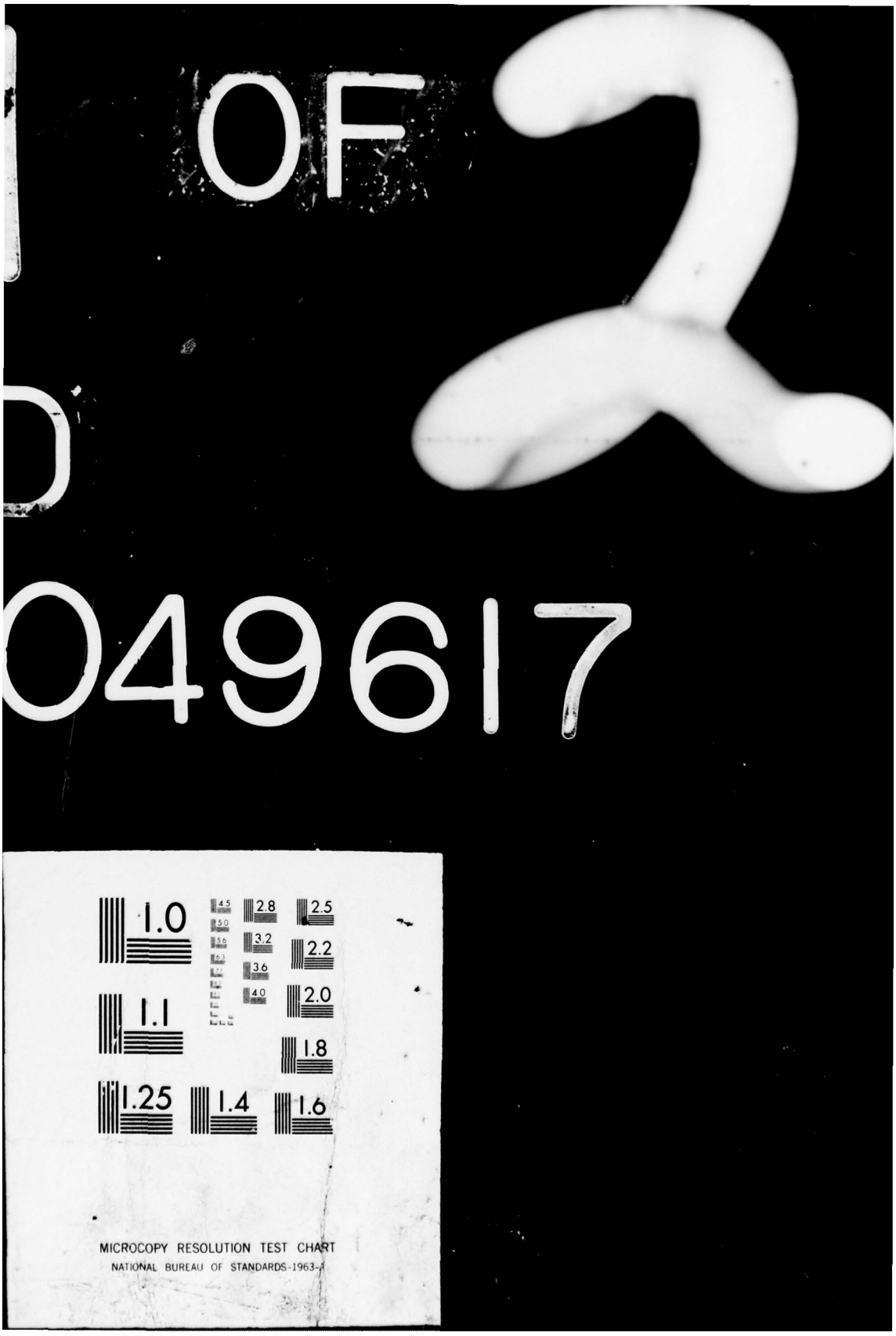
AD-A049 617 SYRACUSE UNIV N Y DEPT OF INDUSTRIAL ENGINEERING AND--ETC F/G 9/2
MATRIX INVERSION USING THE RADC STARAN ASSOCIATIVE ARRAY PROCES--ETC(U)
DEC 77 P B BERRA, E OLIVER F30602-75-C-0121

UNCLASSIFIED RADC-TR-77-386 NL

1 OF 2
AD
A049617


OF
049617
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A049612

AD NO.
DDC FILE COPY

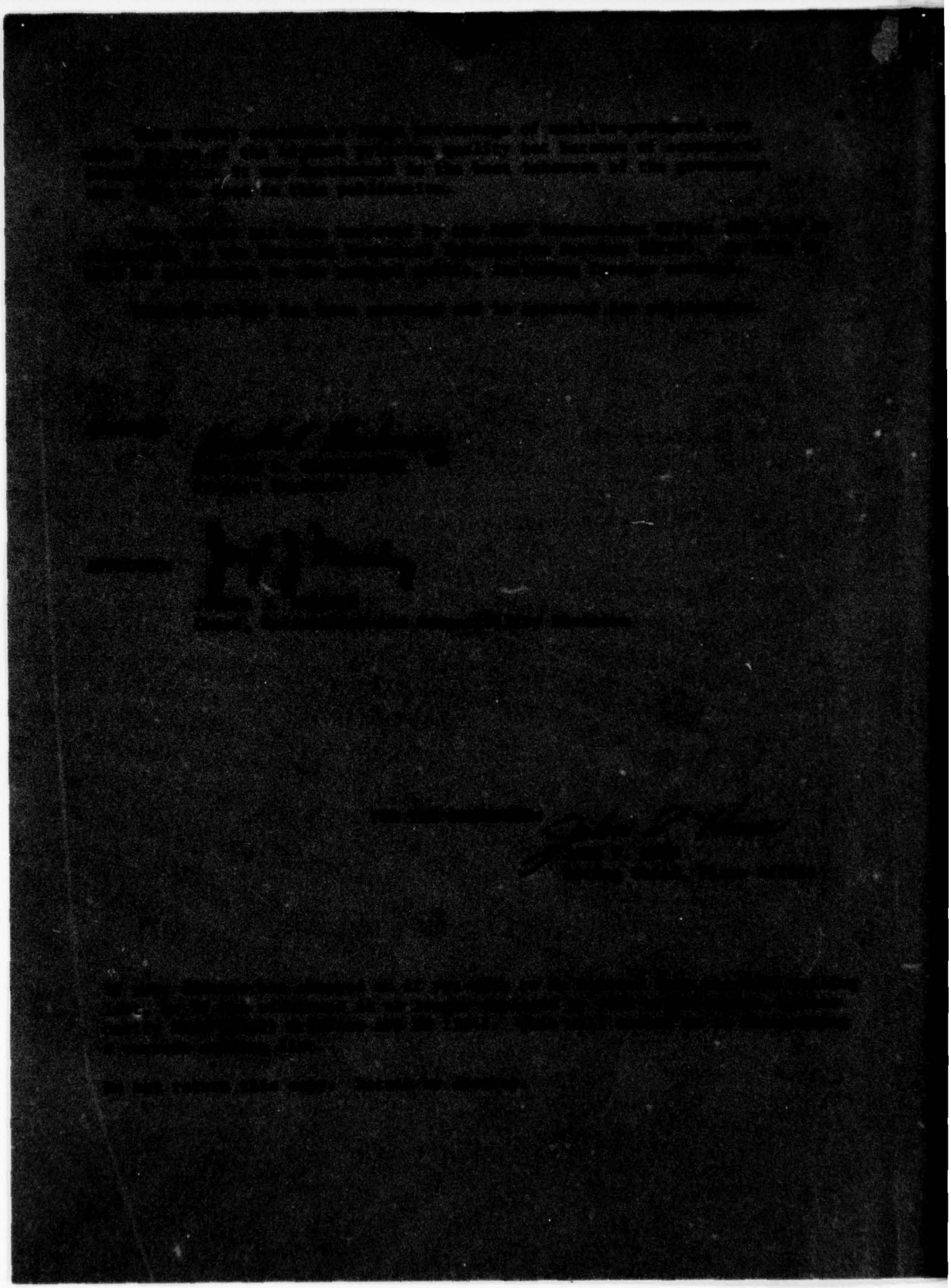
RADC-TR-77-386
Phase Report
December 1977

MATRIX INVERSION USING THE RADC STARAN ASSOCIATIVE ARRAY PROCESSOR

P. Bruce Berra
Eldon Oliver

Syracuse University

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-77-386	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: MATRIX INVERSION USING THE RADC STARAN ASSOCIATIVE ARRAY PROCESSOR.	5. TYPE OF REPORT & PERIOD COVERED: Phase Report, Sep 75 — Apr 77
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): P. Bruce Berra, Ellen Oliver	8. CONTRACT OR GRANT NUMBER(s): F30602-75-C-0121
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Syracuse University/Department of Industrial Engineering & Operations Research, Syracuse NY 13210	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 23380305
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (RBCT), Griffiss AFB NY 13441	12. REPORT DATE: December 1977
	13. NUMBER OF PAGES: 138
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Kenneth R. Siarkiewicz

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Associative Processor
Matrix Inversion
Timing Figures

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The purpose of this report is to provide support data for RADC-TR-75-73. In this research the algorithm reported ~~in this report~~ here is implemented on the RADC STARAN and then used to invert test matrices. Matrices of dimensions 30 x 30, 45 x 45, 60 x 60 and 80 x 80 are inverted and these results are timed. The same matrices were inverted using APL Plus ~~at Syracuse University~~. In the cases of the 30 x 30 and 45 x 45 matrices, a direct comparision was made; this comparison indicates a time savings of 11.82% and 15.49% respectively for the →

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

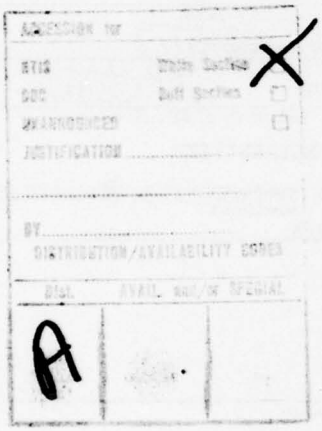
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

409 184 JOB

STARAN. For the two larger matrices, the APL Plus times were extrapolated and the comparison of this extrapolated time with the STARAN time for the 60 x 60 matrix and 80 x 80 matrix indicates a time savings of 31.77% and 62.44% respectively, for the STARAN.

In addition to reporting the test results, background material is presented to acquaint the reader with the RADCAP facility; this includes a discussion of STARAN architecture and the specific procedures required to submit a job to STARAN via the MULTICS system. The object code for inverting a 60 x 60 matrix is included and discussed in detail. Finally, recommendations for future research are discussed based upon the new STARAN Model E which has a larger array size capability.



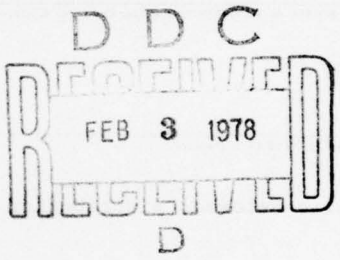


TABLE OF CONTENTS

LIST OF FIGURES (cont'd.)

LIST OF TABLES

TABLE OF ABRIEVIATIONS AND SYMBOLS

a_{ij}: A matrix element located in row i and column j

A: A thirty-two bit field in the arrays (i.e. bits 0 to 31)

ALINK: The STARAN APPLE linker

AP: Associative Processor

APPLE: Associative Processor Programming Language

AS: Array Select Register

B: A thirty-two bit field in the arrays (i.e. bits 32 to 63)

BL: Block Length Counter

D: A thirty-two bit field in the arrays (i.e. bits 64 to 95)

DP: Data Pointer

E: A thirty-two bit field in the arrays (i.e. bits 96 to 127)

EXF: External Function Logic

F: A thirty-two bit field in the arrays (i.e. bits 128 to 159)

FL1: Field Length Counter 1

FL2: Field Length Counter 2

FP1: Field Pointer 1

FP2: Field Pointer 2

FP3: Field Pointer 3

FLE: Field Pointer E

G: A thirty-two bit field in the array (i.e. bits 160 to 191)

H: A thirty-two bit field in the arrays (i.e. bits 192 to 223)

HSDB: High Speed Data Buffer

IMASK: Status Register and Comparator

JCL: Job Control Language

M: Mask Register

MAPPLE: A preprocessor which translates system macros to the assembly language.

MI(J): A mask in array I located in quarter J.

PC: Program Counter

PIO: Parallel Input/Output

RADCAP: Rome Air Development Center Associative Processor

SDM: STARAN Debug Module

X: The X response register

Y: The Y response register

Introduction

This report is a follow-on to RADC-TR-75-73 "Timing Figures for Inverting Large Matrices Using the STARAN Associative Processor" [1]. In that report an algorithm and timing figures were developed for the inversion of the A matrix in the system of linear equations

$$AX = B.$$

In this report the results of another phase of the overall effort are presented; that of actually inverting matrices utilizing Gauss elimination on STARAN. For introductory material on the project, associative processor applications and matrix inversion refer to [1] which is available from the Defense Documentation Center in Alexandria, Virginia (AD A009643).

In this report some background on STARAN is provided along with the utilization of the Rome Air Development Center Associative Processor (RADCAP) facility of which STARAN is a part. The matrix inversion process in general, and a matrix inversion application program in particular, are also discussed

RADC STARAN

The intent of this section is to acquaint the reader with RADCAP. To accomplish this it is divided into three subsections. In the first subsection an overview of the STARAN architecture is presented; next, the functional units utilized by the matrix inversion program are discussed in detail; and the final subsection gives the procedure for submitting a job to STARAN.

1. An Overview

Shown in Figure 1 is a general block diagram of the STARAN architecture. As indicated in the figure the basic components are the associati-

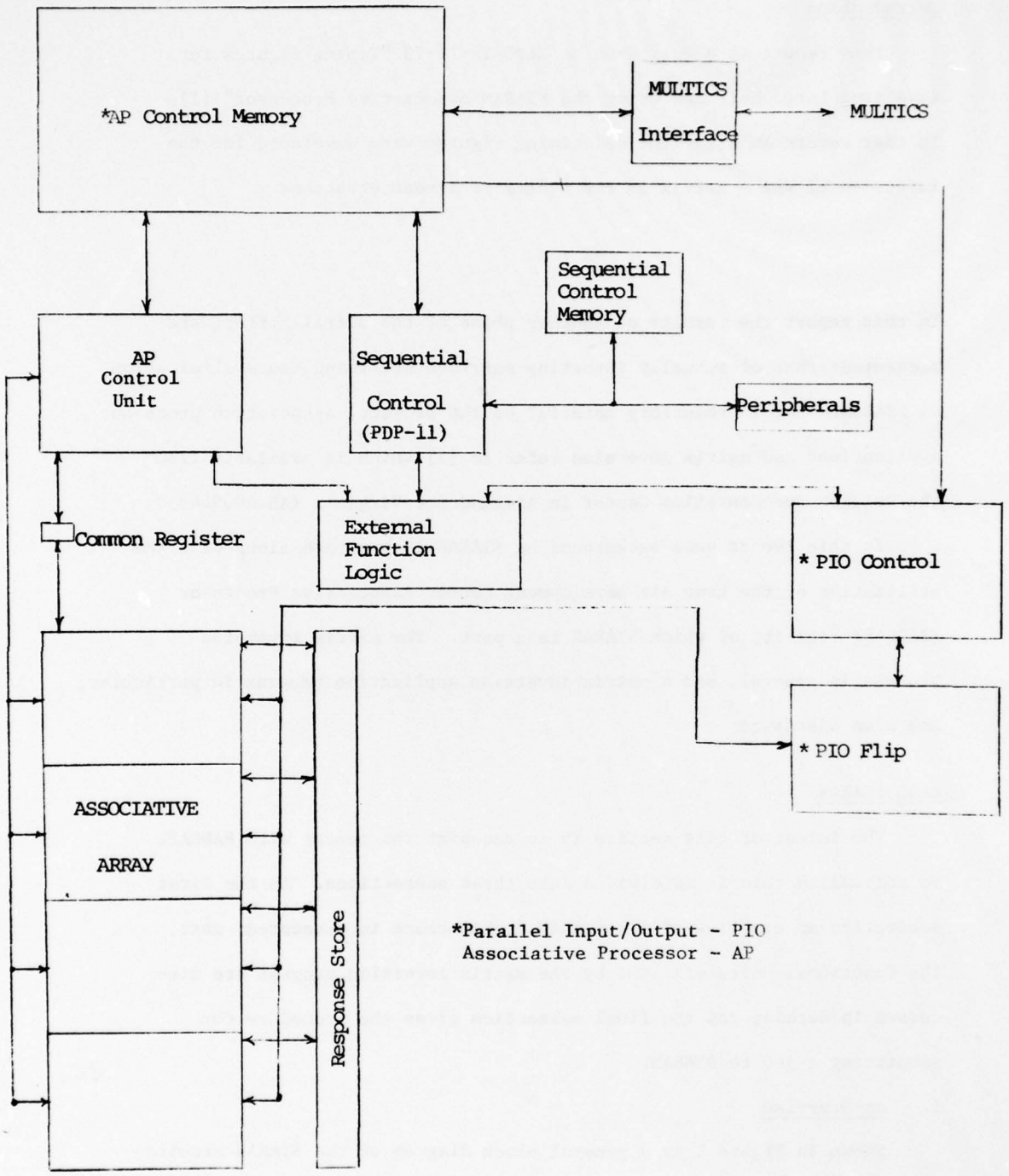


FIGURE.1. Block Diagram of the STARAN Architecture

array, response store, common register, AP control unit, AP control memory, sequential control, sequential control memory, external function logic, PIO control, and PIO flip. In this report the term Associative Processor will be used to include all of these units.

The associative array is a hardware unit in which data are stored. With STARAN the basic array size is 256 x 256. That is, there are 256 words, and each word is 256 bits in length. The RADC STARAN configuration has four (4) arrays, thus yielding a total array size of 1024 words with each word being 256 bits in length or 1024 x 256 = 262,144 total bits of array memory.

Associated with each word of the array is a three bit response store. It provides arithmetic capabilities, read/write capabilities and an indication of the results of logical operations. It is also used for masking words that are not desired in a particular parallel operation.

The common register is 32 bits long and is used in various arithmetic and search operations on the data in the array. It is also used for loading the array in a "parallel by bit," "serial by word" basis.

Among other things, the AP control unit directs the execution of the instructions that are stored in the AP control memory, which contains all or part of the user program that is being executed.

The PIO control unit controls the PIO flip network which can shift and rearrange data so that parallel arithmetic, search and logic operations can be performed in a variety of ways among the words of the array modules assigned to it. While AP control is processing data in some array modules, PIO control can input and output data in other array modules.

System diagnostics and peripheral devices are handled by sequential

control, a Digital Equipment Corporation (DEC) PDP-11 minicomputer. Associated with sequential control is the sequential control memory.

Synchronization of the three control units, AP control, PIO control and sequential control, is coordinated by the external function (EXF) logic.

The AP has a large number of search and arithmetic instructions. In terms of search there are several instructions that can be executed that process all activated words in the array. Some of these are equal (exact match), next higher, next lower, maximum, minimum, less than or equal to, less than and greater than. As an example consider an exact match search. The word that is to be used as the search criteria is placed in the common register. (Actually, the common register may have to be loaded up to eight times since it is only 32 bits long.) The word is then compared on a bit slice basis with each activated word in the array. Those that match on all 256 bit slices (or any prescribed subset of the 256) will be flagged in the response store. These words will have the same content as the one in the common register (for the prescribed subset of the 256). The responders can then be processed as prescribed by the program. The other search instructions are processed in a similar fashion.

The arithmetic instructions include add, subtract, multiply or divide one field by another field and place the result in a third field. This is done on an intraword basis with all activated words taking place in the operation simultaneously. Also, these operations can be performed between the common register and any prescribed field in the array. As an example, suppose there are two columns of numbers $x_1,\ldots,x_{1024}$ and

$y_1,\ldots,y_{1024}$ and one would like to perform the following operation: $x_i + y_i = z_i$ for $i = 1,\ldots,1024$. With a single instruction, all pairs of x and y values could be added simultaneously and the resulting z values placed in a third field.

In addition, one can add, subtract, multiply or divide any field by a scalar. For example, suppose one would like to obtain $2x_i$ for $i = 1,\ldots,1024$. In this case one would multiply the x_i by the number in the common register (two), and obtain the result in the same or an additional field.

2. STARAN Details

Shown in Figure 2 is a block diagram of STARAN as utilized by the matrix inversion program. Each of the blocks is discussed in detail in the following paragraphs.

The matrix inversion program and the matrix data are stored as segments in MULTICS. At execution time, they are moved from MULTICS to the PDP-11 disk. The program and data for STARAN execution are loaded into AP control memory from the PDP-11 disk. After execution, output is moved from STARAN via the PDP-11 disk to MULTICS and stored as a MULTICS segment. Details of these steps are included in the third subdivision of this section.

In addition to providing the communication link between MULTICS and STARAN, the PDP-11 has other functions. Assembly and debugging of STARAN programs are handled by the PDP-11 as well as housekeeping functions, STARAN maintenance and STARAN diagnostics. Also, the PDP-11 provides the means to initially load AP control memory with the previously assembled object module and any required data.

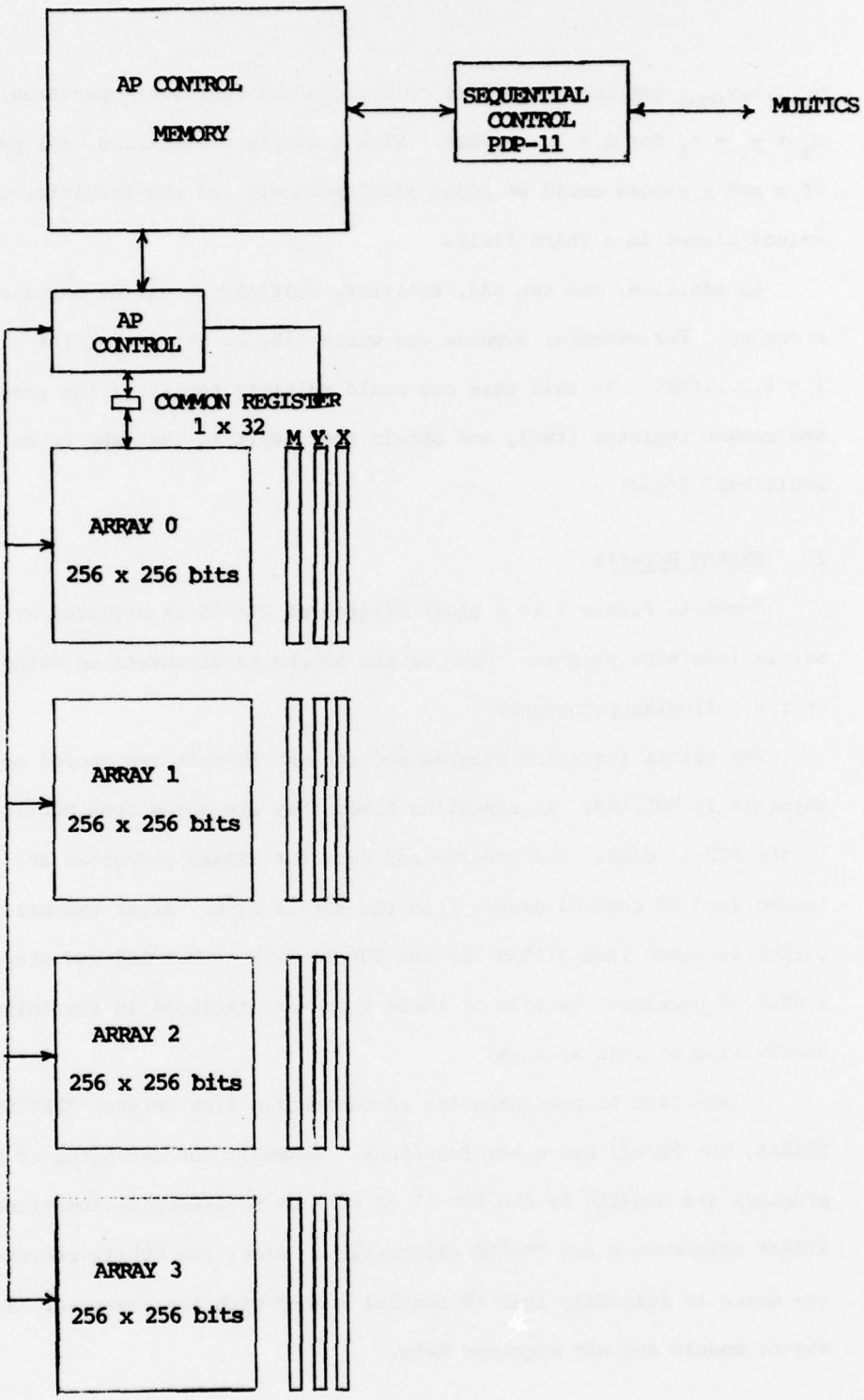


FIGURE 2. A Block Diagram of STARAN Architecture As Utilized By The Matrix Inversion Program.

As shown in Figure 3, AP control memory is partitioned into several sections: the Page Memories, the High Speed Data Buffer (HSDB) and Bulk Core Memory. The three Page Memories each have 512 32-bit words and use solid-state elements which have cycle times of less than 200 nanoseconds. Page 0 is used to store a subroutine library. Pages 1 and 2 are used in a ping-pong fashion; that is, AP control reads instructions from one page while the other page is being loaded by the program pager. Each memory has a port switch to prevent premature access before it is loaded.

The HSDB is a 512 32-bit memory which also uses solid state elements whose cycle times are less than 400 nanoseconds. Its purpose is to provide a convenient place to store data and instructions for quick access.

The Bulk Core Memory has 16,384 32-bit words of storage; it is composed of nonvolatile core with a cycle time of less than one microsecond. Bulk Core Memory is used to store the assembled program and any data required by the program.

The primary function of AP control is to control the associative arrays as directed by instructions stored in AP control memory. Shown in Figure 4 is a detailed block diagram of AP control as it appears in [4] on page 1-10. AP control fetches an instruction from AP control memory and places it in a 32-bit instruction register; the address of the instruction is contained in a 16-bit program counter. AP control consists of nine basic elements which are discussed below.

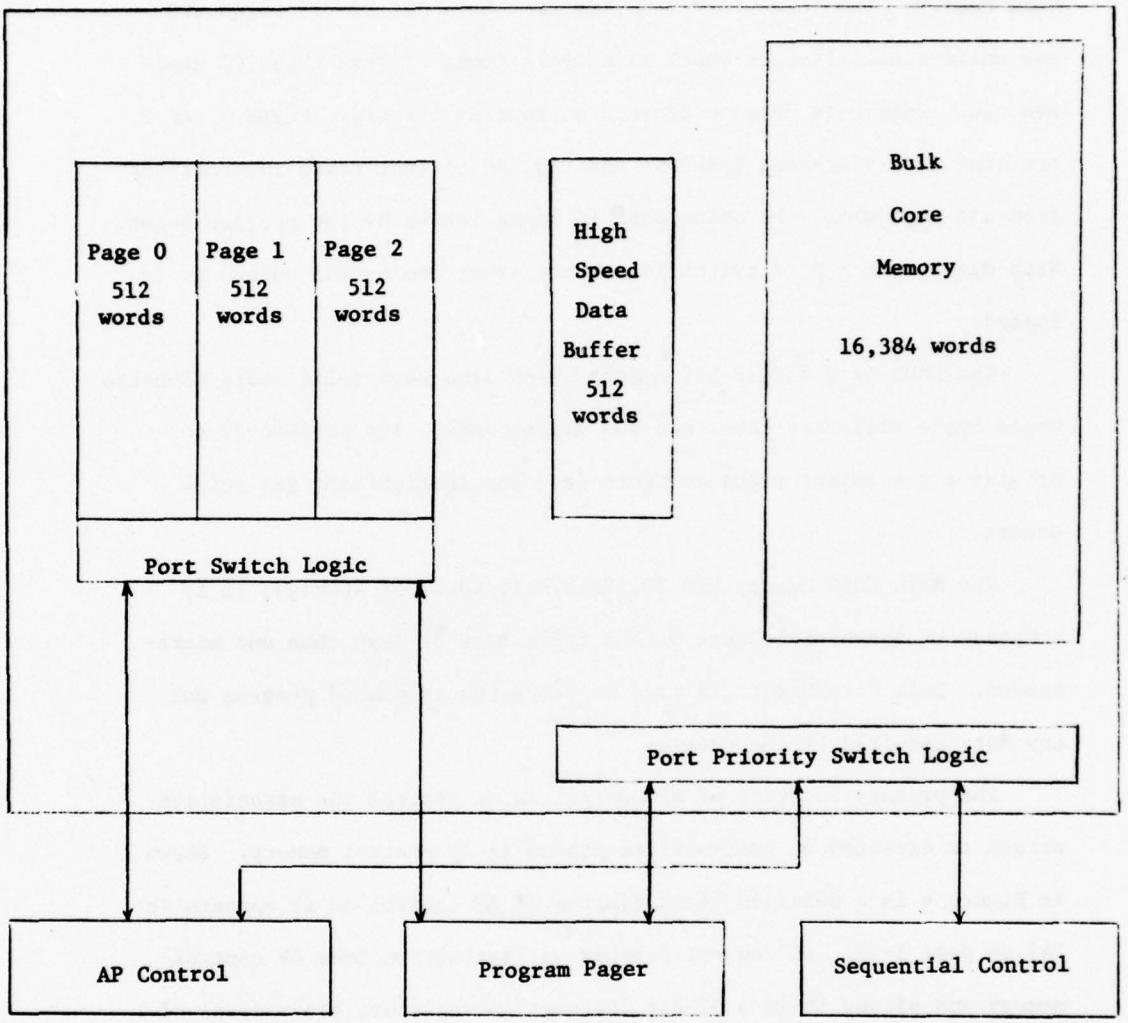


Figure 3: AP Control Memory

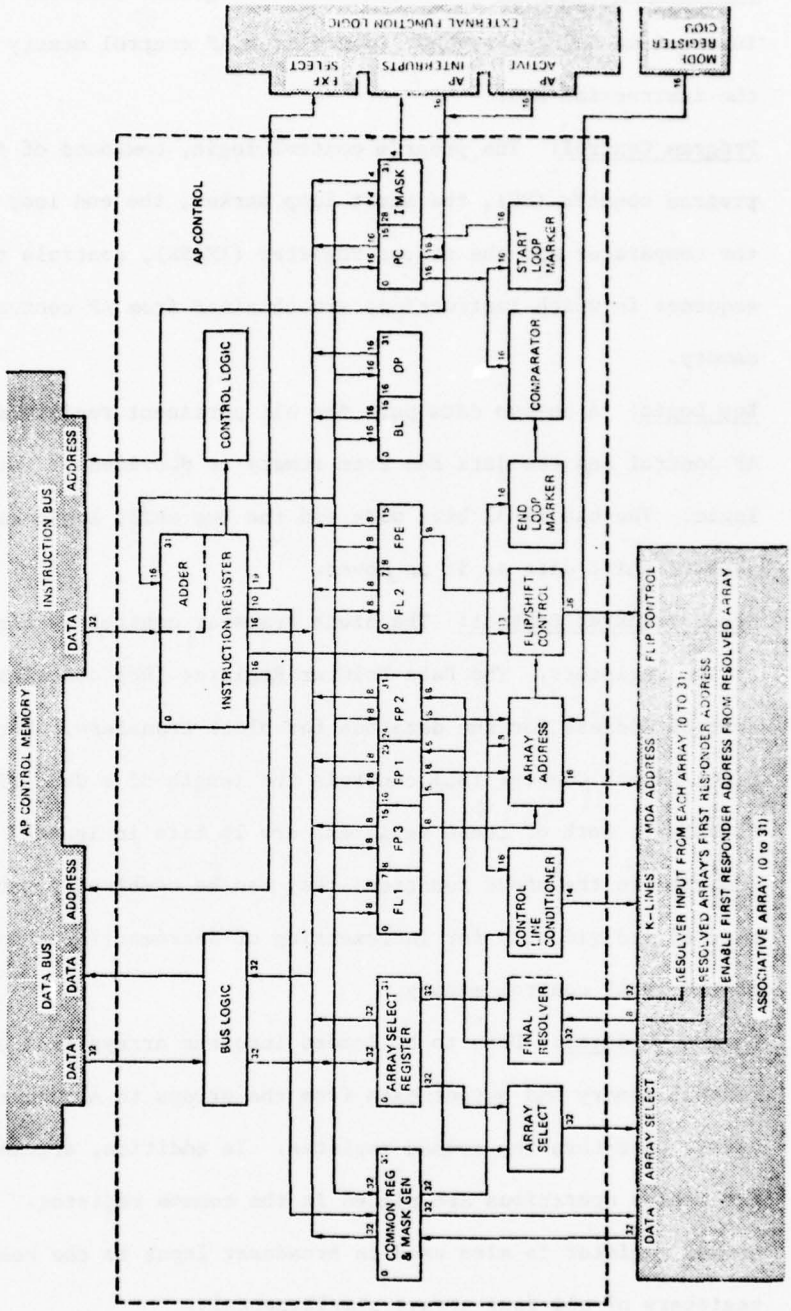


Figure 4: STARAN AP Control

1) Instruction Register: The instruction register contains the instruction being executed, loaded from AP control memory via the instruction bus.

2) Program Control: The program control logic, composed of the program counter (PC), the start loop marker, the end loop marker, the comparator and the status register (IMASK), controls the sequence in which instructions are obtained from AP control memory.

3) Bus Logic: A common data path for all pertinent registers of AP control and the data bus from memory is provided by the bus logic. The bus is 32 bits wide and the bus shift logic can be used to shift data as it is moved.

4) Block Transfer Control: The block transfer control is composed of two registers. The Data Pointer Register (DP) contains the control address for the data bus for block transfers. The Block Length Counter (BL) controls the length of a data block transfer. Both of these registers are 16 bits in length. In addition to the above function, they can be combined together or used individually for incrementing or decrementing counters stored in AP control memory.

5) Common Register: Data to be loaded into the arrays from AP control memory and output data from the arrays to AP control memory pass-thru the common register. In addition, arguments for search operations are placed in the common register. The common register is also used to broadcast input to the response registers of all four arrays simultaneously.

6) Field Pointers and Length Counters: Field pointers one and two (FP1, FP2) are used for indirect addressing of the arrays; FP1 is loaded with the array number and FP2 is loaded with the number corresponding to the word or bit slice desired. The two field length counters (FL1, FL2) are used as counters for branching and loop instructions. Field pointers three and E (FP3, FPE) are used for array bit or word addresses; FPE is also used during multiply and divide routines and to store shift constants. All of the above registers are 8 bits long.

7) Response Store Control: The control signals required by the associative arrays and buffers for correct timing are generated by the response store control The response store control consists of the control line conditioner and control line buffer.

8) Array Control: The array control logic selects the arrays to be used and controls such things as bit/word mode, mask operations and shifting. The array select register (AS) is a 32-bit register used to enable the desired array(s). A one in the bit position corresponding to the array number enables the array. Bit slice or word slice addressing is controlled by the array address mode. The shift logic required for shifting and mirroring operations is generated by control signals from the flip/shift control.

9) Resolver: Resolver logic is used to find the array address and word address of the first responder of some search o

As indicated earlier, the four associative arrays are each organized as a square 256 words by 256 bits of solid-state storage. Access to the arrays is possible in either the word or the bit direction. The arrays may be operated on individually, all at once or in various combinations as enabled by the array select register discussed above. Within each array, it is possible to operate on all or just part of the array by using a masking operation. Masking in the word direction is done by loading the M response store register with ones in the position of the words to be used; masking in the bit direction is accomplished by directing the common register to operate on a specified field. The X and Y response store registers can be used to logically combine data for storage into the arrays or the M register. In the application program discussed subsequently, we make frequent use of the X and Y registers for data movement and for loading the M register.

3. Utilizing RADC STARAN

There are four required steps for utilizing STARAN at RADC. First, the source code must be written and stored as a MULTICS segment; second, the source program must be assembled resulting in an object module; third, the object module must be linked with any required subroutines to produce a STARAN load module; and fourth, the load module must be executed.

The first step is to write the source program. Programs to be executed on STARAN are written in the Associative Processor Programming Language (APPLE). The MULTICS editors "edm" and "QEDX" can be used to create the program, edit it, and store it into a segment.

The second step in submitting a job to STARAN is to create a MULTICS segment which contains PDP-11 batch Job Control Language (JCL) statements

[6]. This segment will reference an APPLE source program, another MULTICS segment, calling it to be moved from MULTICS to PDP-11 disk 0. If the program contains or references any mnemonics, then MAPPLE is called to translate the source program into an APPLE object module. MAPPLE is a collective term used to indicate that both the MAPPLE preprocessor for translating mnemonics as well as the APPLE assembler are executed, [2,3]. The newly created object module can be stored on disk 1 of the PDP-11 or as a MULTICS segment. A sample MULTICS segment to accomplish the above is included in Appendix A1.

The third step is to create a multics segment to run the APPLE linker, ALINK [5]. Its purpose is to accept multiple object modules as input; relocate those object modules and assign them absolute addresses; resolve symbolic references among them; create an overlay structure upon request; generate a load map to indicate the absolute addresses of the load module and the entry point of each load module; and to produce an executable code, the STARAN load module, in a format suitable for loading and execution. The STARAN load module is then stored on disk 1 until required. An ALINK program and load map are included in Appendix A2.

The last step is the execution of the STARAN load module. Again, a MULTICS segment is created with the appropriate JCL statements. In this case the machine is instructed to execute STARAN Debug Module (SDM) which is a system program to detect and locate errors in an application program [5]. The functions provided by SDM include the ability to dump the contents of memory locations, to inspect and change a memory location or register; and to print a table of preselected memory locations and/or registers in AP Control Memory, Parallel I/O Control Memory and Array

Memory. The functions desired are included in the MULTICS segment to be submitted to STARAN. An example program is included in Appendix A3.

Matrix Inversion

The algorithm utilized for matrix inversion in this work is basically Gaussian elimination adapted for use on an associative processor. Column operations are performed rather than row operations, and only one row of the identity matrix is appended at any given time. The reader is referred to [1] for an example that illustrates this method.

Because large matrices are to be inverted, it is desireable in testing to use a matrix which can be generated automatically and whose inverse is known. Matrices generated with the pattern as illustrated in Figure 5 have inverses with the pattern as illustrated in Figure 6. Thus, matrices of this form were chosen for testing.

The system-supplied macros for multiplication, division and subtraction require three empty fields for intermediate operations. Therefore, the largest matrix which will fit into the four arrays for inversion is 63 x 63. A 60 x 60 matrix was chosen since it is large enough to nearly fill all of the arrays. As shown in Figure 7, the data are loaded into the arrays. The numbers in the arrays indicate the columns of the original matrix and the numbers to the left of the arrays indicate the word number. For example, column 1 of the original matrix is replicated four times in field A of each array starting at words 0,64, 128 and 192. The letters at the top of the figure are the names assigned to the 32-bit fields.

1	2	3	4
2	2	3	4
3	3	3	4
4	4	4	4

4 x 4 MATRIX

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	3	3	4	5	6	7	8	9	10	11	12	13	14	15
4	4	4	4	5	6	7	8	9	10	11	12	13	14	15
5	5	5	5	5	6	7	8	9	10	11	12	13	14	15
6	6	6	6	6	6	7	8	9	10	11	12	13	14	15
7	7	7	7	7	7	7	8	9	10	11	12	13	14	15
8	8	8	8	8	8	8	8	9	10	11	12	13	14	15
9	9	9	9	9	9	9	9	9	10	11	12	13	14	15
10	10	10	10	10	10	10	10	10	10	11	12	13	14	15
11	11	11	11	11	11	11	11	11	11	11	12	13	14	15
12	12	12	12	12	12	12	12	12	12	12	12	13	14	15
13	13	13	13	13	13	13	13	13	13	13	13	13	14	15
14	14	14	14	14	14	14	14	14	14	14	14	14	14	15
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

15 x 15 MATRIX

FIGURE 5. Patterned Matrices.

$$\begin{matrix} -1 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -3/4 \end{matrix}$$

INVERSE OF 4 x 4 MATRIX

$$\begin{matrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -14/15 \end{matrix}$$

INVERSE OF 15 x 15 MATRIX

FIGURE 6. Inverses of the Patterned Matrices

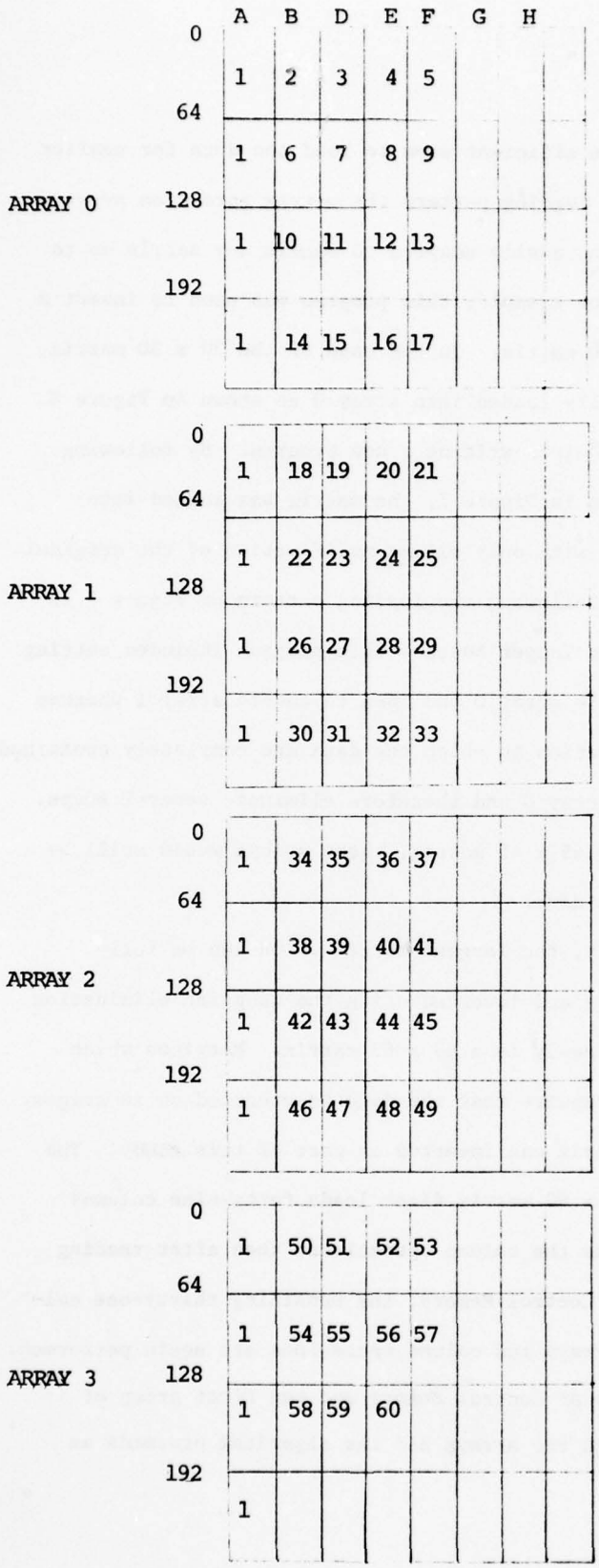


FIGURE 7. A 60 x 60 Matrix Loaded Into the Four Arrays.

Although there are more efficient ways to load the data for smaller matrices, by following this loading pattern the matrix inversion program presented in this work can be easily adapted to handle any matrix up to and including a 63 x 63. For example, this program was used to invert a 45 x 45 matrix and a 30 x 30 matrix. In the case of the 30 x 30 matrix, the data could have been fully loaded into array 0 as shown in Figure 8. However, this would have required writing a new program. By following the loading pattern as shown in Figure 7, the matrix was loaded into arrays 0 and 1 and inverted with only slight modification of the original code. The disadvantage in following the loading pattern of Figure 7 is that the inversion will take longer because this program includes setting the registers first to enable array 0 and then to enable array 1 whereas the more efficient configuration in which the data are completely contained in array 0 would use only array 0 and therefore eliminate several steps. However, in the case of the 45 x 45 matrix, three arrays would still be required to contain all the data.

As mentioned previously, the largest matrix which can be fully contained in the four arrays and inverted using the Gaussian elimination algorithm referred to previously is a 63 x 63 matrix. Matrices which are larger than a 63 x 63 require that the data be operated on in stages. For example, an 80 x 80 matrix was inverted as part of this study. The algorithm to invert the 80 x 80 matrix first loads forty-nine columns into the arrays and performs the column operations; then after reading these results back into AP Control Memory, the remaining thirty-one columns are loaded into the arrays and column operations are again performed. These results are read into AP Control Memory and the first group of columns is again loaded into the arrays and the algorithm proceeds as

ARRAY 0

A	B	D	E	F	G	H	
1	2	3	4	5			
1	6	7	8	9			
1	10	11	12	13			
1	14	15	16	17			
1	18	19	20	21			
1	22	23	24	25			
1	26	27	28	29			
1	30						

Figure 8: An Efficient Way to Load a 30 x 30 Matrix

above. The 80 x 80 matrix was chosen since it was necessary to consider a matrix large enough to require more than the STARAN array capacity, yet small enough to be contained in AP Control Memory.

Matrix Inversion Application Program - General

In this discussion the following notation will be used. Elements of the original matrix will be designated by a_{ij} where i is the row number and j the column number. Recall, the numbers in the boxes of the array diagrams represent the column number of the original matrix. Masks will be denoted as MI(J) where I is the array number (0,1,2,3) and J the position of the mask in the array (1,2,3,4). For example, M0(3) means that array 0 has a mask in the third quarter, that is, words 128 to 191.

The data are loaded into the four arrays as previously illustrated in Figure 7. Each box is sixty-four words in length thus dividing the arrays into four equal quarters in the word direction. In the case of the 60 x 60 matrix, each box will have 60 words of matrix data, one word of identity and three empty words. The identity row is appended to the top of the original matrix and will move down through the matrix row by row as the inversion proceeds.

Each array is divided into eight thirty-two bit fields in the bit direction. Fields A,B,C,E, and F are used to hold matrix data; fields G and H are used for intermediate results of the arithmetic operations; and the last field is used to store the masks and to provide a bit slice required by the arithmetic operations.

The program begins by dividing field A with the first diagonal element of the matrix, a_{11}, to create the identity element (one) in this diagonal position. Note that the identity row is in word 0. Masks M0(1), M1(1),

M2(1) and M3(1) are loaded. Field A is multiplied by elements $a_{1,2}$, $a_{1,18}$, $a_{1,34}$ and $a_{1,50}$ by successively loading the above matrix elements into the common register and using the system-supplied macro to multiply field A by the common register. The results of the multiplication are placed in field H. Next, Masks M0(2), M1(2),M2(2) and M3(2) are loaded, and field A is successively multiplied by elements $a_{1,6}$, $a_{1,22}$, $a_{1,38}$ and $a_{1,54}$ in the same way as above. The third step is to load mask M0(3), M1(3), M2(3) and M3(3) and multiply field A by elements $a_{1,10}$, $a_{1,26}$, $a_{1,42}$ and $a_{1,58}$ as above. Lastly masks M0(4), M1(4), M2(4) and M3(4) are loaded and multiplied by elements $a_{1,14}$, $a_{1,30}$ and $a_{1,46}$. Note that element $a_{1,62}$ does not exist since the matrix dimension is 60 x 60. At this point field H is full; the mask is set for all words; and the system macro for subtraction is used to subtract field H from field B placing the difference in field G. Field G is then moved to field B. This results in creating the identity element (zero) in each of the positions given above. Moving next to field D, the operations are repeated with the first element of each column in field D until field H is full of products. Then, field H is subtracted from field D, again placing the difference in field G and later moving field G to field D. In the same way fields E and F are operated on. The result of the above procedure is that the identity row which was originally appended to the top of the matrix has now been created in the first row of the matrix; what was previously the identity row is the first of the intermediate results leading to the inverse of the original matrix.

The second phase of the algorithm begins by exchanging the position of columns one and two; the result is shown in Figure 9. The identity row is in the word 1; and the same procedure as above is used; that is, first dividing field A by $a_{2,2}$ and then successively multiplying and

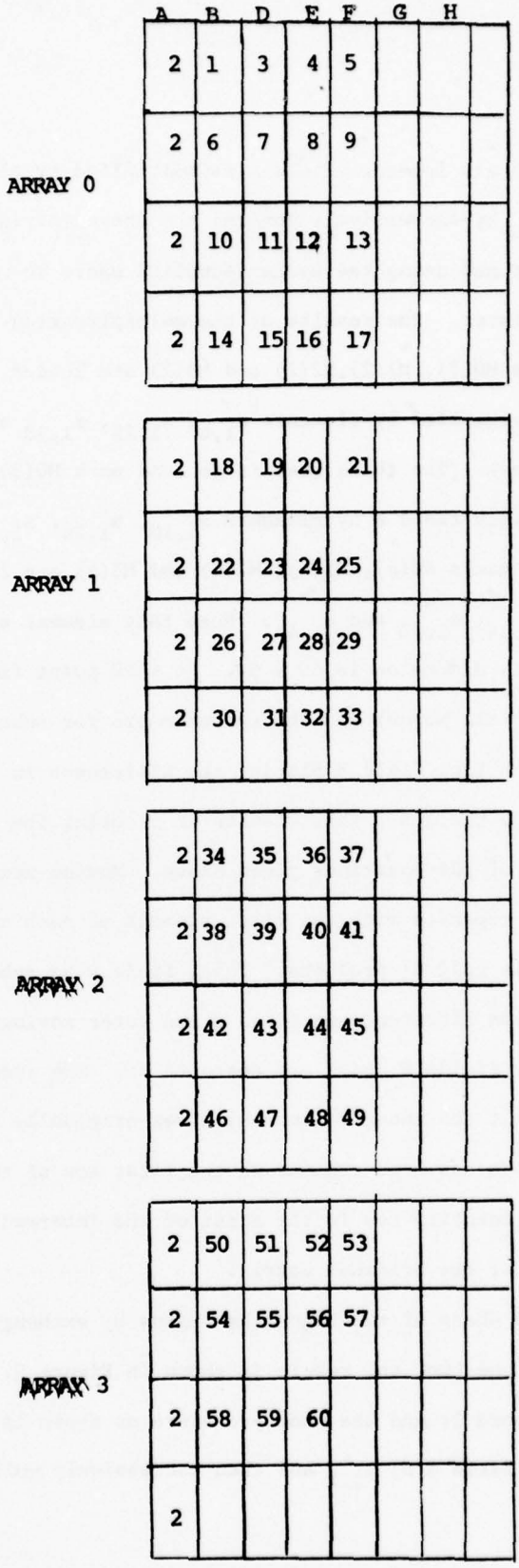


FIGURE 9. The Matrix Configuration after the First Iteration

subtracting using fields B,D,E and F. The result is the creation of the identity row in the word 2. Columns two and three are exchanged, and the arithmetic operations are repeated. Then columns three and four are exchanged and so forth until each column has been successively placed in field A. The final arrangement of the data will be as shown in Figure 10; this will be the inverse of the original matrix. A flow diagram of the general procedure is given in Figure 11.

The reader is referred to Appendix B for a detailed discussion of the inversion program for a 60 x 60 matrix. The appendix includes a figure indicating the program variables, a trace map, detailed flow charts of the program and all subroutines, explanations of the main program and all the subroutines, and and listing of the main program and the subroutines.

Timing of Matrix Inversion

For purposes of comparison, the same patterned matrices inverted on STARAN were inverted using the APL Plus system monadic DOMINO at Syracuse University. Matrices with dimensions 5,10,15,20,25,30,35,40 and 45 were inverted; the times, in seconds, for each of these matrices are summarized in Table I. The program which calculated these times inverted each matrix ten times and returned the average of the ten inversion times.

The matrix inversion routines executed on STARAN were broken down to give the inversion time alone, the time to load AP Control Memory with the data from the PDP-11 disk 0, the time to load the arrays with the data and the time to run the complete routine. It should be noted

ARRAY 0

A	B	D	E	F	G	H
60	1	2	3	4		
60	5	6	7	8		
60	9	10	11	12		
60	13	14	15	16		

ARRAY 1

60	17	18	19	20		
60	21	22	23	24		
60	25	26	27	28		
60	29	30	31	32		

ARRAY 2

60	33	34	35	36		
60	37	38	39	40		
60	41	42	43	44		
60	45	46	47	48		

ARRAY 3

60	49	50	51	52		
60	53	54	55	56		
60	57	58	59	60		
60						

FIGURE 10. The Matrix Configuration at the end of the Program

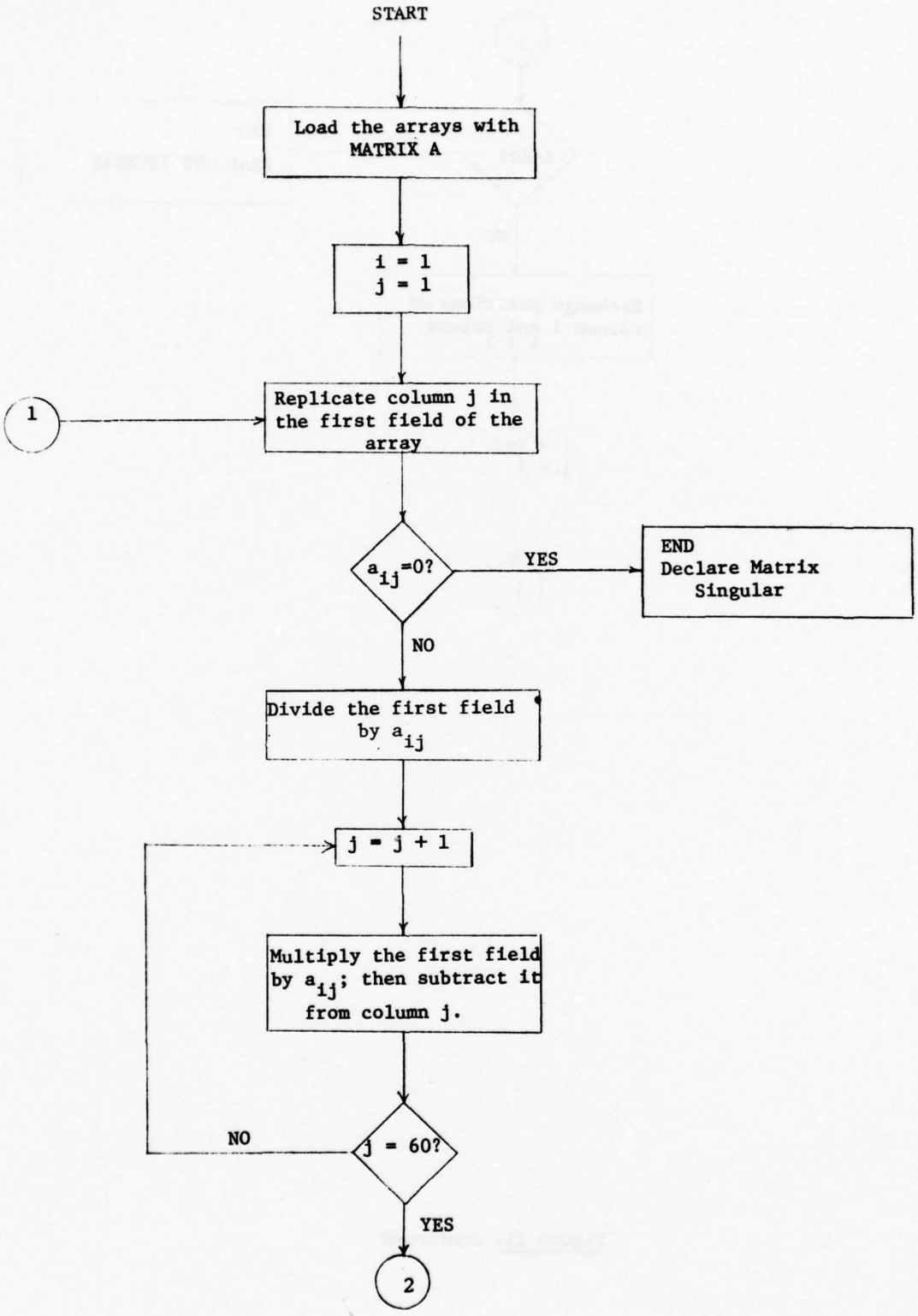


FIGURE 11: Flow Chart of General Matrix Inversion Procedure

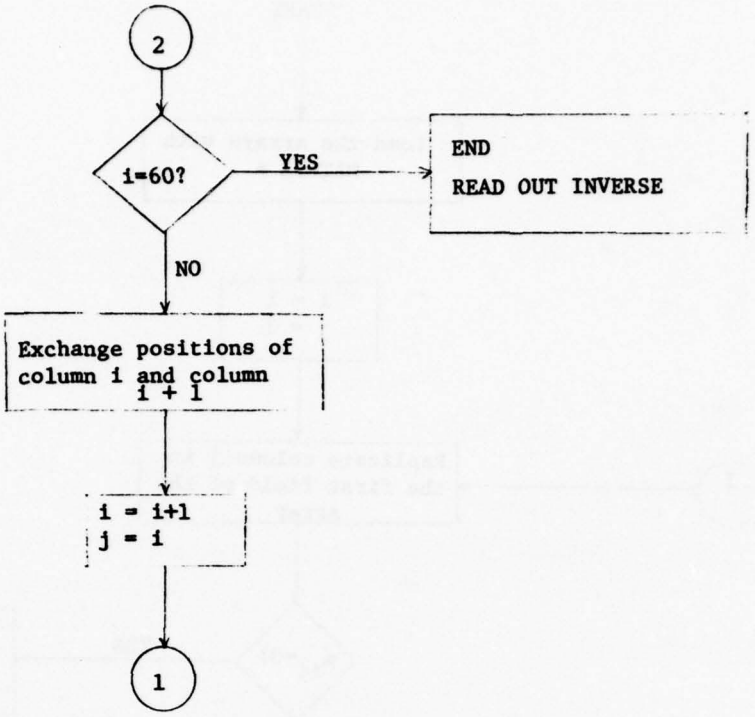


Figure 11: Continued

Table I. Timing Figures for Inverting Matrices using APL at Syracuse University

Rank of Matrix	Time in Seconds
5	0.017
10	0.117
15	0.300
20	0.683
25	1.300
30	2.200
35	3.300
40	4.756
45	6.767

that for the 30 x 30, 45 x 45 and 60 x 60 the arrays were loaded once; however, for the 80 x 80 matrix, the arrays were loaded and unloaded 160 times with intermediate data. Each matrix was inverted five times, and the timing data were collected for each operation stated above. These data are given in Tables II- V. The results are summarized in Table VI.

Shown in figure 12 is a graph which gives a direct comparison of the matrix inversion as executed on the two systems. The upper curve is a plot of the APL Plus inversion times; this curve has been extrapolated to an 80 x 80 matrix. It was necessary to extrapolate to the larger sized matrices since the standard work space provided at Syracuse University could not invert a matrix larger than 47 x 47. Also included in Figure 12 are points corresponding to the matrix inversion times using STARAN; these points are indicated by an "x" on the graph. The STARAN times which are plotted are the inversion times only and do not include loading AP Control Memory or loading the arrays with the exception of the second point for the 80 x 80 matrix. Since data movement in and out of the arrays is necessary to complete the inversion in the 80 x 80 matrix, the time to perform these operations is included in the second point to provide this comparison. It should be noted that when the inversion is executed using APL Plus, the data are in the work space and formatted in the correct way so that the numbers corresponding to the APL Plus inversion times do not include any data movement either.

Summarized in Table VII is a comparison of the inversion times for the two systems.

Future Research

Goodyear Aerospace Corporation has recently introduced STARAN Model E, which is similar in organization to the current STARAN Model B except that the size of the array is increased to 9216 x 256. The larger array size

TABLE II: Timing Data for a 30 x 30 Matrix

OPERATION	TIME IN SECONDS	HIGHEST TIME	LOWEST TIME	AVERAGE TIME
Matrix Inversion	1.9389068 1.9389004 1.9389027 1.9389081 1.9389128	1.938128	1.9389004	1.9389062
Load AP Control Memory from the PDP-11 Disk 0	1.7513823 1.7912171 1.7512208 1.7906924 1.7510153	1.7912171	1.7510153	1.7671056
Load Arrays from AP Control Memory	.0076193 .0076193 .0076193 .0076192 .0076194	.0076194	.0076192	.0076193
Complete Routine	3.6977900 3.7367847 3.7371388 3.6980502 3.7373794	3.7373794	3.6977900	3.7214286

TABLE III: Timing Data for a 45 x 45 Matrix

OPERATION	TIME IN SECONDS	HIGHEST TIME	LOWEST TIME	AVERAGE TIME
Matrix Inversion	5.7187942 5.7187957 5.7175059 5.7187961 5.7187800	5.7187961	5.7175059	5.71853438
Load AP Control Memory from the PDP-11 Disk 0	3.0164728 2.9765930 3.0565808 3.0957294 3.0561704	3.0957294	2.9765930	3.0403092
Load Arrays from AP Control Memory	.0158295 .0158296 .0158294 .0158295 .0158295	.0158296	.0158294	.0158295
Complete Routine	8.7504631 8.7504648 8.7504480 8.8286835 8.7887920	8.8286835	8.7504480	8.77377028

TABLE IV: Timing Data for a 60 x 60 Matrix

OPERATION	TIME IN SECONDS	HIGHEST TIME	LOWEST TIME	AVERAGE TIME
Matrix Inversion	13.2363561 13.2364953 13.2326334 13.2366521 13.2362944	13.2366521	13.2326334	13.2356863
Load AP Control Memory from the PDP-11 Disk 0	4.7941438 4.7163336 4.7159645 4.6763908 4.7558888	4.7941438	4.6763908	4.7317443
Load Arrays from AP Control Memory	.0274109 .0274109 .0274108 .0274109 .0274111	.0274111	.0274108	.0274109
Complete Routine	17.9809878 18.0167450 18.0215646 18.0174254 18.0171516	18.0215646	17.9809878	18.010776

TABLE V: Timing Data for an 80 x 80 Matrix

OPERATION	TIME IN SECONDS	HIGHEST TIME	LOWEST TIME	AVERAGE TIME
Matrix Inversion	22.6889600 22.6884302 22.6870001 22.6889387 22.6895190	22.6895190	22.6870001	22.6885696
Load AP Control Memory from the PDP-11 Disk 0	7.8393370 7.8794026 7.855609 7.779196 7.170973	7.8794026	7.740973	7.81888
Matrix Inversion Including Time to Load Arrays from AP Control Memory and Unload Array to AP Control Memory	33.9439881 33.9435318 33.9429724 33.9414689 33.9437018	33.9439881	33.9414689	33.9431326
Complete Routine	41.7995971 41.7227272 41.6839456 41.8024697 41.7610772	41.8024697	41.68399456	41.753966

TABLE VI: Average Time in Seconds for Inverting Matrices on RADC STARAN

MATRIX DIMENSION	30 x 30	45 x 45	60 x 60	80 x 80
Matrix Inversion	1.939	5.719	13.236	22.689
Load AP Control Memory	1.767	3.040	4.732	7.819
Load Arrays	.008	.016	.027	11.256*
Complete Routine	3.721	8.774	18.010	41.754

*This time includes loading and unloading the arrays with intermediate results.

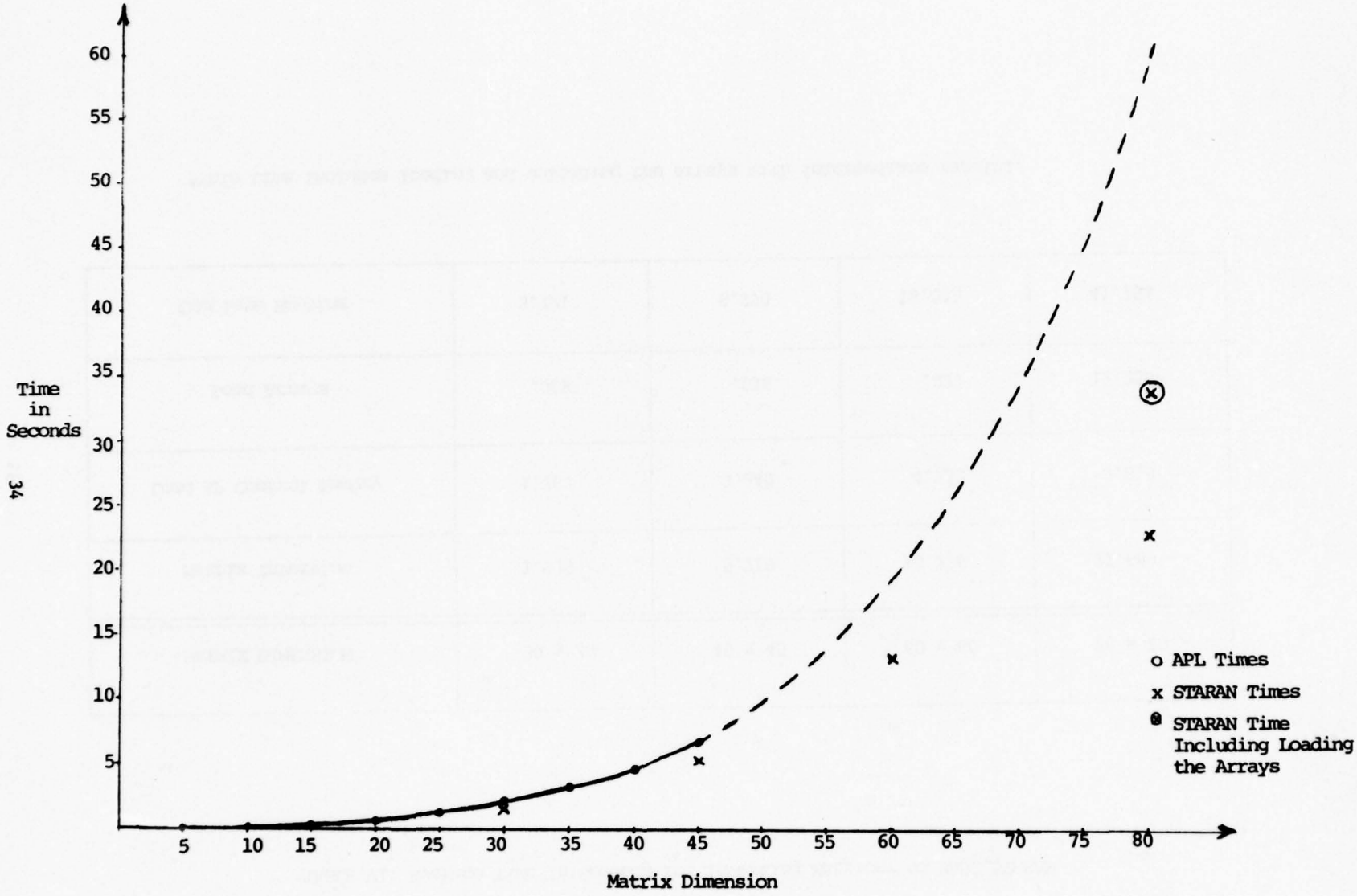


Figure 12: Matrix Inversion Times: APL Times Extrapolated to 80 x 80

TABLE VII: A Comparison of APL and STARAN

MATRIX SIZE	APL time in seconds	STARAN time in seconds	% Savings using STARAN
30 x 30	2.2	1.94	11.82
45 x 45	6.767	5.719	15.49
60 x 60	19.4*	13.236	31.77
80 x 80	60.4*	22.689	62.44
80 x 80 with data movement	60.4*	33.943	43.8

*Times as read from Figure 12.

will have a tremendous impact on the matrix inversion work. With the current array size, the largest matrix which can be fully contained in the array and inverted is 63 x 63. Once the matrix exceeds that size, the inversion time increases due to the necessary data movement. This increase was evident in the inversion of the 80 x 80 matrix. Recall that the arithmetic operations to invert the matrix required 22.689 seconds; however when the data movement required to accomplish the inversion is included, the time increased to 33.943 seconds, approximately a 50% increase in time. Assuming a configuration of four arrays, the larger array size would completely contain a matrix with dimension 510 and invert that matrix without requiring any intermediate data movement.

Another option being introduced by Goodyear Aerospace Corporation is a parallel head per track disk (PHD). The addition of the PHD would provide the capability to more rapidly load the data into the arrays. Data loading is another operation which adds to the matrix inversion time. Again recalling the 80 x 80 matrix, to load data into AP Control Memory from the PDP-11 disk 0 required 7.819 seconds; then from AP Control Memory, the arrays were loaded. Clearly the capability to directly load the arrays from a secondary device would be advantageous.

Since STARAN now has a larger array and the capability to load and unload data in a more efficient way, the inversion of very large matrices seems a feasible problem and should certainly continue to be investigated. The current STARAN, the PDP-11 minicomputer, and the MULTICS system can be used to simulate the capabilities of the STARAN Model E and its peripherals, thus providing an estimate of the time to invert very large matrices on STARAN.

Conclusions

The timing data for matrices inverted on RADC STARAN as shown in Figure 12 indicate that the times to invert matrixes of various sizes compare favorably with the times to invert the same matrices sequentially using APL Plus and thus supports the timing results of RADC-TR-75-73. Since the expected timing data of this report indicate the comparison will prove more favorable as rank increases, a fact supported by the data in this report, it can be seen that the inversion comparisons will become more dramatic as the matrix size increases.

The new technological advances in STARAN architecture discussed previously will provide the system with capabilities which will certainly have a positive impact on the matrix inversion problem.

Therefore, it can be concluded that the results thus far are promising and that further investigation into the solution of these types of problems using STARAN is reasonable.

References

[1] Berra, P.B. and Ashok K. Singhania, "Timing Figures for Inverting Large Matrices Using the STARAN Associative Processor," RADC Report No. RADC-TR-75-73, March 1975. (AD A009643)

[2] STARAN APPLE Programming Manual, GER-15637A, Goodyear Aerospace Corporation, September 1973.

[3] STARAN MACRO-APPLE [MAPPLE] Programming Manual, GER-15643, Goodyear Aerospace Corporation, September 1973.

[4] *STARAN Reference Manual*, GER-15636A, Goodyear Aerospace Corporation, September 1973.

[5] STARAN Users' Guide, GER-15644, Goodyear Aerospace Corporation, September 1973.

[6] Supplemental MULTICS/STARAN Associative Processor Programmers' Manual, Rome Air Development Center, July 1974.

APPENDIX A

The Programs Required to Assemble, Link and Execute the Matrix Inversion Program

A1. The Assembler Program mat.1

The threefold purpose of this job is to move the source program from MULTICS to disk 0 of the PDP-11, create an APPLE object module and an assembled listing of machine instructions with the corresponding address by executing MAPPLE, and finally to store the object module on disk 1 of the PDP-11 for later reference. These three sections are delineated by $FI statements; each section is discussed in detail below.

```
$FI
$JOB PIPMAT1[64,33]
$RU PIP
#DK0:/EN
#DK1:/EN
#DK1:MTX1.APL/DE
#DK0:<MU:MTX1.APL
#MU:DIR0<DK0:/DI
#MU:DIR1<DK1:/DI
```

Section one, the first job, is started by the JCL statement giving the job name and user identification code (*UIC*), *JOB PIPMAT1 [64,33] in* this case. The next statement tells the computer to run PIP. PIP enables the transfer of files from disk to disk in the PDP-11 or from disk to MULTICS and viceversa or to delete files from a disk. Disk 0 and disk 1 are enabled by the next two statements to assure access to them. Since it is required to have the most up-to-date APPLE source program stored on disk 1, delete the program currently stored. Next, move the source program MTX1.APL from MULTICS to disk 0. Note the extension .APL which is used to indicate an APPLE source program. The final two statements update the directory to the PDP-11 disk files which is stored in MULTICS.

```
$FI
$JOB MAPPL1 [64,33]
$RUN MAPPLE
#DK0:MTX1,AOB,MU:MTX1.ALS<DK0:MTX1.APL/S
```

The second section is initialized by the appropriate JCL card. Next, indicate the intention of running MAPPLE. The third statement takes the source program, MTX1.APL which is on disk 0, executes MAPPLE to create an object module, MTX1.AOB, on disk 0 and an assembled listing of the machine instructions with the corresponding addresses, MTX1.ALS, in a MULTICS segment. Note that the extension .AOB is used to indicate an object module, and the extension .ALS is used to indicate a listing.

```
$FI
$JOB PIPOUT [64,33]
$RU PIP
#DK1:MTX1.AOB/DE
#DK1:<DK0:MTX1.AOB
#DK0:MTX1.AOB/DE
#DK0:MTX1.APL/DE
#MU:DIR0<DK0:/DI
#MU:DIR1<DK1:/DI
$FI
```

The third and final section is another PIP job initialized by JCL statement. First, delete the old object module from disk 1 and then store the new object module on disk 1 from disk 0. The next two statements delete the object module and source program from disk 0. Disk 0 is cleaned periodically and also used for library routines and therefore not used for storage. Finally, finish the job by updating the disk directories on MULTICS.

The computer program mat.1 is included in this appendix as Figure 13.

```
pr mat.1

                        mat.1      10/13/76  1219.3 edt Wed

$FI
$JOB PIPMAT1[64,33]
$RU PIP
$DK0:/EN
$DK1:/EN
$DK1:MTX1.APL/DE
$DK0:<MU:MTX1.APL
$MU:DIR0<DK0:/DI
$MU:DIR1<DK1:/DI
$FI
$JOB MAPPL1 [64,33]
$RUN MAPPLE
$DK0:MTX1.AOB,MU:MTX1.ALS<DK0:MTX1.APL/S
$FI
$JOB PIPOUT [64,33]
$RU PIP
$DK1:MTX1.AOB/DE
$DK1:<DK0:MTX1.AOB
$DK0:MTX1.AOB/DE
$DK0:MTX1.APL/DE
$MU:DIR0<DK0:/DI
$MU:DIR1<DK1:/DI
$FI


r 1219 0.090 0.420 20
```

Figure 13: The Assembler Program

A2. The Link Program alink

The overall purpose of the alink MULTICS segment is to link together all object modules needed to run the entire program and to create from these a load module and load map. Note that alink is the MULTICS segment name. The alink program causes ALINK, the STARAN APPLE linker processing program, to be executed. This segment consists of three jobs delineated by $FI, each of which is discussed below.

```
$FI
$JOB PIP [64,33]
$RUN PIP
#DK0:MTX1.ALD/DE
```

The first part of the program executed PIP to delete from disk 0 the old STARAN load module, MTX1.ALD. Note that the extension .ALD is used to indicate a STARAN load module.

```
$FI
$JOB ALINK [64,33]
$RU ALINK
#DK0:MTX1.ALD,MU:MTX1.AMP<DK1:MTX1.AOB/B:9000
#DK1:MTX2.AOB,DK1:MTX3.AOB,DK1:SUBA.AOB
#DK1:SUBB.AOB,DK1:SUBC.AOB,DK1:SUBD.AOB
#DK0:FLTSUB.AOB
#DK0:FDVC.AOB,DK0:FMPC.AOB,DK0:FLTAS.AOB/E
```

The second portion of the MULTICS segment is the ALINK job. In this case, the object modules following "<" are linked together to create the STARAN load module, MTX1.ALD and the STARAN load map MTX1.AMP. The object modules which are linked together in this program are as follows:

1) The main program MTX1.AOB
2) The subroutines

   MTX2.AOB

   MTX3.AOB

   SUBA.AOB

   SUBB.AOB

   SUBC.AOB

   SUBD.AOB

3) System macros

FLTSUB.AOB	floating point subtraction
FDVC.AOB	floating point division
FMPC.AOB	floating point multiplication
FLTAS.AOB	required floating point routines

A sample load map, Figure 15, is included in this appendix following the listing of the ALINK job, Figure 14.

```
$FI
$JOB PIPEND [64,33]
$RU PIP
#DK1:MTX1.ALD/DE
#DK1:<DK0:MTX1.ALD
#MU:DIR0<DK0:/DI
#MU:DIR1<DK1:/DI
$FI
```

The last portion of this segment is again a PIP job. Its purpose is to delete the old load module from disk 1 and then store the new load module on disk 1. Finally, the directories to the disk files are updated.

A3. The Execution Program sdm

The sdm MULTICS segment is composed of three jobs delineated by $FI statements. Note that sdm is the name of the MULTICS segment which will cause SDM, the STARAN Debug Module System program, to be executed. Both the first and the last jobs are PIP jobs for moving data. The middle section is the SDM job which includes several of the debugging features. Each of these sections is discussed below.

```
$FI
$JOB PIP [64,33]
$RUN PIP
#DK0:SDM/UN
#DK0:SDM/DE
#DK0:MAT60.EXT/UN
#DK0:MAT60.EXT/DE
#DK0:MAT60.EXT<MU:MAT60
```

```
pr alink

                        alink      10/13/76  1220.3 edt Wed


$FI
$JOB PIP [64,33]
$RUN PIP
#DK0:MTX1.ALD/DE
$FI
$JOB ALINK [64,33]
$RU ALINK
#DK0:MTX1.ALD,MU:MTX1.AMP<DK1:MTX1.AOB/B:9000
#DK1:MTX2.AOB,DK1:MTX3.AOB,DK1:SUBA.AOB
#DK1:SUBB.AOB,DK1:SUBC.AOB,DK1:SUBD.AOB
#DK0:FLTSUB.AOB
#DK0:FDVC.AOB,DK0:FMPC.AOB,DK0:FLTAS.AOB/E
$FI
$JOB PIPEND [64,33]
$RU PIP
#DK1:MTX1.ALD/DE
#DK1:<DK0:MTX1.ALD
#MU:DIR0<DK0:/DI
#MU:DIR1<DK1:/DI
$FI


r 1220 0.089 0.650 25
```

Figure 14: The Link Program

```
pr MTX1.AMP

                        MTX1.AMP  10/08/76  1101.4 edt Fri

MTX1   .AMP    ALINK V02-01   08-OCT-76   10:51:23


LOAD MODULE NAME: MAIN
TRANSFER ADDRESS: ****
PROGRAM SECTIONS:       LOW     HIGH    SIZE
   <RELOC>              9000    97EB    07EC
   <ABS>                0000    0000    0000

**********
OBJECT MODULE NAME: MAIN
PROGRAM SECTIONS:       LOW     HIGH    SIZE
   <RELOC>              9000    927B    027C
   <ABS>                0000    0000    0000
PROGRAM ENTRIES:
    CONST      901F                    COUNTER    901A
    VALUE      901C                    CONST3     9018
    CONST2     9019                    CONST1     901D
    CONST0     901E
**********
OBJECT MODULE NAME: *********
PROGRAM SECTIONS:       LOW     HIGH    SIZE
   <RELOC>              927C    93E9    016E
   <ABS>                0000    0000    0000
PROGRAM ENTRIES:
    SUB1       927C
**********
OBJECT MODULE NAME: *********
PROGRAM SECTIONS:       LOW     HIGH    SIZE
   <RELOC>              93EA    953B    0152
   <ABS>                0000    0000    0000
PROGRAM ENTRIES:
    SUB2       93EA
**********
OBJECT MODULE NAME: *********
PROGRAM SECTIONS:       LOW     HIGH    SIZE
   <RELOC>              953C    957E    0043
   <ABS>                0000    0000    0000
PROGRAM ENTRIES:
    SUBA       953C
**********
OBJECT MODULE NAME: *********
PROGRAM SECTIONS:       LOW     HIGH    SIZE
   <RELOC>              957F    95C1    0043
   <ABS>                0000    0000    0000
PROGRAM ENTRIES:
    SUBB       957F
**********
```

Figure 15: The Load Map

```
OBJECT MODULE NAME: *********
PROGRAM SECTIONS:      LOW     HIGH    SIZE
   <RELOC>             95C2    9604    0043
   <ABS>               0000    0000    0000
PROGRAM ENTRIES:
    SUBC       95C2
**********
OBJECT MODULE NAME: *********
PROGRAM SECTIONS:      LOW     HIGH    SIZE
   <RELOC>             9605    9647    0043
   <ABS>               0000    0000    0000
PROGRAM ENTRIES:
    SUBD       9605
**********
OBJECT MODULE NAME: FLTSUB
PROGRAM SECTIONS:      LOW     HIGH    SIZE
   <RELOC>             9648    968F    0048
   <ABS>               0000    0000    0000
PROGRAM ENTRIES:
    NORM$      9654                RSTR$      964E
    SAVE$      9648
**********
OBJECT MODULE NAME: FDVC
PROGRAM SECTIONS:      LOW     HIGH    SIZE
   <RELOC>             9690    96EC    005D
   <ABS>               0000    0000    0000
PROGRAM ENTRIES:
    FDVCE$     96C8                FDVC$      9690
**********
OBJECT MODULE NAME: FMPC
PROGRAM SECTIONS:      LOW     HIGH    SIZE
   <RELOC>             96ED    9738    004C
   <ABS>               0000    0000    0000
PROGRAM ENTRIES:
    FMPC$      96ED
**********
OBJECT MODULE NAME: FLTAS
PROGRAM SECTIONS:      LOW     HIGH    SIZE
   <RELOC>             9739    97EB    00B3
   <ABS>               0000    0000    0000
PROGRAM ENTRIES:
    ADF1$      9753                ADC1$      976E
    ISBC2$     97C8                ALIGN$     97D4
    ISBC1$     97BB                FIELD$     9739
    ADF2$      9762                SBF1$      97B7
    ADC2$      977B                SBC1$      97A2
    COMRE$     9747                SBF2$      9796
    SBC2$      97AF

r 1103 0.525 1.698 70 level 2, 9
```

Figure 15: Continued

The first PIP job is initiallized by the appropriate JCL statement. The next two statements refer to a file called SDM which is created initially on disk 0 during the running of STARAN Debug Module. These statements unlock the previous file and then delete it in preparation for the new file. Following this, the data file for the 60 x 60 matrix is unlocked from disk 0, deleted from disk 0 and then copied from the MULTICS segment called MAT60. These steps assure that the most up-to-date version of the data is on disk 0 to be called by the program at execution time.

```
$FI
$JOB BERRAE64,333,DKO:SDM
$RUN SDM
#WA
#X
#LD DKO:MTX1.ALD/NG
#.P+0=APR
#.P+1=X0
#.P+2=Y0
#.P+3=M0
#.P+4=X1
#.P+5=Y1
#.P+6=M1
#.P+7=X2
#.P+10=Y2
#.P+11=M2
#.P+12=X3
#.P+13=Y3
#.P+14=M3
#SAP 9000
#.PI
#TSCM
```

The JCL statement for the second part of the segment has the usual job name and UIC number as well as a statement to write a file called SDM on disk 0. The SDM file on disk 0 will contain all the debugging information called for during the running of the STARAN debug module. A sample print-out of this file is also included in this appendix. After telling the system to execute the STARAN Debug Module (#RUN SDM),

a WAIT command (#WA) is given. This command enables the user's program to execute properly under conditions which might cause unpredictable results or a premature halt. The WAIT is followed by a super clear command (#Y). Next, the STARAN load module is loaded in preparation for execution; the /NG switch indicates that it is not desired to execute at this time. The thirteen statements following the load command are used to establish a print table of AP registers. In this case it is required that the contents of all AP registers and all response registers, namely X, Y, and M, be printed out. SAP is the Start Associative Processor command. The first address following the SAP command is the first address to be executed; the second address is the stopping address for the execution. The print table will print the contents of the specified registers at the stopping address. If a second address is not specified, the entire program is executed. Following the SAP command, the STARAN debug module is asked to create four MULTICS files, each to contain the contents of one of the associative arrays at the time of the stop address given above. The .PI command is used to print the contents of each entry in a Print Table. STARAN Debug Module is then terminated with #TSCM.

```
$FI
$JOB PIP2 [64,33]
$RUN PIP
#MU:<DK0:SDM/FA
#MU:DIR0<DK0:/DI
$FI
```

The final PIP job writes the SDM file with the debugging information from disk 0 to a MULTICS segment for easy access to the debugging information. Directories to the disk files are then updated before the job is concluded.

The complete sdm program is included as Figure 16 and the output of the sdm program is included as Figure 17.

```
.r sdm

                        sdm        10/13/76  1221.4 edt Wed

$FI
$JOB PIP [64,33]
$RUN PIP
$DK0:SDM/UN
$DK0:SDM/DE
$DK0:MAT60.EXT/UN
$DK0:MAT60.EXT/DE
$DK0:MAT60.EXT<MU:MAT60
$FI
$JOB BERRA[64,33],DK0:SDM
$RUN SDM
$WA
$X
$LD DK0:MTX1.ALD/NG
$.P+0=APR
$.P+1=XQ
$.P+2=YQ
$.P+3=MQ
$.P+4=X1
$.P+5=Y1
$.P+6=M1
$.P+7=X2
$.P+10=Y2
$.P+11=M2
$.P+12=X3
$.P+13=Y3
$.P+14=M3
$SAP 9000
$.FI
$TSCM
$FI
$JOB PIP2 [64,33]
$RUN PIP
$MU:<DK0:SDM/FA
$MU:DIR0<DK0:/DI
$FI


r 1221 0.112 1.112 33
```

Figure 16: The Execution Program

```
pr SDM

                        SDM        07/09/76  1209.6 edt Fri

$JOB BERRA[64,33],DK0:SDM
DATE:-05-JUL-76
TIME:-23:14:18
$RUN SDM
STARAN DEBUG
VERSION: V 2.0

*** WA


*** X


*** LD DK0:MTX7.ALD/NG

FLTSUB
LD OK


*** .P+0=APR


*** .P+1=X0


*** .P+2=Y0


*** .P+3=M0


*** .P+4=X1


*** .P+5=Y1


*** .P+6=M1


*** .P+7=X2


*** .P+10=Y2


*** .P+11=M2


*** .P+12=X3


*** .P+13=Y3


*** .P+14=M3
```

Figure 17: sdm Output

```
*** SAP 9000:90D7

AP  BREAKPOINT  SP AT  90D7

*** MU:OUT1/AR:000:0FF

*** MU:OUT2/AR:100:1FF

*** MU:OUT3/AR:200:2FF

*** MU:OUT4/AR:300:3FF

*** .PI

00  PC        = H 8075
01  IR        = H 3800 2000
02  C         = H 0000 0000
03  FL1       = H   00
04  FL2       = H   92
05  FP1       = H   9F
06  FP2       = H   02
07  FP3       = H   1F
10  FPE       = H   00
11  BL        = H 0000
12  DP        = H 0E4C
13  GET       = H 8045
14  PUT       = H 01BB
15  CNT       = H 0000
16  AS        = H F000 0000
17  SLM       = H 8057
20  ELM       = H 805C
21  IMSK      = H 000E
22  HOME      = H   20
01  X0= 00000000 00000007 40000000 00000007 40000000 00000007 40000000 00000007
02  Y0= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
03  M0= FFFFFFFF FFFFFFFF 00000000 00000000 00000000 00000000 00000000 00000000
04  X1= 40000000 00000007 40000000 00000007 40000000 00000007 40000000 00000007
05  Y1= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
06  M1= FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
07  X2= 40000000 00000007 40000000 00000007 40000000 00000007 40000000 00000007
10  Y2= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
11  M2= FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
12  X3= 40000000 00000007 40000000 00000007 FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
13  Y3= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
14  M3= FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF

*** TSCM
$FI
TIME:-23:16:30


r 1210 0.559 1.180 65 level 4, 23
```

Figure 17: Continued

APPENDIX B
Details of a 60 x 60 Matrix Inversion Program

B1. Matrix Inversion Application Program Overview

The listing of the APPLE code for the matrix inversion program and the subroutines required by the program are included in this appendix. MTX1.APL is the main program. MTX2.APL is the subroutine which does the division in field A and the arithmetic operations on fields B and D. MTX3.APL is the subroutine which does the arithmetic operations on fields E and F. SUBA.APL replicates a column from the top quarter of an array in field A; SUBB.APL replicates a column from the second quarter of an array in field A; SUBC.APL replicates a column from the third quarter of an array in field A; SUBD.APL replicates a column from the fourth quarter of an array in field A. Also, each of the replicating subroutines appends the new identity row in the proper position.

A list of the variables used by the program is shown in Figure 18; a trace map of the program and subroutines is shown in Figure 19; finally, detailed flow charts and the listings for the program and subroutines are shown in Figures 20-33.

Each of the programs is broken into small sections for ease of discussion. The small sections are first discussed in general; then the listing of the small section is given; finally, the listing is discussed in detail.

B2. The Main Program: MTX1.APL

The main program is initialized in the first section of the program as follows. The subroutines are defined as external modules giving the main program access to them; the variables in the main program are defined to permit the subroutines to access them; the 32 bit fields used in the program are defined; and the arrays are cleared in preparation for loading the new data.

A: a 32 bit field starting at bit 0

B: a 32 bit field starting at bit 32

D: a 32 bit field starting at bit 64

E: a 32 bit field starting at bit 96

F: a 32 bit field starting at bit 128

G: a 32 bit field starting at bit 160

H: a 32 bit field starting at bit 192

COUNT: The matrix dimension; keeps track of the number of column operations. Count is initialized to sixty and decremented by one after each column is used in field A. When count is zero, the program is terminated.

COUNTER: The number of word-wise division (4) is each matrix; used during the loading operation. Each array is filled by loading sixty-one words in fields B,D,E and F then skipping three words and repeating for a total of four times.

CONST3: Initially 0 but changed during the program; used to point to the desired array during the arithmetic operations of the subroutines, SUB1 and SUB2. SUB1 and SUB2 operate first on array 0, then array 1, array 2 and array 3. The value in CONST3 is loaded into FP1 to enable the correct array for the arithmetic operations.

CONST0: Initially 1 but incremented during the program; used to load into FP2 which points to the word that becomes the new identity word via the arithmetic operations of the subroutines SUB1 and SUB2.

Figure 18: Application Program Variables

VALUE: Contains a number to be loaded into the array select register (AS). The initial number 80000000 enables array 0 when loaded in AS. The number in VALUE will be changed later in the program by moving the number stored in CONST to this location. All the arrays are successively enabled by changing the number in the address VALUE.

CONST: The storage word for the next number to be placed in VALUE; intially 40000000 which will enable array 1 when loaded in AS. The number in this location will be changed by the program.

CONST4: The number 20000000 is stored here to be later loaded into CONST. This number will enable array 2 when loaded in AS.

CONST5: The number 10000000 is stored here to be later loaded into CONST. This number will enable array 3 when loaded in AS.

CONST1 and

CONST2: These are initialized to three and two respectively. During the program they are decremented and control the number to be placed in location CONST.

Figure 18: Continued

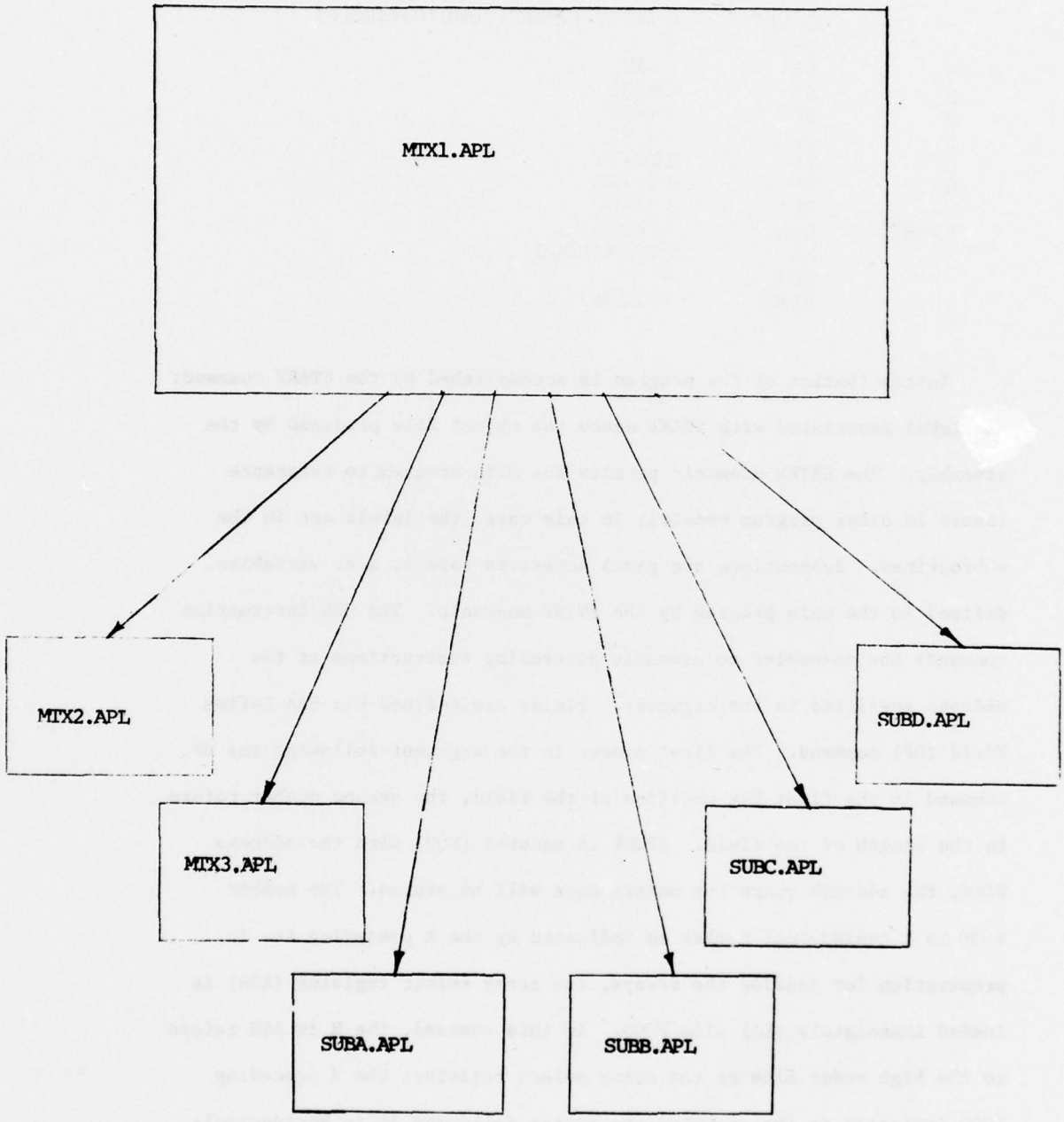


Figure 19: Trace Map

```
MAIN    START
        EXTRN   SUB1,SUB2,SUBA,SUBB,SUBC,SUBD
        ENTRY   COUNTER,FOUR,VALUE,CONST2
        ENTRY   CONST1,CONST,CONST0,CONST3
        ORG     0
A       DF      0,32
B       DF      32,32
D       DF      64,32
E       DF      96,32
F       DF      128,32
G       DF      160,32
H       DF      192,32
STORE   EQU     X'B000'
        LI      ASH,X'F000'
        SET     M
        CLRF    (0,256)
```

Initialization of the program is accomplished by the START command; the label associated with START names the object file produced by the assembly. The EXTRN mnemonic permits the main program to reference labels in other program modules; in this case, the la els are in the subroutines. Subroutines are given access to labels, i.e. variables, defined in the main program by the ENTRY mnemonic. The ORG instruction commands the assembler to assemble succeeding instructions at the address specified in the argument. Fields are defined via the Define Field (DF) command. The first number in the argument following the DF command is the first bit position of the field, the second number refers to the length of the field. STORE is equated (EQU) with the address B000, the address where the matrix data will be stored. The number B000 is a hexidecimal number as indicated by the X preceding it. In preparation for loading the arrays, the array select register (ASH) is loaded immediately (LI) with F000. In this command, the H in ASH refers to the high order bits of the array select register; the X preceding F000 indicates to the computer the number following is in hexidecimal;

the F in F000 which translates from hexidecimal to 1111 in binary enables all four arrays. The M register is set in all four arrays with the SET command in preparation for loading the arrays. Finally, the arrays are cleared (CLRF) in bit positions starting at 0 and ending at 256.

In this next section two parts of the program are considered, one of which occurs at the beginning of the program and the other at the end of the program. These two sections reference each other and work together to load the matrix data into AP control memory from a MULTICS segment.

```
            OPEN        BUFFER
            READ        DATA
WAIT1       IOWAIT      LINK,BUSY
            CLOSE       LINK

BUSY        ILOCK,1     12
            B           WAIT1
ERROR       WAIT
BUFFER      OBUFF       LINK,,DK,MAT60,EXT,4,ERROR
DATA        RBUFF       LINK,STORE,14640,3
SAVE        DS
LINK        DC,2
```

The OPEN command must precede all I/O instructions; it prepares the system for an eventual READ/WRITE instruction. The argument following the OPEN command references the label of an OBUFF instruction, BUFFER in this case. The OBUFF instruction contains the parameter information required by the OPEN instruction. Note, the OBUFF instruction occurs in the latter part of the program. In the OBUFF command are up to seven arguments, six of which are applicable in this case. LINK references the label of an expression referencing a required linkword which is two words of storage in STARAN provided by the programmer and initialized to zero. The second argument is optional; if included, it is the logical name of the data set, if not, as in this case, two successive commas are

used. The third argument refers to the name of the physical device associated with the data set, in this case the disk (DK). MAT60 is the file name of the data set and EXT is the file name extension; these are the fourth and fifth arguments. The sixth argument is a code which specifies how to open the file. In this case, 4 will open a previously created linked file for input via a READ. The final argument, ERROR, is a STARAN address where control is transferred if an error occurs during the OPEN process.

The READ command initiates the transfer of the data into a specified STARAN buffer. There is one required argument, DATA in this case, which references the RBUFF instruction which contains the parameter information required by a READ function. The RBUFF instruction occurs in the latter part of the program. Four arguments are required by RBUFF. LINK, the first argument, references the same linkword provided by the OPEN command. STORE is the address of the first location in bulk core memory for the STARAN input buffer; note that STORE was previously defined. The third argument, 14640, gives the maximum number of eight bit bytes of data to be input. The final argument selects the mode of transfer; in this case, 3 selects unformatted binary.

IOWAIT is an instruction to allow the user to determine when the input buffer is filled. The first argument is required and references the same linkword referenced previously; the second argument is optional and specifies an address to which to transfer control while the I/O process is still in progress. In this case, the control is transferred to BUSY. BUSY is a label in the latter part of the program. If the program goes to BUSY, interlock number twelve is set. The instruction following BUSY is a branch back to IOWAIT.

The close command is always related to a particular OPEN command. When the I/O is finished, the CLOSE command is executed, thus releasing the system for the next instruction. The final instruction of this section, SAVE, is one word of storage used for temporary storage of a counter as the program runs.

In the portion of the program which follows, the initialization of the program continues.

```
                B           $+11
CONST5          DC          X'10000000'
CONST4          DC          X'20000000'
CONST3          DC          0
CONST2          DC          2
COUNTER         DC          4
COUNT           DC          60
VALUE           DC          X'80000000'
CONST1          DC          3
CONST0          DC          1
CONST           DC          X'40000000'
```

The constants (DC) referenced by the main program and the subroutines are defined. Recall that the X preceding numbers indicates they are hexidecimal. The initial statement is a branch (B) around the constant definitions; the branch is necessary since these are non-executable statements. All of the constants above are fully explained in Figure 18.

The objective of this section is to generate masks used by the program during subsequent arithmetic routines and then to store these masks in the arrays for future reference.

```
CLR         X
CLR         Y
CLR         M
LI32        C,X'FFFFFFFF'
GEN,32      X'42008BA0'
GEN,32      X'42208840'
GEN,32      X'40009943'
L           M,Y
GEN,32      X'1AF50001'
GEN,32      X'40C08852'
CLR         M
L           M,Y
GEN,32      X'1AF60001'
GEN,32      X'40C08852'
CLR         M
L           M,Y
GEN,32      X'1AF70001'
GEN,32      X'40C08852'
CLR         M
L           M,Y
GEN,32      X'1AF80001'
```

Before using the response store registers, (X,Y,M) they are cleared (CLR). The LI32 command loads all thirty two bits of the common register (C) with the hexidecimal number FFFFFFFF. When translated to binary, this number is thirty-two contiguous ones. Following the loading of the C, there are three machine instructions which first load the contents of the C into bits 0 to 31 of the Y register; second, load the contents of the C into bits 32 to 63 of the X register; and finally logically OR X and Y leaving the result in Y. The net result is sixty-four ones in bits 0 to 63 of the Y register. Note, it is necessary to proceed in this way since loading a response register from the common register automatically clears the rest of the register.

The contents of Y are loaded (L) into M, the mask register. Since this mask is used many times in the program, it is stored in a bit slice of the array for future reference. The next machine instruction stores the contents of M into bit slice F5(245) of all four arrays.

Recall the Y register has ones in bits 0 to 63, the top quarter of the register. The next machine instruction shifts the contents of Y by 64 bits with the result that there are now ones in the bit positions 64 to 127. Next clear (CLR) M in preparation for loading (L) M with the contents of Y. The machine instruction following the loading of M stores the contents of the M register to bit slice F6(246) of all four arrays.

It is now necessary to generate the mask for the third quarter. This is accomplished by a machine instruction to again shift the contents of Y by 64 bits so that the ones now occur in bit positions 128 to 191. Again clear (CLR) M in preparation for loading (L) Y into M. The next machine instruction stores the contents of M into bit slice F7(247) of all four arrays.

The final section is again a repeat of the above. The Y register is first shifted by 64 bits so that the ones are in positions 192 to 255. Then, after clearing (CLR) M, load (L) the contents of Y into M and finally store M into bit slice F8(248) of all four arrays.

The net result of this section is that there are four masks for future reference which are stored in the arrays for easy access by the program.

In this portion of the program data are loaded into the arrays, specifically field A, which will contain the first column of matrix data. The column is loaded once and then replicated in field A, four times in each array.

```
        LI      FPE,61
        LI      DP,0
        LI      FP12,0
LOADA   NOP
        LR      C,STORE(DP),2
        SCW     A,A
        INCR    FP2
        DECR    FPE
        BNZ,FPE LOADA
        LI      FP12,0
        CLR     X
        CLR     Y
        LOOP,32 MOVEA
        GEN,32  X'63C0A0FD'
        GEN,32  X'42008840'
        GEN,32  X'63C1A0FD'
        GEN,32  X'42E088A0'
        GEN,32  X'40009943'
        GEN,32  X'400088A2'
        GEN,32  X'40C08852'
        GEN,32  X'40009943'
        GEN,32  X'400088A2'
        GEN,32  X'40808852'
        GEN,32  X'40009943'
MOVEA   GEN,32  X'1B600002'
        NOP
```

Initially, FPE, one of the field pointers, is loaded with 61 since 61 words of data are to be loaded. After each word is loaded, FPE will be decremented; when it reaches zero, the program will branch out of the loading operation. DP, FP1, and FP2 are initialized to zero. DP is a counter used to move progressively through the data buffer in STARAN memory. The field pointers initialize the array and word. The first word from the buffer STORE modified by DP, that is with DP added to the address, is loaded into the common register with the load register (LR) command. The value 2 automatically increments DP. The number in the common register is then stored in the array (SCW) from field A of the common register to field A of the array in a position selected via the field pointers, FP1 and FP2. After each word is loaded, FP2 is incremented resulting in a move to the next word in field A. FP1 is not incremented since load array 0 will continue to be loaded. The counter FPE is decremented (DECR); and, as long as it is not zero, the program will branch (BNZ) back to the beginning of the load operation (LOADA).

The next section of the program takes the data from field A and uses the response store registers to replicate it four times in each array. The loop to achieve replication is initialized by setting the field pointers to zero and clearing the response store registers, X and Y. The loop proceeds as follows.

1. Loop thirty-two times to address MOVEA. The loop is performed thirty-two times since each word is thirty-two bits long.
2. Load bit slice selected by FP2 of words 0 to 31, into the common register.
3. Load the common register into Y, bits 0 to 31.
4. Load bit slice selected by FP2 of words 32 to 63, into the common register.
5. Load the common into X, bits 32 to 63.
6. Logically OR X and Y leaving the result in Y. At this point one bit slice of field A is in Y, bits 0 to 63.
7. Store Y into X.
8. Shift Y by 64 bits; this results in the Y register containing the bit slice in position 64 to 127.
9. Logically OR X and Y leaving the result in Y. Y now contains the bit slice replicated twice in bits 0 to 31 and 32 to 63.
10. Store Y into X.
11. Shift Y by 128 bits. Y now contains the bit slice replicated twice in bit positions 128 to 191 and 192 to 255.
12. Logically OR X and Y and store the result in Y. This results in Y containing the bit slice replicated four times.
13. Write Y into all arrays in the bit slice pointed to by FP2; increment FP2.

The final statement is a no-operation used here to allow for automatic speed up execution.

In this section the loading operation is continued by loading arrays 0,1,2, and 3, in that order, until all the matrix data are in the arrays. This section is specifically concerned with loading fields B,D,E and F in arrays 0,1 and 2. Array 3 is loaded in the following section of code since the data configuration for array 3 differs from arrays 0,1 and 2.

```
            LI      FPE,61
            LI      FP2,0
            LI      FP3,3
LOAD        NOP
            LI      BL,4
            LR      C,STORE(DP),3
            SCW     A,B
            LR      C,STORE(DP),3
            SCW     A,D
            LR      C,STORE(DP),3
            SCW     A,E
            LR      C,STORE(DP),3
            SCW     A,F
            INCR    FP2
            DECR    FPE
            BNZ,FPE LOAD
            RPT,3
            INCR    FP2
            SR      DP,SAVE
            LR      (BL,DP),COUNTER
            DECR    DP
            SR      DP,COUNTER
            BZ,DP   NEXTARRAY
            LR      DP,SAVE
            LI      FPE,61
            B       LOAD
NEXTARRAY   LI      (BL,DP),4
            SR      (BL,DP),COUNTER
            LR      DP,SAVE
            DECR    FP3
            LI      FP2,0
            INCR    FP1
            LI      FPE,61
            BZ,FP3  ARRAY3
            B       LOAD
```

Again, initialize (LI) PFE with 61, the number of words to be loaded in each field. FP2, the word pointer, is initialized (LI) to zero. FP3 is loaded (LI) with 3. FP3 is used to keep track of duplicate array operations. In this case, arrays 0,1 and 2 are loaded in the same way; thus, FP3 is initialized by 3. Successively, load (LR) the common register with data from STORE (DP) and store (SCW) it in the array in fields B,D,E and F in the word position pointed to by FP2. Next, increment (INCR) FP2 and decrement (DECR) FPE. If FPE is not zero, branch (BNZ) back to the beginning of the loading operation LOAD. When FPE reaches zero, FP2 must be incremented by three to move to the next 64 word block. (Recall that 61 data words are loaded in each block). Since each array is to be loaded with four sections of columns, COUNTER is initially 4 to keep track of the number of sections that have loaded as follows. First the value in DP must be saved so that the program will be able to continue the loading operation at the correct position in the data buffer by storing the value in DP (SR) in location SAVE. Then the COUNTER is loaded (LR) into the two registers BL and DP. It is necessary to load into BL and DP combined since each is a 16 bit register and the number in COUNTER is a 32 bit number. However, since the number of significant digits of the number is small, it will be fully contained in the lower-order bits, that is, DP. Therefore, to decrement COUNTER it is only necessary to decrement (DECR) DP. The new value of COUNTER is then saved by storing (SR) DP in the memory location COUNTER. The value of COUNTER is tested next, which is still in DP. If the value is zero, branch (BZ) to the address NEXTARRAY to load the next array. If the COUNTER is not zero, continue in the same array by again

initializing (LI) FPE to 61 and recalling (LR) the value from SAVE to DP and branching (B) to LOAD. However, once COUNTER is zero, it is necessary to begin loading the next array. This means a branch to NEXTARRAY. At NEXTARRAY, COUNTER is again initialized (LI) to four and stored (SR) at location COUNTER. FP3, which is keeping track of the number of duplicate array operations, is decremented (DECR). If FP3 is not zero, the loading continues as previously described by initializing FP2 to zero, incrementing FP1 (the array pointer), loading (LI) FRE with 61 and branching to LOAD. However, if FP3 is zero, the program will branch (BZ) to ARRAY3, the loading operation for array 3.

In this section, the last array is loaded. Recall from Figure 5 the configuration of the data in array 3 and note the difference from arrays 0,1 and 2.

```
ARRAY3    SR          DP,SAVE
          LI          (BL,DP),2
          SR          (BL,DP),COUNTER
          LR          DP,SAVE
          LI          FP2,0
LOADA3    NOP
          LI          BL,3
          LR          C,STORE(DP),3
          SCW         A,B
          LR          C,STORE(DP),3
          SCW         A,D
          LR          C,STORE(DP),3
          SCW         A,E
          LR          C,STORE(DP),3
          SCW         A,F
          INCR        FP2
          DECR        FPE
          BNZ,FPE     LOADA3
          SR          DP,SAVE
          LR          (BL,DP),COUNTER
          DECR        DP
          SR          (BL,DP),COUNTER
          BZ,DP       LAST
          LR          DP,SAVE
          RPT,3
          INCR        FP2
          LI          FPE,61
          B           LOADA3
```

This section begins a pattern which is repeated sixteen times. First, branch and link the subroutines which perform the necessary arithmetic operations. These subroutines are explained in full detail in the following sections. Then, after the appropriate masking, the data are moved so that the next column to be used as the pivot is moved into field A. The counter which was initialized to the matrix dimension is checked and, if it is zero, indicating the inversion is complete, the program is exited; otherwise the program continues with column operations.

```
        BAL,R7      SUB1
        BAL,R7      SUB2
NEXT    LR          ASH,VALUE
        GEN,32      X'08F50005'
        MVF         A,G
        MVF         B,A
        MVF         G,B
        LR          (BL,DP),COUNT
        DECR        DP
        BZ,DP       OUT
        SR          (BL,DP),COUNT
        LI          ASH,X'F000'
        BAL,R7      SUBA
```

The branch and link command (BAL) is followed by one of the branch and link registers (R7). This instruction transfers control of the program to the subroutine SUB1 after storing the Execution Location Counter of the next instruction in the branch and link register. Briefly, SUB1 will divide field A to create the identity element and then perform the arithmetic operations between column A and columns B and D. Next branch and link as above to subroutine SUB2; SUB2 performs the arithmetic operations between field A and fields E and F. At this point, all required arithmetic operations with the column currently in field A have been completed and the program proceeds to move a new column into the pivot position. Since the new pivot column will change, the appropriate array must be enabled. Therefore, load (LR), the array select register

(ASH), with the number in the address VALUE. Initially VALUE has 80000000 which will enable array 0. After completing the sixteen repetitions of this portion of the code which follow, all of the columns in array 0 will have been used as the pivot column. The next step will be to move to array 1 which will be accomplished by changing the number in VALUE to 40000000 and repeating the program. After completing array 1, VALUE is changed to 20000000 thus enabling array 2 and again repeat the same code. Finally, VALUE will be changed to 10000000 to enable array 3 and the code repeated until completion of the inversion.

The machine instruction moves the mask stored in bit slice F5(245) to the M register. Note in the sections following that the program will load from bit slice F5, F6, F7 and F8, depending upon which section of the array it is desired to work with. Also, each of these masks is used four times since there are four columns in each quarter of the array to be used as the pivot. With the appropriate mask in place, the column in field A is stored in field G. Then, the next column to be used is moved into field A. (In this case, field B is moved to field A although in subsequent repetitions of this code the field moved into A will change). Finally, the column temporarily stored in field G is moved (MVF) into the place just vacated.

At this point it is desired to check to see if all columns have been used as a pivot. Therefore, load (LR) registers BL and DP with the counter COUNT, which was initialized to the matrix dimension of 60. Then, decrement COUNT; and, if it is zero, branch (BZ) to the end of the program (OUT). Otherwise, store (SR) the new value in the address COUNT. All of the arrays are enabled by loading (LI) the array select register (ASH) with F000. The program will then branch and link SUBA. SUBA, like SUBB, SUBC, and SUBD, takes the new data in field A and replicates

it four times. The arithmetic operations will now begin. The next sections describe this and continue repeating the above operations until the matrix is inverted.

This last portion of the program is needed to change the number stored in VALUE. Recall that VALUE is loaded into the array select register in the previous sections. Another storage word in address CONST is used to accomplish this.

```
        LR          (BL,DP),CONST
        SR          (BL,DP),VALUE
        LR          DP,CONST3
        INCR        DP
        SR          DP,CONST3
        LR          DP,CONST1
        DECR        DP
        SR          DP,CONST1
        BZ,DP       NEXT
        LR          (BL,DP),CONST4
        SR          (BL,DP),CONST
        LR          DP,CONST2
        DECR        DP
        SR          DP,CONST2
        BNZ,DP      NEXT
        LR          (BL,DP),CONST5
        SR          (BL,DP),CONST
        B           NEXT
OUT     NOP
        NOP
        WAIT
```

The first time through the arithmetic routines, VALUE is loaded with 80000000, the initial number, enabling array 0. CONST, which initially is 40000000, is then loaded (LR) into the registers BL and DP and stored (SR) under VALUE. Next, load (LR) register DP with CONST3 and then increment (INCR) DP and store (SR) the new number at CONST3. CONST3 is used by the subroutines SUB1 and SUB2 to be loaded in FP1, the array pointer. At this point, the operations on one array have been completed. CONST3 is therefore incremented so that the program will move to the next array.

CONST1, initially 3, is loaded (LR) into the DP register and then decremented (DECR). The purpose of this counter is to count the number of times VALUE must be changed. If VALUE has been changed three times, it will then be zero and the program will branch (BZ) to the arithmetic routines without changing VALUE. Otherwise registers BL and DP are loaded (LR) with CONST4, initally 20000000 and then stored (SR) under CONST. Next another counter, CONST2, is checked by loading (LR) it in DP, decrementing (DECR) DP and storing (SR) DP back at CONST2. This counter helps to keep track of the number to be stored in CONST. If it is not zero, branch (BNZ) to NEXT, the start of the arithmetic routines. Otherwise load (LR) registers BL and DP with CONST5 and store (SR) CONST5 under CONST. CONST5 has number 10000000. Then branch (B) to NEXT.

The last statement is a NOP at the label OUT. This is the address the program branches to upon completion of the inversion program; that is, when COUNT is zero indicating all columns have been used as the pivot.

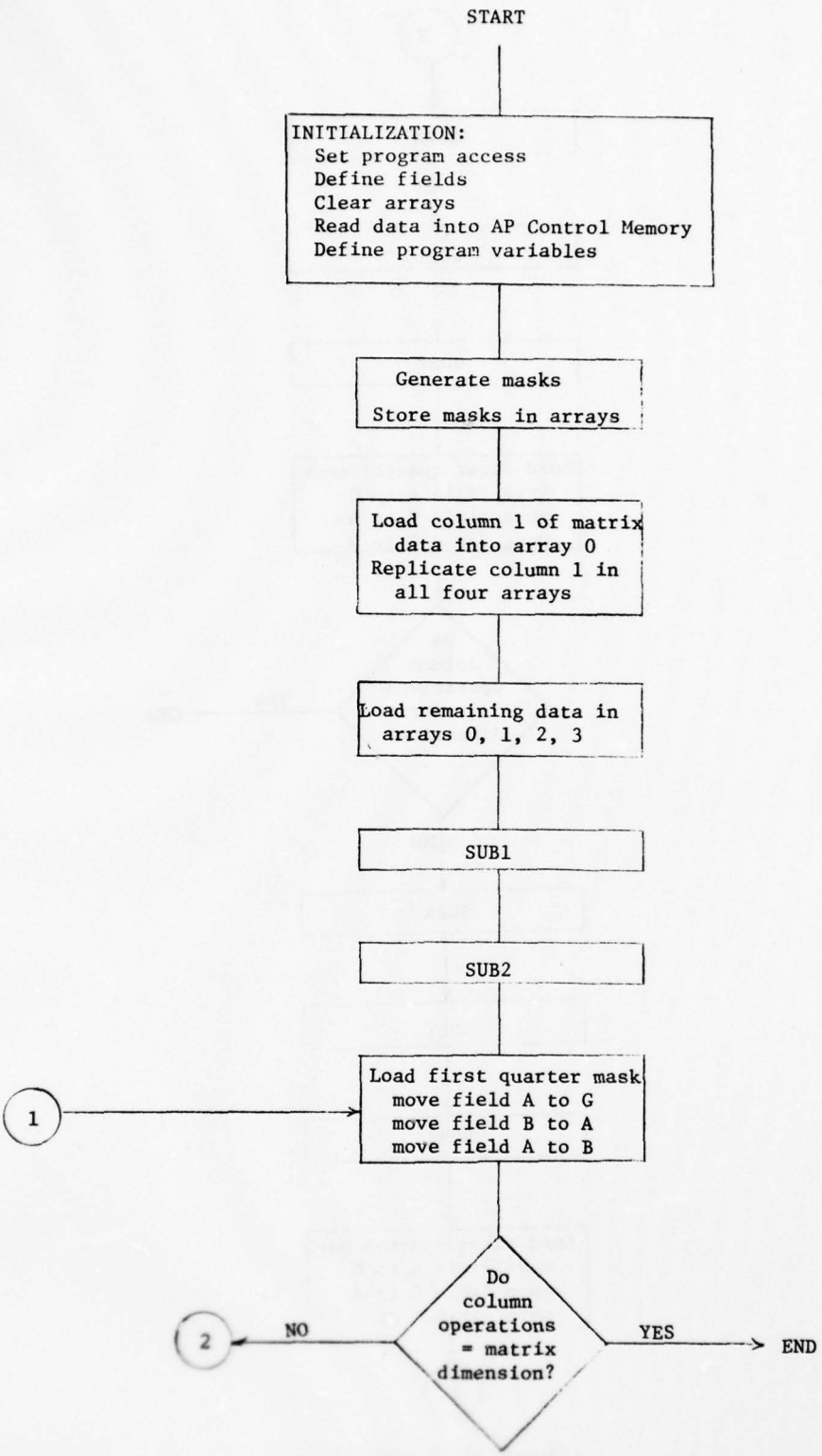


Figure 20: Flow Chart for MTX1.APL

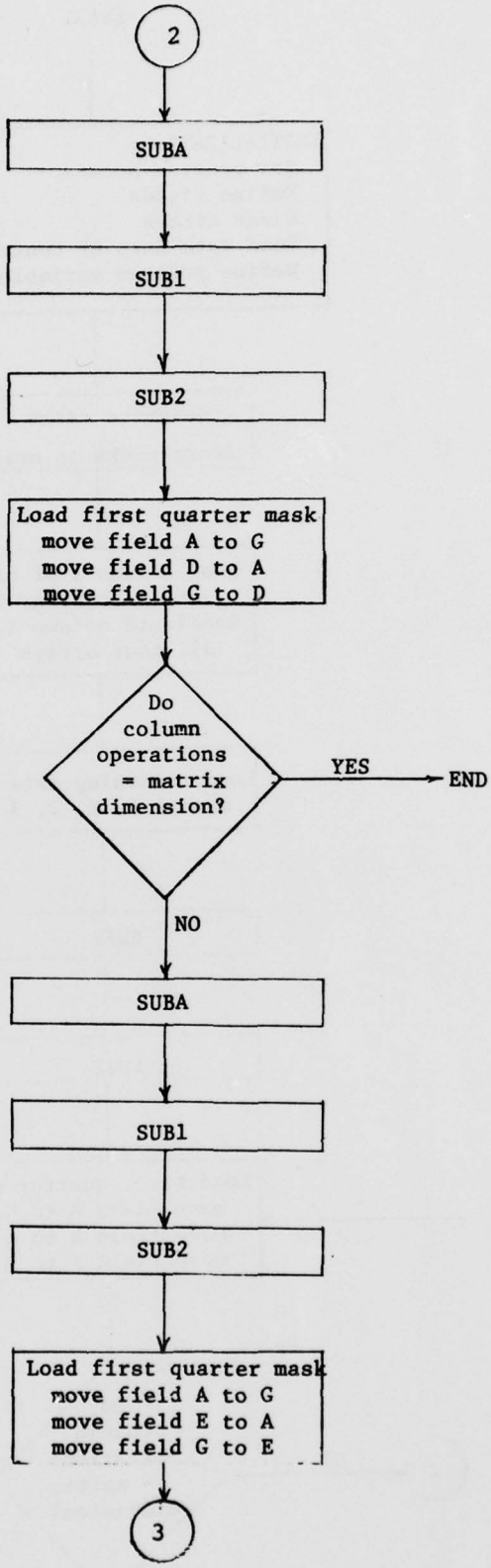


Figure 20: Continued

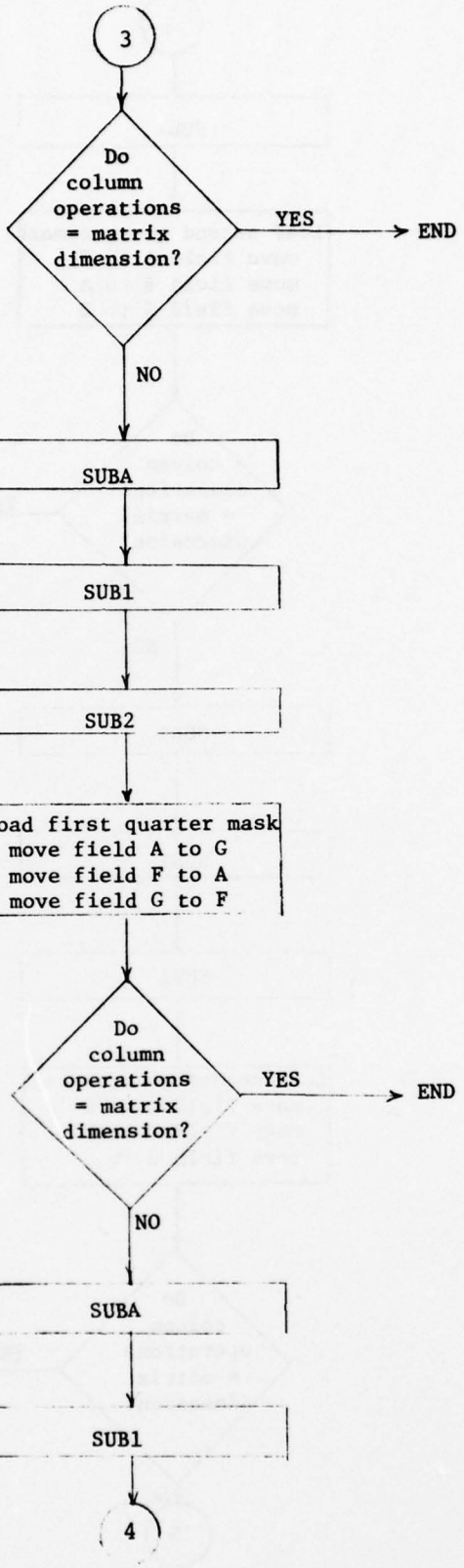


Figure 20: Continued

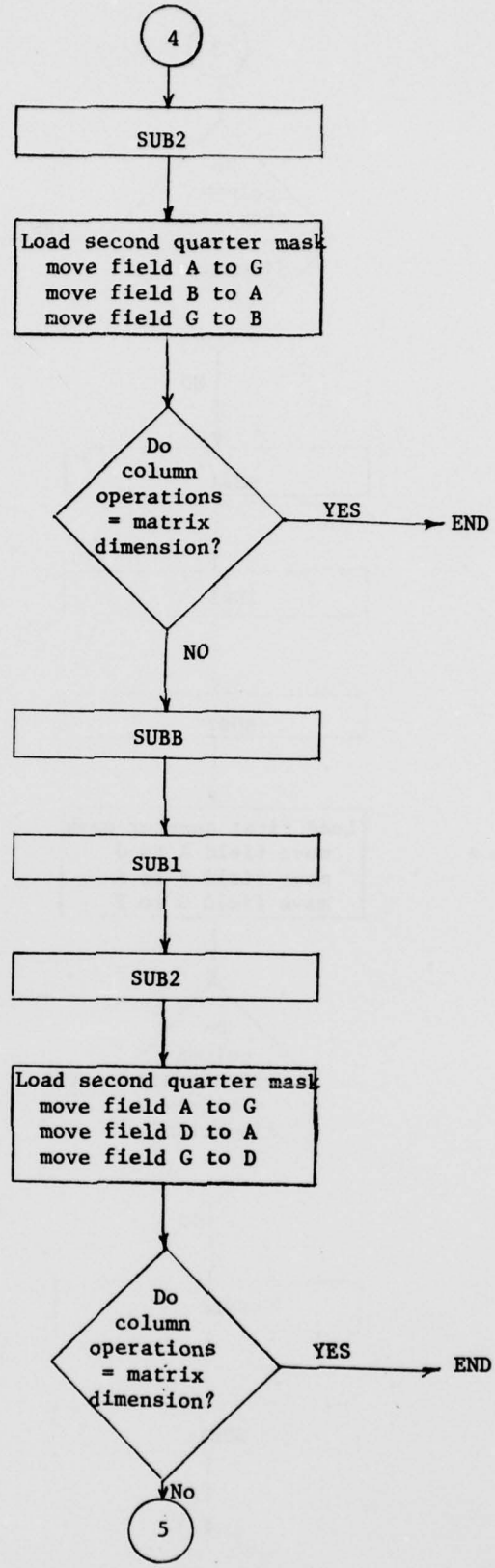


Figure 20: Continued

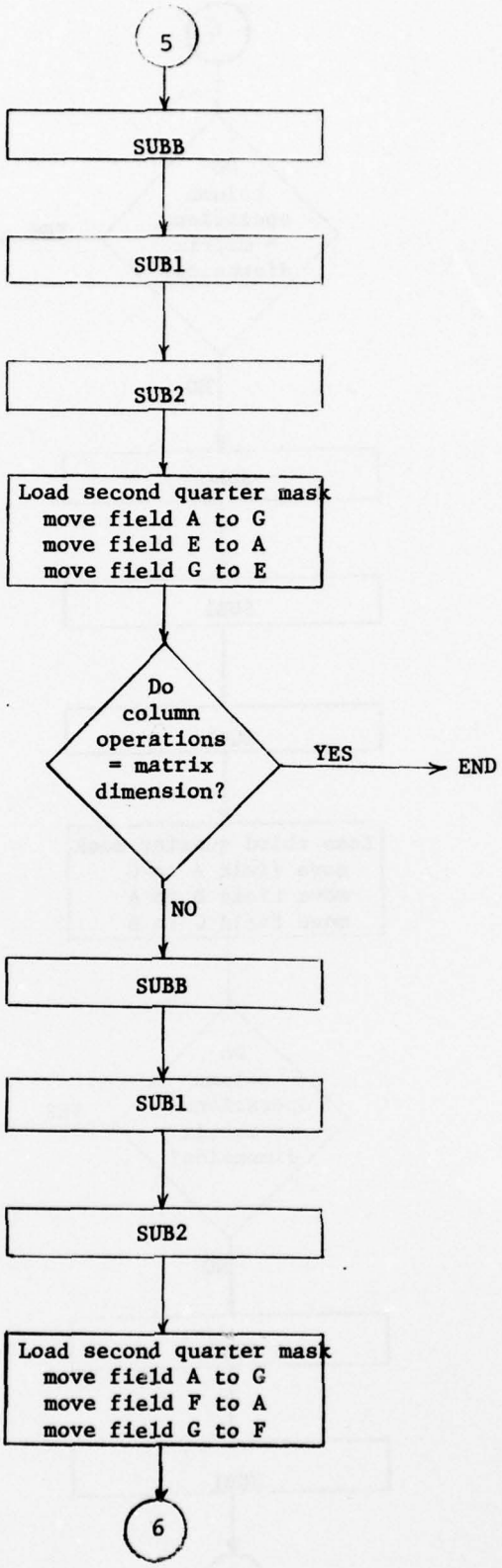


Figure 20: Continued

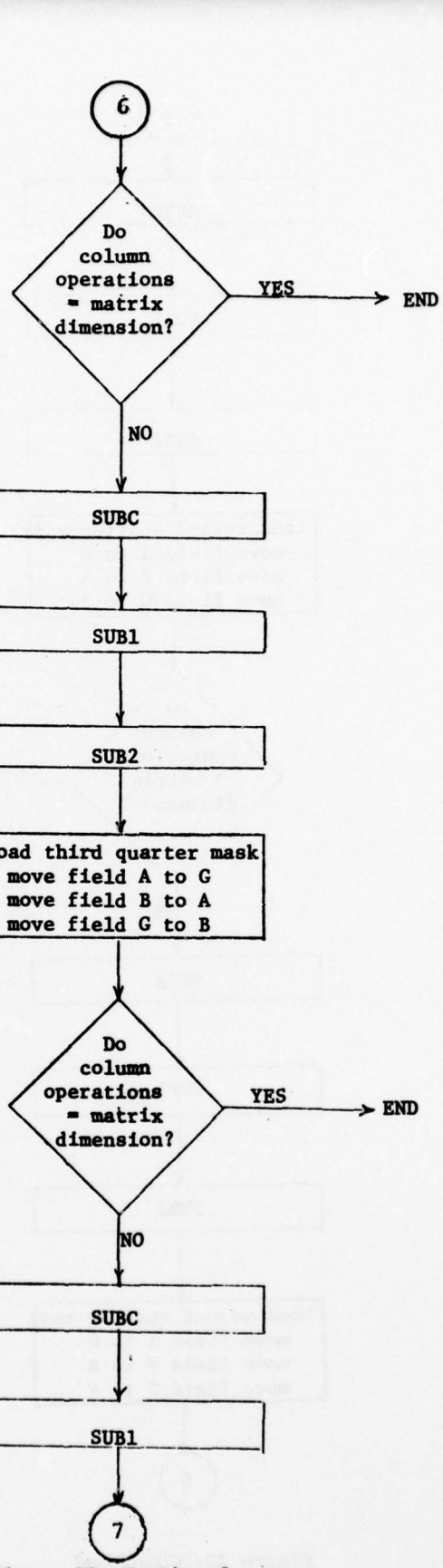


Figure 20: Continued

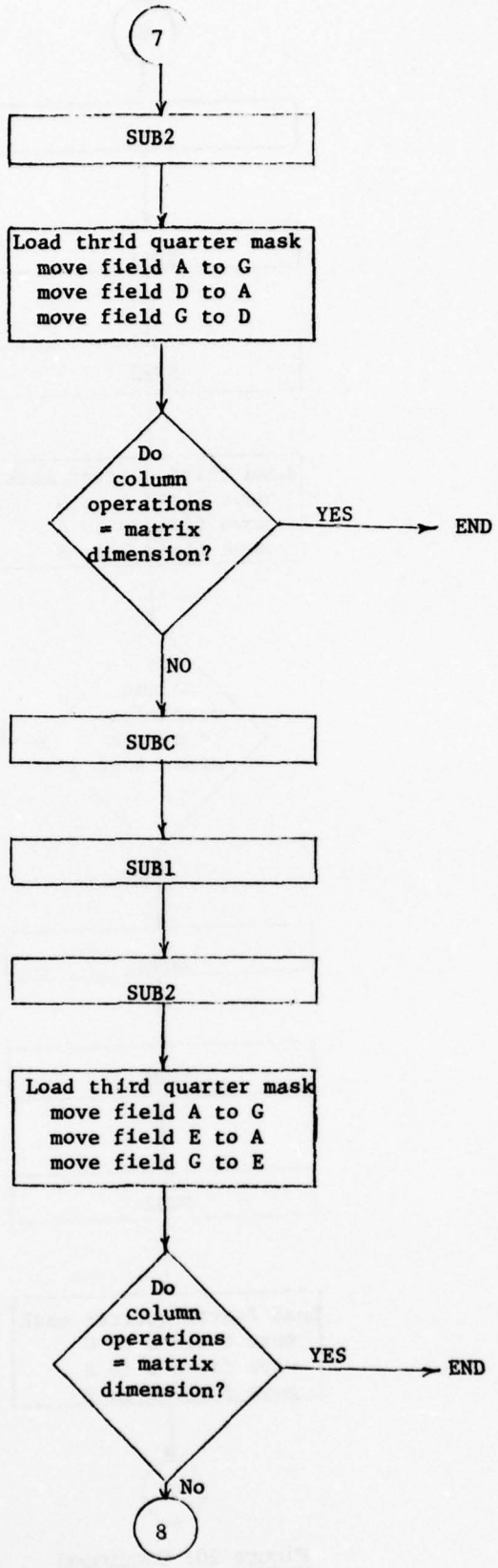


Figure 20: Continued

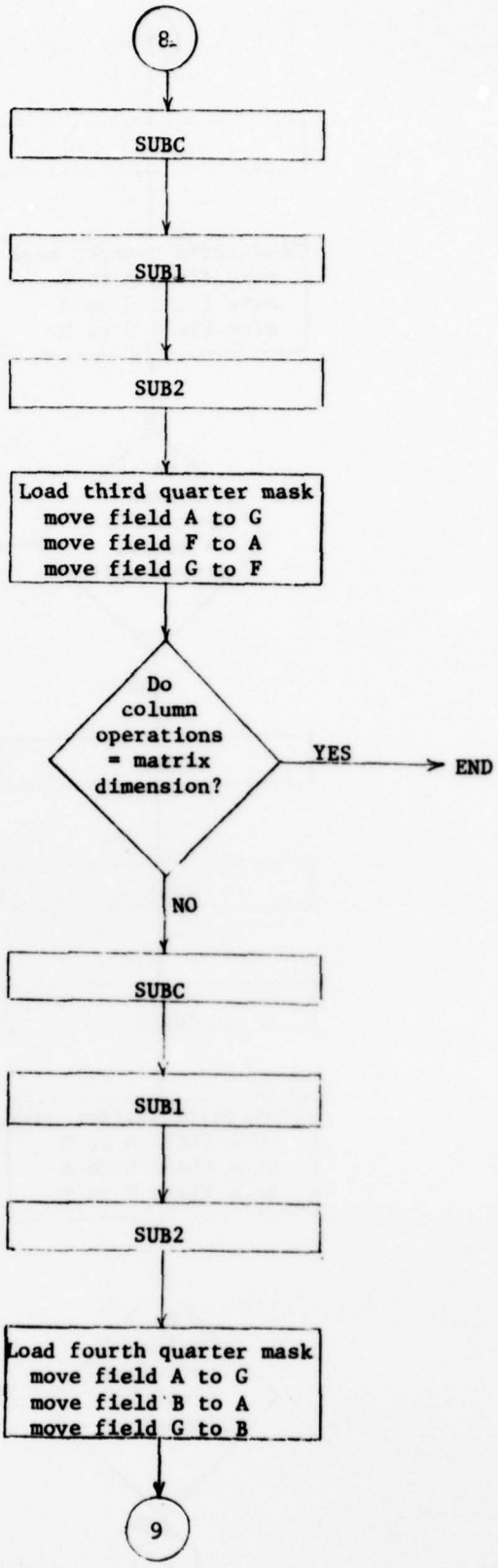


Figure 20: Continued

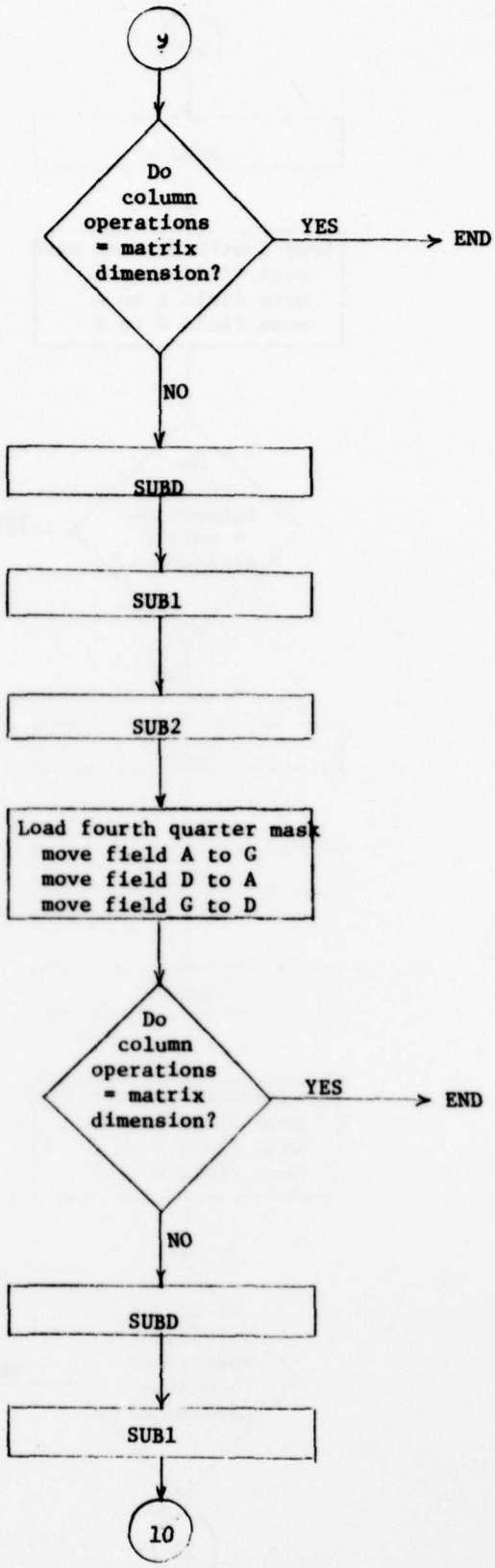


Figure 20: Continued

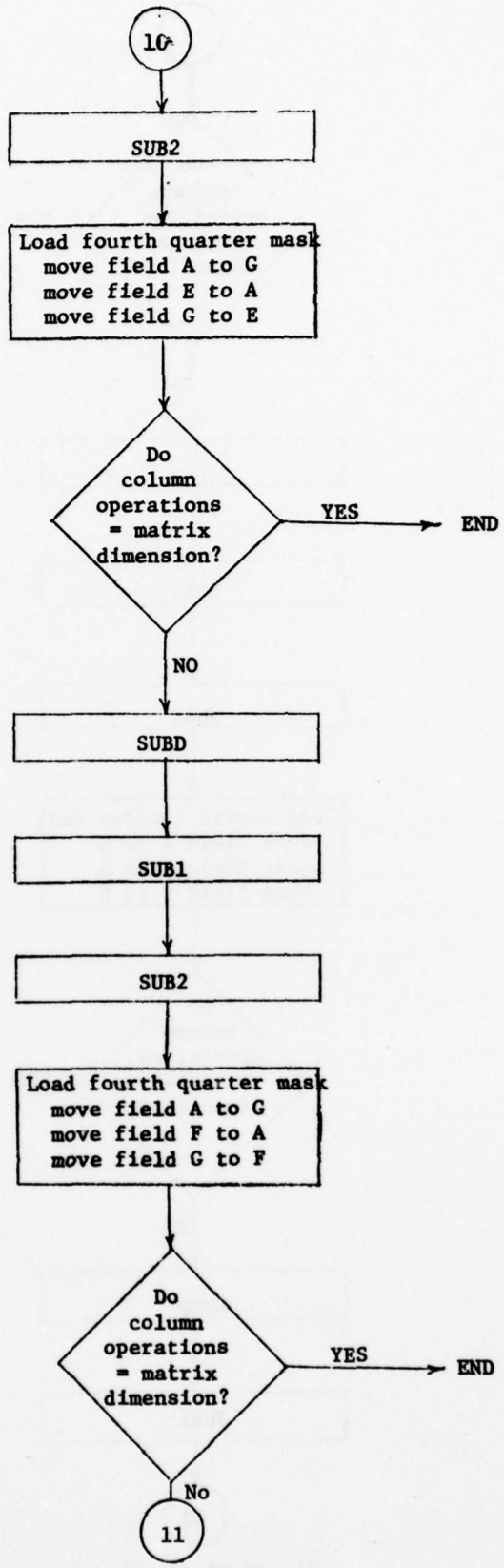


Figure 20: Continued

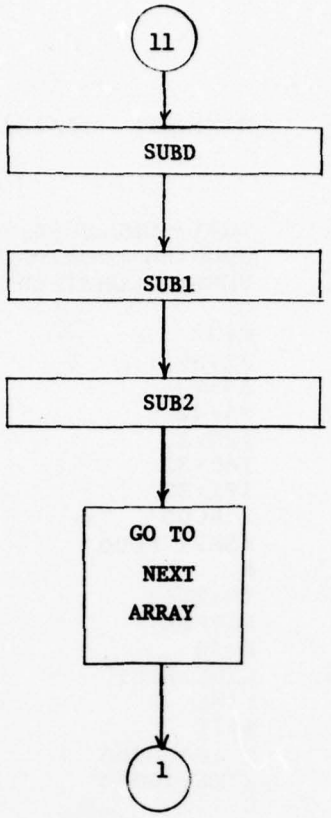


Figure 20: Continued

```
** MTX1.APL

                        MTX1.APL  10/13/76  1222-6 edt Wed


MAIN        START
            EXTRN       SUB1,SUB2,SUBA,SUBB,SUBC,SUBD
            ENTRY       COUNTER,FOUR,VALUE,CONST2
            ENTRY       CONST1,CONST,CONST0,CONST3
            ORG         0
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
STORE       EQU         X'B000'
            LI          ASH,X'F000'
            SET         M
            CLRF        (0,256)
            OPEN        BUFFER
            READ        DATA
WAIT1       IOWAIT      LINK,BUSY
            CLOSE       LINK
            B           $+11
CONST5      DC          X'10000000'
CONST4      DC          X'20000000'
CONST3      DC          0
CONST2      DC          2
COUNTER     DC          4
COUNT       DC          60
VALUE       DC          X'80000000'
CONST1      DC          3
CONST0      DC          1
CONST       DC          X'40000000'
            CLR         X
            CLR         Y
            CLR         M
            LI32        C,X'FFFFFFFF'
            GEN,32      X'420088A0'
            GEN,32      X'42208840'
            GEN,32      X'40009943'
            L           M,Y
            GEN,32      X'1AF50001'
            GEN,32      X'40C08852'
            CLR         M
            L           M,Y
            GEN,32      X'1AF60001'
            GEN,32      X'40C08852'
            CLR         M
            L           M,Y
            GEN,32      X'1AF70001'
            GEN,32      X'40C08852'
            CLR         M
            L           M,Y
            GEN,32      X'1AF80001'
```

Figure 21: MTX1.APL Listing

```
                LI          FPE,61
                LI          DP,0
                LI          FP12,0
LOADA           NOP
                LR          C,STORE(DP),2
                SCW         A,A
                INCR        FP2
                DECR        FPE
                BNZ,FPE     LOADA
                LI          FP12,0
                CLR         X
                CLR         Y
                LOOP,32     MOVEA
                GEN,32      X'63C0A0FD'
                GEN,32      X'42008840'
                GEN,32      X'63C1A0FD'
                GEN,32      X'42E088A0'
                GEN,32      X'40009943'
                GEN,32      X'400088A2'
                GEN,32      X'40C08852'
                GEN,32      X'40009943'
                GEN,32      X'400088A2'
                GEN,32      X'40808852'
                GEN,32      X'40009943'
MOVEA           GEN,32      X'1B600002'
                NOP
                LI          FPE,61
                LI          FP2,0
                LI          FP3,3
LOAD            NOP
                LI          BL,4
                LR          C,STORE(DP),3
                SCW         A,B
                LR          C,STORE(DP),3
                SCW         A,D
                LR          C,STORE(DP),3
                SCW         A,E
                LR          C,STORE(DP),3
                SCW         A,F
                INCR        FP2
                DECR        FPE
                BNZ,FPE     LOAD
                RPT,3
                INCR        FP2
                SR          DP,SAVE
                LR          (BL,DP),COUNTER
                DECR        DP
                SR          DP,COUNTER
                BZ,DP       NEXTARRAY
                LR          DP,SAVE
                LI          FPE,61
                B           LOAD
```

Figure 21: Continued

```
NEXTARRAY LI        (BL,DP),4
          SR        (BL,DP),COUNTER
          LR        DP,SAVE
          DECR      FP3
          LI        FP2,0
          INCR      FP1
          LI        FPE,61
          BZ,FP3    ARRAY3
          B         LOAD
ARRAY3    SR        DP,SAVE
          LI        (BL,DP),2
          SR        (BL,DP),COUNTER
          LR        DP,SAVE
          LI        FP2,0
LOADA3    NOP
          LI        BL,3
          LR        C,STORE(DP),3
          SCW       A,B
          LR        C,STORE(DP),3
          SCW       A,D
          LR        C,STORE(DP),3
          SCW       A,E
          LR        C,STORE(DP),3
          SCW       A,F
          INCR      FP2
          DECR      FPE
          BNZ,FPE   LOADA3
          SR        DP,SAVE
          LR        (BL,DP),COUNTER
          DECR      DP
          SR        (BL,DP),COUNTER
          BZ,DP     LAST
          LR        DP,SAVE
          RPT,3
          INCR      FP2
          LI        FPE,61
          B         LOADA3
LAST      LR        DP,SAVE
          RPT,3
          INCR      FP2
          LI        FPE,61
LASTA     LI        BL,3
          LR        C,STORE(DP),3
          SCW       A,B
          LR        C,STORE(DP),3
          SCW       A,D
          LR        C,STORE(DP),3
          SCW       A,E
          INCR      FP2
          DECR      FPE
          BNZ,FPE   LASTA
          NOP
```

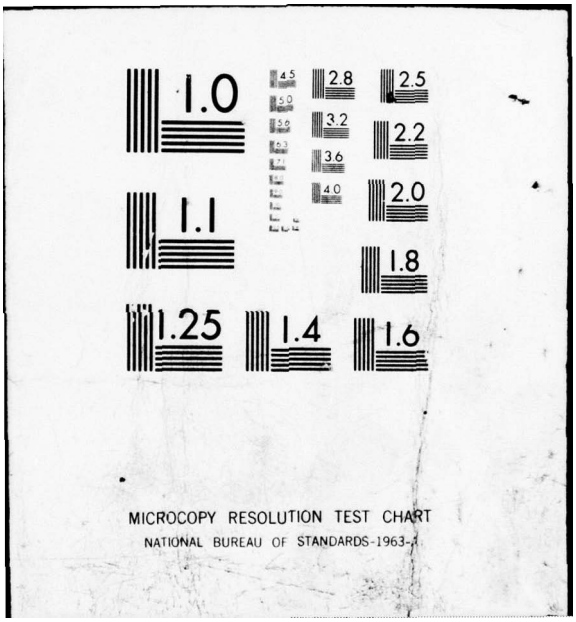
Figure 21: Continued

AD-A049 617 SYRACUSE UNIV N Y DEPT OF INDUSTRIAL ENGINEERING AND--ETC F/G 9/2
MATRIX INVERSION USING THE RADC STARAN ASSOCIATIVE ARRAY PROCES--ETC(U)
DEC 77 P B BERRA, E OLIVER F30602-75-C-0121

UNCLASSIFIED RADC-TR-77-386 NL

2 OF 2
AD
A049617

END
DATE
FILMED
3-78
DDC


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
        BAL,R7      SUB1
        BAL,R7      SUB2
NEXT    LR          ASH,VALUE
        GEN,32      X'08F50005'
        MVF         A,G
        MVF         B,A
        MVF         G,B
        LR          (BL,DP),COUNT
        DECR        DP
        BZ,DP       OUT
        SR          (BL,DP),COUNT
        LI          ASH,X'F000'
        BAL,R7      SUBA
        BAL,R7      SUB1
        BAL,R7      SUB2
        LR          ASH,VALUE
        GEN,32      X'08F50005'
        MVF         A,G
        MVF         D,A
        MVF         G,D
        LR          (BL,DP),COUNT
        DECR        DP
        BZ,DP       OUT
        SR          (BL,DP),COUNT
        LI          ASH,X'F000'
        BAL,R7      SUBA
        BAL,R7      SUB1
        BAL,R7      SUB2
        LR          ASH,VALUE
        GEN,32      X'08F50005'
        MVF         A,G
        MVF         E,A
        MVF         G,E
        LR          (BL,DP),COUNT
        DECR        DP
        BZ,DP       OUT
        SR          (BL,DP),COUNT
        LI          ASH,X'F000'
        BAL,R7      SUBA
        BAL,R7      SUB1
        BAL,R7      SUB2
        LR          ASH,VALUE
        GEN,32      X'08F50005'
        MVF         A,G
        MVF         F,A
        MVF         G,F
        LR          (BL,DP),COUNT
        DECR        DP
        BZ,DP       OUT
        SR          (BL,DP),COUNT
        LI          ASH,X'F000'
        BAL,R7      SUBA
        BAL,R7      SUB1
        BAL,R7      SUB2
        LR          ASH,VALUE
        GEN,32      X'08F60005'
        MVF         A,G
        MVF         B,A
        MVF         G,B
```

Figure 21: Continued

```
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBB
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F60005'
MVF         A,G
MVF         D,A
MVF         G,D
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBB
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F60005'
MVF         A,G
MVF         E,A
MVF         G,E
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBB
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F60005'
MVF         A,G
MVF         F,A
MVF         G,F
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBB
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F70005'
MVF         A,G
MVF         B,A
MVF         G,B
LR          (BL,DP),COUN
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUN
LI          ASH,X'F000'
BAL,R7      SUBC
BAL,R7      SUB1
BAL,R7      SUB2
```

Figure 21: Continued

```
LR          ASH,VALUE
GEN,32      X'08F70005'
MVF         A,G
MVF         D,A
MVF         G,D
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBC
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F70005'
MVF         A,G
MVF         E,A
MVF         G,E
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBC
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F70005'
MVF         A,G
MVF         F,A
MVF         G,F
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBC
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F80005'
MVF         A,G
MVF         B,A
MVF         G,B
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
LI          ASH,X'F000'
BAL,R7      SUBD
BAL,R7      SUB1
BAL,R7      SUB2
LR          ASH,VALUE
GEN,32      X'08F80005'
MVF         A,G
MVF         D,A
MVF         G,D
LR          (BL,DP),COUNT
DECR        DP
BZ,DP       OUT
SR          (BL,DP),COUNT
```

Figure 21: Continued

```
                LI          ASH,X'F000'
                BAL,R7      SUBD
                BAL,R7      SUB1
                BAL,R7      SUB2
                LR          ASH,VALUE
                GEN,32      X'08F80005'
                MVF         A,G
                MVF         E,A
                MVF         G,E
                LR          (BL,DP),COUNT
                DECR        DP
                BZ,DP       OUT
                SR          (BL,DP),COUNT
                LI          ASH,X'F000'
                BAL,R7      SUBD
                BAL,R7      SUB1
                BAL,R7      SUB2
                LR          ASH,VALUE
                GEN,32      X'08F80005'
                MVF         A,G
                MVF         F,A
                MVF         G,F
                LR          (BL,DP),COUNT
                DECR        DP
                BZ,DP       OUT
                SR          (BL,DP),COUNT
                LI          ASH,X'F000'
                BAL,R7      SUBD
                BAL,R7      SUB1
                BAL,R7      SUB2
                LR          (BL,DP),CONST
                SR          (BL,DP),VALUE
                LR          DP,CONST3
                INCR        DP
                SR          DP,CONST3
                LR          DP,CONST1
                DECR        DP
                SR          DP,CONST1
                BZ,DP       NEXT
                LR          (BL,DP),CONST4
                SR          (BL,DP),CONST
                LR          DP,CONST2
                DECR        DP
                SR          DP,CONST2
                BNZ,DP      NEXT
                LR          (BL,DP),CONST5
                SR          (BL,DP),CONST
                B           NEXT
OUT             NOP
                NOP
                WAIT
BUSY            ILOCK,1     12
                B           WAIT1
ERROR           WAIT
BUFFER          OBUFF       LINK,,DK,MAT60,EXT,4,ERROR
DATA            RBUFF       LINK,STORE,14640,3
SAVE            DS
LINK            DC,2        0
                END
                END
```

Figure 21: Continued

B3. The Subroutines.

1. MTX2.APL

The first section of the subroutine consists of the initialization conditions similar to the initialization of the main program.

```
SUB1        START
            ENTRY     SUB1
            EXTRN     COUNTER,FOUR,VALUE
            EXTRN     CONST1,CONST,CONST0
            ORG       0,R
A           DF        0,32
B           DF        32,32
D           DF        64,32
E           DF        96,32
F           DF        128,32
G           DF        160,32
H           DF        192,32
```

The first step in loading array 3 is to store (SR) the value in DP in the location SAVE. The registers BL and DP are loaded (LI) with two and that value is stored (SR) at location COUNTER. In preparation for the loading of array 3, DP is loaded (LR) with the value previously stored in SAVE and FP2 is initiallized (LI) with zero. Note that FPE was initiallized (LI) to 61 by the previous section of code. The loading operation is an exact replica of the previous operation except that the process is repeated only two times; when COUNTER has then decreased to zero the program branches to LASTA. LASTA is also a loading operation; however, in this case the program only loads data into fields B,D, and E. The loading procedure is the same as previously described.

```
LAST    LR      DP,SAVE
        RPT,3
        INCR    FP2
        LI      FPE,61
LASTA   LI      BL,3
        LR      C,STORE(DP),3
        SCW     A,B
        LR      C,STORE(DP),3
        SCW     A,D
        LR      C,STORE(DP),3
        SCW     A,E
        INCR    FP2
        DECR    FPE
        BNZ,FPE LASTA
        NOP
```

The START instruction is required and must precede any statements which generate APPLE code, the label associated with START will occur on the load map. The entry command gives the main program entry to this subroutine at the label SUB1. The program is flagged as being relocatable by the R following the 0 after the ORG statement. The relocation takes place automatically when the programs are linked. It is also necessary to define the fields again in the subroutines in the same way as in the main program.

This portion of the subroutine divides field A to create the identity element in the diagonal position of the matrix.

```
SUB1    LI      FP1,0
        SET     M
        LR      FP2,CONST0
        LC      A
        FDVC,240 A,G
        MVF     G,A
```

The label SUB1 is the entry point of the main program to this subroutine. This operation is indicated by loading (LI) field pointer one (FP1) with zero. Recall, when field pointer one is loaded with zero, the program is directed to array 0. The response register M is set (SET) in all four arrays to enable the division to take place in

each element of field A across all four arrays. Field pointer two (FP2) is loaded (LR) with CONST0; CONST0 is initiallized with the value one. The purpose of CONST0 is to give the word position of the diagonal element. Since the identity row is appended to the top of the matrix, the diagonal element will first occur in word one. This counter will be incremented as the program proceeds. The common register is loaded (LC) with the number from field A pointed to by FP1 and FP2. Division of field A by the common register is accomplished by the floating point routine FDVC. The bit slice (240) following the FDVC command must be provided to save the original contents of the M register. Two entries are required as arguments following FDVC; the first entry represents the dividend and the second the quotient. The quotient is then moved (MVF) from field G to field A.

The general idea behind the multiplication routine is to first perform the multiplication in field B on the top quarter of each array in succession; second, shift the mask by 64 bits and perform the multiplication in field B on the second quarter of each array; third, shift the mask again by 64 bits and multiply field B of each array in succession; and finally shift the mask the final time to operate on the bottom quarter of the array.

```
PROCESS   LI          ASH,X'F000'
          CLR         M
          GEN,32      X'08F50005'
          LI          BL,4
MULTB     LI          FP1,0
          LI          ASH,X'8000'
          LC          B
          FMPC,240    A,H
          INCR        FP1
          LI          ASH,X'4000'
          LC          B
          FMPC,240    A,H
          INCR        FP1
```

```
LI          ASH,X'2000'
LC          B
FMPC,240    A,H
INCR        FP1
LI          ASH,X'1000'
LC          B
FMPC,240    A,H
LI          ASH,X'F000'
DECR        BL
CLR         Y
L           Y,M
GEN,32      X'40C08852'
CLR         M
L           M,Y
NOP
BZ,BL       SUBTB
RPT,64
INCR        FP2
B           MULTB
```

First, all four arrays are enabled by loading (LI) the array select register (ASH) with F000. Then after clearing (CLR) M, the previously stored mask is loaded from bit slice F5(245) into M with a machine instruction. Recall this mask is on bits 0 to 63, the top quarter of the arrays. Then the multiplication is initialized by loading (LI) BL with four to be used as a counter. FP1 is loaded (LI) with zero and the array select register (ASH) with 8000 to indicate the intention of beginning with array 0. The common register is loaded (LC) with the word in field B pointed to by the field pointers. Recall, FP2 was previously loaded with CONST0. FMPC is the floating point arithmetic macro which multiplies all of masked field A by the common register, placing the result in field H. Bit slice 240 is used to store the original mask. FP1 is now incremented (INCR) to one; and the array select register (ASH) is set to 4000, thus enabling array 1. The loading of common register from field B is repeated, this time from array 1. The multiple macro FMPC multiplies field A in array 1 by the common register, placing the result in field

H. FP1 is incremented (INCR) to two, and the array select register is loaded with 2000 to enable array 2. Again (FMPC) is multiplied, this time in array 2. FP1 is incremented (INCR) to three, and the array select register is loaded (LI) with 1000 to enable array 3. After loading the common register (LC) with the word pointed to by FP1 and FP2 in array 3, (FMPC) is multiplied. At this point the top one quarter of each array has been multiplied. The array select register (ASH) is loaded (LI) to enable all four arrays. BL is decremented (DECR) to indicate one quarter of the operation is complete. After clearing (CLR) Y, the mask from M is loaded (L) into Y. A machine instruction shifts Y by 64 bits; then, after clearing (CLR) M, M is loaded (L) from Y. The program is now in a position to operate on the second quarter of the arrays. If the BL register is zero, branch to subtraction (SUBT); otherwise, repeat (RPT) sixty-four times the first instruction following RPT. In this case, increment (INCR) FP2 by sixty-four. This places the program at the correct word position for the columns in the second quarter of the arrays. Finally, branch (B) back to MULTB and continue as before. This code will be executed a total of four times; the first time is on the top quarter of the arrays (words 0 to 63), the second time is on words 64 to 127, the third time is on words 128 to 191; and the fourth time is on words 192 to 255. The final result is that field A will have been multiplied by the word pointed to by FP1 and FP2 in field B for all columns in field B. The result of this multiplication will be in field H.

In this part of the code, the subtraction operation is performed and FP2 is reset to begin in field D.

```
SUBTB     L          Y,M
          SET        M
          FSBF,240   B,H,G
          MVF        G,B
          RPT,192
          DECR       FP2
```

The subtraction routine begins by setting (SET) M to enable all words in each array. FSBF is the floating subtraction routine whose arguments represent the minuend (field B), the subtrahend (field H), and the difference (field G). Bit slice 240 is used to store the original contents of M. Field G is then moved (MVF) to field B; and finally FP2 is decremented by 192 in preparation for a repetition of the previous multiplication steps, this time in field D. This returns FP2 to the value it contained before the program started to multiply in field B.

The rest of the code in this subroutine repeats the above operations with the exception that the arithmetic routines are performed with elements from field D rather than field B.

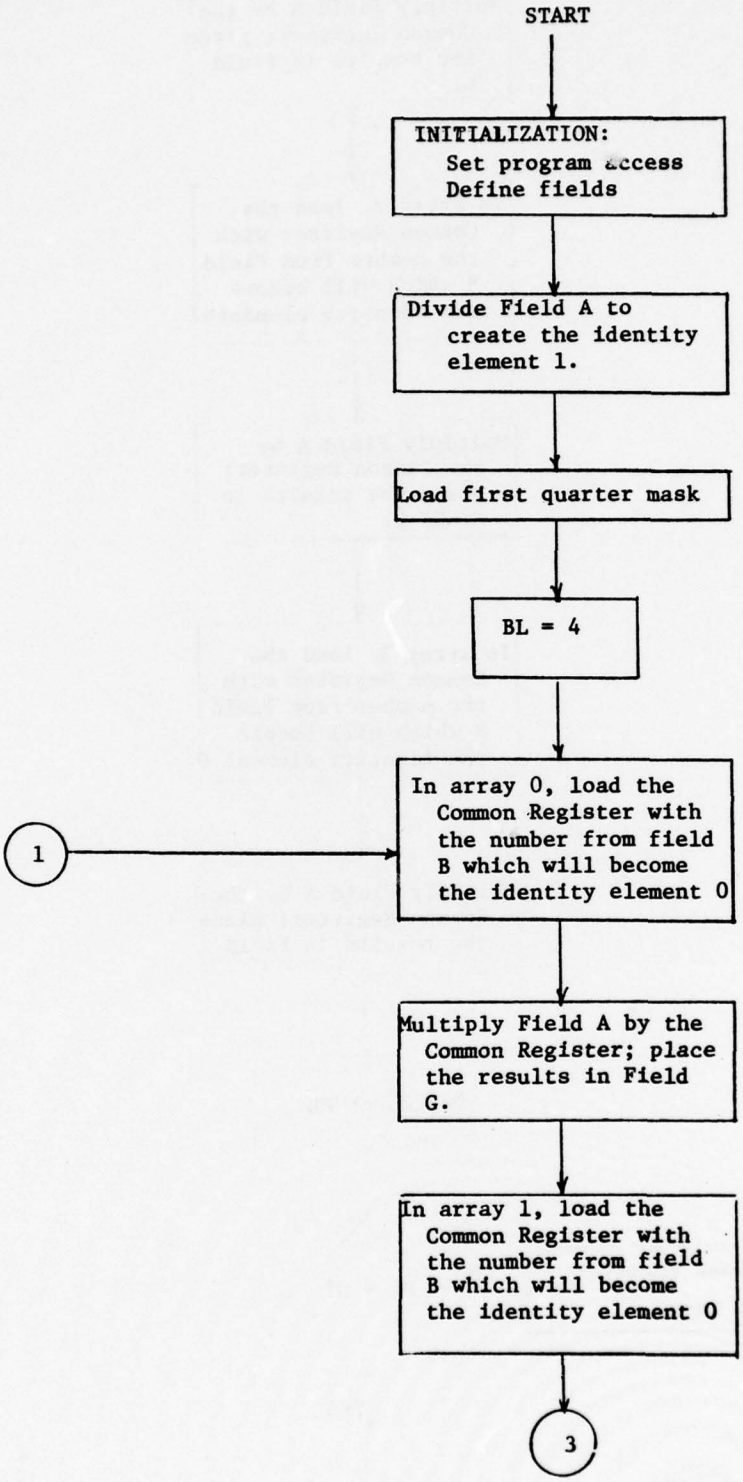
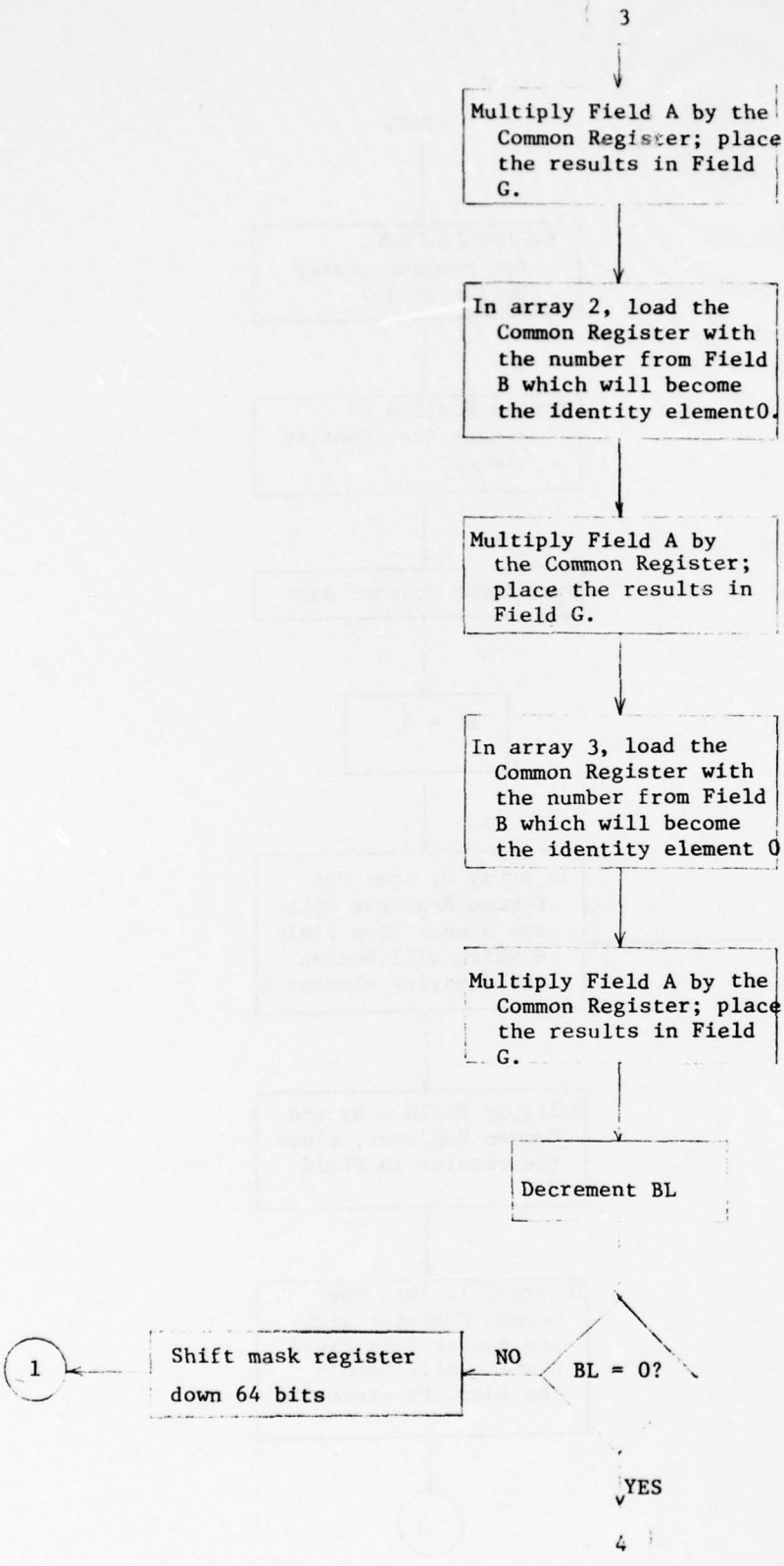


Figure 22: Flow Chart for MTX2.APL



22: Continued

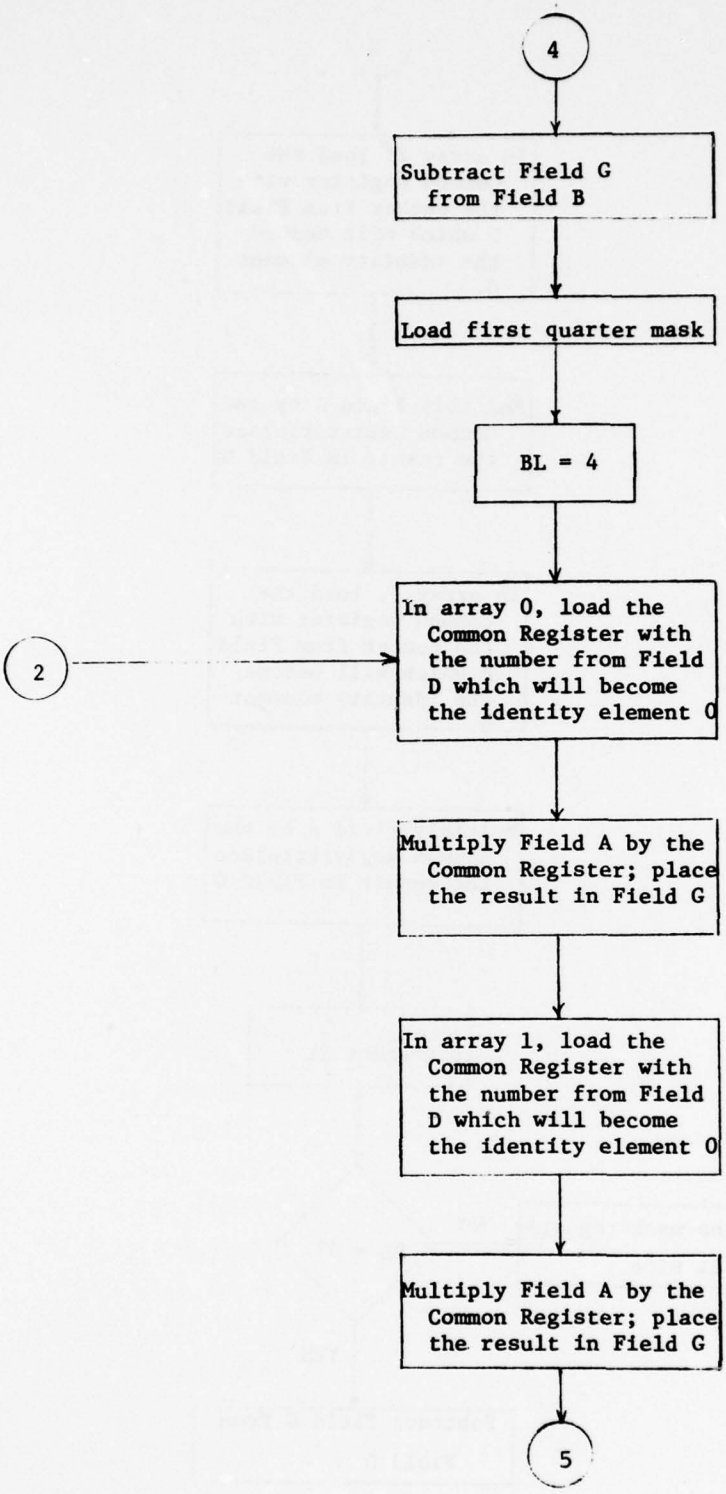


Figure 22: Continued

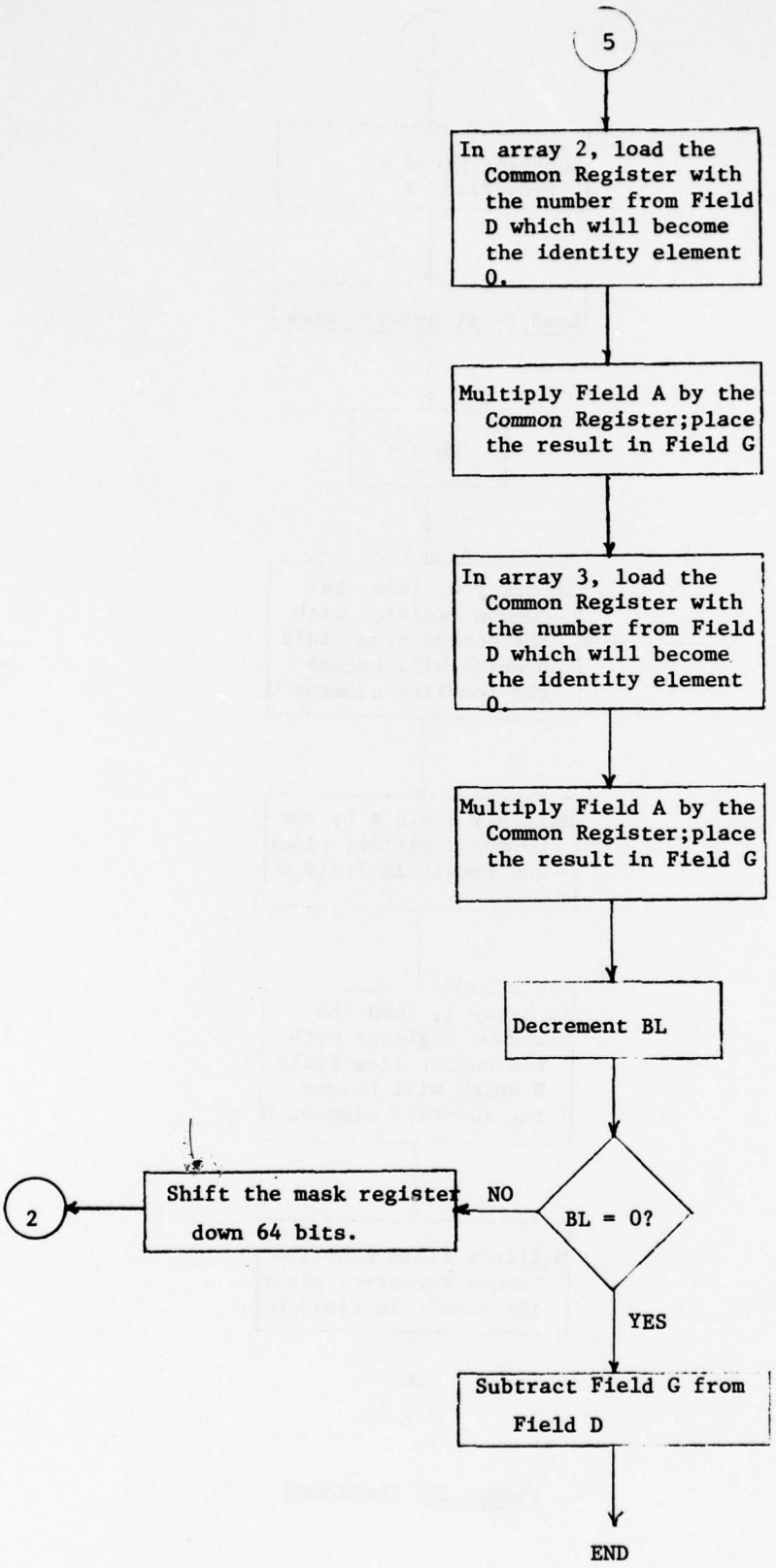


Figure 22: Continued

```
pr MTX2.APL

                                MTX2.APL  10/13/76  1230.3 edt Wed


SUBR1       START
            ENTRY       SUB1
            EXTRN       COUNTER,FOUR,VALUE
            EXTRN       CONST1,CONST,CONST0
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
SUB1        LI          FP1,0
            SET         M
            LR          FP2,CONST0
            LC          A
            FDVC,240    A,G
            MVF         G,A
PROCESS     LI          ASH,X'F000'
            CLR         M
            GEN,32      X'08F50005'
            LI          BL,4
MULTB       LI          FP1,0
            LI          ASH,X'8000'
            LC          B
            FMPC,240    A,H
            INCR        FP1
            LI          ASH,X'4000'
            LC          B
            FMPC,240    A,H
            INCR        FP1
            LI          ASH,X'2000'
            LC          B
            FMPC,240    A,H
            INCR        FP1
            LI          ASH,X'1000'
            LC          B
            FMPC,240    A,H
            LI          ASH,X'F000'
            DECR        BL
            CLR         Y
            L           Y,M
            GEN,32      X'40C08852'
            CLR         M
            L           M,Y
            NOP
            BZ,BL       SUBTB
            RPT,64
            INCR        FP2
            B           MULTB
```

Figure 23: MTX2.APL Listing

```
SUBTB    L          Y,M
         SET        M
         FSBF,240   B,H,G
         MVF        G,B
         RPT,192
         DECR       FP2
         LI         BL,4
         CLR        M
         GEN,32     X'08F50005'
MULTD    LI         FP1,0
         LI         ASH,X'8000'
         LC         D
         FMPC,240   A,H
         INCR       FP1
         LI         ASH,X'4000'
         LC         D
         FMPC,240   A,H
         INCR       FP1
         LI         ASH,X'2000'
         LC         D
         FMPC,240   A,H
         INCR       FP1
         LI         ASH,X'1000'
         LC         D
         FMPC,240   A,H
         LI         ASH,X'F000'
         DECR       BL
         CLR        Y
         L          Y,M
         GEN,32     X'40C08852'
         CLR        M
         L          M,Y
         NOP
         BZ,BL      SUBTD
         RPT,64
         INCR       FP2
         B          MULTD
SUBTD    L          Y,M
         SET        M
         FSBF,240   D,H,G
         MVF        G,D
         RPT,192
         DECR       FP2
         B          0(R7)
         END
         END

r 1232 0.474 1.168 65 level 2, 9
```

Figure 23: Continued

2. MTX3.APL

The second subroutine is almost identical to the first subroutine. Its purpose is to perform the arithmetic operations on fields E and F. Since the identity element in field A was previously created in MTX2.APL, that part of the program is not repeated. However, the multiplication and subtraction portions duplicate MTX2.APL except for the fields they operate on. There is one change, however, that should be noted. At the end of the subtraction in field F, FP2 is decremented by 191 rather than 192. This has the effect of increasing CONST0 so that the next time the subroutine is called, the identity element is created in the next sequential position.

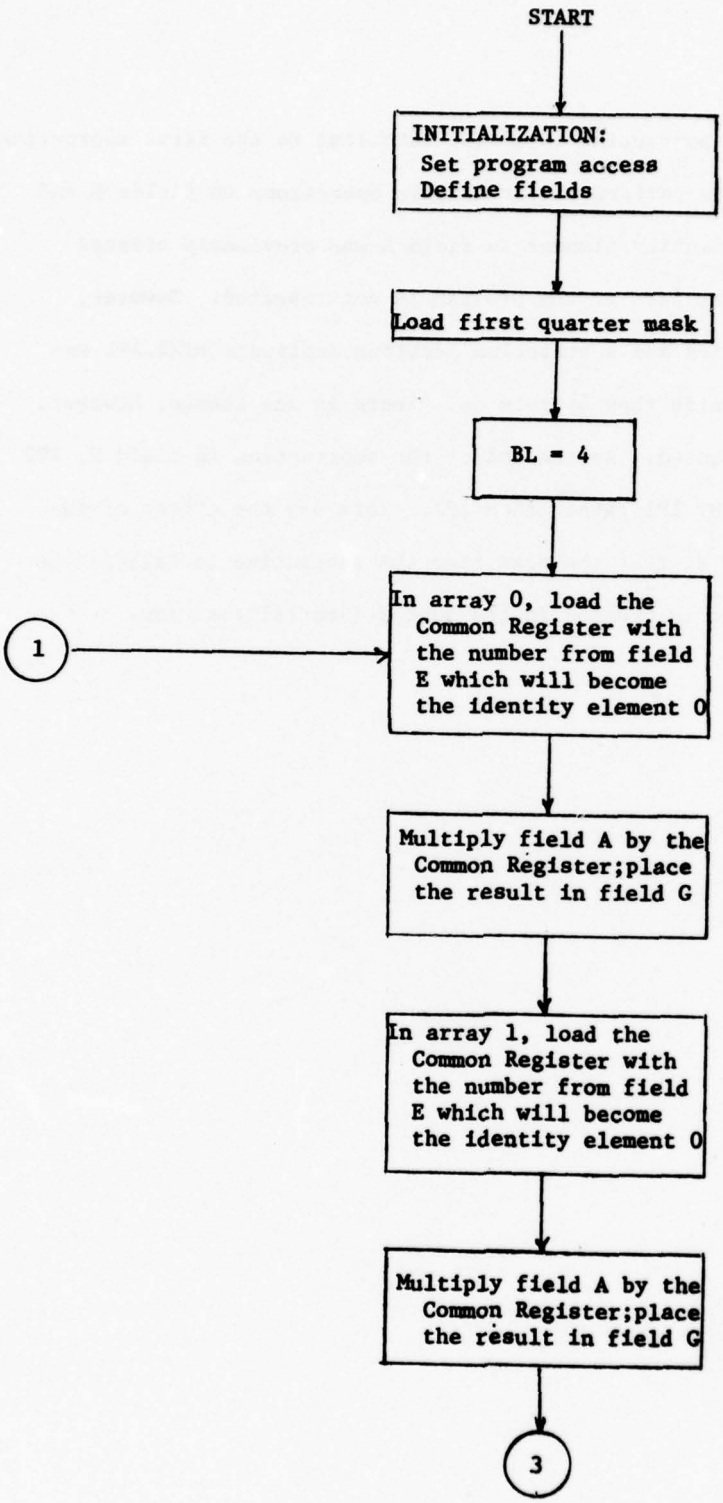


Figure 24: Flow Chart for MTX3.APL

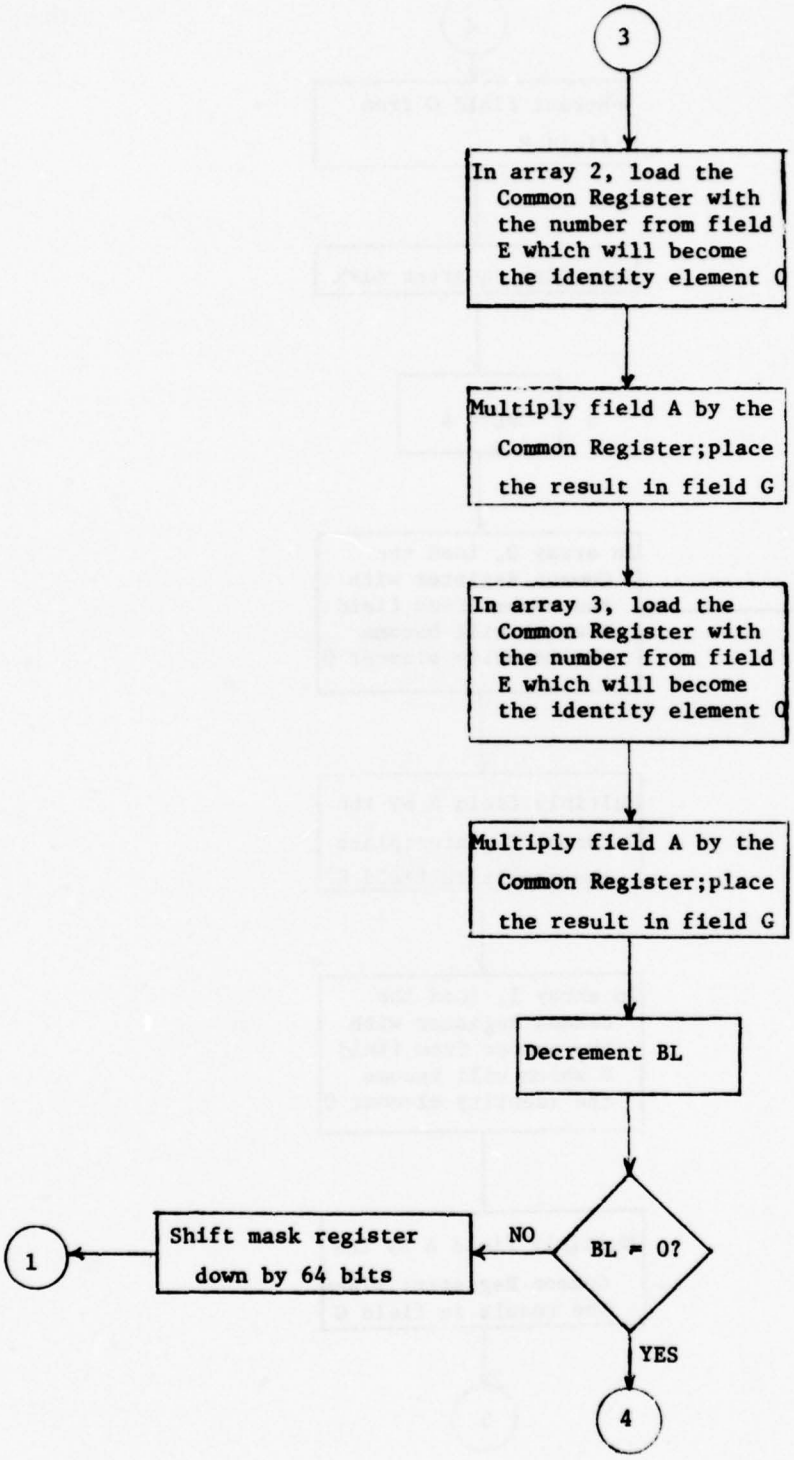


Figure 24: Continued

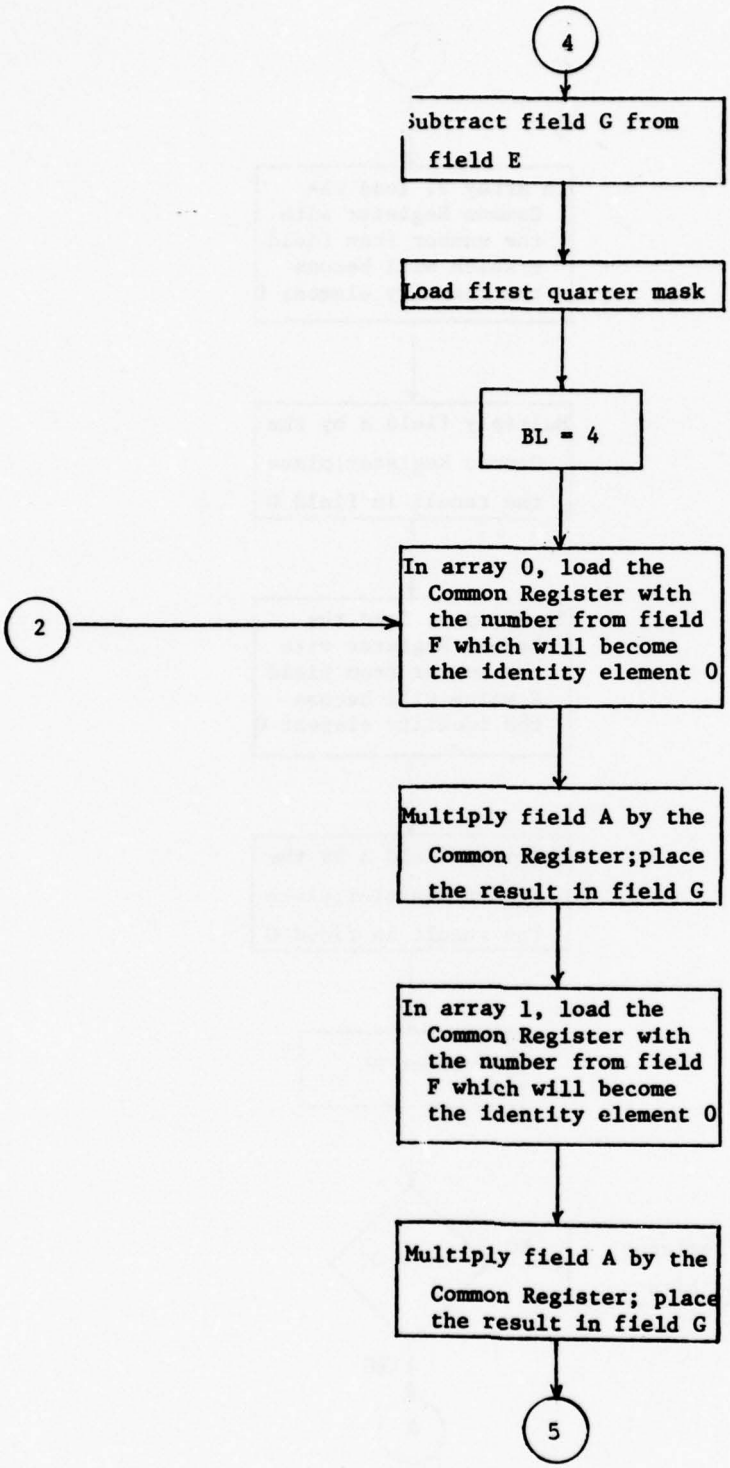


Figure 24: Continued

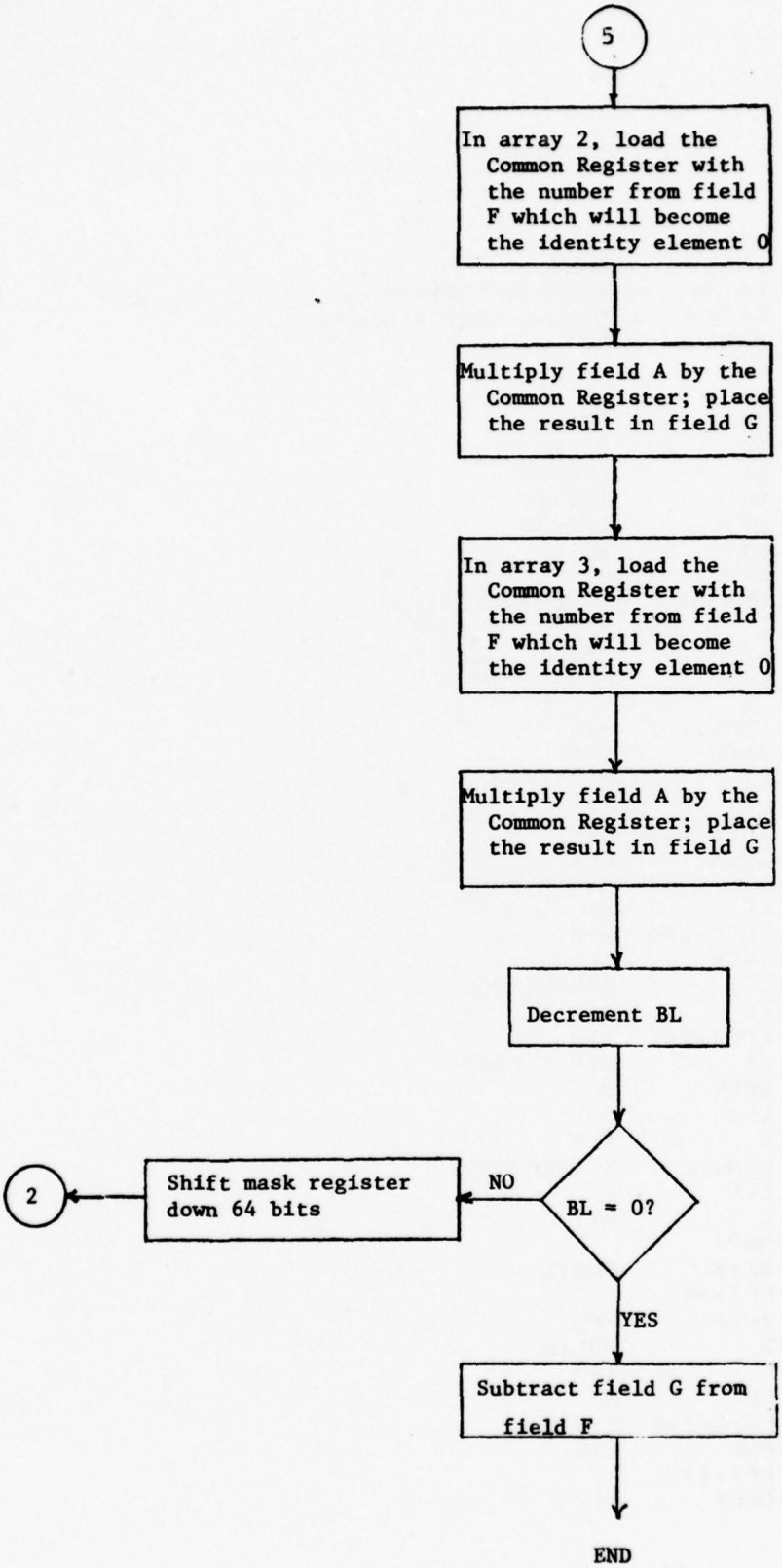


Figure 24: Continued

```
pr MTX3.APL

                        MTX3.APL  10/13/76  1232.7 edt Wed

SUBR2       START
            ENTRY       SUB2
            EXTRN       COUNTER,FOUR,VALUE
            EXTRN       CONST1,CONST,CONST0
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
SUB2        LI          BL,4
            CLR         M
            GEN,32      X'08F50005'
            L           M,Y
MULTE       LI          FP1,0
            LI          ASH,X'8000'
            LC          E
            FMPC,240    A,H
            INCR        FP1
            LI          ASH,X'4000'
            LC          E
            FMPC,240    A,H
            INCR        FP1
            LI          ASH,X'2000'
            LC          E
            FMPC,240    A,H
            INCR        FP1
            LI          ASH,X'1000'
            LC          E
            FMPC,240    A,H
            LI          ASH,X'F000'
            DECR        BL
            CLR         Y
            L           Y,M
            GEN,32      X'40C08852'
            CLR         M
            L           M,Y
            NOP
            BZ,BL       SUBTE
            RPT,64
            INCR        FP2
            B           MULTE
SUBTE       L           Y,M
            SET         M
            FSBF,240    E,H,G
            MVF         G,E
            RPT,192
            DECR        FP2
```

Figure 25: MTX3.APL Listing

```
                LI          BL,4
                CLR         M
                GEN,32      X'08F50005'
                L           M,Y
MULTF           LI          FP1,0
                LI          ASH,X'8000'
                LC          F
                FMPC,240    A,H
                INCR        FP1
                LI          ASH,X'4000'
                LC          F
                FMPC,240    A,H
                INCR        FP1
                LI          ASH,X'2000'
                LC          F
                FMPC,240    A,H
                INCR        FP1
                LI          ASH,X'1000'
                LC          F
                FMPC,240    A,H
                LI          ASH,X'F000'
                DECR        BL
                CLR         Y
                L           Y,M
                GEN,32      X'40C08852'
                CLR         M
                L           M,Y
                NOP
                BZ,BL       SUBTF
                RPT,64
                INCR        FP2
                B           MULTF
SUBTF           L           Y,M
                SET         M
                FSBF,240    F,H,G
                MVF         G,F
                RPT,191
                DECR        FP2
                SR          FP2,CONST0
                B           0(R7)
                END
                END

r 1233 0.446 1.418 62 level 2, 9
```

Figure 25: Continued

3. **SUBA.APL**

This program takes data from the first quarter of any array and replicates it sixteen times in field A. In addition, the routine loads the identity row into the matrix in the new position.

```
SUB3        START
            ENTRY       SUBA
            EXTRN       COUNTER,FOUR,VALUE,CONST3
            EXTRN       CONST1,CONST,CONST0,CONST2
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
```

The initial portion of this program has the same purpose as the initial portion of the previous subroutine. In this case, the entry point to the subroutine is at the label SUBA.

In this portion of the subroutine the data are replicated in field A.

```
SUBA        LR          FF1,CONST3
            LI          FF2,0
            CLR         X
            CLR         Y
            LOOP,32     MOVEB
            GEN,32      X'63C0A0FD'
            GEN,32      X'42008840'
            GEN,32      X'63C1A0FD'
            GEN,32      X'42E088A0'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40C08852'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40808852'
            GEN,32      X'40009943'
MOVEB       GEN,32      X'1B600002'
            NOP
```

Except for initializing FP1 with CONST3, this section of the subroutine is exactly the same as the section of the main program which replicates the data. CONST3 is initially 0, indicating the intention of working in array 0. It is incremented each time the main program has been executed, thus moving the subroutine from array to array.

Here the new identity row is appended to the matrix in preparation for using a new pivot column.

```
            LI          FP3,4
            LI          FP1,0
IDENTA      LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ONE         LI32        C,X'01400000
            SCW         A,A
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ONE
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENTA
```

This operation is intiallized by loading (LI) FP3 with 4 to count the arrays; FP1 with 0 to initiate in array 0 and FP2 with CONST0, which when decremented indicates the word position of the new identity row. Recall that CONST0 was used by the previous arithmetic subroutines to indicate the positon of the number to be used for division and multiplication. FPE is loaded with 4 since the column in field A is replicated four times in each array; the common register is loaded (LI32) with one (0140000 in hex) and the value in the common register is stored (SCW) into field A in the position pointed to by the field pointers FP1 and FP2. FP2 is then incremented 64 times (RPT, 64) to place the program in the second quarter; FPE is

decremented. FPE is now tested and if it is not zero, the program branches (BNZ) to ONE and repeated storing one in field A. Once FPE reaches zero it indicates that an array has been completed. The program increments (INCR) FP1 to point to the next array and decrements (DECR) FP3. FP3 is the counter which counts the number of arrays. If FP3 is not zero, branch (BNZ) to the beginning (IDENTA) of the operation and repeat for the next array; otherwise continue.

In this section, the zero element of the identity row is stored in all columns.

```
            LI          FP3,4
            LI          FP1,0
IDENT       LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ZERO        LI32        C,X'80000000'
            SCW         A,B
            SCW         A,D
            SCW         A,E
            SCW         A,F
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ZERO
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENT
            B           0(R7)
```

This final portion operates in the same way as the previous portion except that a zero (80000000 in hex) is loaded into the common register and that zero is stored into fields B,D,E, and F of each array. The procedure is identical to the previous section except that the program must store four times at each step.

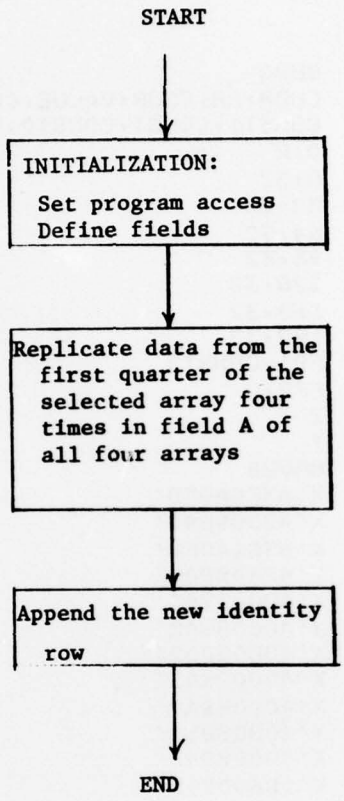


Figure 26: Flow Chart for SUBA.APL

```
SUB3        START
            ENTRY       SUBA
            EXTRN       COUNTER,FOUR,VALUE,CONST3
            EXTRN       CONST1,CONST,CONST0,CONST2
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
SUBA        LR          FP1,CONST3
            LI          FP2,0
            CLR         X
            CLR         Y
            LOOP,32     MOVEB
            GEN,32      X'63C0A0FD'
            GEN,32      X'42008840'
            GEN,32      X'63C1A0FD'
            GEN,32      X'42E088A0'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40C08852'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40808852'
            GEN,32      X'40009943'
MOVEB       GEN,32      X'1B600002'
            NOP
            LI          FP3,4
            LI          FP1,0
IDENTA      LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ONE         LI32        C,X'01400000'
            SCW         A,A
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ONE
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENTA
```

Figure 27: SUBA.APL Listing

```
            LI          FP3,4
            LI          FP1,0.
IDENT       LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ZERO        LI32        C,X'80000000'
            SCW         A,B
            SCW         A,D
            SCW         A,E
            SCW         A,F
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ZERO
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENT
            B           0(R7)
            END
            END
```

```
r 1125 0.332 1.176 56
```

Figure 27: Continued

4. SUBB.APL, SUBC.APL, and SUBD.APL

These subroutines are identical to SUBA.APL except that they use bit slices from successive quarters of field A for replication. Specifically, SUBA.APL loads the common register from words 0 to 63, SUBB.APL uses word 64 to 127, SUBC.APL uses word 128 to 191 and SUBD.APL uses words 192 to 255.

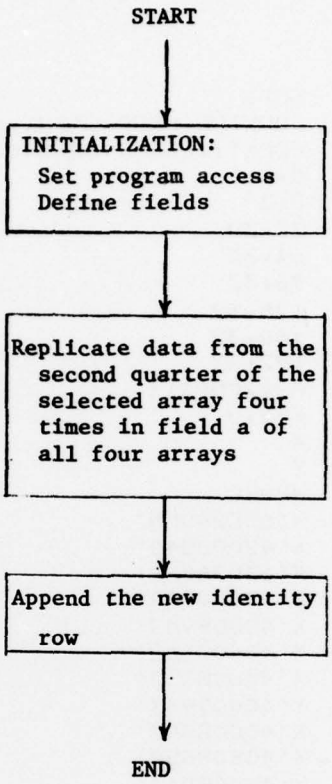


Figure 28: Flow Chart for SUBB.APL

```
pr SUBB.APL

                        SUBB.APL  10/14/76  1125.6 edt Thu


SUB4        START
            ENTRY       SUBB
            EXTRN       COUNTER,FOUR,VALUE,CONST3
            EXTRN       CONST1,CONST,CONST0,CONST2
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
SUBB        LR          FP1,CONST3
            LI          FP2,0
            CLR         X
            CLR         Y
            LOOP,32     MOVEC
            GEN,32      X'63C2A0FD'
            GEN,32      X'42008840'
            GEN,32      X'63C3A0FD'
            GEN,32      X'42E088A0'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40C08852'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40808852'
            GEN,32      X'40009943'
MOVEC       GEN,32      X'1B600002'
            NOP
            LI          FP3,4
            LI          FP1,0
IDENTA      LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ONE         LI32        C,X'01400000'
            SCW         A,A
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ONE
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENTA
```

Figure 29: SUBB.APL Listing

```
            LI          FP3,4
            LI          FP1,0
IDENT       LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ZERO        LI32        C,X'80000000'
            SCW         A,B
            SCW         A,D
            SCW         A,E
            SCW         A,F
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ZERO
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENT
            B           0(R7)
            END
            END

  1126 0.350 1.080 54
```

Figure 29: Continued

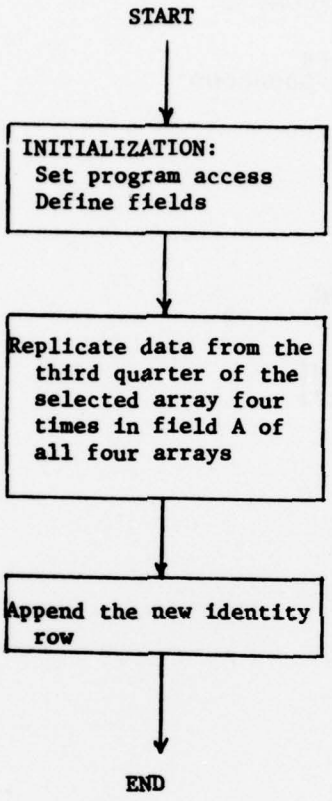


Figure 30: Flow Chart for SUBC.APL

```
pr SUBC.APL

                        SUBC.APL  10/14/76  1253.8 edt Thu


SUB5        START
            ENTRY       SUBC
            EXTRN       COUNTER,FOUR,VALUE,CONST3
            EXTRN       CONST1,CONST,CONST0,CONST2
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
SUBC        LR          FP1,CONST3
            LI          FP2,0
            CLR         X
            CLR         Y
            LOOP,32     MOVED
            GEN,32      X'63C4A0FD'
            GEN,32      X'42008840'
            GEN,32      X'63C5A0FD'
            GEN,32      X'42E088A0'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40C08852'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40808852'
            GEN,32      X'40009943'
MOVED       GEN,32      X'1B600002'
            NOP
            LI          FP3,4
            LI          FP1,0
IDENTA      LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ONE         LI32        C,X'01400000'
            SCW         A,A
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ONE
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENTA
```

Figure 31: SUBC.APL Listing

```
            LI          FP3,4
            LI          FP1,0
IDENT       LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ZERO        LI32        C,X'80000000'
            SCW         A,B
            SCW         A,D
            SCW         A,E
            SCW         A,F
            RPT,64
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ZERO
            INCR        FP1
            DECR        FP3
            BNZ,FP3     IDENT
            B           0(R7)
            END
            END
```

```
r 1254 0.547 0.668 27
```

Figure 31: Continued

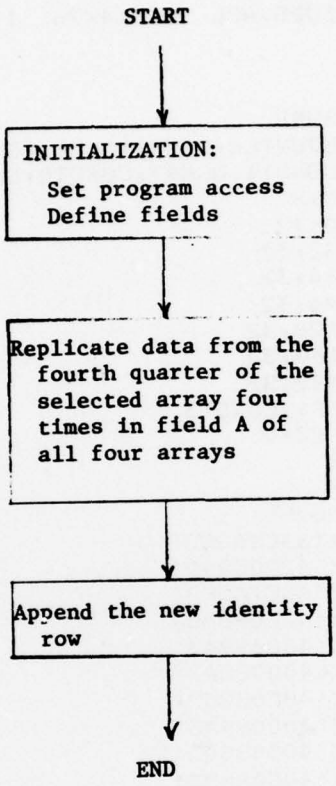


Figure 32: Flow Chart for SUBD.APL

```
pr SUBD.APL

                        SUBD.APL  10/14/76  1127.0 edt Thu

SUB6        START
            ENTRY       SUBD
            EXTRN       COUNTER,FOUR,VALUE,CONST3
            EXTRN       CONST1,CONST,CONST0,CONST2
            ORG         0,R
A           DF          0,32
B           DF          32,32
D           DF          64,32
E           DF          96,32
F           DF          128,32
G           DF          160,32
H           DF          192,32
SUBD        LR          FP1,CONST3
            LI          FP2,0
            CLR         X
            CLR         Y
            LOOP,32     MOVEE
            GEN,32      X'63C6A0FD'
            GEN,32      X'42008840'
            GEN,32      X'63C7A0FD'
            GEN,32      X'42E088A0'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40C08852'
            GEN,32      X'40009943'
            GEN,32      X'400088A2'
            GEN,32      X'40808852'
            GEN,32      X'40009943'
MOVEE       GEN,32      X'1B600002'
            NOP
            LI          FP3,4
            LI          FP1,0
IDENTA      LR          FP2,CONST0
            DECR        FP2
            LI          FPE,4
ONE         LI32        C,X'01400000'
            SCW         A,A
            RPT,6
            INCR        FP2
            DECR        FPE
            BNZ,FPE     ONE
            INCR        FP1
            DECR        FP3
```

Figure 33: SUBD.APL Listing

```
            BNZ,FP3   IDENTA
            LI        FP3,4
            LI        FP1,0
IDENT       LR        FP2,CONST0
            DECR      FP2
            LI        FPE,4
ZERO        LI32      C,X'80000000'
            SCW       A,B
            SCW       A,D
            SCW       A,E
            SCW       A,F
            RPT,64
            INCR      FP2
            DECR      FPE
            BNZ,FPE   ZERO
            INCR      FP1
            DECR      FP3
            BNZ,FP3   IDENT
            B         0(R7)
            END
            END

r 1128 0.323 1.198 57
```

Figure 33: Continued

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C3) activities, and in the areas of information sciences and intelligence. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. Technical and engineering support within areas of technical competence is provided to the Electronic Systems Division's (ESD) POs and other organizations as directed. Technical and engineering support within areas of technical competence is provided to the Air Force Cambridge Research Laboratories and other AFSC organizations as directed. The areas of technical competence include communications, command and control, battle management, information processing, intelligence data collection and handling, solid state sciences, electromagnetic propagation, and electronic reliability, maintainability and compatibility.