



The design of a matrix-oriented cellular computer capable of fault-tolerant operation
by Michael Carl Mulder

A thesis submitted to the Graduate Faculty in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY in Electrical Engineering
Montana State University
© Copyright by Michael Carl Mulder (1970)

Abstract:

The subject of this thesis is the design of a cellular computer specially organized to efficiently implement matrix algorithms, and capable of fault-tolerant operation.

The contents of the thesis are summarized as follows: First, a review of pertinent literature in the areas of cellular logic design, array-structured computers, fault-tolerant computing systems, and large scale integration of digital circuits is presented. Second, an assembly language instruction set and an array-structured, computer simulation program (PAL-1) are described including as an example a Kalman filter algorithm implementation. Third, the computer system organization and logical design of an array-structured computer are described including a discussion of the LSI implementation of a processing cell. Fourth, an adaptation of the array-structured computer design to fault-tolerant operation is presented. Three fault-tolerant computer organizations are described which use a special instruction set allowing fault detection, fault correction, and program rollback operations. Fifth, recommendations are made for further research in the area of array-structured computers capable of fault-tolerant operation.

© Copyright 1971 by Michael Carl Mulder,
Bozeman, Montana. All rights reserved.
No part of this work may be reproduced in
any form, or by any means, without permission
in writing from the author.

THE DESIGN OF A MATRIX-ORIENTED CELLULAR COMPUTER

CAPABLE OF FAULT-TOLERANT OPERATION

by

MICHAEL CARL MULDER

A thesis submitted to the Graduate Faculty in partial
fulfillment of the requirements for the degree

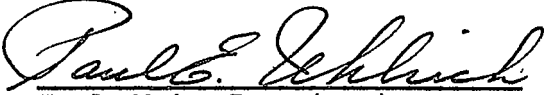
of

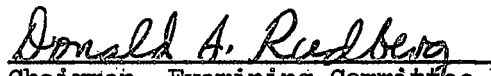
DOCTOR OF PHILOSOPHY

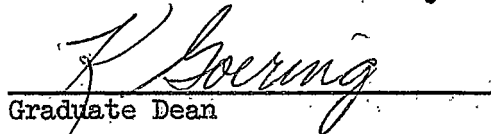
in

Electrical Engineering

Approved:


Head, Major Department


Chairman, Examining Committee


Graduate Dean

MONTANA STATE UNIVERSITY
Bozeman, Montana

August, 1970

ACKNOWLEDGMENT

The author wishes to extend his gratitude for the guidance and encouragement rendered to him by Dr. Donald A. Rudberg during the course of this graduate work and research effort. Concurrently, special appreciation is given to Dr. Robert C. Minnick for his timely suggestions and efforts as a consultant for this research endeavor.

The financial support provided by the Department of Electrical Engineering at Montana State University and the Office of Naval Research under Contract No. N00014-67-C-0477 is gratefully acknowledged.

To a family of cheerful fortitude and endurance--my wife Carol, my daughter Jennifer Leigh, and my son Jonathan Michael, I dedicate this thesis.

TABLE OF CONTENTS

	Page
Chapter 1	INTRODUCTION AND REVIEW OF PERTINENT
	LITERATURE 1
1.1	Introduction 2
1.2	Survey of Microcellular Work 4
1.3	Survey of Array-Organized Computers 8
1.3.1	The Unger Spatially-Oriented Computer 8
1.3.2	The Holland Computer 10
1.3.3	The Comfort Computer 12
1.3.4	The SOLOMON Computer 13
1.3.5	The Gonzalez Iterative Circuit Computer 15
1.3.6	The ILLIAC IV Computer 16
1.3.7	The MCB Computer 19
1.3.8	The Berkeley Array Processor 20
1.4	Survey of the Cannon Cellular Computer 22
1.5	Current Trends in Fault-Tolerant Computers 26
1.6	Current Trends in Large Scale Integrated Circuits 28
1.7	Description of Research Effort 30
1.8	Organization of Remaining Chapters 32

	Page
Chapter 2	DESIGN OF A PARALLEL ASSEMBLY LANGUAGE PROGRAM
	TO SIMULATE AN ARRAY-STRUCTURED COMPUTER . . . 34
2.1	Introduction 35
2.2	Assembly Language Instruction Set 36
2.3	Simulation Program Structure 42
2.4	Memory Storage and Retrieval Subroutines . . . 53
2.5	Arithmetic Subroutines 53
2.5.1	Addition-Subtraction Subroutine . . . 57
2.5.2	Multiplication Subroutine 57
2.5.3	Division Subroutine 71
2.6	Normalization Subroutine 78
2.7	Timekeeping Subroutine 81
2.8	Output Data Structure 82
2.8.1	Assembly Language Instructions 82
2.8.2	Special Instructions 85
2.9	Implementation of the Kalman Filter Algorithm in PAL-1 86
2.10	Results Derived from the Simulation Program . . 90
Chapter 3	DESIGN OF AN ARRAY-STRUCTURED COMPUTER . . . 95
3.1	Introduction 96
3.2	Computer System Functional Organization . . . 97
3.2.1	Introduction 97

	Page
3.2.2 Global Control Unit Functions	97
3.2.3 Exterior Unit Functions	99
3.2.4 X and Y Command Block Functions	103
3.2.5 Processing Array Functions and Organizations	103
3.3 Data Representation and Logic Convention	106
3.3.1 Data Representation	106
3.3.2 Data Normalization	108
3.3.3 Logic Convention	109
3.4 Global Control Unit	109
3.5 Exterior Processing Units	115
3.5.1 Input-Output Unit	115
3.5.2 Data Transfer Microprocessing Unit	117
3.5.3 Dynamic Microprocessing Unit	122
3.6 X and Y Command Blocks	126
3.7 Mathematical Operation Block	131
3.7.1 Introduction	131
3.7.2 MOB Command Generation Module	137
3.7.2.1 Master Control and Sequential Router Section	139
3.7.2.2 Microinstruction Generation Sections	142
3.7.2.2.1 Data Source Selector and Router Generation Section	142

	Page
3.7.2.2.2 Parallel Adder-Subtractor Generation Section	144
3.7.2.2.3 Parallel Carry-Save Multiplier Generation Section	147
3.7.2.2.4 Normalizing and Data Link Generation Section	149
3.7.3 Memory Module	152
3.7.4 Data Source Selector and Router Module	155
3.7.5 Parallel Adder-Subtractor Module	157
3.7.6 Parallel Carry-Save Multiplier Module	161
3.7.7 Normalizing and Data Link Module	166
3.8 Data Transfer Block	170
3.8.1 Introduction	170
3.8.2 Data Transfer Block Organization	172
3.8.3 Data Transfer Operation Bussing Structures	177
3.9 Adaptation of the Array-Structured Computer to Fault-Tolerant Operation	184
3.9.1 Introduction	184
3.9.2 Fault-Tolerant Operation Hardware	185
3.9.2.1 Introduction	185
3.9.2.2 Fault Detection Hardware	186
3.9.2.3 Fault Correction Hardware	188

	Page
3.9.3 Fault-Tolerant Operation Instruction Set	191
3.9.4 Fault-Tolerant Computer Structures	192
3.9.4.1 Introduction	192
3.9.4.2 Software-Oriented Structure	193
3.9.4.3 Triple-Redundant Structure	195
3.9.4.4 Multiple-Processing Array Structure	197
3.10 Implementation of a Single Processing Array Cell	200
3.10.1 Introduction	200
3.10.2 Implementation of a Single Processing Cell	201
Chapter 4 SUMMARY AND RECOMMENDATIONS	208
4.1 Introduction	209
4.2 Summary	209
4.2.1 Parallel Assembly Language Program	209
4.2.2 Array-Structured Cellular Computer	211
4.2.3 Fault-Tolerant Computer Organizations	214
4.3 Recommendations for Future Research	215
Appendix A The Kalman Filter Algorithm	218
Appendix B Pertinent Matrix Algorithms	227
Appendix C Listing of the Parallel Assembly Language Simulation Program	239
Appendix D Listing of the Linearized Kalman Filter Algorithm in PAL-1	303
Bibliography	308

LIST OF FIGURES

Figure		Page
1.1	The Unger Spatially-Oriented Computer	8
1.2	The SOLOMON Computer	14
1.3	The ILLIAC IV Computer	17
1.4	The Cannon Cellular Computer	23
2.1	Memory Doubleword Format	43
2.2	Generalized Flow Chart for PAL-1	44
2.3	Instruction Set Format	45
2.4	Flow Chart for Instruction Decomposition	46
2.5	Flow Chart for Storage of Doubleword in MEMORY	54
2.6	Flow Chart for Generating Element Address	55
2.7	Flow Chart for Retrieving Doubleword from MEMORY	56
2.8	Flow Chart for Subroutine ALNADD	58
2.9	Flow Chart for Subroutine MULTIPLY	64
2.10	Flow Chart for Subroutine RECIP	72
2.11	Flow Chart for Subroutine NORMALIZE	79
2.12	Flow Chart for Subroutine NORMTIME	83
3.1	Array Structured Computer Organization	98
3.2	Global Control and Exterior Units	100
3.3	Data Buss Structure for a 4 x 4 Processing Array	105

Figure		Page
3.4	Flow Chart for the Data Transfer Cycle	107
3.5	Floating Point Data Format	108
3.6	Modular Organization of the GCU	113
3.7	Data Transfer Microprocessing Unit	118
3.8	Data Transfer Macroinstruction Format	119
3.9	Dynamic Microprocessing Unit	123
3.10	Mathematical Operation Macroinstruction Format	124
3.11	Control Lines for a 4 x 4 Processing Array	127
3.12	X Command Block for a 4 x 4 Processing Array	128
3.13	Switching Module for the X Command Block	129
3.14	Y Command Block for a 4 x 4 Processing Array	132
3.15	Mathematical Operation Block	134
3.16	MOB Macroinstruction Format	135
3.17	MOB Command Generation Module	138
3.18	Master Control and Sequential Router Sections	140
3.19	Microinstruction Generation Section for the Data Source Selector and Router Module	143
3.20	Microinstruction Generation Section for the Parallel Adder-Subtractor Module	145
3.21	Microinstruction Generation Section for the Parallel Carry-Save Multiplier Module	148
3.22	Microinstruction Generation Section for the Normalizing and Data Link Module	150

Figure		Page
3.23	Memory Storage Cell	153
3.24	Memory Module	154
3.25	Data Source Selector and Router Module	156
3.26	Parallel Floating Point Adder-Subtractor Module	158
3.27	Parallel Floating Point Carry-Save Multiplier Module	162
3.28	Binary Multiplication Process	164
3.29	Parallel Carry-Save Multiplier	165
3.30	Normalizing and Data Link Module	167
3.31	Normalizing Section	169
3.32	Data Transfer Block	173
3.33	32 Bit Parallel Shift Register	175
3.34	32 Bit Parallel Comparator	177
3.35	Data Bussing Structure for a Rotate Right Operation	178
3.36	Data Bussing Structure for a Rotate Up Operation	180
3.37	Data Bussing Structure for the Transpose Operation	182
3.38	Flow Chart for the Array Comparison Cycle	187
3.39	Restructuring of a 4 x 4 Processing Array	189
3.40	Data Transfer Macroinstruction Modification Process	190
3.41	Software Oriented, Fault-Tolerant Computer Organization	194

Figure		Page
3.42	Triple-Redundant, Fault-Tolerant Computer Organization	196
3.43	Multiple-Processing Array, Fault-Tolerant Computer Organization	199
A-1	Kalman Filter Problem Block Diagram	222

LIST OF TABLES

Table		Page
2.1	Parallel Assembly Language Instruction Set . . .	38
2.2	PAL-1 Output Format	84
2.3	Special Instruction Output Format	85
2.4	Matrix Allocation for the Kalman Filter Example .	86
2.5	State Vector Estimate Comparison	89
3.1	Selected Set of Cellular Logic Units	111
3.2	X Command Block Data Source Selection Code . . .	130
3.3	MOB Macroinstruction Bit Designation	136
3.4	Sequence of Microinstruction Generation	141
3.5	Microinstructions for the Parallel Adder-Subtractor Module	146
3.6	Microinstructions for the Normalizing and Data Link Module	149
3.7	Truth Table for a Memory Storage Cell	152
3.8	Truth Table for the Data Source Selector and Router	155
3.9	Normalizer and Data Source Selection Code . . .	168
3.10	DTB Microinstruction Bit Designation	174
3.11	DTB Microinstructions for a Rotate Right Operation .	179
3.12	DTB Microinstructions for a Rotate Up Operation .	179
3.13	DTB Microinstructions for the Transpose Operation	183

Table		Page
3.14	Special Instruction Set for the Fault-Tolerant Operation	191
3.15	Special Instruction Set Example	192
3.16	Gates Required for the Selected Cellular Logic Units	203
3.17	Gates Required in the MOB	205
3.18	Gates Required in the DTB	206

ABSTRACT

The subject of this thesis is the design of a cellular computer specially organized to efficiently implement matrix algorithms, and capable of fault-tolerant operation.

The contents of the thesis are summarized as follows:

First, a review of pertinent literature in the areas of cellular logic design, array-structured computers, fault-tolerant computing systems, and large scale integration of digital circuits is presented. Second, an assembly language instruction set and an array-structured computer simulation program (PAL-1) are described including as an example a Kalman filter algorithm implementation. Third, the computer system organization and logical design of an array-structured computer are described including a discussion of the LSI implementation of a processing cell. Fourth, an adaptation of the array-structured computer design to fault-tolerant operation is presented. Three fault-tolerant computer organizations are described which use a special instruction set allowing fault detection, fault correction, and program rollback operations. Fifth, recommendations are made for further research in the area of array-structured computers capable of fault-tolerant operation.

CHAPTER 1

INTRODUCTION AND REVIEW
OF PERTINENT LITERATURE

1.1 Introduction

The evolution of computer systems has resulted in significant changes in engineering methods and applications. This evolution began with the development of large general-purpose machines which were capable of many diverse operations. These machines tended to be physically large, expensive, relatively slow, and difficult to program. As computer design technology evolved, these machines became faster, more efficient, easier to program, but remained large and costly. It became apparent that definite applications existed for smaller computing units capable of fast operation. Hence, the special purpose computer was developed, which has a small physical size, low cost, fast execution time, and a small instruction repertoire. The disadvantage of this type of machine was the large initial cost and limited application of a given machine design. To implement another set of algorithms might require an entire system redesign. A classic example of such a limited application machine is the digital differential analyzer (DDA).

Neither of the above approaches appears to be suitable for combinations of on-line applications such as filtering, vehicle navigation, and signal processing. To effectively meet the computational requirements of these and similar algorithms, a third computer design philosophy is evolving which utilizes the inherent structure of a given class of operations to produce a computing system design.

Hence, with a computer so structured, classes of operations can be efficiently processed as in a special-purpose machine, and yet the general-purpose nature can be retained. The specially-organized computer resulting from this research effort is a product of this third computer design philosophy.

An important class of problems whose inherent structure can be used to form the design of a specially-organized computing system is that class which deals with matrix and vector operations. This type of problem suggests a machine structure which is array oriented, and capable of parallel operation. Such an organization would strongly suggest a processing array composed of identical processing cells with appropriate interconnection structure. Examples of this class of problems are Kalman filtering (1), vehicle navigation, phased-array radar control computations, and sonar receiving array data processing.

The design and hardware implementation of a specially-organized computing system will be strongly influenced by the recent advances in solid state technology. The advent of large scale integrated circuit technology has made possible the fabrication of complex digital processors on a single monolithic substrate. Hence, consideration must be given to the impact of this and related technologies on the design of computing systems. In the next section a survey of microcellular research as it pertains to the organization and physical embodiment of a specially-organized computer is presented.

1.2 Survey of Microcellular Work

As the trend in the field of digital circuit implementation shifted to the use of integrated circuit technology, it became apparent that new bodies of digital switching theory must be developed. One such new body of theory undertook design of complex digital circuits in the form of identical cells placed together and connected through a regular, repetitive interconnective structure to form what is called a cellular array. Each of these identical cells consisted of an integrated circuit capable of some specified logical operation. Since logical operations appear with various levels of complexity, it is clear that the corresponding cellular arrays can be divided into a number of categories according to their cell complexity. The category of cellular array to be surveyed in this section will be the low complexity set referred to as microcellular arrays. A microcellular array consists of rather simple cells, with each cell containing no more than a few gates.

In the study of cellular arrays, it is apparent that numerous methods exist for imbedding logical functions into microcellular arrays. Approaches found in reviewed literature are of two general types: fixed cell function units, and variable cell function units. The fixed cell function approach considers cell parameters used only in the modification of the cell interconnection structure, whereas the variable

cell function approach considers cell parameters of both the function and interconnection structure.

One of the earliest efforts in the fixed cell function units was the eight neighbor array proposed in 1961 by Brooking (2) at the AFCRL to allow implementation of any arbitrary switching function. Each cell in this array is an eight input NOR gate; the NOR operation alone forming a complete set of logical primitives. Associated with each cell are eight parameters which provide an electrical disconnect for any subset of the eight nearest neighbor inputs. Several studies were made at AFCRL to implement variations of the eight neighbor arrays.

Combinational logic design methods for eight neighbor fixed cell function (NAND) cellular arrays were developed in 1963 by Spandorfer and Murphy (3) at the UNIVAC Engineering Center. Since the NAND operation forms a complete set, this type of cellular array will also allow implementation of any arbitrary switching function. This initial NAND cellular array consisted of imposing horizontal and vertical interconnection structures called wiggle busses on an eight neighbor array. This work evolved into a diamond-structured cellular array, resulting in more efficient array utilization. Approaches were also developed at UNIVAC (3) whereby faulty cells could be avoided by the introduction of spare rows and columns into the rectangular cellular arrays.

Another type of fixed cell function array that has been considered by several workers utilizes as each cell function a majority or minority element, usually of three inputs. Such majority gates have been considered by Minnick and Short (4), Canaday (5), and by Amarel, Cooke and Winder (6). In 1964, Miyata developed several methods for synthesizing arbitrary combinational functions in terms of arrays of three input minority or majority cells (7). Canaday has also developed a class of canonical synthesis methods that are based on various ways for decomposing switching functions to accomplish the same result.

The use of variable cell function cells will result in a more flexible cellular array. An early effort to utilize variable function cells was proposed by Gluckman at AFCL (8), in which each cell was selected to be an OR gate, NOR gate, or a crossover.

In 1962, Maitra (9) considered one-dimensional arrays in which each cell can be any one of the 16 functions of two variables. This cellular array is known as the Maitra cascade, general function cascade, or as a tributary network. Several cellular arrays have been based on the Maitra cascade, the first of which appears to be the cutpoint array (10) developed by Minnick in 1964. The array consists of a number of vertical Maitra cascades, interconnected by common horizontal busses. Each cell in the cutpoint array can produce any of

eight combinational switching functions or an R-S flip flop. Continuing work by Minnick (11) resulted in the cobweb array in 1965. In this cellular array, cell parameters are used both for choosing among the cell functions, and for changing the interconnection structures. Logical design and fault avoidance algorithms were also developed for the cutpoint and cobweb cellular arrays.

The detailed development of microcellular work is presented by Minnick (12).

As concluding remarks, the survey of microcellular work has yielded two important design considerations. The first is that in designing a cellular processing array, as much of the structure of the problem as possible should be designed into the array interconnection structure. The second consideration is that to implement a processing block in cellular form, a select set of functional cells should be used. Hence, any logical function must be realizable with this canonical set. These concepts are applied to the logical design of the specially-organized computer. In the next section a survey of array-structured computer designs is presented with emphasis placed on features pertinent to the processing of the selected class of problems.

1.3 Survey of Array-Organized Computers

Review of available literature on specially-structured computing systems reveals a number of array-organized computers. Several pertinent computer organizations are discussed in the following sections.

1.3.1 The Unger Spatially-Oriented Computer

One of the first examples of a specially-organized computer was proposed by Unger (13) in 1958. This machine is a stored program computer which can handle spatial problems by directly operating on information in planar form without scanning or format conversion. Hence, this machine is specially organized to efficiently handle pattern detection problems, such as detection of a lower left corner of a binary pattern.

The structure of the proposed machine is shown in Figure 1.1. It consists of a master control unit and a rectangular array of modules. Each module communicates with its four nearest neighbors and receives command signals from the master control. The master control consists of a random access memory for storing instructions, a clock, and decoding circuits, and issues commands in parallel to all modules; it cannot address modules individually. The machine is programmed with a set of 14 assembly language instructions.

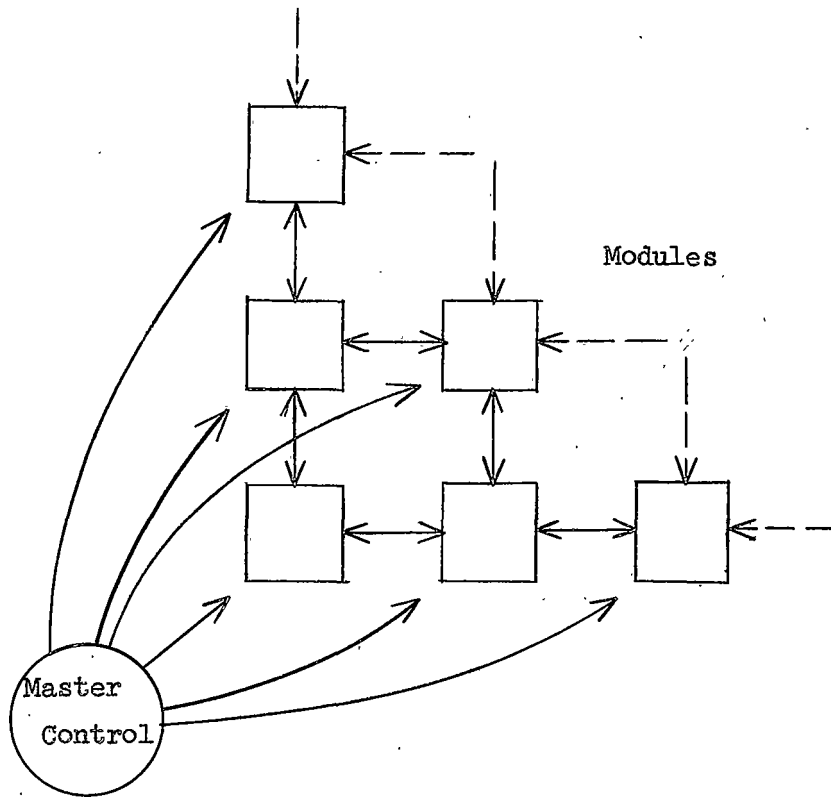


Figure 1.1. The Unger Spatially-Oriented Computer

Each module in the processing array consists of a one bit accumulator, a small amount of random access memory (six bits in one bit words), and some associated logic. Inputs to each module come from the master control and from the accumulators of the nearest neighbor modules.

A logical adder (OR gate) with an input from each module accumulator tells the master control if all of the accumulators contain zero, making possible a transfer on zero order. This instruction, analogous to the conditional transfer orders used in conventional computers, indicates to the master control to skip to the instruction indicated by the transfer zero order. This is the only decision dependent command present in the machine.

It is in this machine organization that one of the first examples of a processing array controlled in parallel by a central control unit is found.

1.3.2 The Holland Computer

A computer organization was described by Holland (14), establishing system control at the local level in the processing array. This organization is in direct contrast to the central control concept suggested by Unger.

The purpose of this computer organization was to provide a basis for investigation into computability and the theory of automata. It consists of a two-dimensional array of identical modules, each module containing a storage register, routing logic, and auxiliary registers. At any given time a module will be active or inactive. If the module is active, it treats the contents of its storage register as an instruction and proceeds to execute the instruction. After a module has

executed its instruction it passes its active status on to its successor, which may be any of its four nearest neighbors in the array. With this computer structure, sequences of instructions are arranged spatially throughout the array of modules, with an arbitrary number of sequences being executed at any given time.

The operation cycle of this computer consists of three phases. In the first phase, module storage registers may be set to values provided by an external source. In the second phase, active modules determine the locations of their operands by causing paths to be logically enabled. The third phase causes the instructions in the storage registers of all active modules to be executed.

This computer appears to be difficult to program efficiently so that more than a small number of array modules are active at any given time. Also, a large amount of hardware is required to accomplish reasonable computing tasks.

One of the important results of this work is the development of an array of locally controlled identical processing modules. However, the utilization of such a processing array results in large hardware expense and a computer that is difficult to program.

1.3.3 The Comfort Computer

The concept of an array-structured computer utilizing local system control was further considered by Comfort (15). The result was a modified Holland computer structure having the form of a fixed size rectangular array of modules. Each module consists of two relatively independent sections; the memory and control section, and the communications section. At one side of the array of modules is a set of arithmetic units. These units do all the mathematical and logic computations. Since the machine contains no central control, each module executes its own instruction when it is enabled. The execution of a sequence of instructions causes the enabling and disabling of successive modules.

This computer was organized to have some important improvements to the Holland computer, which are noted below:

1. Programmability has been improved by several orders of magnitude.
2. Machine size has been reduced by a factor of five.
3. Hardware utilization has been improved by a factor of three.

A possible degradation in system performance may occur, since only one program sequence per arithmetic unit can be executed at one time.

1.3.4 The SOLOMON Computer

The SOLOMON (Simultaneous Operation Linked Ordinal Modular Network) computer is a parallel processing computer introduced by Slotnick, Borck, and McReynolds in 1962 and later revised in 1966 (16). This computer organization was the first attempt to satisfy a class of problems limited by current computing system capabilities at that time. This computer was specially organized to efficiently implement matrix or mesh equations. This class of problems is typified by linear system analysis, matrix calculations, and solutions to systems of ordinary and partial differential equations.

The computer, as shown in Figure 1.2, is composed of three major units. The first is the network control unit (NCU) which is a central control for the machine. The NCU can consist of a minimum of one arithmetic and control subunit, and can be expanded to multiple subunits. The second is the processing array composed of 32 x 32 processing elements (PE). The PE array is designed so that modules of 256 PE's including the associated PE memories can be added or removed from the basic system without design change. The third is the input-output unit (IOU) which consists of five modules of 32 data channels each (a channel being an input-output data link).

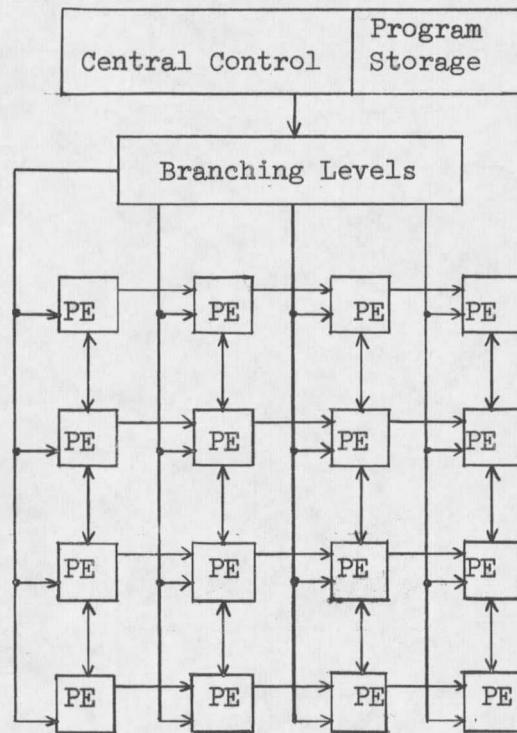


Figure 1.2. The SOLOMON Computer

All processing elements are identical, possessing complete arithmetic capabilities. Each processing element has associated with it two memory frames with 4096 bits of storage (expandable to 16,384 bits) per frame. Each processing element can perform serial-by-bit arithmetic and logic. Words in the processing elements may transfer data serially

to and from its four nearest neighbors in the processing array. An important result of this machine design is the utilization of identical processing cells under central control and capable of four nearest neighbor communication.

1.3.5 The Gonzalez Iterative Circuit Computer

A multilayer iterative circuit computer (ICC) was proposed by Gonzalez (17) in 1963, based on work done by Unger and Holland. This organization resulted in a machine capable of dealing with problems involving spatial relationships between variables. The computer organization consists of three stacked layers of processing elements. The three layers are identical, and are referred to as the program layer, the control layer, and the computing layer. The program layer stores data and instructions, the control layer decodes instructions, and the computing layer performs arithmetical, logical, and geometric operation. The programming is basically the same as the Holland machine with each instruction specifying the module containing the next instruction. The control and program layers can be working on the next two instructions, respectively.

All modules in the three planes of $m \times n$ modules each are identical, and communicate with each other by means of data and control lines running through them. The internal structure of these modules includes

an accumulator, a register, a decoder, and several switching matrices to connect the decoder to the data and control lines.

It is apparent that programming of this machine is difficult and that hardware is not efficiently utilized.

Some of the novel features of this machine organization which warrant design considerations are:

1. A path building method which retains the short time access characteristic of a common buss system, while permitting the simultaneous operation of several paths in the network.
2. A specialization in the functions of the individual layers, separating the flow of control signals from the flow of information.
3. Three-phase operation, with each phase active simultaneously on each layer, and operating on different instructions.

1.3.6 The ILLIAC IV Computer

A parallel array computer called the ILLIAC IV was introduced by Barnes, Brown, Kata, Kunk, Stokes, and Slotnick (18) in 1968. This machine is a direct descendant of the SOLOMON computer, and is directed towards efficient implementation of the matrix or mesh class of problems.

The organization of the ILLIAC IV computer is shown in Figure 1.3.

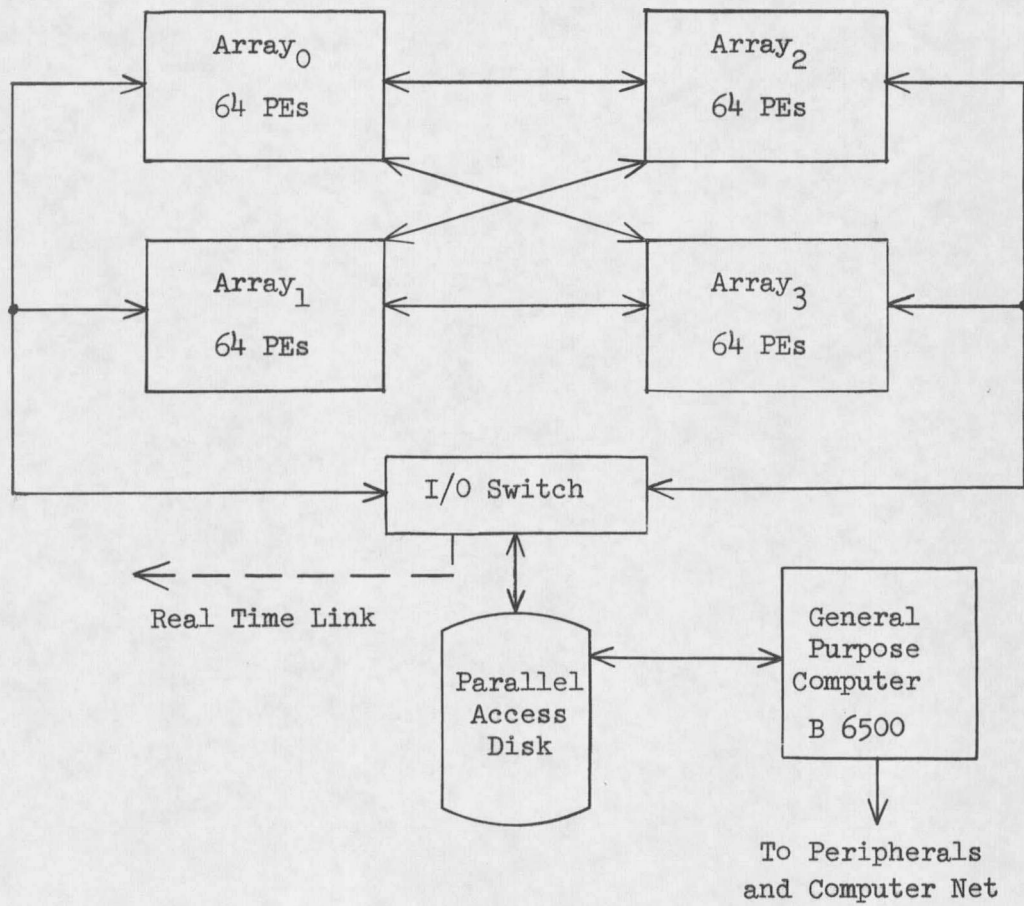


Figure 1.3. The ILLIAC IV Computer

The individual processing elements (PE's) are grouped into four arrays, each containing 64 elements and a control unit (CU). The CU decodes instructions and generates control signals for all processing elements in the array. The four arrays may be connected together under program control to permit multiprocessing or single processing operation. The systems program resides in a general-purpose computer, a Burroughs B6500, which supervises program loading, array configuration changes, and input-output operations internal to the ILLIAC IV system and to the external world. To provide backup memory for the ILLIAC IV arrays, a large parallel access disk system (10^6 bits, 10^9 bits per second access rate) is directly coupled to the arrays. There is also provision for real time data connections directly to the ILLIAC IV arrays.

The ILLIAC IV main structure consists of 256 processing elements arranged in four SOLOMON type arrays of 64 processors each. The individual processors have a 240 nanosecond ADD time and a 400 nanosecond MULTIPLY time for 64 bit operands. Each processor requires approximately 10^4 ECL gates and is provided with 2048 words of 240 nanosecond cycle time thin film memory.

The organization of the ILLIAC IV as a collection of identical units simplifies its maintenance problems. The processing elements, the memories and some parts of the power supplies are designed to be

pluggable and replaceable to reduce system down time. The remaining diagnostic problems are:

1. Location of the faulty subsystem.
2. Location of the faulty package in the subsystem.

Location of the faulty subsystem assumes the B6500 to be fault free, since this can be determined by using the standard B6500 maintenance routines. The B6500 tests the control unit which in turn tests all PE's. PEM's are tested through the disk channel. This capability for functional partitioning of the subsystems simplifies the repair of system faults.

In addition to the design considerations that are identical to the SOLOMON computer, the ILLIAC IV computer design has yielded information on LSI implementation of functional units, and the use of fault detecting software and hardware.

1.3.7 The MCB Computer

The modular computer breadboard (MCB) was proposed in 1969 by the NASA Electronic Research Center (19). This machine takes the form of a modular column structure which can be configured to operate as a number of parallel processors (one processor per column). Each column contains a number of modules, and can be restructured to form modified columns, using spare modules from different physical locations. Each segment or column has the ability to operate as a separate processor

operating on different algorithms, or separate parts of the same algorithm.

The computer is separated into four functional modules; a memory unit, control unit, arithmetic unit, and I/O unit. This machine can be software structured to perform a large number of computing tasks. The configuration is accomplished by the configuration control unit (CCU), and the configuration control switches (CCS). An important configured structure possible with this machine is the triple-modular-redundant (TMR) structure, allowing extremely reliable operation. The concept of LSI implementation was explored in detail in this machine design and forms the base for construction of the modular computer hardware.

The important design considerations resulting from this machine organization are the LSI implementations of functional blocks, and the use of a software structurable processing array.

1.3.8 The Berkeley Array Processor

The Berkeley Array Processor (BAP) was presented by Dere and Sakrison (20) in 1970. This processor is a general-purpose computer designed to efficiently perform the operations of correlation, convolution, recursive filtering, matrix multiplication, and the Cooley-Tukey algorithm.

The BAP is an input-output device interfaced with an IBM 1800 computer. Program and data are resident in the IBM 1800, and are

accessed under control of the BAP. The data is structured as an array and is accessed by the BAP 104 word shift register IC memory which provides a pseudo two channel communication link to the stored array data. The BAP consists of the shift register memory, an array indexing unit, an arithmetic unit, an accumulator, and associated control logic. The computer uses a 16 bit word and operates with a 140 nanosecond clock period.

Since the data structure is an array, the instruction set consists of instructions that set up pointers to arrays, operate on these arrays, that ascertain the completion of an operation, and specify the proper action to take upon completion. The instruction set consists of 17 instructions and is capable of working with complex variables.

This array processor suggests a different approach to an array structure, where a data array is formed in memory and operated upon by a central processor which is specially organized to operate on such an array. This machine is an example of how an array processor can be attached as an appendage to a general-purpose computer to implement array-oriented algorithms. The BAP was designed using known techniques; no significant logic design innovations were developed.

1.4 Survey of the Cannon Cellular Computer

An array-organized machine specially structured to efficiently perform matrix operations was proposed by Cannon (21) in 1969. The intended utilization of this machine was the implementation of the Kalman filter algorithm discussed in Appendix A. This machine design forms the base for the second-generation, array-structured computer design presented in this research effort. Certain conventions developed in Cannon's work are used and referenced in this machine design to allow utilization of pertinent research results, and continuity between this and the preceding design effort.

The organization of the Cannon cellular computer is shown in Figure 1.4. The structure of this machine consists of a two-dimensional square array of processing cells and a global control unit; this structure being similar to the SOLOMON and its descendant, the ILLIAC IV. The processing cells are controlled by the global control unit which supervises the operation of all cells. The global control unit has arithmetic capability and significant storage capability in addition to array control circuitry. This unit decodes instructions and issues the necessary array commands to accomplish the given instruction.

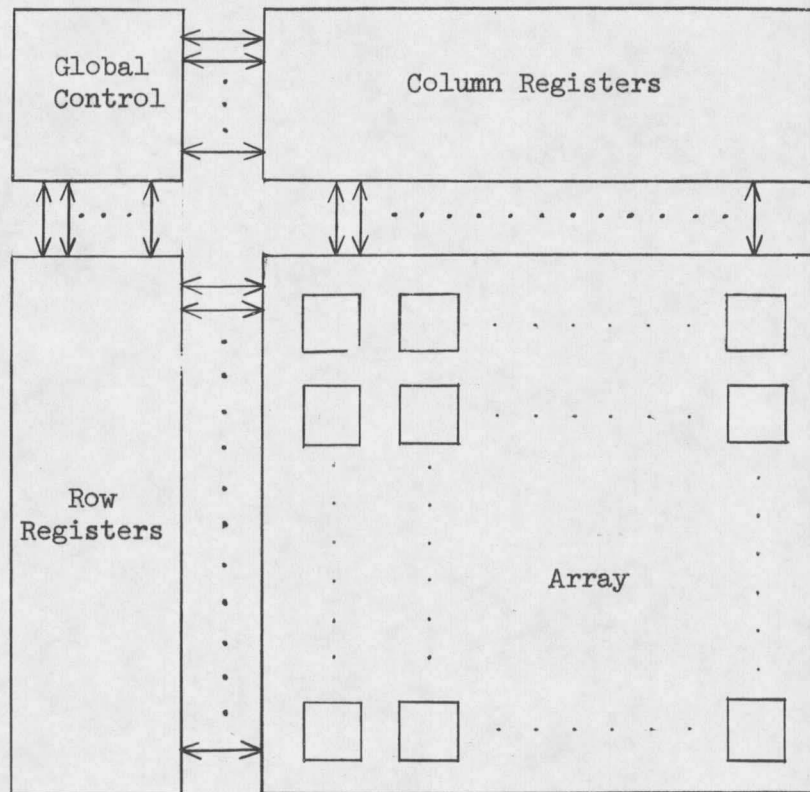


Figure 1.4. The Cannon Cellular Computer

The processing cells are identical, and are composed of sixteen 32 bit words of memory, a floating point adder-subtractor, a floating point multiplier, and logic for the routing of data to and from the processing cell. All cells in the processing array receive the same set of control signals and execute the same instruction. The matrix

data is stored in spatial locations as the (i,j) element of the array being stored in the (i,j) cell location. The cells in the processing array are interconnected uniformly with interconnection structures composed of singular horizontal, vertical, and diagonal data lines. This form of interconnecting structure was selected to allow special data transfer operations to be implemented. These special operations are broadcasting, rotating, skewing, and transposing of matrix data; necessary special matrix operations used in implementing the matrix algorithms selected by Cannon and presented in Appendix B. With these special operations, matrices can be multiplied by method 1, or 2, and matrix inversion can be accomplished by the Gauss-Jordan or escalator inversion schemes.

This specially-structured computer design yields significant improvements in speed over conventionally-organized machines due to its parallel processing capability. However, significant hardware requirements are necessary, since in effect the processing array is composed of N^2 small computing units (N is the dimension of the array).

There are a number of areas requiring improvements in the machine proposed by Cannon. These are listed below:

1. Control of the processing array is not fully defined at either the local or global level.
2. Data transfer in the processing array is very time consuming.
3. Input and output of array data is time consuming.
4. An assembly language has not been developed for the parallel processing machine.
5. Simulation of the computing system has been conducted on the cell level only. A simulation of the entire machine has not been performed.
6. Multiplication of floating point numbers is done serially, which is time consuming.
7. The processing array has no provision for bypassing unused cells.
8. No consideration has been given to machine fault-tolerance.
9. Very little information has been presented on the implementation of the machine in LSI.

These considerations suggest a second-generation design effort on the work begun by Cannon.

1.5 Current Trends in Fault-Tolerant Computers

The continuing increases in the speed and complexity of computing systems will accelerate the demand for fault-tolerant operation. These demands will be reinforced with the critical nature of computing equipment in real time applications and the inability to perform maintenance on long term missions. The most immediate solution in such critical applications has been the replication of the entire system (22), frequently backed up by the transfer of control to a human operator, or to a separate less precise backup system. The replication at the system level becomes extremely costly when larger and faster systems are considered. Furthermore, occurrence of independent faults in two or more backup systems is more probable as the systems become more complex.

In the early stages of development, attention has been directed toward massive redundancy (replication of individual components) (23). The use of component redundancy has been limited by design difficulties and by new development in component technology. The change from discrete components to integrated circuits has largely invalidated the assumptions of independent component failures, (since failures in integrated circuits will quite likely involve more than a single component); without which the advantages of component redundancy are lost.

Considerable effort has been continuously directed toward practical use of massive triple-modular-redundance (TMR) (24) in which

logic signals are handled in three identical channels and faults are identified by vote-taking elements distributed through the system (25). Studies have considered optimal voter placement and analyzed the gain in reliability of mean life of a TMR system (26).

Another form of fault-tolerant computing system utilizes a selective redundancy form. This form will be either concurrent diagnosis (27) or periodic diagnosis (28). Compared to massive redundancy, the selective form requires several additional features; system ability to tolerate interruptions for repair and to execute a rollback reinstatement of a valid former state of the machine for fault correction, sophisticated diagnosis methods, and protection for the system main memory. The most developed techniques of selective redundancy are fault detection by periodic diagnosis and the application of parity and similar error codes (29) to detect and correct errors in data transmission and storage. The periodic diagnosis techniques have progressed from exclusively software implementations to software-hardware combinations (30) and to studies of system design techniques which facilitate their self diagnosis (31). Reorganization of the computing unit structure upon fault detection has also been discussed in recent publications (32); however, much work remains to be done in order to arrive at complete system designs and to derive measures of expected effectiveness.

In summary, it is noted that the rapid development of integrated circuit technology is causing a shift of emphasis from massive redundancy to selective redundancy in which functional units of a system are replaced as single elements. This emphasis may result in additional circuitry being designed into the integrated circuit chips to allow reliable operation in the presence of local faults.

1.6 Current Trends in Large Scale Integrated Circuits

Large scale integrated circuitry has no precise definition, but it is usually reserved for integrated circuits that comprise 100 or more gates (or individual functions), laid down with densities of 50,000 to 100,000 components per square inch. The implementation of a given function is easiest done when the function can be embodied in some modular or cellular manner. Hence, the use of cellular array design in a computer system might be implementable utilizing large scale integration.

Depending on the size and complexity of the proposed digital circuits, hybrid integrated circuit technology may be a better choice for implementation. This unique approach to high density micro-electronic packaging combines on a common plane a number of LSI functional circuits. This type of implementation allows flexibility in design of subunits and the maintenance of the functional package.

Two forms of integrated circuit technology bear mentioning: bipolar and metal oxide silicon (MOS). Bipolar technology will provide speed but at the expense of power dissipation and component density. MOS technology tends to be a factor of 10 slower but a factor of 10 higher in component density; typical bipolar circuits operate with clock period of 10-20 nanoseconds; while typical MOS circuits use clock period of 150-300 nanoseconds. Hence, in selection of an integrated circuit fabrication technology, trade-offs of speed versus allowed circuit complexity must be considered. Currently, arrays of 200-1000 gates can be manufactured with 3000-5000 active components. Larger integrated circuits tend to be the MOS type, or a hybrid of MOS and bipolar. Although speculative, it is not unreasonable to suggest that arrays of 1000-5000 gates with 5000-20,000 active components will be available within five years.

Problems associated with large scale integration are numerous, such as circuit layout and masking, fault detection, fault correction, pin number and location, and heat dissipation. The most formidable of these problems is circuit layout and masking, which quickly passes from human hands to the realm of computer-aided-design (CAD).

Utilization of computer aided design would proceed in the following steps. Initially, the circuit designer would prepare a logic diagram and specify the individual circuit elements required.

This information would be supplied to the computer. A program would then examine the interconnection required and attempt to locate elements to give the simplest connection pattern. The program will test crossovers and bends, suggesting where lines can be laid under resistors to avoid crossovers. After the interconnection structure has been determined, the program will examine the shape and size of various elements and fit them into the smallest possible substrate area. At this point the mask preparation and fabrication process would begin.

1.7 Description of Research Effort

The result of this research effort is the design of an array-structured computer capable of fault-tolerant operation. This computer is a general-purpose machine specially organized to efficiently implement matrix algorithms. This research effort proposes a computing system that should be implementable within five years.

The formulation of this research evolved from the surveys of pertinent literature presented in preceding sections of this chapter. The organization of the designed computer has its origin in the SOLOMON computer and its descendant, the ILLIAC IV. This basic structure was adopted by Cannon, resulting in a specially-organized computing system that efficiently implemented the Kalman filter algorithm. The Unger computer suggests a specially-structured computer utilizing

central processing array control, while the Holland computer suggests the additional use of local processing cell control. The approach adopted in this design effort is a combination of central and local control of the processing array yielding operational flexibility at the expense of additional hardware.

The computer organization and logical design presented in this research effort has a structure similar to the SOLOMON and ILLIAC IV computers resulting in a second generation design of the computing system proposed by Cannon. However, important differences exist between this computer design and the Cannon computer in the following areas: processing cell organization and logical design; speed of operation; speed of data transfer within the processing array; processing cell interconnection structure; development of a computer assembly language instruction set; complete array-structured computer simulation (PAL-1); the introduction of additional control at the local level; and the introduction of parallel arithmetic operations.

A current trend in computer development is the designing of computing systems that are fault tolerant. The literature surveyed suggests many methods that allow such systems to function in the presence of faults. A combination of software and hardware is used in this design effort to form three computer organizations that are

capable of fault-tolerant operation; software-oriented, triple-redundant, and multiple-processing array organizations.

The survey of microcellular literature suggests the use of cellular array logic in this computer design. A selected set of nine cellular logic units is used in the design of this array-structured computer. Consideration is given in this research effort to LSI implementation of a single processing cell.

1.8 Organization of Remaining Chapters

Chapter 2 contains the description of the 58 assembly language instructions selected for the array-structured computer. Also included in this chapter is a description of the array-organized computer simulation (PAL-1), which uses as an example an implementation of the Kalman filter algorithm.

The design considerations derived from Chapters 1 and 2 are used to form the base of the array-structured computer design presented in Chapter 3. This chapter contains a description of the computer system organization and the logical design of this computer. In addition, this chapter contains an adaptation of the array-structured computer design to fault-tolerant operation in the form of three computing system organizations; software-oriented, triple-redundant, and multiple-array organizations. This chapter concludes with a discussion of LSI implementation of a single processing cell.

The concluding chapter consists of a summary of important results and recommendations for future research in the area of array-structured computer systems.

CHAPTER 2

DESIGN OF A PARALLEL ASSEMBLY LANGUAGE
PROGRAM TO SIMULATE AN ARRAY-STRUCTURED COMPUTER

2.1 Introduction

The design of this array-structured computer utilizes an approach different from more common methods. Due to advances in solid state technology, the hardware associated with a computing system has become diverse in operation, small in size, and lower in cost. These advances allow the computer designer to be equally concerned with the software associated with a proposed computer design. Hence, rather than designing a computing system hardware structure and then contriving a workable software ensemble, the design engineer can design the computer hardware to efficiently implement a selected set of assembly language instructions. This design approach places heavy emphasis on the software development of the computing system. In addition, this approach allows the computer designer to fully simulate the operation of the proposed machine and conduct applications programming, before any hardware has been constructed. This design approach is referred to as computer architecture, and involves the simultaneous development of software and hardware for a proposed computing system. The subject of this chapter is the software development and simulation of an array-structured computer.

A computer simulation program has been developed which completely simulates the operation of an array-structured computer. This program is based upon a selected set of 58 assembly language instructions appropriate to the operation of such a machine. To provide

information on the number of parallel processing periods required to implement the assembly language instructions, timing subroutines are included in the program.

To analyze operation of an array-structured computer on a typical problem of the class under consideration, the Kalman Filter algorithm has been implemented with the assembly language instruction set. The results of this implementation are compared with the results of a FORTRAN IV implementation of the same algorithm.

2.2 Assembly Language Instruction Set

To allow efficient implementation of matrix-oriented algorithms, attention has been given to the selection of an effective assembly language instruction set. The occurrence of special instructions composed of sets of these instructions necessitated an investigation of methods and resulting operational requirements for implementing such instructions.

As a result of the above considerations, a selected set of 58 assembly language instructions forms the language for an array-structured computer. This selected set is listed in Table 2.1, and is composed of two types of instructions--simulation and utility. Note that a few of the simulation instructions are special instructions. The simulation instructions relate to array-oriented operations, whereas utility instructions pertain to input-output and book-keeping operations. Associated with the simulated machine is an

external arithmetic processor (called the buffer) to facilitate single element operations. Most of the selected simulation instructions are familiar operations occurring in matrix or associated scalar manipulations. However, a few instructions have been added as a result of work done by Cannon and Rudberg which is presented in Appendix B. These added instructions implement the operations of broadcasting, rotating, and skewing of matrix data, allowing matrix multiplication by algorithms 1 and 2, and matrix inversion by the Gauss-Jordan algorithm.

The simulation program presented in the next section is structured to simulate the array-structured computer implementation of this assembly language instruction set.

Table 2.1 Parallel Assembly Language Instruction Set.

<u>Instruction</u>	<u>Type</u>	<u>Purpose</u>
ABSE	Simulation	Place absolute value of element B,C of A in B,C of A.
ADBE	Simulation	Add buffer to element B,C of A.
ADDA	Simulation	Add elements of array.
ADEB	Simulation	Add B,C of A to buffer.
APBR	Simulation (Special)	Add to row B buffer times elements of row C of A.
BCAC	Simulation	Broadcast column C of B over A.
BCAR	Simulation	Broadcast row C of B over A.
BCBC	Simulation	Broadcast buffer over column B of A.
BCBR	Simulation	Broadcast buffer over row B of A.
BDIZ	Utility	Branch on decrementing INDEX not zero.
CMFA	simulation	Complement array, (negative of each element).
CMFB	Simulation	Complement the buffer.
EXCC	Simulation	Interchange columns B and C of A.
EXCR	Simulation	Interchange rows B and C of A.
INPA	Utility	Read from input cards B by C matrix into A.
INPB	Utility	Read from input cards into buffer.
INPC	Utility	Read from input cards a vector into column B of A.

Table 2.1 (continued)

<u>Instruction</u>	<u>Type</u>	<u>Purpose</u>
INPI	Utility	Read from input cards INDEX
INVA	Simulation (Special)	Invert the $B \times C$ matrix A and place in A.
INVB	Simulation	Invert the buffer.
LDEB	Simulation	Load element B,C of A into buffer.
LDIB	Simulation	Load inverse of element B,C of A into buffer.
MLBC	Simulation	Multiply column B of A by the buffer.
MLBE	Simulation	Multiply the buffer by B,C of A.
MLEB	Simulation	Multiply element B,C of A by the buffer.
MLTA	Simulation	Multiplying corresponding elements of matrices B and C and place the result in A.
MULT	Simulation (Special)	Multiply matrices B and C and place the result in A.
MOVA	Simulation	Move matrix from B to A.
OUTA	Utility	Print out on line printer matrix A.
OUTB	Utility	Print out on line printer buffer.
OUTC	Utility	Print out on line printer column B of A.
OUTR	Utility	Print out on line printer row B of A.

Table 2.1 (continued)

<u>Instruction</u>	<u>Type</u>	<u>Purpose</u>
ROTD	Simulation	Rotate matrix A down B rows.
ROTL	Simulation	Rotate matrix A left B columns.
ROTR	Simulation	Rotate matrix A right B columns.
ROTU	Simulation	Rotate matrix A up B rows.
SETC	Utility	Read from input card the maximum number of columns to be encountered in this algorithm.
SETE	Utility	Store 1 in element B,C of A.
SETR	Utility	Read from input card the maximum number of rows to be encountered in this algorithm.
SKWD	Simulation	Skew matrix A down.
SKWL	Simulation	Skew matrix A left.
SKWR	Simulation	Skew matrix A right.
SKWU	Simulation	Skew matrix A up.
SNEB	Simulation	Find sign of element B,C of A and put in buffer.
SQRB	Simulation	Square the buffer.
STBE	Simulation	Store buffer in element B,C of A.
SUMC	Simulation	Sum columns of B into row C of A.
SUMR	Simulation	Sum rows of B into column C of A.

Table 2.1 (continued)

<u>Instruction</u>	<u>Type</u>	<u>Purpose</u>
TRAC	Simulation	Transfer Column C of B into A.
TRAR	Simulation	Transfer row C of B into A.
TRNC	Simulation	Transfer column C to column B of A.
TRPS	Simulation	Take the transpose of matrix A and place in matrix A.
ZERA	Simulation	Zero the array.
ZERC	Simulation	Zero column B of A.
ZERR	Simulation	Zero row B of A.
\$\$\$\$	Utility	End of instruction set flag.

2.3 Simulation Program Structure

The computer program simulating the operation of an array-structured computer is listed in Appendix C, and is hereafter referred to as PAL-1 (Parallel Assembly Language - 1). The simulated machine consists of a 16 x 16 array of processing cells, an arithmetic unit for each cell, and an external buffer. A clock period computing unit composed of incremental and total accumulated clock period counters and input-output units are included as utility functions. Each processing cell has the capacity for storage of sixteen 32 bit words, allowing matrix operations involving sixteen 16 x 16 matrices for the array-structured machine (expandable to any desirable size). Each cell storage location is denoted by a_{ij} , where i is the row index, j is the column index, and a is the matrix identifier.

Within the program structure, each memory location is considered to be a double word with the format shown in Figure 2.1. The binary fraction of the floating point data word is contained in the first word, and the exponent in the second word. Both words are right justified. Thus, the simulated processing array requires 4096 double words (8192 words) of memory, whose addresses are computed by

$$\text{MEMORY LOC } A(I,J) = \text{MEMORY} + (A + 16(16(I-1) + (J-1))) \quad (2.1)$$

The notation $A(I,J)$ has the usual meaning in matrix notation.

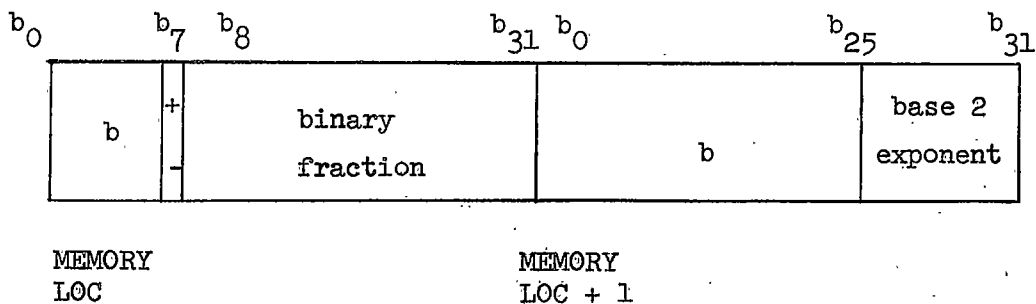


Figure 2.1. Memory Doubleword Format

The simulation program PAL-1 utilizes a FORTRAN IV mainline program, two SYMBOL subroutines (see Appendix C), and a FORTRAN IV subroutine. The FORTRAN IV routines are used for ease of data input and output, whereas the SYMBOL routines are used in the main manipulation portions of the program. The generalized flow chart for PAL-1 is shown in Figure 2.2, illustrating the routines used to implement the assembly language instructions. As shown in this chart, the list of assembly language instructions comprising the algorithm is read by the card reader and placed immediately on the rapid access disc (RAD). The reading process continues until an end of instruction flag (\$\$\$\$) is detected. Then, one by one, beginning with the first instruction read, the instructions are brought from the RAD to the main execution program for execution. The instruction set format used on the input cards is shown in Figure 2.3.

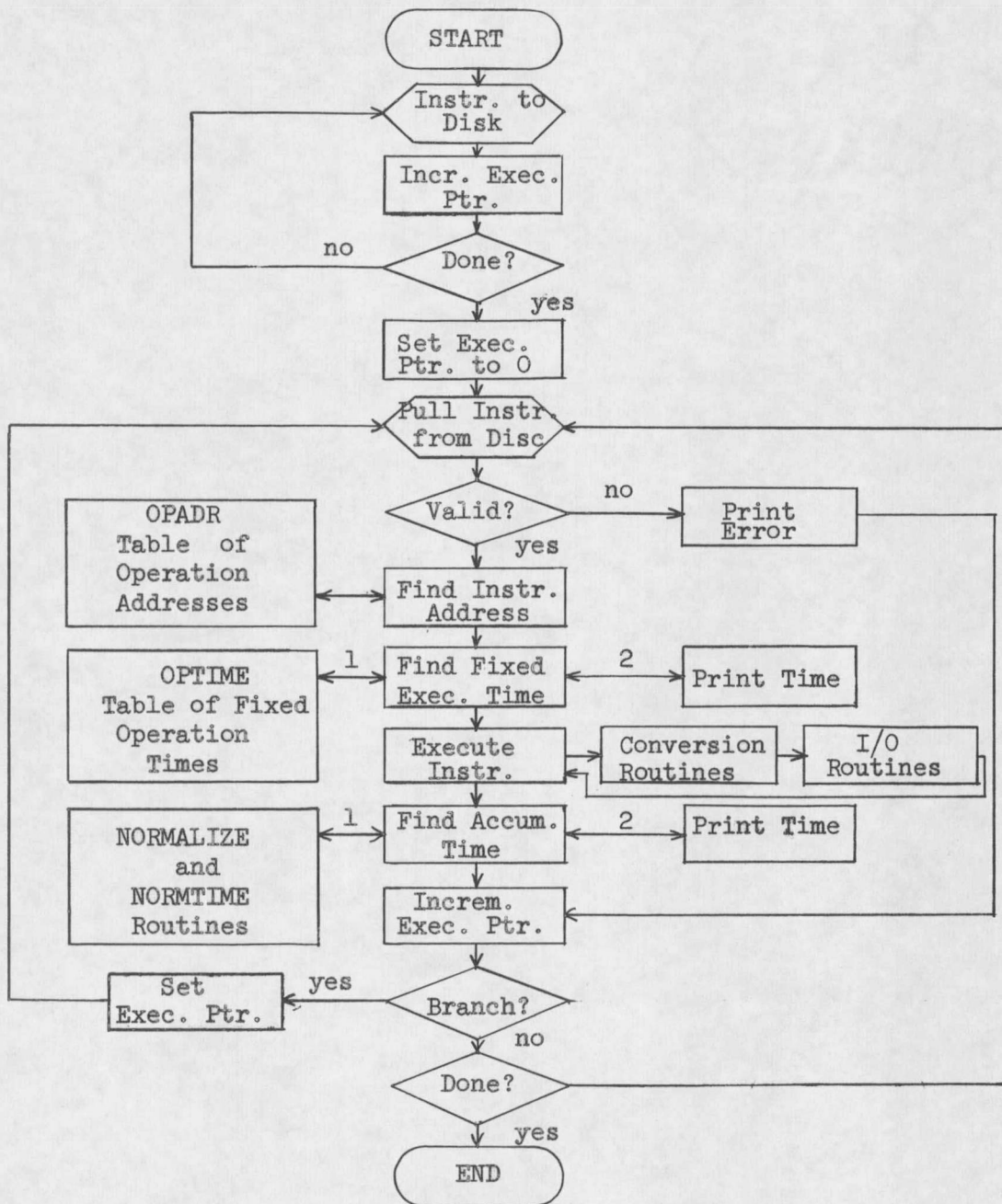


Figure 2.2. Generalized Flow Chart for PAL-1.

Note: Operations that are numbered occur in sequential order.

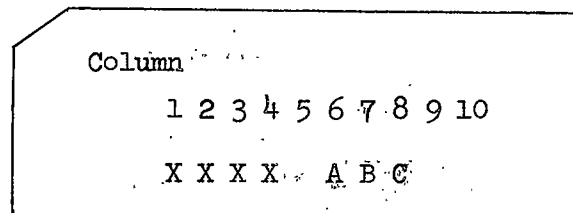


Figure 2.3 Instruction Set Format

XXXX is the instruction mnemonic and A,B,C are the necessary operands of the instructions in hexadecimal notation. The operands use a free field format with at least one or more blanks between the mnemonic and all operands. Once called from the disc, each instruction is checked for correct syntax, and then executed if syntax is verified. Decomposition and execution of each instruction is accomplished by the main execution routine shown in Figure 2.4. Whenever input or output data is required, the FORTRAN IV input-output routines are used. Since the array structured computer operates in base two, conversion routines are required to convert between Sigma 7 base 16 and array structured computer base two. The input data follows the end of instruction flag (\$\$\$\$), and uses a FORTRAN F13.5 format.

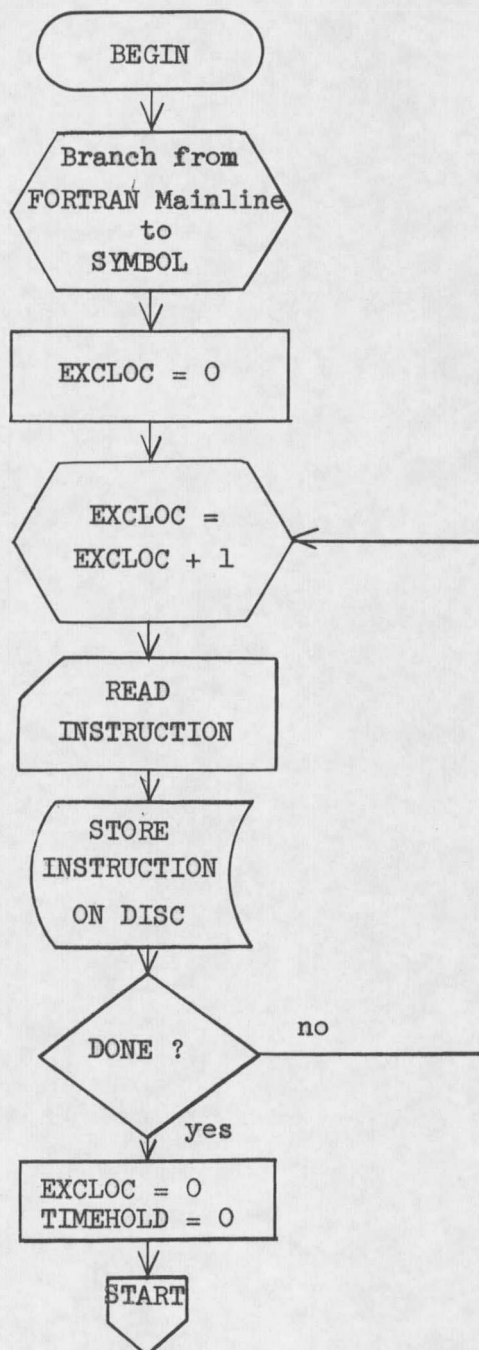


Figure 2.4 Flow Chart for Instruction Decomposition

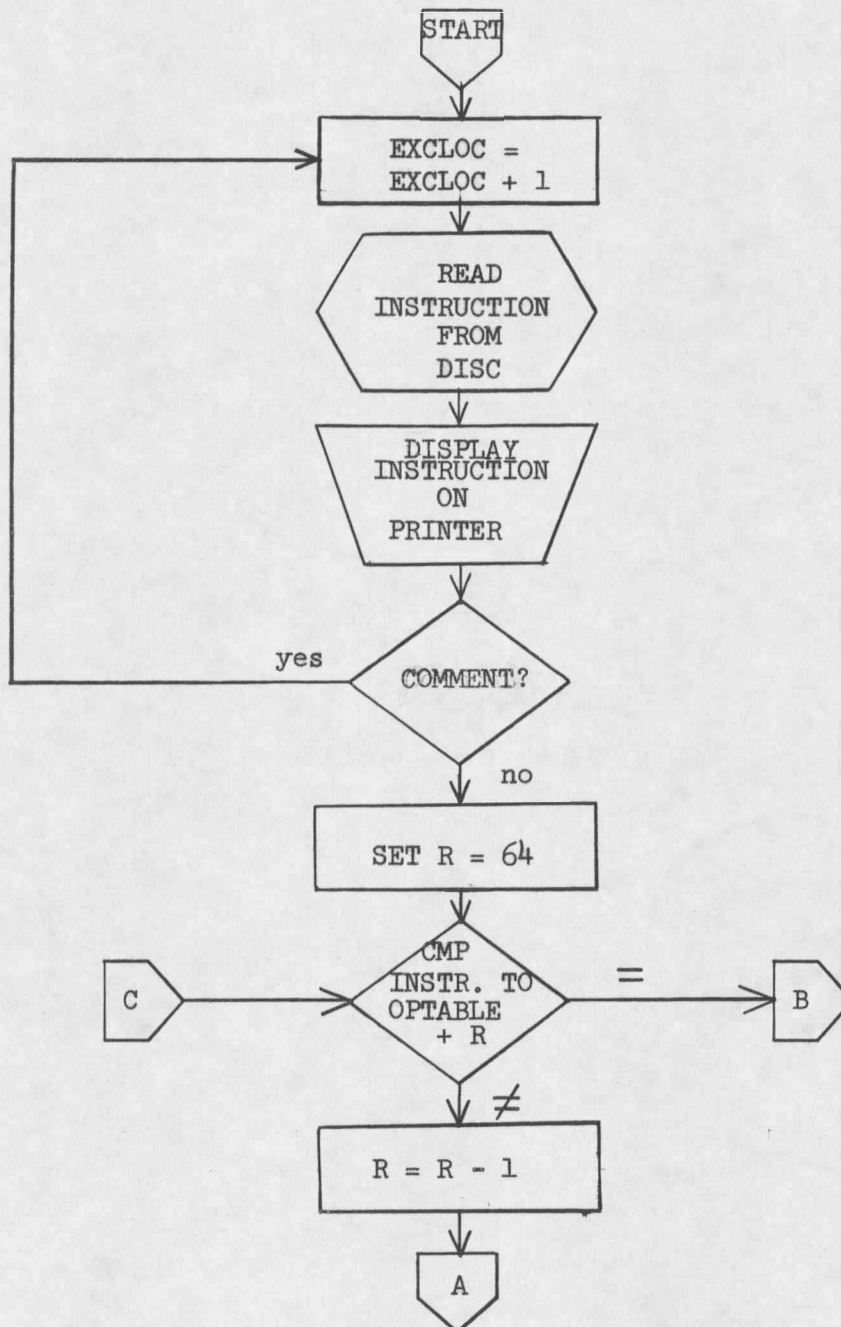


Figure 2.4. (continued)

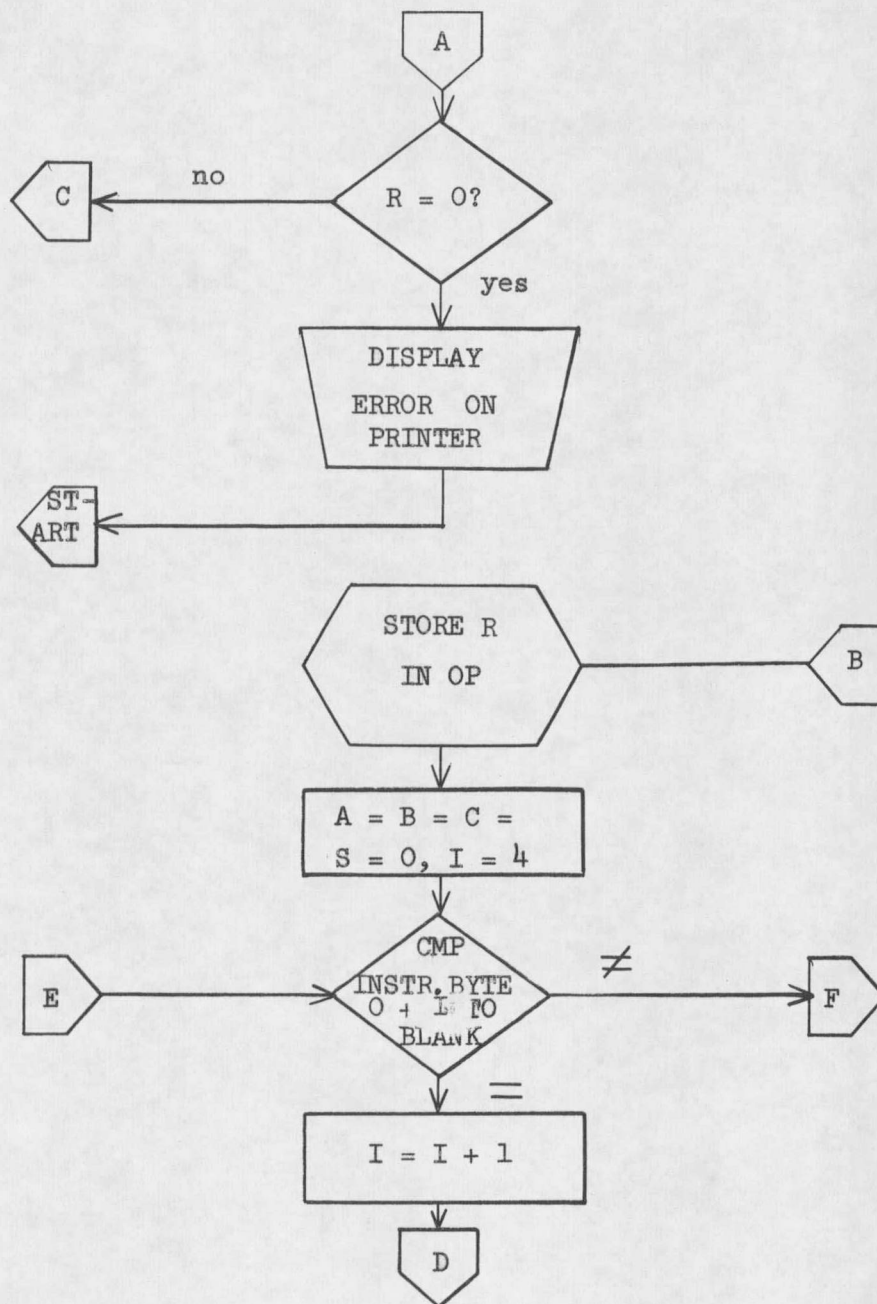


Figure 2.4. (continued)

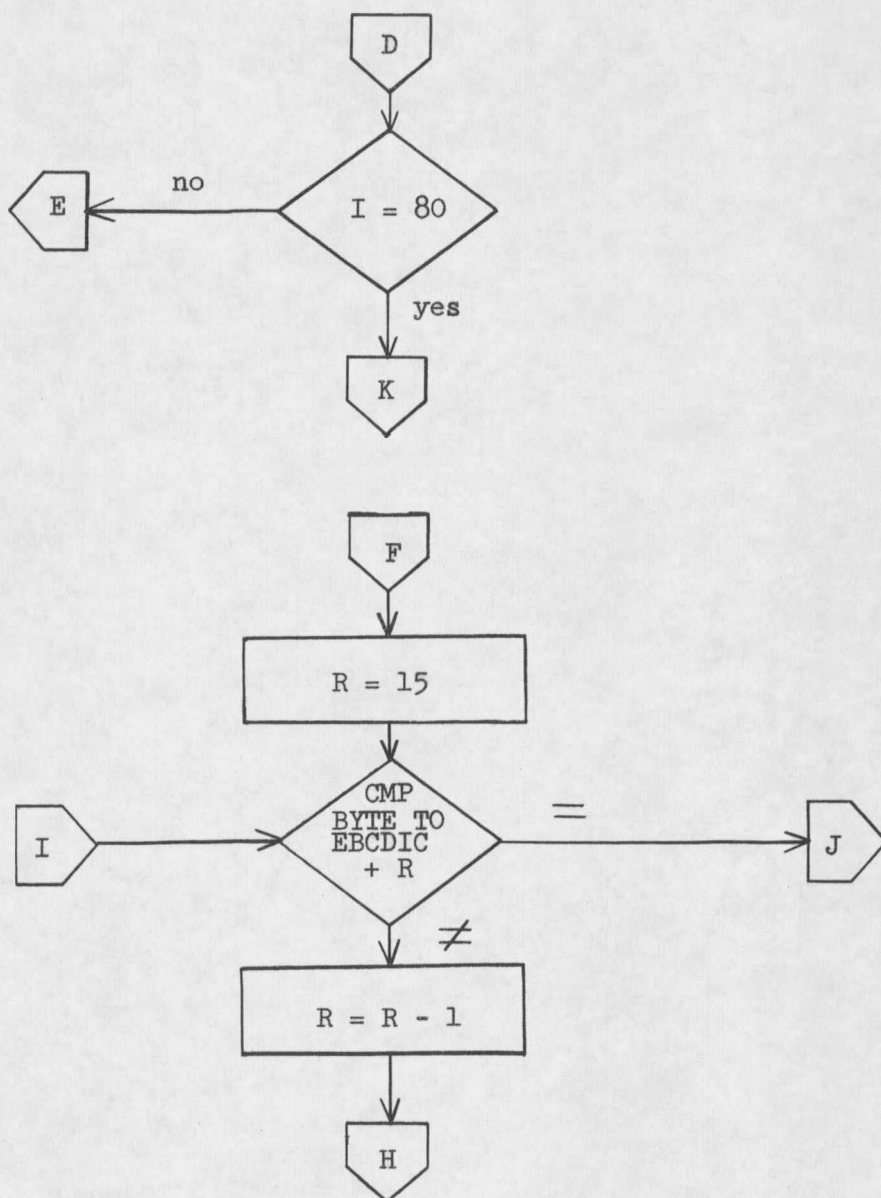


Figure 2.4. (continued)

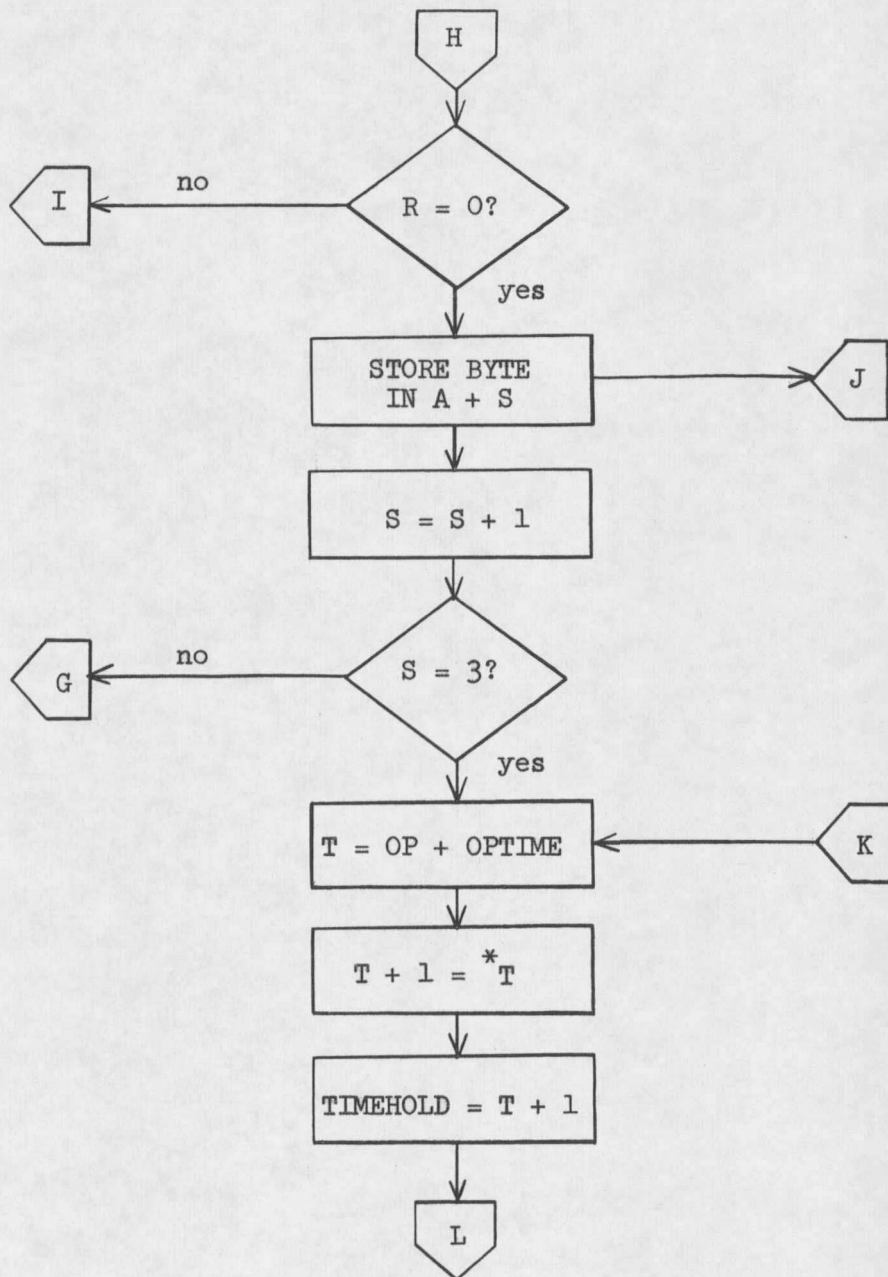


Figure 2.4. (continued)

* denotes indirect addressing

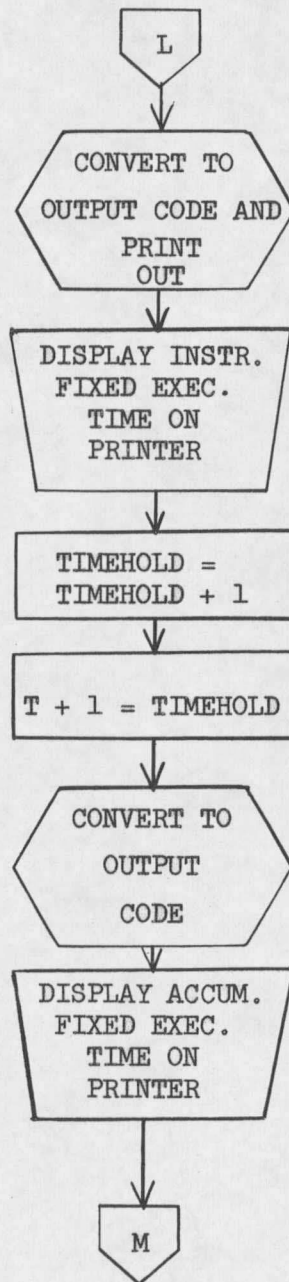


Figure 2.4. (continued)

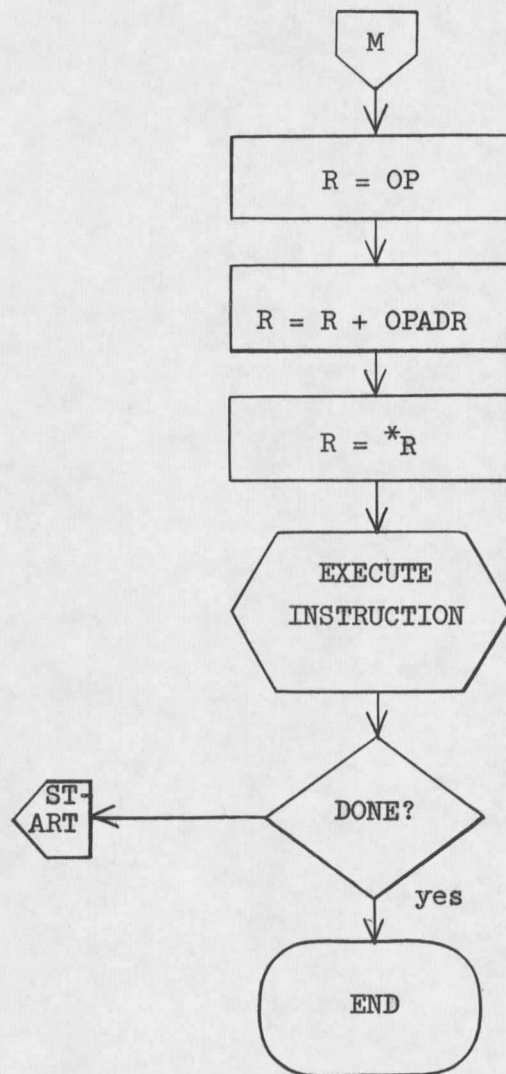


Figure 2.4. (continued)

2.4 Memory Storage and Retrieval Routines

The routines to store and retrieve information from memory warrant comment. The STORE routine is shown in Figure 2.5 and evaluates equation 2.1 via the subroutine ELADRGEN to generate the element doubleword address. This routine, shown in Figure 2.6, places the result of computing the memory location in register R. This result is then used as an index on MEMORY to uniquely select a doubleword in memory. The effective doubleword is then stored in the effective memory address.

To retrieve information from memory, the subroutine LOAD, shown in Figure 2.7, is used. Subroutine ELADRGEN is again used to generate the element indexing number, which is then used to index MEMORY. The doubleword specified by this indexing is then placed in doubleword register R for use within the program.

2.5 Arithmetic Subroutines

Arithmetic subroutines are used to implement the assembly language instructions requiring arithmetic operations. Two's complement arithmetic is used throughout. Although not directly referred to in using the simulation program, these subroutines form the working base of all assembly language instruction operations and are represented by the execute instruction block shown in Figure 2.2. These subroutines call other subroutines that normalize processing array data, and that compute incremental and total execution times in machine clock periods.

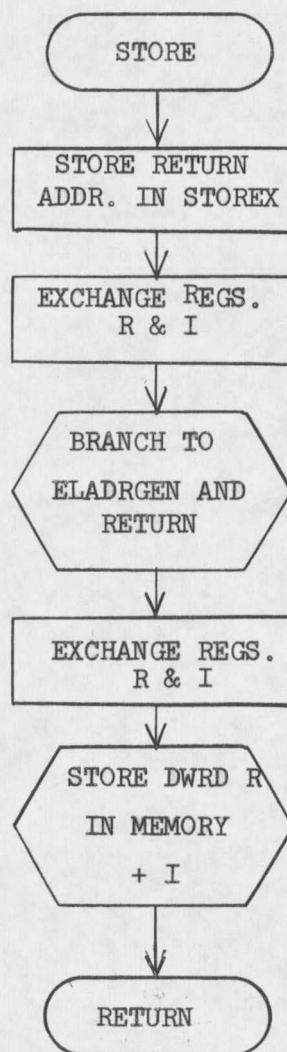
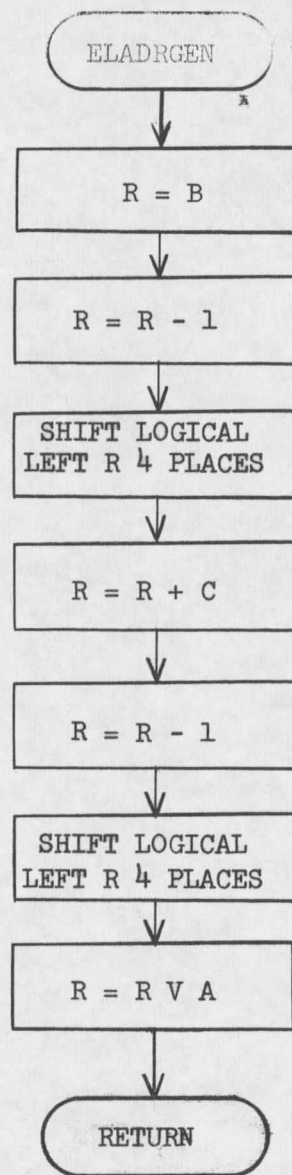


Figure 2.5. Flow Chart for Storage of Doubleword in MEMORY



Note: V denotes the logical OR operation.

Figure 2.6. Flow Chart for Generating Element Address

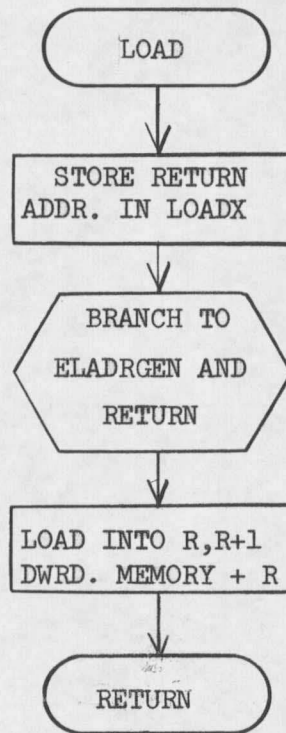


Figure 2.7. Flow Chart for Retrieving Doubleword from MEMORY

2.5.1 Addition-Subtraction Subroutine

Addition of floating point data is accomplished by the ALNADD subroutine shown in Figure 2.8. The subroutine receives the two data words to be added in double registers R and T. The exponent of the smaller number is then aligned to that of the larger number by right shifting its fraction and increasing its exponent for every shift. The shifting occurs a maximum of 24 times, the exponents being compared after each shift. When the exponents are aligned, the fractions are then added and the resulting number stored in double register R. If either of the received fractions is zero, the main operating part of this routine is bypassed and the non-zero number stored in double register R. If both members are zero, the main operating part of the routine is again bypassed, and the result zero is stored in double register R.

2.5.2 Multiplication Subroutine

The MULTIPLY subroutine, shown in Figure 2.9, is used to multiply two floating point numbers. The multiplier and multiplicand are brought into the subroutine in double registers R and T. Initially, the exponent of the product is found and stored in PROD. The fractional part of the product is found by forming doublewords from the fractional parts of registers R and T. The sign bit is then extended left on both extended fractions. The product register is then zeroed. The multi-

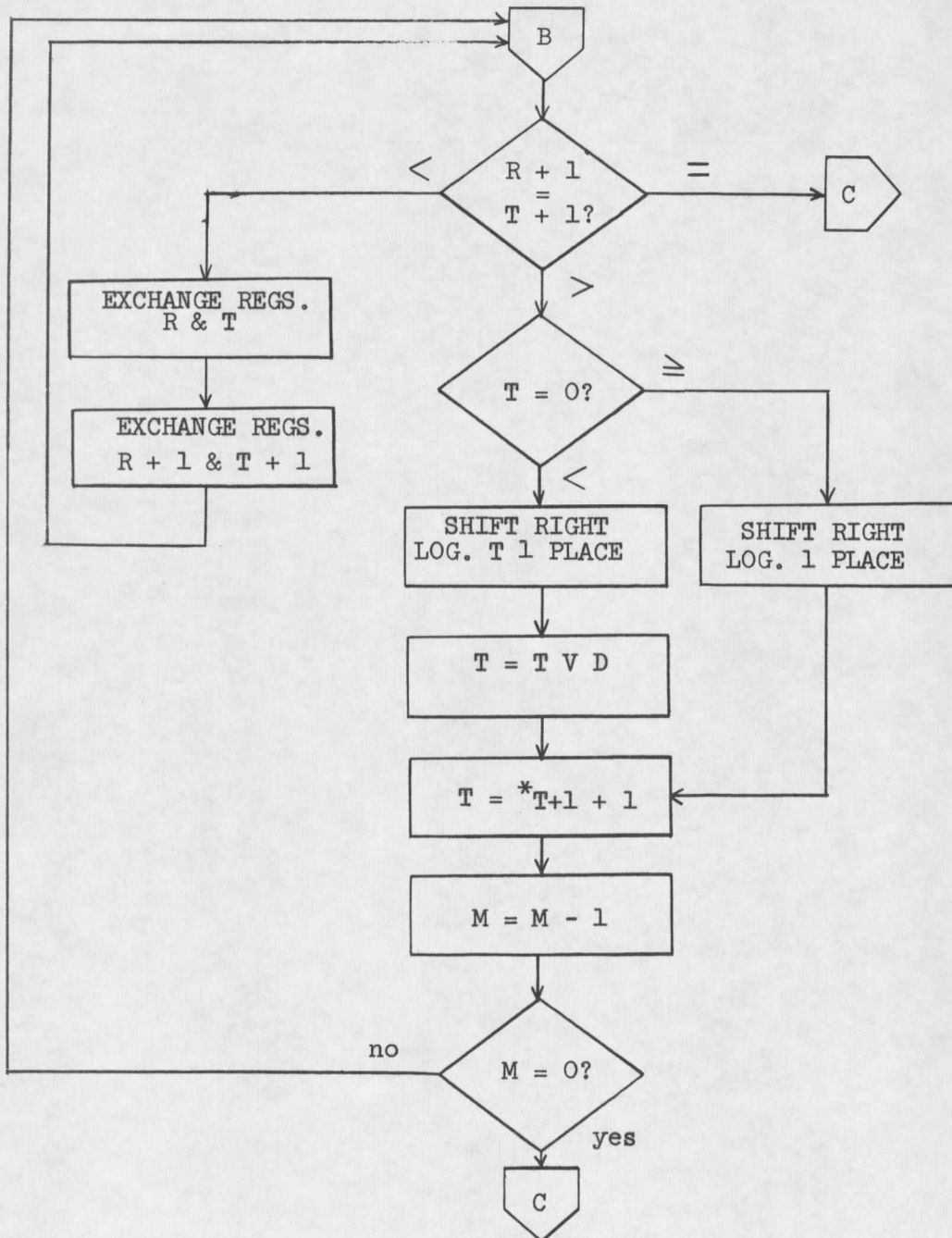


Figure 2.8. (continued)

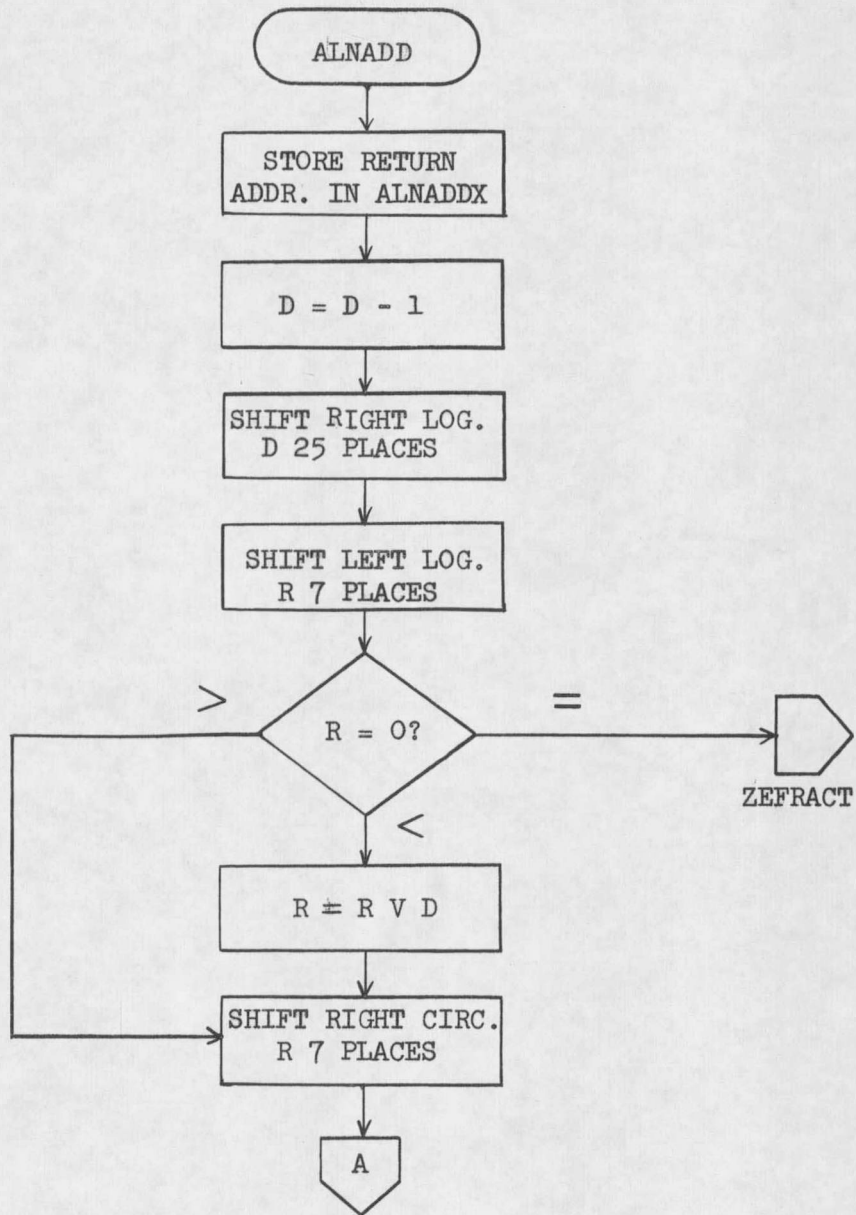


Figure 2.8. Flow Chart for Subroutine ALNADD

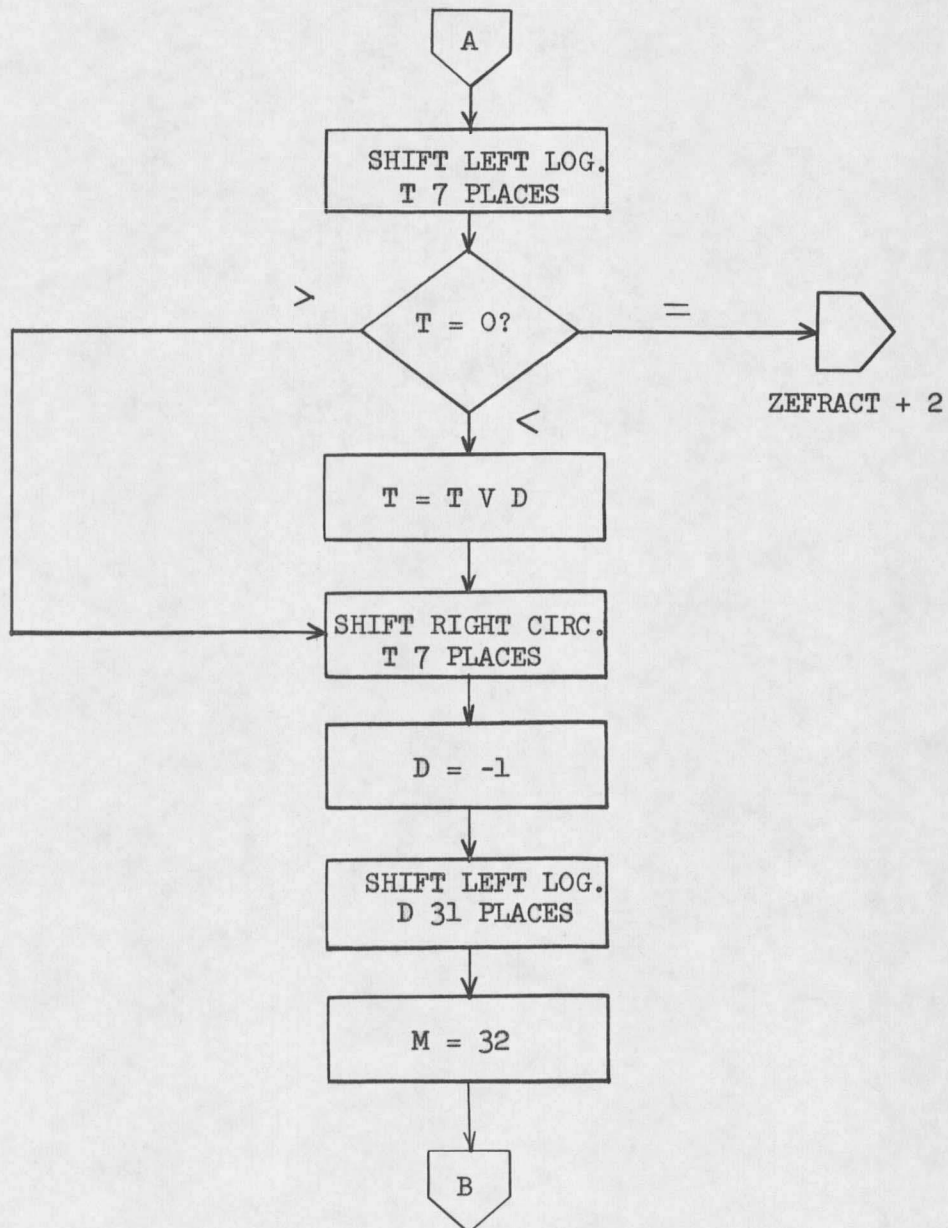


Figure 2.8. (continued)

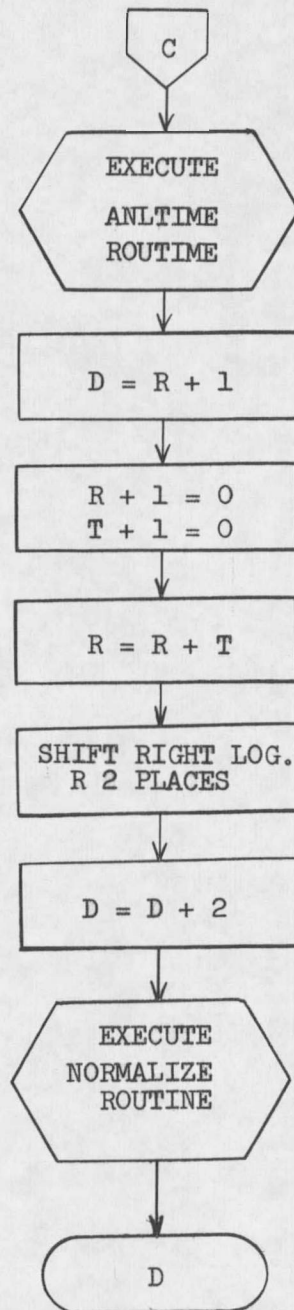


Figure 2.8. (continued)

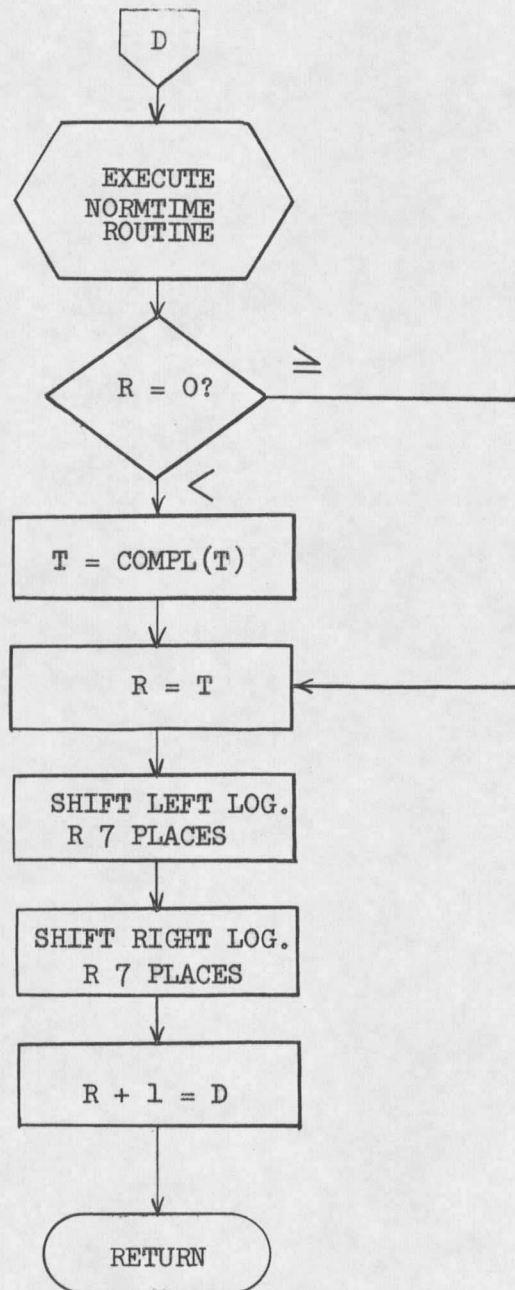


Figure 2.8. (continued)

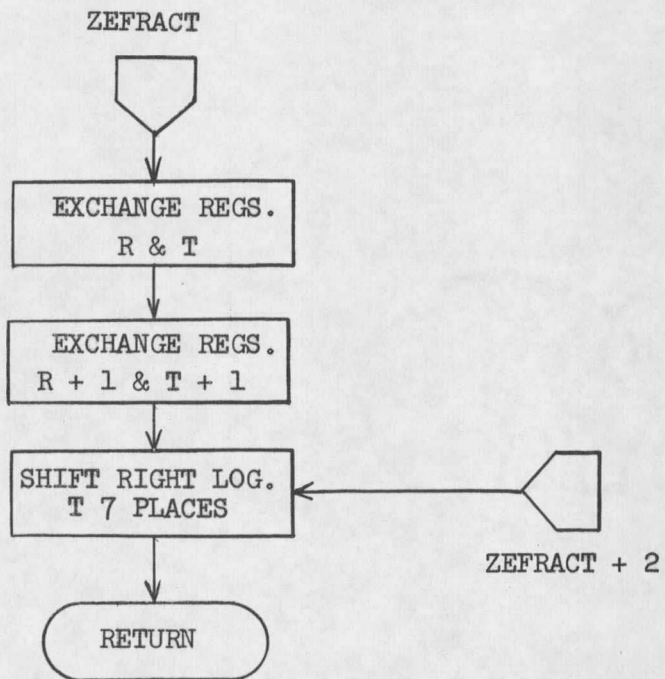


Figure 2.8. (continued)

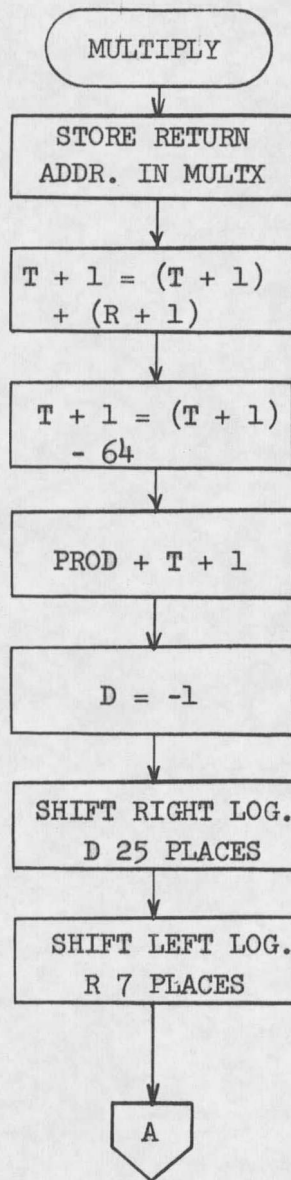


Figure 2.9. Flow Chart for Subroutine MULTIPLY

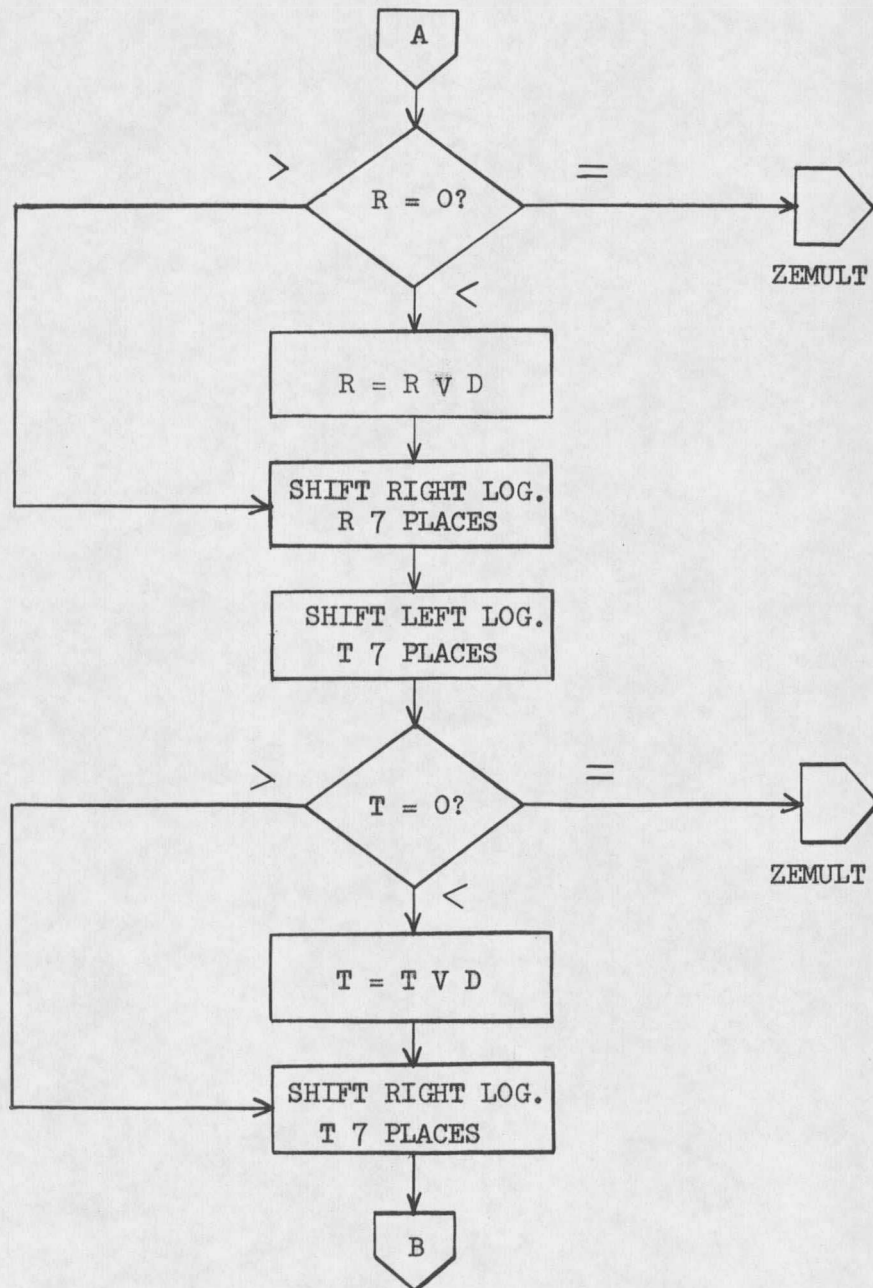


Figure 2.9. (continued)

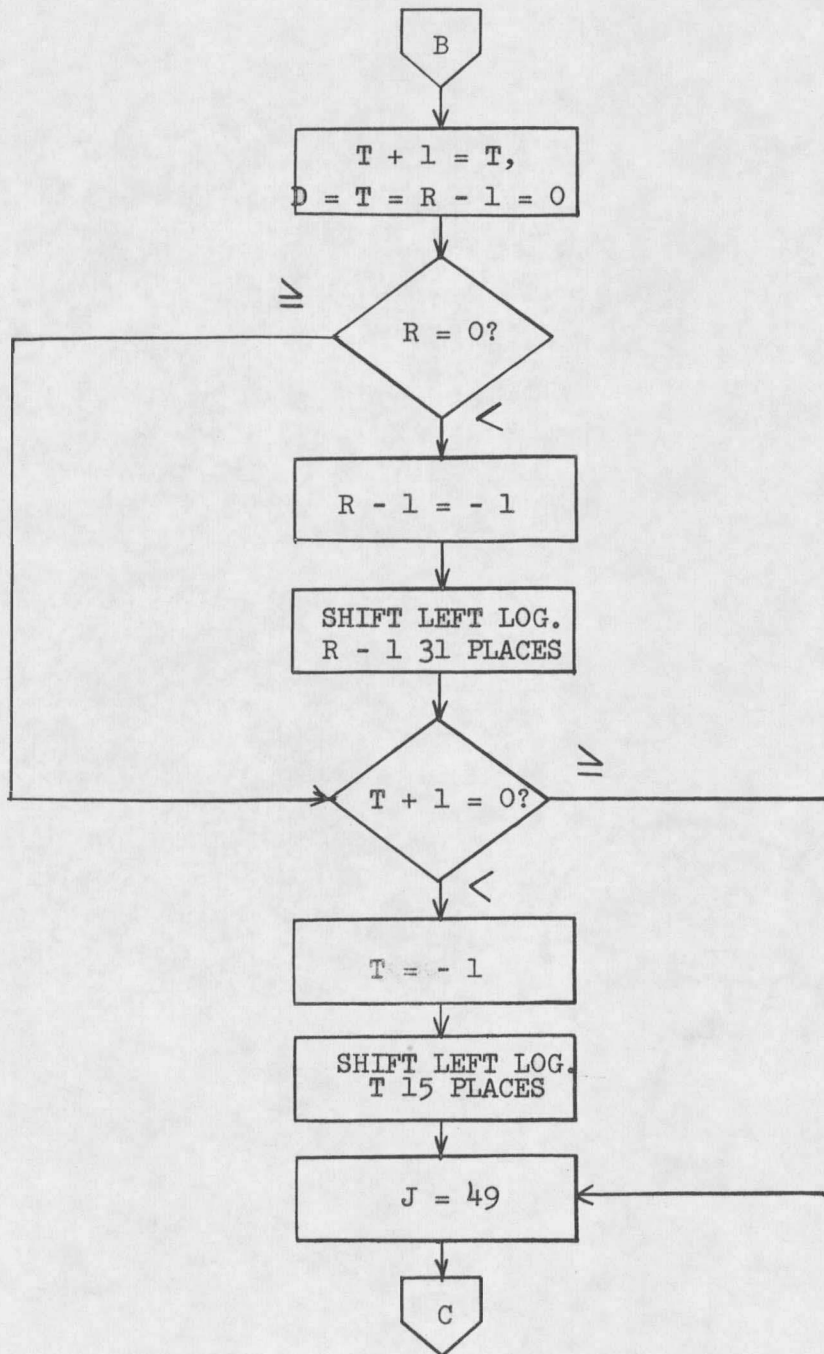


Figure 2.9. (continued)

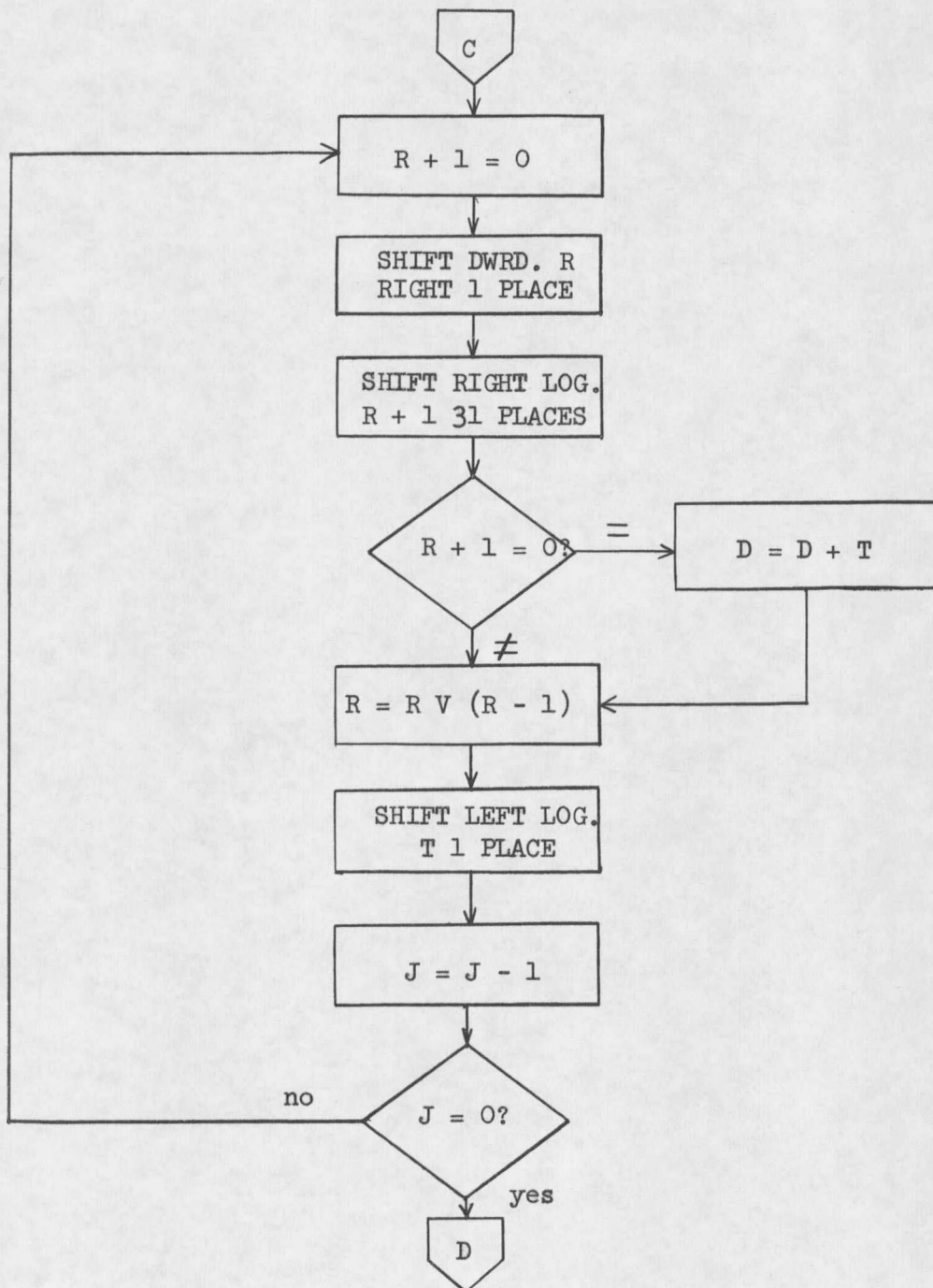


Figure 2.9. (continued)

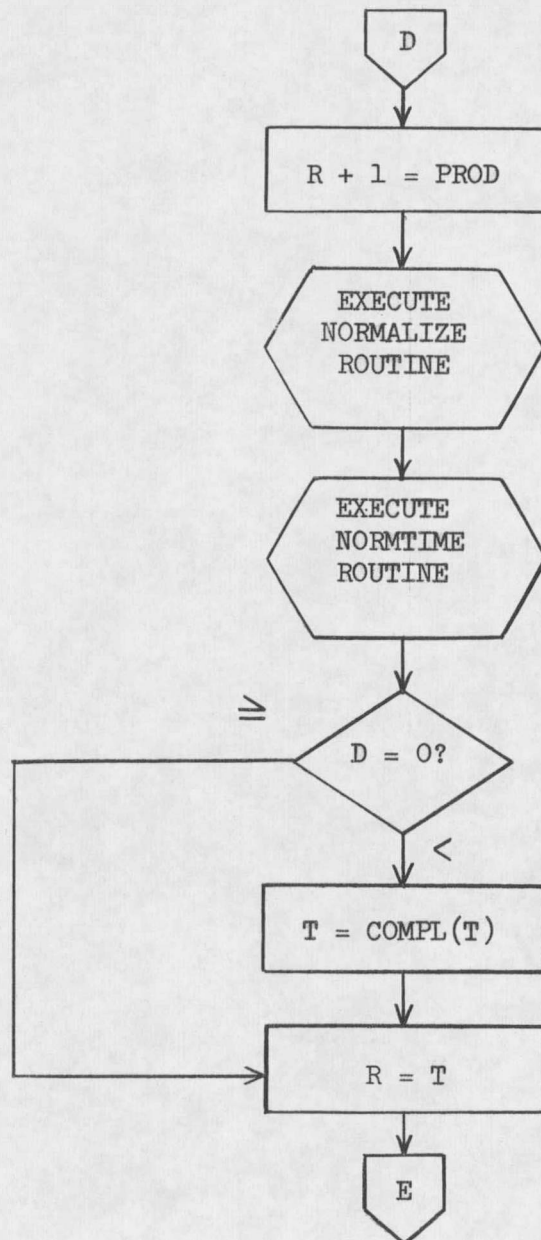


Figure 2.9 (continued)

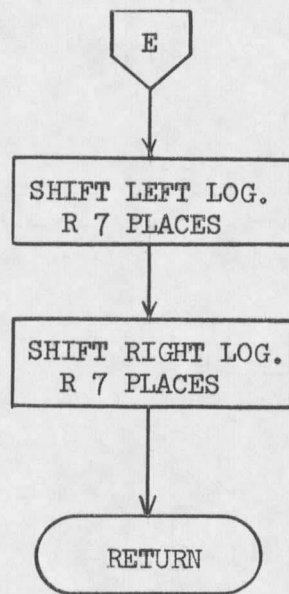


Figure 2.9 (continued)

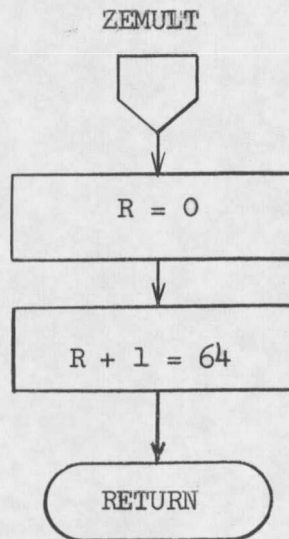


Figure 2.9. (continued)

plication process proceeds by shifting the multiplier left one bit and the multiplicand right one bit. If the lowest order bit is now 1, the multiplier is added to the product register; if the bit is now zero, no addition occurs. Then the lowest order bit in the multiplicand is deleted, and the product register is shifted left 1 bit position. This process is repeated 49 times resulting in the product of the two input floating point fractions. The product fraction is placed in register R and the product exponent in register R + 1.

2.5.3 Division Subroutine

The RECIP subroutine, shown in Figure 2.10, is used to find the reciprocal of an element utilizing restoring binary division. The floating point number to be used is brought into the double register R. Initially, the dividend is set to zero by zeroing registers T and T + 1. If the divisor is negative, NEG is set to -1; otherwise, NEG is set to 0. The divisor is then subtracted from the dividend. If the result is positive, a 1 is stored in register D; if the result is negative, a 0 is stored in register D and the dividend is restored by adding back the divisor. The dividend and quotient are then shifted left one bit position, and the process is repeated. If the subtraction of the divisor from the dividend results in a zero fraction, the process is terminated; otherwise the shifting is repeated 24 times.

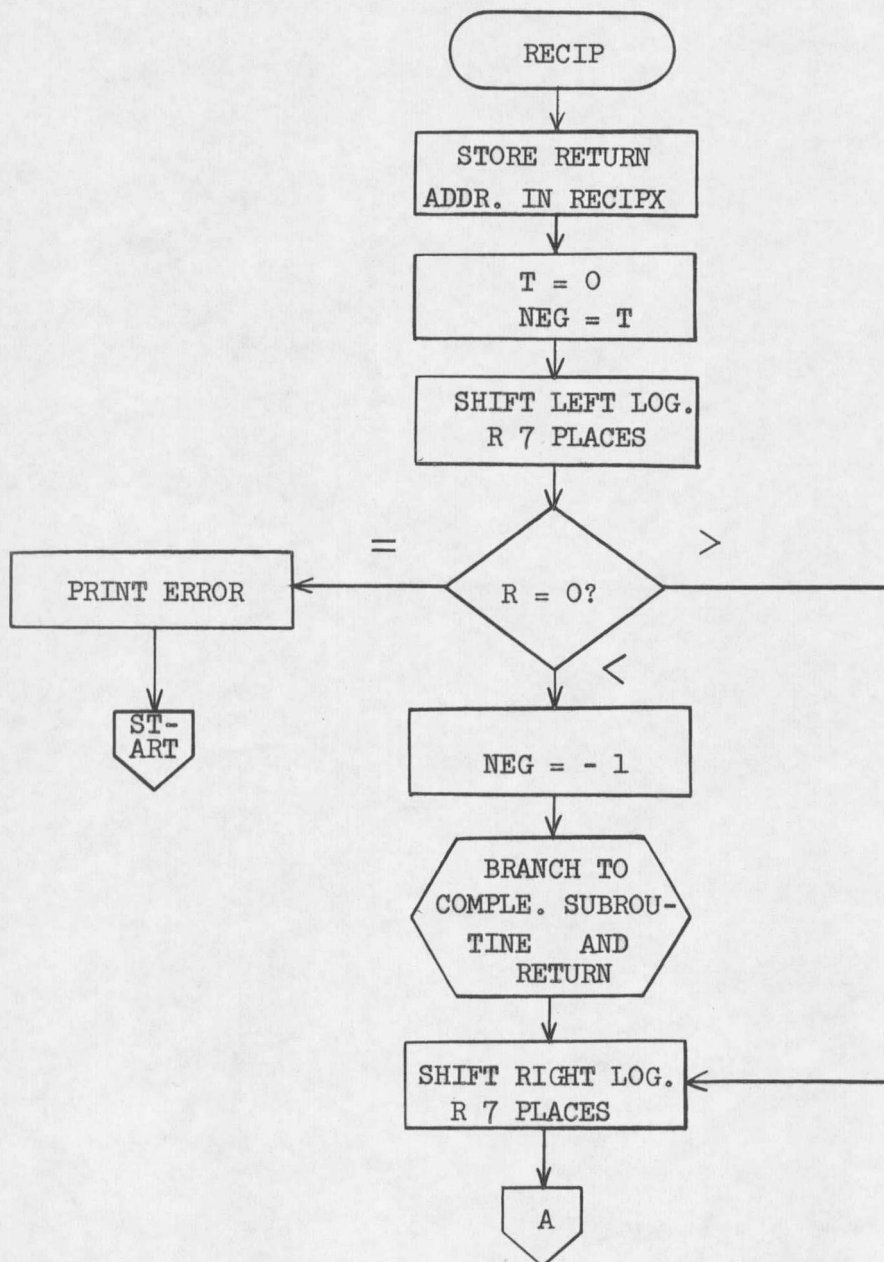


Figure 2.10. Flow Chart for Subroutine RECIP

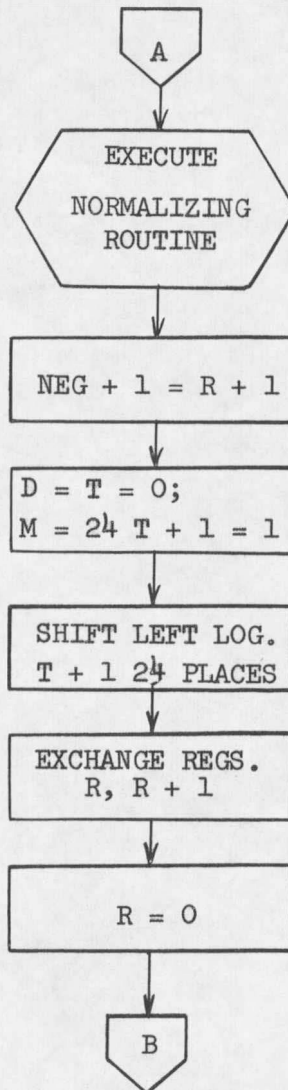


Figure 2.10. (continued)

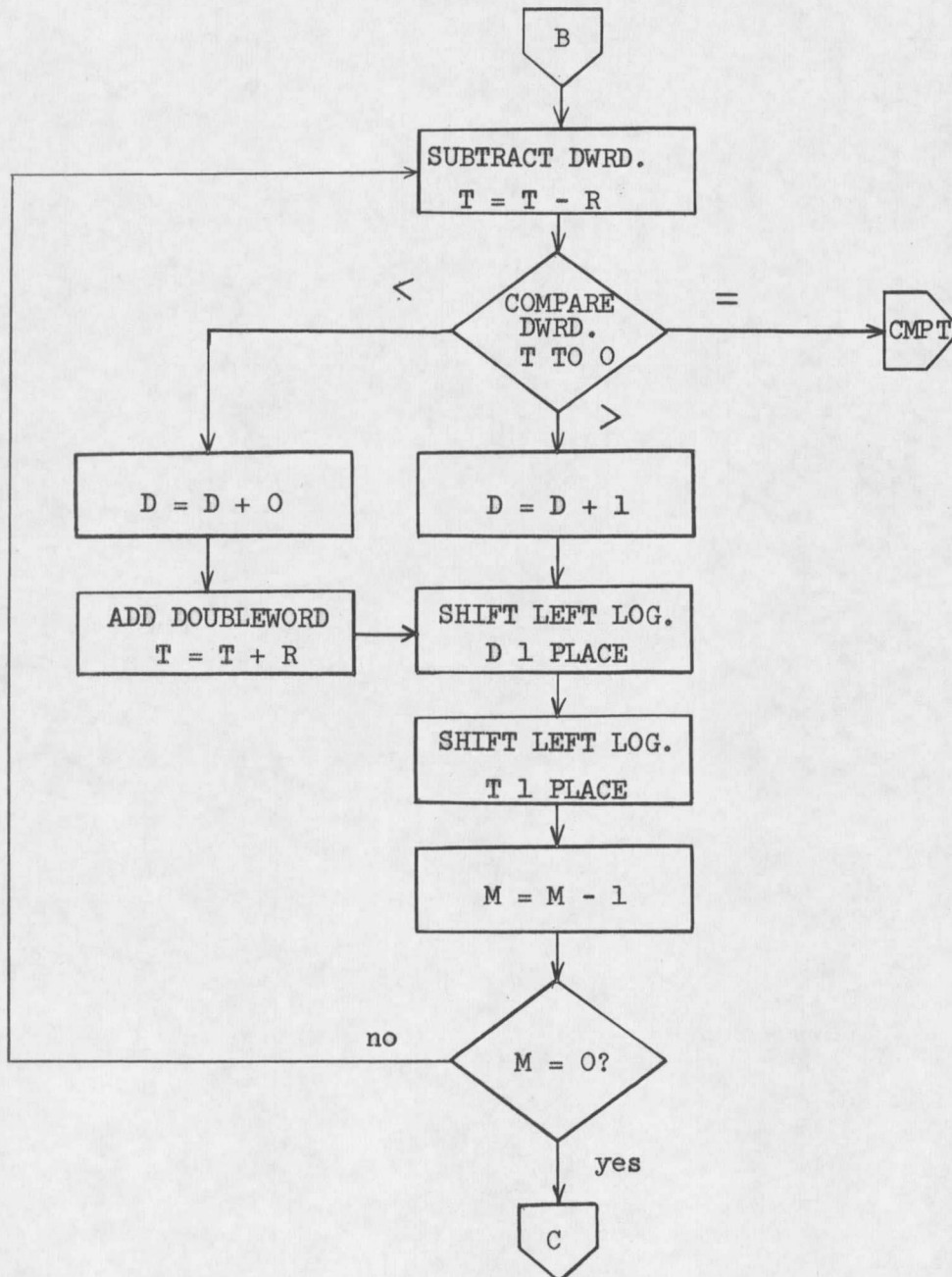


Figure 2.10. (continued)

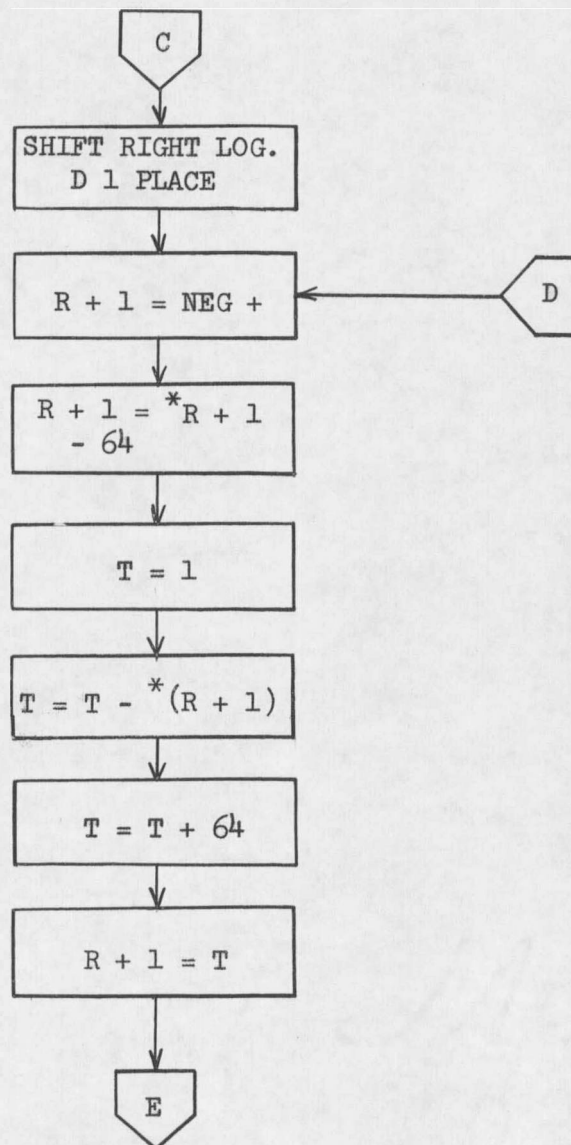


Figure 2.10. (continued)

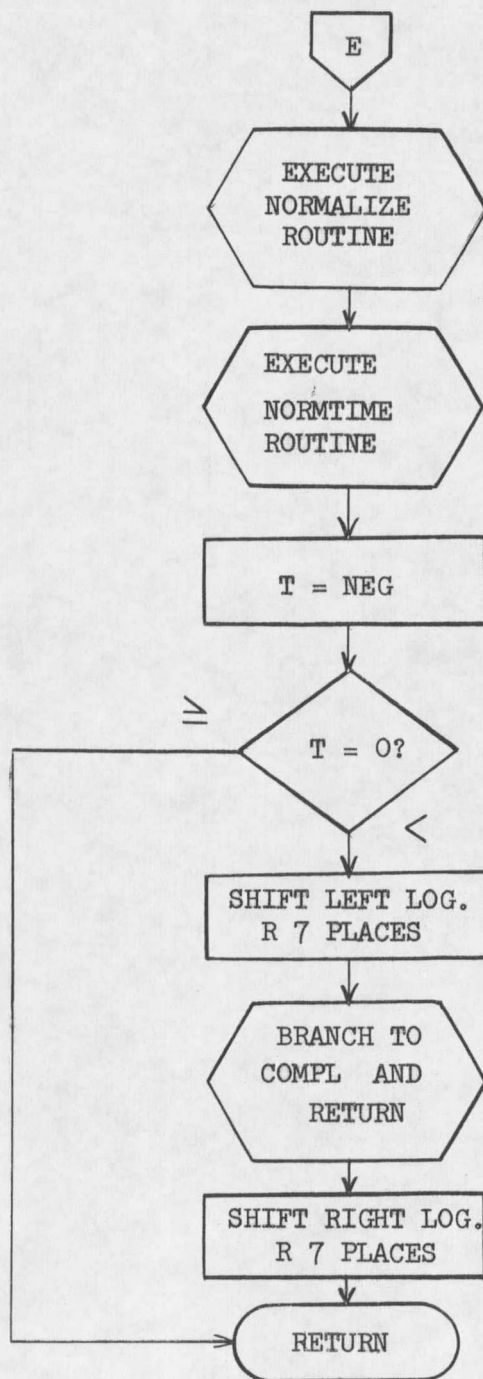


Figure 2.10. (continued)

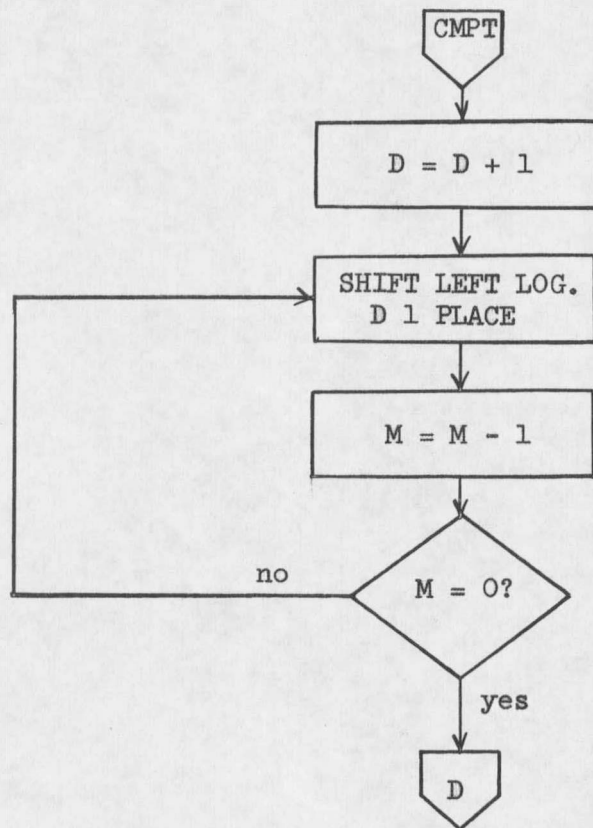


Figure 2.10. (continued)

If the divisor is zero, an ERROR message is printed as output information. The fractional part of the quotient is stored in register R. The calculated exponent part is stored in register R + 1. If NEG is -1, the complement data word is formed.

2.6 Normalization Subroutine

To insure full accuracy during implementation of floating point mathematical operations, all input-output data and resultants are normalized. To accomplish this data normalization, subroutines have been formed from the generalized subroutine NORMALIZE (see Figure 2.2). In these routines, the fractional part of a floating point number is left shifted until a condition code of '4' is found. This corresponds to a non-zero bit detection in the first bit location of the fraction. With every left shift of the fraction, the exponent is decreased by one. This normalization process assures a full 32 bits of binary information, and eliminates the propagation information loss. The test word NORMTEST (designator for a hexadecimal word ('00800000')) is used in all normalizing subroutines to determine the condition code.

The generalized NORMALIZE subroutine is shown in Figure 2.11. The exact form varies, depending upon in which mathematical subroutine it is inserted. Initially, the sign of the floating point number to be normalized must be determined. If positive, the routine continues;

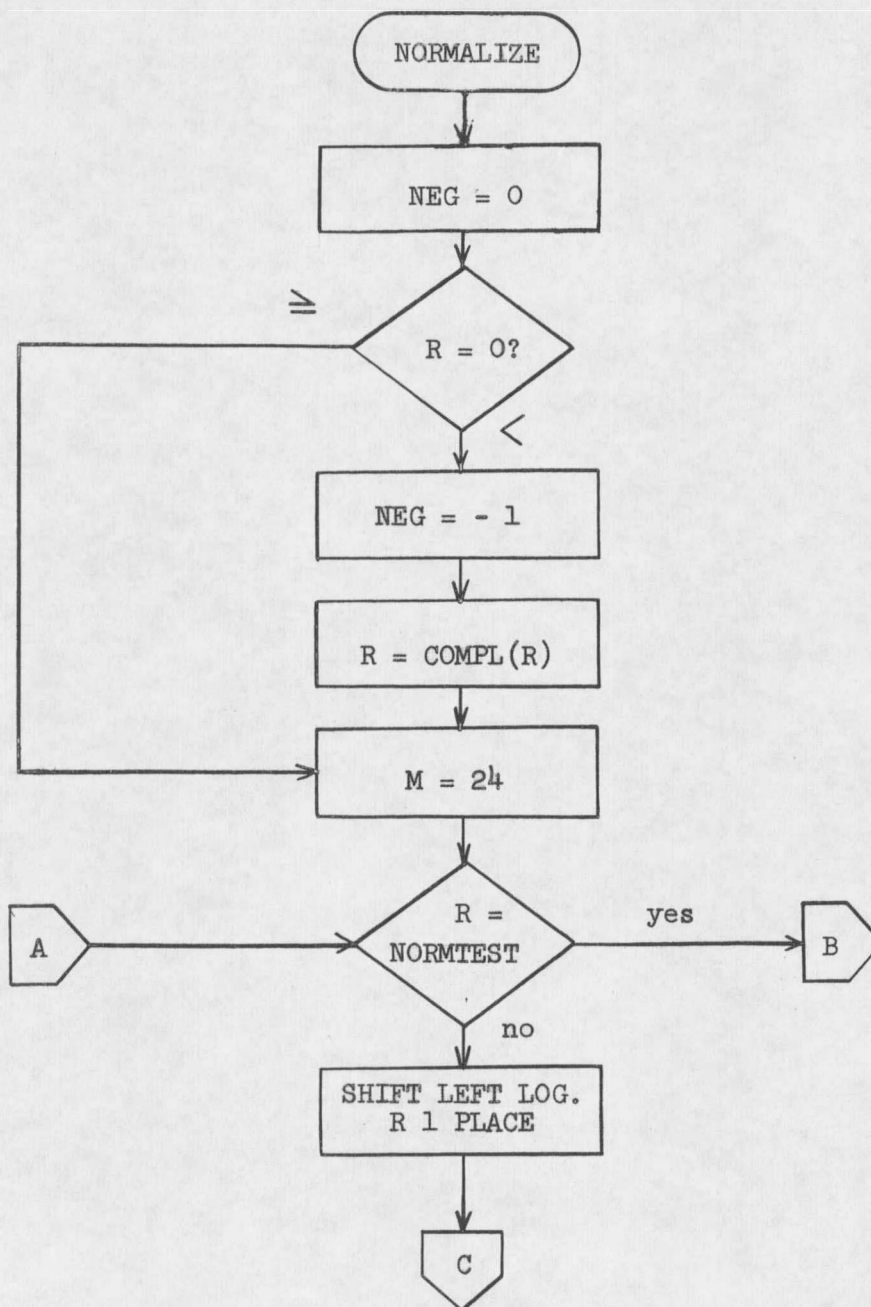


Figure 2.11. Flow Chart for Subroutine NORMALIZE

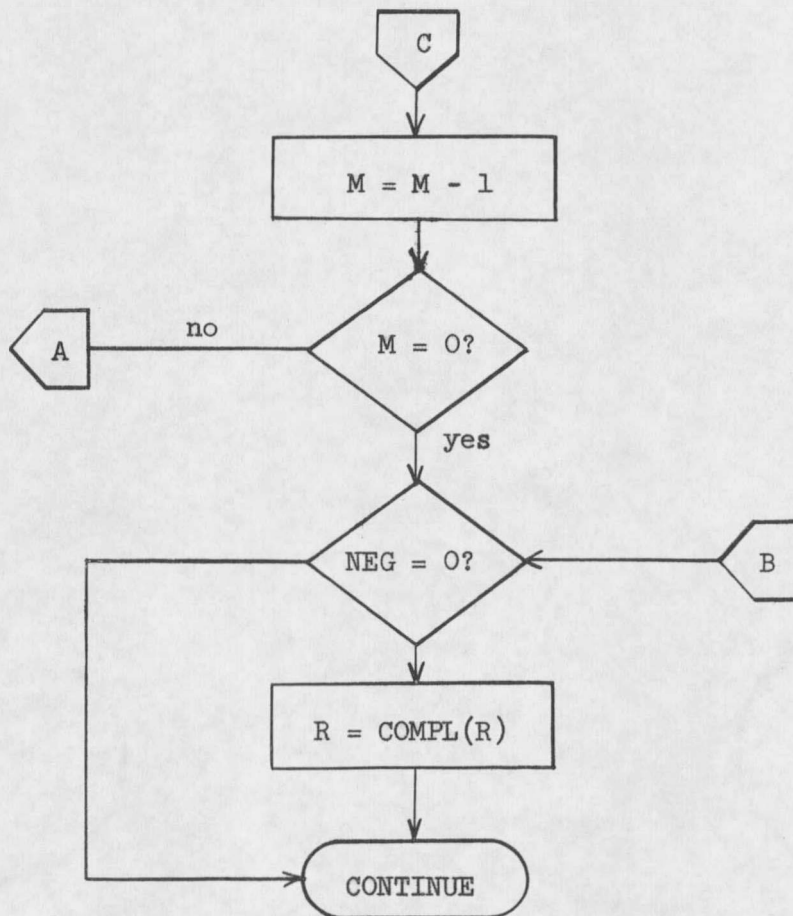


Figure 2.11. (continued)

if negative, the flag NEG is set to -1 and the complement of the fraction is formed. Hence, the NORMALIZE routine is always applied to the absolute magnitude of the floating point fraction. A maximum of 24 shifts is allowed by setting the decrementing register (M) to 24. The fractional magnitude is then left shifted until one or more digits are equal to the test word NORMTEST. Hence, the process ends when a non-zero bit appears in bit position 8. When the shifting process has terminated, NEG is compared to zero. If NEG is negative, the complement of the fraction is formed; if NEG is zero, the fraction remains unchanged. Where applicable to retain all bit information, doubleword normalizing routines are used. In all routines, if the floating point number is zero, the routine is bypassed. The number of left shifts required for normalization in all mathematical routines is stored for use in the timing subroutines.

2.7 Timekeeping Subroutine

Timekeeping subroutines are included in all assembly language execution routines. These subroutines (see Figure 2.2) accumulate fixed and variable clock periods associated with execution of a given instruction. The fixed execution times are table loaded (OPTABLE), and are dependent on the dimension of the matrix operations being performed. The variable times depend on the number of alignment and normalization shifts required in the execution of floating point arithmetic operations.

A generalized subroutine (NORMTIME) to find the number of shifts is shown in Figure 2.12. Register M is used to keep a count of the alignment or normalizing shifts required for a given mathematical operation. However, this register information is subtracted from the beginning index number to find the actual number of shifts that have occurred. This result is then stored for future reference. Included in the timekeeping routines are subroutines that find the maximum shifts and the accumulative shifts required in the execution of an array operation. All timing information is printed out in terms of equivalent array-structured computer clock periods.

2.8 Output Data Structure

2.8.1 Assembly Language Instructions

The output of PAL-1 consists of a printing of the instruction mnemonic, timekeeping data, and the output data if indicated. The output format is the same for all assembly language instructions with the exception of the special assembly language instructions MULT and INVA. The output format is specified in Table 2.2; the item 'Ø' denotes a blank line. If timekeeping information does not exist, the output position is deleted.

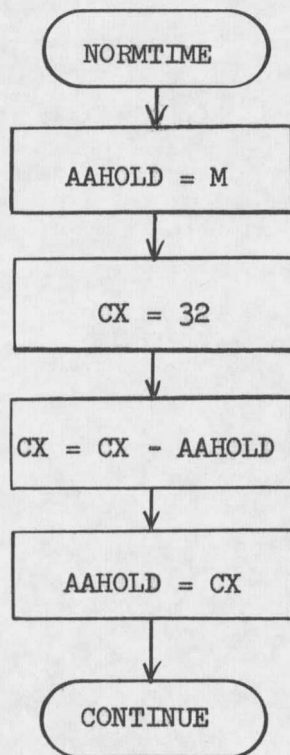


Figure 2.12. Flow Chart for Subroutine NORMTIME

Table 2.2 PAL-1 Output Format

Instruction mnemonic

✎

Fixed execution time for this instruction

Accumulated execution time including the fixed execution time to this point in program

✎

Maximum number of alignment shifts occurring in ALNADD

Maximum number of normalizing shifts occurring in ALNADD

✎

Maximum number of normalizing shifts occurring in MULTIPLY

✎

Maximum number of normalizing shifts occurring in MULTIPLY

✎

Total accumulative execution time to this point in the program execution

✎

Output data if indicated

This output format provides a full analysis of the accuracy and required execution time in clock periods for the implementation of any given algorithm.

2.8.2 Special Instructions

The output of PAL-1 for the special instructions MULT and INVA is similar to the normal assembly language format, but requires a few additions as shown in Table 2.3.

Table 2.3 Special Instruction Output Format

Instruction mnemonic

✕

Fixed execution time for this instruction

Accumulated execution time including the fixed execution time to this point in the program

✕

Maximum number of alignment shifts encountered in any of the matrix additions required in this instruction

✕

Maximum number of total normalization shifts encountered in any matrix

✕

Maximum number of normalizing shifts encountered in any matrix inner product

✕

Maximum number of normalizing shifts encountered in any element reciprocal operation

✕

Total accumulative execution time to this point in the program execution

✕

Output data if specified

2.9 Implementation of the Kalman Filter Algorithm in PAL-1.

To analyze the operation of an array-structured computer, the Kalman filter algorithm has been selected for implementation in PAL-1. This algorithm is discussed in Appendix A and is typical of the type of matrix-oriented algorithms that can be efficiently implemented with this type of machine.

The linearized equation of the Kalman filter algorithm for tracking an object in three-dimensional space are also presented in Appendix A, and are used as an example for the simulation program. The PAL-1 instruction set implementing these equations is listed in Appendix D, with matrix allocations (operand identifier) as shown in Table 2.4.

Table 2.4 Matrix Allocation for the Kalman Filter Example

Matrix Allocation (hexidecimal)	Identifier and Purpose
0	R - Covariance matrix of observation noise
1	Q - Covariance matrix of target exciting noise
2	Γ - Distribution matrix
3	ϕ - State transistion matrix
4	$\sum_{k,k+1}^{k,k+1}$ - Covariance matrix of pre-dicted state error

Table 2.4 (continued)

Matrix Allocation (hexidecimal)	Identifier and Purpose	
5	$x_{k,k-1}$	- Estimate of $x(k)$ based on $k-1$ observations
6	G_k	- Optimal filter gain
7	z_k	- Actual observation
8	H_k	- Linearized observation system
9	u_k	- Known target accelerations
10-C	Matrix working areas subscripts	

The instruction set is divided into two sections: the initialization section and the iteration section. The initialization section specifies the number of iterations to be performed (INPI), the maximum dimension of rows (SETR) and columns (SETC), and zeroes all used memory areas. In addition, this section inputs into the allocated matrix areas algorithm initialization data (matrices 0-6). The iteration section contains the linearized Kalman filter operations beginning with the data input of the new state vector observation (INPC 7 1). In sequence, the program then calculates the following quantities:

1. Covariance matrix of predicted state error
2. Covariance matrix of predicted observation vector error
3. Kalman filter gain factor
4. State vector prediction
5. State vector estimate
6. Covariance matrix of estimated state error

The BDIZ instruction detects when the specified number of iterations have been executed, and then passes the program execution on to the termination flag (\$\$\$\$).

To aid in evaluating the operation of an array-structured computer, the same linearized equations for the Kalman filter algorithm were implemented conventionally in FORTRAN IV. Comparison indicates that the maximum deviation between the results of the two computer programs is less than 0.019%. The comparison is made between state vector estimates after the 12th iteration, as shown in Table 2.5.

The variation between the two program results is accounted for in part by the different matrix inversion routines used in the programs (FORTRAN IV--escalator; PAL-1--Gauss-Jordan), and in part by the data input-output roundoff error that occurs in this simulation of a base two machine on a base sixteen computer (Sigma 7).

When applied to a specific example such as the Kalman filter algorithm, the timekeeping subroutines yield important information on the efficiency of an array-structured computer. The information

resulting from this example is listed below:

1. The 12th iteration required 2681 clock periods to execute.
2. Assuming a 100 nanosecond GCU clock period, this 6x6 matrix-oriented algorithm is updated 3730 times per second.
3. The Gauss-Jordan matrix inversion routine required 500 clock periods (50 microseconds) for a 3x3 matrix inversion.

Table 2.5 State Vector Estimate Comparison

Program	State Vector Estimate--Absolute					
	X ₁	X ₂	X ₃	X ₄	X ₅	X ₆
FORTTRAN IV	3193.742	964.324	5565.360	131.173	560.524	634.053
PAL-1	3194.332	964.266	5565.520	131.188	560.531	634.057
Maximum Deviation %	0.019	0.006	0.003	0.011	0.001	0.001

Due to the mechanics associated with an array-structured computer, the implementation of the Kalman filter algorithm is a minimum of N^2 ($6 \times 6 = 36$) times faster on this machine than is possible on a sequential machine with the same clock period. It is quite likely that this factor of improvement will be even larger since sequential implementations of this algorithm require more time for bookkeeping

and indexing of elemental data. The exact factor of improvement depends on the sequential machine used and the efficiency of the application program.

2.10 Results Derived from the Simulation Program

The parallel assembly language simulation of an array-structured computer has yielded considerable information on the operation of such a machine. This information reveals important design considerations which influenced the array-structured computer design presented in the next chapter. These design considerations are listed below:

1. One of the most critical operations encountered in an array-structured computer is the transfer of data between processing cells. Due to the need to improve the speed of the data transfer operation, the following considerations are presented:
 - a. A comparison of serial and parallel data transfer between cells indicates a 32:1 improvement in speed for parallel data transfer.
 - b. A comparison of serial and parallel data transfer applied to a typical algorithm

implementation (i.e., Kalman filter algorithm in PAL-1) indicated a 4.5:1 speed improvement with parallel data transfer.

Due to these speed improvements, strong considerations must be given to parallel data transfer within the processing array.

2. The control requirements for data transfer operations are not as complex as those required for mathematical operations. This consideration combined with those for speed given above, suggest the use of a separate, faster data transfer cycle, and a segmentation of the processing cell. This segmented organization splits the data transfer operations and the mathematical operations allowing each to operate under a separate clock system.
3. The most complicated data transfer operation is the transpose of matrix data operation. Consideration should be given to a special buss structure to accommodate this operation.
4. To implement the special operations of rotating, skewing, transposing, and broadcasting (see section 2.2), a horizontal and vertical segmented data buss structure composed of 32 parallel lines each should be considered.

5. If serial data input is used, input-output of array data is a time consuming process. Parallel input of data by entire rows or columns may be the best solution to this problem.
6. A separate input-output processor should be used to facilitate input-output of array data, and to allow input-output operations to occur simultaneously with array processing.
7. The most time consuming mathematical operation is that of floating point multiplication. A comparison of serial and parallel multiplication indicates a speed improvement of 16:1 with the parallel unit at the expense of additional hardware. Hence, consideration must be given to the design of a parallel multiplier.
8. Element by element floating point division is not necessary for the matrix class of operations, and may be relegated to a unit external to the processing array. However, provision should be made for future addition in the form of an auxiliary module.
9. Execution time required for arithmetic operations is saved if zero detection circuits are included to

allow bypass of the alignment and normalization cycles if a zero operand is detected.

10. A single floating point normalizing unit can be used to normalize the output of both the adder-subtractor and the multiplier.
11. Control of the array-structured computer at two levels should be considered; intracell control and processing array control.
12. The presence of special instructions (i.e., matrix multiplication and inverse) suggest the use of a separate block of memory between the GCU and the processing array to store sets of repetitive operations.

The simulation program resulted in some additional considerations which may influence the final design and application of this computer. These considerations are listed below:

1. Matrix-oriented algorithms are very easily implemented directly in assembly language, possibly eliminating the need for a GCU compiler.
2. Array-structured computer execution speeds are a factor of N^2 or more (since sequential machines require additional bookkeeping memory and control) faster than possible on sequential machines.

3. Input-output memory buffering should be considered to interface between input-output channels, exterior units, and the processing array.
4. Fault detection and fault correction techniques should be considered due to the complexity and intended application of the array-structured computer.

These design considerations are combined with those resulting from the survey of pertinent literature presented in section 1.3 to form the base for design of an array-structured computer specially organized to efficiently implement matrix operations.

CHAPTER 3

DESIGN OF AN ARRAY-STRUCTURED COMPUTER

3.1 Introduction

The subject of this chapter is the organization and logical design of an array-structured computer capable of efficiently implementing matrix-oriented algorithms. The computer system organization is based upon the design considerations resulting from the array-structured computer simulation (PAL-1) presented in section 2.10. In addition, design considerations resulting from the survey of array-structured computers presented in Chapter 1 are used to aid in forming the computer system organization (see section 1.7). These design features are: use of a two-dimensional processing array of identical modules; use of a horizontal and vertical data transfer structure; and incorporation of local and global control of the processing array. The logical design of the array-structured computer is conducted to the gate level, utilizing a selected set of cellular logic units. The use of these cellular units places emphasis upon an LSI implementation of the designed computer, a concept which should be possible within five years.

This chapter begins with a discussion of the computer system organization and operation. The next section contains the data representation and logic convention used in the logical design of the computer. The following sections contain the gate level design of each of the computer functional units, including a section on the adaptation of the array-structured computer to fault-tolerant

operation. The concluding section of this chapter contains a discussion of the hardware implementation of the designed computing system.

3.2 Computer System Functional Organization

3.2.1 Introduction

The system organization of the array-structured computer, shown in Figure 3.1, is based on the design considerations resulting from the survey of array-structured computers presented in Chapter 1, and the array-structured computer simulation presented in Chapter 2. The array-structured computer consists of four major functional groups: a global control unit (GCU); exterior units; X and Y command blocks; and a processing array composed of N^2 identical processing cells. Each of these is discussed below.

3.2.2 Global Control Unit Functions

Overall system control is provided by the global control unit, which is a 10 Megahertz (Mhz), 36 bit word, general-purpose computer. This unit performs computer system executive functions (system executive program), and initializes and executes assembly language instruction sets. It also contains a separate input-output processor to allow independent input-output processing of array data. To implement a given algorithm, the GCU is loaded with the appropriate

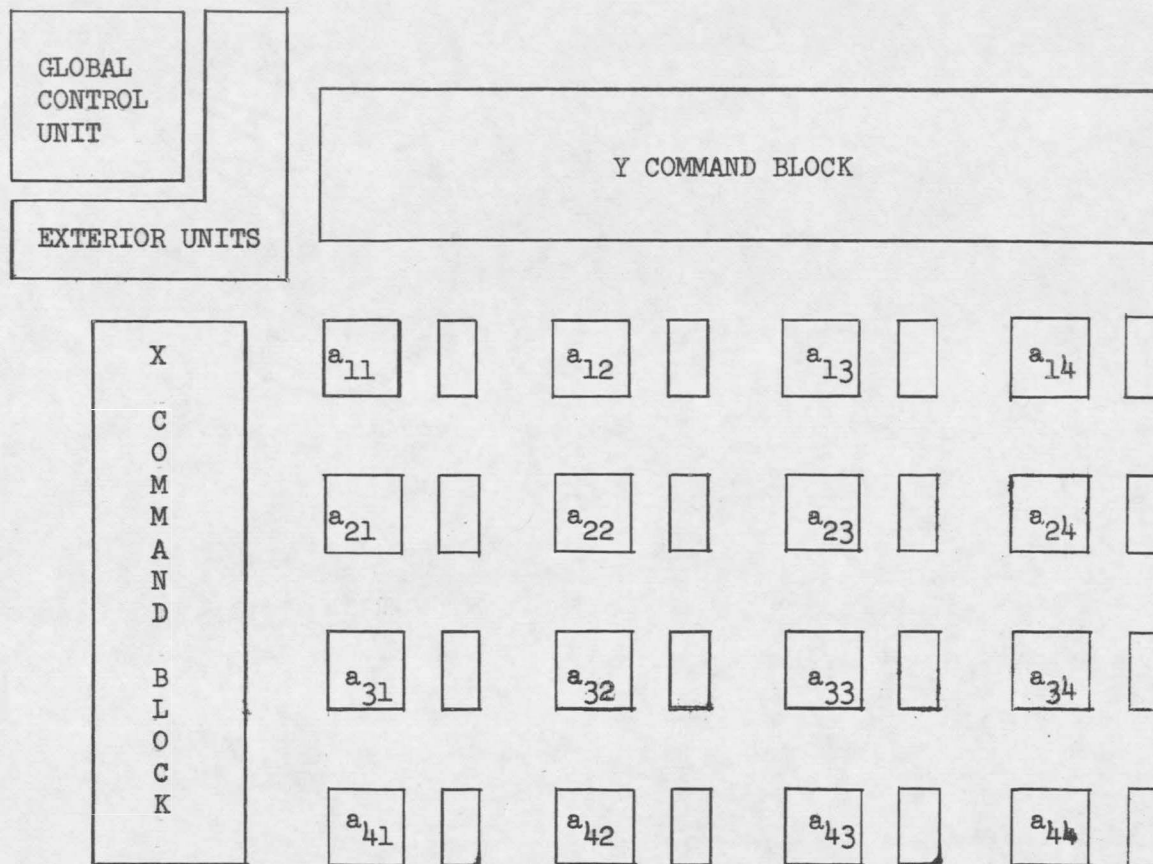


Figure 3.1 Array Structured Computer Organization

assembly language instruction set. The GCU system executive program then initializes and dimensions the given set of instructions, forming an equivalent set of macroinstructions*. The GCU then begins program execution and issues in sequence the macroinstructions to the processing array, data transfer microprocessor, or dynamic microprocessor. The X and/or Y command blocks are used to direct control information from the GCU or the microprocessors to the cells of the processing array. The processing array is reserved for array type operations, leaving scalar operations to be performed by the GCU main processor. Hence, the GCU must have general arithmetic capability in addition to adequate memory. Additional memory is included to permit fault detection and fault correction routines to be stored (see section 2.10). This mode of operation is discussed in section 3.9.

3.2.3. Exterior Unit Functions

Results derived from the simulation program (section 2.10) suggest the use of exterior units to the processing array that are under control of the GCU, as shown in Figure 3.2. These units are the data transfer microprocessing unit used to control data transfer

* An instruction that is equivalent to a set of one or more lower level language instructions (microinstructions).

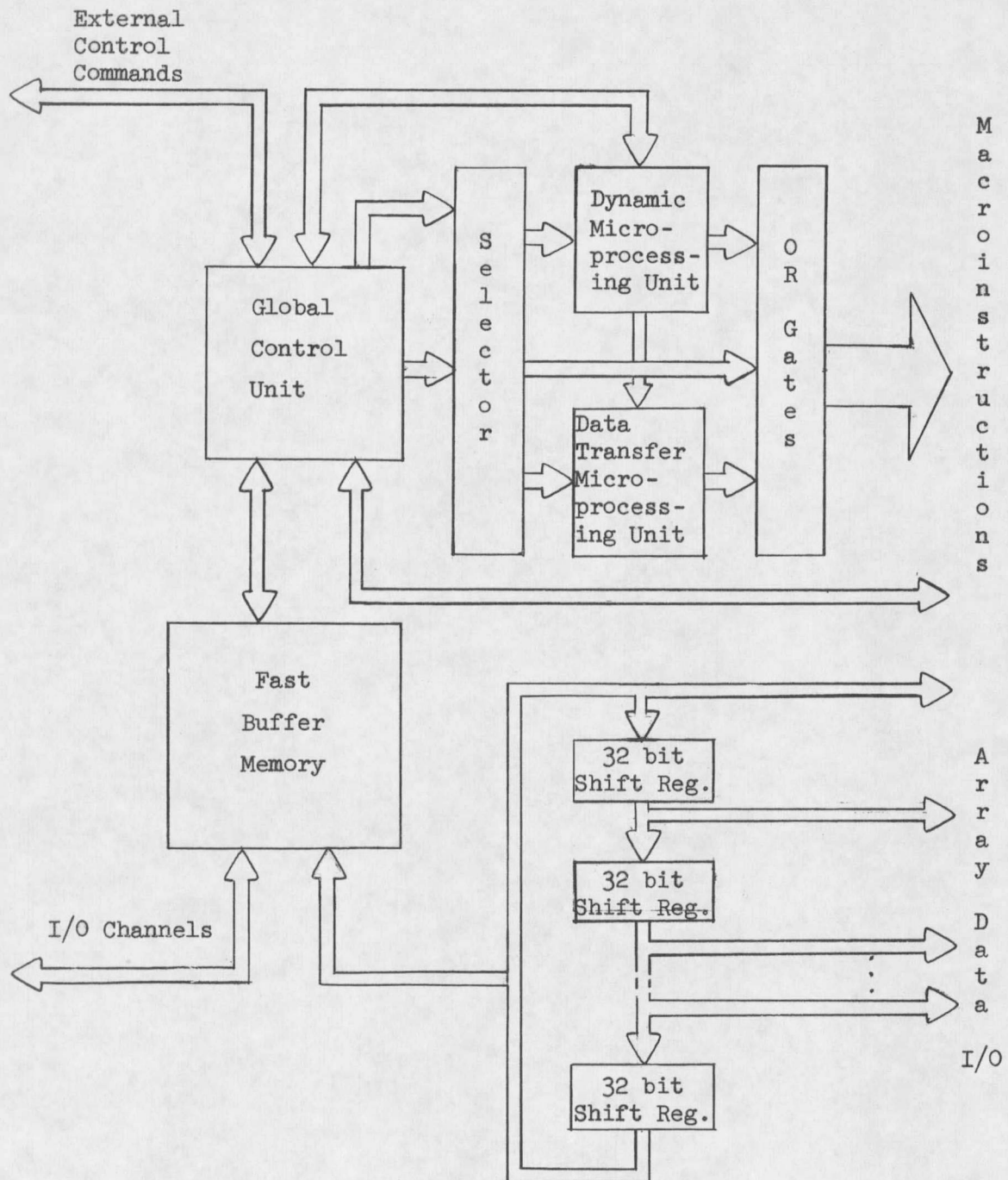


Figure 3.2 Global Control and Exterior Units

operations within the processing array and between the processing array and other units; the dynamic microprocessing unit used to facilitate execution of special instructions, i.e., matrix multiplication and inversion operations; and the input-output unit, consisting of the fast buffer memory and sixteen 32 bit parallel shift registers, used to form a high-speed data communication link between the processing array, GCU and input-output channels. An expanded description of each of these units is presented below.

As suggested by the results derived from the simulation program, all data transfer operations relating to the processing array are controlled by the data transfer microprocessing unit. This unit has access to all processing array data transfer control lines. Upon receiving a data transfer macroinstruction from the GCU, this unit assumes processing array control and executes the given instruction under control of a 6 nanosecond clock (hence, this computer design is a two-clock system), allowing any data transfer operation to be completed in a single 100 nanosecond GCU clock period. The macroinstruction contains all pertinent information to implement any one of the possible data transfer operations. At the end of the data transfer operation, processing array control is returned to the GCU.

The dynamic microprocessing unit is utilized as a temporary storage area (see section 2.10) for sets of repetitive macroinstructions (referred to as special instructions), i.e., matrix

multiplication and inversion. The storage area is segmented into memory modules that are individually addressed, and loaded during the GCU initialization sequence. Once loaded, these special instructions are callable by a single GCU instruction at any time during the execution of a given algorithm. When a special instruction is called, the GCU relinquishes processing control to the addressed dynamic microprocessor memory module. All modules in the microprocessor have access to the processing array control lines, and the data transfer microprocessor. The selected module then issues in sequence the set of macroinstructions implementing the given special instruction to the data transfer microprocessor, or to the processing array. Upon completion of issuing the set of macroinstructions, the dynamic microprocessor returns control to the GCU.

The processing array data communication section consists of the fast buffer memory, the sixteen 32 bit input-output parallel shift registers, and the GCU input-output processor (see Figure 3.2), to allow separate data processing as suggested by the results of the simulation program in Chapter 2. These units operate in coordination with the data transfer microprocessor to allow data transfer to and from the processing array. The shift registers perform the serial to parallel (parallel to serial) conversion required in this machine. Data is transferred in parallel by columns via the X command block under control of the 6 nanosecond (ns) clock. The fast buffer memory

also acts as a data buffer between the processing array; the input-output channels, and the GCU.

3.2.4 X and Y Command Block Functions

The X and Y command blocks route data and command signals from the GCU and exterior units to the processing array. The X command block is used to set the left edge condition of the data transfer busses, and is used in the fault-detection mode by routing fault signals from all of the rows of the processing array. The Y command block receives signals from each row of the processing array indicating the status of the instruction being executed. When the processing array has completed a given operation, the Y command block issues a complete signal to the GCU. Since some of the X and Y command block control lines are used by the GCU and the exterior units, additional logic (not shown) is included to OR these common control signals to the X and Y command blocks.

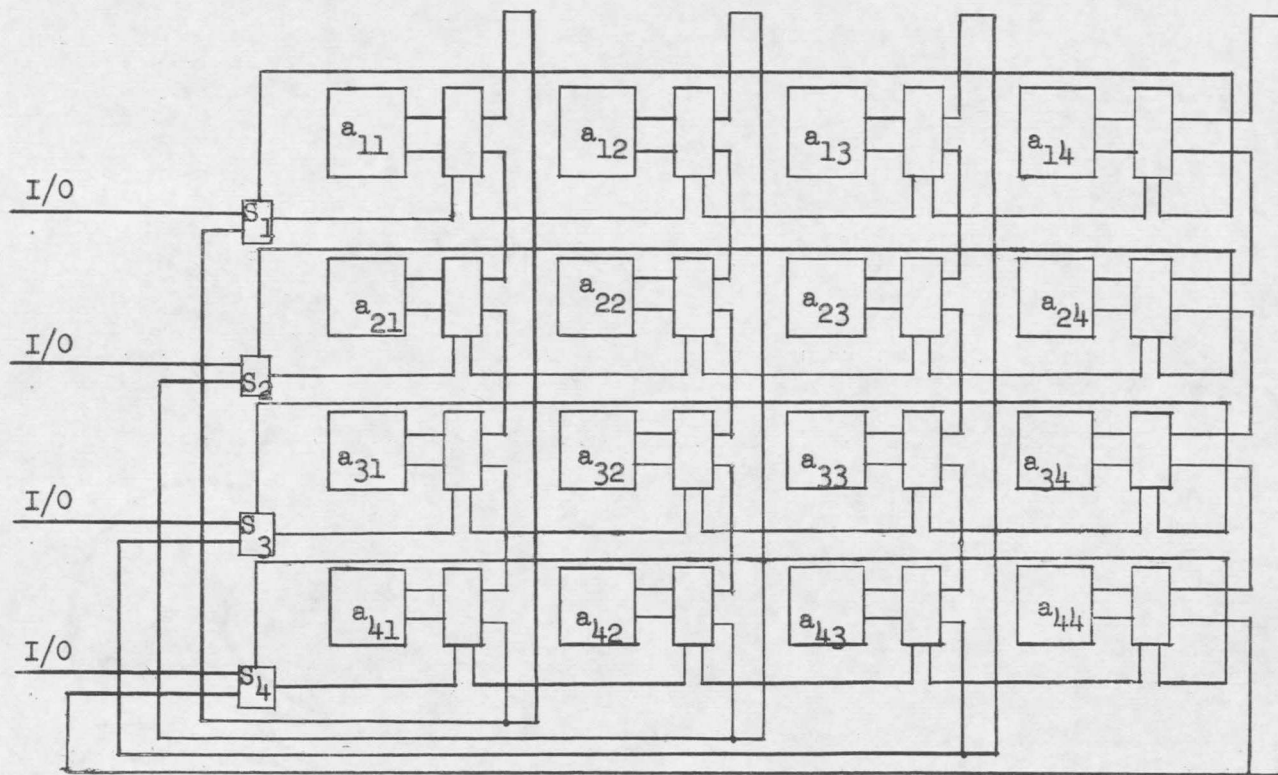
3.2.5 Processing Array Functions and Organization

As a result of the survey of array-structured computers presented in Chapter 1, the processing array of this computer consists of N^2 identical processing cells (labeled a_{ij}), which can be enabled (disabled) by rows or columns. Each processing cell is segmented into a mathematical operation block (MOB) and a data transfer block (DTB), allowing the localized cell arithmetic operations to be

separated from inter-cell and input-output data transfer operations (see section 2.10). A processing cell MOB accepts macroinstruction information (bits 0-18 of the macroinstruction) from the GCU or exterior units, and executes a corresponding set of microinstructions* to sequentially enable a set of MOB modules to execute the given instruction. All MOB's are capable of performing in parallel the operations of floating point addition, subtraction, and multiplication. Provision has been made in the design of the MOB for the addition of an auxiliary operation module, allowing the introduction of additional operations; e.g., floating point division, parity checking (see section 2.10).

The data transfer block (DTB) is used to route data from one cell location to another, and to compare two 32 bit floating point data words. The use of the DTB concept essentially eliminates the data transfer problem encountered by Cannon (21), and tends to isolate fault propagation to single cell locations. The DTB has direct access to the memory module of the processing cell with which it is associated, and to the 128 horizontal and vertical adjacent data buss segments shown in Figure 3.3. The data transfer microprocessor issues microinstructions to structure the data busses for a given

* A lower level language instruction corresponding to a set of one or more command signals to functional modules.



Note: All lines represent 32 parallel data transfer lines.

Figure 3.3 Data Buss Structure for a 4 x 4 Processing Array

operation. The microprocessor then routes data via this bussing structure composed of only enabled cells (unused cells are bypassed), using the data transfer cycle illustrated in Figure 3.4. Data is always shifted right (down), with end around carry to the left (top) of the processing array. The data transfer occurs in the form of long parallel shift registers under the control of the 6 nanosecond clock. At the end of the data transfer cycle, the data in the enabled DTB's is stored in the cell memory module by a GCU store command. Processing array control is then returned to the GCU.

3.3 Data Representation and Logic Convention

3.3.1 Data Representation

The data representation used by the array-structured computer is a two's complement floating point data format, as shown in Figure 3.5. A word of data consists of 32 binary bits; bit 0 is the sign bit, bits 1-24 are the fractional part in two's complement, and bits 25-31 are the excess 64 base 2 exponent. An example of the data representation is given below:

positive: sign bit = 0; $N = F \times 2^{E-64}$

negative: sign bit = 1; $N = -(1-F) \times 2^{E-64}$

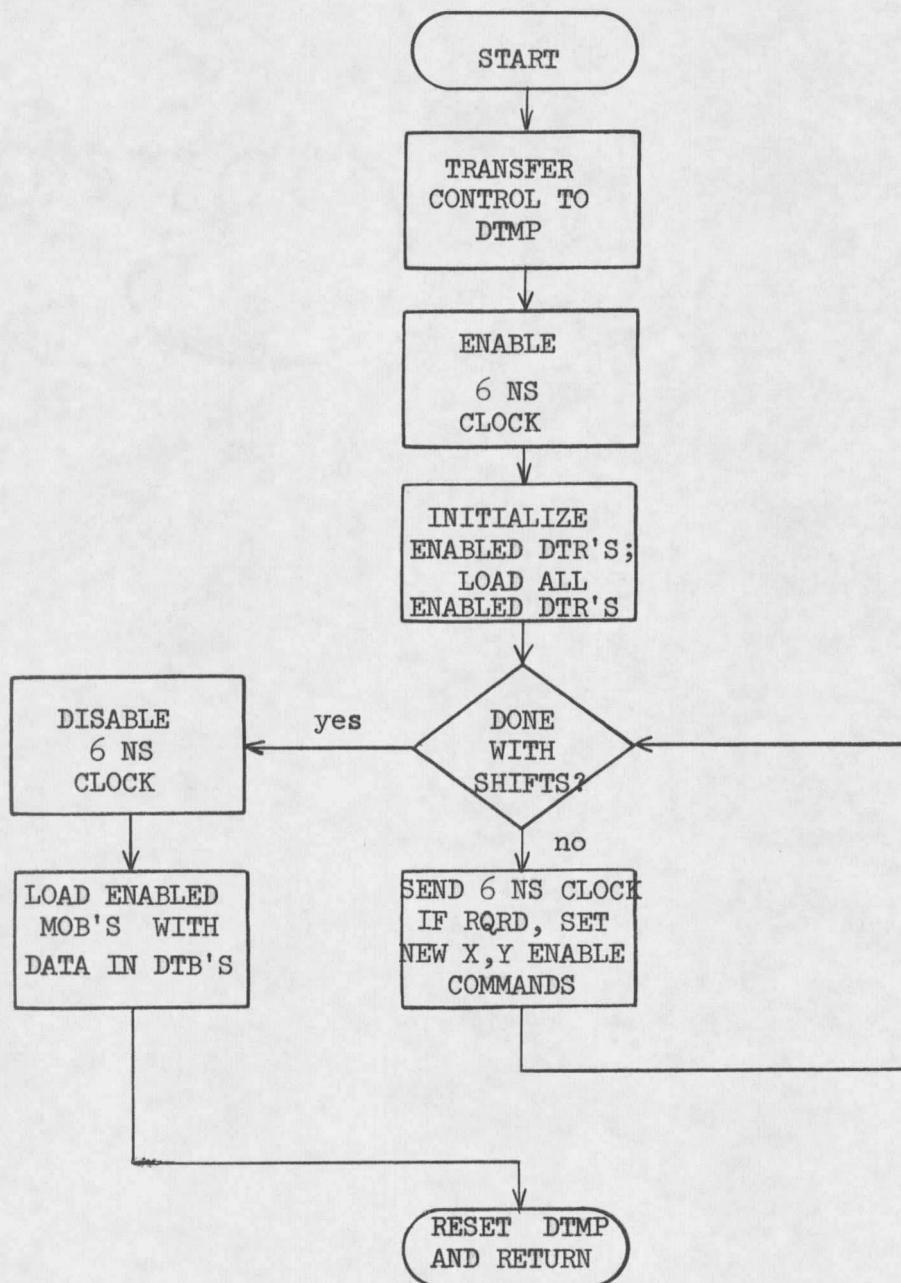


Figure 3.4 Flow Chart for the Data Transfer Cycle

The exponent is represented by the binary equivalent of the data base two exponent value plus 64. Hence, the exponent field is always a positive value (0-127), and the base two exponent of the data is in the range 2^{-64} - 2^{+64} .

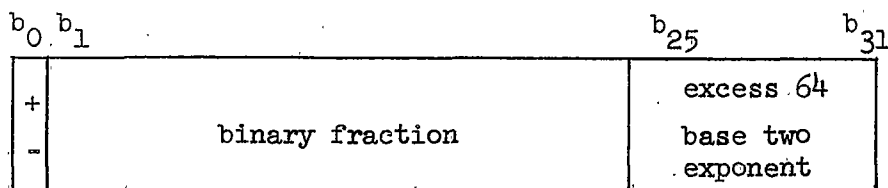


Figure 3.5 Floating Point Data Format

3.3.2 Data Normalization

All data is normalized upon input to the computer and after all mathematical operations. Normalization is defined as follows: if the sign bit is 0 and either bit 1 or bit 2 is a binary 1; if the sign bit is a 1 and either bit 1 or bit 2 is a binary 0. Hence, regardless of sign, the data will be normalized. This floating point representation makes direct comparison more difficult, but makes the logic circuit requirements for implementing this representation less difficult. Normalizing hardware is provided in each processing cell.

3.3.3 Logic Convention

The design of the array-structured computer utilizes the broad concept of cellular logic design. All functional modules comprising the processing cells are designed using the selected set of cellular logic units shown in Table 3.1 which illustrates the logic convention used in this computer design. The use of these cellular units facilitates LSI implementation of the processing cells, a subject discussed in section 3.10.

3.4 Global Control Unit

To execute the assembly language instructions, the array-structured computer uses a combination of local and central control of the processing array. Local control is confined to the processing cell level, and is discussed in detail in section 3.7.2. Overall system supervision and control is provided by the global control unit (GCU); a small to medium size, general purpose machine capable of 36 bit word, 10 Mhz effective execution rate operation. The operations that the GCU must be capable of performing are listed below:

1. Storage of system and user programs, and storage of standard tables.
2. Input-output data processing independent of the main processor execution operations.

3. Issuing and routing of macroinstruction control information to exterior units or the processing array.
4. Detection of special instructions and the transferring of the set of macroinstructions to the dynamic microprocessing unit.
5. Execution of special instruction set used in the fault-tolerant mode of operation.
6. Receiving external control information (slave mode of operation).

Due to the diversity of applications for which the array-structured computer is intended, a modular processor organization is suggested for the GCU. This organization is composed of three basic modules, as shown in Figure 3.6. These modules are the input-output processor module, the memory module, and the central processor module. With this modular structure, the GCU can perform program execution and input-output operations simultaneously and is easily expandable to satisfy large operational tasks (see section 3.9.4.3).

Table 3.1 Selected Set of Cellular Logic Units

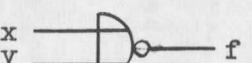
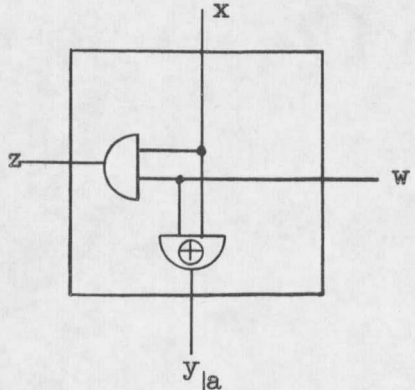
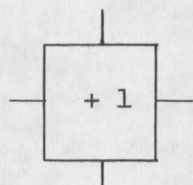
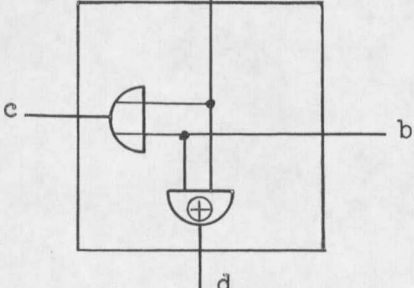
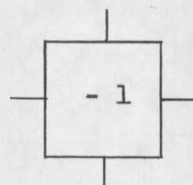
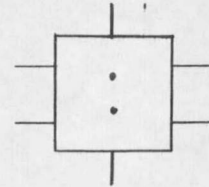
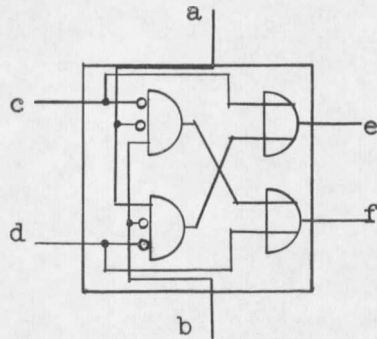
Logical Function	Circuit Schematic	Logical Symbol
AND $f = x \cdot y$		same
OR $f = x + y$		same
EXOR $f = x \oplus y$		same
Inverted Output $f = \overline{x + y}$		same
Add One Cell $z = x \cdot w$ $y = x \oplus w$		
One's - Two's Complementor Cell $c = a + b$ $d = a \oplus b$		

Table 3.1 (continued)

Comparator
Cell

$$e = \frac{c+a}{b \cdot d}$$

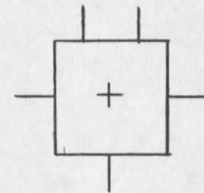
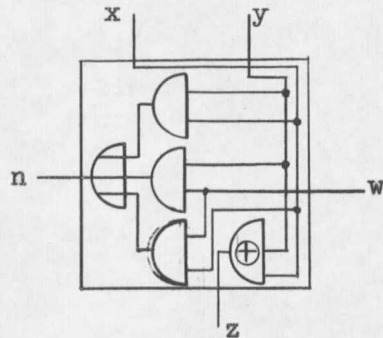
$$f = \frac{d+b \cdot a}{c}$$



Adder Cell

$$z = x \oplus y \oplus w$$

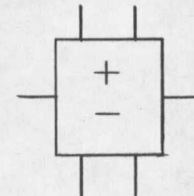
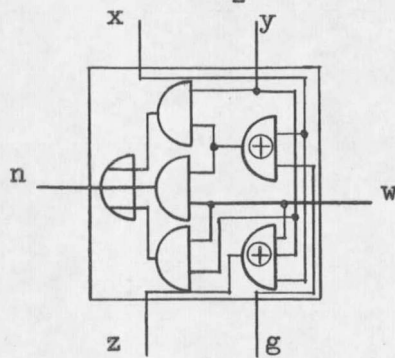
$$n = x \cdot y + y \cdot w + x \cdot w$$



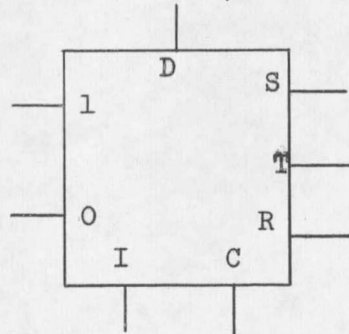
Adder-
Subtractor
Cell

$$z = x \oplus y \oplus w$$

$$n = y \cdot w + (x \oplus y)(w + y)$$



General
Register
Cell



same

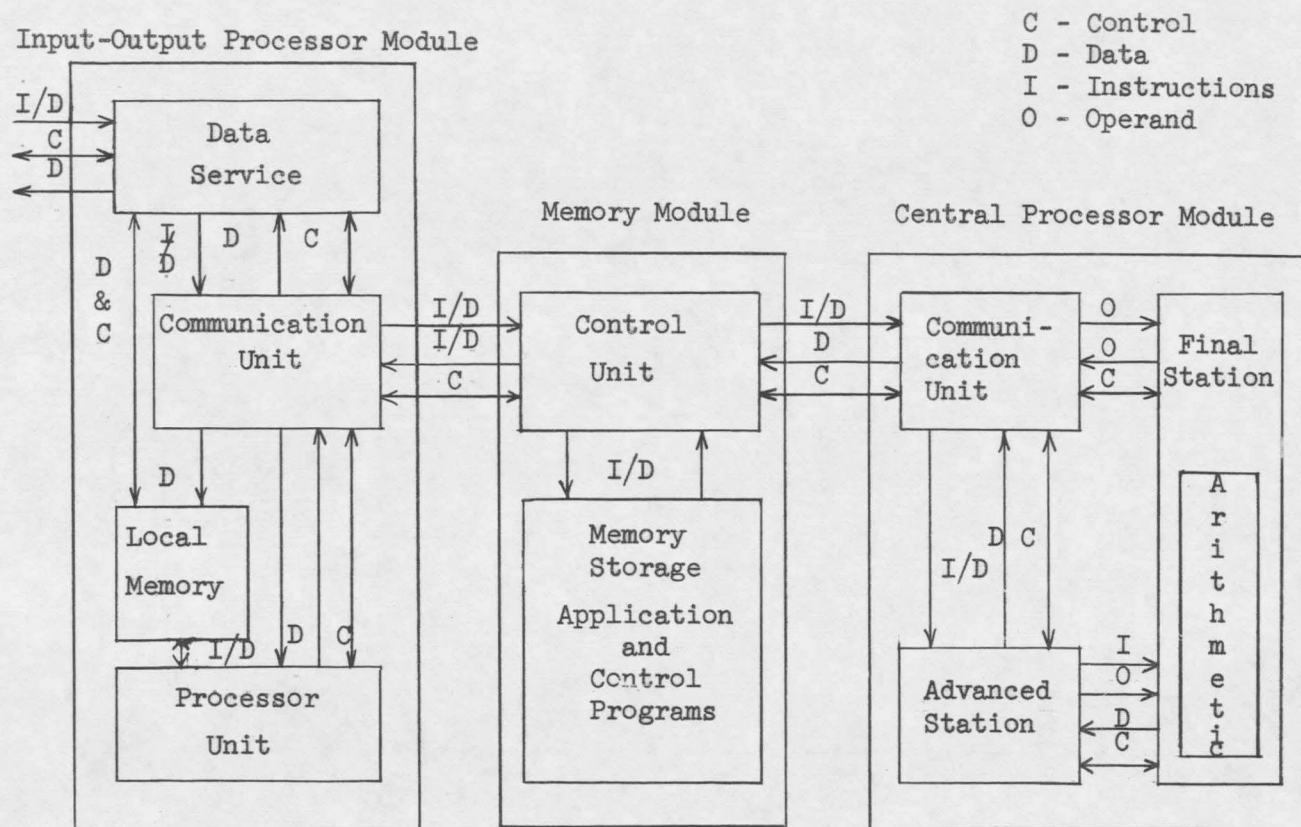


Figure 3.6 Modular Organization of the GCU

The input-output processor module is capable of performing independent input-output operation, and is programmed with its own instruction set. The local memory unit provides storage for use by the processor unit. It is divided to provide an area for each channel serviced by the input-output processor module. These areas are control locations and are used to store basic program information, control words, and termination information. Local memory also provides buffering between internal memory and the input-output devices.

The memory module is divided into two major areas; the control area and the storage area. The control area is responsible for interface with the central processor module and the input-output processor module. Within the storage area, controls exist for instruction and data fetch operations. The modular concept allows the addition of modules to the system to meet expanding memory storage requirements.

The central processor module is organized into three areas: a communication unit, an advanced station, and a final station. The communications unit performs data transfer operations with the memory and input-output processor module in addition to executing internal functions. The advanced station provides fetching via the communication unit and storing of instructions and operands in advance of their use. This look-ahead technique allows a significant reduction in effective program execution time. All arithmetic operations, logical operations, shifts, conversions, and comparisons between operands are performed by the final station.

In summary, the modular computer organization is suggested for the GCU allowing the array-structured computer to be expandable and placing emphasis on an independent input-output processor. Also, the instruction look-ahead characteristic allows fast, effective execution time.

3.5 Exterior Processing Units

3.5.1 Input-Output Unit

The transmission of data to and from the processing array is controlled by the GCU input-output processor (IOP), which simultaneously controls generation of data transfer microinstructions via the data transfer microprocessor and input-output operations of the processing array via the fast buffer memory and the 16 parallel input-output shift registers. Transmission of input-output data occurs in columns, requiring one GCU clock period (100 nanoseconds) per column. The selection of the active column location in the processing array is made by the data transfer microprocessor. Hence, an input (output) data transfer operation for a 16 x 16 array requires 16 GCU clock periods (1.6 microseconds).

The transmission of data to the array-structured computer will most likely be sequential at a high bit rate. If the input data has been processed, it is routed from the input channels through the fast buffer memory to the string of sixteen 32 bit parallel shift registers.

If data processing is required, the GCU IOP directs the data through the fast buffer memory to the GCU main processor. Once processed, the data is then directed to the shift registers via the fast buffer memory. The input-output shift registers are under direct control of the IOP, operating independently of processing array operations. The processing array input process begins with the loading of these shift registers from the fast buffer memory under the control of the IOP. Once loaded, the IOP issues a macroinstruction to the data transfer microprocessor initiating processing array loading operations. While data is being routed to and stored in the processing array cells, the shift registers will be reloading by serial shifting of data on the 32 parallel data bit lines linking the fast buffer memory output to the shift register string. All data transfer operations occur under control of the 6 nanosecond clock.

In the data output operation, the sequence of operations is reversed. The IOP issues a macroinstruction that directs selected data in parallel to the input-output shift registers. Once stored in the shift registers, this data is serially shifted on the 32 data lines from the shift registers to the fast buffer memory. While output data is being shifted to the fast buffer memory, the GCU issues another macroinstruction to route the next output data column to the input-output shift registers. The process continues until the processing array data has been transferred to the fast buffer memory.

If postprocessing is required, the data is directed to the GCU main processor, processed, and directed to the output channels via the fast buffer memory. If no postprocessing is required, the data is directed to the output channels.

3.5.2 Data Transfer Microprocessing Unit

All processing array, data transfer operations are controlled by the data transfer microprocessing unit (DTMU), shown in Figure 3.7. This unit issues microinstructions to the data transfer blocks located within the processing cells. These blocks and their operation are referred to in this section and are discussed in section 3.8. The DTMU consists of a data transfer clock (6 nanosecond period), a two-stage initialization counter, a four-stage down counter, a decoder, associated control and enable logic, and pass through control lines. The DTMU receives a macroinstruction and enable command from the GCU when a data transfer operation is to be performed. The format of the macroinstruction is shown in Figure 3.8. A binary 1 is always present in bit location 0 of data transfer macroinstructions. Bits 1-4 of the macroinstruction correspond to the array designation (1 of 16 array locations within the processing array). Bits 5 and 6 set the left edge condition of the processing array. Bits 7 and 8 indicate the type of data transfer operation to be performed. Bits 9-12 indicate the number of data shifts to be performed. Bits 13-15

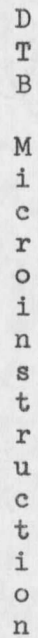


Figure 3.7 Data Transfer Microprocessing Unit

indicate the need for special shift control commands for skew operations. Bits 16-31 are used by the DTMU to dimension the processing array to the data transfer operation to be performed. In increasing order these bits indicate the enabling of rows (columns) of the processing array; a binary 0 indicates an enabled row (column), a binary 1 indicates a disabled row (column).

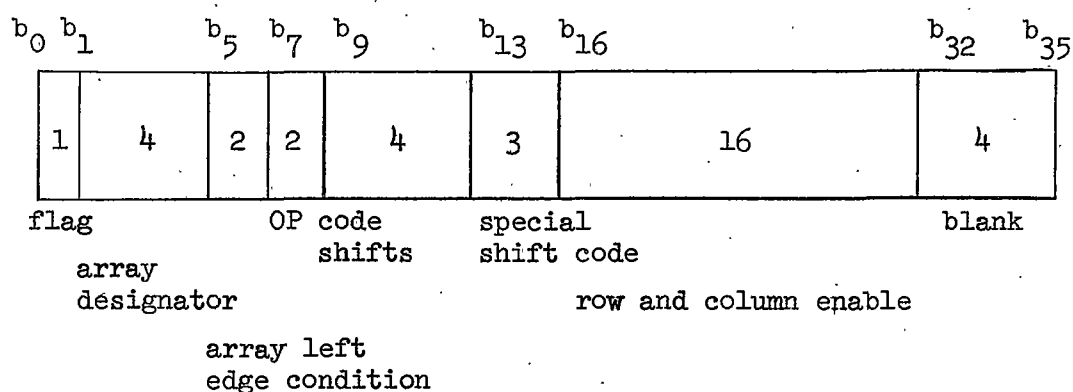


Figure 3.8 Data Transfer Macroinstruction Format

The data transfer operation begins with issuance of a data transfer macroinstruction and enable command from the GCU to the data transfer microprocessing unit. The enable command sets the enable register (gr_1), which in turn enables the data transfer clock. The two stage counter (gr_1, gr_7) is used to allow initialization of the DTMU. Once the data transfer clock is enabled, the two stage counter begins to count the data transfer clock pulses. The first count enables gate g_5 , allowing the loading of the four stage down counter

with macroinstruction bits 1-4. In addition, upon the first count, the DTMU issues a select and route data command to all enabled processing cells, allowing the array data to be transferred to the data transfer blocks within the processing cells. The second count results in a DTMU command (via gate g_7) that loads the selected data in shift registers A and B of all enabled DTB's. In all data transfer operations both registers are loaded regardless of whether both registers are used in the data transfer operation. The third count disables the two stage counter, enables the clock line of the down counter (via gate g_4), and routes bit 13 of the macroinstruction to the DTB's. With the presence of bit 13 and the enable information, the array data busses are determined and the dimension of the data transfer operation defined (unused processing cells bypassed). The down counter then counts down under control of the 6 nanosecond clock until a four bit binary 0 is reached. Each of these counts results in a shift command to the processing array to the right (down). When the 0 count is detected on the four stage down counter (by gate g_{13}), the data transfer clock is disabled by gate g_{11} and the data is routed to the memory module of the respective processing cell. A GCU store command (store) is then sent to all processing cells resulting in the storage of the array data of all enabled cells in array memory (1 of 16 possible array locations). After the array data has been stored, the GCU resets the data transfer microprocessing unit and returns to program execution.

The matrix skewing, data transfer operations require a special set of microinstructions. These operations require that rows (columns) of the processing array execute a different number of shifts depending on their location within the array. In these operations all control information is identical to that discussed above, with the exception of the row (column) enable commands which are selectively set to 1 (disabled) after the correct number of shifts have occurred for that row (column). Bits 13-15 of the data transfer macroinstruction contain information on the skew operation to be performed. Initially, all row (column) enable information is used to enable the entire dimensioned array. As the down counter changes count, the decoder selectively sets registers gr_9 - gr_{24} . The setting of these registers disables the correct row (column) after the correct number of shifts. Hence, in a matrix skew right, data transfer operation, row one remains unshifted, row two is shifted right one place, row three is shifted right two places, etc. Depending on whether the matrix skew operation is to the right (down) or left (up), the decoder accepts the complement of the down counter output or the uncomplemented output, respectively. This requirement on the control logic is a result of the processing array's ability to shift data in one direction only.

3.5.3 Dynamic Microprocessing Unit

In the processing of matrix-oriented algorithms, special instructions are encountered that are composed of sets of assembly language instructions. These special instructions usually contain repetitive operations and the number of assembly language instructions required to implement such special instructions depends on the dimension of the algorithm being executed. To relieve the GCU from processing and storing of such instruction sets, a dynamic microprocessing unit (DMU) is provided as an exterior unit. This unit, shown in Figure 3.9, is composed of a number of memory modules, decoding logic, a seven state counter, enable logic, and end flag detection logic. The DMU has access to the data transfer microprocessing unit and all processing array control lines related to localized array operations.

In the initialization process of the array-structured computer for a given algorithm, the GCU executive program detects existing special instructions and creates sets of macroinstructions implementing the required operations. These dimensioned sets of macroinstructions consist of mathematical operation macroinstructions, data transfer macroinstructions, and an end of instruction set flag. The GCU then initiates the process to store these sets of macroinstructions in the memory modules of the DMU; through the selective enabling of the memory modules and utilizing the seven stage counter (through gate g_1) to decode memory module word locations. The format for data

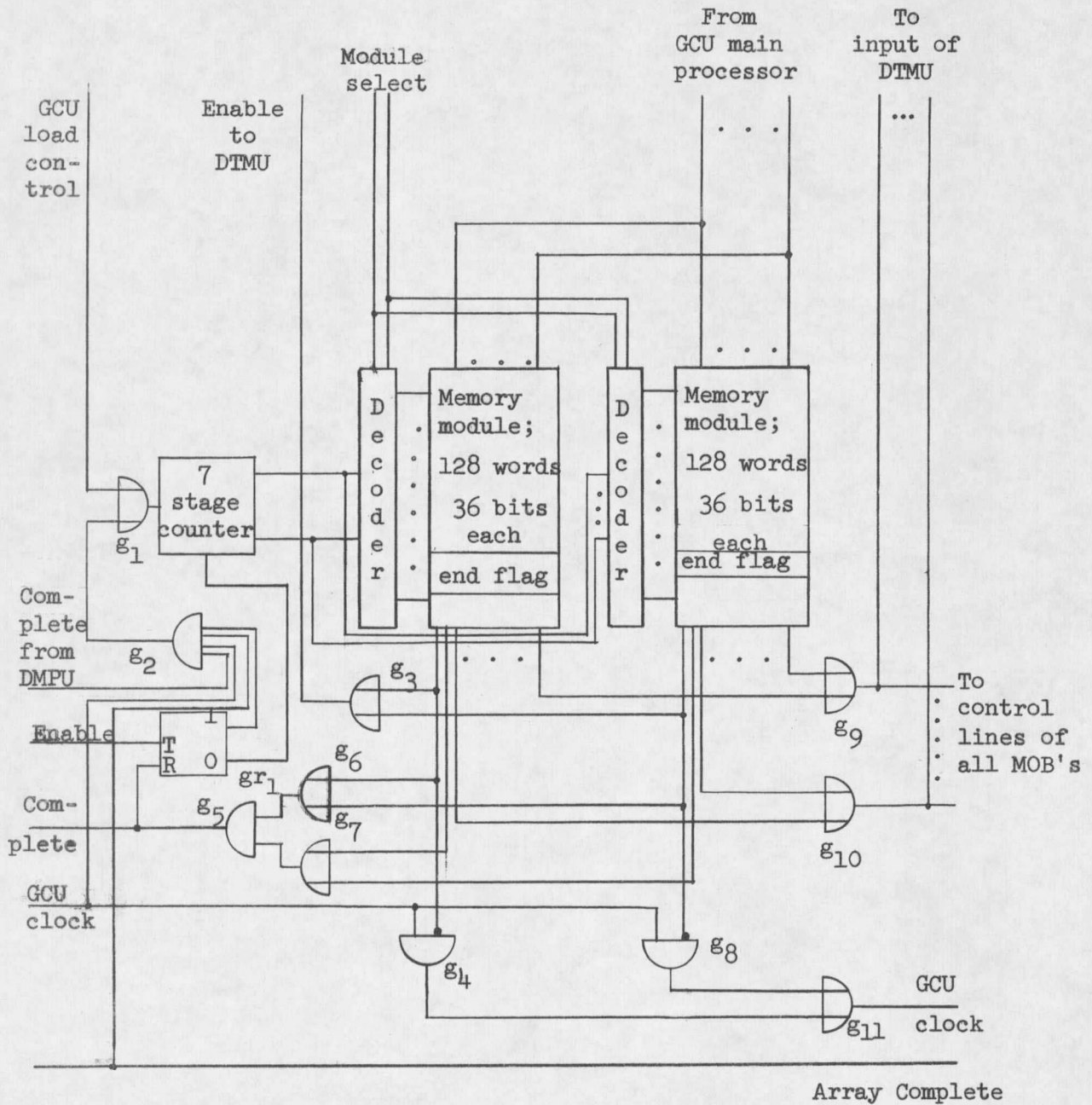


Figure 3.9 Dynamic Microprocessing Unit

transfer operations is shown in Figure 3.8 (bit location 0 always contains a 1), with the format for mathematical operations shown in Figure 3.10.

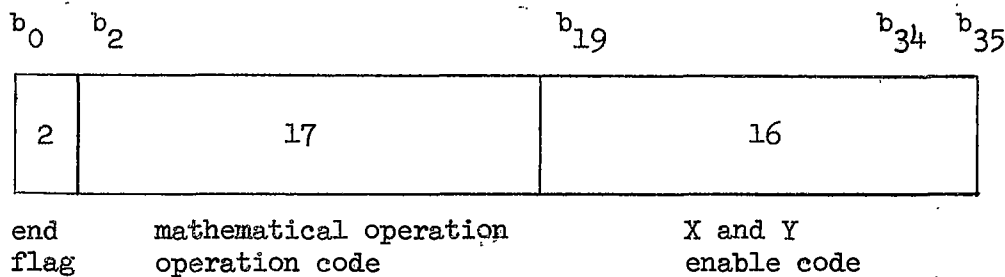


Figure 3.10 Mathematical Operation Macroinstruction Format

These sets of macroinstructions are dynamic in that they are alterable at any time by the GCU. During the processing of matrix-oriented algorithms, these sets of macroinstructions are callable by the GCU, which relinquishes system control to the DMU until the end of instruction set flag is detected. When the end flag is detected, system control is returned to the GCU and the DMU is reset.

When the GCU detects the presence of a special instruction, an enable signal is sent to the DMU along with memory module enable information. The enable register (gr_1) is set, allowing the 100 nanosecond GCU clock pulses to be routed to the seven stage counter

through gates g_1 and g_2 . The counter then provides binary count information to the enabled memory module decoder. The selected memory module sequentially transmits its stored macroinstructions to either the processing array or to the data transfer microprocessing unit. After issuing a macroinstruction to either of these locations, the enable gate (g_2) is disabled until an operation complete signal is received from the processing array or the data transfer microprocessing unit. Upon receiving an operation complete signal, gate g_2 is again enabled allowing the GCU clock pulses to be directed to the seven stage counter. When a data transfer macroinstruction is detected (0 in bit location 1), the GCU clock pulse to the processing array is disabled by gates g_4 and g_8 . The data transfer macroinstruction is then sent to the data transfer microprocessing unit. The DMU is disabled until the data transfer operation is completed at which time an operation complete signal is sent to the DMU (gate g_2). The process of issuing macroinstructions continues until an end of instruction set flag is detected by the flag detection logic (g_5 , g_6 , and g_7). The end flag consists of a 0 in bit location 0, and a 1 in bit location 1 of the stored macroinstruction. When the end flag is detected, the enable register (g_1) is reset and a special instruction complete signal sent to the GCU.

3.6 X and Y Command Blocks

The X and Y command blocks perform interface operations between the GCU, the exterior units, and the processing array. The macroinstruction information supplied to the input control lines is distributed in parallel to each of the processing array cells. Control information is distributed by rows and columns to these cells. The complexity of the control line distribution is illustrated in Figure 3.11 for a 4×4 processing array. The total number of control lines depends on the dimension N of the processing array and is computed by the equation given below:

$$\text{Number of Control Lines} = 3N + 42 \quad (3.1)$$

In addition, macroinstruction complete and fault-detection signals are routed via these blocks to the GCU.

The X command block is shown in Figure 3.12. Control signals to the block consist of row enable commands which define the active rows of the processing array (one line per row), and data source selection signals that select the data source for the left edge of the row bussing structure. The selection of the data source is implemented by the switching modules S_i , one of which is shown in Figure 3.13, and according to the code shown in Table 3.2. Gates g_1 - g_{32} allow data to be routed from the array to the GCU input-output units. Gates g_{33} - g_{64} route data from the right edge of each row data buss to the left edge, forming a closed loop data transfer structure.

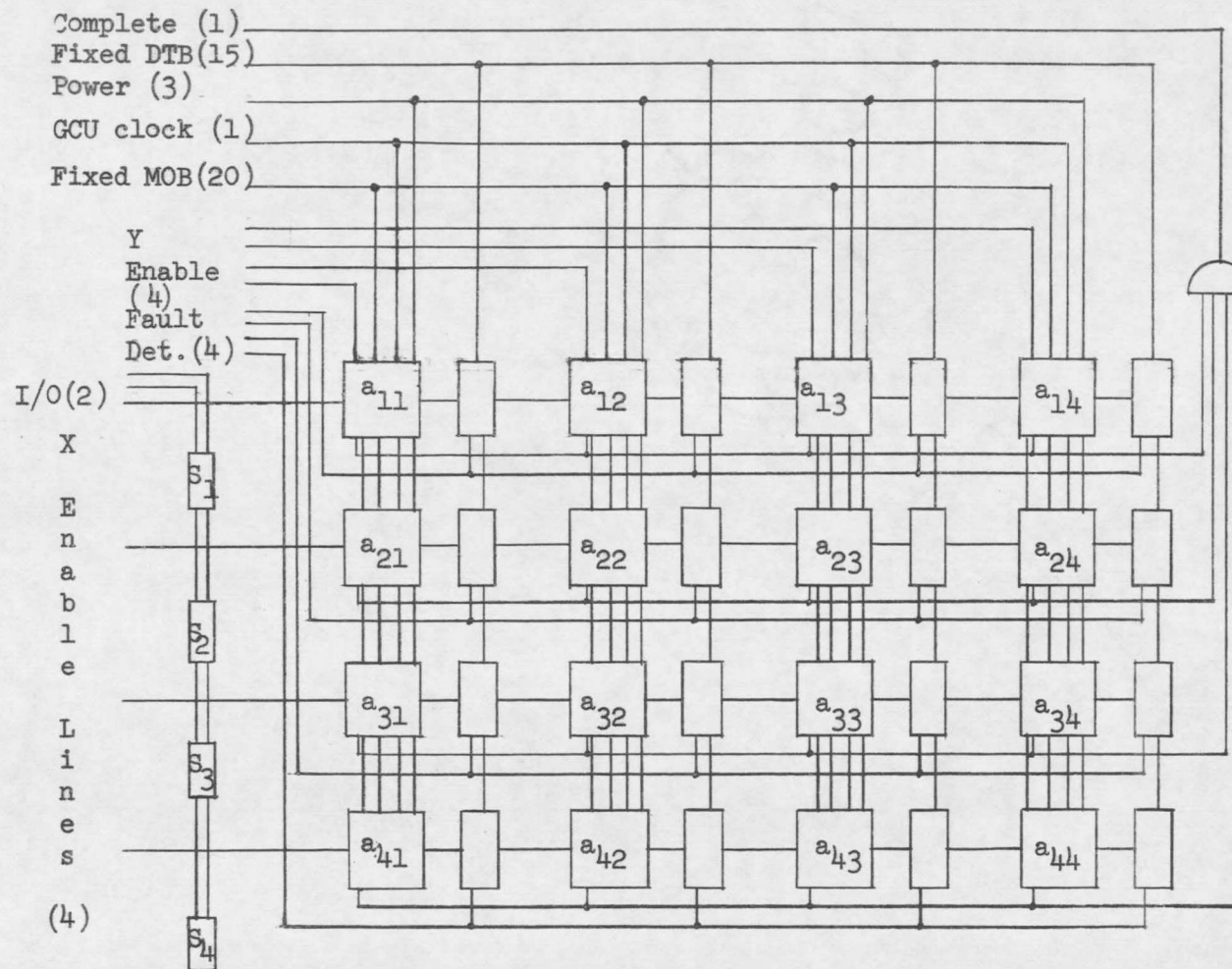


Figure 3.11 Control Lines for a 4 x 4 Processing Array

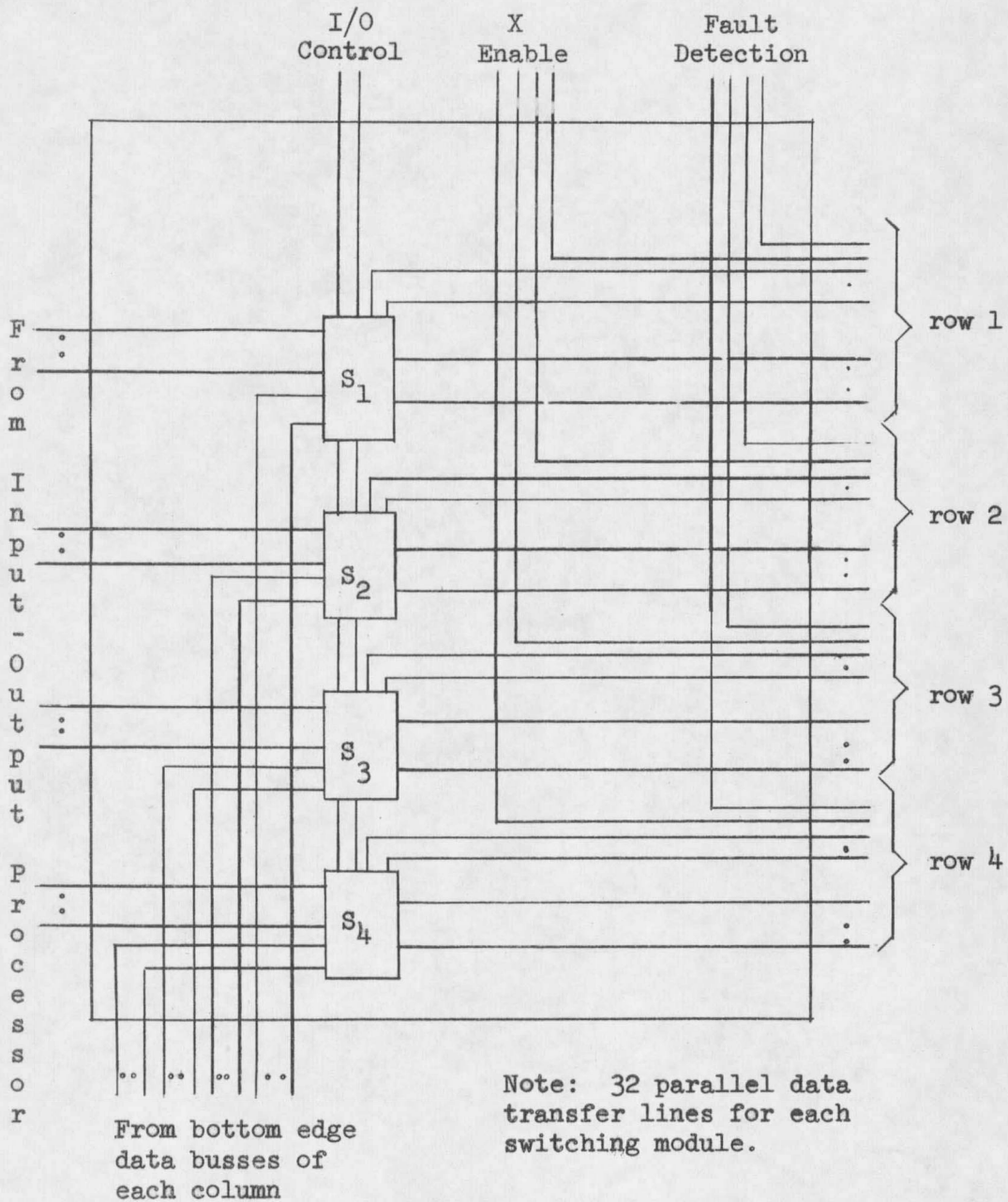


Figure 3.12 X Command Block for a 4x4 Processing Array

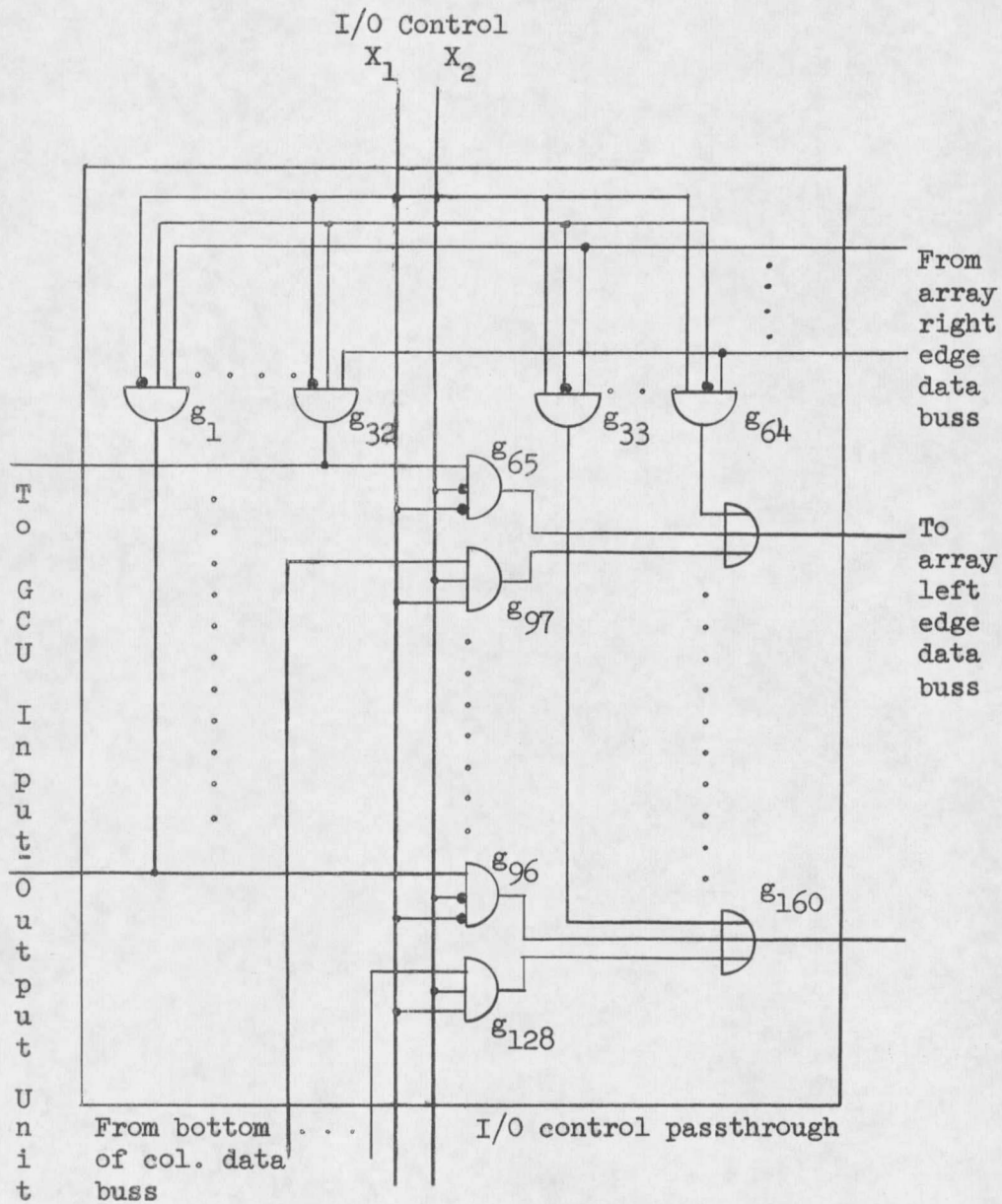


Figure 3.13 Switching Module for the X Command Block

Gates g_{65} - g_{96} allow data to be routed from the input-output units to left edge data busses. The remaining gates (g_{97} - g_{128}) are used only in the transpose operation, allowing data from the bottom of each column of the processing array to be routed into the left edge of the corresponding row of the array.

Table 3.2 X Command Block Data Source Selection Code

X_1	X_2	Data Source and Destination
0	0	right edge data buss to left edge data buss (end around transfer)
0	1	right edge data buss to output lines
1	0	input lines to left edge data buss
1	1	bottom edge data buss to left edge data buss

In the fault-detection mode of operation, each processing cell determines whether a fault exists by direct comparison of computed data to known benchmark data. This comparison is accomplished in parallel with all cells participating in the operation. If a fault exists in any of the cell locations, the fault signal is ORed by rows to the X command block which in turn communicates the existence and location of the row containing the fault to the GCU.

The Y command block is shown in Figure 3.14, and performs only pass-through routing of command signals and a processing array operation complete signal. Included in the block are the Y enable commands, which enable cells of the processing array by columns; fixed MOB command lines; fixed DTB commands; an array operation complete signal; the GCU clock signal; and power lines.

3.7 Mathematical Operation Block

3.7.1 Introduction

Processing array operations which are localized at the processing cell level are implemented by the mathematical operation block (MOB). The MOB forms the first segment of a processing array cell, allowing localized array operations to be separated from the inter-cell operations of data transfer. The second segment of the processing cell is the data transfer block (DTB) which is discussed in section 3.8. The MOB stores all elemental array data and performs mathematical operations and intracell routing of data. The GCU either directly or through the dynamic microprocessing unit issues macroinstructions to the MOB initiating a sequence of localized microinstructions. These microinstructions sequentially enable individual modules within the MOB to execute the macroinstruction. All enabled processing cells in the processing array execute in parallel the same macroinstruction. As processing cells complete the issued macro-

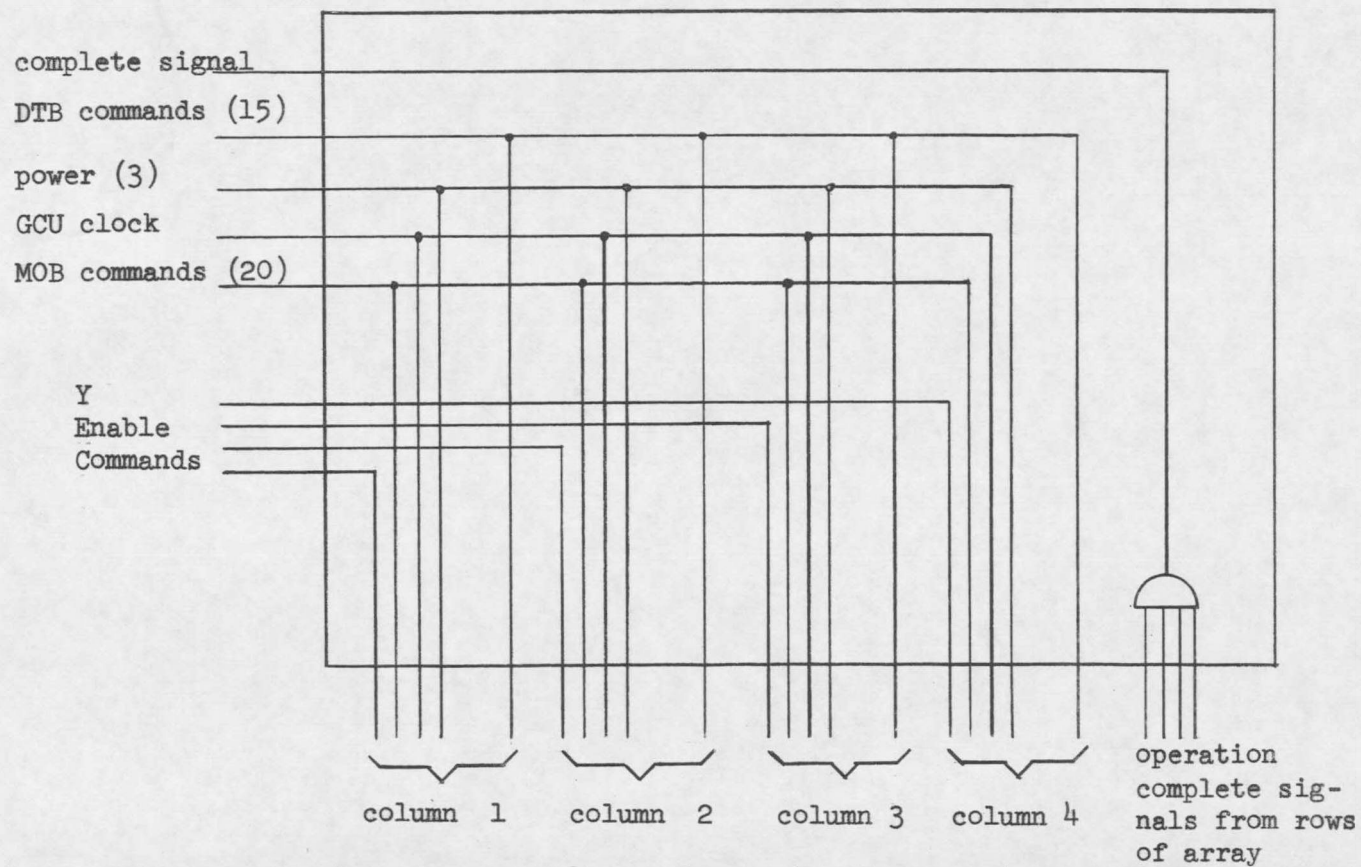


Figure 3.14 Y Command Block for a 4 x 4 Processing Array

instruction, they send complete signals to the GCU, which does not issue a new macroinstruction until all processing cells have executed the previously issued macroinstruction.

A processing cell MOB, shown in Figure 3.15, consists of a microinstruction generation module, sixteen words of 32 bits each in a memory module, a parallel adder-subtractor module, a parallel carry-save multiplier module, an auxiliary operation module, and a normalizer and data link module. Each processing cell MOB has a total of 62 input lines: 32 parallel bit data lines linking the MOB to the DTB; four DTB memory address lines; three power lines; one data source selection command line from the DTB; 19 control information lines; two row-column cell enable lines; and one operation complete line. Each processing cell can communicate with the GCU main processor and has access to external data through the data transfer block.

When executing a localized array operation, the GCU issues a GCU macroinstruction to the processing array. The macroinstruction format is shown in Figure 3.16 with the bit designation defined in Table 3.3. The macroinstruction is composed of two parts; control information and cell enable information. All enabled processing cells receive in parallel the control information (bits 0-18) which defines the memory location of the resulting data, the memory locations of operands A and B, and an operation code indicating which of the MOB internal modules are to be used in executing the array

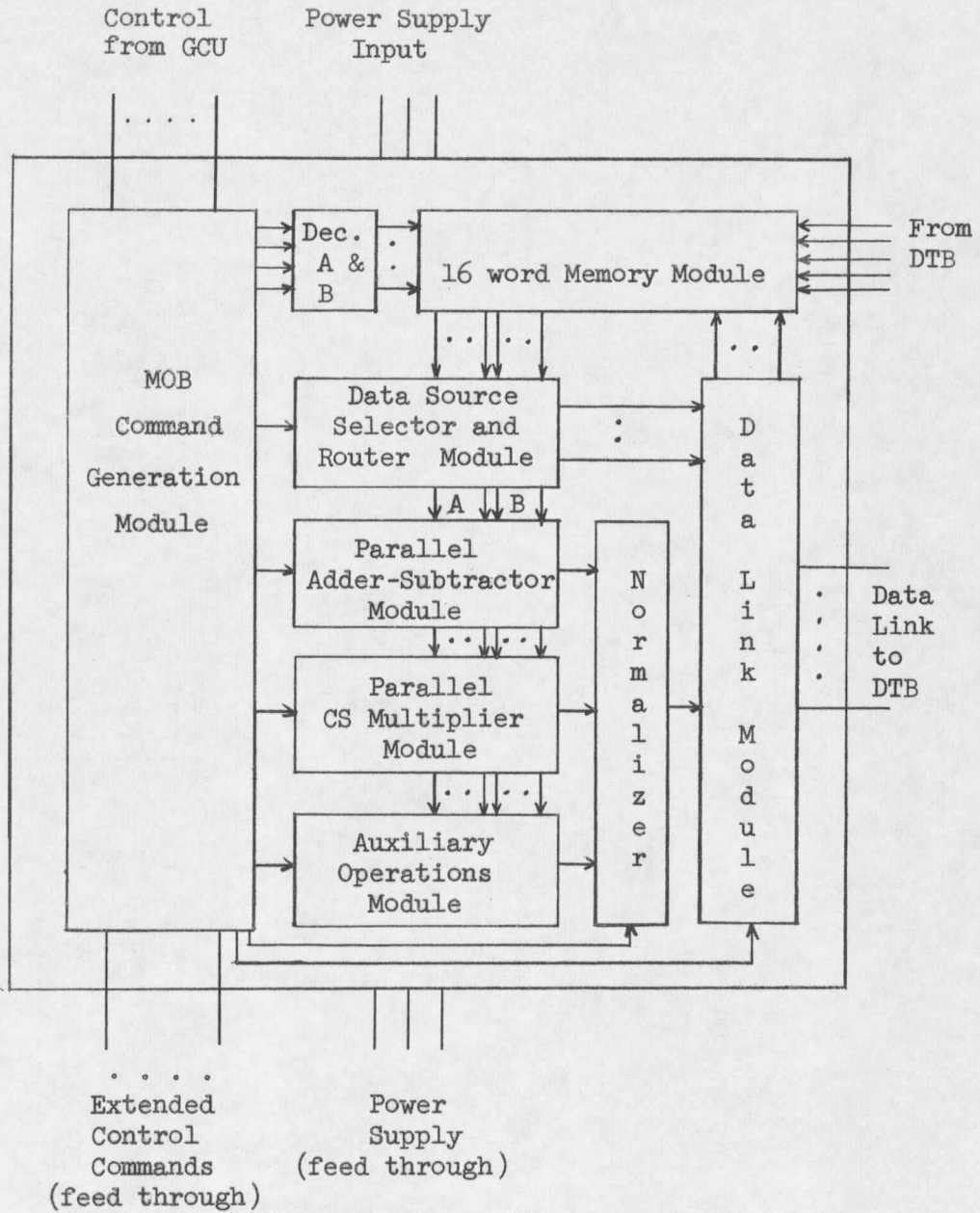


Figure 3.15 Mathematical Operation Block

operation. The second part of the macroinstruction (bits 19-35) is distributed by rows and columns to the processing array (bit 19 is distributed to row 1 and column 1, bit 20 is distributed to row 2 and column 2, etc). In this manner, the processing array is dimensioned to implement the given array operation by localized cell operation.

Each of the modules within the processing cell MOB is discussed in the following sections.

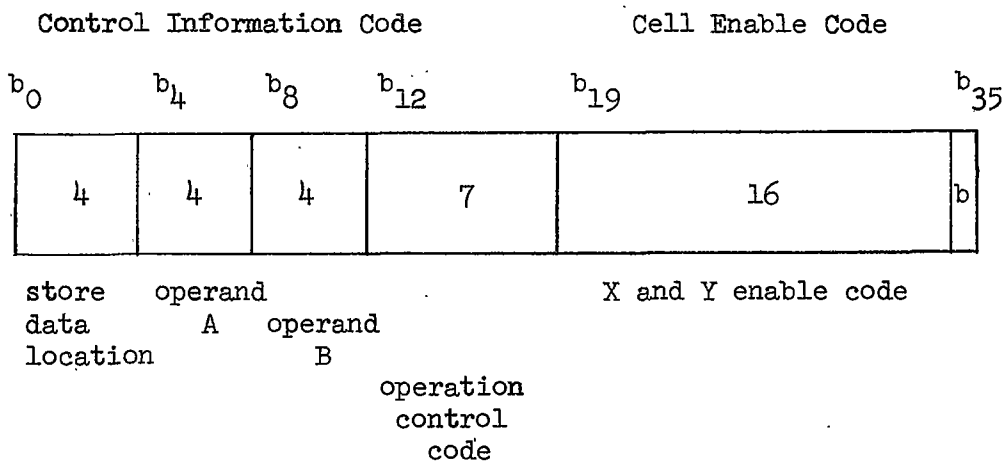


Figure 3.16 MOB Macroinstruction Format

Table 3.3 MOB Macroinstruction Bit Designation

Bit location	Level	Operation
0 - 3	1/0	defines data store location of result
4 - 7	1/0	defines operand B
8 - 11	1/0	defines operand C
12	1	data source selection and routing operation is enabled
	0	above is disabled
13	1/0	defines data sources and where data is routed
14	1	addition or subtraction operation is enabled
	0	above is disabled
15	1	subtraction operation
	0	addition operation
16	1	multiplication operation is enabled
	0	above is disabled
17	1	normalization and input-output operation is enabled
	0	above is disabled
18	1	store data in memory operation
	0	no data store operation
19 - 34	0	row (column) disable
	1	row (column) enable
35		no operation

3.7.2 MOB Command Generation Module

All localized processing cell operations are controlled by the MOB command generation module shown in Figure 3.17. This module is slaved to the GCU and performs the localized control functions discussed in section 1.7. This module consists of two storage registers, a master control section, and microinstruction generation sections that issue commands to the functional modules within the processing cell MOB. All processing cells receive the same GCU control information performing in parallel identical operations on their corresponding array elements. Initially, the control information is stored in the operand registers (three 4 bit address codes), and the control register (seven bit control code). The master control section, in conjunction with the sequential router, then sequentially route the seven bits stored in the control register (macroinstruction bits 12-18) to the microinstruction generation sections. If macroinstruction bits 12, 14, 16, and 17 are binary 0, the operation associated with that bit location (see Table 3.3) is bypassed, and the corresponding MOB functional module is not used in the sequence of operations executing the given macroinstruction. Such a bypass always occurs since addition (macroinstruction bit 14) and multiplication (macroinstruction bit 16) are never enabled in the same macroinstruction. This generation section has the ability to bypass the floating point alignment operation, and both the parallel adder-

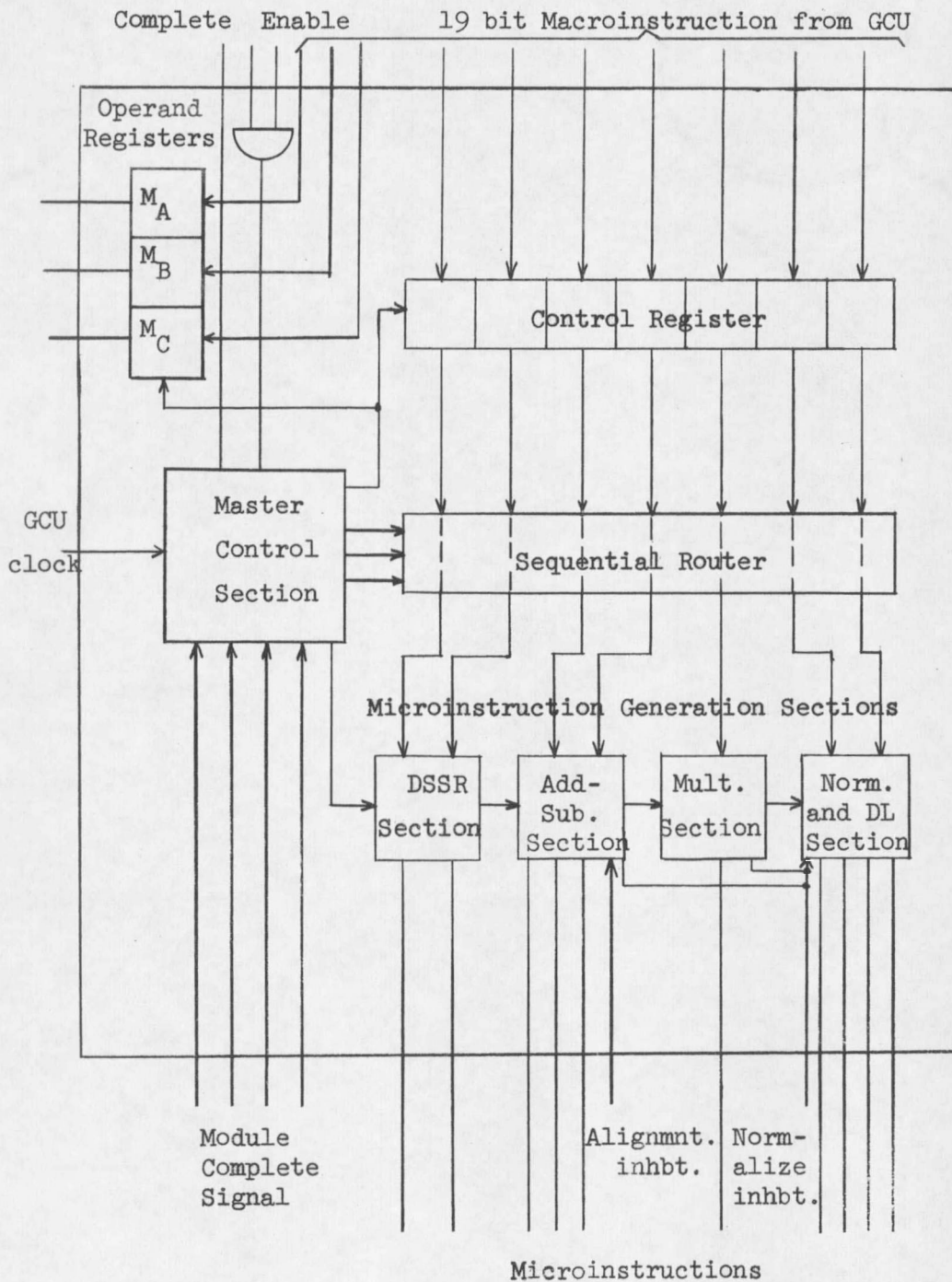


Figure 3.17 MOB Command Generation Module

subtractor and the parallel multiplication generation sections have the ability to bypass the normalization operation. At the end of command generation cycle, an operation complete signal is sent to the GCU and the MOB command generation module is reset.

3.7.2.1 Master Control and Sequential Router Section

The generation of all localized processing cell micro-instructions is controlled by the master control section and sequential router shown in Figure 3.18. These units selectively route control information stored in the control register to the correct micro-instruction generation sections. The master control unit consists of a counter, counter enabling logic, and bypass logic, and is slaved to the 100 nanosecond GCU clock.

The master control section receives a row and column enable signal from the issued macroinstruction and immediately sets the cell enable register gr_4 . The toggle input gate g_9 then passes a single pulse through to the counter (gr_1, gr_2, gr_3) input terminal. Gate g_3 allows the initial count from the reset state to pass. The counter then sets the macroinstruction registers to store the control information on the first count. The counter then continues to count every time the toggle gate (g_9) is enabled from an end of microinstruction generation signal, or a router bypass signal. When the sixth pulse is counted, an operation complete signal is sent to the GCU, and the master control section is reset.

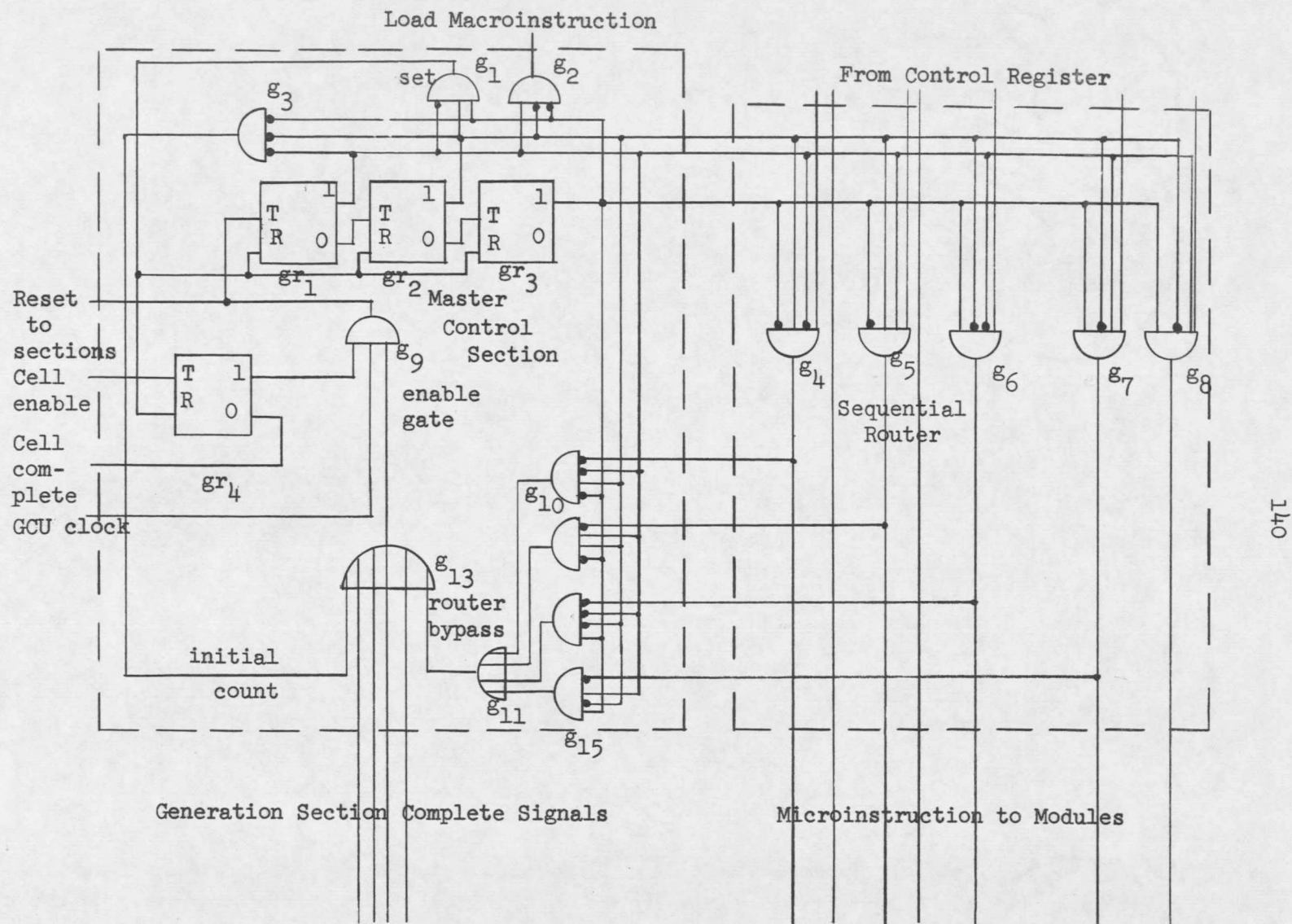


Figure 3.18 Master Control and Sequential Router Sections

The sequence of microinstructions generated in the MOB command generation module is summarized in Table 3.4. Note that cycles 3 and 4 never occur in the same macroinstruction execution.

Table 3.4 Sequence of Microinstruction Generation

Execution time	Cycle period	Operation accomplished by microinstruction
1T	1	set macroinstruction into registers
1T	2	select data source and route data
2T	3	add or subtract
3T	4	multiply
2T	5	normalize result and perform data linking

Any of the above operations (except the set macroinstruction operation) can be inhibited by a 0 in the correct bit location; depending on the bit values of macroinstruction bit locations 12, 14, 16, 17, and 18, shown in Table 3.3.

The sequential router accepts the MC counter output and sequentially routes the macroinstruction control bits to the microinstruction generation sections. Gates g_4 , g_5 , g_6 , g_{7-8} are sequentially enabled by the counter outputs of binary 2,3,4 and 5, respectively. If at each cycle the macroinstruction value is a 1, the microinstruction generation section is enabled. If a 0 is detected in any

of these bit locations, an operation bypass signal is generated.

The bypass condition is detected by the master control section (gates g_{10} , g_{12} , g_{14} , g_{16}) and the counter enabled. The fifth count is used to store data resulting from the localized array operation in cell memory by the enabling of gates g_7 and g_8 (if macroinstruction bit 18 is a 1).

3.7.2.2 Microinstruction Generation Sections

3.7.2.2.1 Data Source Selector and Router Generation Section

The generation of microinstructions for the data source selector and router module is accomplished by the generation section shown in Figure 3.19. This operation requires one GCU clock period, beginning with an enable signal from the sequential router (if macroinstruction bit 12 is a 1). With this enable signal the input gate (g_{24}) is set. The next clock pulse triggers the counter (gr_1), which disables the input gate and sends a complete signal to the master control unit. The output of the counter enables gates g_1 , g_4 , g_7 , g_{10} , g_{13} , g_{16} , g_{19} , and g_{22} . These gates pass the four bit operand codes (macroinstruction bits 4-11) to the cell memory decoders A and B. With this method two data words in cell memory can be selected simultaneously and placed on processing cell internal data busses A and B. The counter enables gate g_{27} allowing bit 13 of the macroinstruction to select the data source for buss B (memory or DTB). The data remains on these busses

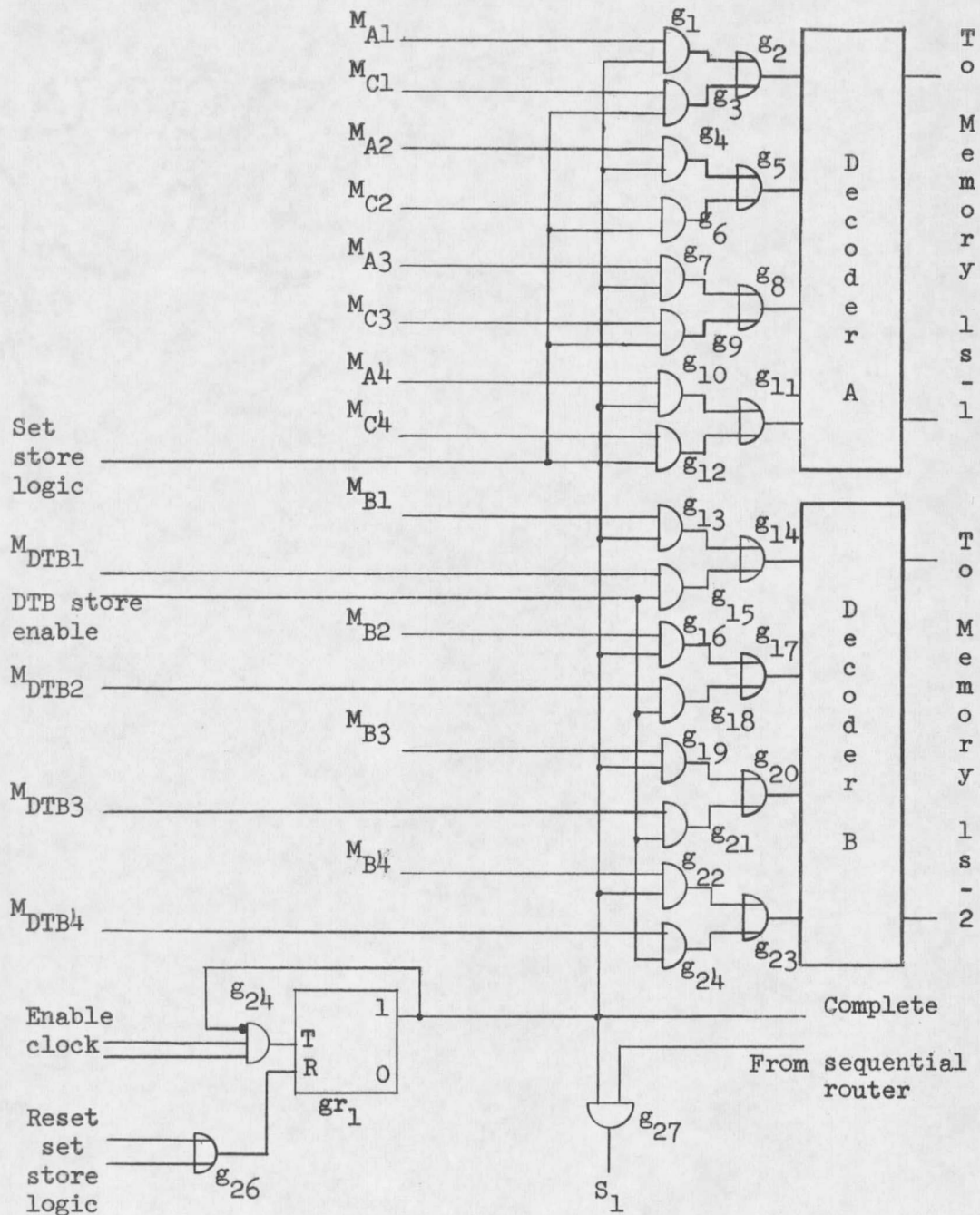


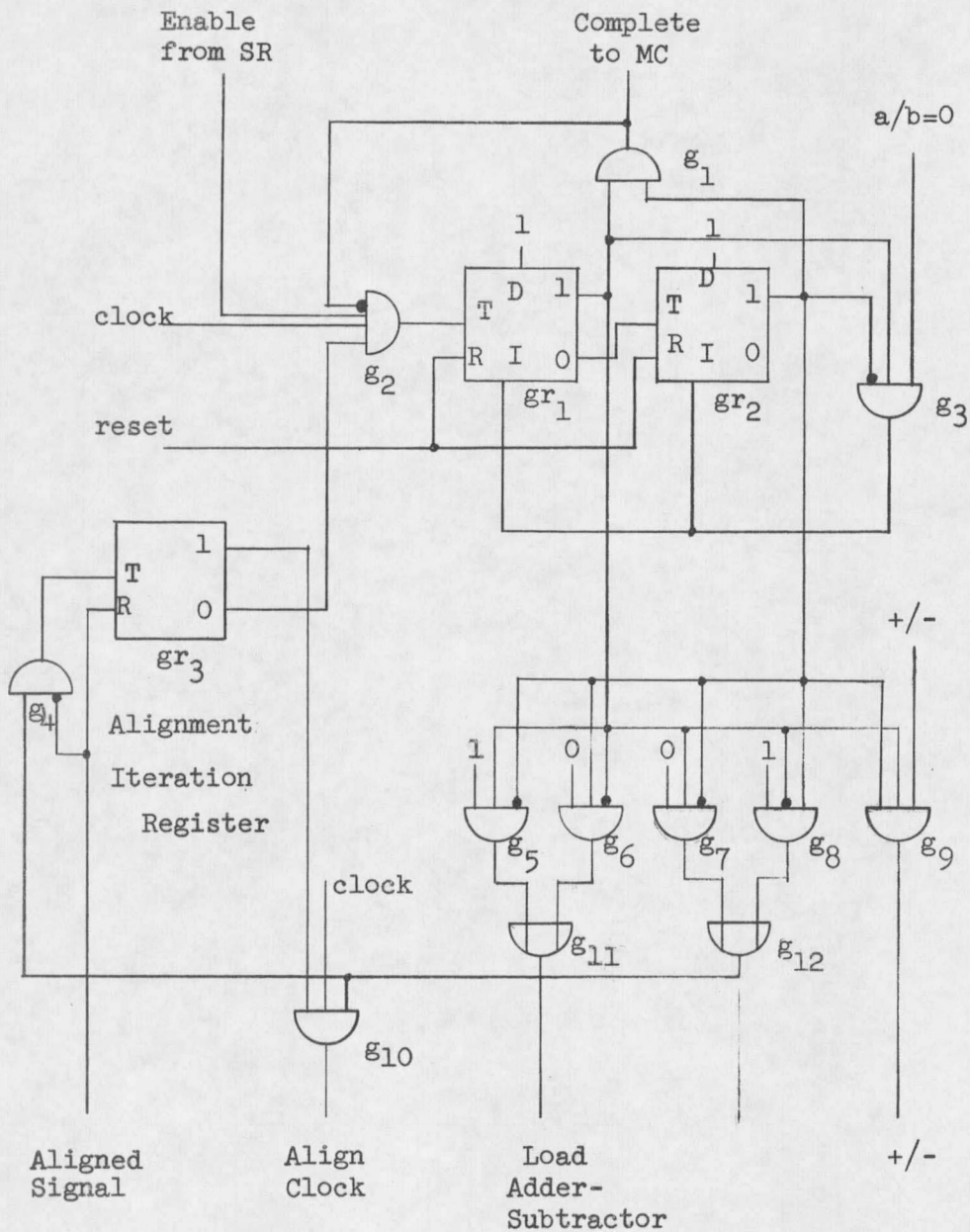
Figure 3.19 Microinstruction Generation Section for the Data Source Selection and Router Module

until the end of the macroinstruction execution unless a data store operation is required. If data is to be stored in cell memory, a store enable signal is sent from the normalizing and data link microinstruction generation section at the fifth cycle period resetting the counter of this generation section. With the resetting of the counter, all gates in this section are disabled except for gates g_3 , g_6 , g_9 , and g_{12} , which are now enabled. With these gates enabled, the memory location of the data to be stored (macroinstruction bits 0-3) is routed to the A memory decoder. The data is stored in cell memory by a command from the GCU sent to all enabled processing cells.

Additional logic (gates g_{15} , g_{18} , g_{21} , g_{24}) is provided to allow cell memory access by the data transfer block. A four bit address code and a route selection command are sent from the data transfer block to the MOB during the data transfer cycle. As with localized operations, a store data command is sent by the GCU to all enabled cells to allow storage of intercell transferred data.

3.7.2.2.2 Parallel Adder-Subtractor Generation Section

The generation of microinstructions for the parallel adder-subtractor module is accomplished by the generation section shown in Figure 3.20. This section generates three operation cycles: the first cycle loads the operand registers, the second cycle aligns the floating point numbers, and the third cycle initiates the addition



Microinstruction to Adder-Subtractor

Figure 3.20 Microinstruction Generation Section for the Parallel Adder-Subtractor Module

or subtraction process. The sequence of microinstructions generated by this section is summarized in Table 3.5.

Table 3.5 Microinstructions for the Parallel Adder-Subtractor Module

Cycle period	Align clock	Load A	+/-
1	0	1	0
2	1	0	0
3	0	0	1/0

This sequence is initiated by the enable signal from the sequential router (if macroinstruction bit 14 is a 1) enabling input gate g_2 . This gate allows the clock pulses to be counted by the two stage binary counter (gr_1 and gr_2). The first count enables the load A signal. The second count disables the counter by setting alignment register gr_3 (assuming that a data word requires alignment). The alignment register then enables the alignment clock (align) by enabling gate g_{11} allowing alignment pulses to be sent to the adder-subtractor sections until an alignment inhibit signal is received. The alignment inhibit signal resets the alignment register, which again enables the counter. The third count allows the +/- signal to pass to the adder-subtractor section, sends an operation complete signal to the master control section, and disables input gate g_2 .

The next clock signal is counted by the MC counter which in turn resets this generation section.

The alignment cycle can be bypassed by detection of an operand and/or operands that are zero. After the first count, gate g_3 is enabled. If an operand zero signal is detected, the counter is set to the third count, effectively bypassing the alignment cycle. If the floating point data words are initially aligned, the alignment register is not set by the input logic, and the counter continues to the third count.

3.7.2.2.3 Parallel Carry-Save Multiplier Generation Section

The generation of microinstructions to implement the operation of parallel multiplication is accomplished by the section shown in Figure 3.21, requiring three clock periods (300 nanoseconds). The enable signal from the sequential router is directed to input gate g_2 of the generation section, and to the initiate line of the parallel carry save multiplier (if the macorinstruction bit 16 is a 1). With this signal the multiplication process is initiated and continues until the counter has counted to the third count. At the third count, input gate g_2 is disabled and a complete signal sent to the master control section. The complete signal allows the counter of the master control section to count another pulse and at the same time sends a reset signal to the multiplier generation section. With this signal the parallel multiplier output stage contains the floating point product.

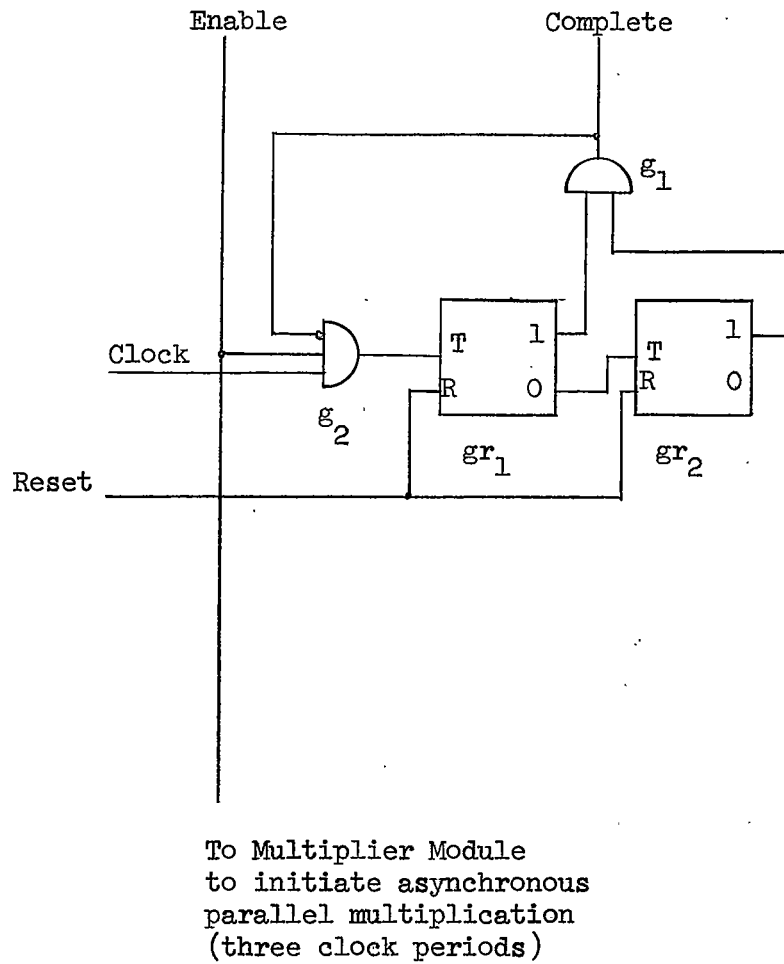


Figure 3.21 Microinstruction Generation Section for the Parallel Carry Save Multiplier Module

3.7.2.2.4 Normalizing and Data Link Generation Section

Generation of control signals for the normalizing and memory storage module is accomplished by the microinstruction generation section shown in Figure 3.22. This generation section generates signals for three operation cycles. The first cycle loads the normalizing register with a 34 bit data word from the selected arithmetic module (adder-subtractor, multiplier, or auxiliary operation modules). The second cycle normalizes the floating point number by an iterative procedure, and the third cycle resets the generation section. The microinstruction information generated during these three cycles is summarized in Table 3.6.

Table 3.6 Microinstructions for the Normalizing and Data Link Module

Cycle	Normclk	Load N	Reset	Selection
1	0	1	0	1/0
2	1 (repeated)	0	0	1/0
3	0	0	1	1/0

The operation of this generation module is initiated by an enable signal from the sequential router (if macroinstruction bit 17 is a 1), enabling gate g_3 . The GCU clock pulses are then counted by the two stage counter (gr_1 and gr_2). The first count allows the normalizing register to be loaded (load N is enabled). The second count

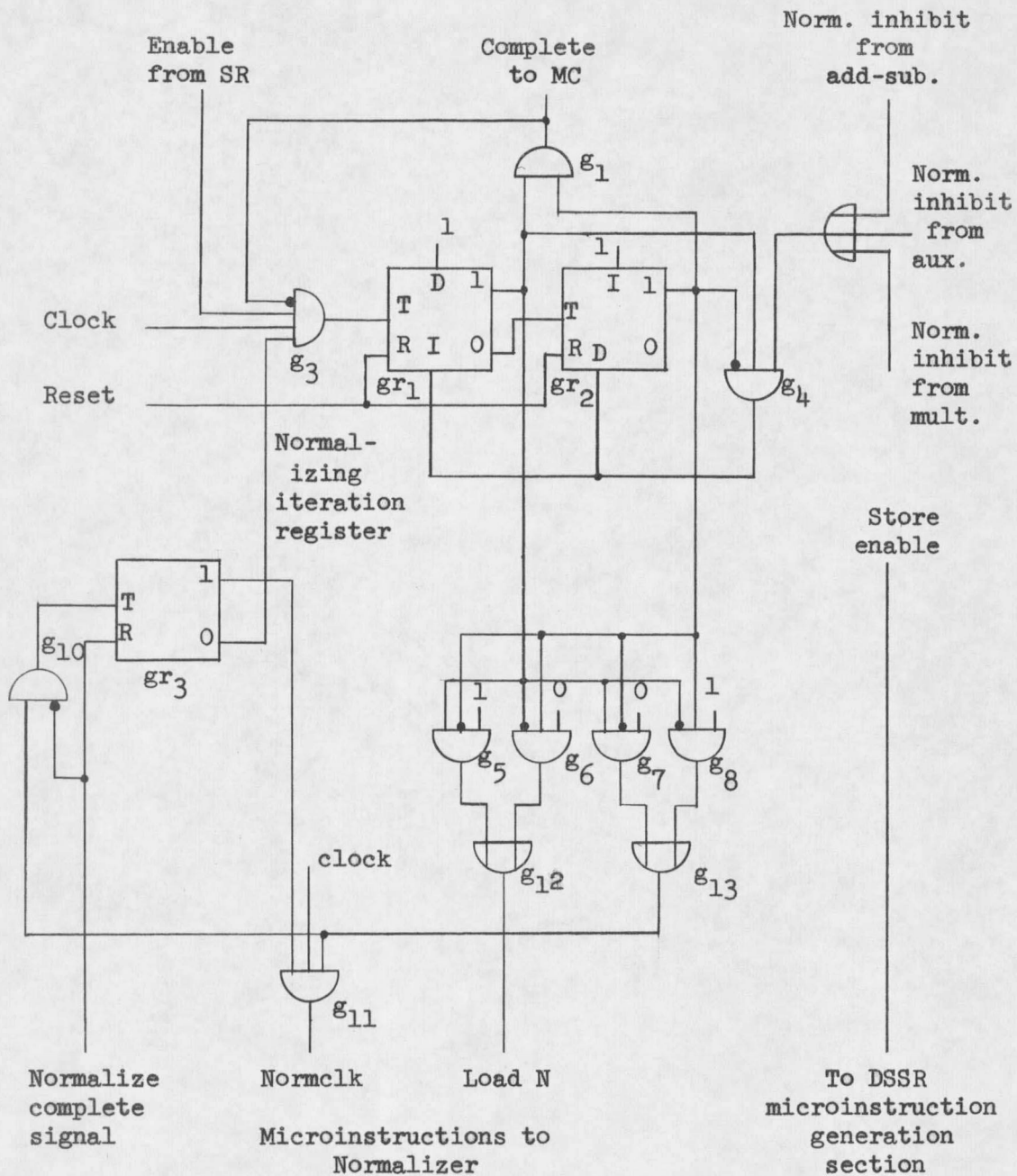


Figure 3.22 Microinstruction Generation Sections for the Normalizing Data Link Module.

disables the counter by setting normalizing register gr_3 . The normalizing register then enables the normalizing pulses (normclk) to be routed to the normalizing section by use of the normclk gate (g_{11}). The disabling of the counter allows the normalize microinstruction to be iterated until a normalize complete signal is received from the normalizing sections. This signal resets the normalizing register enabling the counter. With the third count, an operation complete signal is sent to the master control unit, and the input gate (g_3) is disabled. The next GCU clock pulse is counted by the master control unit counter issuing a reset signal to this generation section. The store enable command to the data source selector and router generation section is present from the initiation of this generation section (if macroinstruction bit 18 is a 1).

The normalizing cycle can be bypassed by detection of a normalization inhibit signal from the adder-subtractor, the multiplier, or the auxiliary operation modules. After the first cycle, normalization bypass gate g_2 is enabled. If a normalization inhibit signal is sent to this section, the counter is set to the third count, effectively bypassing the normalization cycle. If the floating point number is initially normalized, the normalization register is not set (since the logic disables set gate g_{10}), and the counter moves to the third cycle.

3.7.3 Memory Module

Solid state memory units are used to store processing array data in processing cell locations. A single bit memory cell is shown in Figure 3.23, consisting of a general register cellular unit and associated logic. The use of two line select circuits allows the simultaneous reading of stored data on one or both of the read lines. The truth table of the bit memory unit is shown in Table 3.7.

Table 3.7 Truth Table for a Memory Storage Cell

	ws_1	ws_2	r_1	r_2	w	s
read r_1	1	0	1/0	0	0	0
read r_2	0	1	0	1/0	0	0
write w_1	1	0	0	0	1/0	1

The memory module for a processing cell consists of sixteen words of 32 bits organized into a two-dimensional non-destructive memory array and two decoders (A and B), as shown in Figure 3.24. The memory module has the ability to read simultaneously two words of memory into the data source selection and router module. The decoders are supplied with four bit address codes from the MOB control module, which are used to simultaneously select one of sixteen words in processing cell memory for reading. However, only one word can be written into memory at a time. The B decoder uses the

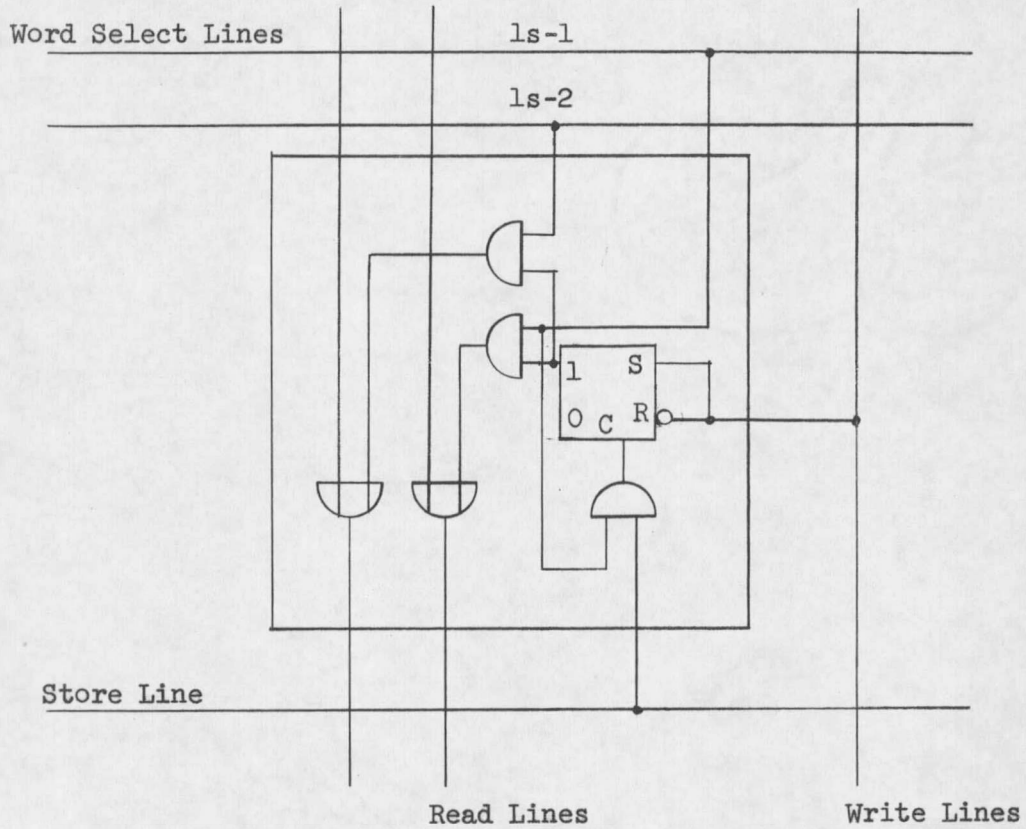


Figure 3.23 Memory Storage Cell

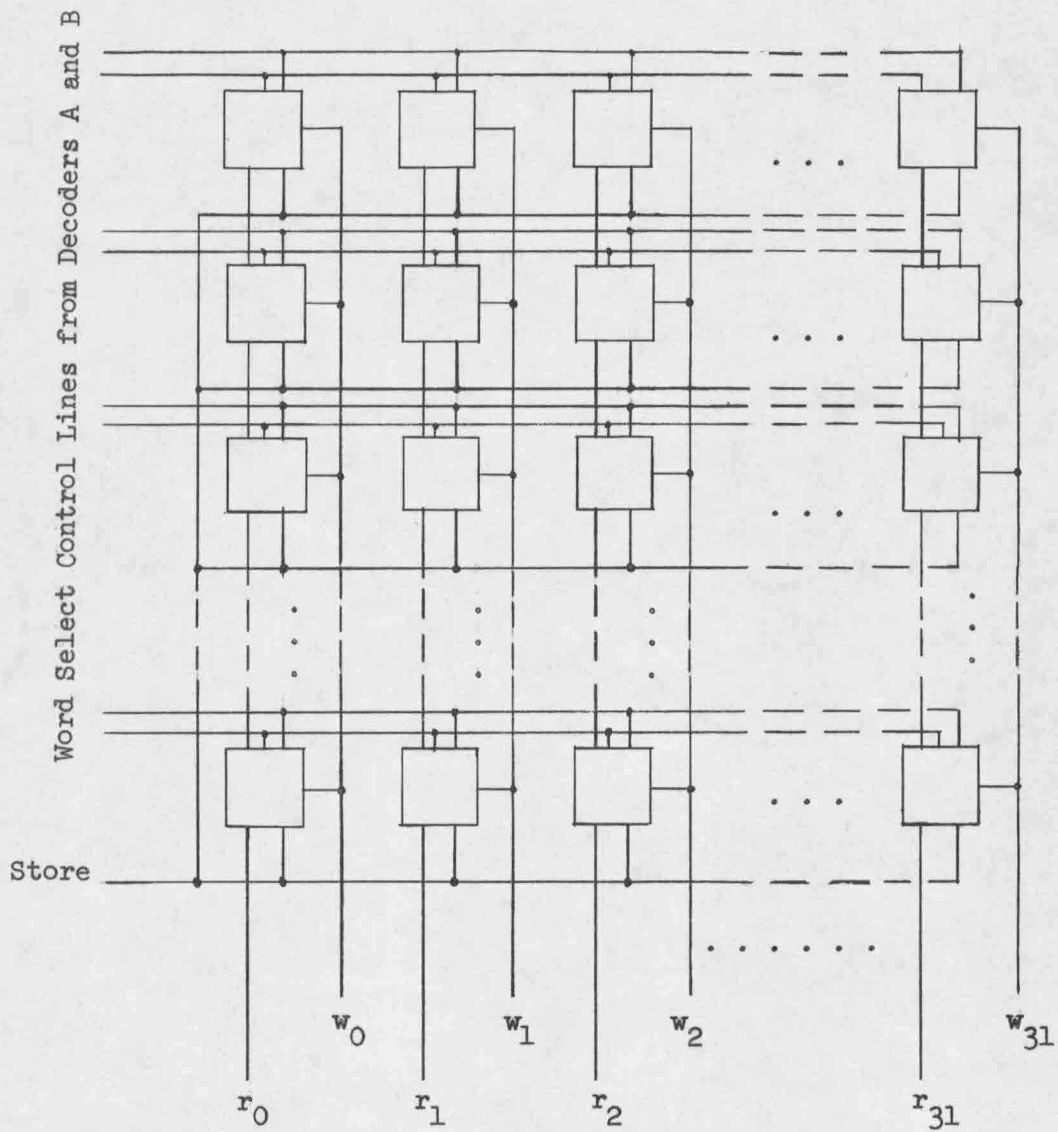


Figure 3.24 Memory Module

four bit address code of M_3 to write a single 32 bit data word into one of a possible sixteen memory locations. Application of a store command from the GCU writes in parallel the word into the selected memory location.

This memory module organization is efficient in that data busses A and B can be directly attached to processing cell memory in one GCU clock period, and a localized store data word operation can also be accomplished in one GCU clock period.

3.7.4 Data Source Selection and Router Module

The data source selector and router module is shown in Figure 3.25. This module is used to route selected cell memory data to data busses A and B, or from cell memory to the data transfer block via data buss A. Data supplied from the data transfer block can also be routed to the B data buss for direct mathematical operation. Module operation is summarized in the truth table shown in Table 3.8.

Table 3.8 Truth Table for the Data Source Selector and Router

S_1	S_2	Operation
0	0	data busses A and B loaded with cell memory data
0	1	cell data on buss A is routed to DTB
1	0	data buss A loaded with cell memory data, data buss B loaded with DTB data
1	1	no operation

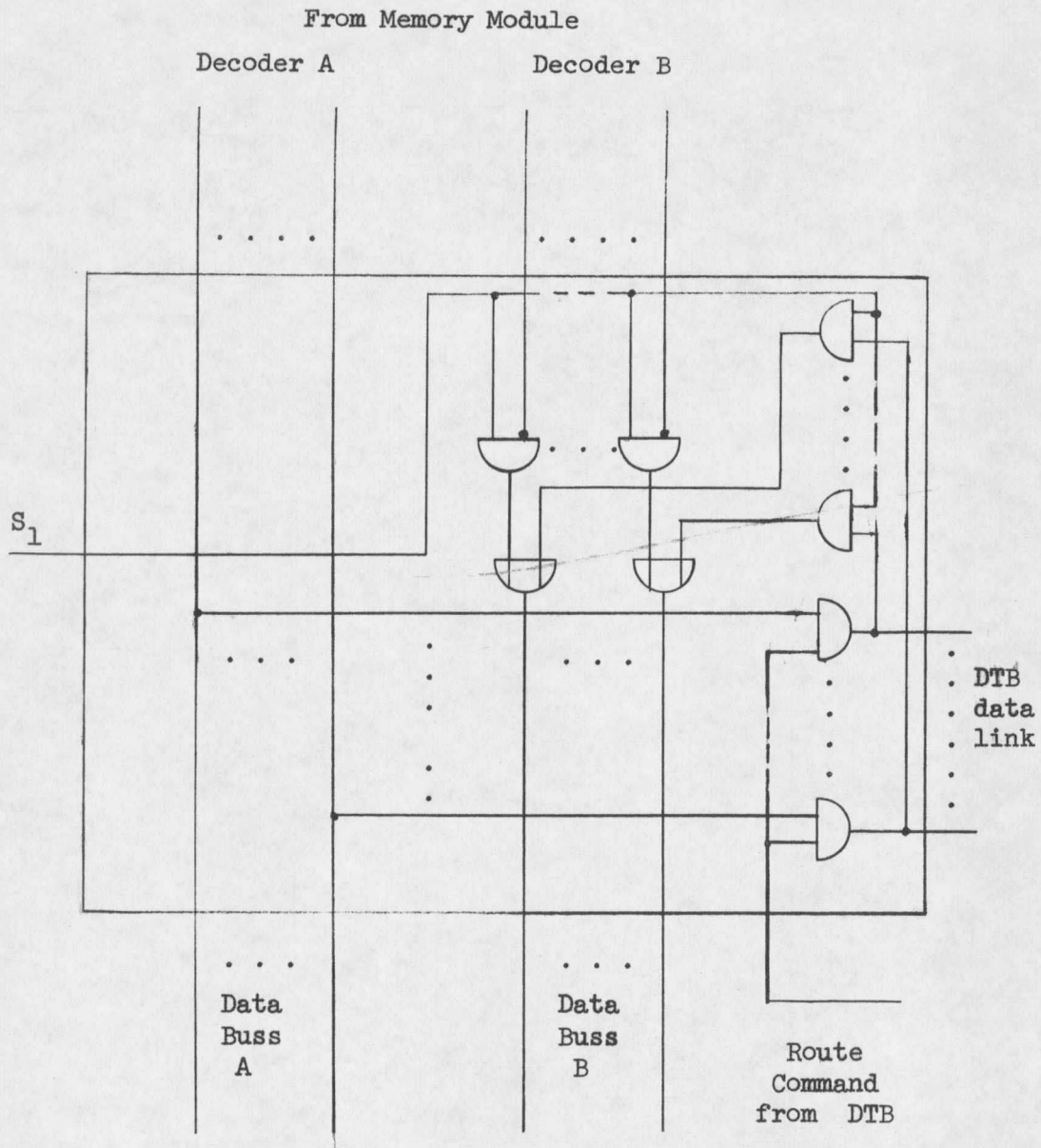


Figure 3.25 Data Source Selector and Router Module

Input lines to this module are connected directly to the cell memory module read lines, allowing the two selected operands to be routed to the A and B data busses in one GCU clock period. The data on buss A can be routed to the data transfer block by setting $S_2 = 1$. Should an operation require external data (that is, broadcast operation), the data can be routed to data buss B by setting $S_1 = 1$. It must be noted that control signal S_2 comes from the data transfer block.

3.7.5 Parallel Adder-Subtractor Module

Addition and subtraction of floating point numbers within processing cells is accomplished by the parallel adder-subtractor module shown in Figure 3.26. This module consists of a zero fraction detection section, an alignment section, and an add-subtract section. This module is similar to the one developed by Cannon (21), with the addition of a zero fraction detection and alignment bypass sections, and the deletion of the normalizing section (the adder-subtractor and multiplier use a common normalizing section). An eight bit format (extendable to 32 bits) is used for illustrative purposes in this discussion.

Under control of the MOB command generation module, the two 32 bit alignment registers are loaded with the data on the MOB A and B data busses. The sign bits are extended to the left one bit

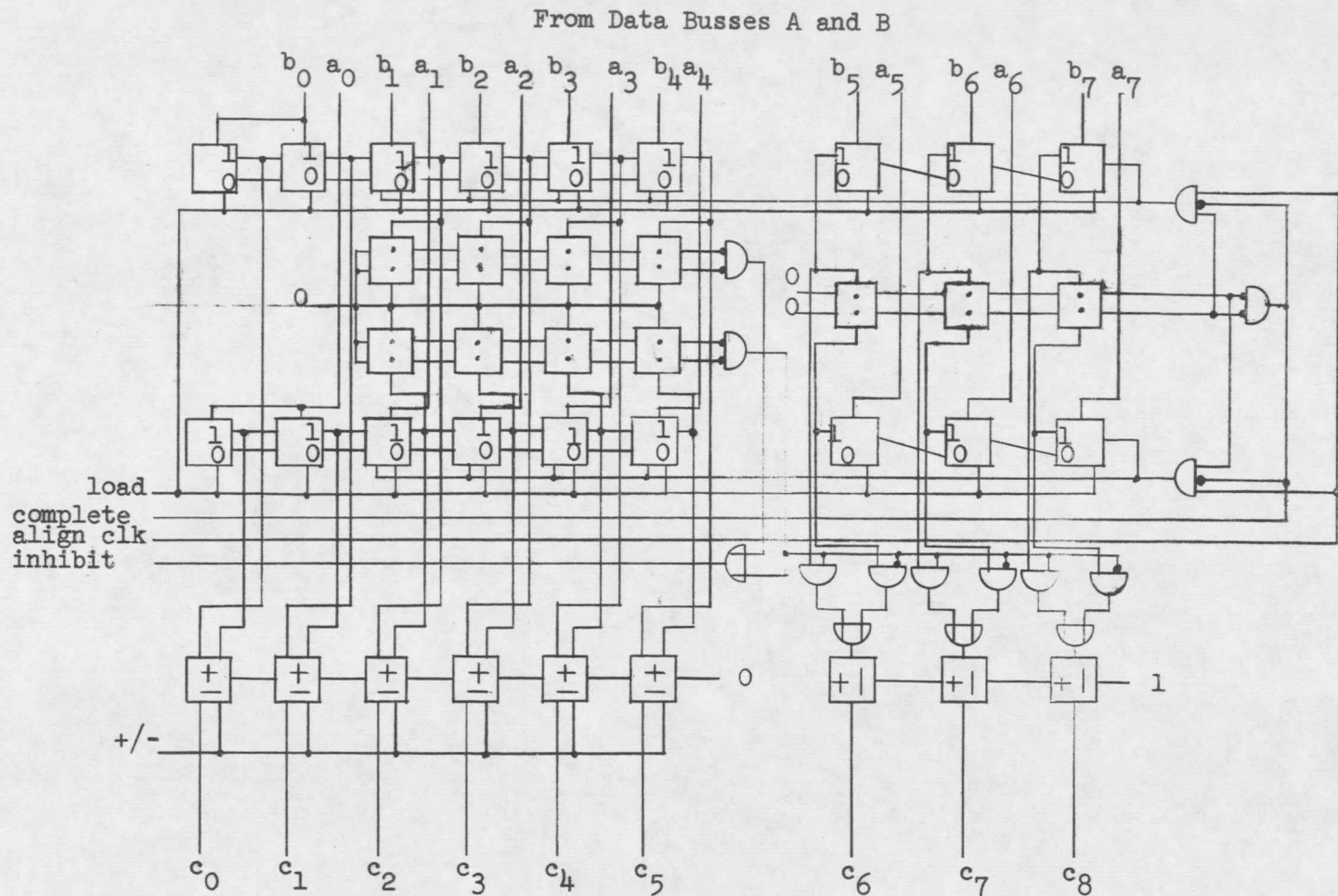


Figure 3.26 Parallel Floating Point Adder-Subtractor Module

position. The zero fraction detection section compares the two floating point fractions to zero. If both fractions are non-zero, alignment and subsequent addition-subtraction proceed. If either of the fractions is zero, the alignment process is bypassed, a normalization inhibit signal is transmitted to the normalizing and data link module, and the addition-subtraction process is initiated. The exponent circuitry is designed to use the exponent of register B in forming the output exponent (since both exponents are equal after alignment) or in the presence of a zero fraction to use the non-zero fractions exponent (or alignment register B exponent if both are zero).

The alignment section contains two 6 bit registers and two 3 bit exponent registers. The alignment process utilizes a process by which the fraction part of the smaller number is shifted right with its exponent part being increased by one for each shift, until the exponents of the two numbers are equal. The shifting process proceeds with a clock signal (align) sent by the MOB command generation module. This clock is inhibited when the two exponents have been aligned. The smaller exponent is selected by using the comparator output to inhibit the clock signal going to the alignment register containing the larger exponent. The exponent registers are connected as ripple-through binary counters and the fractional registers are connected as shift registers. The sign bits do not receive the

clock signal, allowing the sign bit to be extended right in the shift process.

The add-subtract section consists of a string of six add-subtract cells used to form the resulting fraction and a string of three add-one cells used to form the exponent. This section receives in parallel data from the alignment register A and B (upon completion of the alignment process) forming the sum or difference of the input numbers depending on the bit value of the add/sub control line; a binary 0 for addition and a binary 1 for subtraction. The output of this section consists of a sign bit, five fraction bits, and three exponent bits. The additional bit on the left of the fraction allows for a possible bit overflow that occurs if the sum of the two fractions is greater than one. With the possibility of an overflow bit, the decimal point of the output fraction is moved one bit position to the left of the input numbers (allowing retention of an extra bit) and the exponent is increased by one (accomplished by the cascade of add-one cells). Hence, to retain full binary information in the presence of a possible overflow condition, an extra bit position is created resulting in an output number consisting of nine bits (instead of eight). The resulting number format is compatible with the normalizing section to which it is now routed. Regardless of whether an overflow is present, full binary information is retained with this parallel adder-subtractor module.

3.7.6 Parallel Carry-Save Multiplier Module

As a result of the array-structured computer simulation (Chapter 2), it became apparent that a fast floating point multiplier module should be designed (speed improvement of 16:1 compared to serial multiplication). An investigation into fast multiplication resulted in the design of an LSI implementable parallel multiplier based on the Purdue design effort (33), and the cellular array multiplier presented by Ramamoorthy and Economides (34). This parallel carry-save multiplier is capable of multiplying 32 bit positive or negative floating point numbers.

The parallel carry-save multiplier module, shown in Figure 3.27, (shown as eight bits), consists of an input complementing section, a summand generation section, a parallel carry-save multiplier section, an exponent section, and an output complementing section. The multiplication process is asynchronous after being initiated by the multiplier enable signal from the MOB command generation module. Once enabled, the input complementing section forms the absolute value of the input numbers (a requirement of a parallel carry save multiplier), utilizing 48 two's complementor cells in cascade (shown as four cells for each input number) and associated logic. The sign bit of each number is used as an enable/disable command on the input complementing logic; if the sign bit is a 1 (negative number), the fraction bits are complemented and directed to the summand generator section, whereas a

0 (positive number) results in the bypassing of the complementing logic and the input number being directed to the summand generator. The sign bits are also used as input to an exclusive OR sign gate to determine the sign of the product and whether the output complementing section is to be used.

The 24 bit absolute value fractions are supplied to the 576 two input AND gates (shown as 16) that form the summand generation section; the $a_i b_j$ ($i, j = 0, 1, \dots, 23$) summands are required for all parallel multiplication methods investigated. Hence, the basic problem in designing a high speed multiplier is to reduce the time required to ripple add the summands. The carry-save multiplier has been selected, resulting in slightly slower speed and more components than other available methods, but should be the easiest to implement in LSI cellular arrays due to its regular, simple structure. For illustration purposes, the process of multiplication of two 4 bit binary fractions is shown in Figure 3.28. The parallel carry-save multiplier section uses parallel addition and ripple carry to form a product of the columns of summands depicted in Figure 3.28, where these columns of summands are used as column inputs to the adder array. When supplied with these summands, the adder cells ripple their outputs down the array with the product being formed on the output lines (P_i , $i = 1, 2, \dots, 25$), as shown in Figure 3.29 (shown as five bits). Gate delay analysis shows that for the multiplication of two 24 bit

binary fractions, three GCU clock periods (300 nanoseconds) are required (34). The 25 most significant bits (shown as five) are retained and sent to the output complementor section. The multiplier section requires N half adder cells and $N^2 - 2N$ full adder cells in forming the product (where N is the number of fraction bits). Hence, in this multiplier section 24 half adder cells and 528 full adder cells are used.

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				a_3b_0	a_2b_0	a_1b_0	a_0b_0
			a_3b_1	a_2b_1	a_1b_1	a_0b_1	
		a_3b_2	a_2b_2	a_1b_2	a_0b_2		
	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7

Figure 3.28 Binary Multiplication Process

The output complementing section either passes the product fraction to the output lines or complements the product fraction and then passes it on to the output lines, depending on the output of the exclusive OR sign gate; a 1 indicates complementing, whereas a 0

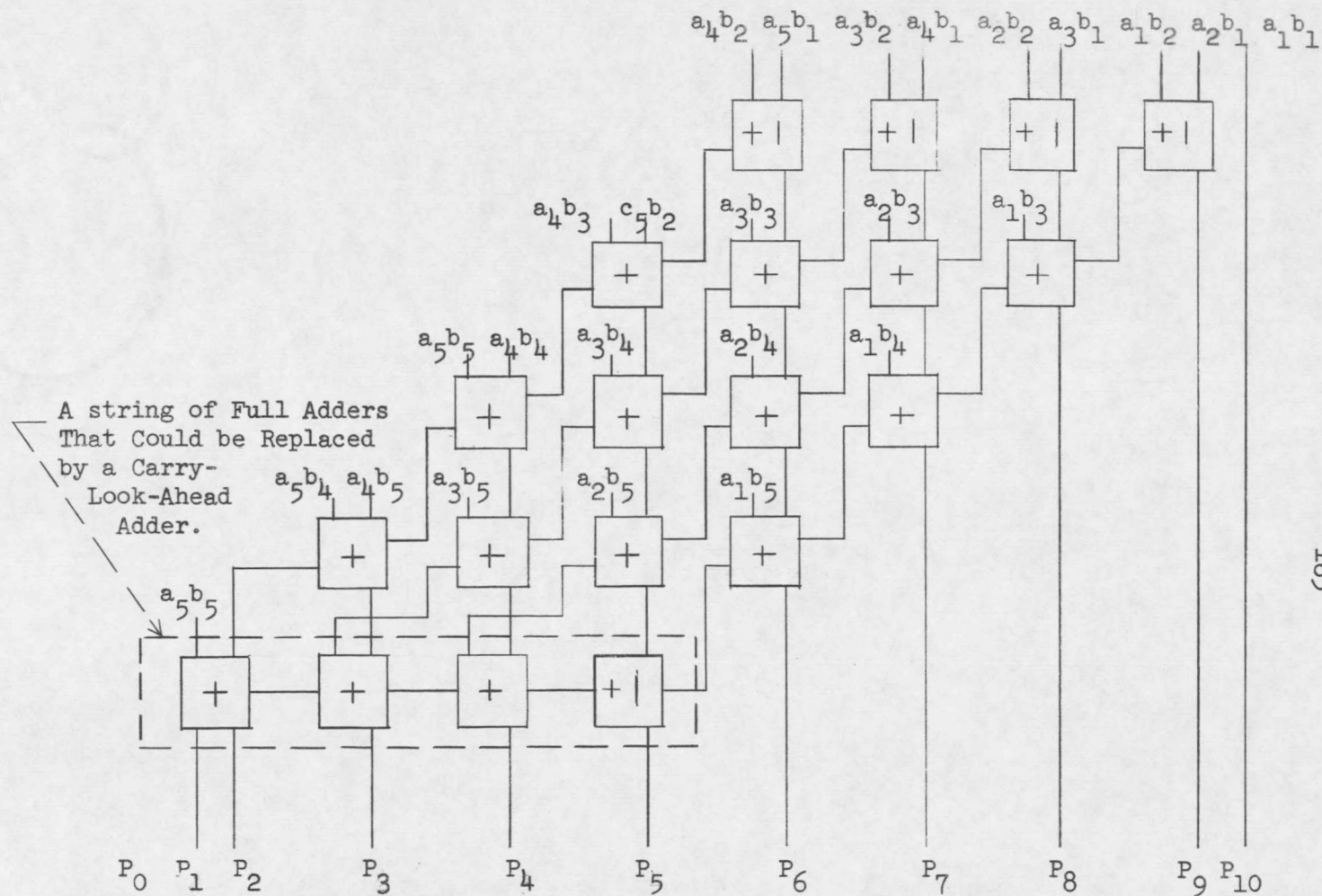


Figure 3.29 Parallel Carry Save Multiplier

indicates bypass of this section. In addition, the sign bit is available as output.

The exponent of the formed product is always positive due to the excess base 64 representation. The product exponent is formed by adding the two input number exponents and then subtracting 64, as shown below:

$$(E_1 + 64) + (E_2 + 64) - 64 = (E_1 + E_2) + 64$$

The upper adder-subtractor cascade is an adder ($g=0$), and the lower cascade is a subtractor ($g=1$). Hence, the output of the parallel carry-save multiplier module is the product of two input floating point numbers, and is available on the 33 output lines (shown as nine bits).

Logic circuitry has been provided to detect the presence of a zero product, resulting in an output number consisting of a sign bit, 25 bits of binary zeroes for the fraction, and an exponent value of +64. In addition, a normalization inhibit signal is sent to the normalizing and data link module so that normalizing of the zero floating point number is not attempted.

3.7.7 Normalizing and Data Link Module

The normalizing and data link module, shown in Figure 3.30, performs two separate operations. The normalizing section performs the operation of normalizing data that results from mathematical

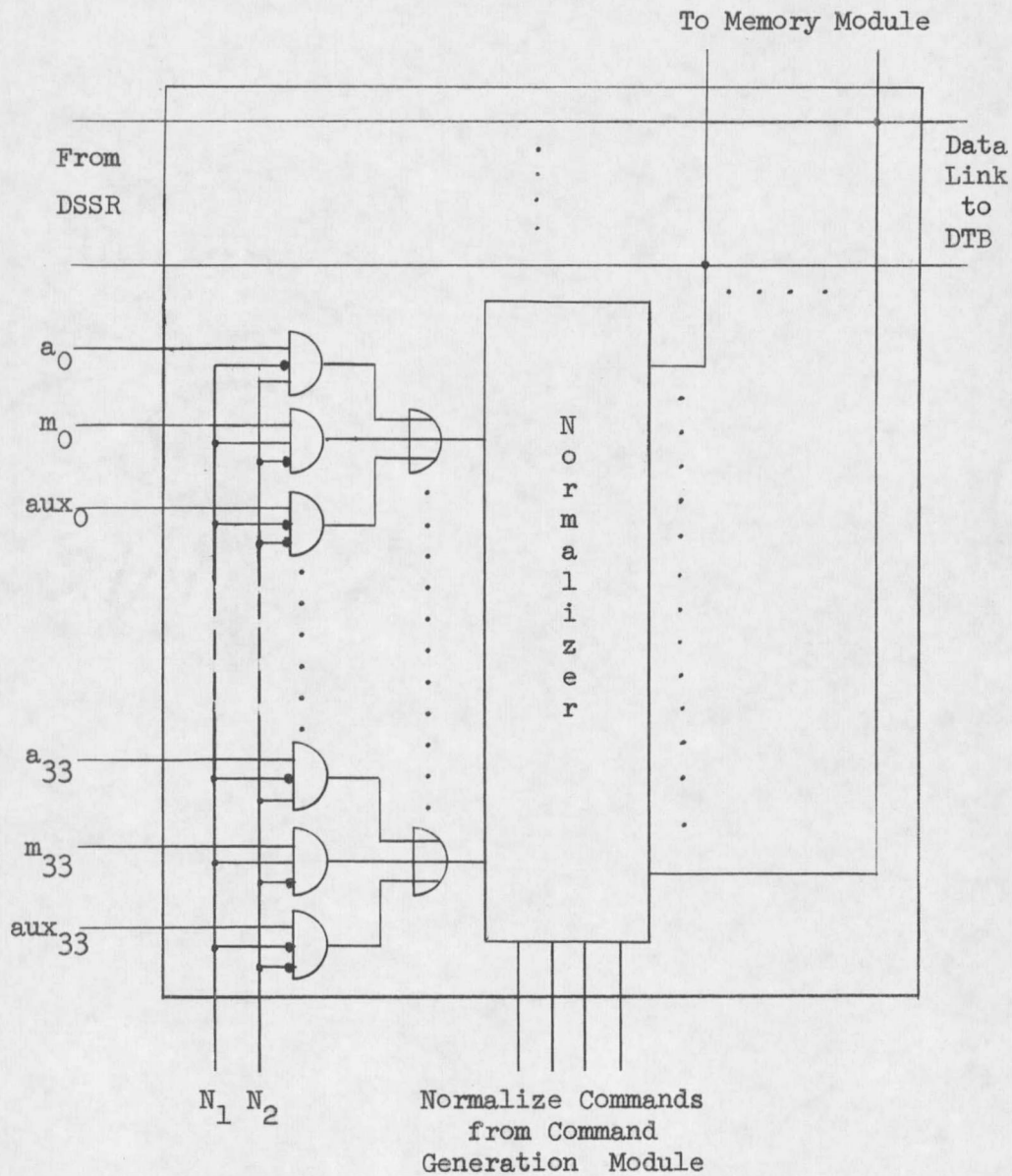


Figure 3.30 Normalizing and Data Link Module

operation on input data. The input-output section forms the data link between the normalizing section and processing cell memory, the two way data link between processing cell memory and the data transfer block, and the one way data link between the data transfer block and the B data buss of the cell MOB. The normalizing section shown in Figure 3.31 (shown as eight bits) is used to normalize the result supplied from the mathematical operation blocks according to the selection code shown in Table 3.9. This section consists of a 26 bit shift register (shown as six bits) used for the fraction, and a seven bit decrementing counter register (shown as three bits) used for the exponent of the supplied floating point number. The registers are loaded in parallel by a command signal (load N) from the MOB command generation module.

Table 3.9 Normalizer Data Source Selection Code

N_1	N_2	Operation
0	0	no operation
0	1	from adder-subtractor
1	0	from multiplier
1	1	from auxiliary section

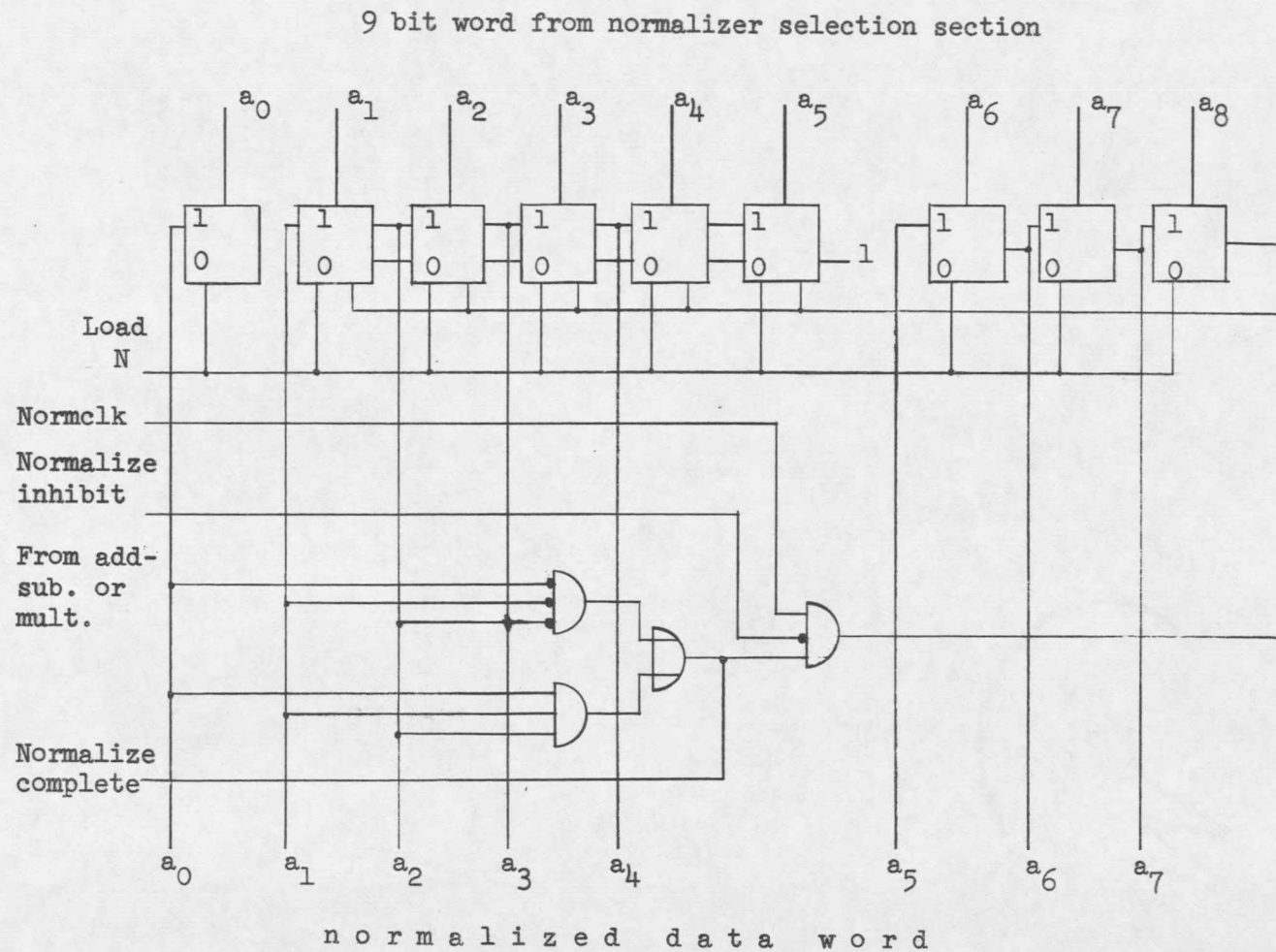


Figure 3.31 Normalizing Section

Normalization of the supplied number is accomplished by clock pulses (normclk) from the MOB command generation module. The clock pulses result in the fraction being shifted to the left by one bit and the exponent being decreased by one for each clock pulse, until the normalize condition is met (see section 3.3.2). The sign bit remains unchanged during the shifting process. Detection of the normalize condition is accomplished by logic that examines the first two bits of the fraction. When the normalization condition is met, an inhibit signal is sent to the MOB command generation module indicating the end of the normalizing cycle. At this time the normalized number is present in the fraction register with the 24 highest order bits retained as the fraction output. The normalized 32 bit floating point number is supplied to the data link section for routing.

The normalizing process can be inhibited by the inhibit N line. This control line is actuated if either of the input numbers supplied to the adder-subtractor or the multiplier are zero.

3.8 Data Transfer Block

3.8.1 Introduction

Transfer of data between cells of the processing array is accomplished by the data transfer block (DTB). This block forms the second segment of a processing array cell, allowing the data transfer operation to be separated from other localized cell operations. This

segmentation allows the data transfer block to be designed separately, with emphasis placed on speed of data transfer operations (a design consideration discussed in section 2.10). To implement data transfer operations, the DTB's use the segmented data bussing structure shown in Figure 3.3. These data busses consist of 32 horizontal and vertical data line segments (one line per bit of a 32 bit data word), which terminate at adjacent DTB's. Row busses are connected to the X command blocks to perform the additional operation of input or output of processing array data. Hence, by supplying control commands, the data busses can be structured to execute a given data transfer operation, including the bypassing of unused processing array cells. Data transfer within the processing array occurs in parallel under control of a 6 nanosecond clock, allowing any data transfer operation to be executed within a single GCU clock period (100 nanoseconds). Hence, the array-structured computer is a two clock computing system: a 10 Mhz clock (100 nanosecond period) for all operations except data transfer, and a 167 Mhz clock (6 nanosecond period) for data transfer operations. The logical design of the DTB is presented in the following section.

3.8.2 Data Transfer Block Organization

The data transfer block is shown in Figure 3.32, consisting of 385 gates, two 32 bit shift registers, and one 32 bit comparator. A total of 158 input-output lines is required, of which 128 are bit transfer lines. The DTB has direct access to processing cell memory, and each of the segments of the X and Y data busses. The DTB has 17 input control commands which are summarized in Table 3.10.

The data transfer cycle for an array-data transfer operation is shown in Figure 3.10. For processing array-data input-output, the cycle is slightly different (the load and store commands are appropriately enabled or disabled). The processing array parallel data comparison cycle also varies slightly (the store command is replaced by the compare command).

To initiate the process of data transfer, the GCU issues a macroinstruction to the data transfer microprocessing unit, which issues a sequence of microinstructions to the DTB. These microinstructions contain all necessary information to accomplish the data transfer operation. A four bit element address code is sent to all array DTB's selecting the word in cell memory to be transferred. The selected word is routed to the two 32 bit shift registers by the control signal X_{16} . These parallel shift registers, one of which is shown in Figure 3.33, are loaded by DTB bit command (X_8).

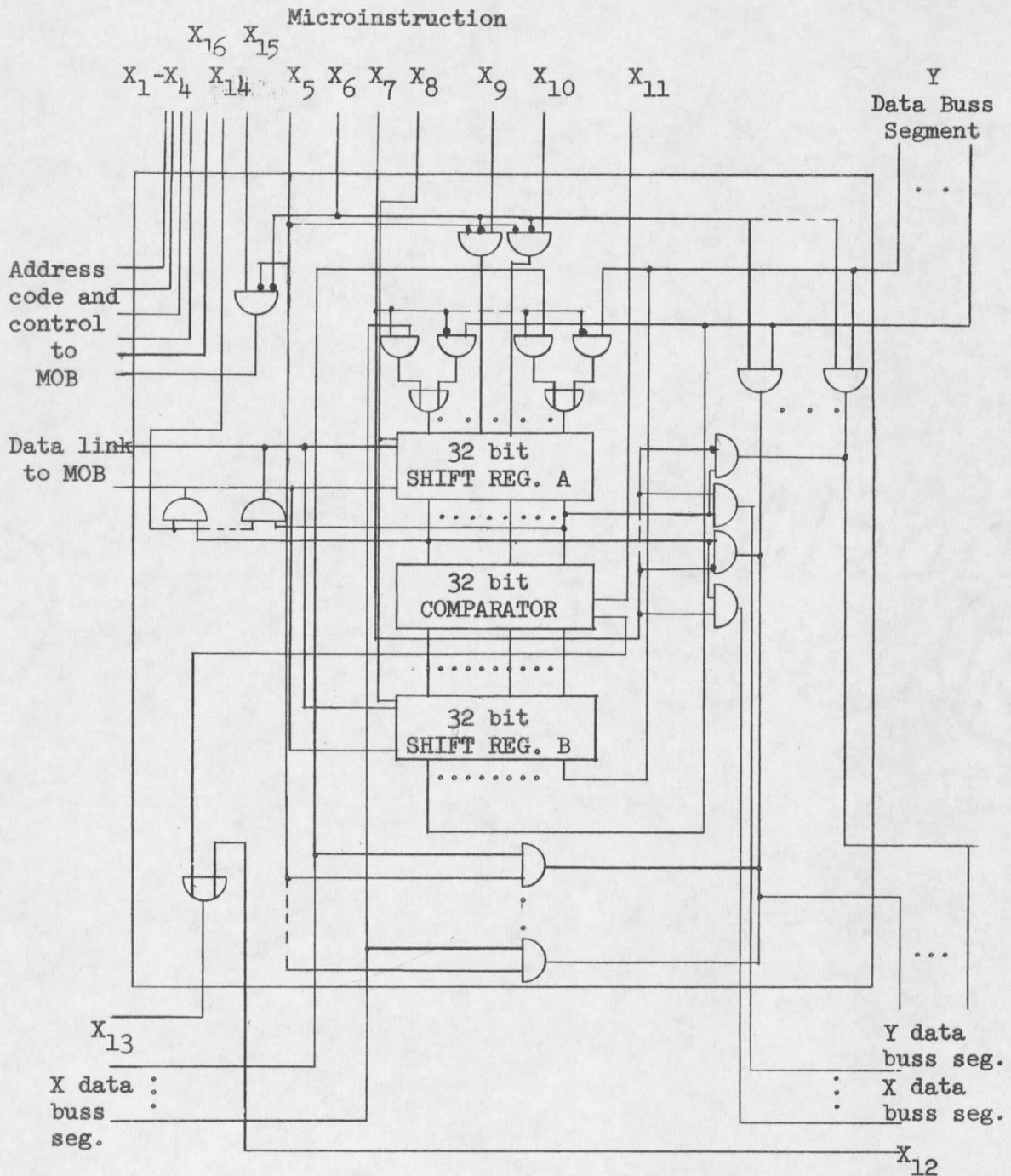


Figure 3.32 Data Transfer Block

Table 3.10 DTB Microinstruction Bit Designation

Control Line	Level	
$X_1 - X_4$	0/1	4 bit code for element address, directed to MOB
X_5	0	X enable
	1	X disable (cell bypass)
X_6	0	Y enable
	1	Y disable (cell bypass)
X_7	1	attach SR A to horizontal buss, SR B to vertical buss
	0	attach SR A to vertical buss
X_8		load SR A and SR B from MOB memory
X_9	1/0	shift clock for SR A
X_{10}	1/0	shift clock for SR B
X_{11}	0	no comparison
	1	compare 32 bits words in SR A and SR B
X_{12}	0/1	fault signal from right DTB
X_{13}	0	no error found, SR A and SR B contain identical data
	1	SR A and SR B do not compare
X_{14}	0	no operation
	1	route data in SR A to MOB data lines (MOB uses this line in external data mode)
X_{15}	0	no operation
	1	store data in MOB memory
X_{16}	0/1	route data from MOB memory to SR A and SR B
X_{17}	0/1	reset

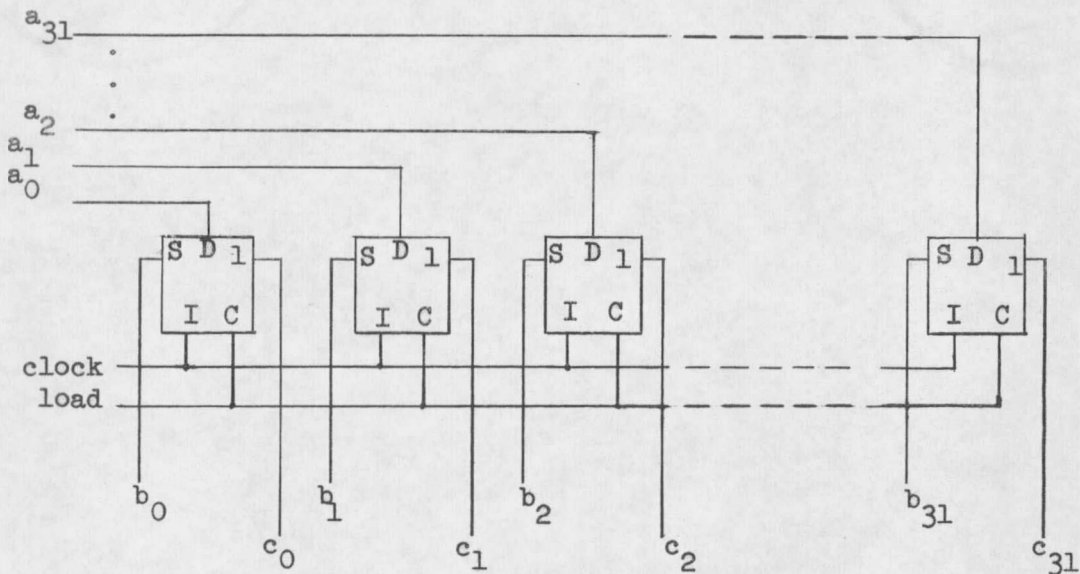


Figure 3.33 32 Bit Parallel Shift Register

Once the shift registers are loaded, the data bussing structure is formed by a DTB command (X_7). The data transfer process then begins with clock pulses sent from the data transfer microprocessor (X_9 and/or X_{10}). Transfer of data in shift register B (SR B) occurs only in the transpose operation.

Provision has been made to bypass unused or unaffected processing cells (either horizontally or vertically) by setting DTB bits X_5 and X_6 to 1; if the bits are 0, data in shift register A is either shifted to the right or down (depending on X_7), under control of the 6 nanosecond clock. The shifting process continues until the data transfer microprocessor detects that the required shifts have occurred,

at which time the data transfer clock is disabled. The processing array utilizes right edge to left edge and bottom edge to top edge data transfer (wrap around data transfer) allowing the formation of long parallel shift registers.

In the transpose configuration, data transfer occurs in two directions; rows of DTB's (SR A) shift right, while columns of DTB's (SR B) shift down. The data from the bottom of the array is routed to the left edge of the array with the right edge to left edge wrap around data structure disconnected (open). In this manner, data from the B shift registers of columns are routed to the A shift registers of rows (column C_i to row R_i data transfer). By the correct number of shifts the column element data is shifted and stored in the row element locations, resulting in a matrix transpose operation.

The 32 bit parallel comparator is used only in the fault-detection mode of operation presented in section 3.9. The comparator is a cellular cascade of 32 comparator cells as shown in Figure 3.34. If the two 32 bit words are equal, the output (X_{12}) is a binary 0; otherwise, the output is a binary 1. The output of all processing cell comparators are ORed to the X command block by rows. The use of these parallel comparators allows a full array comparison to be performed in parallel.

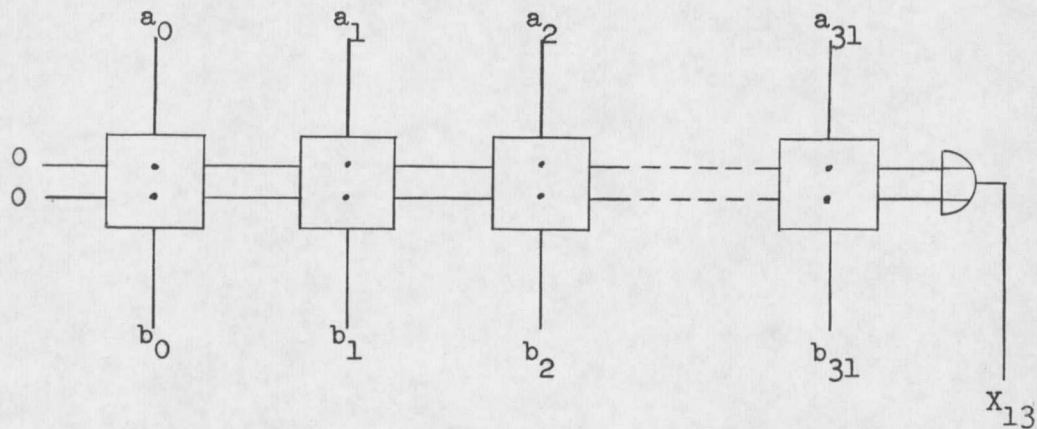


Figure 3.34 32 Bit Parallel Comparator

3.8.3 Data Transfer Operation Bussing Structures

With the exception of the transpose operation, all data transfer operations occur in parallel by rows or columns. Within a row or column, data is shifted under control of the 6 nanosecond clock in the form of long parallel shift registers. All data transfer operations require one GCU clock period (100 nanoseconds) for completion.

As an example of a data transfer operation, the sequence of enabled DTB microinstructions to rotate a matrix to the right five position is shown in Table 3.11. A 6 x 6 matrix is assumed throughout. The data bussing structure shown in Figure 3.35 is formed to execute this data transfer operation.

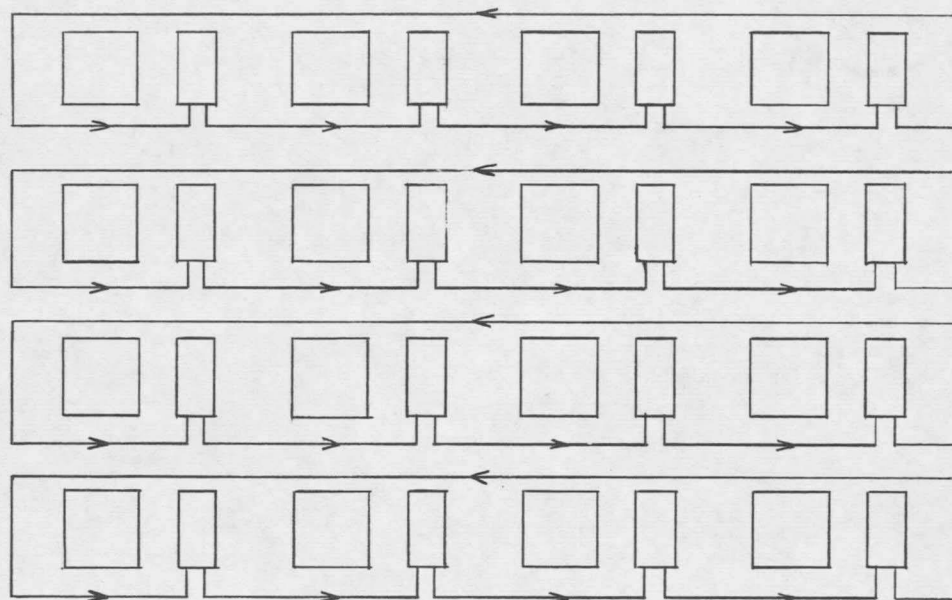


Figure 3.35 Data Bussing Structure for a Rotate Right Operator.

Table 3.11 DTB Microinstructions for a Rotate Right Operation

t	X ₁	X ₂	X ₃	X ₄	X ₅	X ₆	X ₇	X ₈	X ₉	X ₁₀	X ₁₁	X ₁₂	X ₁₃	X ₁₄	X ₁₅	X ₁₆	X ₁₇
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0
3	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
4	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
5	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
6	1	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
7	1	0	1	0	0	0	1	0	1	0	0	0	0	1	0	0	0
8	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
9	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1

The sequence of enabled DTB microinstructions to rotate a matrix up five positions is shown in Table 3.12. The appropriate data bussing structure is illustrated in Figure 3.36.

Table 3.12 DTB Microinstructions for a Rotate Up Operation

t	X ₁	X ₂	X ₃	X ₄	X ₅	X ₆	X ₇	X ₈	X ₉	X ₁₀	X ₁₁	X ₁₂	X ₁₃	X ₁₄	X ₁₅	X ₁₆	X ₁₇
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0
3	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
4	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
7	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0
8	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
9	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1

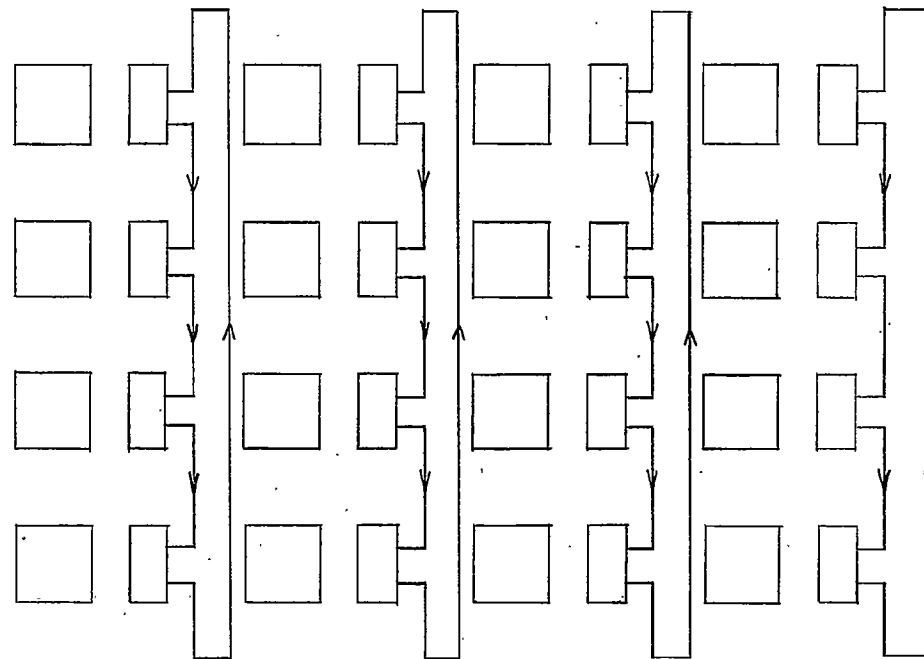


Figure 3.36 Data Bussing Structure for Rotate Up Operator.

As noted in section 3.5.2, in order to implement the skew operations, the X and Y enable commands to the row (column) DTB's must disable appropriate rows (columns) after that row (column) had shifted the correct number of times. The only commands that change in the above tables to implement the skew operations are DTB bit commands X_5 and X_6 , which change from 0 to 1 after the correct number of shifts for that row (column) has occurred.

The data bussing structure for the transpose operation is shown in Figure 3.37. As shown, data transfer proceeds in two directions simultaneously; rows of DTB's (SR A) shift right, columns of DTB's (SR B) shift down. The data from the bottom of the array is routed to the left edge of the array with the right edge to left edge wrap around data structure disconnected (open). In this manner, data is routed into row R_1 of the processing array. The sequence of DTB microinstructions for matrix transpose is shown in Table 3.13.

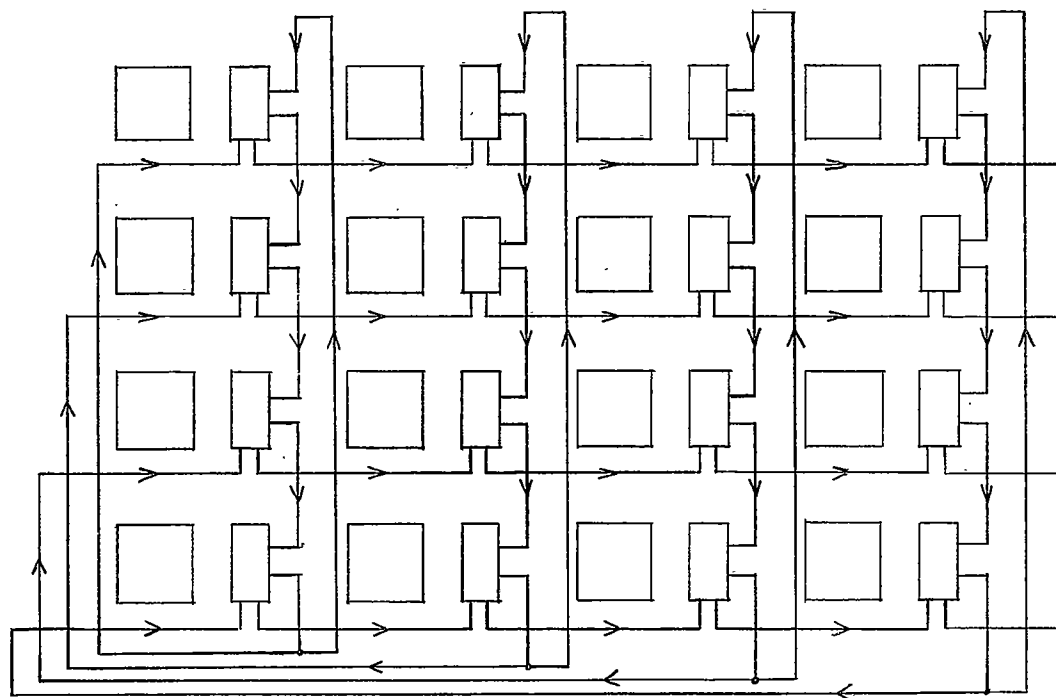


Figure 3.37 Data Bussing Structure for the Transpose Operation.

Table 3.13 DTB Microinstructions for the Transpose Operation

t	X ₁	X ₂	X ₃	X ₄	X ₅	X ₆	X ₇	X ₈	X ₉	X ₁₀	X ₁₁	X ₁₂	X ₁₃	X ₁₄	X ₁₅	X ₁₆	X ₁₇
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0
3	1	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0
4	1	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0
5	1	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0
6	1	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0
7	1	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0
8	1	0	1	0	0	0	1	0	1	1	0	0	0	1	0	0	0
9	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
10	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	1

3.9 Adaptation of the Array-Structured Computer to Fault-Tolerant Operation

3.9.1 Introduction

Continuing increases in speed and complexity of computing systems and the need for highly reliable real-time computing capability have accelerated the demand for fault-tolerant operation. The array-structured computer presented in earlier sections of this chapter will find use in some real-time applications resulting in its adaptation to fault-tolerant operation. The intent of this section is to form an investigative base for future research and development in adapting the array-structured computer to fault-tolerant operation.

This section begins with a discussion of the array-structured computer's fault detection and fault correction hardware. This hardware is used by a special set of three assembly language instructions designed to allow the applications programmer to design the level of reliability of the computing system. Due to tradeoffs between system hardware and effective execution time, three fault-tolerant computing system structures are presented based on the Chapter 1 survey of current trends in fault-tolerant computers. Selection of any one of these structures depends on the required computing capability, execution speed, and level of reliability for the intended application.

3.9.2 Fault-Tolerant Operation Hardware

3.9.2.1 Introduction

Detection and subsequent replacement of faulty cells within the processing array require both array hardware and restructuring routines in the computer executive program. Detection of faults within the processing array is by parallel array comparison of calculated data to known benchmark data. If a fault is detected, the GCU is alerted to its presence and supplied with information on the row location of the faulty cell. With this information the GCU executive program retains system control and restructures the processing array to remove the faulty cell from active participation in all array operations. The method selected for restructuring the array is by row and column substitution. If a fault is detected in the i^{th} row of the array, the i^{th} row and column of the array are removed from active participation by restructuring all macroinstructions (cell enable masks). The removed row and column are replaced by an unused row and column of the array. The presence of multiple faults can be detected and corrected with this method. This method was selected after comparison with a single cell substitution method. Following the restructuring of macroinstructions, the processing computing system returns to normal execution operation.

3.9.2.2 Fault Detection Hardware

Detection of faults within the processing array is by parallel array comparison utilizing the data transfer blocks of all enabled processing cells. The data transfer block, shown in Figure 3.32, has access to both cell memory and external data sources (via the data bussing structure). The array comparison cycle compares data calculated by a test operation (e.g., matrix multiplication) to known benchmark data. A modified input data transfer cycle, shown in Figure 3.38, is used to perform the array comparison; store data command X_{15} is inhibited while compare command X_{11} is enabled. With the initiation of the array comparison operation, the benchmark data is loaded into the processing array. Once loaded, each processing cell contains like elemental data (SR A contains benchmark data, SR B contains calculated data). The array comparison is performed in parallel utilizing the 32 bit parallel comparator of each enabled cell. If calculated data and benchmark data are identical, normal operation of the computer resumes; however, if one or more faults are detected, the processing array alerts the GCU to the presence of a detected fault, and transmits the row location of the faulty cell (each processing cell comparator output within a row is ORed to the left, resulting in a single output per row). With the detection of an array fault, the GCU executive program retains system control, and initiates corrective action.

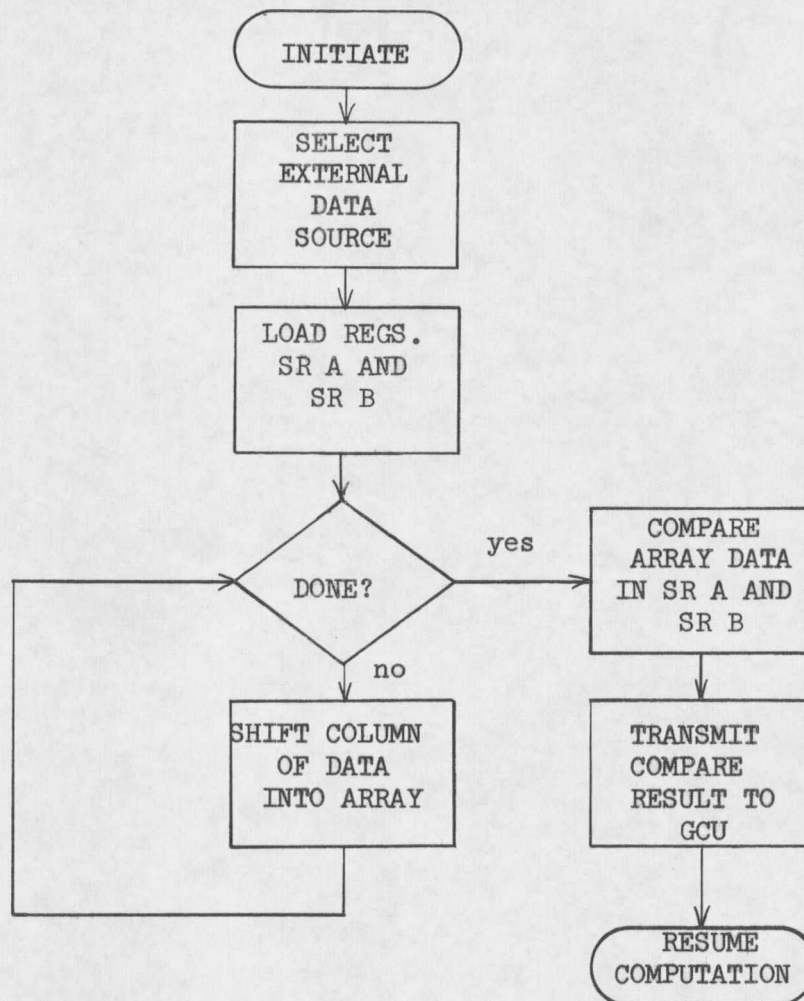


Figure 3.38 Flow Chart for the Array Comparison Cycle

Initiation of the fault detection-fault correction mode of operation is specified by the applications programmer. This mode of operation is callable after any number of operations, and when called turns control of the processing array over to the system executive program. Any operation can be used as a verification operation, but the most effective is matrix multiplication.

3.9.2.3 Fault Correction Hardware

The array-structured computer has been designed to allow processing array restructuring via software by proper selection of array control lines. This method is used to allow the processing array to maintain operation in the presence of faulty processing cells.

Upon detection of a fault within the processing array, the GCU executive program assumes system control and uses the row location of the fault to restructure all array macroinstructions. In the initialization process of the array-structured computer, all array operations have been dimensioned with enabled processing cells to implement a given operation. The dimensioning process results in the formation of a 16 bit enable mask for each macroinstructions; each bit of the mask corresponds to a row and column enable control line (bit 1 corresponds to row 1 and column 1, bit 2 corresponds to row 2 and column 2, etc.). Hence, for a given

assembly language instruction, the initialization process sets the bit values of the enable mask in the corresponding macroinstruction. When the macroinstruction is applied to the processing array, only enabled processing cells are used in implementing the given operation. This method of dimensioning the processing array was adopted in section 3.7 to allow implementation of a number of heterodimensional array operations (row, column, entire array, etc.), to allow bypass of unused or unaffected processing cells, and to allow array operation in the presence of faulty processing cells. To restructure the processing array in the presence of a fault, the enable mask bits of all macroinstruction must be changed under control of the GCU executive system. The restructuring of a 4×4 processing array with a fault in row two is illustrated in Figure 3.39.

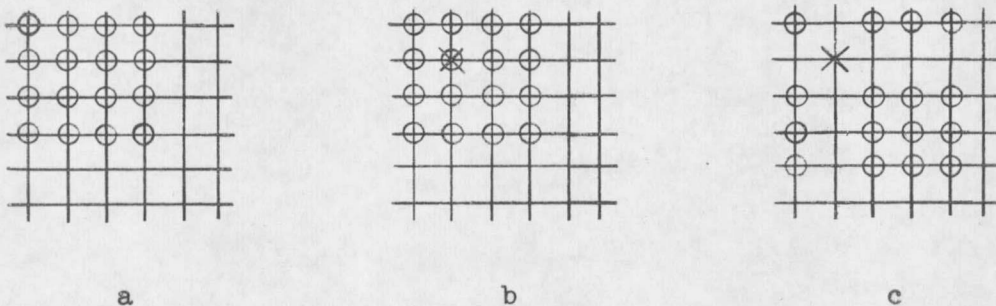


Figure 3.39 Restructuring of a 4×4 Processing Array

Figure 3.39a illustrates a fault-free 4 x 4 processing array. Figure 3.39b indicates a fault in the second row of the array. Figure 3.39c illustrates the restructured processing array having bypassed the faulty cell; the circles in Figure 3.39 indicate enabled cells and the X indicates the location of a faulty cell. An example of the mask modification necessary for a data transfer macroinstruction is shown in Figure 3.40. Original enable mask values are shown in Figure 3.40a, and modified bit values in Figure 3.40b.

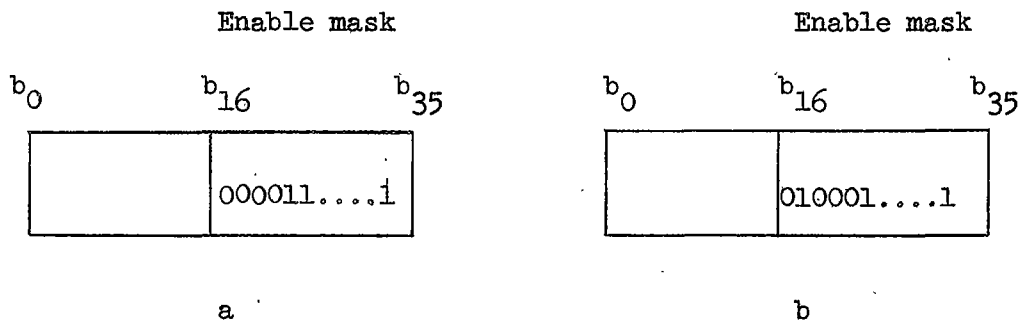


Figure 3.40 Data Transfer Macroinstruction Modification Process

This method of fault correction allows processing array operation in the presence of faulty cells at the expense of bypassing fault-free processing cells.

3.9.3 Fault-Tolerant Operation Instruction Set

The array-structured computer has been designed to allow the applications programmer to select and program the level of reliability. Instructions that check and restructure the processing array, and roll back array data to a previous valid state are listed in Table 3.14.

Table 3.14 Special Instruction Set for Fault Tolerant Operation

Instruction	Purpose
CKAR A B C	Perform the array verification by comparison on the indicated operands.
RSTR	Restructure the processing array if a fault is found; otherwise, no action.
ROLL A B C D E	If the comparison was successful, store these operands in GCU main memory. If the comparison was not successful, restore these operands for previous checked operation.

As an example of these operations, consider the PAL-1 subroutine shown in Table 3.15. For this example, the GCU would be alerted that an array comparison is to be performed on operands A, B, and C, when the CKAR instruction is found. If a fault is detected, the RSTR instruction causes the GCU system executive program to restructure the processing array utilizing fault diagnostics.

If no fault is found, the RSTR instruction initiates no GCU action, but the ROLL instruction causes the transfer of operands A, B, C, D, and E to the GCU main memory for possible use in the next rollback operation. If a fault is detected, the ROLL instruction initiates a program rollback to the last valid state, the point at which the program resumes computation.

Table 3.15 Special Instruction Set Example

```
CKAR  A B C
MULT  A B C
RSTR
ROLL  A B C D E
```

3.9.4 Fault-Tolerant Computer Structures

3.9.4.1 Introduction

Three fault-tolerant computer structures are presented in this section, each using the basic array-structured processing array developed in earlier sections of this chapter. Each succeeding structure allows more computing capability and faster effective execution time at the expense of additional hardware.

3.9.4.2 Software-Oriented Structure

The software-oriented computer organization, shown in Figure 3.41, is intended for applications where fault tolerant operation is important, but high execution speed is not. This structure consists of a processing array, the GCU, control registers, and a fast buffer memory. The operation of this structure relies on the unloading of the array to the fast buffer memory, and then testing the array for faults. The fast buffer memory has access to the GCU main memory, input-output channels, and the processing array. This buffer should be capable of cycle times of about 6 nanoseconds.

The fault-detecting and fault-correcting mode of operation is initiated by the GCU when it reaches a check array instruction (CKAR). At this time the array is unloaded in part or whole to the fast buffer. Known benchmark operands stored in the fast buffer memory are then directed to the processing array. These operands are used by the processing array in performing the required check operation. Additional benchmark data stored in the fast buffer memory is then directed into the processing array and compared with the calculated data. The result of the comparison is sent to the GCU in parallel by rows. Detection of a fault initiates corrective action specified by the system program. If no faults are found, the processing array is reloaded from the fast buffer memory and normal operation resumes.

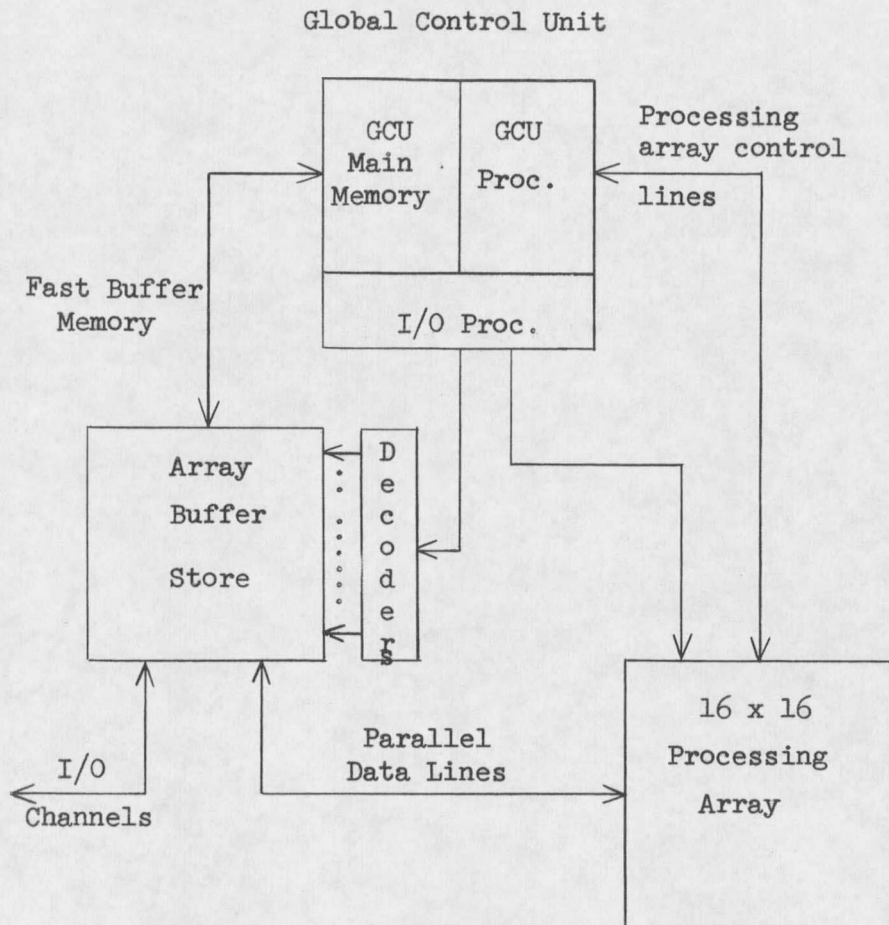


Figure 3.41 Software-Oriented, Fault-Tolerant Computer Organization

It should be noted that since the I/O processor and fast buffer memory are separate from the GCU and processing sector, they can be executing external data transfer operation independently.

3.9.4.3 Triple-Redundant Structure

The triple-redundant computer organization (see section 1.5) allows fault-tolerant operation in applications where both reliability and high execution speeds are necessary. This organization, shown in Figure 3.42, consists of a GCU, a fast buffer memory, control switching networks, and three identical processing arrays. This organization uses the three processing arrays to perform identical operations on identical data under control of the GCU. When the comparison cycle is initiated, the three arrays are compared (three comparison cycles). If no faults are detected, normal program execution resumes; if faults are detected, the three arrays vote to determine which array contains the fault. Once the faulty array is identified, array data from one of the two valid arrays is transferred to the fast buffer memory and retained until all three arrays have been restructured by the system executive program. Once restructured, the three arrays are reloaded with valid data and program execution is resumed. An important advantage of this organization is that since valid data is available in two of the three arrays, rollback to a previous valid state is

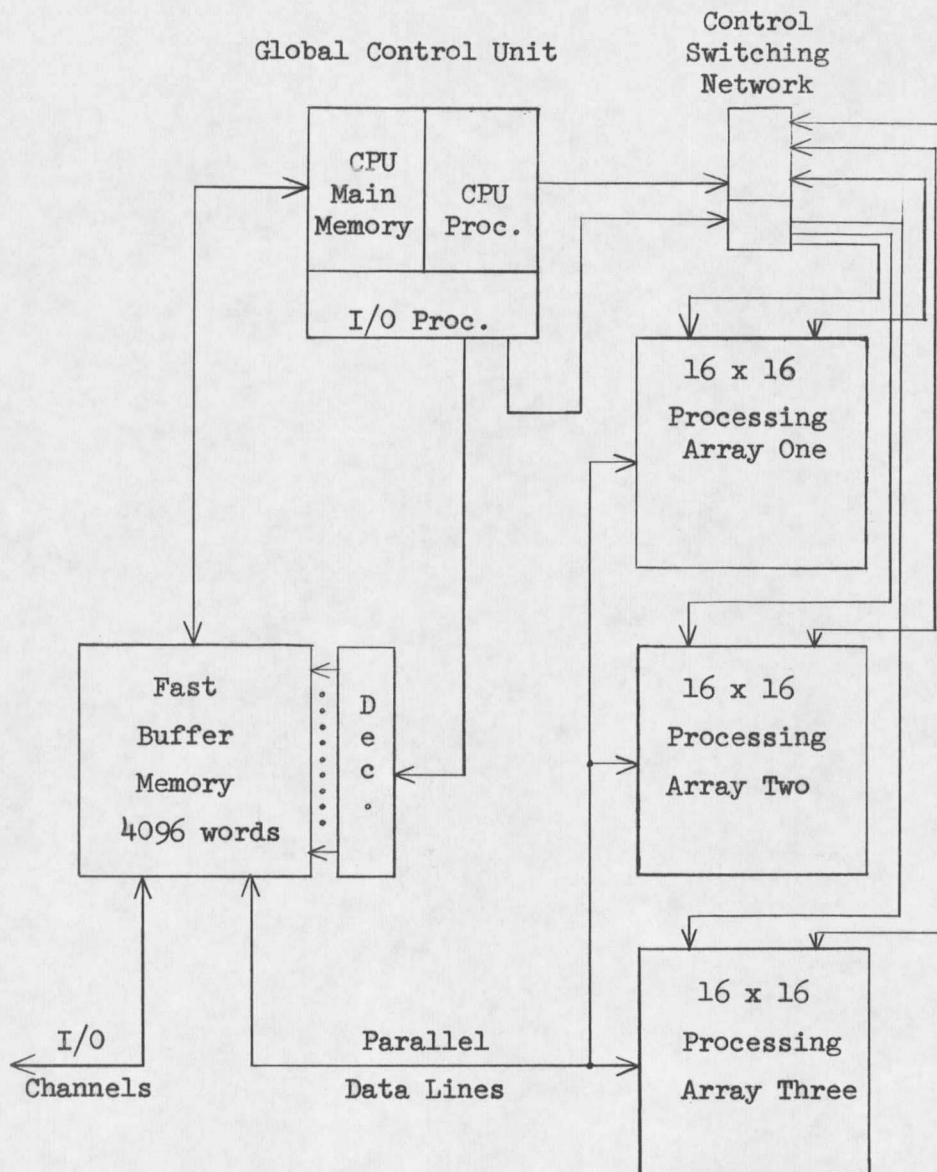


Figure 3.42 Triple-Redundant, Fault-Tolerant Computer Organization

not required. This organization can correct as many faults as there are extra rows and columns pairs in the arrays. Additional processing arrays can be provided as spares, and substituted under control of the GCU, when the restructuring capacity of the original three arrays is exceeded.

An important variation of the above method is the use of continuous comparison between arrays two and three (monitor arrays) in which these arrays are continuously validated. Hence, when array one is compared (normal comparison cycle) to array two or array three, only one comparison cycle is required (since arrays two and three are valid). If a fault is detected, all three arrays are restructured and reloaded as discussed above.

As is apparent, this computer organization can be used with a number of different methods resulting in fault-tolerant operation. Continued research is recommended in utilization of this computer organization.

3.9.4.4 Multiple-Processing Array Structure

The design of the array-structured computer incorporated the use of both local and global control, which allows the processing array to execute a given macroinstruction independent of the GCU. After issuing the macroinstruction, the global control unit is idle until the processing array has completed its operation. This idle

time is significant in that it allows the GCU to operate independently to perform such tasks as data processing, input-output operation, fault tolerant operation, and multiple processing array operation. The concept of multiple processing operation becomes important in applications where a large number of heterodimensional, matrix-oriented algorithms are to be performed. The computer can be adapted to multiple-processing array operation; a structure which lends itself well to fault-tolerant operation.

The multiple-processing array structure is shown in Figure 3.43. This structure consists of a number of identical processing sectors (shown as four), an input-output processor, fast buffer memory switching networks, and the GCU. This system is an extension of the triple-redundant structure presented in section 4.43, but differs in that each processing array is capable of independent operation under control of the GCU. This type of operation places a large systems programming burden on the GCU, which could be reaching the size of a medium scale computer. In addition, each processing array must have access to its own or shared fast buffer memory for input-output and fault-tolerant operation. Switching networks to accomplish these tasks are massive but remain feasible.

Either of the fault-tolerant modes of operation discussed in sections 3.9.4.2 and 3.9.4.3 could be implemented with this computer structure. Another method would require that the GCU

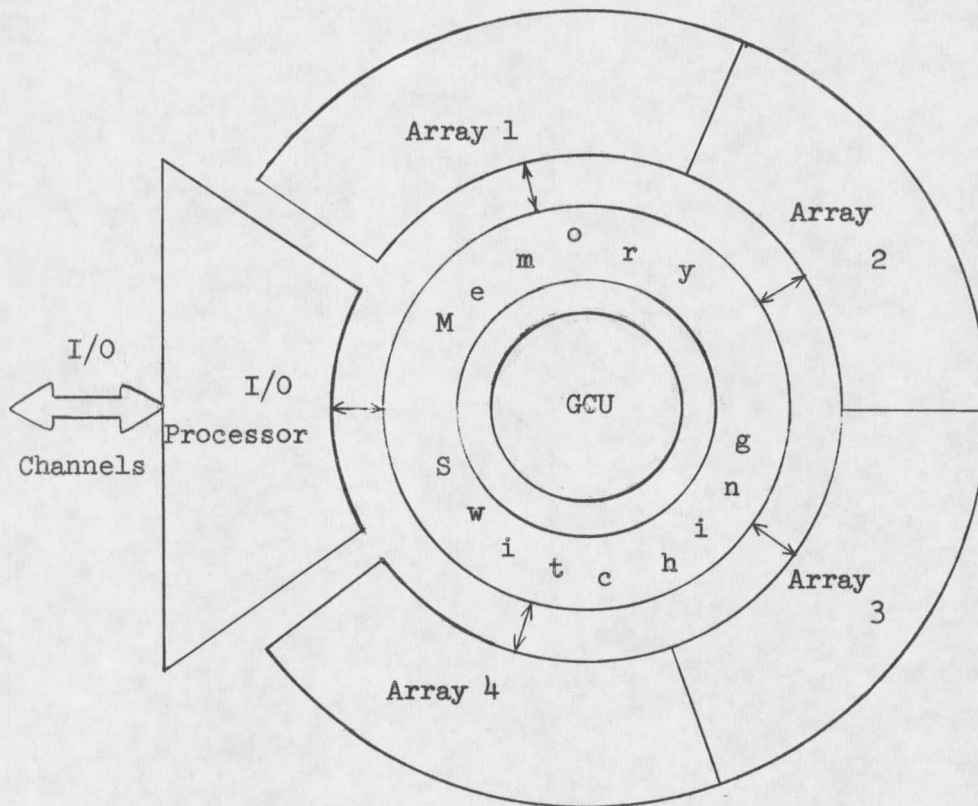


Figure 3.43 Multiple-Processing Array, Fault-Tolerant Computer Organization

designate any number of arrays as roving monitors. These monitors would be selectively attached to active processing arrays to verify acceptable operation. The computer would maintain a number of spare processing arrays to be used as replacements when required. With this method, the redundancy has decreased as the monitors are shared. All of these methods place an additional burden on the GCU system program.

In the event applications are found for this large scale computer structure, this conceptual design should be investigated further.

3.10 Implementation of a Single Processing Array Cell

3.10.1 Introduction

When consideration is given to fabrication of the array-structured computer, attention quickly focuses on the implementation of the processing array cells. Once N^2 (where N is the dimension of the array) of these identical cells have been constructed, the interconnection structure, and the attachment of exterior units and the GCU to the processing array becomes straightforward. The GCU will most likely be a commercially available general-purpose machine, and the sixteen input shift registers and the fast memory buffer will also be commercially

available. Hence, in this section, construction of a single processing array cell is presented with emphasis placed on LSI implementation.

3.10.2 Implementation of a Single Processing Cell

Each processing cell within the processing array has been segmented into a mathematical operation block (MOB) and a data transfer block (DTB). This segmentation resulted from the different functional requirements of these blocks and the intention to separate these operations for LSI implementation.

The design of a processing cell utilizes a set of selected functional cellular units which are listed in Table 3.16 by function and required number of gates. A gate is defined as having three to six transistors, and two to five resistors.

The MOB contains 10,959 gates, as shown in Table 3.17, and operates under control of the 100 nanosecond GCU clock. This block is a candidate for MOS or a combination MOS and bipolar LSI implementation. Digital LSI circuits currently available utilizing these techniques operate at speeds of 150-300 nanoseconds, but current trends (see section 1.6) indicate higher speeds are to be expected. Although future developments in LSI technology may lead to fabrication of this block on a single LSI substrate, it is not recommended that this technique be adopted. To facilitate ease of

manufacturing and replacement within the operating system, a hybrid assembly is recommended. The hybrid assembly should be composed of the three following LSI modules:

Module 1	Memory and Decoders - 5120 gates
Module 2	Parallel Multiplier - 4331 gates
Module 3	Remaining Logic and Control Circuitry - 1508 gates

Due to the complexity of the hybrid assembly interconnection structure, difficulty can be expected in fabrication of such an assembly. However, this technique appears to be well within the capability of current technology.

The second segment of a processing cell is the data transfer block which should be fabricated separately utilizing high speed digital circuits. The only LSI implementation technique capable of the 6 nanosecond clock period requirement is bipolar LSI circuitry. Bipolar LSI technology currently offers 10-20 nanosecond operation with current trends indicating even faster operation (see section 1.6). The DTB should be implementable on a single LSI chip containing the 667 gates shown in Table 3.18. Such implementation appears to be feasible with current technology and is recommended for implementation of the DTB. Interconnection and control lines to the DTB should be short and utilized low capacitance connecting planes to attain maximum speeds.

Table 3.16 Gates Required for the Selected Cellular Logic Units

Logical Unit	Required Number of Gates
AND	1
OR	1
EXOR	1
ADD ONE CELL	2
COMPLEMENTOR CELL	2
COMPARATOR CELL	4
ADDER CELL	5
ADDER/ SUBTRACTOR CELL	6
GENERAL REGISTER CELL	5

Three methods are available for LSI implementation of the digital logic circuits comprising the MOB and the DTB; conventional, integrated circuit modules (MSI, LSI), custom-designed, integrated circuit modules (LSI), and cellular array modules (LSI). Each of these methods is a reasonable choice for implementation, but none is a clear and distinct choice over the others. The first of these methods uses 'off the shelf' integrated circuits to implement the logical functions, utilizing a common interconnection backplane.

This method has advantages of lowest cost and utilization of proven logic designs at the expense of larger physical size and slightly slower speeds than the two other methods. The second method uses design and fabrication of custom LSI modules that perform the required logical functions. This method has advantages of speed, small physical size, and logic designed for a specific application at the expense of high cost and inflexibility in utilization. The third method requires design and fabrication of a generalized cellular LSI array (or several different cellular arrays) that can be easily modified to produce all required logical functions. Such cellular arrays could be patterned after the work done by Minnick (4,10,11) to produce flexible logic modules. Although the first two methods could be used, this method appears to be the best choice for implementation of the logical functions of the processing cells for the following reasons: all modules within the MOB and the DTB are designed utilizing the select set of logical cells shown in Table 3.1; most modules are regular and repetitive in structure; the two largest modules (memory-5120 gates, multiplier-4331 gates) are very regular in organization and have repetitive interconnecting structures; cellular arrays provide small physical size and moderate speed. Design and fabrication of such cellular array modules usually requires large initial costs and time expended in design, but such implementation becomes cost competitive when large numbers of

Table 3.17 Gates Required in the MOB

Module	Number of Logic Gates Required						Total
	Complementor Cell	AND Cell	OR Cell	GR Cell	+/- Cell	Comparator Cell	
MOB Command Generation		54	11	155			220
Memory		1536	1024	2560			5120
Decoder		64					64
Data Source Selector and Router		96	32				128
Adder-Subtractor		17	33	330	191	228	799
Multiplier	144	736	79	--	3372		4331
Normalizer and Data Link		99	33	165			297
TOTAL	144	2602	1212	3210	3563	228	10,959

Table 3.18 Gates Required in the DTB

Unit	Number of Logic Gates Required						Total
	Complementor Cell	AND Cell	OR Cell	GR Cell	+/- Cell	Comparator Cell	
Shift Registers				320			320
Comparator						128	128
Control Logic		226	3				229
TOTAL		226	3	320		128	677

identical cells (such as are used in the designed processing array consisting of N^2 identical cells) are required. A problem with large cellular arrays of this type is that by forming a modified array from a generalized array to satisfy certain logic functions, cells within the array are inefficiently used; the percentage of unused cells can range from 30-90%. However, advances in LSI technology (i.e., number of gates per chip may not only negate this consideration but allow for additional cells which perform other array functions, e.g., fault detection, parity checking, TMR.

Although the development of cellular LSI arrays has lagged behind predictions in recent years, it is recommended that cellular array fabrication of the cells within the processing array be given strong consideration.

CHAPTER 4

SUMMARY AND RECOMMENDATIONS

4.1 Introduction

The result of this research effort is the design of an array-structured computer capable of fault-tolerant operation. This computer is a general-purpose machine specially organized to efficiently implement matrix-oriented algorithms. This computer is useful in situations where high-speed operation is required, but where flexibility and size requirements prevent the use of special-purpose or high-speed, general-purpose computers. This research effort proposes a computing system which should be implementable within five years.

This chapter contains a summary of important results and conclusions, and a section containing recommendations for future research in the areas of design and use of array-structured computers.

4.2 Summary

4.2.1 Parallel Assembly Language Program

An assembly language instruction set consisting of 58 instructions is selected for the array-structured computer based on the usual operations required for matrix operations and some special operations resulting from previous research. The instruction set is embedded in a computer program that simulates the operation of an array-structured computer (PAL-1) and yields execution time information. Design considerations derived from the simulation program are listed and

considered in the design of the array-structured computer. The two salient design considerations investigated are the transfer of data between cells of the processing array (serial versus parallel data transfer), and the use of parallel arithmetic operations. By using parallel data transfer within the processing array, a 32:1 speed improvement for data transfer operations and a 4.5:1 speed improvement for a typical matrix oriented algorithm (Kalman filter algorithm) are realized in comparison to serial data transfer. By using a parallel multiplier a speed improvement of 16:1 is realized in comparison to serial multiplication.

The discrete Kalman filter algorithm is used as an example of the matrix class of algorithms and is implemented with the simulation program. Results compare favorably with a conventional FORTRAN IV simulation of the same algorithm. Assuming a 100 nanosecond clock period, the simulated array-structured computer executed one iteration of a six state variable, three observable Kalman filter algorithm in 2681 clock periods (268.1 microseconds) resulting in 3730 updates per second of the estimated state vector. Inversion of a 3 x 3 matrix by the Gauss-Jordan method requires 500 clock periods (50 microseconds).

4.2.2 Array-Structured Cellular Computer

Based on design considerations resulting from the simulation program and the survey of array-structured computers, system organization and logical design of the array-structured computer is presented. System organization consists of a global control unit, units exterior to the processing array, and a rectangular array of identical processing cells. The global control unit is a 36 bit word 100 Mhz general-purpose computer with a modular organization, including a separate input-output processor. The exterior units are the data transfer microprocessing unit used to control data transfer operations within the processing array and between the processing array and other units; the dynamic microprocessing unit used to facilitate execution of special matrix operations; and the input-output unit used to form a high-speed data communication link between the processing array, GCU and input-output channels. Execution of matrix operations by the processing array utilizes a combination of local and global control.

The processing array consists of a rectangular array of N^2 identical processing cells (where N is the dimension of the array)*.

* Although N was chosen as 16 for the computer organization presented, there is no particular significance attached. N can be any suitable integer.

Matrix data is stored in the processing array with the (i,j) element being stored in the (i,j) processing cell. Processing array cells are enabled by rows and columns so that the processing array is dimensioned to the size of the operation being executed. All processing cells receive the same macroinstruction and operate in parallel.

The processing cell is segmented into a mathematical operation block (MOB) and a data transfer block (DTB), allowing each to operate with a different clock period; 100 nanoseconds and 6 nanoseconds, respectively. The MOB receives a macroinstruction from the GCU or dynamic microprocessing unit and generates the correct sequence of microinstructions to implement the given localized mathematical operation. All processing-array, data-transfer operations are accomplished by the data transfer block. Each processing cell DTB is connected to the cell memory module and the horizontal and vertical adjacent data transfer segments. These segments are 32 bit parallel data lines with bottom to top and right edge to left edge array wrap around. Processing cells are connected to the uniform data-transfer, interconnecting structure only when necessary to implement a data transfer operation. With the above data-transfer, interconnecting structure, matrix data can be rotated, skewed, broadcast, transposed, and input-output.

Since all cells in the processing array are processing data in parallel when performing matrix operations, significant improvements in speed over conventionally-organized computers is achieved with the array-structured organization. Implementation of a typical matrix-oriented algorithm (i.e., Kalman filter algorithm) is a minimum of N^2 times faster than possible with a sequential machine with the same clock period (where N is the dimension of the algorithm). It is quite likely that this factor of improvement will be even larger since a sequential implementation of the same algorithm would require more time for bookkeeping and indexing of elemental data..

Segmentation of the processing cell allows separate LSI implementation of each block. A hybrid LSI assembly is recommended for the 10,959 gates of the MOB consisting of three LSI modules--memory and decoders, parallel multiplier, and the remaining circuitry. The method of LSI implementation should be MOS or a combination MOS/bipolar LSI circuit modules utilizing cellular LSI arrays. A single bipolar LSI module to contain the 667 gates of the DTB is suggested. The fabrication of a single processing cell implementing the above recommendations should be possible within five years.

4.2.3 Fault-Tolerant Computer Organizations

Detection of faults within the processing array of the array-structured computer is accomplished by a parallel array comparison of calculated data to known benchmark data. The global control unit initiates fault-detection and fault-correction cycles under control of the system executive program. The executive program detects the presence of special instruction provided for fault-tolerant operation. This instruction set consists of CKAR, which initiates the array comparison; RSTR, which initiates the restructuring of the processing array to route around detected faults; and ROLL, which loads the processing array memory with the last known valid data and starts the program execution from that point. These instructions allow the applications programmer to design into the program the desired level of reliability.

Three fault-tolerant computer organizations are presented; a software-oriented organization for moderate-speed, fault-tolerant operation; a triple-redundant organization for high-speed, fault-tolerant operation; and a multiple-processing array organization for diverse algorithm applications, some of which might require fault-tolerant operation.

4.3 Recommendations for Future Research

The system organization and logical design of the array-structured cellular computer capable of fault-tolerant operation forms an investigation base for continued research.

Application programming should begin immediately utilizing PAL-1. Suggested areas of applications are matrix inversion techniques, search techniques, dynamic programming, optimal estimation, etc. Due to timing information provided by PAL-1, comparisons can be made with serial implementation of these algorithms. If necessary, additional assembly language instructions or simulation routines should be added to PAL-1. Special consideration should be given to incorporation of processing-array folding methods: problems whose dimensionality is higher than the dimension of the processing array.

Logical design of the array-structured computer has been conducted to a level from which a processing cell and associated control signals should be either logically simulated or bread-boarded with available integrated circuit modules (a discrete version rather than an LSI version). In this manner, logical operation of the designed computer can be verified.

A global control unit should be selected according to the guide lines presented in section 3.4. This unit can be used to investigate interfacing problems between the input-output channels,

global control unit, and processing array. With the availability of this unit, a system executive program could be developed and applications programming could begin.

Considerable research remains to be accomplished in the area of fault detection and fault correction within the processing array. Using section 3.9 as an investigation base, research and development should be conducted on single cell substitution and introduction of local fault detection and correction methods, e.g., parity checking, TMR. If the global control unit is available, the fault detection and correction programs can be written and verified. Since the suggested method of fault detection uses an array comparison, research should be conducted on failure modes and their detection within the processing array; i.e., the effectiveness of fault detection by array comparison. PAL-1 can be used to investigate fault modes within the processing array. To fully investigate fault-tolerant operation, failure modes of units exterior to the processing array should be investigated.

Detailed research should be conducted on developing methods for detecting parallelisms in algorithms. Such methods lead to efficient implementation of such algorithms on this array-structured computer. Efficient schemes for parallel matrix inversion should be developed using the array-structured computer instruction set.

This research effort has resulted in an array-structured computer that is extremely fast and efficient in implementing matrix-oriented algorithms. This computer will yield computational improvements in problems dealing with sampled data control systems (e.g., digital filtering, optimal controllers, adaptive controllers), vehicle navigation, phased radar control computations, and sonar receiving array computations.

APPENDIX A

THE KALMAN FILTER ALGORITHM

A brief outline of the Kalman filter algorithm is presented in this appendix with emphasis placed on the linearized equations required for mechanization of the algorithm in PAL-1. While both discrete and continuous filter algorithms exist, the discrete form is the clear and natural choice for mechanization with a digital computer. For a complete development of the Kalman filter algorithm and related filters, see Liebelt (36) and Lee (37).

The discrete Kalman filter is a linear digital filter whose input is a set of observation variables and whose output is an estimate of another related set of variables. In many applications, and in the example of the sequel, the observation variable set is a sequence of observations of a linear dynamic system, and the output is a sequence of estimates of system state. The Kalman equations themselves are based on observations of a linear (and possibly time varying) system combined with knowledge of the system dynamics. The need to estimate is a consequence of the corruption of ideal measurements by noise (assumed additive) and excitation of the dynamic system by random forcing functions in addition to known inputs. In equation form

$$x_{k+1} = \phi_{k+1,k} x_k + \Gamma_k (u_k + w_k)$$

$$z_k = H_k x_k + v_k$$

where x_k is the state vector at the k th sampling instant

$\phi_{k+1,k}$ is the system transition matrix

Γ_k is the input distribution matrix

u_k is a known control vector

w_k is a random vector with known mean and variance (herein assumed zero)

z_k is the observation vector

H_k is the observation matrix

v_k is a random vector with known mean and variance (herein assumed zero)

Operation of the Kalman filter is described by the set of recursion relations given below:

$$\hat{x}_{k+1,k} = \phi_{k+1,k} x_{k,k}^* + \Gamma_k u_k$$

$$x_{k+1,k+1}^* = \hat{x}_{k+1,k} + G_{k+1} (z_{k+1} - H_k \hat{x}_{k+1,k})$$

$$G_k = \Sigma_{k,k-1} H_k^T (H_k \Sigma_{k,k-1} H_k^T + R_k)^{-1}$$

$$\Sigma_{k,k} = (I - G(k)H(k)) \Sigma_{k,k-1}$$

$$\Sigma_{k+1,k} = \phi_{k+1,k} \Sigma_{k,k} \phi_{k+1,k}^T + \Gamma_k Q_k \Gamma_k^T$$

where $x_{k+1,k}$ is the optimal prediction of x_{k+1} after k observations

$x_{k+1,k+1}^*$ is the optimal estimate of x_{k+1} after $k+1$ observations

G_{k+1} is the optimal gain matrix to provide the above estimates at $k+1$ observations

$\Sigma_{k,k}^*$ is the covariance matrix of error in $x_{k,k}^*$ at k observations

$\Sigma_{k,k-1}$ is the covariance matrix of error in $x_{k,k-1}$

R_k and Q_k are covariance matrices of w_k and u_k , respectively, at k observations

All pertinent sections of an overall system-observation-filter grouping are shown in Figure A-1. It is interesting to note that a noise-free model of the system is imbedded in the filter proper.

The Kalman filter is optimal in several contexts. If consideration is restricted to linear filters (output sequence is a linear combination of present observation and past state) with only means and variances of w_k and v_k sequences known, the Kalman form exhibits optimality in the sense of minimizing the trace of the covariance matrix ($\Sigma_{k,k}$). Secondly, if gaussian distri-

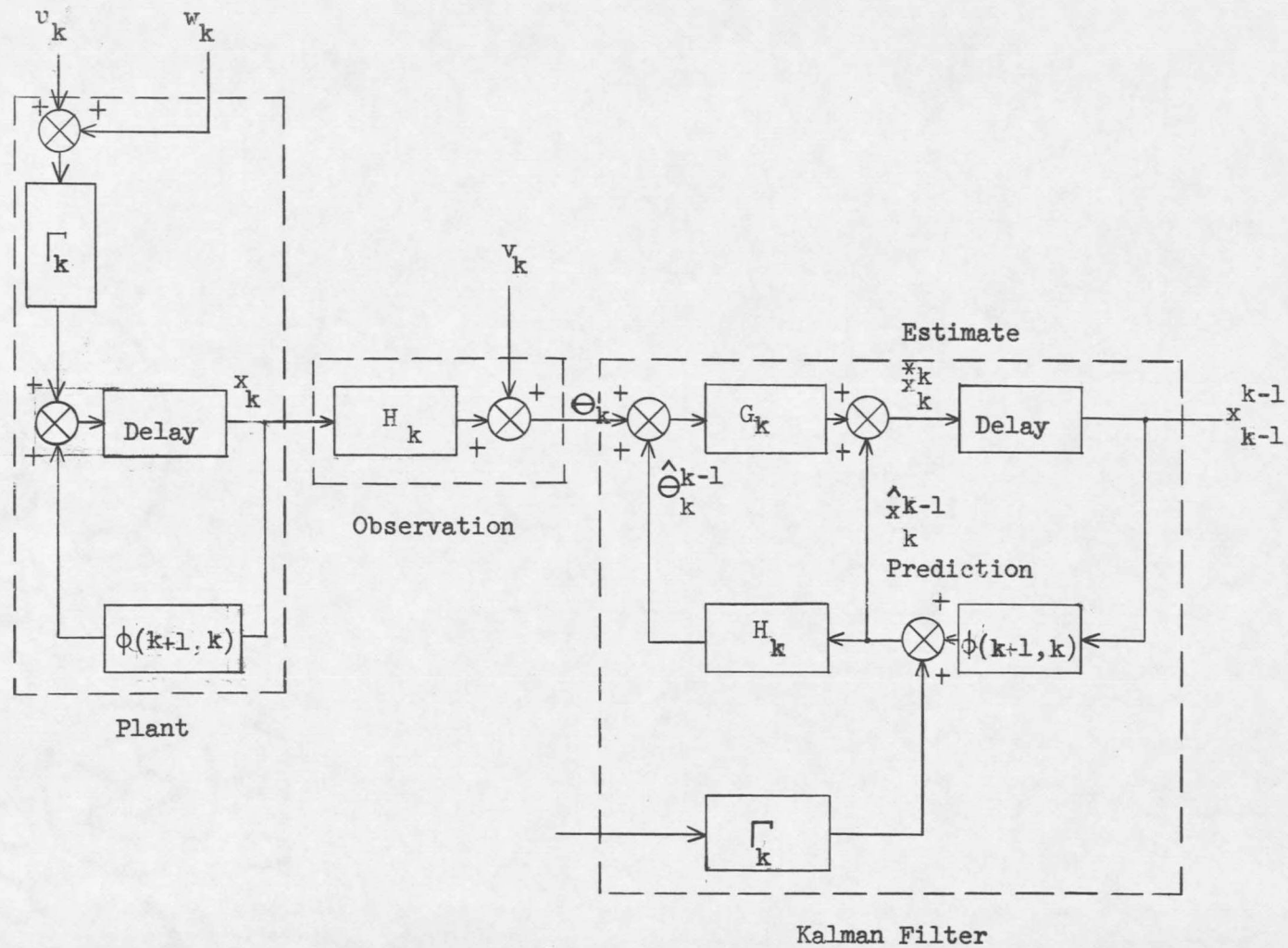


Fig. A-1 Kalman Filter Problem Block Diagram

butions for the random sequences w_k and v_k are assumed, the Kalman form is the minimum variance estimator of any class, linear or nonlinear. Thirdly, it is the result of a recursive least squares filter with known means and variances of random sequences.

As an example of Kalman filter operation and to generate a sequence of typical values against which to check PAL-1, a three-dimensional tracking problem is implemented. A state vector consisting of positions and rates was defined in an orthogonal coordinate system as

$$\bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \Delta \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{bmatrix}$$

A simulated target with maneuvering capability was tracked from a fixed station through three radar observables: Θ (bearing in degrees clockwise from North); Φ (azimuth or elevation angle); and R (range). Using an x_1, x_3, x_5 coordinate system (x_1 = east, x_3 = north, x_5 = altitude), noisy observations are

$$\begin{bmatrix} \theta \\ \phi \\ R \end{bmatrix} = \begin{bmatrix} \tan^{-1} \frac{x_3}{x_1} \\ \tan^{-1} \frac{x_5}{\sqrt{x_1^2 + x_2^2}} \\ \sqrt{x_1^2 + x_3^2 + x_5^2} \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

This observation system is nonlinear and must be linearized in some fashion. Local linearization about the last state vector prediction yields $\frac{\partial z}{\partial x} = H$ as the 3×6 matrix

$$\frac{\partial z}{\partial x} = H_k = \begin{bmatrix} \frac{\hat{x}_3}{\hat{D}^2} & 0 & -\frac{\hat{x}_1}{\hat{D}^2} & 0 & 0 & 0 \\ -\frac{\hat{x}_1 \hat{x}_5}{\hat{R}^2 \hat{D}} & 0 & -\frac{\hat{x}_3 \hat{x}_5}{\hat{R}^2 \hat{D}} & 0 & \frac{\hat{D}}{\hat{R}^2} & 0 \\ \frac{\hat{x}_1}{\hat{R}} & 0 & \frac{\hat{x}_3}{\hat{R}} & 0 & \frac{\hat{x}_5}{\hat{R}} & 0 \end{bmatrix}_k$$

where $\hat{R} = \sqrt{\hat{x}_1^2 + \hat{x}_3^2 + \hat{x}_5^2}$ and $\hat{D} = \sqrt{\hat{x}_1^2 + \hat{x}_3^2}$.

The $\hat{x}_i$ notation indicates evaluation of the matrix at the last state vector prediction.

The locally linearized system equations implemented in PAL-1 are given below:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}_{k+1} = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}_k + \begin{bmatrix} \frac{T^2}{2} & 0 & 0 \\ T & 0 & 0 \\ 0 & \frac{T^2}{2} & 0 \\ 0 & T & 0 \\ 0 & 0 & \frac{T^2}{2} \\ 0 & 0 & T \end{bmatrix} \begin{bmatrix} w_1 + u_1 \\ w_2 + u_2 \\ w_3 + u_3 \end{bmatrix}$$

$$\begin{bmatrix} \delta \Theta \\ \delta \Phi \\ \delta R \end{bmatrix}_k = H \begin{bmatrix} \delta x_1 \\ \delta x_2 \\ \delta x_3 \\ \delta x_4 \\ \delta x_5 \\ \delta x_6 \end{bmatrix}_k + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_k$$

It should be noted that the linearization of the observation system is valid in computing the optimal Kalman gain factor. In addition, the covariance matrices are approximations as a result of this linearization (H matrix) of a non-linear observation system.

APPENDIX B

PERTINENT MATRIX ALGORITHMS

This appendix pertains to the two fundamental matrix operations of multiplication and inversion. The pertinent matrix algorithms selected for implementing these fundamental operations are based on the work done by Cannon (21) and Rudberg (35). The selection of these matrix algorithms necessitates the use of a specially-structured data transfer interconnection structure for the array-structured computer that efficiently performs the data transfer operations of rotation, skewing, transposing, and broadcasting of matrix data.

The selected matrix algorithms presented below assume a processing array of N^2 identical processing modules. Each module is composed of sixteen 64 bit words of memory, two 64 bit shift registers (operand 1 and operand 2), a 64 bit accumulator, and a serial adder.

Matrix Multiplication:

The following two algorithms perform matrix multiplication according to the matrix equation $A = B \times C$ where B is $r \times s$, C is $s \times t$; hence, A is $r \times t$ and $r, s, t \leq n$ is assumed throughout.

Algorithm 1:

With this algorithm for matrix multiplication, re-arrangement of rows and columns of matrices is required. Initially, memory locations of matrices A , B , and C are cleared, and the matrices to be multiplied are loaded into memory locations B and C . The data in matrices B and

C is then shifted according to the following rules:

- A. Elements of the i^{th} row of B are shifted left $i-1$ columns (with end around shift) for $i=1,2,\dots,n$ (skew left).
- B. Elements of the j^{th} row of C are shifted up $j-1$ rows (with end around shift) for $j=1,2,\dots,n$ (skew up).

Once the matrices B and C have been rearranged, operands 1 are loaded with the element data of B, and operands 2 are loaded with the data of C. Element multiplication is performed as each module multiplies operand 1 by operand 2, the product appearing in the accumulator. Element multiplication is performed simultaneously in all modules, so that one multiplication time period (mtp) is required. The product is then stored in memory location for matrix A. Now the data comprising matrix B is shifted right one module with end around shift. In a similar fashion, the data in matrix C is shifted down one module with end around shift. Element multiplication is repeated with the product being stored in the accumulator. Operand 1 is then loaded with the accumulator data while operand 2 is loaded with matrix A data. Element addition is then performed with the result being stored in matrix A. The cycle is repeated N times with the product appearing in matrix A.

An example is given below for a 3 x 3 matrix multiplication where the matrices B and C are to be multiplied.

$$B = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} \quad C = \begin{bmatrix} c_1 & c_4 & c_7 \\ c_2 & c_5 & c_8 \\ c_3 & c_6 & c_9 \end{bmatrix}$$

Operands 1 and 2 are loaded with B and C, and matrix A is zeroed.

$$B = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} \quad C = \begin{bmatrix} c_1 & c_4 & c_7 \\ c_2 & c_5 & c_8 \\ c_3 & c_6 & c_9 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

B and C are shifted using the rules given. B is element-multiplied by C and the result is added to A.

$$B = \begin{bmatrix} b_1 & b_2 & b_3 \\ b_5 & b_6 & b_4 \\ b_9 & b_7 & b_8 \end{bmatrix} \quad C = \begin{bmatrix} c_1 & c_5 & c_9 \\ c_2 & c_6 & c_7 \\ c_3 & c_4 & c_8 \end{bmatrix}$$

$$A = \begin{bmatrix} b_1 c_1 & b_2 c_5 & b_3 c_9 \\ b_5 c_2 & b_6 c_6 & b_4 c_7 \\ b_9 c_3 & b_1 c_4 & b_8 c_8 \end{bmatrix}$$

B is shifted right and C is shifted down one module.

$$B = \begin{bmatrix} b_3 & b_1 & b_2 \\ b_4 & b_5 & b_6 \\ b_8 & b_9 & b_7 \end{bmatrix}$$

$$C = \begin{bmatrix} c_3 & c_4 & c_8 \\ c_1 & c_5 & c_9 \\ c_2 & c_6 & c_7 \end{bmatrix}$$

$$A = \begin{bmatrix} b_1 c_1 & b_2 c_5 & b_3 c_9 \\ b_5 c_2 & b_6 c_6 & b_4 c_1 \\ b_9 c_3 & b_7 c_4 & b_8 c_8 \end{bmatrix}$$

B is element multiplied by C and the result is added to A.

$$B = \begin{bmatrix} b_3 & b_1 & b_2 \\ b_4 & b_5 & b_6 \\ b_8 & b_9 & b_7 \end{bmatrix}$$

$$C = \begin{bmatrix} c_3 & c_4 & c_8 \\ c_1 & c_5 & c_9 \\ c_2 & c_6 & c_7 \end{bmatrix}$$

$$A = \begin{bmatrix} b_1c_1 + b_3c_3 & b_2c_5 + b_1c_4 & b_3c_9 + b_2c_8 \\ b_5c_2 + b_4c_1 & b_6c_6 + b_5c_5 & b_4c_7 + b_6c_9 \\ b_9c_3 + b_8c_2 & b_7c_4 + b_9c_6 & b_8c_8 + b_7c_7 \end{bmatrix}$$

B is shifted right and C is shifted down one module.

$$B = \begin{bmatrix} b_2 & b_3 & b_1 \\ b_6 & b_4 & b_5 \\ b_7 & b_8 & b_9 \end{bmatrix} \quad C = \begin{bmatrix} c_2 & c_6 & c_7 \\ c_3 & c_4 & c_8 \\ c_1 & c_5 & c_9 \end{bmatrix}$$

$$A = \begin{bmatrix} b_1c_1 + b_3c_3 & b_2c_5 + b_1c_4 & b_3c_9 + b_2c_8 \\ b_5c_2 + b_4c_1 & b_6c_6 + b_5c_5 & b_4c_7 + b_6c_9 \\ b_9c_3 + b_8c_2 & b_7c_4 + b_9c_6 & b_8c_8 + b_7c_7 \end{bmatrix}$$

B is element multiplied by C and the result is added to A.

$$B = \begin{bmatrix} b_2 & b_3 & b_1 \\ b_6 & b_4 & b_5 \\ b_7 & b_8 & b_9 \end{bmatrix} \quad C = \begin{bmatrix} c_2 & c_6 & c_7 \\ c_3 & c_4 & c_8 \\ c_1 & c_5 & c_9 \end{bmatrix}$$

$$A = \begin{bmatrix} b_1^c c_1 + b_3^c c_3 + b_2^c c_2 & b_2^c c_5 + b_1^c c_4 + b_3^c c_6 & b_3^c c_9 + b_2^c c_8 + b_1^c c_7 \\ b_5^c c_2 + b_4^c c_1 + b_6^c c_3 & b_6^c c_6 + b_5^c c_5 + b_4^c c_4 & b_4^c c_7 + b_6^c c_9 + b_5^c c_8 \\ b_9^c c_3 + b_8^c c_2 + b_7^c c_1 & b_7^c c_4 + b_9^c c_6 + b_8^c c_5 & b_8^c c_8 + b_7^c c_7 + b_9^c c_9 \end{bmatrix}$$

A now contains the matrix product of the original B and C matrices.

Algorithm 2:

In contrast to the usual formation of elements of A as inner products of rows of B and columns of C, this algorithm forms outer products of columns of B and rows of C, their sum resulting in A. No rearrangement of rows and columns is necessary with this algorithm. First the memory locations for matrix A are cleared. The first column of B is fetched from storage and placed in operand 1 of column 1. Then these contents are broadcast over the rows into operand 1; i.e., all operand 1 registers in row i contain b_{i1} . The first row of C is fetched from storage and placed in operand 2 of row 1, and these contents are broadcast over the columns into operand 2; i.e., all operand 2 registers in column j contain c_{1j} . Element multiplication is performed as each module multiplies operand 1 by operand 2, the product appearing in the accumulator. Element multiplications are carried out simultaneously so that one mtp is required. Contents of memory for matrix A are loaded into operand 1, accumulator contents are loaded into operand

2 and element addition performed. The accumulator contents are then transferred to matrix A storage locations. Then column 2 of B is broadcast into operand 1, row 2 of C is broadcast into operand 2 and the sequence repeats until the s columns of B and s rows of C are exhausted. At this time, matrix A contains the product of $B \times C$.

The second algorithm is preferred in that the matrix size determines the number of times the sequence of operations is repeated, whereas in algorithm 1, the array size determines the number of times the sequence is repeated; (i.e., for a 3×3 matrix multiplication in a 6×6 array, the sequence must be repeated 6 times for algorithm 1, and 3 times for algorithm 2).

Matrix Inversion:

Several inversion methods have been investigated, including Gauss-Jordan, the escalator or bordering method, and the square root method. An algorithm based on the Gauss-Jordan method is presented in a form appropriate for implementation on an array-structured machine.

Gauss-Jordan inversion of an $n \times n$ matrix A is a sequence of elementary row operations reducing the original matrix to an identity matrix. The sequence of elementary row operations is recorded upon an identity matrix which is converted to A^{-1} when operations are complete. The Gauss-Jordan method is well suited to a parallel machine with a capability of horizontal and vertical broadcasting of data. No matrix

scratch pad area is required. For a general discussion of the Gauss-Jordan method (also the escalator or bordering method) that is not allied to a specific machine organization, see Pease (36).

It is assumed that an $n \times n$ matrix presently in storage location A is to be inverted. a_{ij} is in the storage register of the i,j module. An n -fold repetition of the following sequence is required as the index i varies from 1 to n . A global control unit (GCU) is assumed available to perform rapid element division.

1. Compute $1/a_{ii}$ in the GCU. A measurable time is required for the GCU to perform division. During this time, negate and broadcast column i of A into operand 1 of all modules; i.e., row k contains $-a_{ki}$ in all operand 1 registers. Then convert column i of A to the i^{th} column of an identity matrix (1 in the (i,i) location, 0 elsewhere). This effectively introduces the identity matrix one column at a time throughout the computation, eliminating the need for extra storage. Finally, load this modified A matrix into operand 2 of all modules.
2. Return $1/a_{ii}$ from the GCU and broadcast over operand 1 of row i .
3. Perform element multiplication; i.e., each module multiplies its two operands, placing the result in the accumu-

lator. This normalizes row i and produces unused results in other rows.

4. Broadcast row i accumulators vertically over operand 2. Store row i accumulator contents in the A matrix storage. Row i storage is correct and will not be changed again in this cycle.
5. Perform element multiplication.
6. In each module, load operand 2 with accumulator contents and operand 1 with the A matrix.
7. Perform element addition. This provides new rows of the partial inverse.
8. Store accumulator contents in A matrix storage, inhibiting storage of row i .

A numerical example of a matrix inverse by the Gauss-Jordan method using the A register is now given. The original matrix is

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 6 \end{bmatrix}$$

and is found in A matrix storage. After one row operation in Gauss-Jordan inversion the modified matrix is

$$\begin{bmatrix} 1 & 1/2 & 1/2 \\ 0 & 1/2 & 3/2 \\ 0 & 3/2 & 11/2 \end{bmatrix}$$

and the corresponding record on an identity matrix is

$$\begin{bmatrix} 1/2 & 0 & 0 \\ -1/2 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix}$$

It is known that the first column of the modified matrix will be the identity column and that all but the first column of the record matrix will be the remains of an identity matrix. Therefore, only the informative columns are retained in A matrix storage.

Record of one row operation	↓	$\begin{bmatrix} 1/2 & 1/2 & 1/2 \\ -1/2 & 1/2 & 3/2 \\ -1/2 & 3/2 & 11/2 \end{bmatrix}$	↓	Remaining to be processed
--------------------------------	---	--	---	------------------------------

Results of the second row operation are

Record of two
row operations

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 2 & 3 \\ 1 & -3 & 1 \end{bmatrix}$$

Remaining to be
processed

and those of the third

Record of three
row operations
This is A^{-1} .

$$\begin{bmatrix} 2 & -4 & 1 \\ -4 & 11 & -3 \\ 1 & -3 & 1 \end{bmatrix}$$

Remaining to be
processed: None

All of the above results appear successively in A matrix storage. After n repetitions of the sequence, the A storage location contains A^{-1} . Significant operations are n divisions and $2n$ multiplications. For division requiring the same length of time as multiplication, this is $3n$ mtp's. Serial computation requires n division, and $n^2(n-1)$ multiplications for a total of $n^3 - n^2 + n$ mtp. The break-even point is a 2×2 matrix. The linear increase of mtp with n in the parallel processor indicates a reasonable use of the n^2 modules; however, the multiplier of 3 can be reduced to 2 by an increase in module capability to include full division. Reduction of the multiplier to unity seems difficult in view of the required computation of the term $(a_{kj}a_{ij}/a_{ii})$ which has one multiplication and one division to be performed.

APPENDIX C

LISTING OF THE PARALLEL
ASSEMBLY LANGUAGE SIMULATION PROGRAM

The program listed in this appendix is written for the Xerox Data System Sigma 7 computer under the batch processing monitor (BPM). This program is comprised of 4 major routines: a FORTRAN IVH main line routine, a FORTRAN IVH input-output routine, a SYMBOL (symbolic assembly language for the Sigma 7 computer) main manipulation routine, and a SYMBOL base conversion routine. In line SYMBOL is used in the FORTRAN IVH input-output program. Parameters for the program are given below:

Memory Requirements:

FORTRAN IVH	-	169 words
SYMBOL	-	2169 words
Total	-	2338 words

Program Size:

3155 cards
92 processor pages output

Limits:

Time	-	10 minutes
Pages	-	300 pages

Device Assignment:

Input	-	card reader (CRA03)
Output	-	line printer (LPA02)
Temporary		
Storage	-	rapid access disk (RAD)

the RAD uses the following options:

- FILE,TEMP - A file designated TEMP
- KEYED - location of each record in the file is determined by an explicit designator
- DIRECT - specifies next record to be accessed is to be determined by an explicit designator (key)
- OUTIN - specifies the file input-output mode
- REL - all allocated direct access storage for the file is to be released to the system monitor

FORTRAN IVH OPTIONS:

- S - in line SYMBOL is used
- LS - listing source used
- GO - relocatable binary output to go to disc storage
- SI - symbolic input is received from device assigned

SYMBOL Options:

- LO - list output on LO device
- GO - relocatable binary output is to go to disc storage

Loader Options:

OLAY loader is used.

- MAP - output complete listing of external
 reference and definition for the
 load module
- GO - specifies data from users temporary GO
 file is to be included in the root of
 the load module
- UNSAT - library of the designated account is to
 be searched for external definitions
 required for the load module
- HFORLIB - FORTRAN 4H library is to be used

Execution Time:

For a 12 stage iteration of the Kalman filter algorithm listed in Appendix D, the following timing information was found:

Processor execution time - 1.026 minutes

Total job time - 5.314 minutes

JOB. 20470,MULDER,4
 LIMITS (TIME,10),(L0,200)
 ASSIGN F:OUT,(DEVICE,LPA02)
 ASSIGN F:RAD,(FILE,TEMP),(KEYED),(DIRECT),(OUTIN),(REL)
 ASSIGN F:IN,(DEVICE,CRA03)
 FORTRANH S,LS,GO,SI

C
 C THIS PROGRAM SIMULATES THE OPERATION OF A GENERAL
 C PURPOSE ARRAY STRUCTURED COMPUTER THAT EFFICIENTLY
 C IMPLEMENTS MATRIX ORIENTED ALGORITHMS. THE
 C COMMUNICATION LINK WITH THE MACHINE CONSISTS OF A
 C SET OF 58 ASSEMBLY LANGUAGE INSTRUCTIONS. ALL
 C MATHEMATICAL MANIPULATIONS, DATA STRUCTURING AND
 C INFORMATION PROCESSING IS IMPLEMENTED AS IN THE
 C MACHINE LOGICAL DESIGN.
 C

C THIS PROGRAM CONSISTS OF FORTRAN IV INPUT-OUTPUT
 C ROUTINES, AND A SYMBOL ROUTINE AS THE MAIN
 C MANIPULATION PROGRAM.
 C

C NOVEMBER 1969 M. C. MULDER
 C
 C
 C

```
COMMON/DATA/DATA(16)/TEMPBUF/TEMPBUF(32)
CALL BEGIN
END
```

```

SUBROUTINE INPUT
COMMON/DATA/DATA(16)/TEMPBUF/TEMPBUF(32)
COMMON/REG/REG(1)
EXTERNAL FROM, TO
READ(105,105)DATA
S      LW,13      REG
S      B          TO
105    FORMAT (6F13.6)
108    FORMAT (3X,6F13.6)
ENTRY OUTPUT
S      BAL,13     FROM
WRITE(108,108)DATA
S      LW,13      REG
S      AI,13      1
S      B          *13
END
```


SYMBOL LO,GO,CN

*

*

*

THIS ROUTINE IS USED TO CONVERT INPUT DATA
FROM BASE 16 TO BASE 2, AND OUTPUT
DATA FROM BASE 2 TO BASE 16. FORTRAN 4H
INPUT - OUTPUT STATEMENTS USE AS AN
INFORMATION BLOCK, DATA(16). INTERNAL
SYMBOL ROUTINES USE AS AN INFORMATION
BLOCK, TEMPBUF(32).

*

*

	DEF	TEST
	DEF	TO, FROM
	REF	DATA
	REF	TEMPBUF
TEST	DATA	X'00C00000'
SIGN	RES	1
RETURN	RES	1
**	CONVERT DATA TO MACHINE	
**	FLOATING-POINT REPRESENTATION	
TO	AI,13	1
	STW,13	RETURN
	LI,6	32
	LI,7	0
	STW,7	TEMPBUF-1,6 ZERO TEMP. STORAGE.
	BDR,6	\$-1
	LI,6	32
	LI,7	16
TOLOOP	LW,2	DATA-1,7 CONVERTS FORTRAN FLT PT TO KF
	BNEZ	\$+3
	LI,5	0
	B	ENDTO
	LAW,4	2
	LI,5	0
	SCD,4	8
	AI,5	-64
	SLS,5	2
	AI,5	64
	SLS,4	-8

	LI,9	24	
	CW,4	TEST	
	BCS,4	\$+4	
	SLS,4	1	
	AI,5	-1	
	BDR,9	\$-4	
	CI,2	0	
	BGEZ	\$+2	
	LCW,4	4	
	SLS,4	7	
	OR,5	4	
ENDTO	STW,5	DATA-1,7	
KFEXPAND	SCS,5	1	
	LI,2	3	
	LB,3	5,2	LOAD EXPONENT.
	SLS,3	-1	
	STW,3	TEMPBUF-1,6	STORE EXPONENT.
	SCS,5	-1	
	SLS,5	-7	
	STW,5	TEMPBUF-2,6	STORE FRACTION.
	AI,6	-2	
	BDR,7	TOLOOP	
	B	*RETURN	
**	CONVERT DATA FROM MACHINE		
**	FLOATING-POINT REPRESENTATION		
FROM	STW,13	RETURN	
	LI,6	16	
	LI,7	0	
	STW,7	DATA-1,6	ZERO OUTPUT BUFFER.
	BDR,6	\$-1	
	LI,6	16	
	LI,7	32	
KFCOMPRS	LW,5	TEMPBUF-2,7	LOAD FRACTION.
	SLS,5	7	
	LW,3	TEMPBUF-1,7	LOAD EXPONENT.
	OR,3	5	
	STW,3	DATA-1,6	STORE WORD IN KF FORMAT
	AI,7	-2	
	AI,6	-1	
	CI,7	0	
	BNE	KFCOMPRS	DONE WITH COMPRESSION.
	LI,7	16	
FROMLOOP	LW,2	DATA-1,7	
	BNEZ	\$+3	
	LI,6	0	

	B	ENDFROM
	SAD,2	-7
	SLS,3	-25
	AI,3	-64
	LW,4	3
	SLD,4	-2
	SLS,5	-30
	CI,5	0
	BEZ	\$+4
	AI,4	1
	AI,5	-5
	BIR,5	\$+2
	B	\$+3
	SAS,2	-1
	B	\$-3
	AI,4	64
	LAW,6	2
	SLS,4	24
	OR,6	4
	CI,2	0
	BGEZ	\$+2
	LCW,6	6
ENDFROM	STW,6	DATA-1,7
	BDR,7	FROMLOOP
	B	*RETURN
	END	

SYMBOL LO,GO,CN

```

*
*
* * * * *
*
*       THIS SYMBOL PROGRAM IS THE MAIN
*       MANIPULATION ROUTINE.
*       INSTRUCTIONS ARE INTERPRETED AND
*       EXECUTED BY THESE ROUTINES. :
*
* * * * *
*
*       DEF      BEGIN
*       DEF      KEY
*       DEF      ECHO
*       DEF      TEMPBUF
*       REF      F:IN
*       REF      INPUT,OUTPUT,DATA
*       REF      F:OUT
*       REF      F:RAD
*       REF      REG
*       REF      TEST
*
*
*       BTM = 1 FOR BTM MONITOR.
*       BTM = 0 FOR MSU MONITOR.
*
BTM      EQU      1
R        EQU      2
T        EQU      4
S        EQU      5
D        EQU      6
I        EQU      7
M        EQU      8
CX       EQU      9
J        EQU      10
K        EQU      11
ASAVE    EQU      13
BSAVE    EQU      14
CSAVE    EQU      15
*
*
*       WRITE A NUMBER XXX.YYY
*       TO F:OUT

```

* TAKE NUMBER FROM 2,3 AS A 64-BIT NUMBER
 * (TWO'S COMPLIMENTED FORM).
 * CONVERT TO KF MACHINE FORMAT.

* CALLING SEQUENCE IS BAL,13 OUTPUT.
 *
 *

KEY	DATA	1
EXCLOC	DATA	0
* WRITE		
	GEN,8,24	X'11',F:OUT
	GEN,8,24	X'38',16
	DATA	LINEBUFF
	DATA	132
	DATA	KEY
	BOUND	8
LINEBUFF	RES	33
OUTBUF	RES	8
CONTENTS	DATA	0
* DUMPBUFF		

	CALL,1	WRITE
	LI,R	X'40'
	LI,I	131
	STB,R	LINEBUFF,I
	BDR,I	\$-1
	STW,I	CONTENTS
	B	*8

*
 *
 *
 *
 *
 *

COMPLIMENT TWO'S COMPLIMENT OF NUMBER
 STORED IN R,R+1 = 2,3.

COMPLX	RES	1	
	BOUND	8	
ADONE	DATA	1,0	
COMPL	STW,8	COMPLX	SAVE RETURN
	LI,M	-1	FORM 1'S MASK
	EOR,R	M	COMPLIMENT INTEGER PART
	AD,R	ADONE	INCREMENT FRACTION.
	B	*COMPLX	

*
 *
 *

ECHO	CALL,1	WRITEINP	REMOTE: MODIFY,69000000
READ	GEN,8,24	X'10',F:RAD	
	GEN,8,24	X'F8',16	
	DATA	ERROR	
	DATA	ERROR	
	DATA	NUMBUF	
	DATA	80	
	DATA	EXCLOC	
RDINSTR	GEN,8,24	X'10',F:IN	
	GEN,8,24	X'F8',16	
	DATA	ERROR	
	DATA	ERROR	
	DATA	NUMBUF	
	DATA	80	
	DATA	KEY	
STACK	GEN,8,24	X'11',F:RAD	
	GEN,8,24	X'38',48	
	DATA	NUMBUF	
	DATA	80	
	DATA	EXCLOC	
WRITEINP	GEN,8,24	X'11',F:OUT	
	GEN,8,24	X'38',16	
	DATA	NUMBUF	
	DATA	80	
	DATA	KEY	
SKIP	GEN,8,24	X'11',F:OUT	
	GEN,8,24	X'38',16	
	DATA	BLANKS	
	DATA	4	
	DATA	KEY	
BLANKS	TEXT	' '	
NUMBUF	RES	20	
READNUMX	RES	1	
READNUM	STW,8	READNUMX	SAVE RETURN
	MTW,1	EXCLOC	
	CALL,1	READ	READ NUMBER
	CALL,1	WRITEINP	
	LI,J	1	SIGN TO POSITIVE
	LI,CX	1	SET COMPAR TO 1
	LI,S	0	NUM TO 0
	LI,R+1	0	NUM TO 0
	LI,I	0	INITIALIZE CHAR INDEX
	LI,D	0	NO DECIMAL YET
CHARLOOP	LB,T	NUMBUF,I	GET A CHARACTER
	CI,T	X'4B'	NO...IS CHAR DECIMAL

	BNE	\$+4	NO...CHECKMINUS
	LW,R	S	SAVE NUM
	LI,D	1	SET DECIMAL POINT FLAG
	B	ENDLOOP	
	CI,T	X'60'	MINUS SIGN
	BNE	\$+3	NO...CHECK DIGIT
	LI,J	-1	YES...SIGN TO NEGATIVE
	B	ENDLOOP	
	CI,T	X'F0'	DIGIT
	BL	ENDLOOP	NO...IGNORE
	AI,T	-240	YES...CONVERT TO BINARY.
	CI,D	0	DECIMAL YET
	BG	FRACTION	YES...FRACTION PART
	MI,S	10	NO...INTEGER PART
	AW,S	T	NUM=DIGIT+10*NUM
	B	ENDLOOP	
FRACTION	MI,CX	10	SET COMPARE
	LI,K	32	SET BIT COUNTER
	LW,S	T	GET DIGIT
	LI,T	0	ALL BITS TO ZERO
LOOPTOP	MI,S	2	TIMES TWO
	CW,S	CX	CHECK OVERFLOW
	BL	ZERO	NO OVERFLOW BIT IS ZERO
ONE	SW,S	CX	THROW AWAY THE OVERFLOW BIT
	SLS,T	1	SHIFT 1 BIT
	AI,T	1	INSERT BIT
	B	AGAIN	NEXT BIT
ZERO	SLS,T	1	SHIFT 1 BIT
AGAIN	BDR,K	LOOPTOP	NEXT BIT
	AWM,T	R+1	STUFF NEW DIGIT CONTRIBUTION
ENDLOOP	AI,I	1	INCREMENT CHAR. INDEX
	CI,I	60	LAST CHAR.
	BLE	CHARLOOP	NO...GET MORE
	CI,J	0	YES...IS NUMBER NEGATIVE
	BG	POS	NO...LEAVE IT ALONE
	BAL,8	COMPL	YES...COMPLIMENT
POS	B	*READNUMX	
	*		

	*	ERROR ROUTINE	
	*		
ERRORMSG	GEN,8,24	X'11',F:OUT	
	GEN,8,24	X'38',16	
	DATA	ERRBUFF	
	DATA	8	

	DATA	KEY
	BOUND	8
MESSAGE	TEXT	'***ERROR'
ITERNUM	TEXT	'STEP NO.'
ERRBUFF	RES	2

*

ERROR	LD,R	MESSAGE
	STD,R	ERRBUFF
	CALL,1	ERRORMSG
	B	START

*

*

*

*

READ AND INTERPRET INSTRUCTIONS

EBCDIC	TEXT	'0123456789ABCDEF'
OPTABLE	EQU	\$
	TEXT	'ABSE'
	TEXT	'ADBE'
	TEXT	'ADDA'
	TEXT	'ADEB'
	TEXT	'APBR'
	TEXT	'BCAC'
	TEXT	'BCAR'
	TEXT	'BCBC'
	TEXT	'BCBR'
	TEXT	'BDIZ'
	TEXT	'CMPA'
	TEXT	'CMPB'
	TEXT	'EXCC'
	TEXT	'EXCR'
	TEXT	'INPA'
	TEXT	'INPB'
	TEXT	'INPC'
	TEXT	'INPI'
	TEXT	'INVA'
	TEXT	'INVB'
	TEXT	'LDEB'
	TEXT	'LDIB'
	TEXT	'MLBC'
	TEXT	'MLBE'
	TEXT	'MLBR'
	TEXT	'MLEB'
	TEXT	'MLTA'
	TEXT	'MULT'
	TEXT	'MOVA'

TEXT	'OUTA'
TEXT	'OUTB'
TEXT	'OUTC'
TEXT	'OUTR'
TEXT	'ROTD'
TEXT	'ROTL'
TEXT	'ROTR'
TEXT	'ROTU'
TEXT	'SETC'
TEXT	'SETE'
TEXT	'SETR'
TEXT	'SKWD'
TEXT	'SKWL'
TEXT	'SKWR'
TEXT	'SKWU'
TEXT	'SNEB'
TEXT	'SQRB'
TEXT	'STBE'
TEXT	'SUMC'
TEXT	'SUMR'
TEXT	'TRAC'
TEXT	'TRAR'
TEXT	'TRNC'
TEXT	'TRNR'
TEXT	'TRPS'
TEXT	'ZERA'
TEXT	'ZERC'
TEXT	'ZERR'
TEXT	'\$\$\$\$'

*

*

OPADR

*

OPCODE ADDRESSES	
DATA	ERROR

DATA	ABSE
DATA	ADBE
DATA	ADDA
DATA	ADEB
DATA	APBR
DATA	BCAC
DATA	BCAR
DATA	BCBC
DATA	BCBR
DATA	BDIZ
DATA	CMFA
DATA	CMPB

DATA	EXCC
DATA	EXCR
DATA	INPA
DATA	INPB
DATA	INPC
DATA	INPI
DATA	INVA
DATA	INVB
DATA	LDEB
DATA	LDIB
DATA	MLBC
DATA	MLBE
DATA	MLBR
DATA	MLEB
DATA	MLTA
DATA	MULT
DATA	MOVA
DATA	OUTA
DATA	OUTB
DATA	OUTC
DATA	OUTR
DATA	ROTD
DATA	ROTL
DATA	ROTR
DATA	ROTU
DATA	SETC
DATA	SETE
DATA	SETR
DATA	SKWD
DATA	SKWL
DATA	SKWR
DATA	SKWU
DATA	SNEB
DATA	SQRB
DATA	STBE
DATA	SUMC
DATA	SUMR
DATA	TRAC
DATA	TRAR
DATA	TRNC
DATA	TRNR
DATA	TRPS
DATA	ZERA
DATA	ZERC
DATA	ZERR

DATA \$\$\$\$

*

*

*

*

OPTIME DATA ERROR

*

*

DATA	6
DATA	10
DATA	7
DATA	4
DATA	10
DATA	3
DATA	5
DATA	4
DATA	9
DATA	1
DATA	7
DATA	1
DATA	2
DATA	2
DATA	7
DATA	1
DATA	2
DATA	1
DATA	219
DATA	15
DATA	3
DATA	13
DATA	11
DATA	14
DATA	11
DATA	11
DATA	8
DATA	91
DATA	5
DATA	7
DATA	1
DATA	2
DATA	7
DATA	2
DATA	2
DATA	2
DATA	2

DATA	1
DATA	1
DATA	1
DATA	2
DATA	2
DATA	2
DATA	2
DATA	3
DATA	11
DATA	4
DATA	43
DATA	43
DATA	5
DATA	5
DATA	5
DATA	5
DATA	2
DATA	3
DATA	3
DATA	3
DATA	1

*

*

*

*

OP

RES 1

A

RES 1

B

RES 1

C

RES 1

*

* * * * *

*

*

EXECUTION ROUTINE

*

*

MASK OUT FIXED POINT ARITHMETIC TRAPS

*

*

TRAPFPT

GEN,8,24 X'14',0

DATA 1

*

BEGIN

CALL,8 TRAPFPT

LI,R X'40'

LI,I 131

STB,R LINEBUFF,I

BDR,I \$-1

	DO1	1-BTM
	LI,R	0
	DO1	BTM
	LW,R	L(X'03000000')
	STW,R	EXCLOC
	LI,R	X'5B'
PROG	MTW,1	EXCLOC
	CAL1,1	RDINSTR
	CAL1,1	STACK
	CB,R	NUMBUF
	BNE	PROG
	LI,R	0
	DO1	BTM
	LW,R	L(X'03000000')
	STW,R	EXCLOC
	LI,T	0
	STW,T	TIMEHOLD
*		
START	MTW,0	CONTENTS
	BEZ	\$+2
	BAL,8	DUMPBUFF
	MTW,1	EXCLOC
	CAL1,1	READ
	CAL1,1	WRITEINP
	LB,R	NUMBUF
	CI,R	X'5C'
	BE	START
*		
	LW,R+1	NUMBUF
	LI,R	64
	CW,R+1	OPTABLE-1,R
	BE	\$+3
	BDR,R	\$-2
	B	ERROR
	STW,R	OP
*		
	LI,S	0
	STW,S	A
	STW,S	B
	STW,S	C
	LI,I	4
	LB,R+1	NUMBUF,I
	CI,R+1	X'40'
	BNE	\$+5
	AI,I	1

CI,I	80
BLE	\$-5
B	\$+9
LI,R	15
CB,R+1	EBCDIC,R
BE	\$+2
BDR,R	\$-2
STW,R	A,S
AI,S	1
CI,S	3
BL	\$-11

*
*

LW,T	OP	PRINT EXCT. AND TOTAL TIME.
AI,T	OPTIME	
LW,T+1	*T	
STW,T+1	TIMEHOLD+1	
BAL,8	TIMEOUT	
CAL1,1	WRITE	PRINT OUT ACUM. EXECUTIVE TIME.
CAL1,1	SKIP	

*

LW,R	OP
AI,R	OPADR
LW,R	*R
BAL,8	*R
B	START

*

* * * * *

*

CHANGE

*

CALLING SEQUENCE

*

BAL,8 CHANGE

*

CHANGEX	RES	1
CHANGE	STW,8	CHANGEX
	LD,R	ROWMAX
	XW,R	ROWSIZE
	XW,R+1	COLSIZE

```

                STD,R      ROWMAX
                B          *CHANGEX
*
* * * * *
*
*          DOROW
*
*          ROUTINE TO INITIALIZE A ROW LOOP
*
DORX           RES        1
DOROW          STW,8       DORX
                LI,8       1
                STW,8       C
                B          *DORX
*
* * * * *
*
*          DOCOL
*
*          ROUTINE TO INITIALIZE A COLUMN LOOP
*
DOCX           RES        1
DOCOL          STW,8       DOCX
                LI,8       1
                STW,8       B
                B          *DOCX
*
* * * * *
*
*          DOARRAY
*
*          ROUTINE TO INITIALIZE AN ARRAY LOOP
*
DOAX           RES        1
DOARRAY        STW,8       DOAX
                BAL,8       DOCOL
                BAL,8       DOROW
                B          *DOAX
*
* * * * *
*
*          FINROW
*
*          ROUTINE TO TERMINATE A ROW LOOP
*

```

```

FINRX      RES      1
FINROW     STW,8     FINRX
           LW,8      C
           AI,8      1
           STW,8     C
           CW,8      COLMAX
           BLE       *FINRX
           LI,8      1
           AWM,8     FINRX
           B         *FINRX

```

```

*
```

```

* * * * *
```

```

*
```

```

*          FINCOL
*
```

```

*
```

```

*          ROUTINE TO TERMINATE A COLUMN LOOP
*          CALLING SEQUENCE

```

```

*
```

```

*          BAL,8      FINCOL
*
```

```

FINCX      RES      1
FINCOL     STW,8     FINCX
           LW,8      B
           AI,8      1
           STW,8     B
           CW,8      ROWMAX
           BLE       *FINCX
           LI,8      1
           AWM,8     FINCX
           B         *FINCX

```

```

*
```

```

* * * * *
```

```

*
```

```

*          FINARRAY
*
```

```

*
```

```

*          ROUTINE TO TERMINATE AN ARRAY LOOP
*
```

```

FINAX      RES      1
FINARRAY   STW,8     FINAX
           BAL,8     FINROW
           B         *FINAX
           BAL,8     FINCOL
           B         $+4
           LI,8      1
           AWM,8     FINAX
           B         *FINAX

```



```

          BAL,8      DOROW
          B          *FINAX
*
* * * * *
*
*          SHIFTD
*
SHIFTDX   RES        1
SHIFTD    STW,8      SHIFTDX
          LW,BSAVE   ROWMAX
          STW,BSAVE  B
          BAL,8      LOAD
          STD,R      T
          MTW,-1     B
          BLEZ       $+6
          BAL,8      LOAD
          MTW,1      B
          BAL,8      STORE
          MTW,-1     B
          B          $-6
          LD,R      T
          MTW,1      B
          BAL,8      STORE
          B          *SHIFTDX

```

```

*
* * * * *
*
*          SHIFTL
*
SHIFTLX   RES        1
SHIFTL    STW,8      SHIFTLX
          LI,CSAVE   1
          STW,CSAVE  C
          LW,D       COLMAX
          BAL,8      LOAD
          STD,R      T
          MTW,1      C
          CW,D       C
          BL         $+6
          BAL,8      LOAD
          MTW,-1     C
          BAL,8      STORE
          MTW,1      C
          B          $-7
          LD,R      T

```

```

      MTW,-1      C
      BAL,8       STORE
      B           *SHIFTLX
*
* * * * *
*
*           SHIFTR
*
SHIFTRX      RES      1
SHIFTR      STW,8     SHIFTRX
            LW,CSAVE  COLMAX
            STW,CSAVE C
            BAL,8     LOAD
            STD,R     T
            MTW,-1    C
            BLEZ      $+6
            BAL,8     LOAD
            MTW,1     C
            BAL,8     STORE
            MTW,-1    C
            B         $-6
            LD,R      T
            MTW,1     C
            BAL,8     STORE
            B         *SHIFTRX

```

```

*
* * * * *
*
*           SHIFTU
*
SHIFTUX      RES      1
SHIFTU      STW,8     SHIFTUX
            LI,BSAVE  1
            STW,BSAVE B
            LW,D      ROWMAX
            BAL,8     LOAD
            STD,R     T
            MTW,1     B
            CW,D      B
            BL        $+6
            BAL,8     LOAD
            MTW,-1    B
            BAL,8     STORE
            MTW,1     B
            B         $-7

```

```

LD,R      T
MTW,-1    B
BAL,8     STORE
B         *SHIFTUX

```

```

*
* * * * *

```

```

*
*
* ROUTINE TO GENERATE ELEMENT ADDRESS OF
* A(B,C) & PUT IT IN REG. 2.
*

```

```

* CALLING SEQUENCE
* BAL,8 ELADRGEN
*
ELADRGEN LW,R      B      GET ROW ADDRESS
          AI,R      -1
          SLS,R     4      SHIFT OVER
          AW,R      C      PUT IN COLUMN ADDRESS
          AI,R      -1
          SLS,R     4      SHIFT OVER
          OR,R      A      PUT IN ELEMENT ADDRESS
          B         *8     RETURN

```

```

*
*****
*
* * * * *

```

```

* ROUTINE TO LOAD A(B,C) INTO 2,3
*
LOADX    RES      1
LOAD     STW,8     LOADX
          BAL,8    ELADRGEN
          LD,R     MEMORY,R
          B        *LOADX

```

```

*
*
* ROUTINE TO STORE 2,3 IN A(B,C)
*

```

```

STOREX   RES      1
STORE    STW,8     STOREX
          XW,R     I
          BAL,8    ELADRGEN
          XW,R     I
          STD,R    MEMORY, I
          B        *STOREX

```

*

START OF OPERATION IMPLEMENTATION

*

* A B S E

*

ABSEX	RES	1
ABSE	STW,8	ABSEX
	BAL,8	LOAD
	SLS,R	7
	CI,R	0
	BGEZ	\$+2
	BAL,8	COMPL
	SLS,R	-7
	BAL,8	STORE
	B	*ABSEX

*

*

* A D B E

*

ADBEX	RES	1
ADBE	STW,8	ADBEX
	BAL,8	LOAD
	LD,T	BUFFER
	BAL,8	ALNADD
	BAL,8	STORE
	BAL,8	ADDTIME
	BAL,8	PRTTIME
	BAL,8	ACTIME
	B	*ADBEX

*

*

* A D D A

*

ADDAX	RES	1
ADDA	STW,8	ADDAX
	BAL,8	CHANGE
	LW,ASAVE	A
	LW,BSAVE	B
	LW,CSAVE	C
	BAL,8	DOARRAY
	STW,BSAVE	A

```

BAL,8      LOAD
STD,R      T
STW,CSAVE A
BAL,8      LOAD
BAL,8      ALNADD
STW,ASAVE A
BAL,8      STORE
BAL,8      ADDTIME
BAL,8      FINARRAY
B          $-10
BAL,8      CHANGE
BAL,8      PRRTIME
BAL,8      ACTIME
B          *ADDAX

```

```

*
* * * * *
*
*          A D E B
*

```

```

ADEBX      RES          1
ADEB       STW,8        ADEBX
            BAL,8        LOAD
            LD,T        BUFFER
            BAL,8        ALNADD
            STD,R        BUFFER
            BAL,8        ADDTIME
            BAL,8        PRRTIME
            BAL,8        ACTIME
            B            *ADEBX

```

```

*
* * * * *
*
*          A P B R
*

```

```

APBRX      RES          1
APBR       STW,8        APBRX
            LW,BSAVE     B
            LW,CSAVE     C
            BAL,8        DOROW
            STW,CSAVE     B
            BAL,8        LOAD
            LD,T        BUFFER
            BAL,8        MULTIPLY
            STD,R        T
            BAL,8        MULTIME

```

```

STW,BSAVE B
BAL,8      LOAD
BAL,8      ALNADD
BAL,8      STORE
BAL,8      ADDTIME
BAL,8      FINROW
B          $-12
BAL,8      PRTTIME
BAL,8      PMTIME
BAL,8      ACTIME
B          *APBRX

```

```

*
* * * * *

```

```

*
*          B C A C
*
BCACX      RES          1
BCAC       STW,8        BCACX
           LW,ASAVE     A
           LW,BSAVE     B
           LW,CSAVE     C
           BAL,8        DOCOL
           STW,BSAVE    A
           STW,CSAVE    C
           BAL,8        LOAD
           BAL,8        DOROW
           STW,ASAVE    A
           BAL,8        STORE
           BAL,8        FINROW
           B            $-3
           BAL,8        FINCOL
           B            $-9
           B            *BCACX

```

```

*
* * * * *

```

```

*
*          B C A R
*
BCARX      RES          1
BCAR       STW,8        BCARX
           LW,ASAVE     A
           LW,BSAVE     B
           LW,CSAVE     C
           BAL,8        DOROW
           STW,BSAVE    A

```

```

STW,CSAVE B
BAL,8      LOAD
BAL,8      DOCOL
STW,ASAVE A
BAL,8      STORE
BAL,8      FINCOL
B          $-3
BAL,8      FINROW
B          $-9
B          *BCARX

```

```

*
* * * * *

```

```

*
*          B C B C
*

```

```

BCBCX      RES          1
BCBC       STW,8        BCBCX
           LW,BSAVE     B
           LD,R         BUFFER
           BAL,8        DOCOL
           STW,BSAVE    C
           BAL,8        STORE
           BAL,8        FINCOL
           B            $-3
           B            *BCBCX

```

```

*
* * * * *

```

```

*
*          B C B R
*
BCBRX      RES          1
BCBR       STW,8        BCBRX
           LD,R         BUFFER
           BAL,8        DOROW
           BAL,8        STORE
           BAL,8        FINROW
           B            $-2
           B            *BCBRX

```

```

*
* * * * *

```

```

*
*          B D I Z
*
BDIZX      RES          1
BDIZ       STW,8        BDIZX

```

BAL,8	READNUM
MTW,-1	INDEX
BEZ	*BDIZX
AWM,R	EXCLOC
CAL1,1	SKIP
CAL1,1	SKIP
LD,R	ITERNUM
STD,R	ERRBUFF
CAL1,1	ERRORMSG
LW,T+1	INDEX+2
SW,T+1	INDEX
AI,T+1	1
BAL,8	TIMEOUT
CAL1,1	SKIP
CAL1,1	WRITE
CAL1,1	SKIP
B	*BDIZX

*

* * * * *

*

* C M P A

*

CMPAX	RES	1
CMFA	STW,8	CMPAX
	BAL,8	CHANGE
	BAL,8	DOARRAY
	BAL,8	LOAD
	SLS,R	7
	BAL,8	COMPL
	SLS,R	-7
	BAL,8	STORE
	BAL,8	FINARRAY
	B	\$-6
	BAL,8	CHANGE
	B	*CMPAX

*

* * * * *

*

* C M P B

*

CMPBX	RES	1
CMPB	STW,8	CMPBX
	LD,R	BUFFER
	SLS,R	7
	BAL,8	COMPL

SLS,R -7
 STD,R BUFFER
 B *CMPBX

*
 * * * * *

*
 * E X C C
 *

EXCCX RES 1
 EXCC STW,8 EXCCX
 LW,BSAVE B
 LW,CSAVE C
 BAL,8 DOCOL
 STW,BSAVE C
 BAL,8 LOAD
 STD,R T
 STW,CSAVE C
 BAL,8 LOAD
 STW,BSAVE C
 BAL,8 STORE
 LD,R T
 STW,CSAVE C
 BAL,8 STORE
 BAL,8 FINCOL
 B \$-11
 B *EXCCX

*
 * * * * *

*
 * E X C R
 *

EXCRX RES 1
 EXCR STW,8 EXCRX
 LW,BSAVE B
 LW,CSAVE C
 BAL,8 DOROW
 STW,BSAVE B
 BAL,8 LOAD
 STD,R T
 STW,CSAVE B
 BAL,8 LOAD
 STW,BSAVE B
 BAL,8 STORE
 LD,R T
 STW,CSAVE B

	BAL,8	STORE	
	BAL,8	FINROW	
	B	\$-11	
	B	*EXCRX	
*			
* * * * *			
*			
*	I N P A		
*			
INPAX	RES	1	
INPA	STW,8	INPAX	
	BAL,8	CHANGE	
	BAL,8	DOARRAY	
	LI,K	0	INITIALIZE COUNTER.
	LI,1	0	
	LCI	0	SET CONDITION CODE TO 0000.
	STM,0	REGHOLD	
	LI,13	\$+3	
	STW,13	REG	STORE RETURN ADDRESS.
	BAL,13	INPUT	INPUT DATA IN KF FORMAT.
	DATA	0	
	LCI	0	SET CONDITION CODE TO 0000.
	LM,0	REGHOLD	
	LW,R	TEMPBUF,1	
	LW,R+1	TEMPBUF+1,1	
	BAL,8	STORE	STORE KF DATA.
	BAL,8	FINARRAY	
	B	\$+2	FINISHED?
	B	\$+6	
	AI,K	1	
	AI,1	2	
	CW,K	COLMAX	DONE WITH THIS ROW?
	BL	\$-9	
	B	\$-20	
	BAL,8	CHANGE	
	B	*INPAX	
*			
* * * * *			
*			
*	I N P B		
*			
INPBX	RES	1	
INPB	STW,8	INPBX	
	LCI	0	SET CONDITION CODE TO 0000.
	STM,0	REGHOLD	

LI,13	\$+3	
STW,13	REG	STORE RETURN ADDRESS.
BAL,13	INPUT	BRING IN DATA.
DATA	0	
LCI	0	SET CONDITION CODE TO 0000.
LM,0	REGHOLD	
LI,7	32	
LW,R	TEMPBUF-1,7	TRANSFER DATA TO BUFFER.
STW,R	BUFFER-1,7	
BDR,7	\$-2	DONE WITH TRANSFER?
B	*INPBX	

*

*

*

I N P C

*

INPCX

RES

1

INPC

STW,8

INPCX

BAL,8

CHANGE

LW,BSAVE

B

STW,BSAVE

C

BAL,8

DOCOL

LCI

0

SET CONDITION CODE TO 0000.

STM,0

REGHOLD

LI,13

\$+3

STW,13

REG

STORE RETURN ADDRESS.

BAL,13

INPUT

INPUT DATA IN KF FORMAT.

DATA

0

LCI

0

SET CONDITION CODE TO 0000.

LM,0

REGHOLD

LI,1

0

LW,R

TEMPBUF,1

LW,R+1

TEMPBUF+1,1

AI,1

2

BAL,8

STORE

BAL,8

FINCOL

B

\$-5

BAL,8

CHANGE

B

*INPCX

*

*

*

I N P I

*

INPIX

RES

1

INPI	STW,8	INPIX	
	BAL,8	READNUM	
	STD,R	INDEX	
	STD,R	INDEX+2	
	B	*INPIX	
* ***** *			
*	I N V A	GAUSS-JORDAN MATRIX INVERSION.	
*			
ADHOLD	RES	4	
INVAX	RES	1	
INVA	STW,8	INVAX	
	LI,8	1	SET TIMING PRINT BYPASS
	LI,1	4	
	STW,8	NOPRINT-1,1	
	BDR,1	\$-1	
	LW,8	B	
	XW,8	ROWSIZE	
	STW,8	ADHOLD+2	STORE RETURN MAT. ROW PARAM
	LW,8	C	
	XW,8	COLSIZE	
	STW,8	ADHOLD+3	STORE RETURN MAT. COL. PARAMETER.
	BAL,8	CHANGE	
	LW,K	A	
	STW,K	ADHOLD	STORE RETURN MATRIX.
	LI,K	13	
	STW,K	A	
	BAL,8	TEMPSTR	
	BAL,8	ZERA	ZERO WORKING AREA.
	BAL,8	RESTR	
	BAL,8	DOARRAY	
	BAL,8	SETE	SET 1ST ROW IDENTITY MATRIX.
	BAL,8	FINROW	
	B	\$-2	
	BAL,8	TEMPSTR	
	BAL,8	CHANGE	
	BAL,8	SKWD	SKEW IDENTITY MATRIX.
	BAL,8	CHANGE	
	BAL,8	RESTR	
	LW,K	ADHOLD	
	STW,K	B	
	LI,J	12	
	STW,J	A	
	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.

	BAL,8	ZERA	
	BAL,8	RESTR	RESTORE REGISTER INFO.
	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.
	BAL,8	MOVA	MOVE MATRIX TO WORKING AREA.
	BAL,8	RESTR	RESTORE REGISTER INFO.
	LI,D+1	1	SET PIVOT COUNTER.
	STW,D+1	C	SET INITIAL COUNM INDEX
	STW,D+1	B	
STRT	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.
	BAL,8	LDIB	GET ELEMENT INVERSE.
	BAL,8	RESTR	RESTORE REGISTER INFO.
	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.
	BAL,8	MLBR	MULT. ROW B OF A BY BUFFER.
	BAL,8	RESTR	
	LI,8	1	
	AWM,8	A	SET MATRIX INDEX TO 13.
	BAL,8	TEMPSTR	
	BAL,8	MLBR	MULT ROW B OF 13 BY BUFFER.
	BAL,8	RESTR	RESTORE REGISTER INFO.
	LI,8	-1	
	AWM,8	A	SET MATRIX INDEX TO 12.
	LI,D	0	
	LI,J	0	SET ITERATION COUNTER.
	STW,J	B	SET INITIAL ROW INDEX.
	LI,J	1	SET ITERATION COUNTER.
INLOOP	CW,J	D+1	MASK OUT ELEMENT.
	BE	BYPASS	
	AI,J	1	
	LW,D	B	
	AI,D	1	
	STW,D	B	
	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.
	BAL,8	LDEB	LOAD ELEM B,C OF A INTO BUF
	BAL,8	RESTR	RESTORE REGISTER INFO.
	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.
	BAL,8	CMPB	COMPLIMENT ELEMENT.
	BAL,8	RESTR	RESTORE REGISTER INFO.
	BAL,8	TEMPSTR	TEMP. STORE REGISTER INFO.
	BAL,8	APBR	MODIFY ROW.
	BAL,8	RESTR	
	LI,8	1	
	AWM,8	A	SET MATRIX INDEX TO 13.
	BAL,8	TEMPSTR	
	BAL,8	APBR	MODIFY ROW.
	BAL,8	RESTR	RESTORE REGISTER INFO.

LD,R	BUFFER
BAL,8	RECIP
STD,R	BUFFER
BAL,8	DTIME
BAL,8	DPRT
BAL,8	ACTIME
B	*INVBX

*

 *
 *
 * L D E B
 *

LDEBX	RES	1
LDEB	STW,8	LDEBX
	BAL,8	LOAD
	STD,R	BUFFER
	B	*LDEBX

*

 *
 *
 * L D I B
 *

LDIBX	RES	1
LDIB	STW,8	LDIBX
	BAL,8	LOAD
	BAL,8	RECIP
	STD,R	BUFFER
	BAL,8	DTIME
	BAL,8	DPRT
	BAL,8	ACTIME
	B	*LDIBX

*

 *
 *
 * M L B C
 *

MLBCX	RES	1
MLBC	STW,8	MLBCX
	LW,BSAVE	B
	STW,BSAVE	C
	BAL,8	DOCOL
	BAL,8	LOAD
	LD,T	BUFFER
	BAL,8	MULTIPLY
	BAL,8	STORE

BAL,8	MULTIME
BAL,8	FINCOL
B	\$-6
BAL,8	PMTIME
BAL,8	ACTIME
B	*MLBCX

*
* * * * *

*
* M L B E

MLBEX	RES	1
MLBE	STW,8	MLBEX
	BAL,8	LOAD
	LD,T	BUFFER
	BAL,8	MULTIPLY
	STD,R	BUFFER
	BAL,8	MULTIME
	BAL,8	PMTIME
	BAL,8	ACTIME
	B	*MLBEX

*
* * * * *

*
* M L B R

MLBRX	RES	1
MLBR	STW,8	MLBRX
	BAL,8	DOROW
	BAL,8	LOAD
	LD,T	BUFFER
	BAL,8	MULTIPLY
	BAL,8	STORE
	BAL,8	MULTIME
	BAL,8	FINROW
	B	\$-6
	BAL,8	PMTIME
	BAL,8	ACTIME
	B	*MLBRX

*
* * * * *

*
* M L E B

MLEBX	RES	1
-------	-----	---

MLEB	STW,8	MLEBX
	BAL,8	LOAD
	LD,T	BUFFER
	BAL,8	MULTIPLY
	BAL,8	STORE
	BAL,8	MULTIME
	BAL,8	PMTIME
	BAL,8	ACTIME
	B	*MLEBX

*

* * * * *

*

* M L T A

*

MLTAX	RES	1
MLTA	STW,8	MLTAX
	LW,ASAVE	A
	LW,BSAVE	B
	LW,CSAVE	C
	BAL,8	DOARRAY
	STW,BSAVE	A
	BAL,8	LOAD
	STD,R	T
	STW,CSAVE	A
	BAL,8	LOAD
	BAL,8	MULTIPLY
	STW,ASAVE	A
	BAL,8	STORE
	BAL,8	MULTIME
	BAL,8	FINARRAY
	B	\$-10
	BAL,8	PMTIME
	BAL,8	ACTIME
	B	*MLTAX

*

*

* * * * *

*

* M U L T

*

*

	BOUND	8
MUHOLD	RES	3
MULX	RES	1
MULT	STW,8	MULX

LI,8	1	
LI,8	1	SET TIMING PRINT BYPASS.
LI,1	4	
STW,8	NOPRINT-1,1	
BDR,1	\$-1	
LW,J	A	
STW,J	MULHOLD+2	
LW,J	B	
STW,J	MULHOLD	
LW,J	C	
STW,J	MULHOLD+1	
LI,D	12	
LI,D+1	13	
LI,ASAVE	14	
LI,CX	15	
STW,D	A	
BAL,8	TEMPSTR	
BAL,8	ZERA	ZERO WORKING AREA.
BAL,8	RESTR	
STW,D+1	A	
BAL,8	TEMPSTR	
BAL,8	ZERA	
BAL,8	RESTR	
STW,D+1	A	
LW,J	MULHOLD+1	
STW,J	B	
BAL,8	TEMPSTR	
BAL,8	MOVA	BRING MULT. TO WORKING AREA.
BAL,8	RESTR	
STW,D	A	
LW,J	MULHOLD	
STW,J	B	
BAL,8	TEMPSTR	
BAL,8	MOVA	BRING MULT. TO WORKING AREA.
BAL,8	RESTR	
BAL,8	TEMPSTR	
BAL,8	SKWL	
BAL,8	RESTR	
STW,D+1	A	
BAL,8	TEMPSTR	
BAL,8	SKWU	
BAL,8	RESTR	
BAL,8	CHANGE	
LW,K	ROWMAX	
BAL,8	CHANGE	

	STW,ASAVE	A	
	BAL,8	TEMPSTR	
	BAL,8	ZERA	ZERO WORKING AREAS.
	BAL,8	RESTR	
	STW,CX	A	
	BAL,8	TEMPSTR	
	BAL,8	ZERA	
	BAL,8	RESTR	
COMM	STW,ASAVE	A	
	STW,D	B	
	STW,D+1	C	
	BAL,8	TEMPSTR	TEMPORARY STORAGE OF REG. INFO.
	BAL,8	MLTA	INNER MULTIPLICATION.
	BAL,8	RESTR	RESTORE REG. INFO.
	STW,CX	A	
	STW,CX	B	
	STW,ASAVE	C	
	BAL,8	TEMPSTR	TEMPORARY STORAGE OF REG. INFO.
	BAL,8	ADDA	SUM ELEMENTS OF TWO MATRICES
	BAL,8	RESTR	RESTORE REG. INFO.
	STW,D	A	
	LI,J	1	
	STW,J	B	
	BAL,8	TEMPSTR	TEMPORARY STORAGE OF REG. INFO.
	BAL,8	ROTR	ROTATE RIGHT.
	BAL,8	RESTR	RESTORE REG. INFO.
	STW,D+1	A	
	LI,J	1	
	STW,J	B	
	BAL,8	TEMPSTR	TEMPORARY STORAGE OF REG. INFO.
	BAL,8	ROTD	ROTATE DOWN.
	BAL,8	RESTR	RESTORE REG. INFO.
	STW,CX	A	
	BDR,K	COMM	DONE WITH PRODUCT?
	LW,K	MULHOLD+2	
	STW,K	A	
	STW,CX	B	
	BAL,8	TEMPSTR	
	BAL,8	MOVA	RETURN PRODUCT.
	BAL,8	RESTR	
	LI,8	0	
	LI,1	4	
	STW,8	NOPRINT-1,1	
	BDR,1	\$-1	
	BAL,8	PRTIME	

```

        BAL,8      PMTIME
        BAL,8      ACTIME
        B          *MULX
*
*****
*
*          M O V A
*
MOVAX    RES      1
MOVA     STW,8     MOVAX
        BAL,8     CHANGE
        LW,ASAVE  A
        LW,BSAVE  B
        BAL,8     DOARRAY
        STW,BSAVE A
        BAL,8     LOAD
        STW,ASAVE A
        BAL,8     STORE
        BAL,8     FINARRAY
        B          $-5
        BAL,8     CHANGE
        B          *MOVAX

```

```

*
*****
*
*          O U T A
*
OUTAX    RES      1
OUTA     STW,8     OUTAX
        BAL,8     CHANGE
        BAL,8     DOCOL
        BAL,8     DOROW
        LI,7      0
        BAL,8     LOAD
        STW,R      TEMPBUF,7
        STW,R+1    TEMPBUF+1,7
        AI,7      2
        BAL,8     FINROW
        B          $-5
        LCI       0
        STM,0     RECHOLD
        LI,13     $+3
        STW,13    REG
        BAL,13    OUTPUT
        DATA     0

```

SET CONDITION CODE TO 0000.

STORE RETURN ADDRESS.

LCI	0	SET CONDITION CODE TO 0000.
LM,0	REGHOLD	
LI,7	0	
LW,8	CONTENTS	
CI,8	10	
BL	\$+2	
BAL,8	DUMPBUFF	
CAL1,1	SKIP	
BAL,8	FINCOL	
B	\$-23	
BAL,8	CHANGE	
B	*OUTAX	

*

*
* O U T B
*

OUTBX	RES	1	
OUTB	STW,8	OUTBX	
	LI,7	32	
	LW,R	BUFFER-1,7	TRANS. INFO TO TEMPBUF
	STW,R	TEMPFUB-1,7	
	BDR,7	\$-2	DONE WITH TRANSFER?
	LCI	0	SET CONDITION CODE TO 0000.
	STM,0	REGHOLD	
	LI,13	\$+3	
	STW,13	REG	STORE RETURN ADDRESS.
	BAL,13	OUTPUT	PRINT OUT BUFFER.
	DATA	0	
	LCI	0	SET CONDITION CODE TO 0000.
	LM,0	REGHOLD	
	B	*OUTEX	

*

*
* O U T C
*

OUTCX	RES	1
OUTC	STW,8	OUTCX
	BAL,8	CHANGE
	LW,BSAVE	C
	LW,BSAVE	B
	STW,BSAVE	C
	BAL,8	DOCOL
	LI,7	0

BAL,8	LOAD
STD,R	TEMPBUF,7
AI,7	1
BAL,8	FINCOL
B	\$-4
LCI	0
STM,0	REGHOLD
LI,13	\$+3
STW,13	REG
BAL,13	OUTPUT
DATA	0
LCI	0
LM,0	REGHOLD
BAL,8	CHANGE
B	*OUTCX

SET CONDITION CODE TO 0000.

STORE RETURN ADDRESS.

SET CONDITION CODE TO 0000.

*

*
* O U T R
*

OUTRX	RES	1
OUTR	STW,8	OUTRX
	BAL,8	CHANGE
	BAL,8	DOROW
	LI,7	0
	BAL,8	LOAD
	STD,R	TEMPBUF,7
	AI,7	1
	BAL,8	FINROW
	B	\$-4
	LCI	0
	STM,0	REGHOLD
	LI,13	\$+3
	STW,13	REG
	BAL,13	OUTPUT
	DATA	0
	LCI	0
	LM,0	REGHOLD
	BAL,8	CHANGE
	B	*OUTRX

SET CONDITION CODE TO 0000.

STORE RETURN ADDRESS.

SET CONDITION CODE TO 0000.

*

*
* R O T D
*

ROTDX	RES	1
ROTD	STW,8	ROTDX
	BAL,8	CHANGE
	LW,J	B
	BAL,8	DOROW
	BAL,8	SHIFTD
	BAL,*	FINROW
	B	\$-2
	BDR,J	\$-4
	BAL,8	CHANGE
	B	*ROTDX

*
* * * * *

*
* R O T L
*

ROTLX	RES	1
ROTL	STW,8	ROTLX
	BAL,8	CHANGE
	LW,J	B
	BAL,8	DOCOL
	BAL,8	SHIFTL
	BAL,8	FINCOL
	B	\$-2
	BDR,J	\$-4
	BAL,8	CHANGE
	B	*ROTLX

*
* * * * *

*
* R O T R
*

ROTRX	RES	1
ROTR	STW,8	ROTRX
	BAL,8	CHANGE
	LW,J	B
	BAL,8	DOCOL
	BAL,8	SHIFTR
	BAL,8	FINCOL
	B	\$-2
	BDR,J	\$-4
	BAL,8	CHANGE
	B	*ROTRX

*
* * * * *

```

*
*      R O T U
*
ROTUX  RES      1
ROTU   STW,8     ROTUX
        BAL,8     CHANGE
        LW,J      B
        BAL,8     DOROW
        BAL,8     SHIFU
        BAL,8     FINROW
        B         $-2
        BDR,J     $-4
        BAL,8     CHANGE
        B         *ROTUX

```

```

*
*****
*
*      S E T C
*
SETCX  RES      1
SETC   STW,8     SETCX
        BAL,8     READNUM
        CI,R      0
        BLE       ERROR
        CI,R      16
        BG        ERROR
        STW,R     COLSIZE
        B         *SETCX

```

```

*
*****
*
*      S E T E
*
SETEX  RES      1
SETE   STW,8     SETEX
        LI,R      1
        SLS,R     23
        LI,R+1    65
        BAL,8     STORE
        B         *SETEX

```

```

*
*****
*
*      S E T R
*
SETRX  RES      1

```


SETR	STW,8	SETRX
	BAL,8	READNUM
	CI,R	0
	BLE	ERROR
	CI,R	16
	BG	ERROR
	STW,R	ROWSIZE
	B	*SETRX

*

* * * * *

*

*

S K W D

*

SKWDX	RES	1
	SKWD	SKWDX
	BAL,8	CHANGE
	BAL,8	DOROW
	LW,J	C
	AI,J	-1
	BLEZ	\$+3
	BAL,8	SHIFTD
	BDR,J	\$-1
	BAL,8	FINROW
	B	\$-6
	BAL,8	CHANGE
	B	*SKWDX

*

* * * * *

*

*

S K W L

*

SKWLX	RES	1
SKWL	STW,8	SKWLX
	BAL,8	CHANGE
	BAL,8	DOCOL
	LW,J	B
	AI,J	-1
	BLEZ	\$+3
	BAL,8	SHIFTL
	BDR,J	\$-1
	BAL,8	FINCOL
	B	\$-6
	BAL,8	CHANGE
	B	*SKWLX

*

*

*

S K W R

*

SKWRX	RES	1
SKWR	STW,8	SKWRX
	BAL,8	CHANGE
	BAL,8	DOCOL
	LW,J	B
	AI,J	-1
	BLEZ	\$+3
	BAL,8	SHIFTR
	BDR,J	\$-1
	BAL,8	FINCOL
	B	\$-6
	BAL,8	CHANGE
	B	*SKWRX

*

*

*

S K W U

*

SKWUX	RES	1
SKWU	STW,8	SKWUX
	BAL,8	CHANGE
	BAL,8	DOROW
	LW,J	C
	AI,J	-1
	BLEZ	\$+3
	BAL,8	SHIFTU
	BDR,J	\$-1
	BAL,8	FINROW
	B	\$-6
	BAL,8	CHANGE
	B	*SKWUX

*

*

S N E B

*

SNEBX	RES	1
SNEB	STW,8	SNEBX
	BAL,8	LOAD
	SLS,R	7
	CI,R	0
	BGEZ	\$+3

	LI,R	-1
	B	\$+2
	LI,R	1
	LI,R+1	65
	STD,R	BUFFER
	B	*SNEBX
* * * * * *		
	S R Q B	
SQRBX	RES	1
SQRB	STW,8	SQRBX
	LD,R	BUFFER
	STD,R	T
	BAL,8	MULTIPLY
	STD,R	BUFFER
	BAL,8	MULTIME
	BAL,8	PMTIME
	BAL,8	ACTIME
	B	*SQRBX
* * * * * *		
	S T B E	
STBEX	RES	1
STBE	STW,8	STBEX
	LD,R	BUFFER
	BAL,8	STORE
	B	*STBEX
* * * * * *		
	S U M C	
SUMCX	RES	1
SUMC	STW,8	SUMCX
	LW,ASAVE	A
	LW,BSAVE	B
	LW,CSAVE	C
	LI,T	O
	LI,T+1	O
	BAL,8	DOROW
	STW,BSAVE	A

BAL,8	DOCOL
BAL,8	LOAD
BAL,8	ALNADD
XW,T	R
XW,T+1	R+1
BAL,8	ADDTIME
BAL,8	FINCOL
B	\$-6
LD,R	T
STW,ASAVE	A
STW,CSAVE	B
BAL,8	STORE
BAL,8	FINROW
B	\$-14
BAL,8	PRTTIME
BAL,8	ACTIME
B	*SUMCX

*
 * * * * *
 *
 *
 * S U M R
 *

SUMRX	RES	1
SUMR	STW,8	SUMRX
	LW,ASAVE	A
	LW,BSAVE	B
	LW,CSAVE	C
	LI,T	0
	LI,T+1	0
	BAL,8	DOCOL
	STW,BSAVE	A
	BAL,8	DOROW
	BAL,8	LOAD
	BAL,8	ALNADD
	XW,T	R
	XW,T+1	R+1
	BAL,8	ADDTIME
	BAL,8	FINROW
	B	\$-6
	LD,R	T
	STW,ASAVE	A
	STW,CSAVE	C
	BAL,8	STORE
	BAL,8	FINCOL
	B	\$-14

```

      BAL,8      PRITIME
      BAL,8      ACTIME
      B          *SUMRX
*
*****
*
*      T R A C
*
TRACK      RES      1
TRAC      STW,8      TRACK
          LW,ASAVE    A
          LW,BSAVE    B
          BAL,8      DOCOL
          STW,BSAVE   A
          BAL,8      LOAD
          STW,ASAVE   A
          BAL,8      STORE
          BAL,8      FINCOL
          B          $-5
          B          *TRACX
*
*****
*
*      T R A R
*
TRARX      RES      1
TRAR      STW,8      TRARX
          LW,ASAVE    A
          LW,BSAVE    B
          LW,CSAVE    C
          STW,CSAVE   B
          BAL,8      DOROW
          STW,BSAVE   A
          BAL,8      LOAD
          STW,ASAVE   A
          BAL,8      STORE
          BAL,8      FINROW
          B          $-5
          B          *TRARX
*
*****
*
*      T R N C
*
TRNCX      RES      1

```

TRNC	STW,8	TRNCX
	LW,BSAVE	B
	LW,CSAVE	C
	BAL,8	DOCOL
	STW,CSAVE	C
	BAL,8	LOAD
	STW,BSAVE	C
	BAL,8	STORE
	BAL,8	FINCOL
	B	\$-5
	B	*TRNCX

*
* * * * *

*
* T R N R

TRNRX	RES	1
TRNR	STW,8	TRNRX
	LW,BSAVE	B
	LW,CSAVE	C
	BAL,8	DOROW
	STW,CSAVE	B
	BAL,8	LOAD
	STW,BSAVE	B
	BAL,8	STORE
	BAL,8	FINROW
	B	\$-5
	B	*TRNRX

*
*
* * * * *

* T R P S

TRPSX	RES	1
TRPS	STW,8	TRPSX
	BAL,8	CHANGE
	BAL,8	DOARRAY
TRNSLOOP	BAL,8	LOAD
	STD,R	T
	LW,J	B
	LW,K	C
	STW,J	C
	STW,K	B
	BAL,8	LOAD
	XW,T	R

LOAD ELEMENT.

LOAD NEW ELEMENT.

STORE ELEMENT IN NEW LOCATION.

STORE ELEMENT IN NEW LOCATION.
PIVOT AROUND MATRIX DIAGONAL.

BAL,8	DOCOL
BAL,8	STORE
BAL,8	FINCOL
B	\$-2
B	*ZERCX

*
* * * * *

*
* Z E R R

ZERRX	RES	1
ZERR	STW,8	ZERRX
	LI,R	0
	LI,R+1	64
	BAL,8	DOROW
	BAL,8	STORE
	BAL,8	FINROW
	B	\$-2
	B	*ZERRX

*
\$\$\$\$ CALL,9 1

*
* * * * *

*
* A L N A D D

ALIGN AND ADD ELEMENTS.

ALNADDX	RES	1
ALNADD	STW,8	ALNADDX
	LI,D	-1
	SLS,D	-25
	SLS,R	7
	CI,R	0
	BEZ	ZEFRACT
	BGZ	\$+2
	OR,R	D
	SCS,R	-7
	SLS,T	7
	CI,T	0
	BEZ	ZEFRACT+2
	BGZ	\$+2
	OR,T	D
	SCS,T	-7
	LI,D	-1
	SLS,D	31
	LI,8	32

IS FRACTION 1 NEGATIVE?
IF ZERO EXCHANGE REG. INFO.

IF NEG. EXTEND SIGN BIT.

IS FRACTION 2 NEGATIVE?
IF 0 RETURN WITH SUM IN R,R+1.

IF NEG. EXTEND SIGN BIT.

CW,T+1	R+1	COMPARE EXPONENTS.
BE	\$+14	EXPONENTS ALIGNED.
BG	\$+10	EXP. 1 IS GREATER THAN EXP. 2.
CI,T	0	IS FRACTION 1 NEGATIVE?
BGEZ	\$+4	
SLS,T	-1	SHIFT FRACTION.
OR,T	D	EXTEND SIGN BIT.
B	\$+2	
SLS,T	-1	SHIFT FRACTION.
AI,T+1	1	INCREASE EXPONENT.
BDR,8	\$-10	
B	\$+29	
XW,T	R	
XW,T+1	R+1	
B	\$-14	
STW,M	AAHOLD	
LI,9	32	
SW,9	AAHOLD	
STW,9	AAHOLD	STORE ALIGNMENT TIME.
LW,D	R+1	
LI,R+1	0	
LI,T+1	0	
AD,R	T	
SLD,R	-2	
AI,D	2	
LAD,T	R	
LI,M	24	
CW,T	NORMTEST	NORMALIZE RESULT.
BCS,4	\$+4	
SLD,T	1	
AI,D	-1	
BDR,M	\$-4	
STW,M	AAHOLD+1	
LI,9	24	
SW,9	AAHOLD+1	
STW,9	AAHOLD+1	STORE NORMALIZING TIME.
CI,R	0	
BGEZ	\$+2	
LCW,T	T	
LW,R	T	
SLS,R	7	
SLS,R	-7	
LW,R+1	D	STORE EXPONENT IN REG. R+1.
B	*ALNADDX	
XW,T	R	

ZEFRACT

	XW,T+1	R+1	
	SLS,T	-7	
	B	*ALNADDX	

*
 * * * * *
 *
 * M U L T I P L Y
 *

PROD	RES	2	
MULTX	RES	1	
MULTIPLY	STW,8	MULTX	
	AW,T+1	R+1	FIND PRODUCT EXPONENT.
	AI,T+1	-64	
	STW,T+1	PROD	STORE PRODUCT EXPONENT.
	LI,D	-1	
	SLS,D	-25	
	SLS,R	7	
	CI,R	0	IS FRACTION NEGATIVE?
	BEZ	ZEMULT	IF PRODUCT ZERO, ZERO ELEMENT.
	BGZ	\$+2	
	OR,R	D	EXTEND SIGN BIT.
	SCS,R	-7	
	SLS,T	7	
	CI,T	0	FRACTION NEGATIVE?
	BEZ	ZEMULT	IF PRODUCT ZERO, ZERO ELEMENT.
	BGZ	\$+2	
	OR,T	D	EXTEND SIGN BIT.
	SCS,T	-7	
	SLD,D	64	ZERO PRODUCT REGISTER.
	XW,T+1	T	
	SLS,T	32	ZERO REGISTER.
	LI,R-1	0	
	CI,R	0	NEGATIVE NO.?
	BGEZ	\$+3	
	LI,R-1	-1	
	SLS,R-1	31	
	CI,T+1	0	EXTEND MULT. SIGN BIT TO LEFT.
	BGEZ	\$+2	
	LI,T	-1	NEGATIVE NO.
	SLD,T	15	
	LI,J	49	SET INCREMENTING REGISTER.
	LI,R+1	0	
	SLD,R	-1	
	SLS,R+1	-31	SHIFT DESIGNATOR BIT.
	CI,R+1	0	COMPARE DESIGNATOR BIT TO ZERO.

	BEZ	\$+2	
	AD,D	T	
	OR,R	R-1	EXTEND SIGN BIT.
	SLD,T	1	SHIFT MULT LEFT, ADD ZERO
	BDR,J	\$-8	DONE WITH THE PRODUCT?
	LW,R+1	PROD	LOAD EXPONENT INT REG. R+1.
	LAD,T	D	
	SLD,T	-7	
	LI,M	24	
	CW,T	NORMTEST	NORMALIZE RESULT.
	BCS,4	\$+4	
	SLD,T	1	
	AI,R+1	-1	
	BDR,M	\$-4	
	STW,M	MLTHOLD	
	LI,9	24	
	SW,9	MLTHOLD	
	STW,9	MLTHOLD	
	CI,D	0	
	BGEZ	\$+2	
	LCW,T	T	
	LW,R	T	
	SLS,R	7	
	SLS,R	-7	
	B	*MULTX	
ZEMULT	LI,R	0	
	LI,R+1	64	
	B	*MULTX	
* * * * *			
* * * * *			
	R E C I P		FIND ELEMENT INVERSE BY
			RESTORING DIVISION.
* * * * *			
NEG	RES	2	
RECIPX	RES	1	
	BOUND	8	
ZERES	DATA	0,0	
RECIP	STW,8	RECIPX	
	LI,T	0	
	STW,T	NEG	
	SLS,R	7	
	CI,R	0	ELEMENT NEGATIVE?
	BEZ	ERROR	*ERROR* IF DIV. BY 0 ATTEMPTED.
	BGZ	\$+3	

MTW,-1	NEG	
BAL,8	COMPL	CONVERT TO POS TWO'S COMPL.
SLS,R	-7	
LI,M	24	NORMALIZE FRACTION.
CW,R	NORMTEST	NORMALIZE RESULT.
BCS,4	\$+4	
SLS,R	1	
AI,R+1	-1	
BDR,M	\$-4	
STW,R+L	NEG+1	
SLS,D	32	ZERO QUOTIENT.
LI,M	24	SET ITERATION COUNTER.
LI,T	0	SET DIVIDEND.
LI,T+1	1	
SLS,T+1	24	NORMALIZE QUOTIENT.
XW,R	R+1	
LI,R	0	
SD,T	R	SUBTRACT DIVISOR.
CD,T	ZERES	COMPARE RESULT TO ZERO.
BEZ	CMPT	
BL	\$+3	LESS THAN ZERO?
AI,D	1	POS. RESULT, ADD 1 TO QUOTIENT.
B	\$+3	
AI,D	0	NEG. RESULT, ADD 0 TO QUOTIENT.
AD,T	R	RESTORE.
SLD,T	1	
SLS,D	1	
BDR,M	\$-10	REPEAT THE ITERATION.
SLS,D	-1	
LW,R+1	NEG+1	
AI,R+1	-64	
LI,T	1	
SW,T	R+1	
AI,T	64	
STW,T	R+1	STORE RESULTING EXPONENT.
B	\$+6	
CMPT AI,D	1	
SLS,D	1	
BDR,M	\$-1	
SLS,D	-1	
B	\$-11	
LW,R	D	
LI,M	24	
CW,R	NORMTEST	NORMALIZE RESULT.
BCS,4	\$+4	

SLS,R	1	
AI,R+1	-1	
BDR,M	\$-4	
STW,M	DHOLD	
LI,9	24	
SW,9	DHOLD	
STW,9	DHOLD	
LW,T	NEG	NEGATIVE RECIP.?
CI,T	0	
BGEZ	\$+4	
SLS,R	7	
BAL,8	COMPL	
SLS,R	-7	
B	*RECIPX	

*

* * * * *

*

*

TEMPORARY STORAGE AND RECOVERY ROUTINES.

*

	BOUND	8
TEMPSTRX	RES	10
TEMPSTR	STW,8	TEMPSTRX
	STW,D	TEMPSTRX+1
	STW,D+1	TEMPSTRX+2
	STW,J	TEMPSTRX+3
	STW,K	TEMPSTRX+4
	LW,J	B
	STW,J	TEMPSTRX+5
	LW,J	C
	STW,J	TEMPSTRX+6
	STW,I	TEMPSTRX+7
	STW,CX	TEMPSTRX+8
	STW,ASAVE	TEMPSTRX+9
	B	*TEMPSTRX
RESTRX	RES	1
RESTR	STW,8	RESTRX
	LW,J	TEMPSTRX+5
	STW,J	B
	LW,J	TEMPSTRX+6
	STW,J	C
	LW,D	TEMPSTRX+1
	LW,D+1	TEMPSTRX+2
	LW,J	TEMPSTRX+3
	LW,K	TEMPSTRX+4
	LW,I	TEMPSTRX+7

```

      LW,CX      TEMPSTRX+8
      LW,ASAVE   TEMPSTRX+9
      B          *RESTRX

```

```

*
```

```

*****
```

```

*
```

```

*
```

```

*
```

```

*
```

```

*
```

```

*
```

```

*****
```

```

*
```

```

*
```

```

*
```

```

*
```

```

TIMEOUTX RES 1
TIMEOUT STW,8 TIMEOUTX

```

```

      LI,J 8

```

```

      LI,T 0

```

```

      DW,T L(10)

```

```

      AI,T X'FO'

```

```

      SLD,M -8

```

```

      STB,T M

```

```

      BDR,J $-5

```

```

      STD,M LINEBUFF

```

```

      B *TIMEOUTX

```

```

ADDTIMEX RES 1

```

```

ADDTIME STW,8 ADDTIMEX

```

```

      LW,8 AAHOLD+2

```

```

      CW,8 AAHOLD

```

```

      BGE $+3

```

```

      XW,8 AAHOLD

```

```

      STW,8 AAHOLD+2

```

```

      LW,8 AAHOLD+3

```

```

      CW,8 AAHOLD+1

```

```

      BGE $+3

```

```

      XW,8 AAHOLD+1

```

```

      STW,8 AAHOLD+3

```

```

      B *ADDTIMEX

```

```

*
```

```

*
```

```

DTIMEX RES 1

```

```

DTIME STW,8 DTIMEX

```

```

      LW,8 DHOLD+1

```

	CW,8	DHOLD
	BGE	\$+3
	XW,8	DHOLD
	STW,8	DHOLD+1
	B	*DTIMEX
*		
*		
MULTIMEX	RES	1
MULTIME	STW,8	MULTIMEX
	LW,8	MLTHOLD+1
	CW,8	MLTHOLD
	BGE	\$+3
	XW,8	MLTHOLD
	STW,8	MLTHOLD+1
	B	*MULTIMEX
*		
*		
*		
DPRTX	RES	1
DPRT	STW,8	DPRTX
	LW,8	DHOLD+1
	AW,8	DHOLD+3
	STW,8	DHOLD+3
	LW,8	DHOLD+1
	CW,8	DHOLD+2
	BLE	\$+2
	STW,8	DHOLD+2
	LW,8	NOPRINT+2
	CI,8	0
	BGZ	\$+5
	LW,T+1	DHOLD+2
	BAL,8	TIMEOUT
	CALL,1	WRITE
	CALL,1	SKIP
	LI,8	0
	STW,8	DHOLD
	STW,8	DHOLD+1
	B	*DPRTX
*		
*		
PMTIMEX	RES	1
PMTIME	STW,8	PMTIMEX
	LW,8	MLTHOLD+1
	AW,8	MLTHOLD+3
	STW,8	MLTHOLD+3

LW,8	MLTHOLD+1
CW,8	MLTHOLD+2
BLE	\$+2
STW,8	MLTHOLD+2
LW,8	NOPRINT+1
CI,8	0
BGZ	\$+5
LW,T+1	MLTHOLD+2
BAL,8	TIMEOUT
CALL,1	WRITE
CALL,1	SKIP
LI,8	0
STW,8	MLTHOLD
STW,8	MLTHOLD+1
B	*PMTIMEX

PRINT OUT NORMALIZING TIME.

*

*

PRTTIMEX	RES	1.
PRTTIME	STW,8	PRTTIMEX
	LW,8	AAHOLD+2
	AW,8	AAHOLD+4
	STW,8	AAHOLD+4
	LW,8	AAHOLD+2
	CW,8	AAHOLD+5
	BLE	\$+2
	STW,8	AAHOLD+5
	LW,8	AAHOLD+3
	AW,8	AAHOLD+6
	STW,8	AAHOLD+6
	LW,8	AAHOLD+3
	CW,8	AAHOLD+7
	BLE	\$+2
	STW,8	AAHOLD+7
	LW,8	NOPRINT
	CI,8	0
	BGZ	\$+8
	LW,T+1	AAHOLD+5
	BAL,8	TIMEOUT
	CALL,1	WRITE
	LW,T+1	AAHOLD+7
	BAL,8	TIMEOUT
	CALL,1	WRITE
	CALL,1	SKIP
	LI,8	0
	LI,1	4

PRINTOUT ALNMENT TIME.

PRINTOUT NORMALIZING TIME.


```

      STW,8      AAHOLD-1,1
      BDR,1      $-1
      B          *PRTIMEX

```

*

*

*

*

PRINT TOTAL ACCUMULATIVE EXECUTION TIME.

*

*

```

ACTIMEX  RES      1
ACTIME   STW,8     ACTIMEX
          LW,8      NOPRINT+3
          CI,8      0
          BGZ       $+21
          LW,T+1    AAHOLD+4
          AW,T+1    AAHOLD+6
          AW,T+1    DHOLD+3
          AW,T+1    MLTHOLD+3
          AW,T+1    TIMEHOLD
          STW,T+1   TIMEHOLD
          BAL,8     TIMEOUT
          CALL,1    WRITE
          CALL,1    SKIP
          LI,8      0
          STW,8     DHOLD+2
          STW,8     DHOLD+3
          STW,8     MLTHOLD+2
          STW,8     MLTHOLD+3
          LI,1      8
          STW,8     AAHOLD-1,1
          BDR,1     $-1
          STW,8     NOPRINT
          STW,8     NOPRINT+1
          STW,8     NOPRINT+2
          B         *ACTIMEX

```

*

```

          BOUND      8
NORMTEST: DATA     X'00800000'
AAHOLD    DATA     0,0,0,0,0,0,0,0
DHOLD     DATA     0,0,0,0
MLTHOLD   DATA     0,0,0,0
NOPRINT    DATA     0,0,0,0,0
          BOUND      8

```

ROWMAX	DATA	16
COLMAX	DATA	16
ROWSIZE	DATA	4
COLSIZE	DATA	4
REGHOLD	RES	16
TIMEHOLD	RES	2
INDEX	RES	4
BUFFER	RES	32
TEMPBUF	RES	32
MEMORY	RES	8192
	END	

APPENDIX D

LISTING OF THE LINEARIZED KALMAN FILTER ALGORITHM IN PAL-1

```

*
* IMPLEMENTATION OF THE KALMAN FILTER ALGORITHM ON AN
* ARRAY STRUCTURED COMPUTER SIMULATED BY THE SIGMA 7.

```

```

*
* INITIALIZE THE MACHINE.
*

```

```

INPI
2.0
SETR
6.0
SETC
6.0
ZERA 0
ZERA 1
ZERA 2
ZERA 3
ZERA 4
ZERA 5
ZERA 6
ZERA 7
ZERA 8
ZERA 9
ZERA A
ZERA B
ZERA C

```

```

* INPUT INITIAL INFORMATION.

```

```

*

```

```

* INPUT COV. MAT. OF BEARING ERROR.

```

```

INPA 0

```

```

OUTA 0

```

```

* INPUT COV. MATRIX OF TARGET EXCITING NOISE.

```

```

INPA 1

```

```

OUTA 1

```

```

* INPUT DISTRIBUTION MATRIX.

```

```

INPA 2
OUTA 2
*      INPUT STATE TRANSITION MATRIX.
INPA 3
OUTA 3
*      INPUT INITIAL ERROR COV. MATRIX.
INPA 4
OUTA 4
*
*      INPUT INITIAL STATE VECTOR ESTIMATE.
INPC 5 1
OUTC 5 1

```

```

*
*      ITERATIVE LOOP BEGINS HERE.
*

```

```

*      INPUT NEW OBSERVATION.
INPC 7 1
OUTC 7 1
*
*      INPUT NEW LINEARIZED OBSERVATION SYSTEM PARAMETERS.
INPA 8
OUTA 8
*
*      INPUT NEW CONTROL DATA.
INPC 9 1
OUTC 9 1
*
*      INPUT THE NEW COV. MATRIX R(3,3) ENTRY.
INPB
SQRB
OUTB
STBE 0 3 3
*
*      FIND THE NEW KALMAN GAIN FACTOR.
*
TRPS 2
MULT A 1 2

```

```

TRPS 2
MULT B 2 A
TRPS 3
MULT A 4 3
TRPS 3
MULT A 3 A
ADDA A A B
MOVA 4 A
*
*      OUTPUT THE PREDICTED ERROR COV. MATRIX.
OUTA 4
TRPS 8
MULT A 4 8
TRPS 8
MULT B 8 A
ADDA A B 0
*
*      OUTPUT THE COV. MATRIX OF THE OBSERVATION VECTOR(Z).
OUTA A
INVA A 3 3
TRPS 8
MULT B 8 A
TRPS 8
MULT B 4 B
MOVA 6 B
*
*      OUTPUT THE NEW KALMAN GAIN FACTOR.
OUTA 6
*
*      FIND THE NEW STATE VECTOR ESTIMATE.
*
MULT A 2 9
CMPA A
MULT B 3 5
ADDA A A B
MOVA 5 A
*
*      OUTPUT THE CURRENT PREDICTED STATE VECTOR
*
OUTA 5
MULT A 6 7
ADDA A A 5
MOVA 5 A
*
*      OUTPUT THE NEW STATE VECTOR ESTIMATE.

```

OUTA 5
*
* FIND THE NEW COVARIANCE MATRIX.
*
MULT A 8 4
MULT A 6 A
CMPA A
ADDA A A 4
MOVA 4 A
*
* OUTPUT THE NEW COVARIANCE MATRIX.
OUTA 4
*
* FIND THE PREDICTED STATE VECTOR ESTIMATE.
*
MULT A 3 5
*
* OUTPUT THE NEW PREDICTED STATE VECTOR ESTIMATE.
OUTA A
BDIZ
-89.
\$\$\$\$

BIBLIOGRAPHY

1. Kalman, R. E., "New Methods and Results in Linear Prediction and Filtering Theory," RIAS Technical Report 61-1.
2. Brooking, M. R., et al; "Survey of In-house Work on Cellular Logic," Unpublished Memorandum, AFCRL, 1965.
3. Spandorfer, L. M., Murphy, J. V., "Synthesis of Logic Functions on an Array of Integrated Circuits," Scientific Report No. 1 for UNIVAC Engineering Center, Project 4645, 1964.
4. Minnick, R. C., Short, R. A., "Cellular Linear Input Logic," Final Report, SRI Project 4122, Contract AF 19(628)-498, Stanford Research Institute, AFCRL 64-6, DDC No. AD 433802, February 1964.
5. Canaday, R. H., "Two Dimensional Iterative Logic," AFIPS Conference Proceedings, Fall Joint Computer Conference, Vol. 27, Pt. 1, pp. 343-353.
6. Amarel, S., et al; "Majority Network Gates," IEEE Transactions on Computers, Vol. EC-13-1, February 1964, pp. 4-13.
7. Miyata, F., "Realization of Arbitrary Logical Functions Using Majority Elements," IEEE Transactions on Electronic Computers, Vol. EC-12 No. 3, June 1963, pp. 183-191.
8. Glucksman, H. A., "A Parapropagation Pattern Classifier," AFCRL Report No. 62-109, March 1962.
9. Maitra, K. K., "Cascaded Switching Networks of Two Input Flexible Cells," IRE Transactions on Electronic Computers, Vol. EC-11,2, April 1962, pp. 136-143.
10. Minnick, R. C., Short, R. A., "Cutpoint Cellular Logic," IEEE Transactions on Electronic Computers, Vol. EC-13, No. 6, December 1964, pp. 685-698.
11. Minnick, R. C., "Cobweb Cellular Arrays," AFIPS Conference Proceedings, Fall Joint Computer Conference, Vol. 27, Pt. 1, 1965, pp. 327-341.
12. Minnick, R. C., "A Survey of Micro Cellular Research," Journal of the Association for Computing Machinery, Vol. 14, April 1967, pp. 203-241.

13. Unger, S. H., "A Computer Oriented Towards Spatial Problems," IRE Proceedings, Vol. 36, October 1958, pp. 1744-1750.
14. Holland, J. H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub Programs Simultaneously," Proceedings of the Eastern Joint Computer Conference, 1959, pp. 108-113.
15. Comfort, W. T., "A Modified Holland Machine," Proceedings, Fall Joint Computer Conference, 1963, pp. 481-488.
16. Slotnick, D. L., et al; "The SOLOMON Computer," Proceedings, Fall Joint Computer Conference, 1962, pp. 97-107.
17. Gonzales, R. A., "A Multi-Layer Iterative Circuit Computer," IEEE Transactions on Electronic Computers, Vol. 12, December 1963, pp. 781-790.
18. Barnes, G. H., et al; "The ILLIAC IV Computer," IEEE Transactions on Computers, Vol. 17, August 1968, pp. 746-757.
19. Pariser, J. J., Maurer, H. E., "Implementation of the NASA Modular Computer with LSI Functional Characters," AFIPS Conference Proceedings, Fall Joint Computer Conference, 1969, pp. 231-245.
20. Dere, W. Y., Sakrison, D. J., "The Berkeley Array Processor," IEEE Transactions on Computers, Vol. C-19, No. 5, May 1970, pp. 444-447.
21. Cannon, L. E., "A Cellular Computer to Implement the Kalman Filter Algorithm," Ph.D. Thesis, Montana State University, August, 1969.
22. Hamlin, J. E., "A General Description of the NASA Real Time Computing Complex," Proceedings of the 19th National Conference of the ACM, 1964, pp. A2.2-1 to A2.2-22.
23. Creveling, C. J., "Increasing the Reliability of Electronic Equipment by the Use of Redundant Circuits," Proceedings of IRE, Vol. 44, April 1956, pp. 509-515.
24. Avizienis, A., "Design of Fault Tolerant Computers," AFIPS Conference Proceedings, Fall Joint Computer Conference, Vol. 31, 1967, pp. 733-744.

25. Pierce, W. H., Failure Tolerant Computer Design, Academic Press, Inc., New York, 1965.
26. Lyons, R. E., Vanderkulk, W., "The Use of Triple Modular Redundancy to Improve Computer Reliability," IBM Journal of Research and Development, Vol. 6, No. 2, April 1962, pp. 200-209.
27. Avizienis, A., "Concurrent Diagnosis of Arithmetic Processors," Conference Digest of the 1st Annual IEEE Computer Conference, September 1967.
28. Manning, E., "On Computer Self Diagnosis: Part I - Experimental Study of a Processor, Part II - Generalizations and Design Principles," IEEE Transactions on Electronic Computers, Vol. EC-15, No. 6, December 1966, pp. 873-890.
29. Peterson, W. W., Error Correcting Codes, The MIT Press and John Wiley and Sons, New York, 1961.
30. Downing, R. W., et al; "No.1 ESS Maintenance Plan," The Bell System Technical Journal, Vol. 43, No. 5, Part 1, September 1964, pp. 1961-2019.
31. Forbes, R. E., et al; "A Self Diagnosable Computer," AFIPS Conference Proceedings, Fall Joint Computer Conference, Vol. 27, Part 1, 1965, pp. 1073-1086.
32. Terris, I., Melkanoff, M. A., "Investigation and Simulation of a Self Repairing Digital Computer," IEEE Conference Record on Switching Circuit Theory and Logical Design, 1965, pp. 264-272.
33. Habibi, A., Wintz, P. A., "Fast Multipliers," unpublished, received through University Microfilm, January 1970.
34. Ramamoorthy, C. V., Economides, S. C., "Fast Multiplication Cellular Arrays for LSI Implementation," AFIPS Conference Proceedings, Fall Joint Computer Conference, 1969, pp. 89-98.
35. Rudberg, D. A., "A Microcellular Parallel Processor for the Discrete Kalman Filter." Presented at the Rocky Mountain Section of the American Astronautical Society Conference, Denver, Colorado, July 15-16, 1968.

- 36. Liebelt, D. B., An Introduction to Optimal Estimation, Addison-Wesley Publishing Co., Reading, Massachusetts, 1967.
- 37. Lee, R. C. K., Optimal Estimation, Identification, and Control, The M.I.T. Press, Cambridge, Massachusetts, 1966.

MONTANA STATE UNIVERSITY LIBRARIES



3 1762 10011082 2

D378 Mulder, Michael Carl
M897 The design of a
cop.2 matrix-oriented cellu-
lar computer capable
of fault-tolerant
operation

NAME AND ADDRESS

36-3 William J. A. DMP
EE 401-750

R DEC 11 1975 OLC

411N Chris Shuckling
719, 5.5 = A8

D378
M897
Cap. 2