INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106

RICE UNIVERSITY

A HIERARCHICAL, RESTRUCTURABLE MULTI-MICROPROCESSOR ARCHITECTURE

by

Robert Glen Arnold

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

Thesis Director's signature:

Edward Alvin Feustel

Houston, Texas

May, 1976

ACKNOWLEDGMENTS

I would like to take this opportunity to express my deepest gratitude to all those who have assisted in the conduct of this research. In particular, I would like to thank the members of my research committee, Dr. Sigsby Rusk, Dr. Ken Kennedy and especially my thesis advisor, Dr. Edward Feustel.

In addition, my thanks go to Dr. Robert Minnick, without whose assistance, encouragement, and friendship I would never have begun a graduate program; to Dr. Ed Page for the hours of stimulating discussion on this and related subjects; and to Jeanne Fulton for the many long hours she devoted to the transcription of this manuscript. My wife, Natalie, has given me constant encouragement and support for which I will be enternally grateful.

This work has been supported in part by a Rice Research Fellowship and by NSF Grant OCA GJ36471-1.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I. INTRODUCTION

This thesis presents a novel architecture for a data processing system based on a large number of identical modules containing microprocessors. It is intended that this architecture provide a flexible, effective data processing system that may be dynamically restructured to simplify its use and enhance its reliability. A major emphasis maintained throughout this work has been to assure that the architecture developed is practical. As a consequence, the description of modules and the discussion of the examples have been couched in terms of devices that are currently available. Hopefully, parts more specifically adapted to an architecture of this nature will become available in the future and will provide improved performance and simpler construction. The examples discussed have been chosen in part because they are somewhat atypical of applications proposed for the efficient application of multiprocessors. This is to emphasize the flexibility and general purpose nature of this system architecture. In addition, the constraints placed upon the software and firmware by the system are mentioned and a few convenient features are specified.

The discussion of the system architecture begins with a brief description of the goals derived for the system. Next, the historical background of computer architectural development is discussed along with general, historically significant multiprocessor schemes illustrating the typical tightly coupled system. A brief description of the work of T.C. Chen [13] concerning the efficiency of tightly coupled systems is included to provide some motivation for the development of the architecture that is the subject of this work.

The second major section of this thesis describes

concepts and major submodules incorporated within this architecture. The description is presented at a relatively abstract level to provide the reader with an overall conceptual view of the system and its operations without becoming bogged down in detail unnecessarily.

The interprocessor communication facility is basically a set of Pierce Loops with modifications for incorporation in this architecture. The aspects of this communication structure related to delay and the overhead added to the system operation are analyzed in detail. The results of this analysis provide a general characterization of the system operation. Strategies for the application of the architecture are developed based on this analysis. For a further discussion of Pierce Loops, the reader is encouraged to refer to the excellent papers by Pierce [34] , Avi-itzak [3] , and Hayes and Sherman [25]. Verification of the heart of the analysis is provided by the results of a computer simulation of a series of processors communicating via this structure.

One of the common deficiencies of traditional multiprocessors is programming difficulty. In learning from these examples, we can see that it is important to consider the software aspects of a new structure to ensure that the system can be made effective in the solution of problems without unreasonable software demands. On the other hand, it would be impossible to discuss completely the wide range of programming techniques available on as versatile an architecture as described here. The majority of the software always relies upon the ingenuity and imagination of the individual programmer. However, the basic philosophy of the software is presented and includes several examples. One example is presented in extreme detail for a simple,

narrow problem; a second is discussed at a much higher level for a more complex problem and includes routines typical of those that would be found in an operating system for this architecture.

A. Goals

The goal of this thesis is to develop an architecture for a computing system employing a large number of microprocessor based modules as building blocks. These modules will be linked together by a collection of busses in such a manner that the total system architecture will be extremely flexible, allowing the resources of the system to be configured to satisfy the requirements of the particular application. In addition, it is intended that as the application changes, the structure and distribution of the resources may also change. This flexible, restructurable control system will also provide enhanced reliability through the use of redundant modules that may be incorporated into the operation by dynamic reconfiguration and other provisions for graceful degradation.

It is not the purpose of this work to provide a detailed implementation of this system but rather to provide the necessary background required before a system of this nature is constructed. Discussions of possible implementation of hardware and software are included. These are intended to further the reader's understanding of the architecture and indicate how the system modules might operate and how they could be programmed and used.

It is assumed throughout this thesis that there will be a large number of modules within the system, each of which is of insignificant cost. In order to maintain this assumption, extreme simplicity is stressed for those elements of a module exclusive of the microcomputer(microprocessor and memory). In addition, a large supply of redundant modules is assumed to be avail-

able. Although it is not basic to this study, it is also assumed that each microcomputer is relatively slow in comparison to conventional computers or minicomputers.

B. Background

The history of computer development shows a constant demand for more processing power and speed. This demand has been held in check, however, by the economics of the technology involved. The size, power, and speed for a system have been limited by the financial constraints on the builder and his customers. As a consequence, a major emphasis throughout computer research and development has been directed at the reduction of price and size for a given performance or the improvement of the price performance ratios. Generally, a major means of providing this improvement has been the introduction of a new technology in which the basic switching elements are constructed. Each new technology introduced has usually increased the speed of switching elements by an order of magnitude while simultaneously reducing its cost. The results have been significant improvements in price performance ratios. A second means of improvement that has also been dramatic has been the innovation in memory systems producing an almost linear decrease in cost versus bits of information stored. In addition, the improvement in memory systems has continually increased their speed.

As impressive as technology's past advances have been, it is clear that we are currently approaching a turning point that will have a significant impact on the path of continued computer development. The increased speed of a logic gate is no longer of major concern when compared to the propagation delays of the transmission lines connecting gates. Indeed, it is generally accepted that little advance in computing speed can be expected from future advances in the technology of basic gates [Hobbs, 27].

Simultaneously with the increase in the speed of switching circuits, the size of a logic gate has been diminishing. Large scale integration (LSI) has provided the ability to place several thousand logic gates in one package with a highly automated process. The resultant efficiency of production of a gate provides cost/gate and price-performance indexes that are much better than previously available.

This evolution in digital hardware has taken us from room sized central processing units (CPU's) to CPU's that can be placed in the palm of a hand. The associated prices have been reduced from hundreds of thousands of dollars to hundreds of dollars or less for a CPU of comparable power. Today, the cost of a CPU is generally an insignificant fraction of total system cost. The historically consistent demand for faster, more powerful machines is still being maintained, however. We can be sure that it will continue to exceed the capability provided by improved switching elements alone [Chen, 13]. As a result, this demand must be satisfied with innovative architectures.

Historically, economics has also forced the efficient utilization of a CPU. The users have been required to maximize the processing capability of a computing system and realize the CPU's full potential. To satisfy this requirement, an enormous effort and expense was often directed toward the software used on the system. Thus, the cost of the hardware and software have often been of comparable orders of magnitude.

Today, this is not necessarily the case. A CPU in the form of an LSI microprocessor can be economically employed as a universal logic module performing a few simple tasks. Although the majority of the processor's

capability might be wasted, the savings in design effort and lower maintenance costs are far more significant. In addition, the parts cost using microprocessors will often be less than that of the discrete gate logic performing an equivalent function. This is due to the increased production efficiency of LSI.

The result is that the computer architect today is now released from the incessant demand for efficient processor utilization and should begin to intensify the search for innovative applications of processors. Instead of using a single microprocessor at the limits of its capability, the emphasis needs to shift to systems of processors, each performing a set of relatively simple tasks, functioning as a team to provide a quick, economic solution to the particular problem at hand. In addition, these systems should be extremely flexible in order to more adequately fit the computer system to the problem at hand. This flexibility can be realized by employing the inherent power of the processor not otherwise required in the team concept above. The flexibility of a multiprocessor can also be exploited to provide for fault tolerance and graceful degradation. This assumes that provisions for dynamic reconfigurability and triple modular redundancy of a "hard core" test and repair section are incorporated into the system architecture. The STAR computer developed by JPL and discussed by Martin [30] has demonstrated that these techniques can produce highly reliable systems with calculated, apriori probability of failure during a ten year period as low as 0.01 .

Although software is secondary to this research, it seems evident that by reducing the demand for maximum efficiency in the application of a processor, partitioning the problem into conceptually simple blocks and employing a flexible system that easily and naturally adapts to the

problem, the cost of software should also be reduced.

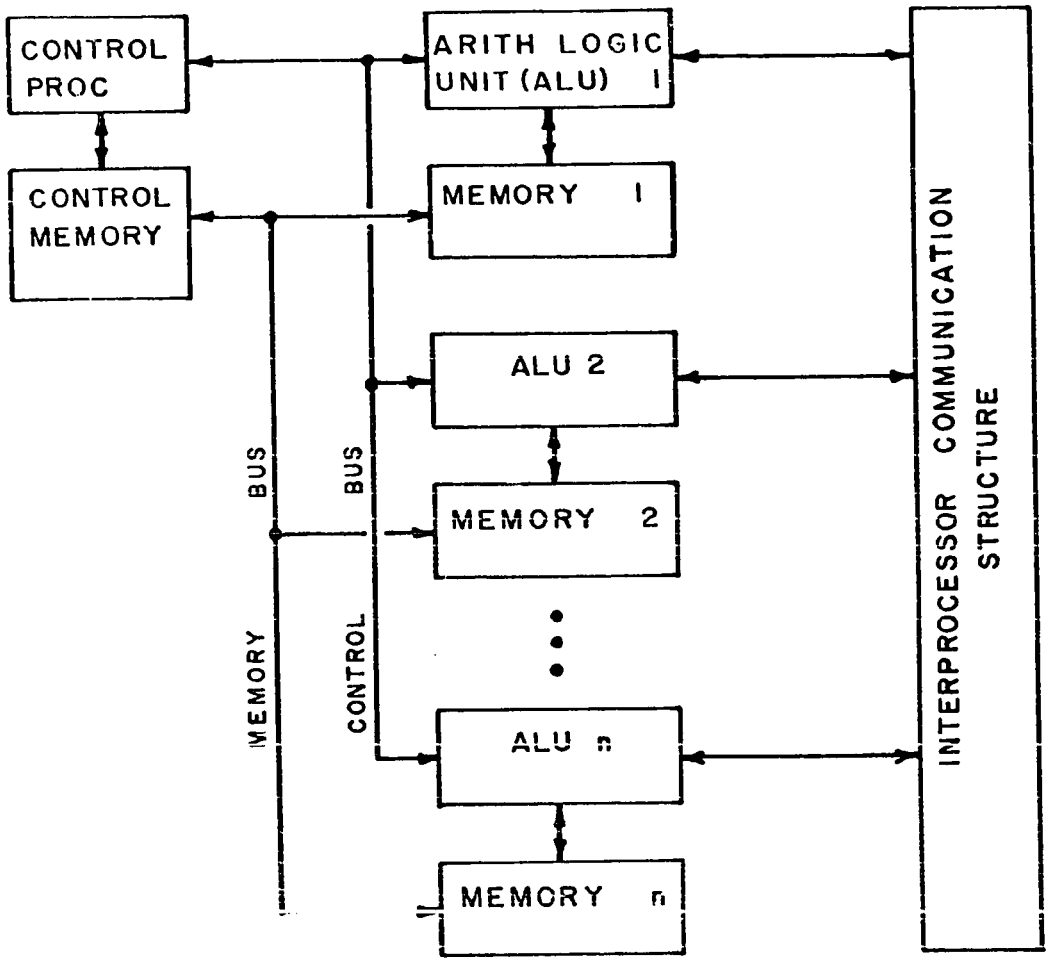
As a place to begin the development of a new multiprocessor architecture, let us first consider the general multiprocessor. In the quest of performance beyond that provided by circuit speed alone, several alternative approaches to multiprocessor design have been taken. One alternative is to subdivide the job to be performed and distribute the sections among many identically constucted processing units. This is commonly termed "parallelism". Another is to form a general purpose arrangement of processing units capable of execution of a portion of a task concurrently with the execution of a succeeding, independent portion of the same overall task. This is generally called "overlap" or "pipelining" [Chen, 13].

Flynn [22] has discussed very high speed computers and recognized that there are several types of architectural organizations employing concurrent operation. He has broken computer systems into four groups:

1. Single Instruction Stream-Single Data Stream (SISD)
2. Single Instruction Stream-Multiple Data Stream (SIMD)
3. Multiple Instruction Stream-Single Data Stream (MISD)
4. Multiple Instruction Stream-Multiple Data Stream (MID)

Stone [51] has further discussed these groups with traditional examples of each (as has Enslow, [15]). A general characteristic of these systems has been that the control arrangements have become less flexible as the number of processors increased.

Consider, as an example, the organization of a somewhat typical SIMD computer. (This description draws heavily on the discussion of the same topic by Stone, [51] and indirectly on the ILLIAC IV). A typical SIMD computer is shown in block diagram form in Fig. I.B.1.



BLOCK DIAGRAM OF TYPICAL
SINGLE INSTRUCTION MULTIPLE DATA
STREAM ARCHITECTURE

Fig. I.B.1

Within this system, each major subelement (ie. memory, arithmetic logic unit (ALU), etc.) is provided with a means of communication with each other major subelement of the same type and indirectly to all other subelements. Thus, it is possible to transmit data throughout the system and to make use of the intermediate results of one processor in another. The control processor is a powerful computer having its own arithmetic capability, registers, memory and the ability to perform conditional branches, etc. On the other hand, although the arithmetic logic units (ALU's) are computers capable of sequentially executing a series of instructions, they are required to be in global synchronization. (Each task or set of instructions in each processor starts simultaneously with that of all other ALU's. No ALU may initiate a second task until all ALU's complete the first task.) All ALU's must either pause or perform the same task. As such, the collection of ALU's is incapable of conditional branches. Each ALU's task is specified by the control processor. It is the function of the control processor to interpret the instruction stream provided by the programmer. Each instruction is either a control instruction and executed by the control processor, or it is a vector operation and broadcast to all the ALU's.

Since an SIMD system can support only one instruction stream, the occurrence of a conditional or data dependent branch can cause difficulties peculiar to this architecture. For example, suppose that one of two operations is to be performed on the data in each ALU and that this choice is made based on whether or not an ALU's accumulator is zero. Those ALU's with a zero accumulator must perform the first operation; those with a non-zero accumulator must perform the second. The SIMD

architecture cannot support independent, concurrent operation of each of the two sets of ALU's. Therefore, the control processor must halt the ALU's that are to perform the second operation and perform the first operation in those remaining. It must then halt the first set of ALU's, enable the previously halted ALU's and perform the second operation. At the conclusion of this process, all ALU's will again be enabled and the program continued. This is illustrated in Fig. I.B.2. As this example shows, the control processor must serially simulate the action of a data dependent branch. It selectively enables only the set of processors on a particular path of the program graph and executes all such paths sequentially.

Analogous to the SIMD system just described is the MISD computer. Consider a typical pipeline processor, a special case of the MISD system. (For example, a single arithmetic processor of the TI-ASC. See Fig. I.B.3.) Within the pipe, several <u>different</u> operations can be in progress simultaneously. However, the system only has one source of operands. Each stage of the pipe gets its data only from the preceding stage.

T. C. Chen[3 , 51] describes pipelining and synchro-parallelism (SIMD as described above for example) as typifying multiprocessing by tight coupling. He defines a repetition ratio, ρ, for a multiprocessor in terms of the two dimensional equipment-time space,

$$\rho = \frac{1}{\frac{\tau_1}{W\tau_2} + 1} \quad ,$$

where τ_1 is the time during which only one processor is required (for overhead, job setup, etc) and τ_2 is the time during which W processors are required. If N is the

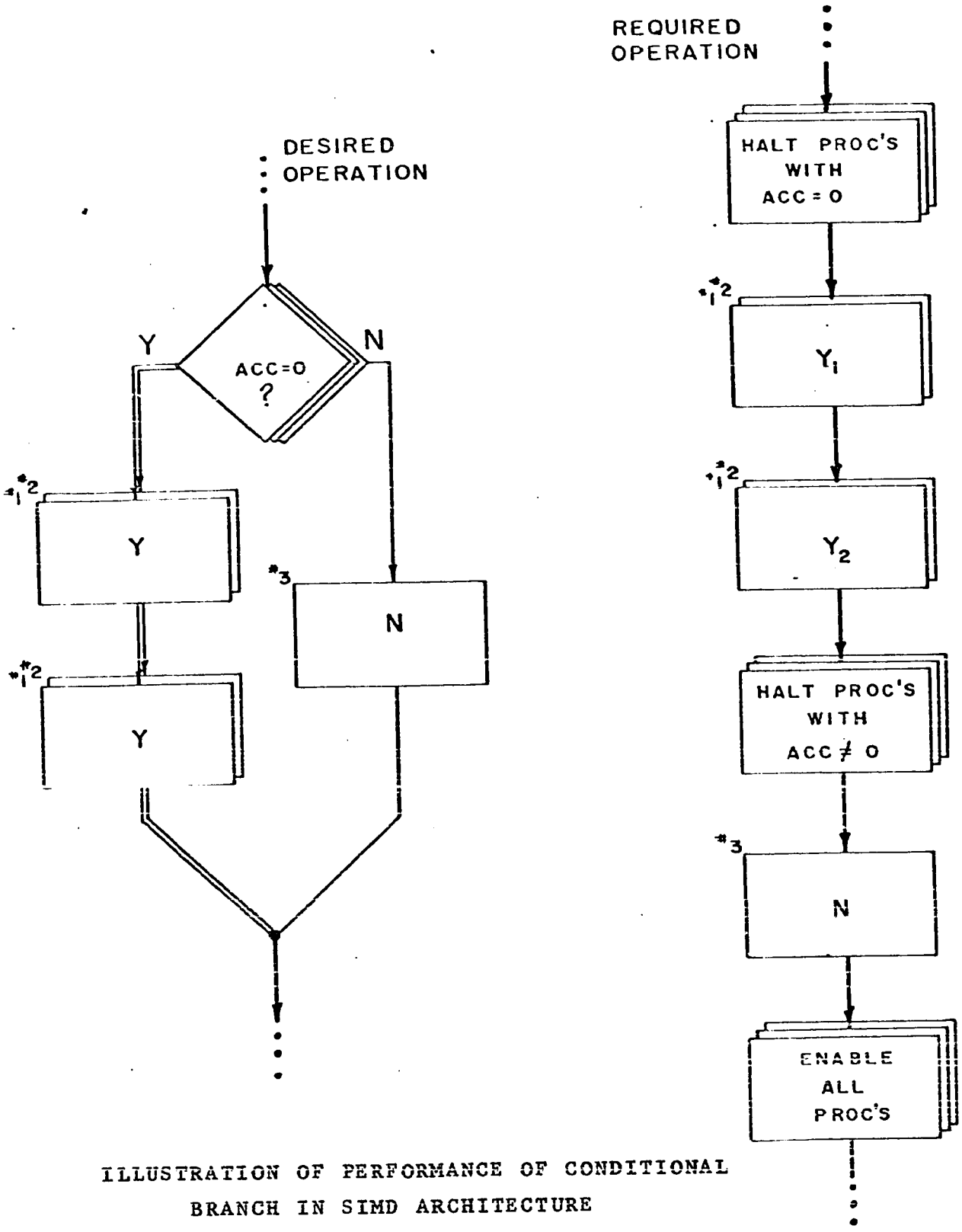
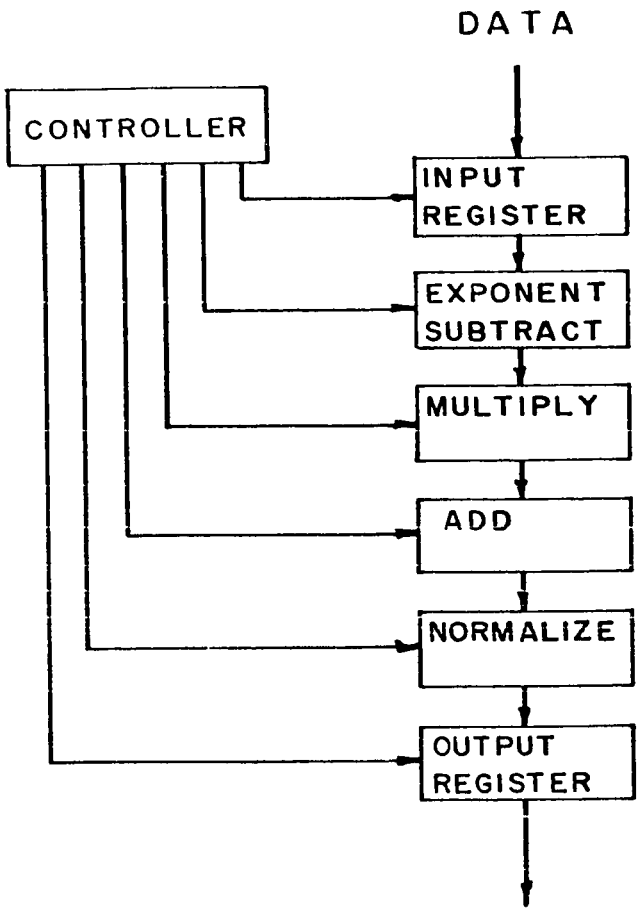


ILLUSTRATION OF PERFORMANCE OF CONDITIONAL BRANCH IN SIMD ARCHITECTURE

Fig. I.B.2



EXAMPLE OF A PIPELINE ARITHMETIC PROCESSOR

Fig. I.B.3

number of processors available in the tightly coupled system, the total efficiency of the synchro-parallel system is

$$\eta = \frac{1}{N}\left[(1 - \rho) + \rho\left\lceil\frac{W}{N}\right\rceil\frac{1}{W}\right]$$

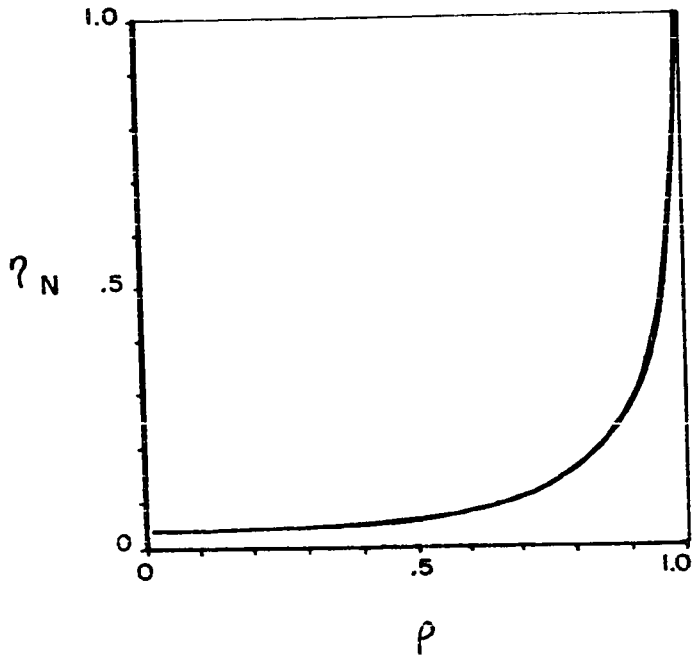
$$\eta = \eta_N \text{ if } W = N$$

$$\eta \leq \eta_N \text{ for } W < N$$

η_N is plotted against ρ for N = 32 in Fig. I.B.4. In this case, it can be seen that the effect on the efficiency of the system is pronounced for very slight deviations of ρ from 1. As Chen points out, the slope of the curve at $\rho = 1$ is (N-1),[3].

Chen also observes that true job parallelism is difficult to find or exploit due to the conditional branches often required (as mentioned in the discussion of SIMD.) Handling conditional branches is difficult and often involves draining pipelines and disabling parallel processors. With the extreme sensitivity of η to small changes in the job parallelism, it is difficult to maintain a high performance level for reasonably parallel jobs not specifically suited to a particular machine configuration. (i.e., 33 x 33 matrix or 31 x 31 matrix operations on a system designed ideally for 32 x 32 matrix operations).

In order to obtain an effective, efficient system architecture, Chen maintains that multiprogramming with the proper overall job mix is required in addition to multiprocessing. The system must actively reassign the priority of jobs to fit the available resources in an attempt toward self-optimization. The parallel system can no longer rely on the synchronization of elements within



EFFICIENCY OF A SYNCHRO-PARALLEL SYSTEM FOR N=W=32

Fig. I.B.4

it. Instead it must become a number of loosely coupled, nearly self-sufficient, processors, each having a large degree of independence and local autonomy.

C. Criteria

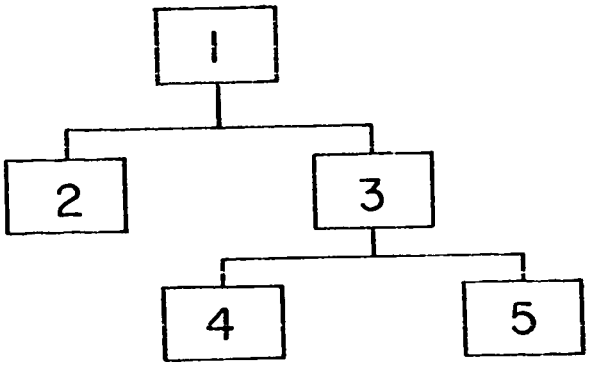
The system architecture under consideration here is intended to provide enhanced speed, flexibility, reliability, etc., while avoiding the difficulties encountered in previously developed architectures. It provides a loose coupling of processors that may cooperate or act independently.

In the attempt to meet these goals, the following criteria have been established for the system architecture:

1. The use of a large number (>100) of processor modules should be possible.
2. The communication/control structure should be uniform throughout the system.
3. Each processor module should be capable of communication with all (or most) other processor modules.
4. The control system should be capable of monitoring and intervening in a process as well as provide for the independent action of a processor. Blocks of processors should be able to function as a team independently of other teams.
5. The control scheme should be simple, uniform, and effective. One such scheme is a hierarchy of control.
6. A dynamic ability to reconfigure the system (i.e., rearrange the hierarchy of control) should be included. This will fit the system to the problem allowing the system to appear as a Von Neuman machine, a parallel array or as an associative parallel processor, etc., as required.
7. In addition, considerations such as reliability, fault tolerance, and graceful degradation demand

the incorporation of redundancy and a capability for dynamic reconfiguration as well as the uniformity of structure previously mentioned. This redundancy can be applied to the parallel or concurrent execution of tasks which could be performed serially as the resources of the system are limited through failures of elements.

Just as structured programming concepts and the hierarchical program development of Dijkstra, [20] provide for simple, effective development of software in a hierarchy of routines, the same basic philosophy, when applied to hardware, should result in an organization effective for the problem at hand. In addition, industry, business, and military organizations have demonstrated the advantage, flexibility, and efficiency of such control schemes. A hierarchical organization will provide the necessary control structure to satisfy the considerations above without relying on a "super" central controller (i.e., it distributes the control allowing each controlling element to be simpler, cheaper, slower, etc., while still producing an effective operation). Although overhead may be increased, this will hopefully be offset by the advantages of a simple, cheap control for a large number of modules. (See Fig. I.C.1).



1 DIRECTLY CONTROLS 2,3

3 DIRECTLY CONTROLS 4,5

ILLUSTRATION OF A HIERARCHY OF CONTROL

Fig. I.C.1

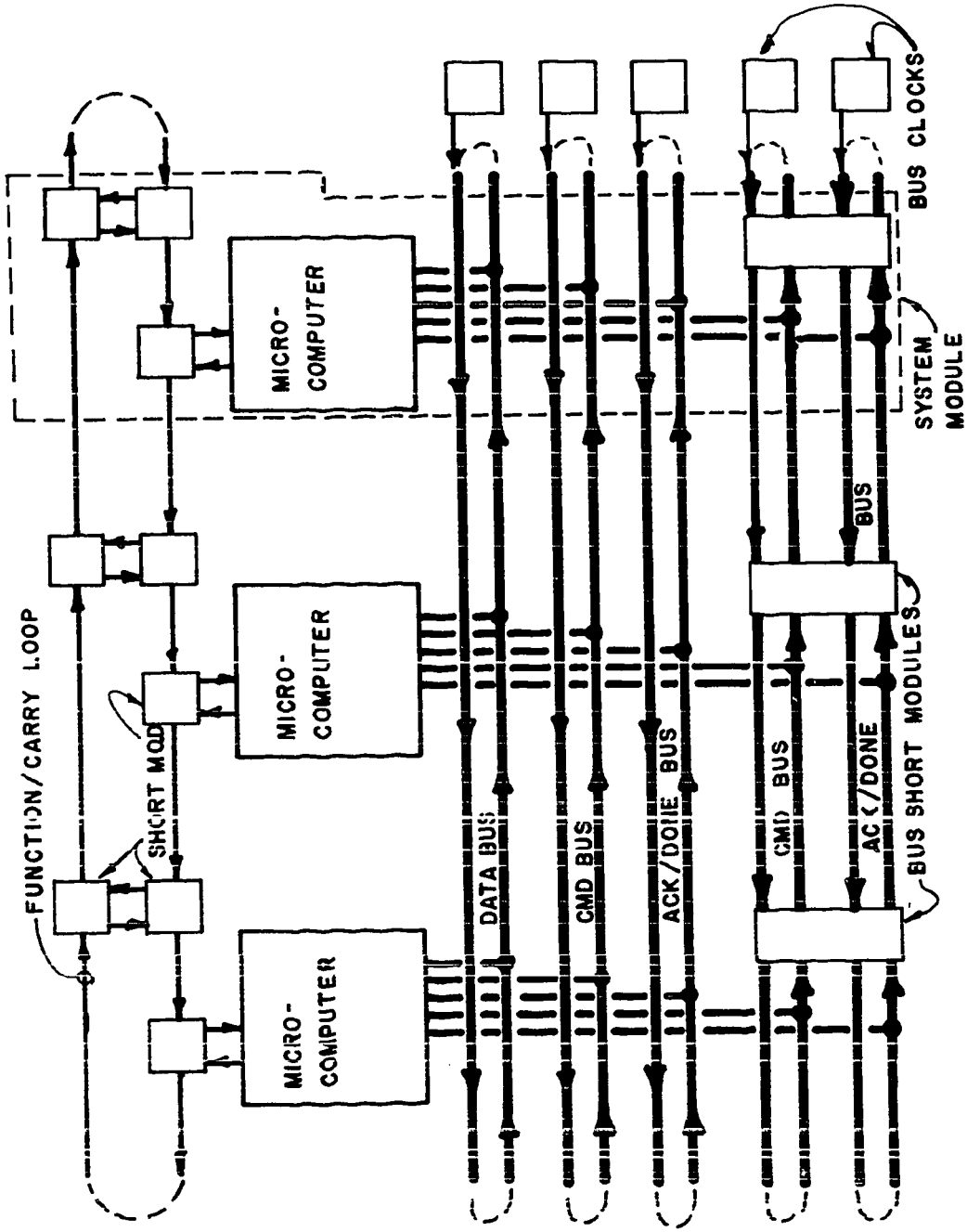
II. DESCRIPTION OF BASIC SYSTEM

A. General Overview

As can be seen in Fig. II.A.1, the system configuration consists of a number of system modules containing microcomputers and ancillary circuits connected by a series of busses, loops, LOOP SHORT and BUS SHORT modules. All interprocessor communication takes place along the various busses and loops. (Previous multiprocessors have employed star networks of I/O channels, multiport memories or crossbar switching matrices, etc., to provide interprocessor communication. These techniques suffer the deficiencies of cost, complexity, etc., that rise linearly as the number of processors grows. In addition they tend to have a fixed maximum practical size. The cost/complexity of a bus tends to grow linearly with $\log_2$ of the number of processors, i.e. only an additional bus address bit is required every time the number of processors doubles.)

Each processor has its own independent memory and generally is capable of performing any of the system tasks assuming it has been suitable programmed. Along with the various elements of hardware in the system, a basic system philosophy and set of protocols is also required. It is intended that this system implement a hierarchical, restructurable organization. As such, there will generally be one processor (any processor) responsible for overall system action. This processor designates subordinates, establishes the chain of command and directs its immediate subordinates in the tasks they are to perform.

In order to implement this philosophy, the following basic characteristics/protocols will be incorporated



OVERALL SYSTEM CONFIGURATION BLOCK DIAGRAM

Fig. II.A.1

into the design:

1. Each module will be named, both with a unique, permanent, physical name (P-name) and with a variable, logical or symbolic name (V-name). Each V-name consists of two parts, a block name and an element name. In addition there is a "universal" name to which all modules respond. This universal name may be employed as either the block name or the element name or both. In this discussion this universal name will be represented as "XX" All communication is carried out by tagging or addressing information packets with their destination name and placing them on a bus. The bus then carries the packets to the receiving module. Data or commands may be passed to a group of modules by specifying only the block name and "XX" for the element name. Likewise, information can be passed to all modules simultaneously by specifying "XX,XX" as the V-name.
2. As is common with communication links between asynchronous systems, all commands sent by a master or controlling module must be received by its subordinate and acknowledged. The subordinate queues the commands pending the arrival of the appropriate operands. (See 5 below).
3. Task completion must be signaled.
4. Several adjacent processors may be strung together to form a wider arithmetic ability than would otherwise be available. (This facility is provided through the FUNCTION/CARRY LOOP).
5. All communication throughout the system will consist of information packets containing the data to be transferred and a series of tags. Since each processor is identified by a name, all ambiguities

associated with the transfer of information are resolved through the use of the processor names. Every packet of information placed on a bus contains the destination address or name (either the permanent modules number or its V-name). In addition to the destination, each packet will contain tags uniquely associating the operands with the commands stored in a queue or other temporary storage medium. For data packets, a 1 bit tag will also indicate the order of the operands for non-commutative operations.

B. Description of System Elements

The heart of each system module is the microcomputer itself. Each microcomputer, the microprocessor with its memory, will be microprogrammed to provide all the basic functions of a standard processor and respond appropriately to the actions of the system. It should perform overhead type tasks as automatically as is practical for a given microprocessor. For example, the processor could be microprogrammed to automatically remove items from the busses and place them in temporary storage. It could also provide for automatic maintenance of a queue of commands. Additionally, the microcomputer would be programmed by the user in a more conventional manner. Generally, the programs would consist of a series of subroutines whose call would be initiated by commands received from more superior elements of the hierarchy.

Each processor must have a priority interrupt capability that masks interrupts occurring below the processor's priority level. It must also have lines for the "carry out" generated by an arithmetic operation or left shift. Likewise, it should also have a "carry in" capability. (Several currently available processors organized on a bit slice basis provide these features [Rattner,38]). There is no requirement as to word length, speed, etc., for the processor imposed by the network architecture. (Obviously, one would want to use the best available processors consistent with these constraints, economics, etc.).

Communication between processors is provided by a system of circulating busses. The circulating bus or C-Bus moves a packet of data in a fixed direction a uniform distance in each unit of time. Thus, the C-Bus functions

much like a shift register and could be considered to be a Pierce Loop [34, 23]. Any processor can transmit by placing an information packet on the bus anytime a gap in the circulating traffic appears at its location. By application of the processors interrupt facility, all users will continually monitor the traffic passing their locations. When a processor recognizes that a packet passing its location contains its address (or name), the processor removes the packet from the bus. The packet's former position in the traffic stream is now a gap, free to be filled with a new packet by any user. The C-Bus thus provides temporary storage of information and is a means by which several independent data transfers can be carried out simultaneously [Pierce, 34].

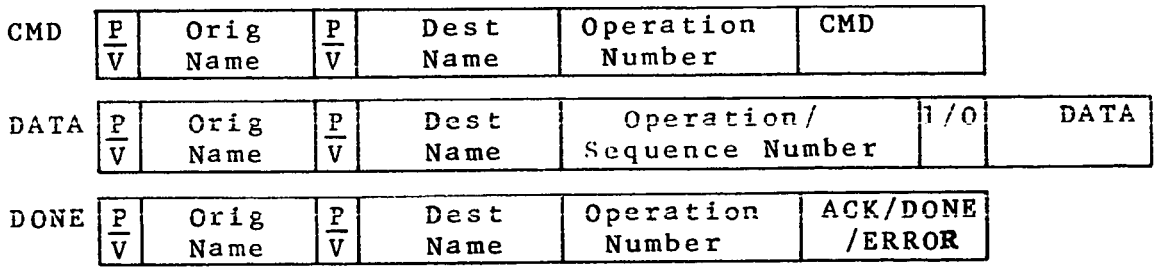
Data transfers are generally carried out on the DATA C-Bus. A data item is placed on the DATA BUS in the form of an information packet containing the data and the destination processor name. As the packet circulates around the bus, the destination name is compared to the name of each processor. When a match occurs between the name on a data item and a processor, that processor is signaled and the data item is removed from the bus.

In order to control the system efficiently, busses providing command and control capabilities have been grouped together into several sets. Each set of busses will collectively be termed a control group (C.G.) Each control group competes for attention from each processor on a priority basis much as in the case of a priority interrupt system. The master control group (M.C.G.) is the highest, most significant priority or 0^{th} level (C.G.[0]). Each additional control group is on level 1,2, etc. Each control group other than the master can be blocked/shorted at the left edge of any processor by activation of the BUS SHORT module. This means that

each circulating bus is "shorted" or the loop is closed (see Fig. II.B.1) dividing the bus into several independent sections. Each control group consists of a command (CMD) bus, and an acknowledge/done (ACK/DONE) bus. The CMD BUS carries commands to the processors. Acknowledgment of receipt and acceptance of the command is returned to the originator on the ACK/DONE bus as well as notification of task completion. When a processor name matches the name attached to a command on a CMD BUS at level "n", an interrupt to the processor is generated on interrupt priority "n". If the processor priority value is set higher than or equal to "n", the interrupt is accepted and the command is recognized as destined for this processor. A processor recognizing a command is obligated to reply on the ACK/DONE BUS with an ACK or positive acknowledge if the command can be accepted into the processors command queue. Otherwise, the processor replies with a negative acknowledge or NAK.

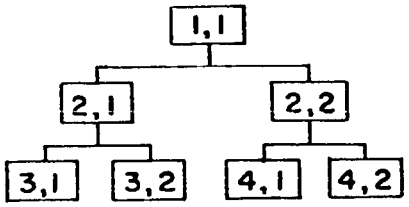
Basic bus formats for information packets with an explanation of the various fields are given below:

Bus Formats

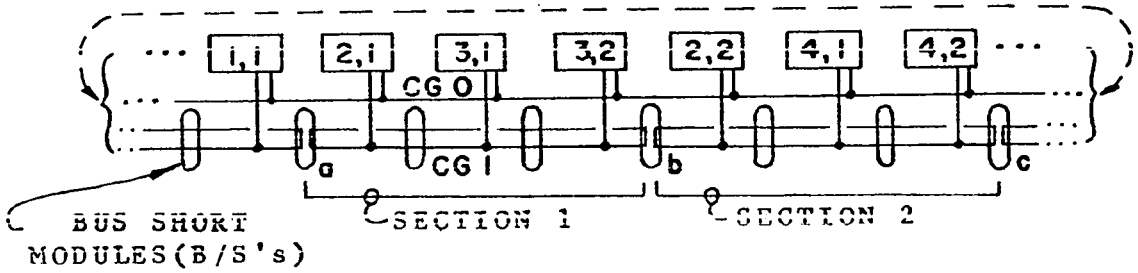


Explanation of Fields

P/V -- 1 bit that indicates that the contents of the name field are to be interpreted as a module's permanent name (P), or its V-name (V).



1,1 CONTROLS 2,1 AND 2,2 ON CONTROL GROUP 0
(MASTER C.G.)
2,1 CONTROLS 3,1 AND 3,2 ON C. G. 1 (SECTION 1)
2,2 CONTROLS 4,1 AND 4,2 ON C. G. 1 (SECTION 2)



1,1 CAUSES B/S a TO BE ACTIVE
3,2 CAUSES B/S b TO BE ACTIVE
4,2 CAUSES B/S c TO BE ACTIVE

ILLUSTRATION OF A CONTROL HIERARCHY ESTABLISHED BY ACTIVATING BUS SHORT MODULES

Fig. II.B.1

Name -- the name of a processor. When interpreted as a V-name, it consists of 2 parts, the block and the element name.

Operation # -- Each command sent to a module is numbered and held in memory in numerical order by the receiving processor until its operands are present and there are no commands having operands present and a lower number in memory. The operands are uniquely identified as belonging with a particular command by a matching Operation #.

Sequence # -- This is a re-interpretation of the Operation # field on the data bus to allow a block of data to be transmitted between processors and ordered upon receipt. This requires, however, that the receiving processors have no commands pending or in progress (except the command to accept this block of data).

I/O -- In the DATA BUS format, this indicates the order of the two operands for non-commutative operations.

CMD -- The command code indicating the operation to be performed.

DATA -- The actual operands, etc., transmitted on the DATA BUS.

ACK -- On the ACK/DONE bus, the two ACK codes indicate the positive acknowledgment (A) of the receipt and acceptance of a command or a negative acknowledgment (N) indicating that the named module is unable to accept or perform the required operation.

DONE/ERROR -- On the ACK/DONE bus, this indicates that the operation whose number is indicated in the operation number field has terminated either successfully or with errors as indicated by the error code.

The FUNCTION/CARRY LOOP also transfers data throughout the system and is designed to transfer information shifted or "carried out" from the arithmetic section of one processor to the arithmetic section of another processor. This allows several processors to function as a single multiprecision arithmetic unit. The FUNCTION/CARRY LOOP passes through each processor module and has no storage of information (i.e., does not shift packets as a C-Bus does). By activation of the appropriate LOOP SHORT modules, the FUNCTION/CARRY LOOP may be gated through the processor proper or past it. In a similar manner, it may also be shorted at the left edge of each processor module (i.e., it may be broken into 2 closed loops at the left end of the module). (See Fig. II.B.2).

In addition to the various busses, the items mentioned previously as BUS SHORT modules and LOOP SHORT modules perform an important function in the implementation of a hierarchical structure. The BUS SHORT modules are a part of every Control Group except the Master Control Group. Their function is to divide a Control Group into independent sections. This allows several teams of modules to operate independently on the same Control Group. Each BUS SHORT module is controlled by the processor to its immediate left. As an example, see Fig. II.B.3.

C. G. [1] is broken into two independent parts with each section functioning just as if it were a complete C.G. Processor ⟨2,1⟩ can then control ⟨3,1⟩ and ⟨3,2⟩ without any interaction with other processors on C.G. [1].

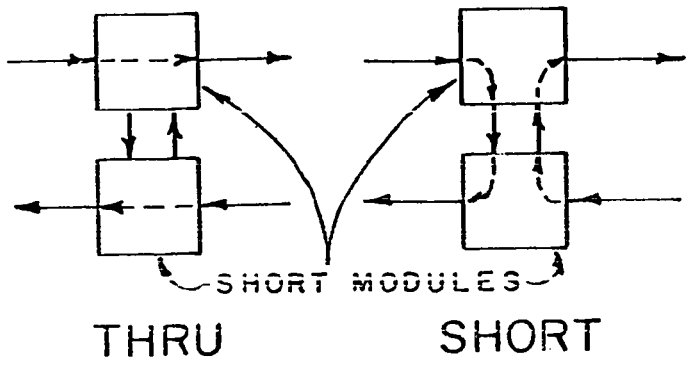
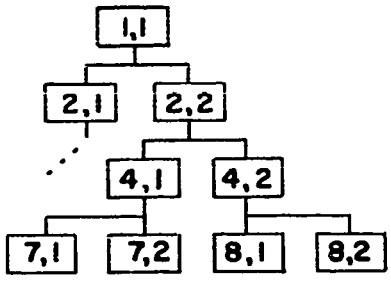


ILLUSTRATION OF THE ACTION OF SHORT MODULES

Fig. II.B.2

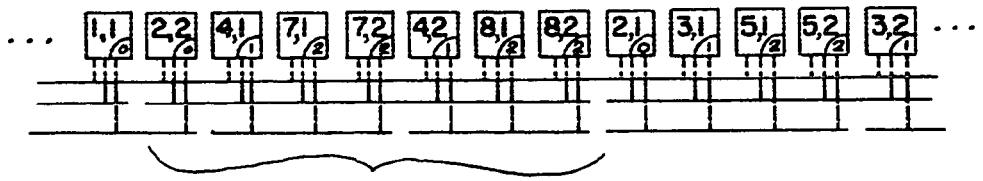


1,1 IS MASTER OF ALL

2,2 IS MASTER OF 4,1 4,2
7,1 7,2 8,1 8,2

4,2 IS MASTER OF 8,1 8,2

ETC.



BETWEEN THESE LIMITS, C.G. 1
HAS EXACTLY THE SAME CAPABILITY
AS C.G. 0 ,ETC.

ILLUSTRATION OF HIERARCHY ESTABLISHED
THRU USE OF SHORT BUS MODULES

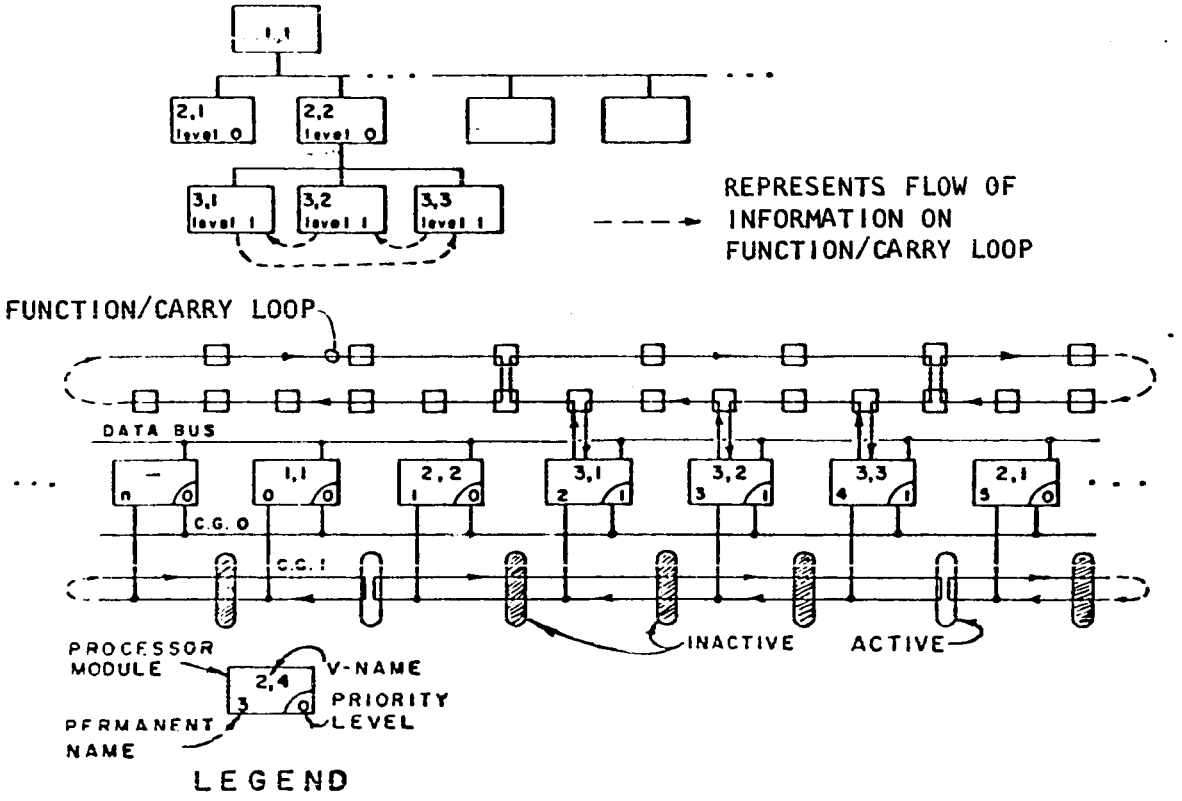
Fig. II.B.3

C. Basic Illustration

The system, when viewed in an unstructured, idle configuration, will appear as a collection of processors arranged in a cylindric fashion connected by a collection of busses. However, this structure, when viewed in an active state, will generally be divided into a collection of processor teams in a hierarchy of responsibility and control. Structuring takes place in the following fashion:

1. Initially, the user will designate a processor as the master and load its memory with the appropriate programs. This processor then begins execution.
2. The master would decide which of the various processors will perform particular tasks (e.g. processors ⟨3⟩ and ⟨10⟩ could be assigned to perform an internal double precision ADD on receipt of a command code of 0001 and a single precision ADD when commanded with a code of 0010).
3. The master commands each processor in turn to load the program being sent to it over the DATA BUS.
4. Upon a command of the master, each processor sets its V-name and priority to the values sent it on the DATA BUS.
5. The appropriate modules are then commanded to activate their BUS SHORT or LOOP SHORT modules, as required.

For example, the hierarchy shown in Fig. II.C.1 may be defined in the system by activating the appropriate BUS SHORT modules, naming the processors appropriately and specifying their priorities (or the level on which the module expects commands). The 0^{th} module has been established with the V-name of ⟨2,1⟩ and designated as



OPERATION OF SYSTEM WITH THREE LEVELS OF CONTROL AND A 3n PRECISION ARITHMETIC SECTION

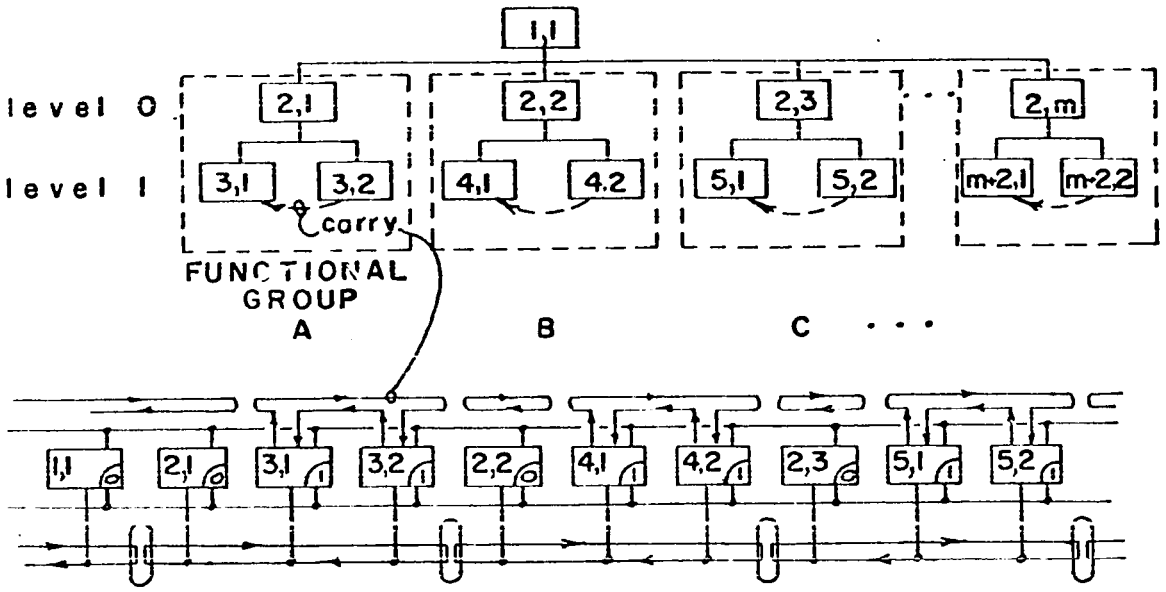
Fig. II.C.1

the most superior element in this structure. Modules 1 and 5, assigned V-names of ⟨2,2⟩ and ⟨2,1⟩, respectively, are both directly controlled by ⟨1,1⟩ and expect commands at the 0^{th} priority level (i.e., from the master control group).

Module 1 (named ⟨2,2⟩) controls directly the three modules, 2, 3, and 4 (named ⟨3,1⟩, ⟨3,2⟩, ⟨3,3⟩ respectively) through commands on the control group at the 1^{st} priority level. Note that since the BUS SHORT modules between modules 0 and 1 and between modules 4 and 5 have been activated, this group of processors is capable of completely independent action without interaction with other modules on the Control Group at level 1. Assuming that the appropriate control modues in the FUNCTION/CARRY LOOP have been activated as shown, the modules named ⟨3,XX⟩ could be considered to be an arithmetic functional unit of 3·n precision where n is the word size of a given module. Module ⟨2,2⟩ would be the controller for this arithmetic section.

As another example, consider a parallel array processor. This configuration, using an arithmetic capability of 2·n bits, would appear as in Fig. II.C.2.

Again each level in the hierarchy is controlled on a different level control group. Module ⟨1,1⟩ is the system controller and actually contains the program to be executed. Each of the modules ⟨3,1⟩ through ⟨M+2,2⟩ contains the appropriate data elements as in any parallel array processor. Module ⟨1,1⟩ would control each of the functional groups A, B, ... by placing a command with the appropriate destination name on the Master Control Group CMD BUS for the specific controlling module desired. On the other hand, ⟨1,1⟩ could control all the functional groups simultaneously with one command addressed to ⟨2,XX⟩. Thus, as in the case of a parallel array pro-



STRUCTURE REPRESENTING A DOUBLE PRECISION PARALLEL ARRAY PROCESSOR

Fig. II.C.2

cessor, a single ADD, MULTIPLY, etc., command would cause all m functional groups to perform the required operation on the appropriate operands in each of their independent memories.

In the case that restructuring is required (due to problem changes or hardware failures), the master need only cause the system to pause while it proceeds through the structuring phase again etc. (The master can interrupt any processor by commands on C.G. [0]. Since C.G. [0] has no BUS SHORT modules, it can never be partitioned and is, therefore, always available for communication with any module. Since it acts at the highest priority level, it can never be masked.

Although the preceding discussion and examples have only two C.G.'s and result in three levels of hierarchy, there could be several more C.G.'s. This would allow several more levels of hierarchy and, at each level, each processor would appear exactly like a master to all those processors subordinate to it.

In each of these examples several considerations should be pointed out:

1. All data transfers take place on the DATA BUS. This bus will, therefore, be a bottleneck and its performance will seriously affect the total system throughout. The DATA BUS must, therefore, be a high speed bus.
2. In order that a group of m processors be connected to form an m.n bit arithmetic section, they must be adjacent or broken only by modules operating independently of the FUNCTION/CARRY LOOP.
3. Although the master controller usually would communicate only with the modules one level below it in the hierarchy, it can send commands to any module through master control group which is never parti-

tioned and cannot be masked. It, therefore, can begin corrective action by reassigning names, etc., should a fault occur.

4. The master of any set of processors is required to keep track off the status and utilization of those processors assigned it.

5. Although this system is dynamically reconfigurable, this feature has limitations. Due to the fragmentation of the busses by the BUS SHORT modules, it is possible for a processor to be cut off and rendered unavailable for application in a group. As a result, restructuring for the purpose of garbage collection may be periodically required. Groups of modules are best formed from modules adjacent to one another. To facilitate restructuring and reduce the requirement for garbage collection, independent groups of modules should be spread as widely as possible.

III. ANALYSIS

A. Introduction and Model

This section will attempt to analyze some of the performance characteristics of a collection of processors organized as described previously. In particular, the interface between a processor and each of the various circulating busses will be studied in detail.

Throughout this discussion all items will be described from the point of view of the bus-processor interface. For example, a processor will be described as having an idle and an active period. For the purposes of this analysis, the active period is that time during which the processor is continually attempting to place messages on the circulating bus. The idle period is that time during which the processor is doing anything else, calculating or awaiting tasks, etc.

The following analyses will be based on models of the processor action and interface having no buffer register or having a single stage of buffering. The case of a multistage queue in the interface is generally felt to be unnecessary but can be found in reference [25]. As shown in Fig. III.A.1, we will be concerned only with one processor and one bus. In each analysis, the bus will be assumed to have traffic distributed on it in a manner such that the arrival of empty slots at the processor of interest obeys a Poisson probability law. The rate at which slots (full or empty) pass a processor is v slots/sec. For a loop containing N processors, a slot will completely circumnavigate the loop in N/v sec.

The first scheme to be studied is illustrated in Fig.'s III.A.2, III.A.3, III.A.4 and III.A.5. As indicated, there is no intermediate buffering, queue, etc.

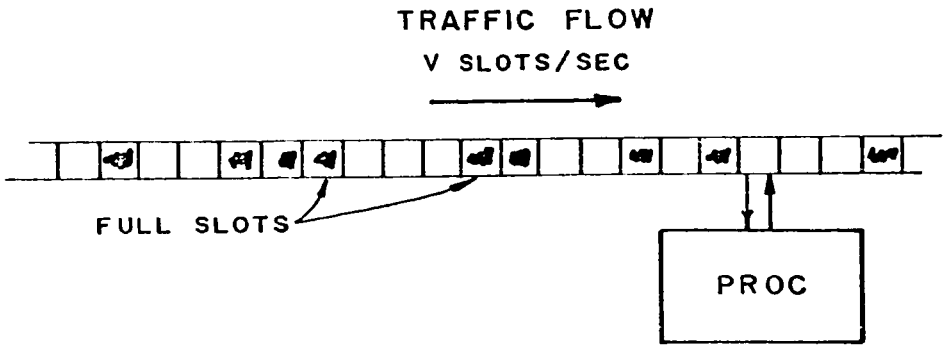


ILLUSTRATION OF PROCESSOR AND BUS

Fig. III.A.1

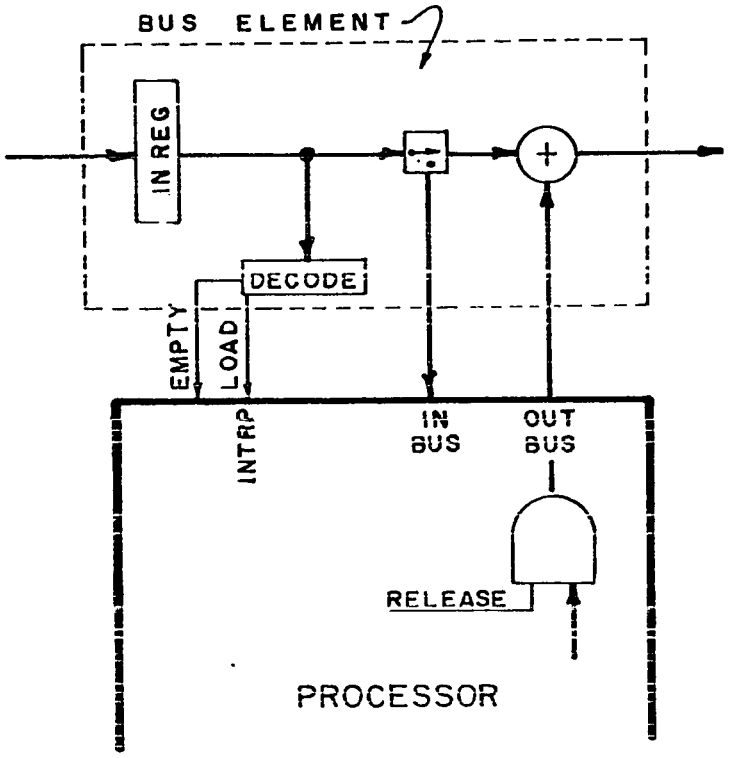


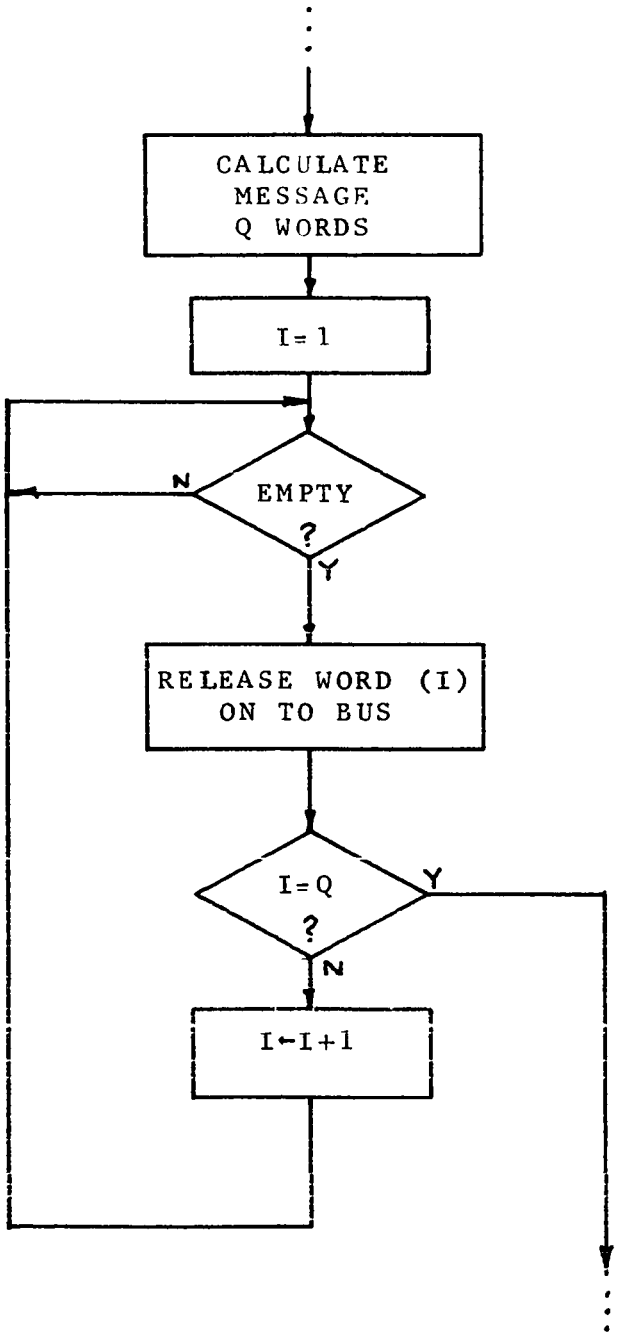
ILLUSTRATION OF BASIC BUS INTERFACE

Fig. III.A.2

```
        .
        .
        .
Calculate message

  I ← 1 ;
  while (message word ≠ end of message) do

        OUTBUFFER ← MESSAGE WORD(I);

        while (BUS_IS_FULL) do

              WAIT ;

        RELEASE OUTBUFFER; ;

        I ← I + 1 ;

  continue

        .
        .
        .
```

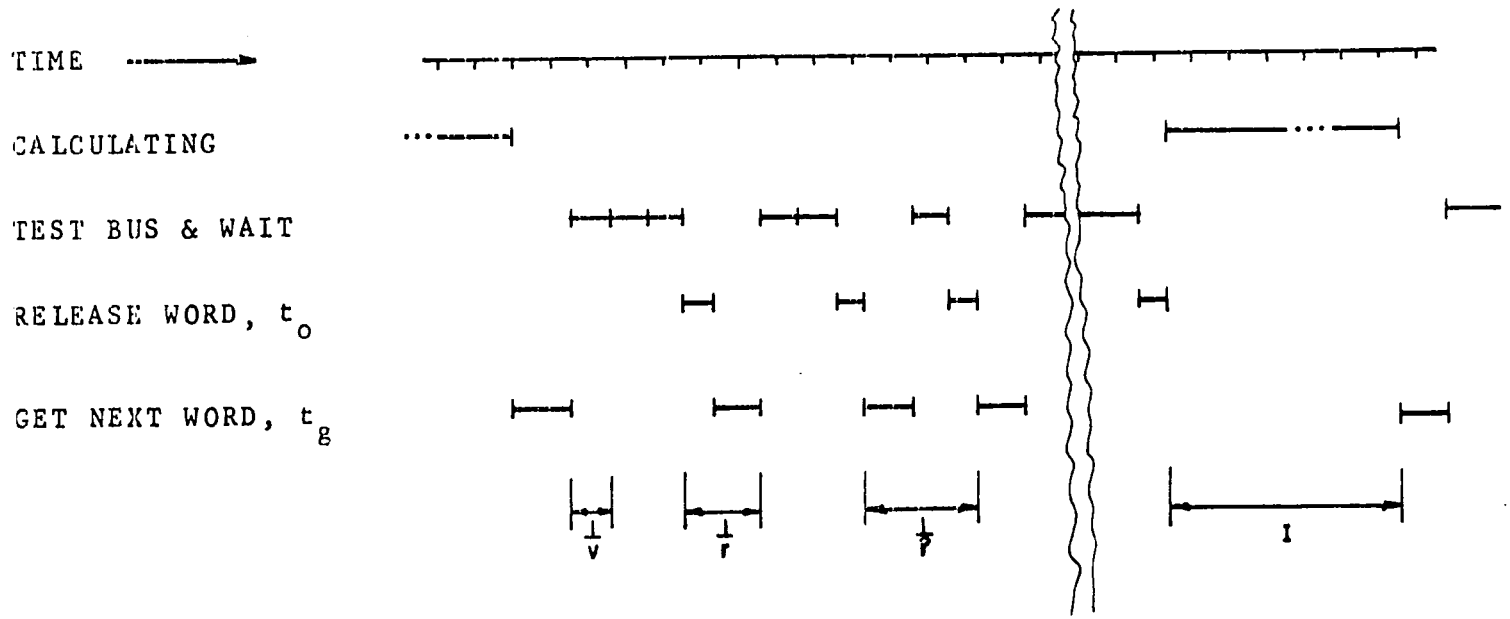
ILLUSTRATION OF PROCEDURE FOR PLACING A MULTIWORD MESSAGE ON A BUS

Fig. III.A.3



FLOW CHART FOR FIG. III.A.3

Fig. III.A.4



TIMING DIAGRAM FOR NON-BUFFERED INTERFACE

Fig. III.A.5

between the processor and the bus. Every output from the processor to the bus necessitates the processor's active monitoring of the bus for an empty slot. Upon detecting an empty slot, the processor must then actively place its message word on the bus.

It should be noted immediately that this scheme will work only in the case that the bus is moving slowly enough that the processor has sufficient time to place the message word into the bus's output register. The time for the processor to recognize that a slot is empty is not of consequence. Should this be a relatively long period, an appropriate amount of "look-ahead" along the bus may be employed. (See Fig. III.A.6).

The basic notation to be used in the following discussions is indicated below with a brief definition of each symbol.

Definitions:

U_i : utilization of the bus at processor i

v : rate at which slots pass a processor

μ_i : rate at which occupied slots pass processor i

$\bar{\mu}_i$: rate at which empty slots pass processor i

r_i : rate at which processor i produces message words and places them on a bus having no traffic.

Q_i : average number of words/message from i^{th} processor

I_i : average idle period of i^{th} processor

θ_i : utilization of processor i

$\hat{r}_i$: actual rate of production of message words (including waiting time) during the active period

N : number of processors in the system on the bus

γ_i : a ratio of processors rate of production of words to bus rate.

$$\gamma_i = \frac{r_i}{v}$$

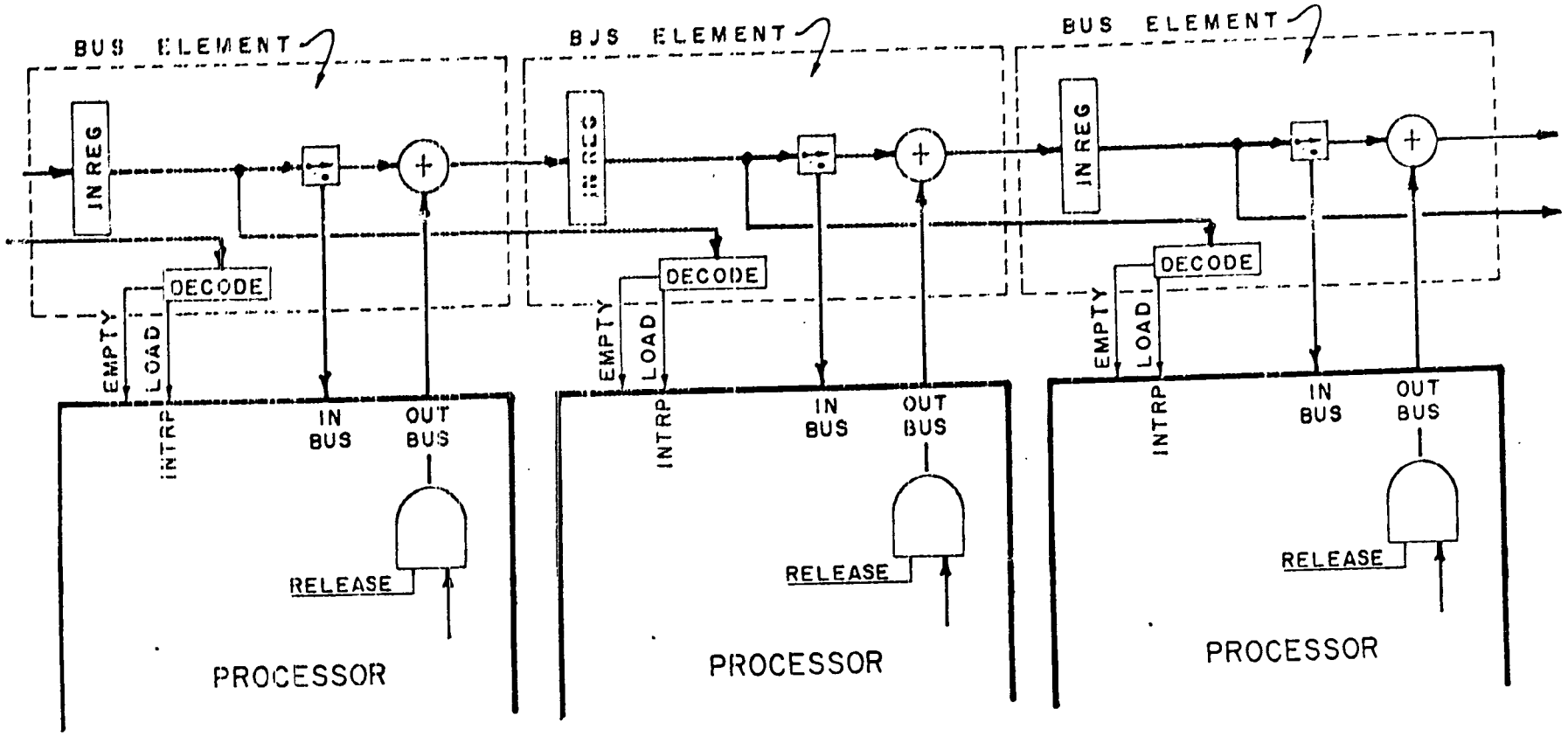


ILLUSTRATION OF 1/V SEC'S "LOOK-HEAD"

Fig. III.A.6

t_{o_i} : time required to output one word

t_{g_i} : time required to obtain one word (ie. retrieve it from memory)

w_i : average time per word the processor waits

w_i' : average wait time normalized to the processor rate

w_i'' : average wait time normalized to the bus rate

$\hat{W}_i$: average wait time corrected to include the slot at the processor, ie. to begin counting with the 0^{th} slot.

$\overline{w}_i$: average time per word the output buffer register waits

m : ratio of retrieval time to the total time required to retrieve and output one message word.

B. Calculation of Waiting Times (no buffering)

The utilization of the bus, U_i, is the fraction of the bus that processor i observes to be occupied. Therefore,

$$U_i = \frac{\text{\# of full slots through i}}{\text{Total \# of slots through i}}$$

Since slots pass processor i, at a rate of v slots/sec., the rate at which full slots leave it must be

$$\mu_i = U_i^* v \qquad (1)$$

(an * on a bus parameter for a processor i indicates that traffic for i has been removed before the parameter is measured). On the other hand, empty slots leave at a rate of

$$\bar{\mu}_i = (1 - U_i^*) v \qquad (2)$$

A processor requires a minimum of $\frac{1}{r_i}$ seconds to generate each word of a message and place the word on the bus, ie., it produces words at a rate of r_i words/sec. Thus, if the average message length is Q_i words, then the average active time of a processor (as viewed by the bus) would be $\frac{Q_i}{r_i}$ seconds. This rate, r_i, is the rate at which words could be produced and placed on the bus, assuming that the bus were completely clear. This, however, will not be the case. The assumed protocol for a processor requires that it produce a word, then place it successfully on the bus, into a buffer register or a bus queue, etc., before producing another word. Hence, if $\frac{1}{r_i}$, is the time required to produce a word and w_i is the average waiting

time required before an item can be placed on the bus, the actual rate of production is

$$\hat{r}_i = \frac{1}{\frac{1}{r_i} + w_i} = \frac{r_i}{1 + r_i w_i} \qquad (3)$$

and the length of the active period is now $\frac{Q_i}{\hat{r}_i}$.

It is assumed that the processors operate in a "burst" mode with relation to the communication structure. The processor will appear idle (actually idle or calculating data for a subsequent message but not attempting to use the bus, etc.) followed by a period in which it is continually trying to access the bus to pass one or more messages. This is a good assumption inasmuch as there is no concrete definition of a message or specification of Q_i and the idle period length is probabilistic in nature.) It is also assumed that the duration of the idle period for the i^{th} processor is statistically independent of the processor's busy period and is an exponentially distributed random variable having a mean, I_i. Since the production of message words is not constant, the overall average rate must be reduced accordingly. The utilization of a processor can be defined as the fraction of time that it is active or

$$\theta_i = \frac{\text{active period}}{\text{total time}} = \frac{Q_i}{\hat{r}_i} \Big/ \left(\frac{Q_i}{\hat{r}_i} + I_i\right)$$

$$= \frac{Q_i}{Q_i + I_i \hat{r}_i} \qquad (4)$$

The result is that the overall average rate is

$$\hat{\hat{r}}_i = \hat{r}_i \rho_i = \frac{\hat{r}_i Q_i}{Q_i + I_i \hat{r}_i} \qquad (5)$$

Let us now concern ourselves with the bus itself. It is obvious that the volume of traffic on the bus is a factor in all our calculations.

Following the work of Hayes and Sherman [25, p. 2955] let P_{ij} be that portion of traffic on the bus destined from processor i to processor j. Also, let R^*_{ik} be the average rate at which traffic from processor i passes <u>through</u> processor k.

Then,

$$R^*_{ik} = \begin{cases} \hat{\hat{r}}_i \sum_{j=1}^{i-1} P_{ij} + \hat{\hat{r}}_i \sum_{j=k+1}^{N} P_{ij} & \forall i: 1<i<k \quad k \neq N \\ \hat{\hat{r}}_i \sum_{j=1}^{i-1} P_{ij} & \forall i: 1<i<k \quad = N \\ \hat{\hat{r}}_i \sum_{j=k+1}^{N} P_{ij} & \forall i: i=1, \quad k \neq N \\ \hat{\hat{r}}_i \sum_{j=k+1}^{i-1} P_{ij} & \forall i: k+1<i \leq N \\ 0 & \text{otherwise} \end{cases}$$

The total average traffic rate through k is

$$R^*_k = \sum_{\forall i} R^*_{ik} \qquad (7)$$

The total average traffic rate out of k is

$$R_k = R_k^* + \hat{\hat{r}}_k \tag{8}$$

Therefore,

$$U_k^* = \frac{R_k^*}{v} \tag{9}$$

and

$$\bar{\mu}_k = \left(1 - \frac{R_k^*}{v}\right) v \tag{10}$$

Assuming that empty slots are randomly distributed along the bus such that the arrival times are statistically independent and exponentially distributed, the arrival of an empty slot at processor i will obey a Poisson probability law. Since Q_i words are contained in a message, it is clear that Q_i empty slots must appear at processor i in order to place the entire message on the bus. It can be shown that the waiting time for Q_i Poisson arrivals obeys a gamma probability law with the probability density function (for integral Q_i) [Parzen, 33; Sevaglian, 45]:

$$f_{WT_i}(t) = \frac{\bar{\mu}_1}{(Q_i - 1)!} (\bar{\mu}_i t)^{Q_i - 1} e^{-\bar{\mu}_i t} \qquad t \geq 0 \tag{11}$$

The expected value of the waiting time for a message of Q_i words is well known [33] and is

$$E\left[WT_i\right] = \frac{Q_i}{\bar{\mu}_i} \tag{12}$$

The variance of the wait time is determined to be

$$\mathrm{Var}\left[WT_i\right] = \frac{Q_i}{(\bar{\mu})^2} \tag{13}$$

The average wait time for an individual word of a message is then given by

$$\frac{E\left[WT_i\right]}{Q_i} = \frac{1}{\bar{\mu}_i} \tag{14}$$

Note that this is the same expression that would have resulted for $Q_i = 1$. The gamma distribution reduces to an exponential distribution with parameter $\bar{\mu}_i$ in this case.

This expression also represents the average length of a busy period† on the bus as seen by processor i. The average wait time of one word at processor i is then

$$w_i = \frac{1}{(1-\bar{U}_i^*)\ v}$$

$$\hat{w}_i = w_i - \frac{1}{v} = \frac{1}{(1 - U_i^*)\ v} - \frac{1}{v} \tag{15}$$

(The term, $-\frac{1}{v}$, is included so that the wait time may be determined by starting with the slot present at the processor rather than after the bus has moved one position.) Normalizing this expression with respect to the time between successive bus slots, $\frac{1}{v}$, yields

†a busy period is defined as the time between the occurrence of two empty slots as seen by a particular processor. This can be shown to agree with the calculation of Hayes and Sherman [25] and their verification by computer simulation.

$$w_i'' = \hat{w}_i \, v = \frac{1}{1 - U_i^*} - 1 \qquad (16)$$

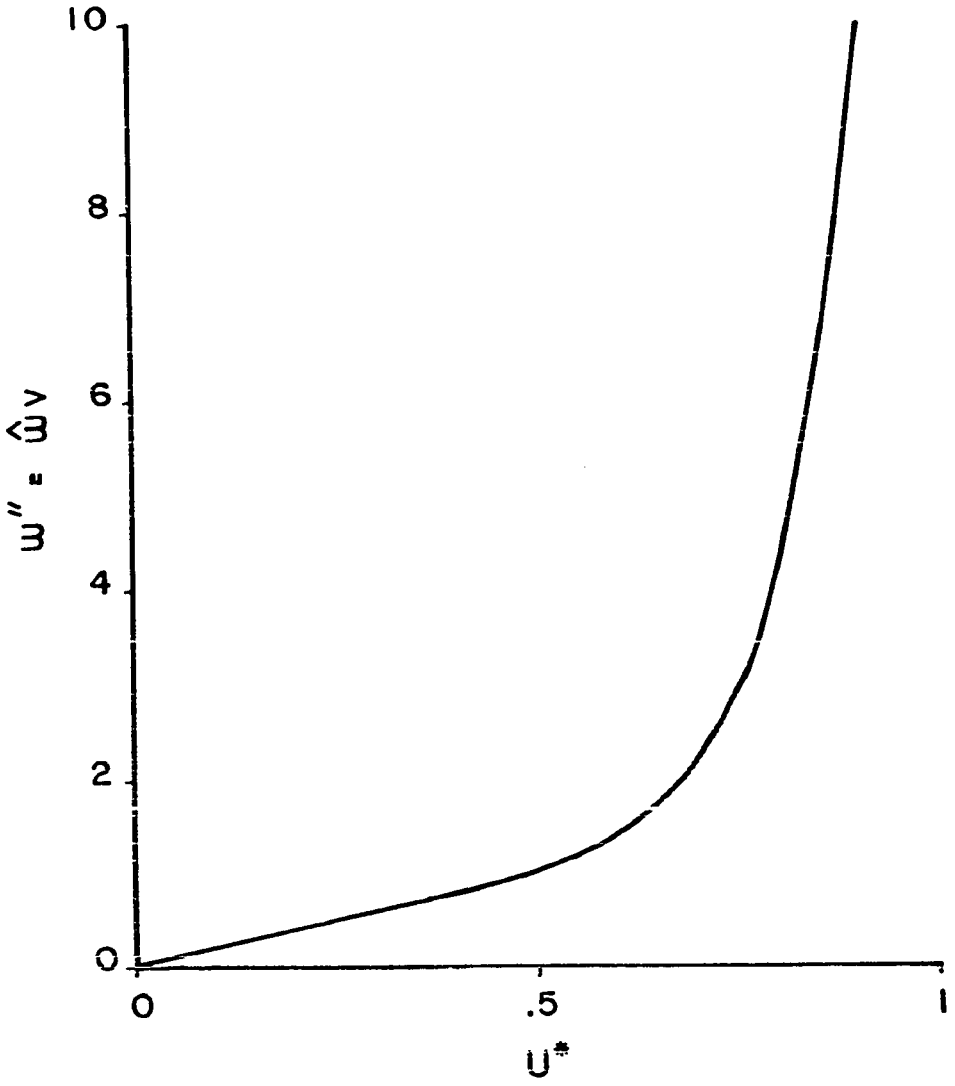
This also represents the wait period in terms of bus slots. Figure III.B.1 is a plot of w_i'' as a function of the bus utilization. It is important to note that the absolute wait time is independent of all system parameters except U^* and v. Thus, for a fixed implementation (ie. having v fixed), U^* completely characterizes the bus interface without any dependence on the distribution of work, relative activity, etc.

Normalizing (15) to the average interword production time for a clear bus, $\frac{1}{r_i}$

$$w_i' = \hat{w}_i r_i = \frac{r_i}{(1 - U_i^*)\, v} - \frac{r_i}{v} = \frac{\gamma_i}{(1 - U_i^*)} - \gamma_i \qquad (17)$$

This is plotted as a function of U_i in Fig. III.B.2. Note that the shape of these curves follow what might be expected intuitively. Equation (17) also forms a basis for comparing the processor speed and communication latency. It indicates the wait period essentially in terms of the memory speed.

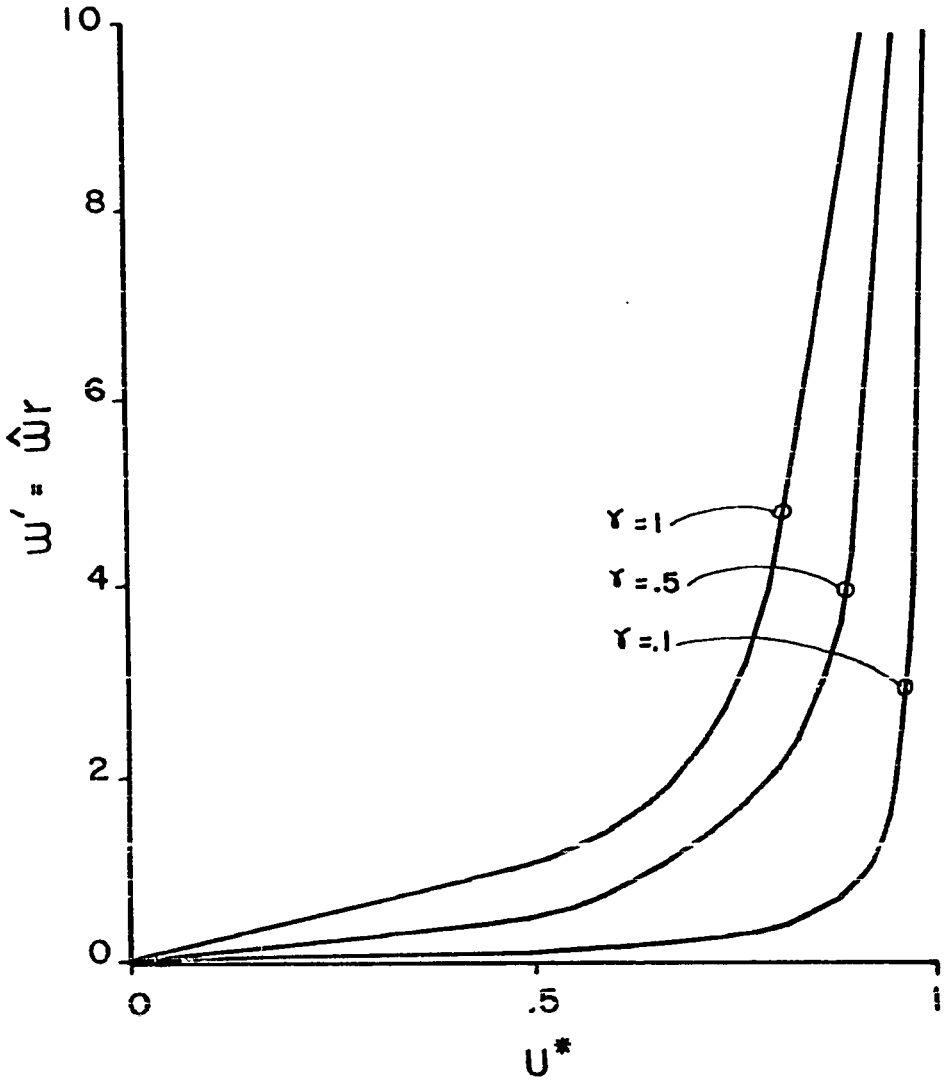
It is insufficient to specify any random variable in terms of its mean alone. It is always important to know how the values of the variable are distributed about the mean, etc. The first moment or variance of the distribution can provide much of this information. The variance per word can be found from (13).



NORMALIZED AVERAGE WAIT TIME/MESSAGE WORD VS.

BUS UTILIZATION

Fig. III.B.1



WAIT TIME VS. BUS UTILIZATION NORMALIZED TO PROCESSOR SPEED

Fig. III.B.2

$$S_i^2 = \frac{\mathrm{Var}\left[WT_i\right]}{Q_i} = \frac{1}{\bar{\mu}_i^2}$$

Using (2)

$$S_i^2 = \frac{1}{(1 - U_i^*)\, v^2} \tag{18}$$

Again normalizing with respect to $\frac{1}{v}$

$$S_i^{2\prime} = \frac{1}{(1 - U_i^*)^2\, v} \tag{19}$$

This has been plotted as a function of U_i^* in Figure III.B.3 for a bus rate of $v = 5X10^6$ slots/sec. As might be expected, S_i^2 becomes very large for large U_i^*.

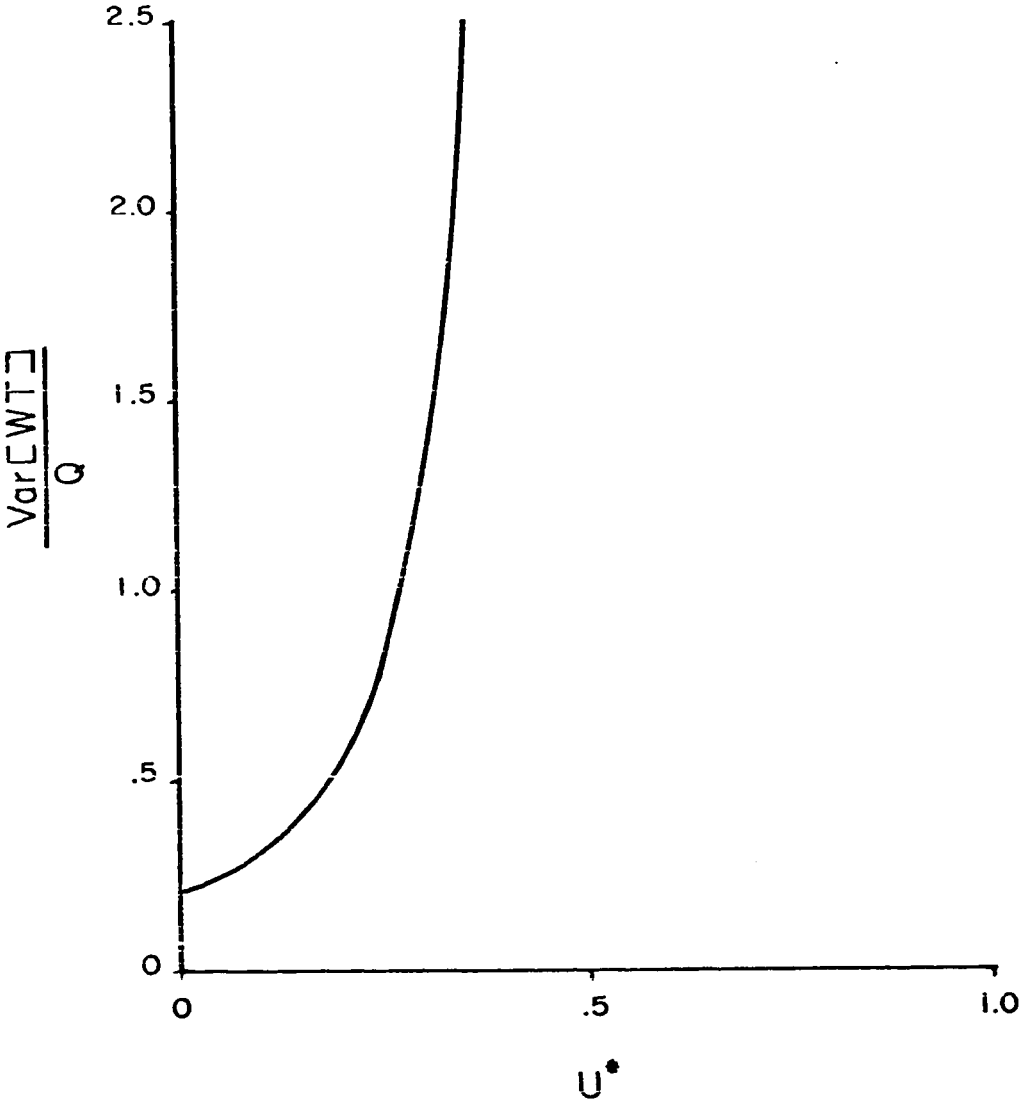
To further characterize the distribution of the wait time per word, we can determine regions having a probability, p, of containing any particular sample of the random variable. According to Parzen [33], for any probability law with finite mean, m, and variance, σ^2, a quantity Q(h) can be defined for $h > 0$ as

$$Q(h) = P\left[x:\ m - h\sigma < x \leq m + h\sigma\right]$$

$$Q(h) = F(m + h\sigma) - F(m - h\sigma) = \int_{m - h\sigma}^{m + h\sigma} f(x)\, dx$$

For the case of an exponential distribution with mean $\frac{1}{\bar{\mu}}$,

$$Q(h) = 1 - e^{-(1 + h)} \quad \text{for } h \geq 1 \tag{20}$$



VARIANCE OF THE WAIT TIME/MESSAGE WORD VS. BUS UTILIZATION, V = 5×10^6 SLOT/SEC

Fig. III.B.3

By Chebyshev's inequality, for any distribution function and $h \geq 0$

$$Q(h) = F(m + h\sigma) - F(m - h\sigma) \geq 1 - \frac{1}{h^2} \qquad (21)$$

This therefore implies that

$$Q(h) = P\left[\, |x-m| \leq h\sigma \,\right] \geq 1 - \frac{1}{h^2} \qquad (22)$$

Chebyshev's inequality then states that the probability that an observed value, X, will lie within $h\sigma$ of the mean is Q(h). For the exponential case,

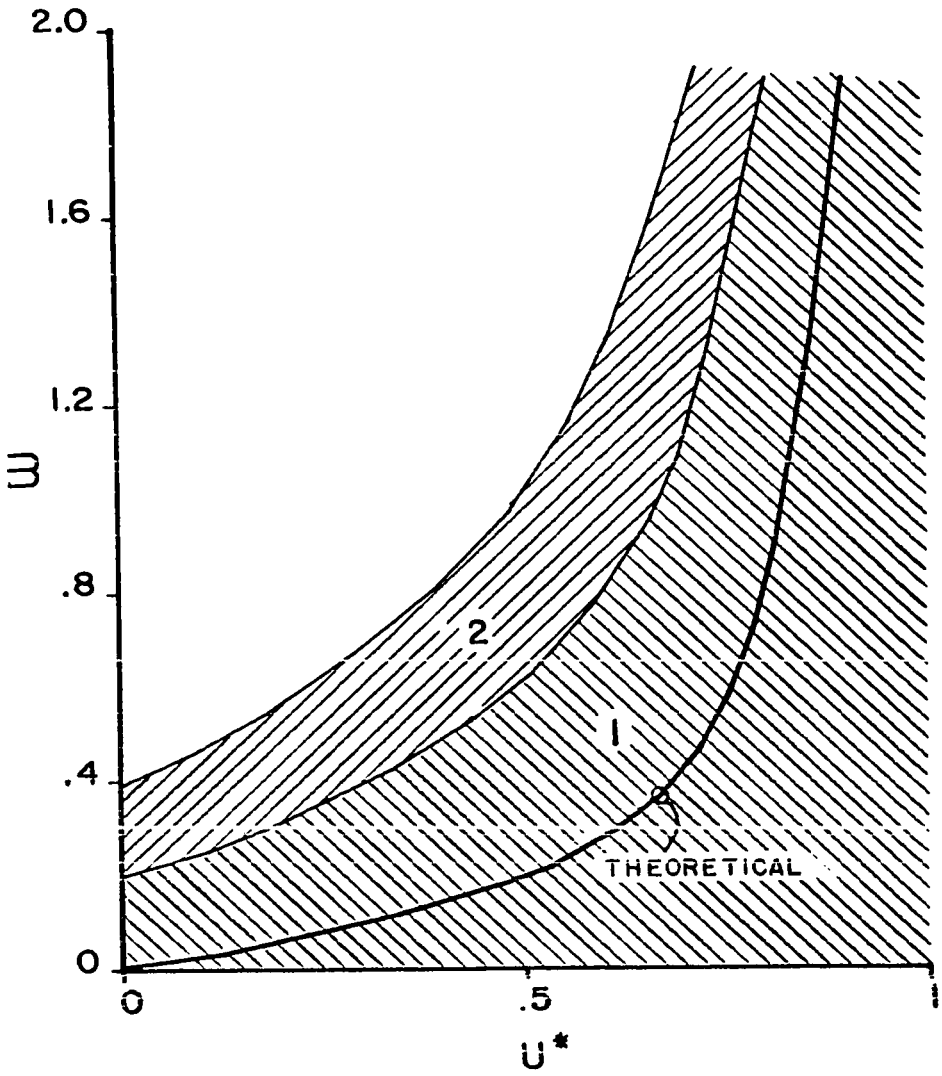
$$P\left[\, | X - \frac{1}{\mu} | \leq \frac{1}{\mu} \,\right] = 0.8646648 \qquad (23)$$

and

$$P\left[\, | X - \frac{1}{\mu} | \leq 2\,\frac{1}{\mu} \,\right] = 0.950213 \qquad (24)$$

Figure III.B.4 plots w_i versus U_i^* for a value of $v = 5X10^6$ slot/sec. In addition, the regions within σ and within 2σ have been indicated by shading. Thus, the probability that a waiting time falls within any of the shaded area is 0.95.

The design of this system architecture provides for the dynamic addition and deletion of processors to a closed section of a bus through the action of the BUS SHORT modules. As a consequence, it is extremely important to know how the response of a bus will vary with the number of processors on it. In order to obtain an expression for the wait time in terms of N, the number of processors, some assumption must be made about the processor's activity and/or the distribution of bus traffic. In general, the activity and distribution of traffic on a



$V = 5 \times 10^6$ SLOT/SEC

PROBABILITY W FALLS WITHIN REGION 1 = 0.86

PROBABILITY W FALLS WITHIN REGION 1 & 2 = 0.95

PROBABILITY REGIONS FOR WAIT TIME VS. BUS UTILZATION

Fig. III.B.4

bus are expected to be a complex, time varying set of quantities and impossible to use in an analysis. However, there are three special cases of traffic distribution that tend to delineate the range of distributions expected and are interesting to study.

The first is the case having one transmitting processor that transmits equally to all other processors (except itself). Obviously, all the traffic will be removed from the bus before it returns to the transmitter; as a consequence, the transmitter will have $U^* = 0$ and will never have to wait. The second case is the opposite of the first. All processors transmit equally to one receiver. In this case, the utilization and wait time of a processor is based solely on its position. The first processor downstream from the receiver sees a clear bus, all the traffic having been removed by the receiver. Therefore, this processor has $U^* = 0$ and no wait time. The first processor therefore transmits its Q word message at the processor rate, r. The second processor downstream from the receiver now sees the traffic from the first, has $U^* = r/v$, and has an appropriate wait time. The third processor sees $U^* = 2r/v$ and must wait appropriately, etc., for all processors.

The third case, symmetric bus traffic, is much more significant than the first two cases. Consider the special case that the bus traffic is symmetric, ie., each processor communicates with every other processor equally. It can be seen that the previous expressions, (6) and (7), simplify greatly

$$P_{ij} = 0 \qquad i = j \qquad \text{(from [Hayes, 25])} \qquad (25)$$

$$= 1/(N-1) \qquad i \neq j$$

$$R_k^* = R^* = \hat{\hat{r}} \left(\frac{N}{2} - 1\right) \tag{26}$$

and from (16) and (9)

$$w = \frac{1}{\left(1 - \frac{R^*}{v}\right) v} \tag{27}$$

Substituting (26) and (5) into (27)

$$w = \frac{1 + rw}{v + vrw - r_\theta\left(\frac{N}{2} - 1\right)} \tag{28}$$

Normalizing with respect to $\frac{1}{r}$, the time to produce one word and place in on the bus is

$$wr = \frac{r\ (\ 1 + rw)}{v + vrw - r_\theta\left(\frac{N}{2} - 1\right)} \tag{29}$$

Simplifying with $\gamma = \frac{r}{v}$,

$$(wr)^2 - (wr)\ \left(\gamma\left(\theta\left(\frac{N}{2} - 1\right) + 1\right) - 1\right) - \gamma = 0 \tag{30}$$

Using the quadratic equation and solving for wr

$$wr = \frac{1}{2}\Big[\gamma\left(\theta\left(\frac{N}{2} - 1\right)+1\right) - 1 \pm \Big[\gamma^2\theta^2\left(\frac{N}{2} - 1\right)^2$$

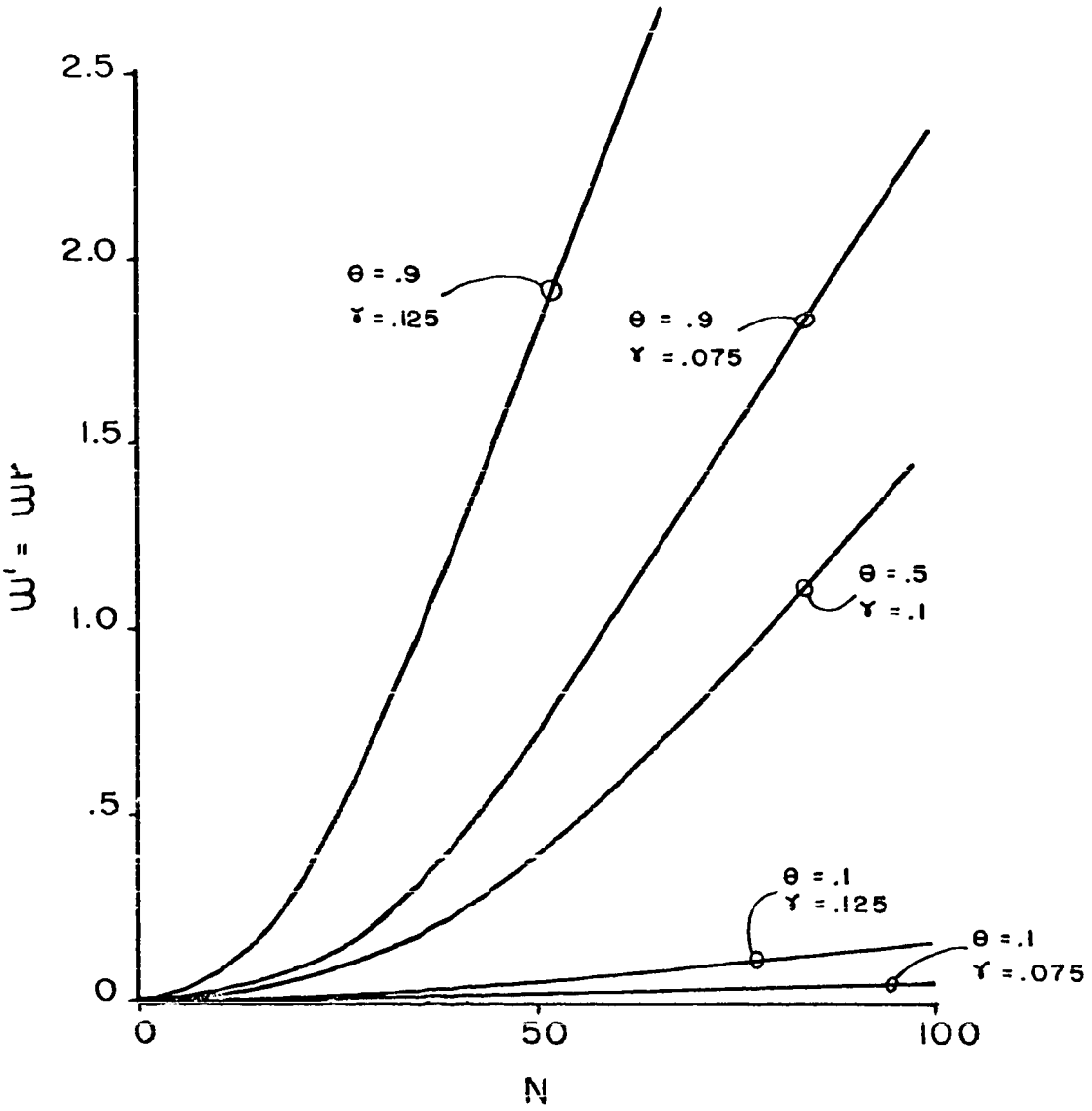
$$+ \gamma^2 2\theta\left(\frac{N}{2} - 1\right) + 2\gamma\theta\left(\frac{N}{2} - 1\right) + 2\gamma + \gamma^2 + 1\Big]^{1/2}\Big] \tag{31}$$

The negative root is discarded as being infeasible since it would imply that $w\ r \leq 0$ which is impossible.

By the same reasoning used for (15)

$$\hat{w}r = wr - \gamma \tag{32}$$

Fig. III.B.5 is a family of curves of $w' = \hat{w}r$ versus N, the number of modules, in a system having symmetric bus traffic. It is important to note that these curves are approximately linear for large N (ie. greater than 25) and for $w' = \hat{w}r \geq .5$. As a consequence, with the dynamic addition or deletion of processor to a section of a bus, the performance of the system is not expected to fluctuate wildly, etc. Indeed, it appears to be fairly well behaved.



FAMILY OF CURVES FOR NORMALIZED WAITING TIME
vs. NUMBER OF PROCESSORS; SYMMETRIC TRAFFIC
Fig. III.B.5

C. Calculation of Waiting Time (Buffering)

As mentioned previously, the protocol for output to the bus discussed so far suffers the deficiency that the bus must be moving slowly enough that the processor can place the output word in the empty slot it has detected before the slot moves on. This then is effectively a constraint on the range of value γ may take on. As seen in Fig. III.A.5, the time required to output a word can be broken into the time to get the word from memory, t_g, and the time to release the word to the bus, t_o. Assuming that the ratio of t_g to t_o is fixed, they may be written as $t_{g_i} = \frac{m}{r_i}$ and $t_{o_i} = \frac{1-m}{r_i}$ for $0 \leq m \leq 1$. As Fig. III.A.5 indicates, $\frac{1}{v}$ must be longer than t_{o_i}, $\forall$ i

$$\text{ie, } \frac{1}{v} > t_{o_i} = \frac{1-m}{r_i} \tag{33}$$

or

$$\gamma_i = \frac{r_i}{v} > r_i t_{o_i} = 1 - m \tag{34}$$

Since the average time a processor must wait per message word is strongly dependent on γ, this could be a severe restriction.

In an attempt to remove this restriction, an output buffer register is inserted between the processor and the bus as shown in Fig. III.C.1. This will allow the processor to place an output word in the buffer and initiate another action (eg. get the next message word from memory). The buffer, operating at essentially the bus speed, can then place the word on the bus automatically upon detection of an empty slot. The processor would still be

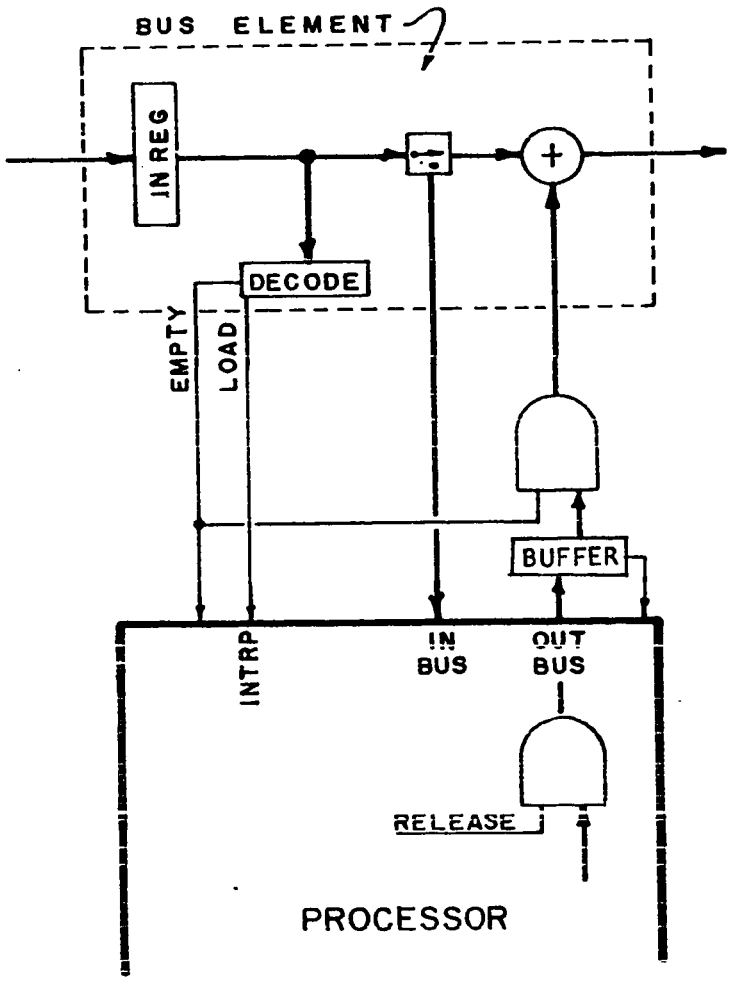


ILLUSTRATION OF BUS INTERFACE WITH OUTPUT BUFFER

Fig. III.C.1

forced to wait should the buffer fail to find an empty slot before the processor is ready to output the next word. However, γ is now free of the previous constraint. Figure III.C.2 illustrates the temporal relationships involved in this processor-bus interface.

The analysis of this second bus interface is very similar to the first. Indeed, from Fig.III.C.2 one can see that with a suitable definition of terms, the form of the result will be exactly the same. In any case, if $t_{g_i} = 0$, $w_i = \bar{w}_i$ where $\bar{w}$ is the time the message word waits in the buffer and w is the time the processor must wait to load the buffer. The result, in this limiting case, is exactly that of the previous section.

As in the first analysis, observe that the time for a processor to place a word on an active bus is

$$\frac{1}{r_i} = \frac{1}{r_i} + w_i .$$

However,

$$\frac{1}{r_i} = t_{g_i} + t_{o_i} \tag{35}$$

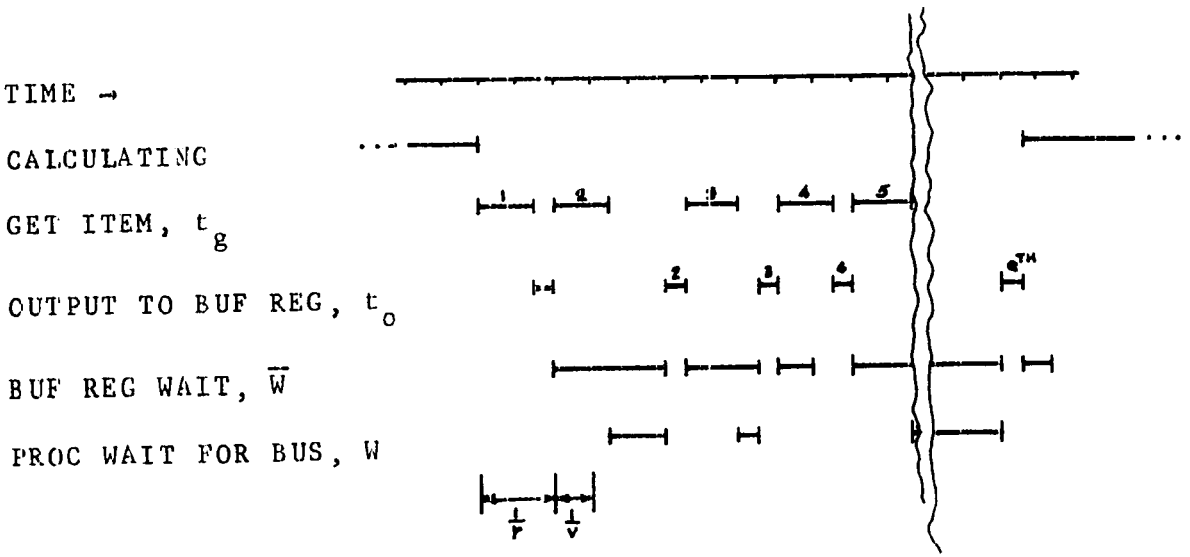
If t_{g_i} and t_{o_i} are assumed to be fixed in proportion to one another, we can then write

$$t_{g_i} = m \frac{1}{r_i} \tag{36}$$

$$t_{o_i} = (1-m) \frac{1}{r_i}, \quad 0 \leq m \leq 1 \tag{37}$$

As seen in Fig.III.C.2,

$$w_i = \bar{w}_i - t_{g_i} \tag{38}$$



TIMING DIAGRAM FOR BUFFERED INTERFACE

Fig. III.C.2

Thus,

$$\hat{r}_i = \frac{1}{\frac{1}{r_i} + w_i} = \frac{1}{\frac{1}{r_i} + \bar{w}_i - t_{g_i}}$$

$$= \frac{r_i}{1 + r_i w_i - m} \tag{39}$$

Defining the processor utilization as before,

$$\theta_i = \frac{\hat{r}_i Q_i}{Q_i + I_i \hat{r}_i} \tag{40}$$

It is expected that the application of a buffer register in this fashion would improve the efficiency of the processor. This improvement would be indicated by a decrease in θ, the utilization of the processor (for communication).

$$w_i = \frac{1}{(1 - U_i^*)\, v}$$

$$= \frac{1}{(1 - \frac{R_i^*}{v})\, v} \tag{41}$$

In the case of symmetric traffic we again have

$$wr_{\theta_2} = \frac{1}{2}\left[\gamma(\theta_2(\tfrac{N}{2} - 1) + 1) - 1 + \left[\gamma^2\theta_2^2(\tfrac{N}{2} - 1)^2\right.\right. \tag{42}$$

$$\left.\left. + 2\gamma^2\theta_2\,(\tfrac{N}{2} - 1) - 2\gamma\theta_2(\tfrac{N}{2} - 1) + 2\gamma + \gamma^2 + 1\right]^{1/2}\right]$$

However, θ has been improved.

$$\bar{w}r = wr_{\theta_1} \tag{43}$$

and is given by the previous equation for wr, (42)

and $$w = \bar{w} - \frac{m}{r} \text{ and for } 0 \leq \bar{w} < \frac{m}{r},\ w = 0 \tag{44}$$

In addition to the improvement in θ and w, there is now no constraint on γ, and it is now to our advantage to minimize γ (or increase v.)

In order to quantify the improvement in θ, a factor, F, may be defined to represent the proportional decrease.

$$F = \frac{\theta_{2_i}}{\theta_{1_i}} = \frac{\dfrac{\hat{r}_{i_2} Q_i}{Q_i + I_i \hat{r}_{i_2}}}{\dfrac{\hat{r}_{i_1} Q_i}{Q_i + I_i \hat{r}_{i_1}}}$$

$$= \frac{Q_i + r_i w_i Q_i + I_i r_i}{Q_i + r_i w_i Q_i + I_i r_i + m_i Q_i}$$

$$\therefore F = \frac{1}{1 + m_i \dfrac{1}{I + r_i w_i + \dfrac{I_i}{Q_i} r_i}} < 1$$

where θ_{1_i} is for the case with no buffer and θ_{2_i} is for the interface model with a buffer.

Since the form of the two interface models is the same, the result is expected to be the same and indeed it is, (using the new values of $\hat{r}$, θ_i, etc.)

D. Overhead and Improvement Factor

A system architecture has now been presented and some of its characteristics determined. However, it still remains to be shown that the use of this system will yield any enhanced performance over a single processor, etc. Also, assuming that execution times are decreased, it is desirable to know what strategies should be employed in applying the architecture to a problem in order to obtain the largest improvement possible over the single processor. As a result, it is important to know something about the overhead of the system inherent in the interprocessor communication structure.

To obtain an estimate of the overhead, consider the case of symmetric bus traffic in which each processor communicates equally with all other processors in a single control group system. Assume also that each communication between processors requires Q words on each of the three busses, DATA, CMD, and DONE. Then the overhead required by this system over a single processor is the sum of the time required to fetch each item and store it on the bus, the wait time of each item (for a bus slot) plus the transit time.

$$\therefore OV = (3 \cdot \frac{1}{r} + 3 \cdot w)\ Q + t \qquad (46)$$

The worst case transit time will be the time to circumnavigate the section of the bus in use.

$$t = N\frac{1}{v} \qquad (47)$$

$$\therefore OV = 3Q\left(\frac{1}{r} + w\right) + N\frac{1}{v} \qquad (48)$$

Assuming also that each processor performs a total of $\frac{1}{N}^{th}$ of the task in parallel with all other processors, the total task will be finished in

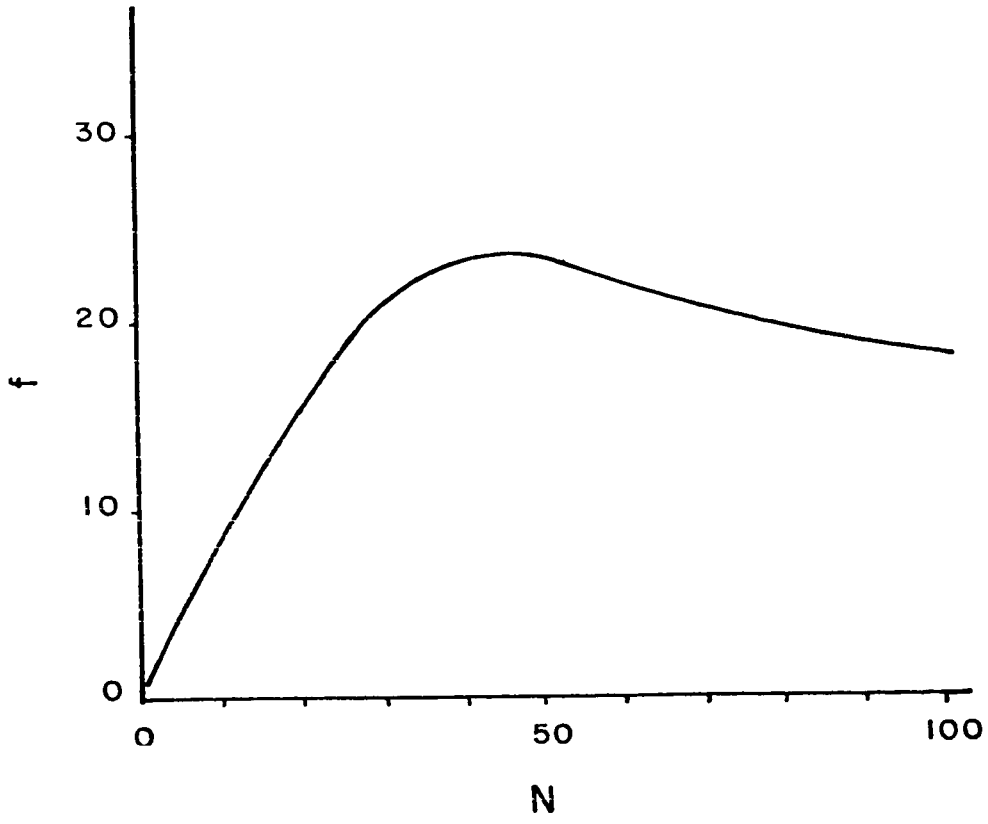
$$ET_p = \frac{1}{N}(ET_s) + OV \qquad (49)$$

where ET_p represents the execution time in the parallel system and ET_s the execution time in a single processor case. Normalizing with respect to ET_s and taking the inverse results in an improvement factor, f.

$$f = \frac{\frac{1}{ET_p}}{\frac{}{ET_s}} = \frac{1}{\frac{1}{N} + \frac{OV}{ET_s}}$$

$$f = \frac{1}{\frac{1}{N} + \frac{1}{ET_s}\left(3Q\left[\frac{1}{r} + w\right] + \frac{N}{v}\right)} \qquad (50)$$

Fig. III.D.1 is a sample plot of f versus N. Note that f climbs very rapidly with increasing N while the value of N is small. The improvement factor then reaches a peak and gradually diminishes thereafter. This indicates that for optimum system performance, the roots of df/dN should be found and f_{max} determined from them. However, it will seldom, if ever, be possible to perform this calculation. Apriori knowledge of the various parameters in general is impossible to obtain. The expression and sample plots similar to Fig. III.D.1 can indicate general strategies which, if employed, will yield increased performance. For example,



θ = 0.9

ET_s = 1 ms

Q = 1

V = 5×10^6 SLOTs/SEC

r = 3.75×10^5 ITEMS/SEC

SYMMETRIC CASE

IMPROVEMENT FACTOR VS. NUMBER OF PROCESSORS

Fig. III.D.1

1. $f \to 0$ as $ET_s \to 0$

   $f \to N$ as $ET_s \to \infty$, hence do not use multiple processors on normally short jobs, just stick with one processor from the system. For long jobs, use as many processors as practical. Although this may not be optimum, it should be relatively close due to the slow decrease in f with increasing N beyond f_{max}.

2. $f \to 0$ as Q as ∞

   $f \to \frac{N \cdot ET_s \cdot v}{ET_s + N^2}$ as $Q \to 0$, therefore be brief in all communications.

3. $f \to 0$ as $N \to \infty$, therefore don't get carried away in applying 1 above.

4. f increases as $\frac{N}{v}$ decreases, therefore allocate processors to a task that are as close as possible to each other and short the bus connecting them to minimize the time enroute for an item on a bus.

etc.

In all cases the programmer's judgment is required but these guides should improve his ability to make sound decisions.

E. Miscellaneous Remarks

Throughout this analysis, an exponential distribution of the idle and busy periods of a bus was assumed. In addition, another assumption is that these periods are independent of each other and stationary. These assumptions then indicate the region of application of this model. In particular, as Hayes and Sherman [25] point out, the model best describes the real system when the processors emit short messages and/or the bus utilization is low (ie., messages of several words remain together during passage from the sender to receiver and are not broken up as they are multiplexed onto the bus).

Another characteristic of circulating bus systems not previously discussed is a phenomenon pointed out by Avi-Itzhak [3] and Anderson, Hayes, and Sherman [1]. In certain cases (generally with $U^* > 0.9$ but occasionally with more moderate loads) competing groups of users can combine to effectually capture the bus and lock out use by other (less active) users. Control of the bus see-saws back and forth between the competing groups. The average waiting time of each group oscillates while the waiting time for the locked out users can grow very large. This is a potentially devastating situation that must be avoided. Therefore, it is important to keep the utilization of the bus as low as possible. Anderson, Hayes and Sherman note that with light to moderate loading, if capture and lock out occur, their effects are transient and eventually die out; they die out more quickly with lighter loading. Additional steps to detect and signal a locked out condition may also be necessary to enable remedial action to be taken.

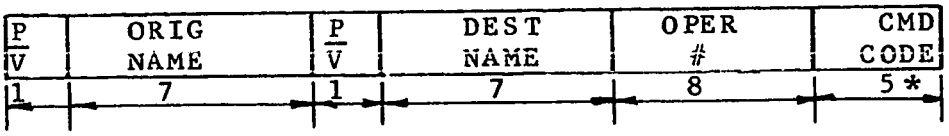
IV. DETAILED DESCRIPTION OF THE SYSTEM

A. Introduction

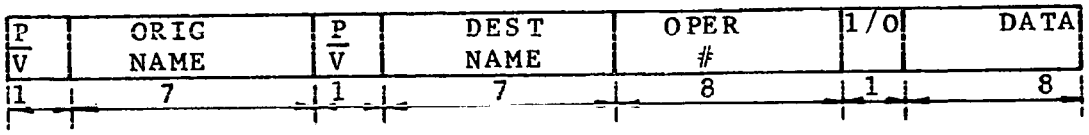
This section discusses the design and operation of a system module in more detail than was previously provided. The discussion also includes possible implementation of several major submodules. Section III of this thesis, concerning the analysis of a processor-bus interface, emphasizes the need for high speed busses. With the emphasis in mind, the design for the various busses attempted to ensure that a conservative, worst-case bus speed of $5X10^6$ slots/sec could be obtained; this figure was easily met for those busses that are elements of C.G. [1] and higher. Because the DATA BUS and C.G. [0] do not have BUS SHORT modules, they are simpler and exceed this speed requirement by a factor of about 1.5.

This section assumes that the system operates as described previously using an eight bit wide microprogrammed microprocessor chip set having TTL standard logic levels. For this discussion, this system will be configured to allow a maximum of 127 system modules, resulting in the following typical bus field widths:

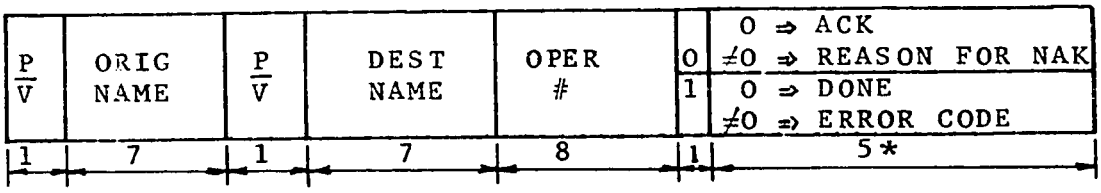
CMD BUS



DATA BUS



ACK/DONE BUS

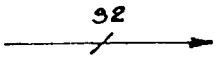


For the purposes of this and subsequent discussions, consider that all status and processor control information is grouped into a processor status word and that all elements of the 1/O and control structure have been mapped into the regularly addressable memory space. All times discussed are worst-case figures and are given in nanoseconds unless otherwise noted.

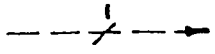
Fig. IV.A.1 defines the non-IEEE and non-MIL standard symbols used throughout this section.

*These fields can be adjusted as required in a finial implementation. The number chosen here are for example only, and should be reasonable for most applications of this architecture

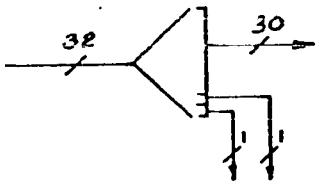
Fig. IV. A.1 BLOCK DIAGRAM SYMBOLS



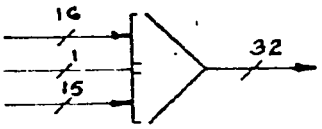
Data path, 32 bits wide
Arrowhead represents direction of data flow



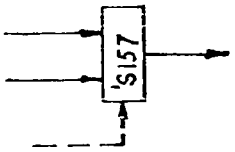
Control path, 1 bit or line wide



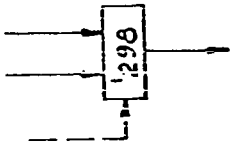
Data path indicating divergence of 2 adjacent end bits.



Convergence of 3 data paths into 1 path with relative position of the converging paths indicated.

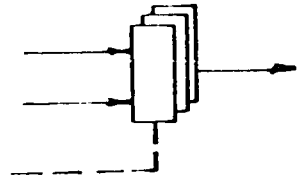


74S157 (or equivalent) data selector, 4 bits wide



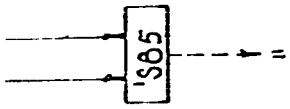
74298 (or equivalent) data selector, with storage register, 4 bits wide

Fig. IV.A.1 BLOCK DIAGRAM SYMBOLS (CONT.)



Multiple circuits of type indicated operating in parallel (ie., to form wider data path, etc.) (no. of circuits is not indicated)

Indicates the portion of the diagram contained on a single system module.



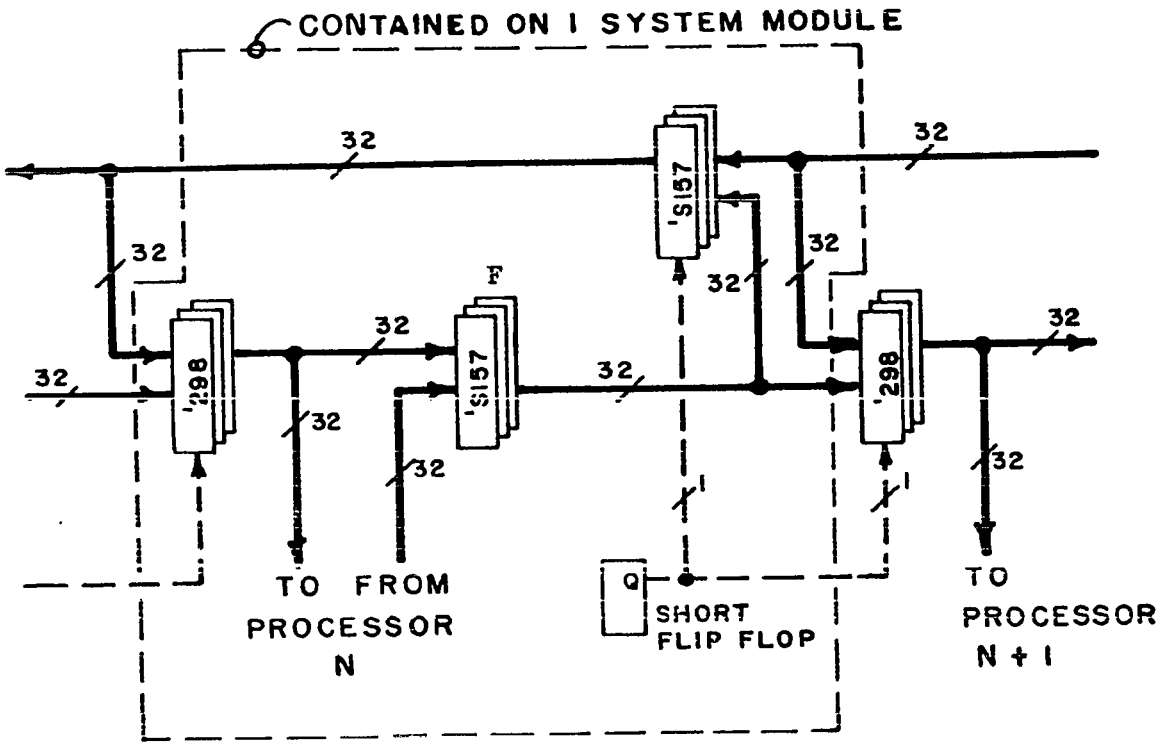
74S85 (or equivalent) comparator.

B. Busses

Each of the busses incorporated into this architecture is a circulating bus or Pierce Loop. The basic configuration of an element of a C.G. bus (other than C.G. [0]) is illustrated in Fig. IV.B.1; as seen in this figure, the mechanism is relatively straightforward and simple. The bus register for each system module is incorporated with a data selector to implement the SHORT operation. As each register consists of edge triggered D-type flip flops, only a single phase clock is required to shift the contents of the bus.

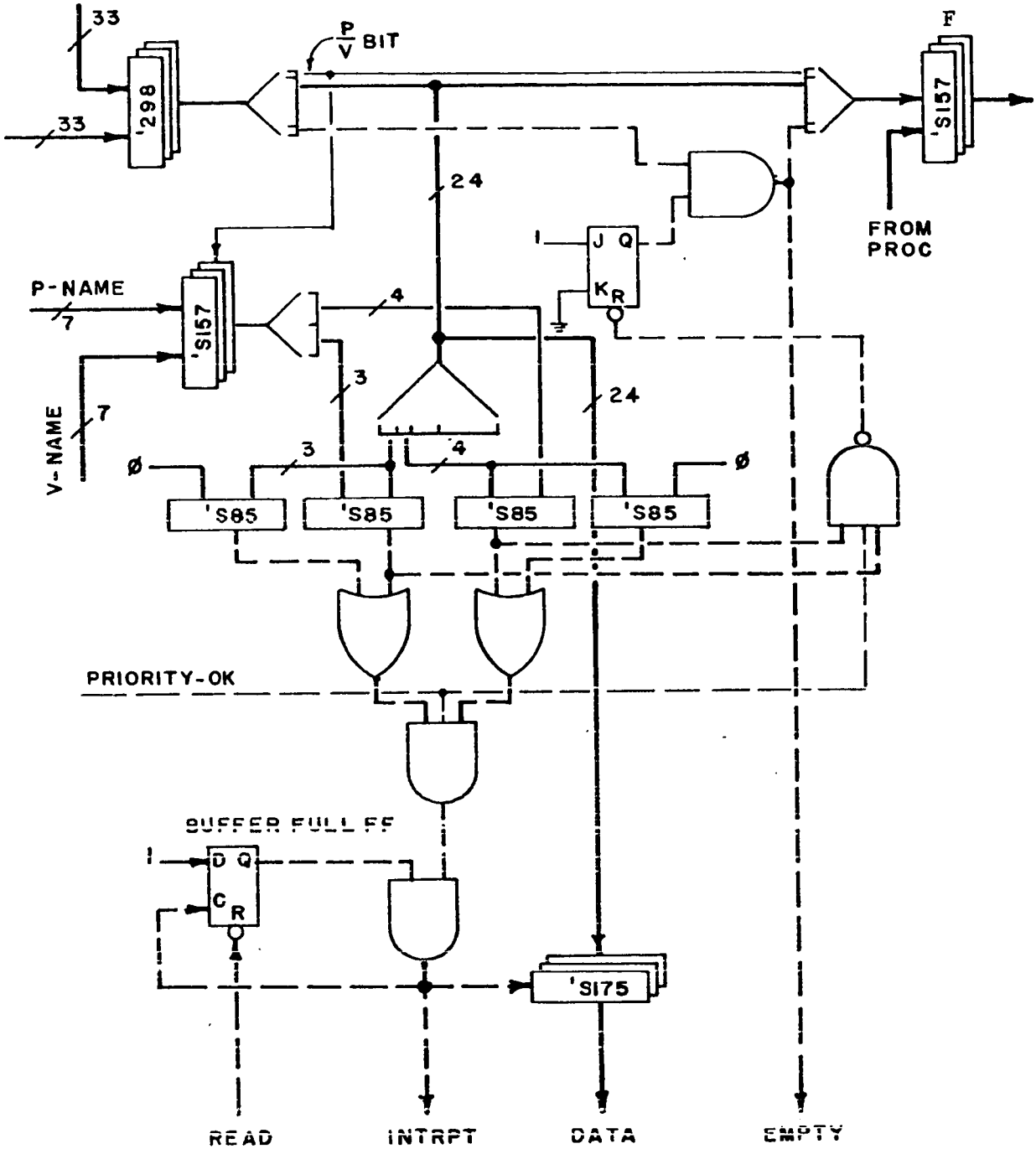
Fig. IV.B.2 illustrates the implementation of the address detection and slot emptying circuits. The destination address of each item stored in the bus register is compared with the module's P-name or the contents of its V-name register as indicated by the P/V bit in the destination address field of the item. In addition, this field is also compared with the universal name code. (See discussion of bus formats in Section A). Assuming that the INPUT BUFFER REGISTER is empty (as indicated by the BUFFER FULL FLIP FLOP, the item whose address matches the module's is automatically loaded into the buffer register and the BUFFER FULL FLIP FLOP set. In addition, the EMPTY flip flop is set, forcing the slot's empty bit to 0. The next clock pulse shifts the bus one position and resets the EMPTY flip flop. If the BUFFER FULL FLIP FLOP had been set or the processor's priority masked operation on this particular bus, no action would have taken place. The processor automatically clears the BUFFER FULL FLIP FLOP by reading the INPUT BUFFER REGISTER.

Fig. VI.B.3 illustrates the output interface. When the OUTPUT BUFFER REGISTER is written by the pro-



BASIC BUS ELEMENT

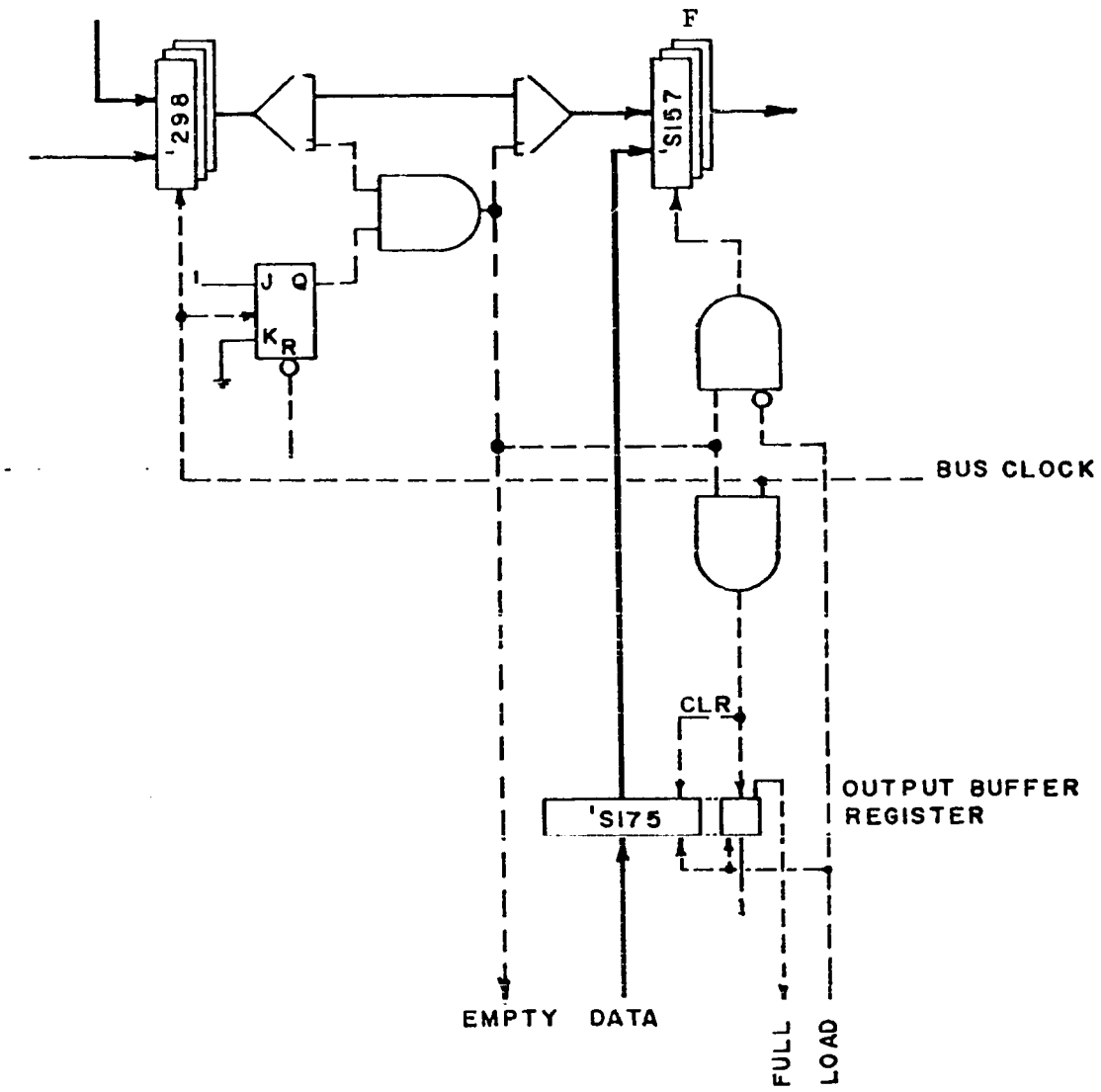
Fig. IV.B.1



PROCESSOR

INPUT FROM BUS TO PROC

Fig. IV.B.2



PROCESSOR

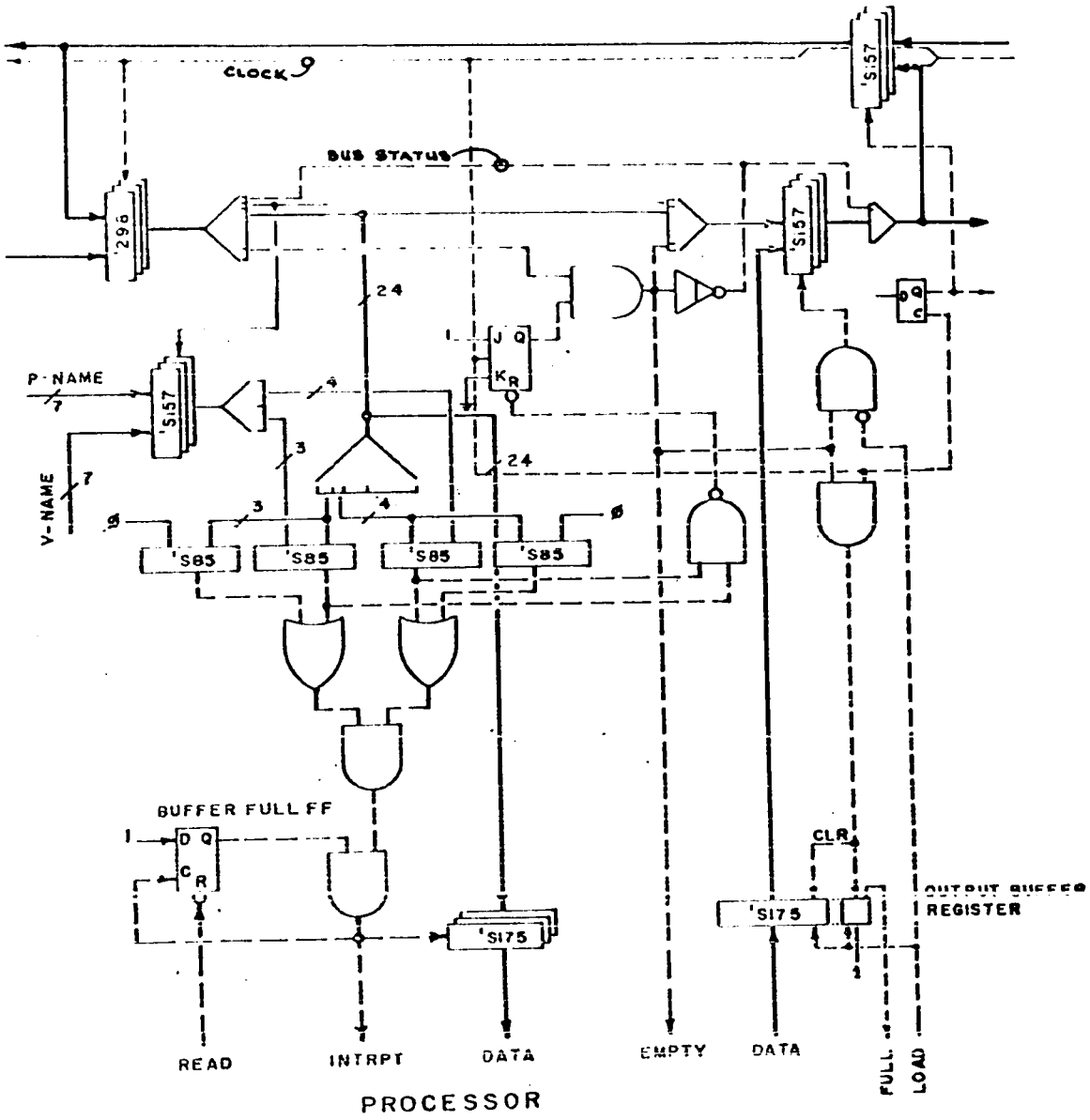
OUTPUT FROM PROCESSOR TO BUS

Fig. VI.B.3

cessor, the FULL FLIP FLOP is set. The processor is required to wait until the the FULL FLIP FLOP is cleared before attempting to write to the OUTPUT BUFFER REGISTER again. When an empty slot is detected, the OUTPUT BUFFER is selected by data selector F. The next clock pulse shifts its contents onto the bus and resets both the register and the FULL FLIP FLOP.

Fig. VI.B.4 is a combination of the previous figures showing the bus plus the interface between the bus and the processor. In addition, it contains an additional line, the BUS STATUS LINE. This line follows the path taken by the information traveling on the bus and is shorted just as the basic bus is. However, it has no register or information storage facility. It is driven by the EMPTY BIT of each bus stage through an open collector or tri-state bus driver. The purpose of this line is to indicate to all modules along a section of the bus whether there is any non-empty position on this bus section. To avoid trapping an information packet on a section of the bus to be shorted, it is necessary that a processor be able to detect the presence of bus traffic. Before activating its BUS SHORT module, the processor must wait until all traffic is removed from the bus or possibly face error correcting requirements. The BUS STATUS line provides this capability. Effectively it provides the (wired) NOR of the EMPTY BIT position of the bus for each module. Thus, if any position is non-empty, the line is pulled low and may be read by each processor on the particular bus section

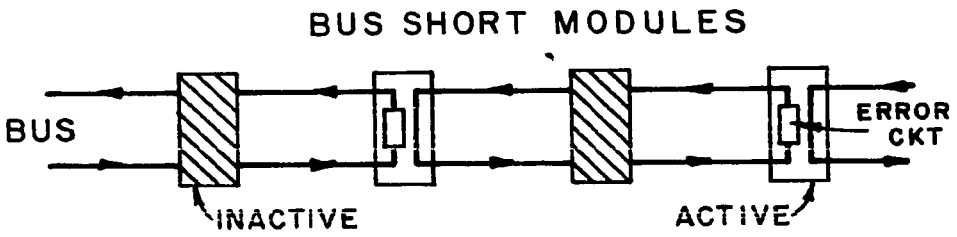
To those portions of the busses already described, it is necessary to add an additional facility. As with any device having a finite size or capacity, it is possible to fill up the bus. In general, this would not be



TOTAL BUS INTERFACE

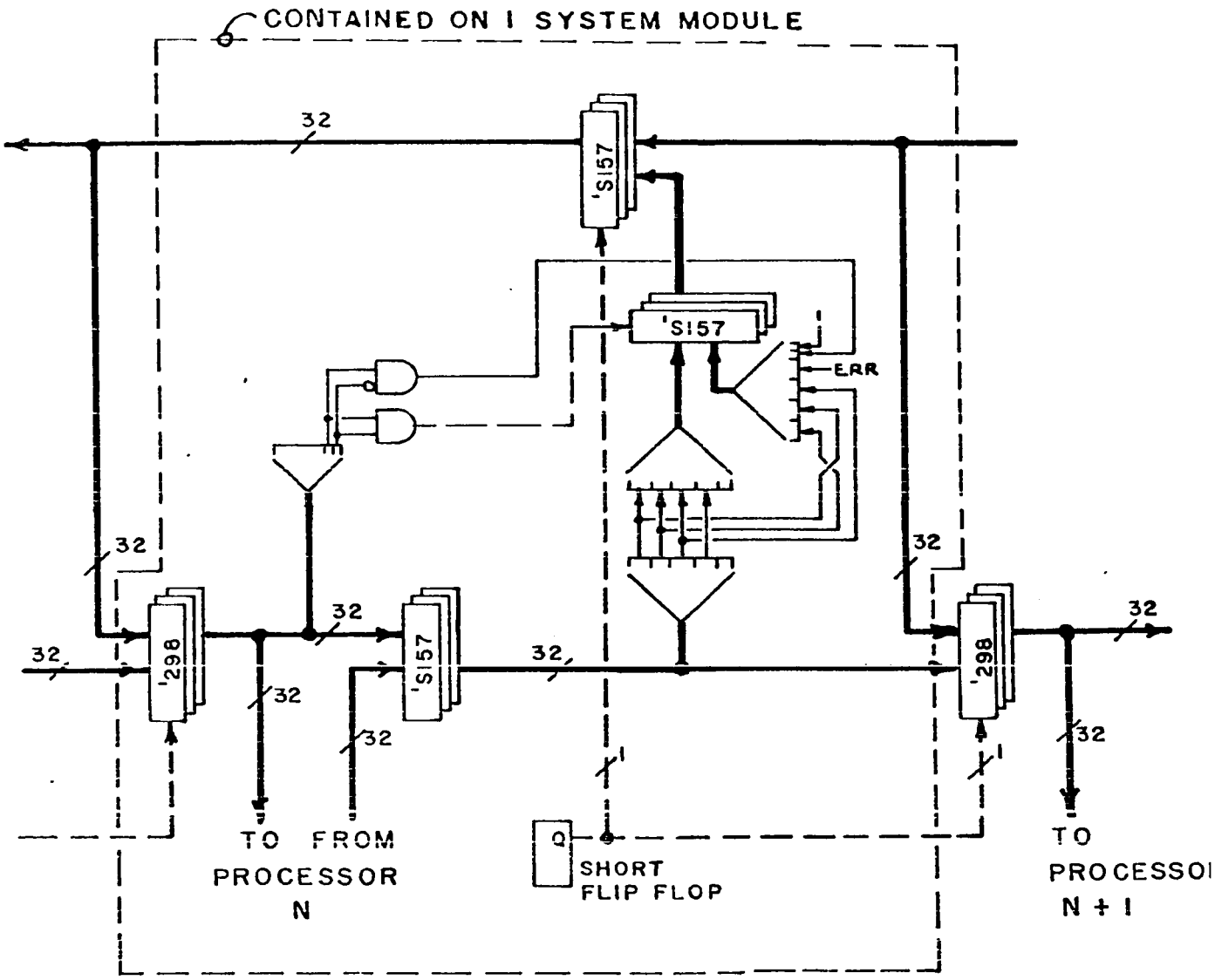
Fig. IV.B.4

serious (apart from the delays it would cause). Since a particular item passes by each module in each cycle around the bus, it would normally be removed by its destination at some time during the first pass. A gap in traffic for addition data would therefore be provided and deadlock prevented. However, it is possible that this might not be the case. For example, if two consecutive items on a single bus are destined for the same processor module, the first will fill the processor's INPUT BUFFER REGISTER and lock out subsequent data items. If the interrupt facility of the processor has a latency in excess of one bus slot time, the processor will be unable to empty the input register in time to receive the second item. The second item must therefore bypass the processor and make a complete cycle of the bus. Thus, each item must be capable of remaining on the bus for several cycles. On the other hand, it is possible that a data item can be placed on the bus with an incorrect or corrupted address (due to programming error or noise). As a result, it has no legitimate destination. In addition, if the destination processor fails, its traffic will never be removed from the bus and a deadlock with a full bus could easily result. Thus, it is imperative that traffic be allowed to remain on a bus for only a finite length of time before an error is declared and corrective action taken. Fig. VI.B.6 shows how this facility can be implemented very simply. This scheme requires the addition of several control bits to each bus. As each word on the bus passes through the bus short module at the right end of the bus section (see Fig.IV.B.5) and begins its return to the left end of the bus, a set of control bits is set to indicate the number of times the word has passed by. When this indication exceeds the maximum, the destination and source address fields are



LOCATION OF ERROR CIRCUIT

Fig. IV.B.5



ADDITION OF ERROR DETECTION TO BASIC BUS ELEMENT

Fig. IV.B.6

interchanged and the data, command, or done field is replaced by an error code. At this point an additional control bit is also set to indicate that this word is now an error word and not valid data, etc. Should the word fail to be taken off the bus by its original source, the error bit will cause the error detecting circuit to remove it from the bus by setting the empty bit appropriately. This prevents an item from "bouncing" between two unacceptable addresses. Fig. IV.B.6 implements this scheme using simply a 74S157 type data selector. Note that the control information is detected before the data selector multiplexing data from the processor onto the bus. This reduces the error detection logic's operating delay by the propagation delay of the 74S157 multiplexer from the processor. This does not cause ambiguous or erroneous operation because the processor cannot place an item on a bus unless the present bus slot is empty. The fact that the slot is empty will disable the error detection logic.

C. Function/Carry Loop

The particular architecture under discussion here is intended to have an extensible arithmetic capability. This capability is to be provided through the designation of several modules as subelements of a larger processing unit; the larger unit will be termed as an extended arithmetic logic unit or EALU. Each module in the EALU will perform exactly the same function as the other modules in this unit with the carries, etc., "rippled" from module to module. The FUNCTION LOOP provides the micro-instruction sequence for the instruction to be performed to each module connected to it. Essentially, the set of modules becomes a SIMD system. Fig. IV.C.1 shows the FUNCTION LOOP and two system modules. As noted in Chapter II, the loops described in this research have no storage of information. However, like the C-bus, the FUNCTION LOOP may be shorted at the left edge of any processor so that groups of processors can function independently of other groups of processors. Each group will be driven by the program and microprogram contained in one module. Fig. IV.C.2 shows two modules configured into a 2·n bit arithmetic processor (n is the word size of an individual system module). Each of the data selectors has been appropriately set to allow the function bus to pass from the microprogram controller around the loop and into each ALU; note that the path from the loop to the ALU is identical for each module. Processor No. 1 has been established as the driving module. In order for this processor to drive the loop, its data selector B must be set appropriately as indicated.

The microprogram controller is a synchronous circuit requiring a clock signal. Normally, in a single processor system, the clock would be fixed at a single

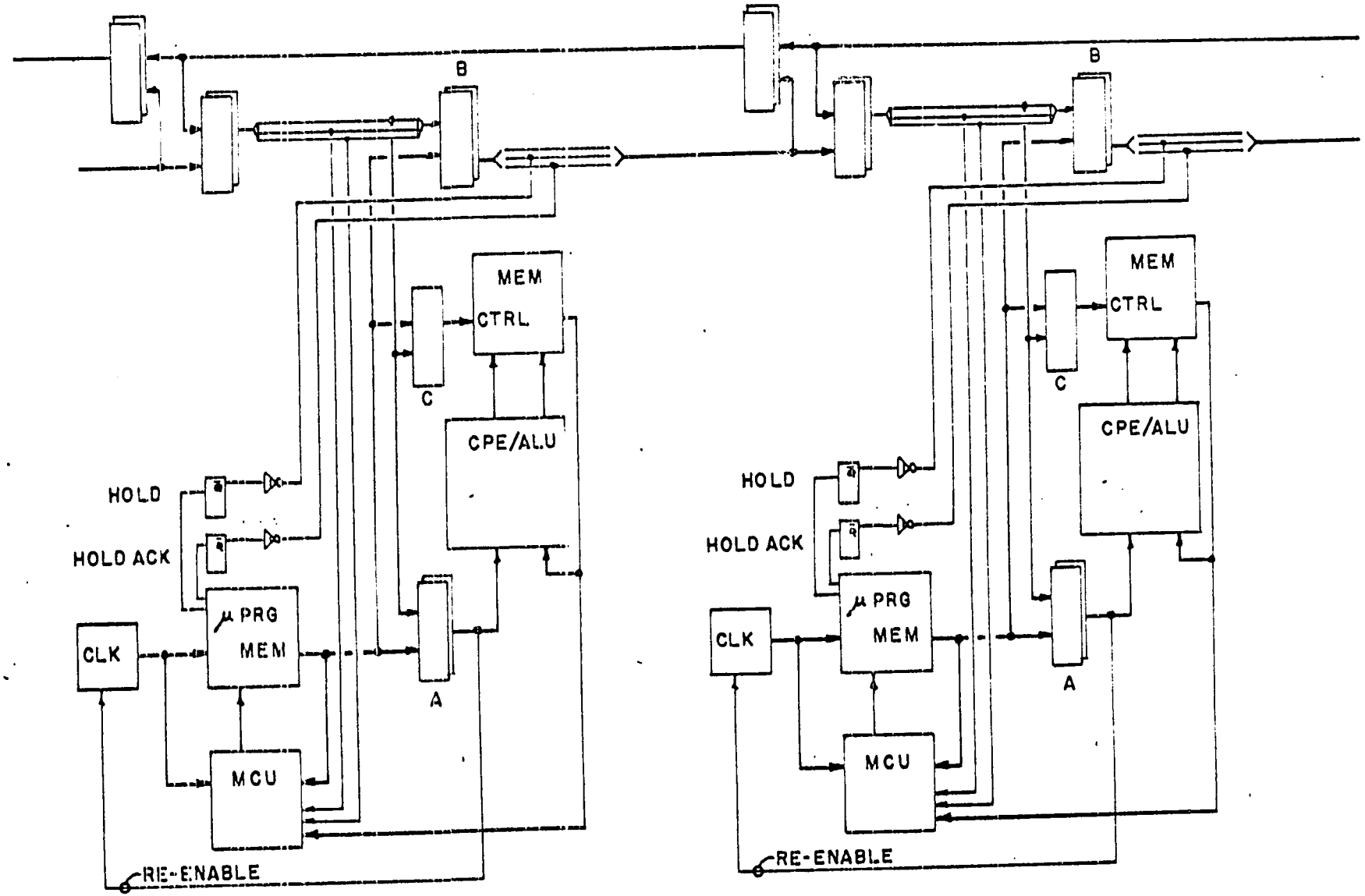
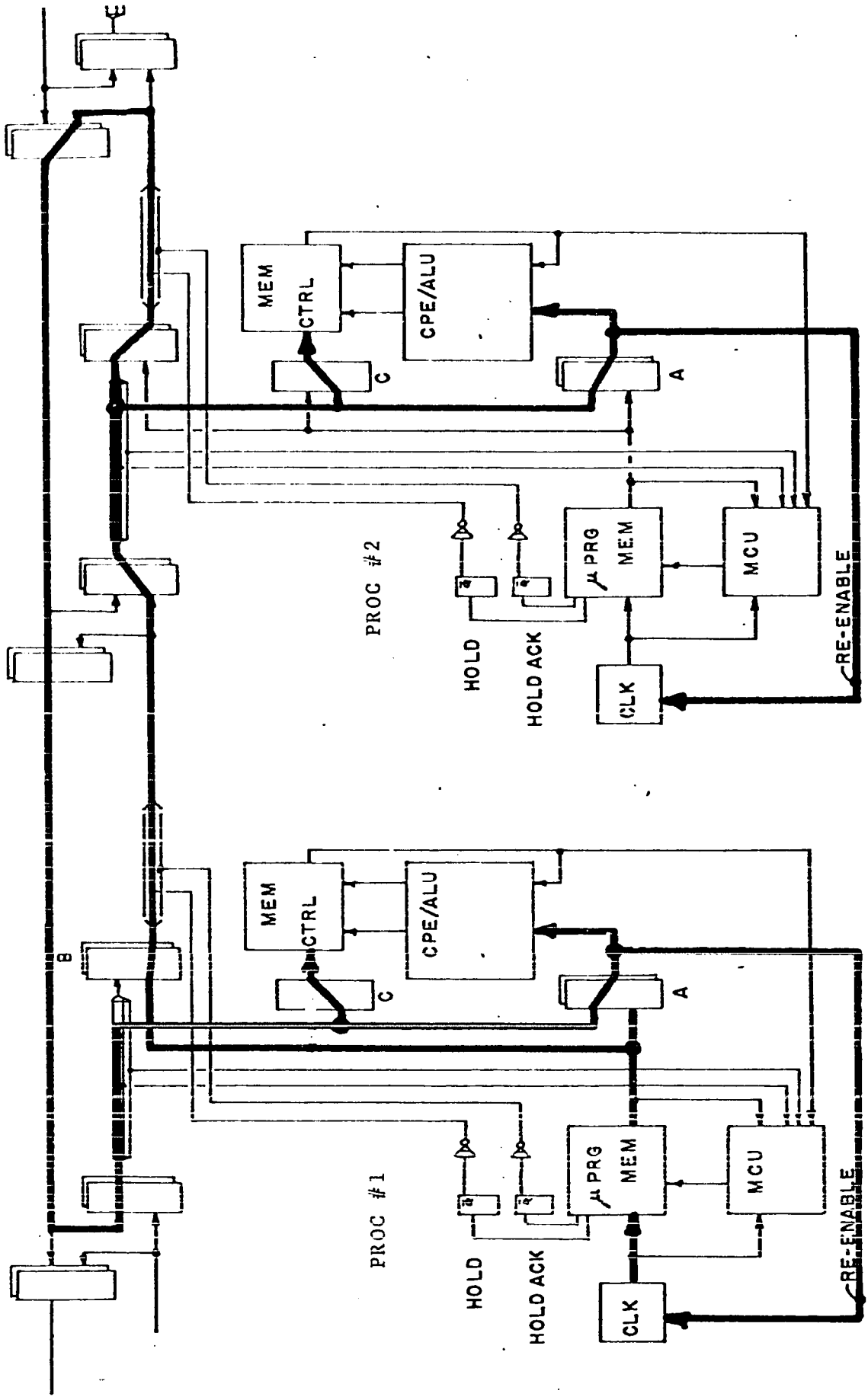


ILLUSTRATION OF THE FUNCTION LOOP FOR MICROPROGRAM CONTROL OF SEVERAL SYSTEM MODULES - Fig. IV.C.1



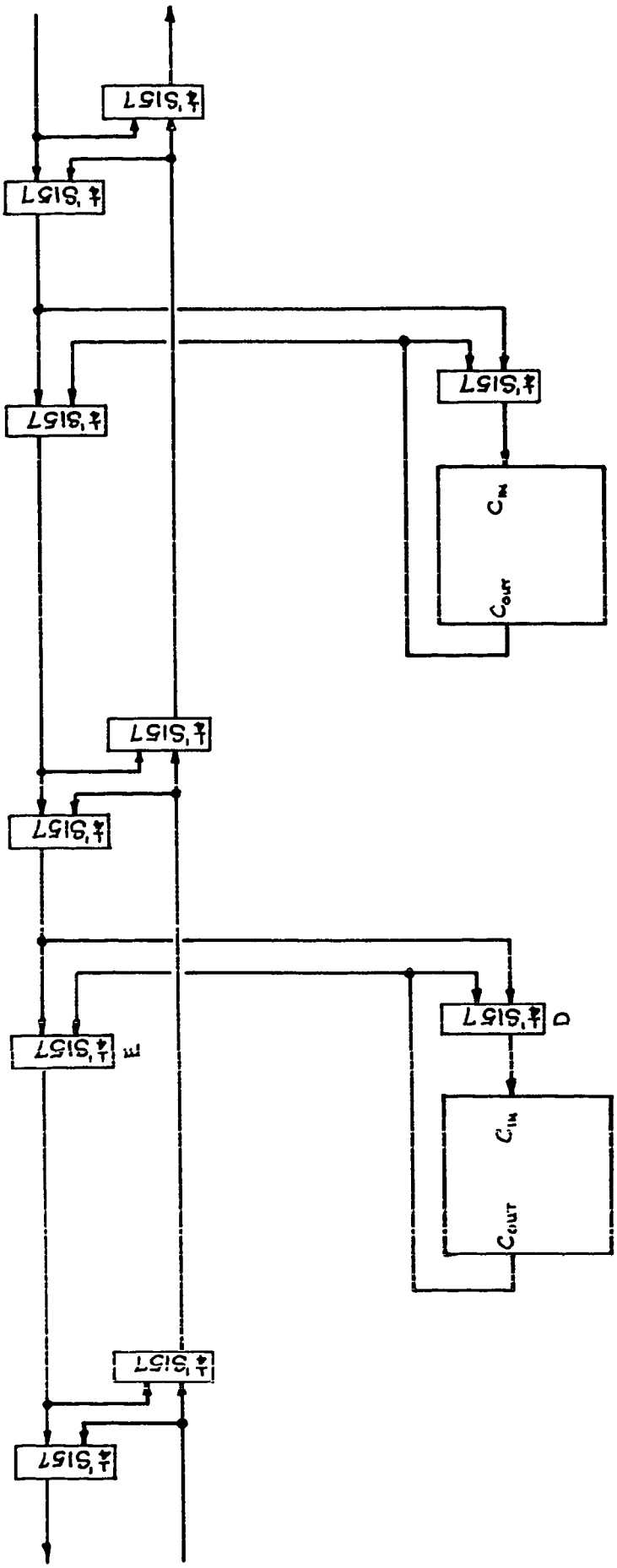
EALU ESTABLISHED VIA THE FUNCTION LOOP Fig. IV.C.2

optimum rate, and the function code provided by the microprogram controller to the ALU would typically change at each clock cycle. Because of the additional propagation delay provided by the logic of the loop when several system modules are active or the modules are widely separted, it is easy to see that the clock rate must either be slow enough to handle the worst case or variable. To slow the clock for the worst-case is not acceptable. Therefore, the clock must be provided with provisions to vary its speed. In this implementation, a clock pulse is never released from the clock until a reply is received as a result of the previous clock transition. By routing the single clock pulse around the loop and back to the originating clock unit in the driving system module to reenable it, the total system is assured of a sufficient clock period for all system modules to function correctly.

Due to the asynchronous aspect of the total system, the time of arrival of operands at each of the modules is indeterminate with the possibility of a module receiving an item, either command or data, etc. at any time. For example, the master module might require that a module within an EALU halt, reset or perform some other function as a result of system errors, interrupts, etc. As a result, each module of an EALU must still be responsive to interrupts from the busses, (especially C.G. [0]). Simultaneously, the other modules in a wider functional unit on the FUNCTION LOOP cannot be allowed to continue operation, leaving the interrupted module behind. The HOLD FLIP FLOP along with the HOLD LINE provide the necessary control for this situation. Any module connected to the FUNCTION LOOP has its microprogram controller free; this controller will continually monitor the interrupt system. Should an interrupt occur, the microprogram controller

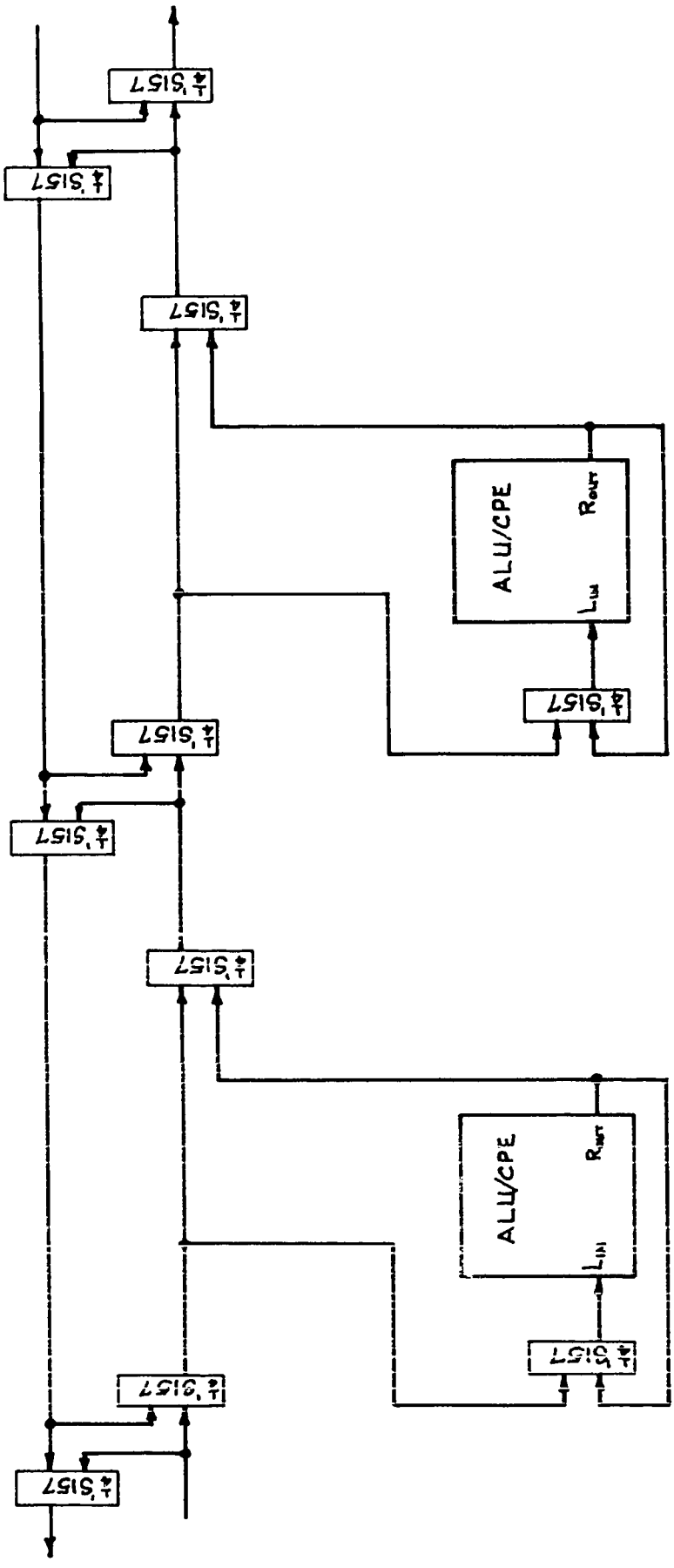
sets the module's HOLD FLIP FLOP. By means of an open collector or tri-state bus driver, the HOLD FLIP FLOP pulls the HOLD LINE low and requests a pause or hold from the driving module. The HOLD LINE is the wired NOR of all the HOLD FLIP FLOPS connected to the section FUNCTION LOOP under discussion delineated by activated LOOP SHORT modules. When the driving module receives a low HOLD LINE, it finishes the current instruction, sets its HOLD ACKNOWLEDGE bit and waits for the HOLD LINE to be released. When the interrupted module receives a response from the driving module on the HOLD ACKNOWLEDGE line, it continues the processing of its interrupt by switching data selectors A and C to enable its own FUNCTION LOOP - ALU link. Following the processors conventional interrupt servicing, the execution of the return from interrupt will clear the HOLD FLIP FLOP releasing the HOLD LINE. The release of the HOLD LINE by all interrupted modules automatically resets the driving module's HOLD ACKNOWLEDGE bit.

In order to complete the extended width arithmetic feature, there must be provisions for carries and left shifts from lower to higher order modules and right shifts in the reverse direction; end around carries are also necessary. Fig's IV.C. 3 and 4 illustrate the Carry Loop. Functioning just as the basic FUNCTION LOOP, the Carry Loop either passes the carry or right shift through a system module's ALU, or bypasses it according to the setting of data selectors D and E. M system modules can appear now as a M·n (partial) ripple carry processing unit.



THE CARRY LOOP - CARRIES AND LEFT SHIFT

Fig. IV.C.3



THE CARRY LOOP - RIGHT SHIFT PORTION

Fig. IV.C.4

D. Limitations of EALU Ability

As described, the EALU is limited as follows:

1. No module within the extent of an EALU (i.e., between shorted loop short modules) can function as a member of a second EALU.
2. There can only be one driving module in an EALU. It will be located logically at the high order position to facilitate testing of sign and carry information.
3. Although the intention is to provide extended arithmetic, most instructions can be executed by the EALU group. However, the ability to provide constants, etc., from the microprogram memory will be limited to the driving module. This limitation should not prove to be a severe restriction, however.

E. Summary

As it has been described in this section, a system module can easily be implemented using standard, readily available IC's such as the Intel 3000 series microprocessors chip set and Schottky TTL. When implemented with these products, conservative calculations of propagation delay through the various circuits using manufacturer's guaranteed worst-case figures indicate that a clock period of 200 ns is acceptable. This yields a bus velocity of $5X10^6$ slots/sec and estimates of γ that are less than one-half. Generally, we may expect to see γ on the order of .25 or less even with the fastest microprocessors chip set available today. The implication of this fact is that the wait time per word of message placed on a bus will be very low. (see Section III). In addition, since the DATA bus and C.G. [1] do not include BUS SHORT modules, the speed of these busses can be much higher yielding values of γ and wait time that are even smaller. It seems clear therefore that close processor interaction via this architecture bus structure is easily attainable.

Each bus implemented as described for an eight bit processor requires 32 information bits as described previously. In addition, an empty bit and two or more control bits are also required for a total approximately 35 bits per bus. Although this is fairly large consdering the small processor word size, it is not unreasonable. As the processor word size increases the relative amount of real information (ie. data, commands, etc.) carried in each bus word increases rapidly. Should it be determined that several parallel busses of approximately 35 bits each are impractical, a reasonable alternative is a blend of parallel and serial busses (eg. DATA and CMD busses parallel; ACK/DONE busses serial). A well chosen blend of this nature should still result in an effective system.

V. SIMULATOR

A. Description of Simulator

Generally, simulations can be broken into two categories by their objectives. One category contains those simulations designed to provide data for the evaluation of the expected performance of a system. This type of simulator is often relatively abstract and relies heavily on the use of random inputs, requests, etc., to provide a load on the system. Statistical information is then obtained about the system's performance under this load. For this type of simulation to be effective, it is important that the statistics of the random load adequately resemble those of the real system, ie., that the abstract model be adequate.

The second type of simulator is designed to provide the opportunity to observe and use the systems in a form that resembles the real system as closely as possible. Thus, this type of simulator will provide statistical information about the system's performance that is as close as possible to that expected of the real system in a particular application. In addition, the ability to use a facsimile of the system can be valuable in discovering faults in the design and implementation and allows the practical usefulness of the system to be evaluated.

The simulator constructed for this research is designed to perform in both of the above categories. As such, it mimics the action of the general system components in discrete time. Each pass through the program updates the elapsed time by one increment, moves each bus one unit and checks the address of each item in each bus slot for a match with the processor monitoring that slot. If a match occurs for a particular slot, a proces-

sor interrupt flag for the bus containing the item is set. With each pass through the program each processor checks its interrupt flags. If a processor flag having a value less than the processor's priority is set, an interrupt is assumed to have occurred. The item that caused the interrupt is removed from the bus and placed in temporary storage for the processor.

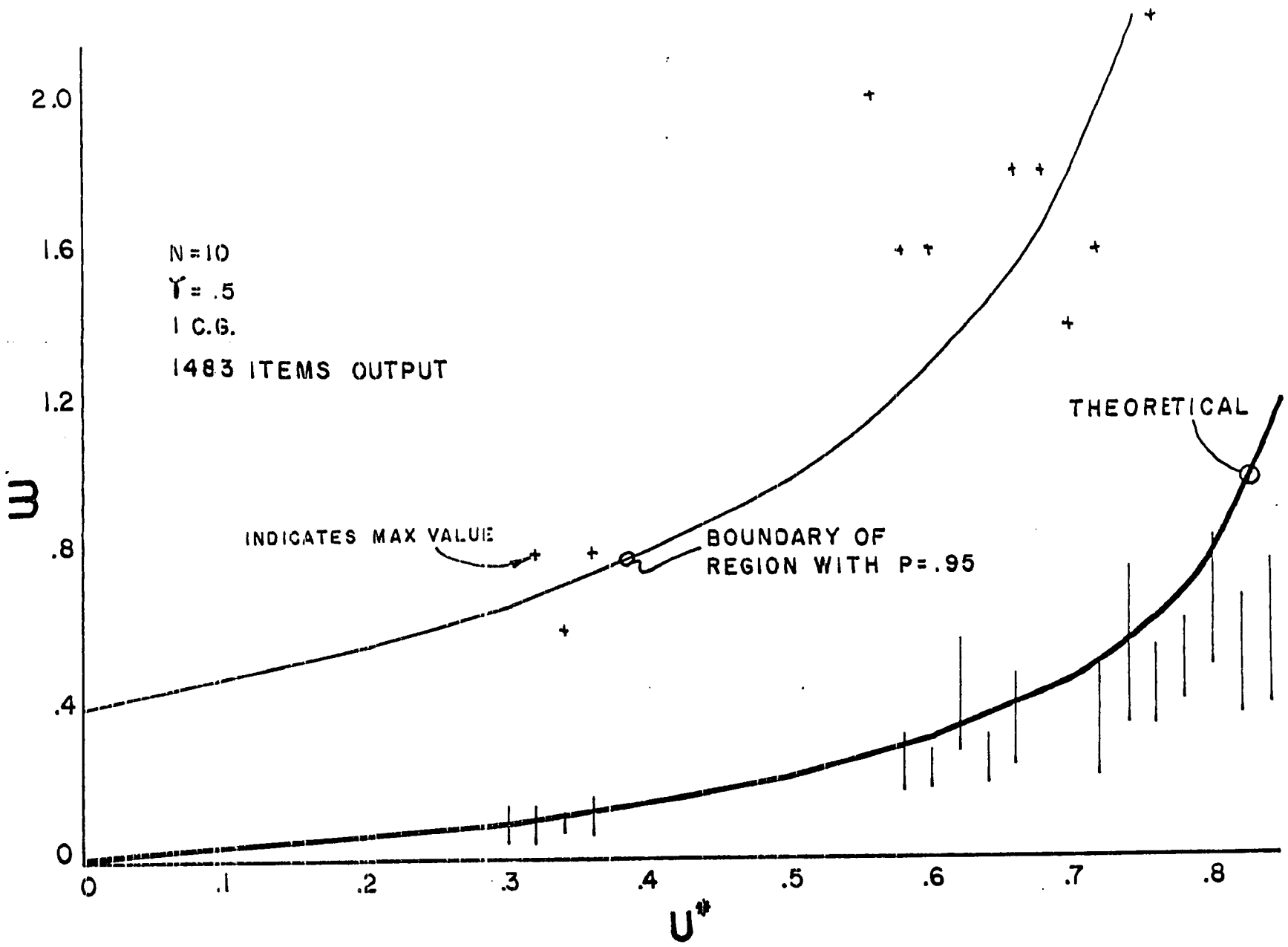
Each action by a processor is assumed to require time for its execution. Each processor, therefore, has a variable, TIMUP, into which is placed the time of execution for a task. The values for TIMUP are fixed or random as appropriate for the operation being mimicked. With each pass through the simulator, TIMUP for each processor is decremented. No new action will be initiated by any simulated processor until its value of TIMUP is less than or equal to zero.

Data is collected each time a processor succeeds in placing an item on a bus. The simulator records the processor number, bus, bus utilization as seen by the particular processor and the time the processor waited for an empty slot. The simulator also keeps a running account of the number of items output, mean and standard deviation of waiting time, the system parameter, θ, and the utilization of the bus for each processor-bus interface. This is displayed on operator command along with a "snap-shot" of the busses. The "snap-shot" shows the complete status of each bus and all data on it. One snap-shot can be obtained during each pass through the program. By observing successive snap shots of system activity, the total action of the system with respect to the interprocessor communication can be seen. An additional item entitled wait time is included in each snap-shot for each processor-bus interface. This figure, if negative, indicates that the processor is currently waiting to output an item. The absolute value

of this number is the elapsed simulated time at which the processor began waiting. If this item is positive, the processor will have successfully output an item after having waited the amount of time indicated. If zero, the processor either was not required to wait or is not attempting to output through the particular processor-bus interface in question. Also, on operator command, the total statistics collected to the current point in the simulation for the waiting time versus bus utilization may be displayed. The utilization is broken into fifty uniform intervals of 0.02 units. All data items falling within an interval are considered as estimates of the expected wait time at the level of utilization within the interval. It is well known that as the number of samples from a population of independent, identically distributed random variables grow large, the distribution of the sample mean approaches the normal distribution. In many cases, it has been found that for n>30 the sampling distribution of the mean is fairly normal [24]. As a consequence, any interval containing less than thirty items is considered to have too few samples to be valid. For those intervals having more than thirty samples, the distribution is assumed to be normal and the mean, standard deviation and 95% confidence levels are determined for each interval..

B. Simulator Results

The simulator results are displayed in Fig. V.B.1. Each vertical bar represents the 95% confidence interval for a set of sample points centered about the sample mean. As can be seen, the simulator results appear to fall along the theoretical curve reasonably well and are therefore considered to be a verification of analytical expression for wait time derived in Section III.



SIMULATION RESULTS: WAIT TIME VS. BUS UTILIZATION

Fig. V.B.1

VI. SOFTWARE

A. Introduction

Programming of any computer is a major part of the task of applying the system to the solution of problems. The software costs in terms of man-hours required, documentation effort and dollars are often equal to or greater than that of the original hardware. As a result, a significant amount of thought must be applied to the programability of a new system architecture to ensure that it may be used practically. This section will therefore discuss software (and firmware) as it applies generally to this multiprocessor scheme and provide indications as to the direction and philosophy required for the software. Several examples will be discussed and general comments made concerning the relative execution times for these examples using this system versus a single processor similar in capability to that incorporated here. As this system is adaptable to variety of microprocessors, a hypothetical processor similar to several that are commercially available will form the basis for these examples.

In attempts to develop guidelines for the construction of software, the trend has been to restrict the programmer to an orderly and structured procedure in which a hierarchical development is emphasized. This is also described as a top down approach and is often referred to as structured programming.

This hierarchical approach applies both to the linguistic and to the actual structure of the program. For example, Brinch Hansen [9] , Dennis [6], and Dijkstra [20] have discussed linguistic hierarchies in which a program is developed at the topmost level as a

single procedure call that is meaningful to the programmer as representing the function to be performed. At the next level, this procedure is developed as another series of statements or procedure calls that are, again, especially meaningful to the programmer as directly representing the tasks to be performed. This process is carried down to the point where each statement consists of a primitive of the programming language. In turn, each primitive is developed in a hierarchical fashion through the language's implementation within the computer's operating system down to the basic operations of the machine itself. In conventional machines, this process can be viewed as establishing a hierarchy of virtual machines, each of which directly interprets the instructions of the layer above it in the form of instructions for the virtual machine in the layer below.

The overall design of a program, from the point of view of the set of tasks to be performed, can be represented as a network of components. Although this network could be a directed graph of aribitrary complexity, there exist some very compelling reasons to avoid this. Among them are:

1. Evidence of design correctness needs to be clearly presented.
2. Programs or program segments often require modification or maintenance. Therefore, program components should be as independent as possible from each other so that the consequences of any changes are as restricted as possible.
3. Design, production, and maintenance of software, to be manageable, often requires a team of people.

> Thus, to facilitate this approach, interdependencies must be minimized.

The result of reducing component interrelations is to structure the system into a partially ordered set of layers called "hierarchical ordering" [Bauer, 6].

The concurrent processability of a program may also be represented as a partially ordered graph displaying the precedence relations between components. Considerable theoretical work has been done to determine the precedence graph given a sequentially coded program and to determine the minimum number of processors required to complete the execution of the task in minimum time [Ramamoorthy, 37].

Each of these techniques involves a hierarchical description that is, on a single processor, simulated or stepped through sequentially. However, on a multiprocessor as described, this sequential simulation is no longer required. In the cases in which a virtual machine concept can be applied, a real machine can easily be used. In addition, although concurrency was not originally intended, this may often result.

B. Simple Example

These concepts, as applied to this architecture, are probably best illustrated by means of a simple example. The following program (which may be incorporated in a larger program) is to find the standard deviation of twelve numbers according to the formula.

$$S = \frac{1}{N}\left[N \sum_{i=1}^{N} X_i^2 - \left(\sum_{i=1}^{N} X_i\right)^2\right]^{1/2}$$

for N = 12

This example was chosen because it is likely to be familiar to most readers and is simple enough to be carried down to the level of individual machine instructions without becoming bogged down in useless detail.

Using PL/1 like notation, and employing a top down approach, the first version of the program is trivial. It would be adequate if the system used recognized STD as an inherent routine causing the standard deviation to be performed.

```
MAIN:PROCEDURE;
     DCL X (12) ;
          CALL STD(12,X,SD) ;
     END MAIN ;
```

It is seldom the case that STD would be recognized by the language used; therefore, we need to proceed to the next level and define it.

```
STD: PROCEDURE(N,XSD);
        DCL X(N);
            BEGIN;
                CALL SUM_X_SQRD(N,X,SXX);
                CALL SQRD_SUM_OF_X(N,SSX);
                NSXX = N*SXX;
                CALL SQRTDIFF(NSXX,SSX,ANS);
                SD = ANS/N;
            END STD;
```

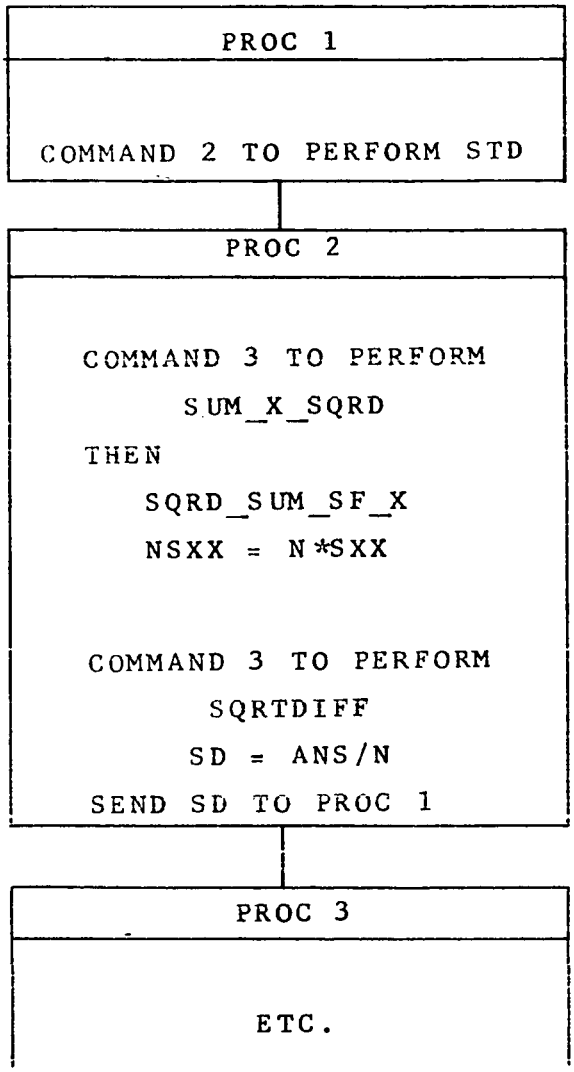
Here STD is shown to be made up of a series of steps. First we find the sum of X squared, then the squared sum of the X's etc. This again makes use of routines not within the language as well as basic elements of the language (ie., arithmetic assignment instructions). Again, another level is required to define SUM_X_SQRD, etc.

```
SUM_X_SQRD: PROCEDURE(N,X,SXX);
        DCL X(N);
        BEGIN;
            SXX = 0
            DO I = 1 TO N;
                SXX = SXX + X(I)**2;
            END;
            END SUM_X_SQRD;

SQRD_SUM_OF_X: PROCEDURE(N,X,SSX)
        DCL X(N);
            SSX = 0;
            DO I = 1 TO N;
```

```
            SSX = SSX + X(I);
        END;
    END SQRD_SUM_OF_X;

    SQRTDIFF: PROCEDURE(NSXX,SSX,ANS);
    BEGIN;
        ANS = NSXX - SSX;
        ANS = SQRT(ANS);
    END SQRTDIFF;
```

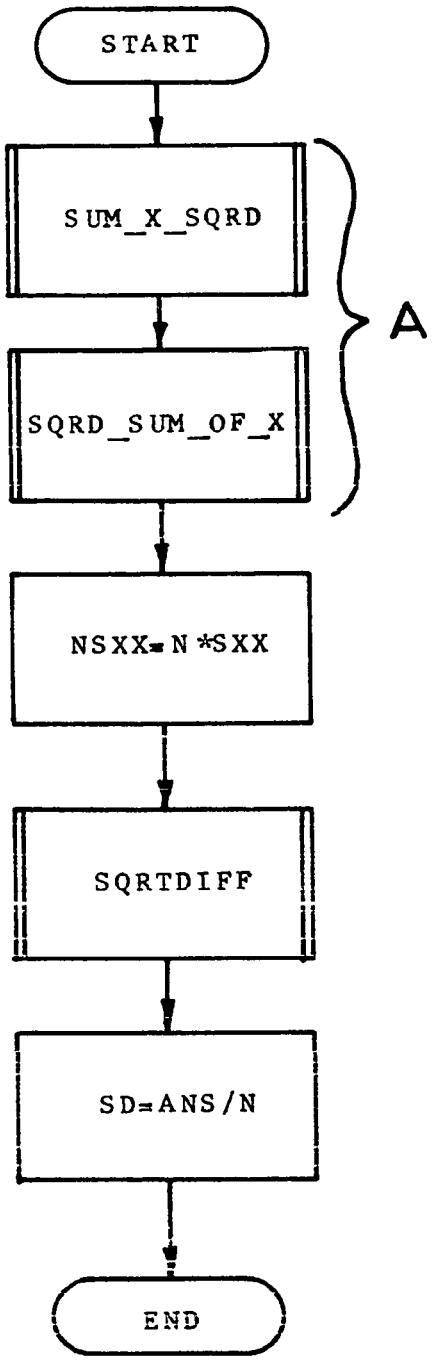
These routines illustrate the linguistic hierarchy employed within this program. The first level could be conceptually considered to be a virtual machine having an STD instruction. In turn, it could call upon a virtual machine of level 2 on which it performs the STD task by executing the "instruction" SUM_X_SQRD, etc., down to the real machine at the lowest level. In addition, each level could be implemented on a real processor in the system described earlier and as shown in Fig. VI.B.1.

We have now reached a situation in which we may exploit the possibilities of concurrency. Thus far, the program discussed has only been viewed as a simple sequential process with each step executed at the termination of the preceding step. A flow chart for the program is shown in Fig. VI.B.2 (not showing the hierarchical development). There is no interdependence between the first and second step of this flow chart nor any precedence relation between them. Therefore, this flow chart may be re-drawn as shown in Fig. VI.B.3. The sections labeled "A" in each flow chart are identical in function and have been programmed using the instruction set outlined in Table VI.B.1 and their execution times compared. Table VI.B.1 is for a hypotehetical processor whose architectural features, instructions, and execution times are



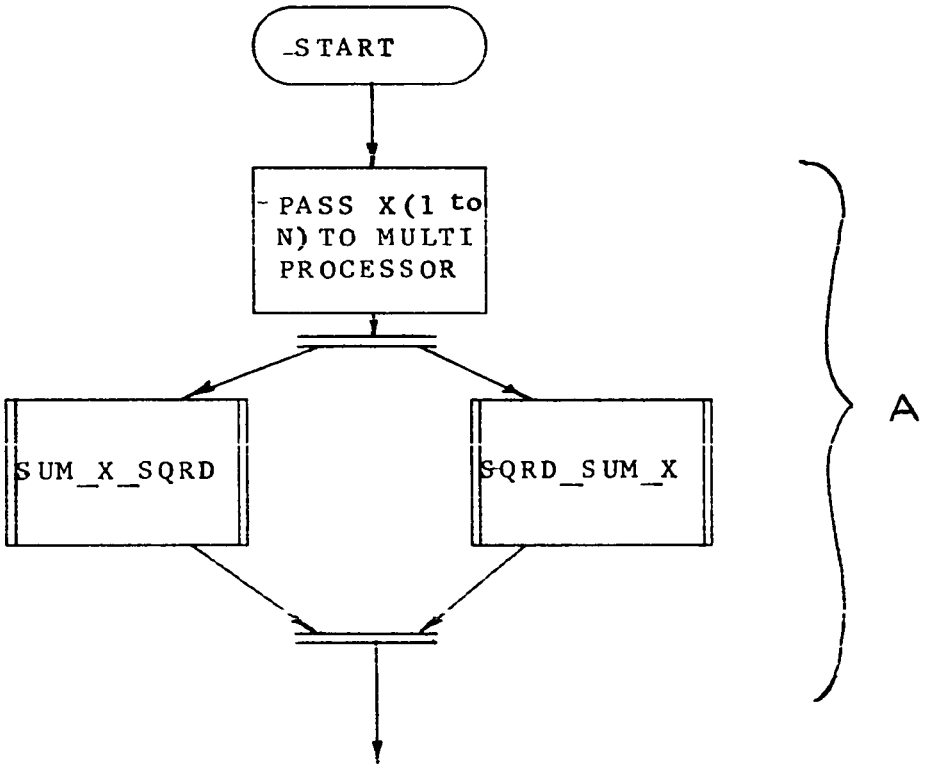
FUNCTIOAL HIERARCHY

FIG. VI.B.1



FLOW CHART FOR EXAMPLE

FIG. VI.B.2



MODIFICATION OF FIG. VI.B.2 FOR CONCURRENT ACTION

FIG. VI.B.3

TABLE VI.B.1 HYPOTHETICAL COMPUTER INSTRUCTION SET

MNEMONIC	FUNCTION	EXECUTION TIME
LDA mem loc	ACC A ← (mem loc)	5μs
LDB mem loc	ACC B ← (mem loc)	5μs
STR{A/B}, mem loc	ACC{A/B} → (mem loc)	5μs
PUSH{A/B}	ACC{A/B} → (mem loc_{SP}), SP-1→SP	4μs
POP{A/B}	ACC{A/B} ← (mem loc_{SP}), SP+1→SP	4μs
ADD{A/B}, mem loc	ACC{A/B} ← ACC{A/B} + (mem loc)	5μs
ADDC{A/B}, mem loc	ACC{A/B} ← ACC{A/B} + (mem loc) + C	
SUB{A/B}, mem loc	ACC{A/B} ← ACC{A/B} - (mem loc)	5μs
*1 MUL mem loc	ACC A,B ← A * (mem loc)	80μs
*2 INTERRUPT→SUBROUT		12μs

TABLE VI.B.1 HYPOTHETICAL COMPUTER INSTRUCTION SET (CONTINUED)

MNEMONIC	FUNCTION	EXECUTION TIME
RTI	Return from interrupt	10μs
ADD{A,B / B,A}	ACC A←ACC A + ACC B ACC B←ACC A + ACC B	2μs
BR	Branch (possibly on a test)	4μs
INX	Increment index register	1μs
INC mem loc	(mem loc) ← (mem loc) + 1	3μs
CLR mem loc	(mem loc) ← 0	6μs
CLR{A / B}	ACC{A / B} ← 0	2μs
LDX	(Index register) ← (mem loc)	5μs

NOTES

*1 - a multiply instruction seldom, if ever, appears in the instruction set of a microprocessor. This is included for convenience in these examples. The time for execution is an estimate based on a shift and add multiplication algorithm using the other instructions shown.

TABLE VI.B.1 HYPOTHETICAL COMPUTER INSTRUCTION SET (CONTINUED)

MNEMONIC	FUNCTION	EXECUTION TIME

NOTES (CONTINUED)

*2 - This is included to provide an indication of the time required between the occurrence of an interrupt and the beginning of the interrupt subroutine. All programmed processor activity is suspended during this interval.

(mem loc) - contents of the specified memory location

$\begin{Bmatrix} A \\ B \end{Bmatrix}$ - either accumulator A or B

@X - the contents of X are used as an address

XR - the index register

#0 - # indicates an immediate operand (ie., 0)

similar to those found in many commercially available microprocessors. This processor is assumed to have two accumulator registers, a stack pointer register, index register and a program counter. It is capable of one level of indirect addressing (generally through the index register). Fig. VI.V.4 is the assembly language routine for the single processor case. Fig. VI.B.5 is the equivalent routine for the multiprocessor case. The master processor has been named ⟨1,1⟩ and is assumed to contain all the values of X initially. The master proceeds by sending each value of X to a processor named ⟨5,1⟩ which calculates a running sum of the X's. Simultaneously, with the transmission of a value of X_i to ⟨5,1⟩, the same value is sent to a processor ⟨4,j⟩ where j = (i mod 4) + 1. Each processor ⟨4,XX⟩ squares each value of X it receives and maintains a running sum of X^2. As a consequence, the block labeled SUM_X_SQRD in Fig. VI.B.3 has been subdivided into four additional concurrent blocks. Processors ⟨5,1⟩ and ⟨4,XX⟩ each send their respective sums back to the master, ⟨1,1⟩, to continue processing.

FIG. VI.B.4 STANDARD DEVIATION ROUTINE FOR SINGLE PROCESSOR

MNEMONIC		FUNCTION	EXECUTION TIME	
		; SUM all X's		
	CLR A	; Clear ACC A, this will hold sum	2μs	
	LDX XPTR	; Place pointer to start -1 of the		
		; array of X's in XR	6μs	
A1:	INX	; Increment XR to point to first X	1μs	A
	ADD A, @XR	; add the X pointed to by XR to SUM	5μs	
	BR on XR ≠ END to A1	;	4μs	
	STR A, TEMP	; Place SUM in a temp location	5μs	
	LDX XPTR	; Store pointer to X's in XR again	6μs	
	CLR A	;	2μs	
	CLR B	; Set up for multiply	2μs	
	CLR M1	;	6μs	
	CLR M2	;	6μs	
B1:	INX	;	1μs	B
	LDA @ XR	; Get X	5μs	
	MUL @ XR	; Square X	80μs	
	ADD A, M1	; Accumulate SUM	5μs	
	ADDC B, M2	;	5μs	

FIG. VI.B.4 STANDARD DEVIATION ROUTINE FOR SINGLE PROCESSOR (CONTINUED)

MNEMONIC	FUNCTION	EXECUTION TIME
STR A, M1	; Place SUM back in	5μs
STR B, M2	; M1 and M2	5μs
BR on XR ≠ END to B1	;	4μs
Sections A & B executed N times for N values		
		35μs + N(120μs)

FIG. VI.B.5 STANDARD DEVIATION ROUTINE FOR MULTIPROCESSOR

MNEMONIC	FUNCTION	EXECUTION TIME
MASTER PROCESSOR - PROCESSOR ⟨1,1⟩		
	LDX XPTR ; Place pointer to X's in XR	6μs
A1:	INX ; Increment XR to point to Next X	1μs
	LDA AD(5,1) ; load A with the code for the bus addr 5,1	5μs
	LDB @ XR ; load B with value of X	5μs
	WAIT(FOR OUTBUF) ; Wait for the bus output buffer to empty	0*
	STR A, OUTBUF 1 ; Place 5,1 in destination addr part of bus	5μs
	STR B, OUTBUF 2 ; Place data in data section of bus word	5μs
	LDA AD(4,1) ; load A with 4,1	5μs
	WAIT(FOR OUTBUF) ; Wait for output buffer to empty from pre- ; vious operation	0*
	STR A, OUTBUF 1 ; etc.	5μs
	STR B, OUTBUF 2 ;	
	LDA AD(5,1) ; •	6μs
	INX ; •	1μs
	LDB @ XR ; •	5μs
	WAIT(FOR OUTBUF) ;	0*
	STR A, OUTBUF 1 ;	5μs
	STR B, OUTBUF 2 ;	5μs

FIG. VI.B.5 STANDARD DEVIATION ROUTINE FOR MULTIPROCESSOR (CONTINUED)

MNEMONIC	FUNCTION	EXECUTION TIME
MASTER PROCESSOR - PROCESSOR ⟨1,1⟩ (CONTINUED)		
LDA AD(4,2)	;	$5\mu s$
⋮	(Continued in the same sequence for AD(4,3), AD(4,4))	⋮
BR on XR $\neq$ end to A1		
		$6\mu s + \frac{N}{4}\overline{(144\mu s)}$

FIG. VI.B.5 MULTIPROCESSOR CASE

MNEMONIC	FUNCTION	EXECUTION TIME
	INTERRUPT ROUTINE FOR ALL PROCESSOR	
	INTRPT	12μs
	LDA INBUF	5μs
	PUSH A	4μs
	RETURN	10μs
	PROCESSOR ⟨4,X⟩	
A1:	POP A	4μs
	STA A M21 ; Square A	5μs
	MUL A, M21 ; Result Left in A&B	80μs
	ADD A, M11 ;	5μs
	ADDC B, M12 ; Double Precision Sum	5μs
	BR ON END TO B2 ;	4μs
B1:	BR on SP ≤ STACKSTART to A1	4μs

FIG. VI.B.5 MULTIPROCESSOR CASE (CONTINUED)

MNEMONIC	FUNCTION	EXECTUTION TIME
	PROCESSOR ⟨4,X⟩ (CONTINUED)	
	BR to B1	
B2:	; Send Result to ⟨1,1⟩	
	PROCESSOR ⟨5,1⟩	
	CLR B	2 *μs
A1:	POP A	4μs
	ADD A,B ; SUM A Result in B	2μs
	BR on END TO B2 ;	
B1:	BR on SP ≤ STACKSTART to A1	4μs
	BR to B1 ;	4 *μs
B2:	; Send Result to ⟨1,1⟩	

*These times do not influence the add time/item and therefore are not included in timing diagram, Fig. VI.B.7.

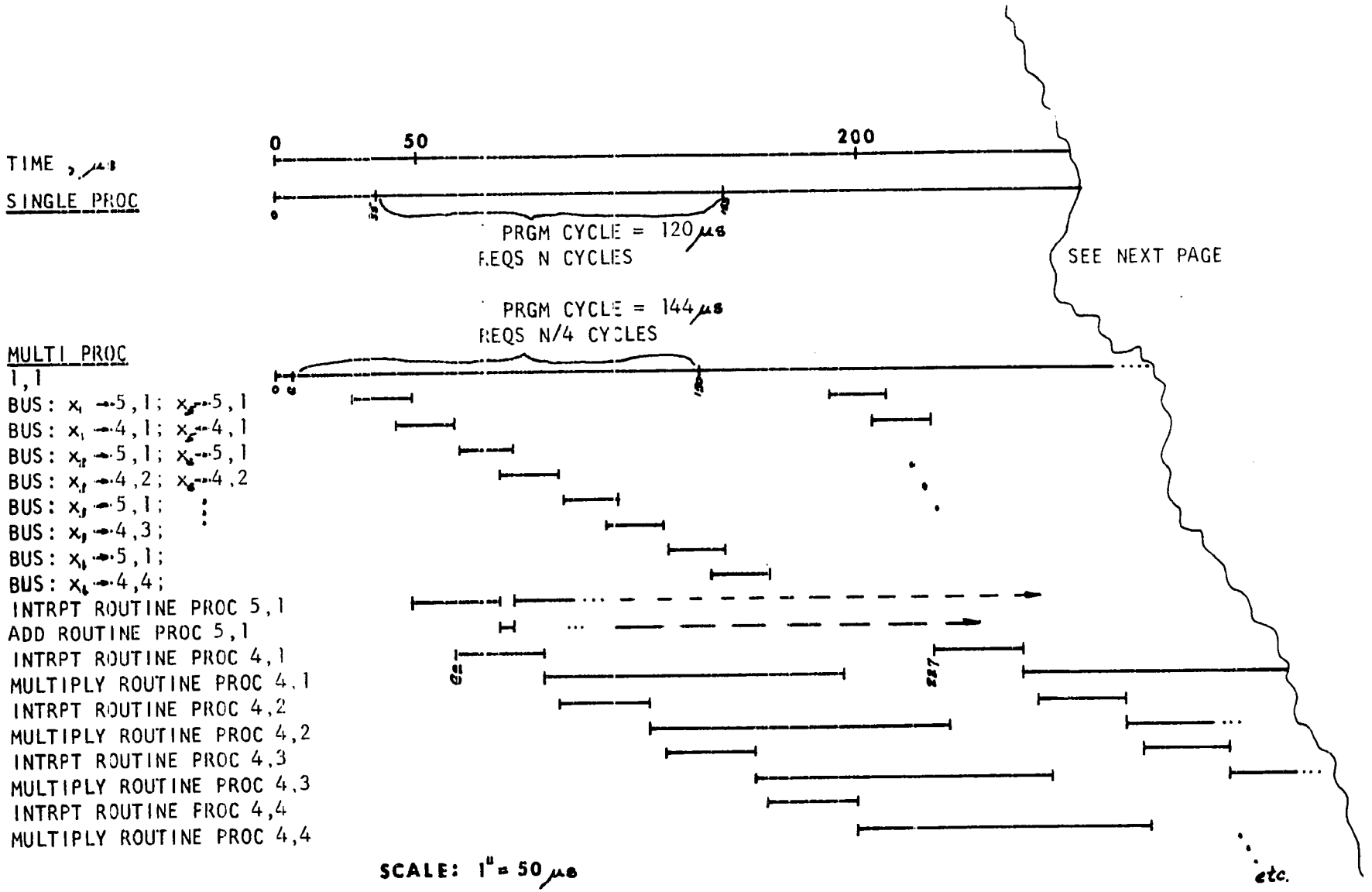
These routines are not intended to be complete, etc. but are to be indicative of the programming techiques and differences between the conventional monoprocessor and the multiprocessor that is the subject of this thesis. In addition these routines provide a basis for developing the timing diagram of Fig. VI.B.6. This figure is a timing diagram of each system's activity and allows the execution times for the flow chart sections A in Fig.'s VI.B.2 and VI.B.3 to be compared. As is easily seen, the single processor requires

$$t_s = 35\mu s + N(120)\mu s$$

to execute. Thus, to form the standard deviation of twelve numbers requires

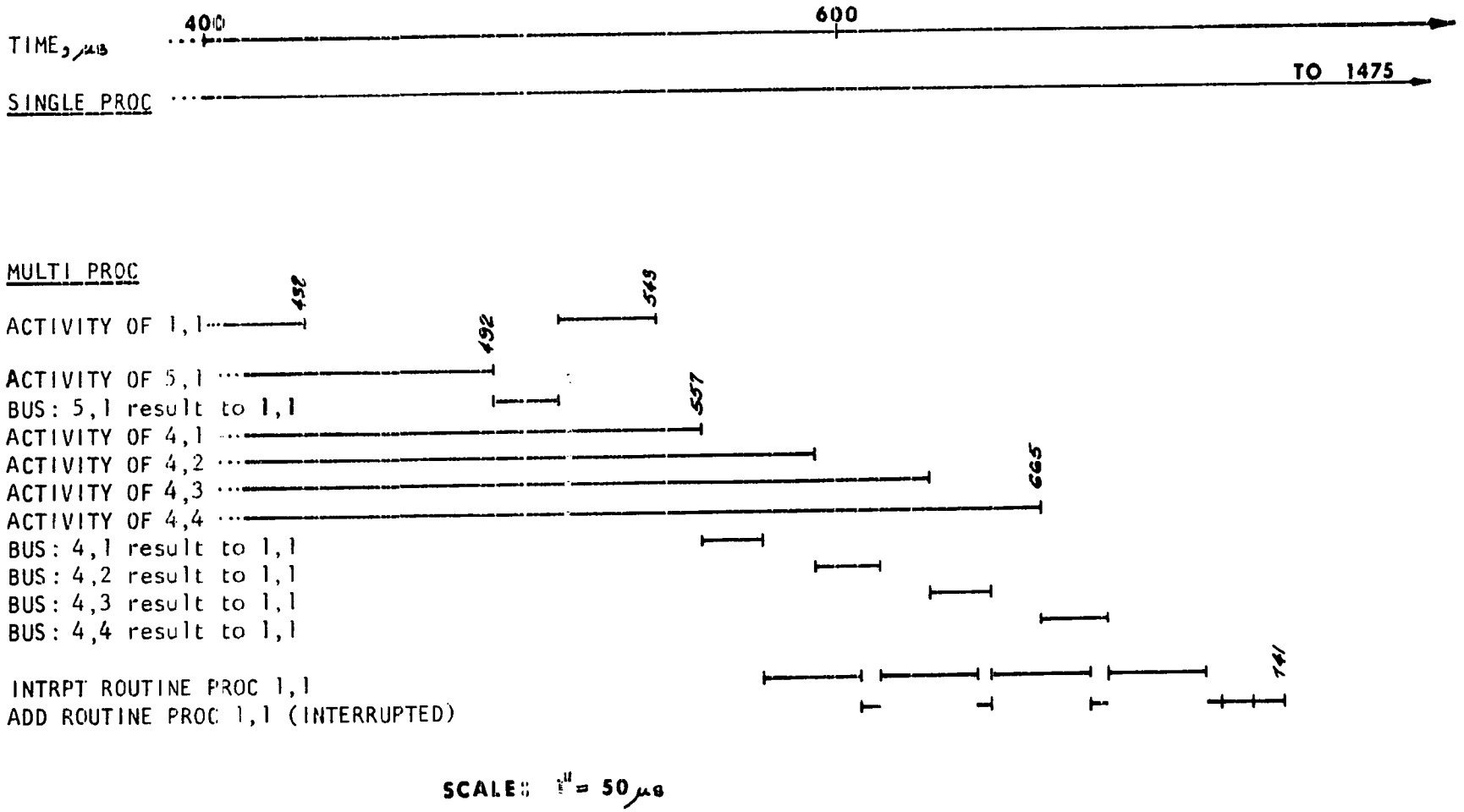
$$t_s = 35\mu s + 12(120)\mu s = 1475\mu s$$

For the multiprocessor case, it is assumed that each bus has a velocity of $\frac{1}{.2}$ μs of $5X10^6$ slots/sec. Since each processor requires 10μs to retrieve a word from memory and place it on a bus, $r = 0.1X10^6$ words/sec. This results in $\gamma = .02$ for each processor. From Section III on the analysis of the bus interface, it can be seen that for $\gamma = .02$, the expected normalized wait time for word will be less than one half of the processor's fetch-store time for bus utilizations of less than 0.96. For utilizations of less than .5, the total expected wait time is less than .4μs which is approximately zero with respect to the time of one program cycle. Therefore, the wait time is neglected. On the other hand, no assumption has been made concerning the placement of the processors around the bus. Assuming a total of 100 processors, the worst case would require 20μs of bus transit time for communication between two processors. (Note that this could easily be considered to absorb any wait time that could



TIMING DIAGRAM FOR FIG. VI.B.5

Fig. VI.B.6



TIMING DIAGRAM FOR FIG. VI.B.5 (CONTINUED)

Fig. VI.B.6

not be neglected). On the other hand, processors would generally be assigned as nearly adjacent to one another as possible; thereby, reducing the required 20μs to the order of 1μs etc., for this example. (Note that the bus is a pipeline-like device; once it is filled, items come out at the rate items are entered; thus, using 1μs or 20μs for the bus transit time will make an insignificant difference for this example. See Fig. VI.B.6).

Using the worst-case bus transit time of 20μs, it can be seen in the timing diagram of Fig. VI.B.6, the multirprocessor configuration requires

$$t_m(1,1) = 6 + \frac{N}{4}(144) + 2(31)\mu s$$

for the operation of the master processor. Processors ⟨5,1⟩, and ⟨4,1⟩, ⟨4,2⟩, ⟨4,3⟩, ⟨4,4⟩ require

$$t_{m(5,1)} = N(31 + 10)\mu s$$

and

$$t_{m(4,x)} = \frac{N}{4}(31 + 103)\mu s$$

respectively. Using this organization, the total time required for forming the standard deviation of twelve numbers is 741μs. This represents an improvement by a factor of 2 over the single processor case.

Although it is not reasonable to include the programming or loading time in a discussion of program execution times, it might be argued that for a system of this nature, the structuring time is a legitimate part of the system execution overhead and should therefore be included. Fig. VI.B.8 illustrates the type of structuring process that might precede this sample program.

As can be seen this will add an additional 300μs

FIG. VI.B.7 STRUCTURING ROUTINE FOR MULTIPROCESSOR

Assume that all processors have been loaded with all the pertinent routines from Fig. VI.B.6. Each buffer register is 32 bits wide, broken into 4 fields of 8 bits indicated by the suffix to the register name. Xmit is automatic after field 4, the DATA/CMD field is written. Field 1 is the destination address; Field 2 is the processor's name and is always loaded with its V-Name; Field 3 is the operation no. The system is assumed to be totally quiescent at the start of this routine.

MNEMONIC	FUNCTION	EXECUTION TIME
CLR A	;	6μs
STR A CØOUTBUF 3	; Set OP # = 0	5μs
STR A DOUTBUF 3	; Set OP # = 0	5μs
LDA CMD1	; Load ACC A with code for command 1 ie. to activate the BUS SHORT module on C.G. [0]	5μs
LDB #last syst Proc no.	; Load ACC B with a constant that is the number of the last syst proc.	5μs
STR B CØOUTBURF 1	; Place that # in dest addr field of C.G. [0] CMD bus output reg.	5μs

FIG. VI.B.7 STRUCTURING ROUTINE FOR MULTIPROCESSOR (CONTINUED)

MNEMONIC	FUNCTION	EXECTUTION TIME
STR A CØOUTBUF 4	; Places CMD code in field 4	5μs
LDB #6	; and automatically Xmits packet	
WAIT(CØ)	; Wait for output reg for CMD bus	~0μs
	; to empty	
STR B CØOUTBUF 1	; Send CMD 1 to proc #6	5μs
STR A CØOUTBUF 4	; This has chopped off a section of	
	; C.G. [1] with proc's 0,1,....6	
	; on it	
LDA CMD2	; Load ACC A with command when 2	5μs
	; ie. to set V-Name to value rec'd	
	; from DATA bus	
WAIT(CØ)	;	~0μs
INC CØOUTBUF 3	; Set OP # = 1	3μs
INC DOUTBUF 3	; Set OP # = 1	3μs
LDB UNIVNAME	; Place univ name code in ACC B	5μs
WAIT(CØ)	;	~0μs
STR B CØOUTBUF 1	; Set up dest addr	5μs
STR A CØOUTBUF 4	; Place CMD code in field 4	5μs
	and automatically Xmit	

FIG. VI.B.7 STRUCTURING ROUTINE FOR MULTIPROCESSOR (CONTINUED)

MNEMONIC		FUNCTION	EXECUTION TIME
		; Sends CMD2 to proc's 0-6	
	LDX ADPTR	; ADPTR is a pointer into a table	6μs
		; of V-Names	
	LDB #0	; for this problem	5μs
A2:	LDA @ XR	; This loop sends V-Name for each proc	5μs
	STR B, DOUTBUF1	; loads dest field	5μs
	STR A, DOUTBUF4	; loads DATA field	5μs
	INC B	;	
	INX	;	1μs
	BR on B≤6 to A2	;	4μs
	LDA CMD3	; Go COMMAND	5μs
	LDB #(Code(5,1)	;	5μs
	WAIT(CØ)	;	~0μs
	STR B CØOUTBUF1	;	5μs
	LDB #2	;	5μs
	STRB CØOUTBUF3	;	5μs
	STR A CØOUTBUF4	; Start operation 2 in ⟨5,1⟩	5μs
	LDB #Code(4,X)	;	5μs

FIG. VI.B.7 STRUCTURING ROUTINE FOR MULTIPROCESSOR (CONTINUED)

MNEMONIC	FUNCTION	EXECUTION TIME
WAIT(CØ)	;	~0μs
STR B CØOUTBUF1	;	5μs
LDB #3	;	5μs
STR B CØOUTBUF3	;	5μs
STR A CØOUTBUF4	; Start operation 3 in proc's of block 4	5μs
	; ie. ⟨4,X⟩	
LDB # Code(1,1)	;	5μs
WAIT (CØ)	;	~0μs
STR B CØOUTBUF1	;	5μs
LDB #1	;	5μs
STRB CØOUTBUF3	;	5μs
STRA CØOUTBUF4	; Start operation 1 in proc ⟨1,1⟩	5μs

to processor ⟨1,1⟩'s task. Assuming that this was incorporated in the worst possible way (ie., tacked onto the first of 1,1's routine) as opposed to interleaving it in the appropriate places within ⟨1,1⟩'s routine to take advantage of the pipeline effect of the bus, etc., the total execution time for the multiprocessor would be

$$t_m = 741\mu s + 300\mu s = 1041\mu s$$

and would represent an increase in speed of 1.42 over the single processor.

From the discussion of the improvement factor in Section III, it is clear that as the number of data items in the standard deviation increases (and hence the execution time in the single processor case) the improvement resulting from use of this multiprocessor architecture also increases. By the time the number of items reaches one hundred, as used in this example, the multiprocessor has a speed advantage that has leveled off at approximately 3. The multiprocessor could also have been configured using additional processor modules to obtain a more dramatic speed advantage. This would not have served the purpose of providing a simple, brief example, however.

C. Firmware/Special System Routines

Thus far, any application of specialized instructions has been avoided to illustrate the utilization of the system using conventional microprocessors and conventional instruction sets of limited capability. However, to most effectively manage the multiple processor system, it is desirable to incorporate several special instructions, system macros or system subroutines especially adapted to this purpose. Assuming the basic instruction set is of a conventional, general purpose type as is found in common micro- and minicomputers, the following type of modifications or additions may be desired.

1. Multiword, Memory To Bus Register Move

For an eight bit machine having its I/O, peripherals, etc., mapped into its normal address space (eg., like the Motorola 6800) and having 32 bits busses, this instruction might take the form:

MV4 EA1, EA2

Move four contiguous bytes beginning at the effective address, EA1, to the four locations beginning at effective address, EA2. This will allow an entire bus register to be filled with one instructions, etc.

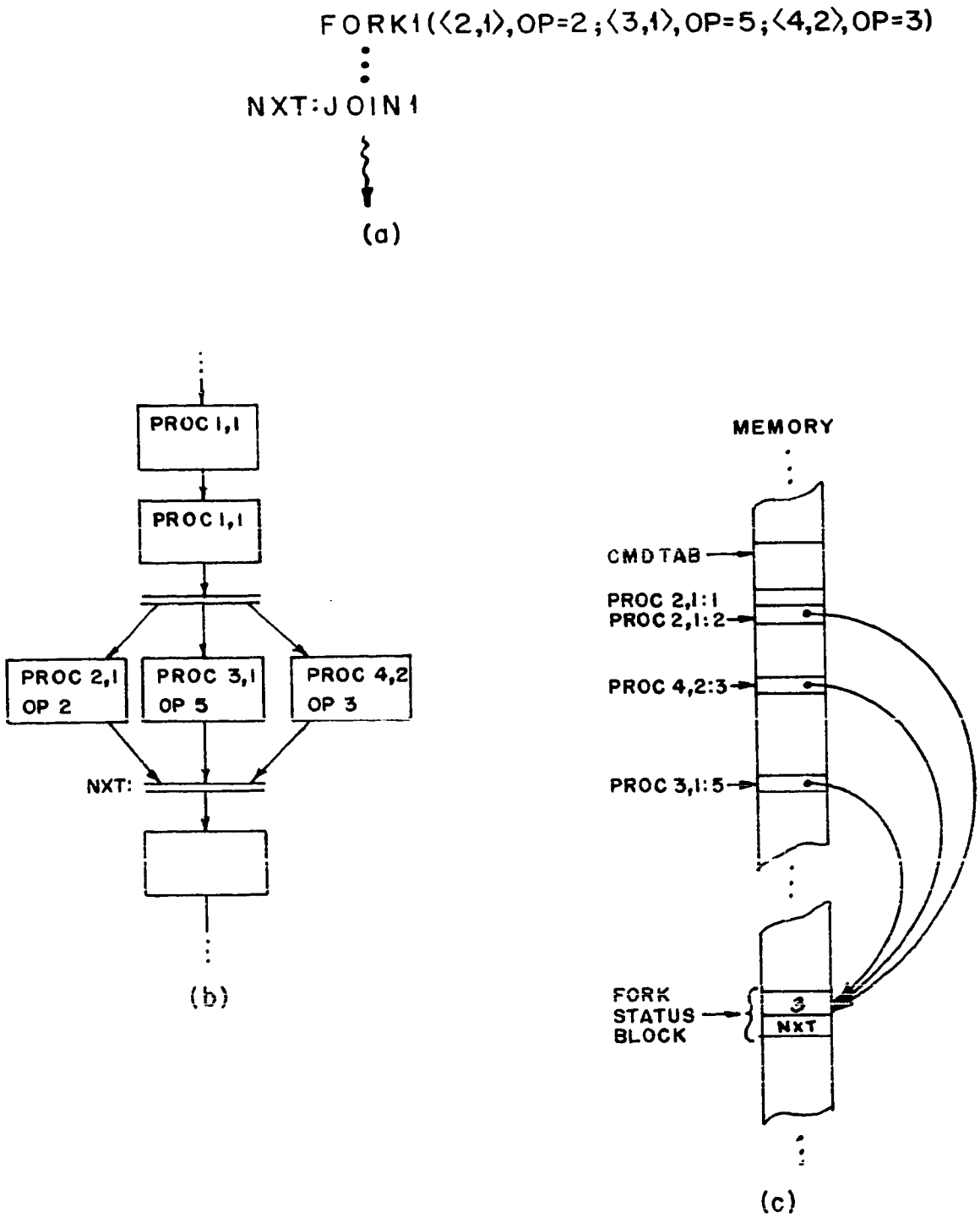
2. Interrupt Processing

During interrupt processing, the first action by the processor must be the setting of the HOLD FLIP FLOP. The processor is then required to wait until it sees the HOLD ACKNOWLEDGE LINE go low. Conventional interrupt processing can then proceed. Upon execution of a return

from interrupt (RTI), the last microprogrammed execution of a return from interrupt (RTI), the last microprogrammed action is to reset the HOLD FLIP FLOP.

3. Clear Mode Addressing

The application for this addressing mode is in the maintenance of a Command Table. Each processor assigned as a master of one or more subordinate processors sends out commands. It must therefore keep track of the status of the operations active in subordinate processors to ensure that a DONE is eventually received for every operation/command it sends out. It must also avoid sending a command with a particular operation number before a previous operation in the same processor with the same operation number is finished. Likewise, when a processor executes a FORM operation spawning processes in subordinate processors, records must be kept allowing the master to ensure that all spawned processes have terminated before resuming action beyond the JOIN instruction. An effective means of maintaining these records is in the form of a table (eg., hash table) of information versus processor numbers and operation number. Indirect and clear mode addressing can be useful in the maintenance of this table. For example, consider a FORK and JOIN sequence. The FORK spawns N independent concurrent processes in N processors. Each possible processor name plus operation number is represented as an empty location in the table, CMDTAB, shown in Fig. VI.C.1. Assume the processes spawned are in processors ⟨2,1⟩, ⟨3,1⟩, ⟨4,2⟩, etc., and are referred to as operations numbered 2, 5, 3, etc., in each processor, respectively. As the FORK begins execution, the entries in CMDTAB for these operation numbers, in their respective processors, are zero. The master processor



ACTION OF FORK AND JOIN

Fig. VI.C.1

issues commands sequentially to each subordinate under the subordinate's respective operation number and enters a pointer to the FORK STATUS BLOCK in CMDTAB at the appropriate position. In addition, the master increments the value contained in the first word of the FORK STATUS BLOCK. This word, initially zero, will contain the number of arcs leaving the FORK (or the number of processors spawned). After the FORK is completed, the master waits for DONE's to be received from each operation and processor containing a spawned task. As each DONE is received from any arbitrary processor, the processor name and operation number are hashed to find their position in CMDTAB. The master will then access indirectly through this position and decrement the first word of the FORK STATUS BLOCK. If this results in a zero value, execution will continue from the address contained in the second word of this status block. Using Clear Mode Addressing when obtaining the address of the FORK STATUS BLOCK after a DONE is received will automatically clear the entry in CMDTAB signifying that the operation and processor are free to be asigned again.

4. EALU Requirements

When a group of modules has been organized as an EALU, the group will appear as a wider processor in most respects. However, with regard to the computation of addresses for storage and retrieval of data and instructions, this is not acceptable. Each system module in an EALU still has a memory independent from its companions. Although the address for each module will be calculated simultaneously and with the same process, they must be independent with no carry between modules. Therefore, when an effective address is being calculated, the system modules must be switched out of the CARRY LOOP.

(This does not imply removal from the FUNCTION LOOP). This then requires that carries, etc., from an EALU be given back to the module's own microprogram controller during address calculation. Each driven module's controller will still be in a "wait for interrupt" mode as discussed earlier. However, it must also monitor the ALU's carry line, etc., to check for overflow or errors in calculating addresses. If an error is detected, it will be handled as though it were an interrupt as discussed in Section IV.

D. Example Using EALU's

A second more complex example may be useful in illustrating the more advanced techniques and applications of this architecture.

A set of simultaneous ordinary differential equations can always be reduced to a set of first order ordinary differential equations by the introduction of auxiliary dependent variables. For example,

$$y'' + 2yy' - 8y = x^2$$

can be reduced to a set of two first order equations by letting

$$y' = u$$

Then

$$y'' = u'$$

and

$$y' = u$$

$$u' - f(x,y,u) = x^2 - 2yu + 8y$$

Assuming that the general set of equations produced in this manner,

$$y_1' = f_1\ (x, y_1, \ldots, y_M)$$

$$y_2' = f_2\ (x, y_1, \ldots, y_M)$$

$$\vdots$$

$$y_M' = f_M\ (x, y_1, \ldots, y_M)$$

has a set of initial conditions

$$(x, y_1, \ldots, y_M)_0$$

specified at one point, a solution can be obtained by numerical integration using the Runge-Kutta method. For additional discussion of this procedure see Southworth [52]. Using a uniform step size, h, the following steps yield the solution points $y_{1,n}, y_{2,n}, \ldots, y_{M,n}$ for $n = 0,1,\ldots,N$, for each of the N intervals of x:

Step 1:
$$K_{1,1} = hf_1(x_n, y_{1,n}, \ldots, y_{M,n})$$

$$K_{2,1} = hf_2(x_n, y_{1,n}, \ldots, y_{M,n})$$

$$K_{M,1} = hf_M(x_n, y_{1,n}, \ldots, y_{M,n})$$

Step 2:
$$K_{1,2} = hf_1\left(x_n + \frac{h}{2}, y_{1,n} + \frac{k_{1,1}}{2}, \ldots, y_{m,n} + \frac{k_{m,1}}{2}\right)$$

$$K_{2,2} = hf_2\left(x_n + \frac{h}{2}, y_{1,n} + \frac{k_{2,1}}{2}, \ldots, y_{m,n} + \frac{k_{m,1}}{2}\right)$$

$$K_{M,2} = hf_M\left(x_n + \frac{h}{2}, y_{1,n} + \frac{k_{m,1}}{2}, \ldots, y_{m,n} + \frac{k_{m,1}}{2}\right)$$

Step 3: $$k_{m,3} = hf_m(x_n + \frac{h}{2}, y_{1,n} + \frac{k_{1,2}}{2}, \ldots, y_{m,n} + \frac{k_{m,2}}{2})$$

for m = 1 to M

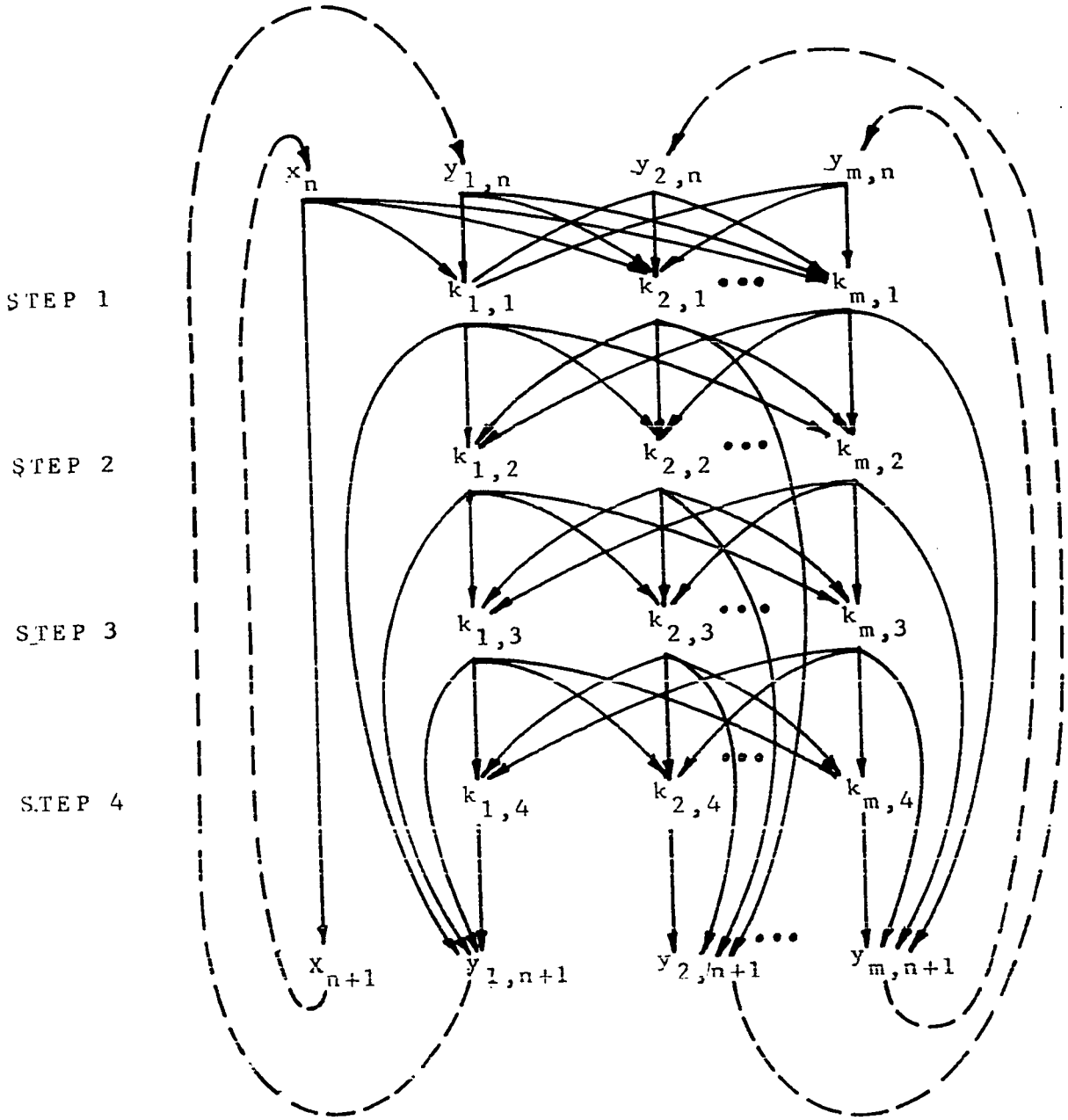
Step 4: $$K_{m,4} = hf_m(x_n 1+h, y_{1,n}+k_{1,3}, \ldots, y_{m,n} + k_{m,3})$$

$$y_{m,n+1} = y_{m,n} + \frac{1}{6}(k_{m,1} + 2k_{m,2} + 2k_{m,3} + k_{m,4})$$

for m = 1 to M

$$x_{n+1} = x_n + h$$

This results in the precedence graph shown in Fig. VI.D.1. Note that each of the operations at each level of the precedence graph are independent of all other operations on the same level. Each vertical column represents those operations pertaining to one of the M equations. The approach taken for this algorithm will be to divide the work between M EALU's. Each EALU will be responsible for one column and will contain four, data driven routines (or operations) for the four steps respectively.



PRECEDENCE GRAPH FOR RUNGE-KUTTA ALGORITHM

Fig. VI.D.1

A program for this architecture written in PL/1 like notation would appear as indicated below. This program will perform Runge-Kutta integration of M simultaneous equations using one level of hierarchy and three word EALU's providing 24 bits of arithmetic precision. Typical basic system routines required are included at the end of the example routine.

Processor 0 is assumed to have been designated by the operator to be the system master and contains the bootstrap that loads and begins execution of the following routine.

```
PROCESSOR 0
     PROC_0  :PROCEDURE MAIN;
```

This routine structures the system, transfers the program for each subordinate from memory to each processor and states each subordinate.

```
          VNAME = 0,0;
```

This sets the processor V-name to 0,0

```
/*                    LEVEL  ;    P-NAME  ; OP NO ; CMD  */
          CALL COUT  (0     ,     ⟨N⟩    ,   1   ,  31);
          CALL COUT  (0     ,     ⟨M*3⟩  ,   1   ,  31);
```

This routine outputs to the specified command bus buffer register; any necessary waiting is provided by this routine.

⟨N⟩ is the largest P-name in the system and is physically located to the immediate left of Proc 0

31 is the command code to cause the destination processor to activate its BUS SHORT module on level 1.

M is the number of equations (defined elsewhere)

```
DO I = 1 TO M * 3;
     CALL COUT (0,   ⟨1⟩  ,   1   ,  28) ;
     CALL DOUT (     ⟨1⟩  ,   1   ,  NAME(I);
     END;
```

These statements send command 28 as operation 1 to each processor. CMD 28 is to be the command to set the processor V-name to the value received on the data but for operation 1

NAME is a vector containing the following V-names, ⟨1,1⟩; ⟨1,2⟩; ⟨1,3⟩; ⟨2,1⟩;...,⟨M,1⟩; ⟨M,2⟩; ⟨M,3⟩

```
CALL COUT (0,   ⟨XX⟩,  2   ,   27) ;
CALL DOUT (     ⟨XX⟩,  2   ,    1) ;
```

Command 27 requires the receiver to set its priority to the value received on the data bus - in this case 1. ⟨XX⟩ represents the universal name;hence, all processors will receive this command.

CALL WAIT

Wait for all processors to reply that they have successfully completed all outstanding tasks

```
DO WHILE MORE_PRGM;
     CALL COUT (0 , ⟨XX⟩,  0,   26);
     DO J=1 TO 255 WHILE NOT_LAST_ITEM;
     CALL DOUT (    ⟨XX⟩,  J,  PROG_WORD(I));
     END
END;
```

Command 26 requires a processor to accept the program being sent over the data bus. This program is in the form of a load module broken into blocks of 255 words or less. The individual words are ordered by using the Data Bus Operation/Sequence number field. Therefore, CMD 26 can only be sent to processors having no outstand-

ing tasks; it is sent only as operation 0; an error would be returned otherwise

```
CALL COUT (0 , ⟨XX,1⟩ , 0  , 25);
CALL COUT (0 , ⟨XX,2⟩ , 0  , 24);
CALL COUT (0 , ⟨XX,3⟩ , 0  , 23);
```

Command 25 - set status to driving module of EALU
24 - set status to driven module of EALU
23 - set status to low order driven module of EALU

```
CALL WAIT
```

Wait for all processors to finish

```
CALL COUT (0 , ⟨XX,1⟩ , 0  , 22);
CALL DOUT (    ⟨XX,1⟩ , 1  , 0  , x_o);
CALL DOUT (    ⟨XX,1⟩ , 2  , 0  , y_o,1);
CALL DOUT (    ⟨XX,1⟩ , 3  , 0  , y_o,2);
                .
                .
                .
CALL DOUT (    ⟨XX,⟩  , M+1, 1  , y_o,M)
```

Command 22 - accept a block of data on data bus, ordered by sequence no (operation no) a 1 in the I/O field (3rd field) indicates last item.

```
CALL COUT (1 , ⟨XX,1⟩ , 0  , 00);
```

Command 00 - "Go", equivalent to pressing a start key in each processor

```
END PROC_0;
```

```
WAIT:  PROCEDURE;
```

This routine checks to see of all the commands it has sent out to subordinate processors have replied DONE.

If not, it waits until all the DONE's are received. It accomplishes this check by observing the entries in CMDTAB for each subordinate processor.

```
AGAIN:  DO I = 1 TO M * 3;
        IF CMDTAB(I).NOT CLEAR.  THEN WAITING = .TRUE.;
        END;
        IF WAITING THEN GO TO AGAIN;
        END WAIT;
```

.NOT CLEAR. should be taken as a special built in logical operation used to detect any non-zero entry in the table CMDTAB.

Each of the M subordinate EALU's (addressed ⟨1,1⟩, ⟨2,1⟩, ..., ⟨M,1⟩, etc.) execute the following routines:
PROCESSOR m,1 for m= 1,2,...,M

```
MAIN: PROCEDURE
      DO I = 0 TO 256;
           ENABLE (I) = .FALSE.;
           END;
```

This is a power up initialization type sequence to ensure that the system starts in a reset state. ENABLE is a global control vector of length 256 (256 possible operation no's). Its status is changed by the interrupt routine that receives commands.

```
AGAIN:
      DO I = 0 TO 256;
           IF ENABLE(I) THEN CALL OP(I);
           END;
      GO TO AGAIN;
Wait for receipt and activation of a command
      END MAIN;
```

The following routine is command to all subordinate processors and is typical of the interrupt routines throughout the system. This routine allows each of the OP routines to be data driven.

```
     INTERRUPT_D:  PROCEDURE;
```
This routine accepts data from the bus and provides it to the processor's operations according to the operation number and originator. Its call is initiated by an interrupt occurring in the DATA bus interface.

```
     HOLD = 1;
```
Requests independence from EALU to service interrupt by setting the HOLD FLIP FLOP to 1.

```
     WAIT:  IF HOLD_ACK = .FALSE.  THEN GO TO WAIT:
```
Wait for independence to be granted as indicated by the status of the HOLD ACKNOWLEDGE LINE.

```
          CALL DIN ( SENDER, OPNO, DATA) ;
          GO TO OP(OPNO) ;
```
OP is a label vector containing starting labels for each routine, ie. OP1, OP2, OP3, OP4. DIN is a routine for obtaining the contents of the INPUT DATA BUFFER.

```
          OP1: BEGIN;
               DECLARE NO_OF_ITEMS;
               NO_OF_ITEMS = NO_OF_ITEMS + 1;
               Y(SENDER) = DATA;
               IF NO_OF_ITEMS = M THEN DO NO_OF_ITEMS = 0;
               WAIT_FOR_1 = .FALSE.;
               END;

               GO TO EXIT;

          OP2: BEGIN;
               DCL NO_OF_ITEMS;
               NO_OF_ITEMS = NO_OF_ITEMS + 1;
               K(SENDER, 1) = DATA;
               IF NO_OF_ITEMS = M THEN DO;
                  NO_OF_ITEMS = 0
                  WAIT_FOR_2 = .FALSE.;
```

```
        END;

    GO TO EXIT;
OP3: (SIMILAR TO OP2 AND OP1)
          .
          .
          .
EXIT: HOLD  = 0;
```

Reset HOLD FLIP FLOP as last function of interrupt routine

The following routines are executed by the driving module in each EALU and actually perform the Runge-Kutta algorithm. Each routine performs one step of the algorithm for one of the M equations. The routines are each data driven, performing no function until their appropriate data arrives. The arrival of the data is signaled by the global variables, WAIT_FOR_... These variables are set by the interrupt routine servicing the data bus.

```
PROCESSOR m, p      m = 1 to M; p = 1 to 3

OP1: PROCEDURE;
     IF WAIT_FOR_1 THEN GO TO EXIT;
     ELSE DO;
          X = X + H;
          K(m,1) = H*F_m(X,Y(1), Y(2),...,
          Y(M);
          CALL DOUT(⟨XX,ELEMENT⟩,2,K(m,1));
          CALL DOUT(⟨XX,ELEMENT⟩,4,K(m,1));
          WAIT_FOR_1 = .TRUE.;
          END;
```

```
EXIT: END OP1;
```

The parameters in the output routine deserve some discussion at this point. This procedure is to drive an EALU. Each module in the EALU will then independently determine the address specified by the variable ELEMENT and fetch its contents. In each case, ELEMENT's contents will be the particular processor's element portion of its V-name, ie. 1, 2, or 3. Thus, when the data is moved to the output register, each output register will be loaded with its processor's own element name in the element field. Therefore, each processor will send its data to all other modules at its same level of significance in the extended word. As a consequence, the bytes making up an EALU word will not get scrambled during transfer from one EALU to another.

```
OP2: PROCEDURE;
     IF WAIT_FOR_2 THEN GO TO EXIT;
     ELSE DO;
     K(m,2) = H*F_m(X+H/2,Y(1)+K(1,1)/2,...,
          Y(M)+K(M,1)/2)
          CALL DOUT(⟨XX,ELEMENT⟩,3,K(m,2))
          CALL DOUT(⟨XX,ELEMENT⟩,4,K(m,2))
          WAIT_FOR_2 = .TRUE.
          END;
EXIT: END OP2;

OP3: PROCEDURE;
     IF WAIT_FOR_3 THEN GO TO EXIT;
     ELSE DO;
     K(m,3) = H*F_m(X+H/2,Y(1)+K(1,2)/2,...,
          Y(M)+K(M,2)/2)
          CALL DOUT(⟨XX,ELEMENT⟩,4, K(m,3))
          WAIT_FOR_3 = .TRUE
          END;
```

```
        EXIT: END OP3;

        OP4: PROCEDURE;
             IF WAIT_FOR_4 THEN GO TO EXIT;
             ELSE DO;
             K(m,4) = H*F_m(X+H,Y(1)+K(1,3),...,Y(M)
                  +K(M),3));
             Y(m) = Y(m)+K(m,1)+2*K(m,2)+2*K(m,3)+
                  K(m,4))/6;
                  CALL DOUT(⟨XX,ELEMENT⟩,1,Y(m));
                  WAIT_FOR_4 = .TRUE.;
                  END;
         EXIT: END OP4;
/** END OF RUNGE-KUTTA ALGORITHM **/
```

Typical basic systems routines are included below. These routines are not necessarility complete but indicate the necessary steps required by the system architecture.

The following routine performs the output function from the processor to a command bus specified in the parameter string. It would normally be considered to be a part of the basic operating system and common to all processors.

```
     COUT: PROCEDURE (LEVEL, NAME, OPNO, CMD);
           DCL 1 OUTPUT_BUFFER (number ofC.G.'s)
                 2 NAME;
                   3 BLOCK;
                   3 ITEM;
                 2 OPNO;
     WAIT:  IF OUTPUT_BUFFER_FULL(LEVEL) = 1 THEN GO TO WAIT;
```

OUTPUT_BUUFER_FULL represents the status register flip flips for each of the bus output registers. LEVEL specifies the particular bus and therefore the particular flip

flop.

When the buffer full flip flop is reset by the bus interface load the buffer with the item to be output.

```
OUTPUT_BUFFER(LEVEL).NAME = NAME;
OUTPUT_BUFFER(LEVEL).OPNO = OPNO;
OUTPUT_BUFFER(LEVEL).CMD = CMD;
```

When the last element is placed in the buffer, the interface is automatically released to attempt to place the contents of the buffer onto the bus.

Since a command has been issued, that fact must be recorded in CMDTAB to wait for ACK and DONE.

```
CMDTABINDEX = HASH(NAME,OPNO);
```

HASH represents a built in function that hashes each unique NAME, OPNO combination to a unique address in CMDTAB.

```
CMDTAB(CDMTABINDEX) = 1;
EXIT: END COUT;
```

Every command sent out is acknowledged on the DONE bus by the subordinate processor. Therefore, the superior processor must have an interrupt routine to service the ACK.

```
INTERRUPT_DONE:  PROCEDURE;
        HOLD = 1;
WAIT:  IF HOLD_ACK = .FALSE.  THEN GO TO WAIT;
        CALL DONE_IN (SENDER, OPNO, CODE);
        IF CODE = NEG_ACK THEN CALL ERROR;
        ELSE IF CODE = DONE THEN GO TO DN;
           ELSE IF CODE≠POS_ACK THEN CALL ERROR;
POS_AK:  INDEX =HASH (SENDER, OPNO);
        IF CMDTAB(INDEX) ≠ THEN CALL ERRO;
```

If the CMDTAB entry is not a 1 signifying that a com-

```
mand has been issued and an ACK is expected then an
error has occurred.  Error is to be an error recovery
routine.
            CMDTAB(INDEX) = 2;
Update the entry to indicate a DONE is now expected.
            GO TO EXIT;
            DN:   INDEX = HASH(SEND,OPNO);
                  IF CODE = DONE_ERROR THEN CALL ERROR;
                  IF CMDTAB(INDEX)≠2 THEN CALL ERROR;

                  CMDTAB(INDEX) = 0;
Clear CMDTAB entry indicating that the operation is com-
plete, etc.

                  EXIT; END INTERRUPT-DONE;
```

The following routines are typical of those routines servicing each of the Control Groups for the system.

```
            INTERRUPT_CO: PROCEDURE;
Interrupt routine for C.G. [0] CMD bus

            HOLD = 1;
Set the processors HOLD F.F. to 1 to request independence
from EALU

      WAIT: IF HOLD_ACK = .FALSE.  THEN GO TO WAIT;
Wait for independence to be granted.

            CALL CIN(0,   SENDER , OPNO , CMD);
Get the contents of the INPUT BUFFER REGISTER for C.G.
[0].  CIN automatically ACK's the command.

            GOTO CMD_ROUTINE( CMD );
Branches to execute routine specified by CMD.  Large
```

values are system routines, small values are user defined (except 00-"Go"). CMD_ROUTINE is a label array containing starting labels for each routine.

```
            .
            .
            .
/*  CMD 00  */

          GO:  ENABLE (OPNO) = .TRUE.;
               GO TO EXIT;
                    .
                    .
                    .
          EXIT:  HOLD = 0;
               END INTERRUPT_CO;
```

Upon receipt of a command, the processor must reply on the ACK/DONE bus.

```
          CIN:  PROCEDURE(LEVEL, SENDER,OPNO,CMD);
               DCL 1 INPUT_BUFFER(number of C.G.'s)
                   2 NAME;
                      3 BLOCK;
                      3 ITEM;
                   2 OPNO;
                   2 CMD;
               SEND = INPUT_BUFFER(LEVEL).NAME;
               OPNO = INPUT_BUFFER(LEVEL).OPNO;
               CMD  = INPUT_BUFFER(LEVEL).CMD;
               IF LEGAL(CMD) THEN CALL DONOUT(LEVEL,
                    SENDER, OPNO, AC_DONE);
               ELSE CALL DONOUT(LEVEL, SENDER, OPNO,
                    NAK_CODE);
          EXIT:  ENDCIN;
```

These routines as described here have not been intended to be complete or in a correct programming language syntax. No attempt has been made to optimize this program beyond using reasonable programming techniques. Instead they are indicative of a programming philosophy that may be applied to this system. For a system M equations, the execution time would be expected to be approximately

$$ET_m \doteq \frac{1}{3} \cdot \frac{1}{M} \cdot ET_s + OV$$

where ET_s is the execution time using a single system module. Since maximum use is made of the ability to broadcast operands, OV will be essentially insensitive to the number of equations. It can easily be seen therefore that this routine could be significantly faster on a multi-system module structure than a single module for appropriate values of M and ET_s.

This example shows the ease with which this architecture may be applied to the solution of problems and software developed for it. With very little difficulty, a problem of a relatively complex nature can be solved with a significant saving in execution time possible.

VII. CONCLUSIONS

A. General

This thesis has attempted to describe a novel data processing architecture consisting of a large collection of identical microcomputer based modules. A significant feature of this architecture is provision for a hierarchy of control that spreads the control functions over several system modules relieving the requirement for an extremely powerful central controller. In addition, several system modules may be grouped together to provide an extended arithmetic capability. This system architecture is therefore very flexible and has been shown to easily adapt to the requirements of a particular problem.

A primary motivation for this research has been the necessity for developing a multiprocessing system capable of allowing a high degree of local autonomy to each processor. In addition, cooperation between processors is required along with multiprogramming. All these facilities are available in this architecture.

A detailed block diagram design for a system module was developed. As a consequence, it can be seen that a practical implementation is easily within grasp. In addition, the software aspects discussed illustrate that the programmability of this architecture is not significantly different than that of a single microprocessor of the type incorporated within it. Although the overhead requirements are increased with the necessity to structure the system into the hierachical form required for a program and distribute parameters accordingly, this is a requirement that must be faced anytime cooperation is required in any system.

The analysis of the communication structure illustrates that a bus system as described here can be applied

satisfactorily in a large system of processors. The expected waiting times required to multiplex an item onto a bus are very reasonable in terms of the microcomputer's basic speed and hence an adequate degree of interaction between processors can easily be maintained. The central result of the analysis has been verified by computer simulation of the system and implies the validity of the conclusions that may be drawn from the analysis.

The analysis of the system and calculation of, f, the improvement factor show that this architecture can provide a significant improvement over single processor systems. Guidelines to insure generally large f were given and are seen to be similar to those provided by common sense. These guidelines are generally easy to apply as well.

As a consequence, it is felt that this architecture satisfies the goals of this research and provides a multiprocessor capable of multiprogramming, having a loosely coupled structure and capable of maintaining a high degree of local autonomy at each processor. It is therefore a very flexible system capable of self optimization and able to structure itself to best solve the problem. This architecture is innovative, easy to use and merits consideration as a model of systems to be constructed in the future.

B. Future Work

This research has provided the basic development of a hierarchical, restructurable multi-microprocessor computer architecture. It has, however, provided several areas for future study. For example, how many Control Groups would typically be required in a practical system; assuming that a mix of serial and parallel busses are employed as mentioned earlier, what effect will this have on the operation of the system?

In order to answer these questions and study the performance of this architecture more fully, the first extension to this work should be a simulator for the system that allows actual programs to be written and executed on the system. The result of this simulation study should be an instruction set that is well suited to the system architecture. Additionally, a characterization of the performance of the system under realistic operating constraints for different numbers of C.G.'s and mixes of serial and parallel busses should be obtained. Based upon the conclusions of this simulation study, further areas of investigation may be developed. Hopefully, a limited prototype of the system architecture will be constructed to further study flexible, restructurable computer architecture of this type.

REFERENCES AND BIBLIOGRAPHY

1. Anderson, R., Hayes, J. and Sherman, D., "Simulated Performance of a Ring Switched Data Network," IEEE Trans. on Comm., Vol. Com-20, 1972.

2. Arnold, R. and Page, E., "A Hierarchical Restructurable, Multi-Microprocessor Architecture," Proc. of Third Annual Symp. on Computer Architecture by IEEE and ACM, Jan. 1976.

3. Avi-Itzhak, B., "Some Heavy Traffic Characteristics of a Circular Data Network," Bell System Technical Journal, Vol. 50, No. 8, pp. 2521-2549, Oct. 1971.

4. Avi-Itzhak, V., and Naor, P., "Some Queuing Problems with Service Station Subject to Breakdown," Oper. Res., Vol. 11, no. 3, pp. 303-320, 1963.

5. Bansemian, H., and Decegama, A., "Evaluation of Hardware-Firmware-Software Tradeoffs with Mathematical Models," AFIPS SJCC, Vol. 38, 1971, pp. 151-161.

6. Bauer, F.L., ed., Lecture Notes in Economics and Mathematical Systems: Advanced Course on Software Engineering, Springer-Verlaz, Heidelberg W. Germ 1973.

7. Bauer, W., and Rosenberg, A., "Software -- Historical Perspective and Current Trends," AFIPS, Vol. 41, part II, 1972.

8. Breuer, M.A., "Adaptive Computers," Information and Control, Vol. II, no. 4, Oct. 1967.

9. Brinch Hansen, P., Operating System Principles, Prentice-Hall, New Jersey, 1973.

10. Burroughs Corp., "Multiprocessing System," U.S. Patent #378816, 1/22/74.

11. Coffman, E.G. and Denning, P.J., Operating Systems Theory, Prentice-Hall, New Jersey, 1973.

12. Chen, T.C., "Distributed Intelligence for User Oriented Computing," AFIPS Conference Proceedings, Vol. 41, part II, pp. 1049-1056, 1972.

13. Chen, T.C., "Parallelism Pipelining and Computer Efficiency," Computer Design, Vol. 10, pp. 69-74, 1971.

14. Chu, W. and Konheim, A., "On the Analysis and Modeling of a Class of Computer Communication Systems," IEEE

Trans. on Comm., Vol. COM-20, 1972.

15. Comptre Corp., H. Enslow, ed., Multiprocessors and Parallel Processing, J. Wiley and Sons, New York, 1974.

16. Crane, M.A. and Iglehart, D.L , "Simulating Stable Stochastic Systems, II: Markov Chains", JACM, Vol. 21, No. 1 Jan. 1974.

17. Crane, M.A. and Iglehart, D.L., "Simulating Stable Stochastic Systems, I: General Multiserver Queues," JACM, Vol. 21, No. 1, Jan. 1974.

18. Dennis, J.B., "First Version of a Data Flow Procedure Language," MIT Project MAC Computer Structures Group Memo 93-1, Aug. 1974.

19. Dennis, J.B. and Fossen, J.B., "Introduction to Data Flow Schemes," MIT Project MAC Computer Structures Group Memo 81-1, Sept. 1973.

20. Dijkstra, E.W., A Short Introduction to the Art of Programming, Technological University, Eindhaven, The Netherlands, Aug. 1971.

21. Fernandez, E. and Bussell, B., "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules.",IEEE Trans, on Computers, Vol. C-20, No.3, Feb. 1971.

22. Flynn, M.J., "Very High-Speed Computing Systems," IEEE Trans. on Comp., pp. 1901-1909, Dec. 1966

23. Graham, R.L. and Pollak, H.O., "On the Addressing Problem for Loop Switching," Bell System Technical Journal, Vol. 50, No. 8, pp. 2495-2519, Oct. 1971.

24. Hald, A., Statistical Theory with Engineering Applications, J.Wiley & Sons, New York, Reprinted, Toppan Co., Ltd., Tokyo, Japan, 1952.

25. Hayes, J. and Sherman, D., "Traffic Analysis of a Ring Switched Data Transmission System." Bell System Technical Journal, Vol. 50, No. 9, pp.2947-2978, Nov. 1971

26. Hoagland, A.S., "Mass Storage - Past, Present and Future," AFIPS 1972, Vol. 41, part II, 1972.

27. Hobbs, L C., et al., Parallel Processing Systems, Technology and Applications, Spartan Books, New York, 1970.

28. Jensen, D., "A Distributed Function Computer for Real Time Control," Proc. of Second Annual Symp. on Computer Architecture by IEEE and ACM, Jan. 1975.

29. Lorin, H., Parallelism in Hardware and Software: Real and Apparent Concurrency, Prentice Hall, New Jersey, 1972.

30. Martin, Benn D., "Data Subsystem for 12 Year Missions," Astronautics and Aeronautics, Sept. 1970.

31. Morris, D. and Treleaver, P.C., "A Stream Processing Network," ACM SIGPLAN, Vol. 10, No. 3, March 1975.

32. Mortelmans, J., An Investigation of a Parallel Reconfigurable Processor, Stanford Electronics Laboratories, Stanford University, Technical Report No. 3605-11, March 1974.

33. Parzen, E., Modern Probability Theory and Its Applications, John Wiley and Sons, Inc. Tappan Company LTD, Toylo, Japan, 1960.

34. Pierce, J.R., "How Far Can Data Go?", IEEE Trans on Communications, Vol. Com-20, pp. 527-530, 1972.

35. Plessey Handel und Investments, A.G., "Multiprocessor Data Processing Systems," U.S. Patent #3787818, 1/22/74.

36. Pomerene, J.H. "Historical Perspective -- Components," AFIPS 1972, Vol. 41, part II, 1972.

37. Ramamoorthy, C., Chandy, K and Gonzalez, M., "Optimal Scheduling Strategies in a Multiprocessor System," IEEE Trans. on Computers, Vol. C-21, No. 2, Feb. 1972.

38. Rattner, et.al., "Bipolar LSI Computing Elements Usher in New Era of Digital Design," Electronics, Vol. 47, No. 18, pp. 89-96, Sept. 1974.

39. Reams, C.C and Liu, M.T., "A Loop Network for Simultaneous Transmission of Variable Length Messages," Proc. of Second Annual Symp. on Computer Architecture by IEEE AND ACM, Jan. 1975.

40. Reams, C.C. and Liu, M.T. "Design and Simulation of the Distributed Loop Computer Network (DLCN)," Proc. of Third Annual Symp. on Computer Architecture by IEEE and ACM, Jan. 1976.

41. Reddi, S.S., "A Parallel Computer with Centralized Control," Rice University Department of Electrical Engineering.

42. Reddi, S.S. and Feustel, E A., "A Restructurable Computer System," Rice University, Department of Electrical Engineering, March, 1975.

43. Reddi, S.S., and Feustel, E.A., "An Approach to Restructurable Computer Systems," Rice University, Department of Electrical Engineering.

44. Selby, S.M., ed., CRC Standard Mathematical Tables, 1yth ed. Chemical Rubber Co., Ohio, 1969.

45. Sevazlian, B.D., and Stanfel, L.E., Analysis of Systems in Operations Research, Prentice Hall, New Jersey, 1975.

46. Southworth, R. and Deleeuw , S., Digital Computation and Numerical Methods, McGraw-Hill, New York, 1965.

47. Spragins, J., "Loop Transmission Systems -- Mean Value Analysis," IEEE Trans. on Comm., Vol. COM-20, 1972.

48. Stone, H.S., Discrete Math Structures and Their Applications, Science Research Associates, Chicago, Ill., 1973.

49. Stone, H.S., "Dynamic Memories with Enhanced Data Access," IEEE Trans. on Comp., Vol. C-21, No. 4, Apr. 1972.

50. Stone, H.S., "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Comp., Vol. C-20, No. 2, Feb. 1971.

51. Stone, H.S., ed., Introduction to Computer Architecture, Science Research Associates, Inc., Chicago, Ill., 1975.

52. Thurber, K., "Interconnection Networks - A Survey and Assessment," National Computer Conference Proceedings, pp. 909-919,1974.

53. Thurber, et.al., "A Systematic Approach to the Design of Digital Bussing Structures," AFIPS Conference Proceedings, Vol. 41, Part II, 1972.

54. Wilkes, M.V., "Historical Perspective -- Computer Architecture," AFIPS 1972, Vol. 41, part II, 1972.