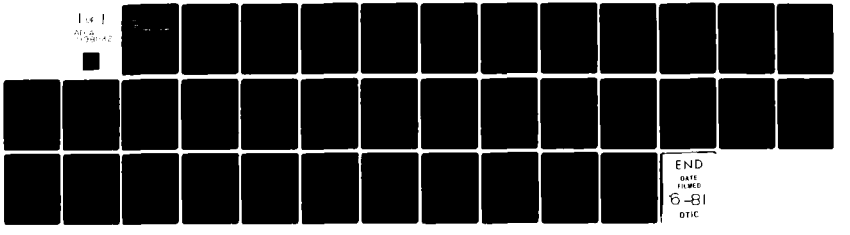
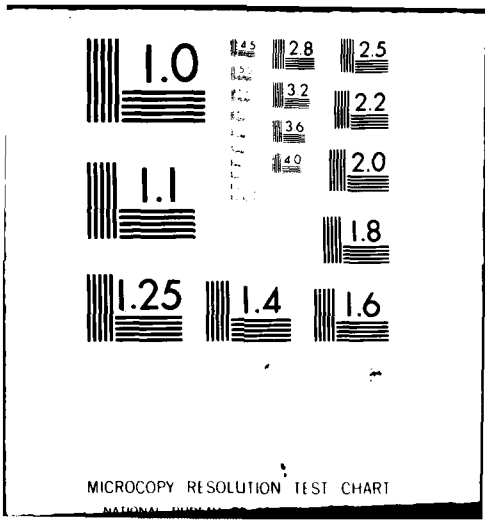
AD-A098 832 OHIO STATE UNIV RESEARCH FOUNDATION COLUMBUS F/G 9/2
A STUDY OF ALTERNATIVE COMPUTER ARCHITECTURES FOR SYSTEM RELIAB--ETC(U)
APR 81 K J BREEDING AFOSR-77-3400

UNCLASSIFIED AFOSR-TR-81-0441 NL



END
DATE FILMED
6-81
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

RF Project 760726/784778
Final Report

LEVEL II

the ohio state university
research foundation
1314 kinnear road
columbus, ohio
43212

A STUDY OF ALTERNATIVE COMPUTER ARCHITECTURES FOR SYSTEM RELIABILITY AND SOFTWARE SIMPLIFICATION

Kenneth J. Breeding
Department of Electrical Engineering

For the Period
August 1, 1977 - December 31, 1980

U.S. AIR FORCE
Air Force Office of Scientific Research
Bolling AFB, D.C. 20332

AFOSR-77-3400

April, 1981

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-81-0441	2. GOVT ACCESSION NO.: AD-A098832
3. RECIPIENT'S CATALOG NUMBER: 9	
4. TITLE (and Subtitle): A STUDY OF ALTERNATIVE COMPUTER ARCHITECTURES FOR SYSTEM RELIABILITY AND SOFTWARE SIMPLIFICATION	5. TYPE OF REPORT & PERIOD COVERED: Final rept. 1 AUG 78 - 31 DEC 80
	6. PERFORMING ORG. REPORT NUMBER: 31
7. AUTHOR(s): Kenneth J. Breeding	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3400
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Ohio State University, 1314 Kinnear Rd., Columbus, Ohio 43212	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2305/B1
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office Of Scientific Research, Bolling AFB, Bldg. 410 - Wash., D.C. 20332	12. REPORT DATE: 22 April 1981
	13. NUMBER OF PAGES: 36
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

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

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

The principle thrust of this work has been the development of special purpose computer architectures aimed at two problem areas. The first area deals with the use of hardward structures to simplify software development and utilization. The second area deals with processing systems for efficiently handling distributed tasds.

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

Unclassified 26726C

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. Introduction

The work described in this report was performed under Grant AFOSR-77-3400. The principle thrust of this work has been the development of special purpose computer architectures aimed at two problem areas. The first area deals with the use of hardware structures to simplify software development and utilization. The second area deals with processing systems for efficiently handling distributed tasks.

Certainly one of the largest cost factors in digital systems today is that of software. The cost of hardware has constantly decreased over the past ten years, and as a consequence, the use of hardware to replace some software functions has become a real alternative. The lowering cost of hardware also makes it possible to consider the application of large multiprocessing structures to spatially distributed problems such as weather forecasting and sociological system modelling. A third area of investigation performed under this grant relates to both areas. This is the problem of designing into large scale systems both fault tolerance and fault diagnosis. These three problems represent the main thrust of the research undertaken which is summarized in what follows.

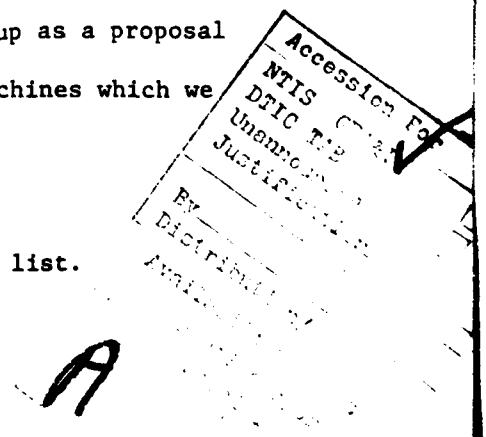
2. Synopsis of Research Accomplishments

This section briefly describes the results of the research carried out over the period August 1, 1978, to December 31, 1980. Basically, ten investigations were undertaken which covered the three major areas of research mentioned above.

In the following sections, the results of this work will be described. Section 3 gives a list of all reviewed publications either currently available or in progress which have resulted from this work.

2.1 Hardware Implementation of Software [4, 5, 15]*

The concept of a piece of hardware to directly execute high-level language has been around since about 1957. The advent of microprocessors some six years ago and the continuing decrease in hardware costs have stirred significant new interest in high level language processors. One of the investigations carried out over the period of the grant was directed at developing a general purpose, high-level language processing architecture. The goals of this work were twofold. First, to have a system which could handle arbitrary block structured, high-level languages. And secondly, to have a system which can execute directly programs written in these languages as fast as or faster than the software compiled version on standard computers. These concepts led to three separate investigations. In the first, a high-level language, or HLL, processor architecture was investigated. From this effort, two further investigations were undertaken. In the first, a specific implantation of the syntactical analysis section was examined. The resulting structure is currently being constructed using five KIM-1 microcomputers which do the syntax analysis and parsing of the source statements in a highly parallel fashion. The second effort spawned by the HLL machine began as an examination of semantics processing and program execution. It ended up as a proposal for a machine different from but similar to data flow machines which we have termed a program structured computer.



* Number in brackets refer to entries in the Publications list.

A summary of these efforts follow.

2.1.1 A High-Level Language Machine [4, 5]

Figure 1 shows, in block diagram form, a proposed architecture for a high-level language processor which meets the goal stated above. As can be seen from this figure, the system consists of four basic parts: the language definition section, the storage section, the execution section, and the analysis section. The operation of these components will be briefly described below.

The language definition section basically consists of an ROM containing the language definition tables in a form easily utilized by the remaining processors. The language definition tables contain both syntactic and semantic information as it pertains to a program statement at all points in its parse. In addition, the table also contains information about special constructs and storage allocation which may not be directly definable in the syntax. Thus, context dependent and some ambiguous grammars can be handled.

The storage unit consists of main storage and a controller. The controller for this unit, basically, is responsible for handling all requests to memory and language definition tables. This includes multiple access results originating in various token and execution processors (see below).

The analysis section is used to both parse a program string and set up the semantic information as it occurs. The main element is a token processor of which there are several. At any given instant of time, one or more (up to 8 for Fortran) token processors are involved in processing a current text string. Basically, the operation of a token processor is as follows. Initially, a set of token processors are enabled to look for all goal symbols which are legal next symbols in the current input string. Typically, there would be one token processor per goal symbol. Each processor will be

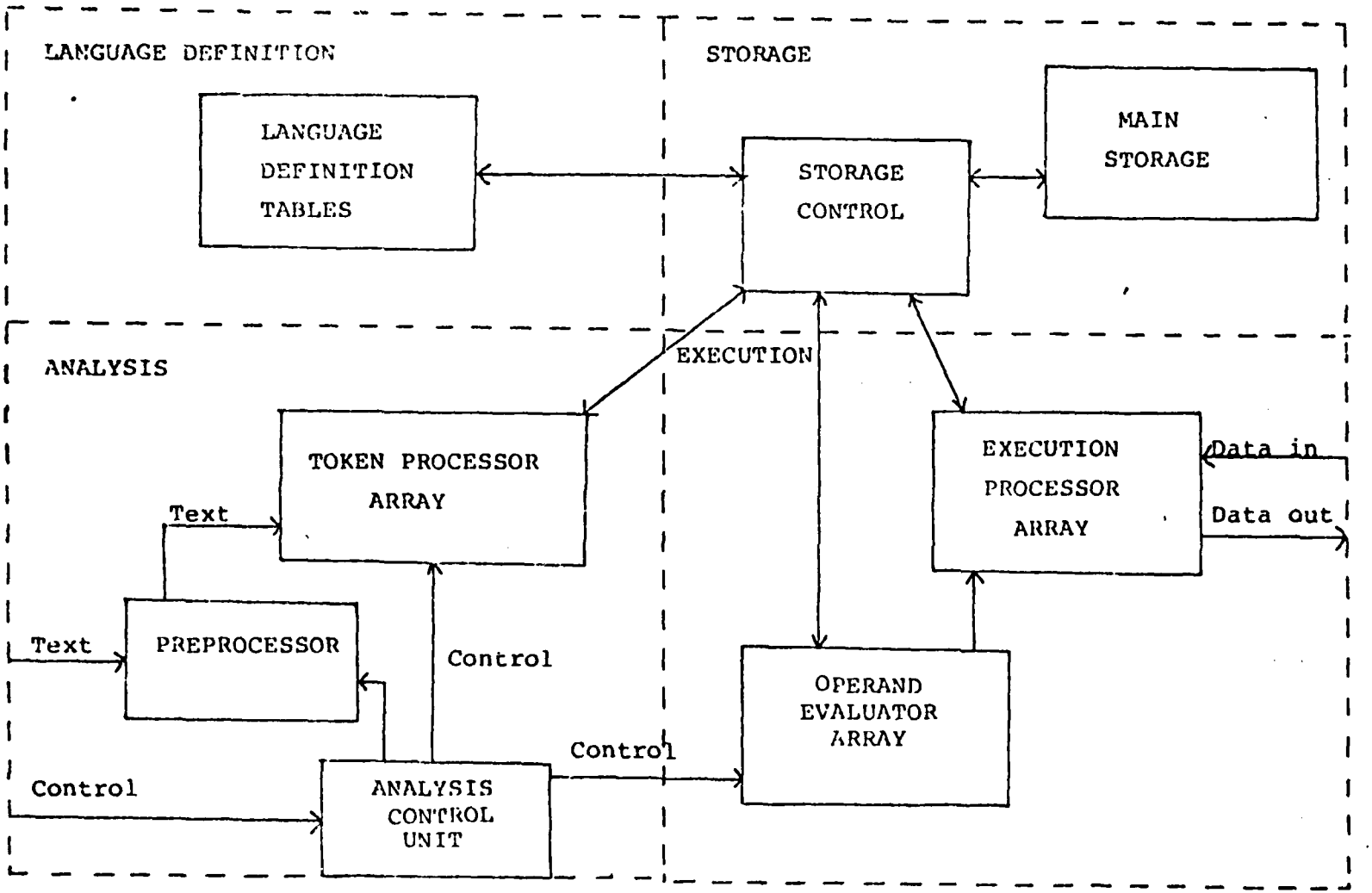


Figure 1. Proposed System Block Diagram.

responsible for preparing the syntactic and semantic information for their particular goal symbol. If the next text symbol is not the goal symbol, then the corresponding token processor will be removed from service. If the next symbol is the anticipated goal symbol, the corresponding token processor will then do two things. First, it will invoke a new set of token processors to look for all legitimate next symbols. Secondly, it will pass on to the execution unit the semantic information associated with its goal symbol.

The execution unit is made up of a multiprocessing system consisting of processor dedicated to specific types of computation processes. For example there may be one or more processors to perform trigonometric computations with others setting up data arrays and still others handling I/O. The main mechanism for invoking an execution processor is an execution tree which is set up by the token processor in the analysis unit. Each element or leaf in this tree contains information about how the various operands are to be evaluated along with points to the next elements, which serve as operands, functions, etc., for the current operation.

A simulation of such a high-level language processor was written for a GRI-99 minicomputer and aimed at a text language called SLANG. In this particular simulation, the input text was first analyzed completely and then the resulting execution tree was processed. The simulated run time for these processes was 20 ms on a particular benchmark program. The corresponding time for compile/execute processing in the conventional way was on the order of 2 sec. Simulations of both Fortran IV and Algel 60 showed similar results. Details of this work may be found in [4, 5].

2.1.2 A Token Processor Emulation [15]

In order to more fully understand and evaluate the architecture proposed in [4, 5], construction of an emulation for the analysis section was

undertaken. This system, based on the KIM-1 microcomputer, is currently under construction. Figure 2 shows the overall organization of the analysis section, language definition tables, and the storage sections, collectively termed the translator.

In these figures, programs are input to the analysis control unit, ACU, which is implemented by a DEC PDP-11/03. The ACU primarily is used to minimally preprocess the programs. In this particular case, the PDP-11/03 serves several other functions as well. It is used to edit and create programs, generate the language definition tables, and generally monitor the overall system operation.

The ACU passes the incoming program tokens onto the token processors, TP1 and TP2, via the buffer registers. The TP's then use the tokens thus received to access the language definition table, LDT, via pointers found in the ROM. Information accessed from the LDT consists of blocks of data or information packets which identify what the appropriate syntax should be at the next token. The TP's use this information to parse the incoming strings and set up the appropriate execution trees in the Memory. The Memory also serves to store intermediate tables generated by the TP's as well as the input programs accessed by the ACU.

The bus protocols in the three-processor system shown in Figure 2 have been developed and simulated on an Amdahl 470. Details of this work may be found in [15].

2.1.2.1 Language Definition Table Generation

The architecture described in Mooney [4, 5] for a high-level language processor requires that the language to be used be described by language definition tables. The form of these tables is generally not convenient for use by humans. Thus, some meta-language must be used for describing the

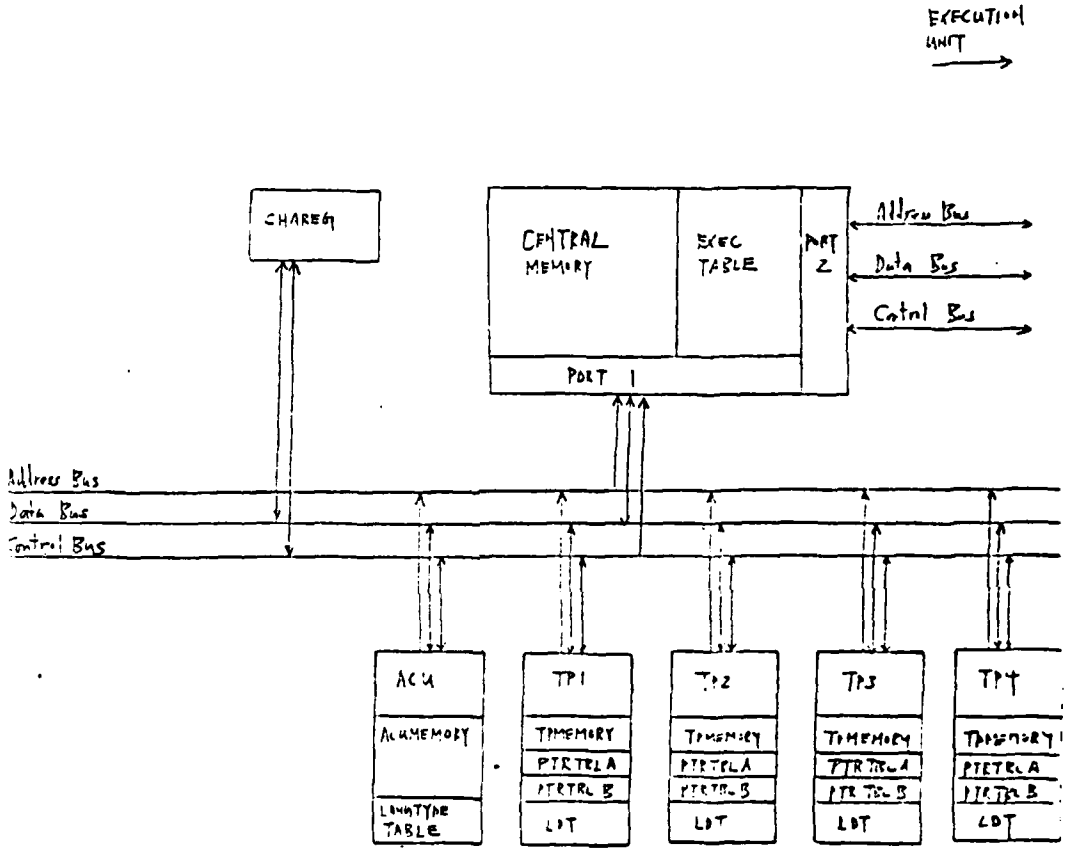


Figure 2. System Configured as a Translator.

various languages and this meta-language must then be translated into an appropriate LDT.

The meta-language used here is the Backus-Naur form. The translation procedure is in two parts. First, the BNF representation of the language used is converted into a two-dimensional grammar which is basically a graph structure. Secondly, the graphical representation of the language is converted into the form required by the TP's and is stored in the language definition tables.

To illustrate the procedure, a simple example will be carried through. Consider the following grammar expressed in BNF as*

$$A \rightarrow Bb \mid aB \mid aCd$$

$$B \rightarrow e \mid eC$$

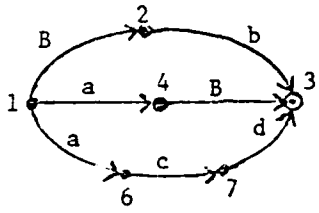
$$C \rightarrow abB \mid cdB$$

where lower case letters represent terminals and upper case letters represent nonterminals.

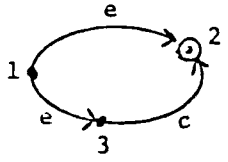
A graph is now created for each production rule as follows: Each production graph starts on a common node and ends on a common node called the entry and exit nodes, respectively. A path is created for each replacement rule with the arcs corresponding to the terminals and nonterminals in the string. The nodes are numbered arbitrarily. Thus, for the above example, the graphs become

* For convenience of typing at a computer console, the "replacement" symbol ::= was replaced by →.

A:

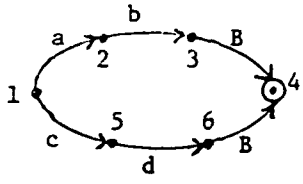


B:



⊙ = exit node

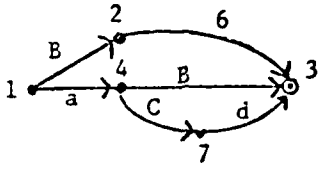
C:



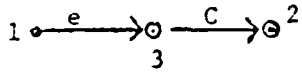
The second step is to combine, into one arc, those arcs labeled the same and having a node in common (either on the left or right).

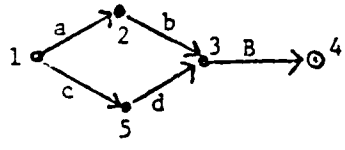
For the example here, the above graphs become

A:



B:





Note in the reduction of B that an exit node and a nonexit node were combined into an exit node.

The BNF graphs may now be converted into LDT entries. This is done by setting up information packets for each node in each BNF graphs which identifies all possible next terminal tokens and a list of nodes passed through in the various graphs. For example, consider node A_4. There are two paths leaving node A_4, one ending on A_3 and the other on A_7, but both arcs are nonterminals. Two partial node stacks are started, one with A_3 on top and the other with A_7 on top. Following the arc with a B label, go to the starting node of rule B, B_1. A single arc leaves B_1 with a terminal symbol, e, and ends on B_3. Thus, the packet is

$$e;\ A_3B_3$$

where e is a possible next token and A_3B_3 represents the partial node stack associated with the corresponding path, where B_3 is the top stack entry. In exactly the same manner, the arc labelled C gives two packets, viz.

$$a;A_7C_2 \quad \text{and} \quad c;A_7C_5$$

Table 1 gives the complete language definition table for this grammar.

This algorithm has been programmed to run on a PDP-11/03 and is currently operational.

2.1.3 A Program Structured Computer [19]

Another type of HHL architecture, a Data Flow architecture, does not attempt to execute an HLL directly, but instead presents execution hardware which is optimal for concurrent task execution. Like the conventional architectures, hardware is optimized for execution speed and this hardware needs

TABLE 1

LANGUAGE DEFINITION TABLE

A_1:	a;A_1	e;A_2B_3	
A_2:	b;A_3		
A_3:	exit -		
A_4:	e;A_3B_3	a;A_7C_2	c;A_7C_5
A_7:	d;A_3		
B_1:	e;B_3		
B_2:	exit -		
B_3:	exit	a;B_2B_2	a;B_2C_5
C_1:	a;C_2	a;C_5	
C_2:	b;C_3		
C_3:	e;C_4B_3		
C_4:	exit -		
C_5:	d;C_3		

a compiler to translate from HLL to machine code. Unlike the conventional architecture, Data Flow architectures are distributed architectures which are also optimized for convurrent processing.

Execution flow of programs for data path oriented architectures may be conceptualized as graphs which indicate data and control flow. Nodes in these

program graphs represent data functions (partical result calculations) or flow control junctions. Data functions include conventional arithmetic and logic functions and control junctions include switches which allow or impede data flow, and comparators which generate switch control information. In a data-driven architecture, function nodes are said to be active or enabled when all the inputs to that node are valid. As an example, the graph of the arithmetic expression:

$$Y = (A*B + C/D) - (E\hat{}F)$$

is represented by a path program resembling its execution tree as shown in Figure 3. In this graph, the *, /, and ^ (exponentiation) nodes are active as soon as the data operands A and B, C and D, and E and F are valid,

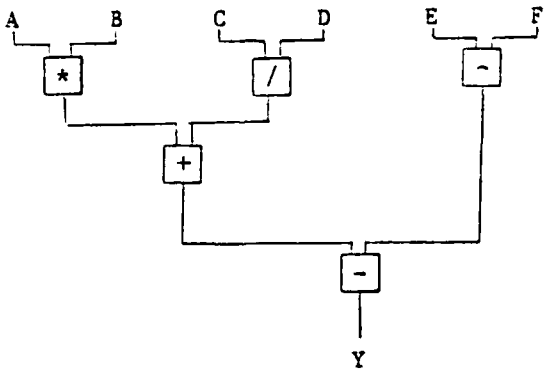


Figure 3. Arithmetic Expression Exeution Tree

respectively. When the results of the multiplication and division are both valid, the addition node is activated. As soon as this sum has been computed and the exponentiation result is valid, the subtraction node is valid.

The different types of node for a Data Flow graph, its "actors," are shown in Figure 4. An operator takes data inputs, performs the specified (arithmetic) operation producing a data output. A decider compares two data

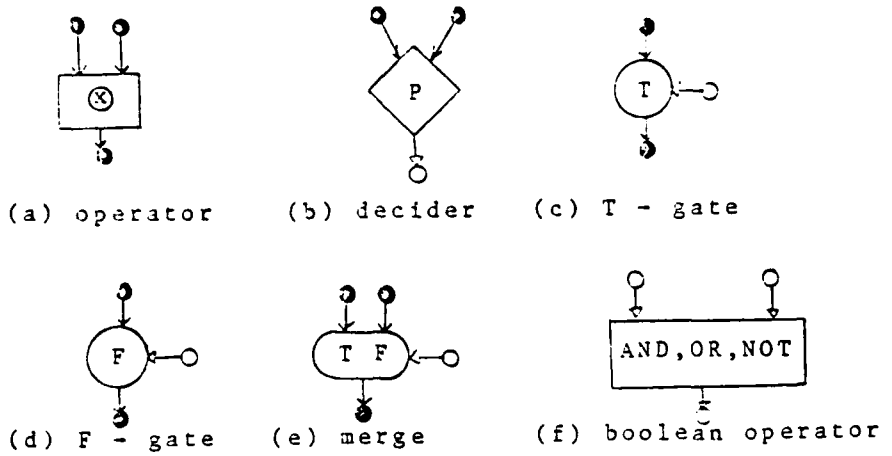


Figure 4. "Actors of the basic data-flow language."

inputs and generates either a TRUE or a FALSE control output. This control output is used by the T and F gates, and Merge actors. The T and F gates are switches, or valves, which allow data flow from their input to their output only when their control input corresponds to their type (i.e., a T gate passes data on a T control signal only, a F gate passes data on an F control signal only). The merge operator outputs one of its two data inputs depending upon the value of its control input. As an example of these operators applied to a simple program, a program to compute N Factorial:

```
INPUT N
F = 1
FOR I= 1 TO N
F = I * F
NEXT I
PRINT F
```

might be represented by a more complicated Data/Control graph of the form of Figure 5.

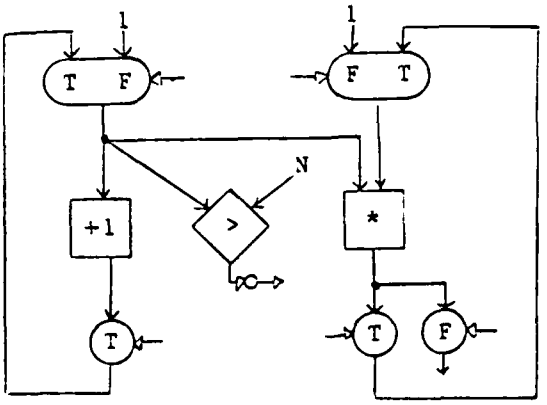


Figure 5. Data driven flow graph for N!

Execution Modules (EMs) are the functional building blocks of the Program Structured Machine. These EMs implement primitive operations which include a basic set of arithmetic and control functions. The PSM interconnects EMs so they execute in conjunction with one another to perform functions more complex than those available as primitives. This EM interconnection, which is similar logically but not physically to an Array Machine net, is also called a net.

Execution Modules are functional blocks which implement the primitive operations of the PSM. As shown in the block diagram of Figure 6, an EM performs an input specified function, Fc, on a set of inputs, X1 X2 ... Xi,

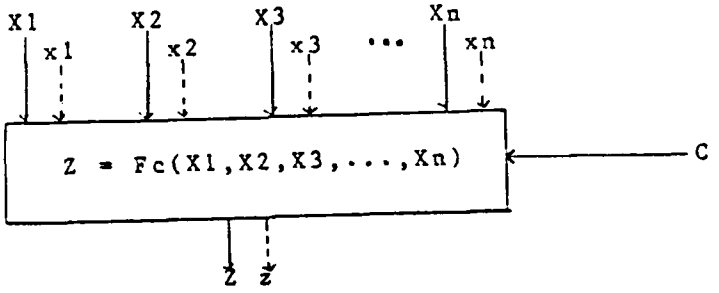


Figure 6. Execution Module Block Diagram

yielding a result which is output as Z. A "ready line" (dotted) associated with each vector of the data lines (X1 X2 ... Xi, Z) indicates when data is valid.

After an EM has received the valid input data necessary for a calculation, it performs the function Fc and outputs the valid data (on Z) and a "data ready" signal (on z).

EMs are not constrained to be identical as in the Array Machine. Some EMs may implement only a single function (e.g., hardware multiply) in which case the control input C is not necessary. Other EMs use the control input to determine which of a list of functions to execute (e.g., a general purpose ALU EM). An EM is a function block. The method used to implement its function, whether it be by software or special purpose hardware, is transparent to the rest of the machine. Functions (Fc's) performed by EMs include data operations (e.g., addition, multiplication), data storage (e.g., variable references and assignments), and sequence control functions (e.g., IF...THEN...ELSE, WHILE loops). Each of these functional modules have been investigated and tentative designs proposed.

The interconnection of these various modules and their reconfiguration for changing programs presents some difficulties. One promising approach which was investigated was a single frequency multiplexed bus. Using this approach, the program structured machine appears as shown in Figure 7. In this organization, each execution module outputs at a unique carrier frequency. The module interconnection is made by programming the various module inputs to listen to the appropriate frequencies. In its simplist form, the controller contains a single register with a field for every execution module input. These fields then define the listening frequency for the various inputs and thus the interconnection of the modules.

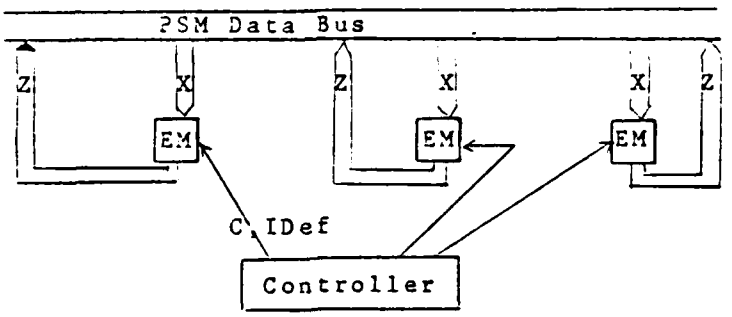


Figure 7. Block Diagram of the PSM

Various controller schemes were investigated including one in which modules could be dynamically reallocated. Thus, when a module completed its task, and is not to be used again in the program, it can be used elsewhere to perform tasks arising in later sections of the program.

The details of this work are described completely in [19].

2.2 Faulty System

Current technology, in particular VLSI, requires that system designers design in not only reliability but testability too. One problem with designing in reliability is measuring the effects of redundancy on the overall system reliability. It is conceivable that the addition of a redundant circuit could reduce reliability rather than increase it. Because of this, an effort was undertaken to derive an accurate measure of the effects of redundancy on system reliability.

Any circuit, no matter how reliable, will eventually fail. When this occurs, it is important that the problem be located and fixed as quickly as possible. Since much of the behavior of sequential circuits can be identified by breaking the feedback paths and testing the resulting combinatorial circuits, procedures were investigated which could be used in the design of combinatorial networks to make them testable for all "stuck at" faults with a very small number of test sequences.

These two efforts are described in the next two subsections.

2.2.1 Fault Tolerant Computing Structures [6, 16]

In designing fault tolerant computing systems, there is little theoretical basis on which to compare and judge the overall reliability. Yet systems requiring a very high degree of reliability are becoming more important in many scientific and business applications. One goal of this research was to develop analysis techniques which could make it possible to identify subsystem reliability and thereby locate components having a higher than average likelihood of failure. Such systems could then be redesigned to make them more fault tolerant.

In estimating the reliability of a hybrid modular redundant (HMR) system, there are two basic levels at which modeling can be done. At the module-level, the reliability, or mission-success probability, is the probability that some subset of the active and spare modules operate correctly over the length of the mission. Such a model, while providing a good first approximation of reliability, does not usually consider the effects of imperfect voting, switching and disagreement-detecting (VSD).

For an accurate model of an HMR system with imperfect, VSD, faults must be examined in the context of specific circuit inputs and states. This level of modeling, while providing extremely accurate results, quickly becomes impractical as the numbers of circuit components and inputs increase.

To combine the accuracy of logic-level reliability analysis with the simplicity of modular analysis, a method has been developed to link the two techniques through the use of module "characteristic parameters" (CP's). The CP's of a module are accurately evaluated at the logic level so that they completely describe the module behavior in the overall system. The module CP's are then combined to provide the overall reliability estimate for the system.

Consider a logic circuit C having a single output line Z. There are four possible, mutually exclusive events that could occur involving Z:

1) Z = 0 correctly
2) Z = 1 correctly
3) Z = 0 incorrectly
4) Z = 1 incorrectly

The occurrence probabilities of these four events will be used as the CP's of circuit C. In particular, if Z_d denotes the desired value of Z at some time and if Z_a is the actual value, then the CP's of circuit C are given by:

$$\Pr[Z = 0 \text{ correctly}] = \Pr[Z_a = 0, \quad Z_d = 0] \tag{1}$$

$$\Pr[Z = 1 \text{ correctly}] = \Pr[Z_a = 1, \quad Z_d = 1] \tag{2}$$

$$\Pr[Z = 0 \text{ incorrectly}] = \Pr[Z_a = 0, \quad Z_d = 1] \tag{3}$$

$$\Pr(Z = 1 \text{ incorrectly}] = \Pr[Z_a = 1, \quad Z_d = 0] \tag{4}$$

To verify that the parameters defined above completely describe the stochastic behavior of a module, consider a circuit C_0 that can be represented by the series combination of single-output modules C_1 and C_2 as in Figure 8. Define:

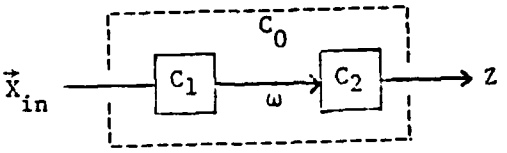


Figure 8. Series Representative of C_0.

$$T(Z) = \{\omega / Z(\omega) = 1\} \quad , \quad F(Z) = \{\omega / Z(\omega) = 0\} \tag{5}$$

where optput Z of C_0 can be represented as a switching function, $Z(\omega)$, of output ω of C_1. Let F_{C2} denote the occurrence of some fault in C_2. It can be proved that:

$$\Pr(Z_a = 0,\ Z_d = 0) = \Pr[\omega_a \varepsilon F(Z);\ \omega_d \varepsilon GF(Z)] - \Pr[\omega_d \varepsilon F(Z)]$$

$$\cdot \Pr[F_{C2}]\{1 - \Pr[Z_a = 0/\omega_d \varepsilon F(Z),\ F_{C2}]\} \qquad (6)$$

The terms on the right side of the Eq. (6) are broken down as follows:

$\Pr[\omega_a \varepsilon F(Z);\ \omega_d \varepsilon F(Z)]$ = directly related to CP's of C_1

$\Pr[\omega_d \varepsilon F(Z)]$ = directly related to CP's of C_1

$\Pr[F_{C2}]$ = physical property of C_2

$\Pr[Z_a = 0/\omega_d \varepsilon F(Z),\ F_{C2}]$ = logical property of C_2

Thus, the CP's of C_1 completely describe the behavior of C_0. Expressions similar to (6) are obtained for the other three CP's of C_0.

The evaluation of the CP's of a circuit C consists of summing the probabilities that input vector $\vec{X}_{in}$ is in the test set of each of the possible faults in C. Assume that the output Z can be written as a function of arbitrary logic line y as follows:

$$Z = yh_0 + \bar{y}h_1 \qquad (7)$$

where yh_0 and h_1 are switching functions of the input $\vec{X}_{in} = (X_1, \ldots, X_n)$ and h_0 and h_1 are independent of y. Then, the conditional error probabilities, given y stuck-at-0 (s-a-0) and stuck-at-1 (s-a-1) are:

$$\Pr[Z_a - 1/Z_d = 0,\ y\ \text{s-a-0}] = \Pr[\vec{X}_{in} \varepsilon F(h_0) \cap T(h_1)] \qquad (8)$$

$$\Pr[Z_a = 1/Z_d = 0,\ y\ \text{s-a-1}] = \Pr[\vec{X}_{in} \varepsilon T(h_0) \cap F(h_1)] \qquad (9)$$

$$\Pr[Z_a = 0/Z_d = 1,\ y\ \text{s-a-0}] = \Pr[\vec{X}_{in} \varepsilon T(h_0) \cap F(h_1)] \qquad (10)$$

$$\Pr[Z_a = 0/Z_d = 1,\ y\ \text{s-a-1}] = \Pr[\vec{X}_{in} \varepsilon F(h_0) \cap T(h_1)] \qquad (11)$$

If it is assumed that at most one fault can be present in C at any one time, then:

$$\Pr[Z_a = 1,\ Z_d = 0] = \sum_{y \in C} [\Pr(Z_a = 1/Z_d = 0,\ y\ \text{s-a-0})\ \Pr(Z_d = 0)\ \Pr(y\ \text{s-a-0})$$

$$+ \Pr(Z_a = 1/Z_d = 0,\ y\ \text{s-a-1})\ \Pr(Z_d = 0)\ \Pr(y\ \text{s-a-1}) \quad (12)$$

and

$$\Pr[Z_a = 0,\ Z_d = 1] = \sum_{y \in C} [\Pr(Z_a = 0/Z_d = 1,\ y\ \text{s-a-0})\ \Pr(Z_d = 1)\ \Pr(y\ \text{s-a-0})$$

$$+ \Pr(Z_a = 0/Z_d = 1,\ y\ \text{s-a-1})\ \Pr(Z_d = 1)\ \Pr(y\ \text{s-a-1}) \quad (13)$$

If multiple faults are to be considered, then an equation similar to (7) can be written in terms of the logic lines of int-rest and expressions similar to those of Eqs. (8) through (11) used to evaluate the various conditional error probabilities.

This work has also been extended to sequential circuits. It is currently being applied to the analysis of a fault tolerant redundant structure which has been propsed that requires no perfect elements. Figure 4 shows a block diagram of a memory module of this multiple structured machine. The details of this work may be found in [6, 16].

2.2.2 Designing in Testability [20]

Given a combinatorial network in a system which is required to be highly reliable requires that diagnostics be available so that faults which may occur in the network be located and replaced as quickly as possible. In highly reliable systems redundancy required to maintain its appropriate reliability will also mask the identification of faults. Thus an effort was undertaken to develop algorithms to add observation and control points in combinatorial circuits to make possible the identification of all stuck-at type faults regardless of the redundancy.

Specifically, two algorithms were developed. In the first, an analysis procedure was investigated which identifies redundancy in combinatorial networks.

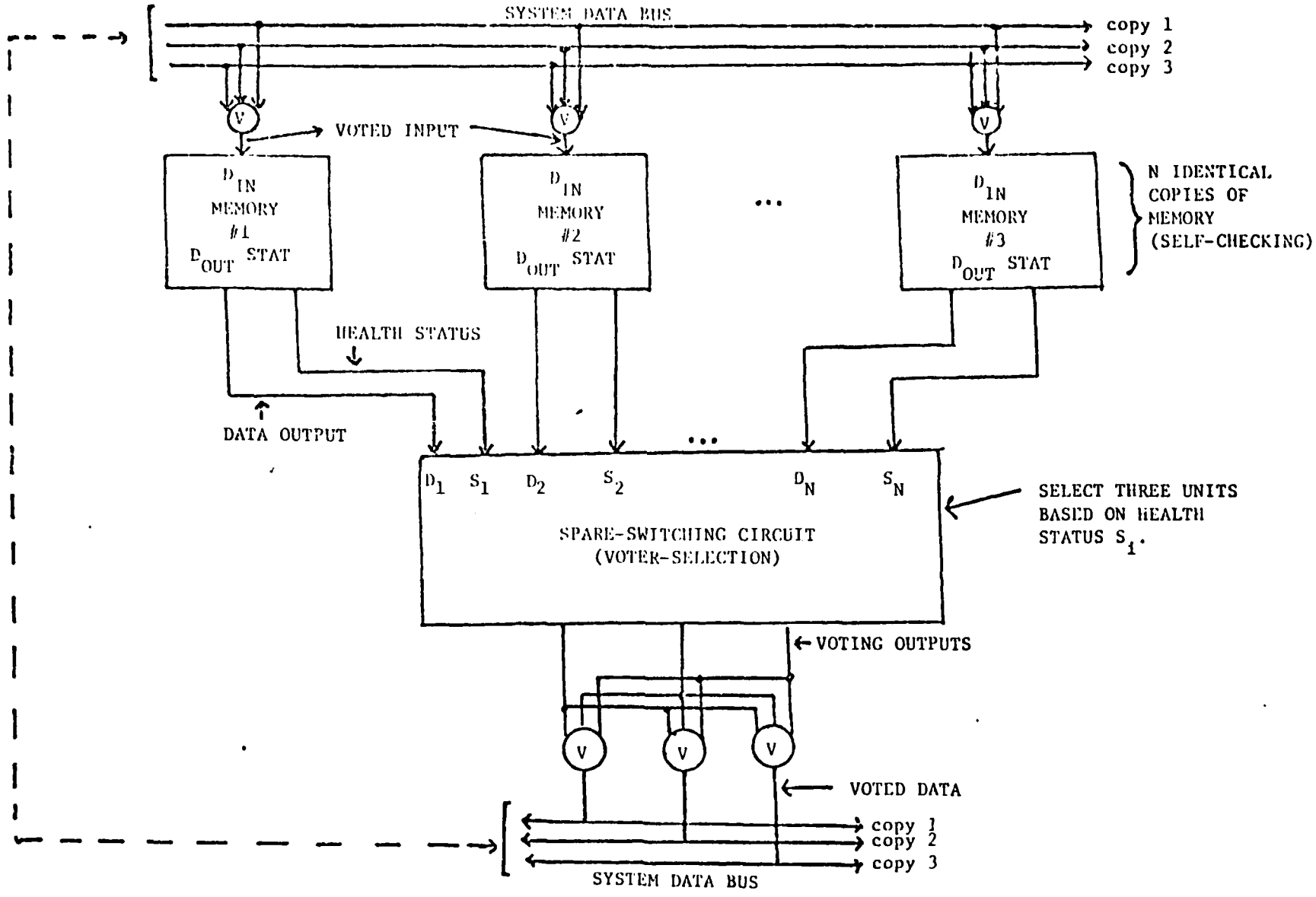


Figure 9. Block Diagram of Redundant, Common Memory Module. All units are operating with three units selected by the switching circuit to participate in voting.

Based on this information a systematic procedure for redesigning the circuit is presented which allows a minimum number of test points and control inputs to be added for complete identification of all stuck-at faults. The second algorithm creates the test set for a complete analysis of all single stuck-at faults. This test set is near minimal and is generated in a very straight forward manner. The details of this work are described in detail in [20].

2.3 Multi Processing System [1,2,3,17]

An increasing number of real-world problems involve the processing of data in an array fashion. Whether the problem involves image processing, spectrum analysis, or meteorological predictions, a very large amount of computing power is required--not to mention system software. To investigate the possibility of special purpose hardware helping resolve these load problems, five tasks were undertaken:

a) a 128 x 128 bit, binary array processor was constructed,
b) a high level array processing language, which is a superset of Fortran, called FAPL, was developed,
c) a general mathematical model for the study of neighborhood transformations was completed,
d) a preliminary study of arrays of microprocessors for complex, two-dimensional system modeling was begun, and
e) inexpensive facsimile equipment coupled to an Apple computer was used to process images.

The results of these projects are described in what follows.

2.3.1 Image Array Processor

It has been proposed that the processing of real time images can be performed most efficiently when the structure of the processor closely resembles the structure of the data to be operated upon. Thus, one would expect that image processing operations can be implemented more easily within an array processor than the normal serial processor. Basically, an array processor

consists of a tesselation of the plane by processing elements, each executing the same instruction stream issued by a single central controller. Such an array processor has been built and is currently in operation. This array processor is implemented as a finite 128 x 128 array of one bit binary processors with facilities built in for expansion to 256 x 256 [1,2]. A rectangular tesselation is used, with each processing element possessing connections to its eight nearest neighbors as well as to its own private accumulator and memory as shown in Figure 10.

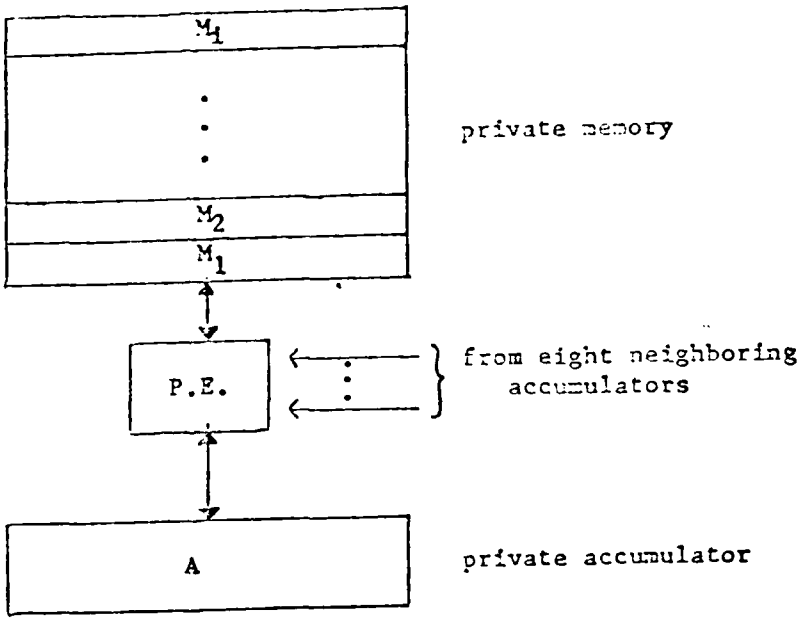


Figure 10. Array Processor Element

Each processing element can form any of the 16 possible Boolean functions of two variables via the Bus Function Generator (BFG) using any two of the following as inputs: the present contents of its accumulator, the present contents of any of its memory cells, and a threshold function of the accumulators of its eight nearest neighbors, denoted f(A). Specifically,

$$f(A) = 1 \qquad \text{if } \sum_{1}^{8} Ai \cdot Wi \leq T \tag{14}$$

$$0 \qquad \text{otherwise}$$

where

$Wi \in \{+1, 0, -1\}$, the 8 weights

$Ai \in \{0, 1\}$, the i^{th} out of 8 nearest neighboring accumulators

$T \in [-8, 7]$, the threshold (15)

This organization is summarized in Figure 11. A bus-oriented structure was used for maximum flexibility in interconnecting the various registers and for future expansion of additional I/O devices, such as a direct TV camera interface and various hard-wired masks. Inputs to the BFG consist of the Main (M) and Secondary (S) buses, each of which can be driven by any of eight possible devices selected under program control. Any or all of the devices can receive information from the Output (O) bus simultaneously, also selected under the program control.

Because of cost considerations, the processor was actually implemented as a serial realization using dynamic shift register memory. This fact, however, is transparent to the user. Pipelining was used extensively in the threshold logic and in the BFG, allowing a clock rate of 5 MHz--the limit of the dynamic shift register memory. Instruction execution times are on the order of 10 ms., as compared with almost two seconds on the array processor simulator which ran on the PDP-9. Image transformations of only moderate complexity consisting of perhaps 2000 operations required over an hour of computer time using the software simulator. The time for such a transformation has been cut to 20 seconds with the array processor, allowing the experimentation with transformations which had been timewise impractical to attempt.

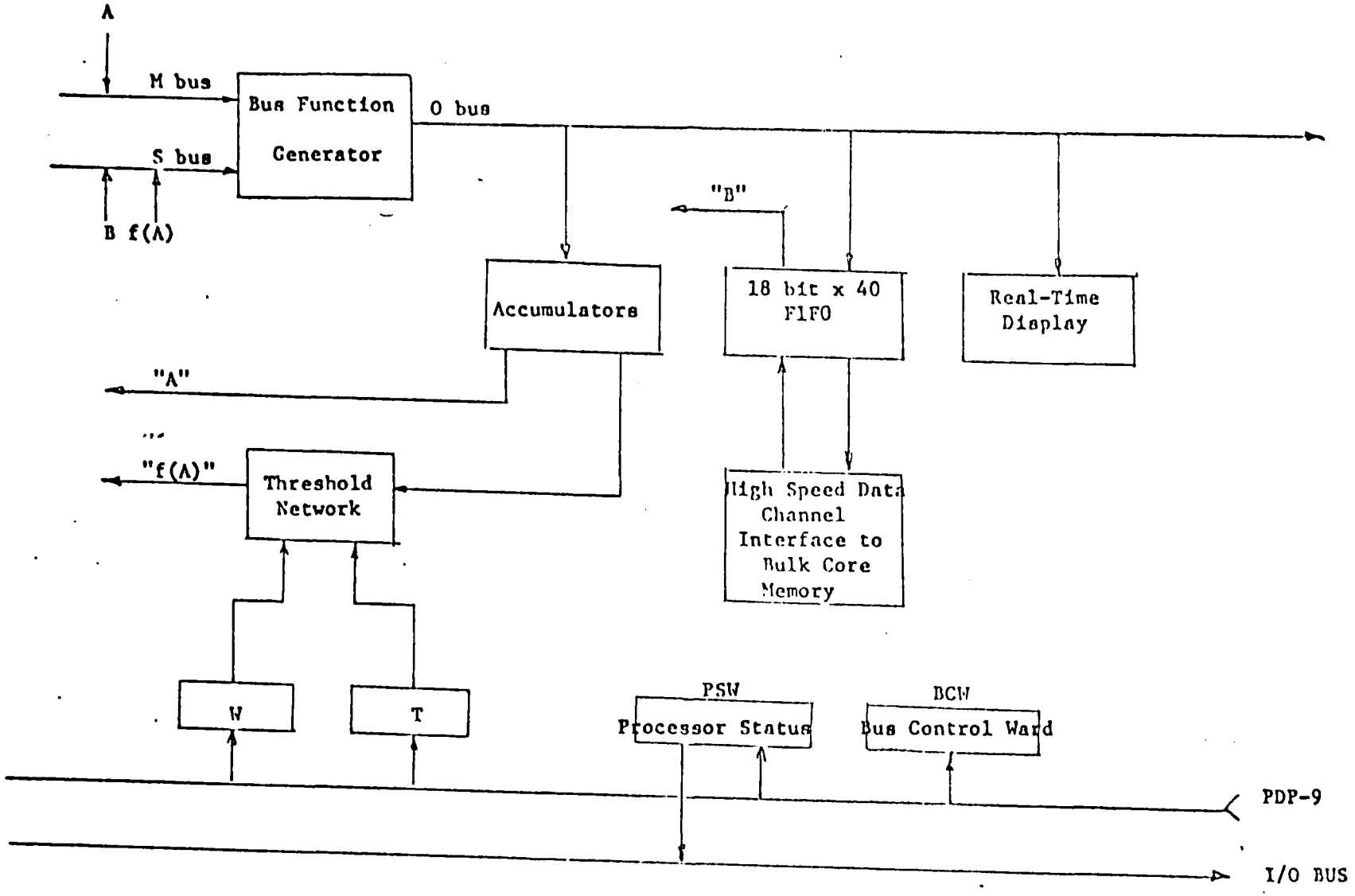


Figure 11. Array Processor Organization

The use of special-purpose hardware is facilitated greatly by high-level languages which relate software constructs to hardware capability. Because of this an array processing language was developed for this system which is a superset of Fortran and has been dubbed FAPL for Fortran Array Processing Language. One goal of this language is to make software portable between two installations which both have array processors of the type described above. To this end, the syntactic definitions of FAPL for the given installation are written in Backus Naur Form (BNF). These definitions are then converted into a tabular form suitable for deriving the parser by an auxiliary program. Both the auxiliary program and the parser were written in Fortran for immediate portability. The parser sets up the semantic processes by emitting object code for host machine. Thus only the object code emitters and the BNF syntax definitions need be altered to make a FAPL program portable from one system to any other. Figure 12 shows the organization of the FAPL compiler. The details and the syntactical descriptions are described in [1].

2.3.2 A Mathematical Model for Array Processing Neighborhood Transformations [3]

In application, array processors operate on data in some local neighborhood, i.e., data coming from neighboring processors. To help in the understanding of such processes, the development of mathematical foundations and models for a neighborhood transformation was undertaken. This investigation was aimed primarily at image processing but easily generalizes to other forms of array processing.

Neighborhood transformations are image transformations in which new values of each element is a function of its "neighborhood" elements. Currently, neighborhood transformations are not particularly useful except in low-level processing functions such as extraction of the perimeter (boundary) of objects, object counting, or "skeleton" extraction. The value of neighborhood transfor-

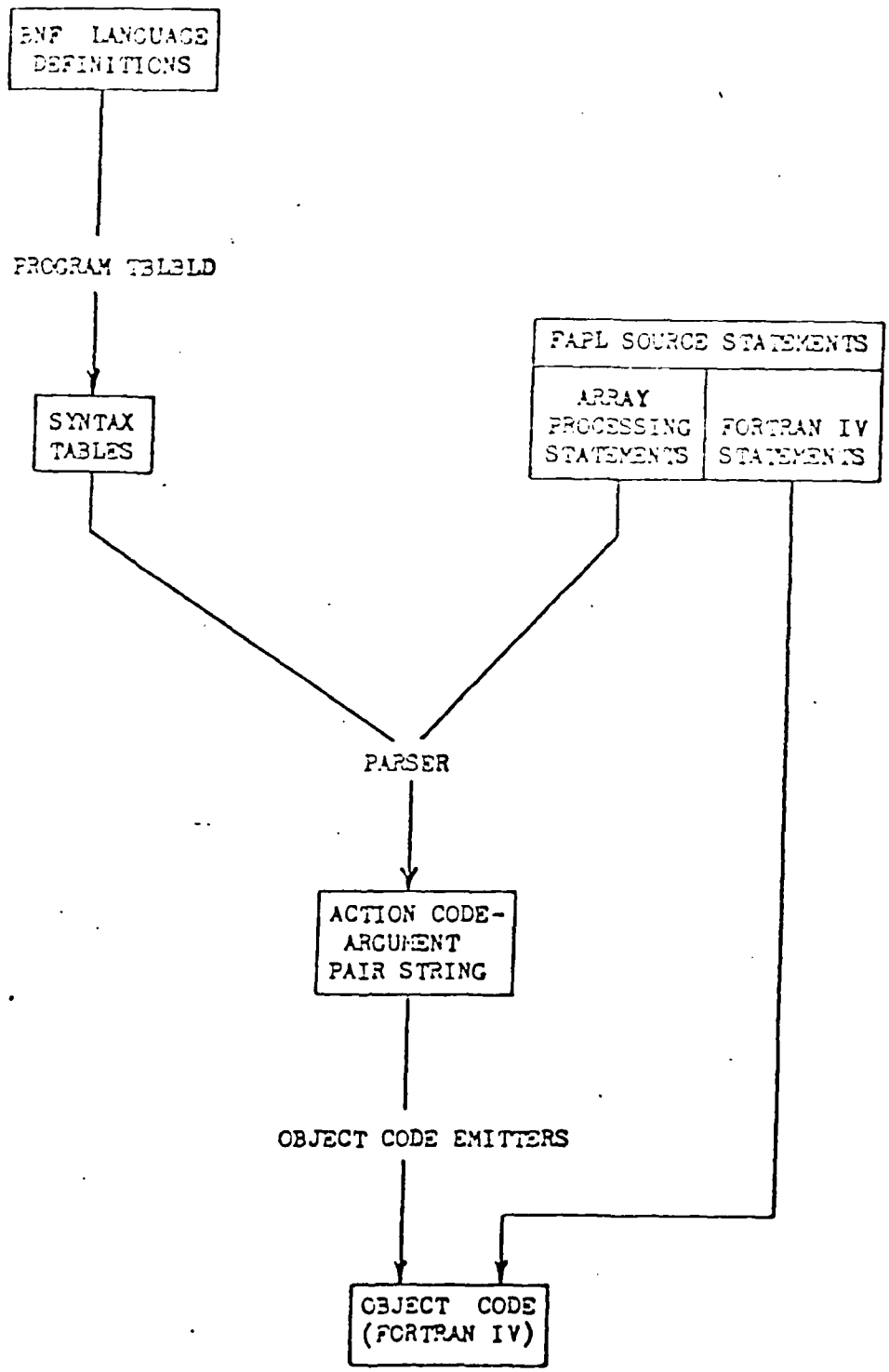


Figure 12. FAPL Compiler Organization

mations lie in the speed at which they can be performed, especially with binary (black and white) images. An array processor, which is a uniform interconnection of elementary processors ("cells"), can perform single neighborhood transformations in one clock time since the transformations for all elements are computed simultaneously. The object of this investigation is to develop an algebraic theory of neighborhood transformations to allow the use of a small set of transformations to become a set of primitives which can be used as necessary to build more sophisticated functions.

Since the purpose of this research is to develop a general mathematical foundation for the use of neighborhood transformations, image and image operations are defined abstractly. An image is an infinite, n-dimensional array of elements, each of which is labelled with a unique n-tuple of integers. The values that the elements can take are members of a Boolean algebra. The images are all considered to be bounded; that is, all the image elements outside some finite region within the image have a common value, known as the background constant of that image.

Five image operations are defined. Image complementation, produce, sum, Boolean convolution, and Boolean division. The first three are the corresponding Boolean operations applied element-by-element to images. Boolean convolution, denoted "*," is the discrete convolution between image elements, using Boolean sum and product. Boolean image division between images A and B is defined in terms of Boolean image convolution and image complement: $A \div B = \overline{\overline{A} * B^t}$, where the element B^t are the elements of B reflected about the origin for each neighborhood. Boolean convolution and division are shown to be generalizations of neighborhood transformations commonly known as expansion, or dilation, and shrink, or erosion.

An algebraic structure for bounded, Boolean-valued images was developed with several lemmas and theorems. This structure, a hybrid of integer-like

arithmetic and Boolean algebra, provides a framework for the analytical manipulation of images. The theorems were applied to the solution of an image equation, providing necessary and sufficient conditions for solutions as well as bounds on the solution. One theorem, a division algorithm for images, provides for a "power" series representation of images, analogous to the common integer radix expression. This power series is shown to have several applications, including template matching, feature extraction, image filtering, and data compression. Several known applications of neighborhood processing, such as noise removal, and boundary smoothing, are shown to be special cases of applications of the series representation of images.

Several application considerations are approached, including minimizing boundary effects in finite image windows and possible extensions to real-valued images. Preliminary investigations suggest that images whose elements are probabilities of being white (or black) belong to a similar algebraic structure.

Further research will focus on refinement of the algebraic structure, extensions to real-valued images, and application of the theorems. It appears that there are yet many unexplored areas in analytical properties of images, especially with respect to a class of functions described as the interior and closure of Boolean-valued images. The applications of this research has been primarily generalizations of known applications, and it is expected that a new set of feature extraction algorithms can be developed from this framework. The details of this work may be found in [3].

2.3.3 A Multiprocessor Array Structure [17].

The modeling of spacial problems by conventional, single processor systems generally creates some very real and difficult software design

problems. The main difficulty arises because of two factors. The first is the overhead requirement for managing the multiple data sets representing the various spatial model points. The second difficulty arises for problems in which the physical model varies from one point in space to another and this requires different programs to represent each model. An example might be the modeling of stresses in metals which contain impure particles, grossly different in characteristic from the surrounding matrix.

The development of software systems for representing such problems can be an extremely time consuming and costly effort. To help alleviate the difficulties encountered and thereby reduce development costs, a special-purpose computer structure is being investigated which should be capable of eliminating all of the overhead which is required by conventional, single processor simulations. This single processor structure basically consists of a conventional microprocessor, in this case a Motorola 6809, a large block of semiconductor memory and special memory addressing hardware. Figure 13 shows a block diagram of this structure. Simulation processing proceeds as follows.

The single processor executes the simulation by assuming that it is a single node of the array, performs the necessary computations for that node, and setps on to the next node. This node scanning is done in a raster fashion. The added addressing hardware allows the processor to load two registers (X and Y) with coordinates of the node to be processed. The processor then need only specify which of the adjacent nodes is to be accessed, and the relative location of the desired data within that node. The addressing hardware performs the necessary address corrections to access the proper location in memory. Provision is also made for the address hardware to properly recognize and direct boundary condition addressing, and new data generated for the next iteration of the array.

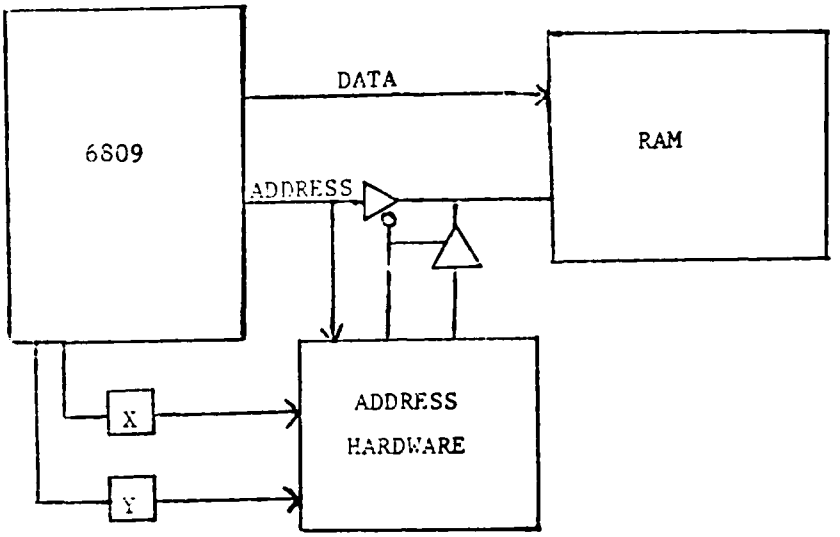


Figure 13. Single Processor Array Processor Simulation.

A design for this structure has been completed and a simulation of the structure is currently being written for a DEC PDP-11/03. More details will be discussed in Section 4 and a complete description of this effort will be presented in [17].

2.3.4 Fascimile Image Processing [21]

A small effort was undertaken to implement image array processing at a very low cost. To this end, a standard Qwip Facsimile transceiver was interfaced to an Apple II computer. Such a system has a number of advantages not least of which is a simple system for long distance image transmission and processing. The images processed in this small effort were primarily medical images. It was found that horizontal resolution of about 1/3 mm and vertical resolution of

roughly the same was possible with a gray level resolution of 8 levels from black to white.

Two experiemnts were undertaken. In one, the circumference of tooth root sections were made. This was accomplished by storing the image digitized by the fascimile equipment and then tracking the boundary of the tooth. The second experiment involved measuring the crystae density in heart cell mitochondria. This was done by forming a spatial first difference at each fixed and summing the results after thresholding. Since the crystae have a very striated structure, a large first difference will be associated with their presence and a small value otherwise. Both experiments were relatively successful and this demonstrated the feasibility of using such simple equipment for a rather specialized and complex problem. Details of this work may be found in [21].

2.4 Distributed Loop Networks

One final area of research which was completed during the past year involved an investigation of a Distributed Loop Computer Network (DLCN). The results of this work have been detailed in [7 to 14] and are briefly summarized below.

1. A loop interface design using the state-of-the-art, off-the-shelf LSI components (the Am2900 series) has been completed [8]. The microprocessor-based implementation will allow great flexibility, power and speed of transmission for the loop at low cost. The design is general enough so that the interface can be used for any type of distributed loop networks [14].

2. A performance study of three popular types of loop networks using queueing analysis has been completed [7]. This study veritifes our previous claim that the DLCN loop has superior performance (shorter message delay and higher channel utilization) over both Newhall and Pierce loops.

3. Performance studies of both DLCN loop communication subnet and whole DLCN network ware also carried out in [9, 11]. Closed formulas for

calculating several design parameters of interest (viz., average response time to the user, mean queue lengths, channel and processor utilizations, etc.) have been obtained.

4. Architectural design of the Distributed Loop Data Base System (DLDBS) for DLCN has been completed [10]. The system architecture of DLDBS is very general, so that new concepts can be applied to many system environments.

5. A general model is proposed for distributed processing that is based on interactions among independent processes [12]. This model generalizes the hierarchical structure that is traditionally used as the model of control and that forces a parent-son relationship between processes. It is shown that the traditional hierarchical control structure can be easily simulated as a special case of our general model.

6. A new class of high-level protocols, called N-process communication protocols, has been proposed to manipulate exchange of messages among the N processes [13]. This class of protocols is a generalization of popular two-process protocols currently used by the ARPANET and NSW, and is application dependent.

The major goal of DLCN research has been to investigate the feasibility of providing efficient, inexpensive, reliable and flexible computing service for a local user community. Such a computing environment is frequently found today in many military installations and university campuses. DLCN system design involves many innovative ideas and careful integration of hardware, software, and communication technologies.

3. Publications

Most of the work described above has been or is being prepared for publication in some form. The following list represents all reviewed publications produced with the support of Grant AF-AFPSR-77-3400. Papers currently in preparation or which have been submitted for publication but have not yet appeared are also listed.

1. Robinson, C.S., A Binary Image Array Processor: Hardware Design and Language Development, Ph.D. dissertation, The Ohio State University, Columbus, Ohio, June, 1977.

2. Robinson, C.S., and Breeding, K.J., "The Hardware Design and Language for a Binary Array Processor," submitted to the IEEE Trans. on Computers, currently in revision.

3. Miller, P.E., The Extraction of Image Features Using a Binary Array Processor, Ph.D. dissertation, The Ohio State University, Columbus, Ohio, December, 1978.

4. Mooney, J.D., Design of a Variable High Level Language Computer Using Parallel Processing, Ph.D. dissertation, The Ohio State University, Columbus, Ohio, December, 1977.

5. Mooney, J.D., and Breeding, K.J., "A General High Level Language Computer," submitted to IEEE Trans. on Computers.

6. Nelson, V.P., An Accurate Reliability Modeling Technique for Hybrid Modular Redundant Digital Systems Using Modular Characteristic Error Parameters, Ph.D. dissertation, The Ohio State University, Columbus, Ohio, December, 1978.

7. Liu, M.T., Pardo, R., and Babic, G., "A Performance Study of Distributed Control Loop Networks," Proc. 1977 International Conference on Parallel Processing, pp. 137-138, August, 1977.

8. Oh, Y., and Liu, M.T., "Interface Design for Distributed Control Loop Networks," Proc. of 1977 National Telecommunication Conference, pp. 31.4.1-6, December, 1977.

9. Liu, M.T., Babic, G., and Pardo, R., "Traffic Analysis of the Distributed Loop Computer Network (DLCN)," Proc. of 1977 National Telecommunication Conference, pp. 31.5.1-7, December, 1977.

10. Pardo, R., Liu, M.T., and Babic, G., "Distributed Services in Computer Networks: Designing the Distributed Loop Data Base System (DLDBS)," Proc. Computer Networking Symposium, pp. 60-65, December, 1977.

11. Babic, G., Liu, M.T., and Pardo, R., "A Performance Study of the Distributed Loop Computer Network (DLCN)," Proc. Computer Networking Symposium, pp. 66-75, December, 1977.

12. Liu, M.T., and Ma, A.L., "Control and Communication in Distributed Processing," Proc. 1977 International Computer Symposium, pp. 123-138, December, 1977.

13. Pardo, R., Liu, M.T., and Babic, G., "An N-Process Communication Protocol for Distributed Processing," Proc. of Computer Network Protocols Symposium, pp. D7.1-10, February, 1978.

14. Liu, M.T., "Distributed Loop Computer Networks," accepted for Advances in Computers, Vol. 17 (m. Rubinoff and M.C. Yovits, editors), Academic Press, New York, 1978.

15. Lee, J., The Analysis Unit of a General High Level Language Computer, M.S. thesis, The Ohio State University, Columbus, Ohio. In preparation, expected completion, June, 1981.

16. Nelson,V.P., and Breeding, K.J., "A Statistical Analysis Technique for Fault Tolerant Structures," submitted to IEEE Trans. on Computers.

17. Bailey, R.E., A Single Processor Emulation for General Array Processing Application, M.S. thesis, The Ohio State University, August, 1979.

18. Klein, C.A., and Breeding, K.J., "Automatic Optical Identification of Faults in Bubble Memory Overlay Patterns," presented at 1979 Pattern Recognition and Image Processing Conference, August, 1979, Chicago.

19. McCord, Jr., Charles F., A Preliminary Study of the Architecture of a Program Structured Computer, M.S. thesis, The Ohio State University, August, 1980.

20. Chugh, R., The Design of Combinatorial Networks Testable by a Small Number of Test Patterns, M.S. thesis, The Ohio State University, June, 1980.

21. Chan, K., A Facsimile System for Image Processing, M.S. thesis, The Ohio State University. In prepraration; expected completion, June, 1981.