THE UNIVERSITY OF TEXAS AT AUSTIN
ELECTRONICS RESEARCH CENTER
AUSTIN, TEXAS 78712

Engineering-Science Building 110, 471-3954

May 16, 1970

Dr. Eugene Vacca
National Aeronautics and Space Administration
Office of Scientific and Technical Information
(Code US)
Washington, D.C. 20546

Dear Dr. Vacca:

Attached is a final report describing the current status of research effort which has been supported during the last twelve months by grant number NGR 44-012-144.

Sincerely,

C.V. Ramamoorthy
Professor of Electrical Engineering

TABLE OF CONTENTS

INTRODUCTION	
SUMMARIES OF RESEARCH PROJECTS	
THE FORTRAN PARALLEL TASK RECOGNIZER	(Enclosure 1)
RESOURCE ALLCCATION AND THE AVOIDANCE OF DEADLOCKS	
FAST MULTIPLICATION ARRAYS FOR LSI IMPLEMENTATION	(Enclosure 2)
A SURVEY OF RESOURCE ALLOCATION DISCIPLINES	(Enclosure 3)
A HEURISTIC OPTIMIZATION ALGORITHM FOR ARRANGING AN N PAGE OPEN STRING	(Enclosure 4)
A CLASSIFICATION AND SURVEY OF COMPUTER SYSTEM PERFORMANCE EVALUATION TECHNIQUES	(Enclosure 5)
UNIFIED HARDWARE/SOFTWARE DESIGN	(Enclosure 6)
ORGANIZING A SPECIAL PURPOSE COMPUTER USING THE FFT TO PERFORM SPEECH ANALYSIS IN REAL TIME	(Enclosure 7)

INTRODUCTION

This report is divided into two parts: the first part consists of a short summary of the work accomplished to date in each of the problem areas covered by this report. Each of the summaries lists the objectives of the research effort, the results achieved to date, and proposed extensions for future effort in the area. The second part of the report contains detailed information on each of the research areas listed in the first part. These details may take the form of self-contained enclosures or brief discussions with attached enclosures.

RECOGNITION OF PARALLEL PROCESSABLE STREAMS IN COMPUTER PROGRAMS

OBJECTIVES

The objective of effort in this area is an investigation of the many implications created by processing in parallel independent segments of individual programs. This effort includes: i) the automatic detection of the parallel processable segments; ii) the determination of the suitability of a program for parallel processing; iii) machine organization to implement parallel processing techniques.

RESULTS

Work has been completed on the final working form of the FORTRAN Parallel Task Recognizer. Briefly, the recognizer is a program which accepts as input source programs written in FORTRAN. As output the recognizer generates a set of tables which indicate the parallel processability of source program tasks. A source program task is considered to be a single statement or a set of such statements.

All of the limitations contained in the early version of the recognizer have been eliminated. In its present form the recognizer analyzes source program statements within a single level in blocks of 200 statements. For programs whose number of executable statements exceeds 200, the result of the recognizer analysis is several sets of output tables rather than a single set. In order to achieve maximum parallelism, all that is necessary is that the various tables be analyzed to permit individual tasks to be shifted from one block of partitions to another.

Program analysis is broken down into three phases. In the first phase, all that is determined is the permissible transitions between source program tasks. In the second phase program loops are detected and labelled with a single task number. In the third phase, input-output relationships between

tasks are determined, and the matrix representation of the resulting task dependency graph is analyzed for the parallel processability by the method of precedence partitions.

Preliminary results of program analysis show that as the number of source program executable statements increases, most of the analysis time occurs during the second phase. In fact, for a source program with 100 executable statements, phase II takes up approximately 90% of the total analysis time. Recall that during the portion of the analysis the goal is the detection of source program loops. After these results were obtained, it was hypotesized that the smaller the number of source program loops the more "suitable" a program would be parallel processing. To test this hypothesis various loop-free programs were analyzed, and it was determined that phase II analysis time is indeed reduced in direct proportion to the number of source program statements. Overall analysis time is reduced only up to a point, however, Although phase II analysis time is reduced by eliminating loops, precedure partitions analysis time increases because the number of individual tasks is larger. Recall that all statements within a program loop are considered a single task. Thus it can be said that a program's suitability is improved as the number of program loops decreases so long as the number of executable statements does not exceed the break point figure.

EXTENSIONS

Program analysis is continuing in an attempt to develop an algorithm which can determine a program's suitability for parallel processing. To obtain this information, timing studies are being performed on test programs to verify the accuracy of the algorithm as it develops.

In addition to this, effort has been initiated in an attempt to determine a multiprocessor machine organization which can implement parallel processing techniques. This includes the development of algorithms for processor and central memory allocation.

RESOURCE ALLOCATION AND THE AVOIDANCE OF DEADLOCKS

OBJECTIVE

The objective of this research effort is the development of a unified software/hardware package, with the hardware part composed of fast, possibly microprogrammable cellular array modules easily adaptable to LSI implementation and dedicated to perform functions related to the dynamic resource allocation activities in a multiprocessing system.

RESULTS

A model for dynamic resource allocation in a multiprocessing environment has been defined. Feasibility requirements and basic assumptions to be considered, pertaining to the model parameters have been set. Basic algorithmic disci lines and system performance requirements for resource utilization, similar to the ones demanded by present day systems have been adopted. In addition, the avoidance of deadlocks is being considered as a criterion of primary importance and the model design is centered around this criterion.

An algorithm, to be associated with the model has been developed.

EXTENSIONS

1. Develop a queueing model which will complement the system and determine the behavior of the job input buffer. In other words adopt disciplines for the dynamic determination of external priority assignments.
2. Apply some probabilistic considerations in the reassignment of jobs to the priority levels within the system and study their effects on the model.
3. Implement and simulate the model by an ALGOL program to determine its behavior, efficiency and timing.
4. Design the required hardware Cellular Array modules to perform the operations called for by the algorithm.

5. Develop the subroutines or software mechanism, possibly in the form of a microprogram, which with the assistance of the hardware modules will perform the resource allocation.
6. Evaluate and compare the efficiency and performance of the algorithm to other existing algorithms performing resource allocation.

MEASUREMENTS AND MODELS

OBJECTIVES

Primary objective was to survey and to study the techniques of computer system performance evaluation. An extensive survey is contained in "A Classification And Survey Of Computer System Performance Evaluation Techniques."

Later efforts were focused upon developing a hueristic optimization algorithm for a specific problem affecting the performance of memory hierarchy componenets such as magnetic tape units and disk units. An original algorithm is presented in "A Heuristic Optimization Algorithm For Arranging An N Page Open String."

RESULTS - SURVEY

The widely distributed software capabilities coupled with increased compatabilities among hardware units has been propagating an ever increasing multitude of computer systems. Such systems aim at a common goal - total system optimization. But achievement of this ultimate goal has been delayed by the absence of a theoretical basis capable of providing a reference for standardized evaluations and comparisons.

This paper has classified and examined the presently available assortment of computer system evaluation techniques. The techniques have been partitioned with respect to computing jobs, hardware units, and operating systems. Within each classification, techniques representing analysis, simulation, and/or synthesis have been examined in order to expose the advantages and disadvantages inherent to the specific techniques.

Characterizing the present technology provides both an outline of what is available and an outline of what needs to be developed. Fundamental among the needs of the computer evaluation technology is a theoretical basis providing a reference for standardized evaluations and comparisons. First, standardized paramenters should be established. Such parameters must characterize global system performance; they must provide common denominators allowing local paramenters to be reduced. Then appropriate techniques may be generated to measure these parameters under known system environments and workloads. Certain typical classes of installations might be represented by standard workloads consisting of standardized kernels or application benchmarks. Special purpose installations for comparative testing in order to evaluate competitive proposals.

Besides evaluation for selection purposes there remains the important problem of monitoring the installed system. Such performance monitoring could be greatly benefited by

designing snuping feature. into the hardware units and the operating systems in order to collect utilizations and usage frequency statistics. Ideally such features would operate in a parallel and interference-free fashion. Those techniques such as software artifact which do cause interference should provide means for their selective use. Built-in snuping features would allow computation centers to continually monitor their systems in order to provide early detection of system environment changes and precipatating problems. Also, such field usage statistics would provide the manufacturer much important input for future system designs.

RESULTS - ALGORITHM

Presented is a heuristic solution to a general optimization problem common to memory hierarchy components which can be characterized as possessing only one degree of access freedom. Included are: disk units which possess only one head per surface and whose head movement time is much greater than the disk rotation time; and tape units possessing bi-directional read/write capabilities. The resulting optimization problem can be formulated in graph theory teminology as follows.

Given a weighted, directed complete graph of N nodes, determine an open string of N nodes (i.e. an ordering) such as to minimize

$$\sum_{j=1}^{N} \sum_{i=1}^{N} (f_{ij}d_{ij} + f_{ji}d_{ji})$$

where

$f_{ij}d_{ij}$ denotes the weight of the edge from node i to node j

f_{ij} is a constant that denotes the density of the path from node i to node j

d_{ij} is a variable that denotes the "length" of the path from node i to node j (paths between consecutive nodes are equal)

The proposed algorithm possesses marked advantages over optimization by means of an exhaustive search of the N! possible permutations. For such an exact optimization technique the run time as a function of the number of nodes increases factorially! Run time for the developed algorithm possesses a lower bond proportional to $(N^3-N)/3$ where N equals the number of nodes.

Presently the algorithm has been implemented in FORTRAN IV and is being run on a CDC 6600 at the University of Texas. After additional refinements of the code it is intended that the algorithm be applied toward optimizing certain memory hierarchy components in the system. Evaluation would include the analysis of gains in the nature operating system.

Algorithm's "optimum" arrangements compare closely to known true optimum arrangements. For strings of 5 or 6 nodes, results have been identical to the true optimums which were exhaustively determined. Comparisons to larger strings become impractical due to the factorial growth rate of the exhaustive technique required to determine the true optimum.

EXTENSIONS

Primary objective will be to develop a simulation model suitable for studying multiprocessing computer systems.

Secondary objective will be to employ the developed model to evaluate various multiprocessing system configurations as a function of system parameters.

A flexible, general purpose simulation model will be required to accommodate the indicated studies of multiprocessing systems as a function of system parameters. The model will allow different hardware resources and configurations to be defined and to be characterized by properties such as average access time, average transfer

rates, and average instruction execution times. From inputted descriptiors of job streams, the model will simulate the behavior of the multiprocessing system.

An unique aspect of this study will be to simulate an adaptive operating system driven by a System Simulator. See Figure 1.

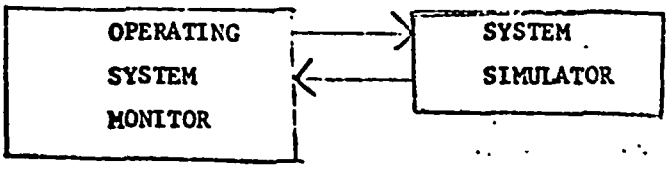


Figure 1

The System Simulator may be considered to be an individual computer unit possessing data paths communicating with the monitor of the multiprocessing system. Physically the System Simulator might be a prepherical processor of the CDC 6600 or a separate mini-computer (low cost). Function of the System Simulator will be to monitor and gather significant everts related to the current status of the host multiprocessing system. Such events would include input job queue descriptions, resource availability, etc. Using such information the Systems Simulator would simulate the future (several minutes) system behavior. As a result of this real time simulation, the System Simulator can feed back system parameter changes which would increase the host system performance. System parameters such as scheduling algorithms would be dynamically altered. Such an adaptive multiprocessing operating system promises significant benefits.

UNIFIED HARDWARE/SOFTWARE DESIGN

OBJECTIVE — This report describes progress made on research concerned with developing some theoretical approaches to handle hardware/software tradeoff decisions in computer design situations whose prospective operating environment is reasonably well known. In particular items covered are:

RESULTS

1) Formulation as a constrained optimization problem.
2) Identification as a multiple choice programming problem and discussion of a very rapid and efficient near optimal solution technique available.
3) Coverage of deficiencies of the multiple choice formulation precluding engineering application.
4) Alleviation of these deficiencies.

SPECIFICALLY:

a) Concept and treatment of sub-systems.
b) Approaches possible for mixed selection of both hardware and software choices for a given subtask.
c) Modification of solution technique to handle a) and b).
d) Consideration of modular memory in software resource usage.
e) Interconnection Costs.
f) Vague data, senstivity information and need for a collection of good solutions as opposed to a single optimal solution.

EXTENSIONS — Currently a considerable proportion of the preceding has been implemented as a computer program. Modular memory, and parametric senstivity considerations are still to be included. For the future, research to develop methods for handling optional subtasks and different types of objective functions other than execution time are planned.

INVESTIGATION OF THE ORGANIZATION OF A SPECIAL PURPOSE COMPUTER FOR SPEECH PROCESSING

OBJECTIVES

Speech properties and signal processing techniques have been investigated with the results being used to develop the design parameters and architecture for a special purpose digital computer for speech analysis.

The many useful applications for speech processing include bandwidth reduction, speeding of speech for the blind, slowing of speech for retarded children and language instruction, frequency division for aid to the partially deaf, automatic speech recognition, and automatic speaker recognition.

RESULTS

Analog Speech processors using the bank-of-filter type spectral analyzers have been in use since the late 1930's and have not yet been displaced by digital speech processors. However, the recently developed fast Fourier transform and cepstrum techniques, reduction in price of digital integrated circuits, and rapid advance of large scale integration (LSI) technology have made it economically feasible to construct special purpose computing devices which can perform speech processing tasks in real time.

Software implementations of the FFT have reduced computation time for many problems by as much as two orders of magnitude. Even greater gains can be realized through special-purpose hardware designed specifically for performing the FFT algorithm in a specific application such as speech processing. Applications for special-purpose FFT processors result from signal processing problems which have an inherent real-time constraint or which involve off-line processing where the volume of data makes processing impractical unless a dedicated machine is used.

Both of these requirements exist for a speech analyzer where real time processing may be required for bandwidth compression or automatic speech processing or large volume of data are required for processing for statistical analysis of speech.

Detailed investigations of speech properties, the fast Fourier transform and cepstral techniques have been performed in order to establish the design parameters and constrains for a special purpose speech analyzer. Research has been performed in the basic areas of organization options, the processor flow chart, sampling requirements, timing requirements and system architecture.

The speech analyzer was designed to obtain the short time spectrum of the speech waveform, the fundamental pitch frequency, and the voicing condition. A sampling rate of 12.8 KHz is used with a record of 20 milliseconds thus giving 256 sample points per record. Although 256 sample points are used per record, a 512 point Fourier transform is used. The 256 sample points are bordered with 256 zeros in order to enhance the cepstral peak and extend the minimum detectable pitch frequency from 100 Hz to 50 Hz.

The processor was designed as a completely hardwired programed computer. However, the architecture provides separate controllers for each algorithm within the processor so that individual algorithms may be modified independently of the other algorithms. In addition it would be possible to add a software capability with out change to the existing architecture. Thus the speech analyzer design could be modified easily to perform signal processing in areas other than speech.

A special purpose computer such as the one considered in this research generally is not a stand-alone device. In other words, it needs to be interfaced with a general purpose computer in order to perform really useful work. The special purpose computer and the

and the general purpose computer may be side by side or they may be many miles apart. In either case there are many items to be considered if the two computers are to communicate with each other. Items to consider include (1) distance of transmission, (2) amount of data to be transmitted, (3) error requirements, (4) coding used, (5) protocol between the two computers and other considerations normally associated with digital data transmission.

EXTENSIONS

Because of the requirements in the paragraph above investigations have begun in the general area of digital data communications with the goal of later applying this general field of knowledge to the requirements for linking special purpose processors to general purpose computers.

ENCLOSURE 1

THE FORTRAN PARALLEL TASK RECOGNIZER

FORTRAN PARALLEL TASK RECOGNIZER

I. INTRODUCTION

The FORTRAN Parallel Task Recognizer is a program which accepts as input a source program also written in FORTRAN. As output the recognizer generates a list of tasks within the source program which can be executed in parallel. The background and theory of operation of the Recognizer in its initial form have been described earlier.[1,2] The Recognizer in its present form, however, is considerably different from its initial form.

This report will describe in detail the working mechanisms of the recognizer program. Throughout this discussion reference will be made to a particular example program, a listing of which is contained in Appendix I. Before discussing the recognizer program, however, the graph model upon which the program is based will be reviewed. The references previously cited have discussed the graph model in a general manner. In contrast to this, the graph model discussion which follows will be keyed to the same example program mentioned earlier.

II. GRAPH MODEL FOR PARALLEL TASK EXECUTION

Throughout this discussion the term task will refer to a single FORTRAN statement or a set of FORTRAN statements. Notice that in the example program (Appendix I) each executable statement has a unique task number associated with it. This task number is assigned to individual statements by the recognizer program.

The first step in the generation of the graph model for the source program is construction of the PERMISSIBLE TRANSITION GRAPH. This graph provides an indication of the transfers of operation permitted by the source program itself. For example, only task number 2 can follow task number 1. That is, after the READ statement of task 1 is completed, the PRINT statement of task 2 can be initiated. That this is so is simply a result of the rules of the programming language. When no explicit transfer is indicated, one task simply follows the other.

The situation is different, however, with the GO TO statement which constitutes task number 33. Depending on the state of the index MI, after task 33 is completed, the next task to be executed is either task 34 or task 36 or task 38. That is, permissible transitions exist between task 33 and each of its possible successors. These transitions can be either actual or potential. The permissible transition graph for the example program is shown in Figure 1.

A model of this type can be represented in a computer by means of a square, binary matrix of dimension NxN, where N is the number of tasks to be analyzed. An entry c_{ij} in this matrix is a "1" if and only if a transition exists from task i to task j and it is "0" otherwise.

The second step in the generation of the graph model is identification and elimination of the loops contained within the permissible transition graph. A loop or maximal strongly connected subgraph (M.S.C.) can be

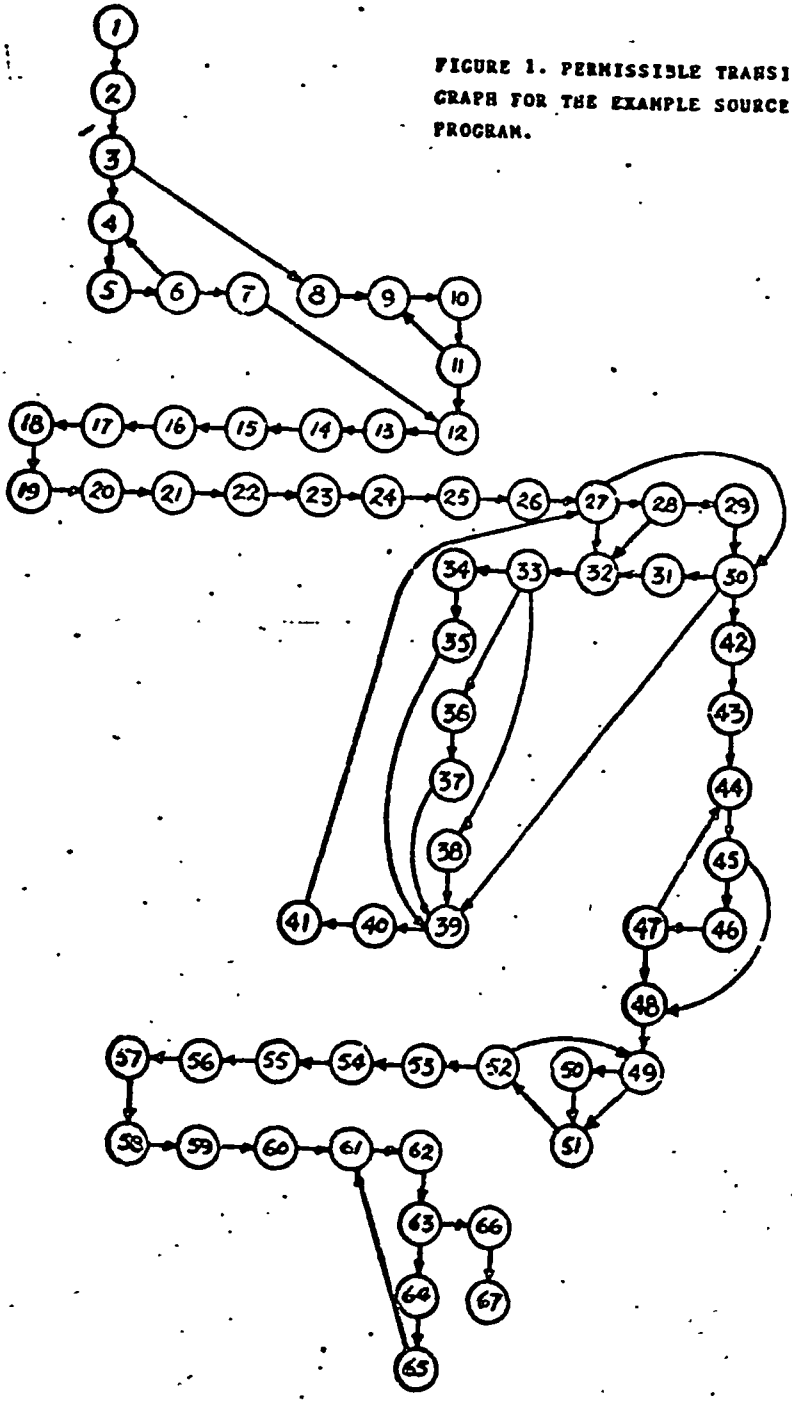


FIGURE 1. PERMISSIBLE TRANSITION GRAPH FOR THE EXAMPLE SOURCE PROGRAM.

identified visually by recognizing branches which "point backwards." That is, a transition exists from one task to another task which preceded it in the sequential instruction stream. More formally, however, a M.S.C. subgraph can be said to consist of a set of tasks which are strongly connected with each other, i.e., any task in the subgraph is reachable from any other. An algorithm for detection of the M.S.C. components is given below.[3]

Let [C] be the matrix which represents the permissible transition graph. Construct the REACHABILITY matrix as follows:

1) Generate an N-dimensional row vector $\underline{R}_1$ equal to the first row of [C].

2) Examine $\underline{R}_1$ for those entries which are non-zero. Form the union between $\underline{R}_1$ and the rows in [C] which correspond to the non-zero entries in the initial $\underline{R}_1$; that is, $\underline{R}_1$ (new) = $\underline{R}_1$ (old) $\cup$ $\underline{C}_i$. Repeat these procedures until no changes occur in $\underline{R}_1$.

3) Repeat Step 2 for $\underline{R}_2, \underline{R}_3, \ldots, \underline{R}_N$. The resulting collection of row vectors is the reachability matrix [R].

4) Construct the reaching matrix, $[R]^T$.

5) Construct $[M] = [R] \cap [R]^T$. The number of M.S.C. subgraphs is given by the number of distinct non-zero row vectors in [M]. The non-zero elements of each of these row vectors are the components of the M.S.C. subgraphs. In terms of the program graph, these elements are the tasks which constitute a loop.

If each of the M.S.C. subgraphs is considered a single task, the result is the REDUCED PROGRAM GRAPH. Using the algorithm described above, the FORTRAN Parallel Task Recognizer detects the M.S.C. subgraphs and assigns a single number to each of the subgraphs. The number assigned to each of the sets of tasks is the task number of the first task (i.e., the lowest numbered task) in the loop. The reduced graph for the example program is shown in Figure 2.

Up to this point all that has been accomplished is the generation of a series of sequentially ordered loop-free tasks. In order to analyze the program for potential parallelism, it is necessary to give consideration to the relationships between individual tasks. If in the reduced program graph a transition exists from task i to task j, it does not necessarily follow that the initiation of task j must follow the completion of task i. However, if task j depends on a result generated by task i, then a definite ordering in time is said to exist.

Consider, for example, tasks 2 and 3 in the example program. The reduced program graph shows a directed branch from task 2 to task 3. Normally this would imply that task 3 cannot be initiated until task 2 is completed. A closer examination of the tasks, however, indicates that task 3 depends only on the outcome of task 1 for its successful execution. The same can be said for task number 2. In fact, then, tasks 2 and 3 can be initiated simultaneously immediately upon completion of task 1. If this type of analysis is performed for all tasks represented in the reduced program graph, the PARALLEL PROGRAM GRAPH which results provides a

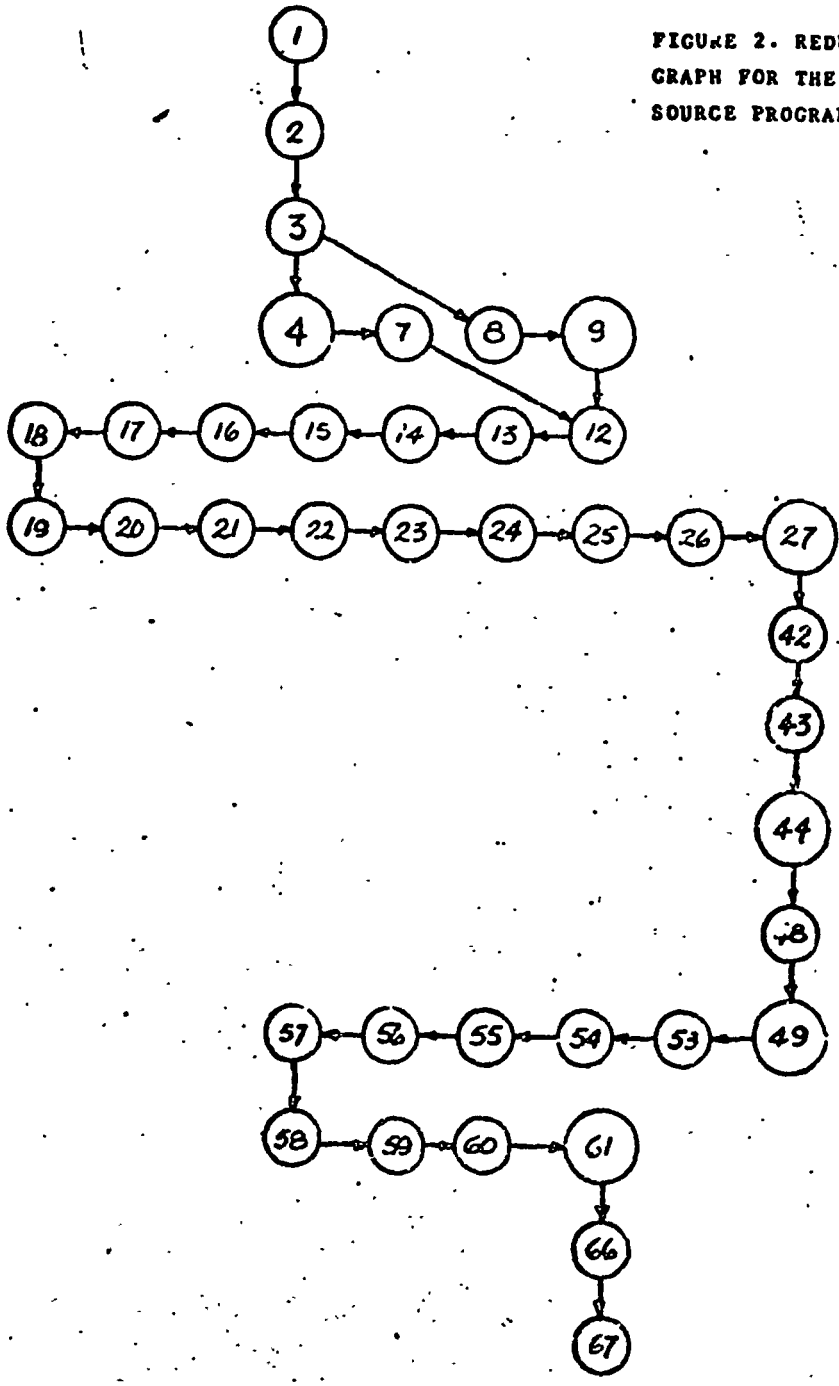


FIGURE 2. REDUCED PROGRAM GRAPH FOR THE EXAMPLE SOURCE PROGRAM.

means of automatically identifying parallel processable tasks. The parallel program graph for the example program is shown in Figure 3.

Before describing the technique for performing this identification, the following comment will be made with regard to Figure 3. If a task within a M.S.C. subgraph depends on the outcome of a task outside the subgraph, the directed branch is drawn from the predecessor task to the task representing the entire subgraph. This follows from the earlier discussion in which it was stated that a single task number is assigned to all the components of a M.S.C. subgraph. In a similar vein, if an ordering exists between members of a M.S.C. subgraph, this fact is not reflected in the parallel program graph. (That is, in a matrix representation of this graph $c_{ij} \neq 1$ for those i and j which belong to a subgraph).

The method used to determine task parallelism is referred to as PRECEDENCE PARTITIONS.[1,2,4] The method relies on the matrix representation of the parallel program graph. Recall that the label of a row or a column in this matrix represents a task number, e.g., the third column represents task number 3, the eighteenth row represents task number 18, etc. To determine the parallel processable tasks, the entire matrix is examined for columns all of whose entries are zero. During each examination, the columns which contain only zeros represent tasks which can be executed in parallel. After these columns have been detected, the

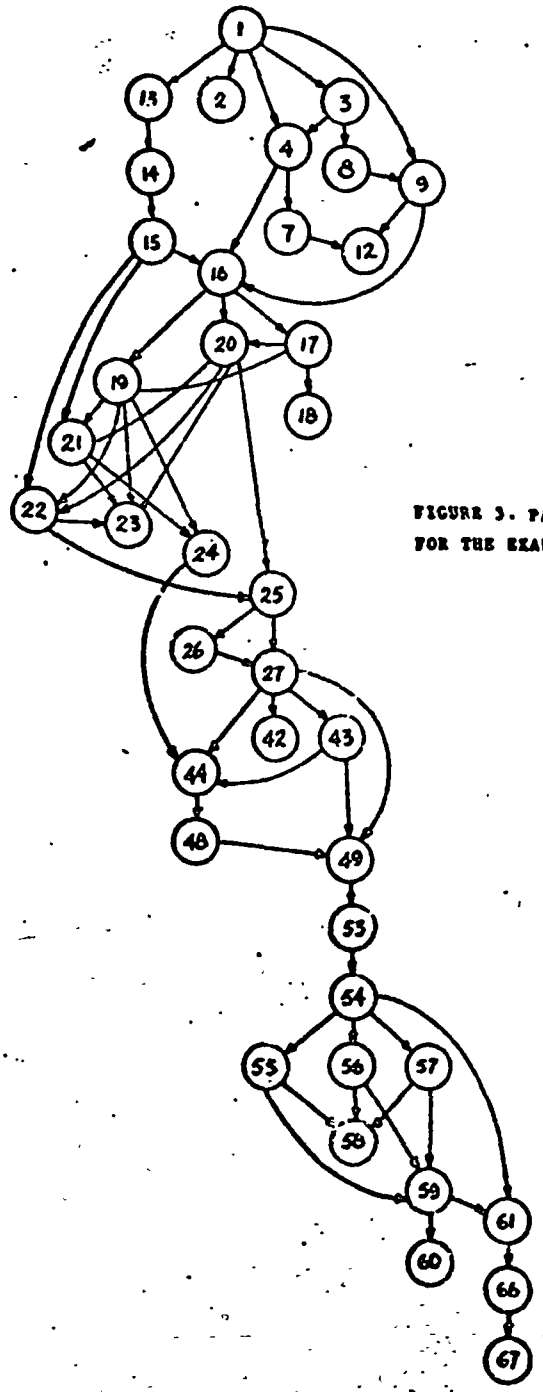


FIGURE 3. PARALLEL TA*X GRAPH
FOR THE EXAMPLE SOURCE PROGRAM.

columns and the corresponding rows are eliminated from further consideration. The process is repeated until the entire matrix has been examined. Each examination or partition corresponds to the earliest time period during which the resultant tasks can be initiated. To determine the latest time during which a task can be initiated, the process is repeated with the following change: instead of looking for all-zero columns, look for all-zero rows. That is, perform row partitions instead of column partitions.

The tables which result from the procedures described above are shown in Appendix II. Reference [2] which contains more details on the graph model described here is enclosed as Appendix V.

III. FORTRAN PARALLEL TASK RECOGNIZER

This section will describe in terms of the actual recognizer program the implementation of the graph model described in the previous section. An overall flow chart of the recognizer program is shown in Figure 4. The discussion which follows will be based on this flow chart. In addition, two other items of information are enclosed to help clarify some of the comments made throughout this discussion. Appendix III contains a listing of some of the intermediate tables generated by the recognizer program. Appendix IV contains a listing of the actual recognizer program.

PHASE I

The objective of Phase I is to analyze the entire source program and to generate a set of tables which will permit generation of the three graphs described in the previous section and shown as Figures 1, 2, and 3.

The FORTRAN Parallel Task Recognizer is a program which analyzes a FORTRAN source program and generates as output a listing of the tasks within the source program which can be executed in parallel.

The types of executable FORTRAN statements which are accepted by the recognizer are listed below.

1) DO
2) READ
3) IF (One branch and two branch logical and three branch arithmetic)
4) PRINT
5) CONTINUE

6) CALL

7) END

8) GΦ TΦ (Assigned, Computed, Unconditional)

9) RETURN

10) ASSIGN

11) ARITHMETIC EXPRESSIONS (Replacement)

Declarations of the following types are acceptable.

1) REAL

2) COMMON

3) INTEGER

4) DIMENSION

5) DATA

6) LOGICAL

7) EQUIVALENCE

8) DOUBLE

9) COMPLEX

The recognizer consists of a main program and seventeen subroutines. The discussion which follows will be based on these items.

Main Program

The basic function of the main program is to extract all the pertinent information concerning a source program statement and to transmit the statement to a subroutine for detailed analysis. In a sense the

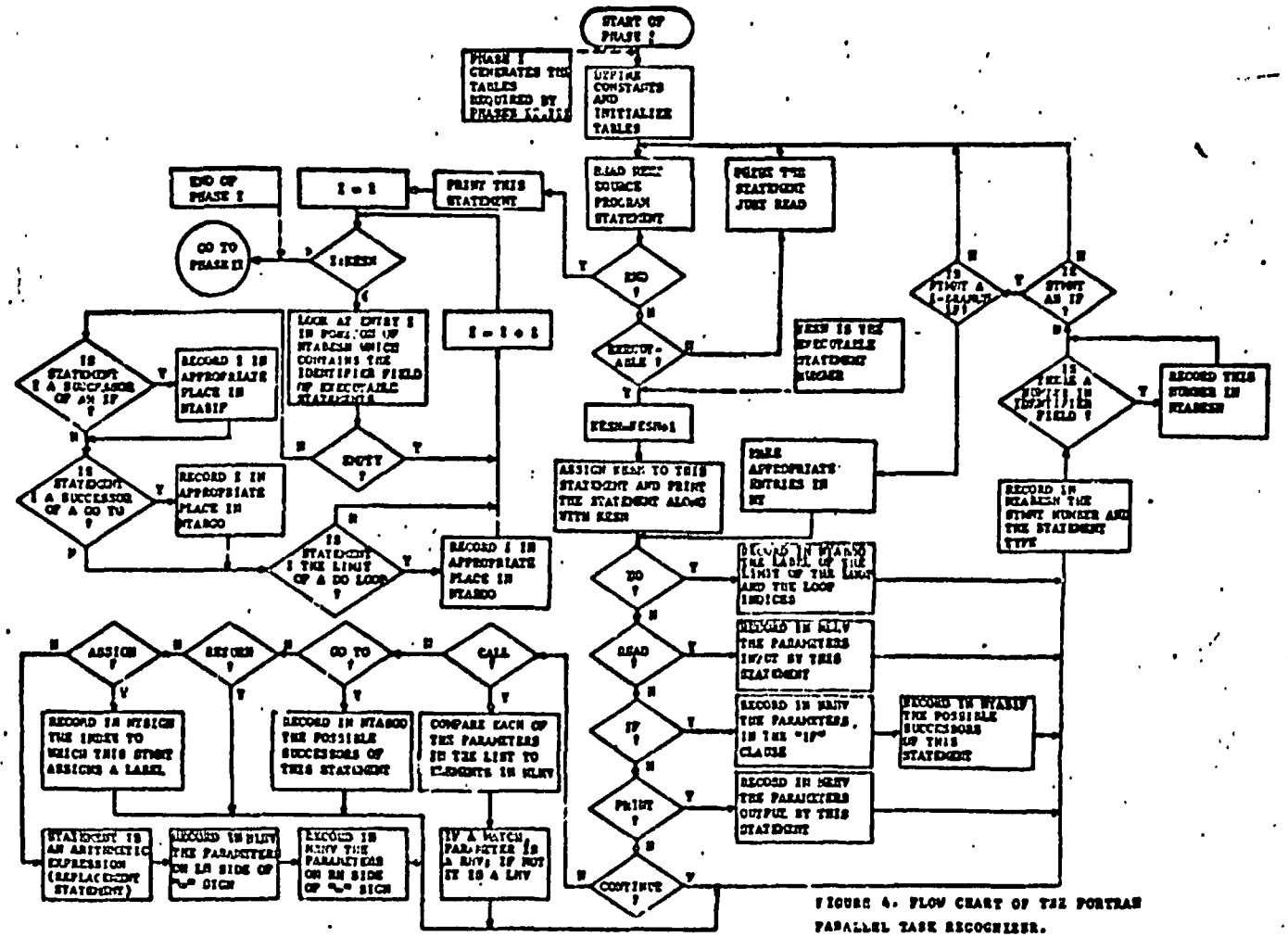


FIGURE 4. FLOW CHART OF THE FORTRAN PARALLEL TASK RECOGNIZER.

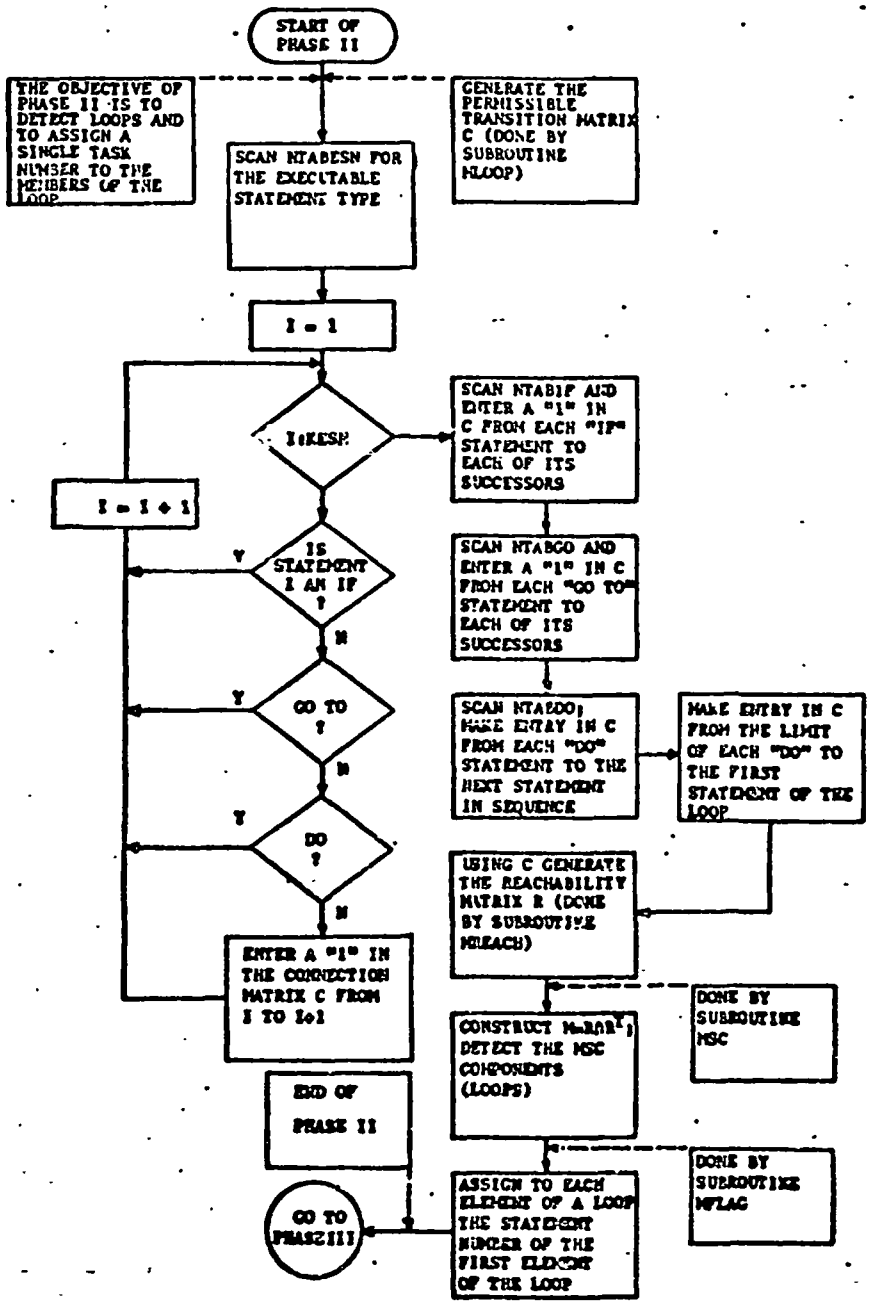


FIGURE 4. FLOW CHART OF THE FORTRAN PARALLEL TASK RECOGNIZER (CONTINUED).

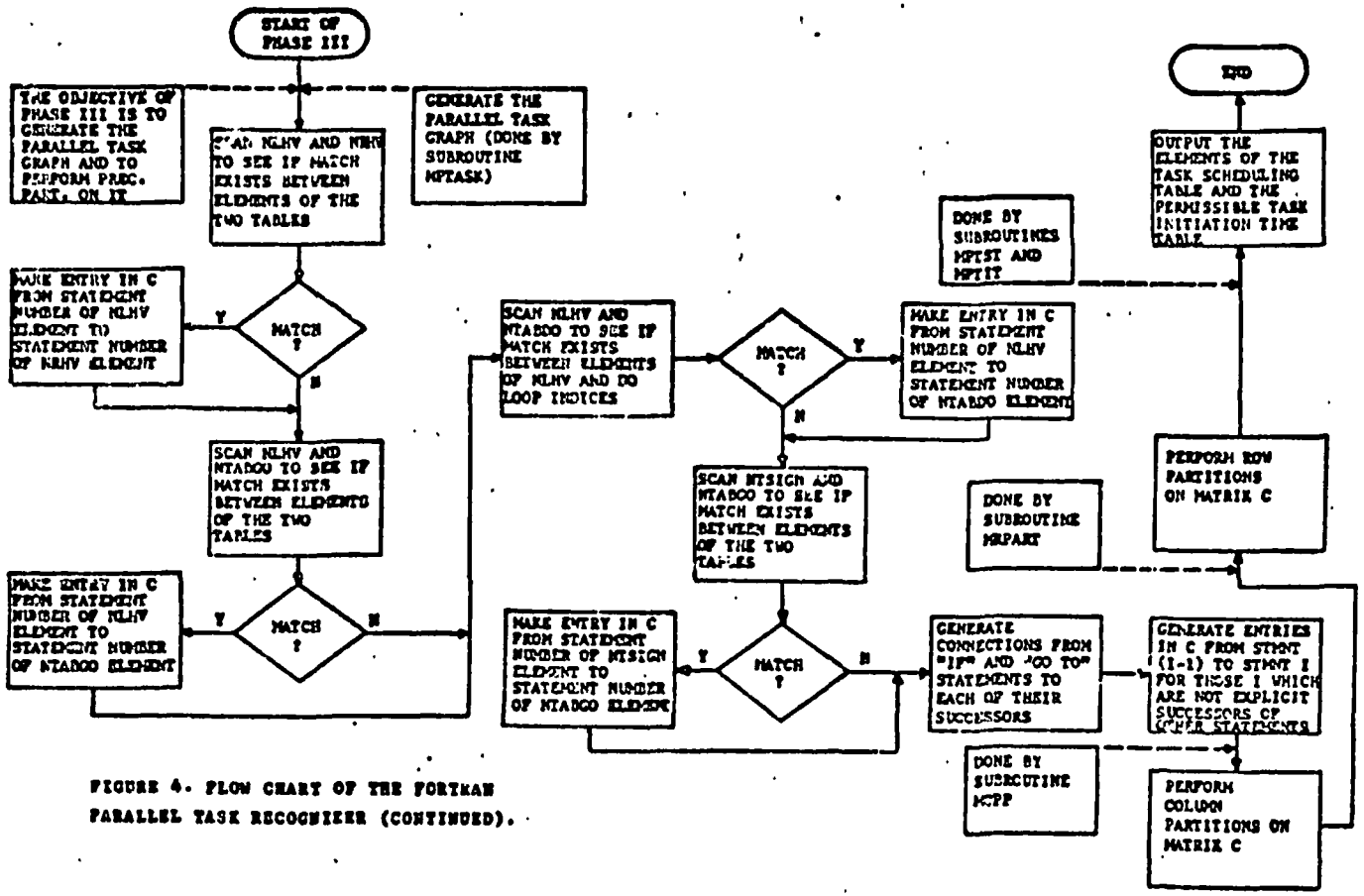


FIGURE 4. FLOW CHART OF THE FORTRAN PARALLEL TASK RECOGNIZER (CONTINUED).

recognizer functions in an interpretive mode since no source program statement is ever looked at more than once.

The acceptable statement types are stored internally in their display code representation. Incoming source program statements are compared to the acceptable statement types until a match is found. If a source program statement is nonexecutable, it is merely printed out and no further action is taken. If the statement is executable, however, several things happen.

First, the executable statement number (KESN) is incremented by one and assigned to that statement. Second, the executable statement type (DO, READ, IF, etc.) is determined. For example, a DO statement is of type A; a READ statement is of type B, and so forth. Third, if a label is found in the identifier field of that statement, that information is stored also. These three items of information are stored in table NTABESN. For example, the appropriate entry in NTABESN for the first DO statement in the example program would be as shown below.

NTABESN

4	A	10

This means that the first DO statement was the fourth executable statement recognized; the statement was of type A (i.e., a DO statement); and the label in the identifier field was the number 10.

After these determinations are made, the source program statement is turned over to the appropriate subroutine for detailed analysis. In keeping with the format of the flow graph of Figure 4 and the way the program is actually written, the recognition subroutine will now be discussed in detail. Before doing this, however, several comments will be made regarding the format of statements which the recognizer will accept.

In general, imbedded blanks within variable names are <u>not</u> permitted. Thus, for example, where the statement

READ 100, NAMEL, NAME2

is in proper form, neither of the following statements is acceptable:

REbAD 100, NAME1, NAME2

READ 100, NAMbE1, NAME2

READ 100, NAME1b, NAME2

READ 100, NAME1, bNAME2

(NOTE: The symbol "b" represents a blank.)

The recognizer requires that all source program statements begin in Column 7 of the punched card representation of the statement. Continuation statements (statements requiring more than one card) are not permitted; an entry in the identifier field of a statement must terminate in Column 5, and only one statement per card is allowed. In addition to those general remarks, additional constraints which may exist will be mentioned when individual subroutines are discussed.

Subroutine MDO

This subroutine is called by the main program and is designed to analyze DO statements. With reference to the recognizer, the following items are significant.

1) The label of the statement which represents the limit of the DO loop.

2) The initial value assigned to the index variable.

3) The largest value assigned to the index variable.

For purposes of illustration, consider the two DO statements which appear in the example program.

a) 10 DO 20 I = 1,10

b) DO 140 L = NW1, NW2

In statement a) only item 1 applies, that is, the limit of the DO is the statement labelled "20". In statement b) all three items apply: the limit of the DO is the statement labelled "140"; the initial value of the index variable is "NW1", and the largest value assigned to the index variable is "NW2". These items along with the executable statement number and the table entry number are stored in table NTABDO. (See Appendix III.)

To determine item 1, scanning begins in Column 10 and continues until a blank is found. The characters between Column 9 and this blank represent the label of the limit of the DO. Scanning continues until an

"=" sign is found. This character represents the left boundary of Item 2. If the next character is a digit, Item 2 is a constant and does not have to be stored. If it is not a digit, however, the variable name between the "=" and the next "," represents the starting value of the variable index. This same comma represents the left boundary of Item 3. As before, if the next character is a digit it is ignored and scanning stops. If it is not a digit, Item 3 is the variable name between the comma and the next blank. The recognizer can accept twenty DO statements in a source program. Imbedded DO statements are acceptable.

Subroutine MREAD

This subroutine is called by the main program and is designed to analyze READ statements. In the previous section it was stated that the generation of the parallel task graph was the final step before implementation of the process of precedence partitions. In the opening paragraph of this section it was stated that once a statement is analyzed it is printed and never processed again. It follows, then, that the information generated during the recognition phase must be stored whether or not it is to be used immediately.

In the case of a READ statement, the parameters input by that statement are strictly of the "left-hand" type. That is, parameters input by a READ statement are in effect outputs, i.e., these parameters serve as inputs to subsequent statements. These concepts can be expressed as shown below.

OUTPUT OF A READ = PARAMETERS INPUT BY A READ

INPUT TO SUBSEQUENT STATEMENTS = OUTPUTS OF A READ

Thus the parameters input by a READ are stored in table NLHV which contains only variables of the "left-hand" type. (See for example, statements (tasks) 1, 15, and 54 in the source program and entries 1, 7, and 22 in NLHV in Appendix III.)

In this subroutine scanning begins in Column 11, but all entries are ignored until the comma following the format statement number if found. Thereafter, input parameters are those found to reside between delimiting commas (or a comma and a blank in the case of the last member of the list). Note that in an implied DO loop in a READ statement, for example,

READ 100, (A(I), I = 1,10), B, C

the delimeters for the input parameter A are the two "(". After the second "(" is found, subsequent entries in the statement are ignored until the second ")" and the following comma are found.

All variables input by a READ statement are stored in NLHV. NLHV (and NRHV to be discussed later) is limited to ten variables per entry and 200 entries. The latter restriction results from the fact that the recognizer can accept only programs which have 200 or less executable statements.

Subroutine MIF

This subroutine is called by the main program and is designed to analyze IF statements. All three types of IF statements are accepted by the

recognizer. The three types are shown below.

1) THREE - BRANCH ARITHMETIC IF

   IF (A) n_1, n_2, n_3

2) TWO - BRANCH LOGICAL IF

   IF (L) n_1, n_2

3) ONE - BRANCH LOGICAL IF

   IF (L) s

In the three types listed above, A is an arithmetic expression, L is a logical expression, the n_i are statement labels, and s is a statement. The recognizer analyzes A or L for the variables found therein. Since these variables must have been generated earlier, they are "right-hand" (RH) variables, that is, the elements of A and L are outputs of previously executed tasks. These RH variables are stored in NRHV. The delimeters for the overall expression are the matched pair of parentheses shown above as (A) or (L).

After the elements of the expression are detected and stored, it is necessary to record the successors of the IF statement. For types 1 and 2 above, these are 3 and 2 statement labels, respectively. These labels along with the statement number and the table entry number are stored in table NTABIF. NTABIF is a table dimensioned 20x8. This means that the recognizer will accept programs with twenty or less IF statements. The successors are stored in columns 3,4, and 5 of NTABIF. Entries in columns 6,7 and 8 are made by the main program and will be discussed later.

Type 3 IF statements are altogether different since a statement rather than a set of statement labels follows the logical expression. When a type 3 IF statement is detected, a flag must be set in the 1x4 array named NT. To set the flag, NT(1) is set to 1; it is 0 otherwise. NT(2) contains the word number of the word containing the first character of the statement following the logical expression. (Input statements are read in by an (R6, 9R8)format. Thus if the first character of the successor statement is found in column 45, NT(2) = 6). NT(3) contains the column number of the first character of the successor statement (in this example, NT(3) = 45). After these conditions are noted, control is returned to the main program. Attempts are made to match the successor statement type to one of the executable statement types until a match is found. At this point processing continues as if the successor statement were a normal statement beginning in Column 7. Successor statements are limited by the recognizer to the following types:

1) GO TO (All 3 types)
2) ASSIGN
3) CALL
4) READ
5) PRINT
6) ARITHMETIC (REPLACEMENT)

Subroutine MAE

This subroutine is called by the main program and is used to analyze ARITHMETIC EXPRESSIONS and replacement statements. The analysis is done in two parts as described below.

1) The first part consists of analysis of everything to the left of the "=" sign. Simple variables are stored in NLHV. Subscripted variables are also stored in NLHV, but the variable index is stored in NRHV. For example, statement 46 in the example program reads as follows:

130 NSTØRE(L) = M4 + M5

Here NSTORE is a LHV. L, however, must have been generated earlier. Therefore, it is a RHV.

2) The second part of the analysis consists of everything to the right of the "=" sign. All variables detected by this part of the analysis are stored in NRHV.

Subroutine MGØTØ

This subroutine is called by the main program and is designed to analyze GØ TØ statements. All three types of GØ TØ statements are accepted by the recognizer. The three types are shown below.

1) UNCONDITIONAL GØ TØ

   GØ TØ n

2) ASSIGNED GØ TØ

   GØ TØ m, $(n_1, n_2, \ldots, n_m)$

3) COMPUTED GØ TØ

   GØ TØ $(n_1, n_2, \ldots, n_m)$, i

In addition, an asterisk (*) is placed in Column 3 of NTABIF to indicate that that statement is a one-branch IF.

Subroutine MRPRINT

This subroutine is called by the main program and is used to analyze PRINT statements. Scanning begins with Column 12, but all entries are ignored until the comma following the format statement number is found. Variables in the output list are found within pairs of commas (or a comma and a blank in the case of the last number of the list) and are stored in NRHV.

Subroutine MCALL

This subroutine is called by the main program and is used to analyze CALL statements. Detection of variables in the subroutine parameter list is accomplished in a manner similar to that which has already been described. All entries are ignored until the "(" indicating the beginning of the parameter list is found. Scanning ceases when a blank is detected.

Determining whether the entries in the parameter list are LH or RH variables proceeds in the following manner. If an entry in the parameter list matches an entry in NLHV, the entry is a RH variable and is stored in NRHV. If no match is found, it is assumed that the variable is generated by the called subroutine. It is, therefore, a new variable and it is stored in NLHV.

Here the n_i are statement labels; i is preset or computed prior to its use in the GO TO; m is a simple integer variable assigned an integer value in a preceding ASSIGN statement.

Scanning begins in Column 12. The presence of a "(" in this column indicates the presence of a computed GO TO. The various statement labels are delimited by pairs of commas or "(" and "," in the case of the first and "," and ")" in the case of the last. The statement labels are stored in table NTABGO which is dimensioned 20x25. This means that the recognizer will accept no more than twenty GO TO statements per source program. Column 1 in NTABGO contains the table entry number; column 2 contains the executable statement number; column 3 contains the number of statement labels contained within the GO TO option list; and column 4 contains the integer variables m or i when the GO TO is of type 2 or 3. Therefore, no GO TO statement can contain more than ten labels in the label option list. The successors of a GO TO statement are entered in NTABGO beginning in column 5. The statement numbers corresponding to these labels are filled in by the main program beginning in the first column following the last successor.

Given that a computed GO TO has been detected, the recognizer expects to find an integer variable name after the label list. The comma after the label list is optional.

If no comma is found in column 12, the GO TO is of type 1 or 2. scanning continues until a blank or a comma is found. If a blank is

found, the GO TO is unconditional. The statement label is stored and scanning ceases. If a comma is found, the GO TO is of the assigned type. The integer variable and the labels are stored until a ")" is found. Scanning ceases and control is returned to the main program at that point.

Subroutine MASSIGN

This subroutine is called by the main program and is used to analyze ASSIGN statements of the form

ASSIGN s to m

Scanning begins in column 19 and continues until a blank is found. At that point the simple integer variable m is stored in table NTSIGN along with the table entry number and the executable statement number. Source programs are limited to ten ASSIGN statements.

Each of the subroutines listed above returns control to the main program where the statement number and type and the contents of the identifier field are stored in NTABESN. If the statement just read is an IF statement, however, it is necessary to determine if it is a one-branch IF. If it is, it is necessary to perform analysis of the successor statement in the manner described earlier. If it is not, the next source program statement is read and the entire process is repeated.

The recognition and table-filling process is continued until an END statement is detected. At that point it is necessary to fill in the vacant entries in NTABIF, NTABGO, NTABDO. This is accomplished by analyzing the third column of table NTABESN. Recall that entries in

this field of NTABESN are statement labels. When an entry is found, an attempt is made to match that entry to one of the entries (statement labels) in NTABIF (columns 3,4,5), NTABGΦ (entries beginning in column 5), and NTABDΦ (column 3). When a match is found, the statement number (found in column 1) of the entry in NTABESN is entered in the appropriate place in each of the tables mentioned above (column 6,7, or 8 in NTABIF; column 4 in NTABDΦ; and beginning immediately after the last successor label in NTABGΦ).

Subroutine NVAR

This subroutine is called whenever a variable or label is detected by any of the subroutines listed above. The reason for this subroutine is that all entries in all tables used by the recognizer have to be right-justified with zero fill. As input NVAR expects: the word number of the word containing the first character of a variable or a label; the column number of the column following the last column occupied by a variable or a label; and the number of characters which make up a variable or a label. As output NVAR returns the variable or label as a right-justified, zero-filled word.

PHASE II

The objective of Phase II is to generate the PERMISSIBLE TRANSITION GRAPH and the REDUCED PROGRAM GRAPH.

The generation of the permissible transition graph is accomplished in the following manner. Column 2 of table NTABESN is searched for every instance of a DØ, IF, or GØ TØ statement. If a particular entry is neither of these three types, a "1" is entered in the connection matrix [C] (called [NC] in the recognizer) from that task to the next task in sequence. All entries in NTABESN are scanned in this manner, ignoring DØ, IF, and GØ TØ statements until the entire table is scanned. After this procedure is complete, the following occur.

1) A "1" is entered in [C] from the statement number of each entry in NTABIF to the statement number of each successor. Thus for the example program the following are true (See table NTABIF in Appendix III).

$$C_{3,4} = C_{3,8} = C_{11,9} = C_{11,12} = C_{28,29} = C_{30,31}$$
$$= C_{30,39} = C_{30,42} = C_{45,48} = C_{45,46} = C_{49,50}$$
$$= C_{49,51} = C_{52,49} = C_{52,53} = C_{63,64} = C_{63,66} = 1$$

Note that these entries merely represent permissible transitions which may result as outcomes of individual IF statements. These transitions are represented as directed branches in Figure 1.

2) A "1" is entered in [C] from the statement number of each entry in NTABGO to the statement number of each successor.

$$C_{7,12} = C_{27,28} = C_{27,30} = C_{27,32} = C_{28,32} = C_{33,34}$$
$$= C_{33,36} = C_{33,38} = C_{35,39} = C_{37,39} = C_{41,27} = C_{65,61} = 1$$

3) A "1" entered in [C] from the statement number of each DO statement to the next statement in sequence. Also, an entry is made in [C] from the statement number of the limit of each DO loop (obtained from NTABDØ) to the statement number of its respective DO statement. This operation, in effect, defines a "DO loop."

$$C_{4,5} = C_{6,4} = C_{44,45} = C_{47,44} = 1.$$

At this point construction of the permissible transition graph is complete. This graph is represented by the connection matrix [C]. Before discussing the construction of matrices [R], $[R]^T$, and [M], some comments will be made on the actual internal machine representation of these matrices as determined by the recognizer program.

Recall that to generate the loop-free graph the state of the following matrices must be preserved: [C], to generate [R]; [R] to generate $[R]^T$; [R] and $[R]^T$ to generate [M]. Since $[R]^T$ does not have to be preserved after the construction of matrix [M], [M] is stored in the same physical memory initially occupied by $[R]^T$. These statements are meant to imply that in order to generate the reduced program graph enough memory must be available to preserve the state of three 200x200 matrices. The number 200 stems from the fact that the recognizer can accept source programs containing up to 200 executable statements. If a single memory word is used to represent each matrix entry, a minimum of $120{,}000_{10}$ words of memory would be required for the matrices alone. This is an undesirable requirement especially since [C] (but not [R], $[R]^T$,

or [M]) is so sparsely populated. In view of this, it was decided to use individual bits within a word as matrix entries.

The CDC 6600 on which the recognizer program was run has a word length of 60 bits. However, for the matrices described here only the rightmost 40 bits of each word are used for matrix entries. The leftmost 20 bits are used for flags, pointers, and so forth. The number 40 (as opposed to say 42, 45, etc.) was chosen primarily because it is a multiple of 200 (the number of bits required to represent the desired matrices). Equally important, however, is the fact that in the CDC 6600 integer constants are limited to a value of $2^{48}-1$ during multiplication and division operations. (A detailed analysis of the recognizer program bit manipulations will show many instances of integer arithmetic wherein this restriction has to be observed).

Therefore, in the recognizer program a 200x200 matrix is represented by an array dimensioned 200x5. Thus for the three matrices described above the memory requirement is $3x(200x5) = 3000_{10}$ words. The 40 to 1 reduction in required memory is obviously a definite advantage. This advantage, however, comes at a cost of the considerable complexity required to access individual bits.

The connection matrix is generated by subroutine MLOOP in the manner described above. However, this subroutine does not make the

actual entries in [C]. This is accomplished by subroutine MCONN. MLOOP merely transmits the matrix coordinates of a "1" entry to MCONN. For example, if $C_{24,43} = 1$, MCONN determines that the entry is to be made in the 24th row and the second word of [C]. This stems from the fact that $40 < 43 < 80$. The matrix representation for this entry would be as shown below, assuming that no other entries have been made.

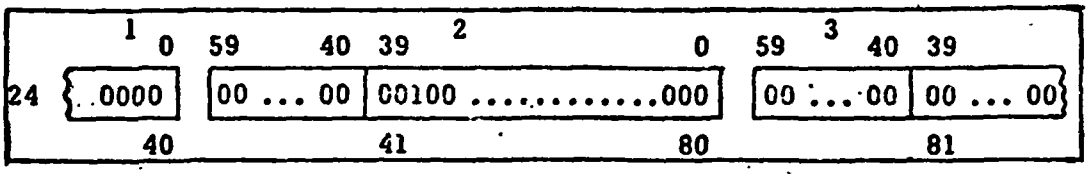


FIGURE 5. Matrix Representation of $C_{24,43} = 1$.

After MLOOP completes construction of [C], control is returned to the main program. At that point, subroutine MREACH is called; this subroutine generates the reachability matrix [R] (called NR in the recognizer program). In this subroutine a library function subroutine is introduced, LSHIFT. A typical application of LSHIFT is shown below

WORD = LSHIFT (WORD, N)

Here WORD is a word whose contents are shifted N bits. If N is positive, the contents of the word are shifted to the left; if N is negative, the contents of the word are shifted to the right.

MREACH forms [R] in the manner described in section II. Starting with the first row vector in [C], the non-zero entries are stored in a stack. The row vectors represented by the elements of the stack are

themselves analyzed for non-zero entries. These in turn generate new entries in the stack. After all of [C] has been analyzed in this manner, the first row vector in [R] consists of the union of the first row vector in [C] with all the row vectors whose labels are stored in the stack. The process is repeated beginning with the second row vector of [C], etc., until the construction of [R] is complete. Control is then returned to the main program which in turn transfers control to subroutine MSC.

Subroutine MSC does several things. First, it forms $[R]^T$, the transpose of the reachability matrix. This is accomplished by detecting each non-zero entry in [R], noting its coordinates, and storing a "1" in [M] in the transpose of those coordinates. [M] at this point is merely a temporary store for $[R]^T$.

At this point matrix [M] can be generated by forming a logical AND operation between each row vector in [R] and the corresponding row vector in $[R]^T$. The MSC subgraphs are all those distinct row vectors in [M] which are non-zero. The row numbers of these row vectors are stored in a temporary holding area. The elements of each of these row vectors are in turn stored in a separate array. The first element of each of these MSC subgraphs is indicated in this array by placing a "1" in bit 40 of the word containing the first element of the subgraph. In the example program, the first loop consists of tasks 4,5, and 6; the second loop consists of tasks 9,10, and 11. The corresponding entries in the array (called LS in the recognizer program) for these tasks are shown below.

LS

	59 ... 40	39 ... 0
1	0 ... 001	0 0100
2	0 ... 000	0 0101
3	0 ... 000	0 0110
4	0 ... 001	0 001001
5	0 ... 000	0 001010
6	0 ... 000	0 001011
7		

FIGURE 6. Representation of Table Which Contains MSC Components.

At this point control is returned to the main program which in turn transfers control to subroutine MFLAG. The sole function of MFLAG is to assign a single task number to each element of a MSC subgraph and to flag all those elements which are part of a subgraph. Those elements which are part of a MSC subgraph are all those elements stored in array LS. The flag is set in bit 40 of the second column of NTABESN. The task number assigned to these tasks is that of the first element in the subgraph. After termination of subroutine MFLAG, the first few entries of table NTABESN would appear as shown below.

	1		2	3
	59 0	59 40	39 0	59 0
1	1		B	
2	2		D	
3	3		C	
4	4	1	A	4
5	5	1	K	4
6	6	1	E	4
7	7		H	
8	8		K	
9	9	1	K	9
10	10	1	K	9
11	11	1	C	9

FIGURE 7. State of Table NTABESN at the end of Phase II.

At this point Phase II has been completed. Control has been returned to the main program and the proper entries have been made for initiation of Phase III.

PHASE III

The objective of Phase III is to generate the Task Scheduling Table, the Permissible Task Initiation Time Table plus associated tables (see Appendix V). The first step in this direction is the generation of the Parallel Task Graph (See Figure 3). The original [C], [R], and [M] matrices

do not have to be retained. The physical memory which was occupied by these matrices will now be used for other purposes. The area occupied by [C] will now be used for the matrix representation of the Parallel Task Graph. The area formerly occupied by [R] will now contain $[C]^T$. The parallel task graph is generated in six steps by subroutine MPTASK as shown below.

1) The first step is to check the relation between all LH and RH variables. Tables NLHV and NRHV are scanned to see if a match exists between elements of the two tables. When a match is found, a "1" is entered in [C] from the statement number of the LH variable to the statement number of the RH variable.

As before, entries in [C] are made by subroutine MCONN which receives a pair of coordinates (i,j) from MPTASK. For every (i,j) not only is an entry made at location c_{ij} but an entry is also made in another matrix at location $(c_{ij})^T$. (That is, [C] and $[C]^T$ are constructed concurrently). This latter operation is also performed by subroutine MCONN. Additional operations performed by MCONN will be discussed later.

In general, the following observation is valid throughout this stage of Phase III. A "1" is never entered at location c_{ij} when $i \geq j$. Although a match between an LHV and a RHV may be found when $i > j$, this would imply a different usage of the variable in question, since at this point loops have been removed from the program graph. When a match is found and $i = j$, it means that a transition exists between two statements of

an MSC subgraph. As was mentioned earlier, no entry is made in [G] under these conditions.

2) Tables NLHV and NTABGΦ are scanned to see if a match exists between a LHV and the variable index of a computed or assigned GΦ TΦ. The effect of this step is to generate a transition from a statement which sets the value of an integer variable to the GΦ TΦ statement which uses that variable.

3) Tables NLHV and NTABDΦ are scanned to see if a match exists between a LHV and the indices of a DO statement. The effect here is to draw transitions from the statements which set the value of the variable indices used by a DO loop to the statement which uses them.

4) The variable index used by an assigned GΦ TΦ statement must have previously been set by an ASSIGN statement. This step generates transitions from the various ASSIGN statements to their respective GΦ TΦ statements.

5) Step 5 draws transitions from each F and each GΦ TΦ statement to each of their explicit successors keeping in mind the "anti-looping" restrictions given in Step 1 above.

6) At this point several statements exist within the source program which are not covered by any of the previously listed conditions. For example, the READ statement of task number 17 in the example source

program is not an explicit successor of any other task. Implicitly, however, it is intended to follow task 16. Therefore, a "1" entered at location c_{ij} where i = 16 and j = 17.

After completion of step 6, construction of the parallel task graph (and its matrix representation) is complete. At this point column precedence partitions are begun by subroutine MCPP which is called by the main program after control is returned to it by subroutine MPTASK.

Recall that in column precedence partitions the objective is to find all-zero columns in the matrix representation of the parallel task graph. However, because of the word-bit organization of [C], this would be a difficult and time-consuming task. Therefore, a roundabout approach has been taken to circumvent this problem. For a column j in [C] to contain only zeros, the following must be true: $c_{ij} = 0$ for all i. Therefore, instead of inspecting [C] for each j and all i, it is only necessary to record the number of $c_{ij} \neq 0$ for each j. For example, references to Figure 3 shows that four transitions exist between predecessor tasks and task number 23. (In particular, i = 19,20,21,22). This information is stored for all j by subroutine MCONN in a portion of memory previously occupied by matrix [M]. Instead of representing a 200x5 matrix, this area is now considered as five stacks whose length is 200. Every time MCONN is called by MPTASK to generate an entry

in [C] at location c_{ij}, the j-th entry of the fourth stack in [M] is incremented by one. At the same time, the j-th row of the fifth stack is incremented by one also. The reason for this will be explained shortly.

Returning to the discussion on partitions, the members of a particular partition are those j whose value in stack M_4 is zero. These members are stored in the first stack of M. Several other items of information must be stored in M_1, also. In addition to the element of the partition (1), the following are retained: the partition number (2); if a task is the first member of a partition, the number of elements in the partition (3); if the task has a new task number (i.e., it is a member of a MSC subgraph) designate it as such (4). All four of these items are stored in a single entry in M_1. Item (1) is stored in bits 0-10; item (2) in bits 11-19; item (3) in bits 20-39; item (4) in bit 40. The first few entries of M_1 for the example program are shown in Figure 8. (See the Task Scheduling Table in Appendix II for an interpretation of the meaning of these entries).

It was mentioned in Section II that after all the zero columns are detected during a particular partition the corresponding rows and the columns are eliminated from further consideration. The effect of eliminating a particular row in the scheme described here is accomplished in the following manner. Given a partition member k, the column numbers of the non-zero entries of row k in [C] are noted. These column numbers are the j coordinates for the members of row k which are not zero.

M_1

	59 — 40	39 — 20	19 — 11	10 — 0
1			1	1
2		3	2	2
3			2	3
4			2	13
5	1	3	3	4
6			3	8
7			3	14
8		3	4	7
9	1		4	9
10			4	15

FIGURE 8. State of Stack M_1 after Performance of Column Precedence Partitions.

Elimination of this row is effected by reducing by one the value of each of these j entries in stack M_4. The members of the next partition are those members of M_4 whose value is now zero.

The process is repeated until all columns and corresponding rows have been eliminated. The results of column precedence partitions could now be printed. However, since the results of row precedence partitions are to be printed alongside the results of column precede ce

partitions, printing is deferred until subroutine MRPART upon being called by subroutine MCPP generates the results of row precedence partitions.

Determining the latest time during which a task can be executed (i.e. performing row precedence partitions) is considerably easier than determining the earliest time during which a task can be executed. This is due primarily to the fact that instead of seeking all-zero columns, the goal is now all-zero rows. And because of the way that the [C] matrix is represented, the search for all-zero rows proceeds forty bits at a time. Unlike column partitions which initiated a search with column 1, row partitions initiate a search with the highest numbered row in [C]. After the members of a partition are determined, the corresponding columns in [C] are set to zero. The rows which represent the members of a partition are not considered further. The process is repeated until all rows and columns in [C] have been eliminated. In a manner similar to column partitions, the results of row partitions are stored in two stacks called NRP(1) and NRP(2) in the recognizer program.

At this point all calculations and manipulations to be performed have been completed. The remainder of the recognizer program is devoted to the printing of the actual tables. Subroutine MPTST prints the Task Scheduling Table. MPTST calls subroutine MPTIT which prints the Permissible Task Initiation Time Table. Control is then returned to MPTST which prints a table which contains all the members of MSC subgraphs

and their task numbers. Finally MPTST prints a table which lists the inputs to each of the tasks in the source program. This is where stack M_5 and matrix $[C]^T$ are used. Recall that for a particular j, the inputs to task j are all those i for which $c_{ij} = 1$. Therefore, the non-zero entries in each row vector j of $[C]^T$ are the inputs to task j. The j-th entry in M_5 contains the number of 1's in row vector j of $[C]^T$. The entries in M_5 are used to reduce the search time of each row vector.

IV. OBSERVATIONS

At this point some comments will be made on some of the results listed in the tables of Appendix II. For example, the third column partition of the Task Scheduling Table indicates that tasks 4,8 and 14 can be executed in parallel. Note, however, that tasks 4 and 8 are successors of the IF statement of task number 3. That is, either task 4 or task 8 but not both will follow task 3. Thus either tasks 4 and 14 or 8 and 14 can be initiated during the third time period. Both 4 and 8 are listed in time period three because that is the earliest time during which they could be initiated.

Similarly the last row partition contains tasks 2,12,18,23,42,58, 60, and 67. Of these all but 42,58, and 67 are PRINT statements. Since it is safe to assume that in most cases all the PRINT statements of single program refer to a single output device, it is obvious that a definite ordering exists between these PRINT statements. All that this entry in the table is indicating, however, is that all printing can be deferred until time period twenty-two.

Other results which at first appear to be contradictory appear as results in these tables. Interpretation in the light of the comments made above should resolve most of them.

The dayfile for a particular run of the recognizer program is included as the last page in Appendix IV. From some of the figures listed

therein the following comments can be made. The entire recognizer program resided in core memory approximately 22 seconds. Of this time, thirteen seconds were required to compile the program. CPU time was approximately 2.5 seconds. The balance of the time the program was in a READY but WAITING state while the processor serviced other jobs.

Preliminary timing studies on the performance of the FORTRAN Parallel Task Recognizer indicate that well over half of its analysis time is spent in the detection and elimination of strongly connected subgraphs. That this is so is due primarily to the fact that individual bits are used to represent entries the matrices which represent the various graphs.

A related observation is that matrix computations do not increase linearly as the length of the program increases; instead their number is proportional to the square of the length of the program.[5] This suggests that program analysis overhead can be reduced considerably by minimizing the effects of this part of the analysis. This can, of course, be accomplished by reducing the number of loops that the recognition process must detect. From this it can be concluded that one of the factors which increases the "suitability" of a program for parallel processing is the incidence of a small number of loops.

Earlier it was stated that the recognizer can accept programs whose number of executable statements does not exceed two hundred.

For programs longer than this, however, the entire process could be repeated for each set of 200 instructions. The result would then be multiple sets of tables rather than a single set. To determine the maximum parallelism, analysis could be performed on the elements of each Task Scheduling Table. Recall that an entry in this table consists of a set of tasks (partition) which can be executed in parallel. Partition members from the set of Task Scheduling Tables can be moved from one table to another to increase maximum parallelism.

Appendix V describes a program as consisting of a hierarchy of levels. The main program, for example, is of level 1; subroutines called by the main program are of level 2, etc. The type of analysis described in this report is applicable at any level in the hierarchy. The idea can be carried one step further. Consider a pair of nested loops as shown below. The outer loop is considered by the recognizer as a single task in Level 1. The inner loop, however, is analogous to a subroutine of Level 2. Once control is passed to the inner loop, execution of the outer loop is suspended until control is returned from the inner loop. Viewing the inner loop in this manner permits the loop itself to be analyzed for parallel processability.

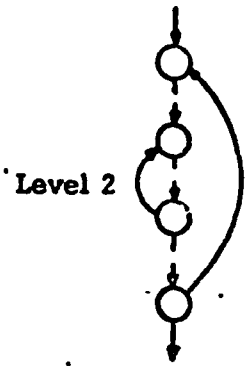


Fig. 9. Hierarchical Representation of Nested Loops.

BIBLIOGRAPHY

[1] Gonzalez, M. J., "Recognition and Representation of Parallel Processable Streams in Computer Programs." Masters Thesis, University of Texas at Austin, August, 1969.

[2] Ramamoorthy, C. V. and Gonzalez, M. J., "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs." Proc. FJCC, 1969.

[3] Ramamoorthy, C. V., "Analysis of Graphs by Connectivity Consideration." JACM, Vol. 13(2), (April, 1966).

[4] Ramamoorthy, C. V., "A Structural Theory of Machine Diagnosis." Proc. SJCC, 1967.

[5] Ramamoorthy, C. V., Personal Communication.

APPENDIX I

THE SOURCE PROGRAM AND THE NUMBERS
ASSIGNED TO THE EXECUTABLE STATEMENTS ARE SHOWN BELOW.

```
      C     THIS IS A TEST PROGRAM DESIGNED TO CHECK OUT
      C     THE ADVANCED VERSION OF THE FORTRAN PARALLEL
      C     TASK RECOGNIZER.

            DIMENSION A(10),B(10),C(10),ANGLE(10),NSTORE(5)
            INTEGER A,B,C,ANGLE
    1       READ 300,(A(I),I=1,10),(B(I),I=1,10),NCODE
    2       PRINT 310,(A(I),I=1,10),(B(I),I=1,10)
    3       IF(NCODE.EQ.0)10,30
    4    10 DO 20 I=1,10
    5       C(I)=SQRT(A(I)**2+B(I)**2)
    6    20 CONTINUE
    7       GO TO 40
    8    30 J=1
    9    35 C(J)=SQRT(A(J)**2-B(J)**2)
   10       J=J+1
   11       IF(J-10)35,35,40
   12    40 PRINT 320,(C(I),I=1,10)
   13       CALL POLAR(A,B,ANGLE)
   14       PRINT 320,(ANGLE(I),I=1,10)
   15       READ 330,NX,NY,NZ
   16       CALL SUB1(C,NX,NY,NZ,NA,NB)
   17       READ 330,NE,NF,NG,NH
   18       PRINT 320,NE,NF,NG,NH
   19       NX1=NE*2**NA
   20       NX2=NF*2**NB
   21       NLH=(NX1*NX)-(NX2/NY)
   22       NRH=(NX-NY+NZ)*(NX1/NX2)
   23       PRINT 320,NX1,NX2,NLH,NRH
   24       CALL SUB2(NLH,NX1,NW1,NW2)
   25       CALL SUB3(NRH,NX2,M1,M2,M3)
   26       ASSIGN 3 TO INDEX
   27    45 GO TO INDEX,(50,60,70)
   28    50 IF((M1-M2).EQ.0)GO TO 70
   29       M4=M3+M2
   30    60 IF(M2-M3)65,110,120
   31    65 M5=(M1+M2)*(M2-M3)/M4
   32    70 CONTINUE
   33       GO TO (80,90,100),M1
   34    80 PRINT 320,M1,M2
   35       GO TO 110
   36    90 PRINT 330,M2,M3
   37       GO TO 110
   38   100 PRINT 330,M4,M5
   39   110 CONTINUE
   40       ASSIGN 2 TO INDEX
   41       GO TO 45
   42   120 CONTINUE
   43       CALL SUB4(M4,M5,K6)
```

```
      DO 140 L=N+1,NM2
      IF(K6,GT,5)150,130
130   STORE(L)=M4+M5
140   CONTINUE
150   L=1
155   IF(L,LE,K6,AND,M4,GT,M5)160,170
160   PRINT 320,M4,M5,L
170   L=L+1
      IF(L,LE,K6)155,180
180   CONTINUE
      READ 340,ABC,DEF,G,H
      X1=ABC+DEF
      X2=DEF-G
      X3=G*H
      X4=X1-X2+X3
      CALL SUB5(X1,X2,X3,X5,X6)
      PRINT 350,X5,X6
185   CALL SUB6(X5,X6,ABC,H,X7)
      X8=X7+X5-X6
      IF(X8-(DEF**2))190,190,200
190   X5=X5-1
      GO TO 185
300   FORMAT(20(I3,1X))
310   FORMAT(1H0,20(I3,1X))
320   FORMAT(1H ,10(I3,1X))
330   FORMAT(20I3)
340   FORMAT(20F4.2)
350   FORMAT(1H ,*OUMMY*,20F4.2)
200   CONTINUE
      END
```

APPENDIX II

TASK SCHEDULING TABLE

COLUMN PARTITIONS		ROW PARTITIONS	
PARTITION NUMBER	TASK NUMBERS	PARTITION NUMBER	TASK NUMBERS
1	1	1	1
2	2 3 13	2	3 13
3	4 8 14	3	8 14
4	7 9 15	4	4 9 15
5	12 16	5	16
6	17	6	17
7	18 19 20	7	19 20
8	21 22	8	22
9	23 24 25	9	25
10	26	10	26
11	27	11	21 27
12	42 43	12	24 43
13	44	13	44
14	48	14	48
15	49	15	49
16	53	16	53
17	54	17	54
18	55 56 57	18	55 56 57
19	58 59	19	59
20	60 61	20	61
21	66	21	7 66
22	67	22	2 12 18 23 42 58 60 67

PERMISSIBLE
TASK INITIATION TIME

TASK	TIME PERIOD
1	1
2	2
3	2
4	3
7	4
8	3
9	4
12	5
13	2
14	3
15	4
16	5
17	6
18	7
19	7
20	7
21	8
22	8
23	9
24	9
25	9
26	10
27	11
42	12
43	12
44	13
48	14
49	15
53	16
54	17
55	18
56	18
57	18
58	19
59	19
60	20
61	20
66	21
67	22

THE NEW TASK NUMBERS AND
THEIR COMPONENTS ARE GIVEN BELOW

NEW TASK NUMBER	TASKS								
4	4	5	6						
9	9	10	11						
27	27	28	29	30	31	32	33	34	
	35	36	37	38	39	40	41		
44	44	45	46	47					
49	49	50	51	52					
61	61	62	63	64	65				

THE INPUTS TO EACH OF THE TASKS ARE GIVEN BELOW

TASK NUMBER	INPUTS			
1				
2	1			
3	1			
13	1			
4	1	3		
8	3			
14	13			
7	4			
9	1	8		
15	14			
12	4	7	9	
16	4	9	15	
17	16			
18	17			
19	16	17		
20	16	17		
21	15	19	20	
22	15	19	20	
23	19	20	21	22
24	19	21		
25	20	22		
26	25			
27	25	26		
42	27			
43	27			
44	24	27	43	
48	44			
49	27	43	48	
53	49			
54	53			
55	54			
56	54			
57	54			
58	55	56	57	
59	55	56	57	
60	59			
61	54	59		
66	61			
67	66			

APPENDIX III

NTADIF

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
1	3	10	30		4	R	0
2	11	35	35	40	9	9	12
3	28	*			29	0	8
4	30	65	110	120	31	39	42
5	45	150	130		48	46	0
6	49	160	170		50	51	0
7	52	155	180		49	53	0
8	63	190	190	200	64	64	66

NTABDO

(1)	(2)	(3)	(4)	(5)	(6)
1	4	20	6		
2	44	190	47	NW1	NW2

NTABGO

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
1	7	1		40	L				
2	27	3	INDEX	50	60	70	1	3	5
3	28	1		70	5				
4	33	3	M1	80	90	100	7	9	-
5	35	1		110	*				
6	37	1		110	*				
7	41	1		45	0				
8	65	1		185	2				

NTSIGN

(1)	(2)	(3)	(4)
1	26	INDEX	
2	40	INDEX	

NLHV

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
1	1	3	A	B	NCOUE		
2	5	1	C				
3	8	1	J				
4	9	1	C				
5	10	1	J				
6	13	1	ANGLE				
7	15	3	NX	NY	NZ		
8	16	2	NA	NB			
9	17	4	NE	NF	NG	NH	
10	19	1	NX1				
11	20	1	NX2				
12	21	1	NLH				
13	22	1	NRH				
14	24	2	NW1	NW2			
15	25	3	M1	M2	M3		
16	29	1	M4				
17	31	1	M5				
18	43	1	K6				
19	46	1	NSTORE				
20	48	1	L				
21	51	1	L				
22	54	4	ABC	DEF	G	H	
23	55	1	X1				
24	56	1	X2				
25	57	1	X3				
26	58	1	X4				
27	59	2	X5	X6			
28	61	1	X7				
29	62	1	X8				
30	64	1	X5				

NWHV

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
1	2	2	A	B			
2	3	1	NCODE				
3	5	6	I	SQRT	A	I	A
4	8	0					
5	9	6	J	SQRT	A	J	A
6	10	1	J				
7	11	1	J				
8	12	1	C				
9	13	2	A	B			
10	14	1	ANGLE				
11	16	4	C	NX	NY	NZ	
12	18	4	NE	NF	NG	NH	
13	19	2	NE	NA			
14	20	2	NF	NB			
15	21	4	NX1	NX	NX2	NY	
16	22	5	NX	NY	NZ	NX1	NX2
17	23	4	NX1	NX2	NLH	NRH	
18	24	2	NLH	NX1			
19	25	2	NRH	NX2			
20	28	2	M1	M2			
21	29	2	M3	M2			
22	30	2	M2	M3			
23	31	5	M1	M2	M2	M3	M4
24	34	2	M1	M2			
25	36	2	M2	M3			
26	38	2	M4	M5			
27	43	2	M4	M5			
28	45	1	K6				
29	46	3	L	M4	M5		
30	48	0					
31	49	4	L	K6	M4	M5	
32	50	3	M4	M5	L		
33	51	1	L				
34	52	2	L	K6			
35	55	2	ABC	UEF			
36	56	2	DEF	G			
37	57	2	G	H			
38	58	3	X1	X2	X3		
39	59	3	X1	X2	X3		
40	60	2	X5	X6			
41	61	4	X5	X6	ABC	H	
42	62	3	X7	X5	X6		
43	63	2	X8	UEF			
44	64	1	X5				

APPENDIX IV

```
      DIMENSION NTABESN(200,3),NES(11),NLHV(200,13),
     1NRHV(200,13),NTABDO(20,6),NTABIF(20,8),NC(200,5),
     2NR(200,5),M(200,5),NIS(10),EX(10),NONEX(11),EXT(11),
     3TEST(11),NX(10),MX(10),NMX(10),NM(8),NTABGO(20,25),
     4NT(4),NTSIGN(10,4),LS(100)
      DATA NBLANK,NSTAR/55B,47B/
      DATA BLANK/5555555555555555B/
C     FORMAT,REAL,COMMON,INTEGER,DIMENSION,DATA
C     TYPE,LOGICAL,EQUIVALENCE,DOUBLE,COMPLEX
      DATA(NONEX(I),I=1,11)/06172215012400000B,2205011400000000B,
     103171515171600000B,  11162405070522000B,  04111505162311178,
     204012401000000000B,24312005000000000B,141707110301140008,
     305212511260114058,04172502140500008,03171520140530008/
C     DO,READ,IF,PRINT,CONTINUE,CALL,END,GO TO,RETURN,ASSIGN
      DATA(EX(I),I=1,10)/04170000000000008,  22050104000000008,
     111060000000000008,  20221116240000008,  03171624111625058,
     203011414000000008,  05160400000000008,
     307175524170000008,22052425221600008,01232311071600008/
C     A,B,C,D,E,F,G,H,I,J,K
      DATA(EXT(I),I=1,11)/01B,02B,03B,04B,05B,06B,07B,10B,
     111B,12B,13B/
      DATA(TEST(I),I=1,11)/77000000000000 0B,  77770000000000008,
     177777700000000008,  77777777000000008,  77777777770000008,
     277777777777700008,  77777777777777008,  77777777777777778,
     31B,                 7777777777778,5555555555558/
      DATA (NM(I),I=1,8)/77B,7777B,777777B,77777777B,7777777777B,
     1777777777777B,77777777777777B,7777777777777777B/
      DATA (NX(I),I=1,8)/770000000000000008,  77000000000000B,
     177000000000B,  7700000000B,  770000000B,  77000008,  7700008,
     27700B,  77B/
      DATA(MX(I),I=1,8)/100000000000000B,  10000000000000B,
     110000000000B,  1000000008,  100000B,  10000B,
     2100B,  1B/
      DATA (NMX(I),I=1,10)/33B,34B,35B,36B,37B,40B,
     141B,42B,43B,44B/
      TYPE INTEGER BLANK,EX,EXT,TEST
      DO 42 J=1,20
      DO 40 I=1,25
      NTABGO(J,I)=0
   40 CONTINUE
   42 CONTINUE
      DO 44 I=1,3
      NT(I)=0
   44 CONTINUE
      DO 48 J=1,10
      DO 46 I=1,4
      NTSIGN(J,I)=0
   46 CONTINUE
   48 CONTINUE
      DO 52 I=1,200
      DO 51 J=1,3
      NTABESN(I,J)=0
   51 CONTINUE
   52 CONTINUE
      DO 56 I=1,200
```

```
      DO 55 J=1,13
      NLHV(I,J)=NRHV(I,J)=0
   55 CONTINUE
   56 CONTINUE
      DO 60 J=1,6
      DO 59 I=1,20
      NTABDO(I,J)=0
   59 CONTINUE
   60 CONTINUE
      DO 62 J=1,8
      DO 61 I=1,20
      NTABIF(I,J)=0
   61 CONTINUE
   62 CONTINUE
      DO 64 I=1,200
      DO 63 J=1,5
      NC(I,J)=M(I,J)=NR(I,J)=0
   63 CONTINUE
   64 CONTINUE
      PRINT 2005
      KESN=NLH=NRH=NTDO=NTIF=NGO=NSIGN=KSN=KMARK=0
C     READ NEXT SOURCE PROGRAM STATEMENT
   70 READ 2000,(NIS(I),I=1,10)
C     CHECK FOR END
      NIST=NIS(2)
      NIST=NIST.AND.TEST(3)
      IF(NIST.EQ.EX(7))290,72
C     CHECK FOR AN ALL-BLANK CARD
   72 DO 73 I=2,10
      IF(NIS(I).EQ.BLANK)73,75
   73 CONTINUE
   74 IF(KMARK.EQ.1)66,67
   66 KMARK=0 $ GO TO 78
   67 KSN=KSN+1
      IF(KSN.GT.50)68,69
   68 KSN=0 $ PRINT 2006
   69 PRINT 2020,(NIS(I),I=1,10)
   78 CONTINUE
      GO TO 70
C     CHECK FOR COMMENT CARD
   75 NIST=NIS(1)
      NIST=NIST.AND.NX(3)
      NIST=NIST/MX(3)
      NIST=NIST.AND.NX(8)
      IF (NIST.EQ.EXT(3))74,76
C     CHECK COLUMNS 1 TO 5
   76 NIST=NIS(1)
      MIST=NIST.AND.TEST(10)
      IF (TEST(11).EQ.MIST)80,77
C     CHECK FOR FORMAT IN COLS. 7-12
   77 NIST=NIS(2)
      MIST=NIST.AND.TEST(6)
      IF(MIST.EQ.NONEX(1))74,80
C     CHECK FOR REAL IN COLS. 7-10
   80 NIST=NIS(2)
```

```
      MIST=NIST.AND.TEST(4)
      IF (MIST.EQ.NONEX(2))74,81
C     CHECK FOR COMMON IN COLS. 7-12
   81 MIST=NIST.AND.TEST(6)
      IF (MIST.EQ.NONEX(3))74,82
C     CHECK FOR INTEGER IN COLS. 7-13
   82 MIST=NIST.AND.TEST(7)
      IF (MIST.EQ.NONEX(4))74,83
C     CHECK FOR DIMENSION IN COLS. 7-15
   83 MIST=NIST.AND.TEST(8)
      IF (MIST.EQ.NONEX(5))74,84
C    CHECK FOR DATA
   84 MIST=NIST.AND.TEST(4)
      IF(MIST.EQ.NONEX(6))74,85
C     CHECK FOR TYPE
   85 IF(MIST.EQ.NONEX(7))74,86
C     CHECK FOR LOGICAL
   86 MIST=NIST.AND.TEST(7)
      IF(MIST.EQ.NONEX(8))74,87
C     CHECK FOR COMPLEX
   87 IF(MIST.EQ.NONEX(11))74,88
C     CHECK FOR EQUIVALENCE
   88 MIST=NIST.AND.TEST(8)
      IF(MIST.EQ.NONEX(9))74,89
C     CHECK FOR DOUBLE
   89 MIST=NIST.AND.TEST(6)
      IF(MIST.EQ.NONEX(10))74,200
C     STATEMENT IS EXECUTABLE
  200 KESN=KESN+1
      NES(1)=KESN
      DO 201 I=1,10
      J=I+1
      NES(J)=NIS(I)
  201 CONTINUE
      KSN=KSN+1 $ KMARK=1
      IF(KSN.GT.50)190,192
  190 KSN=0 $ PRINT 2006
  192 PRINT 2010,(NES(I),I=1,11)
C     CHECK FOR DO
  202 MIST=NIST.AND.TEST(2)
      IF (MIST.EQ.EX(1))220,203
C      CHECK FOR READ
  203 MIST=NIST.AND.TEST(4)
      IF (MIST.EQ.EX(2))230,204
C     CHECK FOR IF
  204 MIST=NIST.AND.TEST(2)
      IF(MIST.EQ.EX(3))240,205
C     CHECK FOR PRINT
  205 MIST=NIST.AND.TEST(5)
      IF (MIST.EQ.EX(4))250,206
C     CHECK FOR CONTINUE
  206 MIST=NIST.AND.TEST(8)
      IF (MIST.EQ.EX(5))260,207
C     CHECK FOR CALL
  207 MIST=NIST.AND.TEST(4)
```

```
      IF (MIST.EQ.FX(6))270,208
C     CHECK FOR GO TO
  208 MIST=NIST.AND.TEST(5)
      IF(MIST.EQ.EX(8))272,209
C     CHECK FOR RETURN
  209 MIST=NIST.AND.TEST(6)
      IF(MIST.EQ.EX(9))274,210
C     CHECK FOR ASSIGN
  210 IF(MIST.EQ.EX(10))276,280
  300 NTABESN(KESN,1)=KESN
      NTABESN(KESN,2)=NEST
      MIST=NSAD
      IF (MIST.EQ.TEST(11))381,305
C     ADDRESS FIELD IS NOT BLANK
  305 NTABESN(KESN,3)=MIST
      GO TO 381
  381 IF(NTABESN(KESN,2).EQ.EXT(3))382,74
  382 IF(NTABIF(NTIF,3).EQ.NSTAR)384,74
C     STATEMENT IS A ONE-BRANCH IF

C      WRT THE SUCCESSOR OF A ONE-BRANCH IF, LIF CONTAINS
C     THE NUMBER OF CHARACTERS BEYOND THE BEGINNING OF A
C     WORD WHICH DO NOT BELONG TO THE SUCCESSOR.  NT(2)=
C     WORD WITHIN STATEMENTS NT(3)=COLUMN CONTAINING
C     INITIAL CHARACTER OF SUCCESSOR.
  384 LIF=NT(3)-(NT(2)-2)*8-7 $ NT(4)=LIF+1
      LT1=NT(2) $ LT2=2**(6*LIF) $ LT3=NT(2)+1
      NISA=(NIS(LT1))*LT2
      NISB=NIS(LT3)
      NISB=NISB.AND.TEST(LIF)
      NISB=NISB/MX(LIF)
      NISB=NISB.AND.NM(LIF)
C     NIST CONTAINS FIRST 8 CHARACTERS OF THE STATEMENT
C     ON THE SAME CARD AS THE ONE-BRANCH IF
      NIST=NISA.OR.NISB
      GO TO 202
  236 I=9
      DO 239 J=1,5
      I=I-1
      IF(I.EQ.8)228,229
  228 NSAD=NIS(1)
      NSAD=NSAD.AND.TEST(7)
      NSAD=NSAD/MX(7)
      NSAD=NSAD.AND.7777777777B
  229 NSD=NSAD
      NSD=NSD.AND.NX(I)
      NSD=NSD/MX(I)
      NSD=NSD.AND.NX(8)
      IF(NSD.EQ.NBLANK)231,239
  239 CONTINUE
  231 IF(I.EQ.8)232,233
  232 NSAD=NIS(1)
      GO TO 300
  233 NSAD=NSAD.AND.NM(8-I)
      GO TO 300
```

```
C       STATEMENT IS A DO
  220 NEST=EXT(1) $ NTDO=NTDO+1
      CALL MDO(NTDO,NIS,NTABDO,KESN)
      GO TO 236
C       STATEMENT IS A READ
  230 NEST=EXT(2)
      CALL MREAD(NLHV,NLH,KESN,NIS,NX,MX,NMX,NT)
      GO TO 236
C       STATEMENT IS AN IF
  240 NEST=EXT(3)
      NTIF=NTIF+1
      CALL MIF(NRHV,NRH,KESN,NIS,NX,MX,NMX,NTIF,
     1NTABIF,NT)
      GO TO 236
C       STATEMENT IS A PRINT
  250 NEST=EXT(4)
      CALL MPRINT(NRHV,NRH,KESN,NIS,NX,MX,NMX,NT)
      GO TO 236
C       STATEMENT IS A CONTINUE
  260 NEST=EXT(5)
      GO TO 236
C       STATEMENT IS A CALL
  270 NEST=EXT(6)
      CALL MCALL(NRHV,NLHV,NRH,NLH,KESN,NX,MX,NMX,NIS,NT)
      GO TO 236
C       STATEMENT IS A GO TO
  272 NEST=EXT(8) $ NGO=NGO+1
      CALL MGOTO (NGO,KESN,NTABGO,NX,MX,NMX,NIS,NT)
      GO TO 236
C       STATEMENT IS A RETURN
  274 NEST=EXT(9)
      GO TO 236
C       STATEMENT IS AN ASSIGN
  276 NEST=EXT(10) $ NSIGN=NSIGN+1
      CALL MASSIGN(NIS,NSIGN,KESN,NX,MX,NTSIGN,NT)
      GO TO 236
C       STATEMENT IS AN AE
  280 NEST=EXT(11)
      CALL MAE(NRHV,NLHV,NRH,NLH,NX,MX,NMX,NIS,KESN,NT)
      GO TO 236
  290 KESN=KESN+1
      NEST=EXT(7)
      NTABESN(KESN,1)=KESN
      NTABESN(KESN,2)=NEST
      NES(1)=KESN
      DO 292 I=1,10
      J=I+1
      NES(J)=NIS(I)
  292 CONTINUE
      PRINT 2010,(NES(I),I=1,11)
      DO 390 J4=1,KESN
      IF(NTABESN(J4,3).EQ.0)390,336
  336 NIST=NTABESN(J4,3)
  340 IF(NTABIF(1,1).EQ.0)371,350
C       SEE IF COLS. 1-5 MATCH SUCCESSORS OF IF STATEMENTS
```

```
  350 DO 365 I=1,NTIF
      IF(NTABIF(I,3).EQ.NSTAR)365,352.
  352 IF(NTABIF(I,1).EQ.0)371,362
  362 DO 360 J=3,5
      IF(MIST.EQ.NTABIF(I,J))363,360
  363 L=J+3
      NTABIF(I,L)=NTABESN(J4,1)
  360 CONTINUE
  365 CONTINUE
C       SEE IF THIS STATEMENT IS A SUCCESSOR OF A GO TO
  371 DO 380 I=1,NGO
      NTGO=NTABGO(I,3)
      IF(NTGO.EQ.0)390,372
  372 DO 378 J=1,NTGO
      NG1=J+4
      IF(MIST.EQ.NTABGO(I,NG1))374,378
  374 NG2=NTABGO(I,3)+NG1
      NTABGO(I,NG2)=NTABESN(J4,1)
      GO TO 380
  378 CONTINUE
  380 CONTINUE
C     CHECK FOR LIMIT OF DO LOOP
      DO 386 I=1,NTDO
      IF(MIST.EQ.NTABDO(I,3))385,386
  385 NTABDO(I,4)=NTABESN(J4,1)
  386 CONTINUE
  390 CONTINUE

 5000 CONTINUE
 5015 PRINT 5020
      DO 5025 I=1,NTIF
      PRINT 5030,(NTABIF(I,J),J=1,8)
 5025 CONTINUE
      PRINT 5040
      DO 5035 I=1,NTDO
      PRINT 5050,(NTABDO(I,J),J=1,6)
 5035 CONTINUE
      PRINT 5060
      DO 5070 I=1,NGO
      PRINT 5065,(NTABGO(I,J),J=1,10)
 5070 CONTINUE
      PRINT 5080
      DO 5090 I=1,NSIGN
      PRINT 5085,(NTSIGN(I,J),J=1,4)
 5090 CONTINUE
      PRINT 5100
      DO 5110 J=1,NLH
      PRINT 5150,(NLHV(J,I),I=1,8)
 5110 CONTINUE
      PRINT 5200
      DO 5300 I=1,NRH
      PRINT 5150,(NRHV(I,J),J=1,8)
 5300 CONTINUE
 2000 FORMAT(R6,9R8)
 2005 FORMAT(1H1,///,20X,*THE SOURCE PROGRAM AND THE*,
```

```
     1* NUMBERS*,/,10X,*ASSIGNED TO THE EXECUTABLE*,
     2* STATEMENTS ARE SHOWN BELOW.*,//)
 2006 FORMAT(1H1,///)
 2010 FORMAT(1H ,8X,I3,1X,R6,9R8)
 2020 FORMAT(1H ,12X,R6,9R8)
 5020 FORMAT(1H1,25X,*NTABIF*,/,10X,*(1)  (2)  (3)*,
     1*   (4)   (5)   (6)   (7)   (8)*)
 5030 FORMAT(1H ,10X,I2,1X,I2,1X,3(R4,1X),3(I4,1X))
 5040 FORMAT(1H0,20X,*NTABDO*,/,10X,*(1)  (2)  (3)  (4)*,
     1*   (5)   (6)*)
 5050 FORMAT(1H ,9X,I2,4X,I2,R4,1X,I4,R5,1X,R5)
 5060 FORMAT(1H0,30X,*NTABGO*,/,10X,*(1)  (2)  (3)   (4)*,
     1*   (5)   (6)   (7)   (8)   (9)   (10)*)
 5065 FORMAT(1H ,10X,2(I2,2X),I2,R7,6(R3,3X))
 5080 FORMAT(1H0,15X,*NTSIGN*,/,10X,*(1)  (2)  (3)  (4)*)
 5085 FORMAT(1H ,9X,2(I3,1X),1X,2(R5,1X))
 5100 FORMAT(1H1,/////,35X,*NLHV*,//,10X,*(1)  (2)  (3)*,
     1*   (4)      (5)      (6)      (7)      (8)*)
 5150 FORMAT(1H ,10X,3(I2,1X),5(R7,1X))
 5200 FORMAT(1H1,/////,35X,*NRHV*,//,10X,*(1)  (2)  (3)*,
     1*   (4)      (5)      (6)      (7)      (8)*)
  500 CONTINUE
C      GENERATE THE PERMISSIBLE TRANSITION MATRIX
       CALL MLOOP(NTABESN,NTABIF,NTABGO,NTABDO,EXT,
     .1NC,NR,KESN,M,NTIF,NGO,NTDO)
C      GENERATE THE REACHABILITY MATRIX
       CALL MREACH(NC,KESN,NR)
C      DETERMINE THE MAXIMAL STRONGLY CONNECTED COMPONENTS
       CALL MSC(NR,KESN,M,LS)
C      ASSIGN NEW TASK NUMBERS TO THE STRONGLY
C      CONNECTED SUBGRAPHS
       CALL MFLAG(NTABESN,LS)
C      ZERO THE PERMISSIBLE TRANSITION MATRIX AND
C      GENERATE THE PARALLEL TASK MATRIX
       DO 590 I=1,200
       DO 580 J=1,5
       NC(I,J)=NR(I,J)=M(I,J)=0
  580 CONTINUE
  590 CONTINUE
       CALL MPTASK(NC,KESN,NTABESN,NRHV,NLHV,
     1NRH,NLH,NTABIF,NTABGO,NTSIGN,NTIF,
     2NGO,NSIGN,EXT,M,NR,NTABDO,NTDO)
C      PERFORM PRECEDENCE PARTITIONS ON THE
C      PARALLEL TASK MATRIX
       CALL MCPP(NC,NTABESN,KESN,NR,M)
       END
```

```
      SUBROUTINE MDO(NTDO,NIS,NTABDO,KESN)
      DIMENSION NIS(10),NTABDO(5,6)
      DATA BLANK,NEQ,ND1,ND2,COMMA/55B,54B,33B,44B,56B/
      TYPE INTEGER BLANK,COMMA
      NTABDO(NTDO,1)=NTDO
      NTABDO(NTDO,2)=KESN
      K3=10 $ K4=K5=0 $ L=1
      DO 300 I=2,10
      IF(I.EQ.2)20,30
   20 K6=4 $ GO TO 40
   30 K6=1
   40 DO 200 J=K6,8
      K=NIS(I)
      K1=(8-J)*6
      K=LSHIFT(K,-K1)
      K=K.AND.77B
      GO TO(50,80,100,120,140,180,186),L
C     CHECK FOR BLANK IN COLUMN K3
   50 IF(K.EQ.BLANK)60,70
   60 K3=K3+1 $ L=1
      GO TO 200
   70 K3=K3+1 $ K4=K4+1 $ L=2
      GO TO 200
   80 IF(K.EQ.BLANK)90,70
C     LOOP LIMIT HAS BEEN READ
   90 CALL NVAR(NIS,I,K4,J,NV)
      NTABDO(NTDO,3)=NV
   95 K3=K3+1 $ K4=0 $ L=3
      GO TO 200
  100 IF(K.EQ.NEQ)110,95
  110 K3=K3+1 $ L=4
      GO TO 200
C     CHECK FOR VARIABLE START FOR LOOP COUNTER
  120 IF(K.GE.ND1.AND.K.LE.ND2)135,130
  130 K5=1
  135 K3=K3+1 $ K4=K4+1 $ L=5
      GO TO 200
  140 IF(K.EQ.COMMA)150,135
  150 IF(K5.EQ.1)160,170
  160 K5=0
      CALL NVAR(NIS,I,K4,J,NV)
      NTABDO(NTDO,5)=NV
  170 K3=K3+1 $ K4=0 $ L=6
      GO TO 200
C     CHECK FOR VARIABLE TERMINATION OF LOOP COUNTER
  180 IF(K.GE.ND1.AND.K.LE.ND2)400,182
  182 K5=1
  184 K3=K3+1 $ K4=K4+1 $ L=7
      GO TO 200
  186 IF(K.EQ.BLANK)188,184
  188 IF(K5.EQ.1)190,400
  190 CALL NVAR(NIS,I,K4,J,NV)
      NTABDO(NTDO,6)=NV
```

```
      GO TO 400
  200 CONTINUE
  300 CONTINUE
  400 RETURN
      END
```

```
      SUBROUTINE MREAD(NLHV,NLH,KESN,NIS,NX,MX,NMX,NT)
      DIMENSION NLHV(200,13),NIS(10),NX(10),MX(10),
     1NMX(10),NT(4)
      DATA COMMA,LP,RP/56B,51B,52B/
      DATA BLANK/55B/
      TYPE INTEGER COMMA,BLANK,RP
      NLH=NLH+1
      NLHV(NLH,1)=NLH $ NLHV(NLH,2)=KESN
   32 IF(NT(1).EQ.1)34,38
   34 K6=NT(4) $ LK=NT(2) $ K3=NT(3)
      GO TO 40
   38 K3=11 $ LK=2
   40 K4=NP=K5=JL=0 $ L=1
      DO 82 I=LK,10
      IF(I.EQ.2)41,42
   41 K6=5 $ GO TO 43
   42 IF(NT(1).EQ.1)43,39
   39 K6=1
   43 DO 80 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      IF(NT(1).EQ.1)8,15
    8 IF(JL.EQ.0)10,15
   10 IF(K.EQ.COMMA)14,12
   12 K3=K3+1 $ GO TO 80
   14 K3=K3+1 $ JL=1 $ GO TO 30
   15 GO TO (44,47,56,62),L
C     CHECK FOR BLANK IN COLUMN K3
   44 IF (K.EQ.BLANK)45,90
   45 K3=K3+1
      IF (K3-72)46,46,99
   46 L=1 $ GO TO 80
C     CHECK FOR LEFT PARENTHESIS
   47 IF(K.EQ.LP)48,49
   49 IF(K.EQ.BLANK)72,70
   48 K5=K5+1
      IF(K4.EQ.0)50,52
```

```
   50 K3=K3+1 $ L=2 $ GO TO 80
   52 CALL NVAR(NIS,I,K4,J,NV)
      LM=K4=0 $ GO TO 73
   54 K3=K3+1 $ L=3 $ GO TO 80
   56 IF(K.EQ.RP)58,54
   58 K5=K5-1
      IF(K5.GT.0)54,60
   60 K3=K3+1 $ L=4 $ GO TO 80
   62 IF(K.EQ.COMMA)64,66
   64 K3=K3+1 $ L=1 $ GO TO 80
   66 IF(K.EQ.BLANK)99,96
C       CHECK FOR END OF VARIABLE
   70 IF(K.EQ.COMMA)71,78
   71 IF(K4.EQ.0)64,72
   72 CALL NVAR(NIS,I,K4,J,NV)
      LM=1 $ K4=0
   73 CONTINUE
      NP=NP+1 $ NP1=NP+3
      NLHV(NLH,3)=NP $ NLHV(NLH,NP1)=NV
      IF(LM.EQ.1)64,54
C     CHECK FORMAT STATEMENT NUMBER IN READ STATEMENT
   90 IF(K4.EQ.0 .AND.K3.LE.16)91,47
   91 DO 92 L=1,10
      IF(K.EQ.NMX(L))93,92
   92 CONTINUE
      GO TO 47
   93 K3=K3+1 $ L=1 $ GO TO 80
   78 K4=K4+1 $ K3=K3+1 $ L=2
   80 CONTINUE
      K6=1
   82 CONTINUE
   96 PRINT 100,K3,K4,K5,K6,KESN
  100 FORMAT(1H ,*NO COMMA FOLLOWING RIGHT PARENTHESIS*,
     1* AFTER PARAMETER NAME*,1H ,3HK3=,I2,2X,3HK4=,I2,2X,
     23HK5=,I2,2X,3HK6=,I2,2X,5HKESN=,I2)
   99 NT(1)=0
      RETURN
      END
```

```
      SUBROUTINE NVAR (NIS,I,K4,J,K)
      DIMENSION NIS(10),N(8),M(8),NM(7)
      DATA (N(L),L=1,8)/77000000000000 B,  77770000000000B,
     177777700000000B,  77777777000000B,  77777777700000B,
     277777777777000B,  77777777777700B,  77777777777777B/
      DATA (M(L),L=1,8)/1B,100B,10000B,1000000B,100000000B,
```

```
     1100000000008,1000000000000B,1000000000000008/
      DATA(NM(L),L=1,7)/77B,7777B,777777B,77777777B,7777777777B,
     1777777777777B,777777777777778/
      IF(J-K4)20,20,10
   10 L1=10-J
      L2=9-L1
      K=NIS(I).AND.N(L2)
      K=K/M(L1)
      K=K.AND.NM(K4)
      GO TO 99
   20 IF(J-1)25,25,30
   25 J1=J
      GO TO 35
   30 J1=J-1
   35 L3=J1-1
      L4=K4-J+1
      L5=6*(J-1)
      L6=8-L3
      L7=9-L6
      K1=NIS(I-1).AND.NM(L4)
      K1=K1*(2**L5)
      IF(J-1)40,40,45
   40 K=K1
      GO TO 99
   45 K2=NIS(I).AND.N(L3+1)
      K2=K2/M(L6)
      K2=K2.AND.NM(L7)
      K=K1.OR.K2
   99 RETURN
      END
```

```
      SUBROUTINE MIF(NRHV,NRH,KESN,NIS,NX,MX,NMX,
     1NTIF,NTABIF,NT)
      DIMENSION NRHV(200,13),MX(10),NMX(10),NT(4),
     1NTABIF(20,8),NIS(10),KK(72),NX(10)
      DATA BLANK,MUL,DIV,ADD,EXP,PER,SUB,LP,RP,COM/
     11R ,47B,50B,45B,4747B,57B,46B,51B,52B,56B/
      DATA NSTAR,NBLANK/47B,55555555555555558/
      TYPE INTEGER BLANK,DIV,ADD,EXP,PER,SUB,COM,RP
      DO 5 I=1,72
      KK(I)=NBLANK
    5 CONTINUE
      NRH=NRH+1
      NRHV(NRH,1)=NRH $ NRHV(NRH,2)=KESN
   30 NTABIF(NTIF,1)=NTIF
```

```
      NTABIF(NTIF,2)=KESN
      NP=K4=KLOG=NT(1)=NT(2)=NT(3)=0
      K3=9 $ L=1 $ K5=K7=KD=0
      DO 160 I=2,10
      IF(I.EQ.2)41,42
   41 K6=3 $ GO TO 43
   42 K6=1
   43 DO 150 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.MX(8)
      KK(K3)=K
      GO TO (44,48,64,80,90,124),L
C     LOOK FOR LEADING LEFT PARENTHESIS
   44 IF(K.EQ.LP)47,45
   45 K3=K3+1 $ L=1 $ GO TO 150
   46 K3=K3+1 $ L=2 $ GO TO 150
   47 K5=K5+1 $ GO TO 46
   48 IF(K.EQ.BLANK)45,50
C     LOOK FOR A DECIMAL DIGIT IN COLUMN K3
   50 DO 52 N=1,10
      IF(K.EQ.NMX(N))53,52
   52 CONTINUE
      GO TO 54
   53 KD=1 $ GO TO 46
C     LOOK FOR A SPECIAL CHARACTER IN COLUMN K3
   54 IF(K.EQ.MUL)46,55
   55 IF(K.EQ.DIV)46,56
   56 IF(K.EQ.ADD)46,57
   57 IF(K.EQ.PER)120,58
   58 IF(K.EQ.SUB)46,59
   59 IF(K.EQ.LP)47,60
   60 IF(K.EQ.RP)110,62
   62 K4=K4+1 $ K3=K3+1 $ L=3
      GO TO 150
C     LOOK FOR SPECIAL CHARACTER WHICH WOULD TERMINATE
C     VARIABLE NAME
   64 IF(K.EQ.MUL)70,65
   65 IF(K.EQ.DIV)70,66
   66 IF(K.EQ.ADD)70,67
   67 IF(K.EQ.SUB)70,140
  140 IF(K.EQ.PER)116,142
  142 IF(K.EQ.COM)70,68
   68 IF(K.EQ.LP) 71,69
   69 IF(K.EQ.RP)115,62
   70 CALL NVAR(NIS,I,K4,J,NV)
      NP=NP+1 $ NP1=NP+3 $ K4=KD=0
      GO TO 72
   71 K5=K5+1
      GO TO 70
   72 CONTINUE
      NRHV(NRH,3)=NP $ NRHV(NRH,NP1)=NV
   75 IF(K.EQ.PER)120,76
   76 IF(K.EQ.RP)77,46
```

```
   77 IF(K5.EQ.0)78,46
   78 K3=K3+1 $ L=4
      GO TO 150
   80 IF(K.EQ.BLANK)78,84
   84 DO 86 N=1,10
      IF(K.EQ.NMX(N))88,86
   86 CONTINUE
      GO TO 130
C     SUCCESSOR OF IF STATEMENT HAS BEEN FOUND
   88 K4=K4+1 $ K3=K3+1 $ L=5
      GO TO 150
   90 DO 92 N=1,10
      IF(K.EQ.NMX(N))88,92
   92 CONTINUE
   94 IF(K.EQ.COM)98,96
   96 IF(K.EQ.BLANK)98,97
   97 PRINT 100,KESN,K3,K4,K6,I,J
      GO TO 170
  100 FORMAT(1H ,*CHARACTER OTHER THAN COMMA FOLLOWING*,
     1* FIRST OR SECOND POSSIBLE SUCCESSOR*,1H ,5HKESN=,I2,2X,
     23HK3=,I2,2X,3HK4=,I2,2X,3HK6=,I2,2X,2HI=,I2,2X,2HJ=,I2)
   98 K7=K7+1 $ K8=K7+2
      CALL NVAR(NIS,I,K4,J,NV)
      K4=0
      NTABIF(NTIF,K8)=NV
      IF(KLOG.EQ.0)102,103
  102 KN=3 $ GO TO 104
  103 KN=2
  104 IF(K7-KN)105,170,170
  105 K3=K3+1 $ L=5
      GO TO 150
  110 K5=K5-1
      IF(K5)78,78,46
  115 K5=K5-1
      GO TO 70
  116 IF(KD.EQ.1.AND.KK(K3-K4-1).EQ.PER)117,70
  117 KD=K4=0 $ GO TO 46
  120 IF(KD.EQ.1)46,122
  122 K3=K3+1 $ L=6 $ KLOG=1 $ GO TO 150
  124 IF(K.EQ.PER)46,122
C     ONE BRANCH LOGICAL IF HAS BEEN DETECTED
  130 NTABIF(NTIF,3)=NSTAR
      NTABIF(NTIF,6)=KESN+1
      NT(1)=1 $ NT(2)=I $ NT(3)=K3
      GO TO 170
  150 CONTINUE
  160 CONTINUE
  170 RETURN
      END
```

```
      SUBROUTINE MPRINT(NRHV,NRH,KESN,NIS,NX,MX,NMX,NT)
      DIMENSION NRHV(200,13),NIS(10),NX(10),MX(10),
     1NMX(10),NT(4)
      DATA BLANK,LP,RP,COM/55B,51B,52B,56B/
      TYPE INTEGER BLANK,RP,COM
      NRH=NRH+1
      NRHV(NRH,1)=NRH $ NRHV(NRH,2)=KESN
   22 IF(NT(1).EQ.1)24,30
   24 K6=NT(4) $ LK=NT(2) $ K3=NT(3)
      GO TO 32
   30 K3=12 $ LK=2
   32 K4=K5=NP=K8=K9=JL=0 $ L=1
      DO 300 I=LK,10
      IF(I.EQ.2)41,42
   41 K6=6
      GO TO 43
   42 IF(NT(1).EQ.1)43,39
   39 K6=1
   43 DO 200 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      IF(NT(1).EQ.1)33,45
   33 IF(JL.EQ.0)36,45
   36 IF(K.EQ.COM)38,37
   37 K3=K3+1 $ GO TO 200
   38 K3=K3+1 $ JL=1 $ GO TO 80
   45 GO TO (44,55,85,100,110,125),L
   44 IF(K.EQ.BLANK)50,55
   50 K3=K3+1 $ L=1
      GO TO 200
C     LOOK FOR PRINT FORMAT STATEMENT NUMBER
   55 DO 60 N=1,10
      IF(K.EQ.NMX(N))70,60
   60 CONTINUE
C     LOOK FOR COMMA AFTER STATEMENT NUMBER
      IF(K.EQ.COM)75,80
   70 K3=K3+1 $ L=2
      GO TO 200
   75 K3=K3+1 $ L=3
      GO TO 200
   80 PRINT 500,K3,K4,K5,KESN
      GO TO 400
  500 FORMAT(1H ,*CHARACTER OTHER THAN COMMA AFTER LAST*,
     1* DIGIT OF FORMAT STATEMENT NUMBER*,1H ,3HK3=,I2,2X,
     23HK4=,I2,2X,3HK5=,I2,2X,5HKESN=,I2)
   85 IF(K.EQ.BLANK)75,87
   87 IF(K.EQ.LP)90,95
   90 K3=K3+1 $ K5=K5+1 $ L=3
```

```
      GO TO 200
   95 K4=K4+1 $ K3=K3+1 $ L=4
      GO TO 200
  100 IF(K.EQ.LP)105,140
  105 K5=K5+1 $ K7=J $ M=I $ K9=1
  106 K3=K3+1 $ L=5
      GO TO 200
  110 IF(K.EQ.RP)115,106
  115 K5=K5-1
      IF(K5.LE.0)120,106
  120 K3=K3+1 $ L=6
      GO TO 200
  125 IF(K.EQ.COM)130,135

  130 CALL NVAR(NIS,M,K4,K7,NV)

      NP=NP+1 $ NP1=NP+3
      NRHV(NRH,3)=NP $ NRHV(NRH,NP1)=NV
  133 IF(K8.EQ.1)400,134
  134 K3=K3+1 $ K4=0 $ L=3
      GO TO 200
  135 IF(K.EQ.BLANK)350,130
  140 IF(K.EQ.COM)145,150
  145 K7=J $ M=I
      GO TO 130
  150 IF(K.EQ.BLANK)155,160
  155 K8=1 $ K7=J $ M=I
      GO TO 130
  160 K3=K3+1 $ K4=K4+1 $ L=4
      GO TO 200
  350 IF(K9.EQ.1)375,400
  375 K8=1 $ K9=0 $ GO TO 130
  200 CONTINUE
      K6=1
  300 CONTINUE
  400 NT(1)=0
      RETURN
      END
```

```
      SUBROUTINE MCALL(NRHV,NLHV,NRH,NLH,KESN,NX,
     1MX,NMX,NIS,NT)
      DIMENSION NRHV(200,13),NLHV(200,13),NX(10),
     1MX(10),NMX(10),NIS(10),NT(4)
      DATA BLANK,COM,LP,RP/55B,56B,51B,52B/
      DATA NBLANK/5555555555555555B/
```

```
      TYPE INTEGER BLANK,COM,RP
      NLH=NLH+1 $ NRH=NRH+1
      NRHV(NRH,1)=NRH $ NLHV(NLH,1)=NLH
      NRHV(NRH,2)=KESN $ NLHV(NLH,2)=KESN
   22 IF(NT(1).EQ.1)24,30
   24 K6=NT(4) $ LK=NT(2) $ K3=NT(3)
      GO TO 32
   30 K3=14 $ LK=2
   32 K4=K5=K8=NPL=NPR=JL=0 $ L=N1=1 $ N2=4
      NMATCH=NLH
      DO 300 I=LK,10
      IF(I.EQ.2)41,42
   41 K6=8
      GO TO 43
   42 IF(NT(1).EQ.1)43,39
   39 K6=1
   43 DO 200 J=K6,8
      K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      IF(NT(1).EQ.1)35,49
   35 IF(JL.EQ.0)46,49
   46 IF(K.EQ.LP)48,47
   47 K3=K3+1 $ GO TO 200
   48 K3=K3+1 $ JL=1 $ GO TO 200
   49 GO TO (44,55,65,80,95),L
   44 IF(K.EQ.LP)50,45
   45 K3=K3+1 $ L=1
      GO TO 200
   50 K5=K5+1 $ K3=K3+1 $ L=2
      GO TO 200
   55 IF(K.EQ.LP)50,60
   60 K4=K4+1 $ K3=K3+1 $ L=3
      GO TO 200
   65 IF(K.EQ.LP)70,150
   70 K7=J $ M=I $ K5=K5+1
   75 K3=K3+1 $ L=4
      GO TO 200
   80 IF(K.EQ.RP)85,75
   85 K5=K5-1
      IF(K5.GT.1)75,90
   90 K3=K3+1 $ L=5
      GO TO 200
   95 IF(K.EQ.COM)100,190
  190 PRINT 500,K3,K4,K5,I,J,KESN
  500 FORMAT(1H ,*NO COMMA FOLLOWING RP WITHIN PARAMETER LIST*,
     11H ,3HK3,I2,2X,3HK4=, I2,2X,3HK5=,I2,2X,
     22HI=,I1,2X,2HJ=, I2,2X,5HKESN=,I2)
      GO TO 400
  100 CALL NVAR(NIS,M,K4,K7,NV)
  101 IF(NLHV(NMATCH,3).EQ.0)105,110
  110 IF(NLHV(NMATCH,N2).EQ.NV)130,103
  103 N1=N1+1 $ N2=N1+3
      IF(N1.GT.NLHV(NMATCH,3))105,110
```

```
  105 NMATCH=NMATCH-1 $ N1=1 $ N2=4
      IF(NMATCH.GT.0)101,140
  123 IF(K8.EQ.1)400,124
  124 K3=K3+1
      L=3 $ NMATCH=NLH $ N1=1 $ N2=4 $ K4=0
      GO TO 200
C     VARIABLE IN PARAMETER LIST IS A RIGHT-HAND VARIABLE.
  130 NPR=NPR+1 $ NP1=NPR+3
      NRHV(NRH,3)=NPR
      NRHV(NRH,NP1)=NV
      GO TO 123
C     VARIABLE IN PARAMETER LIST IS A LEFT-HAND VARIABLE.
  140 NPL=NPL+1 $ NP2=NPL+3
      NLHV(NLH,3)=NPL
      NLHV(NLH,NP2)=NV
      GO TO 123
  150 IF(K.EQ.COM)155,160
  155 K7=J $ M=I
      GO TO 100
  160 IF(K.EQ.RP)170,60
  170 K8=1 $ K7=J $ M=I
      GO TO 100
  200 CONTINUE
      K6=1
  300 CONTINUE
  400 NT(1)=0
      RETURN
      END
```

```
      SUBROUTINE MGOTO(NGO,KESN,NTABGO,NX,MX,NMX,NIS,NT)
      DIMENSION NTABGO(20,25),NX(8),MX(8),NIS(10),NT(4)
      DATA BLANK,LP,COMMA,RP/55B,51B,56B,52B/
      TYPE INTEGER BLANK,COMMA,RP
      NTABGO(NGO,1)=NGO
      NTABGO(NGO,2)=KESN
      IF(NT(1).EQ.1)30,38
   30 K6=NT(4)    $ LK=NT(2) $ K3=NT(3)
      GO TO 40
   38 K3=12 $ LK=7
   40 L=1 $ K4=NP=JL=0
      DO 160 I=LK,10
      IF(I.EQ.2)41,42
   41 K6=6 $ GO TO 43
   42 IF(NT(1).EQ.1)43,39
   39 K6=1
```

```
   43 DO 150 J=K6,8
      IF(NT(1).EQ.1)20,25
   20 JL=JL+1
      IF(JL.GT.5)25,21
   21 K3=K3+1 $ GO TO 150
   25 K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      GO TO (44,50,56,64,70,74,82,86,90,94),L
C     LOOK FOR BLANK IN COLUMN K3
   44 IF(K.EQ.BLANK)45,46
   45 K3=K3+1 $ L=1 $ GO TO 150
C     LOOK FOR LEFT PARENTHESIS
   46 IF(K.EQ.LP)48,80
C     COMPUTED GO TO HAS BEEN DETECTED
   48 K3=K3+1 $ L=2 $ GO TO 150
   50 IF(K.EQ.BLANK)52,54
   52 K3=K3+1 $ L=2 $ GO TO 150
   54 K4=K4+1 $ K3=K3+1 $ L=3 $ GO TO 150
   56 IF(K.EQ.COMMA)60,58
   58 IF(K.EQ.RP)60,54
   60 CALL NVAR(NIS,I,K4,J,NV)
      NP=NP+1 $ NP1=NP+4
      NTABGO(NGO,3)=NP
      NTABGO(NGO,NP1)=NV $ K4=0
C     LOOK FOR TERMINATING RIGHT PARENTHESIS
      IF(K.EQ.RP)62,61
   61 K3=K3+1 $ L=3 $ GO TO 150
   62 K3=K3+1 $ L=4 $ GO TO 150
   64 IF(K.EQ.BLANK)62,66
   66 IF(K.EQ.COMMA)68,72
   68 K3=K3+1 $ L=5 $ GO TO 150
   70 IF(K.EQ.BLANK)68,72
C     RECORD INDEX
   72 K4=K4+1 $ K3=K3+1 $ L=6 $ GO TO 150
   74 IF(K.EQ.BLANK)76,72
   76 CALL NVAR(NIS,I,K4,J,NV)
      NTABGO(NGO,4)=NV
      GO TO 199
   80 K4=K4+1 $ K3=K3+1 $ L=7 $ GO TO 150
   82 IF(K.EQ.COMMA)84,83
   83 IF(K.EQ.BLANK)100,80
C     ASSIGNED TO GO HAS BEEN DETECTED
   84 CALL NVAR(NIS,I,K4,J,NV)
      NTABGO(NGO,4)=NV
      K4=0
   85 K3=K3+1 $ L=8 $ GO TO 150
   86 IF(K.EQ.LP)88,85
   88 K3=K3+1 $ L=9 $ GO TO 150
   90 IF(K.EQ.BLANK)88,92
   92 K4=K4+1 $ K3=K3+1$L=10 $ GO TO 150
   94 IF(K.EQ.COMMA)98,96
   96 IF(K.EQ.RP)98,92
   98 CALL NVAR(NIS,I,K4,J,NV)
```

```
      NP=NP+1 $ NP1=NP+4
      NTABGO(NGO,3)=NP
      NTABGO(NGO,NP1)=NV $ K4=0
      IF(K.EQ.RP)199,102
  102 K3=K3+1 $ L=9 $ GO TO 150
C      UNCONDITIONAL GO TO HAS BEEN DETECTED
  100 CALL NVAR(NIS,I,K4,J,NV)
      NP=NP+1
      NTABGO(NGO,3)=NP
      NTABGO(NGO,5)=NV
      GO TO 199
  150 CONTINUE
      K6=1
  160 CONTINUE
  199 CONTINUE
      NT(1)=0
      RETURN
      END
```

```
      SUBROUTINE MASSIGN(NIS,NSIGN,KESN,NX,MX,NTSIGN,NT)
      DIMENSION NIS(10),NX(8),MX(8),NTSIGN(10,4),NT(4)
      DATA BLANK/55B/
      INTEGER BLANK
      NTSIGN(NSIGN,1)=NSIGN
      NTSIGN(NSIGN,2)=KESN
      KT=0
      IF(NT(1).EQ.1)10,20
   10 K6=NT(4)   $ LK=NT(2) $ K3=NT(3)
      GO TO 22
   20 K3=19 $ LK=3
   22 K4=JL=0 $ L=1
      DO 160 I=LK,4
      IF(NT(1).EQ.1)25,23
   23 IF(KT.EQ.1)24,26
   24 K6=1 $ GO TO 25
   26 K6=5
   25 DO 150 J=K6,8
      IF(NT(1).EQ.1)50,60
   50 JL=JL+1
      IF(JL.GT.12)60,51
   51 K3=K3+1 $ GO TO 150
   60 K=NIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
```

```
      GO TO (30,40),L
   30 IF(K.EQ.BLANK)32,34
   32 K3=K3+1 $ L=1 $ GO TO 150
   34 K4=K4+1 $ K3=K3+1 $ L=2 $ GO TO 150
   40 IF(K.EQ.BLANK)44,34
   44 CALL NVAR(NIS,I,K4,J,NV)
      NTSIGN(NSIGN,3)=NV
      GO TO 200
  150 CONTINUE
      K6=1 $ KT=1
  160 CONTINUE
  200 NT(1)=0
      RETURN
      END
```

```
      SUBROUTINE MAE(NRHV,NLHV,NRH,NLH,NX,MX,
     1NMX,NIS,KESN,NT)
      DIMENSION NRHV(200,13),NLHV(200,13),NX(10),MX(10),
     1NMX(10),NIS(10),KK(72),NT(4)
      DATA BLANK,MUL,DIV,ADD,SUB,PER,COM,LP,RP,NEQ/
     155B,47B,50B,45B,46B,57B,56B,51B,52B,54B/
      DATA NBLANK/55555555555555B/
      TYPE INTEGER BLANK,DIV,ADD,SUB,PER,COM,RP
      NRH=NRH+1 $ NLH=NLH+1
      NRHV(NRH,1)=NRH $ NLHV(NLH,1)=NLH
      NRHV(NRH,2)=KESN $ NLHV(NLH,2)=KESN
      IF(NT(1).EQ.1)502,3
  502 K6=NT(4) $ LK=NT(2) $ K3=NT(3)
      GO TO 5
    3 K3=7 $ LK=2
    5 K4=K5=   K7=NP=0
      DO 4 L=1,72
      KK(L)=NBLANK
    4 CONTINUE
      L=1
      DO 50 N=LK,10
      IF(NT(1).EQ.1)180,181
  181 K6=1
  180 DO 49 J=K6,8
      K=NIS(N)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      GO TO (6,18,35),L
C     LOOK FOR EQUALITY SYMBOL
    6 IF(K.EQ.NEQ)7,10
    7 CALL NVAR(NIS,N,K4,J,NV)    $ K4=0
```

```
      NLHV(NLH,3)=1 $ NLHV(NLH,4)=NV
      GO TO 66
   10 IF(K.EQ.LP)12,11
   11 K4=K4+1 $ K3=K3+1 $ GO TO 49
   12 K5=K5+1
      CALL NVAR(NIS,N,K4,J,NV) $ K4=0
      NLHV(NLH,3)=1 $ NLHV(NLH,4)=NV
   16 K3=K3+1 $ L=2 $ GO TO 49
   18 IF(K6.EQ.1)26,19
   19 DO 20 M=1,10
      IF(K.EQ.NMX(M))16,20
   20 CONTINUE
      IF(K.EQ.MUL)16,22
   22 IF(K.EQ.SUB)16,23
   23 IF(K.EQ.ADD)16,24
   24 IF(K.EQ.DIV)16,26
   26 IF(K.EQ.LP)27,28
   27 K5=K5+1
      IF(K4.EQ.0)16,29
   28 IF(K.EQ.COM)29,37
   29 CALL NVAR(NIS,N,K4,J,NV)
      NP=NP+1 $ NP1=NP+3 $ K6=K4=0
      NRHV(NRH,3)=NP $ NRHV(NRH,NP1)=NV
   32 IF(K5.EQ.0)33,16
   33 K3=K3+1 $ L=3 $ GO TO 49
   35 IF(K.EQ.NEQ)66,33
   37 IF(K.EQ.RP)38,40
   38 K5=K5-1
   39 IF(K4.EQ.0)32,29
   40 IF(K4.NE.0)41,46
   41 IF(K.EQ.MUL)39,42
   42 IF(K.EQ.SUB)39,43
   43 IF(K.EQ.ADD)39,44
   44 IF(K.EQ.DIV)39,46
   46 K6=1 $ K4=K4+1 $ GO TO 16
   49 CONTINUE
      K6=1
   50 CONTINUE
   66 K3=K3+1 $ L=1
      DO 300 I=N,10
      K7=K7+1
      IF(K7.EQ.1)68,69
   68 K6=K3+10-(8*N)
      GO TO 70
   69 K6=1
   70 DO 200 J=K6,8
      K=IIS(I)
      K=K.AND.NX(J)
      K=K/MX(J)
      K=K.AND.NX(8)
      KK(K3)=K
      GO TO (75,115,154),L
C     LOOK FOR DECIMAL DIGIT
   75 DO 80 M=1,10
      IF(K.EQ.NMX(M))85,80
```

```
   80 CONTINUE
      GO TO 90
   85 K3=K3+1 $ L=KD=1 $ GO TO 200
C     LOOK FOR SPECIAL CHARACTER
   90 IF(K.EQ.MUL)85,92
   92 IF(K.EQ.DIV)85,94
   94 IF(K.EQ.ADD)85,96
   96 IF(K.EQ.SUB)85,98
   98 IF(K.EQ.PER)150,100
  100 IF(K.EQ.COM)85,102
  102 IF(K.EQ.LP)85,104
  104 IF(K.EQ.RP)85,106
  106 IF(K.EQ.BLANK)400,110
  110 K4=K4+1 $ K3=K3+1 $ L=2
      GO TO 200
C     LOOK FOR TERMINATING SPECIAL CHARACTER
  115 IF(K.EQ.MUL)130,117
  117 IF(K.EQ.DIV)130,119
  119 IF(K.EQ.ADD)130,121
  121 IF(K.EQ.SUB)130,122
  122 IF(K.EQ.COM)130,124
  123 IF(K.EQ.LP)130,125
  124 IF(K.EQ.PER)156,123
  125 IF(K.EQ.RP)130,127
  127 IF(K.EQ.BLANK)130,110
  130 CALL NVAR(NIS,I,K4,J,NV)
C     A RIGHT-HAND VARIABLE IS RETURNED AND WILL NOW BE STORED
      NP=NP+1 $ NP1=NP+3 $ K4=0
      NRHV(NRH,3)=NP $ NRHV(NRH,NP1)=NV
  144 IF(K.EQ.PER)150,145
  145 IF(K.EQ.BLANK)400,85
  150 IF(KD.EQ.1)85,152
  152 K3=K3+1 $ L=3 $ GO TO 200
  154 IF(K.EQ.PER)85,152
  156 IF(KD.EQ.1.AND.KK(K3-K4-1).EQ.PER)158,130
  158 KD=K4=0 $ GO TO 85
  200 CONTINUE
  300 CONTINUE
  400 NT(1)=0
      RETURN
      END
```

```
      SUBROUTINE MCONN(I,J,NC,NCODE,NR,M)
      DIMENSION NC(200,5),NR(200,5),M(200,5)
      N=1
```

```
    5 IF(J.GT.(N*40))10,20
   10 N=N+1 $ GO TO 5
   20 NW=N
      NB=N*40-J
      NBD=2**NB
      NC(I,NW)=NC(I,NW).OR.NBD
      IF(NCODE.EQ.0)100,40
   40 DO 50 L=1,5
      IF(L*40.GT.I)60,50
   50 CONTINUE
   60 NF=L*40-I
      NH=2**NF
      NR(J,L)=NR(J,L).OR.NH
      M(J,4)=M(J,4)+1
      M(J,5)=M(J,5)+1
  100 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MLOOP(NTABESN,NTABIF,NTABGO,NTABDO,EXT,NC,
     1NR,KESN,M,NTIF,NGO,NTDO)
      DIMENSION NTABESN(200,3),NTABIF(20,8),NTABGO(20,25),
     1NTABDO(5,6),EXT(11),NC(200,5),NR(200,5),M(200,5)
      DATA BLANK/5555555555555555B/
      INTEGER BLANK,EXT
C     SUBROUTINE MLOOP GENERATES THE PERMISSIBLE
C     TRANSITION MATRIX BY SCANNING NTABESN.
      KIF=KGO=KDO=I=NCODE=0
   20 CONTINUE
      I=I+1
   30 IF(I.GE.KESN)500,40
C     CHECK FOR IF
   40 IF(NTABESN(I,2).EQ.EXT(3))20,100
C     CHECK FOR GO TO
  100 IF(NTABESN(I,2).EQ.EXT(8)) 20,200
C     CHECK FOR DO
  200 IF(NTABESN(I,2).EQ.EXT(1)) 20,300
  300 CONTINUE
  310 J=I+1
      CALL MCONN(I,J,NC,NCODE,NR,M)
      GO TO 20
  500 CONTINUE
      DO 520 I=1,NTIF
      KP=6
  505 J=NTABIF(I,KP)
```

```
      IF(J.EQ.0)520,510
  510 K=NTABIF(I,2)
      CALL MCONN(K,J,NC,NCODE,NR,M)
      KP=KP+1
      IF(KP.LT.9)505,520
  520 CONTINUE

      DO 540 I=1,NGO
      KP=5+NTABGO(I,3)
      KT=1 $ K=NTABGO(I,2)
  530 J=NTABGO(I,KP)
      CALL MCONN(K,J,NC,NCODE,NR,M)
      KT=KT+1 $ KP=KP+1
      IF(KT.GT.NTABGO(I,3))540,530
  540 CONTINUE

      DO 560 I=1,NTDO
      K=NTABDO(I,2)
      J=K+1
      CALL MCONN(K,J,NC,NCODE,NR,M)
      J=NTABDO(I,4)
      CALL MCONN(J,K,NC,NCODE,NR,M)
  560 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MREACH(NC,KESN,NR)
      DIMENSION NC(200,5),NR(200,5),NSTACK(200)
      I=K=1 $ NS=N4=0 $ J=KESN
   30 IF(I.GT.5)40,50
   40 I=5 $ GO TO 70
   50 IF((I*40).GT.KESN)70,60
   60 I=I+1 $ GO TO 30
C     N1*40 IS THE SMALLEST MULTIPLE OF 40 WHICH CONTAINS KESN
   70 N1=I $ N2=K
   72 IF(K.GT.KESN)500,75
   75 NZ=NC(K,I)
   80 IF(NZ.EQ.0)300,90
   90 IF((I*40).GT.KESN)110,100
  100 L=0 $ GO TO 120
  110 L=I*40-KESN
  120 IF(J.GT.0)130,310
  130 IF(L.GE.40)300,140
  140 NCOPY=NZ
      IF(L.EQ.0)160,150
```

```
  150 NCOPY=LSHIFT(NCOPY,-L)
  160 NCOPY=NCOPY.AND.1B
      IF(NCOPY.EQ.1)180,170
  170 L=L+1 $ J=J-1
      GO TO 120
  180 N5=I*40-L
      NCOPY=NC(N5,1)
      NCOPY=LSHIFT(NCOPY,-40)
      NCOPY=NCOPY.AND.1B
      IF(NCOPY.EQ.1)170,190
  190 NS=NS+1
      NSTACK(NS)=N5
      GO TO 170
  300 I=I-1
      IF(I.GT.0)175,310
C     END OF ANALYSIS OF ROW K
  310 NCOPY=2**40
      NC(K,1)=NC(K,1).OR.NCOPY
C     NSTACK CONTAINS NODES WHICH NODE K REACHES
  315 N4=N4+1
      IF(N4.GT.NS)400,320
  320 K=NSTACK(N4)
      NCOPY=NC(K,1)
      NCOPY=LSHIFT(NCOPY,-40)
      NCOPY=NCOPY.AND.1B
      IF(NCOPY.EQ.1) GO TO 315
      I=N1 $ J=KESN $ GO TO 75
  400 CONTINUE

C     RESTORE CONNECTION MATRIX TO ORIGINAL CONDITION
      K=1
  445 IF(K.GT.KESN)402,460
  460 NC(K,1)=NC(K,1).AND.177777777777777B
      K=K+1 $ GO TO 445

C     FORM REACH VECTOR FOR NODE N2
  402 CONTINUE
      DO 403 I=1,N1
      NR(N2,I)=NC(N2,I)
  403 CONTINUE
      J=1 $ I=N1
  405 IF(J.GT.NS)440,410
  410 IF(I.GT.0)420,430
  420 N5=NSTACK(J)
      NR(N2,I)=NR(N2,I).OR.NC(N5,I)
      I=I-1 $ GO TO 410
  430 J=J+1 $ I=N1 $ GO TO 405

C     BEGIN CONSTRUCTION OF NEW REACHING VECTOR
  440 N2=N2+1 $ I=N1 $ J=KESN
      NS=N4=0
      K=N2
      LK=K-1
      GO TO 72
  500 CONTINUE
```

```
      RETURN
      END
```

```
      SUBROUTINE MSC(NR,KESN,M,S)
      DIMENSION NR(200,5),M(200,5),LOOP(50),LS(100)
      INTEGER BLANK
      DATA BLANK/5555555555555555B/
      DO 20 I=1,200
      DO 10 J=1,5
      M(I,J)=0
   10 CONTINUE
   20 CONTINUE
C     MATRIX M WILL INITIALLY CONTAIN THE TRANSPOSE OF NR.
      I=K=1 $ J=N=0
   22 IF(K.GT.KESN)200,25
   25 IF(I*40.GT.KESN)150,30
   30 L=I*40
   35 NCOPY=NR(K,I)
      IF(NCOPY.EQ.0)40,34
   34 NCOPY=LSHIFT(NCOPY,-J)
      NCOPY=NCOPY.AND.1B
      IF(NCOPY.EQ.1)36,37
   36 DO 60 LT=1,5
      IF((LT*40).GE.K)62,60
   60 CONTINUE
   62 NEXP=(LT*40)-K
      NCOPY=2**NEXP
      M(L,LT)=M(L,LT).OR.NCOPY
   37 J=J+1 $ L=L-1
      IF(J.LT.40)35,40
   40 J=0 $ I=I+1
      IF(I.LT.5)25,45
   45 I=1 $ J=N=0 $ K=K+1
      GO TO 22
  150 J=I*40-KESN
      L=KESN $ N=N+1
      IF(N.LT.2)35,45
  200 CONTINUE
C     FORM THE PRODUCT OF NR AND NR TRANSPOSE.
      K=1
  202 I=1
      IF(K.GT.KESN)300,205
  205 IF((I*40).GT.(KESN+40))220,215
```

```
  210 K=K+1 $ GO TO 202
  215 M(K,I)=M(K,I).AND.NR(K,I)
      I=I+1 $ GO TO 205
  220 CONTINUE
      GO TO 210
  300 CONTINUE

C     DETECT ALL THE UNIQUE NON-ZERO WORDS.
      DO 310 I=1,50
      LOOP(I)=0
  310 CONTINUE
      I=K=LP=1 $ J=2
  315 IF(K.GT.KESN)500,320
  320 IF((I*40).GT.(KESN+40))330,340
  330 K=K+1 $ I=1 $ J=2
      GO TO 315
  340 IF(M(K,I).EQ.0)350,360
  350 I=I+1 $ GO TO 320
  360 IF(J.GE.(LP+1))370,380
  370 LP=LP+1 $ LOOP(LP)=K
      GO TO 330
  380 IF(LP.EQ.1)370,390
  390 L=LOOP(J) $ I=1
  395 IF((I*40).GT.(KESN+40))330,400
  400 IF(M(K,I).EQ.M(L,I))410,420
  410 I=I+1 $ GO TO 395
  420 J=J+1 $ GO TO 360
  500 CONTINUE
      DO 505 I=1,100
      LS(I)=BLANK
  505 CONTINUE

C     DETERMINE THE STRONGLY COMPONENTS FROM THE NON-ZERO WORDS.
      NQ=2**40 $ LQ=0
      I=K=1 $ J=2
  515 IF(J.GT.LP)700,520
  520 IF((I*40).GT.(KESN+40))530,540
  530 I=1 $ J=J+1 $ K=K+1 $ LQ=0
      GO TO 515
  540 L=39
      NC=LOOP(J)
  545 NCOPY=M(NC,I)
      IF(NCOPY.EQ.0)560,547
  547 NCOPY=LSHIFT(NCOPY,-L)
      NCOPY=NCOPY.AND.1B
      IF(NCOPY.EQ.1)600,550
  550 L=L-1
      IF(L.GE.0)545,560
  560 I=I+1 $ GO TO 520
  600 NCOPY=I*40-L
      K=K+1
      LS(K)=NCOPY
      IF(LQ.EQ.0)610,550
  610 LQ=1
      LS(K-1)=NCOPY.OR.NQ
```

```
      GO TO 550
  700 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MFLAG(NTABESN,LS)
      DIMENSION NTABESN(200,3),LS(100)
      DATA BLANK/5555555555555555B/
      INTEGER BLANK
C     SUBROUTINE MFLAG LABELS THE MAXIMAL STRONGLY
C     CONNECTED STATEMENTS AND ASSIGNS A SINGLE TASK
C     NUMBER TO THE ENTIRE SUBGRAPH.
      DO 40 I=1,200
      NTABESN(I,3)=BLANK
   40 CONTINUE
      DO 100 I=1,100
      IF(LS(I).EQ.BLANK)500,50
   50 L1=LS(I)
      L1=LSHIFT(L1,-40)
      IF(L1.EQ.1)60,70
   60 L2=LS(I).AND.777777B
      GO TO 100
   70 L3=LS(I)
      NCOPY=2**40
      NTABESN(L3,2)=NTABESN(L3,2).OR.NCOPY
      NTABESN(L3,3)=L2
  100 CONTINUE
  500 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MPTASK(NC,KESN,NTABESN,NRHV,NLHV,
     1NRH,NLH,NTABIF,NTABGO,NTSIGN,NTIF,NGO,NSIGN,
     2EXT,M1,NR,NTABDO,NTDO)
      DIMENSION NC(200,5),NTABESN(200,3),NRHV(200,13),
```

```
     1NLHV(200,13),NTABIF(20,8),NTABGO(20,25),NTSIGN(10,4),
     2FXT(11),M1(200,5),NR(200,5),NTABDO(5,6)
      DATA BLANK/5555555555555558/
      INTEGER BLANK,EXT

C     SUBROUTINE MPTASK IS DESIGNED TO GENERATE THE
C     PARALLEL TASK CONNECTION MATRIX AFTER CONSIDERATION
C     IS GIVEN TO THE I/O RELATIONSHIPS BETWEEN TASKS.

C     COMPARE EACH RHV TO EACH OF THE LHV.
      I=M=NCODE=1 $ J=N=4
   20 IF(M.GT.NLH)300,30
   30 NT1=NLHV(M,3)+3
      IF(N.GT.NT1)40,50
   40 M=M+1 $ N=4
      GO TO 20
   50 IF(NLHV(M,2).GE.NRHV(I,2))60,80
   60 I=I+1
      IF(I.LE.NRH)50,70
   70 I=1 $ GO TO 700
   80 IF(NLHV(M,N).EH.NRHV(I,J))110,90
   90 J=J+1
      NT2=NRHV(I,3)+3
      IF(J.LE.NT2)80,100
  100 J=4 $ GO TO 60
  110 NA=NLHV(M,2) $ LT=1
      GO TO 900
  120 NA=NRHV(I,2) $ LT=2
      GO TO 950
  130 N=N+1 $ GO TO 30

C     GENERATE CONNECTIONS BETWEEN IF STATEMENTS
C     AND THEIR SUCCESSORS.
  300 CONTINUE
      I=1 $ L=6
  305 IF(I.GT.NTIF)500,310
  310 J=NTABIF(I,L)
      IF(J.EQ.0)320,330
  320 I=I+1 $ L=6
      GO TO 305
  330 NA=NTABIF(I,2) $ LT=3
      GO TO 900
  340 NA=J $ LT=4
      GO TO 950
  350 L=L+1
      IF(L.LT.9)310,320

C     GENERATE CONNECTIONS BETWEEN GO TO STATEMENTS
C     AND THEIR SUCCESSORS.
  500 CONTINUE
      I=1
  510 L=1
      IF(I.GT.NGO)800,520
  520 NA=NTABGO(I,2) $ LT=5
      GO TO 900
```

```
  530 J=NTABGO(I,3)+5
  540 NA=NTABGO(I,J) $ LT=6
      GO TO 950
  550 J=J+1 $ L=L+1
      IF(L.GT.NTABGO(I,3))560,540
  560 I=I+1 $ GO TO 510

C     CHECK FOR MATCH BETWEEN LHV AND GO TO INDEX.
  700 K=1
  710 IF(K.GT.NGO)755,720
  720 IF(NLHV(M,2).LT.NTABGO(K,2))722,730
  722 IF(NLHV(M,N).EQ.NTABGO(K,4))740,730
  730 K=K+1 $ GO TO 710
  740 NA=NLHV(M,2) $ LT=7
      GO TO 900
  750 NA=NTABGO(K,2) $ LT=8
      GO TO 950
C     CHECK FOR MATCH BETWEEN LHV AND DO LOOP START AND STOP
  755 K=1
  760 IF(K.GT.NTDO)130,770
  770 IF(NLHV(M,2).LT.NTABDO(K,2))775,785
  775 IF(NLHV(M,N).EQ.NTABDO(K,5))790,780
  780 IF(NLHV(M,N).EQ.NTABDO(K,6))790,785
  785 K=K+1 $ GO TO 760
  790 NA=NLHV(M,2) $ LT=11
      GO TO 900
  795 NA=NTABDO(K,2) $ LT=12
      GO TO 950

C     CHECK FOR MATCH BETWEEN ASSIGN AND GO TO INDEX.
  800 CONTINUE
      I=K=1
  810 IF(I.GT.NSIGN)990,820
  820 IF(K.GT.NGO)832,830
  830 IF(NTSIGN(I,2).GT.NTABGO(K,2))850,840
  832 I=I+1 $ K=1
      GO TO 810
  840 IF(NTSIGN(I,3).EQ.NTABGO(K,4))860,850
  850 K=K+1 $ GO TO 820
  860 NA=NTSIGN(I,2) $ LT=9
      GO TO 900
  870 NA=NTABGO(K,2) $ LT=10
      GO TO 950
  900 NB=NTABESN(NA,2)
      NB=LSHIFT(NB,-40)
      IF(NB.EQ.1)910,920
  910 MA=NTABESN(NA,3)
      GO TO 930
  920 MA=NA
  930 GO TO(120,100,340,350,530,550,750,730,870,850,
     1795,785),LT

  950 NB=NTABESN(NA,2)
      NB=LSHIFT(NB,-40)
      IF(NB.EQ.1)960,970
```

```
  960 MB=NTABESN(NA,3)
      GO TO 980
  970 MB=NA
  980 IF(MA.EQ.MB)930,982
  982 DO 983 K3=1,5
      IF(K3*40.GT.MB)984,983
  983 CONTINUE
  984 K1=K3*40-MB $ K2=NC(MA,K3)
      K2=LSHIFT(K2,-K1) $ K2=K2.AND.1B
      IF(K2.EQ.1)GO TO 930
      CALL MCONN(MA,MB,NC,NCODE,NR,M1)
      GO TO 930

  990 CONTINUE
C     CHECK FOR STATEMENTS WHICH ARE STRICTLY
C     OF THE LEFT-HAND TYPE.
      DO 1100 I=2,KESN
      IF(M1(I,4).EQ.0)1010,1100
 1010 NX=NTABESN(I,2).AND.7777B
      IF(NX.EQ.EXT(2))1090,1020
 1020 IF(NX.EQ.EXT(8))1090,1030
 1030 IF(NX.EQ.EXT(7))1090,1040
 1040 IF(NX.EQ.EXT(10))1090,1100
 1090 NA=I-1 $ NB=NTABESN(NA,2)
      NB=LSHIFT(NB,-40) $ NB=NB.AND.1B
      IF(NB.EQ.1)1092,1094
 1092 MA=NTABESN(NA,3) $ GO TO 1095
 1094 MA=NA
 1095 NA=I $ NB=NTABESN(NA,2)
      NB=LSHIFT(NB,-40) $ NB=NB.AND.1B
      IF(NB.EQ.1)1096,1097
 1096 MB=NTABESN(NA,3) $ GO TO 1098
 1097 MB=NA
 1098 IF(MA.EQ.MB)1100,1099
 1099 CALL MCONN(MA,MB,NC,NCODE,NR,M1)
 1100 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MCPP(NC,NTABESN,KESN,NR,M)
      DIMENSION NC(200,5),NTABESN(200,3),NR(200,5),M(200,5)
      DIMENSION NRP(200,2)
C     THIS SUBROUTINE DETERMINES THE PARALLEL
C     PROCESSABLE TASKS.
```

```
C     RECORD THE PARALLEL PROCESSABLE TASKS OF PARTITION I.
      J=I=KS=1 $ K=L1=KM=KR=KT=0
      NQ=2**40
   10 IF(J.GT.KESN)200,20
   20 IF(M(J,3).EQ.0)40,30
   30 J=J+1 $ GO TO 10
   40 IF(M(J,4).EQ.0)50,30
   50 NB=NTABESN(J,2)
      NB=LSHIFT(NB,-40)
      NB=NB.AND.1B
      IF(NB.EQ.1)55,70
   55 IF(J.NE.NTABESN(J,3))60,57
   57 KT=1 $ GO TO 70
   60 M(J,3)=1 $ GO TO 30
   70 KR=1 $ KM=KM+1
      IF(KM.GT.200)80,90
   80 KL=KM-200 $ I=2
   90 KL=KM $ NK=K+1
      M(KL,I)=J
      NK=LSHIFT(NK,10)
      M(KL,I)=M(KL,I).OR.NK
      IF(KT.EQ.1)100,110
C     INDICATE THAT THIS TASK HAS A NEW TASK NUMBER.
  100 M(KL,I)=M(KL,I).OR.NQ $ KT=0
      GO TO 60
C     NOTE EFFECT DUE TO ELIMINATION OF ROW J.
  110 CONTINUE
      GO TO 60
  112 NI=1 $ NF=N+L3
      NROW=M(NF,NI).AND.777B
  115 IF((NI*40).GT.(KESN+40))250,120
  120 IF(NC(NROW,NI).EQ.0)130,140
  130 NI=NI+1 $ GO TO 115
  140 L=39
  150 NX=NC(NROW,NI)
      NX=LSHIFT(NX,-L)
      NX=NX.AND.1B
      IF(NX.EQ.1)160,170
  160 NX=NI*40-L
      M(NX,4)=M(NX,4)-1
  170 L=L-1
      IF(L.GE.0)150,130

  200 CONTINUE
      IF(KR.EQ.0)300,210
  210 KR=0 $ I=1 $ K=K+1
C     K REPRESENTS THE TOTAL NUMBER OF PARTITIONS.
C     L2 REPRESENTS THE NUMBER OF TASKS IN PARTITION I.
C     L3 REPRESENTS THE STARTING POINT OF I IN M( ,1)
C     L2 IS STORED IN THE FIRST ENTRY OF PARTITION
C     I IN M( ,).
      L2=KM-L1 $ L4=L2
      L3=L1+1
      IF(L3.GT.200)220,230
  220 L3=L3-200 $ I=2
```

```
  230 L2=LSHIFT(L2,19)
      M(L3,I)=M(L3,I).OR.L2
      L1=KM $ N=0
  235 IF(KS.GT.L4)240,112
  240 I=J=KS=1 $ GO TO 10
  250 KS=KS+1 $ N=N+1 $ GO TO 235
  300 CONTINUE
      DO 310 I=1,200
      DO 305 J=1,2
      NRP(I,J)=0
  305 CONTINUE
  310 CONTINUE
C     PERFORM ROW PARTITIONS
      CALL MRPART(NC,KESN,NRP,NTABESN,NP,K)
      CALL MPTST(NR,M,K,KESN,NTABESN,NRP,NP)
      RETURN
      END
```

```
      SUBROUTINE MPTST(NR,M,K,KESN,NTABESN,NRP,NP)
      DIMENSION NR(200,5),M(200,5),NTABESN(200,3)
      DIMENSION NSTORE(30),NRP(200,2)
C       SUBROUTINE MPTST LISTS THE RESULTS OF THE
C     PRECEDENCE PARTITIONS OBTAINED BY SUBROUTINE MCPP.
      N=N1=1 $ NY=0 $ NPP=NP
      PRINT 100
      PRINT 105
C     PRINT THE INFORMATION CONTAINED WITHIN PARTITION I.
      DO 500 I=1,K
      DO 20 NST=1,30
      NSTORE(NST)=5555B
   20 CONTINUE
      PRINT 130,I
      NT=M(N,1)
      NT=LSHIFT(NT,-19)
      NT=NT.AND.77B
      NT=NT+N-1
      NJ=N $ NST=1
C     LIST THE ELEMENTS OF PARTITION I.
      DO 400 L=NJ,NT
      NF=N
      IF(N.GT.200)310,320
  310 NF=NF-200 $ N1=2
  320 NX=M(NF,N1).AND.777B
      N=N+1
      NSTORE(NST)=NX $ NST=NST+2
```

```
  400 CONTINUE
      NLIST=NT-NJ+1 $ MLIST=2*NLIST
      IF(NLIST.GT.8)405,410
  405 PRINT 140,(NSTORE(NST),NST=1,16)
      PRINT 141,(NSTORE(NST),NST=17,MLIST)
      GO TO 415
  410 PRINT 140,(NSTORE(NST),NST=1,MLIST)
  415 PRINT 145
      PRINT 135,I
      NZ=NRP(NP,1) $ NST=1
      DO 910 LY=1,NZ
      NSTORE(NST)=NRP(NP,2).AND.777B
      NP=NP+1 $ NST=NST+2
  910 CONTINUE
      NZ1=2*NZ
      IF(NZ.GT.7)915,920
  915 PRINT 147,(NSTORE(NST),NST=1,14)
      PRINT 148
      PRINT 149,(NSTORE(NST),NST=15,NZ1)
      GO TO 925
  920 PRINT 147,(NSTORE(NST),NST=1,NZ1)
  925 PRINT 148
  500 CONTINUE
      PRINT 125

C     GENERATE PERMISSIBLE TASK INITIATION TIME TABLE
      CALL MPTIT(M,NRP,NPP,NTABESN,KESN)

C     LIST THE ELEMENTS OF ANY NEW TASK IN PARTITION I.
      PRINT 150
      PRINT 155
  510 N=1  $ N1=1
      DO 605 I=1,K
      DO 515 NST=1,30,2
      NSTORE(NST)=5555B
  515 CONTINUE
      NT=M(N,1)
      NT=LSHIFT(NT,-19)
      NT=NT.AND.77B
      NT=NT+N-1
      NJ=N $ NST=1
      DO 600 L=NJ,NT
      NF=N
      IF(N.GT.200)520,530
  520 NF=NF-200 $ N1=2
  530 NX=M(NF,N1)
      NX=LSHIFT(NX,-40)
      IF(NX.EQ.1)540,590
  540 NX=M(NF,N1).AND.777B
      PRINT 160,NX

      DO 570 NZ=1,KESN
      IF(NTABESN(NZ,3).EQ.NX)550,560
  550 NY=1 $ NM=NTABESN(NZ,1)
      NSTORE(NST)=NM $ NST=NST+2
```

```
      GO TO 570
  560 IF(NY.EQ.1)565,570
  565 NY=0 $ GO TO 575
  570 CONTINUE
  575 NST=NST-2
      IF(NST.GT.15)576,577
  576 PRINT 170,(NSTORE(NMX),NMX=1,15)
      PRINT 146
      PRINT 175,(NSTORE(NMX),NMX=17,NST)
      GO TO 585
  577 PRINT 170,(NSTORE(NMX),NMX=1,NST)
  585 PRINT 146
  590 N=N+1

  600 CONTINUE
  605 CONTINUE
      PRINT 120

C     LIST THE INPUTS TO EACH OF THE TASKS IN PARTITION I.
      PRINT 190
      N=1 $ N1=1 $ KTIP=0
      DO 608 NST=1,30,2
      NSTORE(NST)=5555B
  608 CONTINUE
      DO 800 I=1,K
      NT=M(N,1)
      NT=LSHIFT(NT,-19)
      NT=NT.AND.77B
      NT=NT+N-1
      NJ=N $ NST=1
      DO 700 L=NJ,NT
      NF=N $ NI=1 $ NS=0
      IF(N.GT.200)610,620
  610 NF=NF-200 $ N1=2
  620 NX=M(NF,N1)
      NX=NX.AND.777B
      NU=M(NX,5)
      PRINT 200,NX
  623 NN=NR(NX,NI)
      NL=39
      NCOPY=NN
      IF(NCOPY.EQ.0)650,625
  625 NCOPY=NN
      NCOPY=LSHIFT(NCOPY,-NL)
      NCOPY=NCOPY.AND.1B
      IF(NCOPY.EQ.1)630,640
  630 NS=NS+1
      NP=NI*40-NL
      NSTORE(NST)=NP $ NST=NST+2
      IF(NS.GE.NU)690,640
  640 NL=NL-1
      IF(NL.GE.0)625,650
  650 NI=NI+1
      IF((NI*40).GT.(KESN+40))690,623
  690 IF(N.EQ.1)694,692
```

```
692 NST=NST-2 $ KTIP=KTIP+1
    IF(KTIP.GT.45)691,693
691 KTIP=0 $ PRINT 210
693 PRINT 144,(NSTORE(NMX),NMX=1,NST)
694 PRINT 146
    N=N+1 $ NST=1
700 CONTINUE
800 CONTINUE
    PRINT 120
100 FORMAT(1H1,////,45X,*TASK SCHEDULING TABLE*,//,6X,
   1*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   2*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   3/,6X,*X*,50X,*X*,50X,*X*,/,6X,*X*,16X,
   4*COLUMN PARTITIONS*,17X,*X*,19X,*ROW PARTITIONS*,
   518X,*X*,/,6X,*X*,50X,*X*,50X,*X*,/,6X,
   6*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   7*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   8/,6X,*X*,11X,*X*,38X,*X*,11X,*X*,38X,*X*,/,6X,*X*,1X,
   9*PARTITION*,1X,*X*,17X,*TASK*,17X,*X*,1X,*PARTITION*)
105 FORMAT(1H+,68X,*X*,17X,*TASK*,17X,*X*,
   A/,6X,*X*,3X,*NUMBER*,
   12X,*X*,15X,*NUMBERS*,16X,*X*,3X,*NUMBER*,2X,
   2*X*,15X,*NUMBERS*,16X,*X*,/,6X,*X*,11X,*X*,
   338X,*X*,11X,*X*,38X,*X*,/,6X,
   4*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   5*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   6/,6X,*X*,11X,*X*,38X,*X*,11X,*X*,38X,*X*)
120 FORMAT(1H ,5X,*X*,14X,*X*,43X,*X*,/,6X,*XXXXXXXXXX*,
   1*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*)
125 FORMAT(1H ,5X,*X*,11X,*X*,38X,*X*,11X,*X*,38X,*X*,/,
   16X,*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   2*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*)
130 FORMAT(1H ,5X,*X*,5X,I2)
135 FORMAT(1H+,61X,I2)
140 FORMAT(1H+,17X,*X*,4X,15(I3,R2))
141 FORMAT(1H ,5X,*X*,16X,15(I3,R2))
144 FORMAT(1H+,25X,15(I3,R2))
145 FORMAT(1H+,56X,*X*)
146 FORMAT(1H+,64X,*X*)
147 FORMAT(1H+,68X,*X*,4X,15(I3,R2))
148 FORMAT(1H+,107X,*X*)
149 FORMAT(1H ,5X,*X*,11X,*X*,38X,*X*,11X,*X*,4X,15(I3,R2))
150 FORMAT(1H1,////,25X,*THE NEW TASK NUMBERS AND*,/,20X,
   1* THEIR COMPONENTS ARE GIVEN BELOW*,//,6X,
   2*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   3*XXXXXXXXXXXXXXXXX*,/,6X,*X*,14X,*X*,43X,*X*)
155 FORMAT(1H ,5X,*X*,3X,*NEW TASK*,3X,*X*,43X,*X*,/,6X,
   1*X*,4X,*NUMBER*,4X,*X*,16X,*TASKS*,22X,*X*,/,6X,*X*,
   214X,*X*,43X,*X*,/,6X,*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
   3*XXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,/,6X,*X*,14X,*X*,
   443X,*X*)
160 FORMAT(1H ,5X,*X*,4X,I3,7X,*X*)
170 FORMAT(1H+,20X,*X*,3X,15(I3,R2))
175 FORMAT(1H ,5X,*X*,14X,*X*,3X,15(I3,R2))
190 FORMAT(1H1,////,12X,*THE INPUTS TO EACH OF THE TASKS*,
```

```
     1* ARE GIVEN BELOW*,//,6X,*XXXXXXXXXXXXXXXXXXX*,
     2*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,/,6X,
     3*X*,14X,*X*,43X,*X*,/,6X,*X*,6X,*TASK*,4X,*X*,
     443X,*X*,/,6X,*X*,5X,*NUMBER*,3X,*X*,15X,*INPUTS*,
     522X,*X*,/,6X,*X*,14X,*X*,43X,*X*,/,6X,*XXXXXXXXXX*,
     6*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,/,
     7SX,*X*,14X,*X*,43X,*X*)
 200 FORMAT(1H ,5X,*X*,6X,I3,5X,*X*)
 210 FORMAT(1H1,5X,*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*,
     1*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX*)
     RETURN
     END
```

```
     SUBROUTINE MRPART(NC,KESN,NRP,NTABESN,NP,K)
     DIMENSION NC(200,5),NRP(200,2),NTABESN(200,3)
C    THIS SUBROUTINE PERFORMS ROW PARTITIONS ON C
     I=KESN $ NQ=2**40
     NI=1 $ NY=NZ=0 $ NP=201
     NV=377777777777B $ NLK=K+1
  10 NLK=NLK-1
  15 IF(I.GT.0)20,200
  20 NX=NC(I,1)
     NX=LSHIFT(NX,-40)
     NX=NX.AND.1B
     IF(NX.EQ.1)30,40
  30 I=I-1 $ GO TO 15
  40 IF((NI*40).GT.(KESN+40))70,50
  50 IF(NC(I,NI).EQ.0)60,55
  55 NI=1 $ GO TO 30
  60 NI=NI+1 $ GO TO 40
  70 NI=1
     NC(I,1)=NC(I,1).OR.NQ
     NX=NTABESN(I,2)
     NX=LSHIFT(NX,-40)
     NX=NX.AND.1B
     IF(NX.EQ.1)80,90
  80 NXL=NTABESN(I,3)
     IF(I.EQ.NXL)90,30
  90 NY=1
     NP=NP-1 $ NZ=NZ+1
C    AN ALL-ZERO ROW HAS BEEN FOUND.
     NRP(NP,2)=I
C    RECORD THE PARTITION NUMBER
     NK=NLK
     NK=LSHIFT(NK,10)
```

```
      NRP(NP,2)=NRP(NP,2).OR.NK
      GO TO 30
  200 IF(NY.EQ.1)210,400
C     ELIMINATE NZ COLUMNS CORRESPONDING TO ALL-ZERO ROWS.
  210 NY=NW=0
C     RECORD NUMBER OF ELEMENTS IN THIS PARTITION
      NRP(NP,1)=NZ
  220 IF(NW.GE.NZ)230,240
  230 I=KESN $ NZ=0
      GO TO 10
  240 I=J=1
      NS=NP+NW
      NU=NRP(NS,2).AND.777B
  250 IF((J*40).LT.NU)260,270
  260 J=J+1 $ GO TO 250
  270 NU=(J*40)-NU
      NT=2**NU
      NO=.NOT.NT
      NN=NO.AND.NV
  280 IF(I.GT.KESN)290,300
  290 NW=NW+1 $ GO TO 220
  300 NC(I,J)=NC(I,J).AND.NN
      I=I+1 $ GO TO 280
  400 CONTINUE
      RETURN
      END
```

```
      SUBROUTINE MPTIT(M,NRP,NP,NTABESN,KESN)
      DIMENSION NTABESN(200,3)
      DIMENSION M(200,5),NRP(200,2),NSTORE(4)
      PRINT 500
      I=1 $ KT=K1=0
      DO 10 J=1,4
      NSTORE(J)=5555B
   10 CONTINUE
   20 IF(I.GT.KESN)200,22
   22 NF=NTABESN(I,2)
      NF=LSHIFT(NF,-40)
      NF=NF.AND.1B
      IF(NF.EQ.1)24,26
   24 NXL=NTABESN(I,3)
      IF(NXL.EQ.I)26,50
   26 K=1
   27 NX=M(K,1) $ NY=NX
      IF(NX.NE.0.AND.K.LT.200)28,50
```

```
 28 NX=NX.AND.777B
    IF(NX.EQ.I)30,29
 29 K=K+1 $ GO TO 27
 30 NT1=NX
    NY=LSHIFT(NY,-10)
    NT2=NY.AND.77B
    NSTORE(1)=NT2 $ J=NP
 40 IF(J.GT.200)50,60
 50 I=I+1 $ GO TO 20
 60 NX=NRP(J,2) $ NY=NX
    NT3=NX.AND.777B
    IF(NT1.EQ.NT3)80,70
 70 J=J+1 $ GO TO 40
 80 NY=LSHIFT(NY,-10)
    NT4=NY.AND.77B
    IF(NT4.EQ.NT2)100,90
 90 NSTORE(3)=NT4 $ K1=1
100 CONTINUE
    KT=KT+1
    IF(KT.GT.45)102,104
102 KT=0
    PRINT 530
104 IF(K1.EQ.0)105,106
105 PRINT 510,NT1,(NSTORE(J),J=1,2)
    PRINT 515 $ GO TO 50
106 PRINT 510,NT1,(NSTORE(J),J=1,4)
    PRINT 515
    K1=0 $ GO TO 50
200 PRINT 520
500 FORMAT(1H1,////,27X,*PERMISSIBLE*,/,22X,
   1*TASK INITIATION TIME*,//,20X,*X*,*XXXXXXXXXXXXXXXXX*,
   2*XXXXXX*,/,20X,*X*,8X,*X*,13X,*X*,/,20X,*X*,2X,*TASK*,
   32X,*X*,1X,*TIME PERIOD*,1X,*X*,/,20X,*X*,
   48X,*X*,13X,*X*,/,20X,*XXXXXXXXXXXXXXXXXXXXXXXX*,/,
   520X,*X*,8X,*X*,13X,*X*)
510 FORMAT(1H ,19X,*X*,3X,I2,3X,*X*,2X,2(I3,R2))
515 FORMAT(1H+,42X,*X*)
520 FORMAT(1H ,19X,*X*,8X,*X*,13X,*X*,/,20X,
   1*XXXXXXXXXXXXXXXXXXXXXXXX*)
530 FORMAT(1H1,////,20X,*XXXXXXXXXXXXXXXXXXXXXX*,
   1/,20X,*X*,8X,*X*,13X,*X*)
    RETURN
    END
```

```
03 APR 70   UNIVERSITY OF TEXAS 6600   UT 1
09.48.12. EPE2381. PPS,,30,100000,20.EEAA01UR.GONZALEZ.
09.48.13. EPE2381. RUN(P)
09.49.03. EPE2381.   CTIME 014.455 SEC.  RUN LEVEL 53V
09.49.03. EPE2381. LGO.
09.49.06. EPE2381.  LOADER UNUSED STORAGE 024034.
09.49.20. EPE2381.  END - PPS
09.49.20. EPE2381. CP   015.609 SEC.
09.49.20. EPE2381. PP   010.021 SEC.
09.49.20. EPE2381. TM   023.498 SEC.     30 (OCTAL)

XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
-5X.1JHBQ
```

APPENDIX V

Reprinted from –
AFIPS – Conference Proceedings, Volume 35

Montvale, New Jersey
07645

A survey of techniques for recognizing parallel processable streams in computer programs*

by C. V. RAMAMOORTHY and M. J. GONZALEZ

The University of Texas
Austin, Texas

INTRODUCTION

State-of-the-art advances—in particular, anticipated advances generated by LSI—have given fresh impetus to research in the area of parallel processing. The motives for parallel processing include the following:

1. Real-time urgency. Parallel processing can increase the speed of computation beyond the limit imposed by technological limitations.
2. Reduction of turnaround time of high priority jobs.
3. Reduction of memory and time requirements for "housekeeping" chores. The simultaneous but properly interlocked operations of reading inputs into memory and error checking and editing can reduce the need for large intermediate storages or costly transfers between members in a storage hierarchy.
4. An increase in simultaneous service to many users. In the field of the computer utility, for example, periods of peak demand are difficult to predict. The availability of spare processors enables an installation to minimize the effects of these peak periods. In addition, in the event of a system failure, faster computational speeds permit service to be provided to more users before the failure occurs.
5. Improved performance in a uniprocessor multiprogrammed environment. Even in a uniprocessor environment, parallel processable segments of high priority jobs can be overlapped so that when one segment is waiting for I/O, the processor can be computing its companion segment. Thus an overall speed up in execution is achieved.

With reference to a single program, the term "parallelism" can be applied at several levels. Parallelism within a program can exist from the level of statements of procedural languages to the level of micro operations. Throughout this paper, discussion will be confined to the more general "task" parallelism. The term "task" (process) generally is intended to mean a self-contained portion of a computation which once initiated can be carried out to its completion without the need for additional inputs. Thus the term can be applied to a single statement or a group of statements.

In contrast to the way the term "level" was used above, task parallelism can exist at several levels within a hierarchy of levels. The statements of the main program of a FORTRAN program, for example, are said to be tasks of the first level. The statements within a subroutine called by the main program would then be second level tasks. If this subroutine itself called another subroutine, then the statements within the latter subroutine would be of the third level, etc. Thus a sequentially organized program can be represented by a hierarchy of levels as shown in Figure 1. Each

*This work was supported by NASA Grant NGR 44-012-144

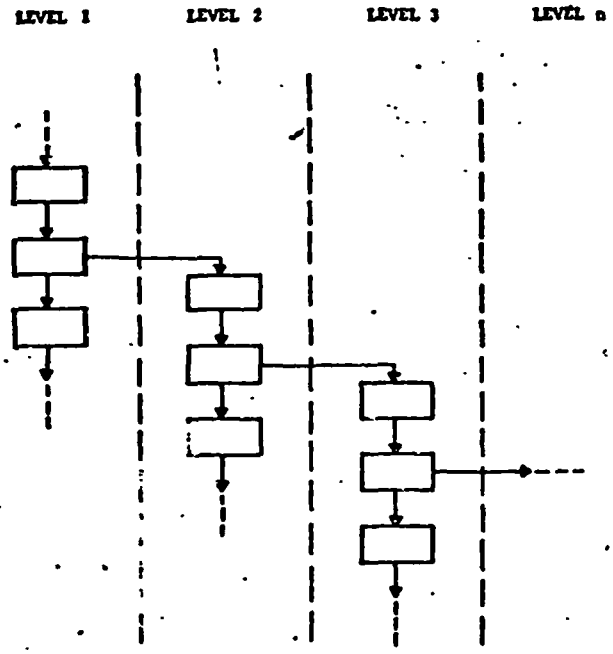


Figure 1—Hierarchical representation of a sequentially organised program

block within a level represents a single task; as before, a task can represent a statement or a group of statements.

Once a sequentially organized program is resolved into its various levels, a fundamental consideration of parallel processing becomes prominent—namely that of recognising tasks within individual levels which can be executed in parallel. Assuming the existence of a system which can process independent tasks in parallel, this problem can be approached from two directions. The first approach provides the programmer with additional tools which enable him to explicitly indicate the parallel processable tasks. If it is decided to make this indication independent of the programmer, then it is necessary to recognize the parallel processable tasks implicitly by analysis of the relationship between tasks within the source program.

After the information is obtained by either of these approaches, it must still be communicated to and utilised by the operating system. At this point, efficient resource utilization becomes the prime consideration.

The conditions which determine whether or not two tasks can be executed in parallel have been investigated by Bernstein.[1] Consider several tasks, T_i, of a sequentially organized program illustrated by a flow chart as shown in Figure 2(a). If the execution of

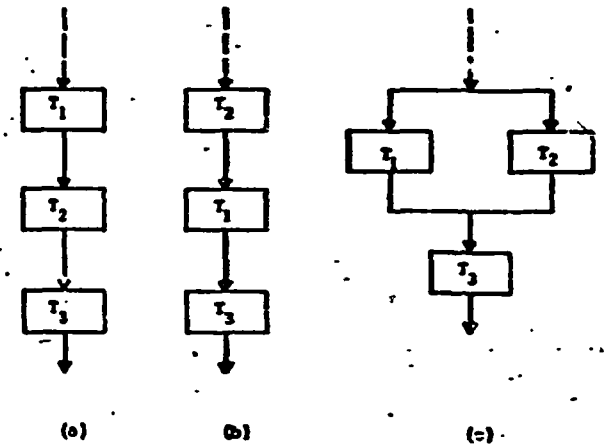


Figure 2—Sequential and parallel execution of a computational process

task T_3 is independent of whether tasks T_1 and T_2 are executed sequentially as shown in Figure 2(a) or 2(b), then parallelism is said to exist between tasks T_1 and T_2. They can, therefore, be executed in parallel as shown in Figure 2(c).

This "commutativity" is a necessary but not sufficient condition for parallel processing. There may exist, for instance, two processes which can be executed in either order but not in parallel. For example, the inverse of a matrix A can be obtained in either of the two ways shown below.

(1)	(2)
a) Obtain transpose of A	a) Obtain matrix of cofactors of A
b) Obtain matrix of cofactors of the transposed matrix	b) Transpose matrix of cofactors
c) Divide result by determinant of A	c) Divide result by determinant of A

Thus obtaining the matrix of cofactors and the transposition operation are two distinct processes which can be executed in alternate order with the same result. They cannot, however, be executed in parallel.

Other complications may arise due to hardware limitations. Two tasks, for example, may need to access the same memory. In this and similar situations, requests for service must be queued. Dijkstra, Knuth, and Coffman[2,3,4] have developed efficient scheduling procedures for using common resources.

In terms of sets representing memory locations, Bernstein has developed the conditions which must be

established before sequentially organized processes can be executed in parallel. These are based on four separate ways in which a sequence of instructions can use a memory location:

(1) The location is only fetched during the execution of T_i.

(2) The location is only stored during the execution of T_i.

(3) The first operation within a task involves a fetch with respect to a location; one of the succeeding operations of T_i stores in this location.

(4) The first operation within a task involves a store with respect to a location; one of the succeeding operations of T_i fetches this location.

Assuming a machine model in which processors are allowed to communicate directly with the memory and multi-access operations are permitted, the conditions for strictly parallel execution of two tasks or program blocks can be stated as follows.

(1) The areas of memory which Task 1 "reads" and onto which Task 2 "writes" should be mutually exclusive, and vice-versa.

(2) With respect to the next task in a sequential process, Tasks 1 and 2 should not store information in a common location.

The conditions listed by Bernstein are sufficient to guarantee commutativity and parallelism of two program blocks. He has shown, however, that there do not exist algorithms for deciding the commutativity or parallelism of arbitrary program blocks.

As an example of what has been discussed here consider the tasks shown below which represent FORTRAN statements for evaluation of three arithmetic expressions.

$$X = (A+B) * (A-B)$$
$$Y = (C-D) / (C+D)$$
$$Z = X+Y$$

Because the execution of the third expression is independent of the order in which the first two expressions are executed, the first two expressions can be executed in parallel.

Parallelism within a task can also exist when individual components of compound tasks can be executed concurrently. In the same manner that individual processors can be assigned to independent tasks, individual functional units can be assigned to independent components within a task. The motivation remains the same—a decrease in execution time of individual tasks. The CDC 6600, for example, can utilize several arithmetic units to perform several operations simultaneously. This type of parallelism can be illustrated by the arithmetic expression which follows.

$$X = (A+B) * (C-D)$$

Normally, this expression would be evaluated in a manner similar to that shown in Figure 3(a). The independent components within the expression, however, permit parallel execution as shown in Figure 3(b) with the same results.

Explicit and implicit parallelism

In the explicit approach to parallelism, the programmer himself indicates the tasks within a computational process which can be executed in parallel. This is normally done by means of additional instructions in the programming language. This approach can be illustrated by the techniques described by Conway, Opler, Gosden, and others[5,6,7]. FORK in the FORK and JOIN technique indicates the parallel processability of a specified set of tasks within a process. The next sequence of tasks will not be initiated until all

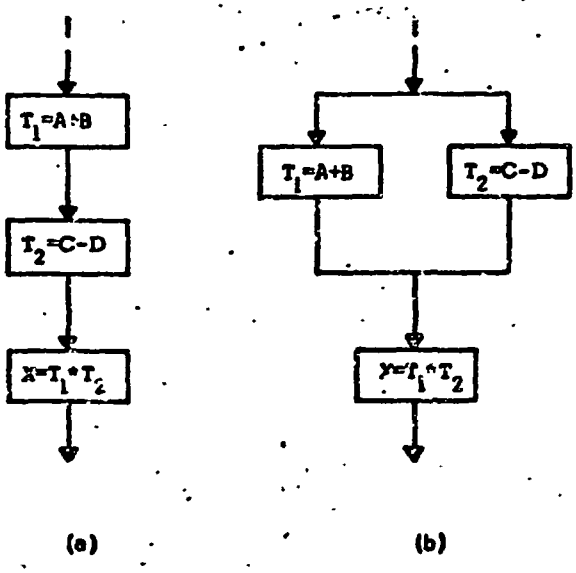


Figure 3—Illustration of parallelism within a compound task

the tasks emanating from a FORK converge to a JOIN statement.

In some instances, some of the parallel operations initiated by the FORK instruction do not have to be completed before processing can continue. For example, one of these branch operations may be designed to alert an I/O unit to the fact that it is to be used momentarily. The conventional FORK must be modified to take care of these situations. Execution of an IDLE statement, for example, permits processors to be released without initiation of further action.[7] The FORK and JOIN TECHNIQUE is illustrated in Figure 4.

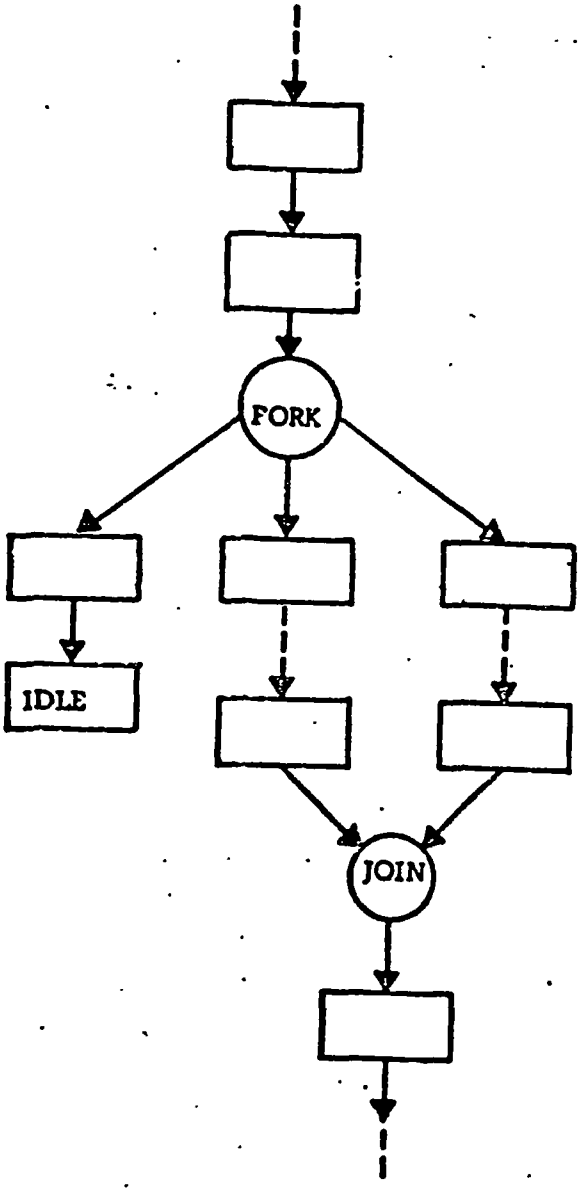


Figure 4—FORK and JOIN technique

Another example of the explicit approach is the PARALLEL FOR[7] which takes advantage of parallel operations generated by the FOR statement in ALGOL and similar constructs in other languages. For example, the sum of two $n \times n$ matrices consists essentially of n^2 independent operations. If n processors were available, the addition process could be organized such that entire rows or columns could be added simultaneously. Thus the addition of the two matrices could be accomplished in n units of time. Another example of this approach is the programming language PL/1 which provides the TASK option with the CALL statement which indicates concurrent execution of parallel tasks.

An additional way of indicating parallelism explicitly is to write a language which exploits the parallelism in algorithms to be implemented by the operating system. This is the case with TRANQUIL,[8,11] an ALGOL-like language to be utilized by the array processors of the ILLIAC IV. The situation is unique in that the language was created after a system was devised to solve an existing problem. "The task of compiling a language for the ILLIAC IV is more difficult than compiling for conventional machines simply because of the different hardware organization and the need to utilize its parallelism efficiently." A limitation of this approach is that programs written in that particular language can only be run on array-type computers and is, therefore, heavily machine dependent.

The implicit approach to parallelism does not depend on the programmer for determination of inherent parallelism but relies instead on indicators existing within the program itself. In contrast to the relative ease of implementation of explicit parallelism, the implicit approach is associated with complex compiling and supervisory programs.

The detection of inherent parallelism between a set of tasks depends on thorough analysis of the source program using Bernstein's conditions. Implementation of a recognition scheme to accomplish this detection is dependent on the source language. Thus a recognizer which is universally applicable cannot be implemented.

An algorithm developed by Fisher[6] approaches the problem of parallel task detection in a general manner. His algorithm utilizes the input and output sets of each task (process) to determine essential ordering and thus inherent parallelism. Given such information as the number of processes to be analyzed, the input and output set for each process, the given permissible

ordering among the processes, and any initially known essential order among the processes, the algorithm generates the essential serial ordering relation and the covering for the essential serial ordering relation. This covering provides an indication of the tasks within the overall process which can be executed concurrently.

Basically, this work formalizes in the form of an algorithm the conditions for parallel processing developed by Bernstein. The conditions for parallel processing between two tasks are extended to an overall process

Detection of task parallelism—A new approach

The next subject covered in this paper involves implicit detection of parallel processable tasks within programs prepared for serial execution. An indication is desired of the tasks which can be executed in parallel and the tasks which must be completed before the start of the next sequence of tasks. Thus the problem can be broken down in two parts—recognizing the relationships between tasks within a level and using this information to indicate the ordering between tasks.

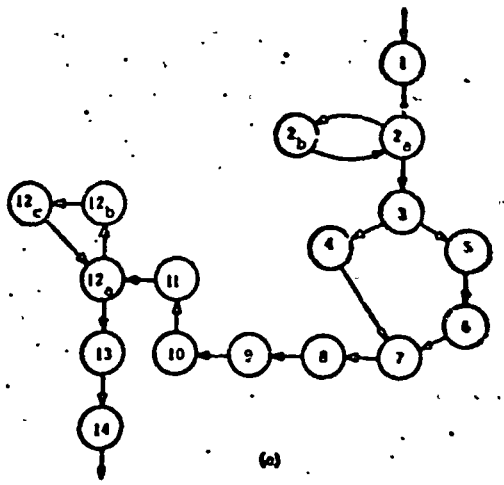
The approach presented here is based on the fact that computational processes can be modeled by oriented graphs in which the vertices (nodes) represent single tasks and the oriented edges (directed branches) represent the permissible transition to the next task in sequence. The graph (and thus the computational process) can be represented in a computer by means of a Connectivity Matrix, C.[10,11] C is of dimension $n \times n$ such that C_{ij} is a "1" if and only if there is a directed edge from node i to node j, and it is "0" otherwise. The properties of the directed graph and hence of the computational process it represents can be studied by simple manipulations of the connectivity matrix.

A graph consisting of a set of vertices is said to be *strongly connected* if and only if any node in it is reachable from any other. A *subgraph* of any graph is defined as consisting of a subset of vertices with all the edges between them retained. A *maximal strongly connected* (M.S.C.) *subgraph* is a strongly connected subgraph that includes all possible nodes which are strongly connected with each other. Given a connectivity matrix of a graph, all its M.S.C. subgraphs can be determined simply by well-known methods.[10] A given program graph can be *reduced* by replacing each of its M.S.C. subgraphs by a single vertex and retaining the edges connected between these vertices and others. After the reduction, the *reduced graph* will not contain any strongly connected components.

The paragraphs which follow will describe the sequence of operations needed to prepare for parallel processing in a multiprocessor computer a program written for a uniprocessor machine.

(1) The first step is to derive the *program graph* which identifies the sequence in which the computational tasks are performed in the sequentially coded program. Figure 5(a) illustrates an example program graph. The program graph is represented in the computer by its connectivity matrix. The connectivity matrix for the example is given in Figure 5(b).

(2) By an analysis of the connectivity matrix, the maximal strongly connected subgraphs are determined by simple operations.[10] This type of subgraph is illustrated by tasks 2 and 12 in Figure 5. Each M.S.C. subgraph is next considered as a single task, and the graph, called the reduced graph, is derived. The reduced graph does not contain any loops or strongly



(a)

	1	2_a	2_b	3	4	5	6	7	8	9	10	11	12_a	12_b	12_c	13	14
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2_a	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2_b	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12_a	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
12_b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12_c	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b)

Figure 5—Program graph of a serially coded program and its connectivity matrix

connected elements. In this graph, when two or more edges emanate from a vertex, a conditional branching is indicated. That is, the execution sequence will take only one of the indicated alternatives. A vertex which initiates the branching operation will be called a *decision* or *branch* vertex. The reduced graph for the example program graph is shown in Figure 6. In this graph, vertex 3 represents a branch vertex.

(3) The next step is to derive the final program graph and its connectivity matrix T. The elements of T are obtained by analyzing the inputs of each vertex in the reduced graph. An element, T_{ij}, is a "1" if and only if the j-th task (vertex) of the reduced graph has as one of its inputs the output of task i; otherwise T_{ij} is a "0". Figure 7 illustrates the final program for the example after consideration is given to the input-output relationships of each task. The connectivity matrix for the final program graph is shown in Figure 8.

From the sufficiency conditions for task parallelism, two tasks can be executed in parallel if the input set of one task does not depend on the output set of the other and vice versa. The technique outlined in Step 4 detects this relationship and uses it to provide an ordering for task execution.

(4) The vertices of the final program graph are

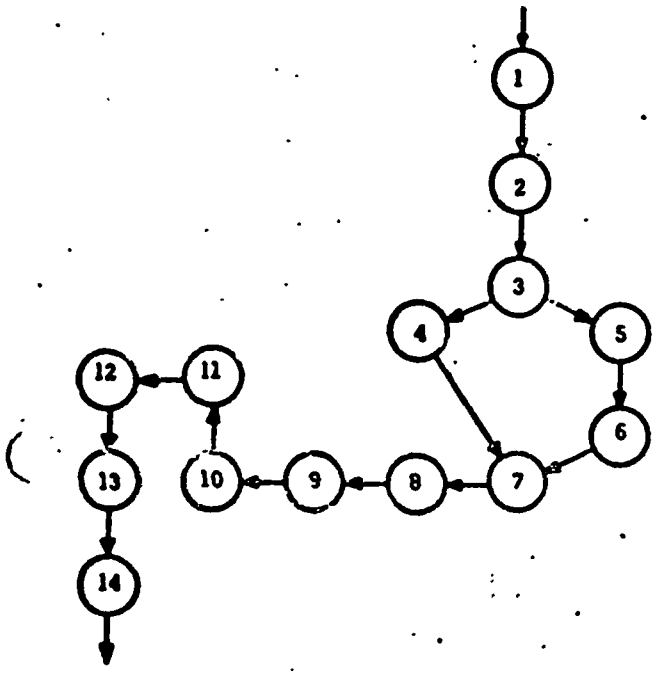


Figure 6—Reduced program graph of the serially coded program

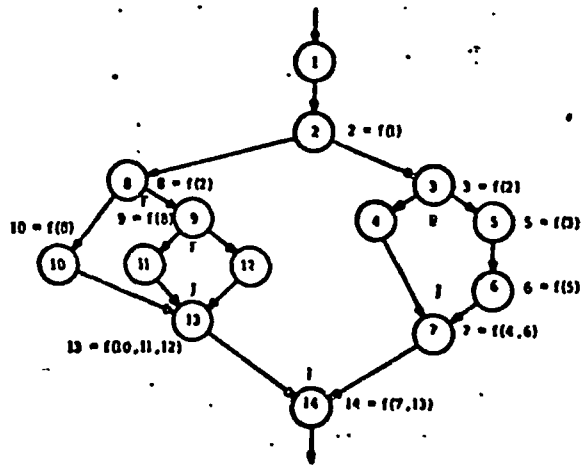


Figure 7—Final program graph of the parallel processable program

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$T =$

Precedence Partitions: {1}, {2}, {3,8}, {4,5,9,10}, {6,11,12}, {7,13}, {14}

Figure 8—Connectivity matrix of the final program graph

partitioned into "precedence partitions"[1] as follows. Using the connectivity matrix T, a column (or columns) containing only zeroes is located. Let this column correspond to vertex v_1. Next delete from T both the column and the row corresponding to this vertex. The first precedence partiton is $P_1 = \{v_1\}$. Using the remaining portion of T, locate vertices $\{v_{21}, v_{22}, \ldots\}$ which correspond to columns containing only zeroes. The second precedence partition P_2 thus contains vertices $\{v_{21}, v_{22}, \ldots\}$. This implies that tasks in set P_2=

$[v_{i1}, v_{i2}, \ldots]$ can be initiated and executed in parallel after the tasks in the previous partition (i.e., P_1) have been completed. Next delete from T the columns and rows corresponding to vertices in P_1. This procedure is repeated to obtain precedence partitions $P_2, P_3, \ldots P_p$, until no more columns or rows remain in the T matrix. It can be shown that this partitioning procedure is valid for connectivity matrices of graphs which contain no strongly connected components.

The implication of this precedence partitioning is that if $P_1, P_2, \ldots P_p$ corresponds to times $t_1, t_2, \ldots t_p$, the earliest time that a task in partition P_i can be initiated is t_i.

The final program graph contains the following types of vertices: (1) The branch or decision type vertex from which the execution sequence selects a task from a set of alternative tasks. (2) The Fork vertex which can initiate a set of parallel tasks. (3) The Join vertex to which a set of parallel tasks converge after their execution. (4) The normal vertex which receives its input set from the outputs of preceding tasks. Figure 7a indicates the final program graph with the first three types of vertices indicated by B, F, and J, respectively.

(5) From precedence partitioning and the final program graph, a Task Scheduling Table can be developed. This table, shown in Table I, serves as an input to the operating system to help in the scheduling of tasks. For example, if the task being executed is a Fork task, a look-ahead feature of the system can prepare for parallel execution of the tasks to be initiated upon completion of the currently active task.

(6) The precedence partitions of Step 4 provide an indication of the *earliest* time at which a task may be initiated. It is also desirable, however, to provide an indication of the *latest* time at which a task may be initiated. This information can be obtained by performing precedence partitions on the transpose of the T matrix. This process can be referred to as "row partitions". The implication here is that if task is in the partition corresponding to time period t_n, then t_n is the latest time that the task i can be initiated.

Using both the row and column partitions, the permissible initiation time for each task can be derived as shown in Table II. Task 4, for example, can be initiated during t_4 or t_5 depending on the availability of processors.

At this point it is desirable to clarify some possible misinterpretations of the implications of this method. The method presented here does not try to determine whether any or all of the iterations within a loop can be executed simultaneously. Rather the iterations executed sequentially are considered as a single task.

TABLE I—Task scheduling table

TIME	INPUTS TO TASKS	TASK NUMBER	TASK TYPE
t_1	-	1	
t_2	1	2	FORK
t_3	2	3	BRANCH
t_3	2	8	FORK
t_4	3	4	
t_4	3	5	
t_4	8	9	FORK
t_4	8	10	
t_5	5	6	
t_5	9	11	
t_5	9	12	
t_6	4,6	7	JOIN
t_6	10,11,12	13	JOIN
t_7	7,13	14	JOIN

For this reason, the undecidability problem introduced by Bernstein is not a factor here.

In addition, precedence partitions may place the successors of a conditional within the same partition. The interpretation of this is that only one of the successors will be executed, and it can be executed in parallel with the other tasks within that partition.

The FORTRAN parallel task recognizer

In order to determine the degree of applicability of the method described above, it was decided to apply the method to a sample FORTRAN program. This was accomplished by writing a program whose input consists of a FORTRAN source program; its output consists of a listing of the tasks within the *first level* of the source program which can be executed in parallel. The program written to accomplish this parallel task

TABLE II—Permissible task initiation time

COLUMN PARTITIONS		PERMISSIBLE TASK INITIATION PERIODS	
TIME	TASK	TASK	TIME
t_1	1	1	t_1
t_2	2	2	t_2
t_3	3,8	3	t_3
t_4	4,5,9,10	4	t_4, t_5
t_5	6,11,12	5	t_4
t_6	7,13	6	t_5
t_7	14	7	t_6
ROW PARTITIONS		8	t_3
t_1	1	9	t_4
t_2	2	10	t_4, t_5
t_3	3,8	11	t_5
t_4	5,9	12	t_5
t_5	4,6,10,11,12	13	t_6
t_6	7,13	14	t_7
t_7	14		

detection is known in its final form as a FORTRAN Parallel Task Recognizer.[11]

The recognizer, also written in FORTRAN, relies on indicators generated by the way in which the program is actually written. Consider the expressions given below.

$$X1 = f_1(A,B)$$
$$X2 = f_2(C,D)$$

Because the right-hand side of the second expression does not contain a parameter generated by the computation which immediately precedes it, the two expressions can be executed in parallel. If, on the other hand, the expressions were rewritten as shown below, the termination of the first computation would have to precede the initiation of the second.

$$X1 = f_1(A,B)$$
$$X2 = f_2(X1,C)$$

The recognizer performs this determination by comparing the parameters on the right-hand of the equality sign to outcomes generated by previous statements.

Other FORTRAN instructions can be analyzed similarly. Consider the arithmetic IF:

IF (X − Y) 3,4,5

Here the parameters within the parentheses must be compared to the outputs of preceding statements in order to determine essential order.

Other FORTRAN instructions are analyzed in a similar manner in order to generate the connectivity matrix for the source program. During this analysis the recognizer assigns numbers to the executable statements of the source program. After this is completed, the recognizer proceeds with the method of precedence partitions described earlier. Precedence partitions yield a list of blocks which contain the statement numbers which can be executed concurrently.

Figure 9 shows a block diagram of the steps taken by the recognizer to generate the parallel processable tasks within the first level of a FORTRAN source program.

Some statements within the FORTRAN set are treated somewhat differently. The DO statement, for example, does not itself contain any input or output parameters but instead generates a series of repeated operations. Because of the loop considerations mentioned earlier, and because the rules of FORTRAN require entrance into a loop only through the DO statement, all the statements contained within a DO loop are considered as a single task. A loop, however, may contain a large number of statements, and a great amount of potential parallelism may be lost if consideration is not given to the statements within the loop. For this reason, the recognizer generates a separate connectivity matrix for each DO loop within the program.

The recognizer itself possesses limitations which must be eliminated before it can be applied to programs of a complex nature. For example, only a subset of the entire FORTRAN set is considered for recogniton. This could be corrected by expanding the recognition process to include a more complete set of instructions.

In addition to the DO statement, loops can also be

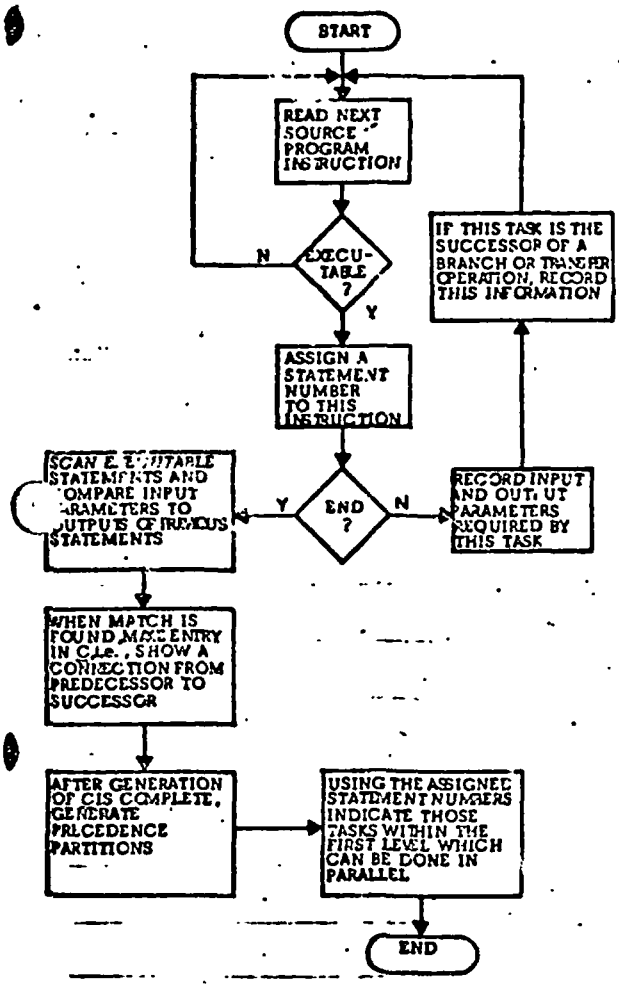


Figure 9—Block diagram of the FORTRAN parallel task recognizer

```
      C     THIS IS A TEST PROGRAM DESIGNED TO CHECK PPS
            DIMENSION A1(10),A2(10),A3(10)
            INTEGER A1,A2,ABC,A2X2,B,C,D
     1      READ 100, (A1(I),I=1,10),B,C,D
     2      READ 100, (A2(I),I=1,10),NS,NST,NSTU
     3      DO 10 I=1,10
     4      IF(A1(I)-A2(I))20,10,40
     5   20 X1=(A1(I))*(B-C)
     6      X2=D*(B/C)
     7      A3(I)=X1*X2
     8   10 CONTINUE
        C   THIS IS A TEST COMMENT
     9   30 PRINT 200,B,C,D
    10   40 CALL ALPHA(A1,A2,ABC,B4,B5)
    11      PRINT 3057,X1,X2,(A3(I),I=1,10)
    12      CALL BETA(X1,X2,A3,B6)
    13      IF(B4-B5)50,50,60
    14   50 READ 315,E,F,G,H
    15      X3=(E*F)*(G-H)
    16      X4=B6*G                                   PARALLEL
    17      X5=X3-X4                                  PROCESSABLE
    18      X6=(B4+B5)*X5                             TASKS
    19   60 PRINT 4,X3,X4,X5                          (1,2)
    20      PRINT 52,(A1(I),I=1,10),ABC,C,(A3(I),I=1,10)   (3)
       100  FORMAT(10I2,3I3)                          (9,10,11,12)
       200  FORMAT(1H0,8 B C D=,/,3I3)                (13)
      3057  FORMAT(1H ,2I3,10F7.1)                    (14)
       315  FORMAT(4F7.4)                             (15,16)
         4  FORMAT(3F7.4)                             (17)
        52  FORMAT(1X,I3,10F7.1)                      (18,19,20)
    21      END
```

(a) (b)

Figure 10—An example of the recognition process.

created by branch and transfer operations such as the IF and GO TO instructions. To eliminate these loops, it would be necessary to analyze the connectivity matrix in the manner mentioned earlier before beginning the process of precedence partitions. The recognizer does not presently perform this analysis.

Nested DO loops are not permitted, and the source program size is limited in the number of executable statements it may have and in the number of parameters any one statement can contain.

Some of these limitations could be eliminated quite easily; others would require a considerable amount of effort. To allow a source program of arbitrary size would require a somewhat more elaborate handling of memory requirements and associated problems. At the present time the recognizer consists of a main program and six subroutines. In its present form the recognizer consists of approximately 1300 statements.

The recognizer is presently written in such a manner that it will detect only first level parallelism. The method it uses, however, can be applied to parallelism at any level.

The theory of operation of the FORTRAN parallel task recognizer will be illustrated by applying the recognition techniques to a sample FORTRAN program. Figure 10(a) is a listing of the sample program showing the individual tasks. Figure 10(b) is a listing of the parallel processable tasks as determined by precedence partitions. The numbers to the left of the executable statements are the numbers assigned by the recognizer during the recognition phase.

Elimination of the limitations mentioned here and other limitations not mentioned explicitly will be the subject of future effort.

Observations and comments

Regardless of the manner in which the subject of parallel processing is approached, common problems arise. Prominent among these is a need to protect common data. If two tasks are considered for concurrent execution and one task accesses a memory location and the other amends it, then strict observance must be paid to the order in which this is done. The

FORTRAN recognizer, for example, may determine that two subroutines can be executed in parallel. At the present time no consideration is given to the fact that both subroutines may access common data through COMMON or EQUIVALENCE statements.

In order to truly optimize execution time for a program which is set up for parallel processing, it would be highly desirable to determine the time required for execution of the individual tasks within the process. It is not enough to merely determine that two tasks can be executed concurrently; the primary goal is that this parallel execution result in higher resource utilization and improved throughput. If the time required for the execution of one task is 100 times that of the other, for example, then it may be desirable to execute the two tasks serially rather than in parallel. The reasoning here is that no time would be spent in allocating processors and so forth.

Determination of task execution time, however, is not a simple matter. Exhaustive measurements of the type suggested by Russell and Estrin[14] would provide the type of information mentioned here.

Another problem area involves implementation of special purpose languages such as TRANQUIL. It was mentioned earlier that programs written in a language of this type are highly machine-limited. It would be highly desirable to be able to implement programs written in these languages in systems which are not designed to take advantage of parallelism. Along these lines, the programming generality suggested by Dennis[15] may be significant.

It should be pointed out that all the techniques which have been discussed here will create a certain amount of overhead. For this reason it is felt that a parallel task recognizer, for example, would be best suited for implementation with production programs. Thus even though some time would be lost initially, in the long run parallel processing would result in a significant net gain.

Conclusions

The method of indicating parallel processable tasks introduced here and illustrated in part by the FORTRAN Parallel Recognizer appears to provide enough generality that it is independent of the language, the application, the mode of compilation, and the number of processors in the system. It is anticipated that this method will remain as the basis for further effort in this area.

In addition to the comments made earlier, some possible future areas of effort include determination of possible parallelism of individual iterations within a loop. It is hoped that additional information can be provided to the operating system other than a mere indication of the tasks which can be executed in parallel. This would include the measurements mentioned earlier and an indication of the frequency of execution of individual tasks.

It is also hoped that a sub-language may be developed which can be added to existing languages to assist in the recognition process and the development of recognizer code.

Detection of parallel components within compound tasks

Several algorithms exist for the detection of independent components within compound tasks.[16,17,18,19] These algorithms are concerned primarily with detection of this type of parallelism within arithmetic expressions. The first three algorithms referenced above are summarized in [19] where a new algorithm is also introduced.

The arithmetic expression which will be used as an example for each algorithm is given below.

$$A + B + C + D * E * F + G + H$$

Throughout this discussion the usual precedence between operators will apply. In order of increasing precedence, the precedence between operators will be as follows: + and −, * and /, and ↑, where ↑ stands for exponentiation.

Hellerman's algorithm

This algorithm assumes that the input string is written in reverse Polish notation and contains only binary operators. The string is scanned from left to right replacing by temporary results each occurrence of adjacent operands immediately followed by an operator. These temporary results will be considered as operands during the next passes. Temporary results generated during a given pass are said to be at the same level and therefore can be executed in parallel. There will be as many passes as there are levels in the syntactic tree. The compilation of the expression listed above is shown in Figure 11.

Although this algorithm is simple and fast, it has two shortcomings. The first is a possible difficulty in implementation since it requires the input string to be in Polish notation; the second is its inability to handle operators which are not commutative.

i	INPUT STRING AFTER THE ith PASS	TEMPORARY RESULTS GENERATED DURING ith PASS
0	A+B+C+D*E*F+G+H+	
1	R1 C+R2 F+G+H+	R1=A+B R2=D*E
2	R3 R4+G+H+	R3=R1+C R4=R2*F
3	R5 G+H+	R5=R3+R4
4	R6 H+	R6=R5+G
5	R7	R7=R6+H

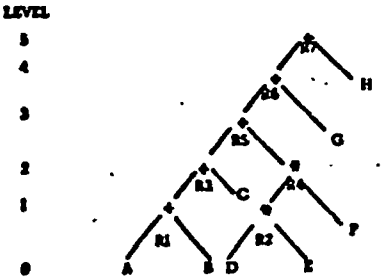


Figure 11—Parallel computation of A+B+C+D*E*F+G+H using Hellerman's algorithm

Stone's algorithm

The basic function of this algorithm is to combine two subtrees of the same level into a level that is one higher. For example, A and B, initially of level 0, are combined to form a subtree of level 1. The algorithm then searches for another subtree of level 1 by attempting to combine C and D. Since precedence relationships between operators prohibit this combination, the level of subtree (A+B) is incremented by one. The algorithm now searches for a subtree of level 2 by attempting to combine C, D, and E. Since this combination is also prohibited, subtree (A+B) is incremented to level 3. The next search is successful, and a subtree of level 3 is obtained by combining C, D, E and F. These two subtrees are then combined to form a single subtree of level 4.

In a similar manner the subtree (G+H), originally of level 1, is successively incremented until it achieves a level of 4; at that time it is combined with the other subtree of the same level to form a final tree of level 5.

The algorithm yields an output string in reverse Polish which does not expressly show which operations can be performed in parallel. Even though the output string is generated in one pass, the recursiveness of the algorithm causes it to be slow, and at least one additional pass would be required to specify parallel computations.

Squire's algorithm

The goal of this algorithm is to form quintuples of temporary results of the form:

Ri (operand 1, operator, operand 2, start level = max [end level op. 1; end level op. 2], end level = start level+1).

All temporary results which have the same start level can be computed in parallel. Initially, all variables have a start and end level equal to zero.

Scanning begins with the rightmost operator of the input string and proceeds from right to left until an operator is found whose priority is lower than that of the previously scanned operator. In the example the scan would yield the following substring:

D*E*F+G+H

Now a left to right scan proceeds until an operator is found whose priority is lower than that of the leftmost operator of the substring. This yields: D*E*F. At this point a temporary result R1 is available of the form:

R1(D,*,E,0,1).

The temporary result, R1, replaces one of the operands and the other is deleted together with its left operator The new substring is then:

R1*F+G+H.

The left to right scans are repeated until no further qunituple can be produced, and at that time, the right to left scan is re-initiated. The results of the process are shown in Figure 12.

Although the example shows the algorithm applied to an expression containing only binary operators, the algorithm can also handle subtraction and division with a corresponding increase in complexity.

A significant feature of this algorithm is that Polish notation plays no part in either the input string or the output quintuples. Because of the many scans and comparisons the algorithm requires, it becomes more complex as the length of the expression and the diversity of operators within the expression increase.

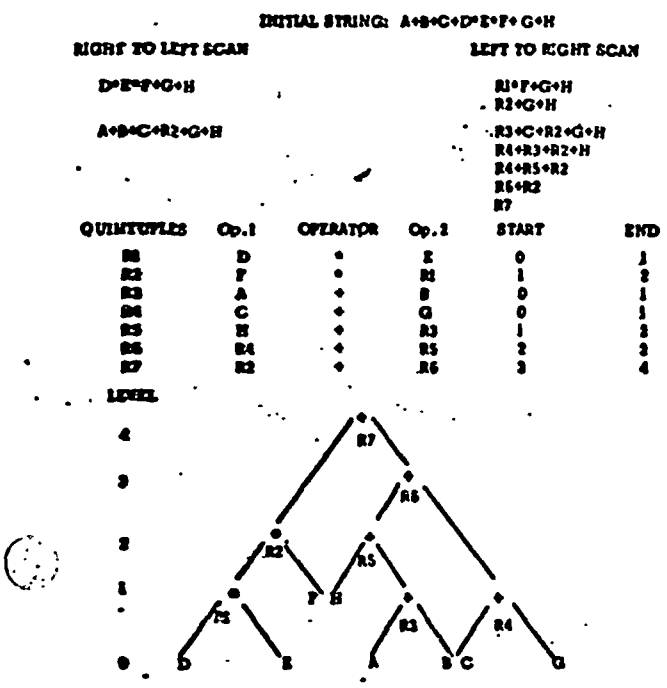


Figure 12—Parallel computation of A+B+C+D*E*F+G+H using Squire's algorithm

Baer and Bovet's algorithm

The algorithm uses multiple passes. To each pass corresponds a level. All temporary results which can be generated at that level are constructed and inserted appropriately in the output string produced by the corresponding pass. Then, this output string becomes the input string for the next level until the whole expression has been compiled. Thus the number of passes will be equal to the number of levels in the syntactic tree. During a pass the scanning proceeds from left to right and each operator and operand is scanned only once.

The simple intermediate language which this algorithm produces is the most appropriate for multiprocessor compilation in that it shows directly all operations which can be performed in parallel, namely those having the same level number. The syntactic tree generated by this algorithm is shown in Figure 13.

A new algorithm

This section will introduce a technique whose goals are: (1) to produce a binary tree which illustrates the parallelism inherent in an arithmetic expression; and

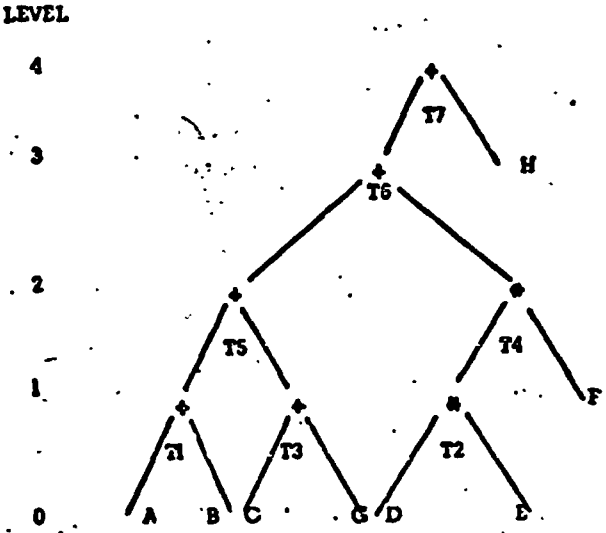


Figure 13—Parallel computation of A+B+C+D*E*F+G+H using Baer and Bovet's algorithm

(2) to determine the number of registers needed to evaluate large arithmetic or Boolean expressions without intermediate transfers to main memory.

This technique is prompted by the fact that existing computing systems possess multiple arithmetic units which can contain a large number of active storages (registers). In addition, the superior memory bandwidths of the next generation of computers will simplify some of the requirements of this technique.

In the material presented below, a complex arithmetic expression is examined to determine its maximum computational parallelism. This is accomplished by repeated rearrangement of the given expression. During this process the given expression in reverse Polish form is also tested for "well formation", i.e., errors and oversights in the syntax, etc.

The arithmetic expression which was used as a model earlier will also be used here, namely A+B+C+D*E*F+G+H. The details of the algorithm follow:

(1) The first step is to rewrite the expression in reverse Polish form and to reverse its order.

+H+G+*F*E D+C+B+A

(2) Starting with the rightmost symbol of the string, assign a weight to each member of the string based on the following procedure:

Assign to symbol S_i the value $V_i = (V_{i-1}) + R_i$ $i = 1,2,\ldots,n$

where $R_i = 1 - \delta(S_i)$ given that

$\delta(S_i) = 0$ if S_i is a variable

$\delta(S_i) = 1$ if S_i is a unary operator

$\delta(S_i) = 2$ if S_i is a binary operator

and $V_{i-1} = V_{i-2} + R_{i-1}$, $V_{i-2} = V_{i-3} + R_{i-2}$, etc.,

such that $V_{i-(i-1)} = V_1 = R_1$, and $V_0 = 0$

Using this procedure, the following expression results:

Root Node

i	15	14	13	12	11	10
S_i	+	H	+	G	+	*
V_i	1	2	1	2	1	2

V_m

9	8	7	6	5	4	3	2	1
F	*	E	D	+	C	+	B	A
3	2	3	2	1	2	1	2	1

Note that for a "well-formed expression" of n symbols $V_n = 1$.

(3) At this point the root node of the proposed binary tree can be determined. Thus the given string can be divided into two independent sub-strings. To determine the root node, draw a line to the left of the first symbol with a weight of 1 ($i = 11$, $S_i = +$, $V_i = 1$) to the left of the symbol with the highest weight, V_m ($i = 7$, $S_i = E$, $V_i = V_m = 3$). The two independent substrings consist of the strings to the left and to the right of this line. The root node will be the leftmost member of the string to the left of the line ($i = 15$, $S_i = +$, $V_i = 1$). Note that V_i also equals 3 for $i = 9$; however V_m is chosen from the earliest occurrence of a symbol with the highest weight.

(4) The next step is to look for parallelism within each of the new substrings. Consider the rightmost substring. Form a new substring consisting of the symbols within the values of $V_i = 1$ to the right and to the left of V_m. Transpose this substring with the substring to the right of it whose leftmost member has a weight of $V_i = 1$.

INITIAL RIGHTMOST SUBSTRING

S_i + * F * E D + C + B A

V_i 1 2 3 2 3 2 1 2 1 2 1

FINAL RIGHTMOST SUBSTRING

i 11 10 9 8 7 6 5 4 3 2 1

S_i + + C + B A * F * E D

V_i 1 2 3 1 3 2 1 2 1 2 1

This procedure is repeated until the initial V_m occupies the position $i = 2$ in the substring. For this example this is already the case. Thus the rightmost substring is in the proper form.

(5) The transposition procedure of step 4 is applied next to the leftmost substring. However, since the leftmost substring of this example consists of only two operands and one operator, no further operations are necessary.

(6) The resultant binary tree is shown in Figure 14. The numbers assigned to each node represent the final weight V_i of the symbol as determined in steps 1–5 above.

Some observations and comments on this algorithm are given below.

(1) The two branches on either side of the root node can be executed in parallel. Within each main branch, the transposition procedure of step 4 yields supplementary root nodes. The sub-branches on each side of the supplementary nodes can be executed in parallel.

(2) The number of levels in the binary tree can be

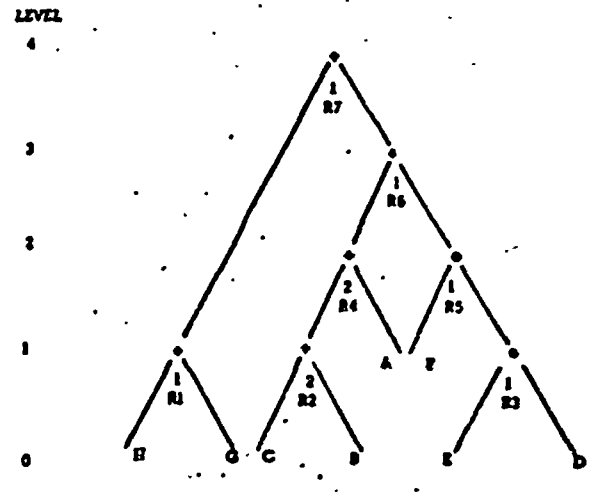


Figure 14—Binary tree for parallel computation of A+B+C+D*E*F+G+H

predicted from the Polish form of the original string.

No. of LEVELS = MAX [NUMBER OF 1's; Vm] in the substring (rightmost or leftmost) containing Vm.

(3) The tree is traversed in a modified postorder form.[20] The resulting expression is

D*E*F+A+B+C+G+H

(4) An added feature of this technique is that the number of registers required to evaluate this expression without intermediate STORE and FETCH operations is obtained directly from the binary tree. This information is provided by the highest weight assigned to any node within the tree. Thus for this example the expression could be evaluated using at most two registers without resorting to intermediate stores and fetches.

(5) This technique of recognizing parallelism on a local level has been applied to a single instruction, in particular, an arithmetic expression. It is worthwhile mentioning that each variable within the expression can itself be the result of a processable task. Thus this technique can be extended to a higher level of parallel stream recognition, i.e., level parallelism.

In order to implement the techniques mentioned here for components within tasks and the techniques mentioned earlier for individual tasks, several system features are desirable. Schemes for detecting parallel processable components within compound tasks are oriented primarily toward arithmetic expressions. For these situations string manipulation ability would be highly desirable. Since individual tasks are represented by a graph and its matrix, the ability to manipulate rows and columns easily would be very important. In this same area, an associative memory could greatly reduce execution time in the implementation of precedence partitions.

ACKNOWLEDGMENTS

The authors would like to thank the referees of the FJCC for their comments and suggestions which resulted in improvements of this paper.

REFERENCES

1 A J BERNSTEIN
Analysis of programs for parallel processing
IEEE Trans on EC Vol 15 No 5 757-763 Oct 1966

2 E W DIJKSTRA
Solution of a problem in concurrent programming control
Comm ACM Vol 8 No 9 569 Sept 1965

3 D KNUTH
Additional comments on a problem in concurrent programming control
Comm ACM Vol 9 No 5 321-322 May 1966

4 E G COFFMAN R R MUNTZ
Models of pure time sharing discipline for research allocation
Proc 1969 Natl ACM Conf

5 M E CONWAY
A multiprocessor system design
Proc FJCC Vol 23 139-146 1963

6 A OPLER
Procedure-oriented statements to facilitate parallel processing
Comm ACM Vol 8 No 5 306-307 May 1965

7 J A GOSDEN
Explicit parallel processing description and control in programs for multi- and uni-processor computers
Proc FJCC Vol 29 651-660 1966

8 N E ABEL P P BUDNIK D J KUCK
Y MURAOKA R S NORTHCOTE
R B WILHELMSON
TRANQUIL: A language for an array processing computer
Proc SJCC 57-68 1939

9 D A FISHER
Program analysis for multiprocessing
Burroughs Corp May 1967

10 C V RAMAMOORTHY
Analysis of graphs by connectivity considerations
Journal ACM Vol 13 No 2 211-222 April 1966

11 C V RAMAMOORTHY M J GONZALEZ
Recognition and representation of parallel processable streams in computer programs—II (task/process parallelism)
1969 Natl ACM Conf

12 C V RAMAMOORTHY
A structural theory of machine diagnosis
Proc SJCC 743-756 1967

13 M J GONZALEZ C V RAMAMOORTHY
Recognition and representation of parallel processable streams in computer programs
Symposium on Parallel Processor Systems Technologies and Applications Ed. L C Hobbs Spartan Books June 1969

14 E C RUSSELL G ESTRIN
Measurement based automatic analysis of FORTRAN programs
Proc SJCC 1969

15 J B DENNIS
Programming generality, parallelism and computer architecture
Proc IFIPS Congress 68 C1-C7

16 H HELLERMAN
Parallel processing of algebraic expressions
IEEE Trans on E C Vol 15 No 1 Feb 1966

17 H S STONE
One-pass compilation of arithmetic expressions for a parallel processor
Comm ACM Vol 10 No 4 220-223 April 1967

18 J S SQUIRE
A translation algorithm for a multiprocessor computer
Proc 18th ACM Natl Conf 1963

19 J L BAER D P BOVET
Compilation of arithmetic expressions for parallel computation

Proc IFIPS 68 D4-B10

20 D KNUTH
The art of computer programming, Vol. 1, fundamental algorithms
Addison-Wesley 316

21 R S NORTHCOTE
Software developments for the array computer ILLIAC IV,
Univ of Illinois Rpt No 313 March 1969

B. RESOURCE ALLOCATION AND THE AVOIDANCE OF DEADLOCKS

Previous work done in the area of Cellular Logic, part of which is contained in Enclosure 2 has indicated the tremendous potential of Cellular Arrays for applications in the rapidly developing area of Large Scale Integration. In particular, the implementation of Cellular Array Logic modules to perform various frequently used arithmetic operations has received considerable attention in recent years. However, very little emphasis has been given to the possible applications of Cellular Array modules on the System level as opposed to the numerous applications proposed, throughout the literature, on the arithmetic unit level.

System management functions, so frequently performed by the operating system in a multiprocessing environment, in a final analysis are simple arithmetic operations like the ones performed by the arithmetic unit. In addition, it is a well known fact that they are being exclusively performed through the assistance of software routines that even though they are quite flexible in their implementation, they have the disadvantage of slow speed performance relative to hardware speeds.

The proposed model resulted in an attempt to investigate the possibilities of using hardware modules, implemented by Cellular Array logic easily adaptable to the LSI technology, to perform certain system management functions and thus taking advantage of their computational speed capabilities.

One of the most important considerations in designing an operating system for a multiprocessing environment is the choice and implementation of the algorithm to allocate the resources to the various jobs demanding the services of these

resources. The proposed model is intended as a unified software/hardware package, with the hardware part composed of fast, possibly microprogrammable Cellular Array modules, dedicated to perform functions related to the dynamic allocation activities in a multiprocessing system.

With this in mind, several existing algorithms for allocating resources in a multiprocessing system were surveyed and their methodology, adopted disciplines and basic decision criteria were studied as indicated in Enclosure 3. New algorithms relative to the resource allocation problem were motivated such as the one presented in Enclosure 4 dealing with the optimal arrangement of an N-Page Open String.

Recently, another criterion, namely the avoidance of system deadlocks has received considerable attention in the design of algorithms for resource allocation. The proposed algorithm and associated model are centered around this latter criterion while other criteria commonly used in existing resource allocation algorithms are also imbeded.

THE DEADLOCK CRITERION

A system deadlock may be a consequence of the fact that two tasks A and B are competing for resources mutually held by the same, thus preventing their mutual progress. If one is not allowed to preempt one of the tasks involved, he has a system deadlock at hand. If he is allowed to preempt, he introduces "swapping overhead", control, problems and increased runtime of the jobs involved.

A. N. Habermann has given a solution to this problem. His algorithm detects a deadlock before it occurs and takes steps to prevent it. The powerful

concept of his algorithm is the "safe state" which is an allocation state (a set of resources allocated to a set of tasks), free of potential deadlocks. Habermann's theorem reads as follows:

THEOREM: When no task will release its resources until it has been allocated all the resources it has claimed, the tasks will not get into a deadlock if and only if the allocation state is safe.

In terms of the notation and parameters defining this model, the necessary and sufficient condition for an allocation state A(t) to be safe can be phrased as follows: An allocation state A(t) at time t is safe if and only if after a subset (or the set) of pending requests is granted to the requesting tasks, the unallocated resources vector $\bar{u}(t)$ at the time instant $t \leq \tau$ can accommodate at least one task j according to its maximum demand, if the task happens to request so. This implies that for every allocation state A(t) and its associated sequence of unallocated resources vectors $\bar{u}(t)$ with $0 \leq t \leq \tau$, there is a set of possible feasible "next allocation states" A(t+1), at least one of which is safe.

Once the notation adopted in the model is introduced, the conditions for a safe allocation state can be stated in more formal form. Thus a safe allocation state will guarantee that at least one of the feasible allocation states at (t+1) will be safe, meaning that it will be possible to accommodate at least one task according to tis maximum demand, free of any deadlocks, while another or more tasks can proceed after the one just accommodated releases its resources. In the event this situation occurs, the first selected task must run to completion.

Following is an example of the above argument. Two types of resources and two tasks are assumed.

Let : System Capacity vector: $\begin{bmatrix}4\\5\end{bmatrix}$

Maximum Demand Matrix: $\begin{bmatrix}4 & 3\\3 & 4\end{bmatrix}$

Initial Allocation Matrix:

$$\begin{array}{c} & T_1 & T_2 \\ R_1 & 1 & 1 \\ R_2 & 1 & 2 \end{array}$$

Unallocated resources vector: $\begin{bmatrix}2\\2\end{bmatrix}$

Pending Requests Matrix: $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$

Disregard releases.

Next feasible allocation state set:

A $\begin{bmatrix}1+1 & 1\\1 & 2+1\end{bmatrix}$ B $\begin{bmatrix}1+1 & 1\\1 & 2\end{bmatrix}$ C $\begin{bmatrix}1 & 1\\1 & 2+1\end{bmatrix}$

For A: $\begin{bmatrix}2 & 1\\1 & 3\end{bmatrix}$ $r=\begin{bmatrix}1\\1\end{bmatrix}$ if P_1 asks for max demand, state becomes

$\begin{bmatrix}4 & 1\\3 & 3\end{bmatrix}$ <u>unsafe</u> $\begin{bmatrix}2\\1\end{bmatrix}+\begin{bmatrix}0\\1\end{bmatrix} \not\leq \begin{bmatrix}4\\3\end{bmatrix}$

If P_2 asks for max demand, state becomes

$\begin{bmatrix}2 & 3\\1 & 4\end{bmatrix}$ <u>unsafe</u>. $\begin{bmatrix}1\\3\end{bmatrix}+\begin{bmatrix}1\\1\end{bmatrix} \not\leq \begin{bmatrix}3\\4\end{bmatrix}$

Following the same procedure, the only choice for a safe state is when

C $\begin{bmatrix} 1 & 1 \\ 1 & 3 \end{bmatrix}$, $r = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$ and if P_2 asks for its maximum demand

$\begin{bmatrix} 1 & 3 \\ 1 & 4 \end{bmatrix}$ is safe since $\begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$

So the request vector for P_2 can be granted safely.

DESCRIPTION OF THE MODEL

1. The set of system resource types S_R represented as a vector of the form:

$$\bar{S}_R = \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_m \end{bmatrix}$$

The number of resources in each resource type is $|R_i| = N_{R_i}$. The resources may be anything from I/O devices to files or system library subroutines. Resources of a certain type that cannot be shared may be considered of a "special" type. Otherwise, no identification is attached to the various resources within a certain type. In addition one may choose to exclude Main Memory and CPU's if there are already efficient algorithms established for their allocation.

2. The set of independent tasks S_T, in the system at time $0 \leq t \leq \tau$ where τ a finite time interval - represented as a vector of the form

$\bar{S}_T(t) = [T_1, T_2, \ldots T_k]$ with $k \leq n$ in case one would like to set $|T_n|$ = max number of tasks that can be handled by the system at any time. In this model an upper bound will not be set, thus leaving the system capacity ope

3. A queing model, to be considered, will serve as an input to the system in the form of an input buffer. One may operate on this queing model independently by re-evaluating task priorities and dynamically rearrange the queue of tasks to come into the system.

4. D – matrix = $[D_{ij}]$ with D_{ij} = max demand of resources of type i, needed for the completion of task j.

5. A(t) – Matrix = $[A_{ij}]$ with A_{ij} = number of resources of type i allocated to task j at the time instant $0 \leq t \leq \tau$.

In parallel with the allocation matrix there will be two other matrices associated, T_{RQ}, T_{RL}, representing the "Pending Requests" matrix and the "Current Releases" matrix of the same order as the current allocation matrix. An entry of zero in the "Pending Requests" matrix indicates no request was issued by the corresponding task for the particular resource type. Releases can be immediately returned to the Unallocated Resources vector to be mentioned shortly.

The column vectors of the above mentioned matrices, representing the max demand request $\bar{D}$, the current resource allocation $\bar{A}(t)$, the requests $\bar{T}_{RQ}$ and releases $\bar{T}_{RJ}$, vectors of any task j, will be of interest for computational purposes in the model rather than the corresponding matrices themselves.

6. $\bar{u}(t)$ – vector of the form $\bar{u}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_m(t) \end{bmatrix}$

with $u_i(t)$ = number of resources that remain unallocated at the time instant $0 \le t \le \tau$ and given by the relation

$$u_i(t) = N_{R_i} - \sum_{j=1}^{k} A_{ij}(t) \pm \sum_{j=1}^{k} T_{ij}(t)$$

where the plus sign in the third term indicates resources are released and are returned to the system and the minus sign indicates that resources are allocated to various tasks.

7. The Allocation state, at time t, of the system may be expressed as $A(t) = (D, A(t), u(t))$.

When testing to find the subset of safe, next allocation states from the set of next feasible allocation states, one needs to generate and look at the "Lookahead" form of the allocation matrices:

$A(t+1) = [A_{ij}]$ with

$A_{ij}(t+1) =$
- D_{ij} for some $j = J$, $1 \le J \le k, \forall i$
- $A_{ij}(t)$ if the request of task j is not granted
- $A_{ij}(t) + T_{ij}(t)$ $j \ne J$, $\forall i$

as shown in the previous example.

8. $\bar{I}(t)$ - vector of the form

$$\bar{I}(t) = \begin{bmatrix} i_1(t) \\ i_2(t) \\ \cdot \\ \cdot \\ \cdot \\ i_1(t) \end{bmatrix}$$

with $i_1(t)$ = number of resources representing the initial request for resources of type i by the task j, candidate to be considered to enter the system.

Each job in the input buffer will be requested to indicate its max demand and initial demand resource vectors, before entering the system.

9. $\bar{W}(t)$ - vector of the form

$$\bar{W}(t) = [W_1(t), W_2(t) \ldots . W_k(t)]$$

with $W_j(t) = \sum_{i=1}^{m} W_i A_{ij}$, $\forall \quad j$

A "weight number " will be associated with each resource type and for the allocation vector of a given task j, the above sum of products will be formed and will be entered in the "weight vector" which will be used for dynamic partitioning of the tasks currently in the system prior to request granting considerations.

The above brings up another feature to be present in the model. The input buffer as well as the various matrices mentioned above together with the weight vector will be partitioned such that the members of each partition will belong to a distinct Priority level. The priority level partitioning of the tasks in the input buffer and the ones currently in the system need not and will not necessarily be the same.

FEASIBILITY REQUIREMENTS

1. The maximum demand vector for a task j, must be at most equal to the resource capacity vector of the system,

$$\text{i.e. } \bar{D} \leq \bar{S}_R \qquad 1 \leq i \leq m.$$

2. The initial demand vector for a task j, must be at most equal to its maximum demand vector, i.e.

$$\bar{I} \leq \bar{D} \quad 1 \leq i \leq m.$$

3. The sum of all allocation vectors corresponding to all tasks j, $1 \leq j \leq k$ currently in the system must be at most equal to the resource capacity vector of the system i.e.

$$\sum_{j=1}^{k} A_{ij} \leq \bar{S}_R \qquad 1 \leq i \leq m.$$

BASIC ASSUMPTIONS

1. Only one task is allowed to request its max demand vector $\bar{D}$ during an allocation state at time $0 \leq t \leq \tau$. If more than one tasks do so, their requests will be held up till the task that acquired its max demand is through using its resources.

2. It is essential that every task j indicates its max demand vector and Initial demand vector for the algorithm - to be presented shortly - to be able to detect and prevent a deadlock.

3. All k - tasks currently in the system will be mutually independent as far as resource requirements are concerned.

4. No task will release any resources it presently holds, unless using them to completion. This non-preemptive discipline will assure file protection and minimize cost resulting by resource swapping between tasks.

BASIC SCHEDULING DISCIPLINES

1. The model and its associated algorithm is designed to satisfy the primary criterion of detecting and preventing a deadlock situation that may occur in the system at hand.

2. The model and its associated algorithm will give first preference to high priority tasks. Priorities in both the input buffer and the system may be at any point dynamically re-assigned to the various tasks.

3. The model and the associated algorithm will try to accomodate as many tasks as possible, satisfying the principle of "good service", provided the deadlock fear has been resolved.

4. The model and its associated algorithm will allocate the resources to the tasks currently in the system to satisfy a near optimal resource utilization, again provided the deadlock fear has been resolved.

5. As mentioned earlier, there is no identification attached to the resources other than the one pertaining to the specific type. Thus, selected files or library routines generally used in the "exclusive control" mode and resident in a certain storage resource of type i, may be considered as different types of resources altogether.

THE ALGORITHMIC DISCIPLINES

1. Each task j in the input buffer will be assigned to a priority level, a label which will carry along upon its entry in the system and which will be dynamically modified as requests and releases for resources occur.

2. All m-resource types will be assigned a unique weighting factor according to predetermined criteria which will be utilized in the dynamic re-assignment of the tasks to the priority levels. Thus, resources may be termed as critical, scarce, accessible only by a selected subset of tasks, etc.

3. A pre-determined "time-slice - τ" will be an important parameter in the algorithm. An attempt will be made to satisfy - in real time - all

arbitrarily occuring requests for resources at discrete time instants $0 \leq t \leq \tau$. In addition, there will be an infinitely small time subinterval associated with the latter part of τ, which shall be called "tolerance". When this tolerance is reached at each τ, a new time slice will be initiated. At the beginning of this new τ the tasks will be reassigned to appropriate priority levels and an attempt will be made to satisfy pending requests collectively, starting at the highest partition. For the remaining portion of τ, additional tasks will be tested to be brought into the system and new requests will be satisfied as they occur, till the tolerance is reached once again.

The important assumption here is that the time required for the necessary computations to dynamically reassign priorities, compute weights, test for safe allocation states, after granting requests, etc. will be very small compared to the duration of the time slice τ, since all required computations will be performed by hardware modules.

THE ALGORITHM

1. Initialize all integral components of the model, such as Allocation and Demand matrices, Requests and Releases matrices, Capacity, Unallocated Resources and Weights vectors.

2. If the Demand and Initial vectors of the first job in the highest priority level of the input buffer satisfy the feasibility criteria:

   A. Start time slice τ.

   B. Set first vector of the allocation matrix equal to the Initial vector of the first job.

C. Set first vector of the demand matrix equal to the Demand vector of the first job.

If the Demand and Initial vectors of the job do not satisfy the feasibility criteria, the input buffer is canned till a suitable job is found.

3. Make an attempt to introduce the next suitable job into the system as long as there are no requests registered for the currently active jobs and the time interval has not reached the tolerance. Otherwise go to step 4.

4. If there are requests from the currently active jobs in the system, make an attempt to satisfy these requests as they occur, till the time tolerance is reached. When this event occurs go to step 5. If time is not within tolerance and no registered request can be satisfied because it does not meet the "safe allocation" state requirements, go to step 3.

5. When tolerance is reached, reset the time slice, use the weights vector as a guideline to dynamically reassign priorities to the currently active jobs in the system and make an attempt to satisfy the maximum number of pending requests collectively, starting at the highest priority level. With the basic assumption that the time required for this process is very small compared to the time slice τ, return to step 3.

A FEW COMMENTS

1. The algorithm will terminate only when the input buffer is empty and all entries in the matrices considered in the model are zeros. Otherwise it will alternate between the above mentioned steps, namely:

A. Attempt to grant the requests for resources by the currently active jobs in the system as they occur.

B. Attempt to enter new jobs in the system after (A) has been considered. When the time tolerance is reached go to (C).

C. Dynamically distribute the jobs in appropriate priority levels and make an attempt to grant pending requests collectively.

2. Releases are returned to the unallocated resources vector just prior to request granting considerations.

3. The basic operations involved in the algorithm and needed to determine the sequence of "safe allocation" states and dynamically reassign jobs into priority levels are addition(subtraction), multiplication and sorting all of which can be very efficiently performed by cellular array hardware modules.

4. When an attempt is made to grant a request in real time, the test performed to determine whether the resulting allocation state is safe, requires at least one addition and subtraction but no more than k, where k is the number of jobs in the current allocation matrix.

5. When an attempt is made to grant all pending requests at the beginning of the new time interval τ, first a sorting operation is required on the weights vector as well as on the max demand, current allocation and request matrices. This sorting not only achieves an efficient assignment of jobs in different priority levels but considerably reduces the number of computations needed to

determine the next safe allocation state in this case. An exhaustive combinatorial procedure involving at each stage computations as in part 4 above would otherwise be needed which might reach a number of the order

$$|P| \sum_{k=1}^{n_i} \frac{n_i!}{k!\,(n_i - k)!} \quad \text{where}$$

$|P|$ = Number of priority levels in the system at time t

n_i = Number of tasks in each priority level.

PROPOSED EXTENSIONS

1. Develop a queueing model which will complement the system and determine the behavior of the job input buffer. In other words adopt disciplines for the dynamic determination of external priority assignments.

2. Apply some probabilistic considerations in the reassignment of jobs to the priority levels within the system and study their effects on the model

3. Implement and simulate the model by an ALGOL program to determine its behavior, efficiency and timing.

4. Design the required hardware Cellular Array modules to perform the operations called for by the algorithm.

5. Develop the subroutines or software mechanism, possibly in the form of a microporgram, which with the assistance of the hardware modules will perform the resource allocation.

6. Evaluate and compare the efficiency and performance of the algorithm, to other existing algorithms performing resource allocation.

REFERENCES

1. Habermann, A. N., "On the harmonious cooperation of abstract machines" Ph.D. Thesis. Math. Department, Technological University, Eindhoven, The Netherlands, October 1967.
2. Habermann, A. N., "Prevention of System Deadlocks" Communications of the ACM, Volume 12, No. 7, July 1969.
3. Dijsktra, E. W. "The structure of "THE" - multiprogramming system Communications of the ACM, Volume 11, No. 4, May 1968.
4. Murphy, J. E., "Resource allocation with interlock detection in a multi-task system". Proc. AFIPS 1968, FJCC Volume 33, part 2.
5. Havender, J. W., "Avoiding deadlocks in multitasking systems" IBM Sys. Journal, Volume 2, 1968.

Fast multiplication cellular arrays for LSI implementation

by C. V. RAMAMOORTHY and
S. C. ECONOMIDES

The University of Texas at Austin
Austin, Texas

INTRODUCTION

The inherent capabilities of Large Scale Integration technology have recently shifted attention toward two major concepts in the design of functional computer subsystems; the concepts of Functional Modules and Cellular Arrays.

The Functional Module concept emphasizes the possible standardization of frequently used common digital subsystem units such as registers, adders, counters, etc. Because of the unique iterative properties also displayed by these units it is common to view them as building blocks (functional modules), built on a single substrate of material, the interconnection of which can expand significantly their functional capabilities. In addition to standardization, their massive production may suggest low cost subsystems.

The Cellular Array concept allows the interconnection of several types of mutually independent logic blocks, the cells, in various geometric configurations to perform a desired operation.

This paper is an attempt to combine the above two approaches in the realization of a Binary Cellular Array multiplication unit easily adaptable to the LSI realization techniques and speculate the possibilities of the realization of other similar such functional units aiming to lower the cost per unit of computation and possibly increase the overall system reliability.

Multiplication was chosen in the study because it forms the basis of division and square root operations by iterative methods as well as others indicated by design trend of present day computing systems.

The methodology and retrospective design procedures of the Multiplication Array are presented. Interconnection arrangements at the cell level, for the array formation, as well as the module level by bringing all module inputs and outputs at the terminals of the "package", for the purpose of assembling larger multiplication units, are also shown.

Since in any LSI circuit testing imposes a complex problem some diagnostic schemes are suggested for reconfiguration and operation under reduced capabilities or even by automatically switching in of a permanently connected spare module.

Other LSI considerations in terms of cell or module fan-in/fan-out, total number of pins required per package, chip sizes and densities and rough cost estimates are also discussed.

Single bit multiplier

Figures 1 and 2 show the integral parts and the detailed cellular array structure of the multiplication unit, in which each row of the array corresponds to one bit of the multiplier. The array uses K-bit operands producing 2K bit product.

To achieve fast execution time the multiplication is done by performing K-1 carry save additions (simple EXCLUSIVE-OR operations) followed by a fast binary addition. Since the cells in the array operate asynchronously, the unit as a whole can operate faster without using a clock pulse.

We shall next explain the single-bit multiplication unit in some detail.

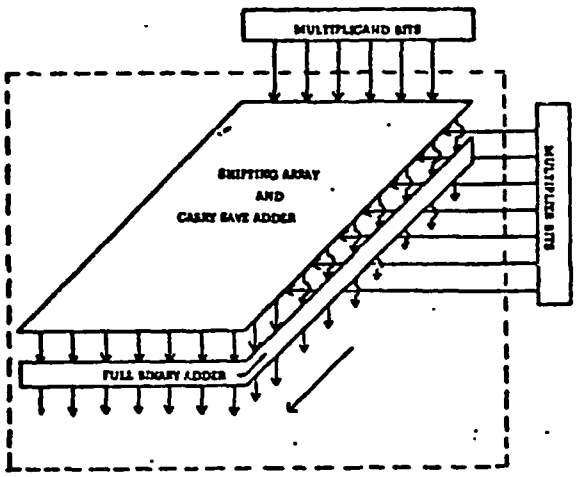


Figure 1—The integral parts of the asynchronous multiplication array

Let the multiplicand be represented by the binary vector $M = (m_1, m_2, \cdots m_k)$ and the multiplier by the binary vector $N = (n_1, n_2, \cdots n_k)$.

A kx2k, P matrix is now generated starting from right to left (whose elements p_{ij} are computed from the relation $p_{ij} = m_i \cdot n_j$, $p_{ij} \in \{0, 1\}$ with the following conditions

$$m_{j-i+1} \text{ if } n_i = 1 \text{ and/or } \begin{cases} 1 \le i \le k \text{ for } i = 1, 2, 3, \ldots k \\ i - 1 < j < k + 1 \text{ for } i = 1, 2, 3 \ldots k \end{cases}$$

$p_{ij} =$

$$0 \text{ if } n_i = 0 \text{ and/or } \begin{cases} 1 \le i \le k \text{ for } i - 1, 2, 3 \ldots k \\ k + 1 \le j \le i - 1 \text{ for } i = 1, 2, 3 \ldots k \end{cases}$$

In terms of the array to be implemented, this condition implies that for the range "i," "j" where $p_{ij} = 0$ no cell will be required to perform a logic function. Thus the [P] matrix has the following form:

$$\begin{matrix} p_{1,2k-1} & p_{1,2k-2} \ldots p_{1,k} \ldots p_{12} & p_{11} & p_{11} \\ p_{2,2k-1} & p_{2,2k-2} \cdots p_{2,k} \cdots p_{22} & p_{21} & p_{21} \\ \cdot & \cdot \quad \cdot \quad \cdot \quad \cdot & \cdot & \cdot \\ \cdot & \cdot \quad \cdot \quad \cdot \quad \cdot & \cdot & \cdot \\ \cdot & \cdot \quad \cdot \quad \cdot \quad \cdot & \cdot & \cdot \\ p_{k,2k-1} & p_{k,2k-2} \quad p_{k,k} \quad p_{k,1} & p_{k,2} & p_{k,1} \end{matrix}$$

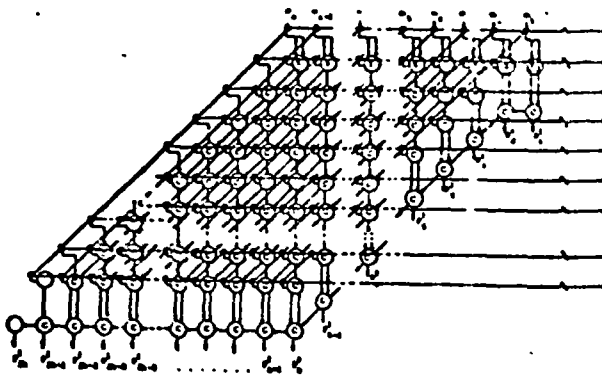


Figure 2—The "single-bit" asynchronous multiplication cellular array

The following example will illustrate the above matrix formation.

EXAMPLE

MULTIPLY . . . $M = (10101)$ and $N = (111111)$

then the P matrix is $P =$

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

The above matrix can be realized by selective ANDing of components of M and N. This "Shifting Network" accomplishes the proper positioning of the numbers to be added before their addition, just as in the conventional multiplication. Arrays of Carry Save Adders are used to perform the addition of these binary numbers utilising Wallace's algorithm.[1]

The first stage of the Carry Save Adder adds the first two rows of the P matrix (first two generated partial products) thus generating two vectors—the first partial sums and the first carry having the form:

$$s = (s_{1,2k-2} \quad s_{1,2k-3} \ldots s_{1,k} \ldots s_{11})$$

$$c = (c_{1,2k-2} \quad c_{1,2k-3} \ldots c_{1,k} \ldots c_{11}); s_{ij}, c_{ij} \in \{0, 1\}$$

The double subscript is used to identify the above vectors with corresponding positions of the P matrix that contributes to their generation.

The logic functions yielding the elements s_{ij} and c_{ij} are:

$$s_{ij} = p_{ij}\bar{p}_{ij} + \bar{p}_{ij}\,p_{ij}$$

$$c_{ij} = p_{ij} \cdot p_{ij}$$

where $j = 2, 3, \cdots 2k - 1$. The composite cells are shown in Figure 3a.

In the subsequent stages the Carry Save Adder will add three vectors: The sum vector generated at the previous stage, the carry vector generated at the previous stage shifted once to the left and the next row vector of the P matrix.

The logic functions producing the new s and c vectors

$$s = (s_{i,2k-1}\, s_{i,2k-2} \ldots s_{i,j} \ldots s_{i1})$$

$$c = (c_{i,2k-1}\, c_{i,2k-2} \ldots c_{i,j} \ldots c_{i1})$$

for $i = 3, 4, \ldots k$, and $j = 1, 2, 3 \ldots 2k - 1$ are:

$$c_{i+1,j} = s_{ij}c_{ij-1} + s_{ij}p_{i+1,j} + c_{ij-1}p_{i+1,j}$$

$$s_{i+1,j} = \bar{s}_{ij}\bar{p}_{i+1,j}c_{ij-1} + \bar{s}_{ij}p_{i+1,j}\bar{c}_{ij-1} + s_{ij}\bar{p}_{i+1,j}\bar{c}_{ij-1} + s_{ij}p_{i+1,j-1}$$

The composite cell 'C' is shown on Figure 3b. After the Carry Save Addition has been performed for all the partial product row vectors of the matrix P, a Ripple Binary Adder is used to add the sum and carry row vectors of the last stage of the Carry Save Adder. The typical cell of this Ripple Binary Adder has the same structure as "cell C" of the Carry Save Adder, except that it ripples through the carries generated to next high order position and puts out the correct binary sum which of course involves any carry incident into it from the previous stage. The output of the Ripple Binary Adder is the final product of the multiplication.

The superposition of the "Shifting Array", the "Carry Save Adder" and the "Ripple Binary Adder" resulting in the "Single Bit Multiplication Cellular Array" is as shown in Figures 1 and 2.

It was found that with a Carry Save Adder there is considerable gain in the time propagation over the choice of Full Binary Adder. Assuming a uniform delay d for each cell in the array, the total execution time T_p of k bit by k bit multiplication is bounded between the limits $(k-1)\,d \leq T_p \leq 2kd$. The lower limit (k-1) d is the total delay in the Carry Save Adder while the upper limit 2kd depends on the choice of the device for the final Full Binary Addition. This, as compared to the maximum delay requirement in a conventional multiplier due to k(k-1) d full binary additions plus k-single bit left shifts. The asynchronous multiplication array, as implemented is shown in Figure 2.

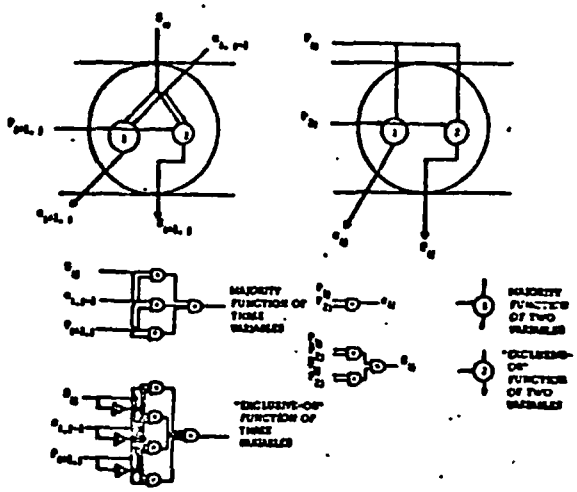


Figure 3a, b—Cell "S", Cell "C".

Two-bit multiplier

Upon examination of this array it was decided that the time propagation and therefore the computational speed could be further improved by reducing the Carry Save Adder stages, in other words, the rows of the array. This also improves the attenuation factor of the cell inputs as they ripple throug the array.

An alternate multiplying algorithm, examining the multiplier bits in subsets of two, was investigated resulting in the block diagram of Figure 4 which displays the integral parts of the modified array. To illustrate the algorithm better, this was assumed to be an m × n instead of a square array and the multiplier parts now are:

1. The m + n + 2—bit register for the multiplicand
2. The n + 2—bit register for the multiplier
3. The m + n + 3—bit registers for the final product
4. The Binary Shifting Array (BSA)
5. The Input Control Circuit (ICC)
6. The Carry Save Adder (CSA)
7. The "End Around Carry" Accumulator (EACA).

Before investigating the above circuits the general algorithm concept must be established. This algorithm

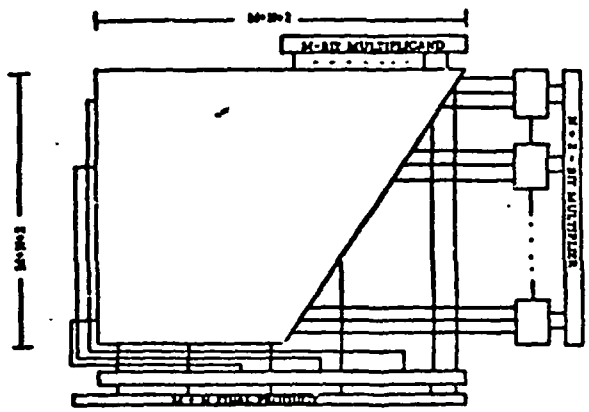


Figure 4—The modified "two bit" multiplier

calls for three types of decisions in each multiplication stage: ADD or SUBTRACT a single multiple of the multiplicand and SHIFT without generating any multiples of the multiplicand. This is opposed to the conventional multiplication which requires only shifts of the multiplicand and their addition. For the possible four 2-bit combination one has the following obvious interpretation:

a. Combination $00_2 = 0_{10}$ Add nothing to the partial product
b. Combination $01_2 = 1_{10}$ Add one times the multiplicand to the partial product
c. Combination $10_2 = 2_{10}$ Add two times the multiplicand to the partial product
d. Combination $11_2 = 3_{10}$ Add three times the multiplicand to the partial product.

Combinations (a), (b) impose no difficulty in their generation. Combination (c) requires a 1-bit shift of the multiplicand to the left according to the obvious simple fact: to generate any 2^n-th multiple of a binary number (the multiplicand in this case), where n is any integer $n \geq 0$, shift the number n-bit position to the left. For example, to generate $16 \times m = 2^4 \times m$, shift m four bit positions to the left. For combination (d) one notices that the multiplicand can be expressed in the following two ways:

(1) $(4 \times m) - (1 \times m)$ (2) $(2 \times m) + (1 \times m)$.

The first representation was chosen for this multiplication algorithm, according to which a complementation (2's complement) of one times the multiplicand is performed and added to the corresponding present stage of the multiplication array while a request is issued to add one times the multiplicand in the following stage *in that order*. The latter request is taken care of by adding "1" to the bit pair of the multiplier corresponding to the next multiplication stage, thus increasing the pair's integer value by one. This is commonly known as a "carryout."

The subtraction of multiplicand from the partial product is performed in two stages. The one's complement of m is "added" into the Carry Save Adder of the row. A "one" in the lowest order bit position corresponding to the row is generated and inserted into the End-Around-Carry Accumulator (EACA), at the appropriate column. Together this constitutes adding the two's complement of m after appropriate shifting. Thus any sequential borrow propagation is prevented at the Carry Save Adder stages. Since the "End-Around-one's" if generated by any or all rows are inserted at distinct columns of the EACA the latter performs at most one accumulation during a complete multiplication cycle. It must be remembered that partial products generated at each row are passed to the next row of cells with a 2-bit left shift.

The following two tables indicate the decisions that have to be made when the various bit pair combinations are encountered at a given stage when no carryout (Table I) or a carryout (Table II) has been generated in the previous stage.[5]

Bits	*Multiple of m generated*	*Carryout*
00	0	0
01	1	0
10	2	0
11	−1	1

Table I

Bits	*Multiple of m generated*	*Carryout*
00	1	0
01	2	0
10	−1	1
11	0	1

Table II

Finally to illustrate the overall performance of the modified multiplier with minimum effort an example of a 4-bit positive multiplicand times a 6-bit positive multiplier producing a $6 + 4 = 10$-bit long product is presented. The extension of the algorithm and techniques involved can be easily extended for an arbitrary bit length multiplicand or multiplier.

The binary shifting array

The BSA generates the elements $p_{ij} \in (0,1)$ and such that

$$p_{ij} = 0 \text{ for } 1 \leq j \leq m + 1; i \geq 3$$
$$j \leq m + n; i \geq 3$$

where

m = no. of bits in M

n = no. of bits in N

with the rest of the p_{ij}'s varying according to the corresponding multiplicand bits. Its implementation procedure is as follows:

1. Provide for one additional bit pair at the most significant part of the multiplier by inserting two zeroes in the register. This will take care of a possible generation of a "carryout" at the two most significant bits of the multiplier. Provide for as many zeros to the left-hand side of the multiplicand register to make it (m + n + 2) bits long.
2. Examine the multiplier bits two at a time from the least significant to the most significant bits.
3. Generate the following three numbers for each multiplier bit pair:
   a. The multiplicand
   b. The multiplicand inverted (one's complecomplement)
   c. Twice the multiplicand.

Repeat for the next bit pair until all n-multiplier bits are used. For this particular example the procedure will yield the formation of possible CSA inputs, where the "boxed in" numbers will be the rows of the P matrix chosen by the control lines of the ICC (Figure 5a).

The two numbers are placed in the registers with the least significant bit of the multiplier starting at the top. For every bit pair of the multiplier there is a corresponding triplet of "AND" gate rows and one of inverters, all together being capable of generating any of the desired forms of the multiplicand called for in Tables I and II.

The "AND" gates have two inputs and one output, one of the inputs being a multiplicand bit bussed across and the other being the appropriate line activated by the ICC. The outputs of the leftmost column are used to keep count of the "End Around Carries" and are directly connected to the appropriate positions of the EACA.

The input control circuit

The ICC is a column of (n/2 + 2) rectangular cells (see Figure 4). Its operation is to select the appropriate multiplicand multiple for each possible bit pair combination, by the way of three output lines: L_0, L_1, L_2

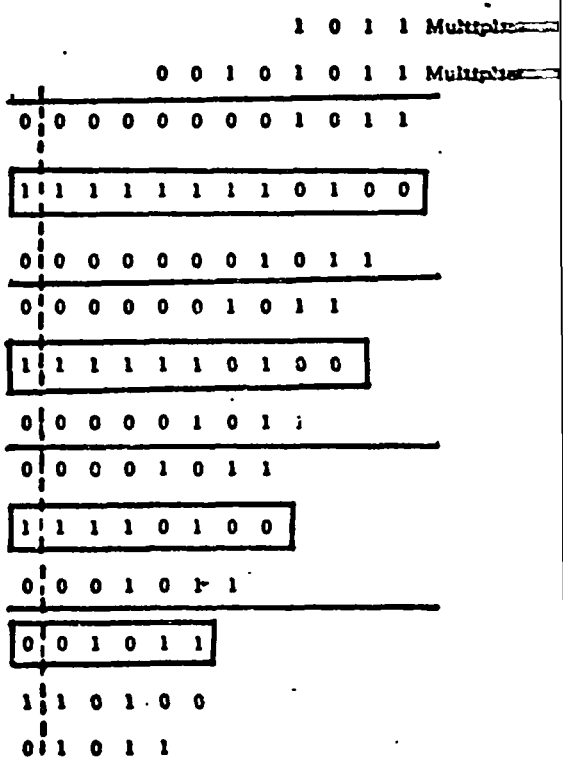


Figure 5a—Multiplication example

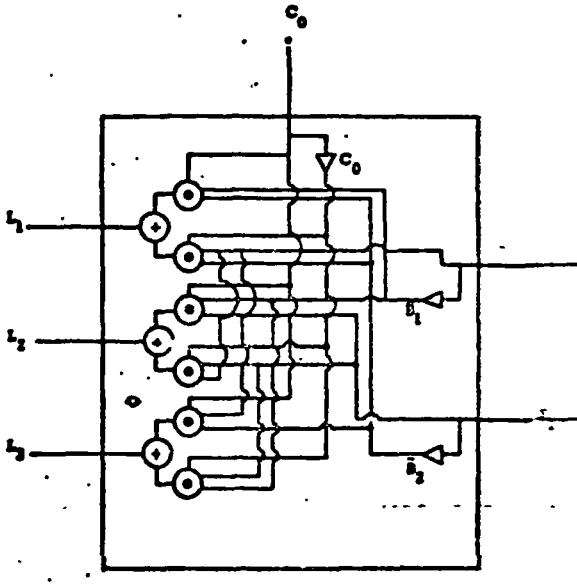


Figure 5b—Cell-K of the input control circuit

L_1 activates the single multiple of the multiplicand (first "AND" gate row of each group of rows in the ESA). L_2 activates the 2's complement of the multiplicand (second "AND" gate row, directly under each row of inverters). L_3 activates the double multiple of the multiplicand. Therefore, the typical cell of the ICC has B_1, B_2, and C_0 as inputs and L_1, L_2 and L_3 as outputs. Its logic functions are shown below. B_1 and B_2 are any two consecutive bits and C_0 is the carryout. The logic:

B_1	B_2	C_0	M L_1	$\overline{M}$ L_2	2M L_3
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	0	1	0
1	1	1	0	0	0

Note: The interpretation of $B_1, B_2 = 01$ is not one times the multiplier as it would obviously appear, but it is instead two times the multiplicand because of the way the multiplier is placed in the register, vertically with the least significant bit on the top. The $B_1, B_2 = 10$ combination is interpreted in a similar manner

$$L_1 = \bar{B}_1\bar{B}_2C_0 + B_1\bar{B}_2\bar{C}_0$$

$$L_2 = \bar{B}_1B_2C_0 + B_1B_2\bar{C}_0$$

$$L_3 = B_2\bar{B}_1C_0 + \bar{B}_2B_1\bar{C}_0$$

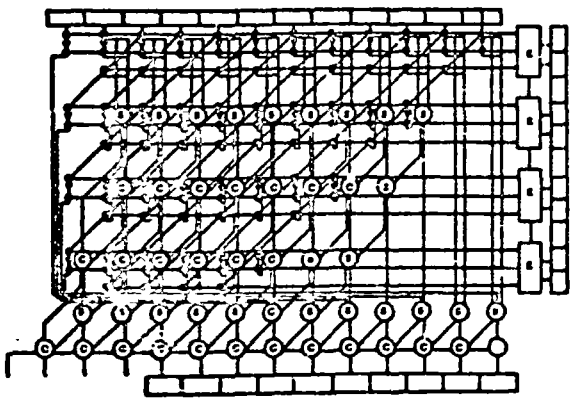


Figure 6—The binary multiplying cellular array

The typical cell "K" of the ICC is shown in detail in Figure 5b.

The carry save adder, end around carry accumulator and full binary adder

A layout of the inputs to the CSA stages, the EACA and FBA is displayed below. The groups of binary numbers between the lines represent the actual inputs to a particular row of cells. The first three groups are CSA row inputs. The fourth group represents the EACA inputs and the final group, those of the FBA. All binary numbers representing partial products are of course *P* matrix row vectors activated by the ICC lines due to a particular multiplier bit pair combination.

```
      1 1 1 1 1 1 1 0 1 0 0    1st partial product
      1 1 1 1 1 0 1 0 0        2nd partial product
  ---------------------------
      0 0 0 0 0 1 0 0 1 0 0    1st partial sum
    1 1 1 1 1 0 1 0 0 0 0 0    1st carry
      1 1 1 0 1 0 0            3rd partial product
  ---------------------------
    1 0 0 0 1 1 0 0 0 1 0 0    2nd partial sum
  0 1 1 1 0 0 1 0 0 0 0 0 0    2nd carry
    0 1 0 1 1                  4th partial product
  ---------------------------
    0 1 0 0 0 1 0 0 0 1 0 0    3rd partial sum
  1 0 1 0 1 1 0 0 0 0 0 0 0    3rd carry
            0 1 1 1            End Around Carries
  ---------------------------
  1 0 0 0 1 1 1 0 1 0 0 0 1    4th partial sum
  0 1 0 0 0 0 0 0 0 1 0 0 0    4th carry
  ---------------------------
  1 1 0 0 1 1 1 0 1 1 0 0 1    Final Sum (Result)
```

Figure 6 shows array after superimposing the individual circuits.

It can be easily noticed that there is a reduction by a factor of two in the total number of cell rows required for the array and therefore in the total final propagation T_p, at the expense of some additional control logic, a number of inverters and an additional stage for the EACA. No further complexity in the cell structure results, thus the originally developed cells were used, with a minor modification for cell S as shown in Figure 7a. This cell may also be present in the single bit multiplication array.

It must also be noticed that the overflow of bits resulting in the left-most significant part of the final

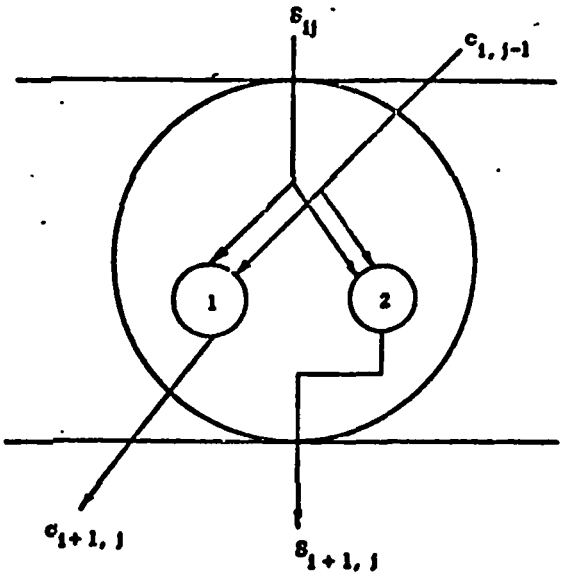


Figure 7a—Cell "S"—A form of Cell "S"

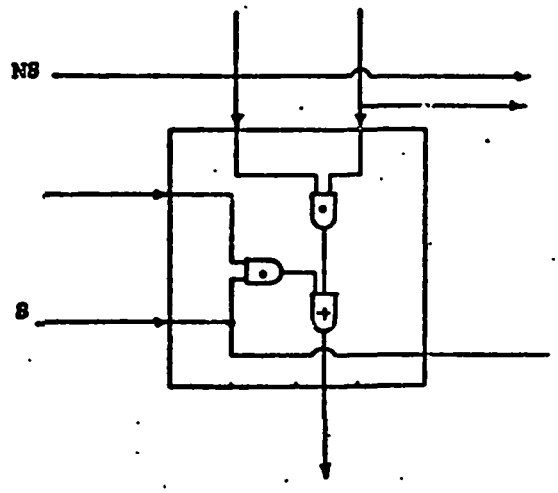


Figure 7b—Cell "R"—Reconfiguration cell

product register may be advantageously utilized for sign and decimal point considerations.

Diagnostics and reconfiguration

In order to incorporate diagnostics in the array and study the interconnection problem, a standard size module had to be assumed. It was felt that the implementation of a 64 × 64 bit multiplier would be a good choice for all practical purposes. An interconnecting scheme of standard dimension 64 × 8 bit modules to realize the 64 bit multiplier was then devised aiming to minimize the number of pins per module necessary for the interconnection.

As seen in Figure 8, the resulting 64 × 64 multiplication unit requires 2-Full Binary addition stages and 4-Carry Save addition stages per module, a total of 32-Carry Save additions and 15-Binary Additions (only one for the first module). However, there is a real time overlap between these various stages, and by utilizing a pipelining technique and a series of flip-flops after each FBA, a 100 percent utilization of the unit during computation is achieved, and the multiplication cycle is considerably faster. This is illustrated shortly in connection with Table III.

The basic module as displayed in Figure 6 has to be modified further for the interconnection. An extra FBA and additional gating for diagnostic purposes is

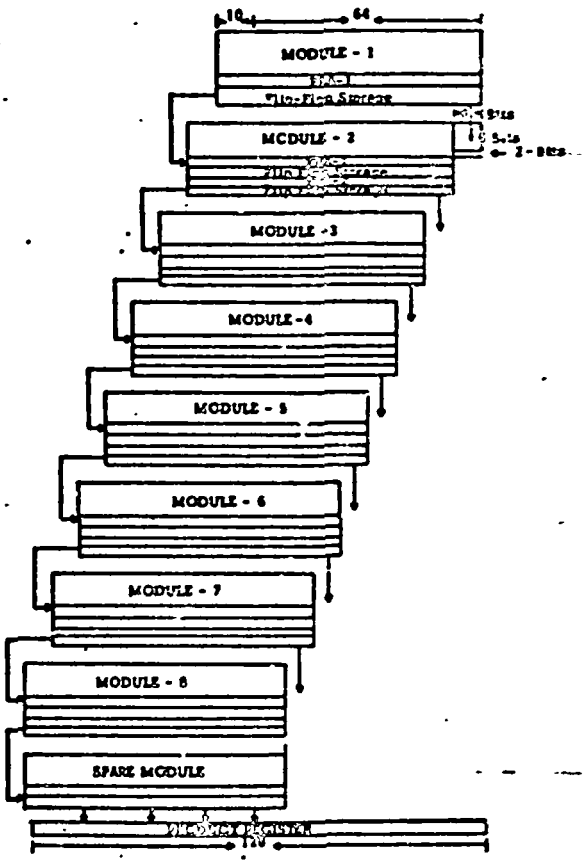


Figure 8—Example of an assembled 64 × 64-bit multiplication unit using the pipelining scheme

introduced in every module between the output of its respective FBA and what is shown as a product register. The typical newly developed cell for the diagnostics and reconfiguration is shown in Figure 7b, while the above mentioned modifications are displayed in detail in Figure 9 for a typical module.

As seen, three additional control lines are needed to perform the following functions.

a. To relay a Fault or No-Fault signal, indicating that a fault has or has not occurred in one particular module (NF/F) (e.g., if F = 0 NF = 1).
b. To relay a No Shift signal for the output of this module, (NS = 1) if no fault has occurred in the preceding module.
c. To relay a shift, eight-bits to the right, (S = 1) for the output of this and all subsequent modules if a fault has been detected in the preceding module.

The detection of the fault could be accomplished by a software routine which may check the final product of the unit periodically and appropriately set the flip-flops of the control signals.

By shifting the outputs of all subsequent modules to the malfunctioning one eight-bit positions to the right while forcing the output of the faulty module to be equal to zero at the same time and simultaneously introducing the spare module which is permanently connected to the unit, one can still achieve 100 percent computational efficiency. If another module fails to function properly, by applying again the same reconfiguration scheme the unit will function with a reduced capability since the eight-least significant bits of the multiplier will be lost. No provision has been made at this point if two modules fail to function properly at the same time. At least one of them must be replaced to put the multiplication unit back in service.

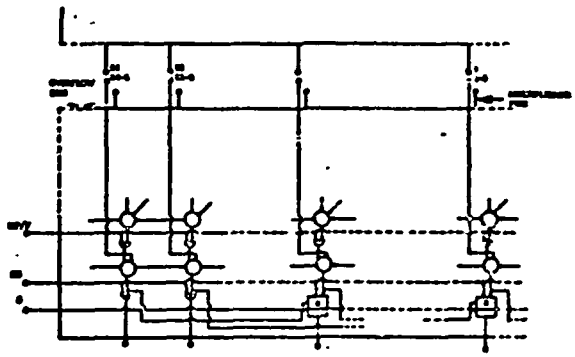


Figure 9—The combinational logic gating for reconfiguration

Aiming to maximize the number of multiplications per unit time, as already mentioned, one can introduce storage elements at intermediate points. This allows the unit to accept a new set of operands without waiting for the total completion of the present computation.

Consider an m × m bit multiplier module. If the intermediate computations are stored after the Carry Save adders, the first Binary adder and the second Binary adder, the rate of multiplications in the module per unit time will be

$$R_m = \frac{1}{\max [t_{cs}, t_b]} \text{ where}$$

t_{cs} = Total time propagation through the CSA.

t_b = Total time propagation through the FBA for the binary addition of two m-bit binary numbers.

Then the number of storage elements required per module is 2m + m + m = 4m. If, however, storage elements are inserted at the outputs of the two Binary Adders only, as shown in Figure 8, the maximum rate of multiplications in each module per unit time will be

$$R_{max} = \frac{1}{t_b + t_{cs}}$$

while the total number of storage elements required will be decreased by half, that is 2m.

The table below gives the sequence of events in the first four modules of the 64 × 64 composite multiplier unit of eight modules, based on the pipelining technique.

Table III

MODULES

TIME UNITS	1	2	3	4
1	B_{11}	B_{11}	B_{11}	B_{11}
2	B_{21}	B_{21}, B_{12}	B_{21}	B_{21}
3	B_{31}	B_{31}, B_{22}	B_{31}	B_{31}
4	B_{41}	B_{41}, B_{32}	B_{41}, B_{22}	B_{41}, B_{31}
5	B_{51}	B_{51}, B_{42}	B_{51}, B_{32}	B_{51}, B_{41}

Each time unit in the above table corresponds to the factor $t_b + t_{cs}$, and B_{ij} represents the j^{th} binary addition of the i^{th} multiplication.

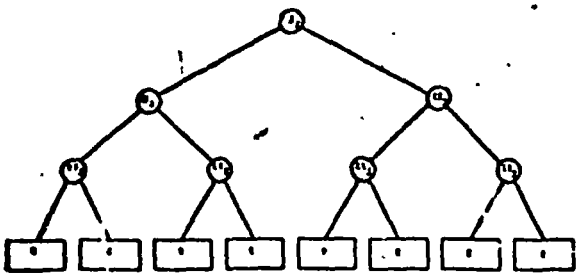


Figure 10—An alternate interconnecting scheme for the 8-modules of the 64 × 64 multiplication unit

Another interconnecting scheme which has not been investigated yet in detail but seems to be equally as efficient, considerably faster and adaptable to the proposed reconfiguration technique is the one shown in Fig. 10, where each level of nodes represents FBA's Figure 10, where each level of nodes represents FBA's performing in parallel with an anticipated multiplication cycle of

$$[1 + \log_2 n]\, t_s + t_{cc}.$$

LSI implementation

The implementation shown for the 64 × 8 module reveals a number of characteristics suitable for large scale integration. Among them are the repetitive interconnections of simple identical cells and the modularity suitable for expansion and reconfiguration. Below some of the approximate hardware requirements are pointed out.

Approximate number of PINS/MODULE

1. m + n + 2 needed for the multiplicand register
2. m + n + 2 needed as inputs to the second FBA
3. m + n + 2 needed for the product
4. n + 2 needed for the multiplier register
5. three-control pins for reconfiguration.

Approximate number of CELLS/MODULE

The cells are the kinds already discussed: C, S, S', R, K. All are present in a module.

1. m × n/2 cells needed for the CSA stages
2. m + n cells needed for the EACA stage
3. m + n reconfiguration cells
4. 2 (m + n + 2) cells needed for the two FBA's
5. n/2 + 1 cells needed for the ICC.

Approximate number of GATES/CELL*

For cell "C" approximately seven-gates are required
For cell "S", "S'" approximately three-gates are required
For cell "R" approximately two-gates are required
For cell "K" approximately nine-gates are required

The above estimates point out the fact that testing at the individual cell or circuit level (item yet to be examined) becomes a problem, especially when the complexity of the chip is increased, with a parallel decrease in reliability and yield of non-defective chips. However, using the modular approach it is advisable to perform the testing externally on the module and discard the malfunctioning units. This would considerably decrease the amount of logic on a chip, which would otherwise have to be inserted for the testing of the individual circuits. This approach seems to be economically feasible since it is estimated that by 1975 an LSI chip of 100 × 100 mils in size may contain 200 components, at five cents per component, while by 1975 an LSI chip of 300 × 300 mils in size may contain as many as 3,600 components at the cost of about one cent per component. Therefore, miniaturization of LSI chips will discourage the testing on the individual circuit level, while the loss due to the discarding of modules after tesing at the frame level will be negligible.

In view of the above considerations and since the present state-of-art high density MOS circuits are being driven at 10 MHs, implementation of the multiplier modules as the one presented by MOS circuits appears very desirable from a manufacturing viewpoint. A reasonable building block might be a 64 × 64 bit multiplication unit requiring an approximate number of 5000 active elements (field effect transistors). One could also visualize the whole unit incorporated in one or two chips. Where speed is the primary requirement, the unit can be designed using fast bipolar transistors, with an expected five ns delay. Assuming then a 64 × 64 bit module is implemented by bipolar transistors, the execution time could be in the neighborhood of 0.225μs, which when pipelined the maximum number of multiplications per second may be approximately 5×10^9. An MOS array of the same module will perform in an order of magnitude slower than in the bipolar case.

* The above gates are mostly "AND" gates with the "OR" gates not included in the count. They are also 2(m + n) additional gates needed for the reconfiguration scheme and m × n gates for shifting each array.

The pin count also indicates that the current design is within the state-of-art of the MOS technology.

The performance figures given above are educated guesses since the circuit and intermodule delays are dependent on the circuit types, their interconnections, the chip topology, etc. In addition,the design examples described in the previous sections indicate the ease with which the array could be partitioned to fit reasonable unit or chip sizes.

CONCLUSION

Since fast multiplication has become the basis of iterative divisions and square roots in fast computers[6,7] there appears to be a need for cheap array type, LSI realizable multiplication subsystems. This paper reports the design methodology and the detailed implementation of one such structure. Ease of diagnosis and capability of reconfiguration were used as twin requirements in the final design. When the unit is composed of a number of modules and a malfunction is detected in one of them, a method of switching automatically in a spare module was presented. An estimate of the logic circuitry in the hard core (that portion of the unit which must be operating without any faults) during testing is found to be less that 14 percent for a 32×32 module, 9.7 percent for 64×64 module and 4 percent for 128×128 module. Therefore, as the size of the multiplication module-unit increases the relative size of the hard core decreases very rapidly.

To conclude, the cellular array implementation of an asynchronous multiplication unit using mostly non-carry-propagating Carry Save adders was accomplished. The final cell design and the control and the reconfiguring circuitry are quite simple.

A number of additional studies needs to be done in the future. The design of self-diagnosable and repairable functional arrays appear quite feasible and worth considering. The possibility of composite design of a multiplication, division and square rooting unit using techniques presented in this paper could be very useful, particularly if the division and square root algorithms are based on the availability of fast multiplication units such as those discussed in this paper.

ACKNOWLEDGMENTS

The authors would like to thank Mr. Gary Wang of the NASA Electronics Research Center for sharing with them some of his thoughts on the subject. Also Mr. W. R. Adrion, graduate student at the University of Texas at Austin for his constructive suggestions.

REFERENCES

1 C S WALLACE
A suggestion for a fast multiplier
IEEE Trans Prof Group on Electronic Computers Vol 13 No 1 Feb 1964

2 *Methods for high-speed addition and multiplication*
NBS Cir No 591 1953

3 O L MacSORELY
High-speed arithmetic in binary computers
Proc IRE Vol 49 No 1 Jan 1961

4 M LEHMAN
Short-cut multiplication and division in automatic binary digital computers
Proc Inst Elec Eng Paper No 2693M Vol 105B Sept 1958

5 I FLORES
The logic of computer arithmetic
Prentice-Hall Inc 1963

6 D FERRARI
A division method using a parallel multiplier
IEEE Trans Prof Group on Electronic Computers Vol 16 No 2 April 1967

7 S F ANDERSON et al
IBM system model 91: Floating point execution unit
IBM Journal of Research and Development Jan 1967

ENCLOSURE 3

A SURVEY OF RESOURCE ALLOCATION DISCIPLINES IN MULTIPROCESSING SYSTEMS

I. INTRODUCTION

An algorithm and the associated implementation of a mechanism to allocate the available resources to the various competitors (jobs), are two of the most important considerations in the operation of present day data processing systems such as the multiprocessing and Pure-Time Shared Systems. They are geared to assign the resources to a work load such as to satisfy some requirement of "good service" while simultaneously achieving efficient utilization of the system and reduced cost of operation.

A great number of algorithms has been developed for this purpose, each taking into consideration several basic scheduling variables and adopting various disciplines that appear to best aid a particular system to achieve the objectives set forth. A number of procedures have also been devised to measure the perfromance of these various algorithms within their environment of application.

Some of the above mentioned variables most commonly considered are job arrival times, explicit priority, time elapsed in the execution of a job, expected job completion time or the various states the resources themselves are in.

Because of the inherent characteristics, type of service, mode of operation and various degrees of availability required from different types of resources and because of the particular role these resources play in the system, algorithms tend to naturally fall in three basic categories:

a. Algorithms for CPU Allocation

b. Algorithms for Main Memory Allocation

C. Algorithms for I/O Device Allocation.

In this paper some of the conventional and most commonly used algorithms for the allocation of resources of the above three mentioned types will be presented as applied to a multiprocessing or Pure-Time Sharing environment. It must be pointed out that rather than presenting a certain algorithm in its formal form, it is felt that the various disciplines, assumptions and trade-off decisions involved are of more interest in gaining insight as to what these algorithms are trying to achieve and so this presentation is tailored accordingly.

It may be appropriate at this point to clarify some of the terms most frequently encountered in the literature in association with the subject of resource allocation.

1. Multiprocessing is the execution of a job by more than one processor working in parallel.

2. Multiprogramming is the parallel execution of more than one job by a

3. Pure Time Sharing is the simultaneous use of a data processing facility by many independent users or programs.

4. Interrupt is an externally or process generated signal causing a processor to divert attention from the presently executing task to a system routine relevant to handling the cause of this signal.

5. Swapping - the exchange of two programs that are resident in the main memory and main storage respectively, when the latter has just been selected for execution while the former is not going to be executed in the upcoming time slice.

6. Mean-flow-time is the average time a job spends in the system while being executed.

7. Foreground, the collection of small interactive type of jobs, with short run-time requirements in a time sharing environment.

8. Background - the batch processing type jobs in a time sharing environment.

SECTION II CPU Allocation - Scheduling Disciplines

The software mechanisms implementing the algorithms for distributing the processes among the processors in a multiprocessing environment are most commonly referred to as Schedulers and therefore the term Scheduling is synonymous to CPU allocation. When multiplexing jobs in such an environment one actually does nothing else but allocating the resource(s) that are referred to as processing units. Some of the activity taking place in the system during such an operation will be summarized later, after some of the most commonly adopted disciplines in job scheduling are presented.

The disciplines of CPU allocation -- and for almost all types of resources -- fall under two major classifications:

a. Disciplines for static allocation

b. Disciplines for dynamic allocation

A typical problem of the first classification is that of scheduling n jobs on k identical processors in the shortest possible time and generally a partial ordering among the n-jobs is assumed.

Dynamic disciplines making extensive use of queueing theory models are almost exclusively used in present day pure time sharing systems. These algorithms tend to favor shorter jobs over longer jobs and make an attempt to minimize mean-flow-times.

In considering the above two kinds of disciplines one must bear in mind two characteristics, pertinent to the computations involved in both cases:

a. The maximum resource capacity of a job which varies from job to job and also within the job as its execution progresses.

b. The more the resource capacity of a job is reduced, the slower its execution becomes.

II.1 Static Scheduling

The basic static scheduling discipline assumes a partially ordered set of jobs each associated with a certain execution time. If job $n_i > n_j$ then n_i must be completed before n_j can be initiated. Thus, the model can be described as a directed graph, where jobs are the nodes and arcs from n_i to n_j for the above ordering. Each n_i has a weight w_i associated with it representing its execution time. Minimizing the mean completion time within a set of independent jobs is usually the objective. Static scheduling may be tailored for a single processor or a multiprocessor system with appropriate restrictions on the structure of the partial ordering of the jobs. Algorithms involving the multiprocessor type system may be

informally termed as generalized "critical path" algorithms encountered in graph theory.

Since most of the static scheduling algorithms may be considered special cases in the dynamic job scheduling will not be discussed any further in this section.

II.2 Dynamic scheduling

Dynamic scheduling, in general, refers to job sequencing in queueing systems, where jobs arrive at random in some probabilistic fashion. Disciplines associated with this problem focus on total processing time, elapsed processing time or remaining processing time, the state of the resources, externally assigned priorities, dynamically computed priorities and others.

The general purpose queueing system may be viewed as the one shown in Fig. 1., and consisting of: (a) Service facility (processor(s)) (b) A system of queues of unprocessed or semi-processed jobs (c) A source of arrivals (d) A feedback path.

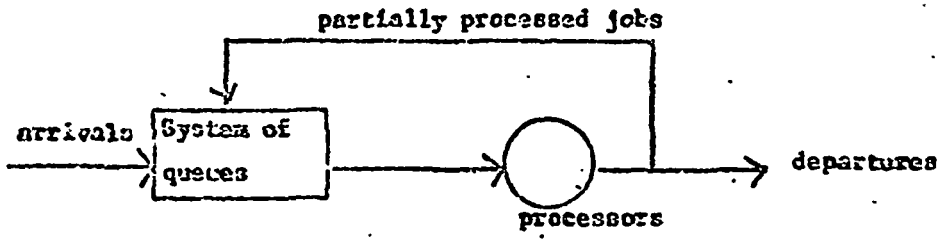


Figure II.1

Two basic assumptions for the scheduling disciplines associated with models of this general type are: (a) If the arrival source is not empty, it keeps generating requests in some probabilistic fashion, (b) The facility will never be idle as long as there is a job ready to be executed.

The decision as to what jobs will have the priority for service, is of utmost importance in classifying the dynamic scheduling algorithms and their associated disciplines. Thus one may clearly distinguish between the following.

Section: Broad classification of dynamic scheduling algorithms.

II.2.1

A. Preemptive/Non-preemptive disciplines

A low priority job is deprived service in favor of a higher priority job which arrived or changed its priority status during execution. The lower priority job is returned to the queue.

B. Return/Restart discipline

Following the action taken in the above discipline, it must be decided whether the interrupted job will be allowed to continue at the point of interruption (return action) or be restarted when it cycles back for service. In the former case provision must be taken to preserve the status of the job at the point of interruption.

C. Disciplines based on source of priority information

According to this discipline a job may be scheduled according to the following criteria.

1. Specific requirements of individual jobs such as running time, I/O requirements, storage, etc.
2. The state of the system as a whole pertaining to availability of resources and current workload.
3. The programming environment such as urgency, special preferences etc.

D. Disciplines based on available information

These disciplines focus primarily on externally assigned priorities for service. It must be pointed out that an overlap of the above mentioned disciplines and associated basic criteria is highly desirable.

Having set the general notions involved in distinguishing between the basic philosophies prevailing in the scheduling algorithms one may further classify these algorithms in a more specialized way as discussed in the following section.

II.3

A more restricted classification of scheduling algorithms.

II.3.1

Scheduling disciplines based on running times.

The underlining characteristic in these disciplines is to minimize waiting time for shorter jobs.

A. Shortest job first discipline (SJF)

The queue is inspected only after the currently running jobs have been served to completion. Jobs requiring the least running time are then entered

into the system. This non-preemptive type of discipline has the obvious advantage of simplicity and at the same time minimizes the mean time of customer waiting in the system. However long running jobs suffer under this type of scheduling.

B. Preemptive shortest job first discipline (PSJF)

The priorities of the jobs for scheduling are re-evaluated at the instant of a new arrival as well as at the completion of a current job, and if the remaining time of the current job is greater than the total time requirement of the incoming one, the former is deprived of service and cycled back to the queue. It is then resumed at the point of interruption when reconsidered for service. Obviously, a preemptive, resume discipline, which has the advantage over the conventional first come first served of SJF previously mentioned in that further alternates the favoritism toward short jobs and further reduces the average waiting time in the system.

An obvious disadvantage is the complexity of the algorithm involved and the "swapping overhead" introduced.

C. Round Robin discipline

First introduced to achieve fast turn around for short jobs in a time sharing environment assuming that running times are not known in advance. A basic time interval -- the quantum -- is an important design parameter. Each job in the system is serviced for a maximum amount of time equal to the quantum. If completed, leaves the system if not is cycled back to the end of the queue and serviced for another quantum later. New arrivals join the end of the queue. Decisions for service are made implicitly on the basis of arrival and running times, after the allocation of one quantum. Swapping is again a disadvantage in this discipline which has motivated several analytical studies to determine system efficiency, waiting times or throughput. In addition, no additional insight about the running time of a job is gained after a quantum of service. This is again a preemptive resume type philosophy.

1. Multiple level feedback (FB) discipline

The quantum parameter is used here also and a new arrival is serviced by immediately receiving as many quanta of service as the job that has received the least amount so far. It preempts all jobs currently executing. This model can be viewed as a multiple level queueing system with the new arrival at the highest priority followed by the job having received the least amount of service quanta etc. Shorter jobs receive better service at the expense of longer ones and the mean flow time is comparable

one wishes to favor short jobs. A disadvantage is that one must keep track of the service already received by each job.

E. Two level feedback (2-FB) discipline

Only a fixed number of quanta of service is allocated to all jobs and if they are not finished they are pushed to the background to receive service only when there are no other jobs in the system. The background queue may be executed according to any other suitable discipline.

F. Declaration of mode discipline

The user is required to state whether his job is of the batching type or interactive type. During the first half of a quantum the interractive jobs are serviced according to the Round Robin discipline and during the second half batching jobs are serviced to completion if possible. As understood there is a great potential of incorporating any of the previously mentioned disciplines in this one.

In conclusion when running time is unknown and fast running jobs can be produced, the ER and FB disciplines are encouraged. It must be also noted that turnaround time, applying any kind of discipline will be considerably effected by the type of jobs that the system may be saturated with at the time.

II.3.2 Scheduling based on state-dependent disciplines.

The desire to reduce swapping overhead costs is the motivation behind this group of disciplines.

A. Queue-oriented RR disciplines

The basic parameter in this discipline is the response time required and hence set, for one RR cycle through all jobs currently in the system.

In the case after completing the first cycle of service, the subsequent cycles are subdivided in quanta according to the number of jobs still requiring service. The discipline is state dependent because the amount of service given to a job in a given cycle depends on the total number of active jobs in the beginning of the cycle.

In another case cycle is determined by the amount of time used to process the foreground queue as well as the background queue. During this interval all processors jobs are given equal quanta and the discipline

initiated. A closer reaction to changes in loading or input activity is achieved. It is implied that the queue is examined at the end of the RR cycle.

C. Input dependent disciplines

This is a variation of the RR discipline in which every time a new arrival is detected the job in service at the moment is allocated an additional quantum. Thus during heavy input activity time allocation for each job is increased and overhead swapping is reduced.

D. Storage allocation based scheduling disciplines

This disciplines will be discussed in connection with the memory allocation algorithms.

II.3.3 Scheduling disciplines based on externally imposed priorities

A. RR disciplines with external priorities

The determination of the quantum size depends on the priorities of the active jobs. Higher priority jobs are allocated larger size quanta than low priority jobs.

The philosophy of this discipline may swing towards the other end of the scale. Each job is assigned an additional priority to the external one, at the time of arrival, based on the external priority and the priority of the currently active jobs. After one quantum of service, the next job to receive service is the one with the lower priority while at the same time its priority is increased by one. It can be seen that in a non-preemptive environment, the priority assigned to a job indicates how many quanta of service are to be allocated to the arriving job before it joins the RR.

B. Other types of dynamic scheduling disciplines

There are numerous other disciplines adopted for scheduling besides the ones presented so far and one would need a considerable amount of time to present them all. A non trivial task in itself may well be when one develops various disciplines based on combinations of the above. In fact this is done many times to accomodate the needs of a particular system. Before closing the subject though it must be pointed out that none of the above disciplines took advantage of the job's waiting time in the input queue. Algorithms exist employing the so called delay dependent discipline according to which jobs enter the queue with zero priority which is dynamically modified proportionally to the job's external priority and its waiting time. A job is then selected according to its "attained priority".

Scheduling algorithms exist according to which the user himself may "buy" his priority. Another important class of algorithms yet, is the one where a defined "cost function" is the basis of selection for service.

II.4 Dynamic schedu[illegible] seen from the point of view of the processor.

So far CPU allocation has been discussed within the context of the system and the restrictions imposed upon it to satisfy certain adopted scheduling disciplines. It is only appropriate to take a brief look at the functions the processor itself has to undertake, in cooperation with its immediate environment, to satisfy the scheduling requirements.

Two essential mechanisms, usually implemented in hardware, cooperate close to the processor. These are the multiplexing and interrupt handling mechanisms. The former serves as a "traffic controller" mainly during the swapping of jobs from main memory to and from main storage. The latter handles either external interrupts or internal when generated by the jobs themselves. The processor before switching must store a state vector. A state vector is a collection of information pertaining to the interrupted job and it basically contains

a. The contents of the program counter
b. The contents of the central registers of the processor
c. The address space of the processor and the contents of every address in it.
d. The state of all I/O devices attached to the processor

It is then quite evident that two basic states a particular job can be in are the running and blocked states. Processors also cooperate with the various system interlock or protection mechanisms but a detailed discussion of this would be beyound the scope of this present action.

New algorithms and disciplines pertaining to memory allocation will be discussed.

III. Memory Management Techniques[1]

III.1 Introduction

The need to break up a program into blocks in order that the program fit the available memory has been recognized from the early days of computers. Early programmers familiar with both the program and the machine were able to accomplish this by dividing their program into blocks which could be overlayed in sequence. The development of higher level programming languages and the corresponding increase in machine complexity, however, have made this process of overlaying sequences increasingly more difficult.

Approaches to the solution of this problem are of two types: static and dynamic. In the static approach, it is assumed that enough is known about the behavior of a program to permit the compiler to generate optimal overlay sequences. In the dynamic approach, memory requirements are allowed to contract and expand as a function of the execution time demands of a program. Increasing demands for computational power by more and more users, and the resultant degree of machine sophistication have made the latter technique the dominant approach for the following system reasons:

(1) The ability to run a partially loaded program.

(2) The ability to vary the amount of memory in use by a given program.

(3) The ability to move a program around in memory during execution.

Programming reasons for the prominence of the dynamic approach include machine independence and program modularity. The former reason can be explained by saying that a change in the coding of the source program. Thus a program can be written independently of the machine in which it is to be run. The latter reason means that a program can be broken up into independent parts which can be compiled separately. A facility of this type permits, for example, different programmers to work on separate components of a single program.

The implementation of the dynamic approach as a solution to the problem of limited memory has to a large extent been made possible by the concept of a virtual memory. In a virtual memory the user has the illusion

that a very large memory is at his disposal, whereas in fact he is actually dealing with a relatively small main memory. Thus there is not a one-to-one correspondence between address space and memory space. Address space refers to the set of names that may be generated by a program as it references information; memory space refers to the set of physical main memory locations in which information items may be stored. This concept relieves the programmer of the responsibility of memory management and assigns it to the computer system. In addition, certain problems arising in multiprogrammed and time shared environments are greatly alleviated.

III.2 Basic System Hardware

The basic computer system to be considered throughout this discussion is shown in Figure III.1. One or more processors have direct access to main memory but not to auxiliary memory. Therefore, information may be processed only when in main memory, and information not being processed must reside in auxiliary memory.

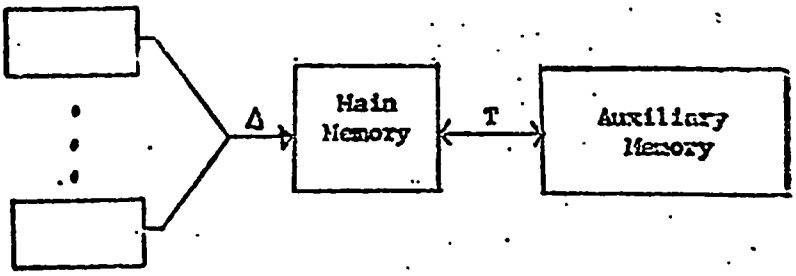


Figure III.1 Basic System Hardware

Memory reference time is measured between the moments at which references to items in memory are initiated by a processor; Δ refers to the average memory reference time. Transport time is the time required to complete a transaction that moves information between the two levels of memory; T refers to the average transport time.

III.3 Definition of Virtual Memory.

Because a virtual memory does not require a one-to-one correspondence between a "name" in the address space and a "location" in the memory space, some means must be provided to translate a virtual address into a physical address. A mechanism for performance of this translation is referred to as the address map or the address translation function and is shown in Figure III.2 (see next page for Figure III.2.)

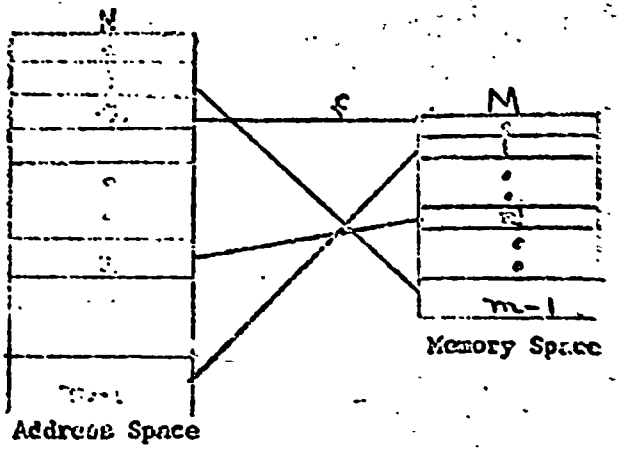


Figure III.2. Mapping from

The address translation function can be considered to be implemented by means of a hardware mechanism placed between a processor and memory as shown below.

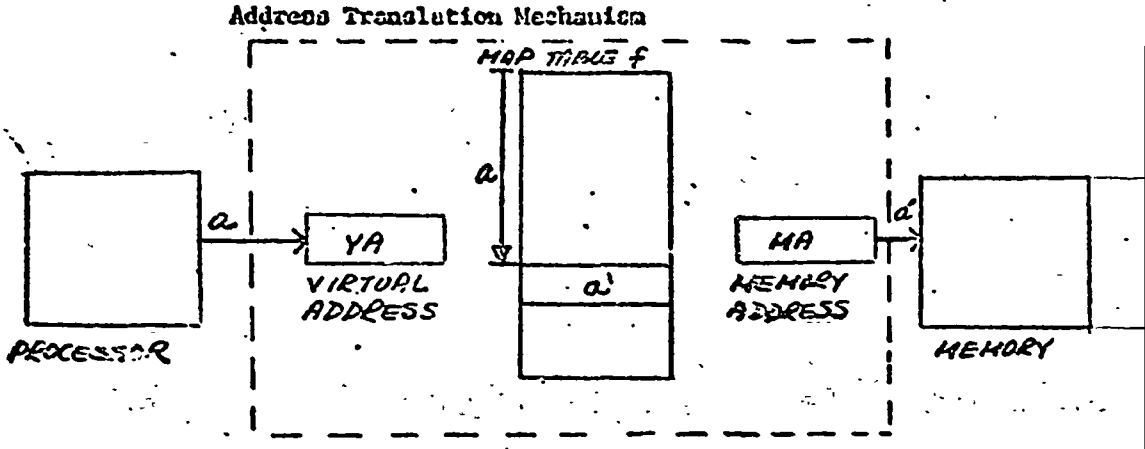


Figure III.3. Implementation of Address Map.

III.4. Implementation of Virtual Memory.

The biggest difficulty with an implementation of the type described above is that the address map would require an amount of memory equal to the available physical memory. The alternative is to group information into blocks of contiguous addresses in address space. Each entry in the address map then refers to a set of addresses with the result that fewer entries are required to contain the same space. The first method used to

generate these blocks is referred to as segmentation .

III.4.1. Segmentation

Segmented address space is regarded as a collection of named segments, each being a linear array of addresses. A segment is a program module which can be referenced and modified as a separate entity. In a segment address space, the programmer references an information item by a two-component address (s,2), in which s is a segment name and w is a word name within s. Figure III.4 shows the essentials of an address translation mechanism that implements segmentation.

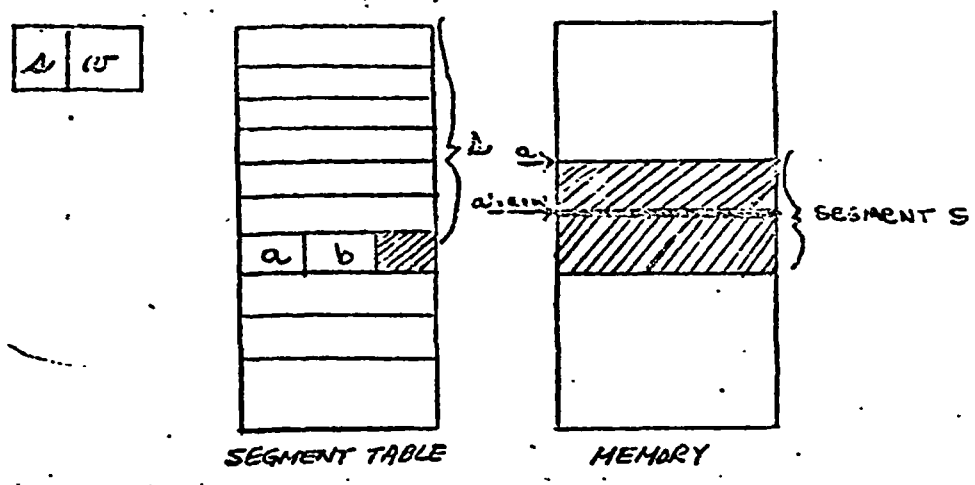


Figure III.4. Address Translation for Segmentation.

The address "a" at which segment s begins is its base address and is found in the s-th entry of the segment table. The physical address is "a+w" which equals (a+w). The number of locations occupied by segment s, "b" is its limit.

Note that in the scheme described here it is assumed that information regarding segment s is in the s-th location in memory. If the segment table can be relocated in memory, however, an additional cycle must be used to add the contents of a relocation register to s in order to access the s-th entry in the segment table.

III.4.2 Paging

Another method of implementing the address map is paging , a method whereby main memory is divided into equal-size blocks known as a page

frames which serve as residences for matching size blocks of virtual addresses known as pages. In segmentation, the elements of each segment occupy contiguous locations in main memory. In paging, all that is required is that the words within a page reside in contiguous locations; the pages themselves may be scattered throughout main memory. A virtual address is equivalent to a pair (p,w) in which p is a page number and w a word number within a page. An implementation of the address map using paging is shown below.

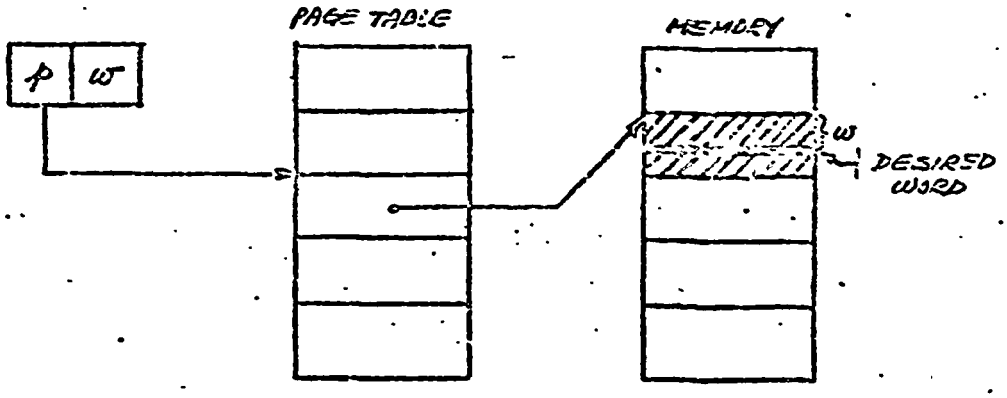


Figure III.5. Implementation of Address Map Using Paging.

III.4.3. Segmentation and Paging.

In order to utilize the best features of both segmentation and paging, a combination of the two approaches can be used. In the situation shown below, entire segments can be brought into memory, but the elements of a segment no longer have to reside in contiguous memory locations. A virtual address now consists of three components: the segment number, s; the page number p; and w, the word number within page p of the desired word.

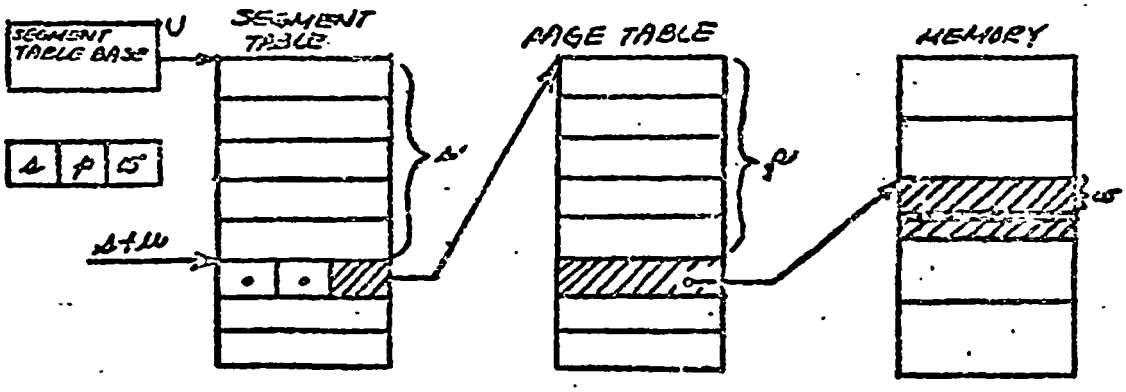


Figure III.6. Implementation of Address Map Using Segmentation and Paging.

III.5. Paging and Segmenting

The preceding paragraphs dealt primarily with the mechanisms used to implement segmentation or paging or both. In addition to these mechanisms, however, policies must be devised to control these implementations. Consider now the placement policy which determines where in memory to place a program block being brought in from auxiliary memory.

III.5.1 Placement Policies

In a paging system the placement policy is straightforward. An incoming page is placed in the first available page frame. If one is not available, the replacement policy will clear out one of the pages currently resident in main memory. In a system using segmentation, the problem is somewhat more difficult. Recall that in pure segmentation a segment must occupy contiguous locations in main memory. The placement policy will find a block of memory of sufficient size to contain an incoming segment. After a period of time, the memory will take on a "checkerboard" appearance as shown below.

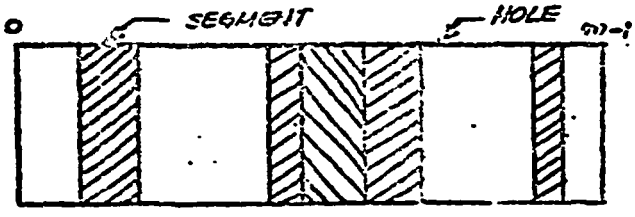


Figure III.7. Checkerboarding of Memory.

The placement algorithm which implements the placement algorithm uses a "hole table" which lists all the holes and their respective sizes. Comments on two algorithms used to place different size segments in memory are given below. Assume there are h holes of sizes $x_1, x_2, \ldots, x_h$, and an insertion request if size s arrives.

1) Best fit. The hole table lists holes in order of increasing size (i.e., $x_1 \leq x_2 \leq \ldots \leq x_h$). Find the smallest i such that $s \leq x_i$.

2) First fit. The hole table lists holes in order of increasing initial address. Find the smallest i such that $s \leq x_i$.

Experiments have shown that the first-fit algorithm is the best of a large class of algorithms. In addition, it has been found that memory size must be at least ten times the average segment size for efficient operation.

III.5.2 Compaction

The techniques described above have assumed a willingness to tolerate a certain amount of overhead to maintain a hole table. An alternative to this requires that at certain points in time all operations must cease in order to permit compaction of all segments in memory in order to eliminate holes between adjacent segments. As shown below, after compaction the low end of memory is filled, new insertions are made at the high end of memory.

Figure III.8. Memory Configuration after Compaction.

It has be found that compaction is less efficient than maintenance of a hole list, for this reason compaction is not often used.

III.6. Storage Fragmentation

Storage fragmentation results from the inability to assign physical locations to virtual addresses that contain information. There are three major types of storage fragmentation. External fragmentation occurs in non-paged memories when checkerboarding becomes so pronounced that every hole is too small to be used. Internal fragmentation occurs in paged memories because storage must be rounded up to an integral number of pages. As a result, storage is wasted because the last page is not fully occupied. Table fragmentation occurs in both paged and non-page memories because storage locations are occupied by the address maps. Consequently, this portion of memory is unavailable for assignment to virtual addresses.

III.6.1. Page Size

Optimum page size is a function of two things: fragmentation and transport time. The smaller the page size, the lower the amount of interval fragmentation. However, as the page size decreases, the number of pages increases; consequently, table fragmentation increases. In addition, page traffic and thus the subsequent total transport time will increase also. If the page size is made large, internal fragmentation increases. Optimum page size lies somewhere between these two extremes.

If fragmentation is the only consideration, optimum page size is of the order of 45 words. If transport time is also considered, the optimum size is then of the order of 1000 words. Since transport time is a direct function of the type of backing store used, optimum page size will depend

on the facilities available at a particular installation. In general, however, moving aim disks should never be used.

III.7. Replacement Algorithms

Placement policies for both paged and non-paged memories were discussed earlier. That is, given that a block of words (a segment or a page) is to be brought into main memory, a portion of memory must be made available for the incoming block. In most instances a vacant area in memory cannot be found to accommodate the incoming block. (In the paragraphs which follow discussion will be confined to a paged memory). The replacement algorithm must determine which page in memory must be released to make room for the incoming page. Generally speaking, if the page with the least likelihood of being reused in the immediate future is retired to auxiliary memory, the best choice has been made. A good measure of performance for a paging policy is page traffic (the number of pages per unit time being moved between memories) because erroneously removed pages add to the traffic of returning pages.

In theory, the way to achieve an optimal replacement algorithm is to preprocess a computational process and to record its history. In practice, however, this procedure is impractical since it would be prohibitively expensive. It does, however, serve as a basis for comparison of other replacement algorithms. Some of these are discussed below.

Random Selection - When a new page is memory is needed, the page to be replaced is selected at random. Although this algorithm is easy to implement, it results in high page traffic since many useful pages are often replaced.

First-In/First-Out (FIFO) Selection - This algorithm replaces the page which has been in memory the longest at the time that a page traffic since it is possible for useful pages to be replaced.

Least Recently Used (LRU) Selection - When a fresh page of memory is needed, the page unreferenced for the longest time is removed. Obviously, mechanisms must be implemented to maintain a history of page usage.

Experiments by Belady[2] have shown that the "ideal" algorithm should possess much of the simplicity of Random or FIFO selection (for efficiency) and some, though not much, accumulation of data on past reference patterns.

III.8. Locality and the Working Set Model

An important factor in the choice of a replacement algorithm is the

principle of locality. This principle states that during short intervals of time, page references will tend to cluster around a small number of pages. One reason for this is that program references may for a time be confined to a subset of the program, a subroutine, for example. In addition, programs often tend to loop for a long time within a small set of pages. The working set of a program, therefore, is expected to contain the "most useful" pages; by the principle of locality the working set changes membership slowly.

If the working set of a program is constructed strictly on a demand basis, then the working set can be expected to grow in size as shown below.

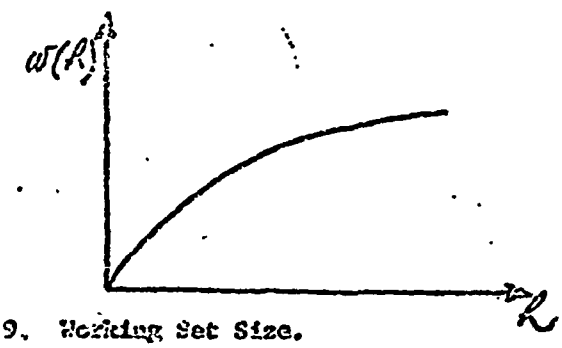


Figure III.9. Working Set Size.

The working set principle is a step in the direction of an optimal page replacement algorithm if the following is observed: A program may run if and only if its working set is in memory, and a page in memory may not be replaced if it is a member of the working set of a running program.

III.9. Multiprogramming and Thrashing

Paging algorithms in a multiprogrammed memory can be implemented in one of two ways. In the local implementation memory is partitioned into blocks of fixed size. A program initiating a page fault may replace a page only from within its own block. In the global implementation, a program requiring an additional page may replace a page from any location in the working memory.

The working set principle is defined in the context of a local policy. Multiprogramming under a global policy is susceptible to thrashing, a collapse of performance that may occur when memory is overcommitted. Briefly, thrashing will occur when a program is brought into memory and attempts to generate its working set at the expense of programs already in memory whose working sets have been built up over a period of time. As a consequence, programs will spend a great deal of time regenerating their working sets with the result that processor utilization will drop drastically.

The working set model under a local policy insures that a program will not be brought into memory unless enough uncommitted memory is available to permit the program to build up its set of useful pages.

III.10. Conclusions

Many different subjects have been presented in this discussion. Although these topics have been discussed individually, collectively they present a unified approach whose goal is the more efficient use of a computer system. The idea which is common to these techniques and serves to bring them together is the concept of a virtual memory. A virtual memory allows a user to consider that all the resources of a computer system are dedicated to him, whereas in fact they are allotted to him on a piecemeal basis. Current reports suggest that actual implementations using combinations of these techniques do indeed accomplish their stated objectives.

IV. File Memory Scheduling Strategies

IV.1 Introduction

In a multiprogrammed, time-sharing computing system the secondary storage is the heart of the system. It is here that input files are stored, output files are compiled as the result of processes, and "scratch work" required by operating processes may be temporarily placed. Consequently, there is a considerable burden on secondary storage, which is often referred to as file memory, resulting from constant traffic in and out of main memory. It is essential then that every element of the computing system interfacing with file memory should do so smoothly and efficiently.

By the nature of file memory, secondary storage is necessarily large capacity and low cost. These requirements consistent with the state-of-the-art technology dictate the use of rotating mechanical devices such as drums and disks. These devices are inherently sequential and slow in comparison with core memory. This disparity makes it necessary to schedule file requests so as to minimize mechanical access time and to maximize the throughput of the system.

In this section, we attempt to summarize Denning's [2] and Coffman's [1] work, delineating some scheduling policies, deriving simple mathematical models, and analyzing some of the results thus obtained.

IV.2 Description of Basic Goals, Environment, and Policy

The computing system can be visualized as two basic entities, main memory and file memory. The fact is that there are considerable traffic between slow, secondary, storage, and fast core storage. System performance clearly depends on the efficiency with which these file memory operations take place. The goals of our scheduling policies must achieve maximum throughput, and yet not discriminate unjustly against any particular request. In addition, it must be reasonably inexpensive to implement.

We assume that there is a basic unit of storage and transmission of data, called page. Programs are made of pages. Core memory is divided into pages; and information is stored page by page on secondary storage devices; it follows that the unit of information transfer is the page. We further assume that requests for file memory use are on a single page basis, and that a process which desires to transfer several pages will have to generate several requests, one for each page.

Because the file system does not operate rapidly enough to satisfy immediately the total system demand, there is bound to be a queue. The simple and basic policy to serve a queue is a first-come-first-served (FCFS) policy. However, in the light of the sequential nature and the slow speed of secondary storage, it is natural to inquire whether the file system can take advantage of opportunities hidden within the queue. For example, it might be possible to serve the last request in the queue during the positioning delay of the first request. The following sections will propose two improved scheduling strategies and compare results with those arrived at by FCFS policy.

IV.3 The Drum System

The first task is to analyze a fixed head storage device. The analysis deals with a drum system, but is sufficiently general for extension to other devices of similar structure. The analysis is approximate, intended only to obtain the expected waiting times so that advantages of scheduling can be illustrated.

IV.3.1 The Drum Organization

As shown in Figure 1, the drum is divided along the axis of rotation into equal regions called fields. Each field is consisted of a group of tracks, and each track has its own head for reading and writing. Along the angular coordinate, the drum surface is divided into equal regions called sectors. The surface area enclosed in the intersection of a field and a sector is a page. For each read/write head, there are two amplifiers, one for read, and the other for write. There is a physical limitation in selection delay involved in switching the states of the heads. Thus, suppose sector K is presently reading and if there is a request in the queue for reading a page on sector k+1, the request is serviced. But if the request is write, it can not be serviced until sector k+2, since the write amplifier can not be switched in soon enough.

We further assume that there is a drum allocation policy which guarantees that there is at least one free page per sector. When a sector is full or over a prespecified level, the drum allocator will selectively relocate one or more pages in that sector to a lower level of storage (say a disk or a tape). If this is done, a write request may commence at once, and it is highly probably that a sequence of write requests generated by a single process will be stored consectively.

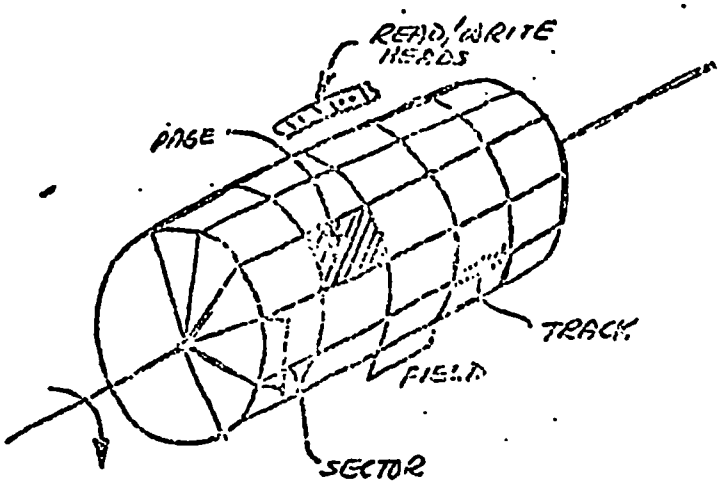


Figure 1, Organization of Drum

IV.3.2 The Shortest-Access-Time-First (SATF) Policy

With respect to the drum structure a timing diagram is developed as shown in Figure 2. The various quantities of interest are displayed and defined as follows:

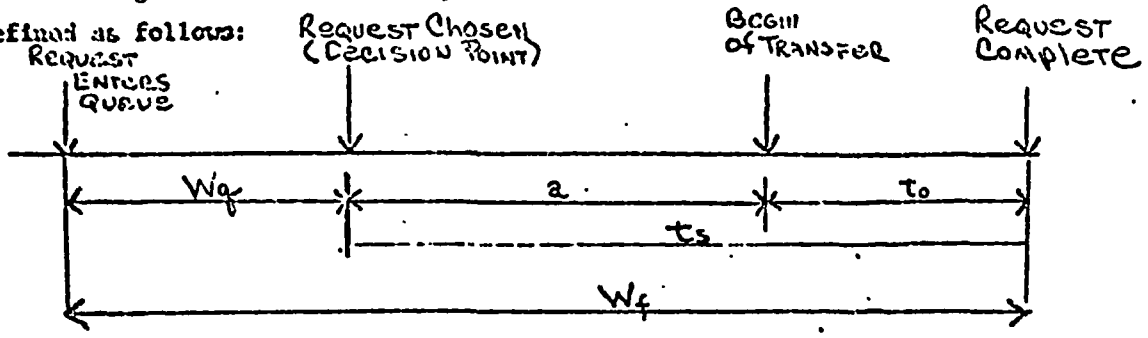


Figure 2, Waiting Time Parameters for Drum

t_o: The transfer time, the time required to read or write a page from or to file memory.

a: the access time, the mechanical positioning delay.

t_s: service time, defined as $t_s = a + t_o$

W_q: the waiting time in queue.

W_f: the total time a request spends in the file system.

To carry out the SATF strategy in coming file requests are sorted by starting address into groups of queues each of which corresponds to a sector on the drum. Two and more requests are in the same queue if and only if they are for the same sector. This is consistent with the hardware since drum is addressable by sectors. As implied by the name, the SATF policy can be simply stated as: choose as next for service that request which requires the short access time.

To help visualize the strategy we have taken a different but equivalent view of the drum system by assuming the drum stationary and the read/write heads as rotating at constant speed around the drum. This is illustrated in Figure 3.

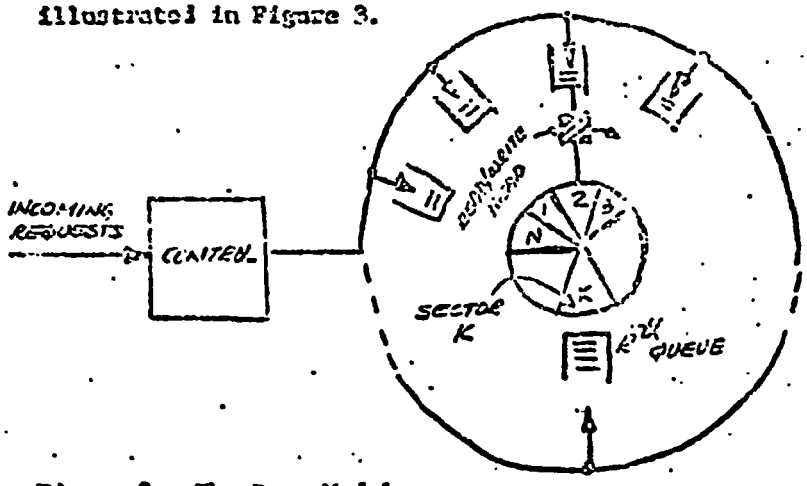


Figure 3, The Drum Model

IV.3.3 The Mathematical Model

First let us define some terms as follows:

E [a] : expected access time.

E [to] : expected service time.

Q : throughput factor to be the number of requests serviced per Unit time.

U : Utilization factor to be the ratio of page transfer time to page transger time plus access time.

T : drum revolution time.

N : the number of sectors.

n : The number of requests in queue

p. : the probability a request is a read.

We now proceed to derive the key parameters, $E[a]$, $E[t_s]$, W_f, U, and Q, for first FCFS policy and then SATF policy.

FCFS Policy

Let the access time per request be a random variable which can assume one of the N values equally likely. $0, \frac{T}{N}, \frac{2T}{N}, \ldots, \frac{N-1}{N}T$

the expected access time is

$$E[a] = \sum_{k=0}^{N-1} \left(\frac{kT}{N}\right) pr\left[a = \frac{kT}{N}\right]$$

$$= \frac{T}{N^2}\left[1+2+3+\ldots+(N-1)\right]$$

$$= \frac{T}{N^2} \cdot \frac{1}{2}(N-1)N$$

$$= \frac{N-1}{N} \cdot \frac{T}{2} \approx \frac{T}{2} \quad ----(1)$$

Page transfer time is constant, $\frac{T}{N}$ Hence, the utilization factor is

$$U = \frac{\frac{T}{N}}{\frac{T}{N} + \frac{N-1}{N} \cdot \frac{T}{2}}$$

$$= \frac{2}{N+1} \qquad (2)$$

$$E[t_s] = E[a] + \frac{T}{N} \qquad (3)$$

AND

$$Q = \frac{1}{E[t_s]} \qquad (4)$$

Assuming that the number of requests, n in the queue stays close to its expectation, n E n . it is known[3] that in a FCFS queue, the expected wait is

$$W_f \approx E[n] \cdot E[t_s]$$

$$\approx n E[t_s]$$

$$\approx n\left(\frac{T}{2} + \frac{T}{N}\right)$$

$$= nT\left(\frac{N+2}{2N}\right) \qquad (5)$$

SATF Policy

The expected minimum access time with n requests in the queue is

$$E[a] = \left[\frac{T}{N+1}\left(1-\frac{1}{2N}\right)^{n+1} + \frac{T(1-p)}{N} + \frac{T(1-p)}{N+1}\left(1-\frac{3}{2N}\right)^{n+1}\right] p^n +$$

$$\frac{T}{N} p^{n+1}(1-p^n)\left[\left(\frac{1}{p} - \frac{1}{N}\right)^n - \left(1-\frac{1}{N}\right)^n\right] \quad ----(6)$$

The derivation can be seen in Appendix 2 of Denning's[1] paper.

$$U = \frac{T/N}{E[a] + T/N} = \frac{T}{N \cdot E[a] + T} \quad - - - - - - (7)$$

To obtain an estimate of W_f under the SATF policy, we reason as follows. If there are n requests waiting, then n/N of them are waiting for a given sector. So when a request enters the file memory, hence a sector queue, (see figure 3), it expects to wait T/2 for its sector to come into position, plus n/N additional drum revolutions before receiving service, and finally T/N time for its own page transfer. Hence

$$W_f = \frac{T}{2} + \frac{nT}{N} + \frac{T}{N}$$

$$= T\left(\frac{1}{2} + \frac{n+1}{N}\right) \quad - - - - - - (8)$$

IV.3.4 Example

To dramatize the effects of scheduling, we give a numerical example. A typical high-speed drum has a revoultion time of 16.7 ms (3600 rpm). Typically, there are 4096 words written around the drums circumference within a field. We use a page size of 64 words yielding 64 sectors around the drum. Typically, the probability of a read will be greater than that of a write, so the choice of 0.7 for p is not unreasonable. Assuming the expected number of requests in queue is 10, we have the following as parameters:

T = 16.7 ms
N = 64 sectors
to = T/N = 0.26 ms
p = 0.7
n = 10

Substituting these parameters into equations (1) through (8); we have the following table.

POLICY	MILLISECONDS			U %	Q REQ/SEC.
	E[a]	E[t_A]	W_f		
FCFS	8.33	8.59	85.9	3	116
SATF	0.23	0.49	11.2	53	2040

The relative improvement is

$$\frac{W_f(FCFS)}{W_f(SATF)} \approx \frac{\frac{1}{2}rT\left(\frac{1}{2}+\frac{1}{N}\right)}{T\left(\frac{1}{2}+\frac{N+1}{4}\right)} = \frac{n(N+2)}{N+2(n+1)}$$

For the set of given parameters, this is 7.65 which means that incoming requests see a drum that is 765 per cent faster! The utilization factor increased by a factor of almost 20.

IV.4 The Disk System

We now turn our attention to movable head storage devices. Typical of equipment in this class is the movable arm disk. Again the model used in the analysis has sufficient generality that the results can be extended for use with other movable head devices of similar structures.

IV.4.1 The Disk Organization

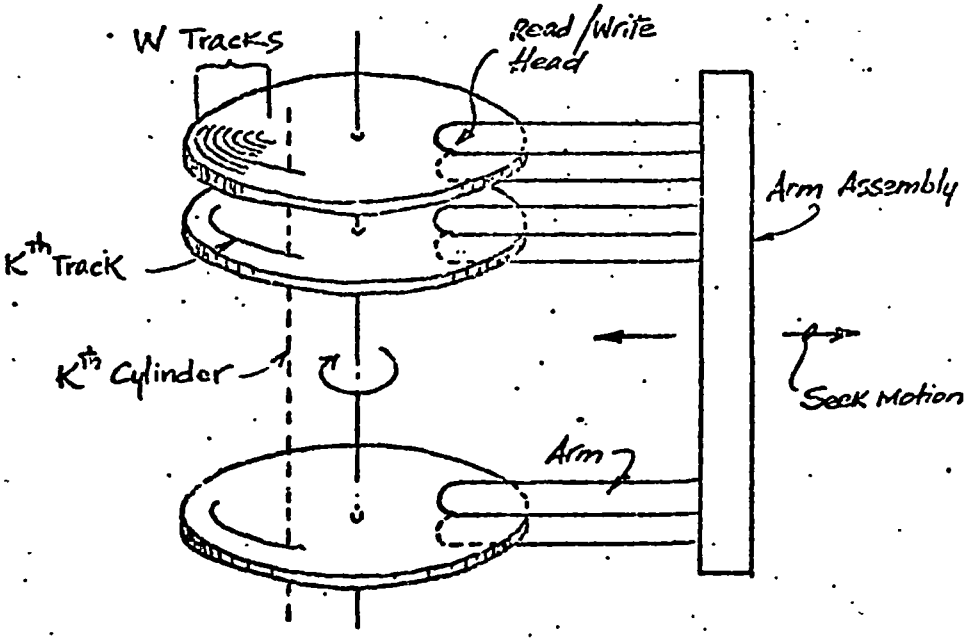


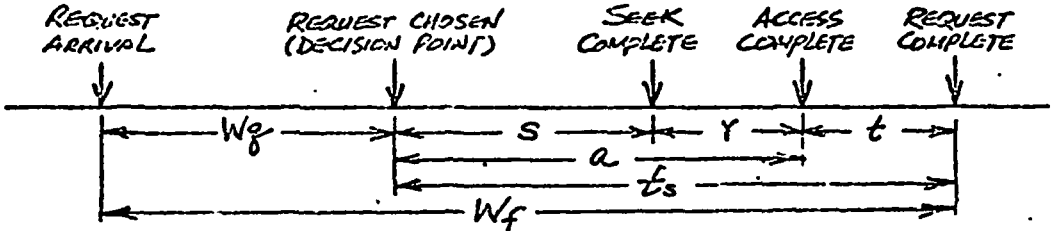
Figure 4, Organization of Disk

The disk system is illustrated in Figure 4. There are a series of disks of like radii equally spaced along a common axis. Both surfaces of each disk are capable of storing information along concentric tracks which are situated near the outer edge. One can imagine that the k^{th} track of each disk is situated on the surface of a cylinder; thus, if the disk is W track wide, there are W concentric cylinders on which to store information.

There is an assembly of movable arms, equipped with read/write heads. There is one set of heads on each side of a disk, sufficient to read or write one track, so each time a new track is requested, it is necessary to reposition the arms. The operation of positioning the arms is known as a Seek. The mechanism for doing this is hydraulic, and is therefore inherently slow. In general seeking one track requires a time S_{min} while a seek the full width of the disk, requires a time S_{max}. Typically S_{min} is in the order of 15ms and S_{max} is about 300 ms. There is little difference between a seek of k tracks and a seek of (k+1) tracks. The bottleneck is getting the arms moving in the first place. Once the arms have been positioned any disk surface on the cylinder is addressable.

IV.4.2 The Seek Strategies

Similar to the drum system, the disk waiting time parameters are shown in Figure 5. The only difference is that added seek time. After the request enters the queue, it waits a time W_q until it is chosen for service. Once it has been chosen, a seek time S elapses. At the completion of the seek, an additional rotation delay passes



until the desired sector has come into position. Only then does transmission begin, and it is assumed to last t seconds. The access time is

$$a = s + r$$

and the service time is

$$t_s = a + t = s + r + t$$

It is reasonable to further assume that once the arms are positioned at a given cylinder, every request for that cylinder is served before another seek is initiated. The reason for this is apparent when one considers the magnitude of the minimum seek time, S_{min}.

Scheduling a disk system may be considered to have two levels. Waiting requests are sorted into W groups, one for each cylinder, in a manner similar to the drum system. The upper level of scheduling is to decide at which cylinder to position the arm. Once the arms are positioned, the lower level SATF scheduling could be used. However, the probability that more than one request is waiting for the same cylinder under normal load conditions,

is quite small. So nothing is to be gained by scheduling policies other than FCFS policy within the cylinder queue. Thus the expected rotation delay is T/2.

Once all requests for a given cylinder are satisfied, the arms are moved away to a new cylinder. Similar to the drum system, it would seem reasonable that a Shortest-Seek-Time-First (SSTF) policy should be used. However, this policy is unsatisfactory since it tends to discriminate against requests for the inner and outer cylinders. To see this, suppose the arm is at the k^{th} track, where K W/2 as indicated in Figure 6. Because requests are assumed to fall uniformly

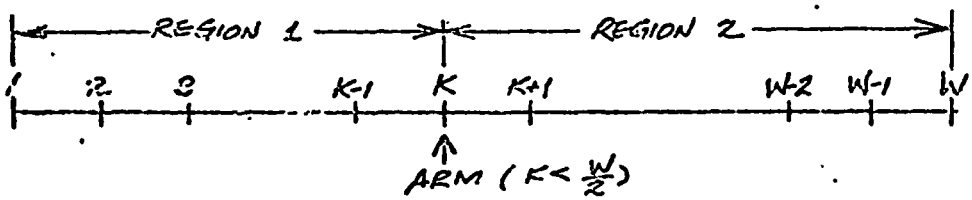


FIG. 6 ALTERNATIVES FOR ARM MOTION

for any of the W tracks, there are more requests in Region 2 than in Region 1, then the likelihood is greater that the arm moves into Region 2. Under heavy usage, the arms tend to yo-yo back and forth around the center of the track region. Further complication occurs in case of a "tie", that is, when the minimum arm movement is the same in either direction.

An alternative strategy which insures reasonable service to all requests is the SCAN policy. The SCAN strategy sweeps the arms back and forth between the inner and outer tracks (or cylinders) servicing any requests in the intervening queues as the arms pass by. One can think of operating the disk arms as a "shuttle bus" between the inner and outer tracks, stopping the arms at any cylinder for waiting requests.

IV.4.3 The Mathematics Model

Although the SSTF policy is unsatisfactory due to its discriminatory effect, it is still possible that the waiting time of a request may still be less than its normal wait under the FCFS policy.

In Appendix 3 of Denning's paper[1], the expected minimum seek time under SSTF policy is derived as

$$E[S] = \left(\frac{N-1}{N}\right)^n \left[\frac{1}{2} + S_{min} + \frac{S_{max} - S_{min}}{2(n+1)} \cdot \left(1 + \frac{1}{n+2}\left(\frac{N}{N-1}\right)^{n+1}\right)\right] \text{---- (9)}$$

The unscheduled seek time under FCFS policy is obtained by setting n=1 in equation (9)

$$E[s] = \frac{W-1}{W}\left[S_{min} + \frac{1}{2} + \frac{S_{max} - S_{min}}{4} \cdot \left(1 + \frac{1}{3}\left(\frac{W}{W-1}\right)^2\right)\right] \quad ------(10)$$

the expected access time for both SSTF and FCFS policies is

$$E[a] = E[s] + \frac{T}{2} \quad ----------(11)$$

the utilization factor is

$$U = \frac{T}{E[a] + T} \quad ----------(12)$$

T is used as the transfer time because it is assumed that each transfer is a full track. The throughput factor Q has the same definition as the drum system

$$Q = \frac{1}{E[t_s]} \quad ----------(13)$$

Due to uncertainties of the SSTF policy, the total wait time W_f under this policy will be estimated. For FCFS policy, however, we have,

$$W_f \approx n E[t_s] \quad ----------(14)$$

SCAN Policy

First assume that there are enough cylinders and few requests that the probability of finding more than one request for a given cylinder is much less than the probability of finding just one request. This is equivalent to assuming that the total number of waiting requests fall randomly within the track region, and that when it arrives the arm is at some random position. The expected distance between the arriving request and the arm is $\frac{W}{3}$. There is probability $\frac{1}{2}$ that the arm travels toward the request for $\frac{W}{3}$ distance, and probability $\frac{1}{2}$ that the arm moves away to the edge and back towards the request, in which case the arm must travel $3(\frac{W}{3}) = W$ distance to reach the request. Hence the expected distance the arm moves for reading the request is

$$\frac{1}{2}\left(\frac{W}{3}\right) + \frac{1}{2}3\left(\frac{W}{3}\right) = \frac{2}{3}W \quad ----------(15)$$

If there are n requests, they divide the track region in (n+1) regions. The seek time for one such distance is

$$E[s] = S_{min} + \frac{S_{max} - S_{min}}{n+1} \quad ----------(16)$$

and the access time is the seek time plus $\frac{T}{2}$ for rotation delay

$$E[a] = E[s] + \frac{T}{2}$$

$$E[t_s] = E[a] + T$$

$$= E[s] + (3/2)T$$

$$= \frac{S_{max} - S_{min}}{n+1} + S_{min} + \frac{3}{2}T \quad \text{-----(17)}$$

Since the time to cross W tracks in nE[t_s] . The time W_f a single request must wait is from equation (14)

$$W_f = \frac{2}{3} n E[t_s]$$

$$= \frac{2}{3} n \left[S_{min} + \frac{S_{max} - S_{min}}{n+1}\right] + nT \quad \text{----(18)}$$

IV.4.4 Example:

To demonstrate the effects of disk scheduling, we present an example using the following parameters:

S_{min} = minimum seek time = 150ms

S_{max} = maximum seek time = 300ms

W = number of tracks = 30

T = disk revolution time = 60ms

n = number of requests in queue = 10

Substituting these parameters into equations from (9) to (18), the following table was obtained.

POLICY	MILLISECONDS				u %	Q/sec.
	E[S]	E[a]	E[t_s]	W_f		
FCFS	195	225	285	2850	21.0	3.5
SSTF	113	143	203	—	29.5	4.9
SCAN	164	194	254	1700	23.6	3.9

When 10 cylinders requested the expected utilization increase with SSTF over FCFS is $\frac{29.5}{21.0} = 1.40$

The gain under SSTF is not-too-impressive 40%. It is the large minimum seek time S_{min} that imparts efficiency. The gain of SCAN over FCFS is

$$\frac{23.6}{21.0} = 1.12$$

which has a 12% increase in utilization. The improvement in waiting time

$$\frac{2050}{1700} = 1.68$$

is quite impressive. It means that each request sees a disk that is 68% faster.

Hence we conclude that SCAN is the preferred disk scheduling policy.

REFERENCES

Part II

1. E. G. Coffman, R. R. Muntz. Models for Pure Time Sharing Disciplines for Resource Allocation. JACM.

2. E. G. Coffman, L. Kleinrock. Computer Scheduling Methods and Their Countermeasures. SJCC, 1968.

3. S. Hellerman. Some Principles of Time Sharing Scheduler Strategies. IBM SYS JOURNAL #2, 1969.

4. S. H. Lampron. A Scheduling Philosophy for Multiprocessing Systems. CACM, Vol. 11, May, 1968.

Part III

1. Denning, P. J. "Virtual Memory", Princeton University Technical Report Number 81, January, 1970.

2. Belady, L. A. "A Study of Replacement Algorithms for Virtual Storage Computers", IBM Sys. J., 5 (2), 1966.

3. Wilkes, M. V. Time Sharing Computer Systems, American Elsevier, 1968.

Part IV

1. E. G. Coffman, Jr. "Analysis of A Drum Input/Output Queue Under Scheduled Operation in a Paged Computer System", JACM, Vol. 16, No. 1, Jan. 1969.

2. P. J. Denning "Effects of Scheduling of File Memory Operations", Proceeding, SJCC, April, 1967.

ENCLOSURE 4

A HEURISTIC OPTIMIZATION ALGORITHM FOR ARRANGING AN N PAGE OPEN STRING

PROBLEM DEFINITION

During the normal operation of a large computing system blocks of data are frequently transferred between different levels of the system's memory hierarchy which may include storage media ranging from ultra-fast bipolar 'cache' buffers, to fast core, to bulk core, to drum, to disk, to tape. Commonly the blocks of data are fixed size and are called pages. When collections of these pages are stored in a member of the hierarchy which can be characterized as possessing only one degree of access freedom, performance of the memory hierarchy depends strongly upon the arrangement of the page string. Using graph theory termology the optimization problem can be stated as follows:

Given a weighted, directed complete graph of N nodes, determine an open string of N nodes such as to minimize

$$\sum_{i=1}^{N} \sum_{j=1}^{N} (f_{ij}d_{ij} + f_{ji}d_{ji})$$

where $f_{ij}d_{ij}$ denotes the weight of the edge from node i to node j

f_{ij} denotes the density of the path from node it to node j

d_{ij} denotes the length of the path from node i to node j (paths between consecutive nodes are equal)

Such an optimization problem has numerous sources. For example consider a disk memory unit possessing one head per surface. If the time required for the intertrack head movement is much greater than the time required for one disk revolution, then the unit can be characterized as possessing only one degree of access freedom. In the corresponding graph model all pages on a given track are considered to exist at a single node.

Path length (d_{ij}) from node i (track i) to node j (track j) is assumed to correspond linearly to the required head movement time between track i and track j. Also, the separation distance between consecutive tracks is assumed to be constant and the cost of self-loops assumed to be zero ($d_{ij} = 0$ if $i = j$). Path densities (f_{ij}) correspond to the expected frequency (number) of head movements from track i to track j. Therefore, the objective function to be minimized equals the summation of all possible intertrack head movements multiplied by their respective frequency.

As a second example consider a magnetic tape unit possessing bi-directional read/write capabilities. With fixed size pages the path length (d_{ij}) is assumed to correspond linearly to the time required to move the tape from page i to page j. The cost of self-loops is assumed to be zero since the tape unit possesses bi-directional read/write capabilities. Path densities (f_{ij}) correspond to the expected frequency (number) of jumps from page i to page j. Therefore, the objective function to be minimized equals the summation of all possible inter page jump distances multiplied by their respective frequency.

Other important applications involve areas such as one-dimensional back board wiring, manufacturing processes (or assembly line) and with appropriate restrictions certain classical operations research problems such as the Chinese Postman problem (fixed terminal nodes and no feedback loop terms in the objective function).

Certainly such an important problem would have already been solved if an exact solution existed. It does. By considering all N! permutations an optimum arrangement can be selected. However, the cost of an exhaustive search of N! permutations increases factorially with N. For example a 10 node graph possesses 3,628,800 permutations while 11 node graph possesses 39,916,800 permutations! Therefore, a heuristic optimization algorithm which offers a cost per node ratio which does not increase factorially is highly desirable.

OBJECTIVE FUNCTION

The problem's Objective Function must measure the total access time for the N pages in terms of the distance traveled to achieve the given number of interpage jumps. Consider a page string consisting of 5 pages. See Fig. 1. For such a page string, the Objective Function to be minimized equals:

$$OF = f_{12}d_{12} + f_{21}d_{21} + f_{13}d_{13} + f_{31}d_{31} + f_{14}d_{14} + f_{41}d_{41} + f_{15}d_{15} + f_{51}d_{51}$$
$$+ f_{23}d_{23} + f_{32}d_{32} + f_{24}d_{24} + f_{42}d_{42} + f_{25}d_{25} + f_{52}d_{52} + f_{34}d_{34} + f_{43}d_{43}$$
$$+ f_{35}d_{35} + f_{53}d_{53} + f_{45}d_{45} + f_{54}d_{54}$$

where d_{ij} equals the distance between page i and page j

and f_{ij} equals the number (frequency) of jumps between page i and page j.

In general for N pages the Objective Function equals

$$\text{Objective Function} = \sum_{i=1}^{N} \sum_{j=1}^{N} (f_{ij}d_{ij} + f_{ji}d_{ji})$$

CONSTRAINTS

The problem possesses unusual constraints which prevent applying the classical optimization methods of Operations Research. All the terms of the Objective Function consist of the produce to a fixed jump frequency (f_{ij}) and a jump distance variable (d_{ij}) which is limited to a fixed set of integers. Therefore the problem must be classified as an integer programming problem.

First, consider the fixed set of jump frequencies (f_{ij}). This set completely specifies the expected frequency of jumping between any two pages in the string. These frequencies may be expressed either as integers corresponding to a number of jump or as fractions corresponding to normalized percentages. In this paper, the examples will employ only integers in order to simplify the presented calculations.

A convenient means of representing the set of page jump frequencies employs a connectivity matrix. Consider an example corresponding to N=5 pages. See Fig. 2. The resulting matrix is called the Jump Frequency Matrix (JFM). Since it completely specifies the interpage jump frequencies, it provides all the required input data for the developed optimization algorithm. Note that when i=j, then $f_{ij}=0$. This condition corresponds to a self-loop or a jump from a page to itself. The cost of such a jump is considered to

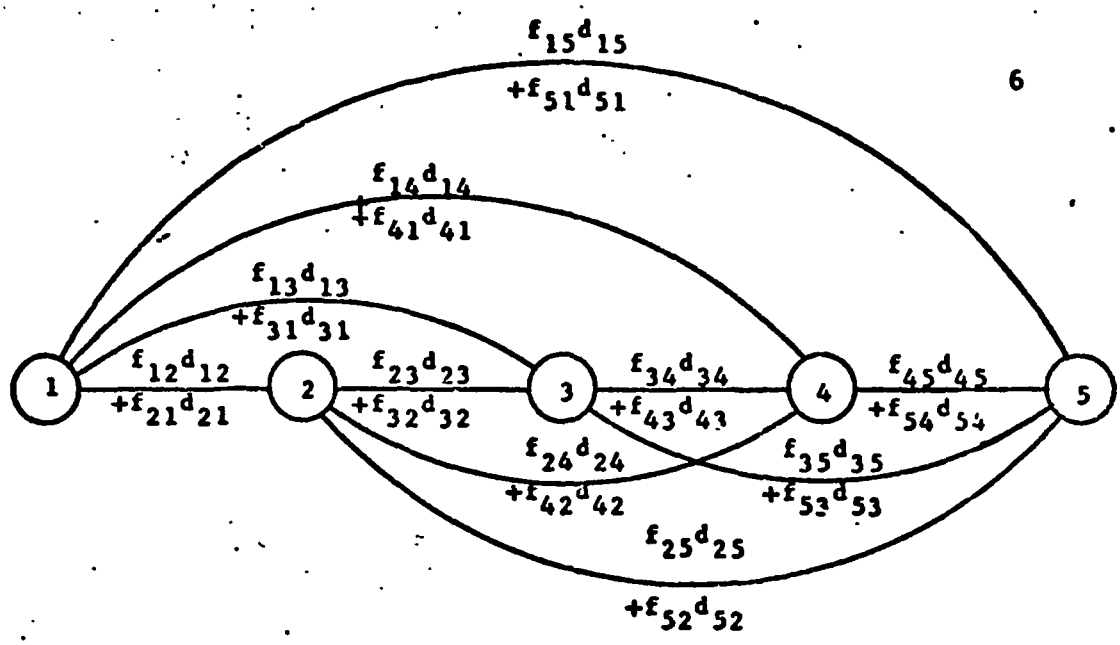


FIGURE 1 OBJECTIVE FUNCTION TERMS (5 PAGE STRING)

PAGE j

PAGE i	1	2	3	4	5
1	0	f_{12}	f_{13}	f_{14}	f_{15}
2	f_{21}	0	f_{23}	f_{24}	f_{25}
3	f_{31}	f_{32}	0	f_{34}	f_{35}
4	f_{41}	f_{42}	f_{43}	0	f_{45}
5	f_{51}	f_{52}	f_{53}	f_{54}	0

FIGURE 2 JUMP FREQUENCY MATRIX (JFM)

be zero since the distance traveled is zero. The tape unit always stops at one boundary of the target page (Inter Record Gap). Since the tape unit operation is bi-directional, the page can be accessed immediately. (It might be noted that actually a tape acceleration delay exists. But since this delay is common to all tape movements and is assumed to be very small with respect to the time required to transverse a page, it will be ignored.)

Now consider the most stringent constraint which must be imposed upon the jump distance variables, d_{ij}. Again consider a 5-page string as shown in Fig. 3. From the linear graph representation it can be observed that for the 5-page string, the jump distance variables must take on only the following set of integers:

$$\text{Jump Distance Integer Set} = \{1,1,1,1,2,2,2,3,3,4\}$$

In general, for an N page string the Jump Distance Integer Set consists of the integers $1,2,\ldots,N-1$ where the integer K is included $N-K$ times.

This set of integers can conveniently be entered into a matrix as shown in Fig. 4. Such a matrix is called the Jump Distance Matrix (JDM).

SUPERIMPOSED TABLEAU

Superimposing the Jump Frequency Matrix upon the Jump Distance Matrix and performing a term wise multiplication generates the Objective Function represented on the same matrix as used for the JFM and the JDM. This representation of the Objective Function will be called the Superimposed Tableau (Fig. 5) and will normally be shown without the jump distance values

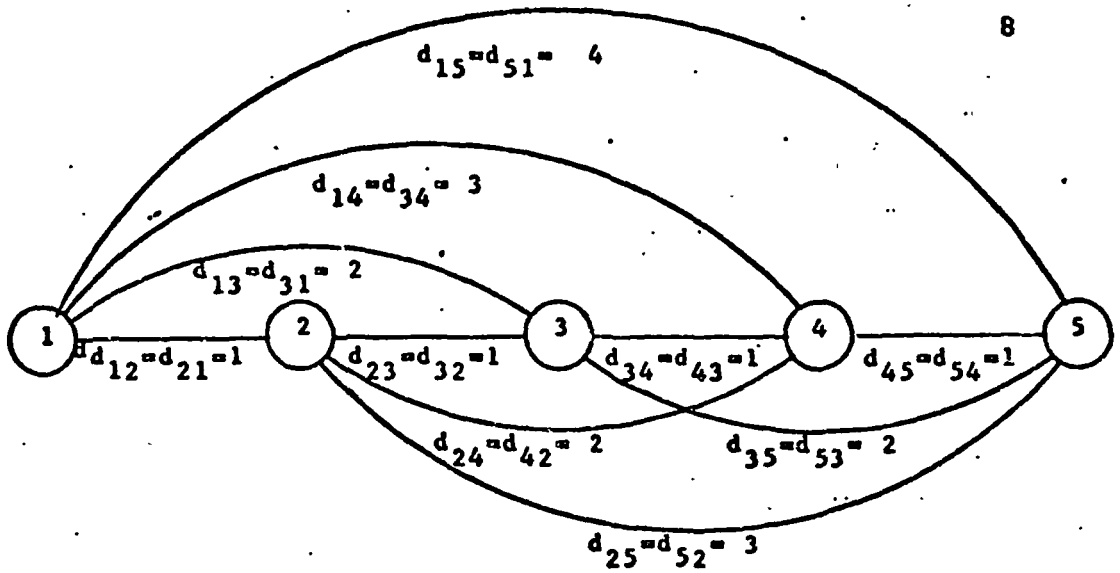


FIGURE 3 JUMP DISTANCES

PAGE 1

PAGE j

	1	2	3	4	55
1	0	1	2	3	4
2	1	0	1	2	3
3	2	1	0	1	2
4	3	2	1	0	1
5	4	3	2	1	0

FIGURE 4 JUMP DISTANCE MATRIX (JDH)

since they remain fixed with respect to the Tableau position. (The jump distances might be shown as corner boxes.) Only the jump frequencies move upon the tableau as the Optimization Algorithm is performed.

PROPERTIES OF SUPERIMPOSED TABLEAU

The Superimposed Tableau representation for the Objective Function possesses several unusual properties. These include:

- representation of a feasible solution
- representation of a page string corresponding to each Objective Function
- grouping of Objective Function terms with respect to jump distance (Level)
- a convenient tableau for evaluating the Objective Function
- a convenient tableau for performing page interchange operations upon the page string.

FEASIBLE SOLUTION

Entering the given initial jump frequencies into the Superimposed Tableau establishes an initial feasible solution. Note that the rows and columns of the Tableau possess both a set of Working Labels corresponding to the given page identification and a set of Tableau Labels which do not change. If all operations (page interchanges) performed upon the tableau preserve feasiblity, then the Superimposed Tableau will always represent a feasible solution to the problem. This feasible solution corresponds to a page string consecutively labeled $1,2,\ldots,N$ using the Superimposed

Tableau Labels. Transformation of the labels to the working page labels requires a one-to-one mapping of the page Working Labels to their corresponding Superimposed Tableau abels. This information is directly available from the Tableau as shown in Fig. 6.

DEFINITION OF PAGE STRING CORRESPONDING TO EACH TABLEAU

Note that row one of the Superimposed Tableau dictates an unique string of N pages. It specifies the distance from page one to each of the remaining N-1 pages. Since the page string consists of consecutive pages, each being one unit of distance farther from page one, this specifies an unique ordering of the pages as shown in Fig. 7. This built-in specification of a page string corresponding to a given tableau circumvents the difficult problem of generating a feasible page string corresponding to a given set of f_{ij} d_{ij} assignments.

LEVELS

The Superimposed Tableau automatically groups the terms of the Objective Function with respect to the jump distances. Each diagonal of the tableau partitions the objective function with respect to the jump distance. See Fig. 8. Each subset will be called a 'level' and the index for each level will correspond to the subset's jump distance. The main diagonal terms will be called the Level 0 subset while the first diagonal above or below the main diagonal will be called Level 1 subset. In general, the Objective Function equals

$$\text{Objective Function} = \{\text{Level 1}\} + \{\text{Level 2}\} + \ldots + \{\text{Level N} - 1\}$$

for a problem of N pages.

		1	2	3	4	5	TABLEAU LABELS
		WL_1	WL_2	WL_3	WL_4	WL_5	WORKING LABELS
1	WL_1	0	$1f_{12}$	$2f_{13}$	$3f_{14}$	$4f_{15}$	
2	WL_2	$1f_{21}$	0	$1f_{23}$	$2f_{24}$	$3f_{25}$	
3	WL_3	$2f_{31}$	$1f_{32}$	0	$1f_{34}$	$2f_{35}$	
4	WL_4	$3f_{41}$	$2f_{42}$	$1f_{43}$	0	$1f_{45}$	
5	WL_5	$4f_{51}$	$3f_{52}$	$2f_{53}$	$1f_{54}$	0	

FIGURE 5 THE SUPERIMPOSED TABLEAU

1	2	3	4	5	TABLEAU LABELS
WL_1	WL_2	WL_3	WL_4	WL_5	WORKING LABELS

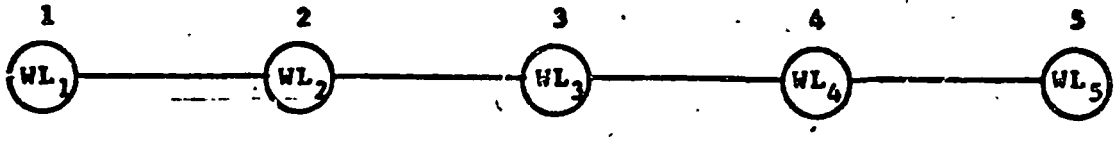


FIGURE 6 FEASIBLE SOLUTION STRING CORRESPONDING TO ONE-TO-ONE MAPPING BETWEEN TABLEAU LABELS AND WORKING LABELS

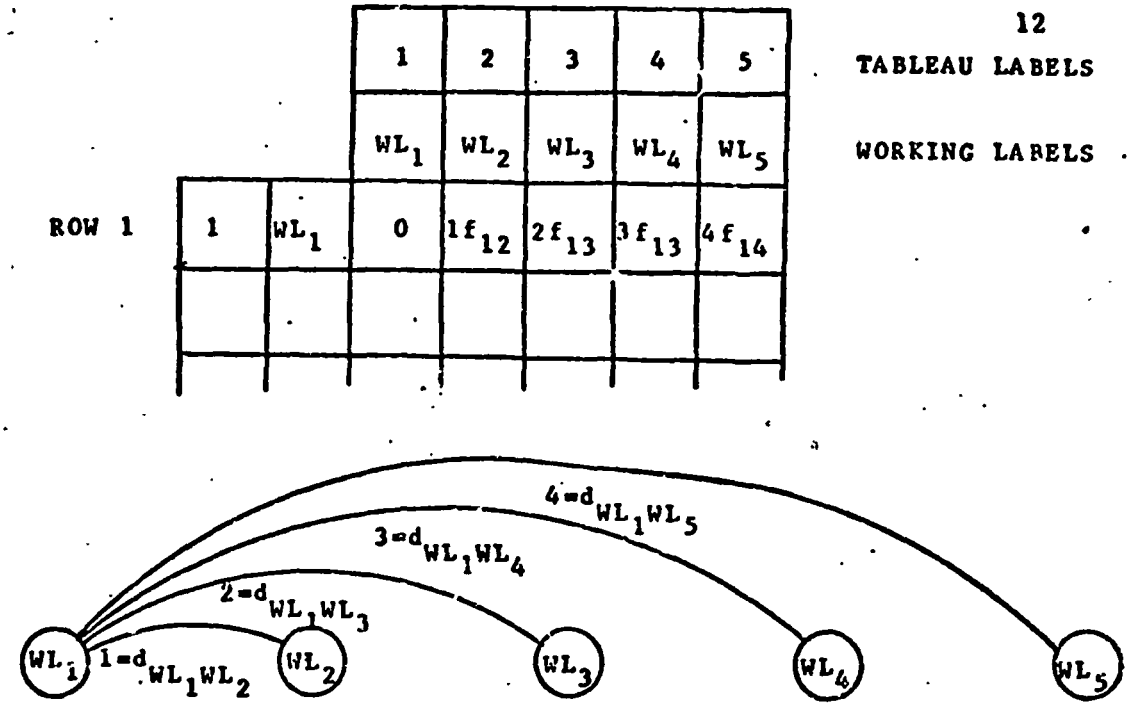


FIGURE 7 FEASIBLE PAGE STRING DEFINITION

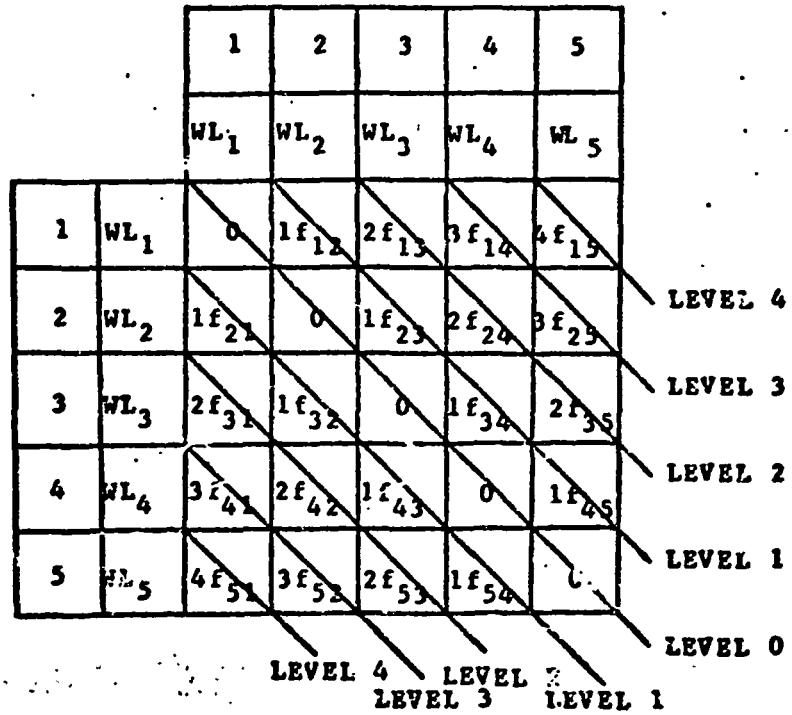


FIGURE 8 PARTITIONING OF OBJECTIVE FUNCTION INTO LEVELS

OBJECTIVE FUNCTION EVALUATION

The concept of partitioning the Objective Function into Levels or subsets with respect to jump distance simplifies evaluation. To evaluate the Objective Function, the terms of each Level are summed and then multiplied by their respective Level index. The summation of these products equals the value of the Objective Function. In general,

$$\text{Objective Function} = \sum_{k=1}^{N-1} \sum_{i=1}^{N} \sum_{j=1}^{N} (f_{ij}d_{ij} + f_{ji}d_{ji}/d_{ij}=d_{ji}=k)$$

Each level cost will be identified by the level index. For example, the Level 1 cost equals $\sum_{i=1}^{N} \sum_{j=1}^{N} (f_{ij}d_{ij} + f_{ji}d_{ji}/d_{ij}=d_{ji}=1)$. See Fig. 9. Therefore evaluation of the Objective Function consists of only a series of additions plus integer multiplications.

PAGE INTERCHANGE RULE FOR SUPERIMPOSED TABLEAU

The Optimization Algorithm to be presented introduces different combinations (changes in the tableau's basis) by interchanging pages. To always maintain feasibility of the tableau (correspondence to a realizable page string) a special operation rule was developed for modifying the tableau as pages are interchanged.

To interchange page k and page kk (for example, page 2 and page 4 in Fig. 10) the following operations are required:

1. Interchange column term k and column term kk for row 1 (f_{14} and f_{12} in Fig. 10). Repeat for rows 2,...,N.

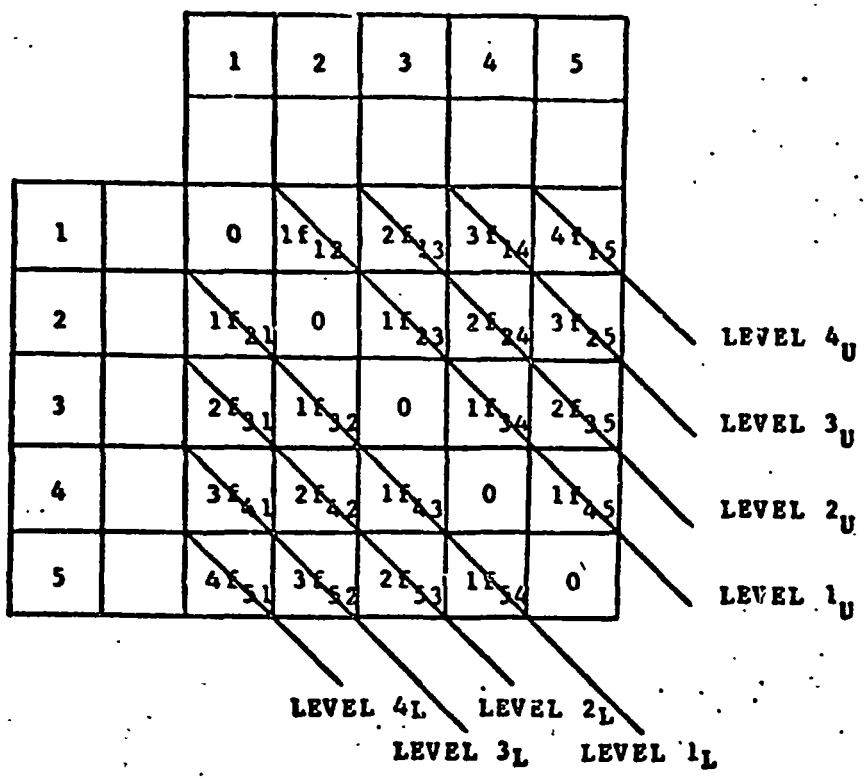


FIGURE 9 OBJECTIVE FUNCTION EVALUATION BY LEVEL COST SUMMATION

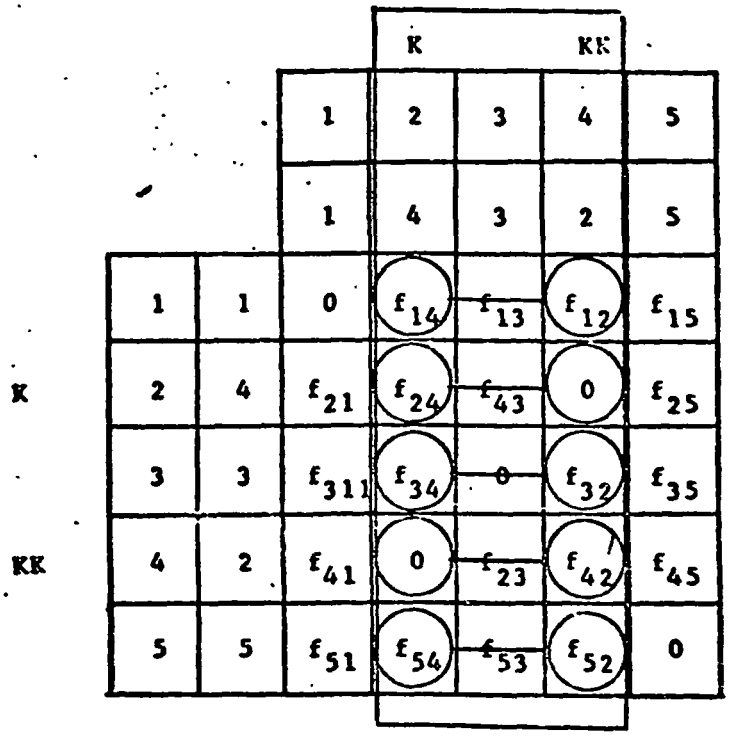


FIGURE 10 TABLEAU AFTER PAGE INTERCHANGE RULE STEP 1 (and step 3)

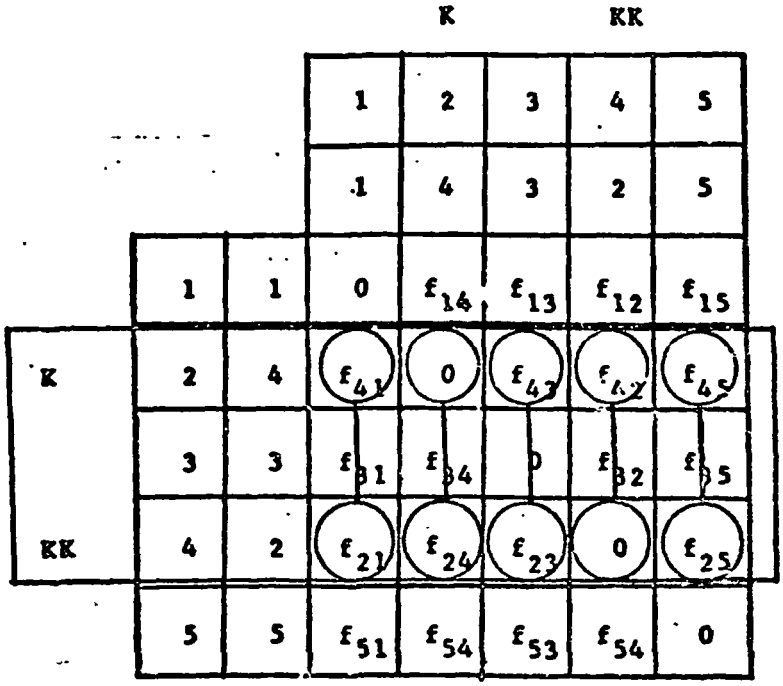


FIGURE 11 TABLEAU AFTER PAGE INTERCHANGE RULE PERFORMED

2. Interchange row term k and row term kk for column 1 (f_{41} and f_{21} in Fig. 11). Repeat for columns 2,...,N.

3. Update the Superimposed Tableau working page labels.

LOCAL TRIANGLES

The developed Optimization Algorithm tests 'local triangles' to identify "promising" page interchanges. A local triangle consists of a given element in the Superimposed Tableau plus two equal distance, lower level elements. The three elements form an isosceles triangle with the single element of higher level being called the vertex element of the local triangle.

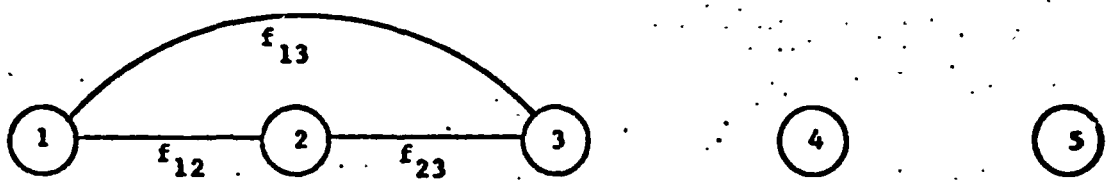
When referenced to the corresponding page string the vertex element represents the element which might be moved to a lower level by interchanging two pages in the page string. Examples of local triangles and their relationship to the page string are delineated in Fig. 12. For the local triangle (f_{12},f_{13},f_{23}) the vertex element (f_{13}) could be moved to a lower level only by interchanging either pages 1 and 2 or pages 2 and 3. By interchanging these pages, one element of a lower level must in turn be moved to a higher level.

The local triangles may be classified according to whether the local cost can be reduced.

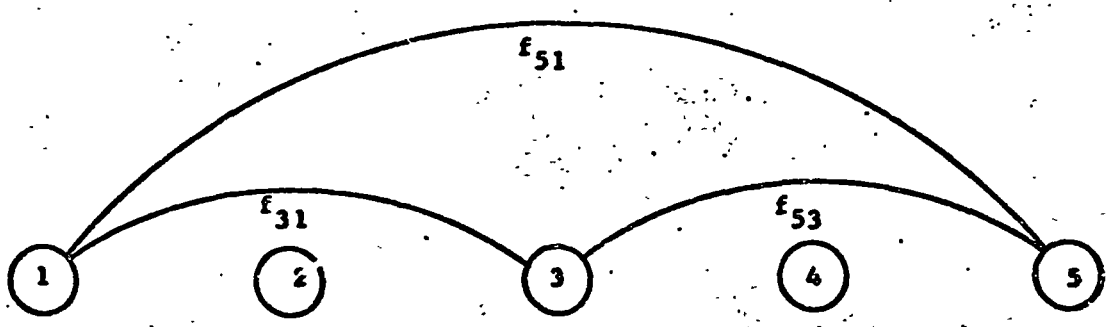
Local Optimum Triangle - Exists whenever the cost of the local triangle cannot be reduced by a page interchange.

Local Gain Triangle - Exists whenever the cost of the local triangle can be reduced by a page interchange.

		1	2	3	4	5
		WL_1	WL_2	WL_3	WL_4	WL_5
1	WL_1	0	$1f_{12}$	$2f_{13}$	$3f_{14}$	$4f_{15}$
2	WL_2	$1f_{21}$	0	$1f_{23}$	$2f_{24}$	$3f_{25}$
3	WL_3	$2f_{31}$	$1f_{32}$	0	$1f_{34}$	$2f_{35}$
4	WL_4	$3f_{41}$	$2f_{42}$	$1f_{43}$	0	$1f_{45}$
5	WL_5	$4f_{51}$	$3f_{52}$	$2f_{53}$	$1f_{54}$	0



LOCAL TRIANGLE f_{12}, f_{23}, f_{13} GRAPH



LOCAL TRIANGLE f_{13}, f_{35}, f_{51}

FIGURE 12 LOCAL TRIANGLES

For a page interchange to show a net gain there must exist at least one local gain triangle. This follows since the net gain of a page interchange is calculated from the summation of a set of local triangle costs as shown in the next section.

In general, for an N page string, the number of local triangles of size D(Dequals the jump distance between pages) can be expressed as:

$$\sum_{L=1}^{N-D} 2(N+1-D-L)$$

For example, in the N=5 page string there exists 18 local triangles of size D=1.

The developed algorithm tests the tableau for triangles of size D=1,...,N-1. By summing the number of local triangles of each size, the total number of local triangles per tableau equals

$$\sum_{D=1}^{N-1} \sum_{L=1}^{N-D} 2(N+1-D-L)$$

EVALUATION OF PAGE INTERCHANGES

By testing local triangles the developed optimization algorithm detects promising page interchanges. But the net effect upon the objective function is unknown since the local triangle involves only the relationship between three pages. The following evaluation rule provides the gain (positive or negative) of interchanging page k and page kk (tableau labels, k less than kk).

For reference consult Fig. 13.

1. Upper Triangular Matrix: Row Pairs + Column Pairs

$$UTM = (kk-k)\left[\sum_{R=1}^{k-1}(T_{R,kk} - T_{R,k}) + \sum_{c=kk+1}^{N}(T_{k,c} - T_{kk,c})\right]$$

2. Lower Triangular Matrix: Column Pairs + Row Pairs

$$LTM = (kk-k)\left[\sum_{c=1}^{k-1}(T_{kk,c} - T_{k,c}) + \sum_{R=kk+1}^{N}(T_{R,k} - T_{R,kk})\right]$$

3. Intersection Terms: Column Pairs + Row Pairs

$$IT = \left[(kk-c) - (c-k)\right]\left\{\sum_{c=k+1}^{kk-1}(T_{kk,c} - T_{k,c}) + \sum_{R=k+1}^{kk-1}(T_{R,k} - T_{R,kk})\right\}$$

4. Net Objective Function Change

$$= UTM + LTM + IT$$

If the calculated change is positive then the value of the objective function can be reduced by interchanging pages k and kk.

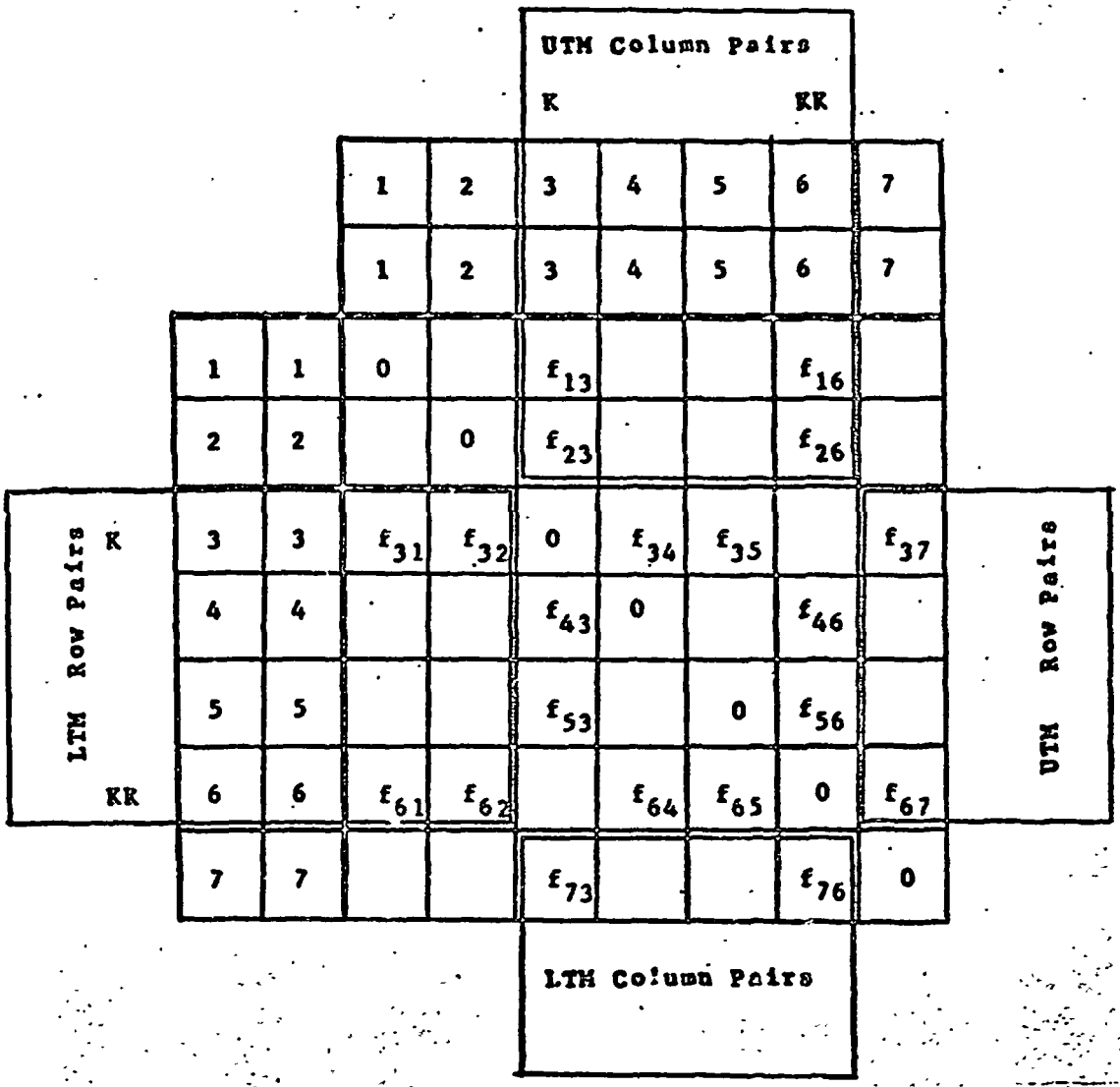


FIGURE 13 EVALUATION OF PAGE INTERCHANGES

OPTIMIZATION ALGORITHM

The preceding definitions and developments provide the necessary background for the algorithm which is detailed in the flowchart shown in Fig. 14. Since no proof of the optimality of the results is presented, the algorithm must be considered heuristic.

The major steps of the algorithm will be briefly discussed.

Page String Ordering.

Initially the page string is ordered with respect to the weight of each page's jump frequency cut set. For a given page K the weight of the jump frequency cut set is calculated by summing the column K entries and the row K entries in the Superimposed Tableau. The page string is then ordered with respect to the weights by interchanging the required pages. This simple procedure provides two advantages.

First, those pages having a greater number of jumps are initially close together. Therefore, the expected number of page interchanges required to achieve an optimum arrangement should be reduced.

Secondly, the expected behavior of the objective function is smoother as the algorithm is executed. Since the algorithm employs only page interchanges which reduce the objective function, the objective function monotonically converges to a local optimum. If the pages were randomly arranged, the expected rate of this convergence would vary wildly. As a result, certain permutation subsets may not be explored.

Starting Page String Arrangement.

The algorithm employs a set of N arrangements for N independent iterations. These N starting arrangements play a role similar to the N basis

vectors of a vector space. From these N starting arrangements the algorithm explores the set of N! permutations.

The N starting arrangements are generated by shifting end-around the ordered page string. Since the algorithm employs only page interchanges, the end-around shift generates a maximumly permuted page string which requires N-1 page interchanges to undo. Restated, an end-around shift operation requires N-1 page interchanges to implement. Note that for a given arrangement N end-around shifts returns the arrangement to its starting arrangement. Therefore, the set of N starting arrangements consists of the given ordered arrangement plus N - 1 arrangements generated by end-around shifts.

Local Gain Triangle Identification.

The key to the algorithm's advantage over an exhaustive search of all N! permutations is the identification of the Local Gain Triangle. As previously explained for a page interchange to reduce the objective function there must exist at least one Local Gain Triangle involving the page pair. Therefore, by testing all local triangles 'promising' page interchanges can be detected.

In order to minimize the number of required page interchanges, testing starts with the maximum size local triangle (D=N-1) and proceeds to the minimum size local triangle (D=1). For example, consider two pages of distance K apart whose interchange would reduce the objective function value. The algorithm would detect the condition with a Local Gain Triangle of size K and one page interchange would result. Conversely, if the algorithm proceeded from minimum to maximum size local triangles, several Local

Gain Triangles of size less than K might be detected and each would result in a page interchange.

Since the Local Gain Triangle identification procedure starts with the maximum size local triangle, it follows that the location of the triangle's vertex elements proceeds from the highest level, Level N - 1, toward Level 2, the smallest level capable of containing a local triangle of size 1. Also, proceeding from the higher to the lower levels allows the algorithm to delay generating strongly connected substrings. Once generated, such substrings are not usually broken and commonly produce undesirable local optimum arrangements.

Evaluation of Interchanges Corresponding to Local Gain Triangles.

If a Local Gain Triangle of size K is detected in a given level, testing is interrupted after completing that level. All page interchanges indicated by the Local Gain Triangles are evaluated with respect to the objective function. Those interchanges possessing positive gains are pushed into the Interchange Stack. Should no positive gain interchanges be found, testing for Local Gain Triangles of size K resumes at the next level.

Page Interchanging.

The page interchange denoted by the top entry of the Interchange Stack is interchanged and a trace generated by pushing the information into the Backtrack Stack. This stack will be employed to backtrack to the proper fork after a local optimum arrangement has been achieved. Note that at each level of testing for Local Gain Triangles multiple positive gain interchanges may be detected. Each of these alternatives correspond to a branch

of the iteration tree. When more than one alternative is detected, a tree fork is established.

After interchanging the page pair the algorithm completely restarts the Local Gain Triangle identification procedure since a Local Gain Triangle of any size and at any level may now exist.

Local Optimum.

Whenever the Superimposed Tableau no longer contains any Local Gain Triangles, then a local optimum arrangement of the page string has been achieved. Such a condition represents a terminal node of the iteration tree. The algorithm must next test if backtracking is required to complete exploring all positive gain interchanges.

Backtracking.

If after achieving a local optimum, entries remain in the Interchange Stack, then the Superimposed Tableau must be rearranged using the trace available in the Backtrack Stack. The Superimposed Tableau is rearranged to correspond to the string arrangement which existed at the tree fork possessing the unexplored page interchange indicated by the top entry of the Interchange Stack.

Iteration Stop.

Whenever a local optimum is reached and the Interchange Stack is empty, then all branches of the iteration tree have been explored. The minimum local optimum detected during the iteration is selected as the Iteration Optimum.

Heuristic Optimum Arrangement.

After completing the N iterations the local optimum having the minimum objective function value is selected as the Heuristic Optimum Page String Arrangement.

FEASIBILITY THEOREM

Every stage of the Superimposed Tableau represents a feasible solution.

PROOF

It is given that the Optimization Algorithm initially enters a feasible solution into the Superimposed Tableau. All operations performed upon the tableau employ only the Page Interchange Rule which preserves feasibility. Therefore, every stage of the Superimposed Tableau represents a feasible solution.

Lower Bound Formula

To develop a lower bound for the number of local triangles which the algorithm examines per iteration, assume that every arrangement of the given page string is optimum.

Since the number of local triangles of size D is

$$(N-D)[(N-D+1]/2$$

then the total number of local triangles of all sizes (D=1,...,N-1) equals

$$\sum_{D=1}^{N-1} (N-D)[(N-D)+1]/2$$

which can be expanded as follows

$$= [N-(N-1)][N-(N-1)+1] + [N-(N-2)][N-(N-2)+1] + \ldots + [N-(1)][N-(1)+1]$$

$$= 1(2) + 2(3) + \ldots + (N-1)\,N$$

$$= \sum_{D=1}^{N-1} i(i+1) = \sum_{i=1}^{N-1} i^2 + \sum_{i=1}^{N-1} i$$

$$= (N^3 - N)/3$$

$$\approx N^3/3 \quad \text{for } N >>> 1$$

Results

The algorithm has been applied to various problems based upon random jump frequencies. The following figures report results for a complete graph of 10 nodes which was incrementally reduced to a 5 node graph.

Figure 15 shows results of run timings performed on the algorithm as a special page string was optimized. This page string possessed the special property that any arrangement was an optimum arrangement. Figure 15a defines the 10 node graph and the corresponding node weights. Figure 15b shows the print out of the optimization results. Note that no page interchanges were required. Figure 15c plots the run time as a function of the number of nodes. A loose correlation to the Lower Bound Formula is shown. For example, consider the run times for N = 10 (.476 seconds) and N = 7 (.138 seconds).

$$RT_{10} \approx \left(\frac{10}{7}\right)^4 RT_7 = \left(\frac{10}{7}\right)^4 (.138) = .574 \approx .476$$

In these preliminary evaluation tests two run options were employed. Option A allowed all pages to be interchanged and the N starting arrangements were generated by end-around shifts of the initial arrangement which was ordered with respect to the node weights. Option B employed the same initial starting arrangement (ordered with respect to the node weights) for every iteration, but fixed a different node each iteration. By holding one node fixed the number of interchange alternatives was reduced. This difference is shown in Figures 16b and 16c. Option A

generated 29 local optimum arrangements while Option B generated only 15. Note that the run times were 6.132 and 2.006 seconds respectively while the final objective function values differed by only 1%.

Figure 16d shows the run time as a function of the number of nodes. The run time difference between Option A and Option B is displayed. However, it should be noted that such run times are dependent upon the jump frequencies of the particular graph being optimized and not just a function of the number of nodes. As a result it is not possible to draw any general conclusions except that the run time of Option A will be greater than the run time of Option B while the final objective function values are very similar.

RUN OPTIONS

T12 = FIXES PAGE POSITION COFRESPONDING TO BASIS NUMBER
T13 = SHIFTS ARRANGEMENT END-AROUND
T14 = ORDERS INPUT ARRANGEMENT WITH RESPECT TO NODE WEIGHT

WEIGHTS	180.00	180.00	180.00	180.00	180.00	180.00	180.00	180.00	180.00	180.00
LABELS	1	2	3	4	5	6	7	8	9	10
	0.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
	10.00	0.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
	10.00	10.00	0.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
	10.00	10.00	10.00	0.00	10.00	10.00	10.00	10.00	10.00	10.00
	10.00	10.00	10.00	10.00	0.00	10.00	10.00	10.00	10.00	10.00
	10.00	10.00	10.00	10.00	10.00	0.00	10.00	10.00	10.00	10.00
	10.00	10.00	10.00	10.00	10.00	10.00	0.00	10.00	10.00	10.00
	10.00	10.00	10.00	10.00	10.00	10.00	10.00	0.00	10.00	10.00
	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	0.00	10.00
	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	0.00

INITIAL TABLEAU FOR PAGE STRING W.
(EVERY ARRANGEMENT IS OPTIMUM)

FIGURE 15a

OPTIMIZATION RESULTS

ITEM.	OBJECTIVE FUNCTION INITIAL	OBJECTIVE FUNCTION MINIMUM	PAGE CHANGES		GAIN TRIANGLES TESTED	GAIN TRIANGLES DETECTED	DET-TEST RATIO	TREE FORKS	MINIMUM ARRANGEMENT
9	3300.00	3300.00	0	0	330	90	.27273	0	1 2 3 4 5 6 7 8 9 10
8	3300.00	3300.00	0	0	330	90	.27273	0	2 3 4 5 6 7 8 9 10 1
7	3300.00	3300.00	0	0	330	90	.27273	0	3 4 5 6 7 8 9 10 1 2
6	3300.00	3300.00	0	0	330	90	.27273	0	4 5 6 7 8 9 10 1 2 3
5	3300.00	3300.00	0	0	330	90	.27273	0	5 6 7 8 9 10 1 2 3 4
4	3300.00	3300.00	0	0	330	90	.27273	0	6 7 8 9 10 1 2 3 4 5
3	3300.00	3300.00	0	0	330	90	.27273	0	7 8 9 10 1 2 3 4 5 6
2	3300.00	3300.00	0	0	330	90	.27273	0	8 9 10 1 2 3 4 5 6 7
1	3300.00	3300.00	0	0	330	90	.27273	0	9 10 1 2 3 4 5 6 7 8
0	3300.00	3300.00	0	0	330	90	.27273	0	10 1 2 3 4 5 6 7 8 9
TOTALS---------- 10		3300.00		0	3300	900	.27273	9	RUN TIME = .477 SECONDS

OPTION A RESULTS
(PAGE STRING W)

FIGURE 15b

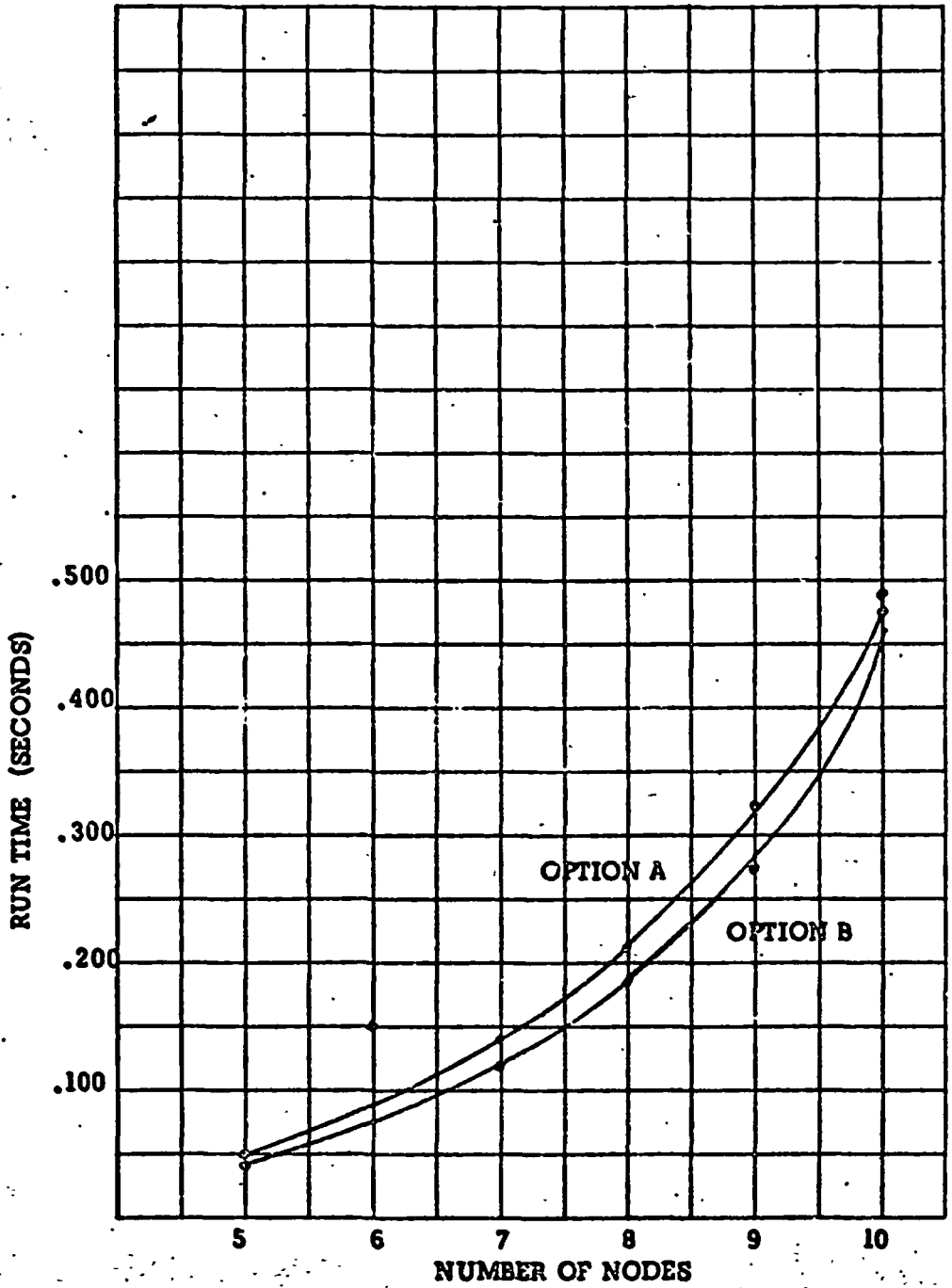

.500
.400
.300
.200
.100
RUN TIME (SECONDS)
OPTION A
OPTION B
5
6
7
8
9
10
NUMBER OF NODES

FIGURE 15 c

RUN OPTIONS

T12 = FIELD PAGE POSITION CORRESPONDING TO BASIS NUMBER
T13 = SHIFTS ARRANGEMENT END-AROUND
T14 = ORDERS INPUT ARRANGEMENT WITH RESPECT TO NODE WEIGHT

WEIGHTS	1260.00	1040.00	960.00	920.00	900.00	840.00	800.00	740.00	740.00	660.00
LABELS	8	5	10	7	1	3	4	2	9	6
	0.00	50.00	90.00	70.00	90.00	90.00	90.00	50.00	10.00	90.00
	50.00	0.00	90.00	70.00	40.00	20.00	90.00	60.00	70.00	30.00
	90.00	90.00	0.00	30.00	80.00	20.00	90.00	50.00	30.00	10.00
	70.00	70.00	30.00	0.00	70.00	20.00	40.00	50.00	50.00	60.00
	90.00	40.00	80.00	70.00	0.00	20.00	20.00	10.00	90.00	30.00
	90.00	20.00	20.00	20.00	20.00	0.00	30.00	80.00	60.00	60.00
	90.00	90.00	90.00	40.00	20.00	30.00	0.00	20.00	10.00	10.00
	50.00	60.00	50.00	50.00	10.00	80.00	20.00	0.00	40.00	10.00
	10.00	70.00	30.00	50.00	90.00	60.00	10.00	40.00	0.00	10.00
	90.00	30.00	10.00	60.00	30.00	80.00	10.00	10.00	10.00	0.00

INTITAL TABLEAU FOR PAGE STRING X

FIGURE 16a

OPTIMIZATION RESULTS

ITER.	OBJECTIVE FUNCTION INITIAL	OBJECTIVE FUNCTION MINIMUM	PAGE CHANGES		GAIN TRIANGLES TESTED	GAIN TRIANGLES DETECTED	DET/TEST RATIO	TREE FORKS	MINIMUM ARRANGEMENT
9	17100.00	13220.00	6	6	742	270	.36388	0	4 10 5 9 1 7 6 2 3 0
8	17640.00	13300.00	15	15	1644	601	.36557	0	4 10 5 1 6 7 9 2 3 0
8	17640.00	13260.00	23	8	2754	975	.35403	1	9 1 7 10 5 4 6 2 3 0
8	17640.00	13300.00	31	8	4192	1482	.35353	2	4 10 5 1 6 7 9 2 3 0
8	17640.00	13300.00	40	9	5696	2003	.35165	3	4 10 5 1 6 7 9 2 3 0
8	17640.00	13320.00	47	7	6846	2398	.35028	4	4 10 5 9 1 7 6 2 3 0
8	17640.00	13300.00	58	11	8434	2972	.35238	5	4 10 5 1 6 7 9 2 3 0
8	17640.00	13280.00	66	8	9544	3346	.35059	6	9 1 7 10 5 4 8 2 3 0
8	17640.00	13300.00	74	8	10982	3853	.35085	7	4 10 5 1 6 7 9 2 3 0
8	17640.00	13300.00	83	9	12486	4374	.35031	8	4 10 5 1 6 7 9 2 3 0
8	17640.00	13320.00	90	7	13636	4769	.34974	9	4 10 5 9 1 7 6 2 3 0
8	17640.00	13220.00	102	12	14978	5208	.35172	10	9 1 10 5 4 7 8 2 3 0
8	17640.00	13300.00	112	10	16600	5827	.35102	11	4 10 5 1 6 7 9 2 3 0
8	17640.00	13300.00	122	10	18030	6319	.35047	12	4 10 5 1 6 7 9 2 3 0
7	18300.00	13220.00	14	14	1180	397	.33644	0	9 1 10 5 4 7 8 2 3 0
6	17600.00	13300.00	13	13	1358	503	.37040	0	4 10 5 1 6 7 9 2 3 0
5	17300.00	13220.00	14	14	1324	476	.35952	0	9 1 10 5 4 7 8 2 3 0
5	17300.00	13220.00	24	10	2571	915	.35603	1	9 1 10 5 4 7 8 2 3 0
5	17300.00	13320.00	32	8	3656	1282	.35066	2	6 3 2 8 7 1 9 5 10 4
5	17300.00	13300.00	38	6	4780	1671	.34958	3	6 3 2 9 7 8 1 5 10 4
4	15000.00	13360.00	12	12	1508	514	.34085	0	6 3 2 9 7 8 1 5 10 4
4	15600.00	13360.00	14	2	1996	681	.34118	1	6 3 2 9 7 8 1 5 10 4
3	17040.00	13180.00	9	9	1336	452	.33832	0	6 3 2 8 4 10 5 7 1 9
3	17040.00	13220.00	14	5	2348	796	.33901	1	6 3 2 8 7 4 5 10 1 9
2	16560.00	13180.00	8	8	1020	361	.35392	0	6 3 2 8 4 10 5 7 1 9
2	16560.00	13180.00	11	3	1758	609	.34642	1	6 3 2 8 4 10 5 7 1 9
2	16560.00	13180.00	13	2	2236	768	.34347	2	6 3 2 8 4 10 5 7 1 9
1	15300.00	13300.00	10	10	1480	468	.32973	0	4 10 5 1 6 7 9 2 3 0
0	15720.00	13220.00	9	9	1252	434	.34665	0	9 1 10 5 4 7 8 2 3 0
TOTALS----------									
10		13180.00		253	35402	12327	.34820	28	RUN TIME = 6.134 SECONDS

OPTION A RESULTS
(PAGE STRING X)

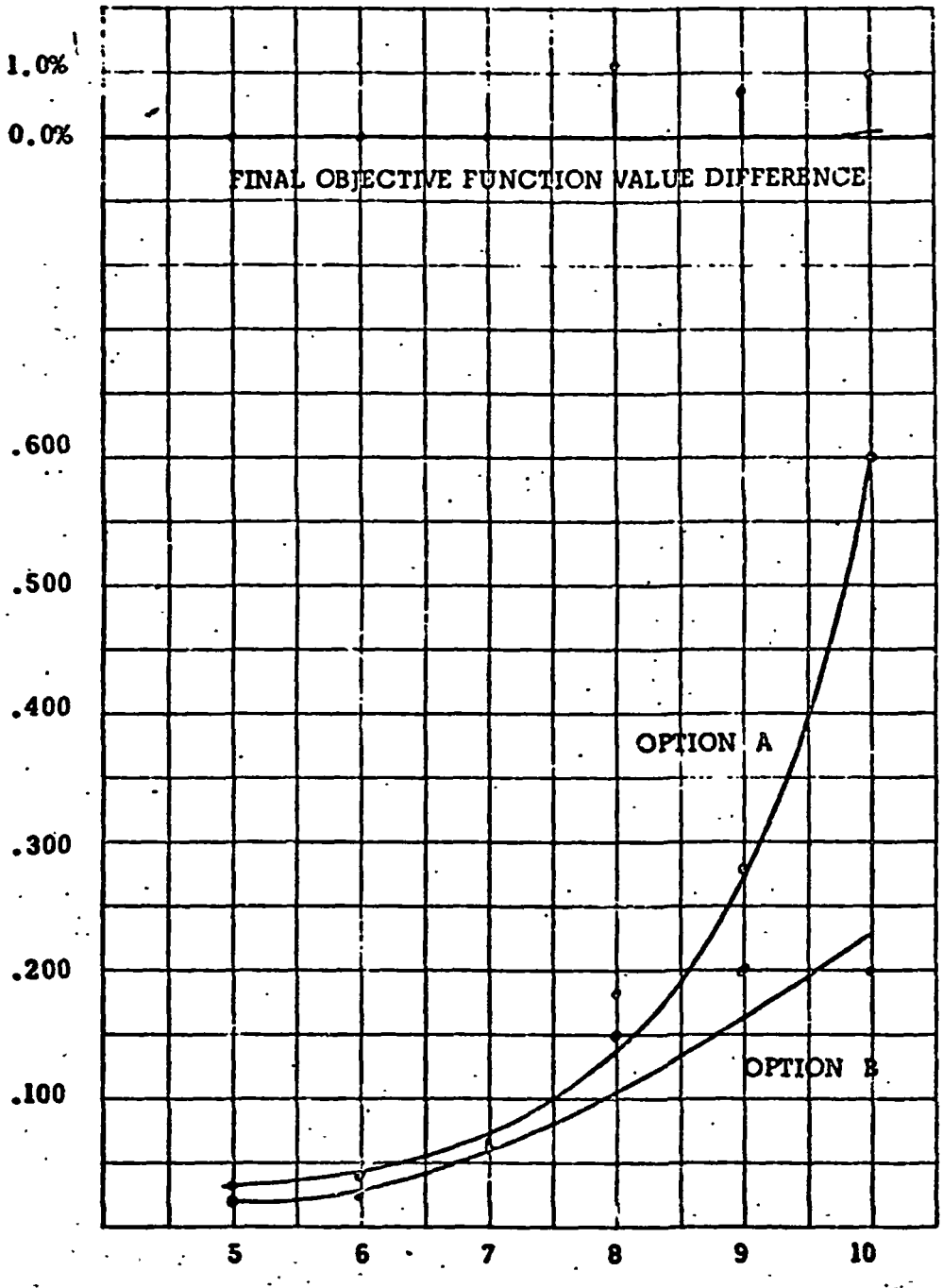
FIGURE 16b

OPTIMIZATION RESULTS

ITER.	OBJECTIVE FUNCTION INITIAL	OBJECTIVE FUNCTION MINIMUM	PAIR CHANGES		GAIN TRIANGLES TESTED	GAIN TRIANGLES DETECTED	DET-TEST RATIO	TREE FORKS	MINIMUM ARRANGEMENT
9	17100.00	13320.00	4	6	742	270	.36388	0	4 10 5 9 1 7 8 2 3 6
8	17100.00	13620.00	4	8	818	292	.35697	0	4 10 5 9 1 7 6 3 8 2
8	17100.00	13620.00	10	2	1268	445	.35095	1	4 10 5 8 1 7 6 3 9 2
8	17100.00	13620.00	16	6	2052	720	.35088	2	4 10 5 8 1 7 6 3 9 2
8	17100.00	13620.00	18	2	2502	873	.34892	3	4 10 5 8 1 7 6 3 9 2
7	17100.00	13320.00	6	6	742	270	.36388	0	4 10 5 9 1 7 8 2 3 6
6	17100.00	14280.00	8	8	1030	397	.38544	0	2 3 9 1 10 5 4 7 8 6
6	17100.00	14280.00	12	4	1786	684	.38298	1	2 3 9 1 10 5 4 7 6 8
5	17100.00	13000.00	7	7	966	365	.35714	0	4 10 5 2 8 3 7 1 9 6
4	17100.00	13320.00	6	6	742	270	.36388	0	4 10 5 9 1 7 8 2 3 6
3	17100.00	13400.00	7	7	742	255	.34367	0	4 10 5 7 8 1 9 2 3 6
3	17100.00	13480.00	10	3	1284	431	.33567	1	4 10 5 7 8 1 9 2 3 6
2	17100.00	13380.00	5	8	534	196	.36704	0	4 5 10 9 1 7 8 2 3 6
1	17100.00	13380.00	5	5	534	196	.36704	0	4 5 10 9 1 7 8 2 3 6
0	17100.00	14760.00	8	8	878	311	.35421	0	8 6 4 10 5 7 1 9 3 2
TOTALS									
10		13320.00		83	10710	3846	.35910	14	RUN TIME = 2.000 SECONDS

OPTION B RESULTS
(PAGE STRING X)

FIGURE 16c



PAGE STRING X

FIGURE 16d

ENCLOSURE 5

A CLASSIFICATION AND SURVEY OF COMPUTER SYSTEM PERFORMANCE EVALUATION TECHNIQUES

ABSTRACT

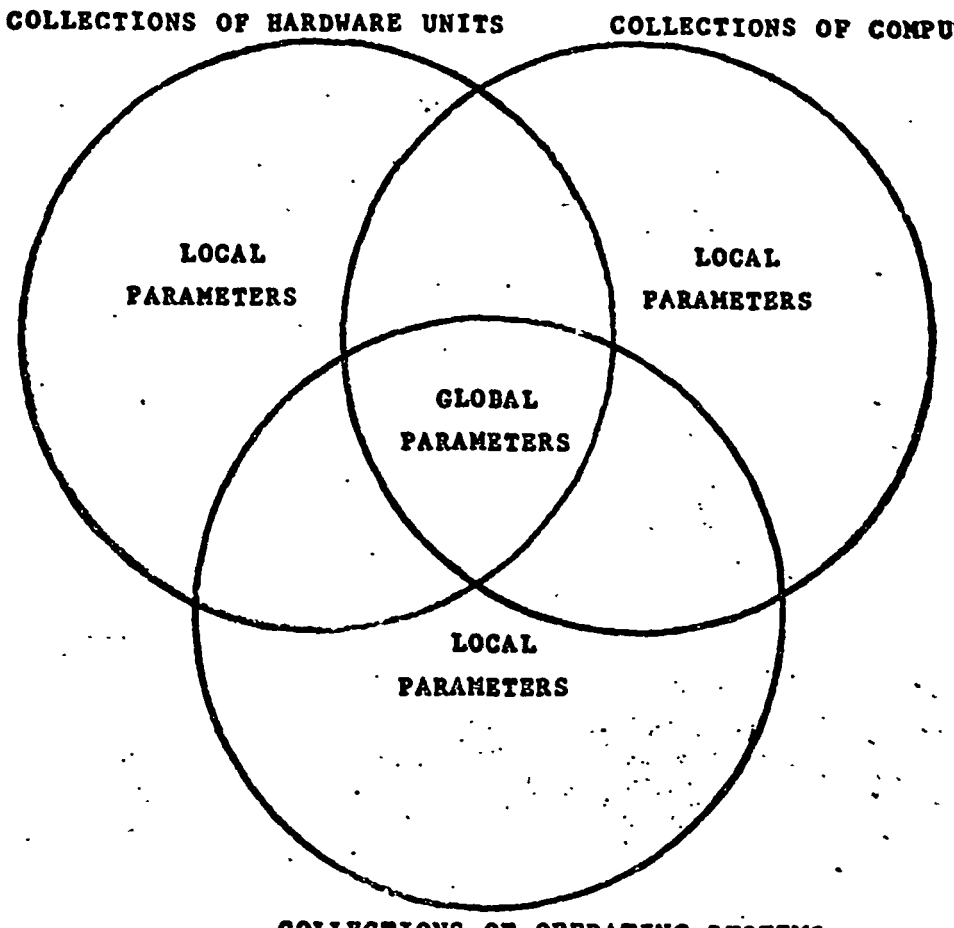
The widely distributed software capabilities coupled with increased compatabilities among hardware units is propagating an ever increasing multitude of computer systems. Such systems aim at a common goal - total system optimization. But achievement of this ultimate goal is being delayed by the absence of a theoretical basis capable of providing a reference for standardized evaluations and comparisons.

The current computer system performance evaluation technology cannot be described in terms of standardized techniques involving established parameters. However, the technology can be partitioned with respect to system component (computing job, hardware unit, and operating system). Within these classifications this paper examines the methodology (analysis, simulation, synthesis) and parameters of typical evaluation techniques presently available. Their advantages and disadvantages are delineated. A comprehensive bibliography is provided both for specific references and for general information.

INTRODUCTION

While awaiting fourth generation computer systems, it is timely to examine the available computer system performance evaluation technology - its methodology, its techniques, and its parameters. This technology cannot be described in terms of standardized techniques involving standardized parameters. No theoretical methods or sound conventions exist. But the assortment of techniques (the technology) can be partitioned with respect to methodology and system component. This paper examines the typical techniques available to evaluate computing jobs, hardware units, and operating systems.

The absence of standardized performance evaluation techniques has not deterred the propagation of an ever increasing multitude of computer systems. Such experimenting has been encouraged by industry compatibilities which have allowed individual computer systems to represent many hardware manufacturers, the operating systems to be either the CPU manufacturer's or any one of several software houses and the workload to range from scientific to commerical in a weird mixture of program languages. As a result a computer system must be defined as the cross product of collections of hardware units, collections of operating system programs, and collections of jobs. Considering the number of terms in the cross product the scope of the evaluation job is indeed great.



THE COMPUTER SYSTEM

FIGURE 1

A general performance evaluation approach must focus upon the isolation and characterization of significant performance parameters - those which characterize the rate and amount of work performed and the availability to perform work. From this viewpoint Computer System Performance Evaluation can be defined as the process of analyzing, simulating, and/or synthesizing the behavior of computer system performance parameters. The absence of adequate performance evaluation parameters has deterred the growth of theoretical methods of computer system organization and design and delayed the ultimate goal - total system optimization. These techniques must provide both static and dynamic measures of components in both a dedicated and a multitask system environment.

But today's system implementation technologies have far outraced performance evaluation technologies. The seriousness of the precipitating problem can be focused by noting that 1969's 21.5 billion dollar investment (53,500 installations) in computer systems is forecast to exceed 50 billion dollars (128,000) by 1975! With such investments at stake computer system optimization promises great rewards for three interest groups, the equipment manufacturers, the computation centers, and the users. Each group is motivated by different primary goals.

Users are interested in performance parameters such as response time(turnaround time), and execute time. In brief they want maximum convenience consistent with economy.

To meet the demands of their many users the computation centers seek to maximinize system resource utilization. They evaluate present systems in order to improve average throughput, average response time (turnaround time), and average operation cost while allowing non-average users to be characterized by parameter variances.

Manufacturers view performance evaluation as the key to meeting present and future demands from the varied and rapidly expanding market. Interests focus upon developing the capability to accurately evaluate present and proposed systems, and to predict performance parameter behavior for systems still in early stages of development. Their individual goals are to maintain a competitive edge by offering computer systems (hardware and software) possessing optimum cost/performance ratios.

METHODOLOGY

Surveying the past decade's literature exposes a wide range and direction of computer system performance evaluation techniques. The decade has produced computer systems ranging from simple batch processing, unit record, uni-processor systems to complex multi-programmed, telecommunication, multiprocessor systems. The corresponding performance evaluation methods have ranged from weighted instruction mix execution time comparisons (ANALYSIS) to time-sharing system simulation models (SIMULATION) to memory hierarchy modeling (SYNTHESIS).

The direction of evaluation methodology has slowly turned from analysis toward synthesis. Analysis includes the separating and breaking up of the computer system into parts so that their functions, relationships, and properties may be understood. Data can be gathered either from real computer systems or from simulation models. By analysis the cost per million instruction executions could be determined based upon the manufacturer's system costs and specifications.

Synthesis includes the combining of known components (hardware and software) into a total computer system or subsystem so that the whole system's (or subsystem's) functions, relationships, and properties may be understood and predicted. Data can be gathered either by measuring real individual parts or simulation models of individual parts. Synthesis represents the highest level of evaluation since it allows performance to be predicted rather than just to be measured. An example of synthesis would be

predicting the average access time of a memory hierarchy based upon the specifications of its components.

Simulation plays a vital role in both analysis and synthesis. This tool allows complex systems to be studied under known conditions and controls. Many real systems are too complex for theoretical analysis and the use of simulation may prove to be the only workable approach. However, the time and cost required to generate accurate simulation models can easily exceed that of direct measurement approaches. But for evaluating proposed systems which yet do not exist in hardware, simulation offers an alternative to building prototypes.

PARAMETERS

Any evaluation technique must consider a set of parameters referenced to a specific environment. Three parameters, throughput, response time, and cost per operation, commonly denote global system capacity. Common local parameters include: average access time, average data transfer rate, average instruction execution time, hardware (processor, memory, tape, drum, channel, etc.) utilization, average executive processing percentage, average job processing percentage, and effective busy time of the processor. The design of specific computer systems reflects the weights assigned to these parameters.

Availability represents a different type of global performance parameter. It is usually expressed as a percentage or as an amount

of time. Manufacturers control availability through component reliability and redundancy, error detection and correction, and preventative maintenance. Computation centers influence availability through planned graceful degradation involving backup equipment (I/O, memory, etc.) and job rescheduling.

Throughput can be considered as the steady state work capacity of a computer system . It is usually expressed as a number of specific tasks performed per time interval. For example, a system's throughput might be described by the number of typical PL/1 statements compiled per hour or by the number of information retrieval requests satisfied per hour. Many applications demand specific throughput levels. As a result the system must be designed to achieve such levels.

Response time can be considered as a transient work capacity of the computer system. It denotes the delay between submitting the job and receiving the job output. [32] Clearly it is a function not only of the system throughput but also of the system environment including the number of jobs and their priorities. For many batch processing systems the response time (turnaround time) may be relatively constant and measured in hours. But for certain time-sharing systems the response time may be only seconds. For example the response time for a stock broker's quotation request or an airline reservation inquiry may be less than 5 seconds.

The third global parameter measures performance in units of money per operation unit time. It is usually given as dollars per hour. Applications exist where more time than money is allowed.

However, definite economies of scale exist for computer systems. Solomon [35] has showed that the IBM System 360 models possess ratios of 1.6 to 2.1; i.e. by increasing the system investment by a factor K the internal operating speed increases by K raised to a power ranging from 1.6 to 2.1 depending upon the model. For example the average monthy rental for models 75 and 30 are approximately $80,000 and $8,000 respectively. With K = 10 an improvement of 100 in the internal operating speed would be expected. For a scientific instruction mix [35] the Model 75 is approximately 40 times faster. Such comparsions are usually referenced to the famous Grosch's Law which equates ''added economy only as the square root of the increase in speed - that is, to do a calculation ten times as cheaply you must do it one hundred times as fast.''[18]

The uninitiated might attempt to express system performance as a single entity involving a relationship between the global parameters, throughput, response time, operation cost, and availability. However, such a figure of merit has not evolved, because system performance is not a single entity!

USER JOB TRANSFORMATIONS

From its definition to execution a problem undergoes many transformations. The total transformation efficiency of this process can be computed as the product of the individual transformation efficiencies. As a result the user who investigates why his program possesses poor performance must carefully consider several major transformation efficiencies including the following:

1. Development of the problem statement and solution algorithm (user)
2. Generation of a high level source language program (programmer)
3. Generation of an assembly language program (compiler)
4. Generation of the appropriate machine language code (assembler or loader)
5. Generation of the problem's solution by managing the job's execution on the system hardware (operating system)

The first transformation is performed by the scientist or engineer who reduces the problem to a statement and develops a solution algorithm. This first transformation is not a part of computer system performance evaluation, but is noted in order to stress the importance of the communication link between the scientist and the programmer. In order to maximinize this transformation efficiency the problem must be precisely stated and an optimum solution algorithm chosen. It is usually helpful to develop several solution algorithms so that the chosen 'optimum' algorithm represents more than just one way to solve the problem.

The second transformation and probably the most important transformation is performed by the programmer who reduces the problem statement and solution algorithm into a high level source language program such as FORTRAN, ALGOL or COBOL. To maximinize this transformation efficiency the programmer must consider many parameters including: source language selection, I/O treatment, data treatment, and system resources and properties. Major errors such as wrong source language selection are usually avoided, but more subtle errors may strongly degrade the transformation effeciency. For example consider a high usage DO LOOP containing numerous initialization statements. Such statements might easily be moved outside the DO LOOP resulting in significant execution time improvements.

Familiarity with available system subroutines must not be underrated. Such subroutines usually represent sophisiticated and highly optimized algorithms in either assembly or machine language. As a result their proper use eliminates the degradation introduced by the compiler and minimizes the inefficiencies introduced by the programmer. For example consider the drastic effect (reduction) upon a program's execution time should a programmer approach the Fibonacci difference equation solution using iterative rather than recursive techniques.

Note that it is not unusual for the programmer to eliminate the step involving the high level source language transformation by directly generating the program in the appropriate assembly language. Directly generating assembly language requires additional

programming skill and time and produces a machine oriented program. However, a skilled programmer can usually generate more efficient assembly language code than the two step transformation including generating the high level source language followed by compilation. Also, for high usage programs elimination of the compiling time becomes an important consideration.

Usually the assembly language is generated by the system's appropriate source language compiler. Such compilers represent one of the most highly optimized system programs. However, each compiler represents a compromise between speed, memory requirements, and output code quality. This compromise should represent the optimum cost/performance ratio for the user, but a common underlying assumption is that only low usage programs are involved. It is this assumption that allows significant improvements to be gained for high usage progams by the programmer working directly in assembly language.

Machine language generation probably represents the most highly efficient of all the transformations. Converting assembly language into machine code is mainly a one to one mapping procedure which needs to be performed automatically due to its tedious nature. The assembler or loader can do little to overcome program inefficiencies embedded one, two or three transformation levels. Note that it has been assumed that each transformation step possesses a maximum efficiency of one. Efficiencies greater than one would imply that the transformation process possesses 'error' correcting capabilities.

EVALUATION TECHNIQUES FOR COMPUTING JOBS

Evaluation techniques for the collections of jobs can produce significant benefits. Developed techniques allow dynamic monitoring of program execution in order to expose the program's traffic patterns and execution time density patterns. Other techniques provide modeling tools for studying program structures . Common purpose is to contribute input data for program improvement processes.

DYNAMIC MONITORING OF SINGLE JOBS

Three general approaches have been taken to achieve dynamic program execution monitoring. They include self monitoring by introduced artifact and external monitoring by hardware/software devices (snuping). Also, various techniques allow programs to be executed during so-called interpretative modes which are either controlled by special programs or by special hardware features.

The basic purpose of all such techniques is to gather dynamic statistics such as subroutine execution times, branching probabilities, subroutine usage and resource usage. Such statistics provide important input to program optimization processes.

ARTIFACT - Let the program be represented by a directed graph with statement(s) corresponding to nodes and statement transitions corresponding to arcs. The activity of the nodes and arcs can be monitored by inserting artifact at the desired monitor points. Such artifact can take the form of subroutine calls. In a FORTRAN

PAGE 12 MISSING

program the following subroutine calls might be used;

CALL COUNTER(I)

and CALL TSTAMP(I)

To record the usage frequency the subroutine, COUNTER, simply counts the number of times that it is called. To record execution times the called subroutine, TSTAMP, fetches the system clock time using the CALL SECOND subroutine and then labels and stores the clock time. Values of the index I correspond to unique monitor points.

The artifact technique is simple to implement, but increases overall execution time of the monitored program. Also, no theoretical methods exist to locate the strategic monitor points required to minimize the execution time cost with respect to the information collected.

The technique allows program traffic patterns and execution time density patterns to be measured as a function of the input data. Since the technique locates the high traffic program segments, it helps to locate those segments which should be considered for optimization. The technique fails to ideally record execution times since the artifact execution times are also measured by the system clock. Compensation for the artifact execution times is tedious, but often it can be ignored if the measured segments are relatively large. Similarly the execution time density patterns pinpoint those program segements which should be optimized in order to improve the program execution time.

EXTERNAL MONITORS - The hardware monitor has been used effectively to gather dynamic statistics of programs. Examples include the IBM Program Monitor [1], the IBM Time Sharing System Performance Activity Recorder (TS/SPAR) [33], and the UCLA Snuper Computer [15]. The last two examples are also highly applicable toward evaluating Operating Systems. Basically these hardware monitors provide means of detecting the contents and/or activity of logic lines and means to store such gathered data. For example the early IBM Program Monitor recorded the contents of the IBM 7090 computer's instruction counter. The monitor was able to keep pace with the 7090 by recording only certain selected types of instructions, buffering and packing the gathered data, and halting the 7090 whenever the buffer capacity was exceeded. Data was recorded on magnetic tape and later processed by several special programs. Final results were presented either as printout or on film. Principal benefits of such a technique includes the generated descriptions of the program segment execution time and frequency. Also, no artifact is introduced to affect the execution timing.

The UCLA Snuper Computer possessed several significant differences. A second computer (Sigma 7) was used to control the monitoring of the object computer (IBM 7090). Means were provided to monitor numerous internal signals and to present unique messages at the interface to the Snuper Computer. In Phase 1, artifacts were introduced into the object computer's programs at significant event points. However, the artifact consisted of so-called emitter calls which caused corresponding unique messages to be presented

at the interface to the Snuper Computer. The messages were interpreted as unique memory addresses by the Snuper Computer and the corresponding memory locations were indexed by one to provide event counters. In summary the Phase 1 Snuper Computer alloved program monitoring with reduced artifact, but required a second computer complete with monitoring hardware and programs.

For Phase 2 the Snuper Computer completely eliminated the artifact in the object computer programs by gathering sufficient data during the program compilation and loading phases to generate a Significant Event Filter (SEF). Interface messages were used to address the SEF and if the corresponding bit were a one the event was defined to be significant. Event counting proceeded as in Phase 1. In both phases gathered data was later processed in order to extract important statistics.

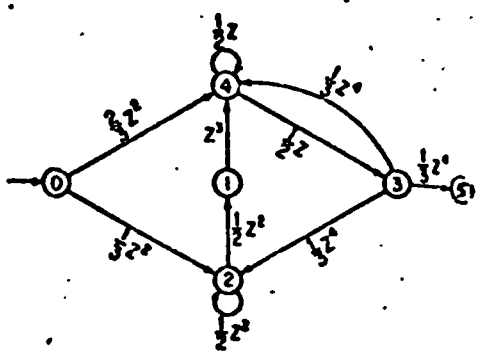
Hardware monitors such as those just described are best suited for laboratory use. They lack portability, introduce object system downtime for connections, are expensive to operate and develop, and are usually designed to monitor only a very small class of computers (probably only one model). However such techniques have been accepted by manufacturers as useful tools in program optimization processes. Even if the use of such tools cannot become wide spread, their benefits may become widespread through the distribution of optimized system programs (a user benefit).

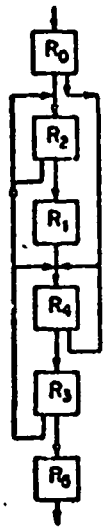
GRAPH ANALYSIS

Since the programmer's flow chart is a directed linear graph, techniques of graph theory are applicable to the analysis and modeling of programs. A general model has the nodes corresponding to computational subsequences (statements or subroutine) and arcs corresponding to execution flow. The model can be expanded by assigning execution times to the nodes and branching probabilities as requirements to the arcs. [23,28] By changing the node assignments (statement, subroutine, job, etc.) the model affords a general approach toward modeling problems associated with program execution.

A number of techniques [14] to transform cyclic graphs into acyclic graphs. Such schemes allow the cyclic program loops to be replaced by deterministic acyclic ones having expected execution frequencies. Such a scheme is useful in estimating a program's expected execution time. However the expected branching probabilities for non-looping such as those iterated under the control of counters depend upon either dynamic program monitoring or knowledgeable estimate. In either case the model is descriptive and a proper range of input data must be

If all the necessary assumptions were valid, the straightforward toward program analysis often results in a useful model. Consider the example [25] shown in Figure 2 assuming that the branching probabilities are statistically independent and therefore insensitive. A discrete Markov analysis of the execution time for the six node graph involves a





MATRIX OF EXECUTION TIMES

R	t	TIME UNITS
R_0	t_0	2
R_1	t_1	3
R_2	t_2	2
R_3	t_3	4
R_4	t_4	1
R_5	t_5	0

MATRIX OF BRANCHING PROBABILITIES

p_{ij}	0	1	2	3	4	5
0	0	0	1/3	0	2/3	0
1	0	0	0	0	1	0
2	0	1/2	1/2	0	0	0
3	0	0	1/3	0	1/3	1/3
4	0	0	0	1/2	1/2	0
5	0	0	0	0	0	0

$$G_{05}(z) = \frac{\frac{2}{3}z^7 - \frac{1}{18}z^9 + \frac{1}{36}z^{12}}{1 - \frac{1}{2}z - \frac{1}{2}z^2 + \frac{1}{4}z^3 - \frac{1}{8}z^5 + \frac{1}{12}z^7 - \frac{1}{12}z^{10}} \qquad (1)$$

$$\bar{t} = \left.\frac{d\,G_{05}(z)}{d\,z}\right|_{z=1} = 29\ 1/3 \text{ units of time.} \qquad (2)$$

FIGURE 2

EVALUATION TECHNIQUES for HARDWARE UNITS

The earliest computer system evaluation techniques focussed on evaluating the hardware units since systems at that time were greatly limited by the hardware technology. During the era of the batch processing systems performance was very strongly characterized by the performance of major hardware components such as the CPU and the memory. Evaluation techniques mostly compared the execution times of single machine instructions such as ADD, frequency weighted instruction mixes, and program kernels.

With the introduction of the multiprogrammed systems attention shifted toward developing application benchmarks. But with the introduction of the time sharing systems and the present generation of super computers (CDC 7600, IBM 369/91,195) attention shifted toward developing simulation techniques.

INSTRUCTION EXECUTION TIME COMPARISONS

Probably the first technique used to compare different CPU's merely used the execution times for single instructions such as ADD or MULTIPLY. This technique is definitely application sensistive and therein is its great weakness. The performance of only a very limited class of programs can by characterized accurately by considering only the ADD and MULTIPY execution times. For example a typical matrix multiplication program consists of only about 25% ADD and MULTIPLY type instructions [35]. Monitoring scientific installations has shown that the ADD/SUBTRACT and MULTIPLY/DIVIDE instruction types compose less than 20% of the instructions executed. [3,25,35]

Other weaknesses of the technique includes the failure to compensate for the CPU's having different operand sizes, different organizations (one address, multi-address, auxiliary registers, multiple execution units, etc.), and different instruction implementation algorithms (pure binary, decimal, microprogrammed, floating point).

STORAGE CYCLE TIME COMPARISONS

Another early evaluation technique compared the storage cycle times . The usefulness of this scheme has been outdated by interleaving memory banks, high speed 'cache' memories, and large storage access widths. For example the IBM 360/85 possesses a storage access width of 16 bytes (128 bits) with either 2 or 4 way interleaving plus a 32k byte 'cache' memory with a 80 nanosecond cycle. [17] Such features would be overlooked if only the .96 microsecond storage cycle time were considered. Considering that the IBM 360/25 has a .90 microsec. storage cycle time, a very false performance ratio might be deduced. The role of the 'cache' memory drastically affects the effective storage cycle of computers. It has been reported that the IBM 360/195 'cache' memory satisfies an average of 99% of all storage requests and that varying the backing bulk memory access time from a fraction to 2 microseconds affects system performance by only 10 to 15%. Clearly the storage cycle time parameter must either be redefined or marked as insignificant when evaluating such modern computers.

The Maximum Storage Bus Rate (MSBR) has evolued as a modern figure of merit.

MSBR = (access width/storage cycle time)x interleave factor

However, MSBR fails to compensate for the 'cache' memory, but would show that the IBM 360/85 has a MSBR= 533.33 megabits while the model 25 has only a MSBR= 17.77 megabits.

INSTRUCTION MIX EXECUTION TIME COMPARISONS

Recognizing some of the weaknesses of the Instruction Execution Time Comparisons, the Weighted Instruction Mix approach was developed. [3,25,35] From actual operating systems, statistics were gathered to weigh individual instructions according to their frequency of occurance in a particular set of applications. A weighted instruction mix execution time can then be calculated. Arbuckle [3] has presentedd the following scientific application mix:

INSTRUCTION TYPE	FREQUENCY PERCENTAGE
Floating Point Add/Subtract	9.5%
Floating Point Multiply	5.6%
Floating Point Division	2.0%
Load/Store	28.5%
Indexing	22.5%
Conditional Branch	13.2%
Other	18.7%

Again this approach suffers from failure to correct for the CPU's operand size, organization and algorithm implementation. Properties of the instruction stream such as sequence and I/O operations are ignored. Instruction sets which may differ greatly

in power are evaluated on the basis of certain universal instructions while leaving 20 to 30% as miscellaneous instructions. Since the statistics are collected from a particular system they reflect that system's properties such as its organization, compiler, assembler, operating system efficiency, and workload. A system's workload depends on both the job types and their number. Excessive amounts of waiting time would certainly bias such statistics abnormally. However, the technique provides very useful information to the hardware designer since high frequency instructions are pinpointed. By optimizing such high frequency instructions the average number of jobs executed per wait time can be increased.

PROGRAM KERNELS

To overcome many of the weaknesses of the Instruction Mix approach, the program kernel was introduced. [3,8,35] The kernel represents the CPU coding required to perform a significant task such as evaluating a polnominal, solving a set of simultaneous linear equations or multiplying two matrices. Obviously the kernel allows a skilled programmer to use all the features of a CPU. A significant property as a yardstick is that it possesses poor correlation with weighted instruction mix results. Just how representative of a system workload can a matrix multiplication kernel be? It is no surprise that system performance based upon such a kernel representing a very specific job possesses poor correlation with the Weighted Instruction Mix technique which is much more representative of a typical system workload. Certainly both techniques offer convenient means of gathering data. The capability of the program kernel to provide data referenced to a specific job should not be ignored.

However, kernels have proved useful in measuring the instruction execution rate of high performance computers such as the IBM 360/195. No single machine cycle can be identified with a single instruction execution due to the overlap, 'pipelining' and lookahead in instruction execution. Iterations of kernels as small as 15-20 instructions have been employed to establish effective instruction execution rates measured in millions of instructions per second. Much attention must be given to the content of such kernels since the effective instruction rate of such computers is strongly affected by the instruction stream and storage requests.

A significant weakness of evaluating system performance based upon individual kernel execution times is the failure to measure the efficiency of the Operating System. The arrival of multiprogramming and multiprocessing initiated the Application Benchmark evaluation technique.

APPLICATION BENCHMARK

Presently the internal performance of computer systems is frequently validated by application benchmarking. [7,19] This technique involves timing the execution of a typical collection of user jobs. One manufacturer suggests that the user benchmark contain 15 frequently used application jobs. Such a technique provides the user with much specific information since his own jobs are actually executed and timed. Such a collection of jobs tends to produce average performance parameter values since more events occur in the sample space of jobs. Also, it provides a test for the multiprogramming and/or multi-processing operating system. By

running the Application Benchmark on competitive systems, performance ratios may be rather accurately established by the user. Contents of the benchmark should be carefully chosen to insure that the user's tasks are fairly represented. Such results are representative of total system performance (hardware and software).

SIMULATION MODELING

Often simulation modeling provides the best if not only means of studying today's sophisticated hardware units. Designing hardware units such as CPU's and memory hierarchies [2] involves optimizing parameters beyond the scope of simply 'tuning' a hardware prototype. For example a hardware prototype would have provided an inadequate tool for evaluating a concept such as the IBM 'cache' memory. [17] Optimizing overall performance as a function of design parameters such as 'cache' memory capacity, transfer block size, backing memory access time, and program structure characteristics could definitely be satisfied best using a simulation model. As a result the development of a prototype is no longer the starting point for design optimization. It frequently represents the hardware implementation of an optimized simulation model whose primary purpose is to prove the feasibility and reliability of unknown components and whose secondary purpose is to prove the accuracy of the 'optimum' simulation model.

However, a close examination of a simulation model exposes the many assumptions and approximations used to construct the model. It is the accuracy of these numerous and often difficult assumptions that establishes the validity of the simulation evaluation results. Probably the major consideration involves the compromise which must be made between simulation detail and cost. Simulation is not the answer to every evaluation problem, but it frequently provides a technique to evalute new concepts without filling warehouses with "golden prototypes."

EVALUATION TECHNIQUES for COLLECTIONS OF OPERATING SYSTEMS

To a degree the performance influences of both the collections of jobs and collections of hardware units can be isolated and characterized. Such limited information can in some cases even establish bounds on total system performance. But total system performance is accurately measured only by global performance parameters such as throughput, response time, and cost per operation. These parameters are functions of all three system collections.

SIMULATION

Simulation models have been widely used to evaluate today's operating system designs. References include articles concerning a wide range of operating system types including real-time [37], time-sharing [16,24], multiprocessing [5,9,34], and multiprogrammed [20]. Simulation has become a vital part of the developement of every operating system. Often manufacturers must include the results of simulation testing as an integral part of large system proposals. The UNIVAC 1108 multiprocessor Evaluation of System Performance (ESP) model [9] exemplifies a typical simulation project. It has been employed to evaluate the Jet Propulsion Laboratory multiprocessor system. Statistics reflecting channel utilization, processor utilization, peripheral equipment utilization, system overhead, and effective processor busy time can be gathered by simulation. The model construction reflects the existing 1108 executive system (EXEC 8) including the Executive Control, Input/Output, and Scheduling functions.

System hardware resources are characterized by average access and transfer times. The user provides input job descriptions. From this information the number of interrupts and I/O transfers are calculated and their execution simulated using detailed descriptions of the executive system (EXEC 8) functions and the hardware resources. System performance can then be predicted. Up to eight reports reflecting system performance statistics are generated. They include: number of I/O channel loads including utilization percentage, executive system routine usage for the individual jobs plus the total workload, executive and workload instruction totals, interrupt summary, software utilization distribution percentages, and hardware utilization plus a general summary reflecting throughput and busy time .

SELF MEASUREMENT

A convientent approach to monitoring an operating system's performance is to introduce artifact into the operating system program. Such artifact allows statistics to be gathered dynamically. A current example is the IBM System Internal Performance Evaluation (SIPE) program which is designed for the System/360 Time Sharing System [12]. This software measurment technique introduces SIPE 'hooks' into the resident supervisor program at strategic locations. Logically these 'hooks' function as subroutine calls to introduce SIPE into the normal program stream. Each 'hook' possesses an unique identifier code which drives the SIPE program to collect specific information about the system status. This snapshot of the system status includes contents of internal registers, specific locations of main memory, and a time stamp. The collected data is buffered and later transferred to tape. From these tapes data reduction programs extract various statistics and format them for display.

SIPE was designed for customer installation use as well as laboratory use. The current IBM 360/TSS resident supervisor programs contain the SIPE 'hooks.' Unless activated the effect of the 'hooks' upon system performance is insignificant. When fully activated system performance is degraded only about 8%. As a result SIPE can be justified for frequent user use such as installation debug.

Software approaches to evaluating operating system performance face one serious compromise since executing the introduced artifact slows down the system and may percipitate abnormal system conditions. Event resolution and system degradation must be compromised.

Significant features of software approaches such as SIPE include portability, flexibility, and relatively low cost. The SIPE 'hooks' (8 bytes each) are actually part of the resident supervisor. SIPE itself must be link-edited into the system during startup. Obviously no special hardware exists in such a totally software technique. Great flexibility exists since only the resident supervisor contains the 'hooks.' All the TSS/360 utility functions are unaltered. Evaluation of new TSS configurations involves only adding or deleting 'hooks' in the supervisor. Once developed the technique offers low operating cost and possesses no holding (storage) costs such as some hardware monitors like the Snuper Computer.

EXTERNAL MONITORS

External hardware monitors offer another approach to evaluating operating systems. Devices can range from simple usage meters and channel activity counters to complex units such as the UCLA Snuper Computer [15], the UNIVAC 1108 Hardware Monitor [29], and the IBM Time Sharing System Performance Activity Recorder (TS/SPAR) [33]. Since such devices are interference free their prime advantage is the capability to gather dynamic statistics in a non-degraded system environment. But sophisticated devices such as the

last three possess properties which limit their general use. They lack portability, flexibility, and economical operation of software approaches such as SIPE. For example their large physical size limits portability and their installation can impose excessive downtime for the monitored system. Since they must interface with hardware, they are designed to monitor a particular system's internal architecture and circuit family. But most of the large hardware monitors were intended only for laboratory use. They can contribute to installation optimization directly through the development of better operating system programs.

CONCLUSION

This paper has classified and examined the presently available assortment of computer system evaluation techniques. The techniques have been partitioned with respect to computing jobs, hardware units, and operating systems. Within each classification, techniques representing analysis, simulation, and/or synthesis have been examined in order to expose the advantages and disadvantages inherent to the specific techniques.

Characterizing the present technology provides both an outline of what is available and an outline of what needs to be developed. Fundamental among the needs of the computer evaluation technology is a theoretical basis providing a reference for standardized evaluations and comparisons. First, standardized parameters should be established. Such parameters must characterize global system performance; they must provide common denominators allowing local parameters to be reduced. Then appropriate techniques may be generated to measure these parameters under known system environments and workloads. Certain typical classes of installations might be represented by standard workloads consisting of standardized kernels or application benchmarks (7). Special purpose installations must continue to develop customized application benchmarks for comparative testing in order to evaluate competitive proposals.

Besides evaluation for selection purposes there remains the important problem of monitoring the installed system. Such performance monitoring could be greatly benefited by designing snuping features into the hardware units and the operating systems (36) in order to collect utilization and usage frequency statistics. Ideally such features would operate in a parallel and interference-free fashion. Those techniques such as software artifact which do cause interference should provide means for their selective use.. Built-in snuping features would allow computation centers to continually monitor their systems in order to provide early detection of system environment changes and precipatating problems. Also, such field usage statistics would provide the manufacturer much important input for future system designs.

REFERENCES

1. C. T. Apple, "The Program Monitor-A Device for Program Performance Measurements," Proceedings 20th ACM National Conference (Aug. 1965), pp. 66-75.

2. W. Anacker and C. P. Wang, "Performance Evaluation of Computing Systems with Memory Hierarchies," IEEE Trans. Comp., Vol. EC-16, No. 6 (Dec. 1967), pp. 764-773.

3. R. A. Arbuckle, "Computer Analysis and Thruput Evaluation," Computers and Automation, Vol. 15, No. 1 (Jan. 1966), pp. 12-15,19.

4. A. J. Bonner, "Using System Monitor Output to Improve Performance," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 290-298.

5. F. R. Baldwin, W. B. Gibson and C. B. Poland, "A Multiprocessing Approach to a Large Computer System," IBM Systems J., Vol. 1, No. 1 (Sept. 1962), pp. 64-76.

6. W. Buchholz, "A Selected Bibliography on Computer System Performance Evaluation," Computer Group News (March 1969), pp. 21-22.

7. W. Buchholz, "A Synthetic Job for Measuring System Performance," IBM System J., Vol. 8, No. 4 (1969), pp. 309-318.

8. P. Calingaert, "System Performance Evaluation: Survey and Appraisal," Comm. ACM, Vol. 10, No. 1 (Jan. 1967), pp. 12-18.

9. J. A. Caron, L. R. Jorze and K. D. Tschetter, "Evaluation of System Performance Model," UNIVAC Advanced Systems and Programming Report P.X.-5426 (Aug. 11, 1969), 67 pages.

10. P. S. Cheng, "Trace-Driven System Modeling," IBM Systems J., Vol. 8, No. 4 (1969), pp. 280-299.

11. E. G. Coffman, Jr., "Analysis of a Drum Input/Output Queue Under Scheduled Operation in a Paged Computer System," ACM J., Vol. 16, No. 1 (Jan. 1969), pp. 73-90.

12. W. R. Deniston, "ATSS/360 Software Measurement Technique," Proceedings 24th ACM National Conference (Aug. 1969), pp. 229-245.

13. G. Estrin and D. Martin, "Experiments on Models of Computations and Systems," IEEE Trans. Comp., Vol. EC-16, No. 1 (Feb. 1967), pp. 59-69.

14. G. Estrin and D. Martin, "Models of Computational Systems-Cyclic to Acyclic Graph Transformations," IEEE Trans. Comp., Vol. EC-16, No. 1 (Feb. 1967), pp. 70-79.

15. G. Estrin, D. Hopkins, B. Coggan and S. D. Crocker, "Snuper Computer: A Computer in Instrumentation Automation," AFIPS Conf. Proc., Vol. 30 (1967 SJCC), pp. 645-656.

16. G. Estrin and L. Kleinrock, "Measures, Models and Measurements for Time-Shared Computer Utilities," 1967 ACM National Meeting Proceedings, pp. 85-96.

17. Donald H. Gibson and W. Lee Shevel, "'Cache' Turns Up A Treasure," Electronics, Vol. 42, No. 21 (Oct. 13, 1969), pp. 105-107.

18. H. R. J. Grosch, "High Speed Arithmetric: The Digital Computer as a Research Tool," Optical Society of America Journal, Vol. 43, No. 4 (April 1953), pp. 306-310.

19. E. O. Joslin, "Application Benchmarks: The Key to Meaningful Computer Evaluation," Proceedings 20th ACM National Conference (Aug. 1965), pp. 27-37.

20. J. H. Katz, "An Experimental Model of System/360," Comm. ACM, Vol. 10, No. 11 (Nov. 1967), pp. 694-702.

21. D. D. Keefe, "Hierarchical Control Programs for Systems Evaluation," IBM Systems J., Vol. 7, No. 2 (1968), pp. 123-133.

22. H. G. Kolsky, "Some Computer Aspects of Meteorology," IBM Journal, Vol. 11, No. 6 (Nov. 1967), pp. 584-600.

23. T. C. Lowe, "Analysis of Boolean Program Models for Time-Shared, Paged Environments," Comm. ACM, Vol. 12, No. 4 (April 1969), pp. 199-205.

24. N. R. Nielsen, "The Simulation of Time Sharing Systems," Comm. ACM, Vol. 10, No. 7 (July 1967), pp. 397-412.

25. E. Raichelson and G. Collins, "A Method for Comparing the Internal Operating Speeds of Computers," Comm. ACM, Vol. 7, No. 5 (May 1964), pp. 309-310.

26. C. V. Ramamoorthy and M. J. Gonzales, "Recognition and Representation of Parallel Processable Streams in Computer Program-II (Task/Process Parallelism)," Proceedings 24th ACM National Conference (Aug. 1969), pp. 387-397.

27. C. V. Ramamoorthy, "Discrete System Representation and Analysis by Generating Functions of Abstract Graphs," IEEE Int. Conv. Rec. 1965, pp. 68-80.

28. C. V. Ramamoorthy, "Discrete Markov Analysis of Computer Programs," Proceedings ACM National Conference 1965, pp. 386-392.

29. D. J. Roek and W. C. Emerson, "A Hardware Instrumentation Approach to Evaluation of a Large Scale System," Proceedings ACM National Conference 1969, pp. 351-367.

30. M. Schatzoff, R. Tsao and R. Wing, "An Experimental Comparison of Time Sharing and Batch Processing," Comm. ACM, Vol. 10, No. 5 (May 1967), pp. 261-265.

31. P. H. Seaman and R. C. Soucy, "Simulating Operating Systems," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 264-279.

32. Allan L. Scherr, "An Analysis of Time-Shared Computer Systems," Massachusetts Institute of Technology, Project MAC Technical Report MAC-TR-18, June 1965.

33. F. D. Schulman, "Hardware Measurement Device for IBM System/360 Time Sharing Evaluation," Proceedings 22nd ACM National Conference (Aug. 1967), pp. 103-109.

34. E. C. Smith, "A Directly Coupled Multiprocessing System," IBM Systems J., Vol. 2, No. 3 (Sept.-Dec. 1963), pp. 218-299.

35. M. B. Solomon, Jr., "Economics of Scale and IBM System/360," Comm. ACM, Vol. 9, No. 6 (June 1966), pp. 435-440.

36. W. I. Stanley, "Measurement of System Operational Statistics," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 299-308.

37. W. E. Stanley and H. F. Hertel, "Statistics Gathering and Simulation for the Apollo Real-Time Operating System," IBM Systems J., Vol. 7, No. 2 (1968), pp. 85-102.

38. E. S. Walter and V. L. Wallace, "Further Analysis of a Computing System Environment," Comm. ACM, Vol. 10, No. 5 (May 1967), pp. 266-272.

39. Peter White, "Relative Effects of Central Processor and Input-Output Speeds Upon Throughput on the Large Computer," Comm. ACM, Vol. 7, No. 12 (Dec. 1964), pp. 711-714.

N71 13662

ENCLOSURE 6

UNIFIED HARDWARE/SOFTWARE DESIGN

UNIFIED HARDWARE/SOFTWARE DESIGN - QUARTERLY REPORT

INTRODUCTION

In our previous report we gave a preliminary statement of the problem to be dealt with as well as structural investigation with a view toward optimal solutions. In this report we shall generalize portions of the problem statement as well as introduce mixed hardware/software considerations. Solution techniques will also be discussed and emphasis will be placed on an efficient, rapid, near-optimal approach as results from the previous report, as well as investigations by others indicate that guaranteed optimal solutions for this type of problem are computationally infeasible.

PROBLEM STATEMENT

Essentially the problem considered in the last report was:

Synthesis of a system to operate in a known environment whose operation could be broken down into a collection of subtasks (computer instruction, instructions, operations, functions, etc.) with a performance criteria defined over these subtasks (i.e. execution time or cost, power, weight, etc.) subject to appropriate constraints. The problem is to select among various methods of implementing the subtasks in order to optimize the performance of the overall system. The formulation:

given $F = [f_1 \ldots \ldots \ldots \ldots f_n]$ denoting a set of relative usage frequencies of the subtasks (environmental information)

$$T = \begin{bmatrix} (t_{11}\ t_{12} \ldots . t_{1m_1}) \\ \\ (t_{n1}\ t_{n2} \ldots . t_{nm_n}) \end{bmatrix}$$

giving performance information (execution time), and corresponding to the performance data a collection of resource usage data such as for example

$$C = \begin{bmatrix} (c_{11}\ c_{12} \cdots c_{1m_1}) \\ (c_{n1}\ c_{n2} \cdots c_{nm_n}) \end{bmatrix} \quad \text{Cost}$$

$$W = \begin{bmatrix} (w_{11}\ w_{12} \cdots w_{1m_1}) \\ (w_{n1}\ w_{n2} \cdots w_{nm_n}) \end{bmatrix} \quad \text{Weight}$$

$$P = \begin{bmatrix} (p_{11}\ p_{12} \cdots p_{1m_1}) \\ (p_{n1}\ p_{n2} \cdots p_{nm_n}) \end{bmatrix} \quad \text{Maximum power.}$$

Then select a collection of n (i, j) pairs for all $1 \le i \le n$

minimizing $\Sigma_i f_i t_{ij}$

s.t. $\Sigma_i c_{ij} \le C_t$ Cost constraint

$\Sigma_i w_{ij} \le W_t$ Weight Constraint

$\Sigma_i p_{ij} \le P_t$ Conservative maximum power constraint.

C_t, W_t and P_t represent system bounds on cost, weight and maximum power usage. Any other constraints that can be reasonably expressed in an additive fashion could of course be appended to the above, also, different performance criteria could be used such as cost, possibly with a constraint representing real time data acquisition requirements, instead of the execution time criteri- given. In any event, the basic problem structure is the same.

There are, however, a host of difficulties encountered in attempting to realistically treat a system design problem in the preceding fashion as shall shortly be seen.

One of the first difficulties is that the preceding problem is not well stated from a computational standpoint especially when attempting to apply results from prior work on problems of this structure. To bring it into a "standard" conventional form let

$$x_{ij} = \begin{cases} 1 & \text{if implementation } j \text{ is chosen for the } i\text{th subtask} \\ 0 & \text{otherwise} \end{cases}$$

Then restate the problem as

$$\text{Min} \quad PX$$

$$\text{s.t.} \quad RX \leq B$$

$$SX = 1$$

$$x_{ij} = 0, 1$$

with $S = \begin{bmatrix} e_{m_1} & 0 & \cdots & 0 \\ 0 & e_{m_2} & \cdots & 0 \\ 0 & 0 & \cdots & e_{m_n} \end{bmatrix}$ and e_{m_i} is a row vector of m_i units.

In the above, P is a row vector reflecting performance, R an appropriately dimensioned matrix containing resource usage data with S denoting the requirement that exactly one implementation be selected per subtask.

Problems of this type are termed multiple choice (MC) programming problems.

MULTIPLE CHOICE PROBLEM - BALINTFY HEURISTIC

The MC problem has been studied to a moderate extent although not as much as the general (1,0) problem. Results from these studies as well as insight gained from the lattice structure study of our previous report have convinced us that obtaining true optimal solutions is not worth the computational effort evidently required especially in view of the likely vagueness of the avilable data. For near optimal

solutions a technique originally developed by Balintfy [5] is available and gives promise of possessing the efficiency necessary for the solution of large problems. Some modification is required for our application and this will be discussed shortly. First the Balintfy approach will be described.

Basically, it consists of two distinct types of iterations. One proceeds, similar to the dual simplex method in linear programming, from the best performing but usually most infeasible solution toward feasibility simply by interchanging (setting the incoming variable to 1 and the present to 0) the variable within each multiple choice group that possesses the best change in performance/cost ratio (minimum) from the present variable. This is done within the most violated resource constraint until feasibility for all the resource constraints are obtained. At this point, the second type of iteration begins consisting of a maximal decrease in the objective function within an MC group as long as feasibility is maintained. These "reverse" iterations are continued until no more feasible changes are possible. The varibles that have been set to 1 at this point define the desired near optimal solution. Studies by Balintfy and some trial runs by the author on simple trade off problems indicate that actual optimality is obtained quite often.

The existence of an efficient solution technique is important and as attention is now addressed to the deficiencies encountered in using the MC approach an upper-ost objective will be to retain the structure that allows application of the Balintfy heuristic.

DEFICIENCIES OF THE MC FORMALISM

Disadvantages of the MC approach include:

1) Implementations are rarely "dedicated" as such to a single subtask. More likely there may be portions of the total system that are utilized by several subtasks. This is clearly the case in microprogramming. "Subsystems" must be considered in any trade off formulation purporting to be realistic.

2) While providing some treatment of pure hardware or pure software trade offs, the MC problem as stated previously does not properly deal with the vital mixed case in which choices exist between both hardware and software implementations of a subtask.

3) The MC approach in effect requires a system synthesis procedure vaguely reminiscent of an assembly line in that an arbitrary selection of implementations should be permissable with a negligible or constant assembly cost. In other words widely varying interconnection costs among different combination of implementations are not taken into account in the MC problem as it has been stated. A constant interconnection cost can easily be accounted for by initially deducting it from C_t.

4) Finally a rather large amount of data is required of which a considerable portion is usually only imperfectly known.

These four areas appear to be the major difficulties precluding the practical use of the MC formulation. In the remainder of the report we shall be concerned with ameliorating these difficulties. Surprisingly enough it appears that, to a large extent, these problems can be circumvented while still retaining the structure that allows rapid near optimal solutions.

SUBSYSTEM CONSIDERATIONS

Subsystems are characterized by two items, their resource usage and their membership (i.e. the set of implementations using a particular subsystem). As far as resource usage is concerned from this point on only cost will be considered. Generalization to multiple resource usage is relatively straight forward.

For notational purposes the symbol s_q will represent cost when used in a resource constraint and

$S_q = [(i_1^q, j_1^q), \ldots\ldots, (i_{1_q}^q, j_{1_q}^q)]$; indicates the membership set when specifying indices.

Next by introducing variables

$$y_q = \begin{cases} 1 \text{ if subsystem } S_q \text{ is used} \\ 0 \text{ if not} \end{cases}$$

and then treating subsystems in effect as a type of "set up cost" as in operations research [7] the trade off problem can be restated as

Objective: $\text{Min} \left\{ \sum_{i=1}^{n} \sum_{j=1}^{m_i} f_i t_{ij} x_{ij} + \sum_{k=1}^{Q} y_{1k} \right\}$

s.t.

Resource: $\sum_{i=1}^{n} \sum_{j=1}^{m_i} c_{ij} x_{ij} + \sum_{k=1}^{Q} s_k y_k \le C_t$

Subsystem Structure $k^{\#}|S_k|$:

$$\sum_{(i,j)\in S_1} x_{ij} - M_1 y_1 \le 0$$

$$\sum_{(i,j)\in S_Q} x_{ij} - M_Q y_Q \le 0$$

These force $y_k = 1$ if any $x_{ij} \in S_k$ are 1. On the other hand, if all $x_{ij} \in S_k$ are 0 the MIN operation $\Rightarrow y_k = 0$.

Multiple Choice Constraints:

$$\sum_{j=1}^{m_1} x_{1j} = 1$$

$$\vdots$$

$$\sum_{j=1}^{m_n} x_{nj} = 1$$

$$y_{11} + y_{12} = 1$$
$$\vdots$$
$$y_{Q1} + y_{Q2} = 1.$$

The use of the y_{k2} allows retention of the MC structure. The preceding is quite similar to the type II problem stated in our previous report, however, here it has been generalized to allow unrestricted access to a subsystem by any implementation.

There is of course a price to be paid for all the maneuvering just shown. In particular, there are the additional subsystem structure constraints to process as the most obvious liability. The main difficulty, however, is the fact that an optimal solution is implicitly assumed while the techniques we plan to use generally yield near optimal solutions. It is unreasonable not to expect an increased error in view of the somewhat artificial manner in which the problem was forced to fit the MC formulation. Small problems involving subsystems that were tried by hand have worked out well but the suspicion is that for larger problems this would not be true.

Another approach to the subsystem problem is to modify the Balintfy heuristic to directly take subsystems into account in a manner similar to the yet to be described modification used for the mixed problem. This approach to the subsystem problem is however not yet in a sufficient state of development for presentation here.

HARDWARE/SOFTWARE - THE MIXED PROBLEM

Up to now we have dealt only with applications of the MC formalism to the so called pure problem i.e. trade-offs possible among hardware only or software only. This section treats the mixed problem occurring when a choice as to a software or a hardware implementation occurs within an MC group.

To do this, the characteristics of a software implementation must be examined more closely. What is needed is performance and resource usage information for the software implementation. As far as resource usage is concerned we will, for the moment at least, utilize a cost/word of memory used basis. Later in this report memory modularity will be considered. It is the performance area, however, that requires the most adjustment when considering a software implementation. The problem is that the performance criteria considered (execution time) is dependent on other implementation performance. This occurs simply because a software implementation consists of a set (eventually) of hardware instructions and the execution time of most of these instructions is typically unknown prior to solution of the problem. Consider a software implementation w_{ij} consisting of a collection of instructions. This can be reduced to a set of relative frequency coefficients, g_k, one for each distinct instruction, k. Suppose we let t_i represent the (variable) execution time for each instruction. Then the total execution time

$T_{ij} = \sum_{k \epsilon w_{ij}} g_k t_k = t_{ij}$ where (i, j) represents the indices of the software implementation in the MC formulation. But $t_k = \sum_{j=1}^{m_k} t_{kj} x_{kj}$ resulting in the objective function becoming

$$\text{MIN} \left[\sum_{i=1}^{n} \sum_{j=1}^{m_i} f_i t_{ij} x_{ij} + \sum_{(i,j) \epsilon W^t} f_i x_{ij} \cdot \left(\sum_{k \epsilon W_{ij}} g_k \cdot \left(\sum_{j=1}^{m_n} t_{kj} x_{kj} \right) \right) + \sum_{l=1}^{Q} y_l \right]$$

$(i,j) \notin W^t$

Here W^t represents the set of all w_{ij} that are software implementations. Note that nothing implies $(k,j) \notin W^t$ reflecting the possibility of software implementations using other software implementations as subroutines. The overall mathematical effect of mixed implementation choices is to add variable product terms to the objective function.

Variable Product Terms - Classical Approach to the Mixed Problem

The presence of the product terms apparently results in an optimization problem that is not longer in the MC class indeed no longer even in the strict "linear" (1,0) class. However, by using some more tricks from operations research the problem can be perturbed at least to an artificial MC structure.

For each product term $\prod_{j \in T} x_j$ with T consisting only of distinct indices, replace the product term with a new variable x_T and for each new variable append two new constraints of the following form:

1) $$x_T \geq \sum_{j \in T} x_j - |T| + 1$$

2) $$x_T \leq \frac{1}{|T|} \sum_{j \in T} x_j .$$

The effect of these new constraints is that 1) is redundant until $\prod_{j \in T} x_j = 1$ at which point 1) forces $x_T \geq 1$ with 2) implying $x_T \leq 1$ thus $x_T = 1$ as desired. When $\prod_{j \in T} x_j = 0$ at least one $x_{j \in T} = 0$ and constraint 2) forces $x_T < 1$ or $x_T = 0$. Usually some additional artificial variables must be added in order to retain the MC structure. In our application product constraints of the 2) form might be eliminated as the Min operation should give the same result.

The computational price paid for this approach to the product variable problem is considerable. When a substantial number of software implementations is involved, a large number of product constraints can be required, particularly if any amount of subroutine calls are used. In addition, the extra artificial variables needed will usually have a zero coefficient in the objective function giving poor results when the Balintfy heuristic is used. The next section discusses another approach to the product problem.

Modified Balintfy Approach to the Mixed Problem.

Each t_{ij} value in the objective function ideally reflects the performance of the (i, j) implementation. However, when product terms are eliminated with the preceding classical approach, t_{ij} may not directly reflect the performance of a software implementation, rather this performance will be eventually reflected if the artificial constraints are satisfied. To correct this, it is necessary to make the t_{ij} term for the software implementation directly reflect this performance. The primary difficulty encountered is that t_{ij} is likely to be dependent upon other as yet not finally determined execution times. Roughly, this is circumvented by using the execution times chosen or in use in the last iteration since these would correctly reflect the change in performance if a software implementation is selected in the next iteration. There are two cases to consider. One occurs when there are no software implementations in the current trial solution. In this case it is necessary to update t_{ij} if and when an iterative change is made in an instruction execution time that is used by software implementations. The second case occurs if there is at least one software implementation in a trial solution. Then, in addition to t_{ij}

updates required in the first case, a modification to some of the relative coefficients in F is necessary. This is true simply because, in effect, additional software is being used (actually "is being considered").

A computer program utilizing this latter approach is currently in preparation. Indications are that manipulative requirements will be far less than that necessary using the classical approach and storage requirements about 50% less. Most of the extra storage consists of certain tables and inverted file structures used to facilitate rapid access to the software implementation descriptors

We are also developing, as mentioned, an approach to the subsystem problem along somewhat similar lines and hope to have this ready for our next report.

Memory Modularity in the Mixed Problem

In the mixed problem discussion resource requirements for software implementations were on cost/word framework. This of course does not adequately reflect the situation. Memory typically comes in modules and it is not generally true that the cost of N memory modules is the same as N times the cost of one module.

With a little ingenuity modularity can easily be introduced into the MC formulation. Suppose an upper bound is known on the number of modules required (possibly obtainable from the number of address bits and extension register size for example) then

Let $$z_l = \begin{cases} 1 \text{ if N modules are used} \\ 0 \text{ otherwise} \end{cases}$$

clearly $\sum_{l=1}^{U} z_l = 1$ where U is the upper bound. Now alter the mixed problem MC formulation as follows:

1) Add one new constraint of the form

$$\sum_{(i,j)\in W^t} m_{ij} x_{ij} - \sum_{l=1}^{U} m_l z_l \leq 0$$

m_{ij} is the amount of memory required for the (i, j)th software implementation and m_l is the amount of memory available in l modules. Note the use of the term "available". If "fixed" software not directly involving trade-off considerations is part of the system its memory requirements must be deducted from the m_l.

2) Set the cost coefficients of the software implementation in the cost resource constraint to zero.

3) Add to the resource cost constraint the sum

$$\sum_{l=1}^{U} c_l z_l \quad \text{with } c_l \text{ representing the cost of l modules.}$$

4) Append to the objective function the sum

$$\sum_{l=1}^{U} l z_l .$$

5) Set in the additional MC structure constraint

$$\sum_{l=1}^{U} z_l = 1.$$

Contraint 1) forces all z_l with insufficient memory to zero. Then any $z_l=1$ with m_l having sufficient or more than sufficient memory will cause 1) to be satisfied. On the other hand, the MIN operation will (hopefully) set the lowest possible weighted $z_l = 1$ as desired. Simultaneous inclusion of other resource usage by the modules such as weight and power is easily done by reflecting their usage in the coefficient of z_l in the appropriate resource constraint.

The preceding is not quite so artificial as some of the previous maneuvers have been and only one constraint has been added. Even so the Balintfy heuristic can again likely be modified to take modularity directly into account. In fact, the

reader may have noticed the similarity between the artificial constraints introduced in modularity, mixed and subsystem considerations.

INTERCONNECTION COSTS

The interconnection cost problem is one of the limitations of the MC approach. In theory it can be exactly accounted for by the introduction of variable product terms in the resource constraint with appropriate cost coefficients reflecting the configuration defined by its product term. Classical or modified techniques could then be used to solve the problem. Practically this approach is limited by the large number of configuration combinations possible. This technique could be used if there are only a small number of configurations incurring significant interconnection costs.

IMPERFECT DATA - SENSITIVITY INFORMATION

The major practical limitation of the MC formulation is obviously the difficulty of obtaining the data. Quite probably implementations would have to be designed down to at least the logic equation level for hardware and the assembly or machine language level for software before truly accurate data is possible. Even then unforseen occurrences that might not show up until the actual manufacturing stage puts a question mark on the validity of even these data values.

It can also be argued that possibly the major cost would be incurred in the design phase itself especially if, as the MC approach seems to require, the way to proceed is to crank out as many implementations per subtask as possible. A little insight quickly shows that this is not the method to use rather judgement on the part of the designer should quickly eliminate a large portion of the implementation choices even before detailed design work begins. The MC approach should come into use when remaining implementation choices have possible features that might cause them to be used in certain situations. Those "certain situations" being specified by appropriate

operating constraints. However, information on such constraints is likely to be even more difficult to obtain than implementation data. It is almost absurd to expect existence of an absolute bound on any resource, especially cost. Instead acceptable ranges are more likely. It is here that sensitivity information obtained from varying the constraint bounds through these ranges is quite valuable. In operations research parlance, this is a type of parametric programming on the constraint bounds. For a single resource constraint the bound itself can be the parameter. However, for the case of multiple resource constraints a trifle more involved relationship is usually necessary where all resource bounds are expressed as a function of a single parameter and the desired information is the objective function value with its solutions as this parameter is varied.

For the MC formulation it appears relatively easy to modify the Balintfy heuristic to produce this bound sensitivity information. In fact, for the single resource constraint case discussed each reverse iteration produces one of the desired sensitivity values. Introducing parametric considerations among several resource bounds is the next obvious step and should be possible with the Balintfy approach.

Of course other sensitivity information in addition to the bound variations such as that derived from changes in F, T and C values is useful. Development of methods to obtain this information is tentatively planned for the future.

Sensitivity approaches are best when most of the data can be considered "reliable". When this is not true, about the only resource left is random variable techniques. A fair amount of work in linear programming under uncertainty has been done notably by Charnes. However, corresponding results in the (1, 0) case are not well known.

Investigation in this area for the MC formulation have only been contemplated not initiated.

In the final analysis a good design requires a great deal of research effort in order to obtain pertinent, reliable design data. There is simply no way around this fact although it occasionally can be sidestepped a little.

REFERENCES

1. Balas, E., "An Additive Algorithm for Solving Linear Programs with Zero-One Variables", Operations Research 13, pp. 517-545, 1965.
2. Chandy, K. M. and C. V. Ramamoorthy, "Optimization of Information Storage Systems", Information and Control 13, pp. 509-520, 1968.
3. Glover, F., "A Multiphase-Dual Algorithm for the Zero-One Interger Programming Problem", Operations Research 13, pp. 879-919, 1965.
4. Gue, R.L., "A Decomposition Principle for the Zero-One Programming Problem", Tech. Report CP67102 Computer Sciences Center, SMU.
5. Balintfy, J.L. "Menu Planning by Computer" Communications of the ACM, April, 1964
6. Liggett, J. C., "A General Multiple Choice Formulation of an Algorithm by Balintfy", Tech. Report CP-68004, Computer Sciences Center, SMU.
7. Healy, W. C. Jr., "Multiple Choice Programming," Operations Research, Vol. 12, 1964.
8. Gue, R. L., Cain, K. C., and Liggett, J. C. "Analysis of Algorithms for the Zero-One Programming Problem," Communications of the ACM, December 1968.

ENCLOSURE 7

ORGANIZING A SPECIAL PURPOSE COMPUTER USING THE FFT TO PERFORM SPEECH ANALYSIS IN REAL TIME

PAGE MISSING FROM AVAILABLE VERSION

PREFACE

The many useful applications for speech processing include bandwidth reduction, speeding of speech for the blind, slowing of speech for retarded children and language instruction, frequency division for aid to the partially deaf, automatic speech recognition, and automatic speaker recognition. The development of the Fast Fourier transform (FFT), reduction in price of digital integrated circuits, and the rapid advance of large scale integration (LSI) technology have made it economically feasible to construct special purpose computing devices which can perform speech processing tasks in real time. This thesis presents an organization of a digital speech analyzer using the FFT and cepstrum. Structure of the speech waveform, FFT considerations, and cepstrum pitch extraction are discussed as they relate to the implementation of a digital speech analyzer. The basic areas of organization options, the processor flow chart, sampling requirements, timing requirements and architecture are discussed in detail.

Submitted to the committee on December 6, 1969.

ABSTRACT

The overall objective of this thesis was to develop design parameters and the architecture for a special purpose digital computer for speech analysis.

Section I of the thesis is an introduction which discusses briefly some of the areas of application for speech processing equipment. Section II provides a very brief discussion of the structure and characteristics of speech which influence the specifications for a speech analyzer.

Some background to the fast Fourier transform (FFT) and an explanation of its operation is given in Section III. Also in Section III some of the properties of the FFT are discussed as they relate to hardware design considerations.

An explanation of the uses of Cepstrum analysis for the purpose of pitch extraction is given in Section IV.

Section V-A discusses advantages of building special purpose hardware for applications such as a speech processor. A review of basic organization options is given in Section V-B. A simplified flow chart of a speech analyzer using the FFT and cepstral analysis is developed in Section V-C. Section V-D presents a development of the sampling requirements as imposed by properties of the

speech waveform and the FFT.

From the sampling requirements and the flow chart, Section V-E developes the number of memory accesses and real multiplies and additions required for each block of the flow chart. From these results the speed requirements for the memory, adder, and multiplier are derived.

The architecture of the speech analyzer is developed in Section V-F and a summary of the hardware implementation is given in Section V-G.

Section VI discusses very briefly other possible approaches to speech analysis and Section VII presents some recommendations for further investigations. A conclusion is presented in Section VIII.

TABLE OF CONTENTS

SECTION			PAGE
I	INTRODUCTION		1
II	STRUCTURE OF THE SPEECH WAVEFORM		5
III	FFT DESIGN CONSIDERATIONS		9
	A	GENERAL	9
	B	THE FFT ALGORITHM	11
	C	FFT BASE CONSIDERATIONS	16
	D	MODULO 4 ALGORITHM	20
	E	THE IN-PLACE AND OUT-OF-PLACE ALGORITHM	21
	F	COEFFICIENTS	22
	G	SAMPLING	26
	H	WINDOWING	28
IV	CEPSTRUM PITCH DETERMINATION		31
V	IMPLEMENTATION OF ANALYZER HARDWARE		36
	A	GENERAL	36
	B	THE BASIC ORGANIZATION OPTIONS	37
		1. The Sequential Processor	37
		2. The Cascade Processor	39
		3. The Parallel Processor	39
		4. The Array Processor	41
	C	THE PROCESSOR FLOW CHART	41
	D	THE SAMPLING REQUIREMENTS	44

SECTION			PAGE
	E	THE TIMING REQUIREMENTS.	49
		1. General.	49
		2. Data Buffering and Weighting of Input Data	49
		3. The FFT Algorithm.	50
		4. The Separation Algorithm	54
		5. Square Magnitude	57
		6. Weighting of the Log of the Power Spectrum	58
		7. Cepstrum Peak Detector	58
		8. Data Output.	60
		9. Summary of Timing and Speed Requirements	60
	F	THE ANALYZER ARCHITECTURE.	65
	G	SUMMARY OF HARDWARE IMPLEMENTATION . .	73
	H	SIMULATIONS.	74
VI		OTHER APPROACHES TO SPEECH ANALYSIS BY DIGITAL TECHNIQUES	79
VII		RECOMMENDATIONS FOR FURTHER INVESTIGATIONS	83
VIII		CONCLUSION	86
		REFERENCES	87
		VITA	89

LIST OF FIGURES

FIGURES		PAGE
1	Schematic of FFT In-Place Algorithm	13
2	Schematic of FFT Out-of-Place Algorithm . .	23
3	Unit Circle Defining FFT Coefficients . . .	25
4	A Real Signal and Its Power Spectrum Displayed in the FFT Algorithm Format.	27
5	Representation of the Folding Frequency . .	27
6	The Leakage of Energy from One Discrete Frequency Into Adjacent Frequencies Resulting from the Analysis of a Finite Record.	29
7	Relationship of the Cepstrum to the Time Waveform and the Spectrum	35
8	The Functional Block Diagram of a Sequential Fast Fourier Transform Processor . . .	38
9	The Functional Block Diagram of a Cascade Processor	38
10	The Functional Block Diagram of a Parallel Processor	40
11	The Functional Block Diagram of an Array Processor	40
12	Flow Chart of a Speech Analyzer Using FFT .	42
13	Effects in the Frequency and Cepstrum Domains from Bordering with Zeros	47
14	Equations for the Separation Algorithm for N = 16.	56
15	Flow Chart for Determining Pitch Peak and Voicing Condition	59

FIGURES

		PAGE
16	Memory Access Time Requirements.	62
17	Speed Requirements for Multiplier vs Adder.	63
18	FFT Speech Analyzer Organization	66
19	Speech Analyzer Registers and Arithmetic Unit	67
20	Flow Chart and Equations for a Mod 4 Pass Through The Arithmetic Unit.	70
21	Partial Flow Chart for FFT with One Multiplier and One Adder	71
22	FFT Memory Timing.	72
23	Simplified Flow Chart of Simulation Driving Program.	75

LIST OF TABLES

TABLES		PAGE
1	Real Multiplications and Additions Required in Performing Base 2, 4, 8, and 16 FFT Algorithms	18
2	Summary of Timing Requirements.	51

SECTION I

INTRODUCTION

For several decades now man has been designing and building machines which process language directly. Some of these machines are the telephone, radio, phonograph, tape recorder and the entire spectrum of the modern communications industry. But these machines deal only with the time waveform (or causal acoustic aspects) of speech, not with its underlying structure.

With the emergence of the digital computer we have machines available that can deal with systems of rules. It is inevitable that we try to use these machines for speech processing to analyze, transform, and synthesize speech. The development of the Fast Fourier Transform (FFT), reduction in price of digital integrated circuits, and the rapid advance of Large Scale Integration (LSI) technology have indicated that small special purpose computing devices may be built which can perform speech processing tasks in real time. These special purpose computing devices may operate as stand alone processors or on-line with a general purpose digital computers. This thesis discusses a special purpose computer configuration which could be used as a speech analyzer to find the

spectrum, pitch fundamental, and the voicing conditions of speech in real time. In most speech processing applications the analyzer is only part of the system with other parts being a speech synthesizer, pattern recognizer and/or other types of signal processing. However, in any speech processor, a speech analyzer is a part of the system.

There are many useful applications for speech processing. A list of major applications is given below. This is certainly *not* a *complete* list but *gives* a *good* idea of the possible uses of speech processing.

1. To facilitate more faithful and efficient transmission of speech and storage of speech signals.
2. Speeding of the syllabic rate of speech for recorded books for the blind.
3. Slowing of speech for retarded children and foreign language instruction.
4. Frequency division for aid to the partially deaf.
5. Automatic speech recognition.
6. Automatic speaker verification.

Speech processing is used to reduce the redundancy of the speech wavefore (as in vocoders). This reduces the bandwidth requirements for transmission of speech and reduces storage requirements for speech in applications such as audio response units.

Speeding of the syllabic rate of speech is used for recording written material for the blind who cannot read ordinary books and magazines. With "speed hearing" of recordings of accelerated speech, the blind person can increase the rate of absorption of information from the outside world by an appreciable factor.

Slowing of speech, the inverse process, is being used for language instruction and as aids for teaching retarded children.

Frequency division methods are being studied as a possible aid to the partially deaf who have lost their high-frequency hearing.

Automatic speech recognition is being studied for applications such as direct audio programming of computers, control of machines, language translation, dictation, etc. While there are still many unsolved problems in automatic speech recognition (especially in recognition of continuous speech) there are in existence experimental working hardware systems utilizing a limited vocabulary of isolated words for such applications as ZIP code recognition, speech transmission over very narrow bandwidth channels, banking operations and probably others.

Studies are being performed on automatic speaker verification for such applications as verification of

individuals in communication systems, remote banking facilities, and identification of employees at plants involved in classified work.

SECTION II

STRUCTURE OF THE SPEECH WAVEFORM

This section will give a very brief discussion of the structure and characteristics of speech which influence the specifications for a speech analyzer. More detailed discussion may be found in reference 1.

From a spectral viewpoint speech can be described as voiced or unvoiced. A voiced sound (such as the vowels) has a harmonic spectrum and an unvoiced sound (such as s and ch) has a continuous spectrum. A voiced sound is produced by the vocal cords allowing puffs of air from the lungs to excite the resonant cavities of the throat, oral, and nasal passages. These puffs of air occur at a pseudo periodic rate which is called the pitch rate or the fundamental of the voiced speech sound. These pseudo periodic puffs of air produce a harmonic spectrum which is modified by the resonant cavities of the speech mechanism to produce the various sounds of speech. The quality of speech is very dependent upon the frequency and stability of the fundamental; however, the sound being produced is dependent upon the tuning of the resonant cavities.

Unvoiced excitation of the vocal tract is produced by a turbulent flow of air created at some point of stricture in the tract. An acoustic noise is thereby generated and provides an incoherent excitation for the vocal system.

A third source of excitation is created by a pressure buildup at some point of closure. An abrupt release of the pressure provides a transient excitation of the vocal tract. The transient excitation can be used with or without vocal cord vibration to produce voiced or unvoiced plosive sounds.

Thus, the three basic parameters which a speech analyzer will extract from the speech time waveform are:

1. Short time frequency analysis
2. Pitch fundamental frequency
3. Voicing condition

The speech waveform contains energy at frequencies as high as 8KHz and above. However, little if any information is lost if the speech signal is band limited at 4KHz. The normal telephone line bandlimits speech to roughly 200 Hz to 3200 Hz with very little loss of intelligibility and quality. Since the articulators; namely, the lips, jaw, tongue, and velum are changed at approximately a 20-25 Hz rate in normal speech, the energy distribution will change at that same rate. That is, the energy within say 100 Hz

bandwidth filters in the speech spectrum will vary at about 25 Hz or less. Thus the vocal mechanism can be considered as a quasi-stationary source of energy.

The conventional mathematical link between an aperiodic time function f(t) and its complex amplitude-density spectrum F(ω) is the Fourier transform-paid

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt \qquad (1.1)$$

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega t}d\omega \qquad (1.2)$$

For the transform to exist, $\int_{-\infty}^{\infty} |f(t)|dt$ must be finite. Generally a continuous speech signal neither satisfies the existence condition nor is known over all time. The signal must consequently be modified so that its transform exists for integration over known (past) values. Further, to reflect significant temporal changes, the integration should extend only over times appropriate to the quasi-steady elements of the speech signal. Essentially what is desired is a running spectrum, with real-time as an independent variable, and in which the spectral computation is made on weighted past values of the signal.

Such a result can be obtained by analyzing a portion of the signal "seen" through a specified time window, or weighting function. Since the articulators change at about

a 20-25 Hz rate the spectrum should be sampled at about a 50 Hz rate to satisfy the Nyquist sampling rate. Thus the length of the time window T should on the order of 20 ms.

The fundamental frequency of voiced speech lies normally in the range of 70-300 Hz for both male and female speakers. For most purposes a time window of 20 ms is adequate for tracking pitch; however, for purposes of analysis - synthesis in bandwidth reduction for communication purposes it is being found that pitch should be up dated every 10 ms.

The voicing condition of speech must be determined to indicate the presence of a harmonic type spectrum or a continuous noise type spectrum. For the unvoiced condition (continuous spectrum) there is no pitch fundamental. The voicing condition should be determined at the window rate T.

SECTION III

FFT DESIGN CONSIDERATIONS

A. GENERAL

An algorithm for the computation of Fourier coefficients which requires much less computational effort than was required in the past was reported by Cooley and Tookey in 1965 (2). This method is now widely known as the Fast Fourier Transform and has produced major changes in computational techniques used in digital spectral analysis, filter simulation and related fields.

The Fast Fourier Transform (FFT) is a method for efficiently computing the discrete Fourier transform (DFT) of a time series (discrete data samples). The efficiency of this method is such that solutions to many problems can now be obtained substantially more economically than in the past and many operations can be performed in real time whereas in the past this was only a dream. Thus the reason for the very great current interest in this technique. The FFT takes advantage of the fact that the calculation of the coefficients of the DFT can be carried out iteratively, which results in a considerable saving of computation time.

Specifically, if the time series consists of $N = 2^n$ samples, then about $1/2\ nN = 1/2\ N\log_2 N$ complex arithmetic operations [additions and multiplications] will be required to evaluate all N associated FFT coefficients. In comparison with the number of operations required for the calculation of the DFT coefficients with straight forward procedures (N^2) this number is so small when N is large as to completely change the computationally economical approach to various problems.

From the linearity property and complex symmetries of the Fourier transform pairs we can derive the important result of doing simultaneous Fourier analysis of two sets of real data (3). In the transform of two sequences of real values at the same time, one set is stored as the real component and the other set is the imaginary component of the complex vector to be transformed. The symmetry property allows separation of the two transforms. This is one of the properties of the FFT which makes its use very advantageous in a vocoder analyzer. One method of detecting the pitch of a speech waveform is to measure the time shift between the autocorrelation function of the speech waveform at the origin and the first major secondary peak. The autocorrelation function is equal to the Fourier transform of the power spectrum of the waveform. The power spectrum is defined as the sum of the squares of the real and imaginary

components of the Fourier transform of the speech waveform. Thus the power spectrum in the discrete case is a real sequence that can be Fourier transformed to give the autocorrelation function. The ability of the FFT to transform two sets of real data simultaneously is advantageous in a speech analyzer. In a speech analyzer an N point sequence representing the power spectrum of an N point input sequence can be transformed to obtain pitch information while another N point sequence can be simultaneously transformed to obtain spectral information.

B. THE FFT ALGORITHM

There are many detailed derivations of the FFT in the literature (2,4,5). Space will not be taken up here with a derivation of the FFT; however, it does seem appropriate to present a brief explination of what the algorithm does.

The equation for the discrete Fourier transform (DFT) is given as:

$$S(k) = \sum_{j=0}^{N-1} x(j)\, w^{jk}, \quad k=0,1,\ldots,N-1 \qquad (2)$$

where $S(k) \triangleq$ equally spaced spectral coefficients where $k=0,1,\ldots,N-1$

$x(j) \triangleq$ equally spaced time samples of a time waveform when $j=0,1,\ldots,N-1$

$$w = e^{\frac{-2\pi i}{N}}$$

$N = 2^n \triangleq$ number of points in the transform.

It can be seen from Equation (2) that N^2 complex multiplications and additions are required to compute a spectrum at N frequencies given N samples of a waveform X(j) when using the DFT. The FFT reduces the number of complex computations to (1/2)Nn multiplications and Nn additions.

The basic features of the algorithm will be explained referring to Figure 1. The figure is drawn for n = 4, N = 16, but it is possible to generalize from it to any value of N (5). Defining some terminology, let the original N data samples form a vector array X_0, with components $X_0(j)$, j = 0,1,..., (N-1) shown as points or nodes in a vertical column on the left of Figure 1. The sequence of the time samples are indicated by the position of the node in the column and the binary number to the left of it.

The next column of nodes, array 1, represented by X_1, with components $X_1(j)$, j = 0,1,...,N, is computed from the array X_0. In Figure 1, each of the nodes of X_1 is entered by a dashed arrow and a solid arrow, and within each node is an integer. The solid arrow brings a complex quantity from one of the nodes in the previous array and multiplies it by W^p where p is the integer in the circle. This complex product is added to the complex quantity brought via

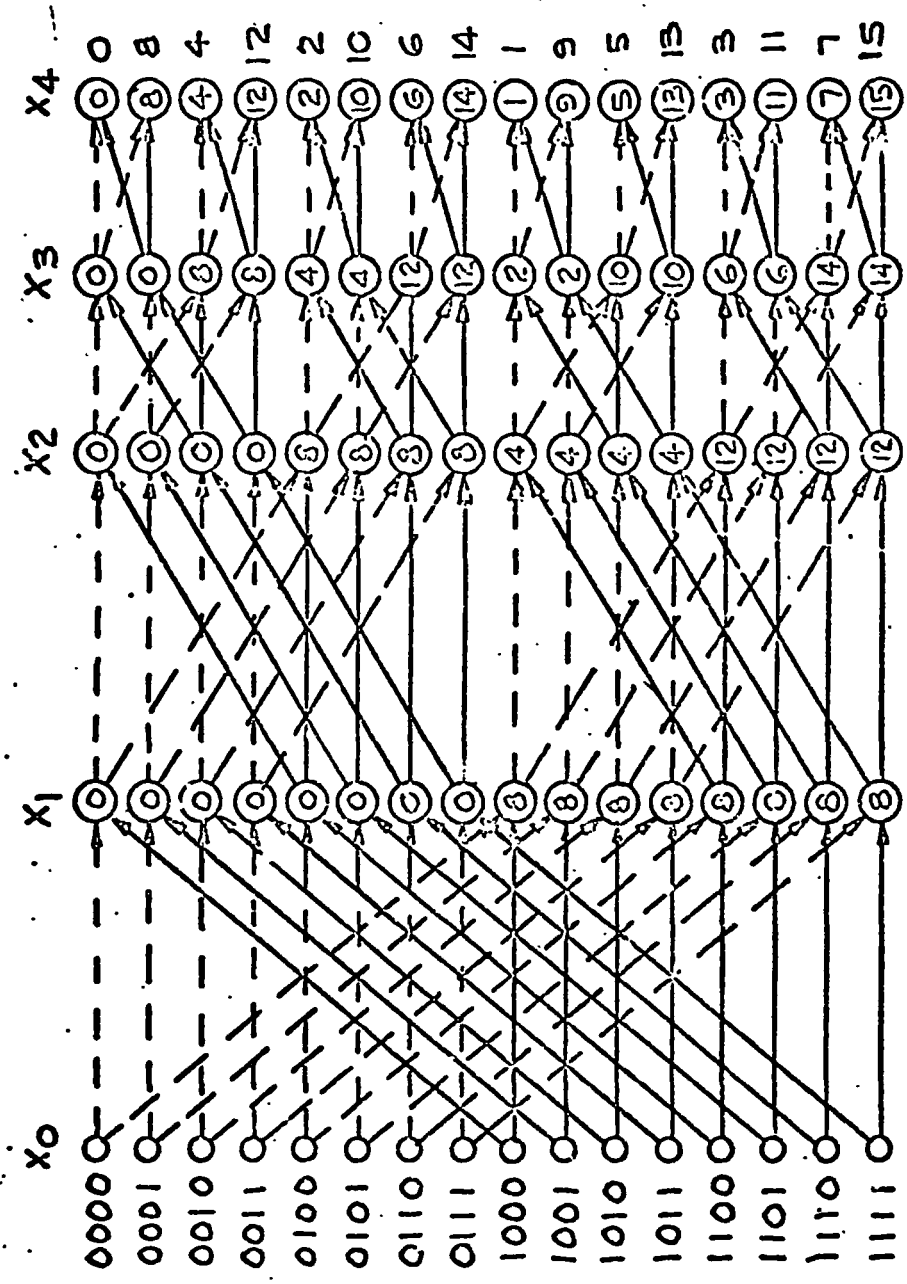


Figure 1

Schematic of FFT In-Place Algorithm

the dashed arrow from another node on the left. Thus the top node in array X_1 implies the computation:

$$X_1(0) = X_0(8)W^0 + X_0(0)$$

After array X_1 is computed, X_2 of points $X_2(j)$, is computed from X_1 in the same way, and the third and fourth arrays are similarly computed. In all, n arrays (in this case 4) are computed, and the last array contains the N values of the spectrum. These do not appear in array X_n in order, however. To find the k^{th} spectrum point, write the binary number k with the order of bits reversed, giving k', and look in the k'^{th} location, $X_n(k')$, for the k^{th} spectrum point.

Now that the basic operation of the algorithm has been explained, rules will be given to construct Figure 1 for any value of n.

1. The N data points of the array X_0 with vertical location $j = 0,1,\ldots,(N-1)$ are drawn. The locations j are given as binary numbers (n bits) $j_{n-1}\cdots j_0$. The next array, X_1, and in general the array X_m, are drawn successively to the right, and the points in the array are also addressed as binary numbers. Nodes on the same horizontal level have the same binary address.

2. The number in the circle of the j^{th} node in the m^{th} array is found by:

a. writing the binary number j

b. shifting it by (n-m) places to the right (filling the newly opened bit positions on the left by 0's)

c. reversing the order of the n bits

Thus, the last element in array 2, $X_2(15)$, has address 15=1111 Scaling by n-2 to the right gives 0011, and reversing the order of the bits gives 1100=12.

3. In the m^{th} array, node j (in binary form $j_{n-1}\cdots j_1 j_0$) has a solid arrow drawn to it from a node in the $(m-1)^{st}$ array, whose location is the same as j except that bit j_{n-m} must be a 1. The dashed arrow comes from a node in the $(m-1)^{st}$ array whose location is the same as j except that bit j_{n-m} must be a 0. Note that this implies that there are two nodes in array m which are affected by the same pair of nodes in array m-1, and that no other nodes in array m are affected by either of the aforementioned nodes in array m-1. The two nodes in array m and the two nodes in array m-1 form a rectangle.

Note that the pair of nodes in the m^{th} array will have solid arrows coming from the same node in the m-1 array, and that the integers in the circles will differ by N/2, since bit j_{n-m} of the address of each node differs, and since rule 2 above says to scale by n-m places to the right

and reverse the order of bits. The multiplications required for these two nodes in the m^{th} array are thus negatives of each other since $W^{N/2} = -1$, and if the computations for both nodes are done together, half of the required multiplications can be saved.

C. FFT BASE CONSIDERATIONS

Cooley and Tukey stated in their original paper (2) that the Fast Fourier Transform algorithm is formally most efficient when the number of samples in a record can be expressed as a power of 3 (i.e., $N=3^n$), and further that there is little efficiency lost by using $N=2^n$ or $N=4^n$. (The base of the algorithm is defined as m where $N=m^n$.)

Later, however, it was recognized that the symmetries of the sine and cosine weighting functions made the base 4 algorithms more efficient than either the base 2 or the base algorithms (6). Bergland then carried the base 4 technique one step further to an even more efficient base 8 algorithm (

As the base of the algorithm increases it is possible to use symmetries of the sine and cosine weighting functions to reduce the number of arithmetic operations. As the base of the algorithm increases, however, the $2^n=N$ point transform becomes more involved (although using fewer arithmetic operations) and it becomes increasingly difficult to let N

be an arbitrary power of 2. For example, 128-, 256-, 1024-, and 2048-point transforms cannot be performed with the base 8 algorithm. However, it is possible to combine the three algorithms, thus preserving the versatility of the base 2 algorithms, while attaining the computational advantage of the base 8 algorithm. It is possible to perform as many base 8 iterations as possible and then finish the computation by performing a base 4 or base 2 iteration as required.

Bergland (7) has derived a set of equations comparing the computations required by base 2, 4, 8, and 16 algorithms for N points where N is any integer power of 2. The equations were derived using the property that no complex multiply is required when the exponent of W is 0 since $W^0=1$. The number of arithmetic operations for different FFT configurations shown in Table 1 were calculated from Bergland's equations assuming a record rate of 48 records per second.

In the design of an all digital speech analyzer it is necessary to use hard wire programming in place of software in order to keep within the speed limitations of memory and logic elements. In addition it is desirable to have the capability of changing the value of N. With these restrictions it seems appropriate to use a base 2 algorithm because of less complex control logic required and greater

Table 1

Real multiplications and additions required in performing Base 2, 4, 8, and 16 FFT algorithms

	M	N	Real Multiplies	Real Adds
Base 2	4	16	68	162
	5	32	196	418
	6	64	516	1026
	7	128	1284	2434
	8	256	3076	5634
	9	512	7172	12802
	10	1024	16387	28674
	11	2048	36868	63490
	12	4096	81924	139266
Base 4	4	16	36	146
	5	32	116	378
	7	128	836	2210
	8	256	4052	5122
	9	512	4867	11650
	10	1024	11268	26114
	11	2048	25604	57858
	12	4096	57348	126978
Base 8	4	16	25	146
	5	32	89	378
	6	64	260	930
	7	128	686	2210
	8	256	1710	5122
	9	512	4098	11650
	10	1024	9558	26114
	11	2048	21842	57858
	12	4096	49140	126978

Table 1 Continued

M	N	Real Multiplies	Real Adds
4	16	24	144
5	32	86	373
6	64	252	918
7	128	668	2182
8	256	1668	5058
9	512	4004	11506
10	1024	9348	25794
11	2048	21380	57154
12	4096	48132	125442

ease of changing the value of N. If N were fixed, then the additional complexity of control logic for a higher base algorithm may be justified.

D. MODULO 4 ALGORITHM

Even in a machine in which a base 2 algorithm is used it is possible to use a modulo 4 memory access algorithm to reduce the memory accesses by a factor of 2 without losing flexibility. A modulo i algorithm is defined as an algorithm which accesses a set of i data points (for use in a scratch pad memory) such that j iterations can be performed, when $2^j = i$, before returning results to the main memory. The number of accesses to main memory is reduced by the factor j at the expense of additional scratch pad memory and control logic.

The mod 4 algorithm accesses the proper set of 4 data points from memory such that two iterations can be performed (to calculate new values for the next two arrays), in place of one iteration, before the results have to be returned to memory. The price paid for a mod 4 algorithm is a very slight increase in complexity of control circuitry and scratch pad registers to hold 4 data words. This is a small price to pay when it can mean using one memory in place of two or of using a 1 μsec memory in place of a 0.5 μsec memory.

E. THE IN-PLACE AND OUT-OF-PLACE ALGORITHM

The algorithm shown in Figure 1 can be called an in-place algorithm because for N data samples only N complex data locations are needed in memory to perform the iteration of the m arrays. Referring to Figure 1 it is seen that the pair of data points in array m-1 are used to generate a pair of data points for the same address for array m. Thus a pair of data points from array m-1 can be accessed from memory, operated on, and then returned to the same location in memory as data points in array m. It should be noticed however from the figure that order of accessing data points differs for each array m. For example, the first access for array X_0 will require points 0000 and 1000. The first access for array X_1 will require points 0000 and 0100. The first access for array X_2 will require points 0000 and 0010 and so forth. This requires the controller which addresses memory to operate in a slightly different mode during the processing of each array, thus adding somewhat to its complexity.

A so called out-of-place algorithm is shown in Figure 2. Some comparison between Figures 1 and 2 will verify that both algorithms produce the same result. However, in the case of Figure 2 it is seen that two points accessed from array m-1 will not be placed back into the

same memory address after being operated on. Thus memory will be required for 2N complex points. An advantage of the out-of-place algorithm is that the points are accessed in the same order for each array m-1 and stored in the same order for each array m. Thus the controller addressing memory will simply run through the same loop for each array m processed. There is required, of course, control circuitry to swap pages of memory after the processing of each array.

The additional complexity of the controller for the in-place algorithm does not offset the additional memory required for the out-of-place algorithm and thus the former approach should normally be used. However, in some applications the speed of the processor may be limited by the speed of the memory in which case it is possible to use two separate memories so that the reading of old points from array m-1 and the storing of new points in array m can be done simultaneously. In a case such as this where memory is doubled anyway for speed advantage it would be desirable to use the out-of-place algorithm.

F. COEFFICIENTS

In Section III B, W is defined as exp $(-2\pi i/N)$ and a procedure was given for obtaining p where p is an exponent of W. Thus we have:

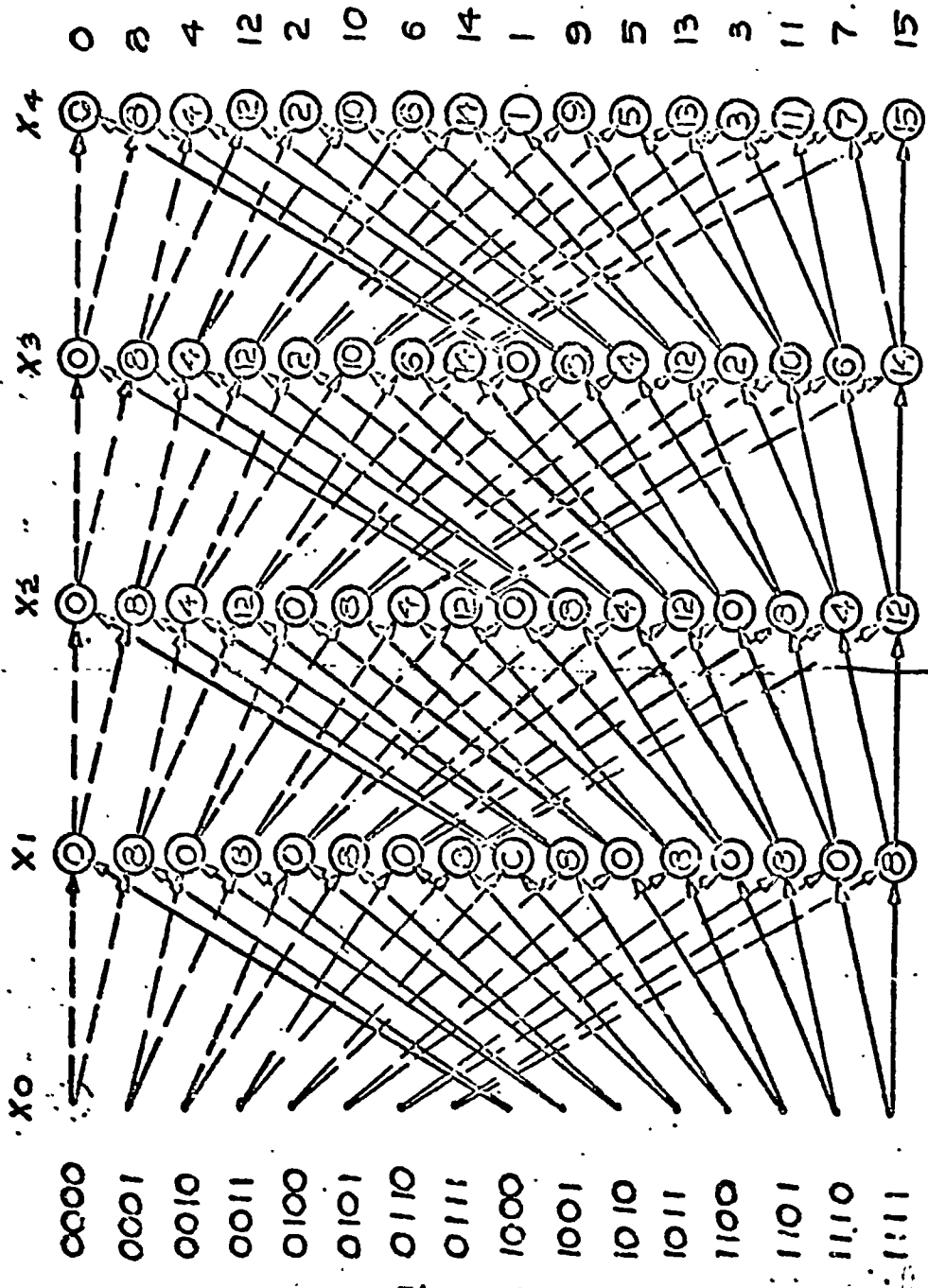


Figure 2

Schematic of FFT Out-Of-Place Algorithm

$$W = \exp(-2\pi i/N) = \cos(2\pi/N) - i \sin(2\pi/N) \quad (3)$$

and $$W^p = \cos(2\pi p/N) - i \sin(2\pi p/N) \quad (4)$$

For the cases where p=0 and p=N/2 we have:

$$W^0 = \cos 0 - i \sin 0 = 1 + i\,0 \quad (5)$$

$$W^{N/2} = \cos(2\pi N/2N) - i \sin(2\pi N/2N)$$
$$= \cos(\pi) - i \sin(\pi) = -1 + i\,0 \quad (6)$$

It is seen in Equations (5) and (6) that $W^{N/2}$ is the negative of W^0. This was pointed out in Section III B and can also be seen in Figure 3 which is the unit circle defining the FFT coefficients. The importance of this is that each node pair uses values of p which differ by N/2 meaning that one coefficient will always be the negative of the other. This property is used to reduce the number of complex multiplications by a factor of two. By referring to Figure 1 and 3 it can also be seen that because of this property only the coefficients in the upper half of the unit circle are needed for the processing of the FFT algorithm. Thus only sine and cosine values ranging from p=0 to p=(N/2)-1 must be available in storage. Discussion of techniques for further reduction in coefficient storage is given in Section V.

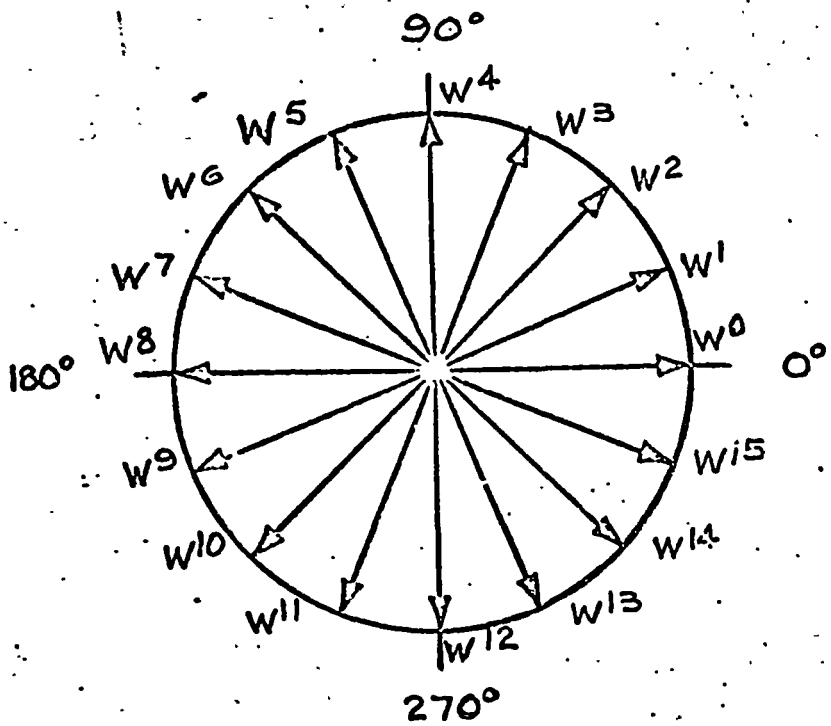


Figure 3

Unit Circle Defining FFT Coefficients

G. SAMPLING

The FFT operates on N evenly spaced samples of a continuous time waveform to produce N evenly spaced spectral coefficients. An example of a real-valued time series and its associated FFT (or DFT) is shown in Figure 4 (in place of showing the complex Fourier coefficients in Figure 4 the power spectrum is shown). The time series $x(k\Delta t)$ is assumed to be periodic in the time domain of period T seconds, so the set of Fourier coefficients is periodic over the sample frequency f_s. The fundamental frequency f_0 of the transform is equal to 1/T where T is the period of the window in seconds. T is equal to $N\Delta t$ where Δt is the sampling period. The sampling frequency f_s is equal to $1/\Delta t$.

The power spectrum will be symmetric about the folding frequency f_f where $f_f=f_s/2$. The Fourier Coefficients between N/2 and N-1 can be viewed as the "negative frequency" harmonics between -N/2 and -1. Likewise, the last half of the time series can be interpreted as negative time (that is, as occurring before t=0).

An example of the symmetry about the folding frequency is shown in Figure 5. Only the coefficients up to the folding frequency are useful since the spectrum from $f_s/2$ up to f_s is just the mirror image of the spectrum from 0 up to $f_s/2$. This agrees with the Nyquist criterion which

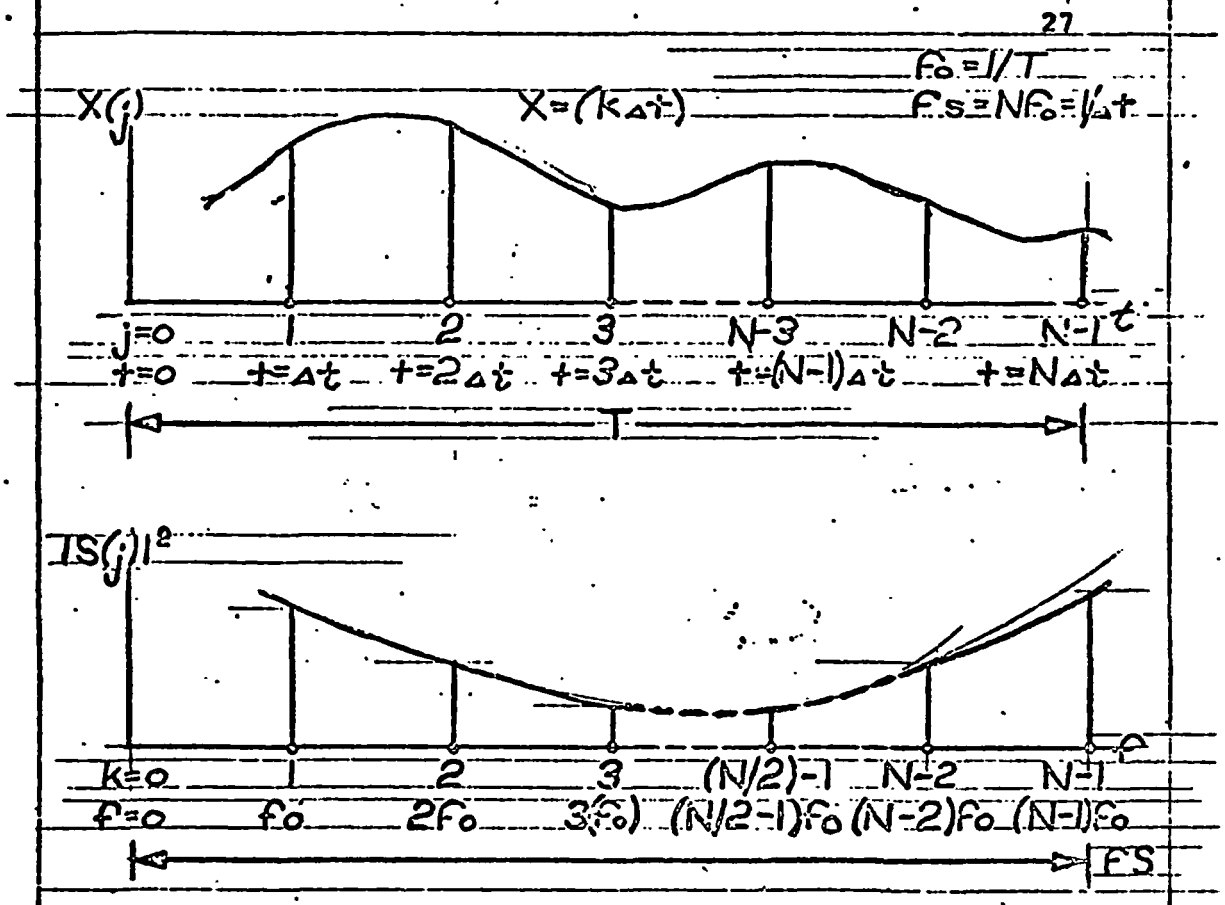


Figure 4

A Real Signal and Its Power Spectrum Displayed in the FFT Algorithm Format

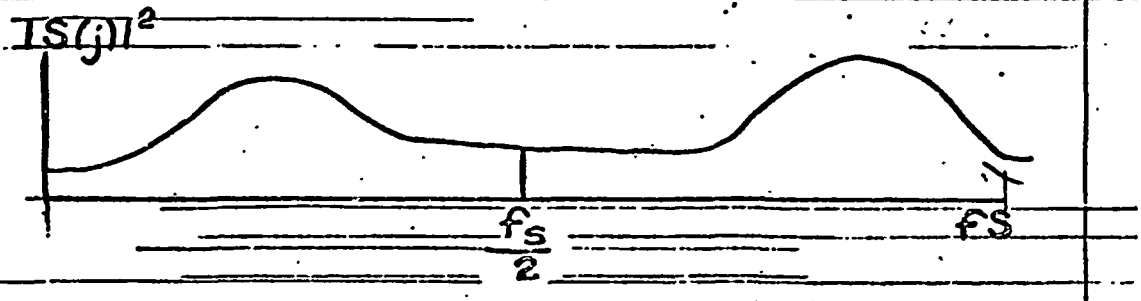


Figure 5

Representation of the Folding Frequency

says that a continuous signal must be sampled at least at twice the highest frequency in the signal to prevent alias error. Also implied is the fact that for an N point transform only N/2 useful spectral coefficients are obtained.

H. WINDOWING

A problem known as leakage (4) is inherent in the Fourier analysis of any finite record of data. The record is formed by taking data samples of the actual signal for a period of T seconds and by neglecting the signal before and after the period T. As shown in Figure 6 this is equivalent to multiplying the signal by a rectangular data window. If the continuous Fourier transform of the pure cosine wave in Figure 6a had been found, its contribution would have been limited to only one point on the frequency axis as shown in Figure 6d. However, the multiplication by the data window in the time domain is equivalent to performing a convolution in the frequency domain. Thus, this impulse function is convolved with the Fourier transform of the square data window, resulting in a function with an amplitude of the (sine x)/x form centered about f_1.

This function is not localized on the frequency axis and in fact has a series of spurious peaks called sidelobes. The objective is usually to localize the contribution

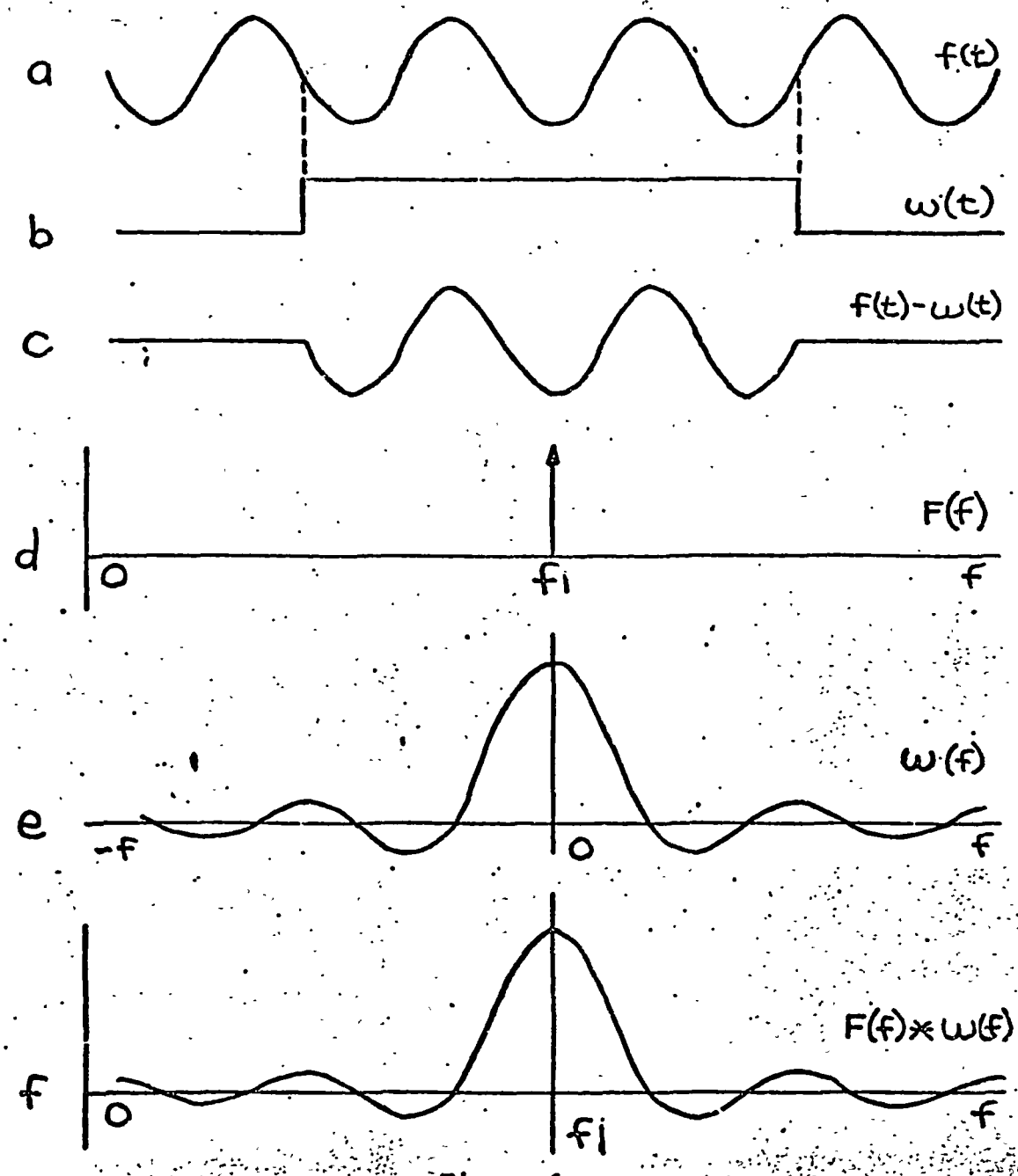


Figure 6

The Leakage of Energy from One Discrete Frequency into Adjacent Frequencies Resulting From the Analysis of a Finite Record

of a given frequency by reducing the amount of leakage through these sidelobes.

The usual approach consists of applying a data window to the time series, which has lower sidelobes in the frequency domain than the rectangular data window. There are a number of different data windows which have been discussed in the literature (8,9,10,11). One example is Tukey's "interim" data window. In this window, a raised cosine wave is applied to the first and last ten percent of the data and a weight of unity is applied in between. Since only twenty percent of the terms in the series are given a weight other than unity, the computation required to apply this window in the time domain is relatively small. However, if computation time required is not an overriding consideration, one can find a number of windows that give rise to more rapidly decreasing sidelobes.

The use of any data window other than the rectangular data window will also reduce the picket-fence effect by widening the main lobe of each spectral window. The picket-fence effect is ripple in the spectrum caused by the signal being analyzed not falling exactly on one of the discrete Fourier coefficients.

SECTION IV

CEPSTRUM PITCH DETERMINATION

The speech signal, f(t), is created by a source signal, s(t), which consists of puffs of air generated at the vocal cords, passing through the vocal tract. If this vocal tract is specified by its impulse response, h(t), the speech signal is the convolution of s(t) and h(t):

$$f(t) = s(t) * h(t) \tag{7}$$

If S(ω) and H(ω) represent the Fourier transforms of s(t) and h(t), respectively, then F(ω), the Fourier transform of f(t), is:

$$F(\omega) = S(\omega) \cdot H(\omega) \tag{8}$$

The power spectrum then consists of harmonics of the source signal frequency, the pitch. The spectrum is, therefore, also periodic with the period equal to the reciprocal of the period of the time waveform signal being analyzed. This fact makes it feasible to Fourier transform the power spectrum to obtain the desired period of the pitch of the signal.

This spectrum of the power spectrum is commonly called the autocorrelation function, $r(\tau)$ of the speech signal defined as:

$$\begin{aligned} r(\tau) &= \mathcal{F}[|F(\omega)|^2] \\ &= \mathcal{F}[|S(\omega)|^2 \ |H(\omega)|^2] \\ &= \mathcal{F}[|S(\omega)|^2] * [|H(\omega)|^2] \\ &= r_s(\tau) * r_h(\tau) \end{aligned} \tag{9}$$

where r_s and r_h are the autocorrelation functions of $s(t)$ and $h(t)$ respectively. Note that $r(\tau)$ consists of a convolution of the autocorrelation functions. This results in broad peaks, and in some cases, multiple peaks in $r(\tau)$. This implies that the autocorrelation method of pitch extraction is impaired by the effects of the impulse respon of the vocal tract, $h(t)$., upon the source signal, $s(t)$.

A solution is to separate the effects of the vocal tract and the source so that they are more easily identifiable. The way to achieve this is to take the logarithm of the power spectrum before Fourier transforming it:

$$\begin{aligned} \log |F(\omega)|^2 &= \log |S(\omega)|^2 \cdot |H(\omega)|^2 \\ &= \log |S(\omega)|^2 + \log |H(\omega)|^2 \end{aligned} \tag{10}$$

The Fourier transform preserves addition:

$$\mathcal{F}[\log |F(\omega)|^2] = \mathcal{F}[\log |S(\omega)|^2] + \mathcal{F}[\log |H(\omega)|^2] \tag{11}$$

The source and vocal tract effects are now additive rather than convolved as in the autocorrelation function. The peak corresponding to the source period can be emphasized by squaring the second spectrum. This function, $\{\mathcal{F}[\log|F(\omega)|^2]\}^2$, of the Fourier transform of the logarithm of the power spectrum is called the cepstrum. The term quefrency is used as the independent variable for the cepstrum as time is used for the time domain and frequency for the spectral domain. The units of quefrency are cycle ÷ cycles/sec = sec.

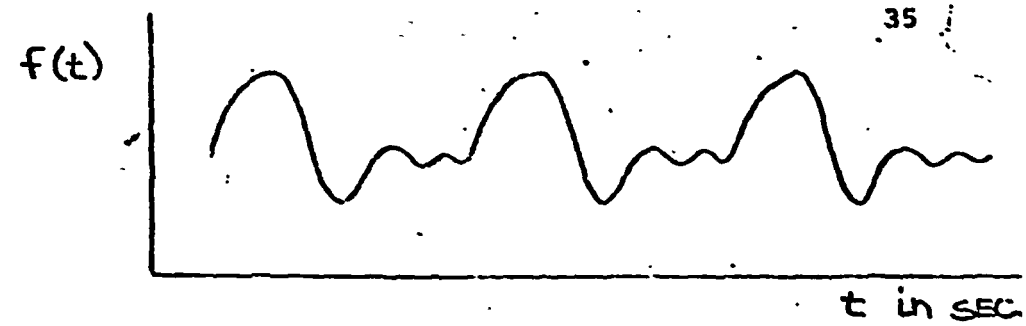
It has been determined by observation of computer simulation that hanning weighting of both the time samples and the logarithm of the powerspectrum have the effect of enhancing the pitch peak with respect to the amplitude of the adjacent lobes.

The hanning window (9) takes the form in the time domain of

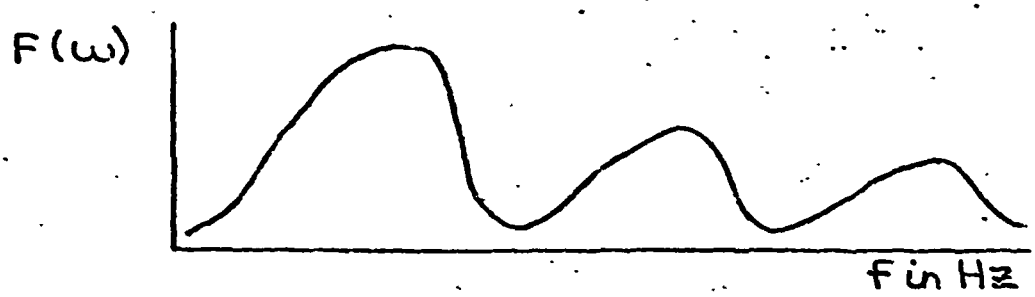
$$\frac{1}{2}\left(1 + \cos\frac{\pi k}{N}\right).$$

This function when multiplied by the original time function has the effect of smoothing the ends of the input sequence and joining them to zero. The hanning window is not necessarily the most effective window. The most effective window can be determined only by cut-and-try inquiry.

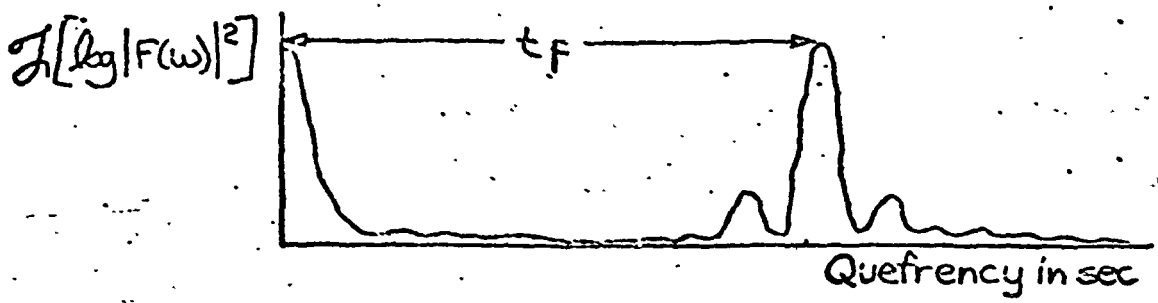
The relationship among the speech time waveform, the spectrum, and the cepstrum is shown in the sketch of Figure 7. For simplicity the continuous speech waveform is shown and the envelopes of the spectrum and cepstrum are shown in place of the discrete samples and coefficients. If t_p is the distance of the cepstral peak from the origin in seconds then the fundamental frequency of the time waveform is $1/t_p$.



a. RAW INPUT SPEECH



b. ENVELOPE OF THE SPECTRUM OF THE RAW INPUT SPEECH



c. ENVELOPE OF THE CEPSTRUM OF THE RAW INPUT SPEECH

Figure 7

Relationship of the Cepstrum to the Time Waveform and the Spectrum

SECTION V

EMPLEMENTATION OF ANALYZER HARDWARE

A. GENERAL

Software implementations of the FFT have reduced computation time for many problems by as much as two orders of magnitude. Even greater gains can be realized through special-purpose hardware designed specifically for performing the FFT algorithm in a specific application. Applications for special-purpose FFT processors result from signal processing problems which have an inherent real-time constraint or which involve off-line processing where the volume of data makes processing impractical unless a dedicated machine is used. Both of these requirements exist for a speech analyzer where real time processing may be required for bandwidth compression or automatic speech processing or large volume of data are required for processing for statistical analysis of speech. Bergland has stated that experience with the FFT processor built by Bell Telephone Laboratories indicates that the cost reduction resulting from special-purpose hardware is nearly as great as the reduction which came with the Cooley-Tukey algorithm (12). The Bell Telephone FFT signal processing system costs five

times less perhour than a large general-purpose computer while performing the fast Fourier transform algorithm twenty times faster. Thus, on the FFT part of the processing, a 100 to 1 cost saving is possible. As a result of this reduction in cost, people who had not even heard of the fast Fourier transform three years ago are now finding that they cannot get along without it.

B. THE BASIC ORGANIZATION OPTIONS

Several FFT processors have been built or are being built both as general purpose and as special purpose processors (13). The organization of an FFT processor is usually dictated by the performance and cost requirements and the technology assumed. Four families of machine organizations, which have appeared in some form in the literature (12), will be described.

1. The sequential processor - a simplified diagram of the sequential processor is shown in Figure 8. This organization is similar to that of a small general-purpose computer except that the table memory, data memory, arithmetic unit, and control unit can usually all operate concurrently. The same memory can be used to store the input data, the intermediate data, the intermediate results, and the resulting Fourier coefficients. Since only one basic

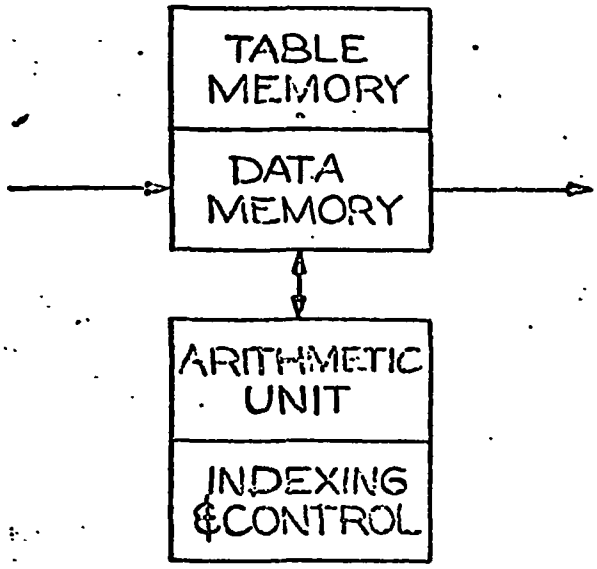


Figure 8

The Functional Block Diagram of a Sequential Fast Fourier Transform Processor

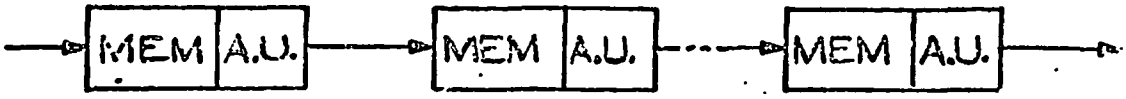


Figure 9

The Functional Block Diagram of a Cascade Processor

operation is involved and the accessing pattern is very regular, the amount of hardward involved can be realtively small.

2. The Cascade Processor - A simplified block diagram of the cascade processor is shown in Figure 9. To improve the speed of the processor, one arithmetic unit is used for each of the n arrays where $N=2^n$. Thus a pipeline type of operation is performed with n sets of data being operated on at once, but with output data being generated n times faster than the sequential processor. The execution time for the sequential processor described above is $B(N/2)\log_2 N$ where B is the time required to perform one basic operation. Since the basic cascade processor has n arithmetic units of the sequential processor type, its execution time is improved by a factor of $\log_2 N=n$ at the expense of approximately n times as much logic and memory. Obviously the amount of hardware required is much greater than for the sequention processor.

3. The Parallel Iterative Processor - A simplified block diagram of the parallel iteration processor is shown in Figure 10. By using N/2 arithmetic units, an entire array can be processed in one execution time B. Thus the execution time for the algorithm is $B(\log_2 N)$. This is an improvement in execution time over the sequential processor

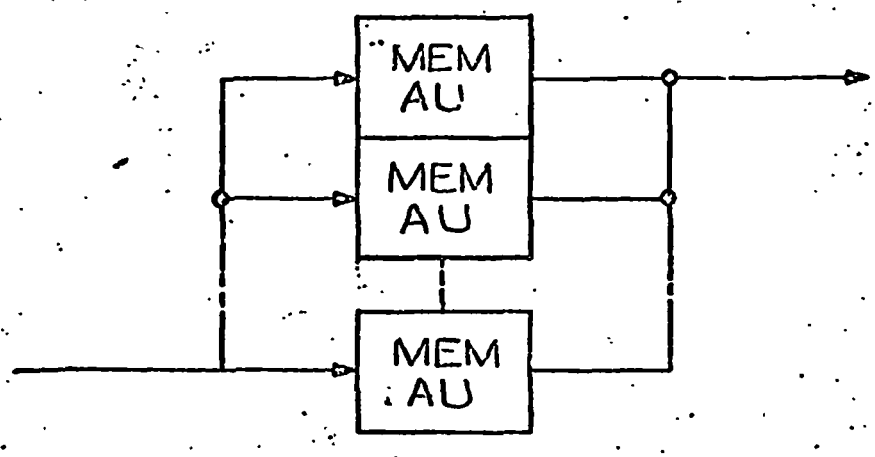


Figure 10

The Functional Block Diagram of a Parallel Processor

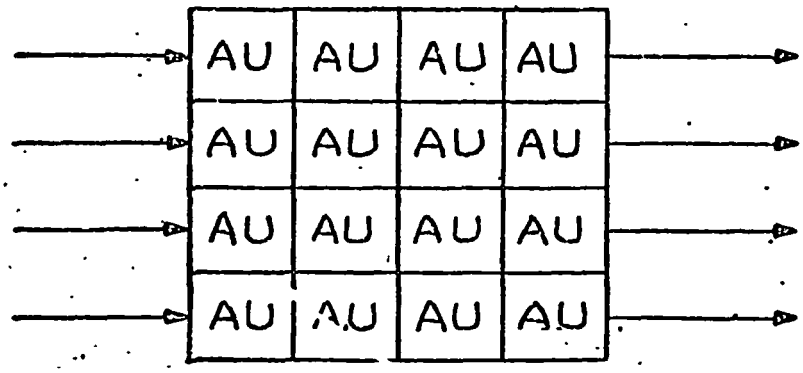


Figure 11

The Functional Block Diagram of an Array Processor

by a factor of N/2 at the expense of approximately N/2 times as much logic and memory.

4. The Array Processor - A simplified block diagram of the array processor is shown in Figure 11. The complete array processor would have (N/2)n arithmetic units so that by pipelining the arrays of data the effective execution time for the algorithm is simply the time B required for performing one basic operation. At this point in time the hardware requirement for an array processor is prohibitive.

C. THE PROCESSOR FLOW CHART

A simplified flow chart for the speech analyzer processor is shown in Figure 12. Provision is made for either a microphone input or an audio tape input to the processor. The input amplifier provides impedance matching and amplification between the input devices and the low pass filter. The low pass filtering provides a bandlimited signal which can be sampled without aliasing error. The cutoff frequency of the filter can be set at any point within the constraints of the 12,800 Hz sampling rate (which is derived in the next subsection). The pre-emphasis operation off sets the natural energy distribution of speech which is essentially flat up to about 1 KHz and then rolls off at about 6db per octave. The reason for the use of the pre-emphasis is to produce a

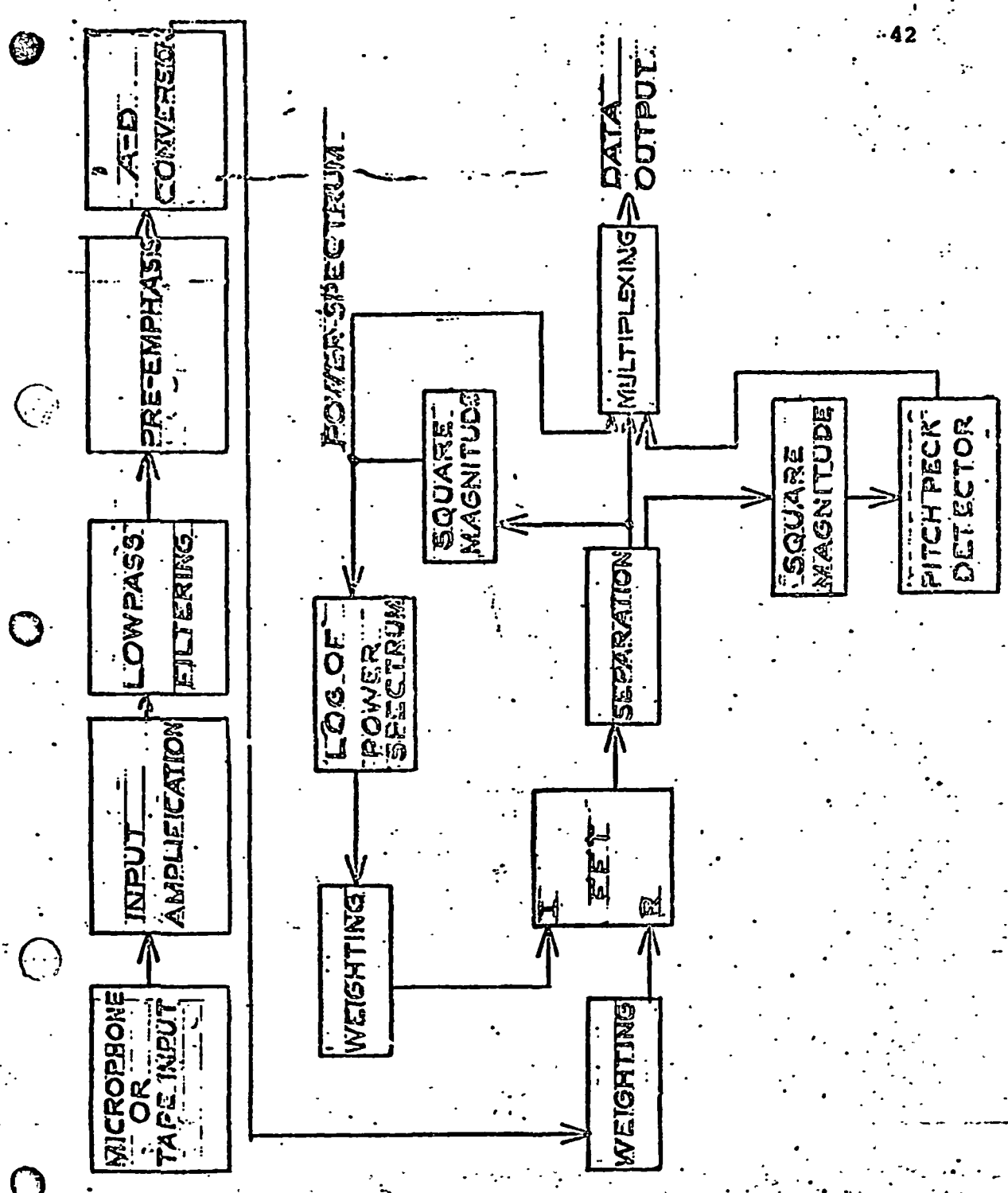


Figure 12

Flow Chart of a Speech Analyzer Using FFT

flat long-term energy distribution to ensure optimal usage of the dynamic range of the analyzer across the spectrum. The analog-to-digital converter codes the input analog speech waveform to a quantized sampled data form. Time waveform weighting is performed to enhance the spectral and cepstral information.

The fast Fourier Transform algorithm is normally applied to complex data, and has both real and imaginary inputs. Because the transform pairs have the properties of linearity and complex symmetry, it is possible to process two real inputs and separate the results at the output. The sampled data is entered at the real input, and the log magnitude of the resulting transform pairs is reentered at the imaginary input. At the FFT output, a separation routine divides these into a complex spectral output (transform of data points) and a complex cepstral output (transform of log magnitude of spectral output). The square magnitude is taken of the complex spectrum to produce the power spectrum and the square magnitude of the complex cepstrum is taken to produce the cepstrum. The log of the power spectrum is taken and weighted to produce the input for the imaginary input to the FFT. The pitch peak detector searches the cepstrum data for the cepstrum peak which determines the fundamental frequency. If no cepstrum peak

is found, then the speech sound is assumed to be unvoiced. The pitch data and the power spectrum data or the complex spectrum data is multiplexed to the output.

D. THE SAMPLING REQUIREMENT

Before discussing the sampling requirements it would be well to review some definitions already used and introduce some new definitions.

$f_s \triangleq$ the sampling rate in samples per second.

$T \triangleq$ the length of a record (or window) in seconds.

$f_0 \triangleq 1/T$, the cell width in cycles or the spacing of the Fourier coefficients on the frequency scale.

$\Delta t \triangleq 1/f_s$, the sampling period in seconds.

$q_0 \triangleq$ the cell width in seconds on the quefrency scale. $q_0 = 1/f_s = \Delta t$.

$P_{min} \triangleq$ the minimum pitch frequency detectable on the quefrency scale.

$N \triangleq 2^n$, the number of sampling points in a record or frame of data.

$f_f \triangleq f_s/2$, the folding frequency

Now the sampling specifications will be established to meet the requirements for speech processing. For a speech analyzer which is expected to perform speech analysis for different purposes in speech processing and speech studies, an absolute minimum high frequency cut off would be 4 KHz. In order to meet the Nyquist sampling rate then,

we have $f_s \geq 2(4\ \text{KHz}) = 8\ \text{KHz}$. Since the articulators change at about a 20-25 Hz rate the spectrum should be sampled at about a 50 Hz rate, thus T can be set equal to 20 ms. If we use 128 data points per record then we have $f_s = 128(50) = 6{,}400$ Hz which is less than the minimum 8 KHz specified above. If 256 data points per record are used then $f_s = 256(50) = 12{,}800$ Hz which is well above the minimum 8 KHz.

From the above considerations we now have:

$f_s = 12{,}800$ Hz

$f_f = 6{,}400$ Hz

$T = 20$ ms

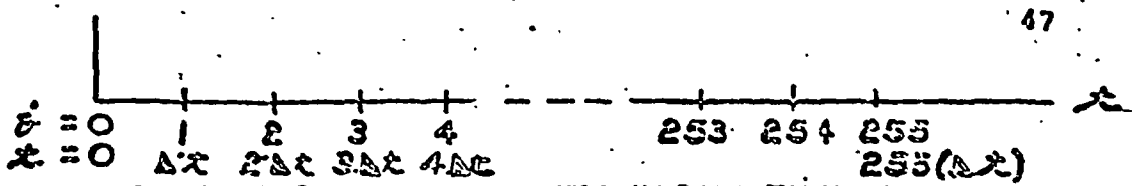
$f_0 = 50$ cycles

$\Delta t = 78.1\ \mu s$

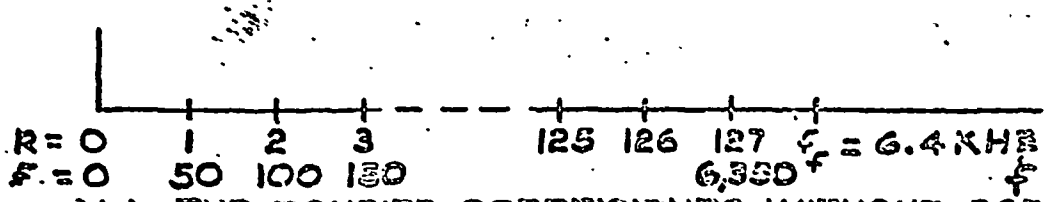
Now the Cepstrum will be considered further based on the specifications above. When the power spectrum is logged and Fourier transformed the resulting coefficients will be on the quefrequency scale which is a time scale. The coefficients will be separated by $1/f_s = 78.1\ \mu s$ and there will be 128 useful coefficients. Thus, a Cepstral peak at the 128th point would indicate a pitch fundamental period of 128(.0781 ms)=10 ms. A period of 10 ms yields a pitch frequency of 100 Hz. Thus the minimum pitch frequency P_{min} which can be obtained with f_s=12.8 KHz and N=256 is not

compatible with the 70 Hz desired as discussed in Section II. A P_{min} of 70 Hz can be obtained by reducing the sampling rate f_s to 8.95 KHz but this would cause T to increase to 29 ms. The speech waveform is not sufficiently stationary over a 29 ms window to give satisfactory results.

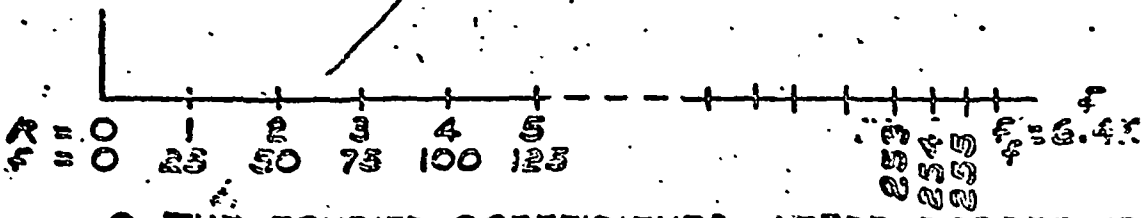
There is a technique available to extend P_{min} and which also produces a better Cepstrum peak over the entire pitch range, particularly in the low frequency region. This approach is to border the 256 data points with 128 zeroes on each side (actually the 256 zeroes could be put either all in front or in back of the 256 data points) (4) and perform a 512 point transform. The results of this operation are shown in Figure 13. Figure 13a illustrates 256 equally spaced sample points in the time domain. If the sampling rate is 12.8 KHz and a 256 point transform is performed then 128 equally spaced Fourier coefficients will be obtained between 0 and 6.4 KHz with a spacing f_0 of 50 Hz as shown in Figure 13b. If the 256 sample points are bordered with 256 zeros and a 512 point transform is performed then 256 equally spaced Fourier coefficients will be obtained between 0 and 6.4 KHz with a spacing f_0 of 25 Hz as shown in Figure 13c. If the cepstrum is obtained from the data of Figure 13b then 128 equally spaced points will be obtained on the quefrency scale with Δt=0.0781 ms



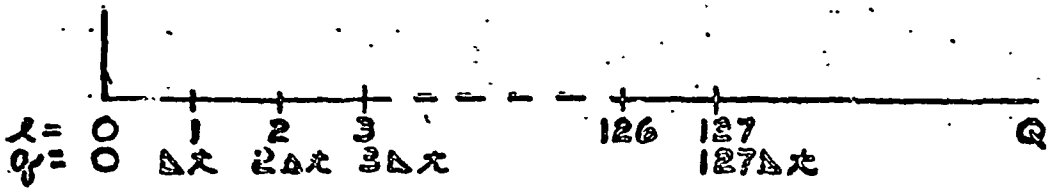
a. 256 SAMPLE POINTS IN THE TIME DOMAINE



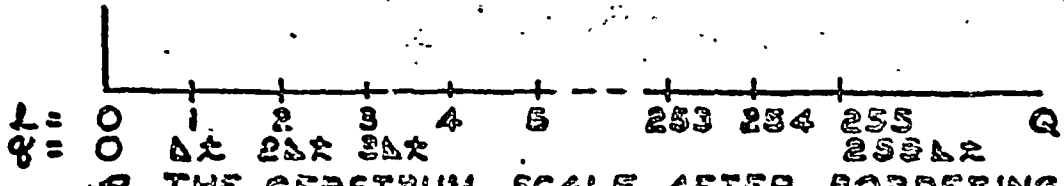
b. THE FOURIER COEFFICIENTS WITHOUT BORDERIN WITH ZEROS



C. THE FOURIER COEFFICIENTS AFTER BORDERING WITH 256 ZEROS



d. THE CEPSTRUM SCALE WITHOUT BORDERING WITH ZEROS, Δt = 0.078IMS



e. THE CEPSTRUM SCALE AFTER BORDERING WITH ZEROS, Δt = 0.078IMS

Figure 13

Effects in the Frequency and Cepstrum Domain from Bordering with Zeros

as shown in Figure 13d. If the cepstrum is obtained from the data in Figure 13c, then 256 equally spaced points will be obtained on the quefrency scale, still with $\Delta t=0.0781$ as shown in Figure 13e. For N=512 and f_s=12.8 KHz there will be 256 coefficients between 0 and 6.4 KHz with f_0=25 cycles. The bordering with zeros does not distort the spectral envelope and in fact provides a finer frequency quantization of the spectrum.

The finer frequency quantization of the spectrum is in effect a higher sampling rate of the log power spectrum and thus a higher rate of ripple (closer spacing of the harmonics) on the frequency scale can be represented on the quefrency scale. The coefficients on the quefrency scale will still be $1/f_s$=78.1 μs apart but for the N=512 case there will be 256 useful coefficients. Thus the longest pitch period detectable is 256 (.0781 ms)=20 ms which corresponds to a P_{min} of 50 Hz. Thus by bordering with zeros the pitch range can be extended below 70 Hz and the cepstral peak actually enhanced over the desired range by the higher effective sampling rate of the spectrum.

So now the speech analyzer can be specified as using 256 data points per record but bordering with 256 zeros to perform a 512 point transform, thus extending P_{min} to 50 Hz.

E. THE TIMING REQUIREMENTS

1. General - This subsection will develop the number of memory accesses and real multiplications and additions required for the speech analyzer.

A core memory is considered as most cost effective for this application when considering the speed, random access requirements, and cost per bit. The number of memory accesses required will be partitioned into full-cycle accesses and half-cycle accesses where half-cycle is considered here to refer to a read operation which does not restore (that is leaves zeros in the address read) and a write operation which does not have to clear (that is assumes zeros are initially stored in the address into which a word is to be written). Values of 1.0 μs and 0.6 μs will be used for the full-cycle and half-cycle times respectively since there are standard off-the-shelf memories with these specifications from several vendors.

2. Data Buffering and Weighting of Input Data - A 256 word section of memory will be set aside as a data input buffer. The output of the analog-to-digital converter will be read into this buffer every 78.1 μs and at the beginning of the processing of each record a block of 256 data points will be transferred from the buffer into the processing portion of the memory. The data will be multiplied

by the windowing weights at this time. Storage locations for 256 weighting constants are set aside so that any windowing function may be used. From the above discussion we see that 256 half-cycle writes are needed to get a new record of data into memory. Then to transfer the new record of data into the processing portion of memory 256 half-cycle reads (hc-reads) are needed to get the data from the buffer, 256 full-cycle reads (fc-reads) are needed to fetch the window constrants, 256 multiplies are performed, and 256 hc-writes are needed to place the new weighted record into the processing portion of memory. These operations are listed in Table 2.

It should be pointed out here that since the buffer is not a separate memory, all controllers in the system must have the capability of being interrupted by the analog-to-digital converter to read data words into the buffer.

3. The FFT Algorithm - The number of real multiplies and real additions will be calculated for a 512 point radix two transform. For a 512 point transform there are nine arrays to be calculated with 512 points in each array. To calculate each new point in the next array there is a complex multiply and complex addition. A complex multiply requires four multiplies and two additions as shown in Equation (12).

Table 2

Summary of Timing Requirements

Operation	Arithmetic Operations		Memory Accesses		Memory Time in ms		
	Mults	Adds	Full Cycle	Half Cycle	Full Cycle	Half Cycle	Total
Data Buffering and Windowing	256	--	256	768	.256	.461	.717
FFT Algorithm	9,216	13,824	1,147	10,240	1.147	6.144	7.291
Separation Algorithm	--	1,024	--	2,048	--	1.228	1.228
Square Magnitude	1,024	512	--	1,536	--	0.920	0.920
Weighting of the Log Power Spectrum	256	--	256	768	.256	.461	0.717
Cepstrum Peak Detector	257	261	32	256	.032	.154	0.186
Output Data	--	--	513	--	.513	--	0.513
Total	11,009	15,621	2,204	15,616	2.204	9.368	11.572

$$(a_1+ib_1)(c_1+id_1) = (a_1c_1-b_1d_1) + i(a_1d_1+b_1c_1) \qquad (12)$$

The subtraction is counted as an addition since it can be done in 2's complement. A complex add requires 2 adds. Therefore each new point requires 4 multiplies and 4 adds. However, 1/2 of the complex multiplies can be saved per pair of points due to the nature of the FFT (use of powers of W which differ by N/2). So a pair of points requires 4 multiplies and six adds. As a result it can be stated that each new point requires 2 multiplies and 3 adds.

So a 512 point FFT will have nine arrays to calculate with 512 points in each array and two multiplies and three adds for each point. Thus the 512 point FFT will need 9,216 multiplies and 13,824 additions as shown in Equations (12) and (13).

$$512 \times 9 \times 2 = 9{,}216 \text{ multiplies} \qquad (12)$$

$$512 \times 9 \times 3 = 13{,}824 \text{ addition} \qquad (13)$$

In calculating the number of memory accesses it will first be assumed that a mod 2 algorithm will be used where mod 2 indicates that two data points are accessed, processed, and stored back into memory before accessing two more data points. Thus there will be 18,432 memory accesses as shown in Equation (14).

$$\frac{512 \text{ complex points}}{\text{array}} \times \frac{2 \text{ reads}}{\text{complex point}} \times 2 \text{ writes} \times 9 \text{ arrays} = 18{,}432 \text{ reads and writes} \qquad (14)$$

Each of these memory accesses can be done in the half-cycle mode of 0.6 μs. Thus memory time required is 18,432 (0.6μs) = 11.06 ms.

In addition to the complex data points to be accessed the sine/cosine coefficients must be fetched. If it is considered that one sine/cosine point is fetched per data pair then there will be (512)(9)(1/2) = 2,294 fetches (assuming that the sine and cosine values are both stored in the same memory location, which is reasonable). Since these fetches are full cycle reads at 1.0 μs per read the time required is 2.294 ≈ 2.3 ms. Thus the total time for memory accesses is 11.06 + 2.30 = 13.36 ms.

If memory accesses and arithmetic operations cannot be performed simultaneously, then only 6.64 ms are available for the arithmetic operations (and this is considering only the FFT operation). Clearly then, it is necessary to have the capability of simultaneous memory access and arithmetic operation. The use of a mod 4 algorithm provides this capability and in addition reduces the number of data point accesses by almost a factor of two.

In the mod 4 algorithm four data points are accessed which will allow the calculation of 4 data points in each of the next two arrays. This can be seen by referring to Figure 1. If the points A_0(0, 4, 8, 12) are accessed then

the points $A_2(0, 4, 8, 12)$ can be calculated before data must be returned to memory. Thus the number of memory accesses using the mod 4 algorithm is:

$$(512)(4)(8)(1/2) + (512)(4)(1) = 10,240$$

Since each of these is a half-cycle access of 0.6 μs the time required is 6.144 ms.

It is a property of the FFT that the sine/cosine coefficients of each set of four data points represent quadrature vectors. Thus with slight additional control circuitry only one sine/cosine coefficient need be accessed from memory for each four data points. Thus the memory time required for accessing sine/cosine coefficients is 1.15 ms, half the time derived for the mod 2 algorithm. So the total time for the memory to support the FFT is 6.144 ms + 1.15 ms = 7.294 ms and the system has the capability of simultaneous operation of the memory and arithmetic unit as will be shown later.

4. The Separation Algorithm - As has been discussed earlier the FFT in the speech analyzer being discussed here performs simultaneous Fourier analysis of two sets of real data. In the transform of the two sequences of real values at the same time one set is stored as the real component and the other set is the imaginary component of the complex vector to be transformed. The symmetry property allows separation of the two transforms.

Let $s_1(kt)$ and $s_2(kt)$ be the two sequences of real values to be transformed. Then:

$$F[s_1(kt)] = S_1(n)$$

$$F[s_2(kt)] = S_2(n) \quad \text{where } n = 0,1,2,\ldots,N-1$$

Also let:

$$s(kt) = s_1(kt) + js_2(kt)$$

then from linearity we have

$$S(n) = S_1(n) + jS_2(n)$$

It can then be shown that

$$S_1(n) = \frac{S(n) + S^*(N-n)}{2}$$

$$S_2(n) = \frac{S(n) - S^*(N-n)}{2j}$$

An example of the separation algorithm for N=16 is shown in Figure 14. From the example it can be seen that it is not necessary to perform calculations for all $S_1(n)$ and $S_2(n)$ where n ranges 0 to N-1. The values of $S_1(n)$ and $S_2(n)$ for $9 \leq n \leq 15$ are complex conjugates of the $S_1(n)$ and $S_2(n)$ for $1 \leq n \leq 7$. Thus it is necessary to calculate only N/2 values for each of $S_1(n)$ and $S_2(n)$.

There are four additions required for each $S_1(n)$, $S_2(n)$ pair. The number of additions required for separation is given by:

$$(N/2)4 = (512/2)4 = 1,024$$

An example for N = 16;

n	s(n)	S(N-n)	$2S_1(n)$	$2S_2(n)$
0	$a_0 + jb_0$	$a_0 + jb_0$	$(a_0 + a_0) + j(b_0 - b_0)$	$(b_0 + b_0) + j(a_0 - a_0)$
1	$a_1 + jb_1$	$a_{15} + jb_{15}$	$(a_1 + a_{15}) + j(b_1 - b_{15})$	$(b_1 + b_{15}) + j(a_{15} - a_1)$
2	$a_2 + jb_2$	$a_{14} + jb_{14}$	$(a_2 + a_{14}) + j(b_2 - b_{14})$	$(b_2 + b_{14}) + j(a_{14} - a_2)$
3	$a_3 + jb_3$	$a_{13} + jb_{13}$	$(a_3 + a_{13}) + j(b_3 - b_{13})$	$(b_3 + b_{13}) + j(a_{13} - a_3)$
4	$a_4 + jb_4$	$a_{12} + jb_{12}$	$(a_4 + a_{12}) + j(b_4 - b_{12})$	$(b_4 + b_{12}) + j(a_{12} - a_4)$
5	$a_5 + jb_5$	$a_{11} + jb_{11}$	$(a_5 + a_{11}) + j(b_5 - b_{11})$	$(b_5 + b_{11}) + j(a_{11} - a_5)$
6	$a_6 + jb_6$	$a_{10} + jb_{10}$	$(a_6 + a_{10}) + j(b_6 - b_{10})$	$(b_6 + b_{10}) + j(a_{10} - a_6)$
7	$a_7 + jb_7$	$a_9 + jb_9$	$(a_7 + a_9) + j(b_7 - b_9)$	$(b_7 + b_9) + j(a_9 - a_7)$
8	$a_8 + jb_8$	$a_8 + jb_8$	$(a_8 + a_8) + j(b_8 - b_8)$	$(b_8 + b_8) + j(a_8 - a_8)$
9	$a_9 + jb_9$	$a_7 + jb_7$	$(a_9 + a_7) + j(b_9 - b_7)$	$(b_9 + b_7) + j(a_7 - a_9)$
10	$a_{10} + jb_{10}$	$a_6 + jb_6$	$(a_{10} + a_6) + j(b_{10} - b_6)$	$(b_{10} + b_6) + j(a_6 - a_{10})$
11	$a_{11} + jb_{11}$	$a_5 + jb_5$	$(a_{11} + a_5) + j(b_{11} - b_5)$	$(b_{11} + b_5) + j(a_5 - a_{11})$
12	$a_{12} + jb_{12}$	$a_4 + jb_4$	$(a_{12} + a_4) + j(b_{12} - b_4)$	$(b_{12} + b_4) + j(a_4 - a_{12})$
13	$a_{13} + jb_{13}$	$a_3 + jb_3$	$(a_{13} + a_3) + j(b_{13} - b_3)$	$(b_{13} + b_3) + j(a_3 - a_{13})$
14	$a_{14} + jb_{14}$	$a_2 + jb_2$	$(a_{14} + a_2) + j(b_{14} - b_2)$	$(b_{14} + b_2) + j(a_2 - a_{14})$
15	$a_{15} + jb_{15}$	$a_1 + jb_1$	$(a_{15} + a_1) + j(b_{15} - b_1)$	$(b_{15} + b_1) + j(a_1 - a_{15})$

Figure 14

Equations for the Separation Algorithm for N = 16

For each new data pair calculated there will be four half-cycle reads to fetch S(n) and S(N-n) and four half cycle writes to store $S_1(n)$ and $S_2(n)$. Thus, there will be 8 half-cycle memory accesses for N/2 data points. The number of half-cycle memory access is then:

$$(N/2)8 = (5.2/2)8 = 2,048$$

At 0.6 μs per half cycle access the memory time will be approximately 1.23 ms.

5. Square Magnitude - A shown in Figure 12, the operation following the separation is to calculate the square magnitude of the complex spectrum and cepstrum. For a complex number a + jb, the square magnitude is $a^2 + b^2$. As noted in subsection 4 above the processor will be working with N/2 points at this stage. Treating a squaring operation as a multiply there will be two multiplies and one add for each point in two sets of N/2 points. Thus the number of adds and multiplies are:

$$(N/2)(2)(2) = 2N = 1,024 \text{ multiplications}$$

$$(N/2)(1)(2) = 512 \text{ additions}$$

Because of fetching complex points and storing real points (square magnitudes) there will be two half-cycle reads and one-half cycle write per data point for two sets of 256 data points. Thus there are 1536 half-cycle memory accesses at 0.6 μs per access for a total memory time of 0.92 ms.

6. Weighting of the Log of the Power Spectrum - As shown in Figure 12 the log of the power spectrum is taken and then weighted prior to introduction to the imaginary input of the FFT box. Assuming a combinational logic circuit to obtain the log to the base 2 of the power spectrum, it is possible to fetch a power spectrum point from memory, pass it through the combinational logic, multiply it by the weighting factor, and then store it back into memory. There are 256 spectral coefficients to be operated on, however, the image of these 256 data points (negative frequency) must also be stored to generate the 512 point input signal for the FFT. That is, $s_2(n)$ will also be stored in $s_2(N-n)$. Thus, there will be 256 multiplies, 256 hc-reads for spectral coefficients, 256 fc-reads for weighting values, and 512 hc-writes to place the results in the A_c array of the FFT. Memory time requirements will be:

$$(256+512)(0.6) + (256)(1.0) = 0.72 \text{ ms}$$

7. Cepstrum Peak Detector - The cepstrum, defined as the power spectrum of the logarithm of the power spectrum, has a strong peak corresponding to the pitch period of the voiced-speech segment being analyzed. A simplified flow diagram of the cepstrum peak detector taken from Noll (14) is shown in Figure 15. Details of the flow chart will not be discussed here. The algorithm will require 256 half-

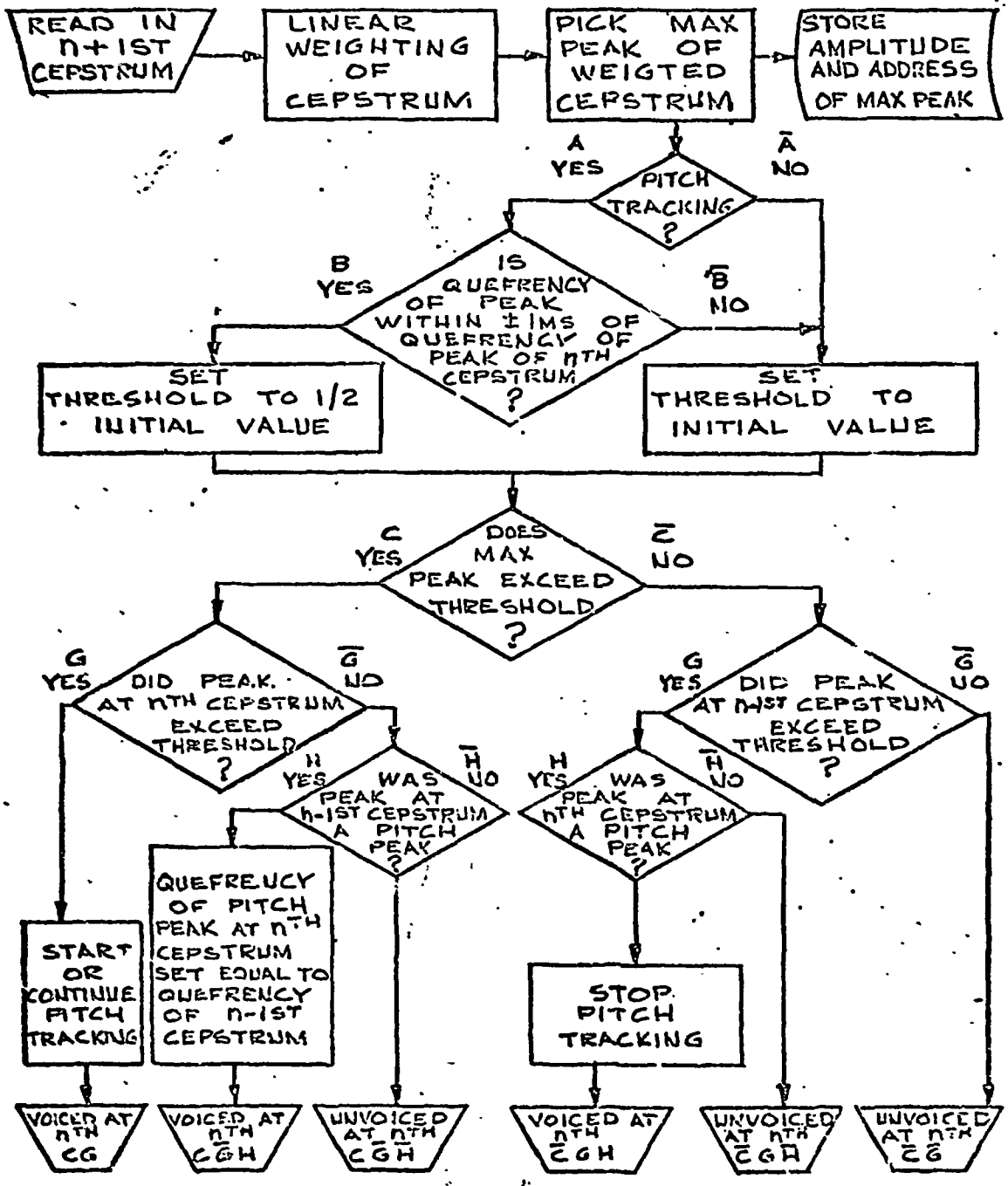


Figure 15

Flow Chart for Determining Pitch Peak and Voicing Condition

cycle memory accesses to fetch the cepstral coefficients and 32 full-cycle reads to fetch weighting values. Eight cepstral values will be multiplied by each weighting value to produce a step wise increased wighting with respect to increase in time (decrease in fundamental frequency) on the quefrency scale. Thus there will be 256 multiplies. There will then be 256 adds (2's complement subtracts) and comparisons to determine the value and location of the peak. There will be four adds to determine if the present peak is within ± 1 ms of the previous peak. There will be one multiply to set a threshold and one add and check to determine if the peak exceeds the threshold. The number of operations given here is summarized in Table 2.

8. Data Output - If the power spectrum is desired as the output then there will be 256 full-cycle reads to output the data. If the complex spectrum is desired then there will be 512 full-cycle reads to output the data. Full-cycle reads are required because the spectrum data must be used in the cepstrum process and thus cannot be destroyed.

9. Summary of Timing and Speed Requirements - A summary of the timing requirements is given in Table 2. The operations discussed above is given in the column to the left of the table. The number of arithmetic operations, memory accesses and memory time requirements are given for each operation. The totals are given on the bottom row.

The memory access time requirements are further presented in bar chart form in Figure 16. The memory time requirements for each operation are plotted on the total available time base of 20 ms. Based on these data there is 8.428 ms slack time with the memory accessing taking fifty eight percent of the available time.

The plot shown in Figure 17 gives an estimate of the speed requirements for the multiplier and the adder where a single multiplier and adder are used. The plot shows that if additions can be performed in zero time then there will be 20,000 μs/11,009 = 1.8 μs available for each multiplication. Conversely if multiplications can be performed in zero time then there will be 20,000 μs/15,621 = 1.29 μs available for each addition. Obviously neither additions nor multiplications can be performed in zero time, so we can pick a realistic point on the line, for example if we allow 0.5 μs per addition then each multiplication must be performed in 1.1 μs.

The time required for additions and multiplications is directly proportional to the number of bits (word length) involved and inversely proportional to the amount of parallel circuitry used. Although the optimum word lengths have not been investigated, the following assumptions should be close to optimum requirements:

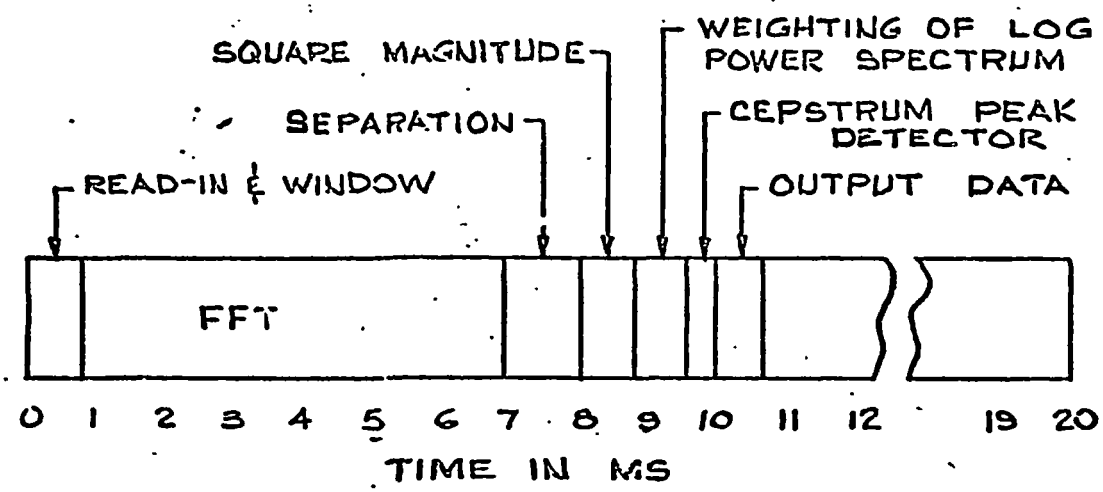


Figure 16

Memory Access Time Requirements

Representation of memory access requirements in ms plotted on the total available time base of 20 ms. Based on these data there is 8.428 ms slack time with the memory accessing taking fifty-eight percent of the available time.

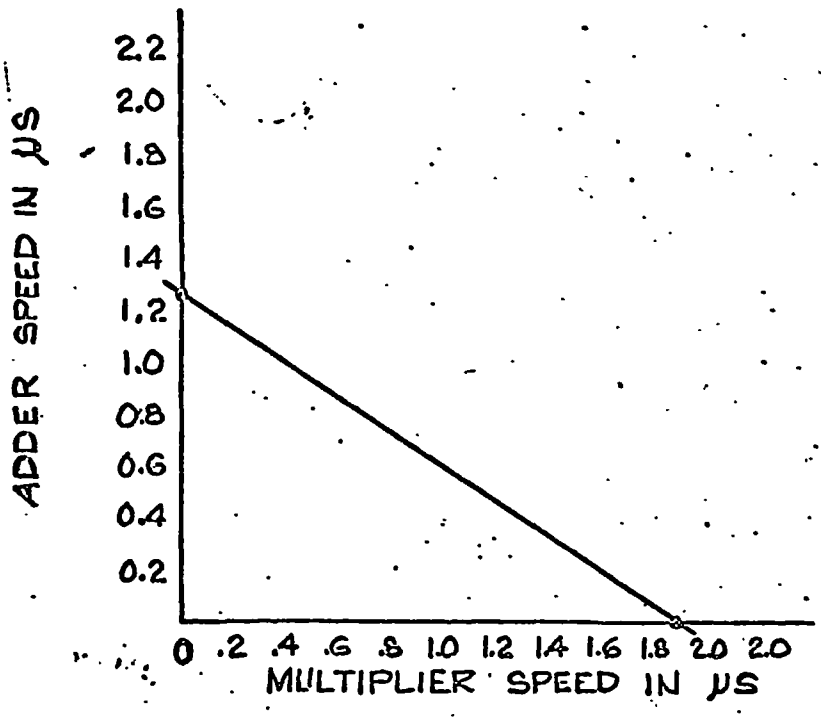


Figure 17

Speed Requirements for Multiplier vs Adder

This curve was generated using the requirement of 11,009 real multiplies and 15,621 real additions and an allowable time of 20 ms. It assumes one multiplier and one adder operating at one hundred percent efficiency.

1. Input data word length 8 bits
2. Processor word length 16 bits
3. Coefficient word length (sine/cosine values) 8 bits
4. Output word length 8 bits

Transistor-transistor logic (TTL) 4-bit full adders are available which produce the sums and carry from two 4-b t words in 60 nanoseconds. Thus 16-bit words can be summed in 240 nanoseconds without using any look-ahead logic. Also, using TTL carry save adders in a Booth and Booth multiplier 16-bit words can be multiplied in 650 nanoseconds (250 nanoseconds if pipelining is used).

So, with circuits commercially available it is possible to perform additions in less than 0.35 μs and multiplications in less than 0.7 μs. This implies a total time for additions and multiplications of

$$(0.35)(15{,}621) + (0.7)(11{,}009) = 13{,}174\ \mu s = 13.174\ ms.$$

This leaves approximately 6.8 ms of slack time for the arithmetic operations.

From the above we have 8.4 ms of slack time for the memory operations and 6.8 ms of slack time for the arithmetic operations. Thus in the design of the architecture of the machine it will not be necessary to insure that the memory and arithmetic unit are operating one-hundred percent of the time; however, they must have the capability of operating simultaneously. The architecture will be covered in the next section.

F. THE ANALYZER ARCHITECTURE

A simplified block diagram of the system architecture for the speech analyzer is shown in Figure 18. The architecture employs a core memory, scratch pad registers, intermediate result registers, data buffer registers, and a fast arithmetic unit designed to operate in parallel with memory accessing. Individual controllers are used for the different algorithms and functions within the algorithms to simplify the controller design and allow easy modification of all or part of an algorithm. The controllers can easily be implemented either as counters and state decoders or as read-only memories. All of the programming is performed from the hardware controllers. The scratch pad registers, intermediate result registers, and the data buffer registers are required to provide parallel operation of the memory and arithmetic unit.

A more detailed block diagram of the registers and the arithmetic unit is shown in Figure 19. The registers are paired as a_i and b_i where the a_i is the real value and the b_i is the imaginary value of a complex word. Thus the a_i and the b_i registers together hold the complex word (a_i+jb_i). The registers a_0 and b_0 through a_3 and b_3 hold the four data points which are needed to calculate the four data values in the next two arrays as explained in Section

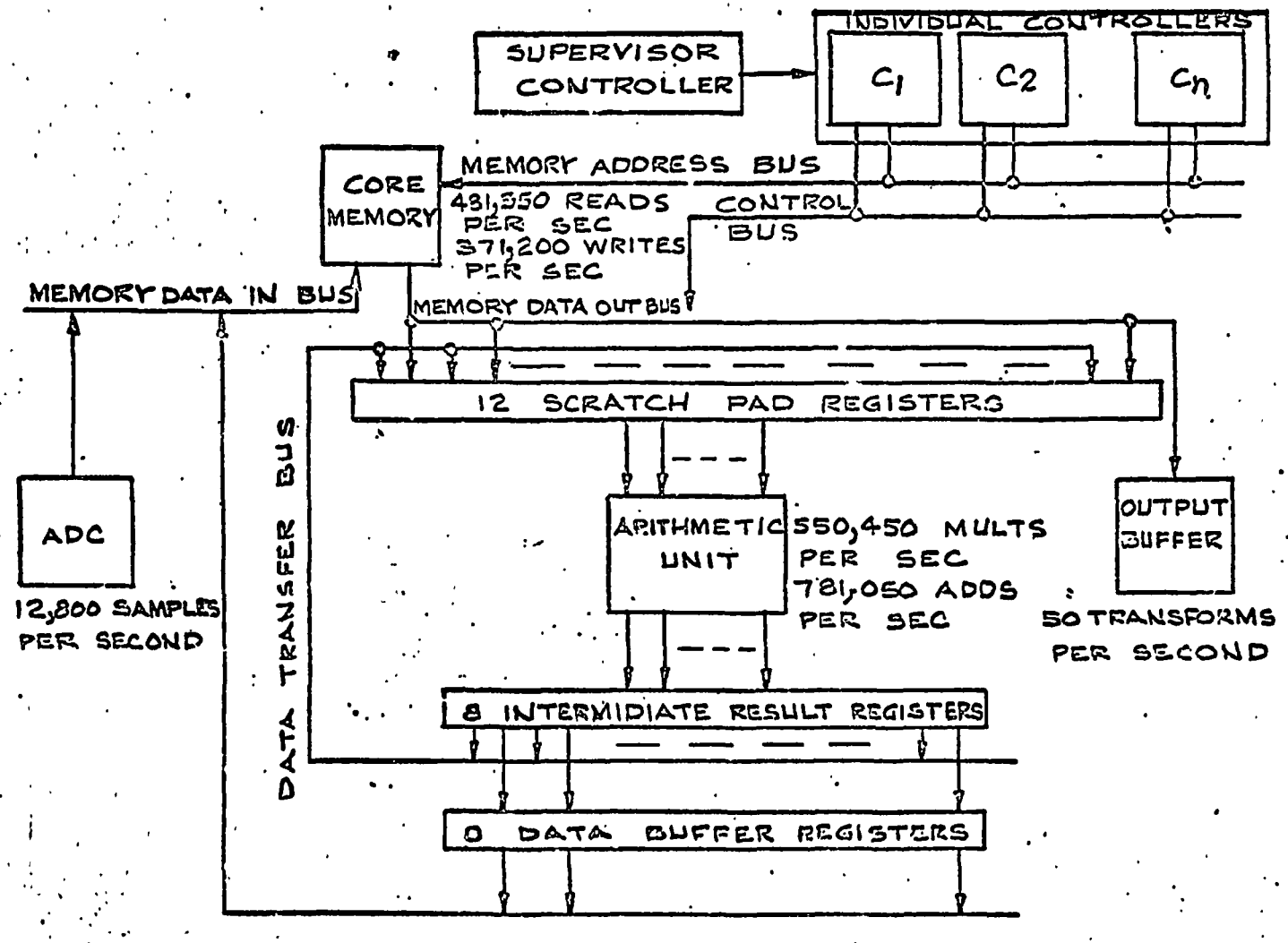


Figure 18

FFT Speech Analyzer Organization

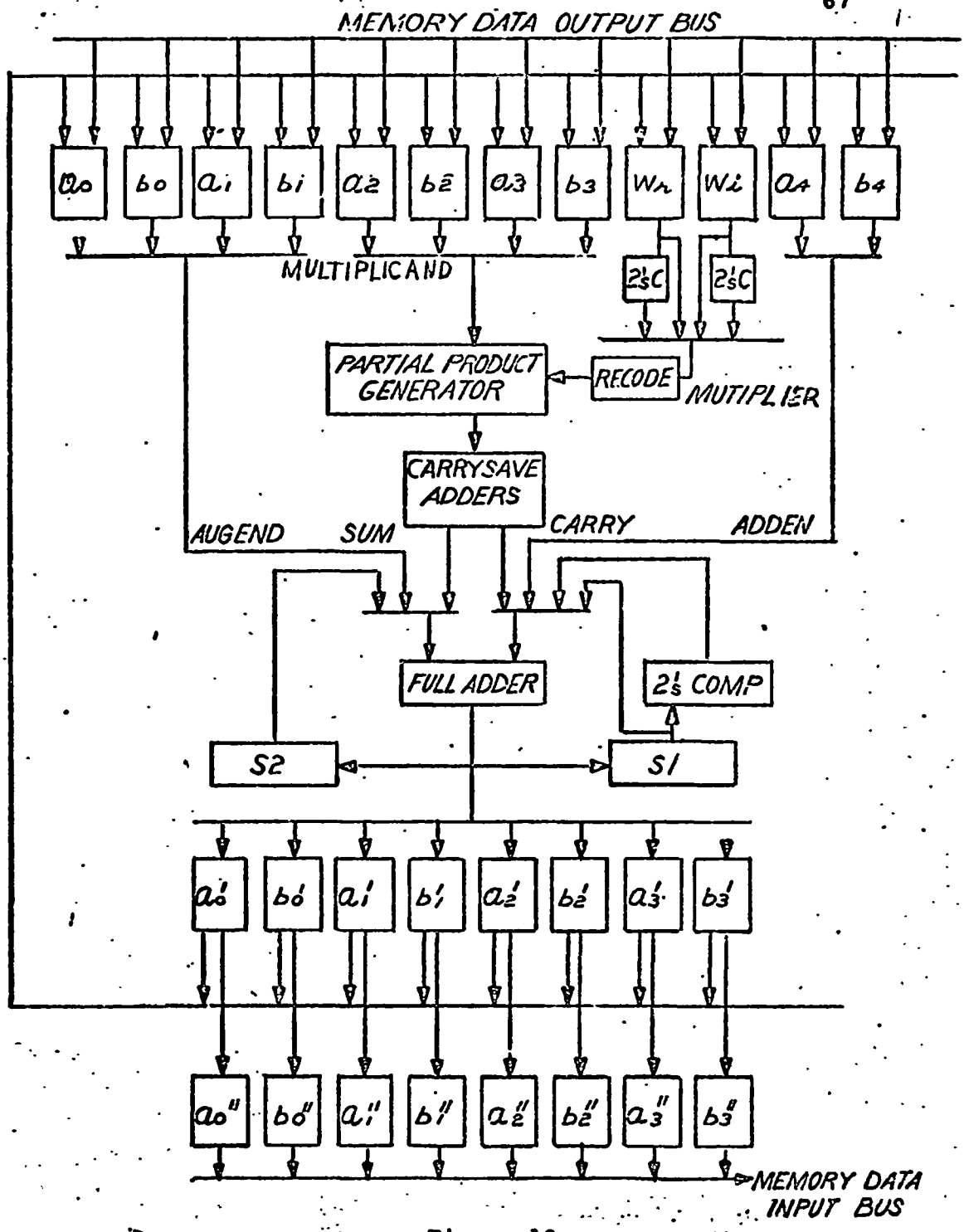


Figure 19

Speech Analyzer Registers and Arithmetic Unit

III D. The a_0, b_0, a_1, and b_1 registers are bussed together as the multiplicand input to the multiplier. The W_R and W_I registers contain the coefficients described in Section III F. The bussing of the real and imaginary parts of the coefficients and their 2's complements allows the use of coefficients from only one quarter of the coefficient circle as described in Section III F. More generally the W_R and the W_I registers hold the multiplier for any multiplication operation. The a_4 and b_4 registers contain the addend for addition operations.

The results of an operation on four data points are placed in the "primed" registers called the intermediated results registers above. Feedback from the intermediate results registers to the scratch pad registers is provided because two passes through the arithmetic unit are required for the Mod 4 algorithm. The "double primed" registers, called the data buffer registers above, allow the intermediate data registers to dump their data so they can be used by the arithmetic unit while data is written back into memory from the data buffer registers.

The recode block provides the recoding of the multiplier required by the Booth and Booth algorithm. The partial product generator generates partial products as a function of the recoded multiplier. The carry-save adders

provide fast generation of the sum and carry for the full adder. When the Sum and carry are gated into the full adder, the output is the product of the selected multiplicand and multiplier. The S1 and S2 registers are used in the Mod 4 algorithm as will be explained below.

A flow chart and equations for a Mod 4 operation is shown in Figure 20. The equations are for only one pass through the arithmetic unit and it should be remembered that two passes are required. It should also be noticed that a slightly different arrangement of data is required in the second pass. In order to keep the algorithm the same for both passes the data transferred back to a_1b_1 and a_2b_2 is interchanged. A more detailed flow chart of one pass through the arithmetic unit for a Mod 4 operation is shown in Figure 21. The use of the S_1 and S_2 registers can be seen in this flow chart.

A Timing diagram of memory accesses and arithmetic operations for one Mod 4 operation is shown in Figure 22. The operations are divided into a first and second pass through the arithmetic unit as discussed above. The top line represents arithmetic operations, and although not labeled, follows the flow of the flow chart of Figure 21. The multiplies can be distinguished from the additions since the former require two clock periods and the later require

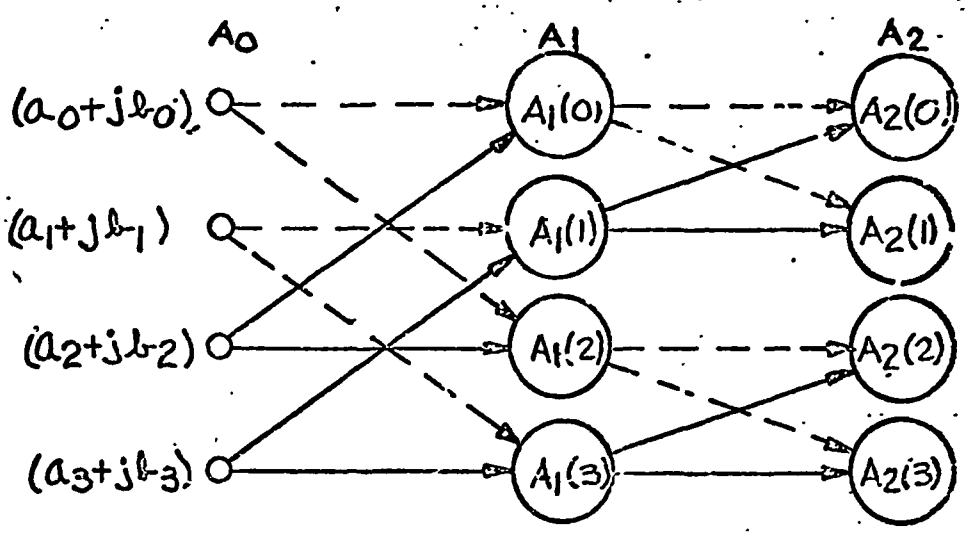


Figure 20

Flow Chart and Equations for a Mod 4 Pass Through the Arithmetic Unit

$$a_0'+jb_0' = A_1(0) = (a_0+jb_0) + (a_2+jb_2)(w_{r0}+jw_{I0}) \quad (1)$$

$$a_2'+jb_2' = A_1(2) = (a_0+jb_0) - (a_2+jb_2)(w_{r0}+jw_{I0}) \quad (2)$$

$$a_1'+jb_2' = A_1(1) = (a_1+jb_1) + (a_3+jb_3)(w_{r1}+jw_{I1}) \quad (3)$$

$$a_3'+jb_3' = A_1(3) = (a_1+jb_1) - (a_3+jb_3)(w_{t1}+jw_{I1}) \quad (4)$$

$$a_0'+jb_0' = (a_0+a_2w_r-b_2w_I) + j(b_0+b_2w_r+a_2w_I) \quad (5)$$

$$a_2'+jb_2' = (a_0-a_2w_r+b_2w_I) + j(b_0-b_2w_r-a_2w_I) \quad (6)$$

$$a_1'+jb_2' = (a_1+a_3w_r-b_3w_I) + j(b_1+b_3w_r+a_3w_I) \quad (7)$$

$$a_3'+jb_3' = (a_1-a_3w_r+b_3w_I) + j(b_1-b_3w_r-a_3w_I) \quad (8)$$

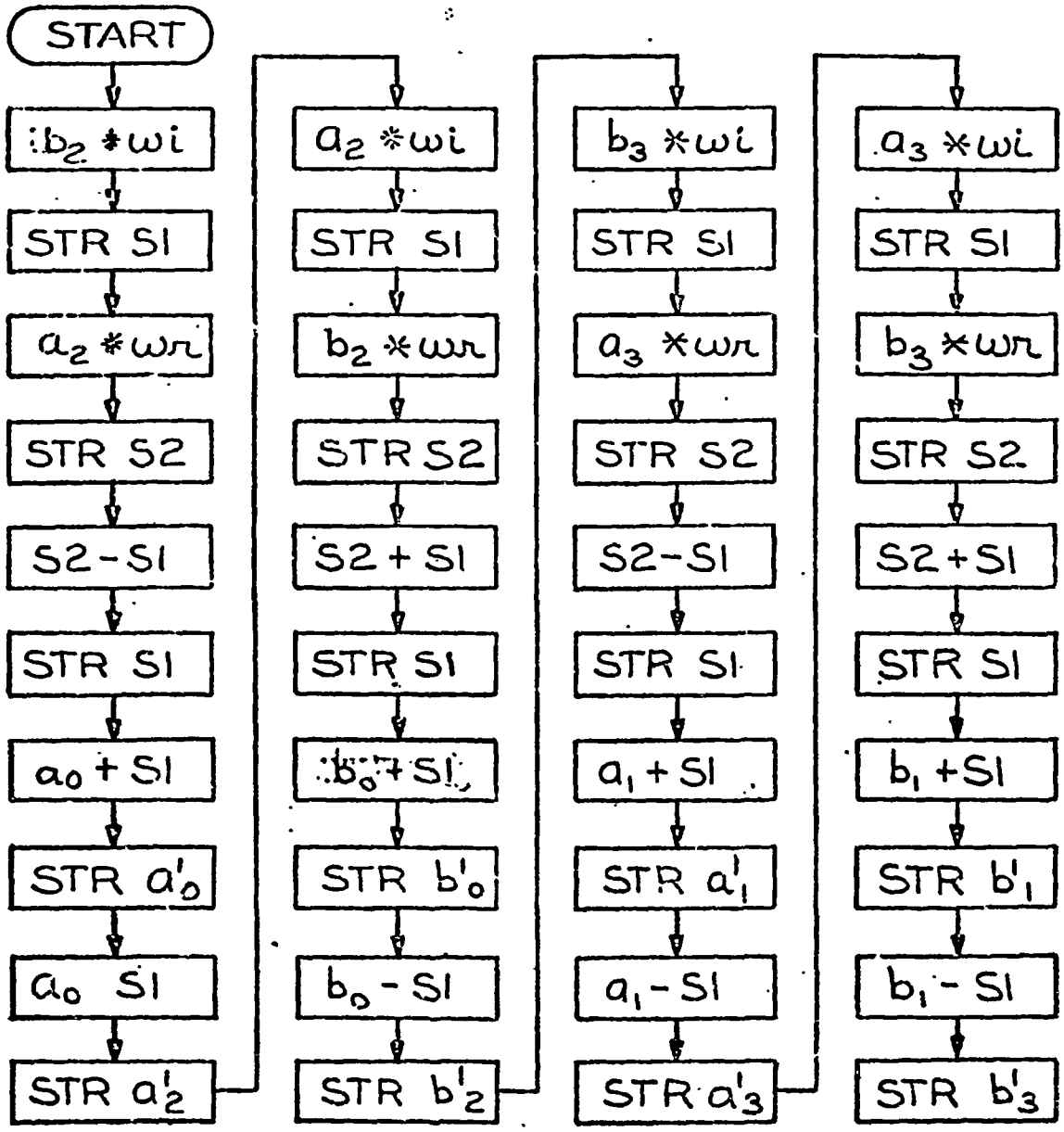
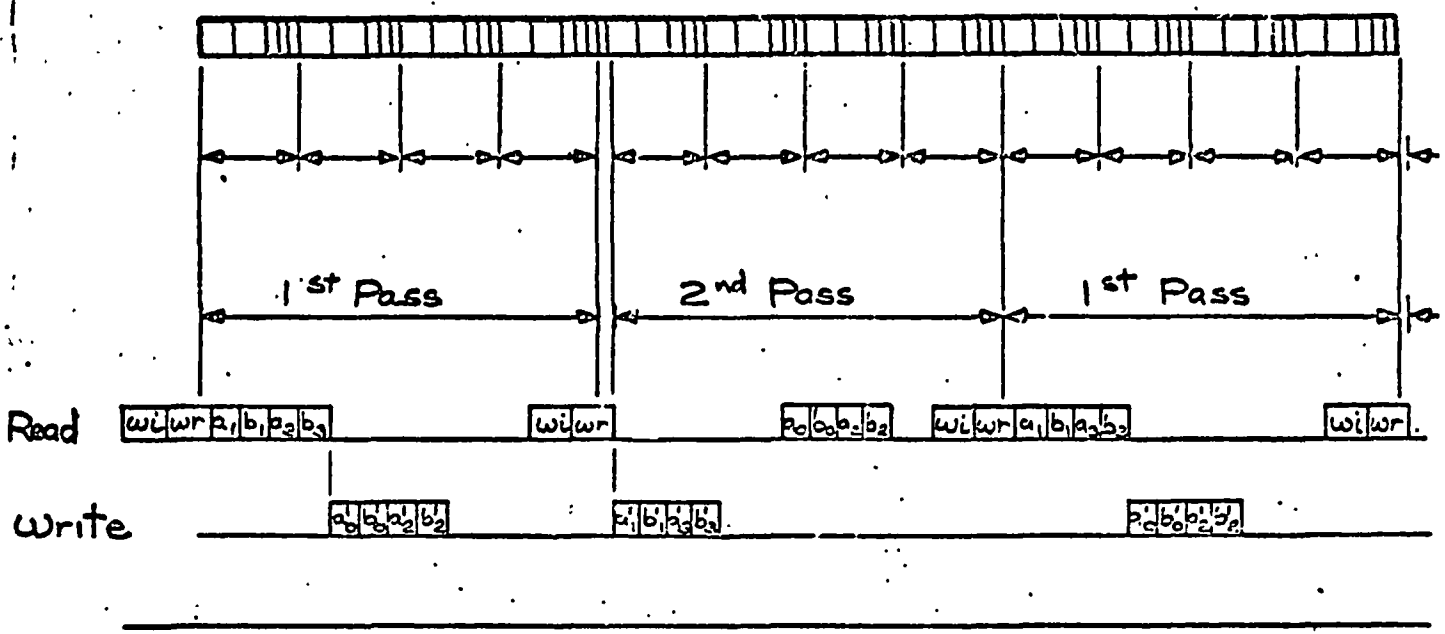


Figure 21

Partial Flow Chart for FFT with One Multiplier and One Adder



FFT MEMORY TIMING

Figure 22

one clock period. The memory accesses require two clock periods for a half-cycle operation and three clock periods for a full cycle operation.

With a clock pulse of 350 ns and two clock pulses per half-cycle operation, each half-cycle operation requires 700 ns in place of the 600 ns capability used in constructing Figure 16. Also, because of the large number of arithmetic operations required the memory is able to operate for only 39 of the 52 clock periods required for a Mod 4 iteration. Thus the memory is operating at only seventy-five percent efficiency. Under these conditions it can be shown that the FFT operation requires just under 12.8 ms to perform in place of the 7.29 ms shown in Table 1. The other algorithms in the speech analyzer do not have as many arithmetic operations per memory access as the FFT algorithm and will in fact be time limited by memory accesses. Thus the times shown in Figure 16 will be essentially correct.

G. SUMMARY OF HARDWARE IMPLEMENTATION

This section has dealt with basic organization options, the processor flow chart, sampling requirements, timing requirements, and the analyzer architecture. Items of great importance but not dealt with here are:

1. The number of bits to use for the input words, processor words, output words, and coefficient words.

2. Dynamic Range.
3. Round off error.
4. Details of the controllers, arithmetic unit, and data addressing.

Obviously all of the above items must be taken into consideration in order to successfully implement a hardware speech analyzer.

H. SIMULATIONS

A complete simulation of the functions shown in Figure 12 was programmed and run on an IMB 1130 computer. The simulation does not employ all the hardware features of the architecture described in this thesis but does implement the functional characteristics.

The IBM 1130 computer is a small general purpose scientific computer. The 1130 used is a 16 bit machine with 8K of core memory and 512,000 word disk cartridge memory. The access cycle time for the core memory is 3.6 microseconds. Access time for one or two tracks on the disk is 15 milliseconds.

The simulation program is controlled by a Driving program written in Fortran IV. The Driving program controls the flow of data into or out of various subroutines. A simplified flow graph of the Driving program is shown in Figure 23. The use of the Driving program provides

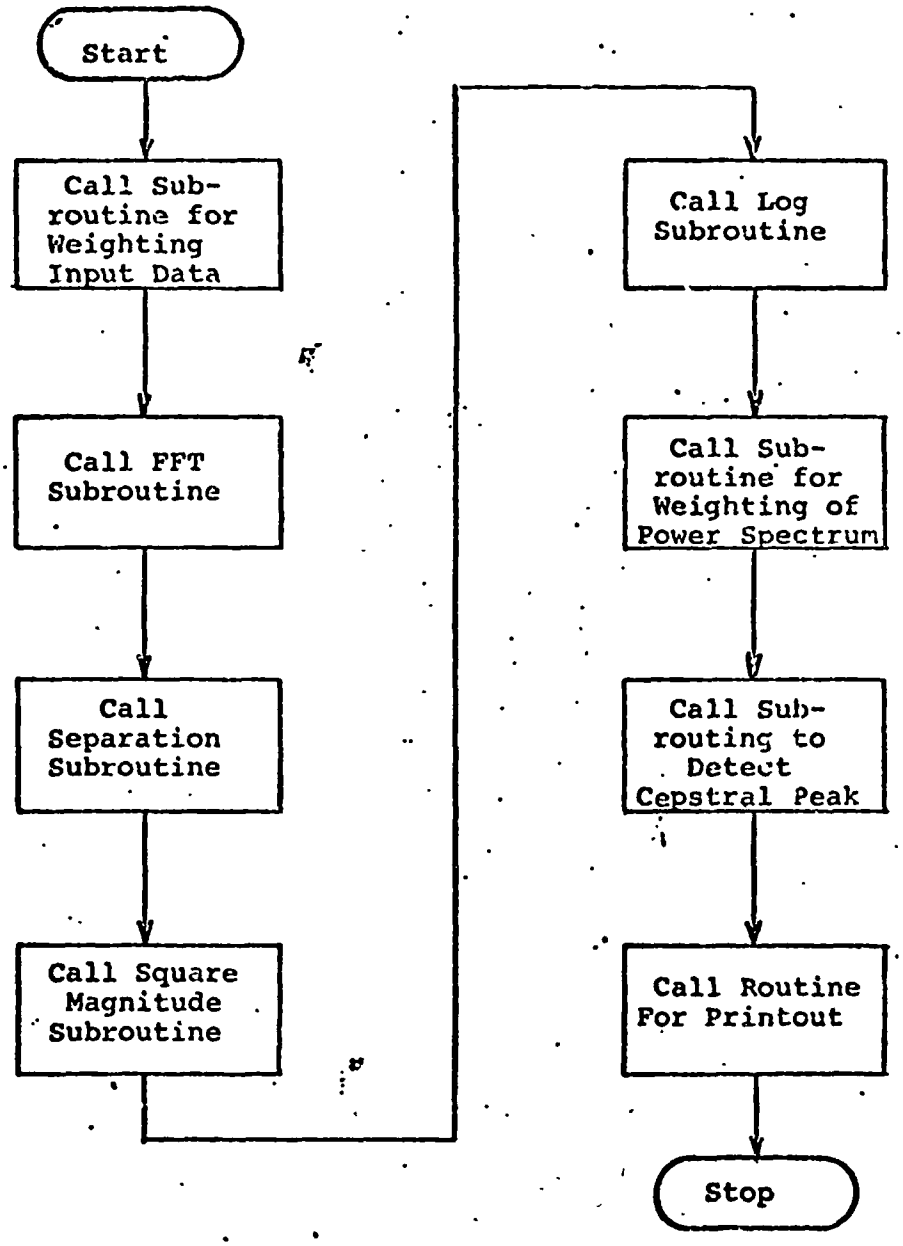


Figure 23

Simplified Flow Chart of Simulation Driving Program

considerable flexibility in the software package to evaluate a wide variety of configurations.

The Driving program can call either a Fortran subroutine or an assembly language subroutine. Most of the subroutines were originally written in Fortran using real constants and variables. In this way it was relatively easy to prove out the concepts involved. For example, the concept of processing two real time series with one FFT and separating the results was verified in this way. Also advantage of Cepstrum over autocorrelation was shown and the concept of borderin with zeros was demonstrated.

All of the Fortran subroutines were eventually replaced with assembly language subroutines. All of the assembly language routines used integer arithmetic except the FFT and Separation subroutines, which continued to use real values. Under this condition the simulation continued to perform satisfactorily. Assembly language FFT and Separation subroutines using integer arithmetic were then written and tried; however, the dynamic range was severely limited under these conditions. This final configuration has not been investigated to determine the problem areas. It is believed that the problem is either an error in the subroutines themselves, a bad choice in scaling within a subroutine and/or mismatch in scaling between subroutines.

Further investigation and modification of the software should be performed until it executes satisfactorily simulating the hardware configuration described in this thesis. It would then be possible to specify the design of the hardware system controllers.

Four basic FFT routines were written for the simulation. Base 2 and base 4 routines were written in both Fortran and assembly language. The base 2 Fortran subroutine uses 88 statements; the base 4 Fortran subroutine uses 134 statements; the base 2 assembly language subroutine uses 296 statements; and the base 4 assembly language subroutine uses 670 statements.

The Fortran subroutine for separation uses 25 statements and the weighting program uses 12 Fortran statements. The square magnitude subroutine uses 60 assembly language statements and the log subroutine uses about 40 assembly language statements. The Cepstral peak detector subroutine uses 181 assembly language statements.

The first version of the simulation (all Fortran) ran at about 4,500 to 1 machine time to real time. At this rate, 90 seconds is required to process one 20 millisecond frame of data. In the final assembly language form, the simulation ran at about 1,000 to 1 machine time to real time. At this rate, 20 seconds is required to process one 20 millisecond frame. Also at this rate, it takes just under one

and one-half hours to process five seconds of speech on the IBM 1130.

SECTION VI

OTHER APPROACHES TO SPEECH ANALYSIS BY DIGITAL TECHNIQUES

There are several possible approaches to implementing a digitalized speech analyzer. This author does not know of any actual hardware implementations although Bell Telephone Laboratories, Lincoln Laboratories, and several universities and companies in industry have achieved computer simulations of digital speech analyzers. However, a large number of special purpose FFT boxes have been built as evidenced by Bergland (13).

There are many different parameters to be considered in comparing special purpose FFT processors if a fair comparison is to be made. A limited comparison will be made here of the FFT processor developed in this thesis and the processors covered by Bergland (13). Of twenty-six processors covered in Bergland's survey, eleven are of the sequential structure as shown in Figure 8 (which is the structure of the processor developed in this thesis). One of the parameters used for comparison is execution time of a transform where N=1024. Most of the sequential processors have an execution time on the order of 30 to 50 ms. Three processors with execution times of less than 30 ms are given.

These execution times are listed as 27 ms, 22 ms, and 3.75 ms. The processor developed in this thesis performs a 512 point transform in 12.8 ms. If the controllers were changed to perform a 1024 point transform then the transform would take 28 ms. This time could be cut in half with a small increase in hardware to provide pipelining in the multiplication hardware and lookahead capibility in the full adder. By using four multipliers to implement a complex multiplier the execution time could be reduced by roughly another factor of 4, thus providing an execution time of roughly 3.5 ms.

Comparison should be made between the special purpose speech analyzer developed here and the use of a general purpose computer for implementing the algorithm. Factors to be considered are cost of the special purpose processor, machine time - to - real time on a general purpose machine, and cost per hour of running on the general purpose machine. A rought estimate of hardware cost for the special purpose machine is $15,000. The algorithm for the speech analyzer of this thesis was programmed to run on an IBM 1130 (small scientific computer) and ran at about a 1,000 to 1 machine time to real time ratio. A reasonable cost per hour to use the IBM 1130 is $25. For $15,000 one could purchase 600 hours of computer time. At a ratio of 1,000 to 1 this would allow only 0.6 hours of continuous speech to be processed.

For many study and research applications only a few seconds of continuous speech are needed and thus it would not be practical to build or purchase a special purpose processor. For other applications, several hours of continuous processing may be required and thus large savings in money and time would be achieved by using a special purpose processor. It is estimated that the algorithm could run in real time on the CDC 6600 computer. If the charges for the 6600 were $1,200 per hour then 12.5 hours could be run for $15,000. Thus if speech processing in excess of 12.5 hours were needed it would be advantageous to use the special purpose machine described in this thesis.

Among the several approaches to implementing a digital speech analyzer are (1) Direct Fourier Transform (DFT), (2) Fast Fourier Transform (FFT), (3) digital filtering (Z-transform), (4) autocorrelation, (5) complex exponential analysis, and (6) least mean square fitting of basis functions. Of the six approaches listed above it is believed that the FFT, and digital filter approaches hold the most promise of performance and economy. Although the FFT approach has been taken in this paper, the tradeoffs of complexity, cost, and performance between the FFT and digital filter approach is not clear cut enough to make either one of them outstanding with respect to the other.

One of the other techniques of interest is the complex exponential transform even though at this time it appears impractical to achieve a real time system with this approach. The complex exponential transform uses damped oscillators as the basis functions in place of the constant amplitude oscillators used in a Fourier transform. The complex exponential does not assume periodicity outside the data window and results in higher resolution for given data length than does the Fourier transform. In addition, placement of frequencies in the resulting spectrum is entirely dependent upon the frequencies present in the input signal whereas the coefficients obtained from the Fourier transform are pre-positioned by the period of the window and the sampling rate. Perhaps the advance of hardware art coupled with the reduction of the general case of the complex exponential process to a specific application will make real time feasible.

It should be stated that analog speech processors using the bank-of-filter type spectral analyzers have been in use since the late 1930's and have not yet been displaced by digital speech processors. However the recently developed cepstral techniques can be implemented easier and perform more reliable with digital circuits than with analog circuits. Thus the coming years should see a switch to all-digital speech processors.

SECTION VII

RECOMMENDATIONS FOR FURTHER INVESTIGATIONS

Specific areas for further investigation have been pointed out in Section V G. These are determination of word length for the input word, output word, processor word, and coefficient words; dynamic range; round off error; and details of the controllers, arithmetic unit, and data addressing. Theoretical investigations should be performed; however, the actual machine should be simulated on a general purpose computer for evaluation.

The architecture described in this paper uses a single multiplier and adder and the speed requirements can be met by TTL integrated circuits. Because of recent advances in MOS-LSI technology, consideration should be given to the type of architecture required with respect to the slower speed of MOS and the partitioning problems of LSI.

Section V G contains a brief discussion of the processor controllers. Individual controllers are used for the different algorithms and functions within the algorithms to simplify the controller design and allow easy modification of all or part of an algorithm. The controllers can easily be implemented either as counters and state decoders

or as read-only memories. Thus the processor is a micro-programmed system. The mechanical design of the processor could be such that any of the controllers could be easily removed and replaced with a controller containing a different program. With this capability the processor could be relatively easily modified and thus be used for signal processing applications other than speech analysis. By changing the sampling rate and N and the control of the algorithms, spectral analysis can be made of blood flow, electroencephelograms, radar, sonar, acoustical signals, and etc. Algorithms other than spectral analysis could be implemented such as convolution and special vector operations.

It would, of course, be desirable to have a software capability within the processor to further extend its general usefulness. Eight of the twenty-six special prupose FFT processors covered by Bergland have software control. In general a software capability can be added at the expense of much greater cost and loss of processing speed. Investigation should be performed to determine an efficient software approach and the trade offs of cost, speed and increased capability of the processor.

Implementation of digital filtering approach should be compared with the FFT approach. The digital filtering approach requires more arithmetic operations than the FFT;

however, the control circuits are much less complicated. This could result in digital filtering being the best of the two for implementation with MOS-LSI.

More basic studies need to be performed for application of the complex exponential transform to speech processing. If these studies prove encouraging, then consideration should be given to a hardware implementation of the complex exponential.

This thesis has dealt entirely with the implementation of a digital speech analyzer. In most speech processing applications the analyzer is only half of the system with a speech synthesizer being the other half. Thus development of a digital speech synthesizer is of prime importance.

SECTION VIII

CONCLUSION

An organization of a digital speech analyzer using the fast Fourier transform and cepstrum has been presented. Structure of the speech waveform, FFT considerations, and cepstrum pitch extraction was discussed as it related to the implementation of a digital speech analyzer.

The development of the FFT, reduction in price of digital integrated circuits, and the rapid advance of large scale integration (LSI) make it economically feasible to implement a digital speech analyzer which will function in real time. Although several important items such as word length, dynamic range, round off error, and circuit details were not discussed, the basic areas of organization options, the processor flow chart, sampling requirements, timing requirements and architecture were discussed in detail.

REFERENCES

(1) Flanagan, J. L., "Speech Analysis, Synthesis, and Perception," 1965 New York, Academic Press, Inc., Publishers.

(2) Cooley, J. W., and Tukey, J. W., "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Comput., vol. 19, pp. 297-301, Apr. 1965.

(3) Cooley, J. W., Lewis, P. A. W., and Welch, P. D., "The Fast Fourier Transform Algorithm and Its Applications," IBM Res. Paper RC-1743, Feb., 1967.

(4) Bergland, G. C., "A Guided Tour of the Fast Fourier Transform," IEEE Spectrum, pp. 41-52, July, 1969.

(5) Brigham, E. O., Morrow, R. E., "The Fast Fourier Transform," IEEE Spectrum, pp. 63-70, December, 1967.

(6) Gentleman, W. M., Sande, G., "Fast Fourier Transforms for Fun and Profit," Fall Joint Computer Conference Proceedings, vol. 29, pp. 563-578, 1966.

(7) Bergland, G. D., "A Fast Fourier Transform Algorithm Using Base 8 Iterations," Math. Computation, vol. 22, pp. 275-279, April, 1968.

(8) Bingham, C., Godfrey, M. D., and Tukey, J. W., "Modern Techniques of Power Spectrum Estimation," IEEE Trans. Audio and Electroacoustics, Vol. AU-15, pp. 56-66, June, 1967.

(9) Blackman, R. B., and Tukey, J. W., The Measurement of Power Spectra. New York: Dover, 1958.

(10) Tukey, J. W., "An Introduction to the Calculations of Numerical Spectrum Analysis," In Spectral Analysis of Time Series, Bernard Harris, ed., New York: Wiley, pp. 25-46, 1967.

(11) Welch, P. D., "The Use of the Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." IEEE Trans. Audio and Electroacoustics, vol. AU-15, pp. 70-73, June, 1969.

(12) Bergland, G. D., "Fast Fourier Transform Hardware Implementations - An Overview," IEEE Trans. on Audio and Electroacoustics, vol. AU-17, No. 2, pp. 104-108, June, 1969.

(13) Bergland, G. D., "Fast Fourier Transform Hardware Implementations - A Survey," IEEE Trans. on Audio and Electroacoustics, vol. AU-17, No. 2, pp. 109-119, June, 1969.

(14) Noll, A. M., "Cepstrum Pitch Determination," The Journal of the Accoustical Society of America, vol. 41, No. 2, pp. 293-309, Feb., 1967.