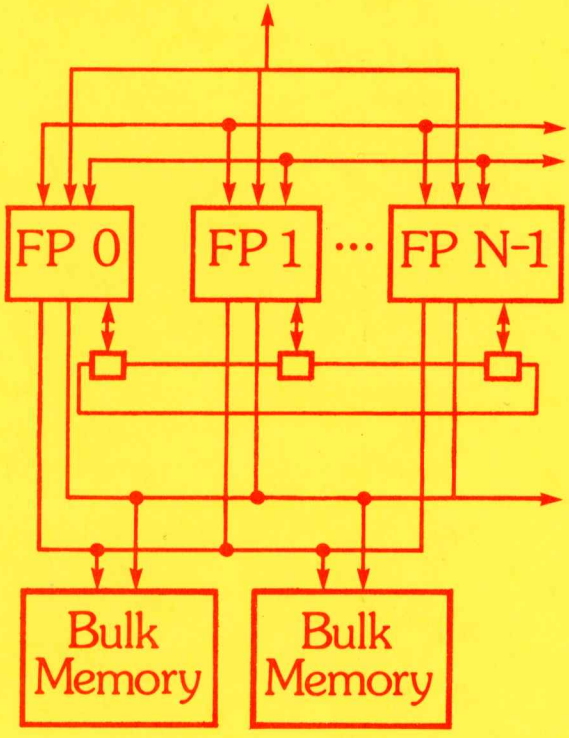
LARS Technical Report 070180
TR-EE 80-33
July 1980

A Multiprocessor Implementation of a Contextual Image Processing Algorithm

Bradley W. Smith
Howard Jay Siegel
Philip H. Swain



Laboratory for Applications of Remote Sensing
School of Electrical Engineering
Purdue University West Lafayette, Indiana 47907 USA

8.0-10313
CR-163402

AgRISTARS

SR-P0-00474
NAS9-15466

"Made available under NASA sponsorship in the interest of early and wide dissemination of Earth Resources Survey Program information and without liability for any use made thereof."

A Joint Program for Agriculture and Resources Inventory Surveys Through Aerospace Remote Sensing

Supporting Research

July 1980

Technical Report

A Multiprocessor Implementation of a Contextual Image Processing Algorithm

by Bradley W. Smith, Howard Jay Siegel and Philip H. Swain

(E80-10313) A MULTIPROCESSOR IMPLEMENTATION OF A CONTEXTUAL IMAGE PROCESSING ALGORITHM (Purdue Univ.) 234 p HC A11/MF A01 .CSCL 05B
N80-32808
Unclas
G3/43 00313

Purdue University
Laboratory for Applications of Remote Sensing
West Lafayette, Indiana 47907

SR-PO-00474
NAS9-15466

TECHNICAL REPORT

A MULTIPROCESSOR IMPLEMENTATION OF A CONTEXTUAL IMAGE PROCESSING ALGORITHM

BY

B. W. Smith
H. J. Siegel
P. H. Swain

This report describes the activity carried out in the Supporting Research Project.

PURDUE UNIVERSITY
LABORATORY FOR APPLICATIONS OF
REMOTE SENSING
1220 POTTER DRIVE
WEST LAFAYETTE, INDIANA 47907

July 1980

Star Information Form

1 Report No	2 Government Accession No	3 Recipient s Catalog No
SR-PO-00474		

4 Title and Subtitle	5 Report Date
A Multiprocessor Implementation of a Contextual Image Processing Algorithm	July 1, 1980
	6 Performing Organization Code
7 Author(s)	8 Performing Organization Report No
Bradley W. Smith, Howard Jay Siegel, and Philip H. Swain	070180
9 Performing Organization Name and Address	10 Work Unit No
Laboratory for Applications of Remote Sensing Purdue University West Lafayette, IN 47907	11 Contract or Grant No NAS9-15466
	13 Type of Report and Period Covered
12 Sponsoring Agency Name and Address	
NASA/Johnson Space Center Houston, TX 77058	14 Sponsoring Agency Code

15 Supplementary Notes

16 Abstract

Contextual classifiers are being developed as a method to exploit the spatial/spectral context of a pixel to achieve accurate classification. Classification algorithms such as the contextual classifier typically require large amounts of computation time. One way to reduce the execution time of these tasks is through the use of parallelism. The applicability of the CDC Flexible Processor system for implementing contextual classifiers is examined. Extensive testing on a CDC Flexible Processor simulator was done. Results show a dramatic increase in throughput can be obtained using the CDC Flexible Processor array.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
Classifying image data, contextual information, contextual classifier, multiprocessor systems, remote sensing	

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price*
Unclassified	Unclassified		

*For sale by the National Technical Information Service Springfield Virginia 22161

NASA - JSC

ACKNOWLEDGEMENTS

This work was supported by the National Aeronautics and Space Administration under Contract NAS9-15466.

This report is based on the Master of Science Dissertation by Bradley W. Smith. Portions of this work have appeared in the following:

Philip H. Swain, H. J. Siegel, and Bradley W. Smith, "A Method for Classifying Multispectral Remote Sensing Data Using Context," Proceedings of the Symposium on Machine Processing of Remote Sensing Data (IEEE Catalog No. 79 CH 1430-8), pp. 343-353, June 1979.

Philip H. Swain, Paul E. Anuta, David A. Landgrebe, and H. J. Siegel, "Volume III: Processing Techniques Development, Part 2: Data Preprocessing and Information Extraction Techniques," LARS Contract Report 113079, November 1979, 160 pages.

Philip H. Swain, H. J. Siegel, and Bradley W. Smith, "Contextual Classification of Multispectral Remote Sensing Data Using a Multiprocessor System," IEEE Transactions on Geoscience and Remote Sensing, Vol. GE-18, No. 2, pp. 197-203, Apr. 1980.

H. J. Siegel, Philip H. Swain, and Bradley W. Smith, "Parallel Processing Implementation of a Contextual Classifier for Multispectral Remote Sensing Data," Proceedings of the Symposium on Machine Processing of Remotely Sensed Data (IEEE Catalog No. 80 CH 1533-9), pp. 19-29, June 1980.

The authors wish to thank David A. Landgrebe and Leah J. Siegel for their comments and suggestions. They also wish to thank Mel Boes, Dave Curry, and Keith Rodwell for helping type the original thesis manuscript.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	v
ABSTRACT	vii
1. INTRODUCTION	1
2. OVERVIEW OF THE FLEXIBLE PROCESSOR	3
2.1 The Hardware	3
2.1.1 Introduction	3
2.1.2 The CDC Flexible Processor	3
2.1.3 Register Files	6
2.1.4 Registers and Arithmetic Units	6
2.1.5 Micro-memory and Input/Output	8
2.1.6 Microprogramming of the Flexible Processor	9
2.1.7 A Flexible Processor Image Processing System	9
2.2. The Software	9
2.2.1 Introduction	9
2.2.2 Registers	12
2.2.3 The Transfer Constant Instruction	12
2.2.4 The Transfer Register Instruction	13
2.2.5 Using the Temporary Files	14
2.2.6 Using the Large Files	15
2.2.7 Programming the Arithmetic Logic Unit	15
2.2.8 The Index Registers	19
2.2.9 Conditional Operations	19
2.2.10 Subroutine Calls, Program Jumps and the Stack	22
2.2.11 The Hardware Multiply	23
2.2.12 Bus Registers	24
2.2.13 Shifting Data	25
2.2.14 Input/Output to the Flexible Processors	25
2.2.15 Interrupts	27
2.3 Conclusions	27
3. THE FLEXIBLE PROCESSOR ARRAY SIMULATOR	28
3.1 Introduction	28
3.2 Organizaton of the Simulator	28
3.3 Operation of the Simulator	29
3.4 Documentation	31
3.5 Changes to Increase Speed	31
3.6 Flexible Processor Micro-Assembler	34
3.7 Conclusions	34

4. FLEXIBLE PROCESSORS SYSTEM IMPLEMENTATION OF MAXIMUM LIKELIHOOD CLASSIFICATION ALGORITHM 35
4.1 Introduction 35
4.2 Implementation of the Maximum Likelihood Classifier on an Flexible Processor Array 36
4.2.1 Introduction 36
4.2.2 Subset of Classes for Each Processor Method . . . 36
4.2.3 Subset of Pixels for Each Processor Method 39
4.3 Conclusions 43

5. FLEXIBLE PROCESSOR SYSTEM IMPLEMENTATION OF A CONTEXTUAL CLASSIFICATION ALGORITHM 45
5.1 Introduction 45
5.2 The Contextual Classifier 45
5.3 Serial Implementation of a Contextual Classifier 47
5.4 Flexible Processor Implementation of a Simple Contexual Classifier 53
5.5 Contextual Classification on a Flexible Processor System 55
5.6 Conclusions 60

6. CONCLUSIONS 61

REFERENCES 65

APPENDICES 66

APPENDIX 1 - Flexible Processor System Simulator Displays. . A-1
APPENDIX 2 - Simulator Flowcharts A-4
APPENDIX 3 - Simulator Listing A-10
APPENDIX 4 - Flexible Processor Micro-Assembler Listing . . A-85
APPENDIX 5 - Implementation of the Maximum Likelihood Classifier on a Flexible Processor A-109
APPENDIX 6 - Contextual Classifier Program Listing A-135

LIST OF FIGURES

Figure		Page
2.1	The Basic Components of a Flexible Processor	4
2.2	Flexible Processor Array Block Diagram	5
2.3	Flexible Processor Structure	7
2.4	The CDC Flexible Processor Coding Form	10
2.5	Typical Flexible Processor Configuration	11
2.6	Complete Listing of the ALU Mnemonics	16
2.7	Complete ALU Instruction Set	18
2.8	ALU Mnemonics	20
2.9	Conditional Mask Functions Implemented on Simulator . .	20
3.1	Flexible Processor Simulator Control Tree Diagram . . .	30
3.2	Sixteen-Flexible Processor Simulator Control Tree . . .	30
3.3	Simulator Commands	32
3.4	Single Step Commands	32
3.5	Memory Editor Commands	32
4.1	An A By B Image Divided Among N Flexible Processors . .	41
5.1	Horizontally Linear Neighborhoods	48
5.2	Pidgeon ALGOL Implementation of the Contextual Classifier -- With Redundant Calculations	48
5.3	Pidgeon ALGOL Implementation of the Contextual Classifier -- Without Redundant Calculations	51
5.4(a)	Underutilization With No Inter-Flexible Processor Communication;	
(b)	Inter-Flexible Processor Data Transfers Required - Full Utilization	56

5.5 Vertically Linear Neighborhoods 56

5.6 Diagonally Linear Neighborhoods 58

5.7 The Diagonals of an A By B Image 58

5.8 Nonlinear Neighborhoods 58

ABSTRACT

Contextual classifiers are being developed as a method to exploit the spatial/spectral context of a pixel to achieve accurate classification. Classification algorithms such as the contextual classifier typically require large amounts of computation time. One way to reduce the execution time of these tasks is through the use of parallelism. The applicability of the CDC Flexible Processor system for implementing contextual classifiers is examined. Extensive testing on a CDC Flexible Processor simulator was done. Results show a dramatic increase in throughput can be obtained using the CDC Flexible Processor array.

1. INTRODUCTION

Since man has been able to fly, he has attempted to gain information about the earth from above. Over the past decade, attempts to extract information from multispectral image data have proved increasingly successful. Traditional methods of pattern recognition applied to individual picture elements have yielded accurate results; however, greater accuracy can be obtained if contextual information is also employed. Accuracy has been increased by up to 55.8% using contextual methods [10]. Because the computational requirements of the contextual classifier are very large, conventional computer systems are not able to handle the processing on a real time basis [10]. One way to reduce the execution time of these tasks is through the use of parallelism.

Various parallel processing systems that can be used for remote sensing have been built or proposed. These include pipelined processors [1], multimicrocomputer systems [8,9], and special purpose systems [4]. The Control Data Corporation (CDC) Flexible Processor system [1,2,3] is a commercially available multiprocessor system which has been recommended for use in remote sensing [5]. The Flexible Processor system includes up to 16 separate processing units called Flexible Processors. In addition to the Flexible Processors, a typical configuration might consist of: a CDC Cyber 170 series computer, a system controller featuring a Cyber 18-20 computer, up to 64K bytes of bulk memory per Flexible Processor, and a high speed data transmission structure called a ring [5]. In depth discussion of both hardware and software aspects of the Flexible Processor system can be found in Chapter 2.

There is a simulator for the Flexible Processor array written in the C programming language [6], which runs on the UNIX operating system. The simulator resides in 64K bytes of main memory, and 161280 bytes of secondary storage during the simulation of 16 Flexible Processors. Further discussion of the simulator is in Chapter 3 and Appendices 1, 2, and 3.

The main computation required by the contextual classifier resembles the Gaussian maximum likelihood classifier. Since the maximum likelihood classifier is considerably less complicated, the software development required for the contextual classifier was based on the maximum likelihood classifier. The logic behind the maximum likelihood classifier, the use of a multiprocessing system to execute a maximum likelihood classifier, and a timing analysis of the maximum likelihood classifier can be found in Chapter 4.

Contextual classifiers are discussed in Chapter 5. A description of the contextual classifier, a serial algorithm, a multiprocessor implementation classifier, and timing analyses are given.

In Chapters 4, 5, and 6, timings of the Gaussian maximum likelihood classifier and the contextual classifier are presented. Both classifiers currently run on the Flexible Processor simulator as discussed in Chapter 3. Chapter 6 draws conclusions on the usefulness of the Flexible Processor array for performing contextual classifications.

2. OVERVIEW OF THE FLEXIBLE PROCESSOR ARRAY

2.1 The Hardware

2.1.1 Introduction

Key elements of the Flexible Processor are discussed first. The focus is on the Flexible Processor, which is the basic building block of the Flexible Processor System. Further details are in [2,3].

2.1.2 The CDC Flexible Processor

The basic components of a Flexible Processor are shown in Figure 2.1. Each Flexible Processor is microprogrammable, allowing parallelism at the instruction level. An example of the way in which N Flexible Processors may be configured into a system is shown in Figure 2.2. There can be up to 16 Flexible Processors linked together, providing much parallelism at the processor level. The clock cycle time of an Flexible Processor is 125 nsec (nanoseconds). Since 16 Flexible Processors can be connected in a parallel and/or pipelined fashion, the effective throughput can be drastically increased, resulting in a potential effective cycle time of less than 10 nsec.

A central feature of the Flexible Processor is its dual 16-bit internal bus structure, enabling the Flexible Processor to manipulate either 16- or 32-bit operands. If 32-bit operands are used, the Flexible Processor can be programmed to execute floating point routines (on its integer hardware) based on the floating point representation of such systems as the IBM 370 and the PDP-11/70. If the needed data width is 16 bits, the Flexible Processor can be programmed to perform different operations on each of the 16-bit words simultaneously.

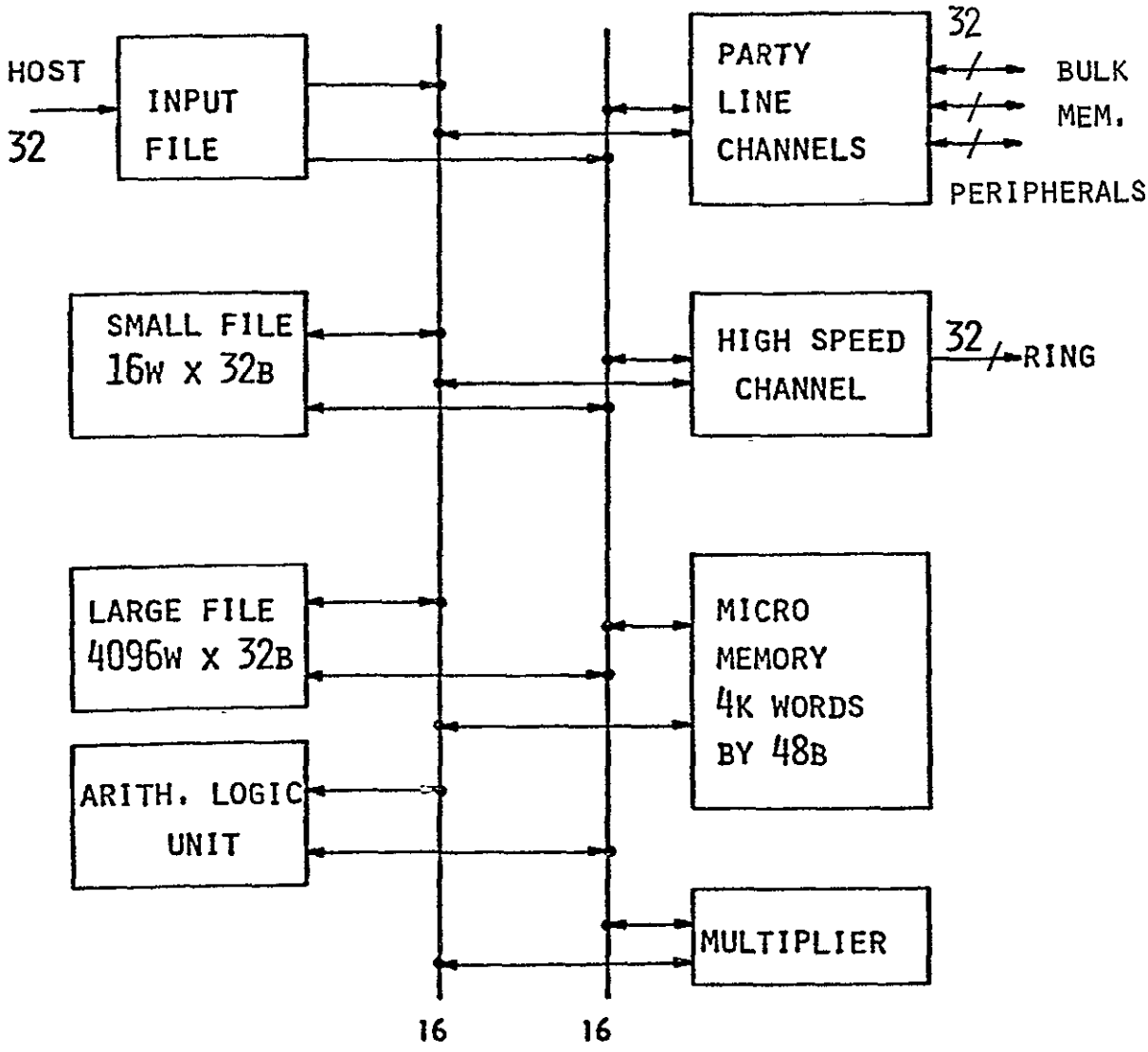


Fig. 2.1. The Basic Components of a Flexible Processor.

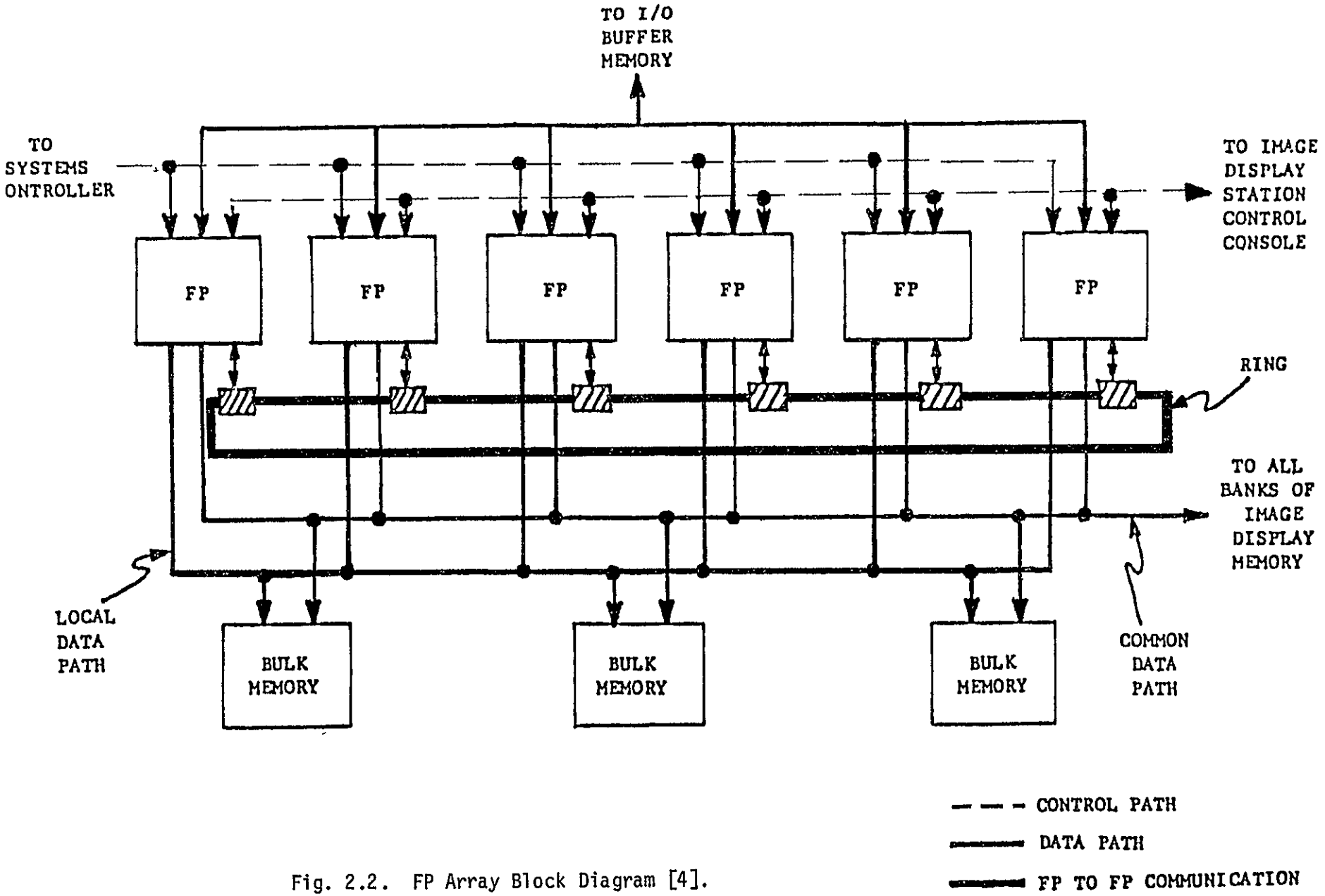


Fig. 2.2. FP Array Block Diagram [4].

2.1.3 Register Files

In each Flexible Processor, there are two files of registers, one called the temporary register file and the other the large register file. Both are divided into 16-bit addressable subunits. If the needed path width is 16 bits, the two files can act like four files, thus creating more addressable space. A special feature of the temporary file is its two separate read and two separate write address registers. This can save much CPU time in many types of matrix operations. The large register file has its own two read/write address registers. It is possible to do either a read or write to either file and simultaneously increment (or decrement) the address register. The temporary file is 16 words, 32 bits each, while the large file is 4096 words, 32 bits each. All of the register files consist of 60-nsec random-access memory.

2.1.4 Registers and Arithmetic Units

Details of the architecture of an Flexible Processor are shown in Figure 2.3. There are three 32-bit general purpose registers called the E, F, and G registers. All of these registers are connected to the arithmetic logic unit (ALU), which can perform 32-bit additions in 125 nsec. The E and G registers are readable directly through the ALU. The general purpose registers can be shifted separately, or the E and F registers can be combined into a 64-bit shift register for double-length shifts. The output of the ALU is a 32-bit register, A, that is addressable by byte (8 bits). This makes a variety of byte manipulations possible. Separate from the ALU is a hardware integer multiplier, which takes two bytes and multiplies them to produce a 16-bit result in 250 nsec. The input registers are the P and Q registers, which are each 16 bits wide. The user can choose which two bytes are to be multiplied. The Flexible Processor is equipped with four index

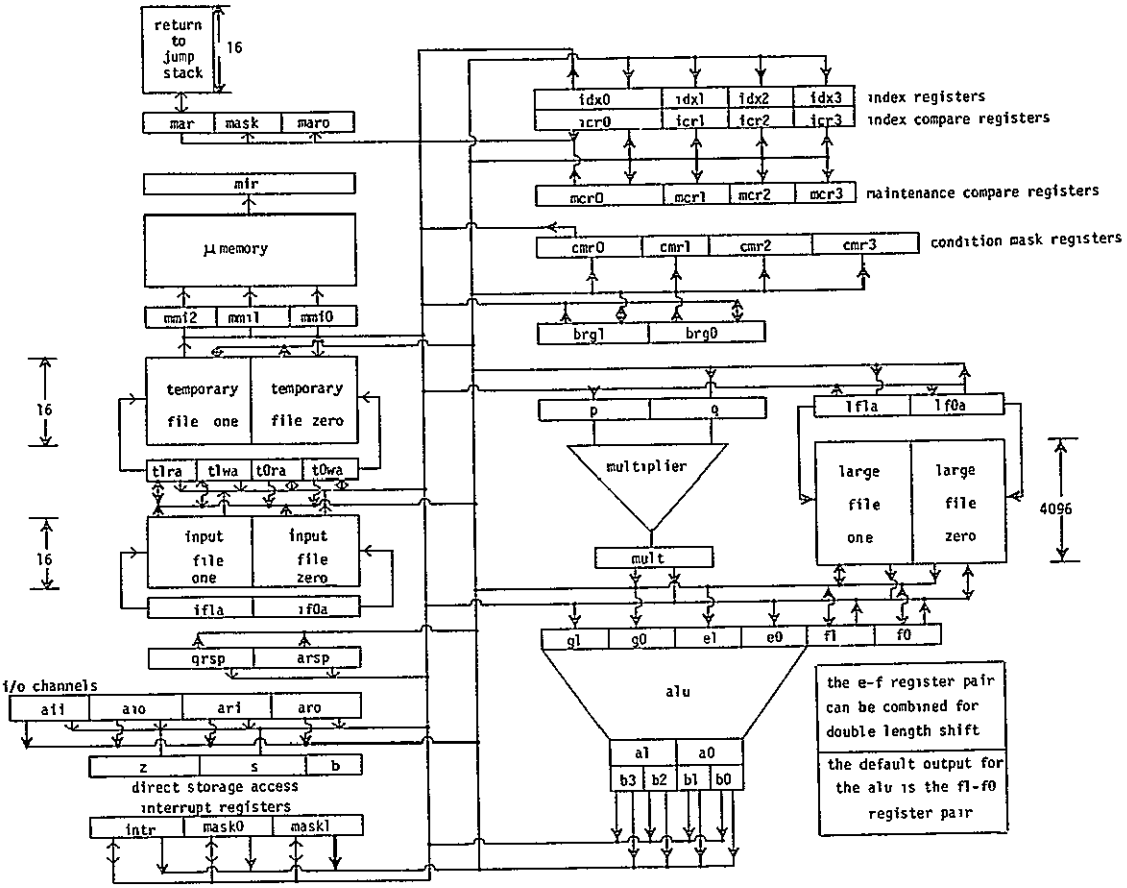


FIG. 2.3 FLEXIBLE PROCESSOR STRUCTURE

ORIGINAL PAGE IS
OF POOR QUALITY

registers and eight corresponding compare registers. The index registers can be used for looping and can be incremented or decremented during any statement not addressing those registers. The Flexible Processor also contains a hardware jump stack, so it is capable of handling standard types of program calls such as subroutine jumps.

2.1.5 Micro-memory and Input/Output

The micro-memory consists of 4K 48-bit words. It stores the microprogram. Each Flexible Processor in a system can contain a different program.

Input/Output (I/O) for the Flexible Processor depends on the overall system (i.e., the Flexible Processor array and its host machine). An Flexible Processor is capable of interrupting another Flexible Processor for I/O. I/O among the Flexible Processors is done one of two ways. The first is a very high speed communication link, arranged in a ring configuration [2,3]. It operates at four mega-words (16-bits per word) per second. Each Flexible Processor has a station on the ring, and each station on the ring is connected to two other stations. When an Flexible Processor does a write to the ring, it gives 16 bits of data and the address of the destination. If a station receives data for another address, it shifts the data to the next station. This is continued until the data reaches the correct station. Special hardware has been added to remove data from the ring in the event of a station failure. The data is loaded into the "input file." This 16 32-bit/word register file can be used as a small buffer. Another form of I/O is through up to 16 64k-byte banks of shared 160-nsec memory. This is not as fast as the previous method; however, for large data transfers, it frees the ring for other communications, as well as providing a buffer between Flexible Processors.

2.1.6 Microprogramming of the Flexible Processor

The Flexible Processor is micro-programmed in "micro-assembly language," allowing parallelism at the instruction level, as indicated in the Flexible Processor coding form shown in Figure 2.4. For example, it is possible to conditionally increment an index register, do a program jump, multiply two 8-bit integers, and add the E and G registers, all simultaneously. This type of operational overlap, in conjunction with the multiprocessing capability of the Flexible Processors, greatly increases the speed of the Flexible Processor array.

2.1.7 A Flexible Processor Image Processing System

Figure 2.5 is provided as an example of one possible Flexible Processor array configuration [5]. The setup of this system has many desirable features for picture processing. The parallel-pipelined architecture of the Flexible Processors enables the system to do rapid matrix multiplications. There are image displays attached, so it is possible to view the pictures. The two 800-bpi tape drives, along with the 50M disk unit, contain enough storage space for jobs that require large amounts of memory. In addition, the system can handle up to eight terminals on its resident operating system (called ICE). Batch jobs can also be run from its 300-card-per-minute reader.

2.2 The Software

2.2.1 Introduction

The host for the Flexible Processor system is programmable in FORTRAN. Flexible Processor programs written in assembly language can be called from the FORTRAN library, enabling the calling programs to be written in FORTRAN [5]. The average user, then, will not have any contact with the Flexible

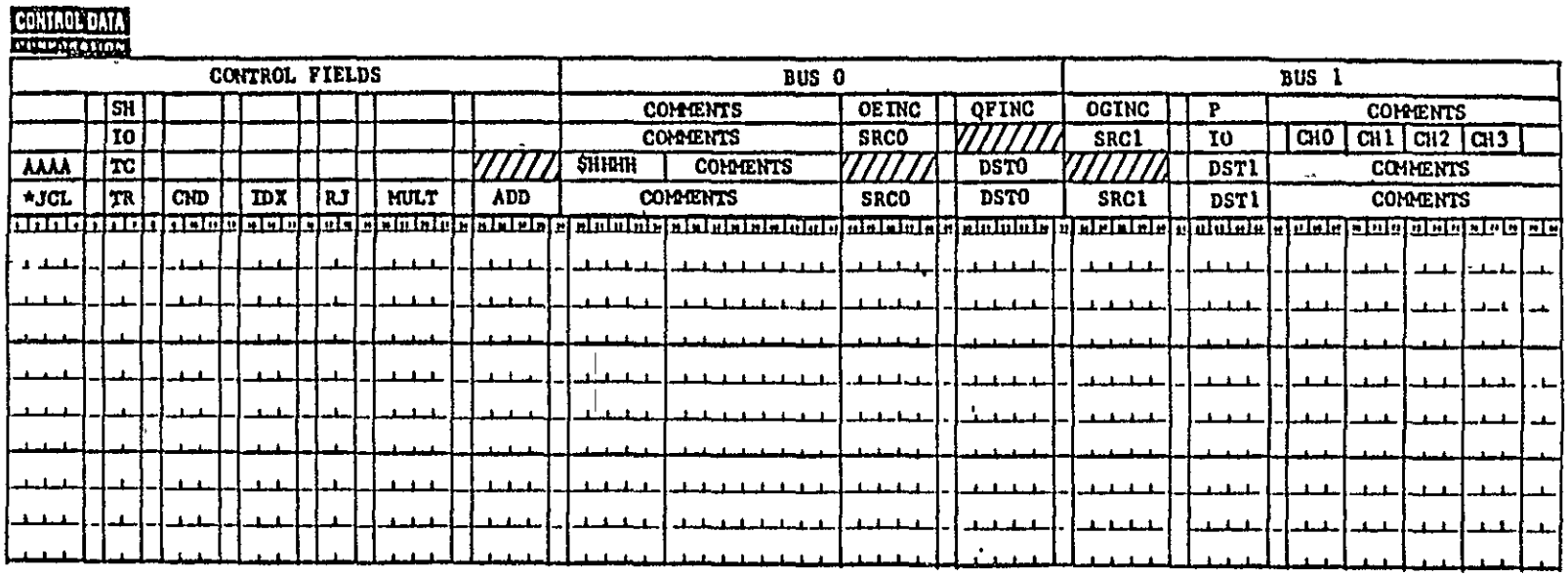


Fig. 2.4. The CDC Flexible Processor Coding Form [8].

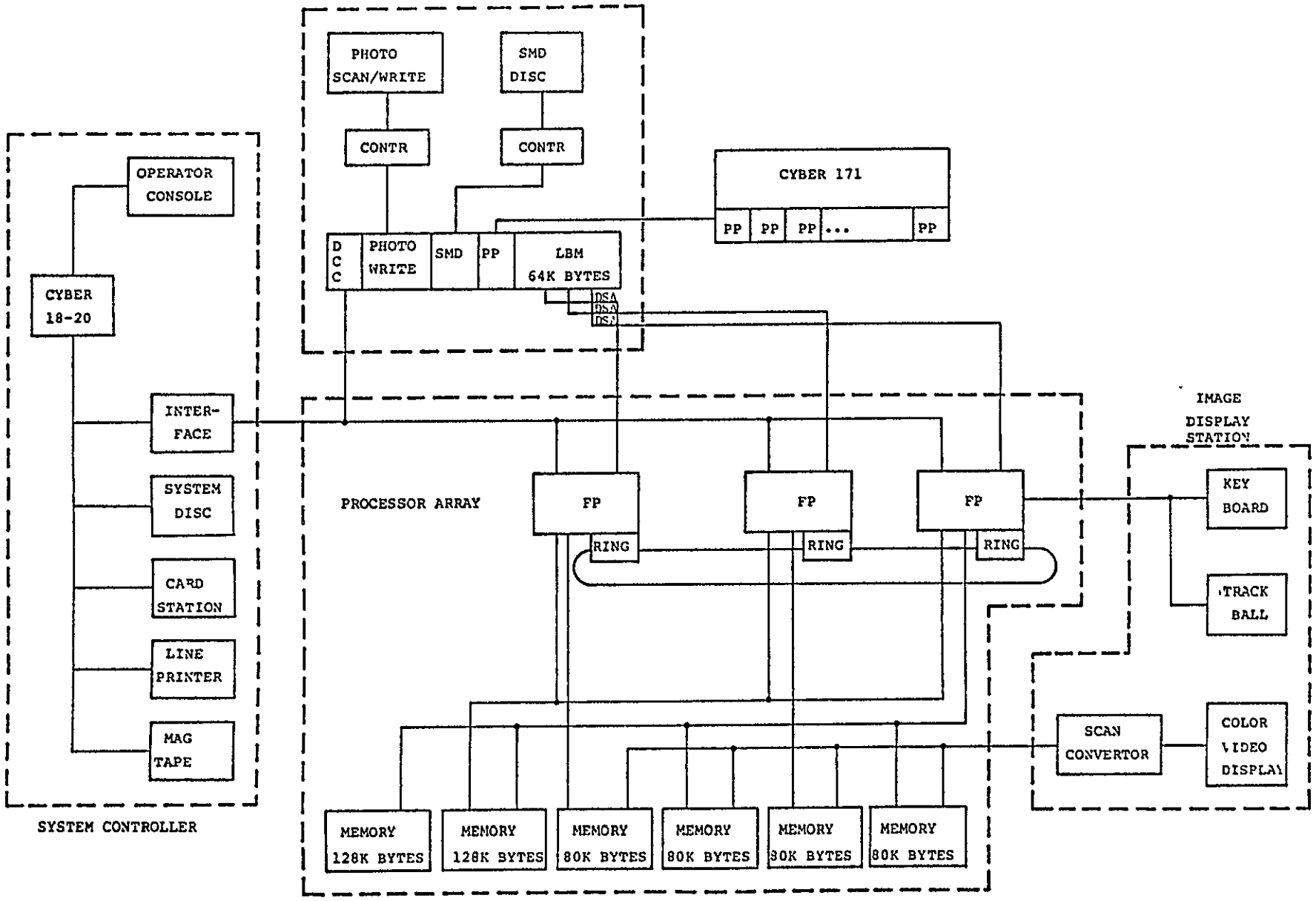


Fig. 2.5. Typical FP System Configuration.

Processor assembly language, making the use of the system much easier. If the necessary Flexible Processor routines exist, data analysis packages, such as LARSYS, which are written in Fortran can, with very simple modifications, run on the Flexible Processor system. The rest of this section overviews how to program an Flexible Processor at the micro-assembly language level.

2.2.2 Registers

The three general purpose registers (E, F, and G) are each divided in halves because they are 32 bits wide and the busses are only 16. The most significant bits of the registers are referred to as the "one" group and the least significant bits are referred to as the "zero" group. For example, the most significant bits of the E register are called E1, and the least significant bits of the E register are called E0.

The ability to address registers in groups of 16 bits allows one to address halves of two separate registers simultaneously. For example, if one wished to write into the upper 16 bits of the F register and the lower 16 bits of the G register, the pair would be referred to as F1G0 in the command. Both will get the same data, but they will get it in one machine cycle instead of two. This increases throughput when, for example, loading initial conditions.

2.2.3 The Transfer Constant Instruction

These registers can be loaded with a constant using the Transfer Constant (TC) instruction. Figure 2.4 shows the coding form. Line three gives the form of the TC instruction format. Omitting the AAAA and the comments, the basic form of the instruction is:

```
TC      $HHHH DST0 DST1
```

The $ tells the assembler that the four following digits are to be interpreted as hexadecimal. This command places the constant on both data lines to enable the loading of two registers simultaneously. The DST (destination) is filled in by an appropriate register which can read off the corresponding bus. Not all registers can provide data to ("source") or receive data from ("destine") both busses. For example, F1 can not read (destine) bus 0, the E and G registers can only be sourced into the arithmetic logic unit, the E1 and G0 registers can read only from bus 1 [3]. Some examples of correct TC instructions are:

```
TC      $FFA8 E0G1 F1G0,
TC      $0100 E0   G0  ,
TC      $0101 E0   NOP .
```

The first command in the example transfers the hexadecimal constant FFA8 to the sixteen-bit registers E0, F1, G0, and G1. The second command transfers the hex constant 0100 to the E0 and G0 registers. In the third command, the NOP indicates bus 1 is not used. Note that while it is not possible to source two different registers at the same time, it is possible to destine two registers off the same bus at the same time.

2.2.4 The Transfer Register Instruction

Another way in which the registers can be used as a source of information is in the Transfer Register (TR) instruction. This is the fourth format shown in Figure 2.4. The basic format of the instruction is:

```
TR    SRC0    DST0    SRC1    DST1
```

This instruction tells the computer to source the register in the SRC0 field to bus 0 and to use the register(s) in the DST0 field as the destination(s). In the event that the other bus is not to be used, a NOP must be placed in

both the SRC and DST fields corresponding to that bus.

2.2.5 Using the Temporary Files

A special feature of the temporary register files, discussed in 2.1.3, is that it has separate read and write indices. The indices are T0RA, T0WA, T1RA, and T1WA, which stand respectively for Temporary file 0 Read Address, Temporary file 0 Write Address, Temporary file 1 Read Address, and Temporary file 1 Write Address. Each is four bits in length. When using the temporary files, one usually initializes the index value and then uses special instructions to increment, decrement, or clear these registers while doing other operations. When storing information to a temporary file, the mnemonic used is TFxf, where x is the file number and f is the function to be performed. The following is a list of the available functions:

U	increment the corresponding index
D	decrement the corresponding index
C	zero the corresponding index
N	perform no operation on the index.

The machine will update the read or write address depending on the context used, i.e., if a temporary file is used as a source, the read address will be assumed, and if it is used as a destination, the write address will be assumed. Some examples are as follows:

```
TC      $0101  TF0U TF1D
TC      $0101  TF0N TF1C
TC      $0101  TF0C TF1C
```

In the examples, the hex constant 0101 is stored in the temporary file while the write pointer is incremented, decremented, unchanged, and cleared.

2.2.6 Using the Large Files

The large files, discussed in Section 2.1.3 have only one pointer per file, but are accessed in the same manner as the temporary file. To access a file, the format is LFxf, where x is the file number and f is the function to be performed on the file. The functions performed are the C, D, and N as defined in Section 2.2.5 and A which adds index register 0 to the corresponding index and uses that location as the desired address. The instruction

```
TC      $0101 LF0U LF1D
```

would store the hex constant 0101 in large files 0 and 1 while incrementing the pointer for large file 0 and decrementing the pointer for large file 1. The length of the large file pointers is 10 bits. Large file pointers are called L0A and L1A. Both the large file and the temporary files pointers can be accessed in the same manner as standard general purpose registers.

2.2.7 Programming the Arithmetic Logic Unit

In the TR instruction there is a field labeled ADD (see Figure 2.4). This field controls the function of the ALU. Output from the ALU is available as the A (accumulator) register, which can be sourced in the same manner as the F and G registers. In the event that the A register is not sourced, the result is moved to the F0-F1 register pair. One feature of the A register is different from the other general purpose registers in that it is byte addressable. This ability makes it one of the most powerful registers on the machine. Figure 2.6 is a listing of the ALU mnemonics and a brief interpretation of their meanings. It is important to remember that it is possible to micro-program this machine; thus there are many possibilities that are not in the mnemonic set. This is the extent of the assembler

Mnemonic:	Function:	Comments:
ADD	A=E+G	Twos complement add the E and G regs.
AND	A=EG	Logical AND the E and G registers.
E	A=E	This is the method for sourcing the E register, making it possible to get data to either bus from the E register.
E+1 E-1 E+E	A=E+1 A=E-1 A=E+E	This makes it possible to increment, decrement, and double the E register without ever having to load a constant.
E-G	A=E-G	Twos complement subtract the E and G register pairs.
E=G	A=E-G-1	The Flexible processor has a branch if negative command. If the E register is less than or equal to the G register, this will branch.
EN	A='E	Logically complement the E register (E NOT).
G	A=G	This makes it possible to use the G register as a data source to both busses.
GN	A='G	Logically complement the G register.
OR	A=E+G	Logically OR the E and G registers.
SB1	A=E-G	Ones Complement subtract the G register from the E register.
SET	A=E+'E	Set A to all ones.
XOR	A=E+G	EXCLUSIVE OR E and G registers.
ZRO	A=E'E	Load A register with all zeros.

Fig. 2.6.

Complete Listing of the ALU Mnemonics.

mnemonics for the ALU, but there are more commands. Figure 2.7 shows a listing of the entire command set. To be able to use this list, first type either an A or a L (for arithmetic or logical) and then a C or a N (for carry or no carry). The A(L) determines the basic function type. The C(N) further determines the type of function by determining the type of carry. With the above, it is possible to use Figure 2.7 to determine the exact function number desired. The only other entity necessary is the function number (from 0 to F). Thus an ANF describes the arithmetic function in the no carry portion of the table that is in the fifteenth row. All three of the function descriptors are places in the column labeled ADD (see Figure 2.4).

As shown in Figure 2.3, the A register is divided into four bytes numbered zero through three. If A0 is sourced, bytes 0 and 1 will be obtained. Likewise, sourcing A1 will yield bytes 2 and 3. If bytes 1 and 2 are needed together, adding an SW (which stands for SWap bytes) to the end of A0 will yield the desired result. If bytes 0 and 3 are needed, adding an SW to the end of A1 will yield the desired result. Thus, A0SW is the correct way to address bytes 1 and 2.

Another feature of the A0 and A1 registers is that they can do a right shift, preserving the signs of the registers. This is accomplished by concatenating a RS (Right Shift) at the end of the desired register. It is possible to do a right shift in conjunction with a byte swap. The ALU has the ability to shift a byte of zeros into either (or both) of the A0 and A1 registers. This is accomplished by shifting both accumulators right by one byte, and loading the upper byte of the pair with zeros. The mnemonic for this is a RZ (Right shift Zero fill) concatenated at the end of the byte pair desired. Figure 2.8 is a list of the possible combinations of the ac-

Function Number	Logical Functions	Arithmetic Operations No Carry	With Carry
0	F = 'E	F = E	F = E+1
1	F = '[E+G]	F = [E+G]	F = [E+G]+1
2	F = ['E G]	F = [E+'G]	F = [E+'G]+1
3	F = ['F F]	F = -1(2's comp)	F = 0
4	F = '[EG]	F = E+[E'G]	F = E+[E'G]+1
5	F = '[G]	F = [E+G]+[E'G]	F = [E+G+E'G]+1
6	F = [E'G+'EG]	F = E-G-1	F = E-G
7	F = [E'G]	F = [E'G]-1	F = [E'G]
8	F = ['E+G]	F = E+[EG]	F = E+[EG]+1
9	F = ['E'G+EG]	F = E+G	F = E+G+1
A	F = G	F = [E+'G]+EG	F = ['E'G+EG]+1
B	F = [EG]	F = [EG]-1	F = [EG]
C	F = [F+'F]	F = E+E	F = E+E+1
D	F = [E+'G]	F = [E+G]+E	F = [E+G]+E+1
E	F = E+G	F = [E+'G]+E	F = [E+'G]+E+1
F	F = E	F = E-1	F = E

[] - contains only logical operations.

Fig. 2.7.

Complete ALU Instruction Set

cumulators and the above operations [3]. The bus numbers are omitted because they can be sourced to either bus. Shift is done before swap. B0, B1, B2, and B3 indicate the four bytes of the A register.

2.2.8 The Index Registers

In the diagram of the machine structure (Figure 2.3), there are four index registers, four index compare registers, and four condition mask registers. None of the registers can be sourced for their contents alone. Index register 0 and its corresponding compare register are 16 bits long, while all the others are only eight bits long. The IDX field, shown in Figure 2.4, is the field that controls the operation of the indices and their compare registers. An INx command, where x is one of the index registers, will increment index register x. A DCx will decrement index register x by one, while a CLx will clear index register x. CLA will clear all registers. The Index compare registers (see Figure 2.4) are used to hold values to be compared to the index registers.

2.2.9 Conditional Operations

The condition mask registers control the condition to be used. These registers do not have a one-to-one correspondence to the index registers. Figure 2.9 is a list of the functions used in the current software (a full listing, appears in [3]). The lengths of the registers are shown in Figure 2.3.

It is possible to test for the conditions in Mask Register 0 by placing a TN in the CND (CoNDition) column. Figure 2.4 shows the location of the CND column in the coding form. To test for the logical "not" of the condition stored in Mask Register 0, an FN is placed in the CND column. To test for

Source A field	Source B field	Bus A		Bus B	
A0	A1	B1	B0	B3	B2
A0	A1RS		Illegal		
A0	A1RZ		Illegal		
A0	A1SW		Illegal		
A0RS	A1		Illegal		
A0RS	A1RS	LS	B1	US	B3
A0RS	A1RZ	Z	B1	US	B3
A0RS	A1SW	B2	B1	US	B3
A0RZ	A1		Illegal		
A0RZ	A1RS	LS	B1	Z	B3
A0RZ	A1RZ	Z	B1	Z	B3
A0RZ	A1SW	B2	B1	Z	B3
A0SW	A1		Illegal		
A0SW	A1RS	LS	B1	B0	B3
A0SW	A1RZ	Z	B1	B0	B3
A0SW	A1SW	B2	B1	B0	B3

Z -one byte of zeros
LS-sign of lower two bytes
US-sign of upper two bytes

Fig. 2.8

ALU Source Mnemonics.

Bit	Condition Mask Reg 0	Condition Mask Reg 3
0	E0 negative	Index Compare reg0 = index 0
1	E1 negative	Index Compare reg0 ≠ index 0
2	F0 negative	Index Compare reg1 = index 1
3	F1 negative	Index Compare reg1 ≠ index 1
4	G0 negative	Index Compare reg2 = index 2
5	G1 negative	Index Compare reg2 ≠ index 2
6	ALU0 negative	Index Compare reg3 = index 3
7	ALU1 negative	Index Compare reg3 ≠ index 3

Fig. 2.9.

Conditional Mask Functions Implemented on Simulator.

the condition in Mask Register 3, an AD is placed in the CND column. Furthermore, the AD must be placed at least two instructions after an increment or decrement of the register in question. If the condition tested is true, the current instruction is executed.

The ability to conditionally execute a statement enables a conditional program jump. Recall that the basic form for a TC statement is:

```
TC      $HHHH DST0 DST1
```

If DST0 is the MAR (Memory Address Register), then after execution of the next statement, the Flexible Processor will do a conditional jump to the value indicated by the hex constant, which can be a program label.

The following is an example of a conditional jump to hex address 1234:

```
TC     $0001 NOP CMR3
TC AD  $1234 MAR NOP
```

The first statement will set the condition mask, while the second statement will jump to memory location 1234 if IDX0 = ICR0. To do an unconditional program jump, omit the AD. The following:

```
TC      NEXT  MAR  NOP
```

will jump to the program label NEXT. Since the MAR and instruction fetch of the Flexible Processor are buffered, it is impossible to do an immediate program jump. This adds little complication to the programming, except that the step to be executed before the jump is placed after the actual jump statement. It is very important, when reading source code for the machine, to remember that the order of execution is reversed.

The Flexible Processor contains two program status words. One can be user loaded and is called PAST. The other contains the current program

status word and is called NOW.

2.2.10 Subroutine Calls, Program Jumps, and the Stack

As shown in Figure 2.3, there is a 16-by-12-bit stack called the return jump stack. This is a typical buffer which is used to hold return addresses as well as temporary data. As indicated in Figure 2.4, there is a field labeled RJ. This controls the return jump stack. There are three possible commands for the stack. SR (SubRoutine jump) will take the current value of the MAR (which is pointing to the next statement), increment it by one and store the result on the top of the stack. This will be the return address. JP (JumP return) takes the current top of stack and places it in the MAR. DF (Delete First item) will delete the top of the stack. The JP does not perform the delete function. Another feature of the SR, JP, and DF is that they all trap out interrupts.

The MAR is buffered, so all operations that seem to be performed on the MAR are actually performed on the buffer. One program cycle is needed to dump the buffer into the MAR. This makes the micro-assembly language somewhat confusing, as the Flexible Processor will execute the statement immediately following any modification to the MAR. For simplicity, the examples use a NOP following a jump. In actual practice, however, this will be replaced by a statement that is more productive.

A typical subroutine jump looks like the following:

(Fields) label	type	RJ	$HHHH	DST0	DST1
	TC	SR	$1234	MAR	NOP
	TC		NOP	NOP	NOP

The above routine will store label+2 on the stack, execute the NOPs, and jump to the hexadecimal location 1234. A typical subroutine return looks

like the following:

(Fields)	TYPE	RJ	$HHHH	DSTO	DST1
	TC	JP	NOP	NOP	NOP
	TC	DF	NOP	NOP	NOP

This will take the top of stack, place it in the MAR, and then delete the top of stack. Since the CND field is valid on all types of instructions, it is possible to do a conditional subroutine jump just by placing the condition in the conditional field. The result looks like:

(Fields)	TYPE	CND	RJ	$HHHH	DSTO	DST1
	TC	AD	SR	$1234	MAR	NOP

This will store the value of the return address, execute the next statement, and continue execution at location 1234.

2.2.11 The Hardware Multiply

The only remaining functional unit to be discussed is the hardware multiply. As shown in Figure 2.3, the inputs are the P and Q registers, which are each 16 bits in length. The result of the multiply is a 16 bit product, which can be the result of the multiplication of any two bytes. This is the only case where the same byte can be sourced twice. The mnemonics for the addressing is L for the lower byte, and U for the upper byte. Thus, to multiply the lower byte of the P register by the upper byte of the Q register, a PLQU would be placed in the MULT field. Caution must be taken when a multiply is initiated. A multiply takes two machine cycles before the result can be sourced. If an interrupt is received before the result is ready, the result will be lost. To prevent such loss, it is necessary to trap out all interrupts. This is accomplished as follows: whenever a multiply is done, an SR is placed in the RJ column of the first statement of the multiply, and a DF is placed in the RJ column of the second multiply statement. The net result is to

push a return address onto the stack and then pop it off the stack. This will trap out interrupts as needed. Further caution must be taken in that the RJ stack is only 16 units long, so overflow is possible. If overflow occurs, no error will be flagged. The following is a routine to square the lower byte of the Q register.

(Fields)	type	RJ	MULT	$HHHH	SRC0	DST0	SRC1	DST1
	TC	SR	QLQL	$0057		MAR		NOP
	TR	DF	QLQL		MULT	F0	MULT	F1

This not only does a multiply, but it also does a program jump and traps interrupts all at the same time, showing how this machine obtains very high processing speeds. (Consider that each program step takes .125 micro-seconds.) If more precision is desired, the following algebraic rule can be used:

$$(a+b)*(c+d)=ac+ad+bc+bd.$$

This rule can be modified to the byte level, yielding the 32- bit result in under three micro-seconds [7].

2.2.12 Bus Registers

The two registers in Figure 2.3 labeled BRG0 and BRG1 are the bus registers. Normally these are used for breakpointing. It is possible to use these registers for general purpose registers (if no breakpointing is needed). To write into these registers, BRG0 and BRG1 are put into the respective columns, while to read from these registers, BSR0 and BSR1 are put into their respective columns.

2.2.13 Shifting Data

The SH field of an instruction is shown in Figure 2.4. The OEINC, OFINC, and OGINC fields determine what type of shift is to take place (left or right, circular or not, padded with ones or zeroes or data from the program status word). The P field determines the Precision of the shift. If the P field is set to S, all of the registers are treated as separate registers; however, if the P field is set to D (Double precision), the E and the F registers are tied together as one register for the shift. There are commands that not only determine the data to be shifted, but they also control the conditions under which the shifts are done [3].

2.2.14 Input/Output to the Flexible Processors

Input/Output (I/O) is one of the most complicated parts of the entire Flexible Processor system. I/O must occur in one of the following forms:

1. Flexible Processor to host
2. Flexible Processor to Flexible Processor
3. Flexible Processor to MOS RAM (shared bulk memory).

For large amounts of data requiring Flexible Processor to Flexible Processor communication, FP to MOS RAM is the most reasonable form of data transfer. If the high-speed communication link, as described in 2.1.5, is used, there is only a buffer for 16 words of information. This requires very closely timed algorithms, as any error would result in the loss of data. Each Flexible Processor is connected to four 16-bit channels, which are called Direct Storage Access (DSA) channels. Each of the channels is connected to four banks of 250 nsec MOS RAM. Each bank of MOS RAM is addressed by bank and channel. Different banks on various channels may be shared. For example, bank 1 on channel 3 may be the same as bank 2 on channel 1. The Flexible Processor is capable of choosing a bank and address to which all the channels are linked through four S registers (Storage location) and B (Bank) re-

gisters. Since the RAM memory is much slower than the clock cycle, the read is done in two stages. The first stage sends the address to the MAR of the specified bank. Upon completion of a read, the Flexible Processor will automatically increment the MAR of the specified bank by one. Within the next four cycles, the data will appear in the Zx register, where x is the channel number (see Figure 2.3). The data will remain in the Zx register until the next read is initiated. In the event of a "memory bank busy," or "data not ready," the Flexible Processor will automatically wait for two machine cycles, after which it will repeat the process. To do a write, the data is sourced directly to the MBR (Memory Buffer Register) of the memory bank corresponding to the bank register. (A write is a one stage process.) The Flexible Processor is programmed to do I/O through the IO statement type. Figure 2.4 shows the form of the statement. The IO statement is similar to the TR statement in that arithmetic calculations can be done simultaneously with I/O. The following statements show how to initialize the S and B registers. (The S and B registers are linked together so that they can be loaded in one statement.)

IO	CND	IDX	RJ	MULT	ADD	SRC0	SRC1	IO	CH0	CH1	CH2	CH3
IO					ZR0	A0	A1	DS	LS	LS	LS	LS
IO						F0	F0	DS	LB	LB	LB	LB
IO			DF	PLQL		MULT	MULT	DS	LSB	LSB	LSB	LSB

The first statement loads all four S registers with 0000. The second loads all four B registers with the contents of F0. The third loads all four S and B registers with the output of the multiplier. The DS stands for DSA I/O. The leading L in the channel column stands for load.

After initializing the S and B registers, the read needs to be initialized, which is done by placing an R in the channel field of the channel to be read. Four cycles later, the data (or a wait) should appear in the

Zx register. To do a write, a W is placed in the channel fields into which the data is to be written. The data to be sourced is in the source fields.

2.2.15 Interrupts

With I/O, interrupts are often needed. The Flexible Processor has the ability to handle up to 16 different interrupts [2,3]. The Flexible Processor can interrupt itself, the host and other Flexible Processors. While processing an interrupt routine, the Flexible Processor sets a flip-flop indicating that an interrupt is being processed. This traps out all lower priority interrupts. The interrupt flip-flops are reset when the program returns to processing the original routine, or until a zero is stored in the interrupt register.

2.3 Conclusions

This has been an introduction to the parts of the Flexible Processor and the parts of the instruction set that will be used in the Gaussian maximum likelihood classifier and contextual classification algorithms discussed in Chapters 4 and 5. For further documentation, consult the CDC Flexible Processor Textbook [3].

3. THE FLEXIBLE PROCESSOR ARRAY SIMULATOR

3.1 Introduction

Each Flexible Processor has a complicated microprogrammable internal architecture. This was overviewed in Chapter 2. As stated earlier, an advantage of this microprogrammable architecture is that it allows parallelism at the instruction level. This makes user verification of the correctness of Flexible Processor algorithms and accurate mathematical timing analyses of these algorithms very difficult. Thus, in order to debug, verify, and time Flexible Processor algorithms, a simulator and micro-assembly language assembler for an array of Flexible Processors have been developed. The simulator and assembler run under the UNIX operating system on a PDP-11 series computer, which has been used successfully to program a maximum likelihood classifier, as discussed in Chapter 4, and a contextual classifier, as discussed in Chapter 5. The simulator displays the contents of the Flexible Processor registers on a terminal screen, in a format demonstrated in Appendix 1. This chapter describes the organization and operation of the simulator.

3.2 Organization of the Simulator

The Flexible Processor system simulator is based on a single FP simulator developed at Purdue [7]. Its capabilities have been extensively expanded.

The current version can simulate up to sixteen Flexible Processors, the maximum number allowed in an actual system. Further, should any further design changes take place in the actual system, the simulator can be modi-

fied to simulate up to forty-eight Flexible Processors. The current maximum program length is 2000 lines. The simulator occupies approximately 64K bytes of main memory.

The simulator is divided into four programs, all written in C [6], a language much like PL/I or PASCAL. Each of the four programs performs a different task. "Monh.c" is the system monitor, which interfaces the simulator to the user. "EXECh.c" is the simulator, which simulates all of the system instructions except the I/O and the shift instructions. "shioh.c" simulates the rest of the instruction set. The final program in the set is "helph.c," which contains a brief help file for the user who is stranded in the monitor routine. In addition, helph.c contains special routines that make the program consistent with all versions of the UNIX operating system. This makes the program portable for use on any system that supports UNIX and the C programming language. In addition, this routine contains all the paging algorithims that are used, making the routines localized, easing possible debugging problems in the future.

3.3 Operation of the Simulator

The program structure for a single Flexible Processor simulation can be represented by the control tree diagram in Figure 3.1. All register files are considered indexed registers. The 16-Flexible Processor system is basically the same tree structure, but there is one more level in the control tree, as shown in Figure 3.2. The structure beneath the command level is the same as for the single Flexible Processor case. If the monitor receives a '#', it will move one node closer to the root of the control tree on any of the branches.

In the Command Level, there are ten possible commands, which are shown in Figure 3.3. If an s is chosen, the simulator will simulate the execution

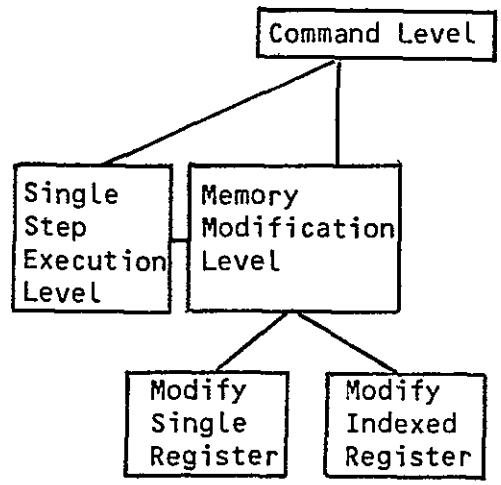


Fig. 3.1. Flexible Processor Simulator Control Tree Diagram.

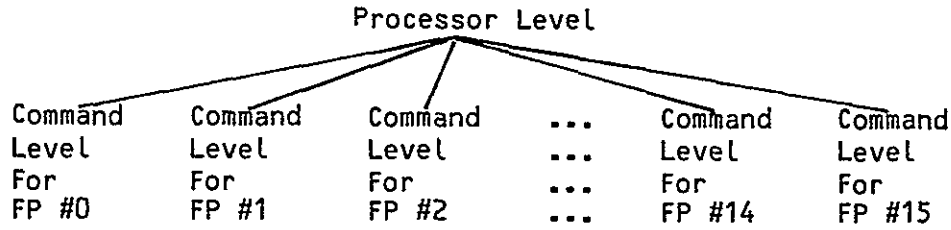


Fig. 3.2. System Flexible Processor Simulator Control Tree.

of one program step and will move to the single step node. Figure 3.4 gives the command set for the single step node. If the m is typed, the only valid arguments are a '#' or a register name. The monitor will print the old value of the register and ask for a new one if the register named is a single register. If the register selected is a register file, the monitor will ask for the index. Upon receiving the index, the monitor will print the old value and prompt the user for input. Valid commands are shown in Figure 3.5. Invalid input will yield a "What?" asking for a correct command.

These are all of the functions supported by the simulator at this time.

Appendix 2 contains flowcharts outlining the operation of the simulator. Appendix 3 contains a source listing of the simulator. As mentioned previously, the maximum likelihood classifier and a contextual classifier have been implemented using the simulator. These are discussed in Chapters 4 and 5.

3.4 Documentation

At the beginning of every major portion of program code, there are comments describing the program flow and variables. This should facilitate understanding of the routine and future simulator modifications, as it translates the routine from a computer language into English.

3.5 Changes to Increase Speed

Normally, output to the terminal is done one character at a time. This requires the program to generate an interrupt to the operating system for each character to be displayed. The operating system then checks several flags, adds special characters where needed, awakens the device driver, tells the device driver which terminal gets the output, and does the output.

s	Single step program.
m	Go to memory level.
l	Load assembled object code.
t	Print the contents of the registers after the input offset (used for debugging simulator).
v	Save the current register values in a file called status.
e XXX	Execute XXX program steps.
stop	Exit from monitor routine.
! unix	Execute system command.
#	Move up one node.
p	Print out all the registers.
h	Print out the help file, current node.

Fig. 3.3. Simulator Commands.

s	Single step program.
m	Go to memory level.
e XXX	Execute XXX program steps.
#	Move up one node to command level.
p	Print out all the registers.
h,H	Print out the help file, followed by the name of the current node.
dtemp	Display the contents of temporary files.
dlarge	Display the contents of large files.
dmem	Display the contents of micro-memory.

Fig. 3.4. Single Step Commands.

c XXX	Changes the old values to XXX.
i	Increments the index without changing the old value.
^	Decrements the index without changing the old value.
#	Return to original level (either the command level or the single step level).

Fig. 3.5. Memory Edition Commands

The output from a single execution step requires exactly one screen, which is 3370 characters. Buffering is done so that the computer handles the interrupt routine once per screen instead of once per character. The only change in the interrupt routine is that instead of displaying one character, the computer displays 3370. This reduces the load on the system by 3369 interrupt routines per screen of output. Most of the time required for output is not due to the physical transfer of data; rather, it is due to the other areas of the interrupt routine. The net result is that the simulator output is over 3300 times faster with buffering than without. While the different command levels require different size buffers, the average increase in speed due to buffering is 4500%.

The PDP-11 series computer uses 16 address bits; thus the maximum amount of data address space is limited to 65,536 bytes. Each simulated Flexible Processor memory and registers require approximately 60,000 bytes, so a special paging routine was written to page the simulated Flexible Processor memories and registers in and out of main memory as required. Output to disk is done in units of 65,536 bytes instead of units of 1 byte. This makes the swapping routine to exchange a part of Flexible Processor memories run in 1 second. Without buffering, this routine took 2.5 hours of straight transfer time. Originally, this program required the total computing power of a PDP-11/70. Now, this program can run on a PDP-11/45 in a time-shared environment.

In a high level language, such as C, PL/I, FORTRAN, or PASCAL, one program step corresponds to many machine steps. To minimize the number of machine steps per program step, frequently used variables were placed in the registers of the machine. For example, the program step:

```
C=C+1
```

requires the computer to load the variable C from memory. The machine then loads C into a register, increments the register, and stores C back into its original location. Frequently accessed variables are placed in a register, so frequent memory fetches are less necessary. This often shortens the number of executed steps by three steps. When C is not used, C is stored and accessed in the usual manner. Thus, the hardware of the computer was used to obtain maximum throughput.

3.6 Flexible Processor Micro-Assembler

The micro-assembler [7] takes the Flexible Processor micro-assembly language and translates it into machine micro-code. A microprogram must end with a # to signal the end of input to the micro-assembler. After the micro-assembler is invoked, it prompts the user for the input file. When it is finished, it will move the assembled output to a file called "object" which can then be loaded into the simulator via the load command. A source listing for the assembler appears in Appendix 4.

3.7 Conclusions

The Flexible Processor micro-assembler and simulator are operational and have been used. Up to 16 Flexible Processors can be simulated. The current versions do not include Flexible Processor-host, inter-Flexible Processor (ring), and Flexible Processor-bulk memory communications.

4. FLEXIBLE PROCESSOR SYSTEM IMPLEMENTATIONS OF A MAXIMUM LIKELIHOOD CLASSIFICATION ALGORITHM

4.1 Introduction

To demonstrate the use of a Flexible Processor system on a task less complex than the contextual classifier, consider the analysis of Landsat data using a Gaussian maximum likelihood classifier. Landsat measurements are taken from four spectral bands and are received by the Flexible Processor as a data vector. Based on decision theory akin to that developed in the contextual classifier model, the vector is classified by determining the probability that it belongs to each information class and assigning it to the class for which this probability is maximum.

The way in which an Flexible Processor may be used in implementing a Gaussian maximum likelihood classifier is demonstrated below. The techniques described are to be extended to the contextual classification algorithm.

In Section 4.2, methods for implementing the maximum likelihood classifier on an Flexible Processor array are presented. The ways in which the contextual classifier can be implemented on an Flexible Processor array are presented in Chapter 5.

4.2 Implementation of the Maximum Likelihood Classifier on an Flexible Processor Array

4.2.1 Introduction

Two methods for implementing the maximum likelihood classifier on an Flexible Processor array are discussed. The first assigns to each Flexible Processor a different set of classes, and each Flexible Processor processes all pixels for its assigned classes. The second method assigns to each Flexible Processor a different subimage, and each Flexible Processor processes the pixels in its subimage for all classes. The basic matrix operations are the same for both methods.

The ability to do a fast matrix multiply is at the heart of efficiently implementing the maximum likelihood classifier. The form for the matrix multiplication portion of the discriminant function calculation is:

$$(X-U_i)^t(C_i^{-1})(X-U_i),$$

where X is the data vector, U_i is the mean vector for the ith class, and C_i is the covariance matrix [10] for the ith class.

4.2.2 Subset of Classes for Each Processor Method

Consider the use of the Flexible Processor array to perform these classifications using the first method. Assume there are m distinct classes and the computer contains p Flexible Processors. Each Flexible Processor is assigned to process m/p classes. The large file in each Flexible Processor is initialized with the inverse of the covariance matrix and mean vector for each class it was assigned. The current data vector is stored in each Flexible Processor in the temporary file. When a new data vector is loaded into an Flexible Processor, it overwrites the previous one. For simplicity, but

without the loss of generality, in the following assume that m = p. If m is greater than p, then in each Flexible Processor instead of applying just one inverse covariance matrix to the data set several would be applied. This will, of course, increase the execution time by a factor of approximately m/p.

In standard arithmetic, one would first multiply the $(X-U_i)^t$ and the C_i^{-1}, creating a new vector. This vector would then be multiplied by $(X-U_i)$ resulting in a scalar. In our implementation, the order has been somewhat altered. $(X-U_i)^t$ is multiplied by a column of C_i^{-1}, accumulating in a variable called "sum." After this is done for a column j of C_i^{-1}, "sum" is multiplied by $(X-U_i)_j$ (the jth element of $(X-U_i)$), accumulating the result in a variable called "hold" and re-initializing "sum" to 0 [1]. The following is a pidgeon ALGOL description of the process:

```
HOLD =0;
FOR J=1 TO N DO
     BEGIN:
     SUM=0;
     FOR I=1 TO N DO
          BEGIN:
          SUM=SUM+X[I]*C[I,J];
          END
     HOLD=HOLD+SUM*X[J];
     END
```

where N is the dimension of covariance matrix, X[I] is the Ith element of the input vector, and C[I,J] is the element in the Ith row and Jth column of covariance matrix. At the end of the routine, the value contained in the "hold" variable is the desired scalar. This algorithm requires fewer stores and fetches than the standard algorithm, so it shortens the run time of the process. All pointers are kept in the index register, further simplifying the process. Temporary file locations are used for sum and hold, so the three general purpose registers can be kept free for the floating point

operations.

One way to perform this algorithm is to have the host send $(C_i)^{-1}$ and U_i to Flexible Processor i. The host then sends the current data vector to Flexible Processor 0, then 1, etc. When the processor receives the data vector, it calculates "hold." After the host gives all Flexible Processors the data for pixel (i,j), it waits until Flexible Processor 0 has calculated the value for its "hold." The host then retrieves the value of "hold," loads Flexible Processor 0 with the data vector for the next pixel, and adds a precomputed constant to calculate the discriminant function. The host executes this process for all Flexible Processors. After the last Flexible Processor has transmitted the result, the host does a compare and stores the class index corresponding to the maximum of the discriminant values computed for this pixel. Thus, the compares and adds are done by the host while the Flexible Processors are computing the "hold"s for the next pixel, minimizing delay.

This maximum likelihood classifier implementation has been programmed on a simulator for a Flexible Processor array at the Laboratory for Applications of Remote Sensing. The simulator displays the contents of the main registers and provides a variety of tools for debugging Flexible Processor microcode, as is discussed in Chapter 3.

Allowing 40 Flexible Processor machine cycles for each floating point addition and 9 Flexible Processor machine cycles for each floating point multiply, the number of machine cycles is as follows, where j = number of pixels and n = number of measurements (size of data vector):

setup and clear registers:	9
load mean:	2n
load covariance matrix:	$4n^2$
load and normalize data vector:	42jn+j

inner loop of algorithm: $56jn^2$

outer loop of algorithm: $61jn$

$$56jn^2 + 103jn + 4n^2 + 2n + j + 9$$

Floating point numbers had an eight-bit exponent and 16-bit mantissa. This assumes that m, the number of classes, equals p, the number of processors. If m is greater than p, the runtime may be approximated by multiplying by $\lceil m/p \rceil$.

Preliminary tests indicate that a single Flexible Processor will perform a maximum likelihood classification faster than a PDP-11/70 with floating point hardware. Exact comparisons of the Flexible Processor array performance with other systems are difficult without detailed information about factors such as pre- and/or post-processing of the data not included in the computation time, data precision used, memory load time, etc. However, to give a general idea of the effectiveness of this approach, consider a classification of 256-by-256 pixels of Landsat data (n=4) using 16 classes and a complete array of 16 Flexible Processors (and a host machine). The total processing time is approximately 10.4 sec. ESL states that their array processor gives up to an increase of 25 times over the IBM 370/158. On the classification of four channels into eight classes, their time is 6.3 sec.

4.2.3 Subset of Pixels for Each Processor Method

An alternative method to perform the pointwise maximum likelihood classification of pixels using a Flexible Processor array is based upon having each Flexible Processor perform the maximum likelihood classifier for a different section of the image. Recall, the contextual classifier performs computations similar to those used by the maximum likelihood classifier, but is complicated by the involvement of "neighboring" pixels.

Consider performing a maximum likelihood classification on an A-by-B image with N Flexible Processors. One way to approach the problem is to divide the image into N subimages and have each Flexible Processor perform the maximum likelihood classification for all pixels in its subimage. This is shown in Figure 4.1. If all subimages have the same number of pixels, then the Flexible Processors will be fully utilized and the classification of the entire image will take approximately 1/N as much time as it would take a single Flexible Processor to perform the entire classification. Thus, maximum improvement, i.e., a factor of N, is obtained.

Consider the case in which each subimage does not contain the same number of pixels, which will occur when (A*B)/N is not an integer. This will lead to underutilization of the Flexible Processors, but this underutilization will be negligible as will now be shown.

One way to approach this situation is as follows. To each of N-1 Flexible Processors, assign a subimage of size

$$\left\lceil \frac{(A*B)}{N} \right\rceil ,$$

where $\lceil x \rceil$, the ceiling of x, is the smallest integer greater than or equal to x. To the remaining Flexible Processor, assign a subimage of size

$$(A*B)-(\left\lceil \frac{(A*B)}{N} \right\rceil * (N-1)) .$$

For example, if A=117, B=196, and N=16, then

$$\left\lceil \frac{22,932}{16} \right\rceil = \lceil 1433.25 \rceil = 1434$$

pixels are in each subimage associated with 15 Flexible Processors. The remaining pixels, of which there are

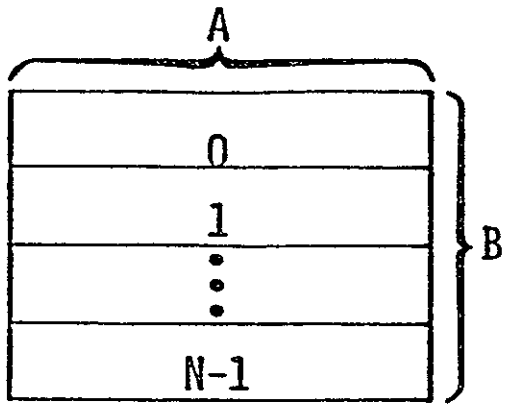


Fig. 4.1.

An A-by-B Image Divided

Among N Flexible Processors.

$$22{,}932-(15*1434)=1422$$

are associated with one Flexible Processor. This sixteenth Flexible Processor will have fewer pixels to classify and thus will finish before the other Flexible Processors (assuming that, on the average, the time for the floating point calculations is approximately the same for all pixels, which implies some underutilization of the Flexible Processor since it must sit idle waiting for the others to finish). Ideally, a factor of N=16 performance improvement over a single Flexible Processor is desired, which, in this case, would require all 16 Flexible Processors to each classify 1434 pixels. To compute the utilization of the Flexible Processor array, divide the number of pixels actually classified by the maximum number that could be classified in the same amount of time if all 16 Flexible Processors were fully utilized. Thus, the utilization is:

$$\frac{22{,}932}{(16*1434)} = 99+\%.$$

Therefore, a factor of 99+% of N improvement is obtained.

In general, using the above assignment of pixels to subimages, the utilization of the system is

$$\frac{A * B}{\lceil (A * B)/N \rceil * N}$$

The maximum value of the denominator is A*B+N-1 and occurs when A*B = k*N+1, where k is an arbitrary integer. Therefore,

$$\min((A * B)/(\lceil (A * B)/N \rceil * N)) = (A * B)/(A * B + N - 1).$$

Based on typical sizes of remotely sensed images and assuming that the maximum size of a Flexible Processor array is 16,

$$A * B > 10 * N,$$

and

$$(A * B)/(A * B + N-1) > 99\%.$$

Thus, in general, the worst case performance is 99+% of the ideal factor of improvement over a single Flexible Processor.

In Appendix 5, the maximum likelihood classifier programs for this Flexible Processor implementation are described. Included are the routines for floating points arithmetic, using a 14-bit exponent and a 16-bit mantissa. The current algorithm, which runs on the simulator described in Chapter 3, uses 3526 125-nsec steps to process one pixel (four floating-point component data vector) and two classes, including choosing the maximum value. Performing a two class maximum likelihood classifier on 400 pixels of actual Landsat data (as used in the tests described in [12]), a single Flexible Processor averaged 410 microseconds per pixel (including the time to move the image data from the bulk memory to the processor). Thus, if 16 Flexible processors were used, each with its own bulk memory, an effective processing rate of $4 * 10^4$ pixels per second can be obtained.

4.3 Conclusions

Two methods of calculating a Gaussian maximum likelihood classifier have been discussed. The timings for both algorithms have been discussed. The first method presented requires the host to do much data collection, while the second does not. It was shown that the second method provides high utilization of the Flexible Processors. The actual micro-assembly language program for the second method is presented in Appendix 5.

In the next Chapter, the way in which a parallel processing system such as the Flexible Processor array can be used to perform contextual classification is examined.

5. FLEXIBLE PROCESSOR IMPLEMENTATION OF A CONTEXTUAL CLASSIFICATION ALGORITHM

5.1 Introduction

This chapter explores the actual implementation of a contextual classifier. Section 5.2 briefly describes the contextual classifier approach. Section 5.3 gives serial algorithms for performing it. Section 5.4 presents a Flexible Processor program to implement the contextual classification algorithm with a simple size three neighborhood and an image size such that the number of rows is a multiple of the number of Flexible Processors in the system. The use of the Flexible Processor system to implement a general contextual classifier is explored in Section 5.5.

5.2 The Contextual Classifier

The image data to be classified are assumed to be a two-dimensional I-by-J array of multivariate pixels. Associated with the pixel at "row i" and "column j" is the multivariate measurement n- vector $X_{ij} \in R^n$ and the true class of the pixel $\theta_{ij} \in \Omega = \{\omega_1,\ldots,\omega_c\}$. The measurements have class-conditional densities $f(X|\omega_k)$, $k = 1,2,\ldots,c$, and are assumed to be class-conditionally independent. The objective is to classify the pixels in the array.

In order to incorporate contextual information into the classification process, when each pixel is to be classified, p-1 of its neighbors are also examined. This neighborhood, including the pixel to be classified, will be referred to as the p-array. Intuitively, to classify each pixel, the con-

textual classifier computes the probability of the given observed pixel being in class k by also considering the measurement vectors (values) observed for the neighbor pixels in the p-array. Specifically, for each pixel, for each class in Ω , a discriminant function g is calculated by summing the weighted probabilities of the p-1 neighbor pixels occurring in all possible classification states. This is described below mathematically for pixel (i,j) being in class ω_k. The description is followed by an example to clarify the notation used. Further details may be found in [10,11,13].

$$g_k(\underline{X}_{ij}) = \sum_{\underline{\theta}_{ij} \in \Omega^p, \theta_{ij}=\omega_k} \left[\prod_{\ell=1}^{p} f(X_\ell | \theta_\ell) \right] G^p(\underline{\theta}_{ij})$$

where

$X_\ell \in \underline{X}_{ij}$ is the measurement vector from the ℓth pixel in the p-array (for pixel (i,j))

$\theta_\ell \in \underline{\theta}_{ij}$ is the class of the ℓth pixel in the p-array (for pixel (i,j))

$f(X_\ell|\theta_\ell)$ is the class conditional density of X_ℓ given that the ℓth pixel is from class θ_ℓ

$G^p(\underline{\theta}_{ij}) = G^p(\theta_1, \theta_2, \ldots, \theta_p)$ is the a priori probability of observing the p-array $\theta_1, \theta_2, \ldots, \theta_p$.

Within the p-array, the pixel locations may be numbered in any convenient but fixed order. The joint probability distribution G^p is referred to as the context distribution.

To clarify the computation of the discriminant function, consider the following example. Let the context array (neighborhood) be the p=3 (two

nearest neighbors) choice shown in Figure 5.1 with the pixels numbered such that the pixel (i,j) to be classified is associated with X_1 and θ_1, and pixel (i,j-1) is associated with X_2 and θ_2, and pixel (i,j+1) is associated with X_3 and θ_3. Assume there are two possible classes: $\Omega=\{a,b\}$. Then the discriminant function for class b is explicitly

$$g_b(\underline{X}_{ij}) = \sum_{\underline{\theta}_{ij}\in\Omega^3,\theta_1=b} \left[\prod_{\ell=1}^{3} f(X_\ell|\theta_\ell)\right] G^3(\underline{\theta}_{ij})$$

$$= f(X_1|b)f(X_2|a)f(X_3|a)G(b,a,a)$$

$$+ f(X_1|b)f(X_2|a)f(X_3|b)G(b,a,b)$$

$$+ f(X_1|b)f(X_2|b)f(X_3|a)G(b,b,a)$$

$$+ f(X_1|b)f(X_2|b)f(X_3|b)G(b,b,b)$$

Note that $G^3(\underline{\theta}_{ij})=G(\theta_1,\theta_2,\theta_3)$ is the relative frequency of occurrence in the scene of the specific neighborhood configuration $(\theta_1,\theta_2,\theta_3)$. After computing the discriminant functions g_a and g_b for pixel (i,j), pixel (i,j) is assigned to the class which has the largest discriminant function value.

5.3 Serial Implementation of a Contextual Classifier

Algorithm 1, shown in Figure 5.2, is one way to implement the contextual classifier. The particular classifier considered here is a horizontally linear p-array of size three. This is shown in Figure 5.1.

First consider the main loop. Let the original image to be classified be an I-by-J array called A. Columns 0 and J-1, the two side edges of the image, are not classified since these pixels will not have both left and right neighbors. The variable "value" will contain the maximum "g"

Fig. 5.1. Horizontally Linear Neighborhoods.

```
Main Loop
for i = 0 to I-1 do /* row */
  begin
  for j = 1 to J-2 do /* column */
    begin              /* for each pixel */
    value = -1         /* max "g" */
    class = -1         /* class with max
                                    "g" */
    for k = 1 to C do /* for each class */
      begin
      current = g(i,j,k)
      if current > value
        then  value = current
              class = k
      end
    print Pixel (i,j) is classified as
          "class"
    end
  end
```

```
Discriminant Function Calculation
function          g(i,j,k)
sum = 0
for r = 1 to C do  /* all possible
                              classes */
  begin
  for q = 1 to C do  /* all possible
                              classes */
    begin
    sum = compf(i,j-1,r)*compf(i,j,k)
          *compf(i,j+1,q)*G(r,k,q)+sum
    end
  end
return (sum)
```

```
Class-Conditional Density Calculation
function compf(a,b,k) /* for pixel (a,b),
                              class k */
x = A(a,b)   /* x is pixel measurement
                              vector */
expo=-[log|Σ_k| + (x-m_k)^T Σ_k^-1 (x-m_k)] * .5
return (e^expo)
```

Fig. 5.2. Pidgeon Algol Implementation of the Contextual Classifier -- With Redundant Calculations.

(discriminant function) value calculated for pixel (i,j). This variable may be updated as the "g" for each class is calculated. The variable "class" is the class associated with "value." In the main loop, "g(i,j,k)" is a call to a function to calculate the discriminant function for pixel (i,j) and class k. This function is called I*(J-2)*C times, once for each class and for each pixel being classified.

Consider the calculation of g(i,j,k). The class of pixel (i,j) is held constant at k, while all other possible class assignments are considers for pixels (i,j-1) and (i,j+1). For each assignment of classes for the pixels neighboring pixel (i,j), of which there are C*C, the product of the class-conditional densities ("compf") is weighted by "G(r,k,q)," the a priori probability of observing the 3-array $(\omega_r,\omega_k,\omega_q)$. The "G" array is predetermined and prestored. For each call "g(i,j,k)," the value of "sum" for that i,j, and k is calculated. "Sum" is then returned as the value of "g(i,j,k)." In this straightforward version of the g(i,j,k) routine, the function to compute a class-conditional density ("compf") is called C*C times each time "g" is called.

Now consider the "compf" routine. This calculates the class-conditional density for pixel (a,b) and class k using the following equation:

$$f(x|k)=e^{-[\log|\Sigma_k|+ (x-m_k)^T\Sigma_k^{-1}(x-m_k)]/2}$$

where the measurement vector for each pixel is of size four, Σ_k^{-1} is the inverse covariance matrix for class k (four-by-four matrix), m_k is the mean vector for class k (size four vector), "T" indicates the transpose, and "log" is the natural logarithm. For each class, the algorithm uses $\log|\Sigma_k|$, Σ_k^{-1}, and m_k as precomputed constants. For each call "compf

(a,b,k)," the value of "e^{expo}" for that a,b and k is calculated. "e^{expo}" is then returned as the value of "compf (a,b,k)."

Algorithm 1 executes the "compf" subroutine ($I*(J-2)*C^3$) times. Since for each pixel there are C "f"s (class-conditional densities), this approach is inefficient by a factor of C^2. Algorithm 2 rectifies this problem by saving certain "f" values rather than recalculating them.

Algorithm 2, shown in Figure 5.3, implements the contextual classifier without the redundant executions of "compf" that occur in Algorithm 1. Let X, Y, and Z correspond to the pixels (i, j-1), (i, j), and (i, j+1), respectively, where (i, j) is the pixel to be classified. Each of X, Y, and Z is a vector of size C. Element t of X will contain the class-conditional density ("compf") for the current (i, j-1) pixel for class t. Y and Z are defined similarly. By using these three vectors to save the class-conditional densities, each density (for a given pixel and class) is calculated only once, instead of C^2 times.

The main loop of Algorithm 2 is modified to calculate the class-conditional densities for the first three columns each time a new row is considered (i.e., each time "i" is incremented). Each time a new pixel in a given row is to be classified (i.e., just before "j" is incremented), these values are updated. In particular, X gets the Y values, Y gets the Z values, and new values are calculated to update Z.

The new discriminant function calculation, g', does not call the subroutine "compf." It gets the values it needs from the X, Y, and Z arrays. For each call "g'(k)," the value of "sum" for that k is calculated. "Sum" is then returned as the value of "g'(k)."

The same "compf" routine is used for both Algorithms 1 and 2. Algorithm 1 calls this routine $I*(J-2)*C^3$ times, while Algorithm 2 calls it only

```
Main Loop
for i = 0 to I-1 do  /* row */
  begin
  for k = 1 to C do
    begin /* compute f's for 1st 3
                                columns */
    X(k) = compf (i,0,k)
    Y(k) = compf (i,1,k)
    Z(k) = compf (i,2,k)
    end
  for j = 1 to J-2 do /* column */
    begin        /* for each pixel */
    value = -1   /* max "g" */
    class = -1   /* class with max "g" */
    for k = 1 to C do
        begin
        current = g'(k)
        if current > value
          then value = current
               class = k
        end
    print Pixel (i,j) is classified as
                                "class"
    if j < J-2
        then /* update X,Y,Z arrays */
             for k = 1 to C do
             begin
             X(k) = Y(k)
             Y(k) = Z(k)
             Z(k) = compf (i,j+2,k)
             end
  end
end
```

```
Discriminant Function Calculation
  function  g'(k)
  sum = 0
  for r = 1 to C do  /* all possible
                              classes */
    begin
    for q = 1 to C do /* all possible
                              classes */
      begin
      sum = X(r) * Y(k) * Z(q)
            *G(r,k,q) + sum
      end
    end
  return (sum)
```

Fig. 5.3. Pidgeon Algol Implementation of the Contextual Classifier -- Without Redundant Calculations.

I*(J-2)*C times.

There are other techniques that can be employed to make Algorithm 2 even more efficient that have not been included in order to avoid obscuring the basic program flow. For example, whenever G(r,k,q) is zero, no multiplications are performed.

The serial complexity of Algorithm 2 can be calculated in terms of assignment statements, multiplications, additions, and "compf" calculations. To initialize X, Y, and Z for new rows, I*C*3 assignments and calls to "compf" occur. For each pixel, at most C+1 assignments to "value" and "class" occur, and C calls to "g'(k)" occur. In addition, for each row, the X, Y, and Z vectors are updated J-3 times, each update using 3*C assignments and C calls to "compf." Each execution of "g'(k)" uses $3*C^2$ multiplications, C^2 additions, and C^2+1 assignments. Thus, the total complexity for Algorithm 2 is:

$I(J(C^3+7C+2)-(2C^3+14C+4)$	assignments;
$3C^3I(J-2)$	multiplications;
$C^3I(J-2)$	additions; and
$I*J*C$	"compf" calculations.

The growth is proportional to $I*J*C^3$ assignments, multiplications, and additions, and I*J*C "compf" calculations.

In this section, a contextual classifier based on a horizontally linear neighborhood of size three has been analyzed. Algorithms for contextual classifiers using other size and shape neighborhoods would be analogous to the algorithms which were presented.

Algorithms 1 and 2 are written for conventional uniprocessor systems. Section 5.4 will examine how to implement Algorithm 2 on a CDC Flexible Pro-

cessor system.

5.4 Flexible Processor System Implementation of a Simple Contextual Classifier

Consider the implementation of a contextual classifier on an array of N Flexible Processors. Assume the neighborhood is horizontally linear as shown in Figure 5.1. Divide the A-by-B image into subimages of B/N rows A pixels long, as shown in Figure 4.1. Assign each subimage to a different Flexible Processor. The entire neighborhood of each pixel is included in its subimage. Each Flexible Processor can therefore execute the uniprocessor algorithm presented in Section 4.1 on its own subimage. No interaction between Flexible Processors is needed, i.e., each Flexible Processor can process its subimage independently.

The LARS Flexible Processor micro-assembler and simulator are being used to gather statistics on the execution time for the size three horizontally linear neighborhood contextual classifier. Due to the fact that each Flexible Processor is microprogrammable, determining program correctness and analyzing execution times is done through the use of the micro-assembler and simulator. The current implementation of the contextual classifier uses 780 microprogram instructions. Execution times per pixel vary because all floating point operations are done in the software. The classification time associated with the first pixel on a line is different than the classification of the rest of the pixels on the same line. This difference is accounted for by the three-pixel window. Data must be calculated for each of the pixels in the window for the first pixel on the line, while for the rest, data must be calculated for only one pixel. The format of the data words of the pixel measurement vectors, covariance matrices, etc., consists of a 14-bit two's complement exponent and a 17-bit sign-magnitude mantissa. The covariance

matrices, logarithms of the determinants of the covariance matrices, a priori probabilities (G^p), and the X, Y, and Z vectors are all stored in the large file. In this way, each Flexible Processor has all the information it needs for performing the classification on its subimage. The subimage data itself would be stored in a bulk memory. A multiple Flexible Processor configuration which associates one bulk memory with each Flexible Processor would be best for this application. For testing the Flexible Processor contextual classifier program, the classification of two rows of eight pixel measurement vectors (stored in the large file) using four classes was evaluated. The data was actual Landsat data, as was used in [12]. Evaluation of the serial Algorithm 2 from section 5.3 showed that a PDP-11/70 required .073 seconds per pixel, while a single Flexible Processor required .075 seconds per pixel. While, at first, it seems that the PDP-11/70 actually ran faster, lack of exponent range in the 11/70 floating point hardware yielded the incorect results due to rounding error. To overcome this error, by normalizing the data, it would require an extra .030 seconds per pixel, thus the Flexible Processor is over 25% faster. The floating point is implemented in software in the Flexible Processor and uses a 14-bit exponent to overcome this problem. These tests are by no means exhaustive. The simulator must run for many hours just to obtain a result for one pixel. Further testing is in progress.

Using .1 seconds per pixel as a rough approximation of the PDP processing time, and .08 seconds per pixel as a rough approximation of a single Flexible Processor processing time, a 16 Flexible Processor configuration, where each processor had its own bulk memory, would perform contextual classifications at a rate of 200 pixels per second as opposed to 10 pixels per second for a single PDP-11/70. As mentioned in section 5.3, additional pro-

gramming techniques that would increase this processing rate can be incorporated (this is currently in progress). Furthermore, as more experience in programming Flexible Processors is obtained, additional improvements in execution time can be expected.

5.5 Contextual Classification on a Flexible Processor System

Consider the implementation of a contextual classifier on an array of Flexible Processors as discussed in Section 5.4. Again, assume the neighborhood is horizontally linear, as shown in Figure 5.1 and the image is divided into subimages of B/N rows A pixels long, as shown in Figure 4.1. If B = kN, where k is a integer, there is 100% utilization of the Flexible Processors. Furthermore, there is no overhead for inter-Flexible Processor data transfers, since the entire neighborhood of each pixel is included in its subimage. Therefore, a factor of N improvement is attained.

If (A*B)/N is an integer, but B = kN + x, $0<x<N$, then Flexible Processors can be underutilized in order to keep neighborhoods within subimages, or Flexible Processors can be fully utilized, dividing neighborhoods between Flexible Processors, necessitating inter-Flexible Processor data transfers. This is shown for a simple example in Figure 5.4, where N=2, A=3, and B=4. In Figure 5.4(a) no inter-Flexible Processor transfers are needed, but Flexible Processor number 1 is not fully utilized. In Figure 5.4(b) both Flexible Processors are fully utilized, but due the horizontally linear neighborhood, at least pixel 11 will have to be sent to Flexible Processor number 1 and at least pixel 12 will have to be sent to Flexible Processor number 0.

If (A*B)/N is not a integer, some inter-Flexible Processor data transfers will be necessary. The number of transfers will be a function of the way in which the pixels are assigned to Flexible Processors, as in the previous paragraph. To determine the computationally fastest approach when-

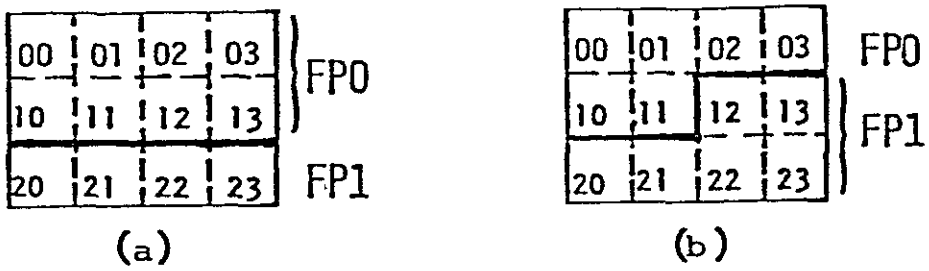


Fig. 5.4.

(a) Underutilization With No Inter FP Communication.

(b) Inter-FP Data Transfers Required -- Full Utilization.

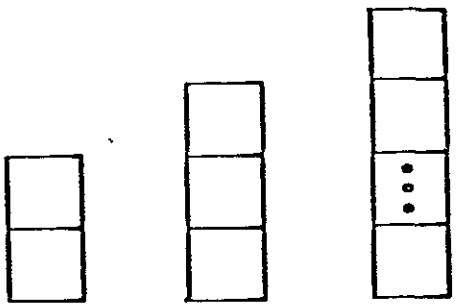


Fig. 5.5. Vertically Linear Neighborhoods.

ever B = kN + x, 0<x<N, requires knowledge of the actual image size, the actual number of Flexible Processors used, the exact time required to execute inter-Flexible Processor transfers, and the length of the neighborhood.

There are two other cases of linear neighborhoods. There are vertically linear and diagonally linear, as shown in Figures 5.5 and 5.6. The vertically linear case is just a 90° rotation of the horizontally linear case. The diagnonally linear case can be simplified to a 45° rotation of the horizontally linear case for B = kN by the proper assignment of pixels to Flexible Processors. Consider an A by B image, $A \leq B$ and $B = Nk$. Label the diagonals from 0 to A+B-2, as shown in Figure 5.7 for A=4 and B=6. The pixels can then be grouped into B sets of A pixels as follows:

1. for each i, $0 \leq i \leq A-1$, the pixels in diagonals i and i+B form a group of size A,

2. for each j, $A-1 \leq j \leq B-1$, the pixels in diagonal j form a group of size A.

Using these rules, each Flexible Processor is assigned k groups. Thus, the problem has been reduced to the equivalent of the horizontally linear case, which has already been discussed. The case for B = kN + x, 0<x<N, is even more complex than the analogous situation in the horizontally linear case and requires a detailed tradeoff analysis based on the actual image size, the actual number of Flexible Processors used, the exact time required to execute inter-Flexible Processor data transfers, and the length of the neighborhood.

Now consider nonlinear neighborhoods, that is neighborhoods which do not fit into one of the linear classes. For example, all of the neighborhoods in Figure 5.8 are nonlinear. Figure 5.8(a) and its rotations represent the simplest nonlinear neighborhood. It is included in all other

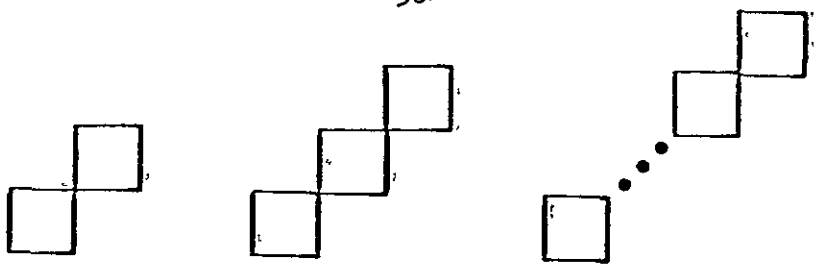


Fig. 5.6. Diagonally Linear Neighborhoods.

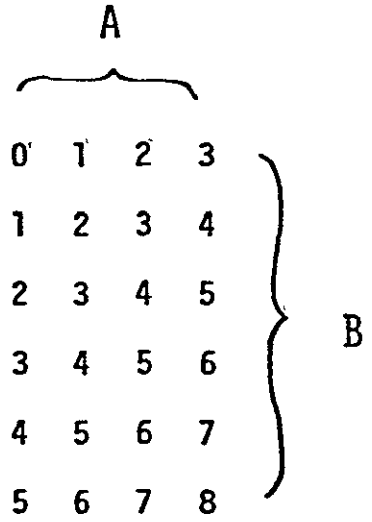


Fig. 5.7. The Diagonals of an A-by-B Image.

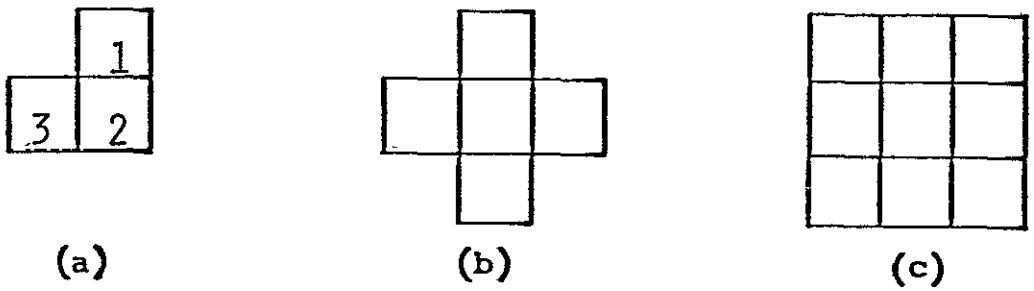


Fig. 5.8. Nonlinear Neighborhoods.

nonlinear neighborhoods. Thus, that neighborhood is called the nonlinear kernel neighborhood.

It can be shown that there is no way to partition an A-by-B image into N (not necessarily equal) sections such that a contextual classifier using a nonlinear neighborhood can be implemented without involving inter-Flexible Processor data transfers. This will be demonstrated for the nonlinear kernel and will thus be true for all nonlinear neighborhoods. There are three cases to consider. If there is a horizontal border between two subimages stored in different Flexible Processors, then pixels 1 and 2 in 5.8(a) will be in different Flexible Processors. If there is a vertical border, pixels 2 and 3 will be in different Flexible Processors. If there is a diagonal border, pixels 1 and 2 will be in different Flexible Processors. The way in which to assign Flexible Processors in order to minimize computation time will depend upon the particular image size, number of Flexible Processors, time required for inter-Flexible Processor communications and the shape and size of the neighborhood. These factors will also determine the effectiveness of the use of the Flexible Processor array for performing context classifications based on a given neighborhood.

The discussion of performing classifications with the Flexible Processor system demonstrates one way in which a multiple-processor system can be used to hasten the processing of image data. Future work involves programming the contextual classifier on the Flexible Processor simulator using different size and shape neighborhoods and determining the most efficient assignment of pixels to Flexible Processors for each case examined. The implementation of the classifier will provide hard data to verify the effectiveness of the parallel processing approach.

5.6 Conclusions

Algorithms for performing contextual classifications using a size three horizontally linear neighborhood was presented. Algorithm 1 was a straightforward approach. Algorithm 2 was a more efficient approach that avoided redundant calculations. The serial computational complexity of Algorithm 2 was shown to have a growth proportional to $I*J*C^3$ assignments, multiplications, and additions, and I*J*C "compf" calculations. The way in which N Flexible Processors could perform the classifications N times faster than a single Flexible Processor was explained.

In summary, contextual classifiers have been shown to be powerful remote sensing tools in other papers. Their main disadvantage is their computational complexity. This Chapter has demonstrated how parallel processing can be used to overcome this disadvantage.

6. CONCLUSIONS

The goal of the research in this project has been to implement a contextual classifier on a simulator of an array of CDC Flexible Processors. To achieve this end, the simpler Gaussian maximum likelihood classifier was first implemented. The maximum likelihood classifier program provided a vehicle for gaining experience in coding for a Flexible Processor and debugging the simulator. Computations performed by the maximum likelihood classifier are identical to many of the computations required for the contextual classifier, but the overall algorithm is considerably simpler. Thus implementing the maximum likelihood classifier provided a useful means for beginning to learn how to program a Flexible Processor system and for debugging the simulator.

The next major step was to implement the contextual classifier on the Flexible Processor simulator. As the program currently runs, it is approximately 25% faster on a single Flexible Processor system than it is on a PDP-11/70. After extensive testing, using 300 pixels from actual Landsat data, the following is a list of average timings for the Flexible Processor floating point algorithms used in the contextual classifier program (using a 14-bit exponent and 16-bit mantissa):

add:	9 microseconds
subtract:	9 microseconds
multiply:	2 microseconds
divide:	3 microseconds
compare:	4 microseconds

A Flexible Processor, operating on actual Landsat data, can perform a contextual classification using a size three horizontally linear neighborhood and four classes at a rate of approximately 75 milliseconds per pixel. Thus, a 16 Flexible Processor system would process approximately 215 pixels per second. When more experience programming the Flexible Processor has been gained, these times can most likely be improved.

It is important to realize that 60 to 90% of the processing time for the contextual classifier is spent in software implementations of floating point algorithms. Thus, the addition of floating point hardware (with the needed precision) would greatly increase the processing speed of the classifications.

Recall that a Flexible Processor is programmed in micro-assembly language, allowing parallelism at the instruction level. For example, it is possible to increment an index register conditionally, do a program jump, multiply two 8-bit integers, and add two 32-bit integers -- all simultaneously. This type of operational overlap, in conjunction with the multiprocessing capability of the Flexible Processors, greatly increases the speed of the Flexible Processor array.

The following list summarizes the important architectural features of an Flexible Processor:

User microprogrammable.

Dual 16-bit internal bus system.

Able to operate with either 16- or 32-bit words.

125 nsec clock cycle.

125 nsec time to add two 32-bit integers.

250 nsec time to multiply two 8-bit integers.

Register file (with 60 nsec access time) of over 8,000 16-bit words.

In order to debug, verify, and time Flexible Processor algorithms, a simulator for an array of up to 16 Flexible Processors has been developed. This simulator runs under the UNIX operating system on a PDP-11/70 series computer. An assembler for the micro-assembly language has also been developed.

The experience gained through the use of the simulator has made evident the following advantages and disadvantages of the system.

Advantages:

Multiple processors (up to 16).

User microprogrammable -- parallelism at the instruction level.

Connection ring for inter-Flexible Processor communications.

Shared bulk memory units.

Separate arithmetic logic unit and hardware multiply.

Disadvantages:

No floating-point hardware.

Micro-assembly language -- difficult to program.

Program memory limited to 4k micro-instructions.

Based on the investigations to date, the advantages of this system appear to outweigh the disadvantages. However, alternative approaches, such as multimicroprocessor systems, should also be considered to determine the most cost-effective approach for implementing the contextual classifier and other computationally demanding image processing operations for remote sensing.

Through the use of parallel, pipelined, and/or special purpose computer systems such as the CDC Flexible Processor system, the types of computations required for the contextual classifier and other computationally demanding

processes can be implemented efficiently. This will not only reduce the time required to do contextual classification, but will also allow the investigation of techniques which may otherwise be considered infeasible.

REFERENCES

[1] G. R. Allen, L. O. Bonrud, J. J. Cosgrove, and R. M. Stone, "The Design and Use of Special Purpose Processors for the Machine Processing of Remotely Sensed Data," Proceedings of the 1973 Conference on Machine Processing of Remotely Sensed Data, (IEEE Cat. No. 73 CH 0834-2GE), pp. 1A-25-1A-42, Oct. 1973.

[2] Control Data Corp., Cyber-Ikon Image Procesing System Design Concepts, Digital Systems Division, Control Data Corp., Minneapolis, MN, January 1977.

[3] Control Data Corp., Cyber-Ikon Flexible Processor Programming Textbook, Digital System Division, Control Data Corp., Minneapolis, MN, November 1977.

[4] K. S. Fu, "Special computer architectures for pattern recognition and image processing-an overview," in Proc. 1978 National Computer Conf., pp. 1003-1013, June 1978.

[5] J. L. Kast, P. H. Swain, and T. L. Phillips, The Feasibility of Using a Cyber-Ikon System as the Nucleus of an Experimental Agricultural Data Center, LARS Contract Report 021678, Laboratory for Applications of Remote Sensing (LARS), Purdue University, West Lafayette, IN, February 1978.

[6] Kernighan, B. W. and Ritchie, D. M., The C Programming Language, Prentive-Hall, Englewood Cliffs, NJ, 1978.

[7] Krause, K. W., "Use of the CDC Cyber-Ikon Simulator," unpublished report, School of Electrical Engineering, Purdue University, West Lafayette, In 47907, August 1978.

[8] H. J. Siegel, P. T. Mueller, Jr., and H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor system," Proceedings of the 1978 International Conference on Parallel Processing (IEEE Catalog No. 78 CH 1321-9C), pp. 9-17, August 1978.

[9] H. J. Siegel, L. J. Siegel, R. J. McMillen, P. T. Mueller, Jr., and S. D. Smith, "An SIMD/MIMD multimicroprocessor system for image processing and pattern recognition," Proceedings of the 1979 IEEE Computer Society Conference on Pattern Recognition and Image Processing (IEEE Catalog No. 79 CH 1428-2C), pp. 214-224, August 1979.

[10] P. H. Swain, H. J. Siegel, and B. W. Smith, "A method for classifying multispectral remote sensing data using context," Proceedings of the 1979 Machine Processing of Remotely Sensed Data Symposium (IEEE Catalog No. 79 CH 1430-8), pp. 343-353, June 1979.

[11] P. H. Swain, and P. E. Anuta, D. A. Landgrebe, and H. J. Siegel, Vol. III: Processing Techniques Development, Part 2: Data Preprocessing and Information Extraction Techniques, LARS Contract Report 113079, Laboratory for Applications of Remote Sensing (LARS), Purdue University, West Lafayette, IN, November 1979.

[12] P. H. Swain, S. B. Vardeman, and J. C. Tilton, Contextual Classification of Multispectral Image Data, LARS Technical Report 011080, Laboratory for Applications of Remote Sensing (LARS), Purdue University, West lafayette, IN, Jan. 1980.

[13] P. H. Swain, H. J. Siegel, and B. W. Smith, "Contextual classification of multispectral remote sensing data using a multiprocessor system," IEEE Transactions on Geoscience and Remote Sensing, Vol. GE-18, No. 2, pp. 197-203, April 1980.

[14] J. C. Tilton, P. H. Swain, and S. B. Vardeman, "Context distribution estimation for contextual classification of multispectral image data," Proceedings of the 1980 Machine Processing of Remotely Sensed Data Symposium (IEEE Catalog No. 80 Ch 1533-9), pp. 19-29, June 1980.

APPENDIX 1

FLEXIBLE PROCESSOR SYSTEM SIMULATOR DISPLAYS

A. Simulator Output Display

B. Simulator Display of Temporary File

C. Simulator Display of Large File

A. SIMULATOR OUTPUT DISPLAY

```
CYCLES:      MAR:          MIR0: MIR1: MIR2:          tr unn nop np pupu
-0000  05f3   016d  0100   0100  0005    ac0  mult brg0 nop nop

IDX0: IDX1: IDX2: IDX3:          ICR0: ICR1: ICR2: ICR3:          E1:  E0:
0003  0004   0004  0000          0003  0004  0004  0004           0000 1f2a

CMR0: CMR1: CMR2: CMR3:          MCR0: MCR1: MCR2: MCR3:          G1:   G0:
0010  0000   0000  0010          0000  0000  0000  9c40           0000 a5ce

BRG1: BRG0:      IMR0: IMR1:     INTR:   MARC:                    F1:  F0:
0000  27d9       0000  0000      0000    0000                     0011 a5ce

NOW:    PAST:    MMT0: MMT1: MMT2:       OUP0: OUP1:              P:   Q:
0014    0014     0000  0000  0000        0000  0000               a265 c500

IF0A:   IF1A:   LF0A:   LF1A:   TF0AR:  TF0AW:  TF1AR:  TF1AW:  MULT:
0000    0000    0010    0010    0002    0002    0002    0002    27d9

AINOUT: QRESIN: ARESIN: AROUT:  QINOUT: AQZIN:
0000    0000    0000    0000    0000    0000
0000    0000    0000    0000    0000    0000
0000    0000    0000    0000    0000    0000
0000    0000    0000    0000    0000    0000
```

B. SIMULATOR DISPLAY OF TEMPORARY FILE

```
temp[0] =        0000  0000
temp[1] =        0001  0001
temp[2] =        0006  d240
temp[3] =        0005  b640
temp[4] =        0006  9400
temp[5] =        0009  c500
temp[6] =        0000  0000
temp[7] =        0000  0000
temp[8] =        0000  0000
temp[9] =        0000  0000
temp[a] =        0000  0000
temp[b] =        0000  0000
temp[c] =        0000  0000
temp[d] =        0000  0000
temp[e] =        0000  0000
temp[f] =        0000  0000
```

C. SIMULATOR DISPLAY OF LARGE FILE

```
lf[0]  =  8fc0  0005  c000  00fd  b080  8002  c800  00fc
lf[4]  =  c000  00fd  9780  0006  9c80  0001  f880  8001
lf[8]  =  b080  8002  9c80  8001  a440  0004  b580  8000
lf[c]  =  c800  00fc  f880  8001  b580  8000  ff80  0003
lf[10] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[14] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[18] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[1c] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[20] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[24] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[28] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[2c] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[30] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[34] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[38] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[3c] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[40] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[44] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[48] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[4c] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[50] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[54] =  0000  0000  0000  0000  0000  0000  0000  0000
lf[58] =  0000  0000  0000  0000  0000  0000  0000  0000
```

APPENDIX 2

SIMULATOR FLOWCHARTS

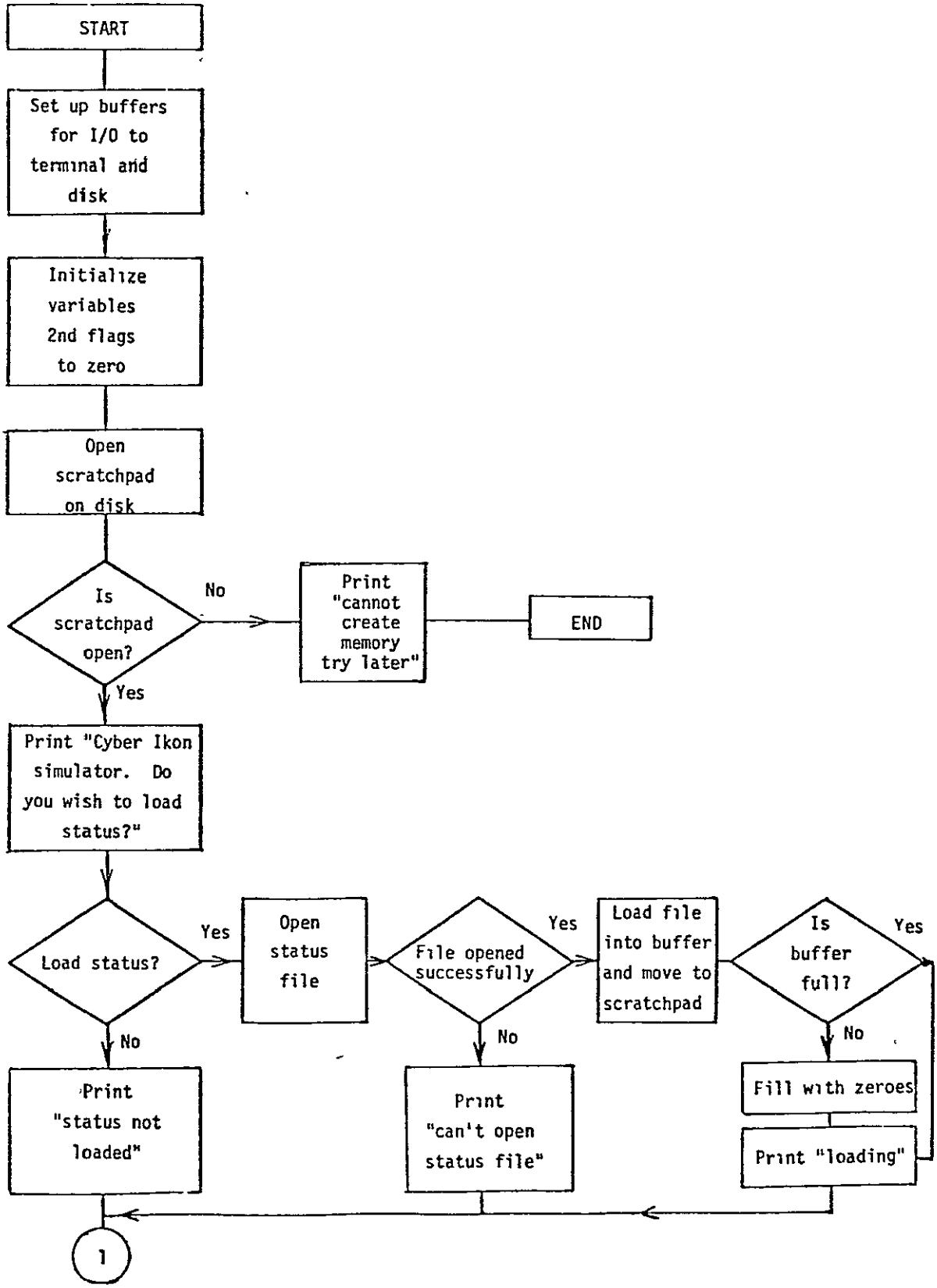
A. Setting Up Simulation

B. Input FP# and Operation to be Performed

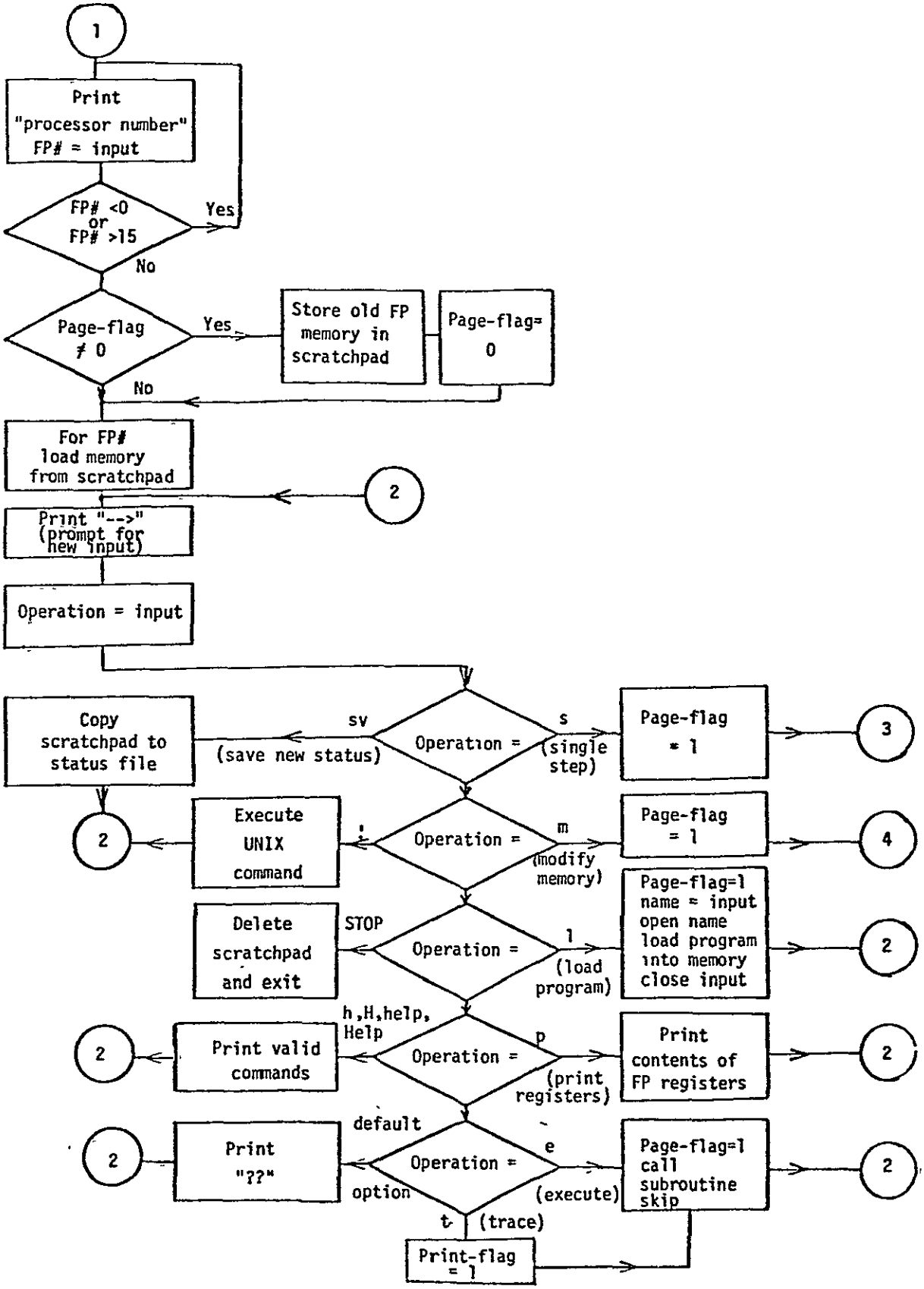
C. Read and Modify Register or Program Memory Content

D. Execute Single Execution Step

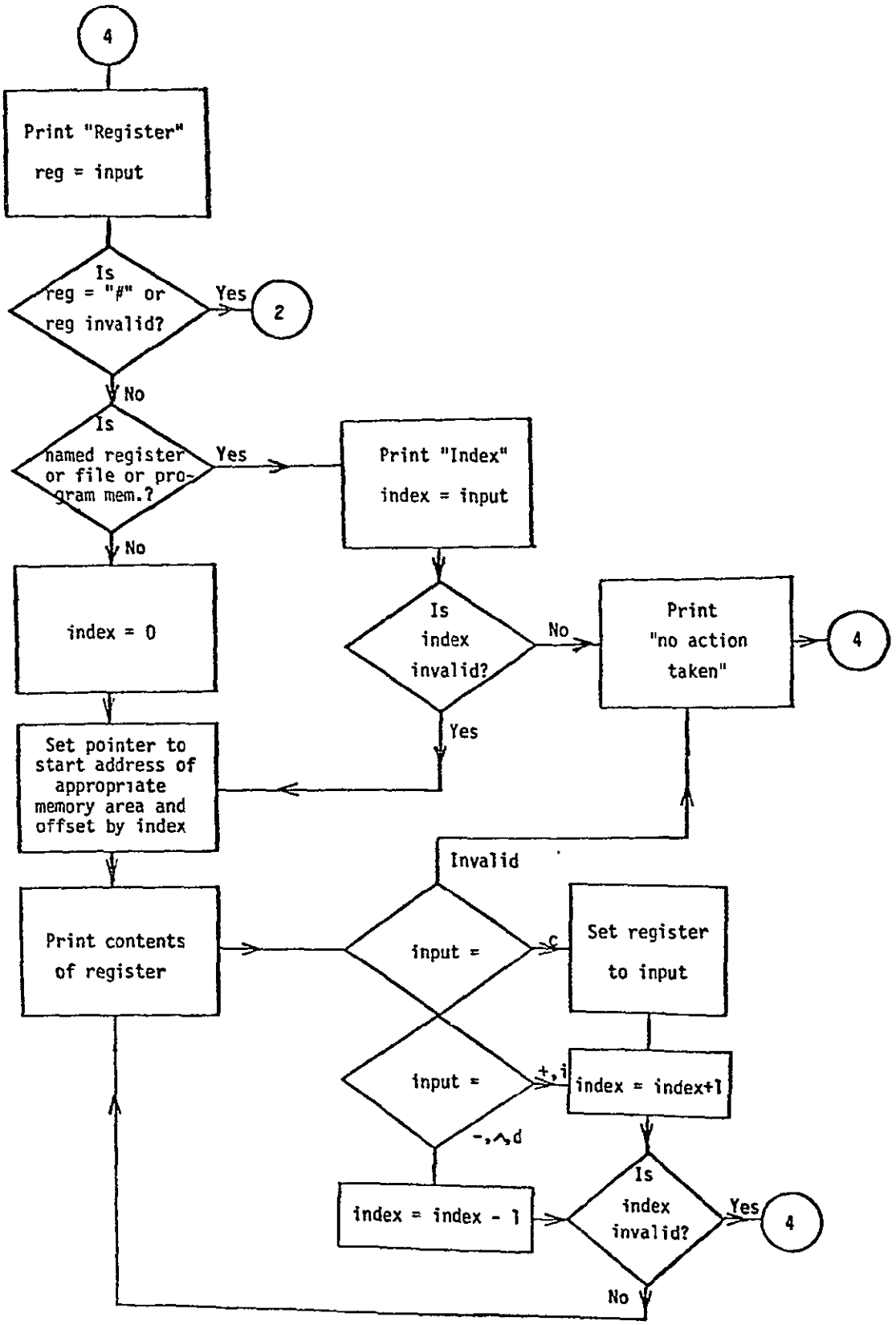
E. Subroutine "Exec" for Executing Single Instructions. Subroutine "Skip" for Executing Sequence of Instructions.



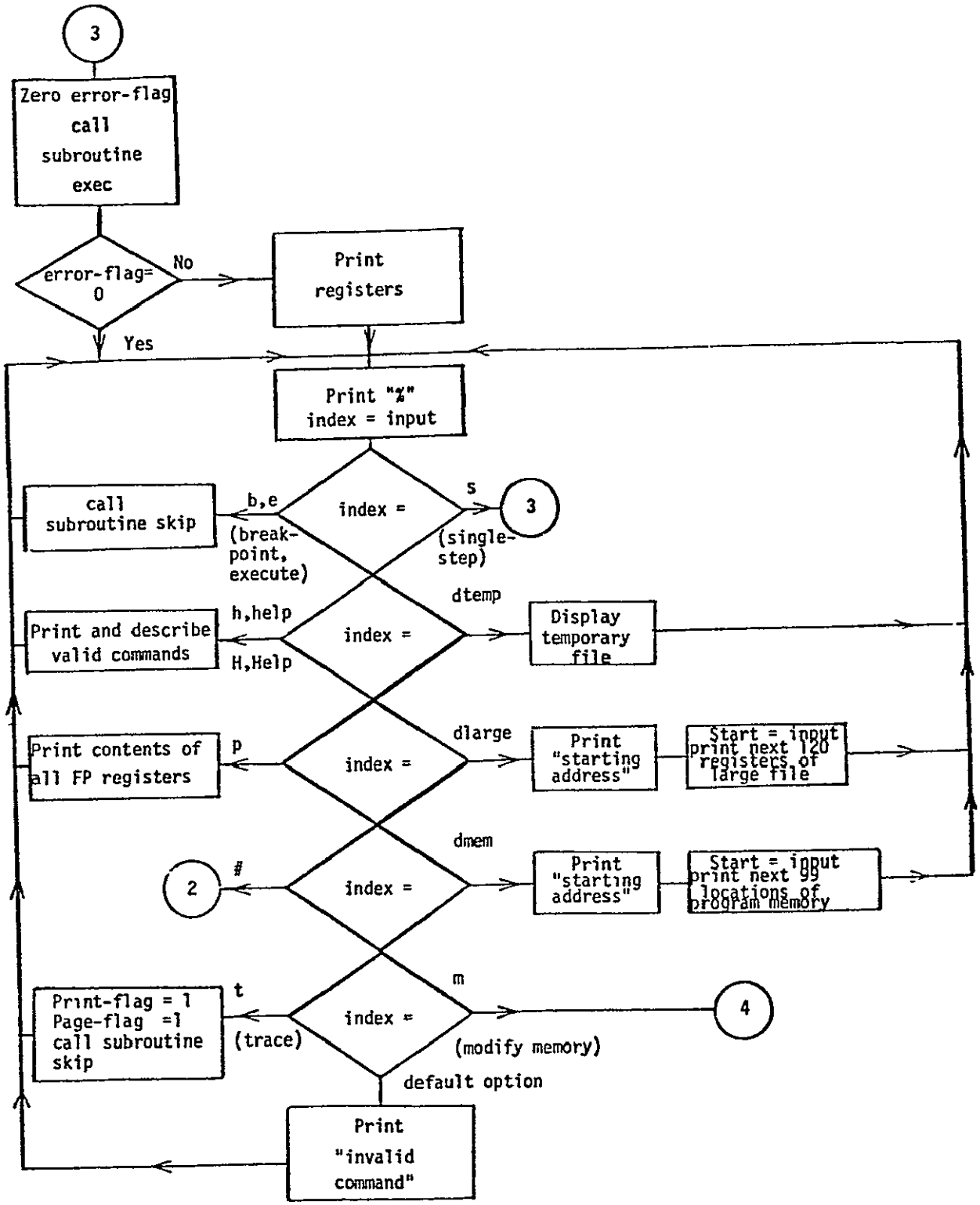
A. Setting Up Simulation.



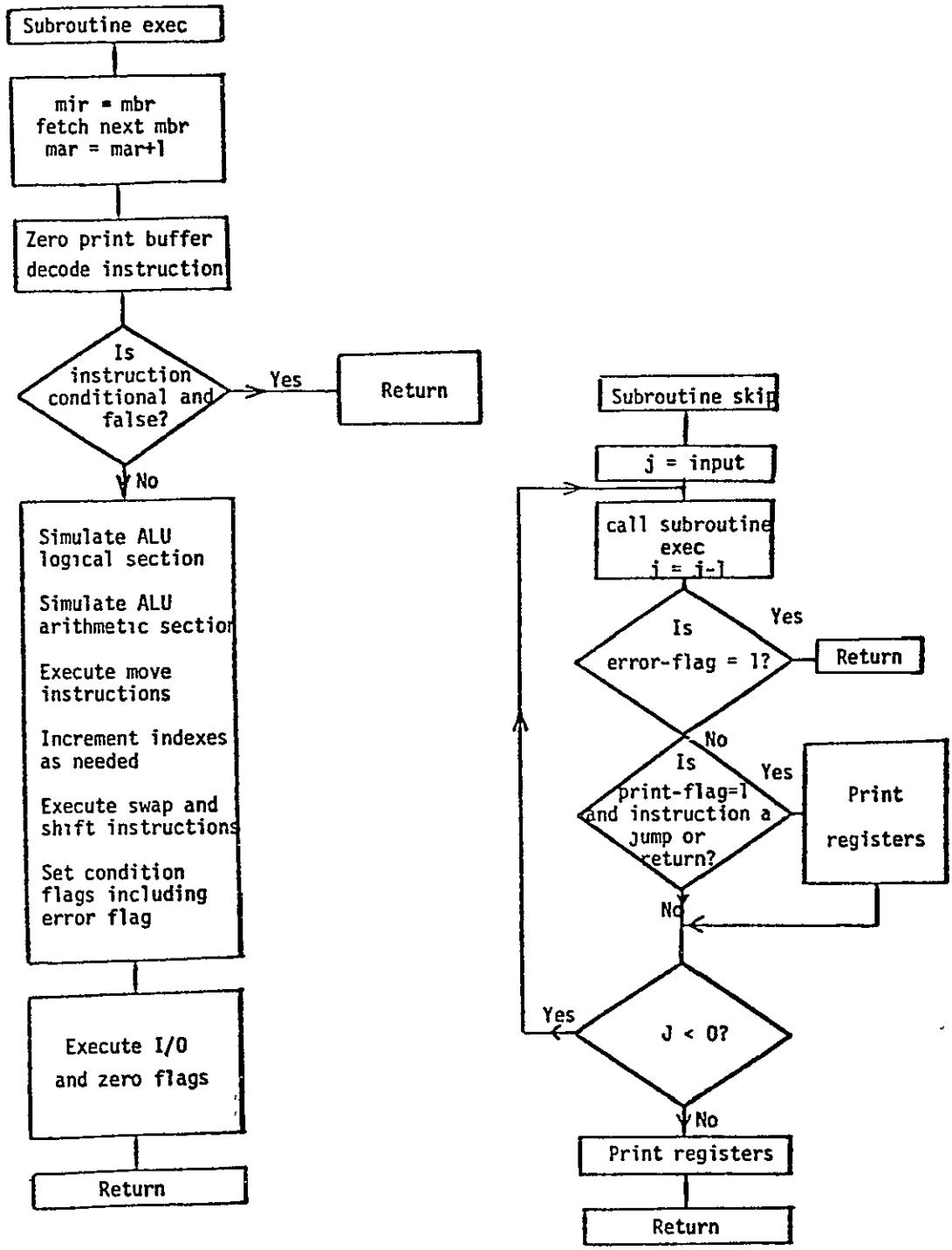
B. Input FP# and Operation to be Performed.



C. Read and Modify Register or Program Memory Content.



D. Execute Single Execution Step.



E. Subroutine EXEC for Executing Single Instructions. Subroutine SKIP for Executing Sequence of Instructions.

APPENDIX 3

SIMULATOR LISTING

```
/*
*
*          X    X    X   XXXX   X         XX            X    X
*          X    XX   X  X    X  X        X  X           X    X
*          X    X X  X  X       X        X  X           XXXXXX
*          X    X  X X  X       X        X  X           X    X
*          X    X   XX  X    X  X        X  X     XX    X    X
*          X    X    X   XXXX   XXXXXX    XX      XX    X    X
*
 Set up FILE pointers for use with new library */

        FILE    *statptr,*inptr,*otptr;
        FILE    *b0ptr,*b1ptr,*b2ptr,*b3ptr,*bufptr;

/* Global int: carry in; double add parameters; opcode; condition
*  flag; number of processors in use;   */

        int     car,par0,par1,par2,par3,op,conflg,lim;

/*  b0-b3 eight bit ALU output chunks ( As defined in textbook )
*   upper sign extended (ALU); lower sign extended (ALU);
*   source values; destination values; error flag; multiply fl..ag   */

        int     b0,b1,b2,b3,us,ls,s1,s0,d1,d0,erflg,merflg;
        int     ffff {0177777};
/*
* Command at * level table for use in lookup routine
*/
                char    *comtab[] {
                "s","m","l",
                "t","v","b","!","stop","#","p","h","help","e","r",
                                -1 };

/* Subcommand lookup table  */
        char    *verbtab[]   {
        "s", "dtemp", "dmem", "dlarg", "m", "b", "p", "h", "help", "e", "#
        "r",  -1  };

/*  Character variables for outputing decoded instruction  */

        char    *f01,f2[5],*f3,*f4,*f5,f6[5],*f7a,*f7,*f7b,
                *f8a,*f8,*f8b,*f9a,*f9,*f9b,*f10,*f11,*f12,*f13,
                        *f14,this[10];

/* bias- register index   input- Hex input form inhex routine  */

                int     bias,input;
/*
```

```
*       Tables for bus decoding for use in table lookup
*/
                char    *srctab[] {
                        "nop",
                        "a0sw",
                        "a0rs",
                        "a0rz",
                        "a0",
                        "a1sw",
                        "a1rs",
                        "a1rz",
                        "a1",
/**/
                        "qrsp",
                        "arsp",
                        "z0",
                        "z1",
                        "z2",
                        "z3",
/**/
                        "bsr0",
                        "bsr1",
                        "f0",
                        "f1",
                        "if0n",
                        "if0u",
                        "if0d",
                        "if0c",
                        "if1n",
                        "if1u",
                        "if1d",
                        "if1c",
                        "imr0",
                        "imr1",
                        "intr",
                        "l0ad",
                        "lf0n",
                        "lf0u",
                        "lf0d",
                        "lf0a",
/**/
                        "l1ad",
                        "lf1n",
                        "lf1u",
                        "lf1d",
                        "lf1a",
                        "mar",
                        "mcr0",
                        "mult",
                        "tf0n",
                        "tf0u",
                        "tf0d",
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
                        "tf0c",
                        "t0ra",
                        "t0wa",
                        "tf1n",
                        "tf1u",
                        "tf1d",
                        "tf1c",
                        "t1ra",
                        "t1wa",
                        -1 };
/*
*       Bus 0 integers
*/
                int     sr0int[] {
        00              /* 0 */,
        0100            /* 40 */,
        02100           /* 440 */,
        04100           /* 840 */,
        06100           /* c40 */,
        0200            /* 80 */,
        02200           /* 480 */,
        04200           /* 880 */,
        06200           /* c80 */,
        0600            /* 180 */,
        02600           /* 580 */,
        0500            /* 140 */,
        02500           /* 540 */,
        04500           /* 940 */,
        06500           /* d40 */,
        01700           /* 3c0 */,
        03700           /* 7c0 */,
        0300            /* c0 */,
        02300           /* 4c0 */,
        01000           /* 200 */,
        03000           /* 600 */,
        05000           /* a00 */,
        07000           /* e00 */,
        01100           /* 240 */,
        03100           /* 640 */,
        05100           /* a40 */,
        07100           /* e40 */,
        01600           /* 380 */,
        03600           /* 780 */,
        07600           /* f80 */,
        0               /* ffff */,
        01400           /* 300 */,
        03400           /* 700 */,
        05400           /* b00 */,
        07400           /* f00 */,
        06700           /* dc0 */,
        01500           /* 340 */,
        03500           /* 740 */,
```

```
        05500           /* b40 */,
        07500           /* f40 */,
        0700            /* 1c0 */,
        05700           /* bc0 */,
        0400            /* 100 */,
        01200           /* 280 */,
        03200           /* 680 */,
        05200           /* a80 */,
        07200           /* e80 */,
        02700           /* 5c0 */,
        04700           /* 9c0 */,
        01300           /* 2c0 */,
        03300           /* 6c0 */,
        05300           /* ac0 */,
        07300           /* ec0 */,
        02400           /* 500 */,
        04400           /* 900 */,
        0500            /* 140 */,
        02500           /* 540 */,
        04500           /* 940 */,
        06500,          /* d40 */              -1 };

/*  For some registers, they are loaded by looking up there
*   index (ie reg[index]) in this table ( See srsw default )  */
            srcint[]        {
        0,1,2,3,4,5,6,7,8,9,
        10,20,21,22,23,24,25,32,33,19,
        20,21,22,23,24,25,26,45,46,44,
        49,31,32,33,34,50,36,37,38,39,
        51,57,8329,43,44,45,46,82,83,49,50,
        51,52,100,101,102,103,104,105,-1};
/*
*       Bus one integers
*/
            int     sr1int[] {
        00              /* 0 */,
        01              /* 1 */,
        021             /* 11 */,
        041             /* 21 */,
        061             /* 31 */,
        02              /* 2 */,
        022             /* 12 */,
        042             /* 22 */,
        062             /* 32 */,
        06              /* 6 */,
        026             /* 16 */,
        05              /* 5 */,
        025             /* 15 */,
        045             /* 25 */,
        065             /* 35 */,
        017             /* f */,
        037             /* 1f */,
```

```
        03              /* 3 */,
        023             /* 13 */,
        010             /* 8 */,
        030             /* 18 */,
        050             /* 28 */,
        070             /* 38 */,
        011             /* 9 */,
        031             /* 19 */,
        051             /* 29 */,
        071             /* 39 */,
        016             /* e */,
        036             /* 1e */,
        076             /* 3e */,
        067             /* 37 */,
        014             /* c */,
        034             /* 1c */,
        054             /* 2c */,
        074             /* 3c */,
        0               /* ffff */,
        015             /* d */,
        035             /* 1d */,
        055             /* 2d */,
        075             /* 3d */,
        0               /* ffff */,
        057             /* 2f */,
        04              /* 4 */,
        012             /* a */,
        032             /* 1a */,
        052             /* 2a */,
        072             /* 3a */,
        027             /* 17 */,
        047             /* 27 */,
        013             /* b */,
        033             /* 1b */,
        053             /* 2b */,
        073             /* 3b */,
        024             /* 14 */,
        044             /* 24 */,
        05              /* 5 */,
        025             /* 15 */,
        045             /* 25 */,
        065             /* 35 */,-1        };
/*
*       Destination codes bus 0
*/
                char    *d0tab[] {
                        "nop",
                        "brg0",
                        "e0",
                        "e0f0",
                        "e0g1",
                        "ef0g",
```

```
                        "f0",
                        "f0g1",
                        "g1",
                        "imr0",
                        "imr1",
                        "intr",
                        "lf0n",
                        "lf0u",
                        "lf0d",
                        "lf0a",
                        "l0ad",
                        "mar",
                        "marc",
                        "mmi0",
                        "mmi1",
                        "mmi2",
                        "p",
                        "pe0",
                        "pe0g",
                        "pef0",
                        "pefg",
                        "pf0",
                        "pf0g",
                        "pg1",
                        "tf0n",
                        "tf0u",
                        "tf0d",
                        "tf0c",
                        "t0ra",
                        "t0wa",
                        "t0ba",
                        "if0a",
                        -1      };
/*
*       Referance table for bus 0 destinations
*/
                int     db0int[]        {
        0,24,30,3,4,5,32,7,35,45,
        46,44,12,13,14,15,49,51,52,54,
        55,56,64,23,24,25,26,27,28,29,
        30,31,32,33,82,83,36,47,-1      };
/**/
                int     d0int[] {
        00              /* 0 */,
        0400            /* 100 */,
        04200           /* 880 */,
        024200          /* 2880 */,
        014200          /* 1880 */,
        034200          /* 3880 */,
        020200          /* 2080 */,
        030200          /* 3080 */,
        010200          /* 1080 */,
```

```
        023200          /* 2680 */,
        027200          /* 2e80 */,
        037200          /* 3e80 */,
        03000           /* 600 */,
        07000           /* e00 */,
        013000          /* 1600 */,
        017000          /* 1e00 */,
        027000          /* 2e00 */,
        035600          /* 3b80 */,
        03400           /* 700 */,
        021600          /* 2380 */,
        025600          /* 2b80 */,
        031600          /* 3380 */,
        01000           /* 200 */,
        05000           /* a00 */,
        015000          /* 1a00 */,
        025000          /* 2a00 */,
        035000          /* 3a00 */,
        021000          /* 2200 */,
        031000          /* 3200 */,
        011000          /* 1200 */,
        02400           /* 500 */,
        06400           /* d00 */,
        012400          /* 1500 */,
        016400          /* 1d00 */,
        026400          /* 2d00 */,
        032400          /* 3500 */,
        036400          /* 3d00 */,
        026000          /* 2c00 */, -1          };
/*
*       Bus one destination codes
*/
                char    *d1tab[] {
                        "nop",
                        "arsp",
                        "brg1",
                        "cmr0",
                        "cmr1",
                        "cmr2",
                        "cmr3",
                        "e1",
                        "e1f1",
                        "ef1g",
                        "e1g0",
                        "f1",
                        "f1g0",
                        "g0",
                        "icr0",
                        "icr1",
                        "icr2",
                        "icr3",
                        "idx0",
```

```
                        "idx1",
                        "idx2",
                        "idx3",
                        "if1a",
                        "lf1n",
                        "lf1u",
                        "lf1d",
                        "lf1a",
                        "l1ad",
                        "mcr0",
                        "mcr1",
                        "mcr2",
                        "mcr3",
                        "mm0",
                        "mm1",
                        "mm01",
                        "mm2",
                        "mm02",
                        "mm12",
                        "mma",
                        "q",
                        "qef1",
                        "qefg",
                        "qe1",
                        "qe1g",
                        "qf1",
                        "qf1g",
                        "qg0",
                        "tf1n",
                        "tf1u",
                        "tf1d",
                        "tf1c",
                        "t1ra",
                        "t1wa",
                        "t1ba",
                        -1      };
/*      Bus 1 register code table
*/
                int     db1int[]        {
        0,1,25,26,27,28,29,31,8,9,
        10,33,12,34,40,41,42,43,36,37,
        38,39,48,23,24,25,26,50,57,58,
        59,60,32,33,34,35,36,37,38,65,
        40,41,42,43,44,45,46,47,48,49,
        50,100,101,53,-1        };
/**/
                int     d1int[] {
        00              /* 0 */,
        0166            /* 76 */,
        02              /* 2 */,
        015             /* d */,
        035             /* 1d */,
```

```
        055             /* 2d */,
        075             /* 3d */,
        021             /* 11 */,
        0121            /* 51 */,
        0161            /* 71 */,
        061             /* 31 */,
        0101            /* 41 */,
        0141            /* 61 */,
        041             /* 21 */,
        0117            /* 4f */,
        0137            /* 5f */,
        0157            /* 6f */,
        0177            /* 7f */,
        017             /* f */,
        037             /* 1f */,
        057             /* 2f */,
        077             /* 3f */,
        0130            /* 58 */,
        014             /* c */,
        034             /* 1c */,
        054             /* 2c */,
        074             /* 3c */,
        0134            /* 5c */,
        0116            /* 4e */,
        0136            /* 5e */,
        0156            /* 6e */,
        0176            /* 7e */,
        027             /* 17 */,
        047             /* 27 */,
        067             /* 37 */,
        0107            /* 47 */,
        0127            /* 57 */,
        0147            /* 67 */,
        0167            /* 77 */,
        04              /* 4 */,
        0124            /* 54 */,
        0164            /* 74 */,
        024             /* 14 */,
        064             /* 34 */,
        0104            /* 44 */,
        0144            /* 64 */,
        044             /* 24 */,
        012             /* a */,
        032             /* 1a */,
        052             /* 2a */,
        072             /* 3a */,
        0132            /* 5a */,
        0152            /* 6a */,
        0172            /* 7a */, -1       };
/**/
/**/
                int     ainout  {0};
```

```
                int     qresin  {4};
                int     aresin  {8};
                int     aresout {12};
                int     qinout  {16};
                int     aqzin   {20};
                int     brg0    {24};
/**/
                int     brg1    {25};
                int     cmr0    {26};
                int     cmr1    {27};
                int     cmr2    {28};
                int     cmr3    {29};
                int     e0      {30};
                int     e1      {31};
/**/
                int     f0      {32};
                int     f1      {33};
                int     g0      {34};
                int     g1      {35};
                int     indx0   {36};
                int     indx1   {37};
                int     indx2   {38};
/**/
                int     indx3   {39};
                int     icr0    {40};
                int     icr1    {41};
                int     icr2    {42};
                int     icr3    {43};
                int     intr    {44};
                int     imr0    {45};
/**/
                int     imr1    {46};
                int     if0a    {47};
                int     if1a    {48};
                int     lf0a    {49};
                int     lf1a    {50};
                int     mar     {51};
                int     marc    {52};
/**/
                int     now     {53};
                int     mmi0    {54};
                int     mmi1    {55};
                int     mmi2    {56};
                int     mcr0    {57};
                int     mcr1    {58};
                int     mcr2    {59};
/**/
                int     mcr3    {60};
                int     oupu    {61};
                int     past    {63};
                int     p       {64};
                int     q       {65};
```

```
                int     tf0     {66};
                int     tf0ar   {82};
/**/
                int     tf0aw   {83};
                int     tf1     {84};
                int     tf1ar   {100};
                int     tf1aw   {101};
                int     spare   {102};
                int     lf0     {106};
/* the size of the large files has been changed to 4k to simulate the real
/* system more closely. 4096 locations allocated
                int     lf1     {4202};         /* was 1131 with 1k */
/**/
                int     mir     {8298};         /* was 2156         */
                int     mbr     {8301};         /* was 2159             */
                int     res0    {8304};         /* was 2162             */
                int     res1    {8305};         /* was 2163             */
                int     bus0    {8306};         /* was 2164             */
                int     bus1    {8307};         /* was 2165             */
                int     stack   {8308};         /* was 2166             */
/**/
                int     ovl     {8324};         /* was 2182             */
                int     ovh     {8325};         /* was 2183             */
                int     shcon   {8326};         /* was 2184             */
                int     mulflg  {8327};         /* was 2185             */
                int     mult    {8328};         /* was 2186             */
                int     mem     {8329};         /* was 2187             */
                int     stptr   {11525};        /* was 4983             */
/**/
                int     cych    {11526};        /* was 4984             */
                int     cycl    {11527};        /* was 4985             */
                int     cycsh   {11528};        /* was 4986             */
                int     s       {11532};        /* was 4990             */
                int     b       {11536};        /* was 4994             */
                int     cycsl   {11540};        /* was 4998             */
/**/
                char    *regtab[] {
        "ainout", "qresin", "aresin", "aresout", "qinout", "aqzin", "brg0",
        "brg1", "cmr0", "cmr1", "cmr2", "cmr3", "e0", "e1",
        "f0", "f1", "g0", "g1", "indx0", "indx1", "indx2",
        "indx3", "icr0", "icr1", "icr2", "icr3", "intr", "imr0",
        "imr1", "if0a", "if1a", "lf0a", "lf1a", "mar", "marc",
        "now", "mmi0", "mmi1", "mmi2", "mcr0", "mcr1", "mcr2",
        "mcr3", "oupu", "past", "p", "q", "tf0", "tf0ar",
        "tf0aw", "tf1", "tf1ar", "tf1aw", "spare", "lf0", "lf1",
        "mir", "mbr", "res0", "res1", "bus0", "bus1", "stack",
        "ovl", "ovh", "shcon", "mulflg", "mult", "mem", "stptr",
        "cych", "cycl", "cycsh", "s", "b", "cycsl", "#",
                        -1      };
/**/
                int     regint[]  {
        0, 4, 8, 12, 16, 20, 24,
```

```
        25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38,
        39, 40, 41, 42, 43, 44, 45,
        46, 47, 48, 49, 50, 51, 52,
        53, 54, 55, 56, 57, 58, 59,
        60, 61, 63, 64, 65, 66, 82,
        83, 84, 100, 101, 102, 106, 4202,
        8298, 8301, 8304, 8305, 8306, 8307, 8308,
        8324, 8325, 8326, 8327, 8328, 8329, 11525,
        11526, 11527, 11528, 11532, 11536, 11540, -1          };
/**/
        struct regs  {int       dum[12000]; } r[1];
int printflag;
/* See incl06.h ( Initialized version of this ) for comments  */

        FILE    *statptr,*inptr,*otptr;
        FILE    *b0ptr,*b1ptr,*b2ptr,*b3ptr,*bufptr;
        int     car,par0,par1,par2,par3,op,conflg,lim;
        int     b0,b1,b2,b3,us,ls,s1,s0,d1,d0,erflg,merflg;
        int     ffff ;
                char    *comtab[] ;
                char    *verbtab[]   ;

                int     bias,input;

                char    *f01,f2[5],*f3,*f4,*f5,f6[5],*f7a,*f7,*f7b,
                *f8a,*f8,*f8b,*f9a,*f9,*f9b,*f10,*f11,*f12,*f13,
                *f14,this[10];

                char    *srctab[] ;
                int     sr0int[] ;
                int     srcint[];
                int     sr1int[] ;
                char    *d0tab[] ;
                int     db0int[];
                int     d0int[] ;
                char    *d1tab[] ;
                int     regint[];
                char    *regtab[];
                int     db1int[];
                int     d1int[] ;
                char    *regtab[] ;
                int     regint[]  ;
                int     ainout;
                int     qresin;
                int     aresin;
                int     aresout;
                int     qinout;
                int     aqzin;
                int     brg0;
/**/
                int     brg1;
```

```
                int     cmr0;
                int     cmr1;
                int     cmr2;
                int     cmr3;
                int     e0;
                int     e1;
/**/
                int     f0;
                int     f1;
                int     g0;
                int     g1;
                int     indx0;
                int     indx1;
                int     indx2;
/**/
                int     indx3;
                int     icr0;
                int     icr1;
                int     icr2;
                int     icr3;
                int     intr;
                int     imr0;
/**/
                int     imr1;
                int     if0a;
                int     if1a;
                int     lf0a;
                int     lf1a;
                int     mar;
                int     marc;
/**/
                int     now;
                int     mmi0;
                int     mmi1;
                int     mmi2;
                int     mcr0;
                int     mcr1;
                int     mcr2;
/**/
                int     mcr3;
                int     oupu;
                int     past;
                int     p;
                int     q;
                int     tf0;
                int     tf0ar;
/**/
                int     tf0aw;
                int     tf1;
                int     tf1ar;
                int     tf1aw;
                int     spare;
```

```
                int     lf0;
                int     lf1;
/**/
                int     mir;
                int     mbr;
                int     res0;
                int     res1;
                int     bus0;
                int     bus1;
                int     stack;
/**/
                int     ovl;
                int     ovh;
                int     shcon;
                int     mulflg;
                int     mult;
                int     mem;
                int     stptr;
/**/
                int     cych;
                int     cycl;
                int     cycsh;
                int     s;
                int     b;
                int     cycsl;
        struct regs  {int       dum[12000]; } r[1];
int printflag;
/*
*
*       X    X   XXXX   X    X    X     X         XXXX
*       XX  XX  X    X  XX   X    X     X        X    X
*       X XX X  X    X  X X  X    X     X        X
*       X    X  X    X  X  X X    XXXXXXX        X
*       X    X  X    X  X   XX    X     X   XX   X    X
*       X    X   XXXX   X    X    X     X   XX    XXXX
*
*  SIMULATOR   SYSTEM  MONITOR MOD 12  FOR VERSION 7 redone by BWS 03/01/80.
*  CRASHPROOFED on 01/21/80 by BWS. UNNECESSARY FUNCTION CALLS REPLACED WITH
*  GOTOs. THIS ALSO WILL SPEED EXECUTION AND SHORTEN PROGRAM LENGTH.
**********************************************************************
*
* This must be compiled as 'cc monh.c exech._ shioh._ helph._ -lS -O '
* where exec and shio can be .o or .c
*   The follwing includes include the standard I/O library,
*  and two global files.  'incl0h.h' contains general global variables
*  while 'incl1h.h' contains global variables pertaining to
*  the processor register variables.  'incl1h.h' and 'incl3h.h'
*  are generated by 'glob.c' with regin as input.  Therefore
*  to change processor variables, the variable is added or changed
*  in 'regin' and new include files are generated by 'glob'.
*  To change general global variables they must be physically changed
*  in both 'incl0h.h' and 'incl3h.h'
```

```
*
*****************************************************************************/

#include <stdio.h>
#include "incl0h.h"
#include "incl1h.h"
/*****************************************************************************/
/* name and stornam are the computer systems name for the swap area   */
/* that will hold the memory for paging.  This will enable the machine*/
/* to run with up to 62 FPs. Procnum is the processor number currently*/
/* in memory. Pagec is page modified flag (=0 if changed). This will  */
/* increase the system responce time if one just wiches to take a peek*/
/* at what one of the otherprocessors is doing.                       */
/* the tmpfd and statfd are the file descriptors for the temporary fil*/
/* and the status file.  This is done in order that the old IO package*/
/* may be used, and thus, the throughput of thesystem increased. The  */
/* fd, is not currently being used in this version, because of the    */
/* flag variable.  If the program being loaded into an FP is the same */
/* as a program loaded into another fp, the output file is not closed */
/* , as the close command takes over 12 seconds!!!!!!! The change in  */
/* IO packages took one evening of time, but the result is that the   */
/* new verion of mon6.c runs almost as fast as the single fp version. */
/*****************************************************************************/


        char name[15],stornam[22];
        int procnum,pagec,tmpfd,statfd,objfd,flag;

/*****************************************************************************/
/* This is to buffer the output. This will give an increase of 17:1 in*/
/* The throughput of the system. The fflush will cause the buffer to  */
/* dump.                                                              */
/*****************************************************************************/

        extern _sobuf[];

main()  {
        int     i;
        char *mktemp();
        setbuf(stdout, _sobuf);
/*****************************************************************************/
/* flag is set to one only at the beginning of the program. The flag  */
/* is set so that the program will not close the first program is loades*/
/* for further proof, see lode(reg,flag) later on.                    */
/*****************************************************************************/

        flag=1;
        procnum=1;
        lim=15;         /*the address of processor max*/

/*****************************************************************************/
/* this section of code creates the name of the temporary file in /tmp*/
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/* /tmp was chosen because it was so much faster than standard disk.  */
/*********************************************************************/

        stornam[0]='0';
        strcat(stornam,"/tmp/sbXXXXXX");
        mktemp(stornam);
        printf("temporary file used: %s0,stornam);
        if ((tmpfd=creat(stornam,0600))==-1){ printf("cannot creat memory,try latei
                                              exit(0);
                                            }
        close(tmpfd);

/*********************************************************************/
/* the original create opened the temporary file for read only. It was*/
/* closed.  The following will re-open the temporary file for both    */
/* read and write. this is why the new i/o library was not chosen.    */
/*********************************************************************/


        if ((tmpfd=open (stornam,2))==-1) {printf("unable to work with tmp fl");
                                           exit(1);
                                          }

/* Prompt for Status load  */

        printf("cyber Ikon simulator, hex mace 0);
        printf("Do you want to load status?  ");
        fflush(stdout);         /* dump print buffer */
        switch(getchar())  {
        case 'y':
              load(&r[0]);
              break;
        default :
              printf("Status not loaded0);
              break;
        }
        pagec=0;
        go();
}

/*  The function 'go' is the first level of command
*   it will call itself and reprompt for the processor in
*   case of an error.  It is also called by 'pros' on receipt
*   of a '#' to transfer control to this higher level       */

go()  {
        printf("32      Processor number?
        fflush(stdout);
        inhex();

/*********************************************************************/
/* if the processor chosen is different than the current processor and*/
```

```
/* the some part of the current page in memory has been modified, save*/
/* the current version of the page. Else, trash it!                   */
/*********************************************************************/

        if (input!=procnum && pagec!=0)  memset(&r[0],procnum);
        pagec=0;
        procnum=input;
        if (input<=lim&&input>=0) { memload(&r[0],procnum);
                pros(&r[0]);
                }
        else go();
}
pros(reg)
        int     *reg;   {
                int     i,j;
        char    com[3],string[40];

/*  Print prompt and scan for command  */

loOp:   printf("-->> ");
        fflush(stdout);
        scanf("%s",com);

/*********************************************************************/
/*      The following routine desides the action to be taken         */
/* after typing a command in the first instruction level.            */
/*********************************************************************/

        switch(lookup(com,comtab))  {

/*********************************************************************/
/*      type in an s, for single step.                               */
/*********************************************************************/

                case 0:
                page_used();
                step(reg);
                break;

/*********************************************************************/
/*      type in an m for modify memory                               */
/*********************************************************************/

                case 1:
                modify(reg);
                page_used();
                break;

/*********************************************************************/
/*      type in a l for load a program.                              */
/*********************************************************************/
```

```
                case 2:
                lode(reg,flag);
                page_used();
                break;

/*****************************************************************/
/*   type in a t for print the current values of the registers   */
/* this is used primarily for monitor debugging.                 */
/*****************************************************************/

                case 3:
                pritst(reg);
                break;

/*****************************************************************/
/*      Save all registers on file 'status'                      */
/*****************************************************************/

                case 4:
                statfd=creat("status",0600);
                oldseek(statfd,0,0);
                if (pagec==1) memset(&r[0],procnum);
                for (i=0;i<=lim;i++)
                {     memload(&r[0],i);
                      write(statfd,reg,24000);
                }
                memload(&r[0],procnum);
                close(statfd);
                break;

/*****************************************************************/
/*    type in a b X for a breakpoint after X steps of execution */
/*    type in a e X for the program to Execute X steps.          */
/*****************************************************************/

                case 5:
                case 12:
                skip(reg);
                page_used();
                break;

/*****************************************************************/
/*      type in an ! to enter a system command.                  */
/*****************************************************************/

                case 6:
                for(i=0;(string[i]=getchar())!='0;i++);
                string[i+1]='0';
                system(string);
                printf("!0);
                break;
```

```
/*****************************************************************/
/*      type in stop to discontinue processing.                  */
/*****************************************************************/

                case 7:
                unlink(stornam);
                exit();

/*****************************************************************/
/*      type in a # to go to the next level up.                  */
/*****************************************************************/

                case 8:
                go();
                break;

/*****************************************************************/
/*      recreate the register display on the screen without      */
/*  executing any of the program steps.  This is used to         */
/*  re-make the display after modifying the registers.           */
/*****************************************************************/

                case 9:
                prireg(reg);
                break;

/*****************************************************************/
/*      execute help command.  this is used to aid the user in   */
/* problems that should arise (if he gets stuck.)                */
/*****************************************************************/

                case 10:
                case 11:
                helpcom();
                break;

/*****************************************************************/
/* This will run a trace on the program, printing the register   */
/* values everytime it encounters a subroutine jump, or a return*/
/* from a subroutine.                                            */
/*****************************************************************/

                case 13:
                printflag=1;
                skip(reg);
                page_used();
                break;

/*****************************************************************/
/*      invalid character entered, go to next upper level.       */
/*****************************************************************/
```

```
                default:
                printf("??0);
                break;
                }

/*****************************************************************/
/*      After the statement has been executed, this part of the */
/* program will call itself.                                     */
/*****************************************************************/

        goto loOp;
}

/*****************************************************************/
/* This will execute the number of program steps corresponding  */
/* to the input value, or until it encounters an error.         */
/*****************************************************************/

skip(reg)
        int     *reg;   {
                int     j;
                inhex();

/*****************************************************************/
/*  enter the number of steps to be executed.                    */
/*****************************************************************/

                for(j=0;j<input;j++)    {
                        exec(reg);
                        if(erflg!=0) break;
                        if (printflag==1&&((f4[0]=='s')||(f4[0]=='j')))
                                prireg(reg);
                        }

/*****************************************************************/
/*      after the execution, print the registers.                */
/*****************************************************************/

                printflag=0;
                if(erflg==0) prireg(reg);


/*****************************************************************/
/* Single step and print registers.                              */
/*****************************************************************/

step(reg)
        int     *reg;  {
lo1p:   erflg=0;
        exec(reg);
        if(erflg==0)   ..
        prireg(reg);
```

```
/*****************************************************************/
/*      jump back to the monitor routine.                        */
/*****************************************************************/

        if(quest(reg) == 13) goto lo1p;
}

/*****************************************************************/
/*Single step subcommand level,issues prompt and decodes command*/
/*****************************************************************/

quest(reg)
        int     *reg;  {
                int     get;
                char    verb[10];
lo2p:   printf("step> ");
        fflush(stdout);
        scanf("%s",verb);

/*****************************************************************/
/*      lookup the action to be taken.                           */
/*****************************************************************/

        get=lookup(verb,verbtab);
        switch(get)     {

/*****************************************************************/
/*      single step--type in an s                                */
/*****************************************************************/

        case 0:
        page_used();
        return(13);
        break;

/*****************************************************************/
/*      dtemp will display the temporary registers.              */
/*****************************************************************/

        case 1:
        distemp(reg);
        page_used();
        goto lo2p;
        break;

/*****************************************************************/
/*      dmem will display the memory registers.                  */
/*****************************************************************/

        case 2:
        dismem(reg);
        page_used();
```

```
        goto lo2p;
        break;

/*****************************************************************/
/*      dislarg will display the large file.                     */
/*****************************************************************/

        case 3:
        dislarg(reg);
        goto lo2p;
        break;

/*****************************************************************/
/*      m will modify the registers                              */
/*****************************************************************/

        case 4:
        modify(reg);
        page_used();
        goto lo2p;
        break;

/*****************************************************************/
/*      execute x steps (b) (e)                                  */
/*****************************************************************/

        case 9:
        case 5:
        skip(reg);
        page_used();
        goto lo2p;
        break;

/*****************************************************************/
/*      h,help  execute help file.                               */
/*****************************************************************/

        case 7:
        case 8:
        helpstep();
        goto lo2p;
        break;

/*****************************************************************/
/*   p = print the registers as they now stand. This will fail   */
/* before the first step of execution.                           */
/*****************************************************************/

        case 6:
        prireg(reg);
        goto lo2p;
        break;
```

```
/*****************************************************************/
/*   This will transfer control to the COMMAND level.            */
/*****************************************************************/

        case 10:
        break;

/*****************************************************************/
/*   This will run a trace on the compiled program.  It will     */
/* print the registers when it encounters an sr or a jp,  effect*/
/* ively enabling one to breakpoint their program.               */
/*****************************************************************/

        case 11:
        erflg=0;
        printflag=1;
        skip(reg);
        page_used();
        goto lo2p;

/*****************************************************************/
/* Default takes monitor to command level                        */
/*****************************************************************/

        default:
        printf("Invalid Command0);
        goto lo2p;
        break;
        }
}

/*****************************************************************/
/*  Modify subcommand level, prompts with 'Register?' and        */
/*   responds to any register which is in the file 'regin'       */
/*****************************************************************/

modify(reg)
        int     *reg;   {
                char    regin[14];
                int     add;
        printf("Modify Register? ");
        fflush(stdout);
        scanf("%s",regin);
        if((bias=lookup(regin,regtab))==-1)  {
                printf("Invalid Register ");
                modify(reg);
                }


/*****************************************************************/
/*  If it is a valid input register, its index is put in 'bias' */
/* and action is taken depending on the register specifically on*/
```

```
/* whether or not it is dimensioned                                    */
/***********************************************************************/


        switch(bias) {

/*  Registers with an index of 3  */

        case 0: case 1: case 2: case 3: case 4: case 5:  case 53:
        case 72: case 73: case 74: case 75:
                add=indreg(reg,3);
                enter(add,reg,3);
                break;

/* Registers with an index of 1 (i.e two elements 0 and 1)  */

        case 43:
                add=indreg(reg,1);
                enter(add,reg,1);
                break;

/* Registers with an index of 15  */

        case 47: case 50: case 62:
                add=indreg(reg,15);
                enter(add,reg,15);
                break;
/* Registers with an index of 4095 ( ie the large files )  */

        case 54: case 55:
                add=indreg(reg,4095);
                enter(add,reg,4095);
                break;
/*  Micro memory  */

        case 68:
                memed(reg);
                break;
/* Registers with an index of 2 */

        case 57: case 56:
        add=indreg(reg,2);
        enter(add,reg,2);
        break;
/* # - go to command level  */

        case 76:
                return;
/* Registers which are not dimensioned */

        default:
                printf("  %s = %x  ",regtab[bias],reg[regint[bias]]);
```

```
                printf(" = ");
                inhex();
                reg[regint[bias]]=input;
                break;
        }
/*  This routine calls itself if no exit (ie not a #)  */

        modify(reg);
}

/* Indreg and enter are sort of a sub-sub-command level where you can
*  modify
*  indexed registers; there is no need to return to the register prompt
*  level to modify other in order addresses                        */

indreg(reg,i)
        int     *reg;   {
                int     inadd,biad;
/**/
        printf("Index? ");
        fflush(stdout);
        if(scanf("%x",&inadd)==0) return;
        if((inadd<0)||(inadd>i))  {
                printf(" Index must be less than %x0,i);
                inadd=indreg(reg,i);
                }
        return(inadd);
}

/*  Enter allows you to enter data, increment or decrement the
*   indexed register.  If the maximum index is exceeded, the monitor
*   returns to the 'Register?' level.                               */

enter(add,reg,limit)
        int     *reg; {
                int     biad;
                char    verb[1];
        for(;(add<limit+1)&&(add>-1);add++)  {
                biad=regint[bias]+add;
                printf("  %s[ %x ] = %x  ",regtab[bias],add,reg[biad]);
                fflush(stdout);
                scanf("%s",verb);

/*  Verb contains the action to be taken  */

                switch(verb[0]) {

/* Increment without changing the contents  */

                case '+':
                case 'i':
                break;
```

```
/* Decrement the same way  */

                case '-':
                case '^':
                case 'd':
                add=add-2;
                break;

/*  Change the register to the number input  */

                case 'c':
                inhex();
                reg[biad]=input;
                break;

/* Got back to 'Reg.' level  */

                default:
                printf("no action taken0);
                return;
                break;
                }
        }
}

/* Input a hex number, if it is not hex; flag it and prompt for hex */

inhex()  {
                fflush(stdout);
        if(scanf("%x",&input)==0)  {
                printf("Hex Please  ");
                fflush(stdout);
                scanf("%s",this);    /*  Dummy scan  */
                inhex();
                }
}

/*  General lookup routine returns the index in a table 'chartab'
* of a character string 'item'. If there is no match, -1 is returned */

lookup(item,chartab)
        char    *item;
        char    **chartab;      {
                int     i,j,r,k;
        for(i=0;chartab[i]!=-1;i++) {
        for(j=0;(r=chartab[i][j]) == item[j] && r != '0';j++);
        if(r == item[j])
                return(i);
        }
        return(-1);
}
```

```
/*  Print routine prints all the processor registers
*  a screen full anyway                                  */

prireg(reg)
        int     *reg;  {
        int     j,ad,i;
        printf("CYCLES:      ");
  printf("MAR:  MIR0: MIR1: MIR2:
        pnt(reg[cych]); pnt(reg[cycl]);printf(" ");
        pnt(reg[mar]-2);
        for(j=0;j<3;j++) pnt(reg[mir+j]);
        if (strcmp(f6,"") != 0)
        {
                printf("  %s  %s%s%s %s%s%s",f6,f7a,f7,f7b,f8a,f8,f8b);
                printf(" %s%s%s %s %s %s %s %s0,f9a,f9,f9b,f10,f11,f12,f13,f14);
                printf("ODX0: IDX1: IDX2: IDX3:              ICR0: ICR1: ICR2: ICR3
        }
        printf("        E1:   E0:0);
        for(i=indx0;i<icr0;i++) pnt(reg[i]);
        printf("        ");
        for(i=icr0;i<intr;i++)  pnt(reg[i]);
        printf("        "); pnt(reg[e1]); pnt(reg[e0]);
  printf("0CMR0: CMR1: CMR2: CMR3:
        printf("                 G1:    G0:0);
        for(i=cmr0;i<=cmr3;i++)  pnt(reg[i]);
        printf("        ");
        for(i=mcr0;i<=mcr3;i++)  pnt(reg[i]);
        printf("        "); pnt(reg[g1]); pnt(reg[g0]);
 printf("0BRG1: BRG0:   IMR0: IMR1:     INTR:   MARC:");
        printf("                        F1:   F0:0);
        pnt(reg[brg1]); pnt(reg[brg0]);
        printf("        ");
        for(i=imr0;i<=imr1;i++) pnt(reg[i]);
        printf("        ");
        pnt(reg[intr]); printf("        "); pnt(reg[marc]);
        printf("
  printf("0NOW: PAST:   MMT0: MMT1: MMT2:       OUP0: OUP1:");
        printf("                P:    Q:0);
        pnt(reg[now]); printf(" ");
        pnt(reg[past]);printf(" ");
        for(i=mmi0;i<=mmi2;i++) pnt(reg[i]);
        printf("        ");
        pnt(reg[oupu]); pnt(reg[oupu+1]);
        printf("
        printf("0IF0A:  IF1A:   LF0A:   LF1A:");
printf("       TF0AR:  TF0AW:  TF1AR:  TF1AW:  MULT:0);
        for(i=if0a;i<=lf1a;i++) {
        pnt(reg[i]); printf("   "); }
        for(i=tf0ar;i<=tf0aw;i++) {
        pnt(reg[i]); printf("   "); }
        for(i=tf1ar;i<=tf1aw;i++) {
        pnt(reg[i]); printf("   "); }
```

```
        pnt(reg[mult]);
    printf("0AINOUT:
        printf("0);
        for(i=0;i<=3;i++)  {
                for(j=0;j<=5;j++)  {
                        ad=j*4+i;
                        pnt(reg[ad]);
                        printf("       ");
                        }
                putchar('0);
                }
}

/*  Called by prireg,  prints out a hex number i with leading zeros  */

pnt(i)  {
        if(i<020&&i>=0) putchar('0');
        if(i<0400&&i>=0) putchar('0');
        if(i<010000&&i>=0) putchar('0');
        printf("%x  ",i);
}

/*  Store the processor registers on 'status' in the format:
*    index  reg[index]----------------reg[index+9]            */


/* Load the processor registers from the file 'status' in the
*  same format as specified by the store command                */

load(reg)
        int     *reg;  {
                int     i,j,k,ad,dumm;
        printf("Loading0);
              if((statfd=open("status",0))==-1) { printf("Null status0);
                                                  return; }
              oldseek(tmpfd,0,0);
              for (i=0;i<=15;i++)
              {       if (read(statfd,reg,24000)<24000)
                      { printf("incompatable status0);
                        printf("data loaded as far as fp %x",i);
                        fflush(stdout);
                        break;
                      }
                      write(tmpfd,reg,24000);
              }
              close(statfd);
}

/*  Print the registers according to there index (found in regin)
*   ten registers are output including and after the one asked for   */

pritst(reg,i)
```

```
        int     *reg;  {
                int     j,k;
        printf("Starting reg bias? ");
        fflush(stdout);
        scanf("%d",&j);
        print10(j,reg);
}

/*  Print the ten regs and if 'g' is input print ten more  */

print10(j,reg )
        int     *reg; {
                int     k;
                char    res[5];
        for(k=j;k<=j+9;k++)
        printf("%d   %x0,k,reg[k]);
        printf("?");
        fflush(stdout);
        scanf("%s",res);
        if(res[0]!='g')return;
        j=j+10;
        print10(j,reg);
}

/* Memory editor, prompts for address then from that computes
*  effective address and calls 'edit'                    */


memed(reg)
        int     *reg;  {
                int     efad;
        printf("Address? ");
        inhex();
        efad=regint[bias]+(3*input);
        edit(reg,efad);
}

/*  Edit memory in much the same way as the indexed registers  */

edit(reg,efad)
        int     *reg;  {
                char    verb[5];
        printf("mem< %x >= %x    ",input,reg[efad++]);
        printf("%x    ",reg[efad++]);
        printf("%x",reg[efad++]);
        fflush(stdout);
        scanf("%s",verb);

/*  Switch on input command  */

        switch(verb[0])  {
```

```
/* Increment, don't change the value  */

        case 'i':
        case '+':
        break;

/*  Decrement the same way  */

        case '^':
        case '-':
        efad=efad-6;
        break;

/*  Change all three locations, keeps expecting input in hex  */

        case 'c':
        efad=efad-3;
        inhex();
        reg[efad++]=input;
        inhex();
        reg[efad++]=input;
        inhex();
        reg[efad++]=input;
        break;

/*  Change memory address  */

        case 'a':
        inhex();
        efad=regint[bias]+(3*input);
        break;
/**/
        default:
        return;
        }

/* Re-compute effective address, check for end of memory  */

        input=(efad-regint[bias])/3;
        if((input<0)||(input>2255)) return;
        edit(reg,efad);
}

/* Display temporary files in readable format  */

distemp(reg)
        int     *reg;  {
        int     i;
        printf("0000");
        for(i=0;i<16;i++)  {
        printf("temp[%x] =       ",i);
        pnt(reg[tf1+i]); pnt(reg[tf0+i]);
```

```
        putchar('0);
        }
}

/*  Display memory in readable format, keeps asking for more as long
* as a valid starting address is input  */

dismem(reg)
        int     *reg;  {
                        int     add;
                        int     i,j,k;
        printf("Starting? ");
        fflush(stdout);
        if(scanf("%x",&i)==0)  {fflush(stdout),scanf("%s",this); return;
        add=i*3;
        for(k=0;k<23;k++)  {
                printf("mem<%x> =  ",i);
                for(j=0;j<3;j++)  {
                        pnt(reg[mem+add]);
                        add++;
                        }
                putchar('0);
                i++;
                if(i>255) break;
        }
        dismem(reg);
}

/*  Display the large files, keep displaying a page at a time until
*   an invalid address is input                                    */

dislarg(reg)

        int     *reg;  {
        int     i,j,k;
        printf("Starting? ");
        fflush(stdout);
        if(scanf("%x",&i)==0)  {fflush(stdout);scanf("%s",this); return;
        for(k=0;k<23;k++) {
                printf("lf[%x]  =  ",i);
                for(j=0;j<4;j++) {
                pnt(reg[lf0+i]); pnt(reg[lf1+i]); printf("      ");
                i++;
                if(i>4095) break;
                }
                putchar('0);
                if(i>4095) break;
        }
        dislarg(reg);
}
/*********************************************************************/
/* This routine will load a pas1 formatted object file into memory   */
```

```
/* Since the close command takes so long, this checks to see if the   */
/* previously loaded file is different from the requested file. If it */
/* is, the previously loaded file is closed and the new file is       */
/* opened. If it is the same, no files are closed or opened, saving 12*/
/* seconds of system time. In any event, this program must be         */
/* optimized.                                                         */
/**********************************************************************/

lode(reg,flag)
        int     *reg,flag;  {
                char    file[20];
                FILE    *srcptr;
                int     x1,x2,x3,add,efad;
                char    oldfil[20];

/*  Input file name to be loaded   */

        fflush(stdout);
        strcpy(oldfil,file);
        scanf("%s",file);
/**********************************************************************/
/* the first time throught, it would be rather difficult to close the */
/* previously opened file, as one does not exit. If flag = 1, it is   */
/* the first time through the file.                                   */
/**********************************************************************/

        if(strcmp(oldfil,file)!=0)
        {       if (flag!=1) {fclose(srcptr);}
                flag=0;
                if((srcptr=fopen(file,"r"))=='0') {
                        printf("Cannot open source file0);
                        return;
                        }

        }
/*  Ignore data until  '#' encountered  */

        while(getc(srcptr)!='#')
                while(getc(srcptr)!='0);

/* Kick new line first then input formatted data until read error    */

        while(getc(srcptr)!='0);
        while(fscanf(srcptr,"%x
                efad=mem+(add*3);
                reg[efad]=x1;
                efad++;
                reg[efad]=x2;
                efad++;
                reg[efad]=x3;
                efad++;
                }
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
}

/*************************************************************/
/* this routine, if called, will store the page in secondary */
/* store, this is the page out routine.                      */
/*************************************************************/

memset(reg,procnum)
        int *reg,procnum;
/*************************************************************/
/* this routine will get the correct starting address for the*/
/* beginning of the page.                                    */
/*************************************************************/

{       lseek(tmpfd,(24000L * (long) procnum),0);

/*************************************************************/
/* this routine does the write into secodary type storage.   */
/*************************************************************/

/* fwrite becomes write(tmpfd,reg,24000);   (fdes,bufptr,sizofbuf)   */
/* write returns -1 on error, or # of bytes written.  */
        if (write(tmpfd,reg,24000) <= 0) printf("cannot write!0);
}

/*************************************************************/
/* This is the routine that loads the "page" into memory.  If*/
/* the page does not exits, the routine will automatically   */
/* zero the memory locations needed.                         */
/*************************************************************/

 memload(reg,procnum)
        register *reg,procnum;
{       register i;
        lseek(tmpfd,(24000l*procnum),0);
/* the new read is done as follows: read(tmpfd,reg,10000) -1 for error
   0 for EOF encountered, else # of bytes read.
        if(read(tmpfd,reg,24000)<=0)
                for(i=0;i<12000;i++)
                        *reg++=0;
}

/*************************************************************/
/*  this is just a way of demonstrating that a page in memory*/
/* has been modified.  This will speed upthe rate of data    */
/* tansfer on the machine.                                   */
/*************************************************************/

page_used()
{       pagec=1;
}
/*************************************************************/
```

```
/* this function (called nargs) is just to satisfy a fluke in*/
/* the UNIX operating system.  Because this program is run   */
/* with separate Instruction and Data space, and UNIX is not */
/* quite equipped to handle such things on system calls, this*/
/* was added to the code to prevent a dump every time that a */
/* system call is executed.                                  */
/*************************************************************/

nargs()
{       return(0);
}
/*
*
*       XXXXXX  X    X  XXXXXX   XXXX   X    X         XXXX
*       X        X  X   X       X    X  X    X        X    X
*       XXXXX     XX    XXXXX   X       X    X        X
*       X         XX    X       X       XXXXXX        X
*       X        X  X   X       X    X  X    X   XX   X    X
*       XXXXXX  X    X  XXXXXX   XXXX   X    X   XX    XXXX
*
*************************************************************
*
*   FIRST PART OF FP EXECUTION ROUTINES
*
* CAUTION:  CODE SHOULD BE ADDED TO SHIOH.C TO AVOID OVERFLOW ERRORS
*           OF THE FORM   'format error exec.o' THIS IS 11/70 FOR
*           ' You can't compile a file that big '
*
*
*************************************************************
*
*   The include files here are explained in monh.c,  the ones
*   used here are of necessity non-initialized
* This is compiled as 'cc monh._ exech.c shioh._ helph._ -O'
*  or as 'cc runh._ exech.c shioh._ helph._ -O'
*
*************************************************************/

#include <stdio.h>
#include "incl2h.h"
#include "incl3h.h"

/*  'exec' does some overhead work, zeros the output variables and
*    updates the cycle count then calls the actual execution routines
*    some of which are in the file                              */

exec(reg)
        int     *reg;   {
                int     i;
        if(reg[cycl]==ffff) {reg[cycl]=0; reg[cych]++; }
        else  reg[cycl]++;
          conflg=0;
```

```
        f01=" ";
      for(i=0;i<=5;i++)   f2[i]=0;
        f3=" ";
        f4=" ";
        f5=" ";
      for(i=0;i<=5;i++)   f6[i]=0;
        f7=" ";
        f8=" ";
        f9=" ";
        f7a="";
        f7b="";
        f8a="";
        f8b="";
        f9a="";
        f9b="";
        f10=" ";
        f11=" ";
        f12=" ";
        f13=" ";
        f14=" ";

/* This simulates the buffered read.*/
        for(i=0;i<=2;i++) reg[mir+i]=reg[mbr+i];
        fetch(reg);
        reg[mar]++;
        conlod(reg);
        excond(reg);
        exalu(reg);
        exmult(reg);

/* The preceeding are executed whether or not the condition is met
*  the rest of the instruction is executed conditionally  */

        if(conflg!=0)   return;
        exindx(reg);
        exrj(reg);
        exmain(reg);
}

/*  Execute the ALU portion; check the arit/log bit and call
* exarit or exlog based on that test                        */

exalu(reg)
        int     *reg;   {
        op=(reg[mir+1]>>12)&017;
        if((reg[mir]&040000)!=0) exlog(reg);
        else   exarit(reg);
}

/*  Arithmetic execution   */

exarit(reg)
```

```
        int     *reg;   {
                f6[0]='a';
                car=0;
                if(reg[mir]>0)  {  car=1; f6[1]='c'; }
                else    f6[1]='n';
/*  Arithmetic */
                switch(op)  {
/* F=E   */
                        case 0:
                        f6[2]='0';
                        par0=0;par1=0;
                        par2=reg[e0]; par3=reg[e1];
                        dadd(reg);
                        break;
/* F=(E+G)  */
                        case 1:
                        f6[2]='1';
                        par0=0; par1=0;
                        par2=reg[e0]|reg[g0];
                        par3=reg[e1]|reg[g1];
                        dadd(reg);
                        break;
/* F=E+ G */
                        case 2:
                        f6[2]='2';
                        par0=0; par1=0;
                        par2=reg[e0]|( reg[g0]);
                        par3=reg[e1]|( reg[g1]);
                        dadd(reg);
                        break;
/* F=ZERO MINUS ONE  */
                        case 3:
                        f6[2]='3';
                        par0=0; par1=0;
                        par2=ffff; par3=ffff;
                        dadd(reg);
                        break;
/* F=E PLUS E G  */
                        case 4:
                        f6[2]='4';
                        par0=reg[e0]; par1=reg[e1];
                        par2=reg[e0]&( reg[g0]);
                        par3=reg[e1]&( reg[g1]);
                        dadd(reg);
                        break;
/* F= (E+G) PLUS E G  */
                        case 5:
                        f6[2]='5';
                        par0=reg[e0]&( reg[g0]);
                        par1=reg[e1]&( reg[g1]);
                        par2=reg[e0]|reg[g0];
                        par3=reg[e1]|reg[g1];
```

```
                        dadd(reg);
                        break;
/* F= E MINUS G MINUS ONE */
                        case 6:
                        f6[2]='6';
                        par0=reg[e0]; par1=reg[e1];
                        par2= reg[g0]+1;
                if(par2==0) par3= reg[g1]+1;
                else            par3= reg[g1];
                        dadd(reg);
                        par0=reg[res0]; par1=reg[res1];
                        par2=ffff;  par3=ffff;
                        car=0;
                        dadd(reg);
                        break;
/* F= E G MINUS ONE  */
                        case 7:
                        f6[2]= '7';
                        par0=ffff; par1=ffff;
                        par2=reg[e0]&( reg[g0]);
                        par3=reg[e1]&( reg[g1]);
                        dadd(reg);
                        break;
/* F= E PLUS EG  */
                        case 8:
                        f6[2]= '8';
                        par0=reg[e0]; par1=reg[e1];
                        par2=reg[e0]&reg[g0];
                        par3=reg[e1]&reg[g1];
                        dadd(reg);
                        break;
/* F= E PLUS G  */
                        case 9:
                        f6[2]='9';
                        par0=reg[e0]; par1=reg[e1];
                        par2=reg[g0]; par3=reg[g1];
                        dadd(reg);
                        break;
/* F= (E+ G) PLUS EG */
                        case 10:
                        f6[2]='a';
                        par2=reg[e0]&reg[g0];
                        par3=reg[e1]&reg[g1];
                        par0=reg[e0]|( reg[g0]);
                        par1=reg[e1]|( reg[g1]);
                        dadd(reg);
                        break;
/* F= EG MINUS ONE */
                        case 11:
                        f6[2]='b';
                        par2=reg[e0]&reg[g0];
                        par3=reg[e1]&reg[g1];
```

```
                        par0=ffff; par1=ffff;
                        dadd(reg);
                        break;
/* F= E PLUS E* */
                        case 12:
                        f6[2]='c';
                        par0=reg[e0]; par1=reg[e1];
                        par3=reg[e1]<<1;
                        if(reg[e0]<0) par3++;
                        par2=reg[e0]<<1;
                        dadd(reg);
                        break;
/* F= (E+G) PLUS E */
                        case 13:
                        f6[2]='d';
                        par0=reg[e0]; par1=reg[e1];
                        par2=reg[e0]|reg[g0];
                        par3=reg[e1]|reg[g1];
                        dadd(reg);
                        break;
/* F= (E+ G) PLUS E */
                        case 14:
                        f6[2]='e';
                        par0=reg[e0]; par1=reg[e1];
                        par2=reg[e0]|( reg[g0]);
                        par3=reg[e1]|( reg[g1]);
                        dadd(reg);
                        break;
/* F= E MINUS ONE */
                        case 15:
                        f6[2]='f';
                        par0=reg[e0]; par1=reg[e1];
                        par2=ffff; par3=ffff;
                        dadd(reg);
                        break;
                }
}

/*  Double register add routine simulates a 32 bit integer add
*  parameters are global par0-3 and set by 'exarit'          */

dadd(reg)
        int     *reg;   {
                int     icar;
        long one,two,three;
        icar=reg[ovl]=reg[ovh]=0;

/* 'icar' is the interim carry generated by the three if's to follow */

        if((par2<0)&&(par0<0)) icar=ffff;
        if((par0<0)&&(par2>0)&&(reg[res0]>=(par0))) icar=ffff;
        if((par2<0)&&(par0>0)&&(reg[res0]>=(par2))) icar=ffff;
```

```
/* Set processor overflow  by the conditions described in the manual */
        one=two=three=0l;
        one=((long) par1<<16) + ((long) par0 & 0x0000ffffl);
        two=((long) par3<<16) + ((long) par2 & 0x0000ffffl);
        three=one+two+(long) car;
        reg[res1] = (int) (three >> 16);
        reg[res0] = (int) (three & 0x0000ffffl);
        if((par0>0)&&(par2>0)&&(reg[res0]<0)) reg[ovl]=ffff;
        if((par0<0)&&(par2<0)&&(reg[res0]>0)) reg[ovl]=ffff;
        if((par1>0)&&(par3>0)&&(reg[res1]<0)) reg[ovh]=ffff;
        if((par1<0)&&(par3<0)&&(reg[res1]>0)) reg[ovh]=ffff;
}

/* Execute logical operations  */

exlog(reg)
        int     *reg;  {
        f6[0]='l';
        switch(op)  {
/* F= E */
                case 0:
                f6[1]='0';
                reg[res0]=  reg[e0];
                reg[res1]=  reg[e1];
                break;
/* F=  (E+G) */
                case 1:
                f6[1]='1';
                reg[res0]=  (reg[e0]|reg[g0]);
                reg[res1]=  (reg[e1]|reg[g0]);
                break;
/* F=  EG */
                case 2:
                f6[1]='2';
                reg[res0]=  (reg[e0])&reg[g0];
                reg[res1]=  (reg[e1])&reg[g1];
                break;
/* F= 0 */
                case 3:
                f6[1]='3';
                reg[res0]=0;
                reg[res1]=0;
                break;
/* F=  (EG) */
                case 4:
                f6[1]='4';
                reg[res0]=  (reg[e0]&reg[g0]);
                reg[res1]=  (reg[e1]&reg[g1]);
                break;
/* F=  G */
                case 5:
                f6[1]='5';
```

```
                reg[res0]=  reg[g0];
                reg[res1]=  reg[g1];
                break;
/* F= E XOR G */
                case 6:
                f6[1]='6';
                reg[res0]= reg[e0]^reg[g0];
                reg[res1]= reg[e1]^reg[g1];
                break;
/* F= E G */
                case 7:
                f6[1]='7';
                reg[res0]= reg[e0]&( reg[g0]);
                reg[res1]= reg[e1]&( reg[g1]);
                break;
/* F=  E+G */
                case 8:
                f6[1]='8';
                reg[res0]= ( reg[e0])|reg[g0];
                reg[res1]= ( reg[e1])|reg[g1];
                break;
/* F=  (E EOR G) */
                case 9:
                f6[1]='9';
                reg[res0]=  (reg[e0]^reg[g0]);
                reg[res1]=  (reg[e1]^reg[g1]);
                break;
/* F= G */
                case 10:
                f6[1]='a';
                reg[res0]= reg[g0];
                reg[res1]= reg[g1];
                break;
/* F= EG */
                case 11:
                f6[1]='b';
                reg[res0]= reg[e0]&reg[g0];
                reg[res1]= reg[e1]&reg[g1];
                break;
/* F= ONE */
                case 12:
                f6[1]='c';
                reg[res0]=1;
                reg[res1]=0;
                break;
/* F= E+ G */
                case 13:
                f6[1]='d';
                reg[res0]= reg[e0]|( reg[g0]);
                reg[res1]= reg[e1]|( reg[g1]);
                break;
/* F= (E+G) */
```

```
                case 14:
                f6[1]='e';
                reg[res0]= (reg[e0]|reg[g0]);
                reg[res1]= (reg[e1]|reg[g0]);
                break;
/* F= E */
                case 15:
                f6[1]='f';
                reg[res0]= reg[e0];
                reg[res1]= reg[e1];
                break;
        }
}

/* Execute conditional field    */

excond(reg)
        int     *reg;  {
        op=(reg[mir+2]>>11)&07;
        conflg=0;
        switch(op)  {
/*  Uncond. ALU now cond to shift */
                case 0:
                f2[0]='u'; f2[1]='n';
                reg[shcon]=reg[now];
                break;
/* True data now */
                case 1:
                f2[0]='t'; f2[1]='n';
                reg[shcon]=reg[now];
                if(reg[cmr0]!=0 &&((reg[now]&reg[cmr0])==0)) conflg=ffff;
/* was !=*/     maskt(reg);
                break;
/* False data now */
                case 2:
                f2[0]='f'; f2[1]='n';
                reg[shcon]=  reg[now];
                if(reg[cmr0]!=0 && ((reg[shcon]&reg[cmr0])==0)) conflg=ffff;
                maskf(reg);
                break;
/* Address */
                case 3:
                f2[0]='a';      f2[1]='d';
                reg[shcon]=  reg[now];
                if((reg[cmr3]&01)!=0)
                        if(reg[icr0]==reg[indx0]) {conflg=ffff; break;}
                if((reg[cmr3]&02)!=0)
                        if(reg[icr0]!=reg[indx0]) {conflg=ffff; break;}
                if((reg[cmr3]&04)!=0)
                        if(reg[icr1]==reg[indx1]) {conflg=ffff; break;}
                if((reg[cmr3]&010)!=0)
                        if(reg[icr1]!=reg[indx1]) {conflg=ffff; break;}
```

```
                if((reg[cmr3]&020)!=0)
                        if(reg[icr2]==reg[indx2]) {conflg=ffff; break;}
                if((reg[cmr3]&040)!=0)
                        if(reg[icr2]!=reg[indx2]) {conflg=ffff; break;}
                if((reg[cmr3]&0100)!=0)
                        if(reg[icr3]==reg[indx3]) {conflg=ffff; break;}
                if((reg[cmr3]&0200)!=0)
                        if(reg[icr3]!=reg[indx3]) {conflg=ffff; break;}
                if((reg[cmr3]&0400)!=0)
                        if(reg[mcr0]!=reg[indx0]) {conflg=ffff; break;}
                if((reg[cmr3]&01000)!=0)
                        if(reg[mcr1]!=reg[indx1]) {conflg=ffff; break;}
                if((reg[cmr3]&02000)!=0)
                        if(reg[mcr2]==reg[indx2]) {conflg=ffff; break;}
                if((reg[cmr3]&04000)!=0)
                        if(reg[mcr3]!=reg[indx3]) {conflg=ffff; break;}
                if((reg[cmr3]&010000)!=0)
                        if(reg[mar]!=reg[marc]) {conflg=ffff; break;}
                        break;
/*  Uncond. ALU past cond to shift */
                case 4:
                f2[0]='u';f2[1]='p';
                reg[shcon]=reg[past];
                break;
/* True data past */
                case 5:
                f2[0]='t';f2[1]='p';
                reg[shcon]=reg[past];
                if(reg[cmr0]!=0 && ((reg[shcon]&reg[cmr0])==0)) conflg=ffff;
                maskt(reg);
                break;
/* False data past */
                case 6:
                f2[0]='f';f2[1]='p';
                reg[shcon]=  reg[past];
                if(reg[cmr0]!=0 && ((reg[shcon]&reg[cmr0])==0)) conflg=ffff;
                maskf(reg);
                break;
/* Input Output */
                case 7:
                f2[0]='i'; f2[1]='o';
                reg[shcon]=  reg[past];
                printf( "I/O conditions not simulated0);
                erflg=ffff;
                break;
        }
}

/*  The following two routines are called by 'excon' to test for
*   conditions called for in 'cmr1'                          */

maskt(reg)
```

```
        int     *reg;  {
                if(reg[cmr1]==0) return;
        if((reg[cmr1]&01)!=0)  { printf("Sense not simulated0);
                erflg=ffff;  }
        if((reg[cmr1]&02)!=0) { printf("Sense not simulated0);
                erflg=ffff;  }
                if((reg[cmr1]&04)!=0)
                if(reg[icr0]<=reg[indx0]) { conflg=ffff; return; }
                if((reg[cmr1]&010)!=0)
                if(reg[icr0]>=reg[indx0]) { conflg=ffff; return; }
                if((reg[cmr1]&020)!=0)
                if(reg[icr0]!=reg[indx0]) { conflg=ffff; return; }
                if((reg[cmr1]&040)!=0)
                if(reg[icr1]<=reg[indx1]) { conflg=ffff; return; }
                if((reg[cmr1]&0100)!=0)
                if(reg[icr1]>=reg[indx1]) { conflg=ffff; return; }
                if((reg[cmr1]&0200)!=0)
                if(reg[icr1]!=reg[indx1]) { conflg=ffff; return; }
                conflg=0;
}

/* The following is the same as 'maskf' except that all conditions
*  must be false for 'conflg' to be set inhibiting execution     */

maskf(reg)
        int     *reg;  {
                if(reg[cmr1]==0) return;
                conflg=ffff;
        if((reg[cmr1]&01)!=0) { printf("Sense not simulated0);
                erflg=ffff;  }
        if((reg[cmr1]&02)!=0) { printf("Sense not simulated0);
                erflg=ffff;  }
                if((reg[cmr1]&04)!=0)
                if(reg[icr0]<=reg[indx0]) { conflg=0; return; }
                if((reg[cmr1]&010)!=0)
                if(reg[icr0]>=reg[indx0]) { conflg=0; return; }
                if((reg[cmr1]&020)!=0)
                if(reg[icr0]!=reg[indx0]) { conflg=0; return; }
                if((reg[cmr1]&040)!=0)
                if(reg[icr1]<=reg[indx1]) { conflg=0; return; }
                if((reg[cmr1]&0100)!=0)
                if(reg[icr1]>=reg[indx1]) { conflg=0; return; }
                if((reg[cmr1]&0200)!=0)
                if(reg[icr1]!=reg[indx1]) { conflg=0; return; }
}

/* Conlod checks the conditions of the bus and sets the now register
*  based on its state (and the state of various registers)
*  note that the conditions reflect the beginning of execution   */

conlod(reg)
        int     *reg;  {
```

```
        reg[now]=0;
        if(reg[e0]<0) reg[now]=reg[now]|01;
        if(reg[e1]<0) reg[now]=reg[now]|02;
        if(reg[f0]<0) reg[now]=reg[now]|04;
        if(reg[f1]<0) reg[now]=reg[now]|010;
        if(reg[g0]<0) reg[now]=reg[now]|020;
        if(reg[g1]<0) reg[now]=reg[now]|040;
        if(reg[res0]<0) reg[now]=reg[now]|0100;
        if(reg[res1]<0) reg[now]=reg[now]|0200;
        if(reg[res0]==ffff) reg[now]=reg[now]|0400;
  if((reg[res0]==ffff )&&( reg[res1]==ffff)) reg[now]=reg[now]|01000;
        if(reg[ovl]!=0) reg[now]=reg[now]|02000;
        if(reg[ovh]!=0) reg[now]=reg[now]|04000;
        f2[2]='n';
        if((reg[mir+2]&02000)!=0) { f2[2]='s'; reg[past]=reg[now]; }
}

/* Execute the multiply. 'mulflg' contains the value of the
*  previous multiply and 'merflg' is set if the current
*  multiply is not the same as the last. Results are not placed
*  in 'mult' until the second consecutive call of the same type */

exmult(reg)
        int     *reg;  {
        op=(reg[mir+2])&017;
        merflg=0;
        switch(op)   {
/**/
                case 0:
                f5="plpl";
  if(reg[mulflg]!=0)  {reg[mulflg]=0;  merflg=ffff; break; }
                reg[mult]= (reg[p]&0377) * (reg[p]&0377);
                break;
/**/
                case 1:  case 4:
                f5="plpu";
  if(reg[mulflg]!=1)  {reg[mulflg]=1;  merflg=ffff; break; }
                reg[mult]= ((reg[p]>>8)&0377) * (reg[p]&0377);
                break;
/**/
                case 2: case 8:
                f5="plql";
  if(reg[mulflg]!=2)  {reg[mulflg]=2;  merflg=ffff; break; }
                reg[mult]= (reg[q]&0377) * (reg[p]&0377);
                break;
/**/
                case 3: case 12:
                f5="plqu";
  if(reg[mulflg]!=3)  {reg[mulflg]=3;  merflg=ffff; break; }
                reg[mult]= (reg[p]&0377) * ((reg[q]>>8)&0377);
                break;
/**/
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
                case 5:
                f5="pupu";
  if(reg[mulflg]!=5)  {reg[mulflg]=5;  merflg=ffff; break; }
                reg[mult]= ((reg[p]>>8)&0377) * ((reg[p]>>8)&0377);
                break;
/**/
                case 6: case 9:
                f5="puql";
  if(reg[mulflg]!=6)  {reg[mulflg]=6;  merflg=ffff; break; }
                reg[mult]= ((reg[p]>>8)&0377) * (reg[q]&0377);
                break;
/**/
                case 7: case 13:
                f5="puqu";
  if(reg[mulflg]!=7)  {reg[mulflg]=7;  merflg=ffff; break; }
                reg[mult]= ((reg[p]>>8)&0377) * ((reg[q]>>8)&0377);
                break;
/**/
                case 10:
                f5="qlql";
  if(reg[mulflg]!=10)  {reg[mulflg]=10;  merflg=ffff; break; }
                reg[mult]= (reg[q]&0377) * (reg[q]&0377);
                break;
/**/
                case 11: case 14:
                f5="qlqu";
  if(reg[mulflg]!=11)  {reg[mulflg]=11;  merflg=ffff; break; }
                reg[mult]= ((reg[q]>>8)&0377) * (reg[q]&0377);
                break;
/**/
                case 15:
                f5="ququ";
  if(reg[mulflg]!=15)  {reg[mulflg]=15;  merflg=ffff; break; }
                reg[mult]= ((reg[q]>>8)&0377) * ((reg[q]>>8)&0377);
                break;
        }
}

/* Return jump  */

exrj(reg)
        int     *reg;  {
                op=(reg[mir+2]>>4)&03;
        switch(op)  {
                case 0:
                f4="np";
                break;
/**/
                case 1:
                f4="jp";
                reg[mar]=reg[stack+reg[stptr]];
                break;
```

```
/**/
                case 2:
                f4="sr";
                reg[stptr]=(reg[stptr]+1)&017;
                reg[stack+reg[stptr]]=reg[mar];
                break;
/**/
                case 3:
                f4="df";
/*              reg[mar]=reg[stack+reg[stptr]];          */
                reg[stptr]=(reg[stptr]-1)&017;
                break;
        }
}

/* Index execution  */

exindx(reg)
        int     *reg;  {
                op=(reg[mir+2]>>6)&017;
                switch(op)  {
                case 0:
                f3="nop";
                break;
/**/
                case 1:
                f3="in0";
                reg[indx0]++;
                break;
/**/
                case 2:
                f3="in1";
                reg[indx1]++;
                break;
/**/
                case 3:
                f3="in2";
                reg[indx2]++;
                break;
/**/
                case 4:
                f3="in3";
                reg[indx3]++;
                break;
/**/
                case 5:
                f3="dc0";
                reg[indx0]--;
                break;
/**/
                case 6:
                f3="dc1";
```

```
                reg[indx1]--;
                break;
/**/
                case 7:
                f3="dc2";
                reg[indx2]--;
                break;
/**/
                case 8:
                f3="dc3";
                reg[indx3]--;
                break;
/**/
                case 9:
                f3="cl0";
                reg[indx0]=0;
                break;
/**/
                case 10:
                f3="cl1";
                reg[indx1]=0;
                break;
/**/
                case 11:
                f3="cl2";
                reg[indx2]=0;
                break;
/**/
                case 12:
                f3="cl3";
                reg[indx3]=0;
                break;
/**/
                case 13:
                f3="ce0";
        printf("End ffs not simulated0);
                erflg=ffff;
                break;
/**/
                case 14:
                f3="ce1";
        printf("End ffs not simulated0);
                erflg=ffff;
                break;
/**/
                case 15:
                f3="cla";
                reg[indx0]=0;
                reg[indx2]=0;
                reg[indx1]=0;
                reg[indx3]=0;
                break;
```

```
        }
}

/* Get the next instruction from memory, put it in mbr */

fetch(reg)
        int     *reg;  {
                int     efad,i;
                efad=(reg[mar]*3)+mem;
                for(i=0;i<=2;i++) { reg[mbr+i]=reg[efad+i];
                                  }
}

/* Execute the main part of the instruction   */

exmain(reg)
        int     *reg;  {
                op=(reg[mir+2]>>14)&03;
                switch(op)      {
                case 0:
                f01="tr";
                extran(reg);
                break;
/**/
                case 1:
                f01="tc";
                extrac(reg);
                break;

/* Shift and io are in the file 'shio.c'  */

                case 2:
                f01="sh";
                exshift(reg);
                break;
/**/
                case 3:
                f01="io";
                exio(reg);
                break;
        }
}

/* Tranfer data from register to register */

extran(reg)
        int     *reg;   {

/* Get source and destination code from instruction  */

        s0=(reg[mir+1])&0007700;
        s1=(reg[mir+1])&0000077;
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        d0=(reg[mir])&0037600;
        d1=(reg[mir])&0000177;

/* Use input codes to get an index returned from a list of valid
*  codes (uses a lookup routine)                              */

        s0=finint(s0,sr0int);
        s1=finint(s1,sr1int);
        d0=finint(d0,d0int);
        d1=finint(d1,d1int);

/* Set the fields for output ( also a table )  */

        f7 = srctab[s0];
        f9 = srctab[s1];
        f8 = d0tab[d0];
        f10 = d1tab[d1];

/* After source, s0 and s1 contain  values of the sourced registers  */

        source(reg);

/* The store routine stores the values in the designated registers */

        stord0(reg);
        stord1(reg);
}
source(reg)
        int     *reg;   {

/* Set ALU eight bit chunks and sign extensions for later referance */

        b0=reg[res0]&0377;
        b1=(reg[res0]>>8)&0377;
        b2=reg[res1]&0377;
        b3=(reg[res1]>>8)&0377;
        ls=(reg[res0]>>16)&0377;
        us=(reg[res1]>>16)&0377;

/*  Switch here really only to take care of odd ALU sources  */

        switch(s0)      {
/**/

/* Default calls another switch  */

        default:
        s0=srsw(s0,reg);
        s1=srsw(s1,reg);
        break;
/*  a0sw */
        case 1:
```

```
        if((s1==2)|(s1==3)|(s1==4)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=8)&(s1>=5))
                switch(s1)      {
/* a0sw - a1sw */
                case 5:
                s0=(b2<<8)|b1;
                s1=(b0<<8)|b3;
                break;
/* a0sw - a1rs  */
                case 6:
                s0=(ls<<8)|b1;
                s1=(b0<<8)|b3;
                break;
/* a0sw - a1rz */
                case 7:
                s0=b1;
                s1=(b0<<8)|b3;
                break;
/* a0sw - a1 */
                case 8:
                printf("Illegal ALU source 0);
                erflg=ffff;
                break;
                }
        else    { s0=(b0<<8)|b1;  s1=srsw(s1,reg);  }
        break;
/* a0rs  */
        case 2:
        if((s1==1)|(s1==3)|(s1==4)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=8)&(s1>=5))
                switch(s1)      {
/* a0rs - a1sw */
                case 5:
                s0=(b2<<8)|b1;
                s1=(us<<8)|b3;
                break;
/* a0rs - a1rs  */
                case 6:
                s0=(ls<<8)|b1;
                s1=(us<<8)|b3;
                break;
/* a0rs - a1rz */
                case 7:
                s0=b1;
                s1=(us<<8)|b3;
                break;
```

```
/* a0rs - a1 */
                case 8:
                printf("Illegal ALU source 0);
                erflg=ffff;
                break;
                }
        break;
/* a0rz  */
        case 3:
        if((s1==2)|(s1==1)|(s1==4)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=8)&(s1>=5))
                switch(s1)      {
/* a0rz - a1sw */
                case 5:
                s0=(b2<<8)|b1;
                s1=b3;
                break;
/* a0rz - a1rs  */
                case 6:
                s0=(ls<<8)|b1;
                s1=b3;
                break;
/* a0rz - a1rz */
                case 7:
                s0=b1;
                s1=b3;
                break;
/* a0rz - a1 */
                case 8:
                printf("Illegal ALU source 0);
                erflg=ffff;
                break;
                }
        else    { s0=b1;  s1=srsw(s1,reg);  }
        break;
/* a0  */
        case 4:
        if((s1==2)|(s1==3)|(s1==1)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=8)&(s1>=5))
        switch(s1)      {
/* a0 - a1 */
                case 8:
                s0=reg[res0];
                s1=reg[res1];
                break;
/**/
```

```
                default:
                printf("Adder source error0);
                erflg=ffff;
                break;
                }
        else    { s0=reg[res0];  s1=srsw(s1,reg); }
        break;
/*  a1sw */
        case 5:
        if((s1==8)|(s1==6)|(s1==7)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=4)&(s1>=1))
                switch(s1)
/* a1sw - a0sw */
                case 1:
                s1=(b2<<8)|b1;
                s0=(b0<<8)|b3;
                break;
/* a1sw - a0rs  */
                case 2:
                s1=(b2<<8)|b1;
                s0=(us<<8)|b3;
                break;
/* a1sw - a0rz */
                case 3:
                s1= (b2<<8)|b1;
                s0= b3;
                break;
/* a1sw - a0 */
                case 4:
                printf("Illegal ALU source 0);
                erflg=ffff;
                break;
                }
        else    { s0=(b2<<8)|b3;  s1=srsw(s1,reg);  }
        break;
/* a1rs  */
        case 6:
        if((s1==5)|(s1==7)|(s1==8)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=4)&(s1>=1))
                switch(s1)      {
/* a1rs - a0sw */
                case 1:
                s1=(ls<<8)|b1;
                s0=(b0<<8)|b3;
                break;
/* a1rs - a0rs  */
```

```
                case 2:
                s1=(ls<<8)|b1;
                s0=(us<<8)|b3;
                break;
/* a1rs - a0rz */
                case 3:
                s0=(ls<<8)|b1;
                s1=b3;
                break;
/* a1rs - a0 */
                case 4:
                printf("Illegal ALU source 0);
                erflg=ffff;
                break;
                }
        break;
/* a1rz  */
        case 7:
        if((s1==5)|(s1==6)|(s1==8)) {
                printf("Adder source error0);
                erflg=ffff;
                }
        if((s1<=4)&(s1>=1))
                switch(s1)      {
/* a1rz - a0sw */
                case 1:
                s0=(b0<<8)|b3;
                s1=b1;
                break;
/* a1rz - a0rs  */
                case 2:
                s1=b1;
                s0=(us<<8)|b3;
                break;
/* a1rz - a0rz */
                case 3:
                s1=b1;
                s0=b3;
                break;
/* a1rz - a0 */
                case 4:
                printf("Illegal ALU source 0);
                erflg=ffff;
                break;
                }
        else    { s0=b3;  s1=srsw(s1,reg);  }
        break;
/* a1  */
        case 8:
        if((s1==5)|(s1==6)|(s1==7)) {
                printf("Adder source error0);
                erflg=ffff;
```

```
                }
        if((s1<=4)&(s1>=1))
        switch(s1)      {
/* a1 - a0 */
                case 4:
                s0=reg[res1];
                s1=reg[res0];
                break;
/**/
                default:
                printf("Adder source error0");
                erflg=ffff;
                break;
                }
        else    { s0=reg[res1];  s1=srsw(s1,reg); }
        break;
        }
        reg[bus0]=s0;
        reg[bus1]=s1;
}

/* Given that their are no special ALU considerations this
*  is where sources are determined. ss is the return
*  parameter but represent the global s0 or s1          */


srsw(ss,reg)
        int     *reg;   {
                int     ret,flg,xx0,xx1,xx2,xx3;
        switch(ss)      {
/* nop */
        case 0:
        break;
/* a0sw  */
        case 1:
        ss=(b0<<8)|b1;
        break;
/* a0rs */
        case 2:
        ss=(ls<<8)|b1;
        break;
/* a0rz */
        case 3:
        ss=b1;
        break;
/* a0 */
        case 4:
        ss=reg[res0];
        break;
/* a1sw */
        case 5:
        ss=(b2<<8)|(b3);
```

```
        break;
/* a1rs */
        case 6:
        ss=(us<<8)|b3;
        break;
/* a1rz */
        case 7:
        ss=b3;
        break;
/* a1 */
        case 8:
        ss=reg[res0];
        break;
/* qrsp */
        case 9:
        printf("A/Q not simulated0);
        erflg=ffff;
        break;
/* arsp */
        case 10:
        printf("A/Q not simulated0);
        erflg=ffff;
        break;
/* ifox */
case 19: case 20: case 21: case 22: case 23: case 24: case 25: case 26:
        printf(" Input file not implemented0);
        erflg=ffff;
        break;
/* lf0n */
        case 31:
        ss=reg[lf0+reg[lf0a]];
        break;
/* lf0u */
        case 32:
        ss=reg[lf0+reg[lf0a]];
        reg[lf0a]=(reg[lf0a]+1)&07777;  /* was 01777 */
        break;
/* lf0d */
        case 33:
        ss=reg[lf0+reg[lf0a]];
        reg[lf0a]=(reg[lf0a]-1)&07777;
        break;
/* lf0a */
        case 34:
        ss=reg[lf0+reg[lf0a]];
        reg[lf0a]=(reg[lf0a]+reg[indx0])&07777;
        break;
/* lf1n */
        case 36:
        ss=reg[lf1+reg[lf1a]];
        break;
/* lf1u */
```

```
        case 37:
        ss=reg[lf1+reg[lf1a]];
        reg[lf1a]=(reg[lf1a]+1)&07777;
        break;
/* lf1d */
        case 38:
        ss=reg[lf1+reg[lf1a]];
        reg[lf1a]=(reg[lf1a]-1)&07777;
        break;
/* lf1a */
        case 39:
        ss=reg[lf1+reg[lf1a]];
        reg[lf1a]=(reg[lf1a]+reg[indx0])&07777;
        break;
/* mult */
        case 42:
   if(merflg!=0) { printf("Multiply source error0); erflg=ffff; }
        ss=reg[mult];
        break;
/* tf0n */
        case 43:
        ss=reg[tf0+reg[tf0ar]];
        break;
/* tf0u */
        case 44:
        ss=reg[tf0+reg[tf0ar]];
        reg[tf0ar]=(reg[tf0ar]+1)&017;
        break;
/* tf0d */
        case 45:
        ss=reg[tf0+reg[tf0ar]];
        reg[tf0ar]=(reg[tf0ar]-1)&017;
        break;
/* tf0c */
        case 46:
        ss=reg[tf0+reg[tf0ar]];
        reg[tf0ar]=0;
        break;
/* tf1n */
        case 49:
        ss=reg[tf1+reg[tf1ar]];
        break;
/* tf1u */
        case 50:
        ss=reg[tf1+reg[tf1ar]];
        reg[tf1ar]=(reg[tf1ar]+1)&017;
        break;
/* tf1d */
        case 51:
        ss=reg[tf1+reg[tf1ar]];
        reg[tf1ar]=(reg[tf1ar]-1)&017;
        break;
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/* tf1c */
        case 52:
        ss=reg[tf1+reg[tf1ar]];
        reg[tf1ar]=0;
        break;
/* z */
/* If the z register is sourced, processor may simulate a four
* cycle wait state                          */

        case 11: case 12: case 13: case 14:
        xx0=ss-11;
        xx1=0;
        xx2=reg[cycl] +  reg[cycsl+xx0];
        if((  reg[cycsl+xx0]<0)&&(reg[cycl]<0)) xx1=ffff;
        if((reg[cycl]<0)&&(xx2>reg[cycl])) xx1=ffff;
        if((  reg[cycsl+xx0]<0)&&(xx2>  reg[cycsl+xx0])) xx1=ffff;
        xx3=reg[cych]+  reg[cycsh+xx0];
        if(xx1!=0) xx3++;
        if((xx3==0)&&(xx2<5)&&(xx2>=1)) {
        xx1=0;
        xx2=reg[cycsl]+4;
        if((4<0)&&(reg[cycsl]<0)) xx1=ffff;
        if((reg[cycsl]<0)&&(xx2>reg[cycsl])) xx1=ffff;
        if((4<0)&&(xx2>4)) xx1=ffff;
        xx3=reg[cycsh]+0;
        if(xx1!=0) xx3++;
        reg[cycl]=xx2;
        reg[cych]=xx3;
        }
        ss=reg[srcint[ss]];
        break;

/* The default case look up the register index and sources the
*  indicated register by bias                              */

        default:
        ss=reg[srcint[ss]];
        break;
        }
        return(ss);
}

/* Store the s0 source into the d0 destination  */

stord0(reg)
        int     *reg;   {
        switch(d0)      {
/**/
        case 0:
        break;
/* e0f0 */
        case 3:
```

```
        reg[e0]=reg[f0]=s0;
        break;
/* e0g1 */
        case 4:
        reg[e0]=reg[g1]=s0;
        break;
/* ef0g */
        case 5:
        reg[e0]=reg[f0]=reg[g1]=s0;
        break;
/* f0g1 */
        case 7:
        reg[f0]=reg[g1]=s0;
        break;
/* lf0n */
        case 12:
        reg[lf0+reg[lf0a]]=s0;
        break;
/* lf0u */
        case 13:
        reg[lf0+reg[lf0a]]=s0;
        reg[lf0a]=(reg[lf0a]+1)&07777;
        break;
/* lf0d */
        case 14:
        reg[lf0+reg[lf0a]]=s0;
        reg[lf0a]=(reg[lf0a]-1)&07777;
        break;
/* lf0a */
        case 15:
        reg[lf0+reg[lf0a]]=s0;
        reg[lf0a]=(reg[lf0a]+reg[indx0])&07777;
        break;
/* p */
        case 22:
        reg[mulflg]=ffff;
        reg[p]=s0;
        break;
/* pe0 */
        case 23:
        reg[p]=reg[e0]=s0;
        break;
/* pe0g */
        case 24:
        reg[p]=reg[e0]=reg[g1]=s0;
        break;
/* pef0 */
        case 25:
        reg[p]=reg[e0]=reg[f0]=s0;
        break;
/* pefg */
        case 26:
```

```
        reg[p]=reg[e0]=reg[f0]=reg[g1]=s0;
        break;
/* pf0 */
        case 27:
        reg[p]=reg[f0]=s0;
        break;
/* pf0g */
        case 28:
        reg[p]=reg[f0]=reg[g1]=s0;
        break;
/* pg1 */
        case 29:
        reg[p]=reg[g1]=s0;
        break;
/* tf0n */
        case 30:
        reg[tf0+reg[tf0aw]]=s0;
        break;
/* tf0u */
        case 31:
        reg[tf0+reg[tf0aw]]=s0;
        reg[tf0aw]=(reg[tf0aw]+1)&017;
        break;
/* tf0d */
        case 32:
        reg[tf0+reg[tf0aw]]=s0;
        reg[tf0aw]=(reg[tf0aw]-1)&017;
        break;
/* tf0c */
        case 33:
        reg[tf0+reg[tf0ar]]=s0;
        reg[tf0ar]=0;
        break;
/* t0ba */
        case 36:
        reg[tf0ar]=reg[tf0aw]=s0;
        break;

/*  The default looks up the register bias again   */

        default:
        reg[db0int[d0]]=s0;
        break;
        }
}

/* This puts the s1 souce into the d1 destination  */

stord1(reg)
        int     *reg;   {
                int     efad;
        efad=reg[mar]*3;
```

```
        switch(d1)      {
/**/
        case 0:
        break;
/* arsp */
        case 1:
        printf("A/Q not simulated0);
        erflg=ffff;
        break;
/* e1f1 */
        case 8:
        reg[e1]=reg[f1]=s1;
        break;
/* ef1g */
        case 9:
        reg[e1]=reg[f1]=reg[g0]=s1;
        break;
/* e1g0 */
        case 10:
        reg[e1]=reg[e0]=s1;
        break;
/* f1g0 */
        case 12:
        reg[f1]=reg[g0]=s1;
        break;
/* lf1n */
        case 23:
        reg[lf1+reg[lf1a]]=s1;
        break;
/* lf1u */
        case 24:
        reg[lf1+reg[lf1a]]=s1;
        reg[lf1a]=(reg[lf1a]+1)&07777;
        break;
/* lf1d */
        case 25:
        reg[lf1+reg[lf1a]]=s1;
        reg[lf1a]=(reg[lf1a]-1)&07777;
        break;
/* lf1a */
        case 26:
        reg[lf1+reg[lf1a]]=s1;
        reg[lf1a]=(reg[lf1a]+reg[indx0])&07777;
        break;
/* mm0 */
        case 32:
        reg[mem+efad]=reg[mmi0];
        break;
/* mm1 */
        case 33:
        reg[mem+efad+1]=reg[mmi1];
        break;
```

```
/* mm01 */
        case 34:
        reg[mem+efad]=reg[mmi0];
        reg[mem+efad+1]=reg[mmi1];
        break;
/* mm2 */
        case 35:
        reg[mem+efad+2]=reg[mmi2];
        break;
/* mm02 */
        case 36:
        reg[mem+efad]=reg[mmi0];
        reg[mem+efad+2]=reg[mmi2];
        break;
/* mm12 */
        case 37:
        reg[mem+efad+1]=reg[mmi1];
        reg[mem+efad+2]=reg[mmi2];
        break;
/* mma */
        case 38:
        reg[mem+efad]=reg[mmi0];
        reg[mem+efad+1]=reg[mmi1];
        reg[mem+efad+2]=reg[mmi2];
        break;
/* q */
        case 39:
        reg[multflg]=ffff;
        reg[q]=s1;
        break;
/* qef1 */
        case 40:
        reg[q]=reg[e1]=reg[f1]=s1;
        break;
/*  qefg */
        case 41:
        reg[q]=reg[e1]=reg[f1]=reg[g0]=s1;
        break;
/* qe1 */
        case 42:
        reg[q]=reg[e1]=s1;
        break;
/* qe1g */
        case 43:
        reg[q]=reg[e1]=reg[g0]=s1;
        break;
/* qf1 */
        case 44:
        reg[q]=reg[f1]=s1;
        break;
/* qf1g */
        case 45:
```

```
        reg[q]=reg[f1]=reg[g0]=s1;
        break;
/* qg0 */
        case 46:
        reg[q]=reg[g0]=s1;
        break;
/* tf1n */
        case 47:
        reg[tf1+reg[tf1aw]]=s1;
        break;
/* tf1u */
        case 48:
        reg[tf1+reg[tf1aw]]=s1;
        reg[tf1aw]=(reg[tf1aw]+1)&017;
        break;
/* tf1d */
        case 49:
        reg[tf1+reg[tf1aw]]=s1;
        reg[tf1aw]=(reg[tf1aw]-1)&017;
        break;
/* tf1c */
        case 50:
        reg[tf1+reg[tf1ar]]=s1;
        reg[tf1ar]=0;
        break;
/* t0ba */
        case 53:
        reg[tf1ar]=reg[tf1aw]=s1;
        break;
/**/
        default:
        reg[db1int[d1]]=s1;
        break;
        }
}

/* Extrac determine the place to store reg[mir+1]  */

extrac(reg)
        int     *reg;   {
        s0=s1=reg[mir+1];
        d0=(reg[mir])&0037600;
        d1=(reg[mir])&0000177;
        d0=finint(d0,d0int);
        d1=finint(d1,d1int);
        f7 = d0tab[d0];
        f8 = d1tab[d1];
        stord0(reg);
        stord1(reg);
}

/* This routine returns the index of an input integer 'it'
```

```
*  in the indicated table. Used for sources and destinations  */

finint(it,table)
        int     table[];        {
                int     i;
        for(i=0;table[i]!=it;i++)
        {       if(i>100) {printf("Source destination error0);
                                        erflg=ffff; return;  }
                if(table[i]==-1){printf("Source destination error0);
                                erflg=ffff;
                                return;
                                }
        }
        return(i);
}
/*
*
*        XXXX   X    X     X     XXXX    X    X        XXXX
*       X       X    X     X    X    X   X    X       X    X
*        XXXX   XXXXXX     X    X    X   XXXXXX       X
*            X  X    X     X    X    X   X    X       X
*       X    X  X    X     X    X    X   X    X XX    X    X
*        XXXX   X    X     X     XXXX    X    X XX     XXXX
*
*****************************************************************
*
*  SHIFT AND I/O EXECUTION ROUTINES
*
*****************************************************************
*
*  The includes and compilations proceedures are the same as
*  exec.c
*/
#include <stdio.h>
#include "incl2h.h"
#include "incl3h.h"
/**/

/* Shift execution routine  */

exshift(reg)
                int     *reg;   {

/* Determine if its a double or single shift, formulate opcodes
*  and call shift routines accordingly                          */

        if((reg[mir+1]&02000) != 0)     {
                f10="d";
                dbshft(reg);
                op= reg[mir]&0377;
                shft(reg,g0,g1,9);
                }
```

```
/* Single shift  */

        else    {
                f10="s";
                op=(reg[mir+1]>>2)&0377;
                shft(reg,e0,e1,7);
                op=((reg[mir]>>8)&077)|((reg[mir+1]&03)<<6);
                shft(reg,f0,f1,8);
                op=reg[mir]&0377;
                shft(reg,g0,g1,9);
                }
}

/* This routine execute the double shift and calls 'shcl' to execute
*  the clear opcodes  */

dbshft(reg)
        int     *reg;   {
                int     op1,inbit;
        op=((reg[mir]>>8)&077)|((reg[mir+1]&03)<<6);
        op1=(reg[mir+1]>>2)&0377;
        if(op!=op1) {  printf("Double shift error0); erflg=ffff;
                return;  }

/* Determine the input bit 'inbit'   */

        switch((op>>3)&07)      {
                case 0:
                f7=f8="zin";
                inbit=0;
                break;
/**/
                case 1:
                f7=f8="oin";
                inbit=1;
                break;
/**/
                case 2:
                f7=f8="cir";
                if(((op>>6)&03)==1) inbit=reg[f0]&01;
                else            inbit=(reg[e1]>>15)&01;
                break;
/**/
                case 3:
                f7=f8="sgn";
                if((inbit=(op>>6)&01)==1) inbit=(reg[e1]>>15)&01;
                break;
/**/
                case 4:
                f7=f8="a15";
                inbit=(reg[shcon]>>6)&01;
                break;
```

```
/**/
                case 5:
                f7=f8="n15";
                inbit=  (reg[shcon]>>6)&01;
                break;
/**/
                case 6:
                f7=f8="a31";
                inbit=(reg[shcon]>>7)&01;
                break;
/**/
                case 7:
                f7=f8="n31";
                inbit=  (reg[shcon]>>7)&01;
                break;
/**/
                default:
                f7=f8="What?";
                break;
                }

/* Shift right left or not at all  */

        switch((op>>6)&03)      {
/**/
                case 0:
                f7a=f8a="n";
                break;
/* Right  shift */
                case 1:
                f7a=f8a="r";
                reg[f0]=(077777&(reg[f0]>>1))|((reg[f1]&01)<<15);
                reg[f1]=(077777&(reg[f1]>>1))|((reg[e0]&01)<<15);
                reg[e0]=(077777&(reg[e0]>>1))|((reg[e1]&01)<<15);
                reg[e1]=(077777&(reg[e1]>>1))|(inbit<<15);
                break;
/* Left  shift */
                case 2:
                f7a=f8a="l";
                reg[e1]=(reg[e1]<<1)|((reg[e0]>>15)&01);
                reg[e0]=(reg[e0]<<1)|((reg[f1]>>15)&01);
                reg[f1]=(reg[f1]<<1)|((reg[f0]>>15)&01);
                reg[f0]=(reg[f0]<<1)|inbit;
                break;
/**/
                case 3:
                printf("Illegal shift0);
                erflg=ffff;
                break;
                }

/* Call general 'shcl' routines  */
```

```
                shcl(e0,e1,reg,&f7b);
                shcl(f0,f1,reg,&f8b);
}

/* General single shift routine op contains the opcode
*  of the register to be shifted, a0 and a1 contain
*  the indexs of the register and k keys where the
*  output f-field is coded                    */

shft(reg,a0,a1,k)
        int     *reg;   {
                int     inbit;
        char    *ff,*fg,*fh;

/* Determine input bit  */

        switch((op>>3)&07)      {
/**/
                case 0:
                ff="zin";
                inbit=0;
                break;
/**/
                case 1:
                ff="oin";
                inbit=1;
                break;
/**/
                case 2:
                ff="cir";
                switch((op>>6)&03)      {
                        case 1:
                        inbit=reg[a0]&01;
                        break;
                        case 2:
                        inbit=(reg[a1]>>15)&01;
                        break;
                        }
                break;
/**/
                case 3:
                ff="sgn";
                inbit=(op>>6)&(reg[a1]>>15)&01;
                break;
/**/
                case 4:
                ff="a15";
                inbit=(reg[shcon]>>6)&01;
                break;
/**/
                case 5:
                ff="n15";
```

```
                inbit=  (reg[shcon]>>6)&01;
                break;
/**/
                case 6:
                ff="a31";
                inbit=(reg[shcon]>>7)&01;
                break;
/**/
                case 7:
                ff="n31";
                inbit=  (reg[shcon]>>7)&01;
                break;
/**/
                default:
                ff="What?";
                break;
                }

/* Determine shift direction */

        switch((op>>6)&03)      {
                case 0:
                fg="n";
                break;
/* Right */
                case 1:
                fg="r";
                reg[a0]=(reg[a0]>>1)&077777;
        if((reg[a1]&01)!=0) reg[a0]=((reg[a0])&077777)|0100000;
                reg[a1]=(077777&(reg[a1]>>1))|(inbit<<15);
                break;
/* Left */
                case 2:
                fg="l";
                reg[a1]=reg[a1]<<1;
                if(reg[a0]<0) reg[a1]++;
                reg[a0]=(reg[a0]<<1)+inbit;
                break;
/**/
                default:
                printf("Bad shift field0);
                erflg=ffff;
                break;
                }

/* Clear the indicated registers  */

        shcl(a0,a1,reg,&fh);

/* Determine where the output strings are to be located  */

        switch(k)       {
```

```
/**/
        case 7:
        f7=ff;
        f7a=fg;
        f7b=fh;
        break;
/**/
        case 8:
        f8=ff;
        f8a=fg;
        f8b=fh;
        break;
/**/
        case 9:
        f9=ff;
        f9a=fg;
        f9b=fh;
        break;
        }
}
/*  Clear the indicated register bytes  */

shcl(a0,a1,reg,ff)
        int     *reg;
        int     *ff;    {
        switch(op&07)   {
                case 0:
                *ff="n";
                break;
/* Clear byte 0 */
                case 1:
                *ff="0";
                reg[a0]=reg[a0]&0177400;
                break;
/*/Claer byte 1 */
                case 2:
                *ff="1";
                reg[a0]=reg[a0]&0377;
                break;
/* Clear byte 2 */
                case 3:
                *ff="2";
                reg[a1]=reg[a1]&0177400;
                break;
/* Clear byte 3 */
                case 4:
                *ff="3";
                reg[a1]=reg[a1]&0377;
                break;
/* Clear bytes 0 and 1 */
                case 5:
                *ff="l";
```

```
                reg[a0]=0;
                break;
/* Clear bytes 2 and 3 */
                case 6:
                *ff="u";
                reg[a1]=0;
                break;
/* Clear all */
                case 7:
                *ff="a";
                reg[a0]=reg[a1]=0;
                break;
                }
}

/* Execute the DSA portion of the I/O; other I/O is not simulated  */

exio(reg)
        int     *reg;   {

/* Decode the source fields   */

        s0=reg[mir+1]&07700;
        s1=reg[mir+1]&077;
        s0=finint(s0,sr0int);
        s1=finint(s1,sr1int);
        f7=srctab[s0];
        f8=srctab[s1];
        source(reg);
        op=(reg[mir]>>12)&03;
        switch(op)      {
/**/
        case 0:
        printf("External interrupts not simulated0);
        erflg=ffff;
        break;
/**/
        case 1:
        printf("FP to FP transfers not simulated0);
        erflg=ffff;
        break;
/**/
        case 2: case 3:
        f9="ds";
        dsaio(reg,0,&f11);
        dsaio(reg,1,&f12);
        dsaio(reg,2,&f13);
        dsaio(reg,3,&f14);
        break;
        }
}
```

```
/* General dsaio routine. 'j' is the channel,ff is the output field */

dsaio(reg,j,ff)
        int     *reg;
        int     *ff;    {
        op=(reg[mir]>>(j*3))&07;
        switch(op)      {
/**/
        case 0:
        *ff="nop";
        break;
/* LOAD B */
        case 1:
        *ff="lb";
        reg[b+j]=reg[bus1]&017;
        break;
/* LOAD S */
        case 2:
        *ff="ls";
        reg[s+j]=reg[bus0];
        break;
/* LOAD S LOAD B */
        case 3:
        *ff="lsb";
        reg[b+j]=reg[bus1]&017;
        reg[s+j]=reg[bus0];
        break;
/* READ */
        case 4:
        reg[cycsh+j]=reg[cych]; reg[cycsl]=reg[cycl];
        *ff="ru";
        dsare(reg,j);
        reg[s+j]++;
        break;
/* WRITE */
        case 5:
        *ff="wu";
        dsawr(reg,j);
        reg[s+j]++;
        break;
/* LOAD S  READ */
        case 6:
        reg[cycsh+j]=reg[cych]; reg[cycsl]=reg[cycl];
        *ff="wl";
        reg[s+j]=reg[bus0];
        dsare(reg,j);
        reg[s+j]++;
        break;
/* LOAD S WRITE */
        case 7:
        *ff="wl";
        reg[s+j]=reg[bus0];
```

```
        dsawr(reg,j);
        reg[s+j]++;
        break;
        }
}

/* DSA read reads from the disk randomly, flags overflow
*  Memory banks are files 'bank0',bank1' etc.
*  Four memory banks are simulated, the read reads on all four channels
*  additional banks can be easily added as required    */

dsare(reg,j)
        int     *reg;   {
                int     i;
        switch(reg[b+j])        {
/**/
        case 0:

/* Flag cycle start for simulating wait state */

        reg[cycsl+j]=reg[cycl]; reg[cycsh+j]=reg[cych];

/* If file non-existant flag error  */

        if((b0ptr=fopen("bank0","r"))=='0') {
                        printf("Bank %d read error0,reg[b+j]);
                                        erflg=ffff; return;    }

/* Scan memory input into z-reg until memad 'reg[s+j]' is matched
*  If the end of the bank is reached, an error is flaged      */

 for(i=0;i<=reg[s+j];i++) if((fscanf(b0ptr,"%x",&reg[aqzin+j]))==-1)  {
        printf("Bank %d read overflow at %x hex0,reg[b+j],reg[s+j]);
                                        erflg=ffff; return;    }
        fclose(b0ptr);
        break;

/* Exactly the same as case 0 except uses 'bank1'  */

        case 1:
        reg[cycsl+j]=reg[cycl]; reg[cycsh+j]=reg[cych];
        if((b1ptr=fopen("bank1","r"))=='0') {
                        printf("Bank %d read error0,reg[b+j]);
                                        erflg=ffff; return;    }
 for(i=0;i<=reg[s+j];i++) if((fscanf(b1ptr,"%x",&reg[aqzin+j]))==-1)  {
        printf("Bank %d read overflow at %x hex0,reg[b+j],reg[s+j]);
                                        erflg=ffff; return;    }
        fclose(b1ptr);
        break;
/**/
        case 2:
        reg[cycsl+j]=reg[cycl]; reg[cycsh+j]=reg[cych];
```

```
        if((b2ptr=fopen("bank2","r"))=='0') {
                        printf("Bank %d read error0,reg[b+j]);
                                                erflg=ffff; return;    }
 for(i=0;i<=reg[s+j];i++) if((fscanf(b2ptr,"%x",&reg[aqzin+j]))==-1)  {
        printf("Bank %d read overflow at %x hex0,reg[b+j],reg[s+j]);
                                                erflg=ffff; return;    }
        fclose(b2ptr);
        break;
/**/
        case 3:
        reg[cycsl+j]=reg[cycl]; reg[cycsh+j]=reg[cych];
        if((b3ptr=fopen("bank3","r"))=='0') {
                        printf("Bank %d read error0,reg[b+j]);
                                                erflg=ffff; return;    }
 for(i=0;i<=reg[s+j];i++) if((fscanf(b3ptr,"%x",&reg[aqzin+j]))==-1)  {
        printf("Bank %d read overflow at %x hex0,reg[b+j],reg[s+j]);
                                                erflg=ffff; return;    }
        fclose(b3ptr);
        break;
/**/
        default:
        printf("Bank %d not simulated0);
        erflg=ffff;
        break;
        }
}

/* The DSA write simulates a random access memory on the disk
*  Their are no errors, if the file doesn't exist it is created  */

dsawr(reg,j)
        int     *reg;   {
                int     i,dum;
        reg[aqzin+j]=reg[bus1];
        switch(reg[b+j])        {
/**/
        case 0:

/* See if file exists, if not create it and append zeros until
*  the proper address is reached, then append the value in the
*  'aqzin' register       */

        if((b0ptr=fopen("bank0","r"))=='0')     {
                b0ptr=fopen("bank0","w");
                if(reg[s+j]==0) {
                        fprintf("%x0,reg[aqzin+j]);
                        return;
                        }
                fprintf(b0ptr,"00);
                wrtpst0(reg,j,1);
                return;
                }
```

```
/* If the file exists, determine whether the value can be
*  appended to the end of the file or whether it has to replace
*  an existing value */

      for(i=0;i<=reg[s+j];i++)        {
            if((fscanf(b0ptr," %x ",&dum))==-1)     {

/* The write address is past the end of the file so append  */

            wrtpst0(reg,j,i); return;              }
            }

/* At this point it has been determined that the value must be inserted
*  in an existing file.  The file is written on to a dummy file
* called ..buff and when the address is reached, the 'aqzin' value
* is written on the disk.  Then the rest of the file is written out  */

      fclose(b0ptr);
      b0ptr=fopen("bank0","r");
      bufptr=fopen("..buff","w");
      for(i=0;i<reg[s+j];i++) {
            fscanf(b0ptr,"%x",&dum);
            fprintf(bufptr,"%x0,dum);
            }

/* Scan for entry to be thrown out  */

      fscanf(b0ptr,"%x",&dum);
      fprintf(bufptr,"%x0,reg[aqzin+j]);
      while((fscanf(b0ptr,"%x",&dum)!=-1))
            fprintf(bufptr,"%x0,dum);
      fclose(bufptr);
      fclose(b0ptr);

/* Restore bank0  */

      system("mv ..buff bank0");
      break;

/* Bank 1 write */

      case 1:
      if((b1ptr=fopen("bank1","r"))=='0')     {
            b1ptr=fopen("bank1","w");
            if(reg[s+j]==0) {
                  fprintf("%x0,reg[aqzin+j]);
                  return;
                  }
            fprintf(b1ptr,"00);
            wrtpst1(reg,j,1);
            return;
            }
```

```
        for(i=0;i<=reg[s+j];i++)        {
                if((fscanf(b1ptr," %x ",&dum))==-1)     {
                wrtpst1(reg,j,i); return;               }
                }
        fclose(b1ptr);
        b1ptr=fopen("bank1","r");
        bufptr=fopen("..buff","w");
        for(i=0;i<reg[s+j];i++) {
                fscanf(b1ptr,"%x",&dum);
                fprintf(bufptr,"%x0,dum);
                }
        fscanf(b1ptr,"%x",&dum);
        fprintf(bufptr,"%x0,reg[aqzin+j]);
        while((fscanf(b1ptr,"%x",&dum)!=-1))
                fprintf(bufptr,"%x0,dum);
        fclose(bufptr);
        fclose(b1ptr);
        system("mv ..buff bank1");
        break;
/**/
        default:
        printf(" Bank %d not simulated0,reg[b+j]);
        break;
        }
}

/* This routine writes on the disk in 'bank0' when it is indicated
*  that an append is sufficient                                  */

wrtpst0(reg,j,k)
        int     *reg;   {
                int     i;
        fclose(b0ptr);
        b0ptr=fopen("bank0","a");
        for(i=k;i<reg[s+j];i++) fprintf(b0ptr,"00);
        fprintf(b0ptr,"%x0,reg[aqzin+j]);
        fclose(b0ptr);
}

/* This routine writes on the disk in 'bank1' when it is indicated
*  that an append is sufficient                                  */

wrtpst1(reg,j,k)
        int     *reg;   {
                int     i;
        fclose(b1ptr);
        b1ptr=fopen("bank1","a");
        for(i=k;i<reg[s+j];i++) fprintf(b1ptr,"00);
        fprintf(b1ptr,"%x0,reg[aqzin+j]);
        fclose(b1ptr);
}
# include<stdio.h>
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
oldseek(fildes,offset,ptrname)
int fildes,offset,ptrname;
{
/*************************************************************************/
/*if ptrname is 0, the pointer is set to offset.                         */
/*if ptrname is 1, the pointer is set  to  its  current  location        */
/* plus offset.                                                          */
/*if ptrname is 2, the pointer is set to the  size  of  the  file        */
/* plus offset.                                                          */
/*if ptrname is 3, 4 or 5, the meaning is as above for 0, 1 and 2        */
/* except that the offset is multiplied by 512.                          */
/*                                                                       */
/*If ptrname is 0 or 3, offset is unsigned, otherwise it is signed.      */
/*************************************************************************/
        lseek(fildes,(long) (offset),ptrname);
}
/*************************************************************************/
/*   This will create the help files to aid a used with the exec*/
/* ution of the programs.                                        */
/*************************************************************************/
```

APPENDIX 4

FLEXIBLE PROCESSOR MICRO-ASSEMBLER LISTING

```
/*
*
*       XXXXX      XX      XXXX       X             XXXX
*       X    X   X  X   X          XX            X    X
*       X    X  X    X   XXXX      X             X
*       XXXXX   XXXXXX        X    X             X
*       X       X    X  X    X     X       XX    X    X
*       X       X    X   XXXX    XXXXX     XX     XXXX
*
**********************************************************************
*
*   MICROCODE ASSEMBLER 1:01 9/10/79
*
*  CAUTION:  IF THIS SEEMS NOT TO WORK, CHANCES ARE ITS YOUR INPUT
*            FILE. ANSWER THE FOLLOWING QUESTIONS YES BEFORE
*            ATTEMPTING TO 'DEBUG' THE ASSEMBLER.
*                1. Is there a # at the end of the source listing?
*                2. Do all the instructions have the proper number
*                                of fields  ?
*                3. Are their no control characters hidden in the file?
*
*   If this doesn't cure the problem check the tables to make sure
*            you have proper input formats,  I could have sworn
*            many times that the assembler was bonkers when it was
*            my source. This assembler should not have bugs
*
**********************************************************************
*
* This program is compiled as 'cc pas1.c -lS'
*  The first line includes the new I/O package
*  And the FILE declarations reserve space for the I/O disk files  */
```

```
#include <stdio.h>
                FILE    *objptr,*srcptr;

/* Storage for labels and their values the symbol table */

        struct {
                char    id[6];
                int     add;
                } sym[100];

/* Globals, line number; table index; error flag; output words;
   ALU op; carry log/arit; memory address  */

                int     lnum,tabin,erf,out0,out1,out2,alu,car,madd;

/* Field inputs */
                char    name[7],add[4],sr0[5],sr1[5],d0[5],d1[5],
                        source[60];

/*********************************************************************
*                                                                    *
*      The following tables are the heart of the assembler           *
*       a mnemonic is read in to one of the field inputs and         *
*       this is "looked up" in one of the following tables           *
*       If a match is found, the index of the table at the matching  *
*       word is returned.  In many cases this is the opcode          *
*       associated with the mnemonic looked up.  If it is not,       *
*       that index is used as an index to another array of integer   *
*       constants whose values are the proper opcodes                *
*       If the input mnemonic is not found, a -1 is returned and     *
*       an error is flaged.                                          *
*                                                                    *
*********************************************************************/


/*
*       Type field lookup table
*/
        char    *typetab[] {
                        "org",
                        "equ",
                        "sh",
                        "io",
                        "tc",
                        "tr",
                        -1      };
/*
*       Condition code lookup table
*/
                char *contab[] {
                        "un",
                        "unn",
```

```
                        "uns",
                        "tn",
                        "tnn",
                        "tns",
                        "fn",
                        "fnn",
                        "fns",
                        "ad",
                        "adn",
                        "ads",
                        "up",
                        "upn",
                        "ups",
                        "tp",
                        "tpn",
                        "tps",
                        "fp",
                        "fpn",
                        "fps",
                        "io",
                        "ion",
                        "ios",
                        -1      };

  /* Lookups for opcodes in condition field*/

                int     conint[]        {0,0,1,2,2,3,4,4,5,6,6,7,
                                         8,8,9,10,10,11,12,12,13,
                                         14,14,15 };
/*
*       Index   lookup table
*/
        char    *idxtab[] {
                        "nop",
                        "in0",
                        "in1",
                        "in2",
                        "in3",
                        "dc0",
                        "dc1",
                        "dc2",
                        "dc3",
                        "cl0",
                        "cl1",
                        "cl2",
                        "cl3",
                        "ce0",
                        "ce1",
                        "cla",
                        -1      };
/*
*       Return jump table
```

```
*/
        char    *rjtab[] {
                        "np",
                        "jp",
                        "sr",
                        "df",
                        -1      };
/*
*       Multiply table
*/
        char    *multab[] {
                        "plpl",
                        "plpu",
                        "plql",
                        "plqu",
                        "pupl",
                        "pupu",
                        "puql",
                        "puqu",
                        "qlpl",
                        "qlpu",
                        "qlql",
                        "qlqu",
                        "qupl",
                        "qupu",
                        "quql",
                        "ququ",
                        -1      };
/*
*               ALU     table
*/
        char    *addtab[] {
                        "e",
                        "g",
                        "add",
                        "sub",
                        "e-g",
                        "fff",
                        "set",
                        "zro",
                        "e+e",
                        "e+1",
                        "e-1",
                        "and",
                        "or",
                        "e=g",
                        "oce",
                        "en",
                        "ocg",
                        "gn",
                        "sb1",
                        "xor",
```

```
                -1      };

/*  Opcode lookups  */

        int     addint[]        {037,032,051,06,06,043,043,023,
                                 054,0,057,033,036,046,020,020,
                                 025,025,046,026 };
/*  ALU tables for non-special lookups  */

        char    *cartab[] {
                        "ac","lc","an","ln",
                -1      };
        char    *funtab[] {
                        "0","1","2","3","4","5","6","7","8","9","a",
                        "b","c","d","e","f",
                        -1              };
/*
*       Shift decode table
*/
        char    *shoptab[] {
                        "nzin",
                        "noin",
                        "ncir",
                        "nsgn",
                        "na15",
                        "nn15",
                        "na31",
                        "nn31",
                        "rzin",
                        "roin",
                        "rcir",
                        "rsgn",
                        "ra15",
                        "rn15",
                        "ra31",
                        "rn31",
                        "lzin",
                        "loin",
                        "lcir",
                        "lsgn",
                        "la15",
                        "ln15",
                        "la31",
                        "ln31",
                        -1      };

/* Shift clear field decode  */

                char    *shctab[] {
                "0", "n", "0", "1", "2", "3", "l", "u", "a",
                        -1      };
        int     shcint[] {0,0,1,2,3,4,5,6,7 };
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
/*
*       Input Output tables
*/
                char    *ioptab[] {
                        "xi",
                        "fp",
                        "ds",
                        "aq",
                        "dp",
                        -1};
                int     iopint[]  { 0,1,2,2,3,3 };
/**/
                char    *iotab[] {
                        "nop",
                        "sx1",
                        "ck0",
                        "aqa",
                        "cx1",
                        "ck1",
                        "red",
                        "c01",
                        "wrt",
                        "cwa",
                        "rhr",
                        "ce0",
                        "whr",
                        "ce1",
                        "rhc",
                        "ceb",
                        "whc",
                        "nop",
                        "lb",
                        "ls",
                        "lsb",
                        "r",
                        "ru",
                        "w",
                        "wu",
                        "lr",
                        "rl",
                        "lw",
                        "wl",
                        -1 };
                int     ioint[]         {0,1,1,1,2,2,2,3,3,4,4,5,5,
                                         6,6,7,7,
                                        0,1,2,3,4,4,5,5,6,6,7,7  };
/*
*  Source codes
*/
                char    *srctab[] {
                        "nop",
                        "a0sw",
```

```
                    "a0rs",
                    "a0rz",
                    "a0",
                    "a1sw",
                    "a1rs",
                    "a1rz",
                    "a1",
/**/
                    "qrsp",
                    "arsp",
                    "ai0",
                    "ai1",
                    "ai2",
                    "ai3",
/**/
                    "bsr0",
                    "bsr1",
                    "f0",
                    "f1",
                    "if0n",
                    "if0u",
                    "if0d",
                    "if0c",
                    "if1n",
                    "if1u",
                    "if1d",
                    "if1c",
                    "imr0",
                    "imr1",
                    "intr",
                    "l0ad",
                    "lf0n",
                    "lf0u",
                    "lf0d",
                    "lf0a",
/**/
                    "l1ad",
                    "lf1n",
                    "lf1u",
                    "lf1d",
                    "lf1a",
                    "mar",
                    "mcr0",
                    "mult",
                    "tf0n",
                    "tf0u",
                    "tf0d",
                    "tf0c",
                    "t0ra",
                    "t0wa",
                    "tf1n",
                    "tf1u",
```

```
                        "tf1d",
                        "tf1c",
                        "t1ra",
                        "t1wa",
                        "z0",
                        "z1",
                        "z2",
                        "z3",
                        -1 };
/*
*       Bus 0 integers
*/
                int     sr0int[] {
        00              /* 0 */,
        0100            /* 40 */,
        02100           /* 440 */,
        04100           /* 840 */,
        06100           /* c40 */,
        0200            /* 80 */,
        02200           /* 480 */,
        04200           /* 880 */,
        06200           /* c80 */,
        0600            /* 180 */,
        02600           /* 580 */,
        0500            /* 140 */,
        02500           /* 540 */,
        04500           /* 940 */,
        06500           /* d40 */,
        01700           /* 3c0 */,
        03700           /* 7c0 */,
        0300            /* c0 */,
        02300           /* 4c0 */,
        01000           /* 200 */,
        03000           /* 600 */,
        05000           /* a00 */,
        07000           /* e00 */,
        01100           /* 240 */,
        03100           /* 640 */,
        05100           /* a40 */,
        07100           /* e40 */,
        01600           /* 380 */,
        03600           /* 780 */,
        07600           /* f80 */,
        0177777         /* ffff */,
        01400           /* 300 */,
        03400           /* 700 */,
        05400           /* b00 */,
        07400           /* f00 */,
        06700           /* dc0 */,
        01500           /* 340 */,
        03500           /* 740 */,
        05500           /* b40 */,
```

```
        07500           /* f40 */,
        0700            /* 1c0 */,
        05700           /* bc0 */,
        0400            /* 100 */,
        01200           /* 280 */,
        03200           /* 680 */,
        05200           /* a80 */,
        07200           /* e80 */,
        02700           /* 5c0 */,
        04700           /* 9c0 */,
        01300           /* 2c0 */,
        03300           /* 6c0 */,
        05300           /* ac0 */,
        07300           /* ec0 */,
        02400           /* 500 */,
        04400           /* 900 */,
        0500            /* 140 */,
        02500           /* 540 */,
        04500           /* 940 */,
        06500           /* d40 */          };
/*
*       Bus one integers
*/
                int     sr1int[] {
        00              /* 0 */,
        01              /* 1 */,
        021             /* 11 */,
        041             /* 21 */,
        061             /* 31 */,
        02              /* 2 */,
        022             /* 12 */,
        042             /* 22 */,
        062             /* 32 */,
        06              /* 6 */,
        026             /* 16 */,
        05              /* 5 */,
        025             /* 15 */,
        045             /* 25 */,
        065             /* 35 */,
        017             /* f */,
        037             /* 1f */,
        03              /* 3 */,
        023             /* 13 */,
        010             /* 8 */,
        030             /* 18 */,
        050             /* 28 */,
        070             /* 38 */,
        011             /* 9 */,
        031             /* 19 */,
        051             /* 29 */,
        071             /* 39 */,
        016             /* e */,
```

```
        036             /* 1e */,
        076             /* 3e */,
        067             /* 37 */,
        014             /* c */,
        034             /* 1c */,
        054             /* 2c */,
        074             /* 3c */,
        0177777         /* ffff */,
        015             /* d */,
        035             /* 1d */,
        055             /* 2d */,
        075             /* 3d */,
        0177777         /* ffff */,
        057             /* 2f */,
        04              /* 4 */,
        012             /* a */,
        032             /* 1a */,
        052             /* 2a */,
        072             /* 3a */,
        027             /* 17 */,
        047             /* 27 */,
        013             /* b */,
        033             /* 1b */,
        053             /* 2b */,
        073             /* 3b */,
        024             /* 14 */,
        044             /* 24 */,
        05              /* 5 */,
        025             /* 15 */,
        045             /* 25 */,
        065             /* 35 */           };
/*
*       Destination codes bus 0
*/
                char    *d0tab[] {
                        "nop",
                        "brg0",
                        "e0",
                        "e0f0",
                        "e0g1",
                        "ef0g",
                        "f0",
                        "f0g1",
                        "g1",
                        "imr0",
                        "imr1",
                        "intr",
                        "lf0n",
                        "lf0u",
                        "lf0d",
                        "lf0a",
                        "l0ad",
```

```
                "mar",
                "marc",
                "mmi0",
                "mmi1",
                "mmi2",
                "p",
                "pe0",
                "pe0g",
                "pef0",
                "pefg",
                "pf0",
                "pf0g",
                "pg1",
                "tf0n",
                "tf0u",
                "tf0d",
                "tf0c",
                "t0ra",
                "t0wa",
                "t0ba",
                "if0a",
                -1      };
        int     d0int[] {
00              /* 0 */,
0400            /* 100 */,
04200           /* 880 */,
024200          /* 2880 */,
014200          /* 1880 */,
034200          /* 3880 */,
020200          /* 2080 */,
030200          /* 3080 */,
010200          /* 1080 */,
023200          /* 2680 */,
027200          /* 2e80 */,
037200          /* 3e80 */,
03000           /* 600 */,
07000           /* e00 */,
013000          /* 1600 */,
017000          /* 1e00 */,
027000          /* 2e00 */,
035600          /* 3b80 */,
03400           /* 700 */,
021600          /* 2380 */,
025600          /* 2b80 */,
031600          /* 3380 */,
01000           /* 200 */,
05000           /* a00 */,
015000          /* 1a00 */,
025000          /* 2a00 */,
035000          /* 3a00 */,
021000          /* 2200 */,
031000          /* 3200 */,
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
        011000          /* 1200 */,
        02400           /* 500 */,
        06400           /* d00 */,
        012400          /* 1500 */,
        016400          /* 1d00 */,
        026400          /* 2d00 */,
        032400          /* 3500 */,
        036400          /* 3d00 */,
        026000          /* 2c00 */          };
/*
*       Bus one destination codes
*/
                char    *d1tab[] {
                        "nop",
                        "arsp",
                        "brg1",
                        "cmr0",
                        "cmr1",
                        "cmr2",
                        "cmr3",
                        "e1",
                        "e1f1",
                        "ef1g",
                        "e1g0",
                        "f1",
                        "f1g0",
                        "g0",
                        "icr0",
                        "icr1",
                        "icr2",
                        "icr3",
                        "idx0",
                        "idx1",
                        "idx2",
                        "idx3",
                        "if1a",
                        "lf1n",
                        "lf1u",
                        "lf1d",
                        "lf1a",
                        "l1ad",
                        "mcr0",
                        "mcr1",
                        "mcr2",
                        "mcr3",
                        "mm0",
                        "mm1",
                        "mm01",
                        "mm2",
                        "mm02",
                        "mm12",
                        "mma",
```

```
                "q",
                "qef1",
                "qefg",
                "qe1",
                "qe1g",
                "qf1",
                "qf1g",
                "qg0",
                "tf1n",
                "tf1u",
                "tf1d",
                "tf1c",
                "t1ra",
                "t1wa",
                "t1ba",
                -1      };
        int     d1int[] {
00              /* 0 */,
0166            /* 76 */,
02              /* 2 */,
015             /* d */,
035             /* 1d */,
055             /* 2d */,
075             /* 3d */,
021             /* 11 */,
0121            /* 51 */,
0161            /* 71 */,
061             /* 31 */,
0101            /* 41 */,
0141            /* 61 */,
041             /* 21 */,
0117            /* 4f */,
0137            /* 5f */,
0157            /* 6f */,
0177            /* 7f */,
017             /* f */,
037             /* 1f */,
057             /* 2f */,
077             /* 3f */,
0130            /* 58 */,
014             /* c */,
034             /* 1c */,
054             /* 2c */,
074             /* 3c */,
0134            /* 5c */,
0116            /* 4e */,
0136            /* 5e */,
0156            /* 6e */,
0176            /* 7e */,
027             /* 17 */,
047             /* 27 */,
067             /* 37 */,
```

```
        0107            /* 47 */,
        0127            /* 57 */,
        0147            /* 67 */,
        0167            /* 77 */,
        04              /* 4 */,
        0124            /* 54 */,
        0164            /* 74 */,
        024             /* 14 */,
        064             /* 34 */,
        0104            /* 44 */,
        0144            /* 64 */,
        044             /* 24 */,
        012             /* a */,
        032             /* 1a */,
        052             /* 2a */,
        072             /* 3a */,
        0132            /* 5a */,
        0152            /* 6a */,
        0172            /* 7a */               };
/**/
main() {
/**/
                int     adpar,i,hexnum;
                char    it[8];

/* Prompt for source and set input buffer pointer  */

        printf("Source?  ");
        scanf("%s",source);
        if((srcptr=fopen(source,"r"))=='0')  {
                printf("Cannot open source file0);
                return;  }
        objptr=fopen("object","w");
        erf=madd=tabin=0;
        lnum=1;
/*
* Read and assemble symbol table on first pass
*/
        while((name[0]=getc(srcptr))!='0') {
                lnum++;

/* Action is based on first character in line
*  # says end of file, go to pass two
*  * says comment
*  space or tab says instruction, no label
*  anything else is a label                     */

                switch(name[0]) {
/* End of file, call pass two  */
                        case'#':
                        newline();
                        if(erf==0)  {
```

```
                                pas2();
                                fprintf(objptr,"#0);    }
                        return;
                        break;
/*  Comment, kick line pointer only  */
                        case'*':
                        newline();
                        break;
/* If its an origin input address, else kick address */
                        case' ': case'
                if(fscanf(srcptr,"org %x",&hexnum)!=0)
                                madd=hexnum;
                else            madd++;
                        newline();
                        break;

/* Default to build the label from the first characters ( until
*  space or tab is encountered)                                    */
                        default:
                        for(i=1;i<6;i++) {
                                name[i]=getc(srcptr);
                                if(name[i]==' '||name[i]=='
                                        break;
                                }
                        name[i]='0';
                        if(fscanf(srcptr,"equ %x",&hexnum)!=0)
                                adpar=hexnum;
                else            adpar=madd++;
/*****************       adpar--;                ***TEST *****/
                        adtab(adpar);
                        newline();
                        break;
                }
        }
}
newline() {
        while(getc(srcptr)!='0);
}

/* Add global character "name" to symbol table with value adpar   */

adtab(adpar)
        int     *adpar; {
                        int     i;
        cktab();
        for(i=0;(sym[tabin].id[i]=name[i])!='0';i++);
        sym[tabin].add=adpar;
        tabin++;
}

/* Check to see if name is in table  */
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
cktab() {
                int     i,j;
        for(i=0;i<=tabin;i++) {
                for(j=0;name[j]!='0';j++)
                        if(name[j]!=sym[i].id[j])
                                goto nomatch;
        printf("Double Definition %s Line %d0,name,lnum);
                erf=1;
nomatch:        ;
                }
}

/*********************************************************************
*
*  PASS TWO !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*
*
*********************************************************************/
pas2(){
                int     i;
                char    f1[5];
                lnum=1; madd=0;

/*  Rewind source,  output symbol table to file "object"  */

        rewind(srcptr);
        printf(" Pass one complete:0    0");
        fprintf(objptr,"        Symbol Table0");
        for(i=0;i<=tabin;i++)
                fprintf(objptr,"%s      %x0,sym[i].id,sym[i].add);
        fprintf(objptr,"##0);
        while((name[0]=getc(srcptr))!='0') {

/* Key on first symbol, same action as in pass one  */

        switch(name[0]) {
/*  # says done  */
                case '#':
                return;
                break;
/*  Comment  */
                case '*':
                break;
                default:
                switch(name[0]) {
/*  Instruction, not labeled  */
                        case ' ':       case'
                        break;
/* Label says lookup in symbol table  */
                        default:
                        for(i=1;i<6;i++) {
                                name[i]=getc(srcptr);
                                if(name[i]==' '||name[i]=='
```

```
                                        break;
                                    }
                                name[i]='0';
                                break;
                        }

/* Decode first field. org-> read next field into madd
                       equ-> ignore
                       valid field-> assemble it  */

                        fscanf(srcptr,"%s",f1);
                        switch(lookup(f1,typetab)) {
/*  ORG  */
                                case 0:
                                fscanf(srcptr,"%x",&madd);
                                break;
/* EQU */
                                case 1:
                                break;
/* SH */
                                case 2:
                                fprintf(objptr,"%x      ",madd);
                                shift();
                                madd++;
                                break;
/* I/O */
                                case 3:
                                fprintf(objptr,"%x      ",madd);
                                inout();
                                madd++;
                                break;
/* TC */
                                case 4:
                                fprintf(objptr,"%x      ",madd);
                                trancon();
                                madd++;
                                break;
/* TR */
                                case 5:
                                fprintf(objptr,"%x      ",madd);
                                transfer();
                                madd++;
                                break;
/**/
                                default:
                                printf("Invalid TYPE field  Line %d0,lnum);
                                break;
                                }
                        break;

        newline();
        lnum++;
```

```
        }
}

/* General table lookup routine
*
*  This routine returns the index of the item in the
* specified table                */

lookup(item,chartab)
        char    *item;
        char    **chartab;      {
                int     i,j,r,k;
        for(i=0;chartab[i]!=-1;i++) {
        for(j=0;(r=chartab[i][j]) == item[j] && r != '0';j++);
        if(r == item[j])
                return(i);
        }
        return(-1);
}

/* Decode shift instruction  */

shift() {
                char    esh[5],fsh[5],gsh[5],pre[5];
                int     lo6f,hi2f;
        dec2();
        out2=out2|0100000;
        fscanf(srcptr,"%s%s%s%s%s",add,esh,fsh,gsh,pre);
        hi2f=shdec(fsh);
        lo6f=hi2f&077;
        hi2f=hi2f>>6;
        addec();
/**/
        out1=(alu<<12)|(predec(pre)<<10)|(shdec(esh)<<2)|hi2f;
/**/
        out0=(car<<14)|(lo6f<<8)|shdec(gsh);
        fprintf(objptr,"%x      %x      %x0,out0,out1,out2);
}

/* Decode the input output instruction  */

inout() {
                char    iop[5],ch3[5],ch2[5],ch1[5],ch0[5];
        dec2();
        out2=out2|0140000;
        fscanf(srcptr,"%s%s%s",add,sr0,sr1);
        addec();
        dec1();
        fscanf(srcptr,"%s%s%s%s%s",iop,ch0,ch1,ch2,ch3);
 out0=(iopdec(iop)<<12)|(car<<14)|(iodec(ch3)<<9)|(iodec(ch2)<<6)
                        |(iodec(ch1)<<3)|iodec(ch0);
        fprintf(objptr,"%x      %x      %x0,out0,out1,out2);
```

```
}

/* Decode the transfer constant instruction */

trancon() {
                char    lable[10];
        dec2();
        out2=out2|040000;
        fscanf(srcptr,"%s",lable);

/*  Determine if constant is hex or a label
*   $ means hex number but the $ itself must be delimeted
*   by spaces tabs or returns                                   */

        switch(lable[0]) {
                case '*':
                out1=0;
                break;
                case '$':
                if(fscanf(srcptr,"%x",&out1)==0)
                printf("Invalid HEX CONSTANT   Line %d0,lnum);
                break;
                default:
                out1=0;
                findx(lable);
                if(out1==0)
                printf("Invalid HEX LABLE   Line %d0,lnum);
                break;
        }
        fscanf(srcptr,"%s%s",d0,d1);
        car=0;
        dec0();
        fprintf(objptr,"%x      %x      %x0,out0,out1,out2);
}

/*  Decode the transfer  instruction  */

transfer() {
        dec2();
        fscanf(srcptr,"%s%s%s%s%s",add,sr0,d0,sr1,d1);
        addec();
        dec1();
        dec0();
        fprintf(objptr,"%x      %x      %x0,out0,out1,out2);
}

/* Decode the ALU; carry log/arit returned if car; funcion number in
*  global variable alu                                          */

addec() {
                char    carcd[2];
                int     ret;
```

```
                car=alu=0;
        if((add[0]=='*')&&(add[1]=='0'))    return;

/*  See if item add[] is in the addtab, it will be
*   only if it is a special mnemonic ( Table 5-1 )  */

        if((ret=lookup(add,addtab))!= -1) {
                car=addint[ret]>>4;
                alu=addint[ret]&017;
                return;
                }

/*  Not a special mne. so decode in two parts, car-log/arit
*   and the funtion number                                  */

        carcd[0]=add[0];
        carcd[1]=add[1];
        carcd[2]='0';
        if((car=lookup(carcd,cartab))==-1) {
                printf("Invalid ALU field  Line %d0,lnum);
                return(-1);
                }
        carcd[0]=add[2];
        carcd[1]='0';
        if((alu=lookup(carcd,funtab))!=-1)
                return;
        printf("Invalid ALU field  Line %d0,lnum);
}

/*  General shift decode function
*   Takes shop, the input character mnemonic and
*  Returns the proper numerical opcode          */

shdec(shop)
                char    *shop; {
                int     ret,ret1;
                char    ci[2];
        if((shop[0]=='*')&&(shop[1]=='0')) return(0);
        ci[0]=shop[4];
        ci[1]='0';
        shop[4]='0';
        if((ret=lookup(shop,shoptab)) != -1)
        if((ret1=lookup(ci,shctab)) != -1)
                return((ret<<3)|(shcint[ret1]));
        else
                        printf("Invalid SHIFT field  Line %d0,lnum);
        else
                        printf("Invalid SHIFT field  Line %d0,lnum);
}

/*  Decode the shift precision field
*/
```

```
predec(pre)
        char    *pre ; {
        if((pre[0]=='*')&&(pre[1]=='0')) return(0);
        if(pre[0]=='s'&&pre[1]=='0')
                return(0);
        if(pre[0]=='d'&&pre[1]=='0')
                return(1);
        printf("Invalid SHIFT field  Line %d0,lnum);
}

/* Build word two for all instructions  */

dec2() {
        char    con[4],idx[4],rj[3],mult[5];
        fscanf(srcptr,"%s%s%s%s",con,idx,rj,mult);
 out2=(condec(con)<<10)|(idxdec(idx)<<6)|(rjdec(rj)<<4)|multdec(mult);
}

/*  Decode condition field  */

condec(con)
                char    *con; {
                int     ret;
        if((con[0]=='*')&&(con[1]=='0')) return(0);
        if((ret=lookup(con,contab)) != -1)
                return(conint[ret]);
        else
                        printf("Invalid CND field  Line %d0,lnum);
}

/* Decode index field  */

idxdec(idx)
                char    *idx; {
                int     ret;
        if((idx[0]=='*')&&(idx[1]=='0')) return(0);
        if((ret=lookup(idx,idxtab)) != -1)
                return(ret);
        else
                        printf("Invalid IDX field  Line %d0,lnum);
}

/* Decode return jump field  */

rjdec(rj)
                char    *rj; {
                int     ret;
        if((rj[0]=='*')&&(rj[1]=='0')) return(0);
        if((ret=lookup(rj,rjtab)) != -1)
                return(ret);
        else
                        printf("Invalid RJ field  Line %d0,lnum);
```

```
}

/* Decode multiply field */

multdec(mult)
                char    *mult; {
                int     ret;
        if((mult[0]=='*')&&(mult[1]=='0')) return(0);
        if((ret=lookup(mult,multab)) != -1)
                return(ret);
        else
                        printf("Invalid MULT field  Line %d0,lnum);
}

/* Decode input output operation field  */

iopdec(iop)
                char    *iop; {
                int     ret;
        if((iop[0]=='*')&&(iop[1]=='0')) return(0);
        if((ret=lookup(iop,ioptab)) != -1)
                return(iopint[ret]);
        else
                        printf("Invalid IOP field  Line %d0,lnum);
}
/* Decode general I/O field */

iodec(io)
                char    *io; {
                int     ret;
        if((io[0]=='*')&&(io[1]=='0')) return(0);
        if((ret=lookup(io,iotab)) != -1)
                return(iointEret]);
        else
                        printf("Invalid I0 field  Line %d0,lnum);
}

/* Build word 1 for sh and tr  */

dec1() {
        out1=(alu<<12)|(sr0dec())|sr1dec();
}

/* Decode source codes  */

sr0dec() {
                int     ret;
        if((sr0[0]=='*')&&(sr0[1]=='0')) return(0);
        if((ret=lookup(sr0,srctab)) != -1)
                if(sr0int[ret]!=-1)
                return(sr0int[ret]);
        else
```

```
        printf("Invalid  SOURCE (Bus 0) field  Line %d0,lnum);
        else
        printf("Invalid  SOURCE (Bus 0) field  Line %d0,lnum);
}

sr1dec() {
                int     ret;
        if((sr1[0]=='*')&&(sr1[1]=='0')) return(0);
        if((ret=lookup(sr1,srctab)) != -1)
                if(sr1int[ret]!=-1)
                return(sr1int[ret]);
        else
        printf("Invalid SOURCE (Bus 1) field  Line %d0,lnum);
        else
        printf("Invalid SOURCE (Bus 1) field  Line %d0,lnum);
}

/*  Find a label in the symbol table, return its value.  */

findx(lable)
        char    *lable; {
                int     i,j;
        for(i=0;i<=tabin;i++) {
                for(j=0;lable[j]!='0';j++)
                        if(lable[j]!=sym[i].id[j])
/* As soon as there is no match, try next entry */
                                goto nomatch;
                out1=sym[i].add;
nomatch:                ;
                }
}

/* Build word 0 for tr and tc instructions  */

dec0() {
        out0=(car<<14)|d0dec()|d1dec();
}

/* Decode destinations   */

d0dec() {
                int     ret;
        if((d0[0]=='*')&&(d0[1]=='0')) return(0);
        if((ret=lookup(d0,d0tab)) != -1)
                return(d0int[ret]);
        else
        printf("Invalid DESTINATION (Bus 0) field  Line %d0,lnum);
        printf("It got to here.  the value for d0 is %c 0,d0[0]);
}

d1dec() {
                int     ret;
```

```
if((d1[0]=='*')&&(d1[1]=='0')) return(0);
if((ret=lookup(d1,d1tab)) != -1)
        return(d1int[ret]);
else
printf("Invalid DESTINATION (Bus 1) field  Line %d0,lnum);
```

APPENDIX 5

IMPLEMENTATION OF THE MAXIMUM LIKELIHOOD CLASSIFIER ON A FLEXIBLE PROCESSOR

A. Initialization of the FPs by the Host Computer

B. Interrupt Routine for Flexible Processor

C. Overview of Maximum Likelihood Classifier Flexible Processor Algorithm

D. Flowchart of Floating Point Addition Routine

E. Flowchart of Floating Point Multiplication Routine

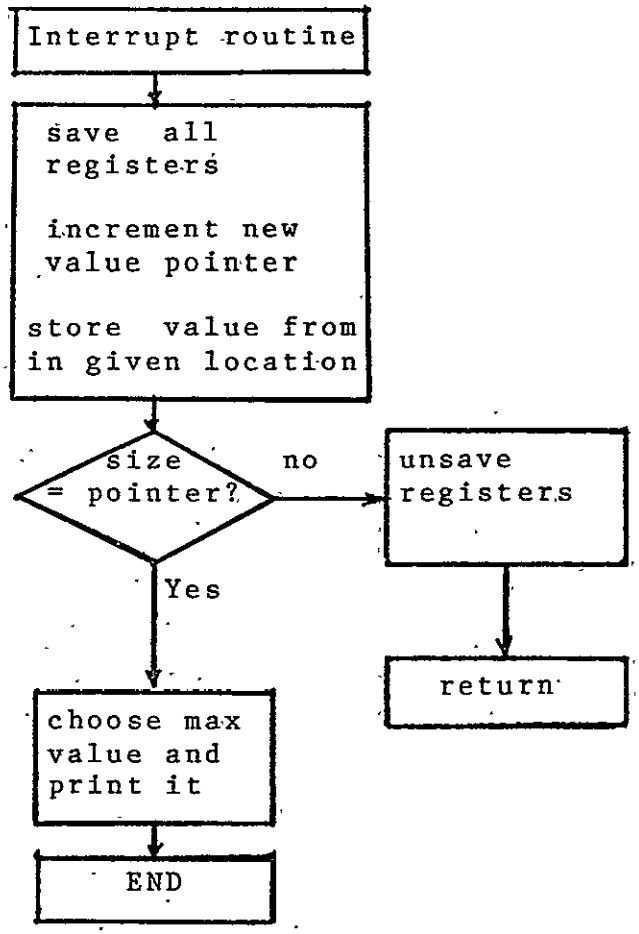
F. Flowchart of Floating Point Compare Routine

G. Actual Flexible Processor Program for Maximum Likelihood Classification

A. INITIALIZATION OF THE FPs BY THE HOST COMPUTER

1. Initialize memory to zeroes.
2. Send FP size, p, sigma, X, and U.
3. Calculate det(sigma)
4. Calculate inv(sigma)
5. Calculate ln(det(sigma))
6. Calculate ln(p(w))
7. Send FP ln(p(w)) - ln(det(sigma)) * .5
8. Send FP inv(sigma)

B. INTERRUPT ROUTINE FOR FLEXIBLE PROCESSOR



C.1. Load Data Into FP.

1) Zero all registers. This includes all index registers, index compare registers, large file address registers, maintenence compare registers and temporary file address (both read and write) registers.

2) Read the first number and store it in register F.

3) Copy the number stored in the F register into the index compare registers number 0 and 1. (This number is the dimension of sigma.)

4) Load all conditions. (This means that the index compare registers are going to test for equality to n.) Index register three will check for equality to zero.

5) Test and increment Index register three. If it is not equal to zero read a number, load it into the F register.

6) Move the F register to temporary file zero while incrementing the write counter.

7) If index register 0 does not equal n, go to step 5.

8) Zero all index registers while moving n to the P register of the multiply while trapping interrupts. (This can be done using the "sr" command.)

9) With interrupts trapped, move multiply output to condition register 2. (This means that the condition registers are now set to check for index registers 0 and 1 equal to n, index register 2 equal to n squared and index register 3 equal to zero.)

10) Test and increment index register 2. If it is n squared, exit.

11) Read a number, store it in large file zero, while simultaneously incrementing its address buffer.

12) Jump to step 10.

C.2. Storage Format

The Storage format used in the processing scheme is as follows:

Temporary files		Large Files	
	n	Sigma[1,1]	First Covariance
	Hold	Sigma[2,1]	Matrix.
normalixed data	X[1,1]	:	
vector for	X[1,2]	Sigma[n,1]	
class one	X[1,3]	Sigma[1,2]	
	X[1,4]	:	
normalized data	Y[1,1]	Sigma[2,n]	
vector for	Y[1,2]	:	
class two	Y[1,3]	Sigma[n,n]	
	Y[1,4]	:	
		Sigma[n,n]	
		:	
		Sigma[1,1]	Second Covariance
		Sigma[2,1]	Matrix
		:	
		Sigma[n,1]	
		Sigma[1,2]	
		:	
		Sigma[2,1]	
		:	
		Sigma[2,n]	
		:	
		Sigma[n,n]	
	mean vector for	U[1,1]	
	class one	U[1,2]	
		U[1,3]	
		U[1,4]	
	mean vector for	V[1,1]	
	class two	V[1,2]	
		V[1,3]	
		V[1,4]	
ln(p(w))-ln(det(sigma))*.5		K[1]	
		K[2]	

C.3. First Matrix Multiplication

1) Initialize all registers. Move 1 to the read address of temporary file zero. Zero all other index registers, large file addresses, temporary file addresses.

2) Move temporary file 0 to the E register (while incrementing the read address pointer.)

3) Move large file 0 to the G register (while incrementing the address pointer.)

4) Call floating point multiply routine.

5) Store result in temporary file 1, while increasing the write pointer.

6) If index 0 is n, jump to the subroutine called sum.

7) If index 1 is n, jump to the next multiply routine.

8) Increment index reg 0.

9) Go to step 1.

10) Increment index 1 by 1.

11) Zero F register. (This is used as the accumulater for the floating point add.)

12) Zero index register 0.

13) Zero temporary file 1 read address.

14) Test and increment index 0. If it = n, go to step 16.

15) Call floating point add subroutine. (40 cycles.) (this routine has been modified to increment the temporary file 1 read pointer as it goes along, so this is not necessary.)

16) Go to step 14.

17) Temporary file 0 pointer = 1.

18) Store f in large file 1 (while incrementing the pointer.) (F contains the result of the n floating point adds.)

19) Return to calling routine

C.4. Second Matrix Multiplication

1) Zero all pointers to large files and temporary file 1 address.

2) Write a 0 to temporary file address 0.

3) Transfer temporary file zero memory location 0 to index register 3.

4) Test and decrement register 3. If zero go to wrap up.

5) While incrementing the pointer to temporary file 0, move the contents to the E register.

6) While incrementing the pointer to large file 1, move the contents to the G register.

7) Call the floating point multiplication routine.

8) Call the floating point add routine.

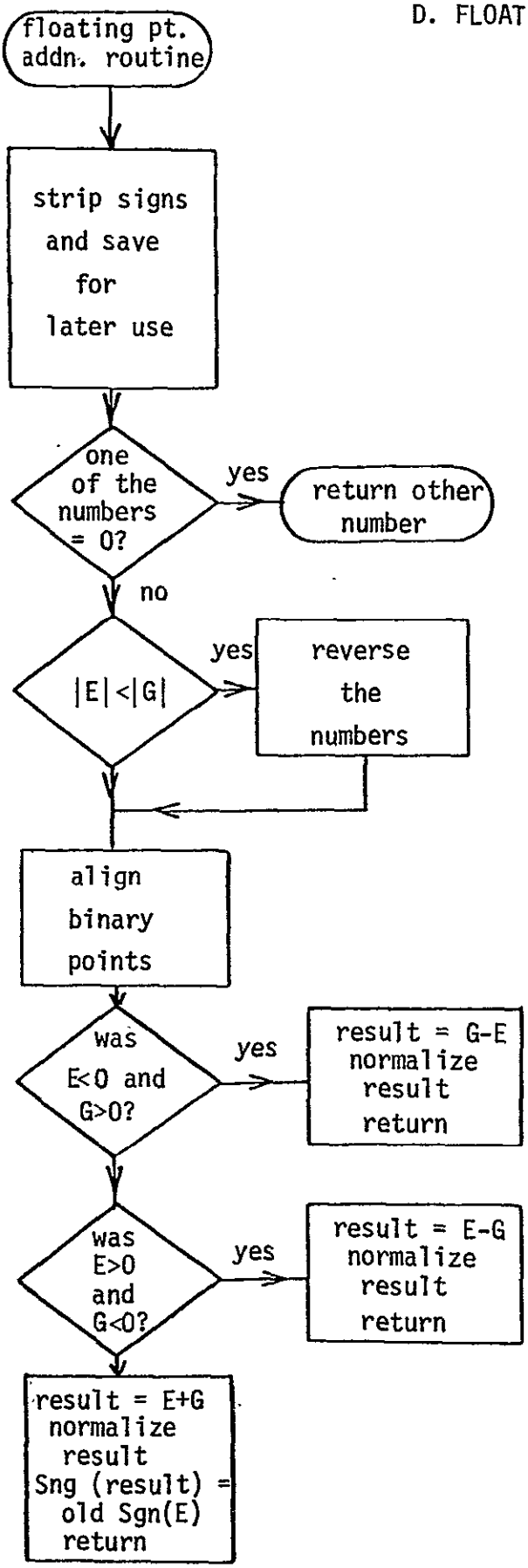
9) Send the result to temporary file 1.

10) Go to step 4.

ORIGINAL PAGE IS
OF POOR QUALITY

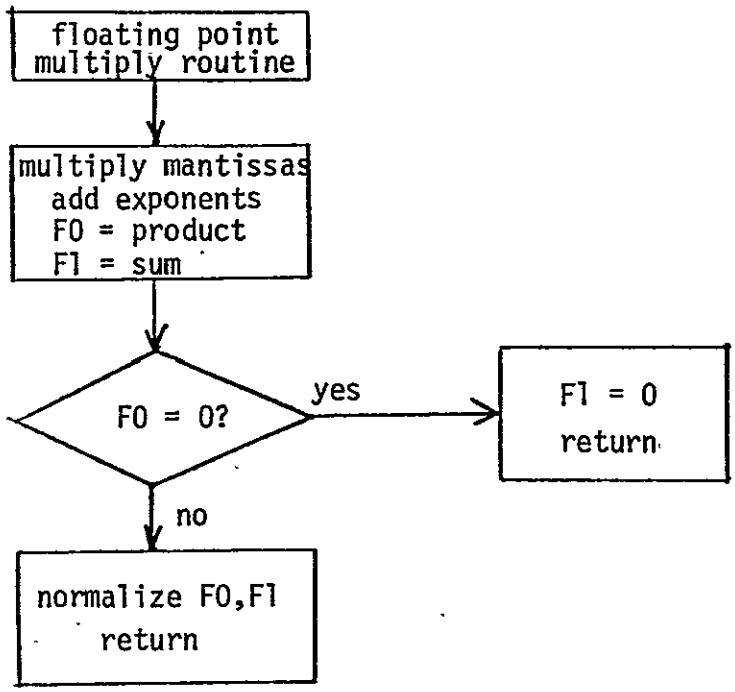
D. FLOATING POINT ADDITION ROUTINE

F = E + G

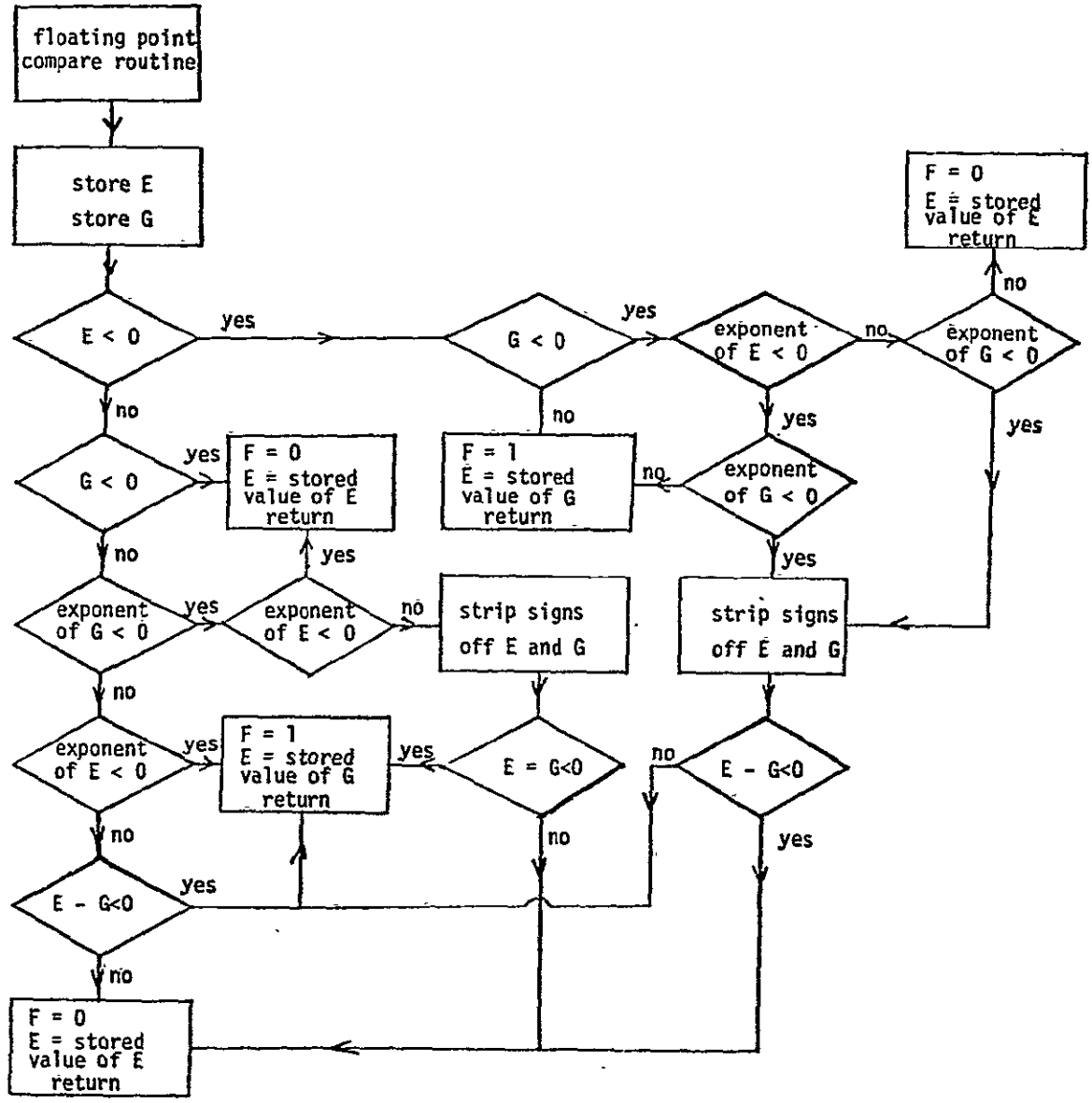


E. FLOATING POINT MULTIPLICATION ROUTINE

F = E + G



F. FLOATING POINT COMPARE ROUTINE



E = max (E,G)

F = 1 if E = G

= 0 if E ≠ G

G. Actual Flexible Processor Program for Maximum Likelihood Classification

```
 org    001a
*          Average time: 612 cycles per pixel (down 10%)
*          min time:     21e cycles per pixel (down 30%)
*
*************************************************************
*****BAYES MAXIMUM LIKELIHOOD CLASSIFIER VER. 021580  2:50****
*****For TWO PIXELS.......................................****
*************************************************************
     tc *   *   *  *          fplr              mar         nop
     tc *   *   *  *         $ 0000             *           *
*************************************************************
*   first interrupt routine. This routine handles the inter- *
*   rupt to load the covariance matrices, the mean vectors   *
*   and the data vector.                                     *
*************************************************************
 org    001e
     tc *   *   *  *          vinr              mar         nop
     tc *   *   *  *         $ 0000             *           *
*************************************************************
* this routine will handle the interrupt when the host just  *
* needs to enter the data vector.                            *
*************************************************************
 org    00fe
     tc *   *   *  *         $ 0000             *           *
*************************************************************
* there values are to be loaded into compare register 3.     *
* These will test the respective registers for inequality to *
* their compare registers.                                   *
*************************************************************
     tc *   cla *  *         $ 000a             t0wa        t1wa
     tc *   *   *  *         $ 0001             tf0n        tf1n
*************************************************************
*  this will clear all of the index registers and zero the   *
*  temporary file write addresses.                           *
*************************************************************
     tc *   *   *  *         $ 0000             nop         cmr0
*************************************************************
*  This will clear the temporary file 0 read address and the *
*  condition register to prevent spurious results.           *
*************************************************************
     tc *   *   *  *         $ 0000             nop         cmr2
*************************************************************
* this will zero the other condition register and the temp   *
* file read address. The dimension of the incoming data is   *
* assumed to be 4X4.  If thematic mapper data is to be used  *
* the matrix will be five by five.                           *
*************************************************************
     tc *   *   *  *         $ 0004             f0          icr3
```

```
*************************************************************
* This will store N in the index compare registers,         *
*************************************************************
     tr *   *   *  *    *                    nop   nop   f0    icr1
     tc *   *   *  *         $ 0010           nop         icr2
*************************************************************
* this is just setting up the counter variables for the loop.*
*************************************************************
wait tc *   *   *  *         wait             mar         nop
     tc *   *   *  *         $ 0000           brg0        brg1
     tr *   *   *  pupu *              mult     e0     *     *
*************************************************************
* the host will start execution at 100 and wait here for the *
* host to interrupt the FP, at which point the FP will do a  *
* program jump to $0007, where there will be a jump to the   *
* correct routine.                                           *
*************************************************************
*   This is the wait routine, which waits for an interrupt.  *
*************************************************************
*
*****************************Begin  Routine**********************************
fpmr tr *   *   *  *    e                   a1    e0    a0    q
*************************************************************
*  load multiplicand                                        *
*************************************************************
     tr *   *   *  *    g                   a0    p     a1    g0
*************************************************************
* load multiplier                                           *
*************************************************************
     tc *   *   *  *           $ 0004          *           cmr0
*************************************************************
* this condition will check to see of f0 < 0.               *
*************************************************************
     tc *   *   *  *           $ 0002          *           cmr3
*************************************************************
*    is index0 = compare register 0?                        *
*************************************************************
     tr *   cl0 *  pupu add                 a1    *     a0    f1
     tr *   *   *  pupu *                  mult  f0    f1    e1
     tr *   *   *  *    *                  *     *     f0    icr0
*************************************************************
*If the value returned is zero, zero both registers,   return*
*************************************************************
     tc ad  *   jp *           $ 0000          f0          f1
     tc ad  *   df *           $ 0000          *           *
*************************************************************
* If f0 is justified, return. The product is normalized.    *
*************************************************************
     tc *   *   *  *           $ 0000          *           cmr1
     tc tnn *   jp *           $ 80ff          g1          *
     tr tnn *   df *    and                  a0    *     a1    f1
*************************************************************
```

```
*    Save the exponent in g1, clear E1 for a counter          *
*************************************************************
     tr *   *   *  *    zro                f1    g1    a0    e1
     tr *   cl0 *  *    zro                a0    e0    *     *
*************************************************************
* By here, product cannot be zero.The normalization process  *
* will take less than four repeats of this loop. If it ever  *
* takes more, there is something branching directly to this  *
* process.                                                   *
*************************************************************
nrm  tr fnn in0 *  *    e+1                a0    e0    a1    *
     sh fnn *   *  *    *                  nzin  lzin  nzin  s
*************************************************************
*    By now, the result must be normalized!!!!!              *
*************************************************************
     tr *   *   *  *    g                  a0    *     a1    f1
     tr *   *   *  *    e                  a0    g1    a1    g0
     tr *   *   *  *    *                  f0    e0    f1    e1
     sh *   *   *  *    *                  lcir  lcir  lzin  s
     tr *   *   *  *    e-g                a0    e0    a1    e1
*************************************************************
*This will take the normalized result, shift it left, adjust *
*the expontent, so that it agrees with the mantissa.        *
*************************************************************
     tc *   *   *  *              $ 01ff           g1          *
     tc *   *   *  *              $ ffff           *           g0
     tr *   *   jp *    acb                a0    *     a1    f1
*************************************************************
* this will "mask off" any carries into the unused portion of*
* the exponent.                                              *
*************************************************************
     sh *   *   df *    *                  nzin  rcir  nzin  s
*
*****************************End of Routine*************************************
*
**********************Begin Routine*****************************
* This routine does the initial setup of the variables        *
*************************************************************
*
fplr tc *   cl2 *  *          $ 0000              l0ad        l1ad
*************************************************************
*this clears all the index registers and the large file write*
* pointers.                                                  *
*************************************************************
     tc *   *   *  *          $ 0100              nop         icr2
     tc *   *   *  *          $ 0010              nop         cmr3
*************************************************************
*   the 0010 tests for index2 <> its compare.                *
*This will load the compare register to check for index      *
*regiter equal to its stored value.                          *
*************************************************************
*This loads the temporary file with the octal location of the*
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
*mean vector.                                                    *
*****************************************************************
     tc *   *   *  *         $ 0000              f0             *
     io *   *   *  *    *                  f0    *     ds     lsb  * * * *
*****************************************************************
*This loads the bank and address location of the covariance      *
*matrix                                                          *
*****************************************************************
*
*
*******************************End   Routine*********************************
*
*
*******************************Begin Routine*********************************
*        This routine loads the covariance matrix.                *
*****************************************************************
imr  io *   in2 *  *    *                  *     *     ds     r  * *     *
     tr *   *   *  *    *                  z0    f0    z1     f1
*****************************************************************
*    this loads the mantissa into the f1 register.               *
*    and  loads the exponent into the f0 register.               *
*****************************************************************
     tc ad  *   *  *         imr                 mar            nop
     tr *   *   *  *    *                  f0    lf0u  f1       lf1u
****************************End   Routine****************************
*
*
****************************Begin Routine*********************
*    This is the routine that loads the mean vectors.           *
*    This routine loads all 16 mean vectors at once.            *
*********i*******************************************************
mnr  tc *   cl2 *  *         $ 0010              nop            icr3
imnr tr *   in2 *  *    *                  z0    f0    z1     f1
*****************************************************************
*  This does the i/o call and loads the number into the f0-f1*
*  register pair                                                *
*  This does the i/o call for the next number                   *
*****************************************************************
     sh *   *   *  *    *                  *     ln31  *        s
*****************************************************************
* this shifts the mean vector to the left, and negates the      *
* sign bit.                                                     *
*****************************************************************
     sh *   *   *  *    *                  *     rcir  *        s
*****************************************************************
*                                                               *
*This negates the sign of the mean vector and shifts it in to*
*the sign position.  This is done because the vector mantissa*
*is in S&M form.  This way, an addition to the vector will     *
*actually  perform the operation of subtracting the vector     *
*from the addend.                                               *
*****************************************************************
```

```
*
     io *   *   *  *    *                  *     *     ds    r  * *     *
*************************************************************
*This does the i/o call for the next number (if there is one)*
*************************************************************
     tc ad  *   *  *         imnr                mar         nop
     tr *   *   *  *    *                  f0    lf0u  f1    lf1u
*************************************************************
*Load the next element in the vector, and store the new value*
*************************************************************
*Load precomputed constant  for each class (8 cycles, not in code)
*********************************End   Routine*********************************
*
*
*********************************Begin Routine*********************************
*This routine loads the normalized the data vector.  It can  *
*be called to execute by itself.                             *
*************************************************************
*
     tc *   *   *  *         $ 0004              *           icr3
vinr tc *   *   *  *         $ 0040              nop         cmr3
*************************************************************
*   the 0040 tests for index 3 <> its compare (4 elements of *
*   the data vector.)                                        *
*************************************************************
     tc *   *   *  *         $ 000a              t0wa        t1wa
     tc *   cl3 *  *         $ 0151              f0          nop
     io *   *   *  *    *                  f0    *     ds    lsb * *     *
*************************************************************
*   This initializes the location of the read pointer.       *
*   and this initializes the first read.                     *
*************************************************************
     io *   *   *  *    *                  *     *     ds    r   * *     *
*************************************************************
*This sets the large file hold registers to a minimum value. *
*************************************************************
     tc *   *   *  *         $ 014f              l0ad        l1ad
     tc *   *   *  *         $ 807f              *           f1
     tc *   *   *  *         $ ffff              f0          *
     tr *   *   *  *    *                  f0    lf0u  f1    lf1u
     tr *   *   *  *    *                  f0    lf0u  f1    lf1u
loip tr *   in3 *  *    *                  z0    tf0u  z1    tf1u
*************************************************************
* this loads the mantissa and exponent into the temp file    *
*************************************************************
*    The following does the I/O call.
     io *   *   *  *    *                  *     *     ds    r   * *     *
*
*    This is executed before the jump and it will load the needed data into
*    the z0-z1 register pair  before it is neeeded, eliminating a two cycle
*    not ready wait.
*
```

```
     tc ad  *   *  *        loip                mar        nop
     tc *   *   *  *        $ 0000              brg0       brg1
     tc *   *   *  *        $ 0006              t0wa       t1wa
     tc *   *   *  *        $ 0000              tf0n       tf1n
*
*    the 0 in location 6 of the temporary files is a cycle counter.
*    it keeps track of the class currently being worked upon.
*
*    After normalizing the data vector, store it, repeat until all
*    the elements are finished, then repeat the cycle until all four elements
*    are finished being processed.
*
     tc *   *. *  *        $ 000a              t0ra       t1ra
     tc *   cl3 *  *        $ 0002              t0wa       t1wa
     tc *   *   *  *        $ 0151              l0ad       l1ad
lo1p tr *   in3 *  *   *                 tf0u  brg0  tf1u brg1
     tc *   *   sr *        fpar                mar        *
     tr *   *   *  *   *                 lf0u  f0    lf1u  f1
     tc *   *   *  *        $ 0040              *          cmr3
     tc ad  *   *  *        lo1p                mar        *
     tr *   *   *  *   *                 f0    tf0u  f1    tf1u
     tc *   cl3 *  *        $ 000a              t0ra       t1ra
*
*    This stores the data normalized data vector in locations 2-5 of the
*    temporary file.  The second vector will appear in locations 6-9 of
*    the temporary file.
*
*    This will fall through to the matrix processing routine.
*
*
*
*    This is the beginning of the matrix multiply routine.
*
stra tc *   cla *  *        $ 0000              brg0       brg1
     tc *   *   *  *        $ 0006              t0ra       t1ra
     tr *   *   *  *   *                 tf0n  p     *     *
     tc *   *   *  *        $ 0010              *          q
     tc *   *   *  plql     $ 0000              *          nop
     tr *   *   *  plql *              mult  l0ad  mult  l1ad
     tc *   *   *  *        $ 0001              t0ba       t1ba
     tc *   *   *  *        $ 0001              tf0u       tf1u
     tc *   *   *  *        $ 0002              t0ba       t1ba
mlty tr *   in1 *  *   *                 tf0u  e0    tf1u  e1
*
* This loads the multiplicand into the e0-e1 register pair.
*
     tc *   *   sr *    fpmr                      mar        nop
*
* This does the program jump to the floating point multiply routine.
*
     tr *   *   *  *   *                   lf1u  g1    lf0u  g0
*
```

```
*
* This step is done before the jump is actually executed. This will load the
* multiplier into the g0-g1 register pair. (F=EXG floating point mult)
*
     tc *   *   sr *         fpar                 mar          nop
*
* This step will do a jump to the floating point addition routine.This rout-
* ine calculates the sum of the contents of the F register and the BRG regis-
* ter pair. The result of the add is then stored in the F register.
*
     tc *   *   *  *         $ 0004               nop          icr1
     tc *   *   *  *         $ 0004               nop          cmr3
*   the 0004 tests for index1 <> its compare
*
*
* This is executed before the jump.  It will just load the condition register
* with the next condition to be tested.
*
     tc ad  *   *  *         mlty                 mar          nop
     tr *   *   *  *    *                    f0   brg0  f1     brg1
*
* On index register 1 not equal to its compare, jump to beginning of multiply
* routine.
*
*
     tc *   *   *  *         $ 0001               t0ba         t1ba
     tr *   *   *  *    *                    tf1n e0    nop    nop
*
*    get address of jth item in the data vector.
*
     tr *   *   *  *    ac0                  a1   tf0d  a0     tf1d
     tr *   *   *  *    ac0                  a0   t0ra  a0     t1ra
*    tr *   *   *  *    ac0                  a1   *     a0     t1ra
* the above was a change to insure that the program works, this is kept.
* this will update the address for the next round, store it, and point to the
* item in question.
*
     tr *   in2 *  *    *                    tf0c e0    tf1c   e1
*
* this will load the multiplier for the second multiply into the e0-e1 reg-
* ister pair.  Simultaneously, this will zero the temp file pointers. They
* will now point to the location of the accumulator.
*
     tr *   *   *  *    *                    bsr0 g1    bsr1   g0
     tc *   *   sr *    fpmr                      mar          nop
     tr *   *   *  *    g                    a0   g1    a1     g0
*
* this is just a subroutine jump to the floating point multiply routine.
* f=EXG
     tc *   *   sr *         fpar                 mar          nop
     tr *   *   *  *    *                    tf0n brg0  tf1n   brg1
*
```

```
*    F=F+BRG. This calculates the subtotal of the matrix multiply.
*
     tr *   *   *  *    *                  f0    tf0n  f1    tf1n
     tc *   *   *  *         $ 0002              t0ba        t1ba
*
* The above two steps load the sub total into the temporary file location
* zero.  It them resets the read and write pointers of the temporary file to
* location two.
*
     tc *   *   *  *         $ 0004              nop         icr2
     tc *   *   *  *         $ 0010              nop         cmr3
*   the 0010 tests for index2 # its compare.
* This will do a test for index 0 not equal to its compare register.
*
     tc ad  cl1 *  *         mlty                mar         nop
     tc *   *   *  *         $ 0000              brg0        brg1
     tr *   *   *  puql *                  *     *     mult  mcr3
     tc *   *   *  *         $ 0000              t0ba        t1ba
     tc *   *   *  *         $ 014f              l0ad        l1ad
* Add precomputed constants (150 cycles, not in code)
* This is the compare routine.
*
     tc *   *   *  *         $ 0000              t0ba        t1ba
     tr *   *   *  *    *                  tf1n  g1    tf0n  g0
     tc *   *   sr *         fcmp                mar         *
     tr *   *   *  *    *                  lf0n  e0    lf1n  e1
*
*  here, if G > E,  (f0=f1=0) return tf1[6] as the class (to location 150)
*
     tr *   cl0 *  *    *                  *     *     f1    icr0
     tc *   *   *  *         $ 0002              *           cmr3
*
* this will increment tf0n[6], the pointer to this array.
*
     tc *   *   *  *         $ 0006              t0ba        t1ba
     tr ad  *   *  *    *                  tf0n  e0    *     *
     tc ad  cl0 *  *         chck                mar         *
     tr ad  *   *  *    e+1                 a0    tf0n  *     *
     tr *   *   *  *    e                   a0    lf0u  a1    lf1u
     tr *   *   *  *    *                   tf0n  lf0u  tf1n  lf1u
     tr *   *   *  *    *                   tf0n  e0    *     *
     tr *   *   *  *    e+1                 a0    tf0n  *     *
chck tc *   *   *  *         $ 0010              *           icr0
     tc *   *   *  *         $ 0001              *           cmr3
     tc *   *   *  *         $ 0004              p           *
     tr *   *   *  *    *                   *     *     tf0n  q
     tc *   *   *  plql      $ 0151              *           g0
     tr *   *   *  plql *                  mult  e0    tf0n  idx0
     tr *   *   *  *    add                 a0    l0ad  a1    *
     tr *   *   *  *    add                 a1    *     a0    l1ad
     tc *   *   *  *         $ 0000              t0ba        t1ba
     tc *   *   *  *         $ 0000              tf0u        tf1u
```

```
     tc *   *   *  *        $ 0001              tf0u       tf1u
     tc adn *   *  *        lo1p                mar        *
     tc *   cl0 *  *        $ 0000              e0         e1
finl tc *   *   *  *        $ 0150              l0ad       l1ad
otpt io *   *   *  *   *                  *     lf0d  ds   w    * * * *
     io *   *   *  *   *                  lf0d  *     ds   w    * * * *
     io *   *   *  *   *                  *     lf0d  ds   w    * * * *
     io *   *   *  *   *                  lf0d  *     ds   w    * * * *
     tr *   *   *  pupu *                 mult  brg0  *    *
     tc *   *   *  *        wait                mar        nop
*
*
*
*****************************End   Routine*****************************
*
*
*
************************begin routine*************************************
*****************************Begin Routine*****************************
*
*    This is the floating point addition routine. 9/4/79.  3:45:00.
fpar tr *   *   *  *   *                  bsr1  g1    f1   e1
     sh uns cl0 *  *   *                  lzin  nzin  lzin s
     sh *   *   *  *   *                  rzin  nzin  rzin s
*
*    This will strip the sign of the mantissa and save it for future use.
*
     tr *   *   *  *   *                  *     *     bsr0 icr0
     tc tnn *   *  *        $ 0000              *          cmr0
     tc tnn *   *  *        $ 0010              *          cmr1
     tr tnn *   jp *   *                  *     *     *    *
     tr tnn *   df *   *                  *     *     *    *
     tr *   *   *  *   zro                  a0    *     a1   cmr1
*
*    this will compare the brg to zero, if it is, return.
*
     tr *   *   *  *   zro                  a0    e0    a1   g0
*
*    This will zero the registers to prevent spurious results.
*
     tr *   *   *  *   e-g                  a0    nop   a1   icr0
*
*    if |e|<|g|, the program will reverse the numbers and continue.
*    since addition is commutative, this should not affect the results.
*
     tr *   *   *  *   xor                  a1    e0    a0   nop
     tc *   *   *  *        $ 0080              *          g0
     tr *   *   *  *   and                  a1    *     a0   g0
     tc *   *   *  *        $ 8000              e0         *
     tr *   *   *  *   e-g                  a1    *     a0   g0
     tc *   *   *  *        $ 0010              *          cmr0
     tc fnn *   *  *        nsh                 mar        *
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
*
*    If the exponent on one of the two numbers is less than zero
*    and the other is not, subtraction to yield the number of shifts
*    will not yield the correct answer, and thus special handling
*    must be added to compensate for this problem. The way that this
*    routine handles the problem is it exclusive ors the two numbers
*    together and then strips off everything but the sign bit.  This
*    is then subtracted from a constant (for speed).The constant is
*    8000, thus if there is a 1 in the sign position, the result will
*    not be negative, indicating that the correction must take place.
*
     tc tnn *   *  *        $ 0020              *          cmr0
     tr *   *   *  *    e-g                a1    g1    *    *
*
*    This will test for E-G->G negative. This is to insure that |brg| >=
*    |f|, simplifying the algorithm greatly.
*
     tc tnn *   *  *    swap                     mar        nop
*
*    This involkes the swap routine that will force the above to be true.
*
     tr *   *   *  *    zro                *     *     a0   cmr0
     tc *   *   *  *    $ 0010                   *          cmr1
*
* the zeroes that are loaded into condition mask register 0 tell
* the machine not to check for any of the conditions represented.
* the 0010 loaded into cmr1 tell the machine to check for the compare
* register greater than index register one.  In this case, this will
* determine whether the two numbers are equal or equal and opposite in
* magnitude and sign.
*
     tr *   *   *  *    zro                a0    g1    a1   e1
     tc tnn *   *  *    equl                     mar        nop
*
*    If they are, the program will jump to a special routine.
*
*
*    By this point in the program, |e|>|g|.
*
     tr *   *   *  *    *                  bsr0  e0    f0   g0
     tc *   *   *  *        $ 0010              *          idx0
     tc *   *   *  *        $ 0020              *          cmr1
     tc tnn *   *  *        rtnf                mar        nop
*
*    If the number of shifts required > 16, return the value in the
*    F register.
*
     tr *   *   *  *    zro                a1    g1    a0   cmr1
     tc *   cl0 *  *    $ 0001                  *          cmr3
*
*    this loads the data to be processed and it programs the CPU to check
*    for reg0#indx0. This is represented by a one in the first position. Thi
```

```
*    check is involked by the AD command.
*
shft sh *   in0 *  *    *                  rzin  nzin  nzin  s
     tc ad  *   *  *    shft                     mar         nop
*
* Index register contains the amount by which G>E, (the number of orders
* of magnitude. This routine shifts E to the right until the two orders of
* magnitude are equal.
*
     tc *   *   *  *    $ 0000                   g1          *
     tc *   *   *  *    $ 0020                   *           cmr0
     tc fpn *   *  *    gpos                     mar         nop
*
*    if g1 >= 0, its sign is taken to be positive, and the numbers are
*    handeled in a corresponding manner.
*
*
*    By this point, g must be negative.
*
     tc *   *   *  *    $ 0002                   *           cmr0
     tc tpn *   *  *    ssgn                     mar         nop
*
*    If E is negative, and G is negative, the signs are the same and the
*    two numbers are just added and one of the signs is preserved.
*
     tc *   *   *  *                        *     *     *     *
*
*    At this point, |G|>|E|, the resultant sign will be that of G.
*    Without regard to sign, the result will be the old sign of G
*    plus |g-e|.
*
dsgn tr *   *   *  *    en                   a0    e0    *     *
     tr *   *   *  *    e+1                  a0    e0    *     *
     tr *   *   *  *    add                  f1    e0    a0    g0
*
*    This calculates g-e.
*
     tc *   *   *  *    $ 0010                   *           cmr0
*
*    If the result is >= zero, there is not a one in tthe first bit position,
*    so the number is not normalized, and must be shifted until there appears
*    a '1' in the first bit position.
*
norm tr fnn *   *  *    e-1                  a0    e0    *     *
     tc fnn *   *  *    norm                       mar         nop
     sh fnn *   *  *    *                    nzin  nzin  lzin  s
*
*    This routine normlizes the data
*
     tr *   *   *  *    g                    a0    f0    *     *
     tc *   *   *  *         $ 80ff                *           g0
     tr *   *   *  *    and                  *     *     a0    f1
```

```
     tc *   *   *  *        $ 0002              *          cmr0
     sh *   *   *  *    *                 nzin  lzin  nzin  s
     sh tpn *   *  *    *                 nzin  roin  nzin  s
     sh fpn *   *  *    *                 nzin  rzin  nzin  s
     tc *   *   jp *        $ 0000              *          *
     tc *   *   df *        $ 0000              *          *
*
*    this routine sets the sign to the sorrect sign and returns to the calling
*    routine.
*
gpos tc tpn *   *  *    dsgn                    mar        nop
     tr *   *   *  *    *                 *     *     *    *
*
*    Before the jump to GPOS, the condition register was set to check for
*    e<0. If it is, the signs are opposite and the data is treated
*    correspondingly.
*
*
*    By default, both G and E have the same sign, so the results are just
*    added.
*
     tr *   *   *  *    *                 *     *     f0   g0
ssgn tr *   *   *  *    add               a1    g1    a0   g0
     sh *   *   *  *    *                 nzin  nzin  rcir s
     tc *   *   *  *    $ 0010                nop        cmr0
*
*    this checks for a carry out of he MSB, indicating normalization is
*    necessary.
*
     tr tnn *   *  *    g                 f1    e0    *    *
     tr tnn *   *  *    e+1               a0    e0    *    *
     tr tnn *   *  *    g                 a0    f0    *    *
     tc tnn *   jp *         $ 80ff             *          g0
     tr tnn *   df *    and               *     *     a0   f1
*
*    If it is, then the number is normalized and the subroutine returns.
*
     sh *   *   jp *    *                 nzin  nzin  lcir s
     tr *   *   df *    g                 a0    f0    *    *
*
*    This routine exchanges the two registers involved so that |G|>|E|
*
swap tr *   *   *  *    *                 f1    g1    f0   g0
     tr *   *   *  *    *                 bsr0  f0    bsr1 f1
     tc *   *   *  *          fpar              mar        nop
     tr *   *   *  *    g                 a0    brg0  a1   brg1
*
*    This calls the original routine.
*
*
*    This is the action taken when the routines have the same magnitude.
*
```

```
equl tc *   *   *  *    $ 0000                    nop          cmr1
     tc *   *   *  *    $ 0002                    nop          cmr0
     tc fpn *   *  *    epos                      mar          nop
     tc *   *   *  *    $ 0020                    nop          cmr0
     tc tp  *   *  *    ssgn                      mar          nop
     tc fp  *   *  *    swap                      mar          nop
     tc *   *   *  *    $ 0000                    nop          nop
epos tc fp  *   *  *    ssgn                      mar          nop
     tc *   *   *  *    $ 0100                    nop          cmr0
     tr *   *   *  *    e=g              nop      nop    nop   nop
     tc tn  *   *  *    zapp                      mar          nop
     tc *   *   *  *    $ 0020                    nop          cmr0
     tr *   *   *  *    e-g              a1       g1     a0    nop
     tc tn  *   *  *    dsgn                      mar          nop
     tc *   *   *  *    $ 0000                    g1           cmr0
     tc *   *   *  *    dsgn                      mar          nop
     tr *   *   *  *    *                nop      nop    bsr1  f1
     tc *   *   *  *         $ 0000               *            *
zapp tr *   *   jp *    zro              a0       f0     a1    f1
     tc *   *   df *    $ 0000                    nop          cmr0
*
*    this routine handles numbers that have different exponential signs.
*
nsh  tc *   cl0 *  *         $ 0010               nop          cmr0
*
*    bug in assembler.  null line will not be assembled.  By this point
*    in the program, the exponent on one of the two numbers must
*    be less than zero. This part of the routine will force the negative
*    part to be stored in brg register. Since a swap can take place,
*    all the original flags must be reset in the event of ashift.
*
     tr *   *   *  *    g                a1sw     *      a0rz  g0
     tc tnn *   *  *         glz                  mar          nop
*
*    The g/brg register contains the negative exponent, no swap needed.
*
     tr fnn *   *  *    *                f1       g1     f0    g0
     tr *   *   *  *    *                bsr0     f0     bsr1  f1
     tr *   *   *  *    g                a0       brg0   a1    brg1
     tr *   *   *  *    *                bsr1     g1     f1    e1
     sh uns *   *  *    *                lzin     nzin   lzin  s
     sh *   *   *  *    *                rzin     nzin   rzin  s
*
*    This swaps the two numbers and resets all the flags needed by the
*    rest of the routine.
*
glz  tc *   *   *  *         $ 0000               e0           g0
     tr *   *   *  *    e-g              a1sw     *      a0sw  g0
     tr *   *   *  *    g                a1rz     nop    a0rs  icr0
*
*    Calculate the number of shifts needed, if it is < 0, it is
*    actually > 80 (16), so return the value in the F register.
```

```
*
     tc tnn *   *  *         rtnf                 mar          nop
     tr *   *   *  *    g                  a0sw  e0    a1rz   nop
     tc *   *   *  *         $ 0010               *            g0
     tr *   *   *  *    e-g                a1    nop   a0     g0
     tc fnn *   *  *         rtnf                 mar          nop
*
*    If the number of shifts required is > 16, return the data in the
*    F register.
*
     tc *   *   *  *         $ 0000               *            cmr0
     tc *   *   *  *         $ 0000               *            e1
     tr *   *   *  *    *                  bsr0  e0    f0     g0
     tc *   *   *  *         shft                 mar          nop
     tc *   *   *  *         $ 0001               *            cmr3
*
*    prepair to shift the data and return to shifting routine,
*
rtnf tr *   *   jp *    *                  *     *     *      *
     tr *   *   df *    *                  *     *     *      *
*
*    Return the contents of the F register.
*
*************************************************************
* this is just for a break point and it is to be removed when*
* the program is actually inserted into the code.           *
*************************************************************
fcmp tc *   *   *  *         $ 0000               t0ba         t1ba
*************************************************************
* This accepts the data in the E register and G register as  *
* Inputs. Initially, the program stores the original data in *
* temporary file. the E register goes in location 0 and the  *
* G register goes in location 1. The following will also     *
* strip off the sign bit                                     *
*************************************************************
     tr *   *   *  *    e                  a0    tf0u  a1     tf1u
     tr *   *   *  *    g                  a0    tf0u  a1     tf1u
*************************************************************
* This routine strips off the sign bit. The correct sign bit *
* is saved in the PAST register.                             *
*************************************************************
     sh uns *   *  *    *                  lzin  nzin  lzin   s
     sh *   *   *  *    *                  rzin  nzin  rzin   s
     tr *   *   *  *    e                  a1sw  e0    a0sw   e1
     tr *   *   *  *    g                  a0sw  g1    a1sw   g0
*************************************************************
* The 0002 in cmr0 will check for E1 negative. This is done  *
* in the past sense. If E1<0, then jump to the routine that  *
* will handle that case.                                     *
*************************************************************
     tc *   *   *  *         $ 0002               nop          cmr0
     tc *   *   *  *         $ 0000               nop          cmr1
```

```
     tc tpn *   *  *         emng                 mar          *
*************************************************************
* by this point, the E register must not be negative (>=0)   *
* the 0020 in the cmr0 will test for g<0. If g<0, e is the   *
* greater of the two numbers. If not, they are both >= 0.    *
*************************************************************
     tc *   *   *  *         $ 0020               *            cmr0
     tc tpn *   *  *         egrt                 mar          *
*************************************************************
* This will determine if there is a difference in exp sgn.   *
*************************************************************
     tc tpn *   *  *         $ 0000               *            cmr1
     tc tnn *   *  *         gxng                 mar          *
*************************************************************
* This will do a jump if the sign of G is 1, or G negative   *
* in the exponent portion.                                   *
*************************************************************
     tc *   *   *  *         $ 0002               *            cmr0
     tc tnn *   *  *         ggrt                 mar          *
*************************************************************
* By here, the exponent of g is positive. If the exponent of *
* E is negative, both mantissas being positive, e<g          *
*************************************************************
     tr fnn *   *  *    e-g                 a0    e0    a1     e1
     tc tnn *   *  *         ggrt           mar         *
     tc fnn *   *  *         $ 0000               f0           f1
     tc *   *   jp *         $ 0000               t0ba         t1ba
     tr *   *   df *    *                   tf0n  e0    tf1n   e1
*************************************************************
* Since both exponents and mantissas are nonnegative, this   *
* routine calculates e-g, exponents in the HOBP and mantissas*
* in the LOBPs. If the result is < 0, g>e, else return E.    *
*************************************************************
ggrt tc *   *   *  *         $ 0001               t0ba         t1ba
     tc *   *   jp *         $ 0001               f0           f1
     tr *   *   df *    *                   tf0n  e0    tf1n   e1
*************************************************************
* if f1>=0, e>g, return tf[1]                                *
*************************************************************
egrt tc *   *   *  *         $ 0000               f0           f1
     tc *   *   jp *         $ 0000               t0ba         t1ba
     tr *   *   df *    *                   tf0n  e0    tf1n   e1
*************************************************************
* This is the section of the program that is called if E is  *
* negative. (mantissa)                                       *
*************************************************************
emng tc *   *   *  *         $ 0020               *            cmr0
     tc fpn *   *  *         ggrt                 mar          nop
*************************************************************
* This section does the compare if both the operands are < 0 *
* This will determine if there is a difference in exp sgn.   *
*************************************************************
```

```
ncmp tc *   *   *  *        $ 0000        *          *
     tc tnn *   *  *        gbng          mar        *
*****************************************************************
* This will do a jump if the sign of G is 1, or G negative      *
* in the exponent portion.                                      *
*****************************************************************
     tc *   *   *  *        $ 0002              *          cmr0
     tc fnn *   *  *        nnpp          mar        *
*****************************************************************
* By here, the exponent of g is positive. If the exponent of    *
* E is negative, both mantissas being negative, e>g             *
*****************************************************************
     tc tnn *   *  *        $ 0000              f0         f1
     tc *   *   jp *        $ 0000              t0ba       t1ba
     tr *   *   df *   *                  tf0n  e0   tf1n  e1
*****************************************************************
* The above will return e                                       *
*****************************************************************
gbng tc tnn *   *  *        ebng          mar        *
*****************************************************************
* Both G's exponent and sign are negative. If true, the same    *
* holds true for E. If this is false, return g.                 *
*****************************************************************
     tc *   *   *  *        $ 0001              t0ba       t1ba
     tc *   *   jp *        $ 0001              f0         f1
     tr *   *   df *   *                  tf0n  e0   tf1n  e1
*****************************************************************
ebng tr *   *   *  *   e-g                  a0    e0   a1    e1
     tc fnn *   *  *        ggrt          mar        *
     tc *   *   *  *        $ 0000              f0         f1
     tc *   *   jp *        $ 0000              t0ba       t1ba
     tr *   *   df *   *                  tf0n  e0   tf1n  e1
*****************************************************************
*  Both the mantissa and the exponent of both E and G are       *
* less than zero.  calculate e-g. if result positive, g>e       *
*****************************************************************
gxng tc fnn *   *  *        egrt          mar        nop
     tc *   *   *  *        $ 0000        *          *
test tr *   *   *  *   e-g                  a0    e0   a1    e1
     tc fnn *   *  *        egrt                mar        nop
     tc tnn *   *  *        ggrt                mar        nop
     tc *   *   *  *        $ 0000        *          *
*****************************************************************
* at this (preceeding line) both E and G are positive. The      *
* sign of the exponent of g is negative. If the sign of the     *
* exponent of E is positive, e>g, hence return E.               *
*****************************************************************
nnpp tr *   *   *  *   e-g                  a0    e0   a1    e1
     tc tnn *   *  *        egrt                mar        nop
     tc fnn *   *  *        ggrt                mar        nop
     tc *   *   *  *        $ 0000        *          *
#
```

APPENDIX 6

CONTEXTUAL CLASSIFIER PROGRAM LISTING

```
 org  0100
     tc *   *   *  *        $ 0000              l0ad        l1ad
*
*       THREE PIXEL LINEAR NEIGHBORHOOD -- 04 classes!
*       CONTEXTUAL CLASSIFIER MOD VI
*       4/5/1980
*
*
*
     tc *   *   *  *        $ 0010              nop         icr0
     tc *   *   *  *        $ 0002              nop         cmr3
lold tc *   *   *  *        $ 0000              tf0u        tf1u
     tc adn in0 *  *        lold                mar         nop
     tr adn *   *  *    *                  if0u  lf0u  if1u  lf1u
*
*    Load |Covariance Matrix|
*
     tc *   cl0 *  *        $ 0030              nop         icr0
     tc *   *   *  *        $ 8000              g1          t1ba
     tc *   *   *  *        $ 0000              t0ba        g0
loam tr *   in0 *  *    *                  if0u  e0    if1u  e1
     tc adn *   *  *        loam                mar         nop
     tr *   *   *  *    xor                 a0    lf0u  a1    lf1u
*
*    Load mean vectors
*
     tc *   cl0 *  *        $ 0120              nop         icr0
loac tc adn *   *  *        loac                mar         nop
     tr adn in0 *  *    *                  if0u  lf0u  if1u  lf1u
     tr *   *   *  *    zro                 a0    lf0u  a1    lf1u
     tr *   *   *  *    zro                 a0    lf0u  a1    lf1u
*
*    Load covariance matrix.
*
     tc *   cl0 *  *        $ 0032              nop         icr0
load tc adn *   *  *        load                mar         nop
     tr adn in0 *  *    *                  if0u  lf0u  if1u  lf1u
     tc *   cl0 *  *        $ 0152              l0ad        l1ad
     tc *   *   *  *        $ 0533              nop         icr0
loag tc adn *   *  *        loag                mar         nop
     tr adn *   *  *    *                  if0u  lf0u  if1u  lf1u
*
*    the above routine will load the data vector & the a-priori configuration
*    al probabilities.
*    The configuration of the large file is as follows:
*                                        Base 10 Base 16
*    -ln |Covariance matrices|           000-016 000-00f
*    Mean Vectors ( X -1 )               017-064 010-03f     /* 12 classes */
*    covariance matrices                 065-255 040-0ff     /* 12 classes */
*    data vectors                        256-287 100-121     /* 8 pixels   */
*    Not currently used                  289-337 122-151
*    G[r,j,q]                           338-1668 152-684     /* 11 classes */
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
*    Not currently used                    1668-1776 684-6f1
*    a[r]                                  1777-1792 6f1-700
*    b[j]                                  1793-1808 701-710
*    c[q]                                  1809-1824 711-720
*    Not used                              1825      721
*    i                                     1826      722
*    j                                     1827      723
*    k                                     1828      724
*    r                                     1829      725
*    q                                     1830      726
*    value                                 1831      727
*    class                                 1832      728
*    jj                                    1833      729
*    [r,j,q]                               1834      72a
*
     tc *   *   *  *        $ 0722                    l0ad       l1ad
     tc *   *   *  *        $ 0000                    lf0u       lf1u
ilp1 tc *   *   *  *        $ 0723                    l0ad       l1ad
     tc *   *   *  *        $ 000f                    t0ba       t1ba
     tc *   *   *  *        $ 06f1                    tf0n       tf1n
     tc *   *   sr *        comf                      mar        nop
     tc *   *   *  *        $ fffe                    lf0n       lf1n
     tr *   *   *  pupl *                  nop   nop   mult  mcr3
*
*    lf1,0[1826]=0 lf0,1[1827]=-2;call comf to calculate a[j]
*    where 0<=j<=3      comf calculates the 3  classes for pixel
*    k+2. comf also assumed that l0ad and l1ad are 1827.
*
*    comf also assumes that in tf0[07] the location of the destination
*    is stored.
*
*    for i=0 to I-1 do:
*
     tc *   *   *  *        $ 0723                    l0ad       l1ad
     tc *   *   *  *        $ 000f                    t0ba       t1ba
     tc *   *   *  *        $ 0701                    tf0n       tf1n
     tc *   *   sr *        comf                      mar        nop
     tc *   *   *  *        $ ffff                    lf0n       lf1n
     tr *   *   *  pupl *                  nop   nop   mult  mcr3
*
*    call comf to calculate b[j], where 0 <= j <= 4.
*
     tc *   *   *  *        $ 0723                    l0ad       l1ad
     tc *   *   *  *        $ 000f                    t0ba       t1ba
     tc *   *   *  *        $ 0711                    tf0n       tf1n
     tc *   *   sr *        comf                      mar        nop
     tc *   *   *  *        $ 0000                    lf0n       lf1n
     tr *   *   *  pupl *                  nop   nop   mult  mcr3
*
*    call comf to calculate c[j], where 0<= j <= 4.
*
     tc *   *   *  *        $ 0723                    l0ad       l1ad
```

```
     tc *   *   *  *          $ 0003                lf0u        lf1u
     tc *   *   *  *          $ 0001                lf0u        lf1u
*
*    lf0,1[1827]=3,lf0,1[1828]=1.
*    for k=1 to J-2 do:
*
klp1 tc *   *   *  *          $ 0727                l0ad        l1ad
     tc *   *   *  *          $ 8001                nop         e1
     tc *   *   *  *          $ 8000                e0          nop
     tr *   *   *  *    e                  a0      lf0u  a1    lf1u
     tr *   *   *  *    e                  a0      lf0u  a1    lf1u
*
*    lf0,1[1831] = -1; lf0,1[1832] = -1;
*    value=class=-1
*
     tc *   *   *  *          $ 0723                l0ad        l1ad
     tc *   *   *  *          $ 0000                lf0u        lf1u
*
*    lf0,1[1827]=0;
*    j=0 (for j=0 to c-1 do:)
*
jlp1 tc *   *   *  *          $ 0725                l0ad        l1ad
     tc *   *   *  *          $ 0004                p           nop
     tc *   *   *  *          $ 0000                lf0u        lf1u
*
*    lf0,1[1829]=0; p=3  (always = number of classes C)
*    for r=0 to c-1 do:
*
     tc *   *   *  *          $ 0723                l0ad        l1ad
     tr *   *   *  *    *                  nop     nop   lf0n  q
     tc *   *   *  plql       $ 072a                l0ad        l1ad
     tr *   *   *  plql *                  mult    e0    nop   nop
     tc *   *   *  *          $ 0152                nop         g0
     tr *   *   *  *    add                a0      lf0n  a1    lf1n
*
*    lf0,1[1834]=(lf1,0[1827] X C ) + base address of G
*    this will provide the address for g[0,j,0];
*
rlp1 tc *   *   *  *          $ 0726                l0ad        l1ad
     tc *   *   *  *          $ 0000                lf0u        lf1u
     tc *   *   *  *          $ 0000                brg0        brg1
*
*    sum=0; lf1[1830]=0;
*    for q=0 to c-1 do:
*
qlp1 tc *   *   *  *          $ 0725                l0ad        l1ad
     tc *   *   *  *          $ 06f1                nop         g0
     tr *   *   *  *    add                a0      l0ad  a1    l1ad
*
*    e = 1777 + lf1[1829]; l0ad,l1ad = e; (a[r])
*
     tr *   *   *  *    *                  lf0n    f0    lf1n  f1
```

```
*
*     f0,1 = lf0,lf1
*
      tc *   *   *  *          $ 0701                nop          g0
      tc *   *   *  *          $ 0723                l0ad         l1ad
      tr *   *   *  *    *                    lf0n  e0    nop    nop
      tr *   *   *  *    add                  a0    l0ad  a0     l1ad
*
*     e = 1793 + lf1[1827]; load,l1ad = e; (b[j])
*
      tr *   *   *  *    *                    lf0n  e0    lf1n   e1
      tc *   *   sr *          fpmr                  mar          nop
      tr *   *   *  *    *                    f1    g1    f0     g0
*
*     f=eXg -- f = a[r] X b[j];
*
      tc *   *   *  *          $ 0711                nop          g0
      tc *   *   *  *          $ 0726                l0ad         l1ad
      tr *   *   *  *    *                    lf0n  e0    nop    nop
      tr *   *   *  *    add                  a0    l0ad  a0     l1ad
      tr *   *   *  *    *                    lf0n  e0    lf1n   e1
*
*     calculate location of c[q]. (=1809+lf1[1830])
*
      tc *   *   sr *          fpmr                  mar          nop
      tr *   *   *  *    *                    f1    g1    f0     g0
*
*     f=eXg -- f = a[r] X b[j] X c[q];
*
      tc *   *   *  *          $ 072a                l0ad         l1ad
      tr *   *   *  *    *                    lf0n  l0ad  lf1n   l1ad
*
*     l0ad,l1ad = lf1[1834] (g[r,j,q])
*
      tr *   *   *  *    *                    lf0n  e0    lf1n   e1
      tc *   *   sr *          fpmr                  mar          nop
      tr *   *   *  *    *                    f1    g1    f0     g0
*
*     f=eXg -- f = a[r] X b[j] X c[q] X g[r,j,q]
*
      tc *   *   sr *          fpar                  mar          nop
      tr *   *   *  *    *                    nop   nop   nop    nop
*
*     f = f + sum
*
      tr *   *   *  *    *                    f0    brg0  f1     brg1
*
*     sum = f
*
      tc *   *   *  *          $ 072a                l0ad         l1ad
      tr *   *   *  *    *                    lf0n  e0    lf1n   e1
      tr *   *   *  *    e+1                  a0    lf0n  a0     lf1n
```

```
*
*    update pointer into g[r,j,q] to next q
*
     tc *   *   *  *         $ 0726              l0ad        l1ad
     tr *   *   *  *    *                  lf0n  e0    nop   nop
     tr *   *   *  *    e+1                nop   nop   a0    idx0
     tc *   *   *  *         $ 0004              nop         icr0
     tc *   *   *  *         $ 0002              nop         cmr3
     tc adn *   *  *         qlp1                mar         nop
     tr *   *   *  *    e+1                a0    lf0n  a0    lf1n
*
*    q=q+1 (lf0,1[1830] = lf0,1[1830] + 1;
*    if q !=  4 goto qlp1
*
     tc *   *   *  *         $ 072a              l0ad        l1ad
     tr *   *   *  *    *                  lf0n  e0    nop   nop
     tc *   *   *  *         $ 006f              nop         g0
     tr *   *   *  *    add                a0    lf0n a0     lf1n
*
*    e=lf0,1[1834]
*    e=e+133
*    lf0,1[1834]=e
*
*    the program has just gone through all possible values  of q
*    in the combination g[r,j,q], it must now update to the next
*    value of r, as j is held constant.  12 if necessary because
*    g is a  4 X  4 X  4 matrix. The program is pointing  to the
*    last element of a given r, and j.
*
     tc *   *   *  *         $ 0726              l0ad        l1ad
     tc *   *   *  *         $ 0000              lf0d        lf1d
*
*    lf1[1830] = 0 (q = 0)
*
     tr *   *   *  *    *                  lf0n  e0    nop   nop
     tr *   *   *  *    e+1                a1    nop   a0    idx0
     tc *   *   *  *         $ 0004              nop         icr0
     tc *   *   *  *         $ 0002              nop         cmr3
     tc adn *   *  *         rlp1                mar         nop
     tr *   *   *  *    e+1                a0    lf0n  a0    lf1n
*
*    lf1[1829] = lf1[1829] + 1  (r=r+1)
*    store updated value of r.
*    if r !=  4 (base 10) goto rlp1
*
     tc *   *   *  *         $ 0727              l0ad        l1ad
     tr *   *   *  *    *                  lf0n  f0    lf1n  f1
     tr *   *   *  *    *                  bsr0  e0    bsr1  e1
     tc *   *   *  *         fcmp                mar         nop
     tr *   *   *  *    *                  f1    g1    f0    g0
*
*    g=lf1,0[1831]; e=brg1,0
```

```
*    f=g; floating point compare e and g.
*
*
     tr *   *   *  *    *                    nop   nop   f1    idx0
     tc *   *   *  *         $ 0000          nop   nop         icr0
     tc *   *   *  *         $ 0002          nop   nop         cmr3
     tr ad  *   *  *    *                    bsr0  lf0u  bsr1  lf1u
     tc ad  *   *  *         $ 0723          lOad  lOad        l1ad
     tr ad  *   *  *    *                    lf0n  e0    lf1n  e1
     tc ad  *   *  *         $ 0728          lOad  lOad        l1ad
     tr ad  *   *  *    e                    a0    lf0u  a1    lf1u
*
*    f will be set to zero or one, depending on whether g
*    or e is greater. If g is greater, the new value is less
*    than the old value. If e is greater, its value and class
*    are the new one for the pixel under consideration.
*
*    in program code: compare e,g;
*                     if (f!=0)
*                     {  lf0,1[1831] = brg0,1;
*                        lf0,1[1832] = lf0,1[1827];
*                     }
*
     tc *   *   *  *         $ 0729          lOad  lOad        l1ad
     tc *   *   *  *         $ 0000          lf0n  lf0n        lf1n
*
*    lf1,0[1833]=0; (jj=0)
*
     tc *   *   *  *         $ 06f1          lOad  lOad        l1ad
     tr *   *   *  *    *                    lf0n  e0    nop   nop
*
*    e0 = lf1[1777](a[0])
*
     tc *   *   *  *         $ 0701          lOad  lOad        l1ad
     tr *   *   *  *    *                    lf0n  g1    lf1n  g0
     tr *   *   *  *    g                    a0    g1    a1    g0
*
*    g1 = lf1[1793](b[0])
*
     tc *   clO *  *         $ 0001          nop   nop         e1g0
     tc *   *   *  *         $ 0002          nop   nop         cmr3
     tc *   *   *  *         $ 0016          nop   nop         icr0
*
*    This will be used to augment the original two values, so that
*    the program can shift b[jj] -> a[jj]  the 16 represents the 24
*    pixels allowed (max)
*
ljjp tr *   in0 *  *    g                    a0    lOad  a0    l1ad
     tr *   *   *  *    *                    lf0n  f0    lf1n  f1
     tr *   *   *  *    e                    a1    lOad  a0    nop
     tr *   *   *  *    e                    a0    nop   a1    l1ad
     tr *   *   *  *    add                  a1    nop   a0    g0
```

```
     tr *   *   *  *    add              a0    nop   a1    e1
     tc adn *   *  *         ljjp             mar         nop
     tr *   *   *  *    *                f0    lf0n  f1    lf1n
*
*    since  there  are  24  data  vectors maximum, this will move all of
*    them, whether they are there or not.  the two add's are the update.
*    of the address pointer.
*
     tc *   *   *  *         $ 000f           t0ba        t1ba
     tc *   *   *  *         $ 0711           tf0n        tf1n
     tc *   *   sr *         comf             mar         nop
     tc *   *   *  *         $ 0724           l0ad        l1ad
*
*
*    Calculate new c[jj]'s
*
*
     tc *   *   *  *         $ 0724           l0ad        l1ad
     tr *   *   *  *    *                lf0n  e0    lf1n  e1
     tr *   *   *  *    e+1              a0    e0    a0    e1
     tr *   *   *  *    e                a1    nop   a0    idx0
     tc *   *   *  *         $ 0001           nop         icr0
     tc adn *   *  *         ilp1             mar         nop
     tr *   *   *  *    e                a0    lf0n  a1    lf1n
     tr *   *   *  pupl *                nop   nop   mult  mcr3
*    Break point.
*
*    Update the value of i, where i is the number of rows.
*
 org 200
fexp tc *   *   *  *         $ 0001           g1          nop
     tc *   *   sr *         fpmr             mar         nop
     tc *   *   *  *         $ b8aa           nop         g0
*
*  Assume E=exp(E);
*  E=E*log(e) (log to the base 2)
*
     tc *   *   sr *         floo             mar         nop
     tr *   *   *  *    *                f0    brg0  f1    brg1
*
*  ent = floor(E)
*
     tc *   *   *  *         $ 0007           t0ba        t1ba
     tr *   *   *  *    *                f0    tf0u  f1    tf1u
*
*  save ent in temp file[7]
*
     tr *   *   *  *    *                nop   nop   f1    e1
     tc *   *   *  *         $ 8000           g1          nop
     tc *   *   sr *         fpar             mar         nop
     tr *   *   *  *    xor              a0    nop   a1    f1
*
```

```
*  fract = E - ent
*
     tc *   *   sr *          fpar                mar          nop
     tc *   *   *  *          $ 8000              brg0         brg1
*
*  fract=fract-0.5
*
     tr *   *   *  *    *                  f0     tf0u   f1     tf1u
*
*  Store fract in temporary file 8
*
     tr *   *   *  *    *                  f0     e0     f1     e1
     tc *   *   sr *          fpmr                mar          nop
     tr *   *   *  *    *                  f1     g1     f0     g0
*
*  xsq=fract*fract
*
     tr *   *   *  *    *                  f0     tf0d   f1     tf1d
*
* Store xsq in temporary file 9
*
     tr *   *   *  *    *                  f0     e0     f1     e1
     tc *   *   *  *          $ 0006              g1           nop
     tc *   *   sr *          fpmr                mar          nop
     tc *   *   *  *          $ f275              nop          g0
*
* temp1=p2*xsq
*
     tc *   *   *  *          $ 000f              nop          brg1
     tc *   *   sr *          fpar                mar          nop
     tc *   *   *  *          $ ec9d              brg0         nop
*
* temp1=temp1 + p1
*
     tr *   *   *  *    *                  f0     e0     f1     e1
     tc *   *   *  *          $ 0009              t0ra         t1ra
     tc *   *   sr *          fpmr                mar          nop
*
* temp1 = temp1 * xsq
*
     tr *   *   *  *    *                  tf1d   g1     tf0d   g0
     tc *   *   *  *          $ 0015              nop          brg1
     tc *   *   sr *          fpar                mar          nop
     tc *   *   *  *          $ fdf4              brg0         nop
*
* temp1 = temp1 + p0
*
     tr *   *   *  *    *                  f0     e0     f1     e1
     tc *   *   sr *          fpmr                mar          nop
     tr *   *   *  *    *                  tf1u   g1     tf0u   g0
     tr *   *   *  *    *                  f0     tf0u   f1     tf1u
*
```

```
* temp1 = temp1 * fract
* store in location 8 of the temporary file.
*
     tr *   *   *  *    *                  tf0n  f0    tf1n  f1
     tc *   *   *  *         $ 000b              nop         brg1
     tc *   *   sr *         fpar                mar         nop
     tc *   *   *  *         $ daa9              brg0        nop
*
* temp2 = xsq + q2
*
     tr *   *   *  *    *                  f0    e0    f1    e1
     tc *   *   sr *         fpmr                mar         nop
     tr *   *   *  *    *                  tf1n  g1    tf0n  g0
*
*  temp2 = temp2 * xsq
*
     tc *   *   *  *         $ 0013              nop         brg1
     tc *   *   sr *         fpar                mar         nop
     tc *   *   *  *         $ a005              brg0        nop
*
*  temp2 = temp2 + q1
*
     tr *   *   *  *    *                  f0    e0    f1    e1
     tc *   *   sr *         fpmr                mar         nop
     tr *   *   *  *    *                  tf1d  g1    tf0d  g0
*
*  temp2 = temp2 * xsq
*
     tc *   *   *  *         $ 0017              nop         brg1
     tc *   *   sr *         fpar                mar         nop
     tc *   *   *  *         $ b730              brg0        nop
*
*  temp2 = temp2 + q0
*
     tr *   *   *  *    *                  f0    tf0n  f1    tf1n
*
*  store temp2 in temporary location 9  (temp2 is already in f)
*
     tc *   *   sr *         fpar                mar         nop
     tr *   *   *  *    *                  tf0u  brg0  tf1u  brg1
*
*  F=temp1+temp2
*
     tr *   *   *  *    *                  f0    e0    f1    e1
     tc *   *   *  *         $ 0001              g1          nop
     tc *   *   sr *         fpmr                mar         nop
     tc *   *   *  *         $ b505              nop         g0
*
*  F=F*sqrt(2)
*
     tr *   *   *  *    *                  tf0d  brg0  tf1d  brg1
     tr *   *   *  *    *                  f0    tf0n  f1    tf1n
```

```
     tr *   *   *  *    *                tf0u  f0    tf1u  e1
*
*  BRG=temp2
*  save f in temporary location 9
*  F=temp1
*
     tc *   *   *  *         $ 8000             g1          nop
     tc *   *   sr *         fpar               mar         nop
     tr *   *   *  *    xor                     a0    nop   a1    f1
*
*  F=temp2-temp1
*
     tc *   *   sr *         fdiv               mar         nop
     tr *   *   *  *    *                tf0n  brg0  tf1n  brg1
*
*  F=BRG/F=(temp2+temp1)/(temp2-temp1)
*
     tc *   *   *  *         $ 0007             t0ba        t1ba
     tr *   *   *  *    *                tf1n  g1    tf0n  g0
*
*  get ent
*
     tc *   *   *  *         $ 3fff             nop         e1
     tr *   cl0 *  *    and                     a0    e0    a1    icr0
     tc *   *   *  *         $ 0001             nop         cmr3
     tc *   *   *  *         $ 0000             e0g1        nop
l00p tc ad  *   *  *         l00p               mar         nop
     sh ad  in0 *  *    *                nzinn nzinn lzinn s
*
*  get integer value of ent.
*
     tr *   *   *  *    *                nop   nop   f1    e1
     tr *   *   jp *    add              a0    nop   a1    e1
     tr *   *   df *    *                f0    e0    mult  mcr3
*
*  add to current power of two. Routine is now over.
*
fdiv tc *   cl0 *  *         $ 0000             nop         e1
     tr *   *   *  *    *                bsr0  e0    f0    g0
     tc *   *   *  *         $ 8000             g1          nop
     tc *   *   *  *         $ 0004             *           cmr0
     tc *   *   *  *         $ 0001             *           cmr3
loop tr *   *   *  *    e-g                     a0    f0    a0    icr0
     tr fnn *   *  *    *                       f0    e0    f0    icr0
     tr fnn *   *  *    add                     a0    nop   a1    e1
     tc ad  *   *  *         loop               mar         nop
     sh *   *   *  *    *                nzinn nzinn rzinn s
     tr *   *   *  *    e                       a1    f0    bsr1  e1
     tr *   *   *  *    zro                     f1    g1    a0    g0
     tc *   *   *  *         $ 4000             e0          nop
     sh *   *   *  *    *                lcir  nzinn lcir  s
     tr *   *   *  *    e-g                     a0    e0    a1    e1
```

```
     sh *   *   *  *    *                  rcir  nzinn nzinn s
     tc *   *   *  *         $ bfff              g1          g0
     tr *   *   *  *    and                a1    e0    a0    e1
     tr tnn *   jp *    e+1                a0    e0    a1    e1
     tr tnn *   df *    and                a1    nop   a0    f1
     sh *   *   jp *    *                  nzinn lzinn nzinn s
     tr *   *   df *    e                  a1    nop   a0    f1
floo tr *   cl0 *  *    zro                bsr1  e0    a1    e1
     sh uns *   *  *    *                  lzinn nzinn nzinn s
     tc *   *   *  *         $ 0001              nop         cmr0
     tc *   *   *  *         $ 0001              nop         cmr3
     tc tpn *   *  *         next                mar         nop
     sh *   *   *  *    *                  lzinn nzinn nzinn s
     tc tnn *   jp *         $ 0000              f0          f1
     tc tnn *   df *         $ 0000              e0          e1
     tc *   *   *  *         $ 0034              nop         g0
     tr *   *   *  *    e-g                a0    e0    a0    f1
     tr fnn *   jp *    *                  bsr0  f0    bsr1  f1
     tr fnn *   df *    *                  nop   nop   nop   nop
     sh *   *   *  *    *                  nzinn rzinn nzinn s
     sh *   *   *  *    *                  nzinn rzinn nzinn s
     tr *   *   *  *    *                  bsr0  e0    bsr1  icr0
l01p tc ad  *   *  *         l01p                mar         nop
     sh ad  in0 *  *    *                  lzinn nzinn nzinn s
     tr *   cl0 *  *    e                  a1    e0    a0    nop
l02p tc fnn *   *  *         l02p                mar         nop
     sh fnn *   *  *    *                  lzinn nzinn nzinn s
     tr *   *   jp *    e                  a0    f0    bsr1  f1
     tc *   *   df *         $ 0000              e0          e1
next tc tnn *   jp *         $ 8001              f0          f1
     tc tnn *   df *         $ 8000              f0          e1
     tc *   *   *  *         $ 0034              nop         g0
     tr *   *   *  *    e-g                a0    e0    a0    f1
     tr fnn *   jp *    *                  bsr0  f0    bsr1  f1
     tr fnn *   df *    *                  nop   nop   nop   nop
     sh *   *   *  *    *                  nzinn rzinn nzinn s
     sh *   *   *  *    *                  nzinn rzinn nzinn s
     tr *   *   *  *    *                  bsr0  e0    bsr1  f1
     sh *   *   *  *    *                  nzinn lzinn nzinn s
     sh *   *   *  *    *                  nzinn rzinn nzinn s
     tc *   *   *  *         $ 0000              nop         e1
     tr *   *   *  *    *                  nop   nop   f1    icr0
ls1p tc ad  *   *  *         ls1p                mar         nop
     sh ad  in0 *  *    *                  lzinn nzinn nzinn s
     tr *   cl0 *  *    e                  a1    e0    a0    nop
     tr ad  *   *  *    e+1                a0    e0    nop   nop
ls2p tc fnn *   *  *         ls2p                mar         nop
     sh fnn *   *  *    *                  lzinn nzinn nzinn s
     tr *   *   jp *    e                  a0    f0    bsr1  f1
     tc *   *   df *         $ 0000              e0          e1
fpmr tr *   *   *  *    e                  a1    e0    a0    q
*****************************************************************
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
*  load multiplicand                                                  *
*****************************************************************
     tr *   *   *  *    g                    a0    p     a1    g0
*****************************************************************
* load multiplier                                                   *
*****************************************************************
     tc *   *   *  *            $ 0004            *           cmr0
*****************************************************************
* this condition will check to see if f0 < 0.                       *
*****************************************************************
     tc *   *   *  *            $ 0002            *           cmr3
*****************************************************************
*    is index0 = compare register 0?                                *
*****************************************************************
     tr *   cl0 *  puqu add                  a1    *     a0    f1
     tr *   *   *  puqu zro                  mult  f0    a0    e1
     tc *   *   *  *         $ 0000                g1          g0
     tr *   *   *  *    *                    nop   nop   f0    g0
     tr *   *   *  puql g                    a0sw  g1    a1sw  g0
     tr *   *   *  puql *                    mult  e0    nop   nop
     tr *   *   *  plqu add                  a1    g1    a0    g0
     tr *   *   *  plqu *                    mult  e0    *     *
     tr *   *   *  *    add                  a1    g1    a0    g0
     tr *   *   *  *    g                    a0rz  f0    a1sw  nop
     tr *   *   *  *    *                    *     *     f1    e1
     tr *   *   *  *    *                    *     *     f0    icr0
*****************************************************************
*If the value returned is zero, zero both registers,   return*
*****************************************************************
     tc ad  *   jp *            $ 0000            f0          f1
     tc ad  *   df *            $ 0000            *           *
*****************************************************************
* If f0 is justified, return. The product is normalized.            *
*****************************************************************
     tc *   *   *  *            $ 0000            *           cmr1
     tc tnn *   jp *            $ bfff            g1          *
     tr tnn *   df *    and                  a0    *     a1    f1
*****************************************************************
*    Save the exponent in g1, clear E1 for a counter                *
*****************************************************************
     tr *   *   *  *    zro                  f1    g1    a0    e1
     tr *   cl0 *  *    zro                  a0    e0    *     *
*****************************************************************
* By here, product cannot be zero.The normalization process         *
* will take less than four repeats of this loop. If it ever         *
* takes more, there is something branching directly to this         *
* process.                                                          *
*****************************************************************
nrm  tr fnn in0 *  *    e+1                  a0    e0    a1    *
     sh fnn *   *  *    *                    nzin  lzin  nzin  s
*****************************************************************
*    By now, the result must be normalized!!!!!                     *
```

```
*************************************************************
     tr *   *   *  *    g                  a0    *     a1    f1
     tr *   *   *  *    e                  a0    g1    a1    g0
     tr *   *   *  *    *                  f0    e0    f1    e1
     sh *   *   *  *    *                  lcir  lcir  lzin  s
     tr *   *   *  *    e-g                a0    e0    a1    e1
*************************************************************
*This will take the normalized result, shift it left, adjust *
*the expontent, so that it agrees with the mantissa.        .*
*************************************************************
     tc *   *   *  *           $ 7fff           g1          *
     tc *   *   *  *           $ ffff           *           g0
     tr *   *   jp *    acb                    a0    *     a1    f1
*************************************************************
* this will "mask off" any carries into the unused portion of*
* the exponent.                                              *
*************************************************************
     sh *   *   df *    *                      nzin  rcir  nzin  s
*****************************Begin Routine FPAR*****************************
*
*    This is the floating point addition routine. 3/1/80
fpar tr *   *   *  *    *                      bsr1  g1    f1    e1
     sh uns cl0 *  *    *                      lzin  nzin  lzin  s
     sh *   *   *  *    *                      rzin  nzin  rzin  s
*
*    This will strip the sign of the mantissa and save it for future use.
*
     tr *   *   *  *    *                      *     *     bsr0  icr0
     tc tnn *   *  *         $ 0000                 *           cmr0
     tc tnn *   *  *         $ 0010                 *           cmr1
     tr tnn *   jp *    zro                    a0    e0    a1    e1
     tr tnn *   df *    zro                    a0    g1    a1    g0
     tr *   *   *  *    zro                    a0    *     a1    cmr1
*
*    this will compare the brg to zero, if it is, return.
*
     tr *   *   *  *    zro                    a0    e0    a1    g0
*
*    This will zero the registers to prevent spurious results.
*
     tr *   *   *  *    e-g                    a0    nop   a1    icr0
*
*    if |e|<|g|, the program will reverse the numbers and continue.
*    since addition is commutative, this should not affect the results.
*
     tr *   *   *  *    xor                    a1    e0    a0    nop
     tc *   *   *  *         $ 2000                 *           g0
     tr *   *   *  *    and                    a1    *     a0    g0
     tc *   *   *  *         $ 8000                 e0          *
     tr *   *   *  *    e-g                    a1    *     a0    g0
     tc *   *   *  *         $ 0010                 *           cmr0
     tc fnn *   *  *         nsh                    mar         *
```

```
*
*    If the exponent on one of the two numbers is less than zero
*    and the other is not, subtraction to yield the number of shifts
*    will not yield the correct answer, and thus special handling
*    must be added to compensate for this problem. The way that this
*    routine handles the problem is it exclusive ors the two numbers
*    together and then strips off everything but the sign bit.  This
*    is then subtracted from a constant (for speed).The constant is
*    8000, thus if there is a 1 in the sign position, the result will
*    not be negative, indicating that the correction must take place.
*
     tc tnn *   *  *         $ 0020              *          cmr0
     tr *   *   *  *    e-g                 a1   g1    *     *
*
*    This will test for E-G->G negative. This is to insure that |brg| >=
*    |f|, simplifying the algorithm greatly.
*
     tc tnn *   *  *    swap                      mar        nop
*
*    This involkes the swap routine that will force the above to be true.
*
     tr *   *   *  *    zro                 *     *     a0   cmr0
     tc *   *   *  *    $ 0010                    *          cmr1
*
* the zeroes that are loaded into condition mask register 0 tell
* the machine not to check for any of the conditions represented.
* the 0010 loaded into cmr1 tell the machine to check for the compare
* register greater than index register one.  In this case, this will
* determine whether the two numbers are equal or equal and opposite in
* magnitude and sign.
*
     tr *   *   *  *    zro                 a0   g1    a1   e1
     tc tnn *   *  *    equl                      mar        nop
*
*    If they are, the program will jump to a special routine.
*
*
*    By this point in the program, |e|>|g|.
*
     tr *   *   *  *    *                   bsr0 e0    f0   g0
     tc *   *   *  *         $ 0010               *          idx0
     tc *   *   *  *         $ 0020               *          cmr1
     tc tnn *   *  *         rtnf                 mar        nop
*
*    If the number of shifts required > 16, return the value in the
*    F register.
*
     tr *   *   *  *    zro                 a1   g1    a0   cmr1
     tc *   cl0 *  *    $ 0001                    *          cmr3
*
*    this loads the data to be processed and it programs the CPU to check
*    for reg0#indx0. This is represented by a one in the first position. This
```

```
*    check is involked by the AD command.
*
shft sh *   in0 *  *    *                  rzin  nzin  nzin  s
     tc ad  *   *  *    shft                     mar         nop
*
* Index register contains the amount by which G>E, (the number of orders
* of magnitude. This routine shifts E to the right until the two orders of
* magnitude are equal.
*
     tc *   *   *  *    $ 0000                   g1          *
     tc *   *   *  *    $ 0020                   *           cmr0
     tc fpn *   *  *    gpos                     mar         nop
*
*    if g1 >= 0, its sign is taken to be positive, and the numbers are
*    handeled in a corresponding manner.
*
*
*    By this point, g must be negative.
*
     tc *   *   *  *    $ 0002                   *           cmr0
     tc tpn *   *  *    ssgn                     mar         nop
*
*    If E is negative, and G is negative, the signs are the same and the
*    two numbers are just added and one of the signs is preserved.
*
     tc *   *   *  *                        *     *     *     *
*
*    At this point, |G|>|E|, the resultant sign will be that of G.
*    Without regard to sign, the result will be the old sign of G
*    plus |g-e|.
*
dsgn tr *   *   *  *    en                   a0    e0    *     *
     tr *   *   *  *    e+1                  a0    e0    *     *
     tr *   *   *  *    add                  f1    e0    a0    g0
*
*    This calculates g-e.
*
     tc *   *   *  *    $ 0010                   *           cmr0
*
*    If the result is >= zero, there is not a one in tthe first bit position,
*    so the number is not normalized, and must be shifted until there appears
*    a '1' in the first bit position.
*
norm tr fnn *   *  *    e-1                  a0    e0    *     *
     tc fnn *   *  *    norm                       mar         nop
     sh fnn *   *  *    *                    nzin  nzin  lzin  s
*
*    This routine normlizes the data
*
     tr *   *   *  *    g                    a0    f0    *     *
     tc *   *   *  *         $ 3fff               *           g0
     tr *   *   *  *    and                  *     *     a0    f1
```

```
     tc *   *   *  *        $ 0002              *          cmr0
     sh *   *   *  *   *                 nzin  lzin  nzin  s
     sh tpn *   *  *   *                 nzin  roin  nzin  s
     sh fpn *   *  *   *                 nzin  rzin  nzin  s
     tc *   *   jp *        $ 0000              e0g1       e1g0
     tc *   *   df *        $ 0000              *          *
*
*    this routine sets the sign to the sorrect sign and returns to the calling
*    routine.
*
gpos tc tpn *   *  *   dsgn                    mar        nop
     tr *   *   *  *   *                 *     *     *    *
*
*    Before the jump to GPOS, the condition register was set to check for
*    e<0. If it is, the signs are opposite and the data is treated
*    correspondingly.
*
*
*    By default, both G and E have the same sign, so the results are just
*    added.
*
     tr *   *   *  *   *                 *     *     f0   g0
ssgn tr *   *   *  *   add               a1    g1    a0   g0
     sh *   *   *  *   *                 nzin  nzin  rcir s
     tc *   *   *  *   $ 0010                 nop        cmr0
*
*    this checks for a carry out of he MSB, indicating normalization is
*    necessary.
*
     tr tnn *   *  *   g                   f1    e0    *    *
     tr tnn *   *  *   e+1                 a0    e0    *    *
     tr tnn *   *  *   g                   a0    f0    *    *
     tc tnn *   jp *        $ bfff               *          g0
     tr tnn *   df *   and                 *     *     a0   f1
*
*    If it is, then the number is normalized and the subroutine returns.
*
     sh *   *   jp *   *                   nzin  nzin  lcir s
     tr *   *   df *   g                   a0    f0    *    *
*
*    This routine exchanges the two registers involved so that |G|>|E|
*
swap tr *   *   *  *   *                   f1    g1    f0   g0
     tr *   *   *  *   *                   bsr0  f0    bsr1 f1
     tc *   *   *  *         fpar                mar        nop
     tr *   *   *  *   g                   a0    brg0  a1   brg1
*
*    This calls the original routine.
*
*
*    This is the action taken when the routines have the same magnitude.
*
```

```
equl tc *   *   *  *    $ 0000                    nop          cmr1
     tc *   *   *  *    $ 0002                    nop          cmr0
     tc fpn *   *  *    epos                      mar          nop
     tc *   *   *  *    $ 0020                    nop          cmr0
*
*    is e1 >= 0  (originally ie) was f   => 0?)
*
     tc tp  *   *  *    ssgn                      mar          nop
     tc *   *   *  *    $ 0000                    nop          nop
*
*    if f1 < 0 then the two have similar signs, and should be added.
*    By this point, the brg is negative for sure, and the f1 register
*    is positive. Reverse the two and continue processing.
*
     tr *   *   *  *    *                 f1      e0     bsr1  g0
     tr *   *   *  *    e                 a0      g1     nop   nop
     tr *   *   *  *    g                 a1      nop    a0    e1
     tr *   *   *  *    *                 bsr0    e0     f0    g0
     tr *   *   *  *    e                 a0      f0     a1    f1
     tr *   *   *  *    g                 a0      brg0   a1    brg1
     tr *   *   *  *    g                 a0      e0     a1    e1
     tr *   *   *  *    *                 f1      g1     f0    g0
     sh uns *   *  *    *                 lzinn   lzinn  lzinn s
     sh *   *   *  *    *                 rzinn   rzinn  rzinn s
     tc *   *   *  *         lbll                 mar          nop
     tr *   *   *  *    zro               a0      g1     a1    e1
*
*    the two numbers are reversed the the program can continue processing
*    (ie. e contains the negative number.)
*
epos tc fp  *  -*-*    ssgn                      mar          nop
lbll tc *   *   *  *    $ 0100                    nop          cmr0
     tr *   *   *  *    e=g               nop     nop    nop   nop
     tc tn  *   *  *    zapp                      mar          nop
*
*    by here E and g have different signs. If they have the same
*    mantissa, the result should be zero. This is what the above
*    lines of code determine.
*
     tc *   *   *  *    $ 0020                    nop          cmr0
     tr *   *   *  *    e-g               a1      g1     a0    nop
     tc tn  *   *  *    lbl1                      mar          nop
     tc fn  *   *  *    lbl1                      mar          nop
     tr *   *   *  *    *                 nop     nop    bsr1  f1
*
*    if the mantissa of g is greater than the mantissa of e, force the
*    resultant sign and exponent to be that of g. Else force it to be
*    e.
*
lbl1 tr *   *   *  *    e-g               a0      g1     nop   nop
     tr *   *   *  *    g                 a1      e0     nop   nop
     tr tnn *   *  *    en                a0      e0     nop   nop
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
     tr tnn *   *  *    e+1                 a0    e0    nop   nop
     tr *   *   *  *    e                   f1    e0    a0    g0
     tc *   *   *  *    $ 0010                    *           cmr0
nnrm tr fnn *   *  *    e-1                 a0    e0    *     *
     tc fnn *   *  *    nnrm                      mar         nop
     sh fnn *   *  *    *                   nzin  nzin  lzin  s
*
*    This routine normlizes the data
*
     tr *   *   *  *    g                   a0    f0    *     *
     tc *   *   *  *         $ 3fff               *           g0
     tr *   *   *  *    and                 *     *     a0    f1
     tc *   *   *  *         $ 0002               *           cmr0
     sh *   *   *  *    *                   nzin  lzin  nzin  s
     sh tpn *   *  *    *                   nzin  rzin  nzin  s
     sh fpn *   *  *    *                   nzin  roin  nzin  s
     tc *   *   jp *         $ 0000               e0g1        e1g0
     tc *   *   df *         $ 0000               *           *
*
*    this routine sets the sign to the correct sign and returns to the calling
*    routine. It is seperate because somewhere a sign convention changed.
*
zapp tr *   *   jp *    zro                 a0    f0    a1    f1
     tc *   *   df *    $ 0000                    nop         cmr0
*
*    this routine handles numbers that have different exponential signs.
*
nsh  tc *   cl0 *  *         $ 0010              nop         cmr0
*
*    bug in assembler.  null line will not be assembled.  By this point
*    in the program, the exponent on one of the two numbers must
*    be less than zero. This part of the routine will force the negative
*    part to be stored in brg register. Since a swap can take place,
*    all the original flags must be reset in the event of ashift.
*
     tr *   *   *  *    g                   a1sw  *     a0rz  g0
     tc tnn *   *  *         glz                 mar         nop
*
*    The g/brg register contains the negative exponent, no swap needed.
*
     tr fnn *   *  *    *                   f1    g1    f0    g0
     tr *   *   *  *    *                   bsr0  f0    bsr1  f1
     tr *   *   *  *    g                   a0    brg0  a1    brg1
     tr *   *   *  *    *                   bsr1  g1    f1    e1
     sh uns *   *  *    *                   lzin  nzin  lzin  s
     sh *   *   *  *    *                   rzin  nzin  rzin  s
*
*    This swaps the two numbers and resets all the flags needed by the
*    rest of the routine.
*
glz  tc *   *   *  *         $ 0000              e0          g0
     tr *   *   *  *    e-g                 a1sw  *     a0sw  g0
```

```
     tr *   *   * *    g                  a1rz  nop   a0rs  icr0
*
*    Calculate the number of shifts needed, if it is < 0, it is
*    actually > 80 (16), so return the value in the F register.
*
     tc tnn *   * *         rtnf                  mar         nop
     tr *   *   * *    g                  a0sw  e0    a1rz  nop
     tc *   *   * *         $ 0010                *           g0
     tr *   *   * *    e-g                a1    nop   a0    g0
     tc fnn *   * *         rtnf                  mar         nop
*
*    If the number of shifts required is > 16, return the data in the
*    F register.
*
     tc *   *   * *         $ 0000                *           cmr0
     tc *   *   * *         $ 0000                *           e1
     tr *   *   * *    *                  bsr0  e0    f0    g0
     tc *   *   * *         shft                  mar         nop
     tc *   *   * *         $ 0001                *           cmr3
*
*    prepair to shift the data and return to shifting routine,
*
rtnf tr *   *   jp *    zro                a0    e0g1  a1    e1g0
     tr *   *   df *    *                  *     *     *     *
*
*    Return the contents of the F register.
*
comf tc *   *   * *         $ 0000                brg0        brg1
     tc *   *   * *         $ 0006                t0wa        t1wa
     tc *   *   * *         $ 0000                tf0n        tf1n
*
*    the 0 in location 6 of the temporary files is a cycle counter.
*    it keeps track of the class currently being worked upon.
*
*    After normalizing the data vector, store it, repeat until all
*    the elements are finished, then repeat the cycle until all four elements
*    are finished being processed.
*
     tc *   *   * *         $ 000a                t0ba        t1ba
     tr *   *   * *    *                  lf0n  e0    lf1n  nop
     tc *   *   * *         $ 0102                nop         g0
*
*    the data vectors are stored in location 100 of the large file
*
     tr *   *   * *    add                a0    l0ad  a0    l1ad
     tr *   *   * *    *                  lf0u  tf0u  lf1u  tf1u
     tr *   *   * *    *                  lf0u  tf0u  lf1u  tf1u
     tr *   *   * *    *                  lf0u  tf0u  lf1u  tf1u
     tr *   *   * *    *                  lf0u  tf0u  lf1u  tf1u
     tc *   *   * *         $ 0003                nop         icr3
     tc *   *   * *         $ 000a                t0ba        t1ba
     tc *   cl3 * *         $ 0002                t0wa        t1wa
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
     tc *   *   *  *        $ 0010              l0ad        l1ad
lo1p tr *   in3 *  *    *                tf0u  brg0  tf1u  brg1
     tc *   *   sr *        fpar                mar         *
     tr *   *   *  *    *                lf0u  f0    lf1u  f1
     tc *   *   *  *        $ 0040              *           cmr3
     tc ad  *   *  *        lo1p                mar         *
     tr *   *   *  *    *                f0    tf0u  f1    tf1u
     tc *   cl3 *  *        $ 000a              t0ra        t1ra
*
*    This stores the data normalized data vector in locations 2-5 of the
*    temporary file.  Location 6 is used for a counter. The results will
*    be stored in the value pointed to by location 7 of the temporary file.
*
*    This will fall through to the matrix processing routine.
*
*
*
*    This is the beginning of the matrix multiply routine.
*
stra tc *   cla *  *        $ 0000              brg0        brg1
     tc *   *   *  *        $ 0006              t0ra        t1ra
     tr *   *   *  *    *                tf0n  p     *     *
     tc *   *   *  *        $ 0010              *           q
     tc *   *   *  plql     $ 0040              *           g0
* was $ 0000 to * nop!
     tr *   *   *  plql *                mult  e0    nop   nop
*
*    tc *   *   *  *        $ 0040              nop         g0
*    this was here.
*
     tr *   *   *  *    add                 a0    l0ad  a0    l1ad
     tc *   *   *  *        $ 0001              t0ba        t1ba
     tc *   *   *  *        $ 0001              tf0u        tf1u
*
*    For indirectly addressing the current row of the normalized
*    data vector
*
     tc *   *   *  *        $ 0002              t0ba        t1ba
*
*
*
mlty tr *   in1 *  *    *                tf0u  e0    tf1u  e1
*
* This loads the multiplicand into the e0-e1 register pair.
*
     tc *   *   sr *    fpmr                    mar         nop
*
* This does the program jump to the floating point multiply routine.
*
     tr *   *   *  *    *                lf1u  g1    lf0u  g0
*
*
```

```
* This step is done before the jump is actually executed. This will load the
* multiplier into the g0-g1 register pair. (F=EXG floating point mult)
*
     tc *   *   sr *          fpar                 mar          nop
*
* This step will do a jump to the floating point addition routine.This rout-
* ine calculates the sum of the contents of the F register and the BRG regis-
* ter pair. The result of the add is then stored in the F register.
*
     tc *   *   *  *          $ 0004               nop          icr1
     tc *   *   *  *          $ 0004               nop          cmr3
*   the 0004 tests for index1 <> its compare
*
*
* This is executed before the jump.  It will just load the condition register
* with the next condition to be tested.
*
     tc ad  *   *  *          mlty                 mar          nop
     tr *   *   *  *    *                  f0     brg0  f1     brg1
*
* On index register 1 not equal to its compare, jump to beginning of multiply
* routine.
*
*
     tc *   *   *  *          $ 0001               t0ba         t1ba
     tr *   *   *  *    *                  tf1n   e0    nop    nop
*
*    get address of jth item in the data vector.
*
     tr *   *   *  *    ac0                a1     tf0d  a0     tf1d
     tr *   *   *  *    ac0                a0     t0ra  a0     t1ra
*    tr *   *   *  *    ac0                a1     *     a0     t1ra
* the above was a change to insure that the program works, this is kept.
* this will update the address for the next round, store it, and point to the
* item in question.
*
     tr *   in2 *  *    *                  tf0c   e0    tf1c   e1
*
* this will load the multiplier for the second multiply into the e0-e1 reg-
* ister pair.  Simultaneously, this will zero the temp file pointers. They
* will now point to the location of the accumulator.
*
     tr *   *   *  *    *                  bsr0   g1    bsr1   g0
     tc *   *   sr *    fpmr                      mar          nop
     tr *   *   *  *    g                  a0     g1    a1     g0
*
* this is just a subroutine jump to the floating point multiply routine.
* f=EXG
     tc *   *   sr *          fpar                 mar          nop
     tr *   *   *  *    *                  tf0n   brg0  tf1n   brg1
*
*    F=F+BRG. This calculates the subtotal of the matrix multiply.
```

```
*
     tr *   *   *  *    *                    f0    tf0n  f1    tf1n
     tc *   *   *  *         $ 0002          t0ba          t1ba
*
* The above two steps load the sub total into the temporary file location
* zero.  It them resets the read and write pointers of the temporary file t
* location two.
*
     tc *   *   *  *         $ 0004          nop           icr2
     tc *   *   *  *         $ 0010          nop           cmr3
*   the 0010 tests for index2 # its compare.
* This will do a test for index 0 not equal to its compare register.
*
     tc ad  cl1 *  *         mlty                  mar           nop
     tc *   *   *  *         $ 0000                brg0          brg1
     tr *   *   *  puql *                    *     *     mult    mcr3
     tc *   *   *  *         $ 0000                t0ba          t1ba
     tc *   *   *  *         $ 8000                g1            nop
     tc *   *   *  *         $ 0000                nop           g0
     tr *   *   *  *    *                   tf0n  e0    tf1n    e1
     tr *   *   *  *    xor                 a0    brg0  a1      brg1
*
*    quadratic = quadratic * -1
*
     tc *   *   *  *         $ 0006          t0ba          t1ba
     tr *   *   *  *    *                   tf0n  l0ad  tf0n    l1ad
*
*  location of log(|sigma|)
*
     tc *   *   sr *         fpar                  mar           nop
     tr *   *   *  *    *                   lf0n  f0    lf1n    f1
     tr *   *   *  puql *                    *     *     mult    mcr3
*
*    calculate log(det(sigma))+quadratic
*
     tr *   *   *  *    zro                 a0    e0    f1      e1
     tc *   *   *  *         $ 0001                g1            nop
     tr *   *   *  *    zro                 f0    e0    a0      g0
     sh *   *   *  *    *                   lcirn lcirn lcirn   s
     sh *   *   *  *    *                   lcirn lcirn lcirn   s
     tr *   *   *  *    e-g                 a0    e0    a1      e1
     sh *   *   *  *    *                   rcir  rcir  rcir    s
     sh *   *   *  *    *                   rcir  rcir  rcir    s
     tr *   *   *  plqu *                    nop   nop   mult    mcr3
etst tc *   *   *  *         $ 0002                nop           cmr0
     tc fnn *   *  *         pos                   mar           nop
*
*    For some reasons, exp only works for positive exponents. This
*    addition should change the data so that it is positive. It will
*    then take the result and invert it by dividing by one.
*
     sh tnn *   *  *    *                   lzin  nzin  nzin    s
```

```
*
*    By here the number is negative. This strips off the sign.
*
     tc *   *   sr *         fexp                mar          nop
     sh *   *   *  *    *                  rzin  nzin  nzin   s
*
*    call the exponentiation routine.
*
     tc *   *   *  *         $ 8000              brg0         nop
     tc *   *   *  *         $ 0001              nop          brg1
*
*    This is 1.0 in the computers notation.
*
     tc *   *   sr *         fdiv                mar          nop
     tr *   *   *  *    e                   a0    f0    a1     f1
     tr *   *   *  plqu *                    nop   nop   mult   mcr3
*
*    This fdiv routine will calculate 1/f, which is the same result
*    as the exp should give using the correct exponent.
*
     tc *   *   *  *         meet                mar          nop
     tr *   *   *  *    *                   f0    e0    f1     e1
*
*    Skip the next few statements.
*
pos  tc *   *   sr *         fexp                mar          nop
     tr *   *   *  *    *                   *     *     *      nop
*
*    by here, the routine is "OK" for positive numbers. No special processing
*    needed here. When done, just fall through
*
meet tr *   *   *  *    *                   nop   nop   *      *
*
* OKOKOKOKOKOKOKOKOKOKOKOKOKOKOKOKOKOKOK to the first time through!!!!!
*
*    this calculates  exp{.5[log(det(inv(sigma)))-quadratic]}
*
     tr *   *   *  plqu *                    nop   nop   mult   mcr3
     tc *   *   *  *         $ 000f              t0ba         t1ba
     tr *   *   *  *    *                   tf0n  e0    nop    nop
     tr *   *   *  *    e+1                 a0    tf0n  a0     tf1n
     tr *   *   *  *    e                   a0    l0ad  a0     l1ad
     tr *   *   *  *    *                   bsr0  lf0n  bsr1   lf1n
*
*    store new value in the location given in tf0n[f]
*
*
* this will increment tf0n[6], the pointer to this array.
*
     tc *   *   *  *         $ 0006              t0ba         t1ba
     tr *   *   *  *    *                   tf0n  e0    *      *
     tr *   *   *  *    e+1                 a0    tf0c  a0     idx0
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
     tr *   *   *  *    *                    tf0n  lf0n  tf1n  lf1n
     tc *   *   *  *         $ 0004               *           icr0
     tc *   *   *  *         $ 0001               *           cmr3
     tc *   *   *  *         $ 0004               p           *
     tr *   *   *  *    e                    a1    *     a0    q
     tc *   *   *  plql      $ 0151               *           g0
     tr *   *   *  plql *                    mult  e0    tf0n  idx0
     tr *   *   *  *    add                  a0    l0ad  a1    *
     tr *   *   *  *    add                  a1    *     a0    l1ad
     tc *   *   *  *         $ 0000               t0ba        t1ba
     tc *   *   *  *         $ 0000               tf0u        tf1u
     tc *   *   *  *         $ 0001               tf0u        tf1u
     tc *   *   *  *         $ 000a               t0ba        t1ba
     tc *   *   *  *         $ 0002               t0ra        t1ra
     tc adn *   *  *         lo1p                 mar         *
     tc *   cla *  *         $ 0000               e0          e1
finl tc *   *   *  *         $ 0150               l0ad        l1ad
     tr *   *   jp pupu *                    mult  brg0  *     *
     tc *   *   df *         $ 0000               nop         nop
*
*
*
fcmp tc *   *   *  *         $ 0000               t0ba        t1ba
*****************************************************************
* This accepts the data in the E register and G register as     *
* Inputs. Initially, the program stores the original data in    *
* temporary file. the E register goes in location 0 and the     *
* G register goes in location 1. The following will also        *
* strip off the sign bit                                        *
*****************************************************************
     tr *   *   *  *    e                    a0    tf0u  a1    tf1u
     tr *   *   *  *    g                    a0    tf0u  a1    tf1u
*****************************************************************
* This routine strips off the sign bit. The correct sign bit    *
* is saved in the PAST register.                                *
*****************************************************************
     sh uns *   *  *    *                    lzin  nzin  lzin  s
     sh *   *   *  *    *                    rzin  nzin  rzin  s
     tr *   *   *  *    e                    a1sw  e0    a0sw  e1
     tr *   *   *  *    g                    a0sw  g1    a1sw  g0
*****************************************************************
* The 0002 in cmr0 will check for E1 negative. This is done     *
* in the past sense. If E1<0, then jump to the routine that     *
* will handle that case.                                        *
*****************************************************************
     tc *   *   *  *         $ 0002               nop         cmr0
     tc *   *   *  *         $ 0000               nop         cmr1
     tc tpn *   *  *         emng                 mar         *
*****************************************************************
* by this point, the E register must not be negative (>=0)      *
* the 0020 in the cmr0 will test for g<0. If g<0, e is the      *
* greater of the two numbers. If not, they are both >= 0.       *
```

```
*****************************************************************
     tc *   *   *  *        $ 0020              *           cmr0
     tc tpn *   *  *        egrt                mar         *
*****************************************************************
* This will determine if there is a difference in exp sgn.   *
*****************************************************************
     tc tpn *   *  *        $ 0000              *           cmr1
     tc tnn *   *  *        gxng                mar         *
*****************************************************************
* This will do a jump if the sign of G is 1, or G negative   *
* in the exponent portion.                                    *
*****************************************************************
     tc *   *   *  *        $ 0002              *           cmr0
     tc tnn *   *  *        ggrt                mar         *
*****************************************************************
* By here, the exponent of g is positive. If the exponent of *
* E is negative, both mantissas being positive, e<g          *
*****************************************************************
     tr fnn *   *  *   e-g                a0    e0    a1    e1
     tc tnn *   *  *        ggrt          mar         *
     tc fnn *   *  *        $ 0000              f0          f1
     tc *   *   jp *        $ 0000              t0ba        t1ba
     tr *   *   df *   *                  tf0n  e0    tf1n  e1
*****************************************************************
* Since both exponents and mantissas are nonnegative, this   *
* routine calculates e-g, exponents in the HOBP and mantissas*
* in the LOBPs. If the result is < 0, g>e, else return E.    *
*****************************************************************
ggrt tc *   *   *  *        $ 0001              t0ba        t1ba
     tc *   *   jp *        $ 0001              f0          f1
     tr *   *   df *   *                  tf0n  e0    tf1n  e1
*****************************************************************
* if f1>=0, e>g, return tf[1]                                 *
*****************************************************************
egrt tc *   *   *  *        $ 0000              f0          f1
     tc *   *   jp *        $ 0000              t0ba        t1ba
     tr *   *   df *   *                  tf0n  e0    tf1n  e1
*****************************************************************
* This is the section of the program that is called if E is  *
* negative. (mantissa)                                        *
*****************************************************************
emng tc *   *   *  *        $ 0020              *           cmr0
     tc fpn *   *  *        ggrt                mar         nop
*****************************************************************
* This section does the compare if both the operands are < 0 *
* This will determine if there is a difference in exp sgn.   *
*****************************************************************
ncmp tc *   *   *  *        $ 0000        *           *
     tc tnn *   *  *        gbng          mar         *
*****************************************************************
* This will do a jump if the sign of G is 1, or G negative   *
* in the exponent portion.                                    *
```

225


```
*****************************************************************
     tc *   *   *  *         $ 0002              *           cmr0
     tc fnn *   *  *         nnpp          mar         *
*****************************************************************
* By here, the exponent of g is positive. If the exponent of *
* E is negative, both mantissas being negative, e>g          *
*****************************************************************
     tc tnn *   *  *         $ 0000              f0          f1
     tc *   *   jp *         $ 0000              t0ba        t1ba
     tr *   *   df *    *                   tf0n  e0    tf1n  e1
*****************************************************************
* The above will return e                                     *
*****************************************************************
gbng tc tnn *   *  *         ebng          mar         *
*****************************************************************
* Both G's exponent and sign are negative. If true, the same *
* holds true for E. If this is false, return g.              *
*****************************************************************
     tc *   *   *  *         $ 0001              t0ba        t1ba
     tc *   *   jp *         $ 0001              f0          f1
     tr *   *   df *    *                   tf0n  e0    tf1n  e1
*****************************************************************
ebng tr *   *   *  *    e-g                 a0    e0    a1    e1
     tc fnn *   *  *         ggrt          mar         *
     tc *   *   *  *         $ 0000              f0          f1
     tc *   *   jp *         $ 0000              t0ba        t1ba
     tr *   *   df *    *                   tf0n  e0    tf1n  e1
*****************************************************************
*  Both the mantissa and the exponent of both E and G are     *
* less than zero.  calculate e-g. if result positive, g>e     *
*****************************************************************
gxng tc fnn *   *  *         egrt          mar         nop
     tc *   *   *  *         $ 0000        *           *
test tr *   *   *  *    e-g                 a0    e0    a1    e1
     tc fnn *   *  *         egrt                mar         nop
     tc tnn *   *  *         ggrt                mar         nop
     tc *   *   *  *         $ 0000        *           *
*****************************************************************
* at this (preceeding line) both E and G are positive. The    *
* sign of the exponent of g is negative. If the sign of the   *
* exponent of E is positive, e>g, hence return E.             *
*****************************************************************
nnpp tr *   *   *  *    e-g                 a0    e0    a1    e1
     tc tnn *   *  *         egrt                mar         nop
     tc fnn *   *  *         ggrt                mar         nop
     tc *   *   *  *         $ 0000        *           *
#
```

ORIGINAL PAGE IS
OF POOR QUALITY

SEP 1980
RECEIVED
NASA STI FACILITY
ACCESS DEPT.