECTION VII - RECOMMENDATIONS FOR FOLLOW-ON EFFORT

1. GENERAL

As a result of the work performed on this study, many areas in need of further investigation have been identified. It is the purpose of this section to describe some of these areas.

It is recommended that future work on this program be carried out with the machine organization effort independent of the applications and programming effort. There are many facets of the machine organization work in need of further study, but it is believed that the greatest knowledge can be obtained if the programming and applications efforts are not forced to be continually modified as a result of machine changes. Furthermore, Machines I and II contain many capabilities that have not been utilized and hence it is believed that the application programming effort should focus its attention on how to exploit this existing capability.

2. MACHINE ORGANIZATION

a. Parallel Nonnumeric Processor

It is recommended that the development of the parallel nonnumeric processor started in the study be continued.

b. Parallel Input/Output

Without some means for the parallel inputting and outputting of data, the parallel processor will lose much of its speed advantage on many problems. Even conventional computers are I/O bound on many problems, especially those which process large formatted files. It should be noted that the parallel processor organization, as it exists at this time, is already capable of accepting or transmitting data on many lines simultaneously. Hence, what is needed is a device that is capable of receiving or providing information on these many lines at the same time. An approach should be developed or postulated for further use in the study.

c. Interrupts

The interrupt mechanism by which extra program conditions are sensed and acted on should be investigated and extended in the light of the multiprocessing facility of the parallel processor.

Input/output, processor and priority interrupts may occur simultaneously and will require servicing.

Hardware, software, and logic techniques will be needed to devise a method of solution.

Means of enabling any set of interrupts for each of the programs being executed in parallel should be developed; i.e., different programs may specify different sets of interrupts to be recognized.

d. Priority

In any computing installation, some form of priority is operative. With a multiprocessor facility where the priorities of the problems in the machine cover a wide range, a method of assigning processors to problems is demanded.

The method of implementation may be absorbed by both the hardware and software. Special instructions that allow specification of a problem's priority may be implemented. Some ideas for a dynamic priority scheme were developed for Machine II, but they are in need of elaboration and must also be added to the other machine organizations.

e. Multiaccess Systems

There is considerable current interest in multiaccess systems, such as JOSS, in which many users can have simultaneous access to some central computing facility. Parallel processors seem ideally suited to multiaccess computing. Implemented in a multiaccess system, a parallel processor could handle, simultaneously, on-line computing requests from many users. Unused processing units could be used efficiently to do processing not requiring on-line execution.

Hence, not only will a parallel processor allow multiple user access but will permit highly efficient hardware utilization.

f. Other Applications of Machine Concepts

Portions of the computer organization, such as the sorting network, have potential applications other than in computers. As these portions of the design are developed, their other uses can also be investigated.

g. Feasibility Model

In order to prove the concepts developed and to help the study of items above, it is advisable to build a model of a sorting network. It can be used as a multiaccess memory and by connecting some small-scale computers, I/O gear, etc., to it, a parallel processor can be built. The cost of such a model can be extrapolated to arrive at good cost figures for larger networks.

3. APPLICATIONS - PROGRAMMING

a. Macro Instructions

Concurrent with efforts directed toward the design and efficient utilization of parallel processors has been the realization that processing capabilities resident in parallel processors give rise to new ways of thinking about and solving problems. Attempts to write parallel solution models and express the operations involved in a compact notation have already led to the development of a preliminary system of macro instructions for a parallel processor (see Appendix IX). The development of the preliminary list of macro instructions was due to an effort to express compactly the operations characterizing problems and structuring possible methods of solution. Investigations aimed at the further development of macro instructions should suggest new conceptual modes in which problems and possible solutions may be analyzed. and provide insights into the nature and significance of parallelism within a problem and methods for exploiting it by new computational procedures.

b. Flow Diagramming Techniques

Techniques to indicate parallelism in the problem on a flow charting level are needed. Indications of time required for completion of sections of a program would be useful in allocating processors to enable processor usage distribution.

c. Program Comparison Problems

In many areas of investigation, appropriate problems should be chosen as a means of demonstrating and comparing the relative efficiency of the parallel processor with a sequential machine.

Large formatted file processing where the input/output load is high could be used to compare system efficiencies while exercising input/output and interrupt areas of investigation.

Sample problems could demonstrate the efficiency of the parallel compiler and execution of the object program. Subroutines using the content-addressing and sorting capabilities of the parallel processor could be compared using the same problem.

This effort will aid in determining how to make use of all the capabilities resident in the machine organization.

d. Compiler

The investigation of the MAD compiler should be continued with a detailed examination of some of the other parallel compilation algorithms in Appendix XI. In particular, the statements currently being considered should be extended to include Boolean expressions.

This investigation naturally leads to a consideration of the tradeoff between a fact compiler with relatively slow object-program execution and a relatively slow compiler with fast object-program execution. In the former, maximum parallelism resides in the compiler and in the latter in the object program.

It is believed that the compiler output may influence hardware organization and that parallel execution of the object program may be easier with a revised machine organization.

A closely related problem is the one of allowing the programmer to specify by statement the parallelism in the problem.[4]

e. Library Subroutines

A study of typical subroutines generally available to computers should be conducted. The objective would be to determine how the multiprocessing, associative, content-addressing, and sorting capabilities of the parallel processor can be used to execute these types of subroutines more efficiently.

LIST OF REFERENCES

1. Batcher, K. E.: A New Internal Sorting Method. Akron, Ohio, Goodyear Aerospace Corporation, GER-11759.

2. University of Michigan Computing Center: Michigan Algorithm Decoder. Ann Arbor, Mich., June 1963.

3. Arden, B., Galler, B., and Graham, R.: "An Algorithm for Translating Boolean Expressions." Journal of the ACM, April 1962.

4. "Procedure Oriented Language Statements to Facilitate Parellel Processing." Communications of the ACM, May 1965.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Goodyear Aerospace Corp	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE
Advanced Computer Organization Study
Volumes I and II

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Final Report August 1964 - November 1965

5. AUTHOR(S) (Last name, first name, initial)
Rohrbacher, Donald L.

6. REPORT DATE April 1966	7a. TOTAL NO. OF PAGES 558	7b. NO. OF REFS L

8a. CONTRACT OR GRANT NO. AF30(602)-3550	9a. ORIGINATOR'S REPORT NUMBER(S) GER-12314
b. PROJECT NO. 4594	
c. Task 459406	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d.	RADC TR ~~TDR~~-66-7

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center GAFB, N.Y. 13440.

13. ABSTRACT Advanced general-purpose computer organizations capable of parallel data processing were studied. To achieve maximum system performance from highly parallel computer organizations, new solution models and programming techniques must be developed. Hence, the following three areas were investigated simultaneously:

1. Applications - Study of problems and their inherent degree of parallelism, and development of theoretical solution models for use on a parallel processor.
2. Programming - The programming of parallel solution models on the postulated computer organizations.
3. Machine Organization - Development of machine implementations capable of parallel data processing.

This study resulted in the design of two computer organizations (designated Machine I and Machine II) capable of parallel data processing and fast sorting and table searching in memory. These machine organizations were possible because of the development of a special memory that permits many processing and input-output units to access memory simultaneously without conflict.

The applications effort was focused on the development of solution models which exploited the maximum amount of parallelism resident within a problem. Two major problems were investigated: a dynamic programming problem, and parallel compilation. Detailed programs were written for the dynamic programming problem on Machine I and a parallel compilation algorithm on Machine II. These same problems also were programmed on the IBM 7090 to provide a standard of comparison. In both cases, the parallel processing capability of the machines afforded significant increases in speed of program execution.

DD FORM 1 JAN 64 1473

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer Programming Numerical Analysis						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through
______________________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through
______________________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through
______________________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

UNCLASSIFIED
Security Classification