SLAC-157
CS 309
UC-32
(MISC)

DYNAMIC CONTROL STRUCTURES AND THEIR USE IN EMULATION*

VICTOR R. LESSER
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY
Stanford, California 94305

PREPARED FOR THE U. S. ATOMIC ENERGY
COMMISSION UNDER CONTRACT NOS.
AT(04-3)-515 and AT(04-3)-326, P.A. No. 23

October 1972

Printed in the United States of America. Available from National Technical Information Service, U. S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22151.
Price: Printed Copy $3.00; Microfiche $0.95.

*Ph.D. Dissertation.

DEDICATED TO THE MEMORY OF MY FATHER

REUBEN DAVID LESSER

WHOSE INSPIRATION HAS FINALLY BLOSSOMED

ABSTRACT

This thesis describes an architecture for a parallel microcomputer system that permits a systematic and flexible approach to the emulation of a wide variety of complex sequential and parallel intermediate machine languages in a dynamically varying Processor-Memory-Switch (PMS) environment. This architecture is based on the view that complex emulators can be best structured in terms of a set of microprocessors that interact in a highly structure manner. These highly structured interaction patterns are defined through the concept of a virtual PMS environment. This concept embodies the capability for reconfiguring both the internal and the external environment of a microcomputer system: the number of internal working registers of each microprocessor; the structure of memory, e.g., its size and word length; and the number of microprocessors and functional units, and their interconnection and interaction patterns. The virtual PMS is implemented in the microcomputer architecture by adding a new global level of hardware control. A particular virtual PMS is dynamically defined by modifying the syntax (i.e., the number of data elements and their relationship) of the data structure for control used by this global hardware control level.

The representational capabilities of this architecture have been examined through the microprogramming of an emulator for a sophisticated parallel machine language, Adams' Graph Machine Language. The emulator of this machine language has demonstrated the versatility and usefulness of the concept of a virtual PMS by requiring less than 600 64-bit microinstructions to be programmed, while at the same time being able to exploit fully the implicit parallelism of a graph machine program. In addition, the dynamic execution characteristics of this architecture have been studied through the use of a detailed simulator of a hardware organization for this microcomputer architecture. The simulator has been used to verify quantitatively that this organization permits parallel activity on the virtual PMS to be mapped without significant overhead onto the physical PMS. In particular, the simulation results indicate that where sufficient parallel activity exists, the addition of microprocessors to the PMS configuration will reduce in a linear way the time it takes to execute the computation. The simulation results have also indicated that the logical hardware design, with the appropriate PMS configuration, can efficiently handle sustained parallel activity, involving highly structured interaction patterns, of greater than sixteen microprocessors.

Acknowledgements

I wish to express my sincere thanks to Professor William F. Miller who has shepherded me through this long arduous thesis. I would also like to acknowledge the advice, support and friendship given to me by the other members of my reading committee — Professors Edward Davidson, Harry Saal and Forest Baskett — without which I would never have finished. Over the years there have been many people whose encouragement of my research efforts and fruitful discussions have been very important to my work. Special thanks to Professors William McKeeman, Robert Fabry and Alan Kay, and to my fellow graduate students and friends Lee Erman, Bill Riddle and John Levy. I would also like to thank Edward Nelson for the considerable effort he devoted on my behalf so that I could use his graph program simulator.

There have been many other people who have been very helpful in preparing this manuscript — Harriet Canfield for her numerous retypings, Joe Wells for his help in using WYLBUR, and finally to Guynn Perry and Hugo Smith for their help in the final moments.

Finally, I would like to acknowledge the deep debt of gratitude I owe to my Uncle Harold Freed, whose friendship, support, advice and even criticism I will always cherish.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

Page

LIST OF TABLES

I. INTRODUCTION

> "Pragmatically important problems such as the design of programming languages appropriate for given problem areas, design of computer systems well matched to given programming languages, and defining efficient structures for translators are capable of being adequately handled only within a model that assigns similar structures to programming languages and computer languages both globally and locally". (Nar67)

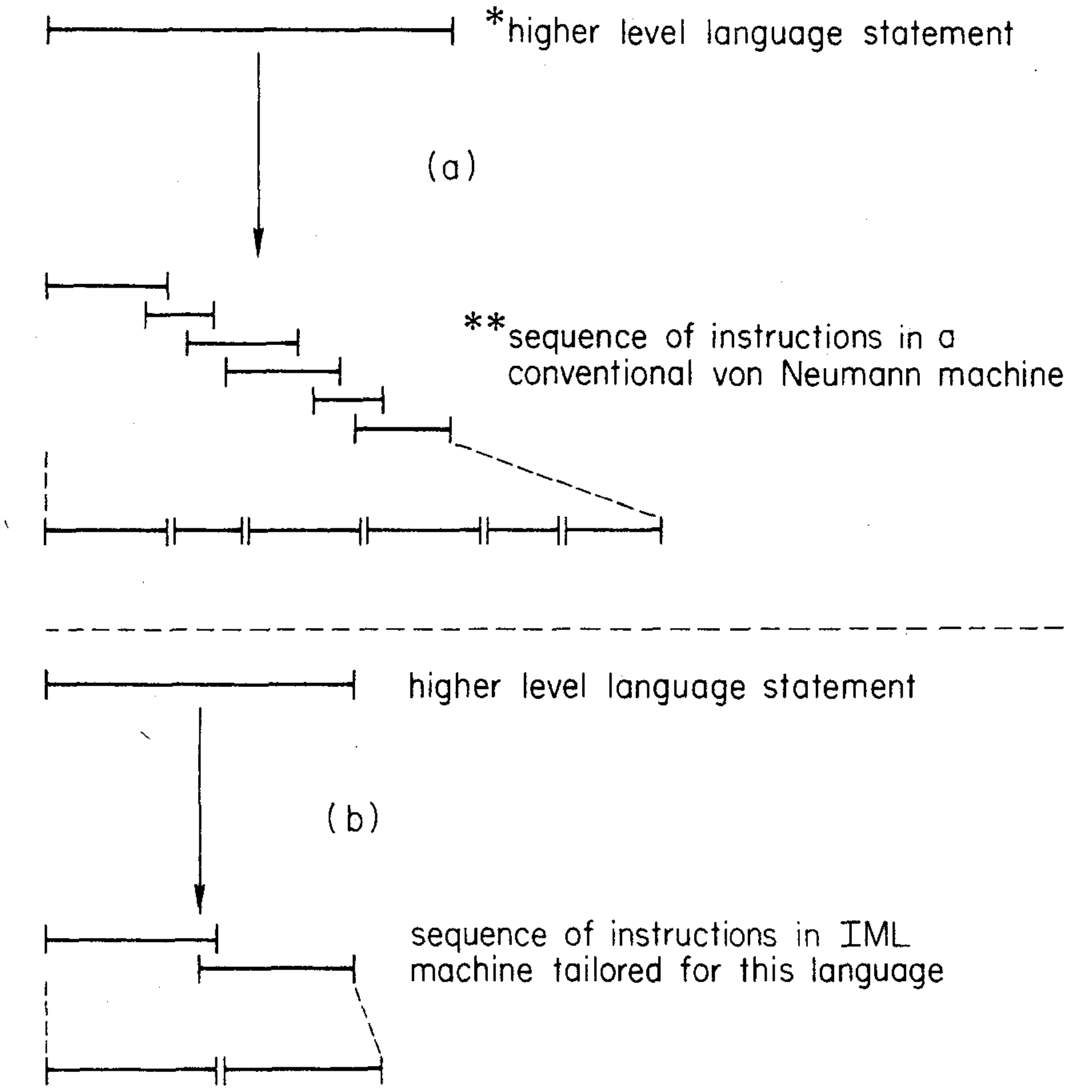
I.1 Unification of Three Trends in Computer Architecture

Over the past few years, there has been a growing trend toward the design of computers whose architecture differs considerably from that of the classic von Neumann type computer*. This departure from the von Neumann type computer architecture has occurred on three levels: 1) the Instruction-Set-Processor (ISP) level of which the Burroughs B5500 is an example, 2) the PMS Processor-Memory-Switch (PMS) level of which the ILLIAC-IV is an example, and 3) the Processor Implementation Technique (PIT) level of computer design of which the IBM 360/40 is an example (terminology from BEL70).

The first trend, that is on the Instruction-Set-Processor level, has led to the development of computers whose machine languages

*A von Neumann type computer is considered to have a sequential control structure, and instructions which operate on single units of data accessed from a linear address space.

are optimized for a particular higher level language or operating system environment. This trend is exemplified in the languages of machines such as the Burroughs B6500(HAV68) for Algol, Fairchild's SYMBOL machine (RIC71) for string manipulation, Abrams' APL machine(ABR70), Melbourne and Pugmire's Fortran machine(MEL65), etc. These machine languages represent a broader class of languages than are conventionally considered (von Neumann) machine languages. We shall refer to this broader class as Intermediate Machine Languages (IML). The tailoring of an IML to a specific higher level language is accomplished by incorporating instructions and data types in the IML which directly implement (i.e., mirror) the primitive operations of the higher level language. For instance, an ALGOL procedure call including the modification of the addressing environment is directly mirrored by the ENTER instruction in the B6500. Thus, instead of implementing the semantics of higher level language primitive operations through an unnecessarily long and complicated sequence of instructions (see Figure 1a), the IML is designed so that there is a single instruction or, at worst, a short sequence of instructions that efficiently carry out the primitive operation (see Figure 1b). Therefore, by tailoring a machine language more closely to a particular higher level language, the mapping between the higher level language and the machine language is simpler and results in a more compact and efficient generated code (MCK67). This trend should accelerate as the cost of software is recognized as the major cost component of a computer system (DEN71).


*higher level language statement
(a)
**sequence of instructions in a conventional von Neumann machine
higher level language statement
(b)
sequence of instructions in IML machine tailored for this language

* The length of the line is intended to give some relative measure of the inherent computational activity involved in the execution of a statement or machine instruction.

** The sequence of machine instructions is intended to indicate the inefficient use of the computational activity of each instruction, and the high overhead of instruction fetching and decoding due to the large number of instructions required to be executed.

Figure 1. Mapping of a Higher Level Language to a Machine Language

The second trend, that is on the Processor-Memory-Switch level, has led to the development of computers that are able to carry out parallel activity at the functional unit level, instruction level, or process level. These different levels of parallel activity are exemplified by machines such as the CDC6600(THO64), which permits clocked asynchronous parallel operation of functional units, the ILLIAC-IV(SLO67), which permits lock-step execution of multiple copies of a single instruction stream on identical processors and the multiprocessor B825(AND62), which permits the execution of multiple asynchronous instruction streams on identical CPU's. This trend towards parallelism at the PMS level has occurred mainly in the design of high performance computer systems. However, as LSI technology brings down the cost of logic and as reliability of the computer system becomes an important component of the design, this trend toward parallelism should extend to many more types of computer systems(BEL72).

These two design trends on the ISP and PMS levels are not disparate but rather are separate aspects of a more general trend towards the design of complex problem oriented computers whose architecture departs considerably from a classical von Neumann architecture. The B6500, the ILLIAC-IV, and the SYMBOL machine represent to varying degrees an integration of these two trends in computer architecture. The B6500 configured as a multiprocessor system permits the allocation of multiple processors to the execution of a single Algol program, the ILLIAC-IV permits highly parallel execution of problems involving an array structured data base, and the SYMBOL machine permits a set of non-identical processors to work in a parallel coroutine structure to interpret and execute a sophisticated IML instruction repetoire.

In parallel with these first two trends, there has been a third trend towards providing a systematic and flexible technique for implementing a processor in hardware. This third trend, that is on the Processor-Implementation-Technique level, has led to the to the development of the concept of a microcomputer (WIL69, HUS70), which provides a systematic and regular technique for specifying control at the circuit gate level. The major use, to date, of microcomputers has been in the implementation (emulation) of the processor of a specific von Neumann type computer; e.g. IBM 360/40(IBM66), with the microcomputer usually having a read-only control memory. Recently, there have been the beginnings of an attempt to combine complex problem-oriented computer design with microcomputer design (WEB67,ROS69), implementing a specific architecture by modifying the READ-WRITE control memory of the microcomputer. It is hoped that the goal of emulating a wide range of problem-oriented computers can be realized by dynamically modifying the control memory of a single microcomputer system. This goal cannot be effectively attained on microcomputers whose architecture is essentially designed for the emulation of the instruction set of a von Neumann type computer in a non-parallel PMS environment. This thesis offers an architecture for a microcomputer system that permits a systematic and flexible approach to the emulation of a wide variety of complex sequential and parallel intermediate machine languages in a dynamically varying PMS environment which contains multiple microprocessors and functional units.

I.2.1 Traditional Microcomputer Architectures

The conceptual architecture of a conventional microcomputer system is shown in Figure 2. The memory subsystem contains a machine language program (and its corresponding data) which is to be executed on the emulated computer. The microprogram memory contains microinstructions that are used to interpret instructions of the emulated computer. The formats of microinstructions on existing microcomputers can be characterized into two general classes: horizontal and vertical microinstruction formats (see Figure 3 and Figure 4 respectively). In a microinstruction specified in terms of the horizontal format, each bit of the microinstruction word controls a particular internal operation of the microprocessor (e.g., the opening or closing of a hardware data path between internal working registers, or the arithmetic operation to be performed on a data path). The vertical format microinstruction word is broken into a series of fields as in a conventional machine instruction, where each field is used to specify either one of a set of internal registers or one of a set of built-in arithmetic operations. These microinstructions are executed on a single microprocessor which is connected to a set of functional units. The term functional unit is used in a very broad context to refer to input/output devices, their corresponding controllers, and arithmetic units, such as a floating-point adder. In conventional microprocessors, the functional units connected are usually input/output devices or their corresponding controllers rather than arithmetic units.

In order to perform an emulation using a conventional microcomputer architecture, the microprogrammer must first imbed the state image of the emulated computer, S(e), which includes the set of working registers of the computer (accumulator, index register, program counter, etc.) and its main memory into the state image of the microcomputer, S(m), which includes the Memory Subsystem and the internal state and working registers of the microprocessor. For efficiency, frequently accessed elements of S(e) (e.g. the program counter of the emulated machine, etc.) are stored, if possible, in the internal working registers of the microprocessor. An emulator constructed out of microinstructions has a conceptual microprogram structure shown in Figure 5. The "control process" activates the "decoding process" with data that identifies the next instruction of the emulated computer to be executed; the decoding process then analyzes the instruction to be executed so as to determine the "semantic routine" together with its appropriate calling sequence, whose activation will perform the semantics of the emulated instruction. After the appropriate semantic routine has been executed, the flow of control returns to the control process which, based on the results of executing the decoding process and the

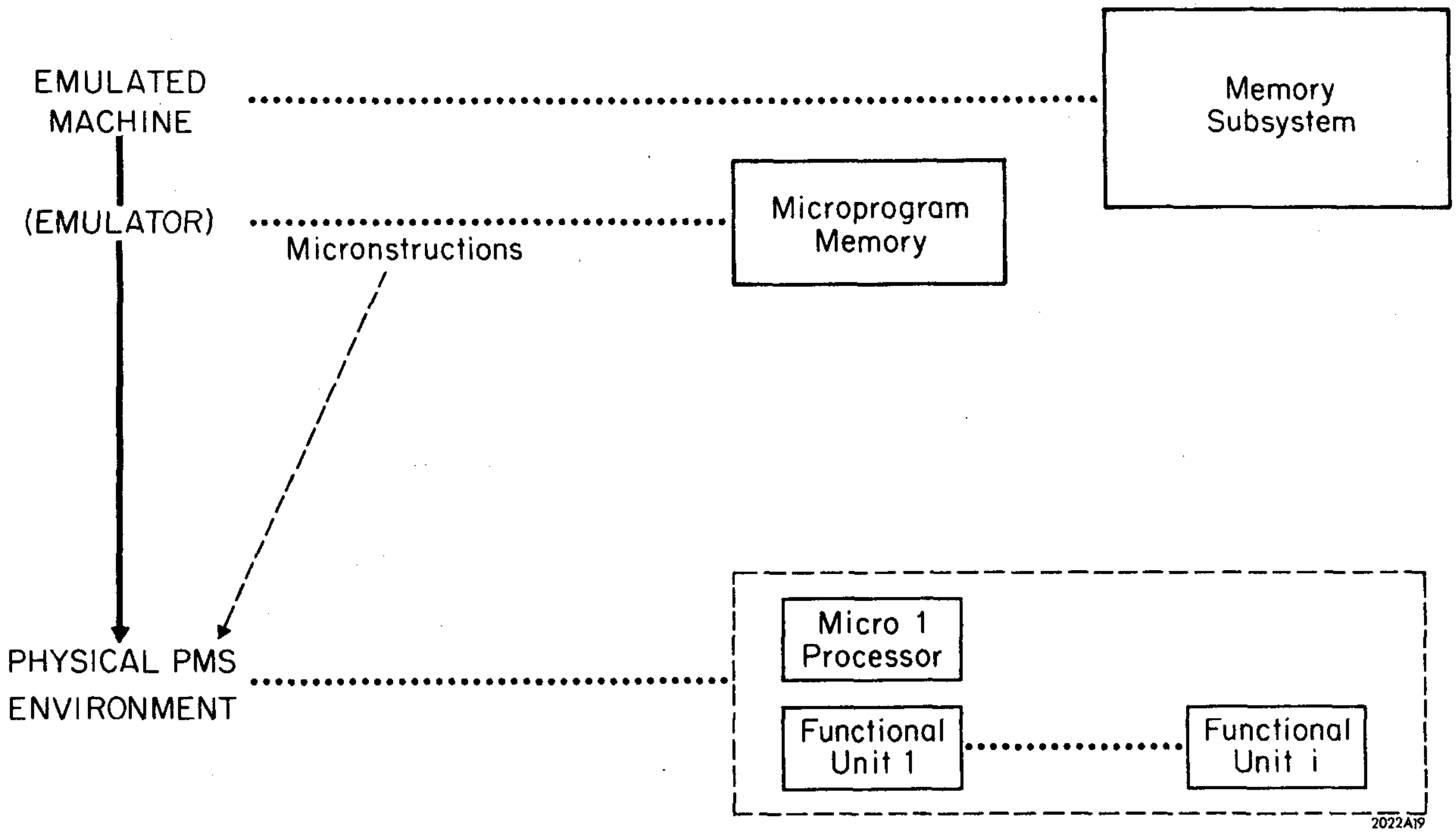


Figure 2. The Relationship between the Emulation Process and a Conventional Microcomputer Architecture

Field	ROS Bits	Function of Field
	0	Parity of bits 0-30
LU	1-3	Left input to mover
MV	4-5	Right input to mover
ZP	6-11	Bits 0-5 of next ROS address
ZF	12-15	Source of bits 6-9 of the next ROS address
ZN	16-18	ROS addressing mode
TR	19-23	Destination of adder latch contents
	24	Spare
WS	25-27	Source of local storage address
SF	28-30	Local storage function
	31	Parity of bits 32-55
IV	23-24	Invalid digit test and instruction address register control
AL	35-39	Shift control and gating into adder latch
WM	40-43	Mover destination
UP	44-45	Byte counter function
MD	46	MD counter control

Field	ROS Bits	Function of Field
LB	47	LB counter control
MB	48	MB counter control
DG	49-51	Length counter and carry insertion control
UL	52-53	Mover function -- left digit
UR	54-55	Mover function -- right digit
	56	Parity of bits 57-89
CE	57-60	Emit field (used as data)
LX	61-63	Left input to adder
TC	64	True/complement control of left adder input
RY	65-67	Right input to adder
AD	68-71	Adder function
AB	72-77	Condition branch test A (furnishes bit 10 of next ROS address)
BB	78-82	Condition branch test B (furnishes bit 11 of next ROS address)
	83	Spare
SS	84-89	Stat setting and miscellaneous control

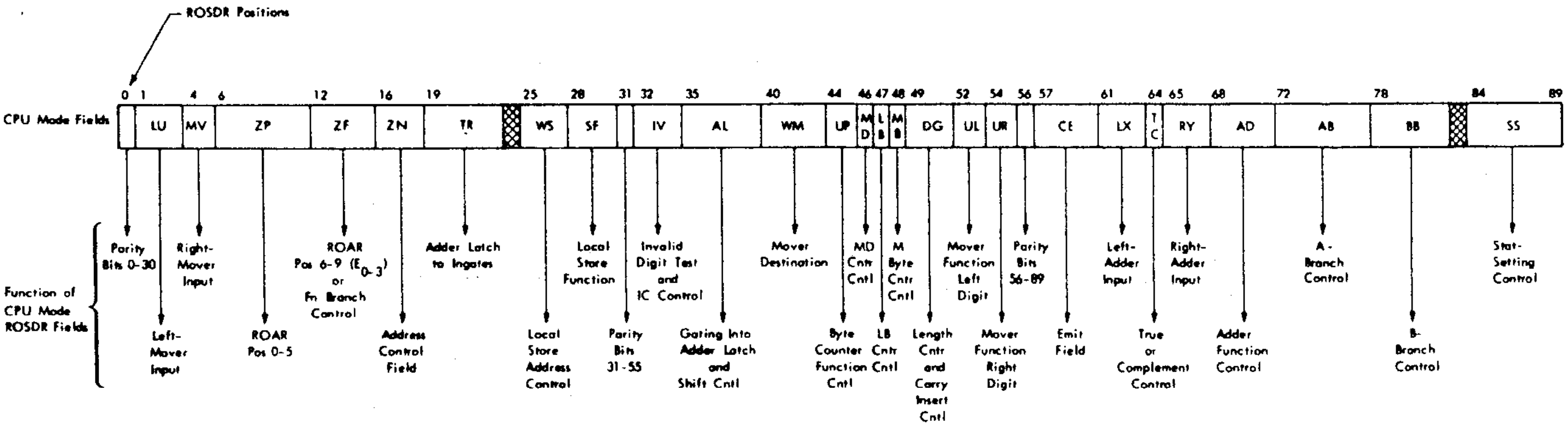


Figure 3. Horizontal (minimal encoded) Microcode Format of IBM 360/50 (from HUS 70)

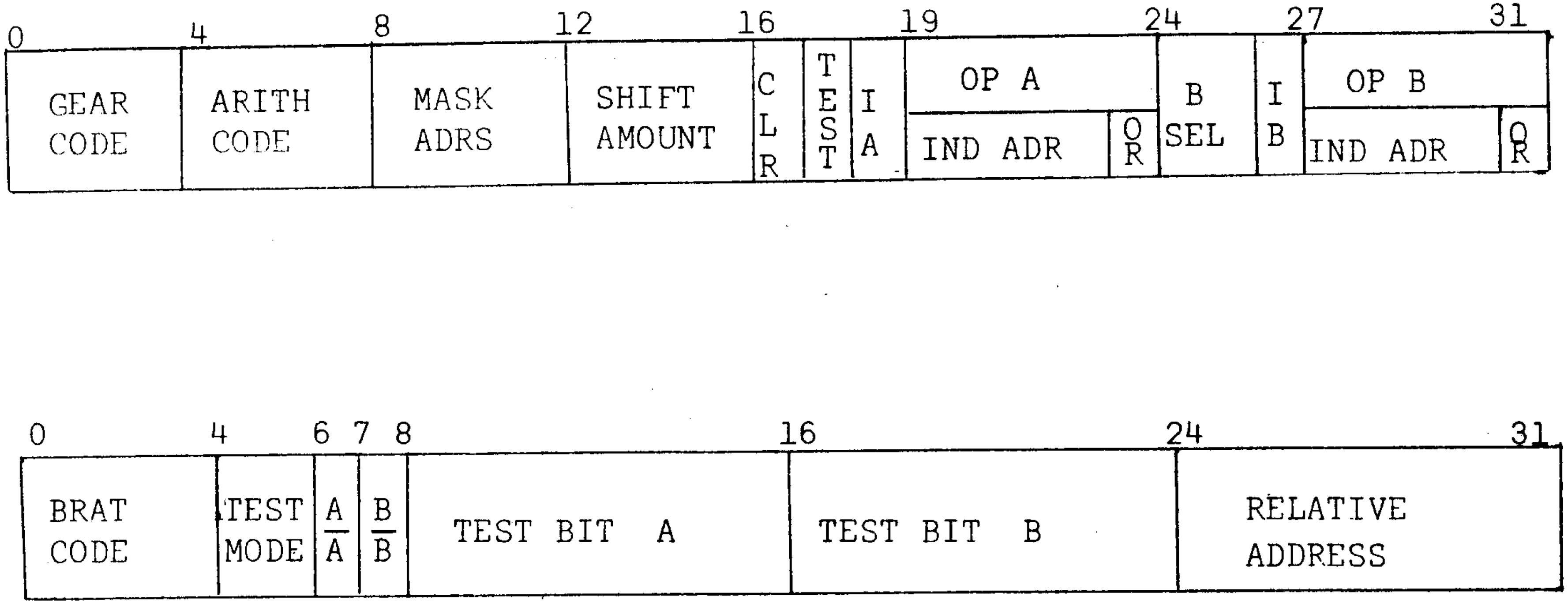


Figure 4. Vertical (highly encoded) Microcode Format of MLP-900 (from LAW 71)

semantic routine, selects the next instruction to be emulated. This basic cycle is conventionally called (TUC65) the "Do Interpretive Loop" (DIL). This two step design process for an emulator is represented in terms of a commutative diagram in Figure 6*.

I.2.2 Basis for a New Microcomputer Architecture

In microcomputers designed to emulate a specific computer architecture, or family of computers with similar architectures, the imbedding of the state image is straightforward. There are internal registers dedicated to holding commonly accessed state information of the emulated computer. The control and decoding processes of the emulator are usually directly implemented in the hardware taking their data from the dedicated internal registers. This control and decoding hardware is usually directly integrated into the microprocessor's control structure used for the sequencing of microinstructions so as to create an extended control structure. Through this concept of an extended control structure, the sequencing of microinstructions is driven directly by the sequencing of emulated instructions. In addition, the internal data paths and microperations are tailored so as to make microprograms that carry out the semantic phase of an

*The left hand side of the commutative diagram represents the effect of executing an instruction of the emulated computer on the state image of the emulated computer. The right hand side represents the sequence of transformations that the microcomputer must perform on its own state image in order to emulate this instruction.

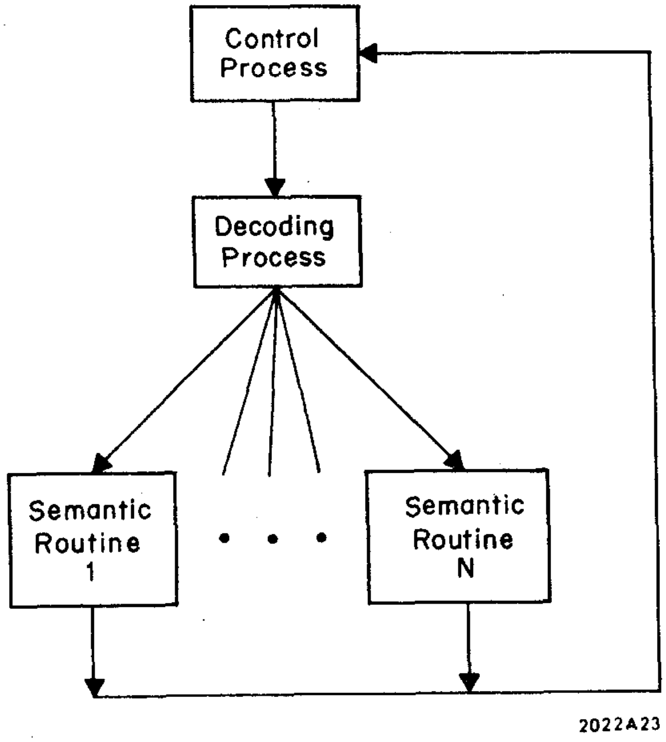


Figure 5. Conceptual Program Structure of an Emulator

$$\begin{tikzcd}
\underline{\text{Emulated Machine}} & \underline{\text{Physical Microcomputer}} \\
S^{I}_{\text{Emulated machine}} \arrow[r, dashed, "\text{implicit imbedding}", "\text{of state image}"'] \arrow[d, dash] & S^{I}_{\mu} \arrow[d, dash] \\
\text{Emulated instruction } I \arrow[r, "\text{emulator}", "\text{(control and decoding process)}"'] \arrow[d] & \mu \text{ instructions of semantic routine for instruction } I \arrow[d] \\
S^{I+1}_{\text{Emulated machine}} & S^{I+1}_{\mu} \arrow[l, dashed, "\text{implicit extraction}"', "\text{of state image}"]
\end{tikzcd}$$

Figure 6. Commutative State Diagram of Conventional Emulation Process

emulation efficient and compact. Thus, a microcomputer is a flexible and efficient technique for emulating computers that have been anticipated.

However, this tailored microcomputer architecture is inefficient when used to emulate a machine language (IML) that is dissimilar in its instruction format, control structure or instruction semantics to the machine languages anticipated by the designer. This inefficiency occurs because:

1) the imbedding of the S(e) into S(m) is not straightforward (e.g., mapping a machine which has a 36 bit wide word into a microprocessor which has a 32 bit wide word) and dedicated (specific function) internal registers cannot be used directly to hold commonly accessed state information (TUC65);

2) the hardware implementation of the control and decoding processes cannot be used directly;

3) the microinstructions and the internal data paths they manipulate that were designed for a specific set of instruction semantics are clumsy when applied to the microprogramming of the control and decoding processes, and different instruction semantics.

These problems with a tailored microcomputer architecture are analogous in many respects to the previously discussed problems with the execution of higher level languages on a von Neuman machine.

In response to these problems with a conventional microcomputer architecture, new types of microcomputer architectures are beginning to be developed, most notably the QM-1 (ROS71) and MLP-900 (LAW71), which are designed more for general purpose emulation rather than for implementation of a specific processor. These new microcomputer architectures differ from conventional architectures by providing the capability of configuring a set of non-specific internal registers of the microprocessor and their corresponding interconnection pattern into the specific configuration appropriate for the emulation of a particular IML. Once the particular configuration is set, the semantics of the microinstruction, when executed, operate directly in the context of the chosen configuration. This flexible configuration capability, referred to in less general contexts as residual control (FLY71), leads to ease of representation, code compactness, and efficient use of microprocessor resources*. The concept of residual control represents a design trade-off between conventional microprocessors which are efficient but inflexible and non-specific microprocessor architectures which are flexible but inefficient because there is no specification of functions or internal resources for particular types of emulation. The cost of this capability for configurability is extra levels of hardware logic, and high speed memory to hold configuration specifications, which implies a slower microprocessor cycle time, and thus a more costly microprocessor.

---

*The concept of residual control represents the extraction from the microinstruction of the enviromental information which remains static during the execution of a sequence of microinstructions. This environmental information specifies gating paths and adder configurations and modes, and is held in set-up registers which are used by the hardware to determine how to interpret a microinstruction.

This concept of dynamic reconfigurability for representational ease has also been employed in the design of other components of a computer system. In particular, the idea of virtual memory (DEN66) is directly analogous in its use and techniques for implementation to the idea of residual control. Both ideas represent attempts to match the structure of the computer system more closely (in this case, dynamically) to the structure of the problem to be programmed.

The conventional microcomputer, augmented with the capability for dynamic configuration of bus interaction patterns (i.e., QM-1) and for generalized bit string extraction and manipulation (i.e., MLP-900, B1700(WIL72)), provides an appropriate environment for emulating a wide range of machine languages which have simple control structures and instruction semantics that operate on simple data structures (e.g. von Neuman type computers). However, intermediate machine languages that are tailored for the execution of higher level languages or for the execution of operating system implementation languages are not so simple since the complexity of the higher level language operations is reflected in the semantics of the IML instructions and control structure. If the current trend in the development of higher level languages is maintained, language-oriented IML's will employ increasingly more sophisticated control structures, such as recursion, coroutines, parallelism, etc., and instructions that access complex data structures, such as lists, trees, arrays, etc., and perform operations such as sort (LEV72) matrix manipulation (GRA70, ABR70), etc. As will be argued below, these IML's call for a more sophisticated control structure in the microcomputer.

The control structure of a machine language or higher level language consists of a set of control rules(CR) and a data structure for control(CDS) commonly called Program Status Word(PSW) or processor state, on which the control rules operate. The control rules determine at each meaningful unit of activity of the language which statement or statements of the language will next be executed. For example, if the CDS of a simplified computer consisted of a program counter and an interrupt register, then the CR of this simplified computer might be the following paradigm: if there are no interrupts pending, then execute the instruction at the location specified by the program counter, otherwise, store the program counter at a fixed location in the program memory, reset the interrupt flip-flop, place the address of the interrupt handling routine in the program counter, and then execute the first instruction of the interrupt handling routine. This definition of a control structure makes a clear distinction between the control structure of a language and the execution of control statements of a language, e.g., conditional branch instructions, etc. The control statements of a language implicitly, rather than explicitly, affect sequencing by modifying only one part of the control structure, namely, the CDS; the actual sequencing of statements occurs only by the interpretation of the control data structure by the control rules. For example, consider the results of executing the control statement "BRANCH TO LOCATION X" in terms of the control structure of the simplified computer discussed previously. The branch statement, when executed, places the address X in the program counter; however, the next instruction to be executed may not be at address X since during the time the branch instruction was executed an interrupt could have occurred.

The simple sequential control structure of a conventional microcomputer is inappropriate for emulation of sophisticated IML's in a parallel PMS environment for the following reasons:

> 1) The control structure component of the state image of sophisticated IML's is not easily imbedded in the control structure component of S(m); in particular, all parallel activity specified in the control structure component of S(e) must be sequentialized when imbedded in the control structure component of S(m); in essence, if the emulated machine contains instructions capable of fork-join type parallelism(CON68), there should be a simple and short sequence of microinstructions that modify control structure components of S(m) so that the microcomputer system will directly start to emulate in parallel the newly created instruction stream defined by the fork instruction.
>
> 2) The control structure for sequencing the different phases (tasks) required in the emulation of sophisticated IML's may not be sequential: the instruction decode, and fetch, and semantics phases may be pipelined, as in the 360/91(AND67), or the phases may interact in a parallel or quasi-parallel coroutine as in the SYMBOL machine.
>
> 3) The control structure may be required to represent the coordination, on a very fine interaction level, of multiple microprocessors and functional units, such as the lock-step execution of processors in the ILLIAC-IV, or scheduling of asynchronous functional units in the CDC-6600.

Thus, the flexibility of the control structure of the microcomputer is crucial to the effective emulation of sophisticated IML's. In particular, the control structure of a microcomputer should be able to be dynamically restructured, in a manner similar to but more general than the reconfigurability specified through residual control and virtual memory, so that it more directly mirrors the control structure of the emulated machine and its emulator.

I.2.3 A New Microcomputer Architecture

The microcomputer architecture to be presented in this thesis is based on unifying in a single framework the concepts of residual control, virtual memory, and dynamic (restructurable) control structure. These concepts have been integrated through the idea of a virtual PMS environment; this idea embodies the capability for reconfiguring both the internal and the external environment of a microcomputer system. The concept of residual control as used in this context allows the varying of the number of internal working registers of each microprocessor; the concept of virtual memory in this context allows the varying of the structure of memory, e.g., its size and word length; the concept of a dynamic control structure allows the varying of the number of microprocessors and functional units, their interconnections and interaction patterns.

The concept of a virtual PMS environment leads to a new view of emulation pictured in Figure 7a and 7b, where S(vm) represents the virtual state image of the microcomputer system created through the specification of a particular PMS environment and the additional level of hardware is used to map microoperations, performed in the context of the virtual PMS environment, onto the actual (physical) PMS environment. This additional level of hardware is analogous to the hardware in a virtual memory system which manages the page tables and performs the mapping of virtual addresses to physical addresses. The extra dimension of representational freedom provided by the concept of a virtual PMS environment allows:

1) The virtual state image of the microcomputer system, S(vm), to be structured so as to make the imbedding of the state image of complex IML's, S(e), straightforward;

2) The microinstructions to operate directly in the context of an appropriate S(vm) so as to make the coding of the emulator compact and simple;

3) The emulator to be coded so as to be independent of the physical PMS environment but, at the same time, exploit physical resources when available.

The concept of a virtual PMS environment also leads to a new view, as pictured in Figure 8, of a microcomputer architecture. In this new architecture, there are two distinct hardware levels of control that are structured in a hierarchical fashion: the conventional level of control, contained in each microprocessor, for the sequencing of microinstructions, and a new level of control for the sequencing of microprocessors and functional units; thus, the microcomputer system contains both local, distributed control structures and a global, system-wide control structure. The control rules for this new level of control are implemented in hardware which is distributed in each microprocessor and in the controller for the bus(ses) which are used for inter-processor communication. This new level of control must be an integral part of the hardware organization for reasons which are analogous to the use of special mapping hardware

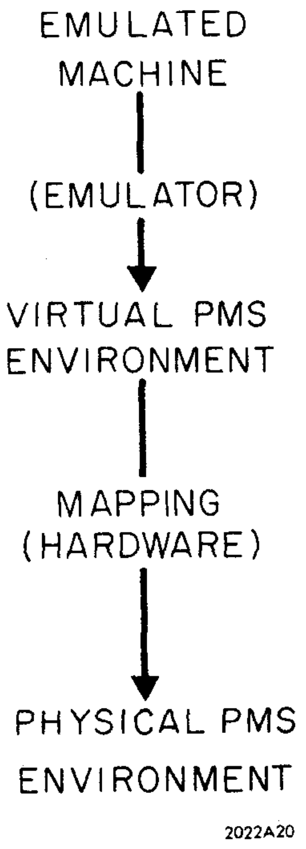


Figure 7a. A New View of Emulation Process

$$\begin{tikzcd}
\text{Emulated Machine} & \text{Virtual Microcomputer System} & \text{Physical Microcomputer System} \\
S^{I}_{\text{Emulated machine}} \arrow[r, dashed, "\text{implicit imbedding}", "\text{of state image}"'] \arrow[d, dash] & S^{I}_{V\mu} \arrow[r, "\text{mapping}", "\text{hardware}"'] \arrow[d, dash] & S^{I}_{\mu} \arrow[dd] \\
\text{Emulated instructions} \arrow[r, "\text{emulator}"] \arrow[d] & \mu \text{ instructions} \arrow[r, "\text{mapping}", "\text{hardware}"'] \arrow[d] & {} \\
S^{I+1}_{\text{Emulated machine}} & S^{I+1}_{V\mu} \arrow[l, dashed, "\text{implicit extraction}"', "\text{of state image}"] & S^{I+1}_{\mu} \arrow[l, "\text{mapping}"', "\text{hardware}"]
\end{tikzcd}$$

2022A21

Figure 7b. A New View of Commutative State Diagram of Emulation Process

for virtual addressing. Otherwise, the overhead in implementing highly structured parallel interaction patterns*, where the parallel activity is of short duration, will overwhelm the inherent parallelism of the interaction patterns. The control data structure for this new level of control is contained in a separate memory called the Process Space Memory (M.PSM).

A particular virtual PMS environment is dynamically defined by constructing an appropriate global control structure for the microcomputer system. An appropriate global control structure is constructed by dynamically modifying the syntax, i.e., the number of data elements and their relationships, of the control data structure (CDS) contained in the Process Space Memory. In a conventional computer or microcomputer system, the data structure for control contains a fixed set of data elements whose relationships are predefined. Thus, in a conventional system, control can only be modified by changing the value of data elements in the CDS. The ability added here to modify the syntax of the data structure for control, as will be seen later, is the key to tailoring a virtual PMS environment for a particular emulated machine.

*A highly structured interaction pattern among microprocesses implies that there is a high degree of coordination among microprocesses. This is in contrast to an unstructured interaction pattern which implies that once one microprocess has initiated the activity of another microprocess, there is no further coordination of the activity of these two microprocesses.

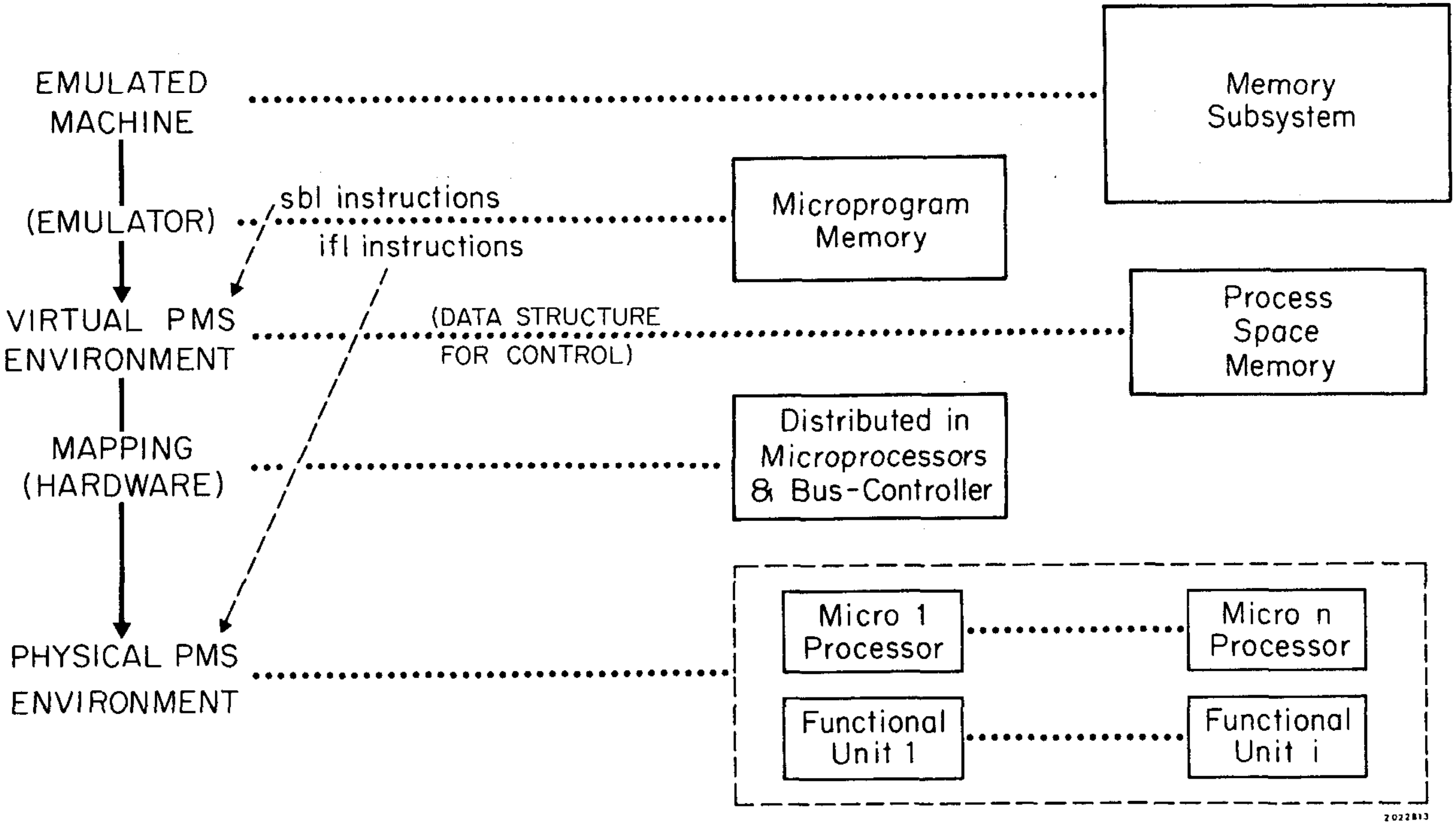


Figure 8. A New View of Relationship between Emulation Process and Microcomputer Architecture

There are two general classes of microinstructions in the microcomputer. One class, called the Integer Function Language (IFL), deals with internal registers of the microprocessor, and are like conventional vertical microinstructions. The other class, called the Structure Building Language (SBL), deals with the external environment of the microprocessor by modifying the CDS contained in the Process Space Memory. The SBL can be thought of as a control structure definitional language which is designed so as to regularize control at the microprocessor/microprocessor interaction level, microprocessor/functional unit interaction level and microprocessor/Memory Subsystem interaction level.

This new level of hardware control can also be thought of as a simple, hardware operating system which controls the scheduling and interactions among microprocessors and functional units. In this context, the CDS stored in the Process Space Memory (M.PSM) is analogous to the control blocks and queues that describe the interaction and existence of tasks (or processes) in a multiprogrammed operating system. The SBL statements are analogous to requests for those supervisor services that affect interprocess interaction patterns in such an operating system. The SBL can manipulate and build up the CDS only in ways understandable to the global control rules of the microcomputer system; the CDS, in a very general sense, can be considered a control structure definition program which, when interpreted by the global control rules of the microcomputer system, defines a particular sequential or parallel control structure for sequencing of virtual microprocessors (microprocesses*) and functional units. The CDS can also be thought of as a variable structure template that defines a particular internal and external structure for the microcomputer system, thus the idea of a virtual PMS environment. An SBL program is quite different from a sequence of control statements since the control structure definition program (the CDS), constructed by the SBL, is external to the microprogram. The separation of the control structure definition program permits the static parts of the virtual PMS environment to be generated only once for repeated executions of emulation.

SBL statements dynamically modify the CDS to directly reflect the state transitions occurring in the emulated computer. SBL statements reflect these state transitions by modifying the CDS so as to change: 1) the data environment of a microprocess, 2) the activity state of a microprocess, or 3) the interaction patterns among microprocesses (only this third case results in a modification of the syntax of the CDS). The CDS explicitly represents the relationship between the execution of a microprogram and the immediate data environment (parameters) in which the instructions of the microprogram operate. This relationship between the control and data environments, as will be seen in more detail later, allows 1) the representation of data environment interrelationships among microprocesses, and 2) the state of the emulated computer to be integrated directly into the CDS (e.g., the IML program counter could be a parameter of a microprocess

---

*The relationship between a microprogram and microprocess is analogous to the relationship between a program and a process(LAM68).

defined in the CDS). Thus, an IML control statement, such as a conditional branch in a pipelined emulator, can be implemented by an SBL microinstruction that modifies the data environment (e.g., the IML program counter parameter) of the microprocess that asynchronously fetches the next instruction to be executed. Likewise, the processing of an IML interrupt can be handled by an SBL instruction that suspends the activity of the microprocess that emulates the IML interrupt handling process. IML control statements that specify the creation of new paths of controls (e.g., fork-join instruction , etc.) can be implemented by an SBL statement that builds up the appropriate structure in the CDS for emulating IML instructions along this newly created control path. Additionally, the SBL can be used to construct in the CDS: 1) control structures for sequencing microprocesses which carry out the semantics of emulated instructions, 2) control structures for I/O, and 3) control structures for data accessing operations.

I.3 An Outline of the Justification for This New Microcomputer Architecture

The remainder of the thesis will develop the following conclusions:

1) The concept of a virtual PMS provides a representational framework in which a wide variety of sequential and parallel control structures can be easily expressed.

2) The SBL can be used to simply and compactly code emulators for complex IML's.

3) A computer organization which implements the concept of a virtual PMS can be designed such that highly parallel activity specified on the virutal PMS can be translated without undue overhead into highly parallel activity on the physical PMS.

Chapter II contains a detailed discussion of the SBL and the associated global control structure, and their applicability for representing particular types of control structures.

Chapter III reviews in a step by step manner the design and coding of an emulator for a complex IML. The emulator for this complex IML represents a comprehensive test case that is used to illustrate how control structure concepts, such as distributed control, pipelining and recursion are coded in the SBL.

Chapter IV discusses the computer organizational issues involved in implementing this proposed microcomputer architecture. Specifically, the following organizational issues will be discussed:

1) the bussing structures to access memory, and for interprocessor communication;

2) the hardware algorithm for scheduling of virtual microprocessors on actual microprocessors.

3) the design requirements necessary to insure no hardware deadlocks are introduced which are not already present as software deadlocks.

4) the issues involved in the use of a memory cache per microprocessor.

5) the internal microprocessor organization necessary to implement the concept of a virtual microprocessor.

Chapter V contains an evaluation of the performance capability of a possible hardware implementation of this microcomputer architecture when executing the emulator discussed in Chapter III. This evaluation is based on statistics produced from a detailed hardware simulator which permits the varying of hardware parameters, such as the number of microprocessors, the number of busses, the interleaving of memory, the size of the cache, and the cycle times of a microprocessor, memory, or cache. This evaluation will attempt to indicate the crucial parameters that affect systems performance. Finally, Chapter VI summarizes the major results of the thesis.

II. Structure Building Language (SBL) and the Data Structure for Control (CDS)

II.1 Motivation and Important Design Considerations

The design of the SBL and its associated data structure for control is based on the view that complex emulators can be best expressed in terms of a set of (virtual) microprocessors that interact in a highly structured manner. Further, these highly structured interaction patterns (e.g., a virtual PMS environment) are different for different types of emulators. This view represents a modular, task oriented approach to managing the complexity of emulation, which is, in fact, the technique used to design sophisticated computer organizations such as the IBM 360/91, CDC 6600, BCC-500(LAM70) and the SYMBOL machine.

The CDS has been defined so as to (1) allow the flexible structuring of a virtual PMS environment, and (2) insure that the hardware algorithm for the mapping of virtual microprocessor activity to actual microprocessor activity is straightforward. The SBL microinstructions are not oriented toward specifying any particular method of microprocessor interaction patterns, but rather are building blocks on which different interaction patterns can be defined. For example, Dijkstra's semaphore (DIJ65), Saltzer's wakeup-waiting switch (SAL66) and message-queuing (SAA70,RID71) are all communication patterns that can be emulated by a short sequence of SBL microinstructions. However, SBL microinstructions are of sufficient complexity so as to provide information to the hardware mapping algorithm which allows the mapping algorithm to take advantage of similarities between the structure of the virtual PMS environment and that of the actual PMS environment. For instance, if it is desired to broadcast the same data to 64 virtual microprocessors and there are at least 64 actual microprocessors, then the mapping algorithm should be able to broadcast the data directly in one step to all 64 microprocessors, rather than sequentially transferring the data to each microprocessor.

The remainder of this chapter is divided into three sections: the Data Structure for Control, the Structure Building Language, and the Generation of the Data Structure for Control. The first section, on the CDS, describes the syntax of microprocess interaction patterns, e.g., "how" microprocesses can communicate and with "whom" they can communicate. The second section, on the SBL, describes the semantics of microprocess interaction patterns, e.g., "how" and "when" different syntactically defined interaction patterns are invoked. The third section, on the generation of the CDS by the SBL, describes how different syntactic interaction patterns are dynamically constructed.

II.2 Data Structure for Control (CDS)

The CDS defines the syntax of microprocess interconnection and interaction patterns. The CDS consists of an arbitrary number of microprocess state vectors (MSV); each MSV has a structure, pictured in Figure 9a, which has 13 components; different microprocess interaction patterns are defined by varying the number of state vectors and the values their components. Changing the values of MSV components, as will be seen shortly, changes the relationship among microprocesses.

A microprocess state vector is contained in two disjoint structures, a primary state vector (PMSV) having 7 components, and a state vector extension (EPSV) containing 6 components; these two disjoint structures are connected by a state vector extension pointer contained as a component of the primary state vector. The MSV is separated into two structures so as to allow the sharing of state vector extensions among microprocesses; the sharing of a state vector extension by two or more microprocesses defines a FORTRAN subroutine type control structure, i.e., each microprocess has its own local statically assigned storage and a common storage area for communication with other microprocesses. In addition, this separation allows the global environment within which a microprocess executes to be changed with the modification of a single pointer, i.e., the state vector extension pointer. One of the methods for microprocesses to interact is for the initiating microprocess to change the state vector

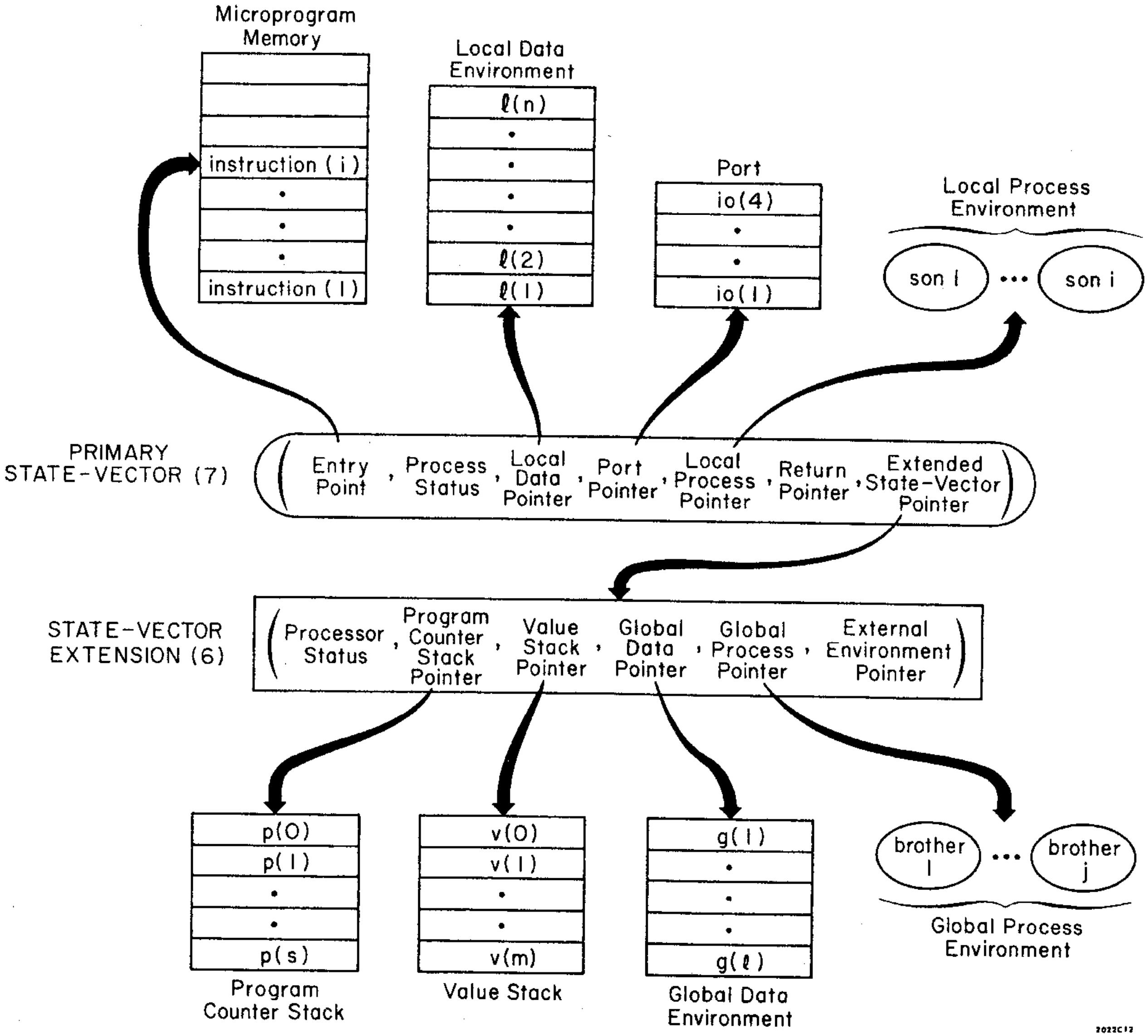


Figure 9a. Structure of Microprocess State Vector

extension pointer of the microprocess to be activated, as will be discussed more fully in the next section.

The remaining components of the microprocess state vector (MSV), for purposes of explanation, can be broken into two overlapping classes: external-environment components and internal-environment components. Each of these classes can be further subdivided into control-environment components and data-environment components. The external control-environment components define the set of microprocesses that a microprocess can directly communicate with. The external data-environment components define how other microprocesses can transfer data to a microprocess. The internal control-environment components define the local CDS for the sequencing of microinstructions of a microprocess. The internal data environment components define the internal working registers of the microprocess. Figure 9b contains a diagram of this categorization of the components of a MSV.

The values of the components of an MSV are integers, pointers to MSV's, or pointers to registers that contain descriptors of either an array of registers or an array of MSV's. The Process Space Memory holds the collection of MSV's that define the CDS as well as the registers pointed to by components of the MSV's. The MSV, together with the registers it points to, define the state image of a virtual microprocessor S(vm).

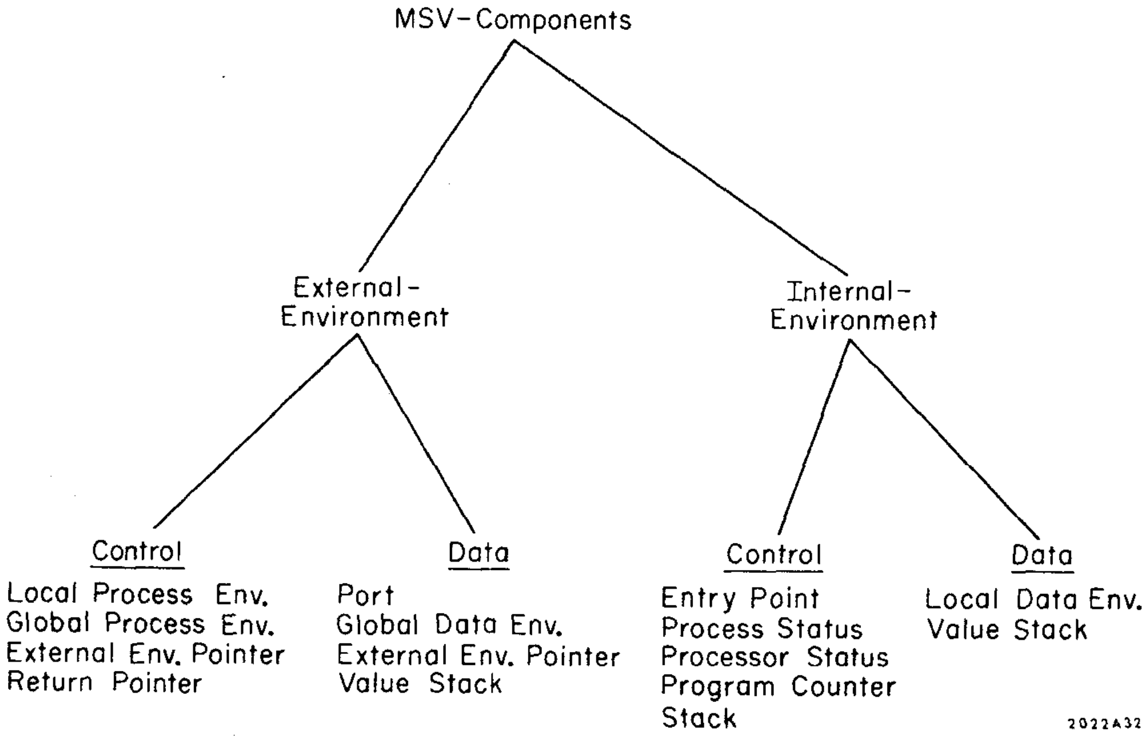


Figure 9b. Functional Classification of Microprocess State Vector Components

II.2.1 External Control Environment

There are four external control-environment components contained in an MSV: a local process environment pointer, a global process environment pointer, an extended environment pointer , and a return pointer. The first three of these configure the CDS in terms of a tree of microprocesses. In this context of a tree of microprocess state vectors, the local process environment component specifies a set of son MSV's, the global process environment component specifies a set of brother MSV's, and the external environment component specifies the father MSV. The external environment pointer provides a mechanism for tracing back up the tree so as to allow communication with the global process environment of the father, grandfather, great grandfather microprocesses, etc. The external environment pointer thus allows the nested structuring of control environments.

> Example 1: Consider the design of an emulator which works in a pipelined manner. In this pipelined emulator, there are separate, asynchronous microprocesses for fetching, decoding and carrying out the semantics of emulated instructions, for fetching and storing operands, and for controlling I/O channels. A possible CDS for this pipelined emulator is pictured in Figure 10. The microprocess computer controls the Channel-Controller microprocesses, and pipelined instruction emulator. The microprocess ISEQ (Instruction SEQuencer), IDECODE, and ITYPE-j implement, respectively, the control, decoding, and semantic processes of the pipelined emulator.

The tree of microprocesses constructed by the first three components represents static control linkages among microprocesses, whereas the

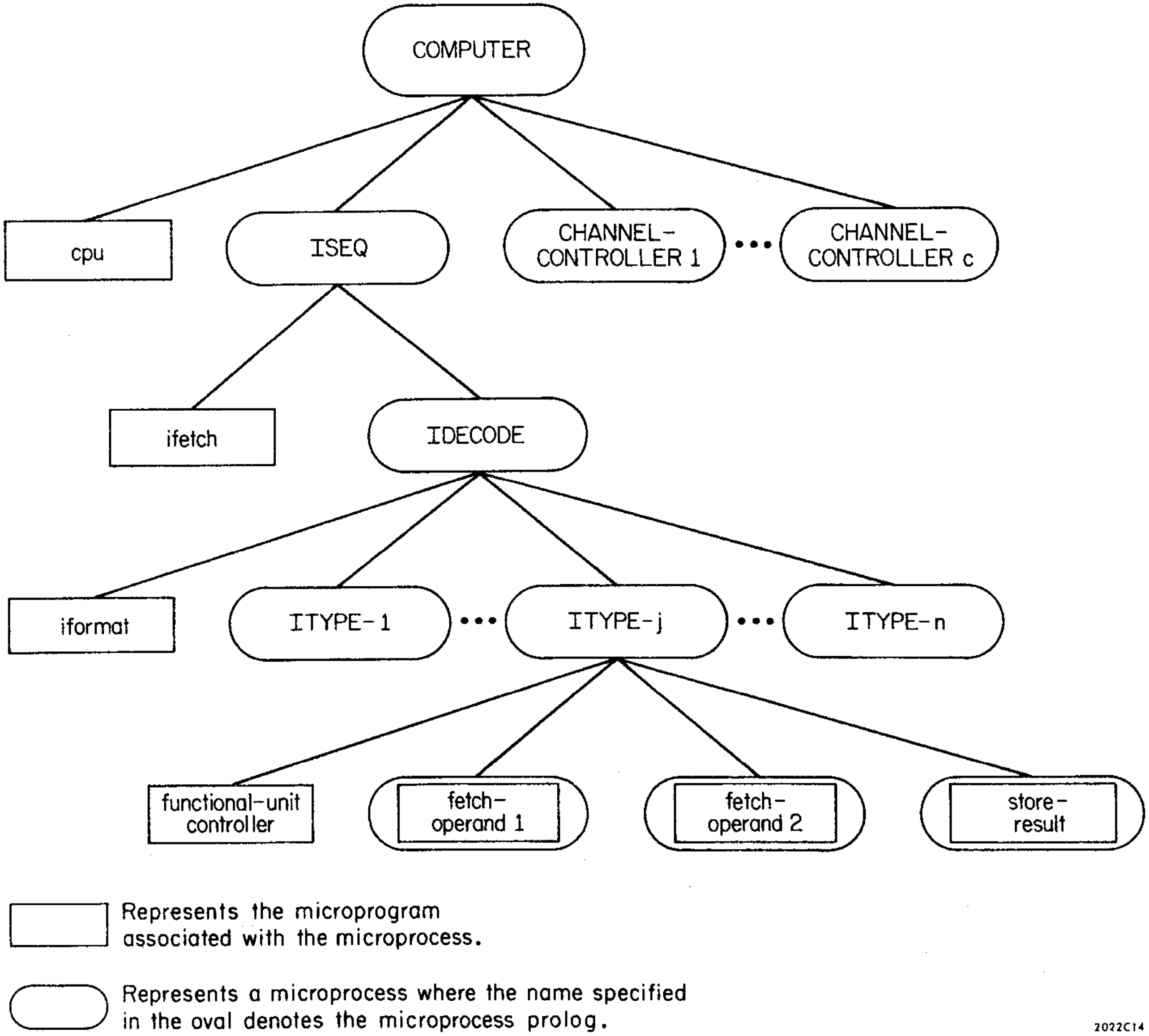

COMPUTER
cpu
ISEQ
CHANNEL-CONTROLLER 1
CHANNEL-CONTROLLER c
ifetch
IDECODE
iformat
ITYPE-1
ITYPE-j
ITYPE-n
functional-unit controller
fetch-operand 1
fetch-operand 2
store-result
Represents the microprogram associated with the microprocess.
Represents a microprocess where the name specified in the oval denotes the microprocess prolog.
2022C14

Figure 10. Control Data Structure for a Pipelined Emulator of a Conventional Computer

return pointer component represents a dynamic control linkage. These static control linkages provide a syntactic framework for the specification of dynamic sequencing among microprocesses, whereas the return pointer provides a means for specifying a dynamic control connection between the initiating and initiated microprocesses.

The CDS is in the form of a tree in order to easily specify control concepts such as hierarchical structure (functional decomposition), parallelism, coroutines, and recursion. Representation of hierarchical structure and recursion is possible because additional levels may be dynamically built in the tree by filling in the local process environment component of the MSV. Representation of parallel and coroutine structures is possible because brother MSV's in the tree may be treated as distinct, independent processes, each with its own state information. In addition, a set of brother microprocesses is a convenient framework for specifying multiple activation patterns, e.g., the 64 Processing Elements(PE) of an ILLIAC-IV can be thought of as a set of brothers which are executed together. Brother MSV's are stored in consecutive locations in the Process Space Memory. This method of storing brothers permits any brother to be accessed in one M.PSM memory reference*.

*An arbitrary size block of MSV's can be specified in terms of three parameters: the beginning of the array of MSV's, the starting address in the array, and the length of the subarray. This concise representation of a block of MSV's could be possibly used to implement efficiently a hardware broadcast operation.

A tree data structure is also a convenient syntax framework (using father, son, and brother relationships among MSV's) for defining distributed control systems. The control structure of a complex system can sometimes be conveniently represented through a hierarchical structure where in each sibling set (or structural level) of the tree there is embedded a simple control rule (via a clocking process) (HOR69) that initiates the sequencing of its son microprocesses. If additional clocking processes are contained in the sibling set, control may pass to these son microprocesses after initialization. Thus, instead of one complex control rule for the entire system, the control can be distributed throughout the system. In addition, since the control rules can be coded such that their addressing structure is not based on their absolute locations in the tree, but only on their relative position in the tree, a single microprogram could be used by clocking processes throughout the tree.

A distributed control structure can be used to define, depending upon the number of clocking processes that are simultaneously executed, either quasi-parallel (DAH66) or parallel control structures. Further, many sequential control structures can also be easily defined in terms of a quasi-parallel control structure. For example, a subroutine call mechanism can be considered a quasi parallel control structure (BIN69): the execution of the subroutine call suspends the activity of the caller and activates the called subroutine; the return from the subroutine then terminates the activity of the subroutine and reactivates the caller. The block structure and procedure calls of Algol and coroutines are other examples of sequential distributed control structures. In essence, the tree structure of the CDS allows the structure of a complex process to be functionally decomposed into a set of executions of less complex processes.

II.2.2 External Data Environment

There are four external data-environment pointers contained in a MSV: a port pointer, a global data environment pointer, a value stack pointer, and an external environment pointer. The port component, which specifies a block of up to four registers, allows the transfer of data to a microprocess to occur at the same time the microprocess is activated. In addition, the communicating microprocess does not have to know the location or structure of the port. The concept of a port allows for the construction of communication patterns where there are many possible microprocesses that may communicate and their sequence of communication is undefined. This type of communication pattern commonly occurs when a microprocess acts as synchronizing (clocking) process for asynchronously communicating microprocess e.g., Dijkstra semaphores, message queuing, etc. The port component may also be used to define broadcast control structure, e.g., multiple microprocesses having the same port.

The global data environment component, which specifies a block of registers of arbitrary length, allows the transfer of data to a microprocess to be separated from the activation of that microprocess. This type of communication pattern generally is used when 1) there are many microprocesses that access a single global data base, and 2) the values of the data base cannot be simultaneously modified by multiple microprocesses nor when other microprocesses are accessing those values.

The value stack component allows two microprocesses to communicate in a coroutine manner. This coroutine communication pattern is defined by setting the value stack components of the MSV's of both microprocessors to the same value.

The external environment pointer component, which points to an MSV, provides a mechanism for accessing the global data environments of a nested structure of microprocesses. This ability to define a nested structure of data environments is very useful in defining Algol-like (block) control structures.

> Example 2: Consider the pipelined emulator discussed in example 1. The microprocesses that make up this emulator could be structured so as to communicate with each other in two ways. One communication pattern is through a shared global data base where the frequently accessed data elements of the state image of the emulated computer are held. The other communication pattern is through the ports of each microprocessor. In particular, the microprocess, ISEQ, that asynchronusly fetches instructions transmits the fetched instructions, to the microprocess, IDECODE, through IDECODE's port. IDECODE, as will be discussed in more detail later, can define when a communication through its port from ISEQ will be consummated. In this way, the ISEQ microprocess does not have to worry about whether the IDECODE microprocess has already decoded the previously fetched instruction. However, if the fetched instruction was transmitted to the IDECODE microprocess through the

shared global data base then some explicit interlock mechanism would be needed to guarantee that IDECODE has already decoded the previously fetched instruction.

II.2.3 Internal Control Environment

There are four internal control-environment components contained in an MSV: an entry point component, a process status component, a processor status component, and a program counter stack. These four components define the microprocessor state. The entry point component specifies the beginning address of the microprogram that will be invoked when the microprocess is executed; the process status component specifies the execution status of the microprocess, and the activation-type requested of the microprocess (the process status will be discussed in detail in the next section); the processor status component specifies the internal status of the microprocessor, e.g., the condition code of the last arithmetic result; the program counter stack component specifies a block of registers that will be used as a stack to hold the microprocessor program counter when the microprocess is suspended or when a microprogram subroutine is invoked.

II.2.4 Internal Data Environment

The remaining set of internal environment components defines the internal working registers of the microprocessor. The internal data environment is specified in terms of two components: a local data environment component, and a value stack component. The local data environment component specifies a block of registers that can be directly addressed by a microinstruction. The local data environment is often used to hold data items that are not modified over repeated executions of the microprocess. Thus, it serves a function similar to the "STATIC VARIABLES" of a PL/I procedure. The value stack component specifies a block of registers that will be used as a stack to hold temporary results that are generated by the execution of microinstructions.

II.3 Structure Building Language (SBL)

The SBL consists of eight different types of microinstructions, as summarized in Table 1*. The SBL has two functions: a syntactic function and a semantic function. the syntactic function involves the dynamic construction of the CDS discussed in the previous section, while the semantic function involves the dynamic invocation of microprocess interaction patterns defined in the CDS. In essence, the syntactic microinstructions dynamically define static, time-independent interrelationships among

*In the original formulation of the SBL discussed in (LES69), there was an additional SBL semantic microinstruction: SCP (Sequential Clocking Process) which was designed to iteratively activate an array of microprocess to simulate the effect of a sequential, parallel or overlap FOR statement. This SBL statement was removed because, through a combination of ASP, SEL and IFL microinstructions, the function of SCP statement could be easily implemented without significant affect in code density nor execution speed.

microprocesses. Whereas the semantic microinstructions use these syntactic interrelationships among microprocesses as a convenient representational framework within which to define dynamic, time-dependent interrelationships among microprocesses. The semantic microinstructions are similar in function to the control statements of a conventional computer since both implement different control structures by modifying values of data elements of CDS; except, the semantic microinstructions operate in the context of CDS which can be dynamically restructured.

There is a clear distinction in the SBL between syntactic and semantic operations. This clear distinction allows semantic operations to be clearly divorced from syntactic modifications to the CDS. As will be discussed more fully in the next section, this divorce permits syntactic modification to the CDS to be generated only when absolutely necessary.

> Example 3: Consider the CDS for an IML emulator which allows fork-join type parallelism. The CDS for this emulator can be structured in two possible ways. One approach is to dynamically generate in the CDS an appropriate syntactic structure to interpret a new stream of emulated instructions every time a fork operation is emulated. The other approach is to allow only a fixed number of fork operations to be invoked at any one time; thus, a static syntactic structure can be generated, when the CDS for the emulator is initially constructed, that permits the interpretation of up to some fixed number of emulated instruction streams. In either of these approaches, the same set of semantic operations can be employed because of the clear separation of syntactic generation from semantic operations.

The microinstructions under the category "Structure Building" in Table 1 are classified as syntactic microinstructions while the remaining

Single Microprocess Activation:

ASP (Activation and Synchronization Clocking Process) a general mechanism for coordinating the activity of two microprocesses;

Multiple Microprocess Activation:

SEL (Selection and Broadcast Clocking Process) activates in a broadcast manner an array of microprocesses;

Functional-Unit/Microprocess Interaction:

FCP (Functional -Unit Clocking Process) coordinated activity of functional unit with microprocesses that generates the inputs and stores the outputs of the functional unit;

Memory/Processor Interaction:

MEM (Memory Clocking Process) fetches (or stores) a bit-string from memory-subsystem into the microprocess port.

Microprogram Invocation:

MSC (Microprogram Subroutine Call) defines the parameters for a subroutine call and then executes this call;

Structure Building:

GEN-PMSV allocates and initializes an array of primary microprocess state vectors;

GEN-EPSV allocates and initializes an extended microprocess state vector;

GEN-REG allocates a block of registers or creates a descriptor to a subarray of registers.

2022A36

Table 1: SBL (Structure Building Language) Instructions

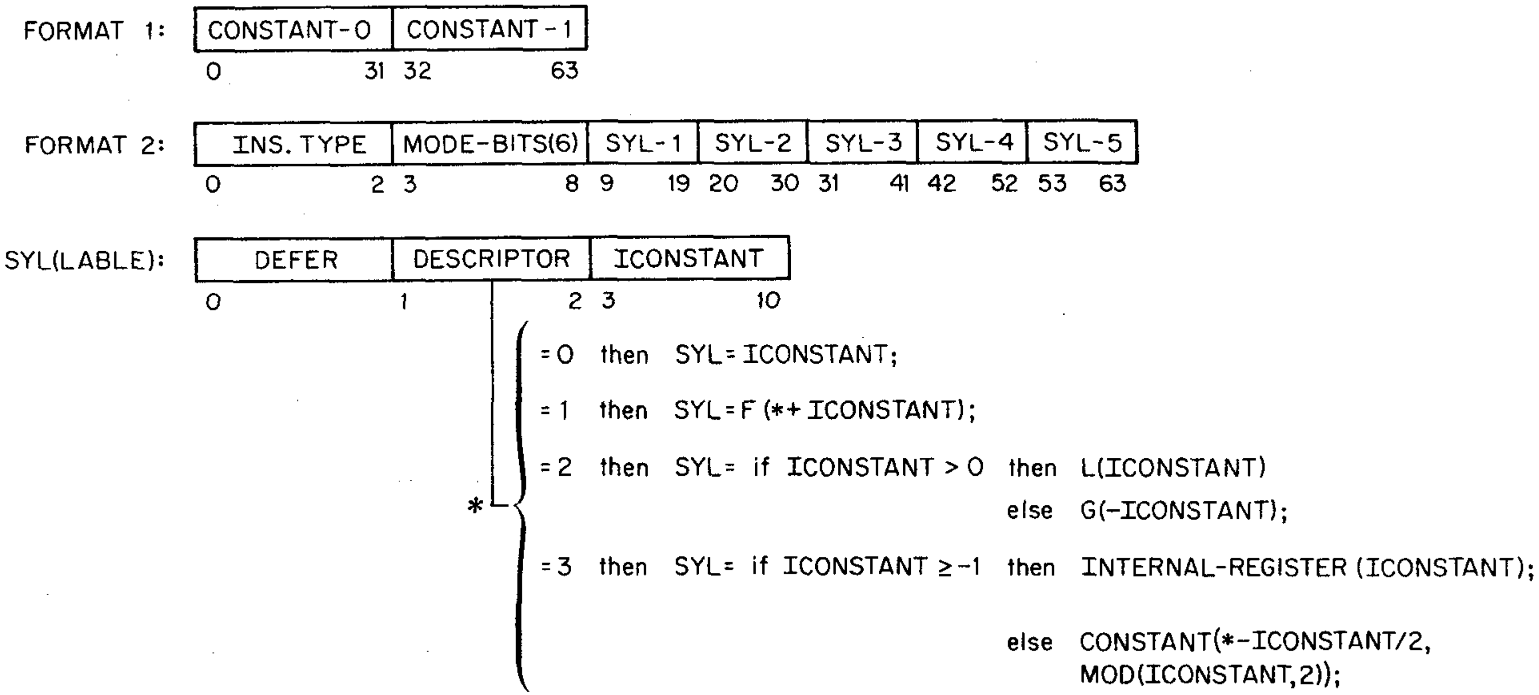
microinstructions type are classified as semantic microinstructions. The syntactic microinstructions will be discussed in the next section.

The same microinstruction internal representation is used for all SBL microinstruction types and is pictured in Figure 11. Each microinstruction word contains six mode bits and five syllables. The execution of SBL microinstructions has two phases: 1) the evaluation of the five syllables, and 2) the execution of a specific control or structure building operation based on the instruction type, computed syllable values and mode bits*.

There are three dimensions to the specifications of dynamic interaction patterns among microprocesses: "when", "who", and "how". The "when" dimension, which has not been discussed up to now since it has no syntactic component, specifies at what time points in the activity life of a microprocess can certain types of communications be received. The "when" dimension is an integral part of the specification of highly structured interaction patterns; this is especially true since 1) the built-in communication mechanism is primitive, e.g., no message queuing, and 2) it is desired to be able

---

*The SBL microinstructions have been referred to in a previous paper(LES71) as SBL macros because of their two phase execution cycle. In this context, the microinstruction type can be considered to define a control structure definitional template (prototype) that is expanded, based on the values of the syllables of a microinstruction, when a microinstruction is executed. The specification of particular values for the parameters of the template then defines a particular instance of a basic control rule or structure building operator.



*A SYLLABLE CAN BE EVALUATED IN FOUR WAYS DEPENDING UPON THE VALUE OF THE DESCRIPTOR. THE FOUR WAYS ARE: (0) AN IMMEDIATE DATA ITEM; (1) THE RESULT OF A CALL TO A MICROPROGRAM FUNCTION; (2) THE VALUE OF A REGISTER OF THE LOCAL, OR THE GLOBAL DATA ENVIRONMENT OF THE MICROPROCESS; OR (3) THE VALUE OF AN INTERNAL STATE REGISTER OR A CONSTANT STORED IN THE MICROPROGRAM MEMORY.

2022617

Figure 11. Microprogram Memory Format

to emulate many different types of communication patterns. The "when" dimension is based on the execution-state of the microprocess that is to receive the communication, and the type of communication (activation-type) desired by the microprocess that is to initiate the communication. The set of possible execution states represent the different phases in the life cycle of a microprocess. The semantics of SBL microinstructions are defined so that communication between two microprocesses is only consummated when the execution-state and type of communication (activation-type) are agreeable for communication*. The agreeable states are specified in Table 2. The set of agreeable states is designed so that a microprocess can 1) sequentially accept and process multiple communications, 2) selectively accept only certain types of communications, and 3) asynchronously accept requests for communication.

The ability to sequentialize the acceptance of multiple communications is crucial to the emulation of synchronization primitives such as the Dijkstra semaphore, message queuing, etc. This ability to sequentialize the acceptance of multiple communications, combined with the ability to transmit data to a microprocess through its port at the same time as an activation, provides a mechanism for creating a single non-interruptable data path to the controller of a

*During the period when a communication is checked whether it can be consummated and when the communication is being consummated, there are hardware locks on the Process Space Memory which guarantee that no other initiating microprocess can examine the MSV of the microprocess to be communicated to.

resource; the creation of a single non-interruptable data path to a resource controller is the basic building block of asynchronous synchronization mechanisms.

> Example 4: Consider the pipelined emulator discussed previously and pictured in Figure 10. The microprocess COMPUTER has the responsibility for controlling the microprocesses ISEQ and CHANNEL-CONTROLLER(1-c). Suppose two of the CHANNEL-CONTROLLER microprocesses are executing and when each terminates, it wants to signal this fact to the microprocess COMPUTER through an execute activation-type and also transmit the termination status of the I/O operation. In addition, suppose the execution-state of the COMPUTER microprocess is in a suspended execution-state after it has initiated the two CHANNEL-CONTROLLER microprocesses. Then the first CHANNEL-CONTROLLER that finishes will be able to consummate a communication with the microprocess COMPUTER. However, not until the microprocess COMPUTER has finished processing of the first communication and set its execution-state to suspended can the other CHANNEL-CONTROLLER initiate a communication.

The ability to selectively accept communications is important in the construction of hierarchical control structures, e.g., multiple levels of clocking (supervisory) processes.

> Example 5a: Consider the following hierarchical control structure where microprocess A is supervising microprocess B, which is, in turn, supervising microprocess C. Suppose B has initiated a communication with C and is waiting for a response from C before it continues executing microinstructions. Additionally, suppose A which controls B, happens to attempt to initiate a communication with B while B is waiting for a communication back from C. However, B may not want to accept the communication from A until the communication with C is completed. (This is especially true if C is a functional unit.) In addition, the complexity of B's coding may increase considerably if B needs to determine which microprocess, A or C, has initiated the communication, and to postpone the response to the communication if it was the inappropriate microprocess. Thus, there is needed some mechanism for B to selectively listen for only communications from C during certain time periods.

This selective listening capability is accomplished by having two execution-states that indicate a microprocess has stopped execution of microinstructions but expects to be restarted: the "waiting" and "suspended" execution-states. Corresponding to these two execution-states, there are two activation-types: execute and wakeup type activations, respectively. By observing the agreeable states in Table 2, the desired selective listening will occur when microprocess A initiates communication with B with an execute activation-type, microprocess C initiates communication with B with a wakeup activation-type, and microprocess B when it only wants to receive a communication from C places itself in the waiting execution-state.

In essence, through the use of agreeable-states approach for selective listening, a two level priority interrupt scheme is defined for message communication. The major differences between the agreeable states approach and the conventional implementation of priority interrupt schemes is that the agreeable-states approach associates priority with type of message rather than with the process generating the message, and it has only two levels of priority. Associating priority with the message rather than the process allows for the building of a hierarchy of clocking processes since a clocking process must have one message priority for communiation with its supervisory process and another for the process it supervises; additionally, interrupt schemes which have n levels of priority can be emulated by constructing multiple levels of clocking processes, each

AGREEABLE COMMUNICATION STATES

	Execution-State							
Type of Activation	UNEXPANDED 1	EXPANDING 2	EXPANDED 3	EXECUTING 4	EXECUTING SINGLE CYCLE 5	SUSPENDED 6	WAITING 7	TERMINATED 8
1 EXPAND	2							2
2 EXECUTE	2,3,4*		4			4		4 or 2,3,4++
3 EXECUTE SINGLE CYCLE	2,3,5		5			5		5 or 2,3,5++
4 WAKEUP						4 or 5+	4 or 5+	
5 SUSPEND				5				
6 RETRIEVE						6	4 or 5+	8
7 TERMINATE	8	8	8	8	8	8	8	8

LEGEND

*The number in the box refers to the new execution states, if blank than activation is not consumated.

+The new execution state depends upon the previous activation type.

++The new execution state depends upon rebuild-condition of the microprocess.

2022A33

Table 2: Agreeable Communication States

having two levels of priority. This multiple level approach is in contrast to the conventional approach of one centralized clocking process (e.g., the CPU of a computer) having n levels of priority.

> Example 5b: Consider the microprocesses COMPUTER, ISEQ and IDECODE in figure 10. Suppose the COMPUTER microprocess receives a signal from one of the CHANNEL-CONTROLLER microprocesses that an I/O operation is complete; the COMPUTER microprocess then wants to interrupt the current sequence of emulated instructions, and restart the sequencing of instructions at a different program location. The microprocess COMPUTER accomplishes this modification of sequencing by transmitting a message to the microprocess ISEQ. However, ISEQ may be waiting for a message from IDECODE specifying that IDECODE can accept another emulated instruction to decode. Thus, microprocesses COMPUTER, ISEQ, and IDECODE relate to each other in a similar manner to, respectively, microprocesses A, B and C discussed in example 5a; consequently, a similar selective listening mechanism discussed previously can be used to handle these communication requirements of a pipeline emulator.

The ability to accept requests for communication asynchronously allows for the construction of interrupt driven control structures; an interrupt driven control structure occurs when a microprocess continues to execute if no communication is pending, but if there is a communication pending, it puts itself into an appropriate state to receive the pending communication. This is in contrast to what will be called "message driven control structure" where the microprocess explicitly waits at certain time points for a communication to be received before continuing to execute microinstructions.

> Example 6: Consider the microprocesses COMPUTER and ISEQ discussed in the previous example. The microprocess ISEQ continues to fetch emulated instructions until it asynchronously receives a request for communication from microprocess COMPUTER. This communication from COMPUTER,

when accepted, specifies where ISEQ next fetches instructions from. However, ISEQ does not accept the communication until its dialogue with IDECODE is completed.

An interrupt driven control structure is constructed using the execution-state "execute-single-cycle" and activation-type "suspend". A microprocess whose execution-state is execute-single-cycle indicates that the microprocess should suspend its activity in order to receive a communication. The suspend activation-type changes the execution-state of a microprocess from execute to execute-single-cycle. Once the suspend activation-type is consummated, the initiating microprocess may then perform the desired communication. It should be noted that the suspend activation-type is only consummated when the the microprocess to be communicated with has an execute execution-state. Once the suspend type of activation is consummated, no other suspend activation-type will be consummated until the execution-state of the microprocess to be communicated with returns to execute. Thus, when the microprocess to be communicated with eventually places itself in a position to accept the communication, only the microprocess which initiated the first suspend will be able to communicate. A microprocess that is in execute-single-cycle execution-state will automatically suspend its activity at the end of a microinstruction which has an appropriate mode-bit set or the microprocess may periodically examine its execution-state to determine whether a communication is requested.

Though a use of an interrupt driver control structure has been detailed in the previous example, it is worthwhile to observe the following: in a computer system where there are many processors and the cost of an individual processor is not significant, compared to the total system cost, message driven control structures may be preferable to interrupt driven control structures because of the ease and clarity of programming, and simpler hardware. This is the case, in fact, in the CDC 6600 where the 10 PPU's control I/O devices without an interrupt structure, each PPU being responsible for controlling only a single device at a time. The use of message driven control structures is very convenient in this microcomputer architecture because a virtual PMS can be dynamically constructed which contains an arbitrary number of virtual microprocessors, each dedicated to a specific control function. In addition, the built-in hardware scheduling algorithm will automatically deallocate a microprocess from the microprocessor to which it is connected to if the microprocess is inactive.

There are three SBL microinstructions that, when executed, initiate communications among microprocesses:

1) ASP (Activation and Synchronization Clocking Process) which is used to specify a single microprocess activation pattern.

2) SEL (Select and Broadcast Clocking Process) which is used to specify a multiple microprocess activation pattern.

3) FCP (Functional Unit Clocking Process) which is used to specify a sequence of activations between a functional unit and a set of microprocesses.

The semantics of each of these microinstructions is based largely on the process-state component of the a microprocess state vector. The structure of the process-state component is pictured in Figure 12. The internal-activation, return-condition, and type-of-transfer subcomponents of the microprocess state vector are used to provide additional information to the microprocess about the type of communication initiated by another microprocess or a functional unit. The internal-activation subcomponent is used to specify one of 16 possible message types, and is generally used to indicate information about either the status of the initiating microprocess or the operation desired by a functional unit. The return-condition subcomponent is used to specify whether the microprocess should signal back to the initiating microprocess after it has terminated its activity. The type-of-transfer subcomponent specifies whether the message to be communicated is a descriptor of the desired data or the actual data which can be up to four registers long. The bussing structure of the microcomputer is designed for direct communication between microprocesses, of up to 128 bits, thus the restriction on the size of the port to four registers.

II.3.1 Single Microprocess Interaction Patterns

The ASP microinstruction is the basic building block on which complex clocking processes are built. The ASP microinstruction combines the control functions of microprocess activation, including parameter passage, and microprocess synchronization. These control functions are implemented through modifications to the MSV of the microprocess to be communicated with and that of the microprocess

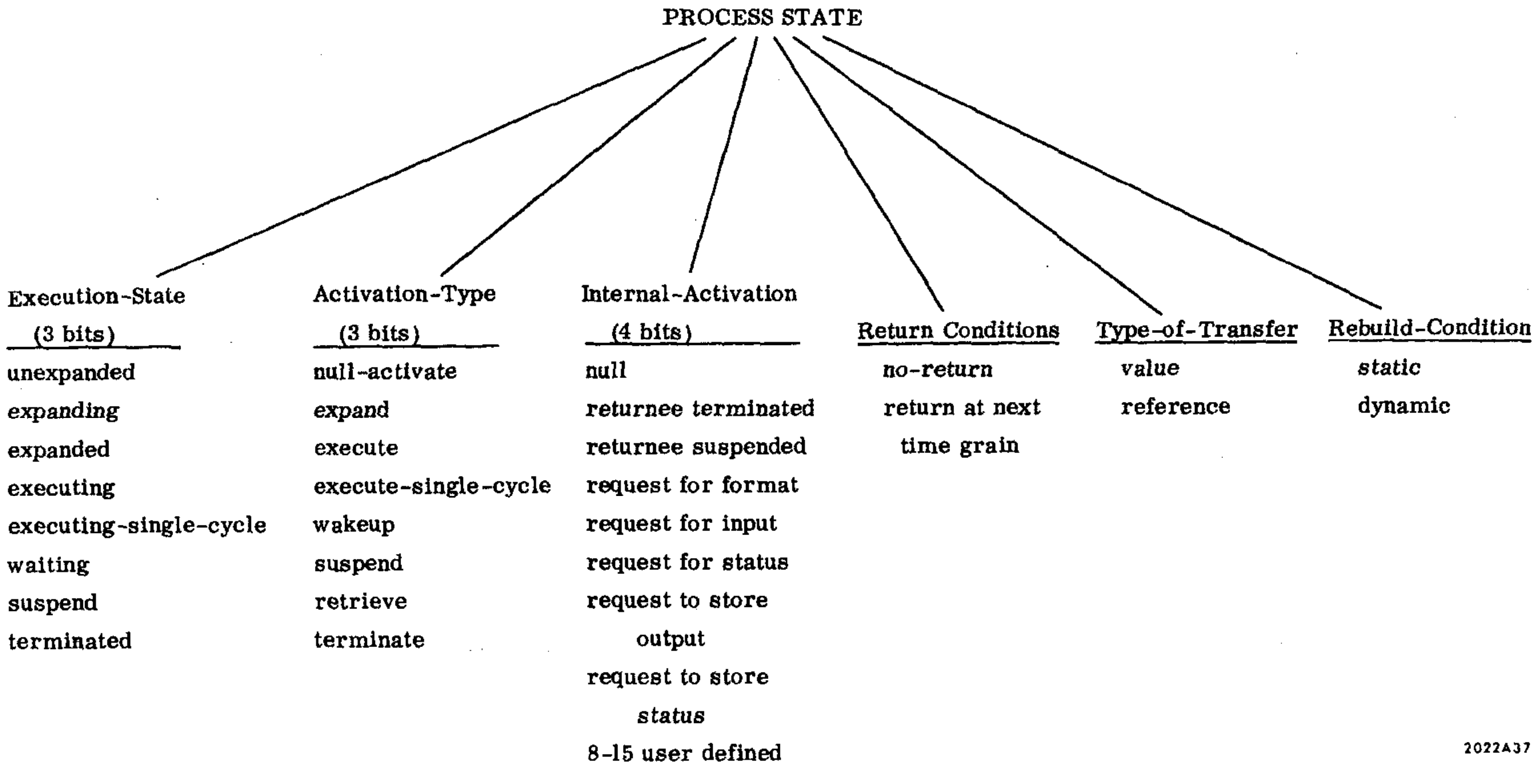


Figure 12. Substructure of Process State

which executes the ASP microinstruction. The microassembler syntax for the ASP microinstruction is the following:

```
<ASP>:=<ACTIVATE> NODE (<P1>) WITH INPUT = <P2>,
     RETURN=<P3>, EPSV=<P4>;
```

The <P1> parameter specifies the address of the MSV of the microprocess to be communicated with. The <ACTIVATE> parameter defines a new process-state component for the MSV defined by the <P1> parameter. Specifically, the <ACTIVATE> parameter specifies the activation-type, return-condition, type-of-transfer, and internal-activation subcomponents of a process-state component. The activation-type parameter is used, as previously discussed, to determine whether the communication can be consummated. If the communication cannot be consummated, then the ASP microinstruction can be repeatedly retried (e.g., busy wait) or the next microinstruction will be skipped over; this option is specified by a mode-bit in the ASP microinstruction word. In addition, if the communication is consummated, the other mode-bits of a microinstruction word specify whether the initiating microprocess will continue to execute microinstructions, or go into a waiting or suspended execution-state. This latter option provides a mechanism for synchronization of the activity of the initiating microprocess with that of the initiated microprocess. The <P2> parameter specifies either an immediate data item or a descriptor of a data item that will be transferred to the port of the called microprocess. In the case that the <P2> parameter is a descriptor, the type-of-transfer specified by the <ACTIVATE>

parameter determines whether the descriptor or the data pointed to by the descriptor will be transferred. The <P3> parameter specifies the address of an MSV to be placed in the return pointer component of the MSV of the called microprocess. The <P4> parameter specifies the address of a state vector extension (EPSV) to be placed in the state vector extension pointer component of the MSV of the called microprocess. The <P3> or <P4> parameters may be null which implies that, respectively, the return pointer or the state vector extension pointer components of this called MSV are not modified.

> Example 7: Consider the implementation of Dijkstra's P and V semaphore operations in terms of ASP microinstructions. Let PV be a microprocess that implements the P and V semaphore operations. A microprocess M performs a P operation by executing the following ASP microinstruction:
>
> ```
> EXECUTE (BUSY_WAIT, WAIT_RESPONSE, VALUE,
> ACT_CODE=9) NODE(L(1)) WITH INPUT=S,
> RETURN_ADDRESS=P(SELF);
> ```
>
> where L(1) is a local data register of M that contains the address of microprocess PV, and S is the top of the value stack that contains the descriptor of the semaphore variable. This ASP microinstruction initiates a communication with the PV microprocess with an execute activation-type, and internal-activation Code equal to 9. The internal-activation code is used to distinguish between a request for a P or V operation. In addition, it transmits a descriptor of the semaphore variable to PV's port and modifies the return pointer component of PV's MSV to point to the microprocess M. After the communication has been consummated, the execution-state of M is set to waiting. If the ASP microinstruction cannot consummate the communication, then the ASP microinstruction will be retried until consummation. However, the busy-wait is not on the semaphore variable but only on the microprocess which updates the semaphore. In addition, the hardware scheduling algorithm in this busy-wait situation will, if there are other uses for the microprocessor executing the microprocess M, reschedule M to run at some later time. The PV process, when activated for a P operation, checks whether the semaphore variable can be decremented; if it can, then the microprocess M is restarted by the following

microinstruction:

```
   WAKEUP(SUSPEND) NODE(P(RETURN));
```

and then the PV microprocess is suspended until another request is received. However, if the semaphore variable cannot be decremented, then the PV microprocess places the address of M in a queue associated with the semaphore variable, and then suspends itself without restarting M. In addition, if there is no more queue space in the data environment of PV to hold addresses of blocked microprocesses, then PV will put itself in the execution-state waiting rather than that of suspended. As will be seen shortly, if PV is in execution-state waiting then only V type operations can be consummated which do not make any more demands for queue space (e.g., a nice use of selective listening). A V semaphore operation is specified by the following ASP microinstruction:

```
   WAKEUP(BUSY_WAIT, CONTINUE, VALUE, ACT_CODE=10)
   NODE (L(1)) WITH INPUT=S;
```

This ASP microinstruction initiates a communication with the PV microprocess with a WAKEUP activation-type, and internal-activation code equal 10. In addition, it transmits a descriptor of the semaphore variable to PV's port. After the communication has been consummated, the M microprocess continues to execute microinstructions. Thus, the V operation goes on in parallel with execution of microinstructions of M. The PV microprocess, when activated for a V operation, increments the appropriate semaphore variable, and checks whether there is a queued microprocess waiting on that semaphore variable. If so, an ASP microinstruction detailed previously, is used to restart the queued microprocess.

II.3.2 Multiple Microprocess Interaction Patterns

The SEL microinstruction is the basic building block of multiple microprocess activation patterns. These patterns include those generated by control structure concepts such as lock-step execution as used in the ILLIAC-IV, fork-join type parallelism, etc. The SEL microinstruction activates in a broadcast manner a selected subarray of microprocesses and then waits for an arbitrary number of these microprocesses to signal completion. A microprocess generally uses this microinstruction to control its son microprocesses. The microassembler syntax for the SEL microinstruction is the following:

```
<SEL>:=<ACTIVATE> <P1> SONS STARTING AT SON (<P2>)
              WITH INPUT=<P3> THEN WAIT
              FOR <P4> SONS TO SIGNAL RETURN;
```

The <P1> parameter specifies the number of sons the activation is broadcast to. The <P2> parameter specifies the number of the first son in the broadcast array. In essence, the <P1> and <P2> parameters select a subarray of son microprocesses to initiate a communication with. The <ACTIVATE> parameter, which has an identical interpretation to the <ACTIVATE> parameter of the ASP microinstruction discussed previously, specifies the type of communication to be broadcasted to the subarray of microprocesses. The <P3> parameter specifies the data that is to be broadcast to the ports of these microprocesses. The <P4> parameter specifies the number of return signals to wait for, before executing the next microinstruction.

Example 8: Consider the CDS pictured in Figure 13. The CONTROL-1 microprocess implements the FORK-JOIN operation on microprocesses A and B through the execution of the following SEL microinstruction:

```
EXECUTE 2 SONS STARTING AT SON(1)
THEN WAIT FOR 2 SONS TO SIGNAL RETURN;
```

The SEL microinstruction has been designed based on the view that the control primitives of 1) broadcasting data to n

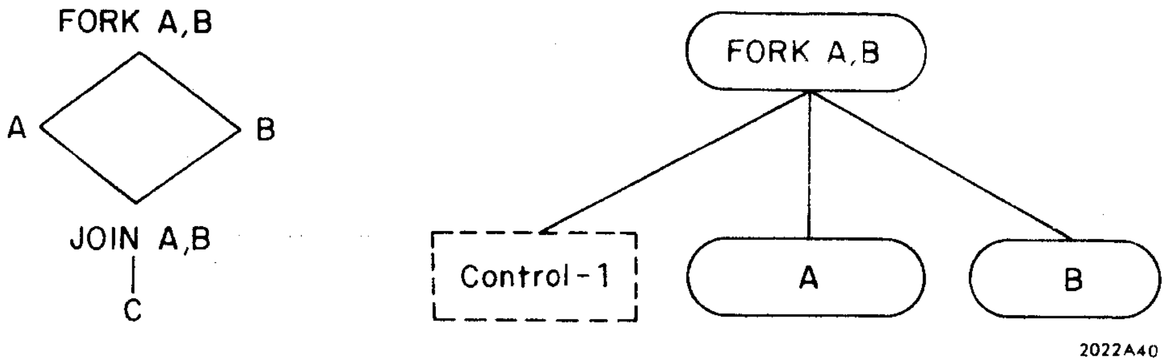


Figure 13. Control Data Structure for a FORK Control Structure

microprocesses, and 2) waiting for k return signals are fundamental to the efficient implementation of highly structured parallel control structures. By imbedding these control primitives in the semantics of a microinstruction, it provides information to the hardware scheduler and bus-controller which can be used to more efficiently map parallel activity on the virtual PMS onto the physical PMS*.

II.3.3 Microprocess/Functional-Unit Interaction Patterns

The FCP microinstruction is used in the microprogramming of the semantics of I/O control structures; I/O devices in future discussions will be referred to as functional units. The microcomputer architecture can contain an arbitrary set of functional units**. Each of these units can be independently activated and can have an arbitrary number of inputs and outputs, where that number need not be fixed and may be data dependent. For example, a functional unit could be a floating-point multiplier, or more generally, an arbitrary input/output device such as a disk controller. A functional

*The use of this information has been postulated and proposed techniques developed, however, these proposed techniques are not incorporated into the computer organization to be discussed in Chapter IV. Especially interesting is how to design a bussing structure so that broadcast operations on the virtual PMS can be directly mapped on to the physical PMS when microprocessors are already connected to the microprocesses that will receive the broadcast operations.

**The hardware scheduling mechanism used to schedule multiple microprocessors is also used to schedule identical functional units.

unit can receive input data from three sources: the memory subsystem, another functional unit, or the microprocessor subsystem. A functional unit obtains (and stores) data by requests to the microprocessor subsystem, which has complete responsibility for determining the source (or sink) of the data that are requested and for generating the appropriate control signals to accomplish the data transfer. In this manner, the microprocessor subsystem acts as a generalized I/O controller and separates the process of data accessing from that of computation. The idea of a generalized I/O control structure to control arithmetic units has been proposed in an earlier paper by the author(LES68) and by Lass(LAS68), as a basis of the design of a high-speed computer.

The FCP microinstruction performs the following functions: 1) creates a connection between a functional unit and the microprocess executing the FCP microinstruction; 2) activates the connected functional unit with control information specifying the desired operation; and 3) controls the generation of input and output data sets for the connected functional unit. The connected functional unit can be a physical functional unit or a virtual function unit, i.e., a microprocess programmed to behave like a functional unit. The input and output data sets are generated by the son microprocesses of the microprocess executing the FCP microinstruction. In essence, the FCP microinstruction is a clocking process which controls the interaction between the functional unit and the son microprocesses that fetch the inputs for the functional unit, and stores its outputs. A functional unit requests a particular service from its clocking process by communicating with the clocking process through a wakeup activation-type. The particular service requested is specified by appropriately setting the internal-activation code (see Table 2). The FCP microinstruction has been parameterized so that both simple and complex interaction (handshaking) patterns between a functional unit and its input and output generating microprocesses can be specified in a uniform framework.

The microassembler syntax for the FCP microinstruction is the following:

```
<FCP>:= ACTIVATE FUNCTIONAL-UNIT (<P1>) WITH
        CONTROL-INFORMATION = <P2> USING <P3>
        INPUT-GENERATORS INITIATED BY <ACTIVATE>
        COMMAND AND STORE STATUS IN <P5>.
```

The <P1> parameter specifies the number of a functional unit or the address of a microprocess. The <P2> parameter specifies control information to be transferred to the functional unit when initially connected. The <P3> parameter specifies the number of son microprocesses that will act as input generators, starting from the first son; the remaining son microprocesses are used as output generators.

Example 9a: Consider the ITYPE-j microprocess in Figure 10, where the microprogram FUNCTIONAL-UNIT CONTROLLER is just a single FCP microinstruction. The <P3> parameter, in this case, would have the value 2 to indicate microprocesses FETCH-OPERAND 1 and FETCH-OPERAND 2 would generate the input data for the functional unit, and microprocess STORE-RESULT would store its output data. In addition, suppose the functional unit that is being controlled can perform a

> floating-point add, a floating-point subtract or a floating-point multiply operation, then the <P2> parameter would specify which operation would be desired.

The son microprocesses that generate the input data set are sequentially activated from the first input son to the last input son to generate input data; an analogous activation pattern is used for activating the son microprocesses that store the output data. The <ACTIVATE> parameter specifies the mode for activating the input and output generator microprocesses, e.g., EXECUTE, EXECUTE-SINGLE-CYCLE, RETRIEVE, and the mode of transferring (and fetching) data to (and from) microprocess, e.g., VALUE or REFERENCE. The EXECUTE mode indicates only one input or output value will be handled by each input and output generator, while the EXECUTE-SINGLE-CYCLE mode implies many values can be handled; the RETRIEVE mode indicates the desired input data has already been prefetched and resides in the microprocess's port; the VALUE mode indicates the FCP microinstruction will handle the transfer of data between the functional unit's port and son microprocess's port; while the REFERENCE mode indicates that son microprocess will directly handle the transfer of data between ports. The interaction patterns defined by these different modes are detailed in Tables 3 and 4, and Fiqures 14 and 15. The <P5> parameter specifies a descriptor of a register (or the address of a microprogram to be invoked) that status data is retrieved and stored from. This ability to invoke a microprogram to store the status information provides a convenient mechanism for the monitoring of special conditions.

1) Device requests clocking-process to generate an input;*

2) Clocking-process activates next available input-generator microprocess to generate an input value;
   - a) Clocking-process transfers address of device to input-generator's port;

3) Input-generator signals clocking-process that input is in the input-generator port;
   - a) and whether more data can be generated;

4) Clocking-process transfers data from input-generator port to device's port.

5) Clocking-process updated address of next available input-generator.

*Device can request next input as soon as the previously requested data has been stored in its port.

Input Modes	Phases Required
EXECUTE (VALUE)	Step 1, 2, 3, 4, 5
EXECUTE (REFERENCE)	Step 1, 2a, 5
RETRIEVE	Step 1, 4, 5
EXECUTE_SINGLE_CYCLE (VALUE)	Step 1, 2, 3, 3a, 4, 5
EXECUTE_SINGLE_CYCLE (REFERENCE)	Step 1, 2a, 3a, 5

2022A34

Table 3: Input Request Protocol for Control of Functional Unit

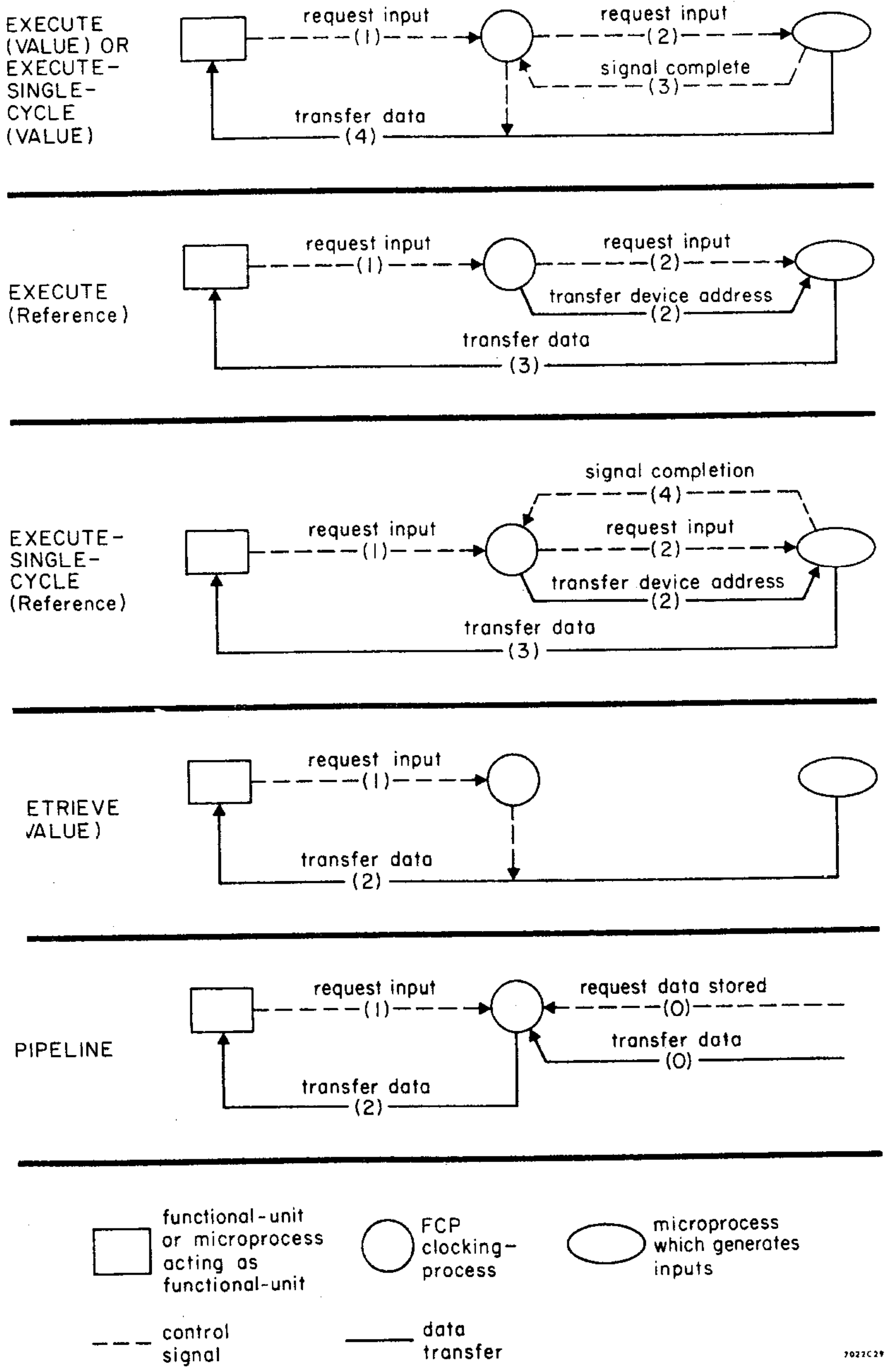


Figure 14. Schematic of Input Request Protocol

1) Device requests clocking-process to store output;*

2) Clocking-process requests output-generator node to store output;

   a) clocking-process transfers address of device to output generator's port;

   b) clocking-process initiates transfer to data in device's port to output-generator's port;

   c) clocking-process initiates transfer of data to its own port;

3) Output-generator signals clocking-process that it has completed transfer, and that it can or cannot accept more data;

4) Clocking-process updates address of next available output-generator.

*Device can request to store next output as soon as data is removed from its port.

Possible Output Modes	Phases Required
EXECUTE (VALUE)/RETRIEVE	1, 2, 2b, 4
EXECUTE_SINGLE_CYCLE (VALUE)	1, 2, 2b, 3, 4
EXECUTE (REFERENCE)	1, 2, 2a, 4
EXECUTE_SINGLE_CYCLE (REFERENCE)	1, 2, 2a, 3, 4
PIPELINE	1, 2c

2022A35

Table 4: Output Request Protocol for Control of Functional Unit

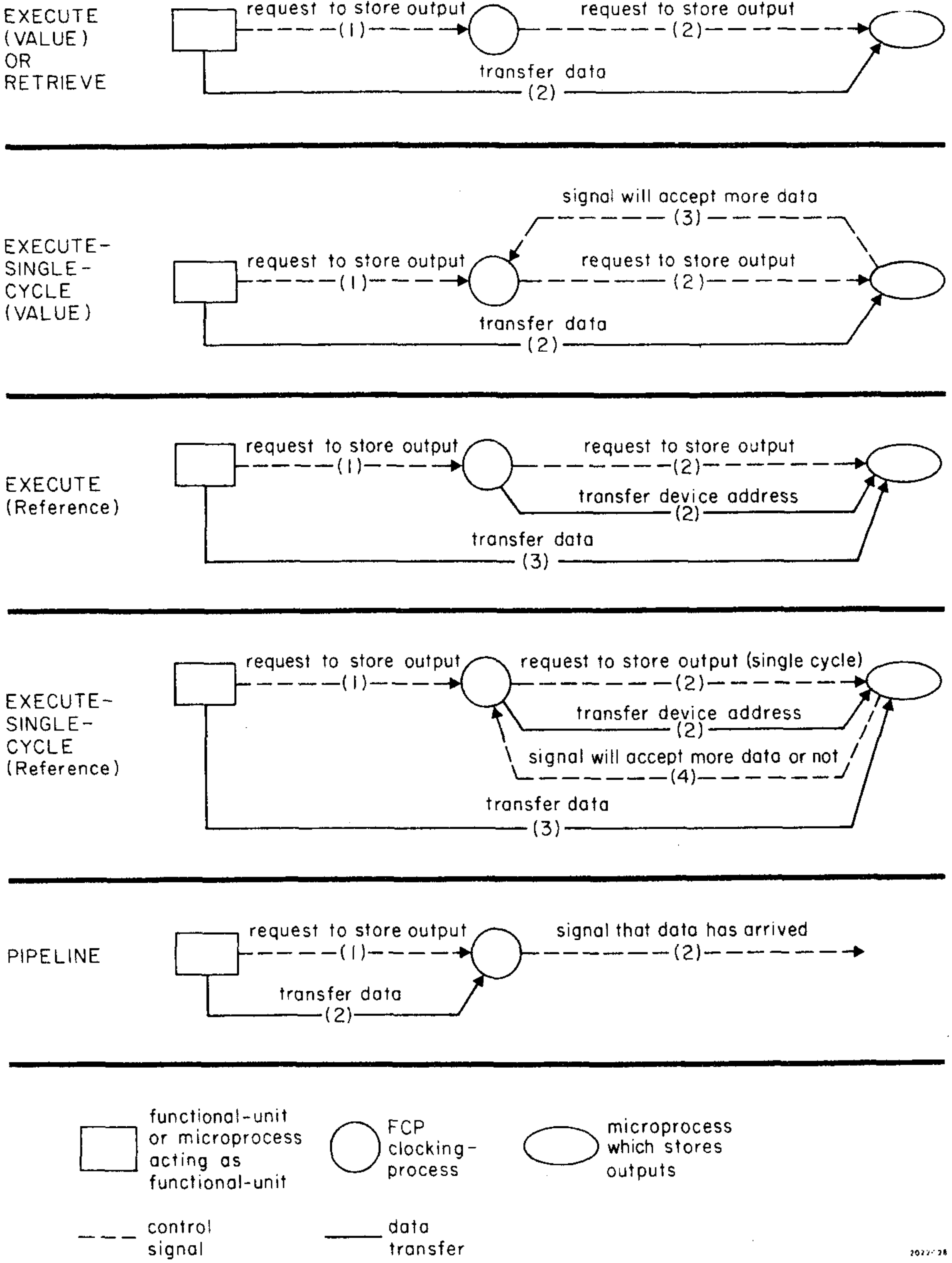

EXECUTE (VALUE) OR RETRIEVE
request to store output (1)
request to store output (2)
transfer data (2)
EXECUTE-SINGLE-CYCLE (VALUE)
signal will accept more data (3)
request to store output (1)
request to store output (2)
transfer data (2)
EXECUTE (Reference)
request to store output (1)
request to store output (2)
transfer device address (2)
transfer data (3)
EXECUTE-SINGLE-CYCLE (Reference)
request to store output (1)
request to store output (single cycle) (2)
transfer device address (2)
signal will accept more data or not (4)
transfer data (3)
PIPELINE
request to store output (1)
signal that data has arrived (2)
transfer data (2)
functional-unit or microprocess acting as functional-unit
FCP clocking-process
microprocess which stores outputs
control signal
data transfer
2022-28

Figure 15. Schematic of Output Request Protocol

> Example 9b: Consider again the microprocess ITYPE-j discussed in the previous example, except this time with a FUNCTIONAL-UNIT CONTROLLER microprogram which is more complex. In this case, it is desired not to connect (reserve) the floating-point functional unit until the input operands have arrived from the Memory Subsystem; this type of functional unit control scheme, commonly called reservation station concept, is used in pipelined computers such as the IBM 360/91 where there may be many instructions competing for the same resource; this control scheme can be implemented by two SBL microinstructions, the first an SEL microinstruction activates simultaneously microprocesses FETCH-OPERAND 1 and FETCH-OPERAND 2, and then waits until both microprocess signal completion; the two input microprocesses,when they are complete, leave their fetched data in their ports. The second SBL microinstruction would be an FCP microinstruction parameterized as in Example 3, except with its <ACTIVATE> parameter specifying a RETRIEVE operation.

The FCP microinstruction semantics clearly separate the operation of data accessing from the computational algorithm requesting the data. This separation facilitates the definition of control structures that 1) directly emulate different types of IML instruction formats, e.g., one address, two address, etc.; 2) specify dynamic data interconnection patterns among functional units*, e.g., a pipeline of functional units, a tree of functional units, etc.; and 3) allow the incorporation of functional units into the functional unit subsystem that have complex input and output requirements, e.g., a matrix multiply unit.

*The method generating a CDS for these alternative functional unit control structures is discussed more fully in an earlier paper by the author (LES70).

II.3.4 Microprocess/Memory Subsystem Interaction Patterns

The MEM microinstruction defines an access path between a microprocess and the Memory Subsystem. This path is used to fetch (store) data into (from) the microprocess's port from (into) the Memory Subsystem. The Memory Subsystem is bit addressable, and can be activated to store or retrieve a bit string of up to 128 bits. The memory subsystem is bit addressable so as to simplify the embedding of the state image of an emulated machine S(e) into the virtual state image of the microcomputer S(vm), and to allow in a single memory operation the fetching of the appropriate unit of data to be worked on. Once the data has been fetched into the working registers of the microprocessor, IFL masking operations can be used to perform bit extraction on 32 bit length registers.

> Example 10: Consider an emulated computer whose instruction word is 48 bits long with an opcode field in the first six bits. A single MEM operation is used to extract the desired instruction from the Memory Subsystem into two consecutive, 32 bit working registers of the microprocess. The opcode of the emulated instruction is then extracted, by a single IFL SHIFT_MASK operation (See Table 6), from the first working register of the microprocess's port.

This two level bit extraction scheme represents a compromise among the high overhead cost of accessing a bit-addressable memory, the ease of programming and the hardware efficiency of internal microprocessor operations on fixed size data elements.

The MEM microinstruction is parameterized so as to allow for the automatic hardware mapping of the address space of a wide variety of emulated machines directly into the physical address space in the Memory Subsystem. This capability permits addresses in the address space of the emulated machine to be directly manipulated without first converting these addresses into addresses in the address space of the microcomputer. This automatic hardware mapping capability of the MEM microinstruction represents how the concept of virtual memory can be used in the context of emulation.

The microassembler syntax for MEM microinstruction is the following:

```
<MEM>:= "READ/STORE" ELEMENT (<P1>) WITH FORMAT
        = <P2> AND LENGTH = <P3>) "FROM/INTO"
        MEMORY ARRAY (DESCRIPTOR = <P4>,
        OFFSET=<P5>);
```

The <P4> parameter specifies a descriptor of a memory array (i.e., (1) base, (2) dimension, and (3) size of data element) in the Memory Subsystem, and the <P5> parameter specifies an offset quantity with respect to the base of this memory array. These two parameters define the physical address space in the Memory Subsystem that the virtual address will be mapped into. The <P1> parameter specifies an index of a data element in the offset memory array, i.e., the address of the data element in the emulated machine space; the <P3> parameter specifies the length of the data element, e.g., for an IBM 360 whether data is 1, 2, 4 or 8 bytes long; and the <P2> specifies the format of the data element. These five parameters are used to map a data item in the virtual address space into a data item in the physical address memory space. This mapping function defines a bit string of, P3 x (P4:size of data element), bits starting at bit address, (P4:base) + (P1+P5) x (P4:size of data element) in the Memory Subsystem. If this bit string lies outside the bound of the memory array, a condition code is set in the processor status, and the memory operation is bypassed.

The MEM microinstruction may also be used, depending upon the process-state, to fetch or store the data for a functional unit. In this mode of operation, the format and length parameter is transmitted along with the data to the functional unit. The use of a formal field in the specification of both the input and output data allows the functional unit to be very sophisticated in being able to perform, if desired, arithmetic operations involving operands and results of different types and lengths(HAU68).

II.3.5 Microprogram Invocation

The MSC microinstruction performs a microprogram subroutine call. The MSC microinstruction is mainly used in conjunction with the GEN_EPSV microinstruction to define what will be called a microprocess prologue (discussed in the next section). The MSC microinstruction in the prologue context is used to initialize the data environment of microprocess before the execution of the main microprogram body of the microprocess. The microassembler syntax for the MSC microinstruction is the following:

```
<MSC>:= INVOKE MICROPROGRAM (<P1>) WITH
        PROCESSOR_STATUS=<P2>, INDEX=<P3>,
        VALUE_STACK=<P4>, INITIALIZE_ROUTINE
        =<P5>;
```

The <P1> parameter specifies the address of a microprogram subroutine. The remaining parameters initialize the data environment before invoking of the microprogram. The <P2> parameter specifies the initial setting of the condition codes specified in the processor-status. The <P3> parameter specifies an initial value for the index register. The index register is not specified in terms of a unique register location in the MSV, but rather is pushed and popped on the program counter stack at the same time the program counter is pushed or popped. The <P4> parameter specifies a value for the top of the value stack and the <P5> parameter specifies another microprogram which is invoked to initialize the local data environment and miscellaneous working registers.

II.4 Generation of Data Structure for Control

The CDS is dynamically generated in the form of a tree of microprocesses through the execution of syntactic SBL microinstructions; however, this method of generating the CDS does not necessarily reflect the dynamic activity patterns of microprocesses. The separation between the generation and sequencing of microprocesses is possible because the execution of a microprocess is factored into three discrete, separable phases: a binding phase, an expansion phase, and an activation phase. The generation of a CDS caused by the binding and expansion phases can thus be separated from the sequencing of a CDS caused by the activation phase, detailed in the previous section. This separation is extremely important because once the overhead cost has been incurred for defining the CDS, there is little overhead cost for each dynamic interaction pattern invoked.

The binding phase of a microprocess involves the generation and storage in the Process Space Memory of the microprocess state vector that defines the microprocess. At the completion of the binding phase, the microprocess's execution-state is set to unexpanded. For example, the binding phase analog for an ALGOL procedure is the allocation of memory for the local variables of a procedure, and the setting up of the static linkage pointers required for uplevel addressing in the block structure.

The expansion phase of a microprocess involves the generation of the microprocess' local process environment, e.g., its son microprocesses, the initialization of its working registers, and the specification of the microprocess entry point e.g., the initial value of the microprogram counter. Thus, the expansion phase of a microprocess results in the completion of the binding phase of its son microprocesses whose expansion will, in turn, generate additional microprocesses. This recursive sequence of microprocess binding and expansion phases leads to a dynamic tree generation mechanism for constructing the CDS. This dynamic mechanism for generating the CDS is somewhat similar to the concept of run time macro expansion or dynamic compilation. For example, the expansion phase of an ALGOL procedure would be the generation of procedure substructure and the initialization of the local variables of the procedure.

The activation phase, which has been discussed in the last section, involves the execution of microinstructions starting at the specified entry point. For example, the activation phase analog in an ALGOL procedure is the transferring of parameters to the procedure and the execution of machine code for the procedure.

> Example 11: Consider the CDS pictured in Figure 13; the three phases of execution of microprocess FORKA,B are pictured in Figure 16. The binding phase results in the generation of microprocess state vector that defines the microprocess FORKA,B. The expansion phase results in the generation of the son microprocesses A and B. Finally, the activation phase results in the execution of microprogram CONTROL-1 which contains the SEL microinstruction that performs the FORK-JOIN control operation.

The expansion phase of a microprocess, which is indicated by execution-state "expanding", occurs when a microprocess is in an "unexpanded" or "terminated" execution-state and is initiated with an activation-type of expand, execute or execute-single-cycle. The expand activation-type explicitly separates the expansion phase from the activation phase; this separation permits the expansion phase to be initiated with different parameters than the activation phase. In addition, based on the rebuild-condition subcomponent of a MSV, part or all aspects of the expansion phase can be bypassed when a

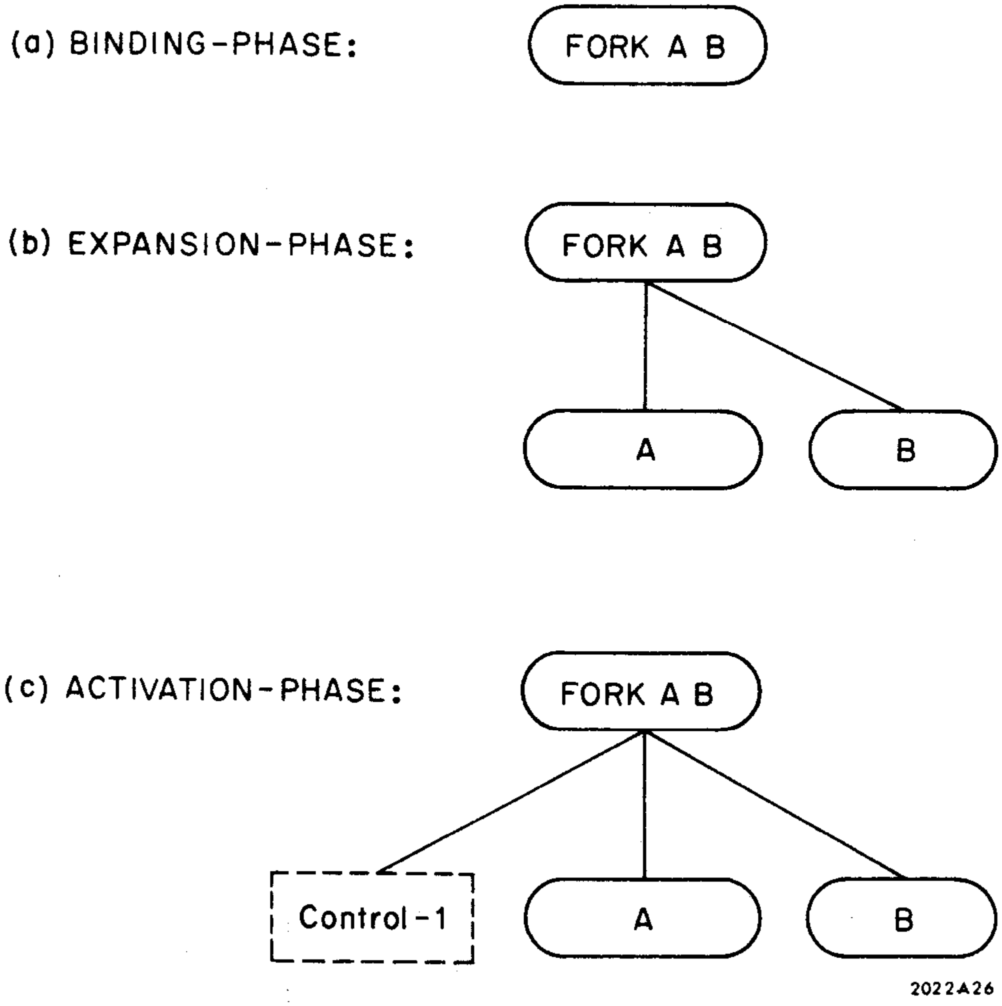


Figure 16. Three Phases of Microprocess Execution

microprocess is re-expanded when in the "terminated" execution-state. This ability to control the activity of the expansion phase permits the rebuilding of the CDS only when the microprocess's substructure will vary on repeated executions of the microprocess. Thus, the static aspects of the CDS once defined need not be regenerated.

The expansion phase is specified in terms of a set of microinstructions that will be called the microprocess prologue. The starting address of the microprogram that defines the prologue is specified by the entry point component of a MSV. The microprocess prologue consists of a two microinstruction sequence: a GEN_PMSV microinstruction followed by a MSC microinstruction. The execution of the GEN_PMSV microinstruction will generally, in turn, result in the execution of the two other types of syntactic microinstructions: GEN_EPSV and GEN_REG. The GEN_PMSV microinstruction, when executed as part of the microprocess prologue, fills in the local process environment component of a MSV with a pointer to a descriptor of an array of MSV's. This array of MSV's defines the son microprocesses. In addition, one of the mode bits of the GEN_PMSV microinstruction specifies the rebuild-condition of the microprocess: static or dynamic. If the microprocess is re-expanded at some later time and the rebuild-condition is set to static, then the GEN_PMSV microinstruction of microprocess prologue will be bypassed. Thus, the microprocess substructure will not be rebuilt in this case.

The MSC microinstruction, when executed as part of the prologue, is only partially executed during the expansion phase; only the syllables of MSC microinstruction that do not have their defer bit set will be evaluated. The syllables of the MSC microinstructions that are deferred will be evaluated only when microprocess begins its activation phase. This two step evaluation scheme provides a mechanism for selective initialization of the environment of the microprocess each time it is executed. For example, the microprocess entry point address could be recalculated for each execution of the microprocess or calculated once for all executions. In the same manner, the local data environment could be re-initialized for each execution or initialized only once when the MSC microinstruction is executed for the first time. Thus, the concept of microprocess prologue provides a simple mechanism for specification of the particular initialization sequence required by a microprocess.

There are three syntactic microinstructions: GEN_PMSV, GEN_EPSV, and GEN_REG. The GEN_PMSV microinstruction allocates and initialzes an array of primary microprocess state vectors in the Process Space Memory. In addition, at the completion of the execution of a GEM_PMSV microinstruction, a descriptor (or pointer to the register containing the descriptor) for this PMSV array, is placed on the top of the value stack of the microprocess that executed this microinstruction. As mentioned previously, if GEN_PMSV is executed as part of the microprocess prologue, the local process environment subcomponent is initialized with a pointer to the descriptor of the array of MSV's. The GEN_PMSV does not have to bc executed in the context of a microprocess prologue. Thus, alternative means of generation of a CDS can be programmed instead of the tree generator scheme previously described. However, when an alternative means of generation is used, there are no automatic mechanisms for specifying when structures will be rebuilt or reinitialized. The microassembler syntax for the GEN_PMSV microinstruction is the following:

```
<GEN_PMSV>:= S= SUBSTRUCTURE CONTAINS <P1> SONS
             WITH PROGRAM = <P2>, PORT = <P3>,
             LOCAL_DATA = <P4>, EPSV=<P5>;
```

The <P1> parameter specifies the number of elements in the array of primary microprocess state vectors. The other four parameters define the value of, respectively, the entry point, the port, local data environment, and extended process state vector pointer components of a PMSV. These parameters are re-evaluated for each PMSV in the array if the defer bit of the syllable associated with parameter is set to 1. Otherwise, the same parameter value is used to initialize all PMSV's of the array.

The GEN_EPSV microinstruction allocates and initializes an extended state vector in the Process Space Memory. At the completion of its execution, a pointer to the EPSV is placed on the top of the value stack. The microassembler syntax of the GEN_EPSV microinstruction is the following:

```
<GEN_EPSV>:= S= P(EPSV WITH GLOBAL_DATA=<P1>,
                GLOBAL_PROCESS=<P2>
                VSTACK=<P3>,
                PSTACK=<P4>,
                EXT_ENV=(<P5>));
```

The five parameters, respectively, define pointers to the global-data environment descriptor, global-process environment descriptor, value stack descriptor, program counter stack descriptor and a pointer to PMSV that defines the first level of extended environment. If a parameter is left out then value of the corresponding component of EPSV of the microprocess executing this microinstruction is used.

The GEN_REG microinstruction can be used either to allocate an array of registers in the Process Space Memory or to create a descriptor for a subarray of registers which have already been allocated. In addition, the GEN_REG microinstruction can be used to create a descriptor for an array of words in the Memory Subsystem. At the completion of the execution of a GEN_REG microinstruction, a descriptor (or pointer to the register containing the descriptor) for the array of registers is placed on the top of the value stack. There are three types of register descriptors: a register-block descriptor, a stack descriptor, and an I/O descriptor. Each of these descriptors specify the base and dimension of a block of registers in the Process Space and information which specifies how the block of registers can be accessed. the access-control information specifies an access mode attribute, e.g., read, read/write, or write, and a "sharability" attribute, e.g., local, coroutine or global. The "local" attribute indicates that only one microprocess will ever access this data, the "coroutine" attribute indicates that only one microprocess at a time will access the data, and the "global" attribute indicates that multiple microprocess may simultaneously access the data*. In addition, the stack descriptor contains a component which indicates the current top of stack, while the I/O descriptor contains components that specify the format and length of the data; in the latter case, the value of the format and length components are specified through the MEM microinstructions. The I/O descriptor is generally used to define the port of a microprocess that is used in the generation for input or output data sets for a functional unit.

The microassembler syntax for the GEN_REG microinstruction is the following:

```
<GEN_REG>:= S=DESCRIPTOR OF "MEMORY/REGISTER_BLOCK/
               STACK/IO_BLOCK" DEFINED FROM
               (DESCRIPTOR=<P1>, OFFSET=<P2>) WITH
               DIMENSION=<P3>, ACCESS_CONTROL=<P4>,
               "WORD_LENGTH/INITIAL_POSITION"=<P5>;
```

The <P1> and <P2> parameters specify the beginning address of a subarray of registers that were previously allocated in the Process Space Memory. If these two parameters are null, then a new block of registers will be allocated in the Process Space Memory rather than using previously allocated registers. The <P3> parameter specifies the number of registers in the block; the <P4> parameter specifies the access control attributes; and the <P5> parameter specifies either the

*The sharability attributes are used to specify the store through mode of data when a cache per microprocessor organization scheme is employed; this concept of cache per microprocessor will be discussed more fully in Chapter IV.

initial position of the top of the stack or the word length of a memory array in the Memory Subsystem.

III. A COMPREHENSIVE TEST CASE

This chapter reviews in a step by step manner the design and the coding of an emulator for a complex IML. The choice of an IML was not based on the practicality of the IML as a machine language, but rather on its appropriateness as a vehicle to test:

1) The sufficiency of the SBL and the associated global control structure to represent a wide variety of sequential and parallel control structures.

2) The suitability of the SBL for the compact and simple coding of IML emulators.

The IML chosen is based on an asynchronous parallel programming schema (language) developed by D. Adams (ADA68) called the Adams' Graph Machine Language (AGML). The use of the AGML as an IML forms an appropriate test case because the emulator for the AGML can be designed to employ the following control structure concepts: distributed parallel control, pipelining, recursion, finite resource scheduling, message queuing, and the reservation station approach to the scheduling of arithmetic units. Furthermore, the emulator for this IML is interesting in its own right because "no computing systems have yet been designed or translators created according to the principles of asynchronous programming" (ERS72). In addition, the AGML permits the direct expression of highly parallel algorithms whose emulation provides an interesting set of simulation test cases for evaluating certain aspects of the computer organization to be discussed in the next chapter.

III.1 Discussion of Adams' Graph Machine Language

The AGML is based on a data flow model (KAR66,ROD67) for representing the sequencing aspects of a computation. In a language based on a simple data flow model for sequencing, the instructions of the language can be thought of as nodes of a graph, where the nodes are connected to each other through links. These links are uni-directional data paths where one terminal point of the link is denoted as an output link of a node while the other terminal point is denoted as an input link of a node. An instruction (node) is executed when each of its input links contains a data item; a node executes by removing the input data from its input links, performing a calculation on this data, and storing the output of the calculation on zero or more of the output links. After the node has stored the results of the calculation on its output links, the node can be re-executed when each of the input links again has data items. An example of a graph program is shown in Figure 17.

The data flow model for sequencing allows the implicit expression of parallel activity because if there exists no data dependencies among a group of nodes, then these nodes may be executed simultaneously. For example, the two multiplication nodes in Figure 17 can be executed simultaneously whereas the plus node must await the completion of both multiplication nodes. A data flow model can also be thought of as a distributed control system since each node can independently decide, based on local information, whether it should execute.

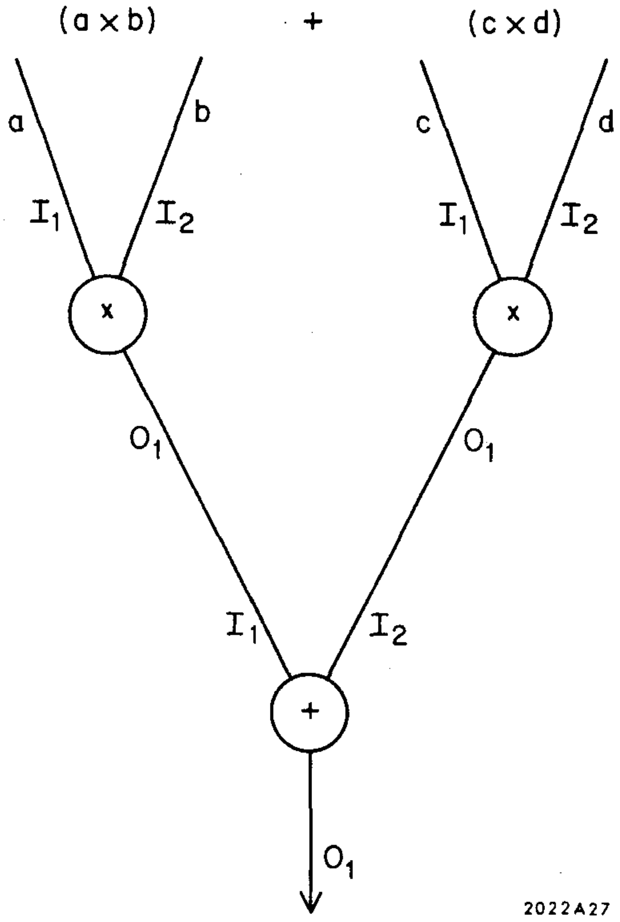


Figure 17. A Simple Graph Program

Adams' formulation of a graph machine language, AGML, is an extension of the simple data flow model previously described. The simple data flow model has been extended in the following ways by Adams' so as to increase the expressible parallelism and to simplify the coding of algorithms:.

1) links between nodes are fifo (first-in-first-out) queues;

2) there are three types of nodes: parallel, procedure, and sequential nodes;

3) a data item can be an array of data items.

The parallel node allows for the expression of pipeline (vector) parallelism. The parallel node is defined so that it may be immediately re-executed rather than waiting for the compututation to complete. This re-execution of the parallel node can occur once the input data items have been removed from their links provided there is another set of input data items on the input links. Thus, multiple instances of a node may be concurrently executing giving the effect of a pipeline. The procedure node allows for the expression of recursive control structures. The procedure node instead of invoking a primitive arithmetic operation, causes the invocation of a graph procedure. The input parameters of the invoked graph procedure are the input data items of procedure node. The invoked graph procedure may, in turn, contain procedure nodes, thus leading to a recursive, parallel control structure. Finally, the sequential node allows for the expression of data-dependent sequencing of the graph, e.g., loop-control, etc. The sequential node is defined so that its semantics are affected by its previous execution history. In particular, the execution history is used to select, for the next execution of the sequential node, a subset of its input data links from which input data will be accepted.

The version of the AGML that was emulated as an IML differs in two ways from Adams' original formulation discussed above:

1) the data items are single units rather than arrays of data items;

2) the conditions for termination of a graph procedure have been changed.

The first difference does not impact the basic organizational structure of the emulator but does simplify the coding of the primitive node operations, and the dynamic space allocation algorithm. However, the second difference does impact the basic organizational structure of the emulator. In Adams' original formulation of the AGML, a graph procedure is terminated when all the nodes of the graph procedure are inactive. This termination condition has been changed so that now a graph procedure is terminated when there is a data item on each of the external output links. An external output link is a link of the graph procedure that is not connected as an input link to any node in the graph procedure. These external output links are used to transmit the output of the graph procedure to the procedure node that invoked the graph procedure. Thus, if there is an infinite loop in a graph procedure, which does not affect the generation of the outputs of the procedure, then this graph procedure would never be terminated based on Adams' original termination condition, whereas this graph procedure would eventually terminate based on this new termination condition*. Otherwise, the termination conditions are identical.

This new termination condition was introduced so that the AGML could be emulated in a highly parallel manner. Monitoring for Adams' original formulation of the termination condition is very difficult to do in a highly parallel manner. This is especially true if no assumptions are made about the actual number of physical microprocessors. In essence, the monitoring process overlays the highly parallel distributed control structure of the graph machine with a control structure which requires sequential accessing of a shared, global data base. In contrast, the monitoring process for the new termination condition does not affect the basic distributed control nature of the AGML. In fact, the process of monitoring for this new termination condition is precisely the same as the process of monitoring for whether a node is ready to execute.

*The new termination condition makes the AGML only output determinate rather than completely determinate.

III.2 The Design of an Emulator for the Adams' Graph Machine Language

A modular task oriented approach has been used in the design of the AGML emulator. The main emphasis in the design of this emulator has been the exploitation of the implicit parallelism of an AGML program. This exploitation of parallelism has been accomplished by:

1) making parallel, whenever possible, the overhead functions required to sequence an AGML program;

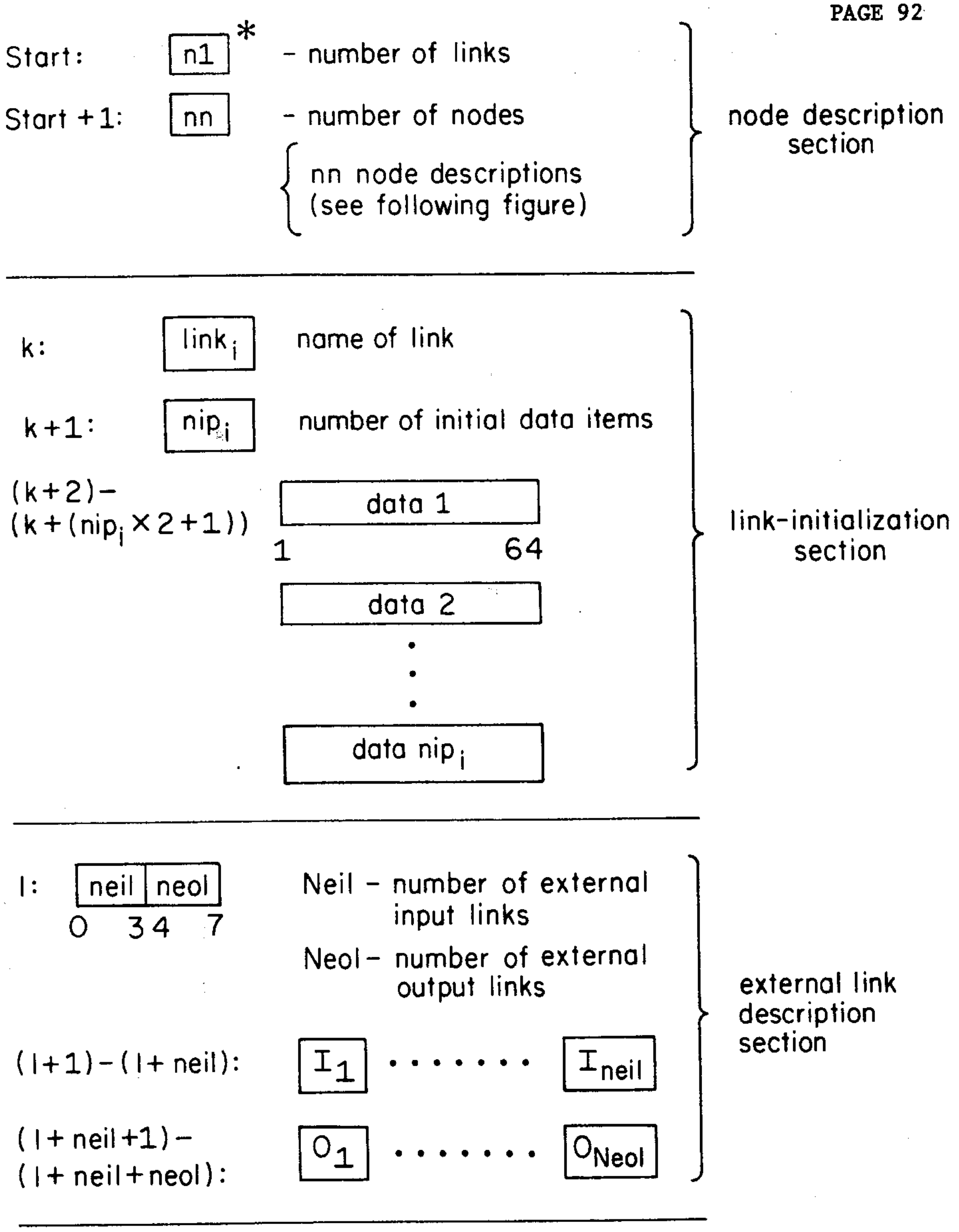
2) dynamically tailoring of the CDS, not only to the structure of AGML emulator, but also to the structure of the specific AGML program to be emulated.

This tailoring of the CDS for a specific graph program is accomplished by creating a separate, distinct control structure for sequencing each node of the graph. This control structure for sequencing each node is tailored to the particular type of node and the node's input and output requirements. Thus, the CDS for the AGML emulator closely mirrors the distributed control structure of the particular AGML program. In addition, the CDS may be dynamically modified during the execution of a graph program so as to take advantage of the potential parallel activity that is generated when a graph procedure is dynamically invoked.

III.2.1 Machine Language Format and Memory Layout

The first step in the design of the emulator is choosing an internal representation (machine language format) for specifying a graph program to the emulator. A graph program consists of one or more graph procedures, where the format of a graph procedure is pictured in Figure 18a. The graph procedure description is broken into three sections: a node definition section, a link-initialization section, and an external-link definition section.

The node definition section begins with two bytes that specify the number of links (nl) and the number of nodes (nn) in the graph procedure. The remaining part of the first section specifies the format of each of the nn nodes of the graph procedure (see Figure 18b). The first byte of each node description indicates the type of node, e.g., parallel, procedure or sequential, the operation code, and the type of arithmetic unit this operation code can be executed on. The second byte specifies the number of input links (IN) and the number of output links (OUT) of the node. The third byte in the case of a procedure node is concatenated with operation code of the procedure node to specify the beginning address of the graph procedure to be invoked. In the case of a sequential node, the third and fourth bytes define the initial input link status of the node. This link status information determines from which input links data will be accepted on the first execution of the node. The remaining bytes of the node description specify the names of the input and output links.



*An unlabeled box represents an 8 bit byte

2022A31

Figure 18a. Machine Language Format for a Graph Procedure

The link initialization section specifies initial data for the links of the graph procedure. The first byte specifies the name of the links to be initialized. If the name is zero, then the link initialization section is terminated. Otherwise, the next byte specifies the number of initial data items to be stored on the link. The remaining bytes specify the initial data items as 64 bit values.

The external link definition section specifies the names of the external input and output links. The external input links are those links that will receive the initial input data items transmitted by procedure node that invokes the graph procedure. The external output links are those links that will hold the output of the graph procedure. At the termination of the graph procedure, this output data will be transmitted back to the invoking procedure node. The first byte of this section specifies the number of external input and output links. The remaining bytes specify the names of these links. An example of a graph program and its corresponding machine language format is detailed in Appendix D.

The internal representation of the AGML program is stored in the Memory Subsystem. The Memory Subsystem is also used to store the data held on the links. All the other state information of the emulator is held in the Process Space Memory.

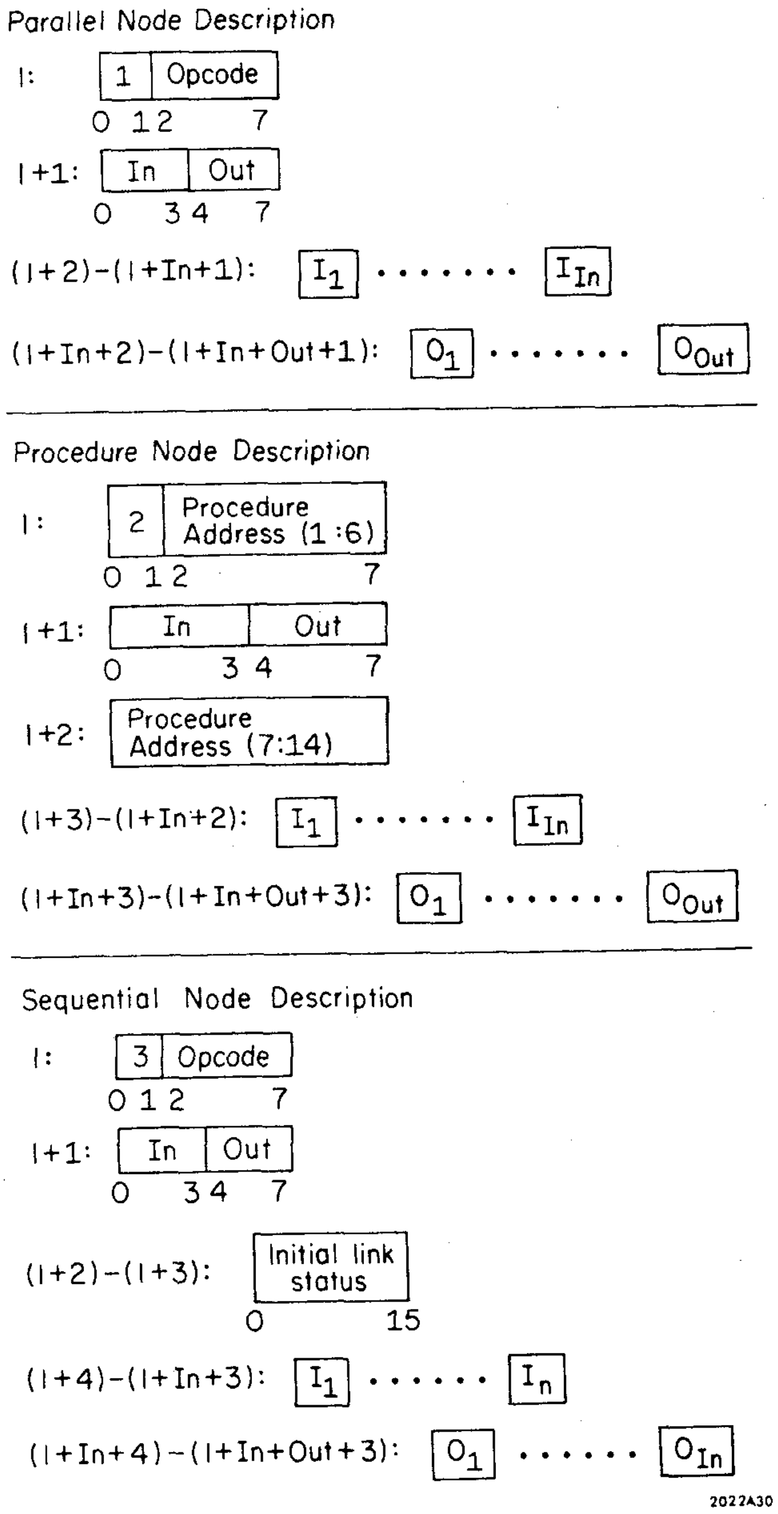


Figure 18b. Machine Language Format for a Node

III.2.2 The CDS for AGML Emulator

The second step in the design of the emulator is the specification of the emulator's CDS. The CDS provides a syntactic framework within which the emulator can be conveniently microcoded. This syntactic framework has been constructed so as to strongly reflect a task oriented approach to the design of the emulator.

The CDS of the AGML emulator can be thought of in terms of two parts: 1) a CDS for resource management (e.g., the dynamic allocation of memory in the Memory Subsystem for link queue space) and 2) a CDS for sequencing of a graph program. These two parts of the emulator's CDS form a two level hierarchy where the CDS for resource management is at the top level. The CDS for resource management is implemented as a fixed structure which is independent of the particular AGML program being emulated, whereas the CDS for sequencing of the graph program has a dynamic structure which is dependent on the particular graph procedures currently being executed.

III.2.2.1 The CDS for Resource Management

The CDS for resource management is pictured in Figure 19a. The resource management functions are carried out by the SPACE-MANAGER, and SCHEDULER(1-Ns) microprocesses. The SPACE-MANAGER microprocess dynamically allocates blocks of storage in the Memory Subsystem for link queue space. Each link is allocated a fixed length block of 16 64-bit words in the Memory Subsystem. This storage allocation cannot be done statically since graph procedures can be dynamically invoked during the execution of a graph program. In addition, there can be many graph procedures which are simultaneously requesting storage for their links. Thus, the storage allocation has to be done dynamically in a central place. As discussed in Chapter II, the SPACE-MANAGER microprocess, by appropriate manipulation of its execution-state, can sequentialize the acceptance and processing of communications which either request the allocation of storage or specify the release of previously allocated storage.

The scheduling function of the resource manager is implemented through a set of SCHEDULER microprocesses where each type of primitive node operation could conceivably have its own SCHEDULER microprocess. A SCHEDULER microprocess is used to assign, depending upon the type of operation, either a functional unit or a microprocess to carry out the primitive operation of a node. Each SCHEDULER microprocess has a fixed length queue to hold requests for a device (e.g., functional unit or microprocess) that cannot be currently honored. If this queue becomes full, then the SCHEDULER microprocess will employ the "waiting" execution state that permits selective listening rather than the suspended state. In this selective listening state, a communication to the SCHEDULER that requests a device will not be consummated, whereas a communication to the SCHEDULER that specifies the termination of a device will be consummated.

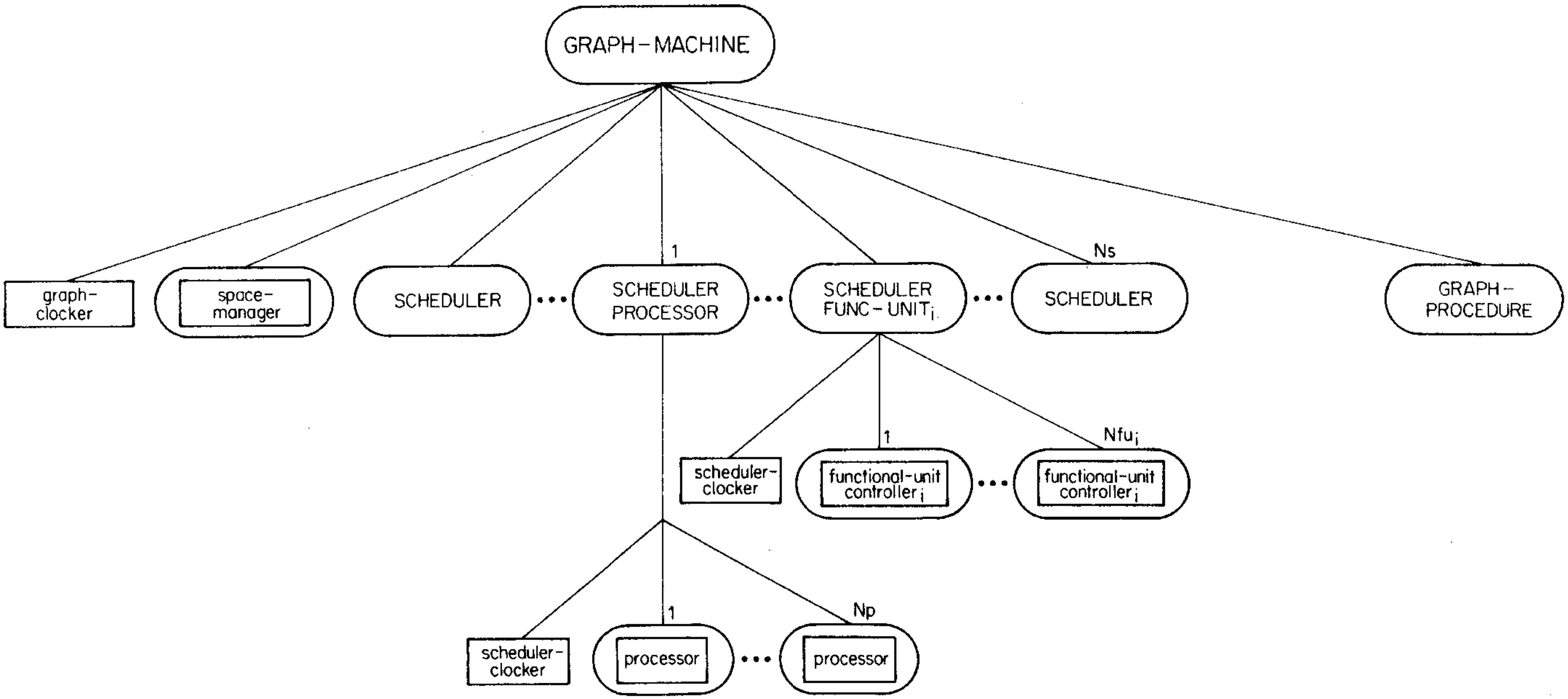


Figure 19a. Control Data Structure for Resource Management

The AGML emulator could have been organized without this centralized scheduling function. In essence, the centralized scheduler is scheduling virtual microprocesses which are in turn being scheduled on physical microprocessors by the built-in hardware scheduler. Thus, the emulator could have been organized so as to use the built-in scheduler alone. There are two main reasons for taking the centralized scheduler approach. The first reason stems from the simplicity of the built-in hardware algorithm for scheduling. Specifically, the two level scheduling approach allows the design of a sophisticated graph scheduler which takes into account the structure of the graph procedure so as to utilize available microprocessors* more efficiently(NEL72). The second reason stems from the semantics of the parallel node that permit the concurrent initiation of an arbitrary number of primitive operations for each parallel node. In order to take advantage of this potential parallelism of the parallel node in a non-centralized scheduling approach either 1) each time a primitive node operation was initiated, the sequencer of a parallel node would have to dynamically generate the MSV of a microprocess to carry out the operation; or 2) the fixed CDS structure of the appropriate SCHEDULER microprocess would have to be duplicated for each parallel node in the graph procedure. Thus, either the structure building overhead involved in sequencing of the graph procedure would greatly increase or the size of the CDS for the graph procedure would greatly increase. On the other hand, a centralized scheduler has a

*The virtual scheduler could query the hardware system to find out the number of physical processors, and uses this information as a parameter in the scheduling function.

fixed CDS structure which does not vary during the execution of the graph, and there is only one MSV for each device that can be scheduled. For these reasons, a centralized scheduling approach was used.

III.2.2.2 The CDS for Sequencing of a Graph Procedure

The sequencing of a graph procedure is implemented through the microprocess GRAPH-PROCEDURE. This microprocess generates the CDS for sequencing of a graph procedure, initializes and allocates storage for the links of the graph procedure, and monitors for the termination of the graph procedure. The CDS for sequencing of a graph procedure, as previously discussed, is tailored to the particular graph procedure being emulated. The template for a tailored CDS is pictured in Figure 19b. This tailored CDS contains nl LINK microprocesses and nn NODE microprocesses, where there are three types of node microprocess: PARALLEL, SEQUENTIAL and PROCEDURE. This CDS for sequencing of a graph procedure has been designed so that once generated its structure need not be modified. Thus, the structure building overhead is only incurred once and consequently is not a function of the number of node executions. In addition, the generation of the CDS for all NODE microprocesses can be done in parallel.

The LINK microprocess is responsible for retrieving and storing data from a link's queue space in the Memory Subsystem and updating the queue pointers. The LINK microprocess acts as a

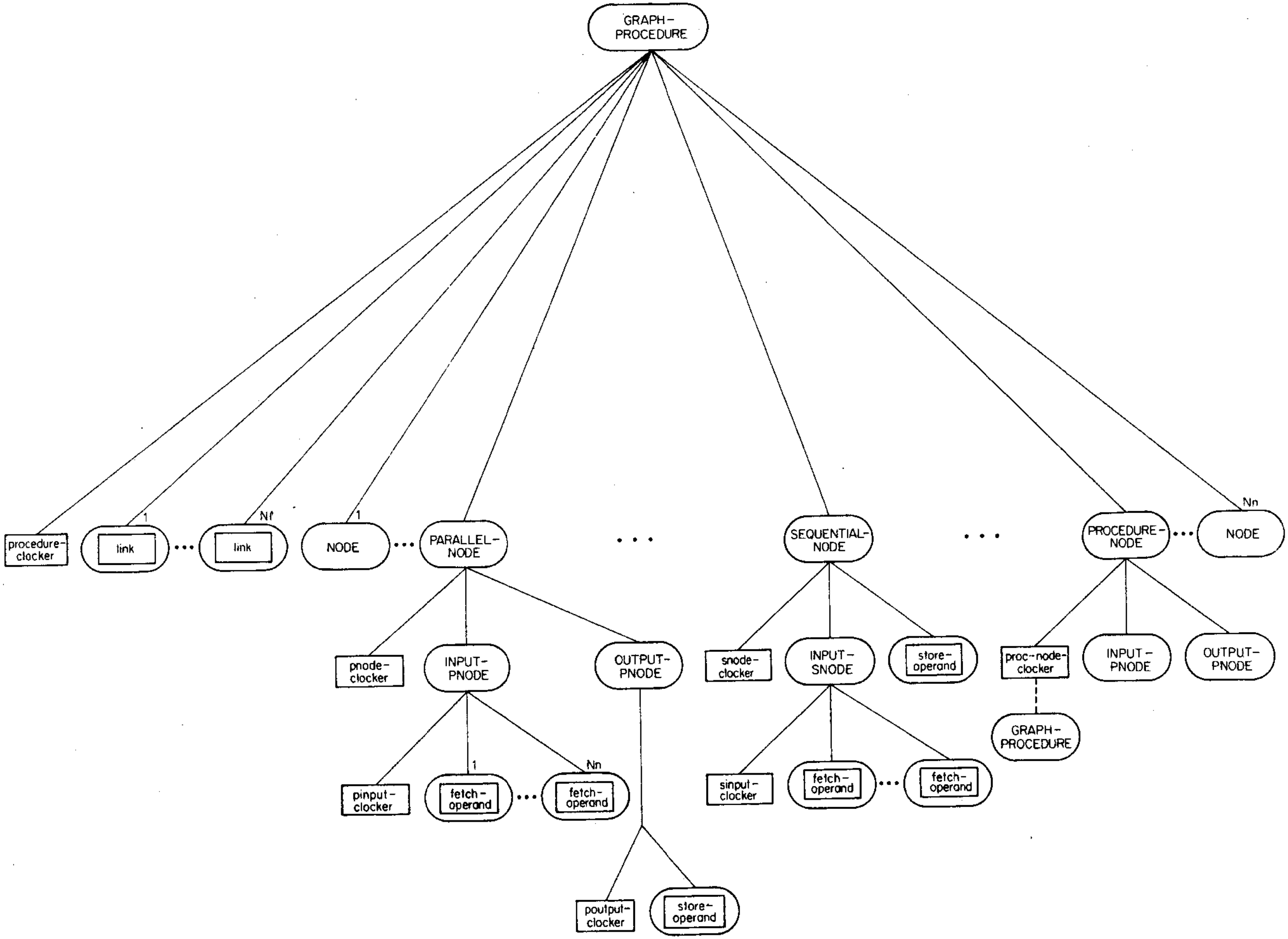


Figure 19b. Control Data Structure for GRAPH-PROCEDURE

semaphore process for controlling access to the link's queue space. A semaphore process is required for controlling access to a link queue because, at the same time, one node may desire to place data on the queue while another node may desire to remove data from the queue. The LINK microprocess is also used to avoid "busy waiting" when a node desires a data item and the link queue is empty. In this case, instead of the node repeatedly querying the LINK microprocess whether input link data is available, the LINK microprocess will accept a request for data from the node and then when the data is available, transfer the data to the node which is in a waiting execution-state. Thus, as will be seen in more detail later, the LINK microprocess allows the trigger function of a node (i.e., deciding when a node is ready to execute) to be monitored in a non-busy way. A similar handshaking mechanism is used to avoid a node "busy waiting" until there is room on the link to store output link data. In addition, the LINK microprocess allows the updating of queue pointers to go on in parallel with a node's further processing.

The NODE microprocess implements the following overhead operations required to sequence a node: 1) fetching the input data items from the appropriate input links, 2) deciding when the node operation is ready to be executed, 3) transferring the input data items to the appropriate microprocess that will perform the node operation, and 4) transferring the output of the node operation to appropriate output links. The CDS associated with each NODE microprocess is designed so that as many of these overhead operations can be either done in parallel or overlapped between consecutive executions of a node.

The overall CDS for the AGML emulator is pictured in Figure 19c. This section has presented the AGML emulator in terms of a set of microprocesses that each performs a small independent task. The next section will discuss how this set of microprocesses dynamically interact to perform the emulation of a graph program. These interaction patterns will be detailed through a discussion of the PARALLEL-NODE microprocess.

III.2.3 The Microcoding of the Parallel Node

The PARALLEL-NODE whose CDS is pictured in Figure 20a, is the most complex of the three types of NODE microprocesses because of the control structures required to generate and keep track of the multiple concurrent initiations of the primitive operations of the node. In order to generate multiple initiations, the fetching of input link data for an operation, which is done by the INPUT-PNODE microprocess, is separated from the storing of output link data for an operation which is done by the OUTPUT-PNODE microprocess. This separation of the input and output phases of a PARALLEL-NODE permits the fetching of input data for one operation to be performed concurrently with the storing of output data for a previously initiated operation. In order to insure the output-determinancy of the graph procedure, multiple initiations of an operation must

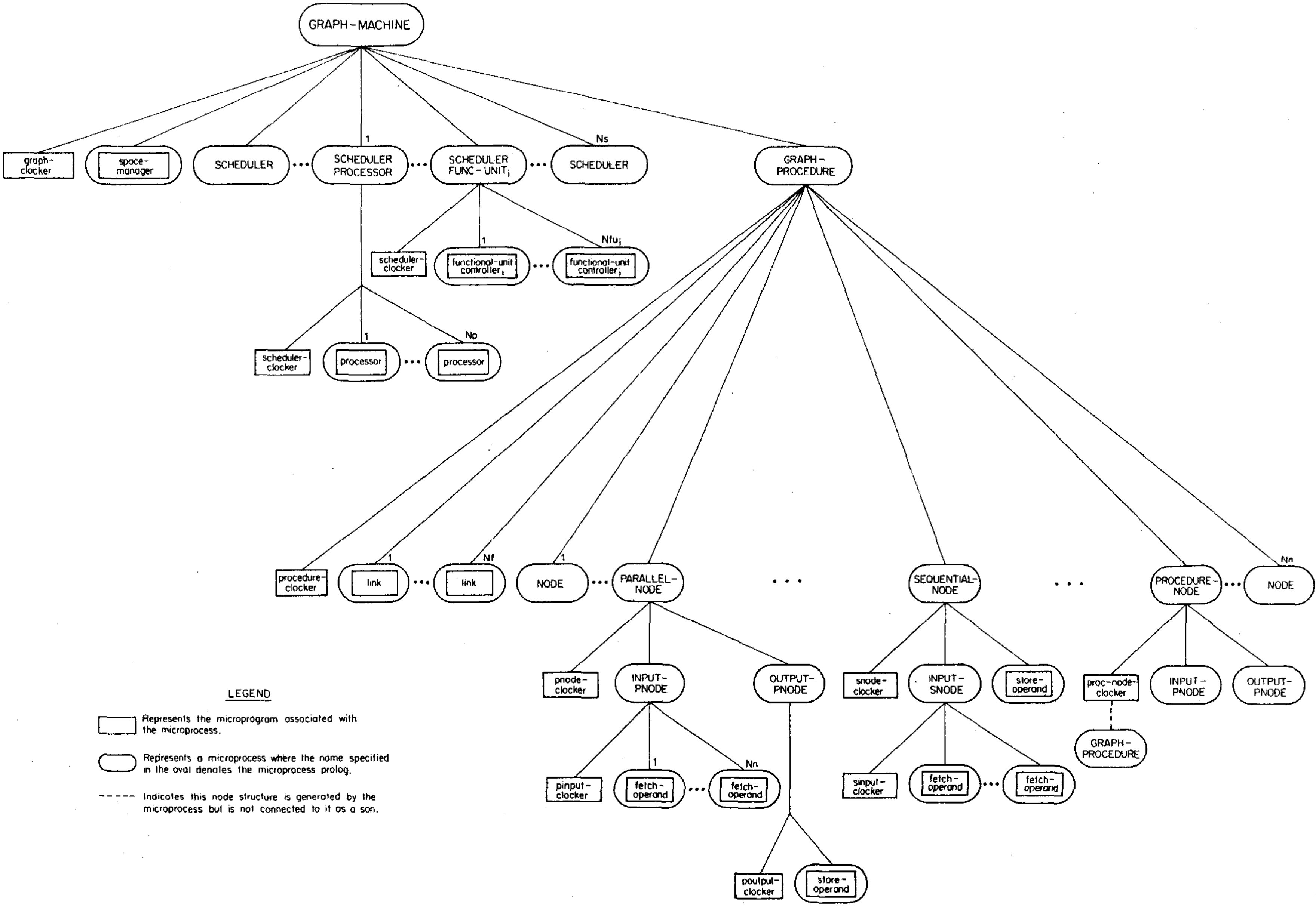


Figure 19c. Control Data Structure for AGML

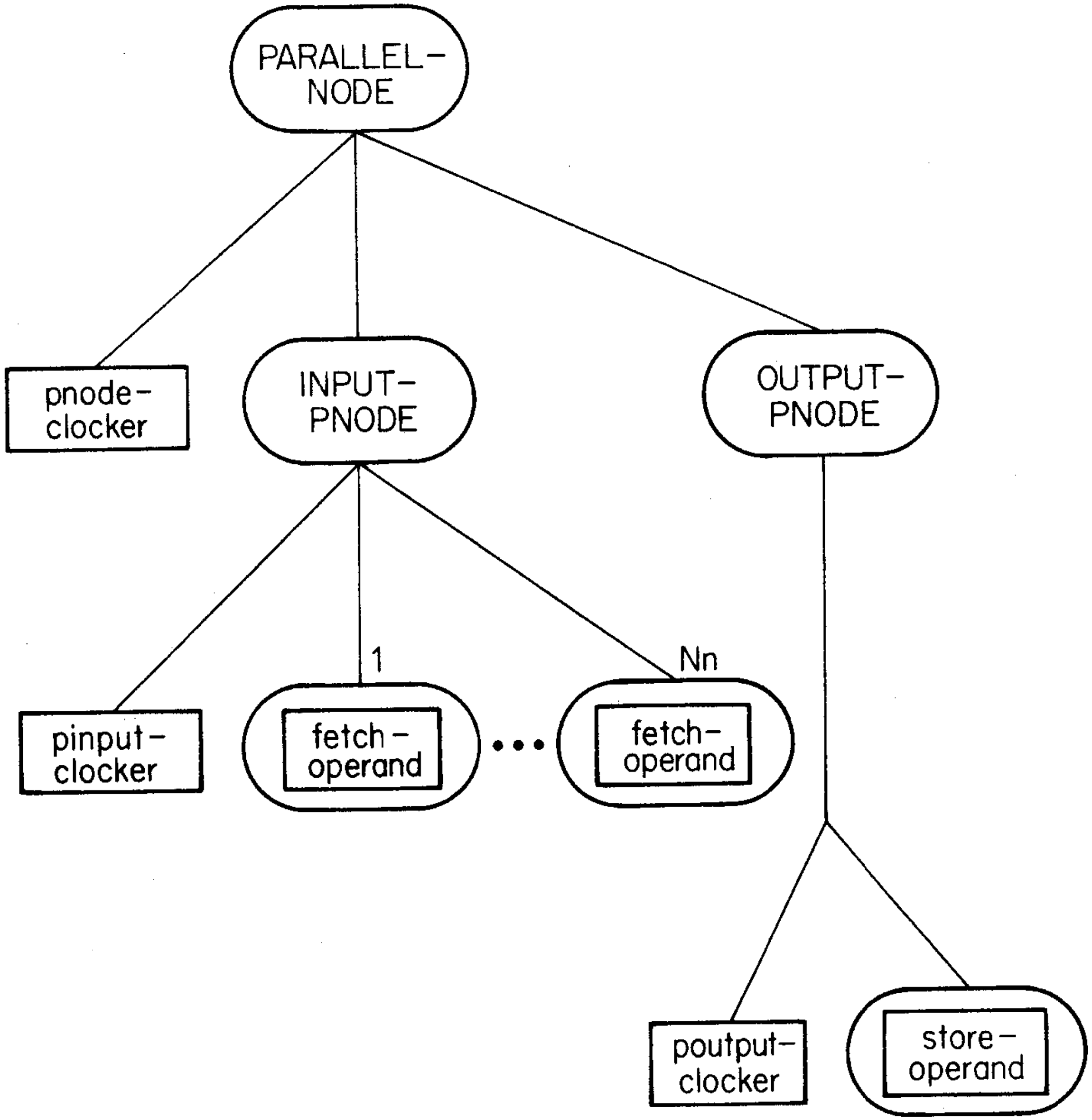


Figure 20a. Control Data Structure for PARALLEL-NODE

terminate in the same order as they were initiated. The PARALLEL-NODE maintains the correct ordering of multiple initiations through a queuing mechanism which holds up the storing of the output of an operation until the output of all previously initiated operations have been stored.

The PARALLEL-NODE interacts directly with the following microprocesses: GRAPH-PROCEDURE, SCHEDULER, INPUT-PNODE, OUTPUT-PNODE, and PROCESSOR(1-n). The interaction patterns of the PARALLEL-NODE with these microprocesses is indicated in Figure 21. The control and data environment interrelationships among these microprocesses are pictured, respectively, in Figure 20b and 20c. Figure 20b indicates how the PARALLEL-NODE microprocess and its son microprocesses locate the addresses of the microprocesses that they will communicate with. The PARALLEL-NODE control environment is a two level hierarchy of global process environments. The top level allows for access to SCHEDULER microprocesses while the lower level allows for access to the LINK microprocesses. For example, the FETCH-OPERAND microprocesses pictured in Figure 20b locate the appropriate LINK microprocesses by accessing their global process environment descriptor, while the PARALLEL-NODE locates the appropriate SCHEDULER microprocess by indirectly accessing through its external environment pointer the global process environment descriptor of the GRAPH-PROCEDURE microprocess. Figure 20c indicates how the PARALLEL-NODE microprocess and its son microprocesseses communicate data with each other. Each of these microprocesses contains its own

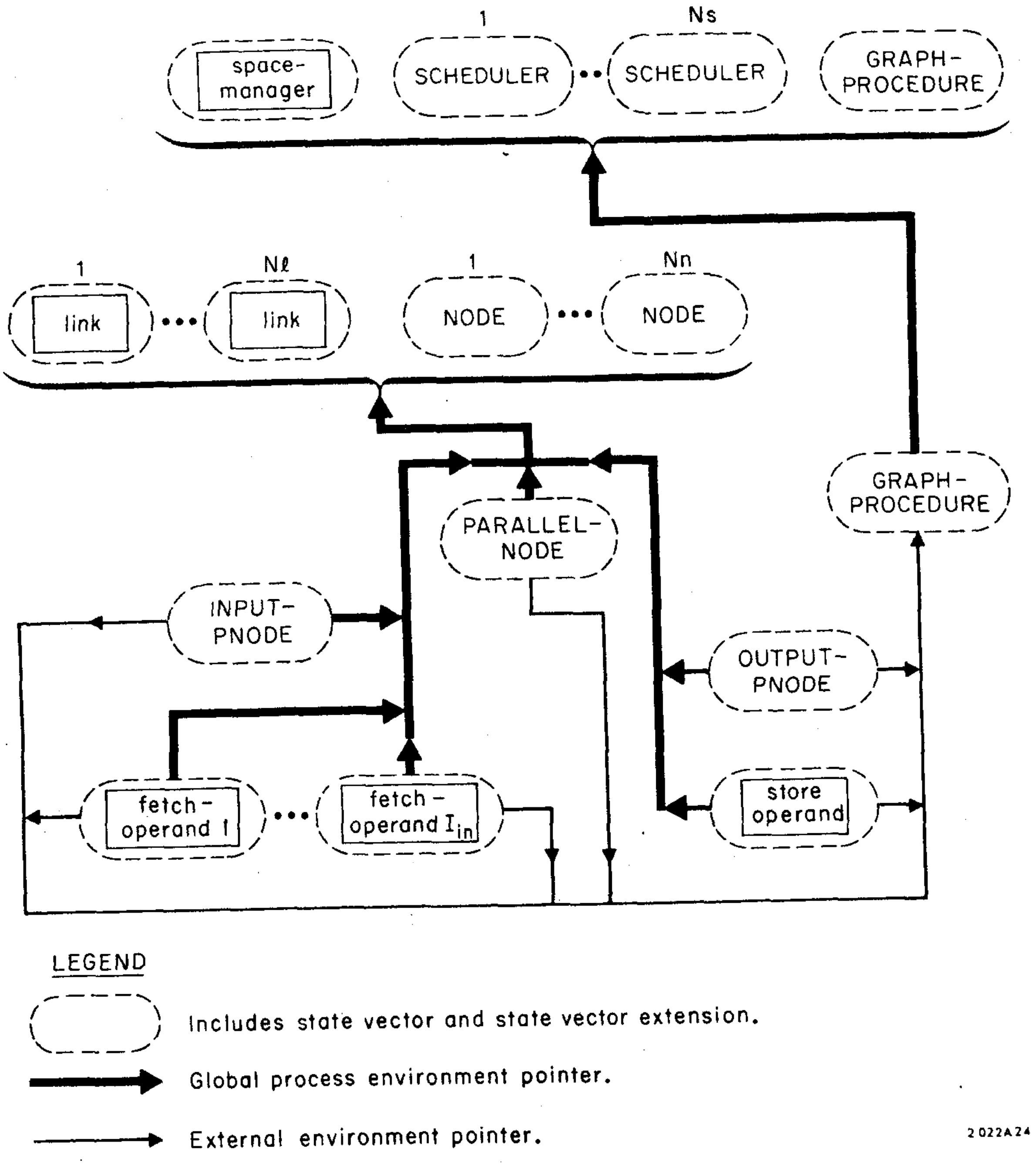


Figure 20b. Control Environment of PARALLEL-NODE

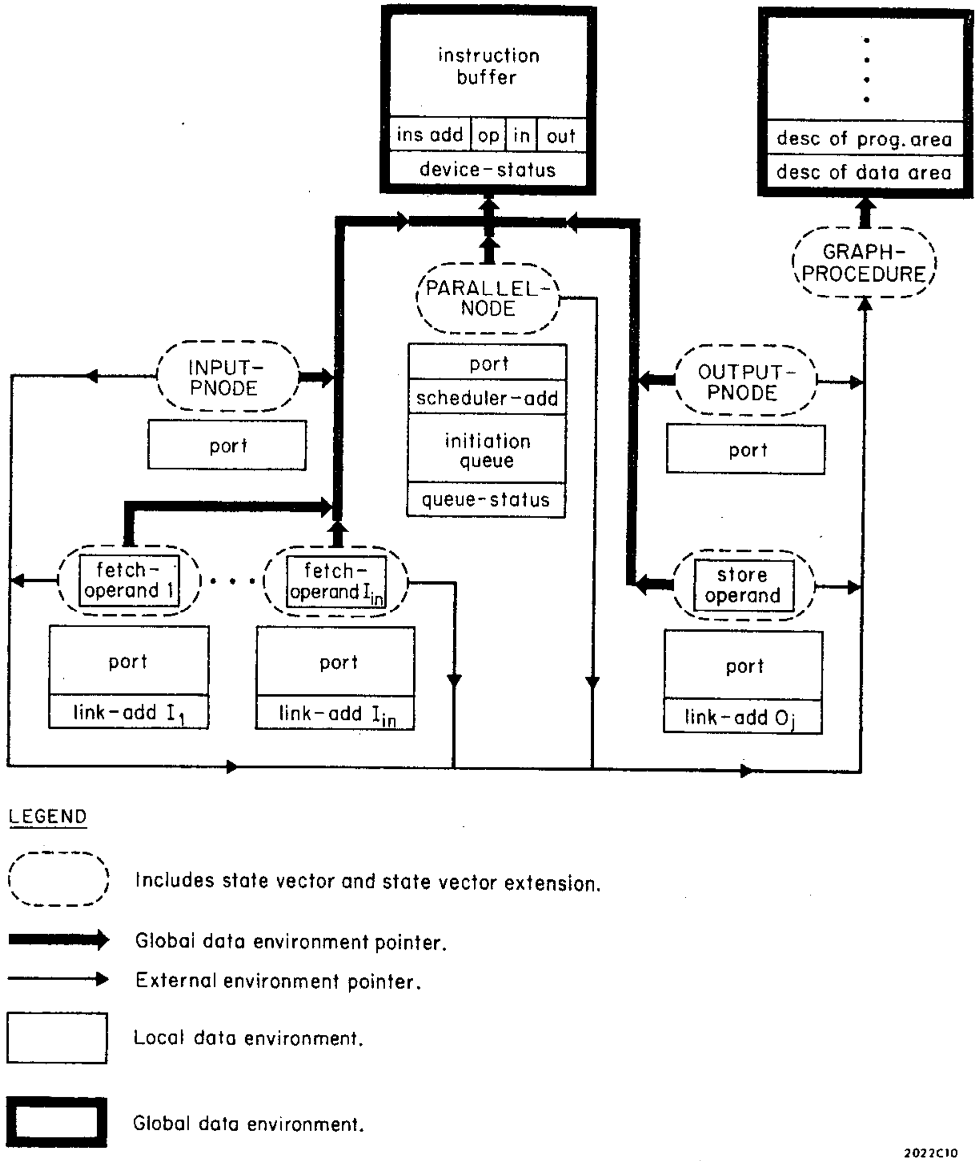
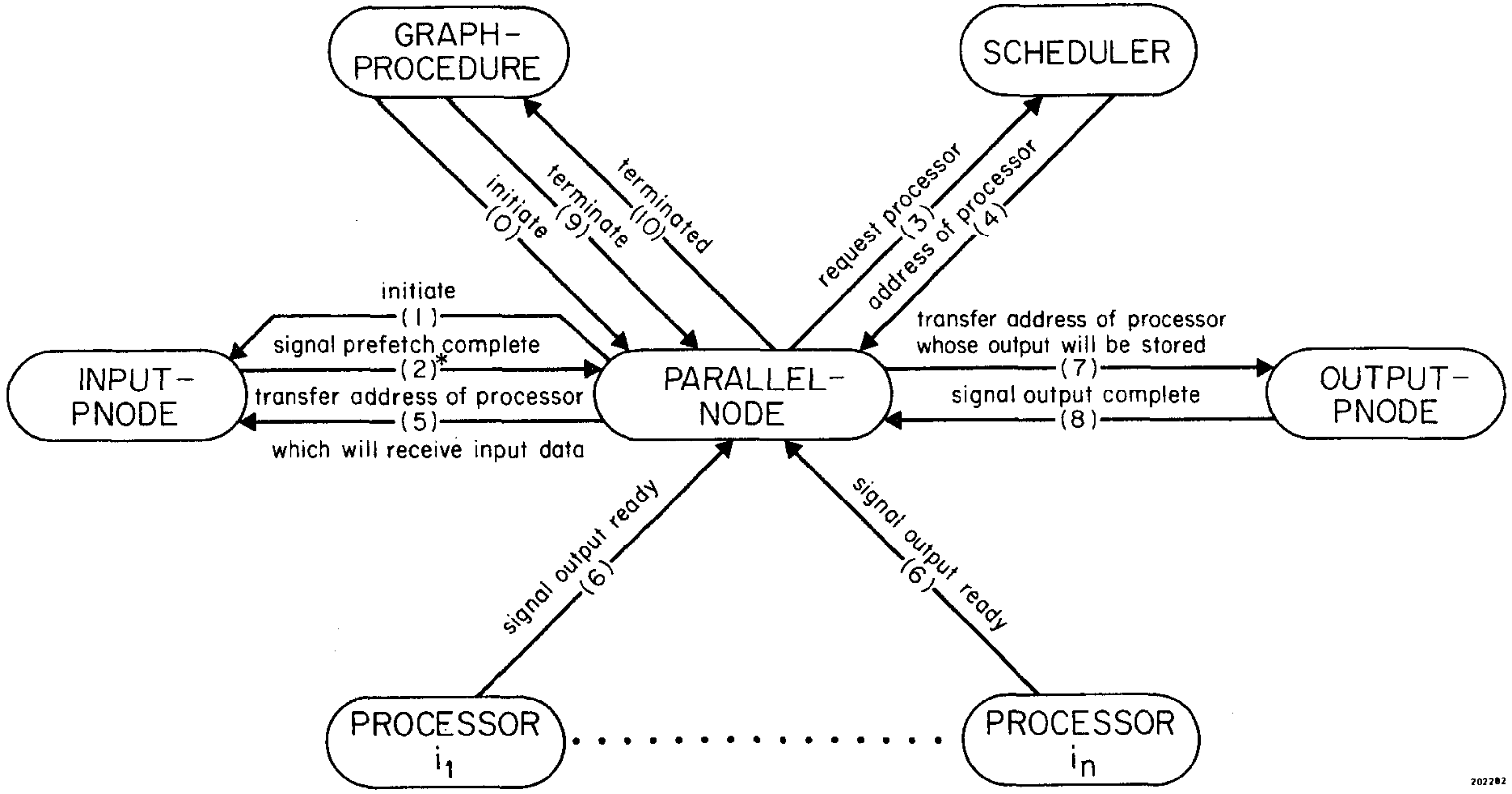


Figure 20c. Data Environment of PARALLEL-NODE



*Steps 2 – 8 represent the sequence of interactions required for a single node computation.

Figure 21. Interaction Patterns of PARALLEL-NODE

PORT which is located in its local data environment. In addition, there is a global data environment which these microprocesses all share. This global data environment contains the description of the particular PARALLEL-NODE in the graph procedure that is being executed.

The GRAPH-PROCEDURE microprocess initiates the PARALLEL-NODE microprocess, and then when the graph procedure termination condition has been met, signals the PARALLEL-NODE to terminate. The PARALLEL-NODE after it has received the terminate signal, waits until all outstanding node operations are completed, and then signals back to the GRAPH-PROCEDURE its termination.

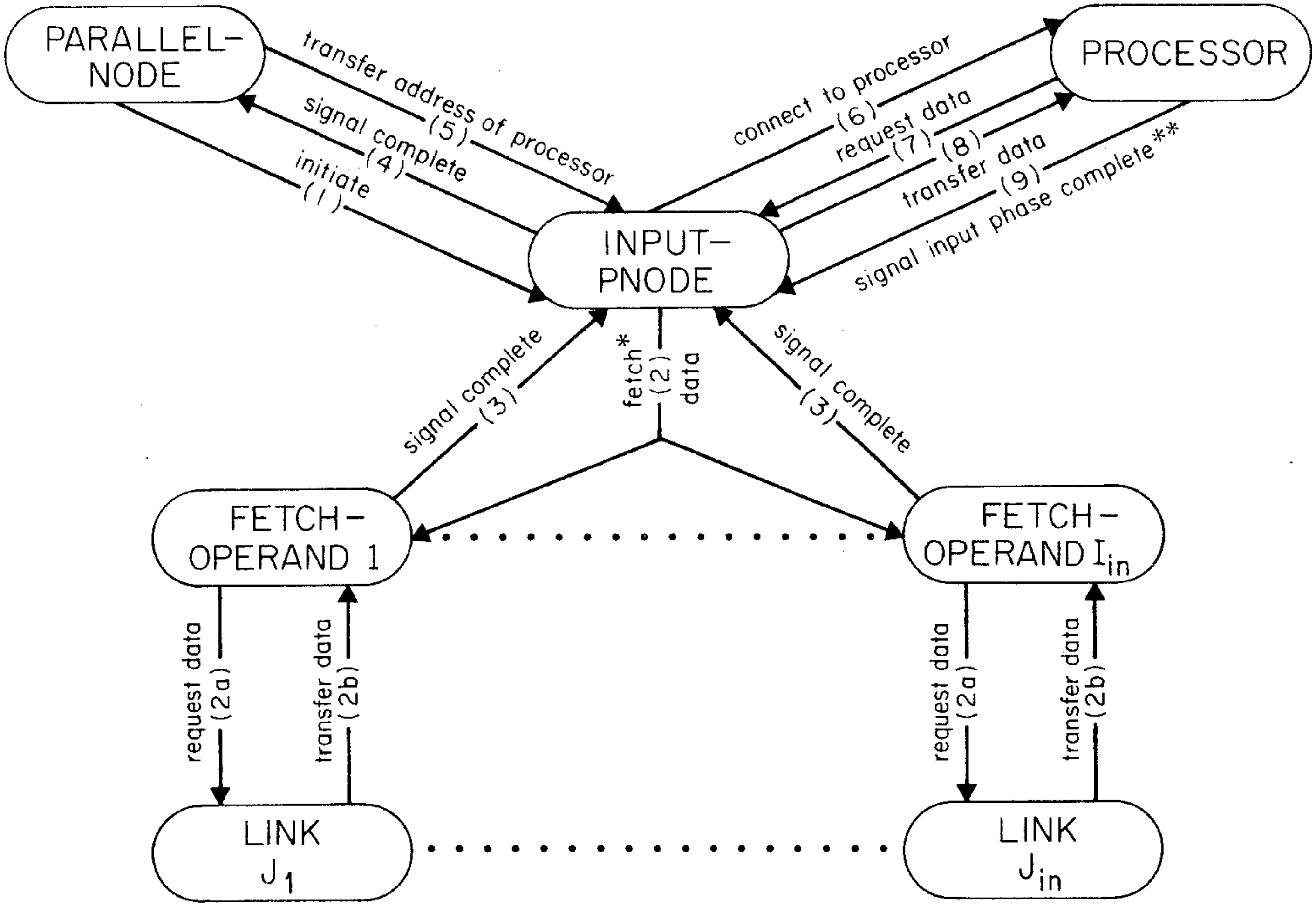
The PARALLEL-NODE, once initiated, activates the INPUT-PNODE microprocess to fetch the input data from the appropriate input links (see Figure 22). After receiving the prefetch complete signal from the INPUT-PNODE, the PARALLEL-NODE then activates the appropriate scheduler microprocess to assign a PROCESSOR to perform the operation*. In this way, a PROCESSOR is not assigned to perform a node operation until the data necessary for the operation has been fetched. This technique for scheduling a processor is called the "reservation station concept".

*The PROCEDURE-NODE is precisely the same as the PARELLEL-NODE except that the PROCEDURE-NODE, instead of calling the SCHEDULER microprocess, generates an MSV of the GRAPH-PROCEDURE microprocess. The address of this newly defined GRAPH-PROCEDURE microprocess is then treated in the same way as the address of the assigned PROCESSOR microprocess.

The PARALLEL-NODE, after receiving the address of the assigned PROCESSOR from the SCHEDULER microprocess, queues the address and activates the INPUT-PNODE with this address. The INPUT-PNODE then transfers the prefetched input data to the assigned PROCESSOR. After the input data has been transferred, the INPUT-PNODE attempts to prefetch the input data for the next operation.

The PROCESSOR(i) microprocess, after completing the desired operation, signals back to PARALLEL-NODE that the output data is ready. The PARALLEL-NODE then checks whether PROCESSOR(i) is at the top of the initiation queue. If PROCESSOR(i) is at the top of the queue, then the address of PROCESSOR(i) is transferred to the OUTPUT-PNODE microprocess. Otherwise, an indicator is set in the initiation queue that PROCESSOR(i) is ready to store its output data. Thus, through the initiation queue mechanism, the outputs of the PARALLEL-NODE are FIFO ordered so as to make the PARALLEL-NODE determinate.

The OUTPUT-PNODE microprocess, upon receiving the address of PROCESSOR(i), transfers PROCESSOR(i)'s output data to the appropriate output links. After the completion of this transfer, the PARALLEL-NODE is notified. The PARALLEL-NODE then examines the initiation queue to determine whether the PROCESSOR(j) at the top of the queue has already signaled that its output is ready. If so, then the OUTPUT-PNODE is reactivated with the address of PROCESSOR(j).



NOTES:

* This represents a broadcast operation.

** After step 9 is completed, the cycle is repeated starting at step (2).

2022825

Figure 22. Interaction Patterns of INPUT-PNODE

These interaction patterns allow the fetching of input link data, storing of output link data, the execution of an arbitrary number of primitive node operations, and the processing of requests to store the output of an operation to all proceed in parallel. In addition, the INPUT-PNODE fetches in parallel, through the use of a broadcast operation, the input link data (see Figure 23). The different types of communications that a PARALLEL-NODE microprocess can receive are distinguished by the particular internal-activation code used in the communication. The PARALLEL-NODE uses this code to jump indirectly in a single microinstruction to the particular microcode routine that handles the communication. The PNODE-CLOCKER microprogram, which is the collection of these microcode routines for handling communications to the PARALLEL-NODE, is less than 70 microinstructions long, and is detailed in Appendix C.

III.3 An Evaluation of the Suitability of the SBL for the Coding of the Graph Machine Emulator

This chapter has reviewed the design and coding of an emulator for a complex IML, i.e., the Adams' Graph Machine Language. The complete microprogram for the AGML emulator is presented in Appendix C. This emulator has been tested, using a simulator of the logical design, on a variety of graph programs under varying PMS configurations in order to validate the emulator's correctness. These test cases, which indicate the dynamic behavior of the AGML emulation, will be discussed in Chapter V.

The AGML emulator has demonstrated the versatility and usefulness of the SBL and the concept of a dynamically restructurable CDS in the following ways:

1) It has shown how an S(vm) can be constructed so as to make the embedding of the state image of a complex IML, S(AGML), straightforward. In particular, it has indicated how a CDS can be tailored so that it directly mirrors the distributed control structure of the AGML.

2) It has shown that an emulator can be compactly and simply coded when the microinstructions directly operate in the context of the appropriate S(vm). The microprogram memory required for the AGML emulator microprogram, including the storage for constants, is less than 600 microinstruction words*.

3) It has shown how a CDS can be dynamically structured so as to easily represent a wide variety of different types of control structures, i.e., distributed control, semaphore processes, message queuing, broadcast control, etc. Further, it has indicated how these different types of control structures can be integrated together in a single CDS.

4) It has shown how a modular task approach to design of an emulator can be implemented naturally within the framework of a restructurable CDS.

---

*Of the 600 microwords, approximately 220 are used to hold the microinstructions for building up the CDS, 300 to hold microinstruction for dynamic control, and the remainder to hold data constants.

IV. A Hardware Implementation of the Microcomputer Architecture

> "Adequate performance of parallel processing systems is......predicated on an appropriately low level of overhead. Allocation, scheduling, and supervisory strategies, in particular, must be simplified and the related procedures minimized to comprise a small proportion of the total activity in the system..........thus a unified and integrated design approach is required in which software and hardware, operating system and processing units, lose their separate identities and merge into one overall complex, for which allocation and scheduling procedures, for example, are as basic and as critical as arithmetic operations." (LEH66)

The main thrust of this chapter is to demonstrate that the microcomputer architecture presented in this thesis can be implemented in hardware in such a way that parallel activity expressed on the virtual PMS level can be mapped, without significant overhead, onto parallel activity on the physical PMS level. In addition, the hardware organization must guarantee that the mapping (scheduling) of virtual activity to physical activity neither introduces hardware resource deadlocks nor changes a deterministic virtual activity into a non-deterministic physical activity. In particular, the latter requirement implies that SBL synchronizing primitives must work correctly, independent of the number of physical microprocessors and the particular interconnection pattern of these microprocessors to microprocesses.

The hardware organizational problems of minimizing overhead, guaranteeing no resource deadlocks, and correctly implementing synchronization mechanisms are all faced in the design of a conventional multiprocessor (LOR72). However, these design problems are significantly magnified in the context of this new microcomputer architecture since microprocessor interaction patterns can be very highly structured and can occur on very fine time grain, i.e., the time between successive interactions can be of very short duration. Thus, there is a greater likelihood for microprocessors to interfere with each other when they 1) access the Microprogram Memory for microinstructions, 2) access the Process Space Memory for shared data items, and 3) communicate with one another. In addition, the time to perform a context switch in a microprocessor from one microprocess to another is especially critical because of the potential for a high rate of context switches. A high rate of context switches may occur because 1) the time between successive interactions of the microprocess is generally of short duration, and 2) in order to avoid a deadlock in mapping virtual activity to physical activity, the microprocessor must be multiprogrammed when there are more active microprocesses than microprocessors. Thus, when a microprocessor is connected to a microprocess that is waiting for a response to a communication, the microprocessor is context switched if an active but not connected microprocess exists.

These conventional problems in the design of a multiprocessor are overlayed with problem of efficiently implementing the concept of a virtual PMS. The efficient implementation of the concept of virtual PMS implies that:

1) There is a built-in hardware scheduling algorithm.

2) The internal working registers of a microprocessor are dynamically reconfigurable so as to conform to the particular microprocess being executed.

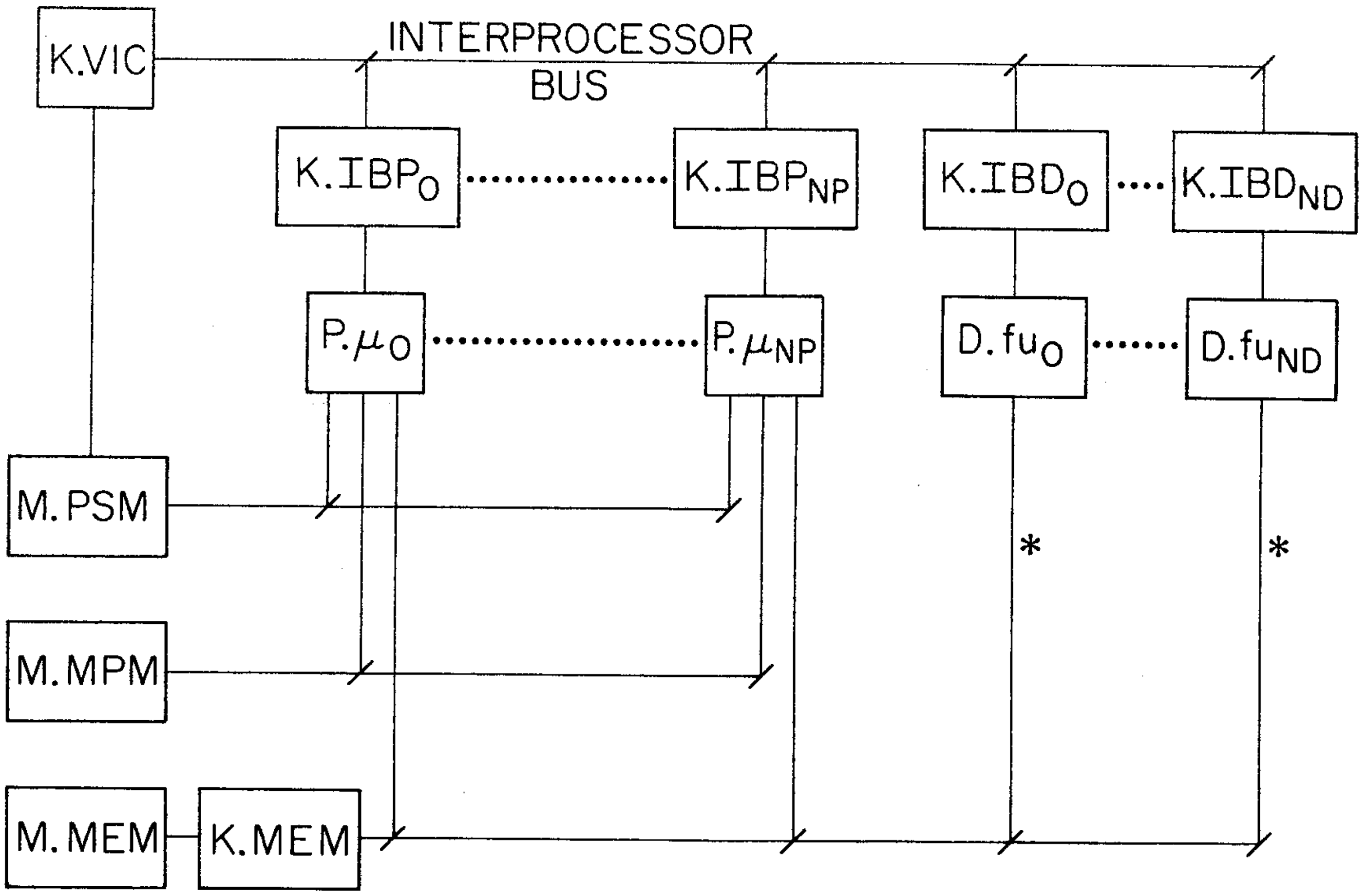
3) Microprocess interaction patterns whenever possible, dependent upon the particular connection between the virtual PMS and physical PMS at the time of the interaction, are directly implemented as microprocessor interaction patterns rather than indirectly implemented as modifications to the Process Space Memory.

The remainder of this chapter will be a discussion of the hardware organization for the microcomputer architecture. This organization demonstrates that there exists a plausible solution to the design problems previously outlined and, further, that this design is a coherent, integrated solution to these problems. However, this chapter will not discuss the hardware technology required to implement the bus structure and memories specified in the design(BEL71,LOR70,MIL70), but rather, this chapter will focus on the interconnection patterns and interaction protocols among the "black boxes" that define the PMS environment of the microcomputer architecture, i.e., the logical hardware organization. The next chapter will, however, investigate the dynamic aspects of the designed system in order to justify the claim that virtual microprocess activity can be mapped without significant overhead into physical microprocessor activity. In addition, this next chapter will examine how varying the interleaving, access paths and access time characteristics of the memories, and the bandwidth and access time characteristics of the bussing structures affect the performance of the microcomputer architecture.

IV.1 A PMS Configuration for the Microcomputer Architecture

An overview of the PMS configuration for the microcomputer computer architecture is pictured in Figure 23 (notation is that of Bell and Newell, BEL70). This configuration consists of NP microprocessors and ND devices (functional units) which can directly communicate with each other over an Interprocessor Bus (IB). The Interprocessor Bus is 128 bits wide so as to allow the transfer of an MSV, an EPSV, or up to 4 data words in one cycle. Each microprocessor and functional unit has separate control hardware, respectively K.IBP and K.IBD, for interacting with the IB. The overall control of IB resides in the Virtual Interaction Controller, K.VIC. K.VIC, together with the K.IBP and K.IBD, represent the hardware for mapping microprocess interaction patterns into microprocessor interaction patterns.

There are three external memories contained in the PMS: M.PSM, M.MPM, and M.MEM. M.PSM, which is the Process Space Memory, holds the global CDS. A microprocessor directly accesses M.PSM in order to retrieve and modify the working registers of the microprocess it is executing. A microprocess indirectly accesses the M.PSM through K.VIC in order to obtain an MSV. The M.MPM, which is the Microprogram Memory, is accessed by microprocessors directly in order to fetch



*These data paths are optional depending upon the bandwidth required by the device.

Figure 23. PMS for Microcomputer Architecture

microinstructions. The M.MEM, which is the main Memory Subsystem, is a bit addressable memory. The K.MEM is the control circuitry required to perform the appropriate shifting to align the desired bit string. The Memory Subsystem is accessible by all the microprocessors and some of the functional units. The normal operation mode for a functional unit assumes that a microprocessor performs all the fetching and storing of data required by the functional unit. This normal mode, as discussed in Chapter II, permits a functional unit to be used in the emulation of many computers since it does not have to be preprogrammed to be aware of the location of its input and output data. Thus, a functional unit in this mode of operation does not need a direct path to the Memory Subsystem. However, there may be functional units which require a very high data rate which cannot be sustained in this normal mode of operation; therefore, some functional units may require a direct path to the Memory Subsystem.

The PMS configuration presented in Figure 23 does not indicate the number of independent communications that a bus can handle, nor the interleaving of a memory. These PMS characteristics have been purposely omitted because they can be varied in the simulator. In addition, the simulator permits the reconfiguration of the bussing structures so that the bussing structure for interprocessor communication can have the additional function of being an access path to any one of the three external memories. In this manner, the PMS can be configured to have from 1 to 4 independent bussing structures.

A more detailed configuration of the PMS for the microcomputer architecture is pictured in Figure 24. In this configuration, there are no functional units and 16 microprocessors. Each microprocessor is connected to the Interprocessor Bus which can sustain up to 8 communications simultaneously. The Interprocessor Bus also serves as the access path from the microprocessors to M.PSM and M.MPM. The M.PSM is 16 way interleaved on lower order address bits, and is connected to Interprocessor Bus through an 8 x 16 crosspoint Switch, S.PSM. The M.MPM is 8 way interleaved and is connected to the Interprocessor Bus through an 8 x 8 crosspoint switch, S.MPM. The Memory Subsystem, M.MEM, has its own separate bussing structure which can sustain up to 4 communications simultaneously. M.MEM is 4 way interleaved and is connected to bussing structuure through a 4 x 4 crosspoint switch.

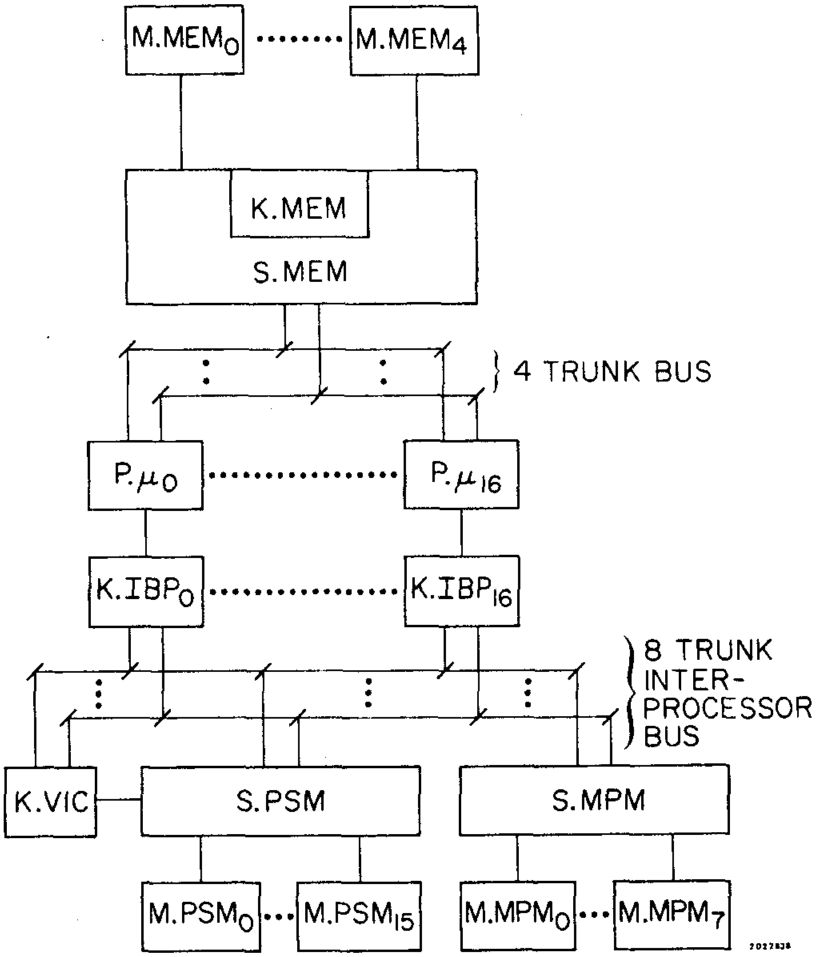
IV.2 The Interprocessor Communication Structure and the Virtual Interaction Controller

This section will discuss the major logical design issues involved in mapping virtual PMS activity to physical PMS activity:

1) The bussing structure for interprocessor communication.

2) The design requirements necessary to insure no hardware deadlocks are introduced which are not already present as software deadlocks.

3) The hardware algorithm for scheduling of microprocesses on microprocessors.



$M.MEM_i$ - Main Memory module i, where M.MEM is interleaved on low order bits 4 ways, cycle time of 2 μsec

S.MEM - 4 by 4 crosspoint switch for access to modules of Main Memory

K.MEM - control used for aligning bit string

$P.\mu_j$ - one of 16 microprocessors, cycle time of 1 μsec

$K.IBP_J$ - control used for scanning 8 trunk Interprocessor Bus

K.VIC - Virtual Interaction Controller

S.PSM - 8 × 16 crosspoint switch for access to modules of Process Space Memory

$M.PSM_i$ - Process Space Memory module i, where M.PSM is interleaved 16 ways, cycle time of 2 μsec

S.MPM - 8 × 8 crosspoint switch for access to modules of Microprogram Memory

$M.MPM_i$ - Microprogram Memory module i, where M.MPM is interleaved 8 ways, cycle time of 2 μsec

Figure 24. A Detailed PMS for Microcomputer Architecture

One of the major considerations in this design is to multiprocess rather than to multiprogram virtual microprocessor activity whenever possible. The design of the Interprocessor Bus (IB) strongly reflects this major design consideration. The IB is designed so that, whenever possible, microprocess interaction patterns are directly implemented as microprocessor interaction patterns. The IB transforms a microprocess interaction pattern directly into a microprocessor interaction pattern by acting in a manner similar to the common data bus on 360/91 (AND67) or equivalently the UNIBUS on PDP-11(DEC69); a similar technique for handling interprocessor communications has also been suggested by Lehman(LEH66).

A microprocessor P1, when executing a microprocess V1 that desires to initiate a communication with another microprocess V2, sends out a request on IB for the MSV of microprocess V2. Each request sent out on the IB is scanned by the control circuitry K.IBP, associated with each microprocessor, in order to determine whether that microprocessor is currently connected to the microprocess V2. If a microprocessor P2 is connected to microprocess V2 and is not currently involved in a dialogue with another microprocessor, then microprocessor P2 will honor the request for the MSV of V2 and transmit this MSV directly over IB to microprocessor P1. However, if there exists no microprocessor to honor the request, then the Virtual Interaction Controller will take over responsibility for fetching the MSV of V2 from the M.PSM and then transmitting this MSV to microprocessor P1. Once a connection has been established between microprocessor P1 executing microprocess V1 and microprocessor P2 executing microprocess V2, the IB can be used to transmit a communication between microprocesses V1 and V2 directly. In particular, if a communication can be consummated between V1 and V2 then the IB can be used to transmit the new MSV of V2, and the data to be stored in the port of V2. After the communication has been consummated, a signal is sent by P1 to P2 on the IB to break off the connection between the microprocessors so that microprocessor P2 can again directly receive communications from other microprocessors. This ability to implement microprocess communications directly significantly decreases the time required to perform a communication in comparison to the time required to implement the communication indirectly through multiple fetches and stores to the M.PSM.

The Virtual Interaction Controller(K.VIC), when it receives a request for a MSV, checks before fetching the MSV from the M.PSM whether some other microprocessor is currently accessing the desired MSV, or the desired MSV is currently connected to a microprocessor. The latter case implies that another microprocessor is already involved in a dialogue with the microprocessor connected to the desired MSV. If either case is true, the Virtual Interaction Controller will place on a FIFO queue associated with the locked-out MSV the address of the microprocessor requesting access. The request to access the locked-out MSV will then eventually get honored when that request is at the head of the queue and the microprocessor currently accessing the MSV again permits access to the MSV. Only one microprocessor can access an MSV at a time because of the semantics of the SBL microprocess interaction patterns which specify that a microprocess can only receive one communication at a time. Thus, a microprocessor must wait until the microprocessor, which is currently attempting to communicate with the desired microprocess, either decides that communication cannot be consummated or completely consummates the communication.

The FIFO queues associated with the locked-out MSV's are contained in a local storage area associated with K.VIC. In addition, this local storage contains the current activity status of each microprocessor and functional unit, and the M.PSM addresses of the MSV's that are currently connected to microprocessors. The maximum number of requests that can be queued is, NP+ND-1, since a microprocessor or functional unit can only initiate a single communication at a time. Thus, the size of the local storage area required by K.VIC is only a small linear multiple of the number of microprocessors and functional units.

A request for an MSV, if it cannot be immediately honored, is queued rather than being reissued by the requesting microprocessor at some later time for two reasons. The first and most important reason is that queuing eliminates the potential for a microprocessor resource deadlocks. A simple example where microprocessor resource deadlock can occur when there is no queuing of requests is the following*.

Example 12: Consider four microprocesses A,B,C, and D, where microprocesses B,C, and D are all connected to microprocessors and are all simultaneously attempting to initiate a communication with A. Suppose that microprocess A is not in the appropriate execution-state to receive a communication from B or C, but is in an appropriate execution-state to receive a communication from D. Given this situation there then exists the possibility that every time microprocess D attempts to fetch the MSV of A, microprocess B or C has locked the MSV of A in an attempt to determine whether A is in an appropriate execution-state to receive a communication. Thus, microprocess D never consummates a communication with A and A never alters its execution-state so that B or C can consummate a communication. Therefore, without a queuing mechanism there exists the potential for introducing in the mapping of virtual PMS activity to physical PMS activity, a hardware deadlock which is not present as a software deadlock.

The situation described in the example is not unrealistic where there are highly structured interaction patterns among a large number of microprocesses. In addition to the queuing mechanism contained as part of the K.VIC, there is also a need for a bus control mechanism e.g., a bus commutator, to guarantee that eventually a microprocessor will be able to get a free bus cycle in order to send out a request for an MSV.

The second reason for queuing requests for a locked-out MSV rather than a microprocessor repeatedly reissuing the request relates to interference on the Interprocessor Bus. The repeated reissuing of

*Knuth (KNU66) has discussed a similar result with respect to Dijkstra's P and V semaphore operations. In particular, if the P and V operations are implemented without queuing, then there exists the possibility that a process Q(i) which performs a P operation will never consummate the operation. This case occurs when there are multiple processes Q(1-n) that attempt to perform a P operation, and there are unbounded number of P operations to be performed.

the request until satisfied would greatly increase the traffic on the Interprocessor Bus and activity of K.VIC. Such an increase in traffic implies a higher probability of a request for an MSV being delayed because either the bus is fully loaded or the K.VIC is busy.

IV.2.1 Microprocessor Scheduling Strategy

The Virtual Interaction Controller is also responsible for dynamic hardware resource allocation in the microcomputer system. There are three types of resources that can be dynamically allocated: 1) blocks of registers in the M.PSM, 2) microprocessors, and 3) functional units. The allocation strategy used for the M.PSM will not be discussed further, except to mention that there does exist hardware implementable techniques for managing the storage of the M.PSM(RIC72, KNU68).

The Virtual Interaction Controller dynamically allocates microprocessors and functional units in a manner analogous to how the data elements of a cache are allocated (GIB67). In particular, the Interprocessor Communication Structure is analogous to a cache in the following ways:

> 1) The relationship between a data element in the cache and its corresponding data element in the large primary memory is analogous to the relationship between a microprocessor contained in the physical PMS and its connected microprocess defined by the virtual PMS.

2) The association of a data word in the cache with a word in the large primary memory is analogous to the connection of a microprocessor to the a microprocess.

3) A direct hit to the cache is analogous to a microprocess interaction pattern being directly implemented over the Interprocessor Bus.

4) A miss in the cache which then requires access to the large primary memory is analogous to a microprocess interaction that cannot be implemented directly over the Interprocessor Bus but instead must be indirectly implemented through modifications to the Process Space Memory.

5) Associating a data word in the cache with a different word in the large primary memory, is analogous to context switching a microprocessor to another microprocess.

6) An empty data word in the cache is equivalent to a microprocessor which is not connected to any microprocess ; a read only data word in the cache is equivalent to a microprocessor which has performed all the work necessary for a context switch, but has not yet been switched to another microprocess; and, a read-write data word in the cache is equivalent to a microprocessor which is currently executing a microprocess.

This analogy has been explored in depth because it provides a convenient framework within which to think about how to schedule microprocesses on microprocessors. In addition, this analogy leads to some interesting ways of looking at the relationship between the number and structure of microprocesses, and that of microprocessors. In particular, the concept of "working set"(DEN68) which is normally applied to data, seems applicable also to microprocesses, i.e., "control working set"; and, the concept of fetching multiple consecutive data items for each line of a cache can also be applied to context switching a group of microprocesses rather than a single microprocess. Though these ideas have not been explored further in their implication on the hardware organization, simulation data will be presented which explores the phenomenon of the "control working set".

The allocation (scheduling) mechanism of the Virtual Interaction Controller is invoked when a request is received to write into the Process Space Memory an MSV which is not currently connected to a microprocessor. If there exists a microprocessor which is not currently connected, then this microprocessor is connected to the MSV. Otherwise, the MSV is connected to the least recently used microprocessor whose associated microprocess is in an expanded, waiting, or suspended execution-state. If there exists no microprocessor whose microprocess satisfies this execution-state condition, then the address of the MSV is placed on a FIFO queue stored in M.PSM until a microprocessor is available.

A microprocessor that is executing a microprocess that reaches the terminated execution-state first unloads into the M.PSM all the context information (local and global data environment registers) associated with the connected microprocess and then signals the Virtual Interaction Controller that it is disconnected from the microprocess. A microprocessor that is executing a microprocess that reaches the expanded, waiting, or suspended execution-state, first signals the Virtual Interaction Controller that it is in an inactive state, and then begins to unload all the context information associated with the microprocess. However, during this unloading period and until the microprocessor is connected to another microprocess, K.IBP is still scanning requests on the Interprocessor Bus which permits the microprocess to be restarted directly over the Interprocessor Bus by another microprocessor. A microprocess also becomes a candidate for being context switched when it gets into a busy-wait state, i.e., when it repeatedly attempts to initiate a communication with a microprocess which is not in an appropriate execution-state to receive the communication. If the queue of microprocesses which desire scheduling is not empty, then this microprocess in a busy-wait state will be disconnected from its microprocessor and put on the scheduler queue. Otherwise, after a suitable number of bus cycles, so as not to saturate the Interprocessor Bus, the busy-waiting microprocess will again attempt to consummate the desired communication. However, a microprocess which is attempting to communicate with a microprocess whose MSV is currently locked-out will not be considered a candidate for disconnection from its microprocessor. This scheduling strategy distinction between a microprocessor waiting to access a locked-out MSV and a microprocessor busy-waiting on an execution-state of an MSV is made because there is some short bound (in part, because of the queuing of locked-out requests) on the maximum time required for an MSV to become unlocked, whereas there is no bound on the time required for an MSV to change to the desired execution-state.

In summary, this scheduling strategy, implemented in the Virtual Interaction Controller for microprocessors is simple, independent of the number of microprocessors, and also maps virtual PMS activity into physical PMS activity without introducing deadlocks. In addition, this scheduling strategy, in conjunction with the Interprocessor Bus, attempts to maximize the number of microprocess interaction patterns that can be directly implemented as microprocessor interaction patterns. As will be seen in the hardware simulation results, this scheduling strategy will take advantage of additional microprocessors added to the microcomputer system to 1) increase the parallel activity of the microcomputer system, 2) increase the number of microprocess interaction patterns that can be directly implemented, and 3) decrease the number of context switches.

IV.3 The Microprocessor Organization

The microprocessor organization has been designed based on the following goals:

1) to configure the internal registers of a microprocessor so as to match the particular internal register configuration of the virtual microprocessor (microprocess) being executed.

2) to reduce the interference among microprocessors caused by accessing of the Microprogram Memory and the Process Space Memory.

3) to make the overhead time required to context switch small.

The first goal is necessitated by the ability to define an arbitrary number of registers that will be contained in the local and global data environment, port, value stack and program counter stack of a virtual microprocessor. Thus, the conventional solution of assigning a fixed set of microprocessor hardware registers for each of these data structures cannot be used in this context. The second goal is necessitated by the fact there may be 32 or more microprocessors all attempting to access the Process Space Memory and Microprogram Memory, possibly some of these microprocessors attempting to access the same microinstruction. Conventional solutions of interleaving memory and multiple access paths to memory are only partial solutions because of the very high traffic to these memories. Thus, some technique is required for cutting down the total traffic to these memories. The third goal is necessitated by the requirement to multiprogram microprocess activity in order to avoid deadlocks. This requirement to multiprogram combined with the short length of the computational activity of a microprocess between successive waits for a communication from other microprocesses leads to a high number of context switches. Thus, a short time for a context switch is quite important. These three design goals can all be satisfied through the concept of a cache per microprocessor.

The microprocessor organization contains an extremely small number of dedicated (specific function) internal hardware registers. The remainder of the internal storage of the microprocessor is structured as a memory cache. The memory cache is used to hold either microinstructions or M.PSM registers which contain the local and global data environment, port, value stack and program counter stack of a virtual microprocessor. The cache per microprocessor concept satisfies the three design goals mentioned previously in the following ways:

> 1) It permits the internal hardware registers to be configured so as to match the register configuration of a particular virtual microprocessor; the cache accomplishes this configuring by attaching tag information to each internal register of the cache; this tag information dynamically associates the contents of a cache register with the contents of a particular M.PSM register of the virtual microprocessor; these internal registers are associatively addressed, based on the tag information, in order to access the contents of a particular M.PSM register.
>
> 2) It reduces the traffic to the Microprogram Memory and the Process Space Memory by allowing a percentage of the accesses and stores of M.PSM registers and the accesses of microinstructions to be accomplished without interaction with the Microprogram Memory and Process Space Memory. In addition, it reduces the likelihood of two microprocessors simultaneously attempting to access the same location in the Microprogram Memory since a copy of the desired microinstruction may already reside in one of the microprocessors.
>
> 3) It reduces significantly the time to context switch because only the M.PSM registers that have been changed need be stored, and only the registers that are required to execute the microprocess need be loaded into the microprocessor. In addition, a microprocessor may immediately begin executing a microprocess as soon as the MSV has been read out of the M.PSM.

The implementation of the cache and its associated control in each microprocessor differs in two ways from how caches are conventionally implemented. The first difference stems from the fact that this microcomputer system contains multiple microprocessors. These multiple microprocessors can be simultaneously executing microprocesses that are communicating through a shared data area in the Process Space Memory. A copy of the contents of these M.PSM registers contained in a shared data area cannot be held in the cache.

If copies of these registers are held in the cache, then there exists the possibility that contents of these cache copies are incorrect because during the period a copy of the register resides in the cache another microprocessor could have modified the original register in the M.PSM. The problem of deciding whether a M.PSM register should be stored in the cache is solved by requiring all accesses and stores of M.PSM registers to be indirect through a descriptor. A M.PSM descriptor specifies among other things the "sharability attribute" of the M.PSM register to be accessed or modified. There are three modes, as previously described, of the sharability attribute: global, coroutine, and local. A register of the virtual microprocessor which has the global attribute will never be held in the cache; a register which has the coroutine attribute may reside in the cache until the virtual microprocessor either is about to enter the waiting, suspended, expanded, or terminated execution-state or is disconnected from its microprocessor. The coroutine attribute indicates that only one virtual microprocessor at a time will access the shared data. Thus, while the virtual microprocessor is in an active execution-state, the coroutine data may reside in the cache. The local attribute indicates that the data may reside in the cache as long as the virtual microprocessor is connected to a microprocessor since no other virtual microprocessor can access the data.

The second difference from a conventional implementation of the cache stems from efficiency considerations. The data environment of a virtual microprocessor is specified by the components of an MSV which, in turn, point to M.PSM descriptors. The M.PSM registers that holds these descriptors are stored in the cache like any other M.PSM register. However, they are being accessed on each microinstruction in order to generate a M.PSM address, for instance, of an element in the local data environment or the top of the value stack, etc. A cache access is normally broken up into two steps: 1) determine whether and where the desired element is in the cache, 2) if it is, then fetch or modify this desired element. In order to make the accessing of the M.PSM descriptors pointed to by the MSV more efficient, the first step required in cache accessing is bypassed. This step is bypassed by having a special set of registers that indicate whether the M.PSM descriptor is in the cache and if it is, then its address in the cache. Thus, the M.PSM descriptors which define the data environment of the microprocess can be directly accessed in the cache eliminating the time required for the associative search step. Additionally, if a M.PSM descriptor is not needed, then it is not brought into the cache.

In addition to these new issues discussed above that arise in implementing a cache memory in the context of this microcomputer system, there are also the following more conventional design issues:

1) The number of registers in each line of the cache.

2) The number of lines in the cache.

3) The technique for line replacement in the cache.

4) The store-through mode, i.e., when a register in the cache is modified, the time at which its corresponding register in the M.PSM is updated.

The technique used for line replacement in the cache is a modified least-recently-used algorithm, where a read-only line will always be replaced before a line which has been modified. The store-through mode, the dimension of the cache, and the size of each line has not been fixed, but rather are parameters that can be set in the simulator of the microprocessor organization. There are three possible store-through modes: never store through until context switch, store through only when bus and memory are free, and always store through. The next chapter will examine the effect of varying these cache parameters on the performance of the microcomputer system.

In summary, this chapter has indicated a coherent and simple design for the microcomputer system which allows parallel activity on the virtual PMS level to be mapped correctly and without significant overhead onto parallel activity in the physical PMS level. The main design techniques applied were:

1) An Interprocessor Bus and its associated control which works like a common data bus so as to map whenever possible microprocess interaction patterns into microprocessor interaction patterns.

2) A built-in FIFO queue mechanism in the Virtual Interaction Controller for ordering access to a locked-out MSV so as to guarantee that no resource deadlock will be introduced in the mapping of virtual activity to physical activity.

3) A memory cache per microprocessor which allows the internal registers to be configured so as to match the particular microprocess being executed and to reduce the memory interference among microprocessors and the time to context switch.

The next chapter will demonstrate these conclusions by examining the results of simulating this microcomputer organization while running the microprogram emulator of Adams' Graph Machine Language described in Chapter III.

V. Simulation Results

This chapter justifies in a quantitative way, through data gathered from a simulator of the hardware organization* discussed in the previous chapter, the following conclusions:

> 1) The emulator for Adams' Graph Machine language is correct, takes advantage of implicit parallelism of a graph procedure, and performs in a parallel manner the overhead operation required to sequence a graph procedure.
>
> 2) The logical hardware design maps virtual PMS parallel activity correctly and without significant overhead into equivalent parallel activity on the physical PMS.

This chapter also examines the effect of varying such paramaters as the number of microprocessors, the number of busses, the interleaving of memory, the size of the cache, and the cycle time of the Process Space Memory on the performance of the microcomputer system. In addition, experimental data that indicates the phenomenon of the "control working set" is presented and discussed.

*The simulator does not simulate exactly the hardware organization described in the previous chapter. In particular, all microprocess interactions are implemented indirectly through the Virtual Interaction Controller rather than some directly over the Interprocess Bus. However, a count is made of the number of interaction patterns that could be directly consummated over the Interprocessor Bus. In addition, the overhead required for allocation of storage in the M.PSM is not counted; this variation does not affect the experimental results to be reported since storage allocation in AGML emulation is only done at the beginning of the run, rather than distributed throughout.

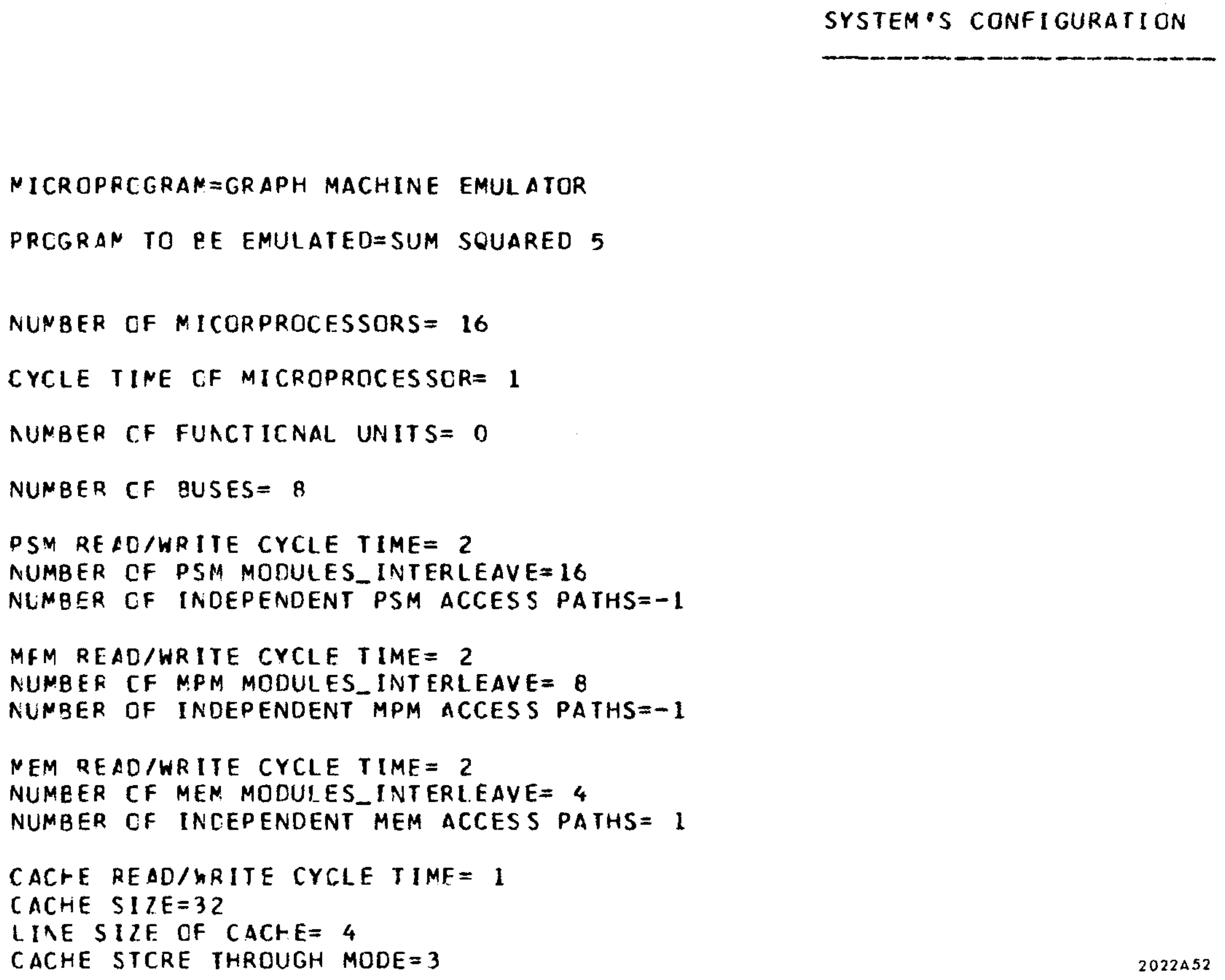
V.1 The Simulator

The simulator is written in PL/I and is structured as a set of coroutine procedures. Each component of the PMS of the microcomputer system that operates asynchronously is simulated by a distinct coroutine procedure. A simulation is performed by executing in a round robin fashion each of these coroutine procedures. Each coroutine procedure is executed until communication to or from another coroutine is desired. Each coroutine keeps track of its internal simulated time, and will not be allowed to consummate the desired communication until its simulated time is equal to the simulated time of the coroutine to which it communicates.

V.1.1 Configuring the Simulator

The input data required by the simulator are (1) a microprogram to be executed, which is stored in the Microprogram Memory, (2) a program and its data to be emulated, which is stored in the Memory Subsystem, (3) a detailed specification of the PMS configuration to be used as an environment within which to execute the microprogram, and (4) trace level specification. The microprogram inputs to the simulator are created by a microassembler. The microassembler program, which is also coded in PL/I, takes as input a symbolic microprogram whose syntax is described in Appendix B. The trace level specification allows the level of summarization of the output data to vary from tracing each bus request to summary statistics for each independent component of the PMS. This trace information has been extremely helpful in debugging the simulator, the hardware design, and the Adams' Graph Machine emulator.

The parameters used to construct a particular PMS configuration for the simulator are shown in Figure 25. These parameters define the number of microprocessors, the cycle time of each microprocessor, the number of functional units and their type, the number of busses to be used for interprocessor communication, the characteristics of the Process Space Memory (M.PSM), Microprogram Memory (M.MPM), and Memory Subsystem (M.MEM), (i.e., their cycle times, interleaving, and the number of requests that each can handle simultaneously), and the characteristics of the cache associated with each microprocessor. If the "NUMBER OF THE INDEPENDENT ACCESS PATHS" parameter is set to -1, then the Interprocessor Bussing Structure is also used to access the desired memory instead of a separate bussing structure dedicated only to that memory. The "CACHE STORE THROUGH MODE" parameter can specify one of four ways of maintaining the cache: (1) always store through, (2) only store through non-descriptor registers, (3) never store through and (4) only store through when bus and memory module are free. The PMS configuration, pictured in Figure 24, is specified by Figure 25.



```
                                                          SYSTEM'S CONFIGURATION
                                                          ----------------------


MICROPRCGRAM=GRAPH MACHINE EMULATOR

PRCGRAM TO BE EMULATED=SUM SQUARED 5


NUMBER OF MICORPROCESSORS= 16

CYCLE TIME CF MICROPROCESSOR= 1

NUMBER CF FUNCTICNAL UNITS= 0

NUMBER CF BUSES= 8

PSM READ/WRITE CYCLE TIME= 2
NUMBER CF PSM MODULES_INTERLEAVE=16
NUMBER CF INDEPENDENT PSM ACCESS PATHS=-1

MPM READ/WRITE CYCLE TIME= 2
NUMBER CF MPM MODULES_INTERLEAVE= 8
NUMBER OF INDEPENDENT MPM ACCESS PATHS=-1

MEM READ/WRITE CYCLE TIME= 2
NUMBER CF MEM MODULES_INTERLEAVE= 4
NUMBER CF INDEPENDENT MEM ACCESS PATHS= 1

CACHE READ/WRITE CYCLE TIME= 1
CACHE SIZE=32
LINE SIZE OF CACHE= 4
CACHE STCRE THROUGH MODE=3                                                  2022A52
```

Figure 25. An Example of a System Configuration for the Simulator

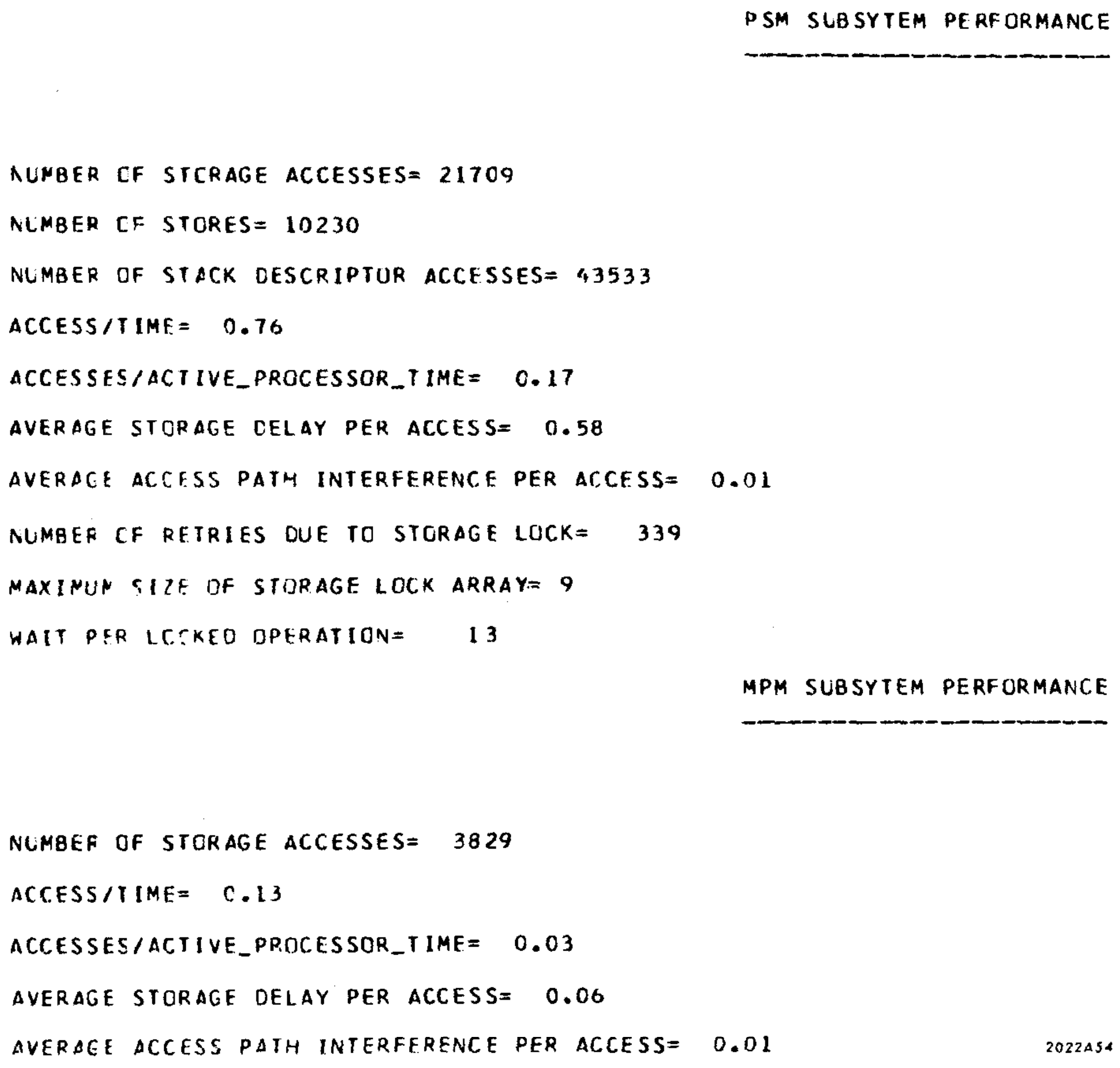
V.1.2 Summary Statistics of the Simulator

The statistics produced from the execution of the simulator display (1) the resource utilization of all components of the PMS configuration, (2) the effect of interference among microprocessors, and (3) the dynamic activity of the computation. An example of the statistics produced from simulating the PMS configuration defined in Figure 25 are displayed in Figures 26a, b, c, and d.

The primary measurements of systems performance are shown in Figure 26a. The "TOTAL TIME TO EXECUTE" indicates the number of bus cycles required to execute the microprogram. The "MICROPROCESSOR_ACTIVITY/TIME" indicates the average number of microprocessors executing during each cycle. This measure of microprocessor activity is referred to as the "allocated parallel activity" of the system. The "INSTRUCTION RETRIES" indicates how many times an ASP or SEL microinstruction was reexecuted, without an intervening context switch, because the communication could not be immediately consummated. The "NUMBER OF MICROPROCESSES GENERATED" indicates the number of MSV's that were created during the simulation, and the "NUMBER OF M.PSM REGISTERS ALLOCATED" indicates the register storage required to define the data environment of these microprocesses. The "QUEUE HISTOGRAM" data indicates the fraction of the "TOTAL TIME TO EXECUTE" during which exactly i microprocesses were queued waiting for an available microprocessor. The "USAGE OVER TIME" indicates the fraction of the "TOTAL TIME TO EXECUTE" during which

```
                                          MICROPROCESSOR SUBSYSTEM PERFORMANCE
                                          ------------------------------------

TOTAL TIME TO EXECUTE = 28465

NUMBER OF INSTRUCTIONS EXECUTED=  9702

MAXIMUM NUMBER OF MICROPROCESSORS USED=16

MICROPROCESSORS_ACTIVITY/TIME=  4.52

AVERAGE INSTRUCTION EXECUTION TIME= 13.25

INSTRUCTION RETRIES=   977

PSM_ACCESSES/INSTRUCTION=  2.24

NUMBER OF MICROPROCESSOR CONTEXT SWITCHS=   853

NUMBER OF MICROPROCESS STEPS QUEUED=  160

NUMBER OF DIRECT ACTIVATIONS OVER BUS=  446

NUMBER OF MICROPROCESSES GENERATED= 54

NUMBER OF PSM REGISTER ALLOCATED= 1111

MAXIMUM NUMBER OF MICROPROCESS STEPS QUEUED= 4

CACHE_ACCESSES= 80775

HIT RATIO=0.884

STORES REQUIRED PER CONTEXT SWITCH TO UNLOAD CACHE=  5.65

STORES REQUIRED IN CACHE MAINTENACE PER SWITCH=  0.06
QUEUE HISTOGRAM=     0     1     2     3     4
PERCENTAGE=        0.972 0.023 0.004 0.001 0.000


NUMBER OF PROCESSOR=           0     1     2     3     4     5     6     7


USAGE OVER TIME              0.00  0.41  0.10  0.10  0.04  0.03  0.04  0.04


                               8     9    10    11    12    13    14    15    16

                             0.04  0.04  0.03  0.03  0.03  0.02  0.01  0.01  0.03
                                                                             2022A51
```

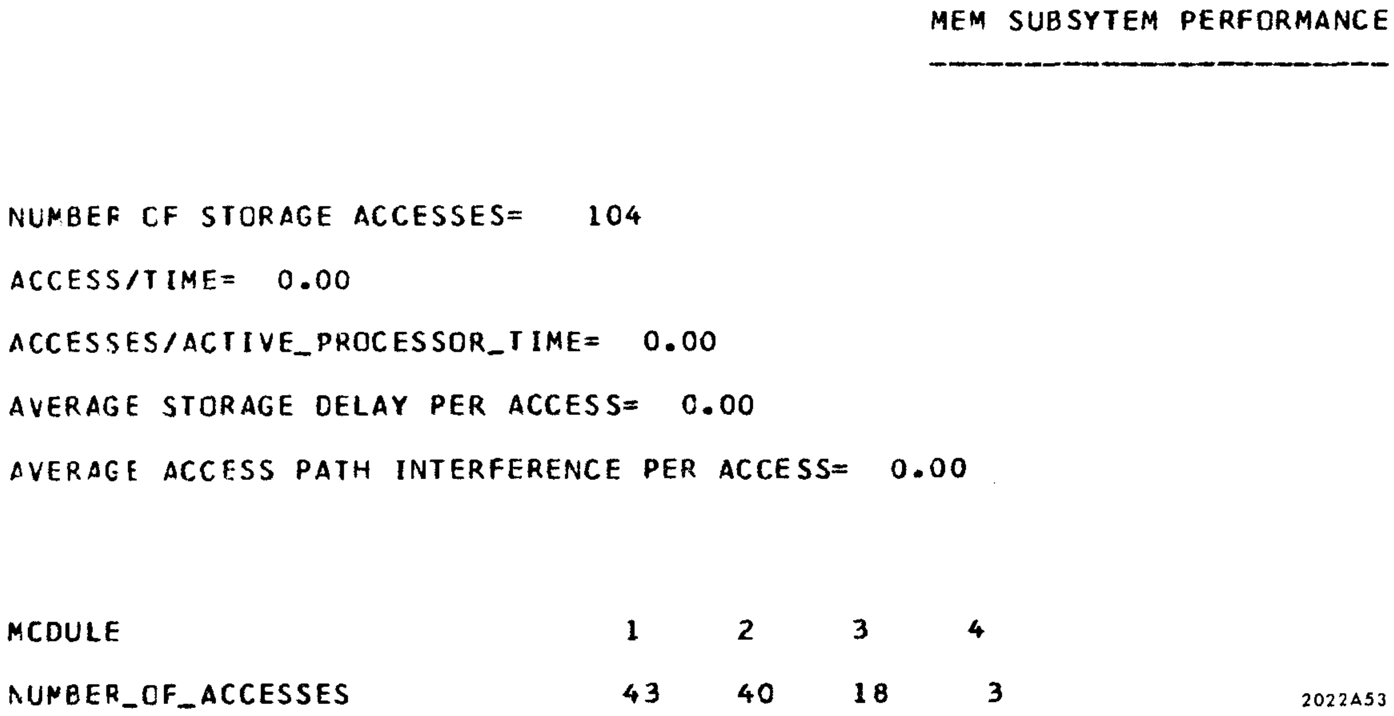
Figure 26a. An Example of Microprocessor Subsystem Performance Statistics



```
                                                    PSM SUBSYTEM PERFORMANCE
                                                    ------------------------


NUMBER OF STORAGE ACCESSES= 21709

NUMBER OF STORES= 10230

NUMBER OF STACK DESCRIPTOR ACCESSES= 43533

ACCESS/TIME=  0.76

ACCESSES/ACTIVE_PROCESSOR_TIME=  0.17

AVERAGE STORAGE DELAY PER ACCESS=  0.58

AVERAGE ACCESS PATH INTERFERENCE PER ACCESS=  0.01

NUMBER OF RETRIES DUE TO STORAGE LOCK=   339

MAXIMUM SIZE OF STORAGE LOCK ARRAY= 9

WAIT PER LOCKED OPERATION=    13
                                                    MPM SUBSYTEM PERFORMANCE
                                                    ------------------------


NUMBER OF STORAGE ACCESSES=  3829

ACCESS/TIME=  0.13

ACCESSES/ACTIVE_PROCESSOR_TIME=  0.03

AVERAGE STORAGE DELAY PER ACCESS=  0.06

AVERAGE ACCESS PATH INTERFERENCE PER ACCESS=  0.01
```

2022A54

Figure 26b.   An Example of M.PSM and M.MPM Performance Statistics



```
                                                  MEM SUBSYTEM PERFORMANCE
                                                  ------------------------


NUMBEF CF STORAGE ACCESSES=   104

ACCESS/TIME=  0.00

ACCESSES/ACTIVE_PROCESSOR_TIME=  0.00

AVERAGE STORAGE DELAY PER ACCESS=  0.00

AVERAGE ACCESS PATH INTERFERENCE PER ACCESS=  0.00



MCDULE                            1     2     3     4

NUMBER_OF_ACCESSES               43    40    18     3
```

2022A53

Figure 26c.   An Example of M.MEM Performance Statistics

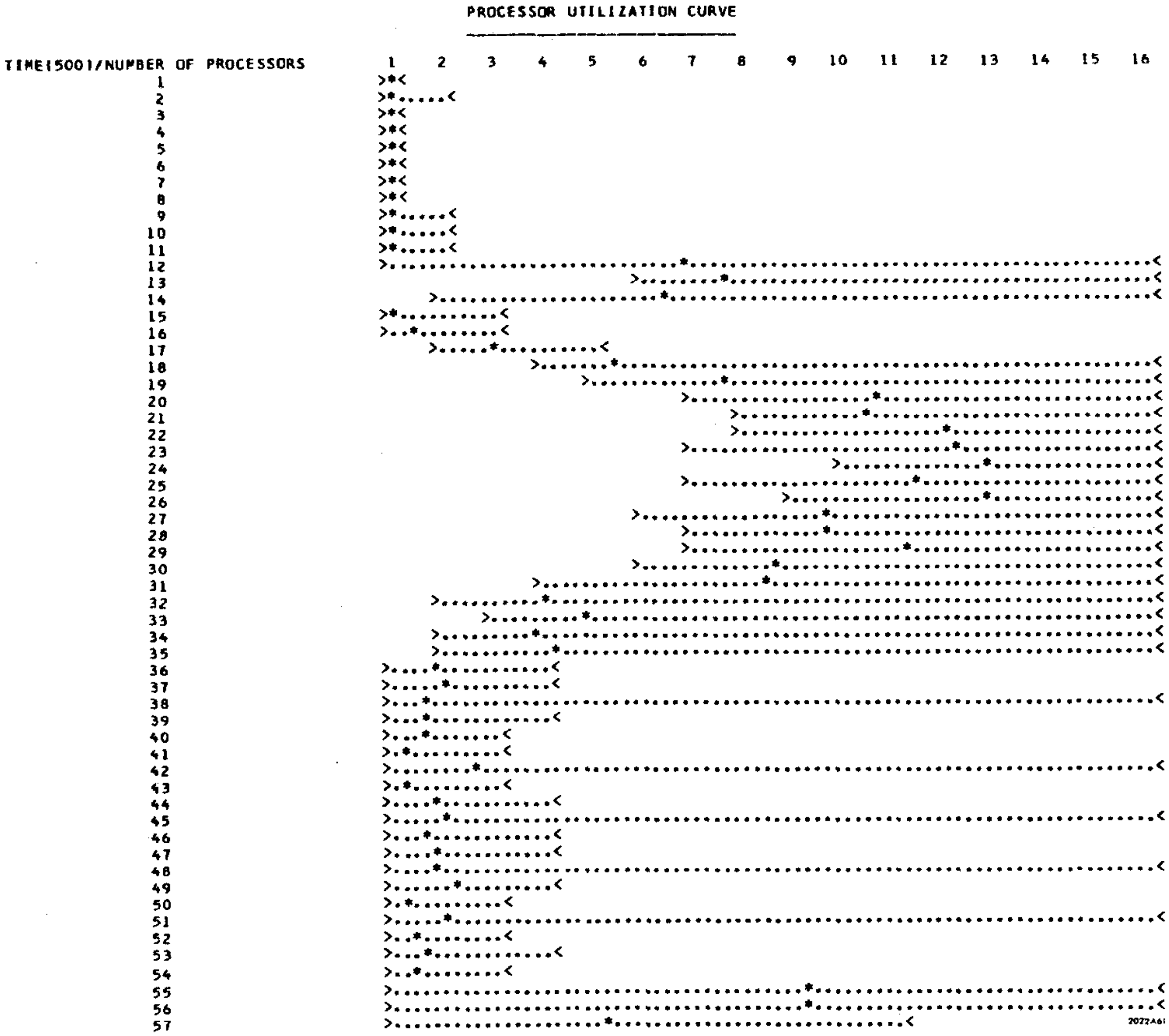


Figure 26d. An Example of Processor Utilization Statistics

exactly i microprocessors were executing. The "(MICROPROCESS) QUEUE HISTOGRAM" and "(MICROPROCESSOR) USAGE OVER TIME" statistics combined together provide a good measure of the distribution of parallel activity on the virtual PMS level.

The performance measurements of the memories and bussing structures are pictured in Figures 26b and 26c. The "ACCESS/TIME" indicates the average number of requests per bus cycle that each memory received. The "ACCESS/ACTIVE-PROCESSOR-TIME" is a normalization of the "ACCESS/TIME" statistics based on the average allocated parallel activity. The "AVERAGE STORAGE DELAY PER ACCESS" indicates the average time requests to memory were delayed in servicing because of hardware interference. There are two points at which interference is measured: (1) access to the bussing structure which connects to the memory and (2) access to a particular module of the memory. The "AVERAGE ACCESS PATH INTERFERENCE PER ACCESS" separately specifies the interference caused by overloading the bussing structure to memory. The "NUMBER OF RETRIES DUE TO STORAGE LOCK" indicates how many requests for MSV's, from the M.PSM, were queued by the Virtual Interaction Controller. The "MAXIMUM SIZE OF STORAGE LOCK ARRAY" indicates the maximum number of microprocessors that were simultaneously waiting for access to locked-out MSV's. The "WAIT PER LOCKED OPERATION" indicates the average time a request for an MSV spent on the queue before it was serviced. In addition, there is data available on the distribution of these queued requests based on MSV's.

The dynamic activity of the computation is characterized by Figure 26d. The dynamic activity is plotted by displaying the minimum (>), maximum (<), and average (*) number of microprocessors that were utilized during the lifetime of the computation. This microprocessor utilization curve is based on a time period of 500 bus cycles. In addition, statistics are available for constructing the activity curve for each microprocess that was created during the computation. In particular, these statistics specify the total time spent in executing a microprocess, and the time periods in which a microprocessor was executing this microprocess.

V.2 The Dynamic Performance Characteristics of the AGML Emulator and the Hardware Organization

The first step in evaluating the performance of the AGML emulator and the hardware organization for the microcomputer architecture, is to verify their correctness. Their correctness was experimentally verified by simulating, on a wide range of PMS configurations, the AGML emulator emulating a variety of graph programs. The performance statistics to be presented in the remainder of the chapter are based on two of these graph programs, Figures 27 and 28.

The first graph program, Sum-Squared in Figure 27, calculates the sum of the squares of the elements of a vector of numbers; the vector is placed on the external input link with its last

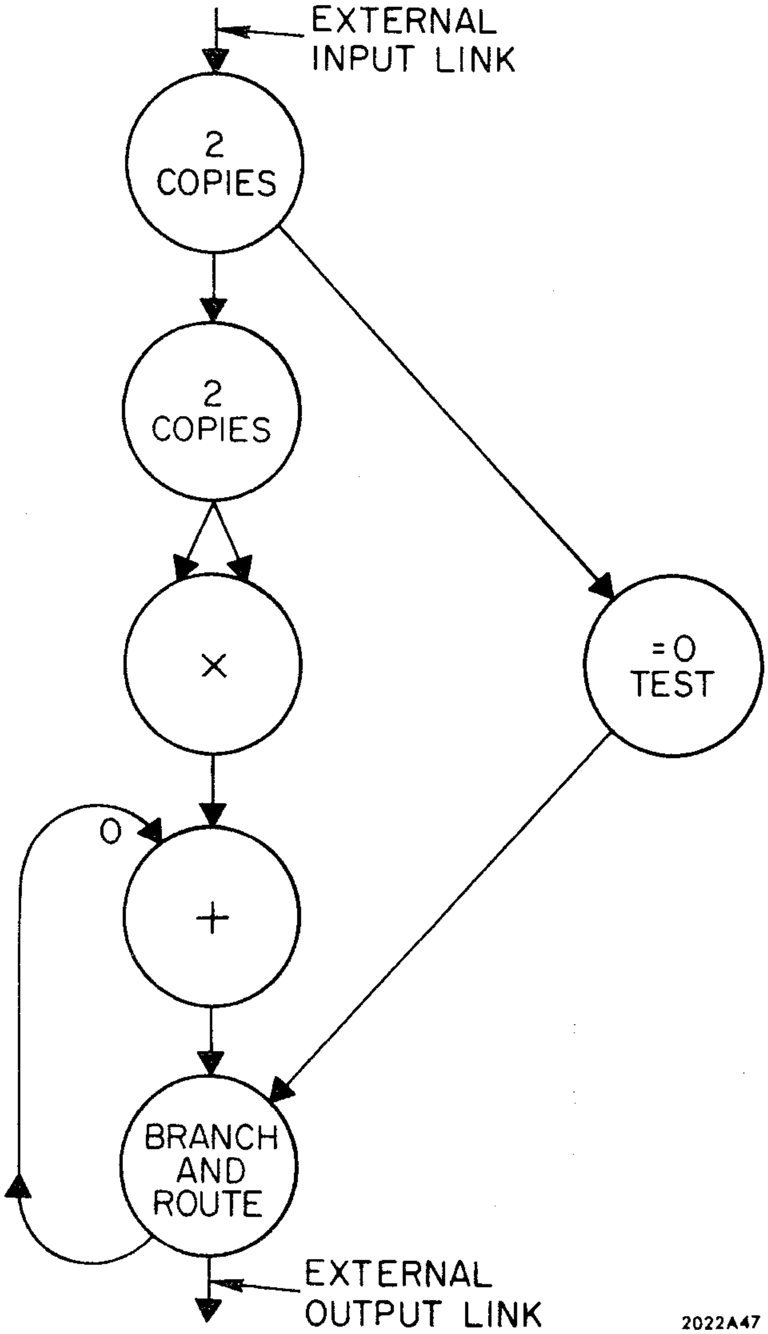


Figure 27. Sum-Squared Graph Program

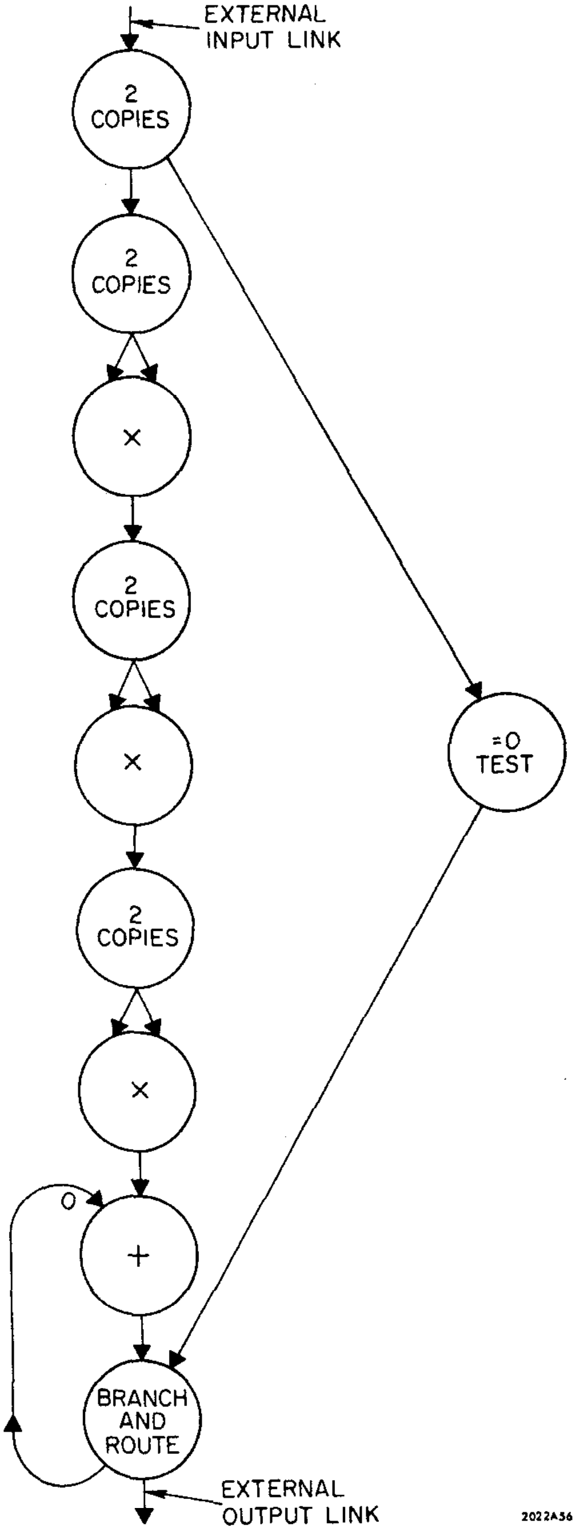


Figure 28. Sum-Eighth-Power Graph Program

element being zero. The node "2 copies" copies the data on its input data link to its two output links. The node "Branch and Route" routes the data on its first input link (connected to the "+" node) to the external output link if its second input link (connected to the "=0" node) contains a true value; otherwise, the output data from the "+" node is routed back to the "+" node for continued summing. The computational structure of this graph program can be thought of as a three level pipeline that flows into an iterative summation network.

The second graph program, Sum-Eighth-Power in Figure 28, calculates the sum of the eighth power of the elements of a vector of numbers. The computational structure of this second graph program is similar to that of the first graph program except that the pipeline part of the computation has seven levels. This extension of the pipeline, as will be verified by performance statistics, increases the inherent parallelism of the Sum-Eighth-Power graph program in comparison to that of the Sum-Squared program.

V.2.1 Measuring Parallel Activity

The parallel activity of an AGML graph program can be measured on three "levels of abstraction"(RID71,HOR70), as pictured in Figure 29. The comparison of parallel activity among these three levels is used to justify the conclusion stated at the beginning of the chapter.

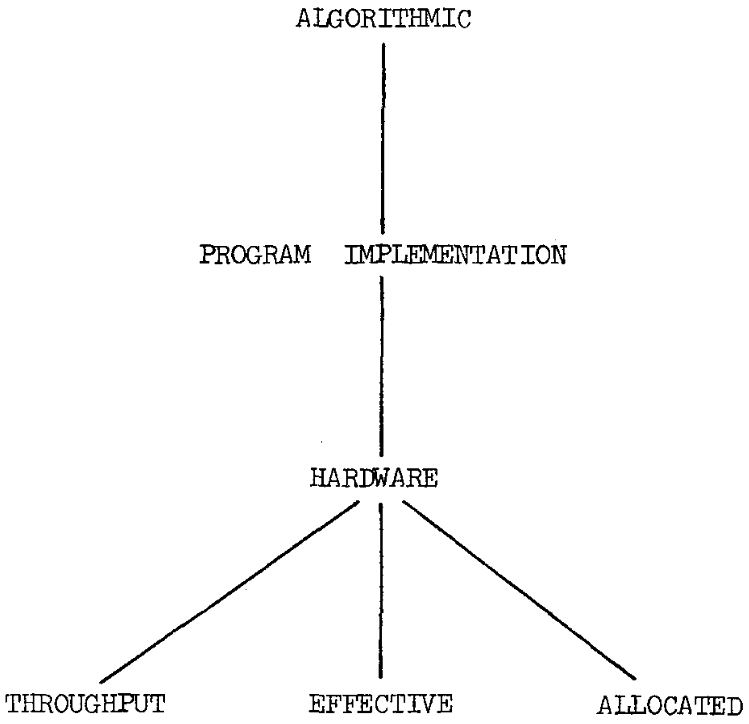


Figure 29. Three Levels of Parallel Activity Measurement

The first measure of parallel activity, "algorithmic parallelism", is based soley upon the sequencing rules of the AGML, i.e., the rules which define when a node may execute. The algorithmic parallelism does not take into consideration any of the bookkeeping operations, both software and hardware, required to implement the parallel activity of the graph program, e.g., the fetching and storing of data on links, the monitoring for when a node can fire. The algorithmic parallelism of the AGML program, as a function of the number of processors, can be measured through the use of a simulation technique developed by Nelson(NEL70)*.

The second measure of parallel activity, "program implementation parallelism" is based on the parallel activity on the virtual PMS level of the AGML emulator when emulating the graph program. The program implementation parallelism takes into consideration the bookkeeping operations at the programming level required to implement a graph program but does not take into consideration (1) the bookkeeping operations at the hardware level required to map virtual PMS activity into physical PMS activity, (2) the hardware interference among microprocessors caused by simultaneous access to a memory module or the bussing structure, and (3) the program interference among microprocesses caused by a microprocess repeatedly attempting to communicate with another micorprocess which

*In using Nelson's simulator to compute the algorithmic parallelism, it has been assumed that all node operations require the same amount of time.

is not in the appropriate execution-state to receive a communication, i.e., the number of instruction retries.

The program implementation parallelism of the AGML program cannot be precisely measured through statistics produced by the simulator. Thus, instead of a precise measurement, an approximation of the program implementation parallelism has been obtained by altering a graph program to reflect the major bookkeeping operations at the programming level. A graph program has been altered by adding dummy nodes along the links of the graph program so as to reflect the overhead operation of fetching and storing of link data (see Figure 30). The algorithmic parallelism of this altered graph program is then used as an approximation to the program implementation parallelism*.

The third measure of parallel activity, "hardware parallelism", is based on statistics produced by the simulator interpreting the AGML emulator when emulating the graph program. There are three related measures of hardware parallelism that are important to the evaluation of the performance of logical hardware design: 1) the "throughput parallel activity" which is calculated by dividing the "TOTAL TIME TO EXECUTE" for n microprocessors by that for a single microprocessor; 2) the "effective parallel activity" which is

*In computing the program implementation parallelisms, it has been assumed that the overhead operation represented by dummy nodes requires twice as much time as a regular node operation. This relationship of overhead to computation was determined experimentally through statistics produced by the simulator on how much real-time each microprocess took to compute.

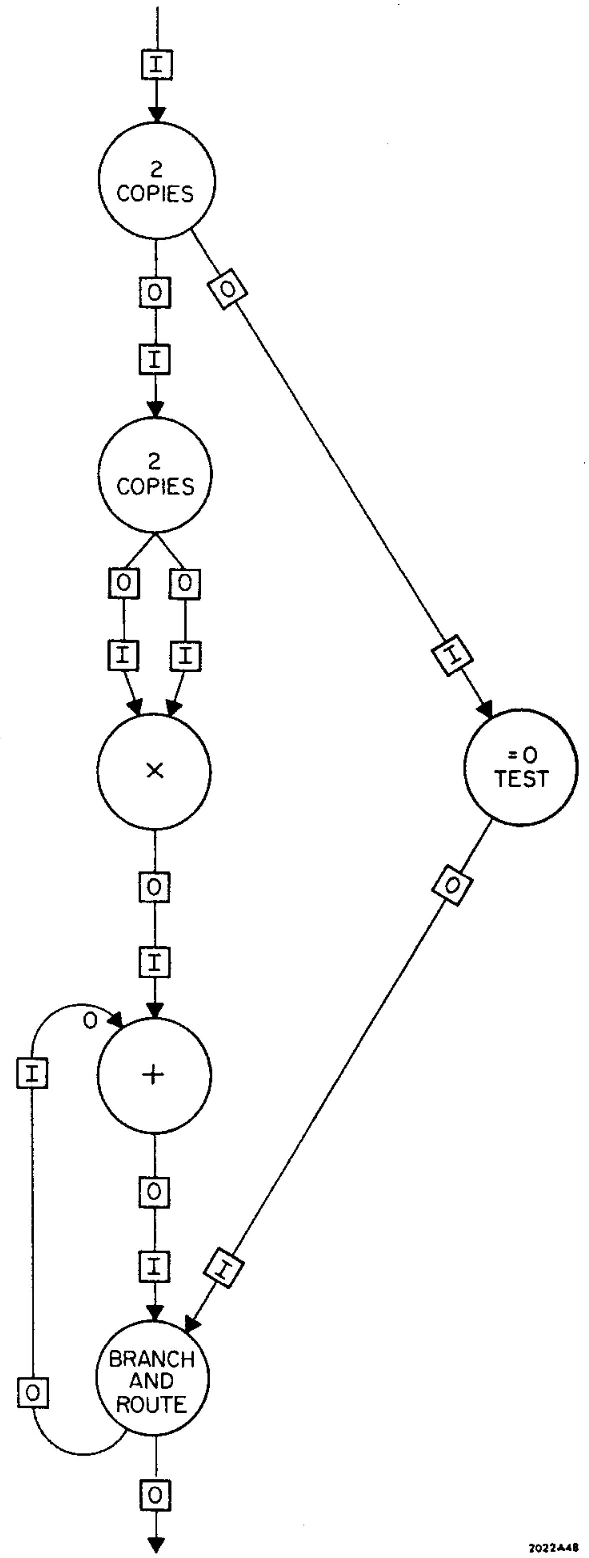


Figure 30. Modification to Sum-Squared Graph Program to Measure Program Implementation Parallelism

calculated by normalizing the "allocated parallel activity" by the results of dividing the "AVERAGE INSTRUCTION EXECUTION TIME" for a single microprocessor by that for n microprocessors; 3) the "allocated parallel activity" which, as previously described, is a measure of the actual parallel activity exhibited by the microprocessors. The effect of program interference as a function of the number of microprocessors can then be observed by comparing the throughput hardware parallelism to the effective hardware parallelism. In the same way, the effect of hardware interference and hardware bookeeping operations as a function of the number of microprocessors can be observed by comparing the effective hardware parallelism to the allocated hardware parallelism.

V.2.2 The Performance Characteristics of the AGML Emulator

The performance characteristics of the AGML emulator have been evaluated by comparing the algorithmic parallelism, the program implementation parallelism, and the hardware throughput parallelism of the Sum-Squared graph with a 5 element vector. The graphical display of each of these parallelism measures as a function of the number of available (micro)processors is pictured in Figure 31. A comparison of these parallelism curves indicates the following conclusions concerning the performance of the AGML emulator:

1) The emulator takes advantage of the implicit parallelism of a graph program (compare algorithmic to throughput curve);

2) The emulator performs in a parallel manner the overhead operations required to sequence a graph program (compare algorithmic to virtual and throughput curves).

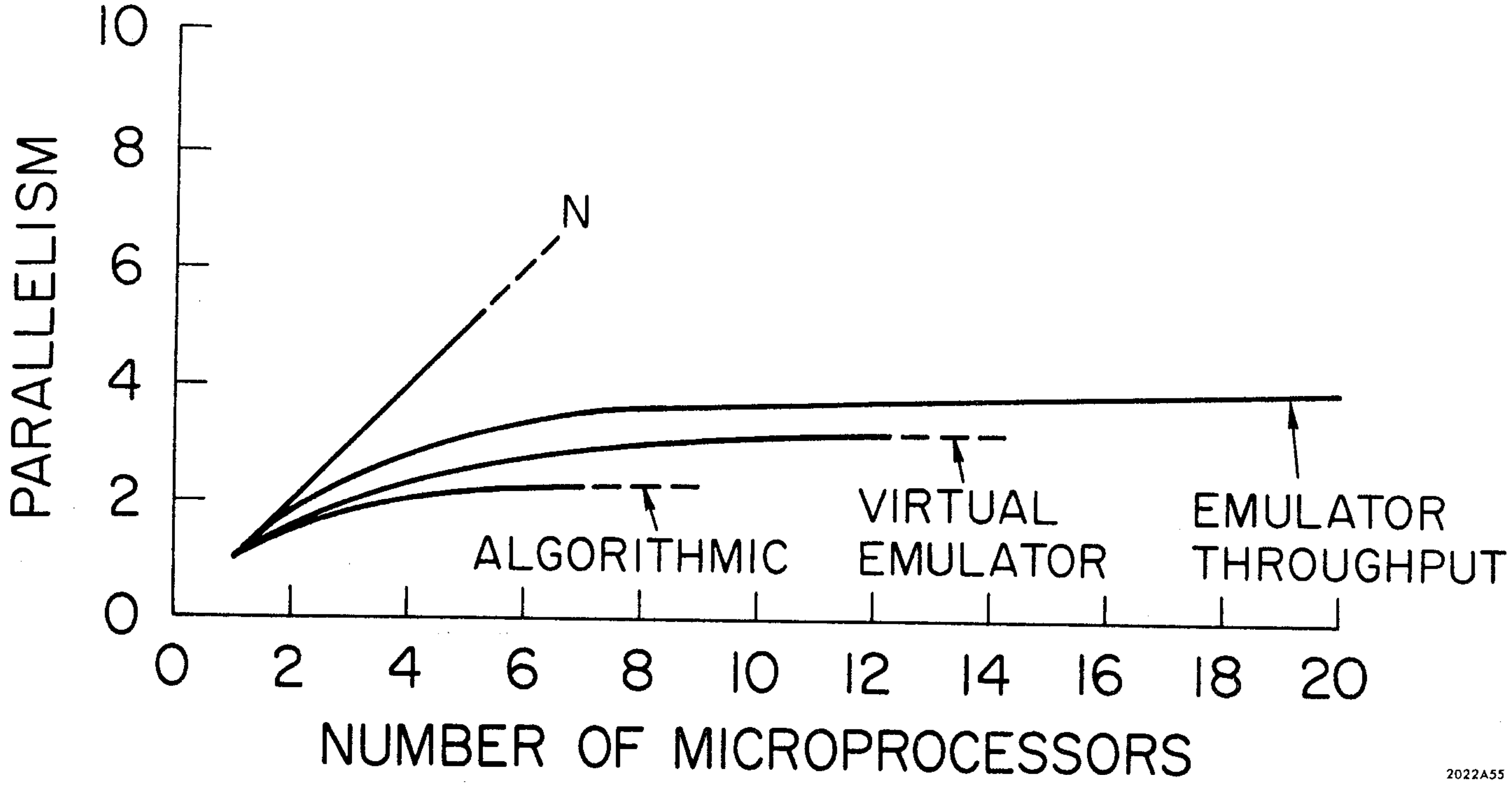


Figure 31 Performance Characteristics of AGML Emulator on Sum-Squared Graph Program(5)

In addition, the comparison between the virtual and throughput parallelism curves indicates that the logical hardware organization configured appropriately can map virtual PMS activity to physical PMS activity without significant overhead.

These conclusions can be substantiated on a more quantitative basis by examining the fit(DER69) of the throughput parallelism curve to the following hyperbola:

$$T(n)= a+b/(n-c),$$

where n is the number of microprocessors and T(n) is the run time in terms of units of a 1000 bus cycles. The result of the fitting is:

$$T(n)= 30 + 120/(n-0.19).$$

A similar curved fitting to the throughput curve of the Sum-Eighth-Power graph with a 10 element vector, pictured in Figure 32, is:

$$T(n)= 40 + 270/(n-0.1).$$

Both curve fittings, which at maximum varied less than 3 percent from the experimental data, indicate that a major part of the computational activity of the AGML emulator can be performed in a parallel way as a function of 1/N.

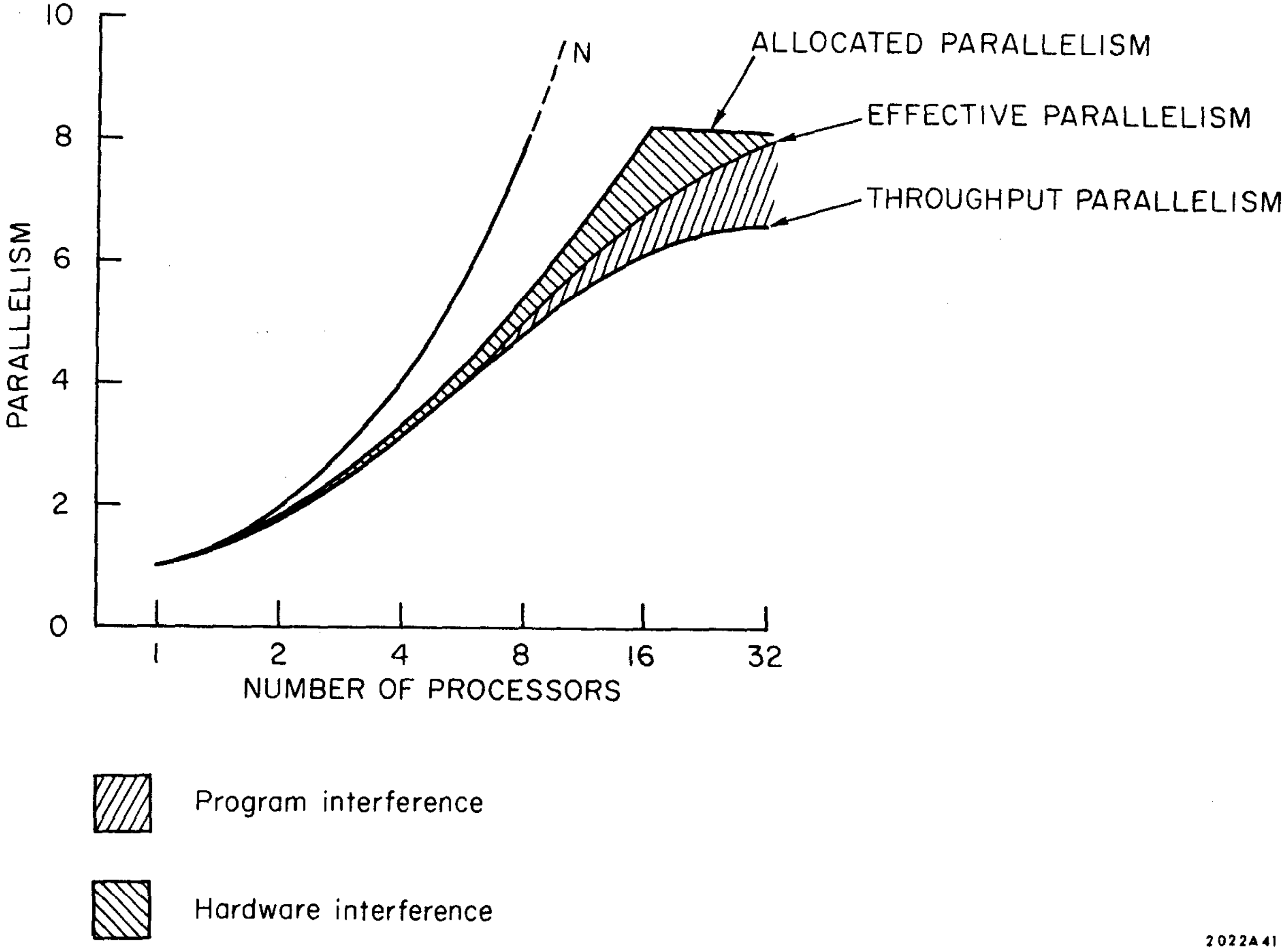


Figure 32 The Effect of Hardware and Program Interference on Sum-Eighth-Power Graph Program(10)

The dynamic performance characteristics of the AGML emulator can also be observed by examining the Microprocessor Utilization Curves of the Sum-Eighth-Power graph program in Figures 33a-e. The dynamic activity of the AGML emulator can be partitioned in terms of six sections, as labeled in Figure 33a. The activity of the first section, which is mostly sequential, represents the dynamic construction of the CDS for the particular graph program being emulated. The activity of the second section represents the initiation of all nodes in the graph, and their subsequent activity involved with determining whether they can execute. The activity of the third section mirrors the gradual initiation of the pipeline part of the graph computation. The activity of the fourth section mirrors the execution of a fully loaded pipeline. The activity of the fifth section mirrors the unloading of the pipeline part of the computation followed by the iterative summation part of the computation. Finally, the activity of the sixth section represents the termination of all the nodes of the graph after the final output appears on the external output link. This sequence of microprocessor utilization curves indicates that an AGML emulator can use available microprocessors, where sufficient parallelism exists, to reduce in a linear way the time it takes to complete each of the sections of the curve. In addition, Figure 33d indicates that the logical hardware design can efficiently handle sustained parallel activity, involving highly structured interaction patterns, of greater than sixteen microprocessors.

V.2.3 The Performance Characteristics of the Hardware Organization

The previous section showed that the virtual PMS activity can be mapped without significant overhead into equivalent parallel activity on the physical PMS. This section will discuss, in detail, how this overhead varies as a function of the amount of parallel activity and the number of microprocessors.

The two components of overhead, hardware interference and program interference, are analyzed by comparing the three measures of hardware parallelism: throughput, effective and allocated hardware parallelism. These three measures are plotted as a function of the number of microprocessors for the Sum-Eighth-Power graph program in Figure 32.

The first observation which can be made from Figure 32 is that the hardware interference and program interference increase as a function of the number of microprocessors. This observation on hardware interference can be explained in the following way: the more microprocessors, the more virtual PMS parallel activity that can be exploited as physical PMS parallel activity; in turn, the more physical PMS activity increases the likelihood of overloading the bussing and memory structures; this overloading leads to a longer time

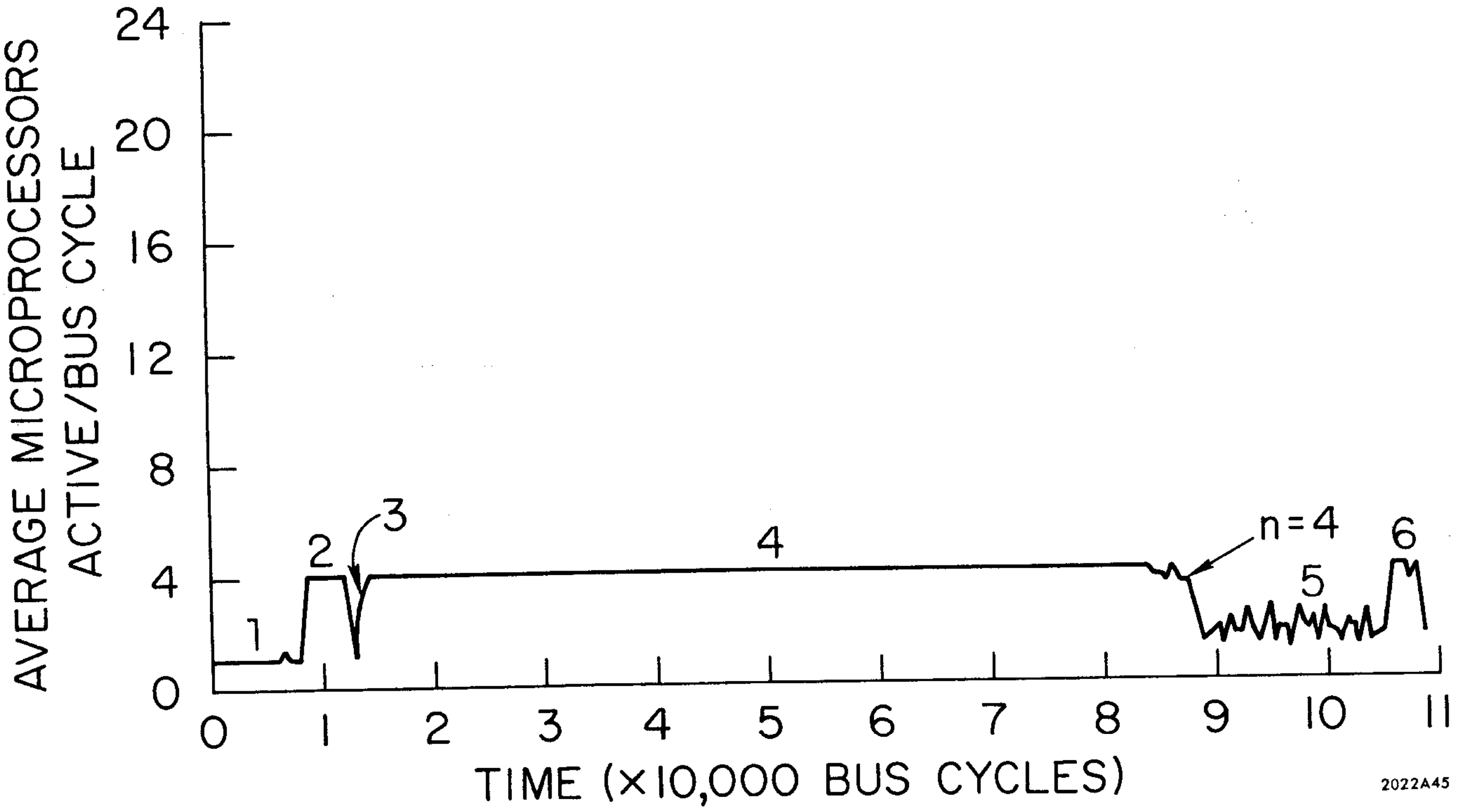


Figure 33a. Microprocessor(4) Utilization Curve for Sum-Eighth-Power Graph Program(10)

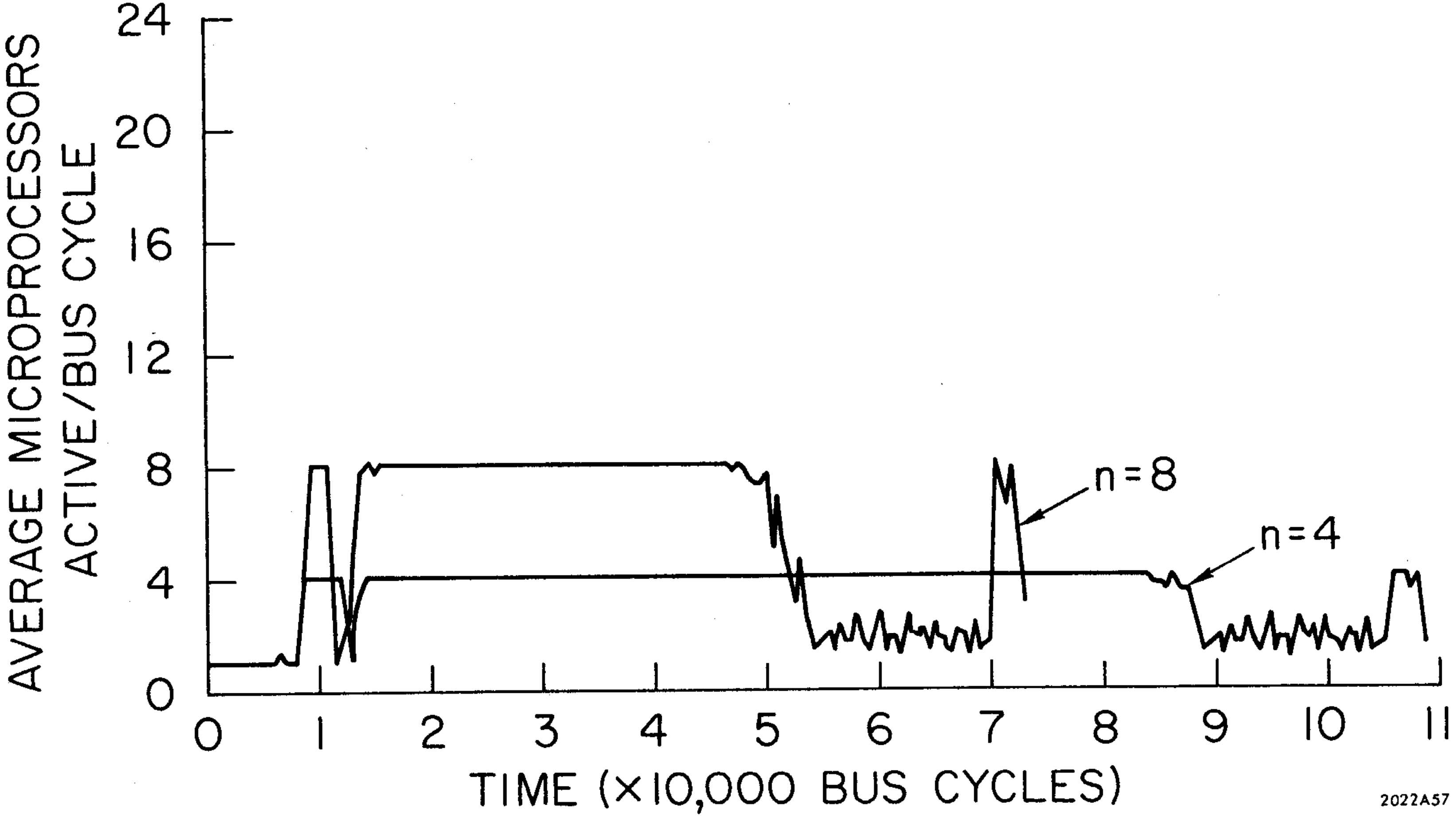


Figure 33b. Microprocessor (4 vs 8) Utilization Curve Sum-Eighth-Power Graph Program(10)

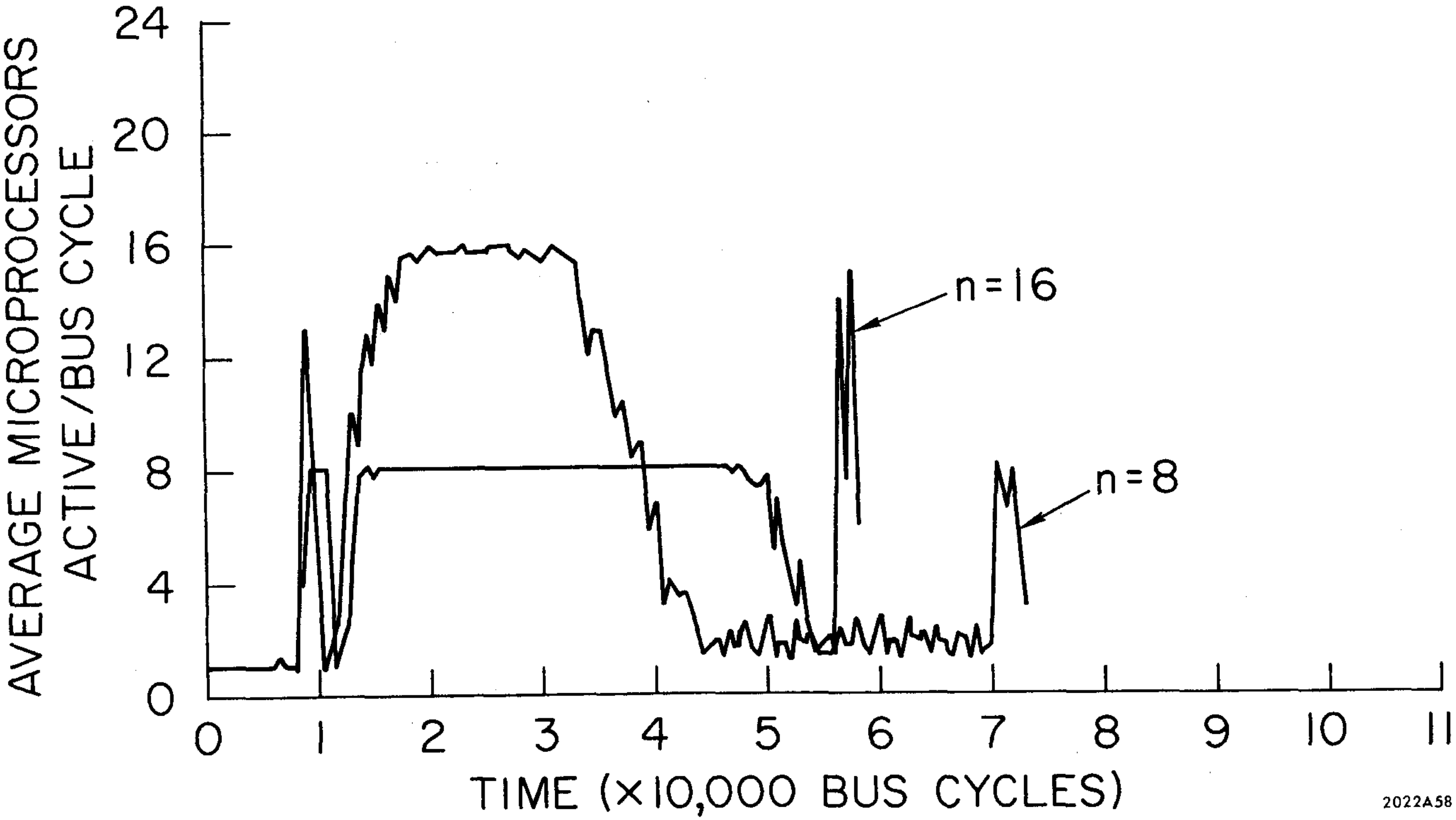


Figure 33c. Microprocessor (8 vs 16) Utilization Curve Sum-Eighth-Power Graph Program(10)

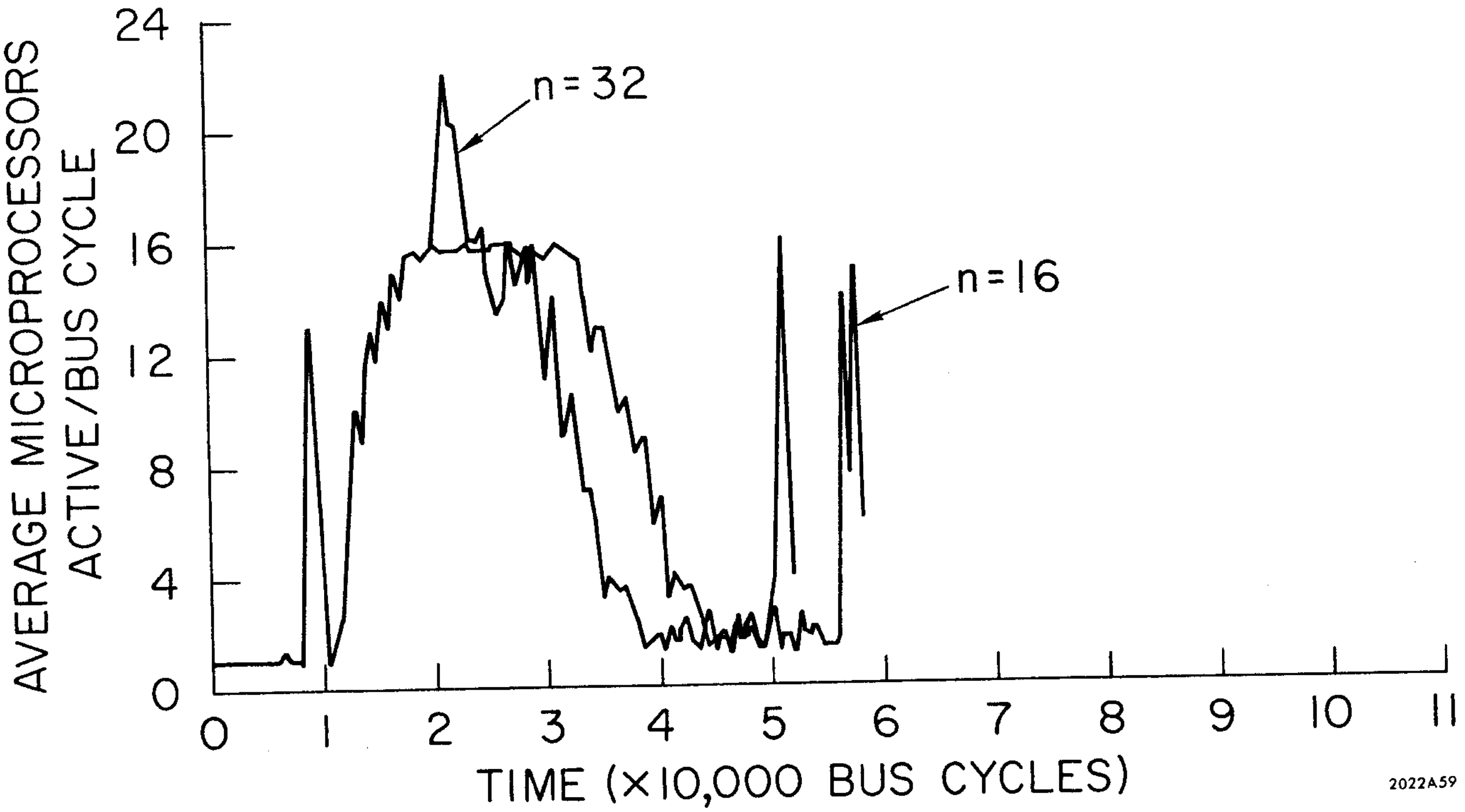


Figure 33d. Microprocessor (16 vs 32) Utilization Curve Sum-Eighth-Power Graph Program(10)

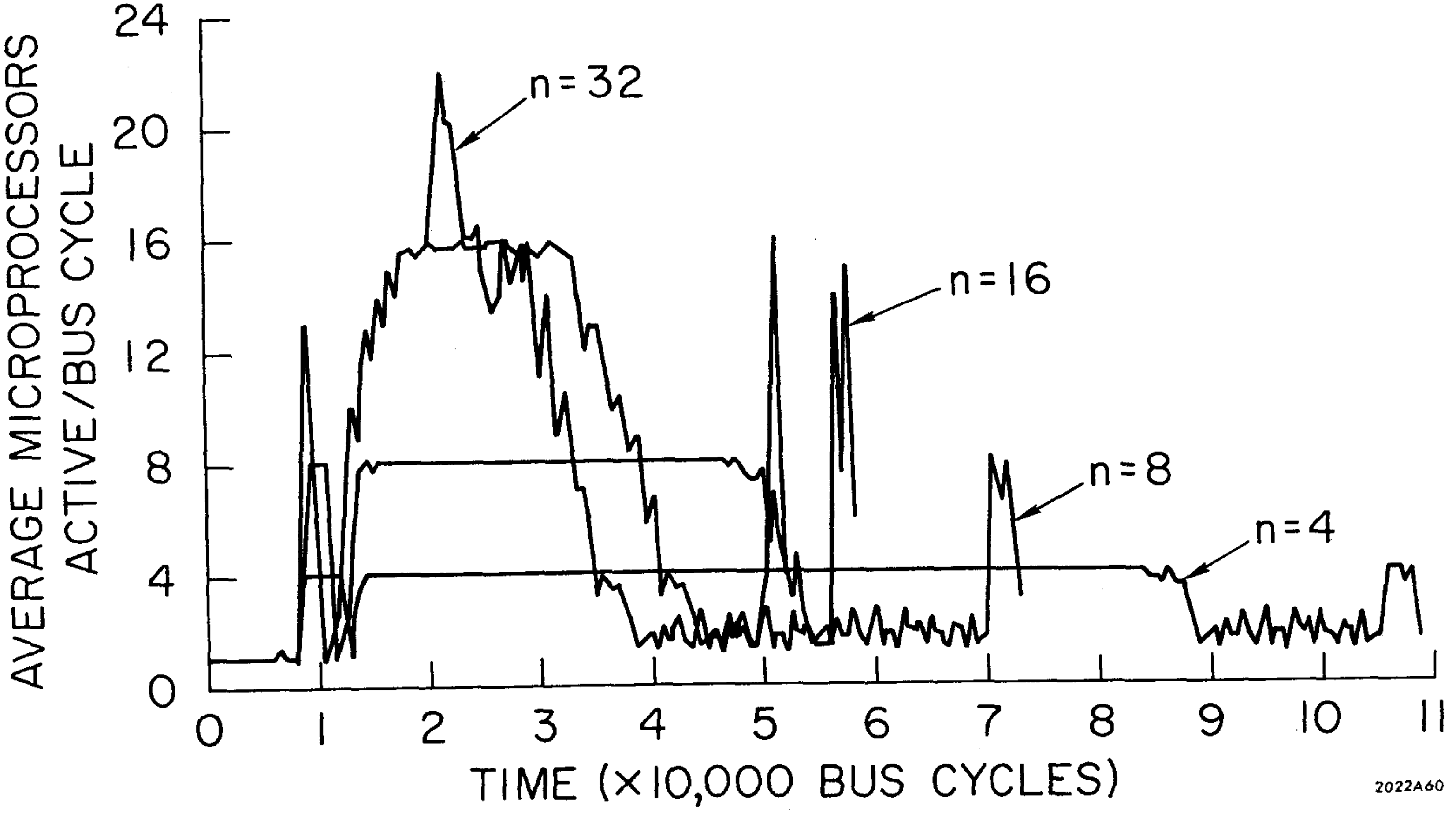


Figure 33e. Microprocessor (4,8,16,32) Utilization Curve
Sum-Eighth-Power Graph Program(10)

to execute microinstructions in the more highly parallel sections of the emulator activity; therefore, the allocated parallelism which measures the amount of parallel microprocessor activity increases in relation to the effective parallelism which measures the amount of parallel microprocessor activity normalized to the the average execution time of a microinstruction.

The increase in program interference can be explained in a similar way, except in this case, the resource causing interference is a microprocess: in a computation involving highly structured interaction patterns, the more parallel activity leads to the greater likelihood that a microprocess will attempt to communicate with another microprocess which is busy; thus, there will be an increase in busy-waiting time caused by repeatedly attempting to consummate a communication. This busy-waiting time is reflected as an increase in the number of microinstructions executed, which causes the increasing difference between the throughput and effective parallelism curves.

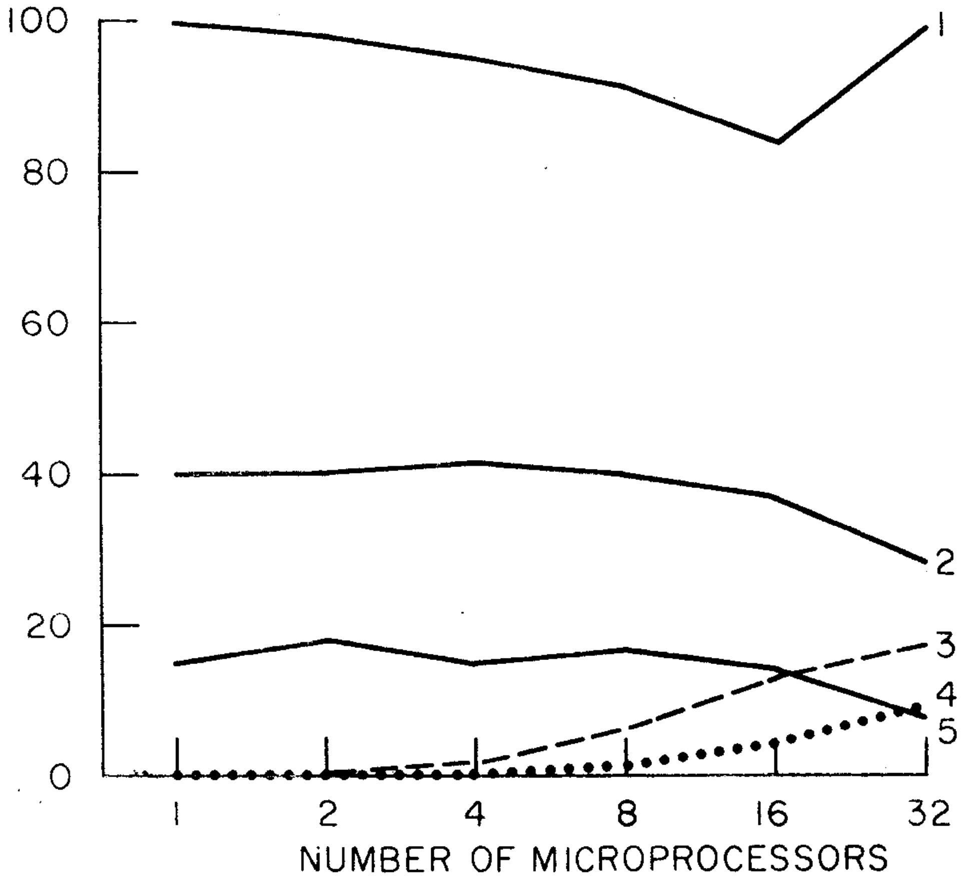
The second observation which can be made from Figure 32 is that the difference between the allocated and effective parallelism above sixteen microprocessors decreases rather than increases or remains constant, as would be expected. This anomaly occurs because the difference between the allocated and effective parallelism, pictured as curve 1 in Figure 34, not only is a measure of hardware interference but also measures the hardware overhead functions involved in mapping virtual PMS activity onto physical PMS activity.

The major components of this hardware overhead are the time to context switch and the time to fetch data for a miss in the cache. As seen from Figure 34, these two components of hardware overhead (see Curves 2 and 4) decrease above sixteen microprocessors counter-balancing the effect of increasing hardware interference (see Curves 3 and 5). The proportional decrease in time spent doing hardware overhead functions can be explained in terms of the "control working set" phenomenon previously discussed in Chapter IV.

The phenomenon of the control working set is graphically displayed in Figure 35: above 16 microprocessors there is a sharp drop off in the number of context switches, the number of microprocesses queued, and a correspondingly sharp increase in the number of direct activations over the interprocessor bussing structure. The decrease in the time spent on hardware overhead function is thus easily explained:

> 1) The decrease in the time spent on context switching occurs because there are significantly fewer context switches.
>
> 2) The decrease in time spent on fetching data for a microprocessor's cache occurs because there is a much higher probability that a microprocess can remain connected to a microprocessor while waiting for a communication. The data working set of microprocesses will thus have to be reassembled much fewer times. This phenomenon is substantiated on another level through looking at the cache miss ratio which significantly decreases with more than sixteen microprocessors, i.e., 13 percent to 10 percent.

This control working set phenomenon of a sharp decrease in hardware overhead above a certain number of microprocessors is directly



TA(n) = (Time to run with n microprocessors) ×
(Average allocated parallel activity)

1 (Average instruction execution time for single microprocessor)/
(Average instruction execution time for n microprocessors)

2 (Time required to fetch data on cache miss for n microprocessors)/TA(n)

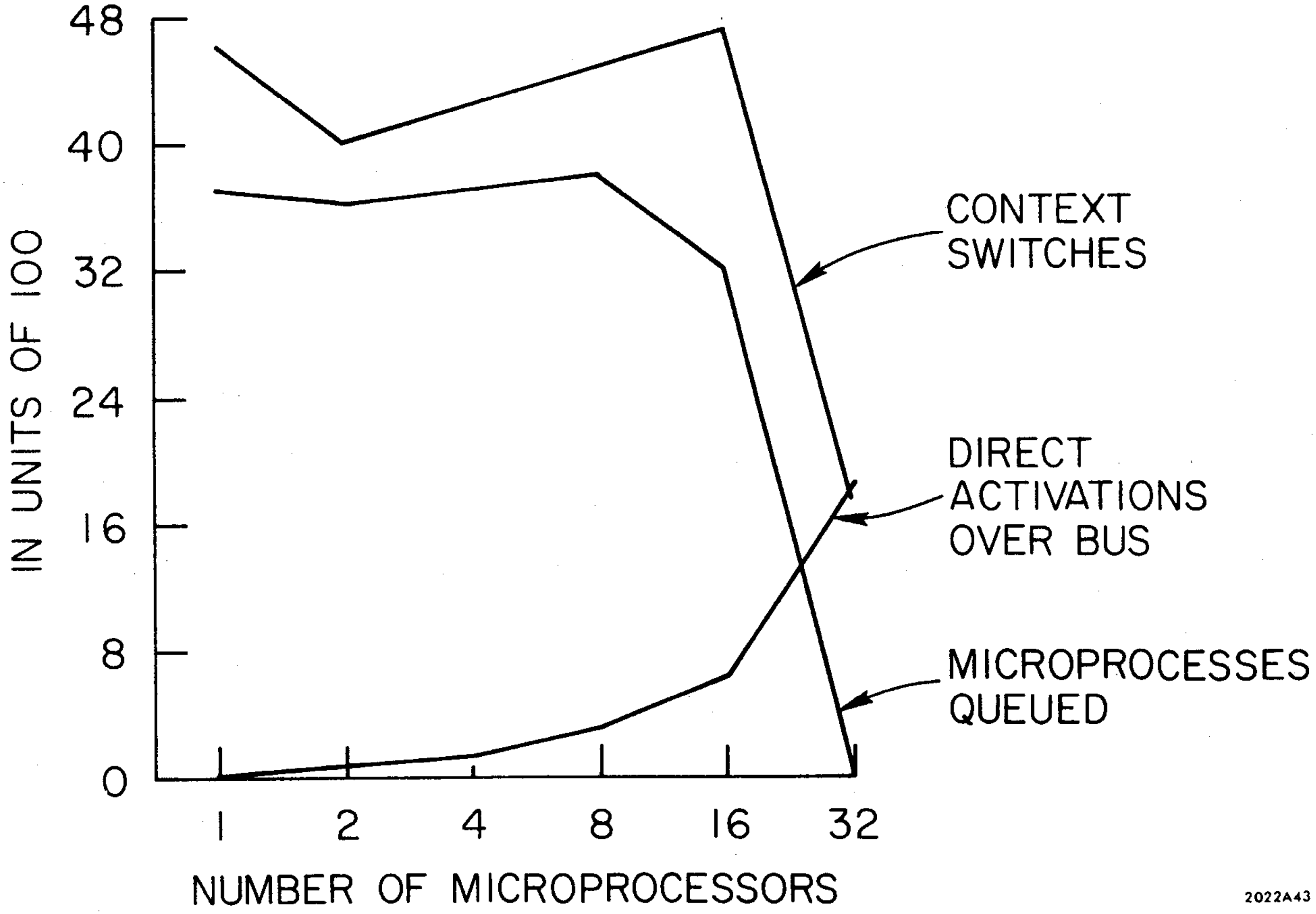
3 (Time caused by PSM interference for n microprocessors)/TA(n)

4 (Time caused by delay in accessing locked MSV for n microprocessors)/TA(n)

5 (Time required to context switch for n microprocessors)/TA(n)

2022B42

Figure 34. Distribution of Hardware Interference for Sum-Eighth-Power Graph Program(10)



2022A43

Figure 35. Effect of Varying the Number of Microprocessors on the Number of Context Switches, Interprocessor Communications, and Microprocesses Queued for Sum-Eighth-Power Graph Program (10)

analogous to the data working set phenomenon of thrashing which occurs when there are too few physical data pages to hold the data working set of the program.

V.2.4 Future Research -- Choosing a PMS Configuration

A major research area that needs further exploration is the development of techniques for choosing an optimal PMS configuration. The choice of an optimal PMS configuration is very difficult considering the large number of parameters that must be set in order to specify a configuration. The configuration that was used for the simulation results presented in the previous section is pictured in Table 5. This configuration was chosen through a trial and error approach, together with some systematic varying of parameters, shown in Figures 36a, b, and c.

Some tentative conclusions from this trial and error search are the following:

1) The interprocessor bussing structure configured to handle 8 bus requests simultaneously with at least up to 16 microprocessors connected, can be used as the access path to the M.PSM and M.MPM without major interference problems.

2) The cache store through mode of never storing through unless necessary seems to be optimal. The mode of only storing through when there are available bus and memory cycles is surprisingly not the best. The explanation seems to be that though the memory is available, the storing into memory will tie up the memory for cycles in the future, thus causing interference for future accesses. However, if the storing through is allowed in some fraction of the available bus cycles, rather than on every available cycle, then this

mode approaches (or could be possibly better than) the never store through mode.

3) The interleaving of the Process Space Memory should be at least as much as the average allocated parallelism expected in the system.

4) A cache of 128 32 bit words configure as 32x4 seems appropriate for handling microprograms of short duration with small data working sets. A cache configuration of 64x2 has significantly lower cache hit ratio.

5) The proportional effect of varying M.PSM cycle time on throughput seems to be independent of the number of microprocessors.

These conclusions are extremely tentative because of the small sample space of microprograms that were emulated. There should be future research directed toward a more careful examination of these conclusions.

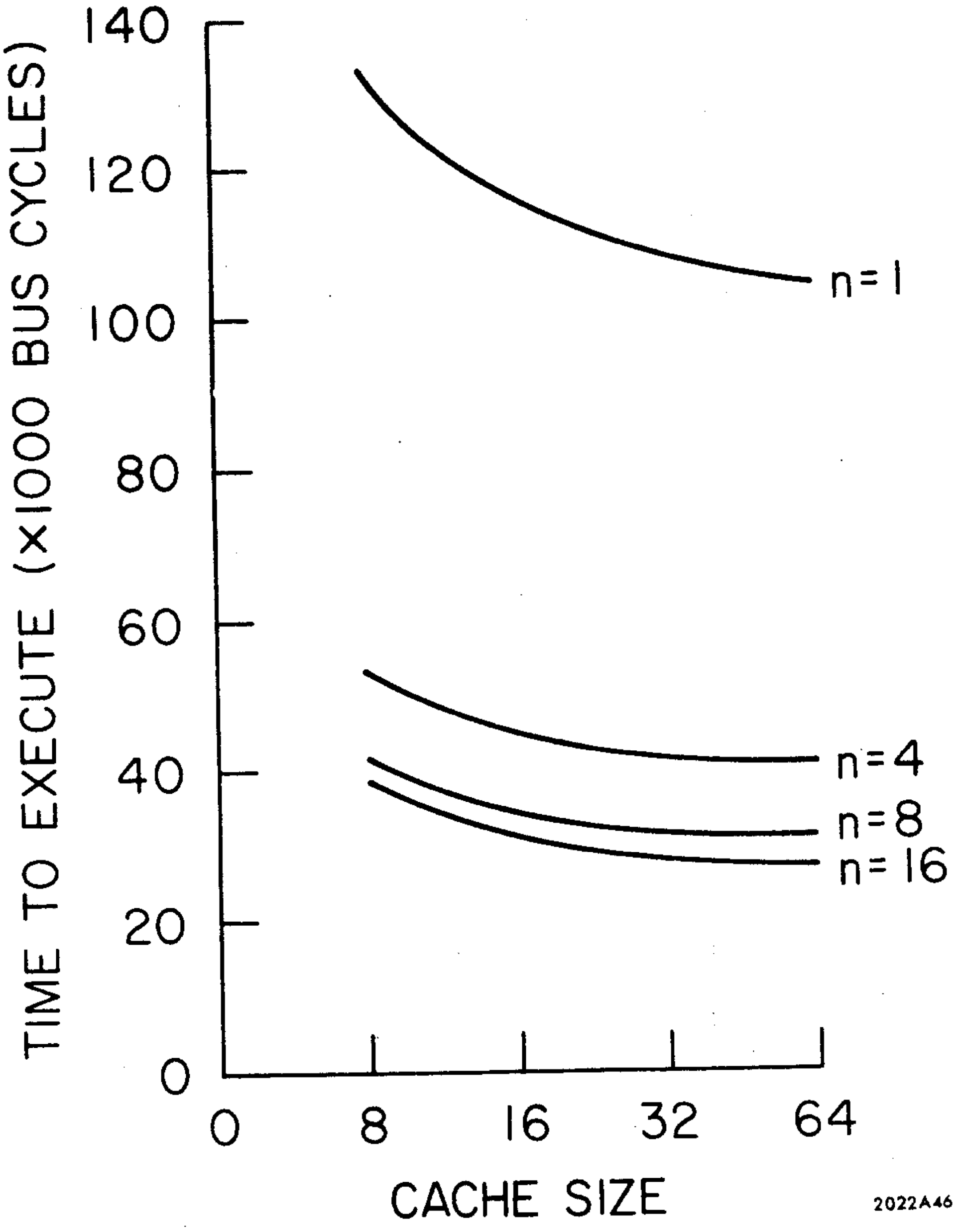


Figure 36a. Effect of Cache Size on Throughput of Sum-Squared Graph Program(5)

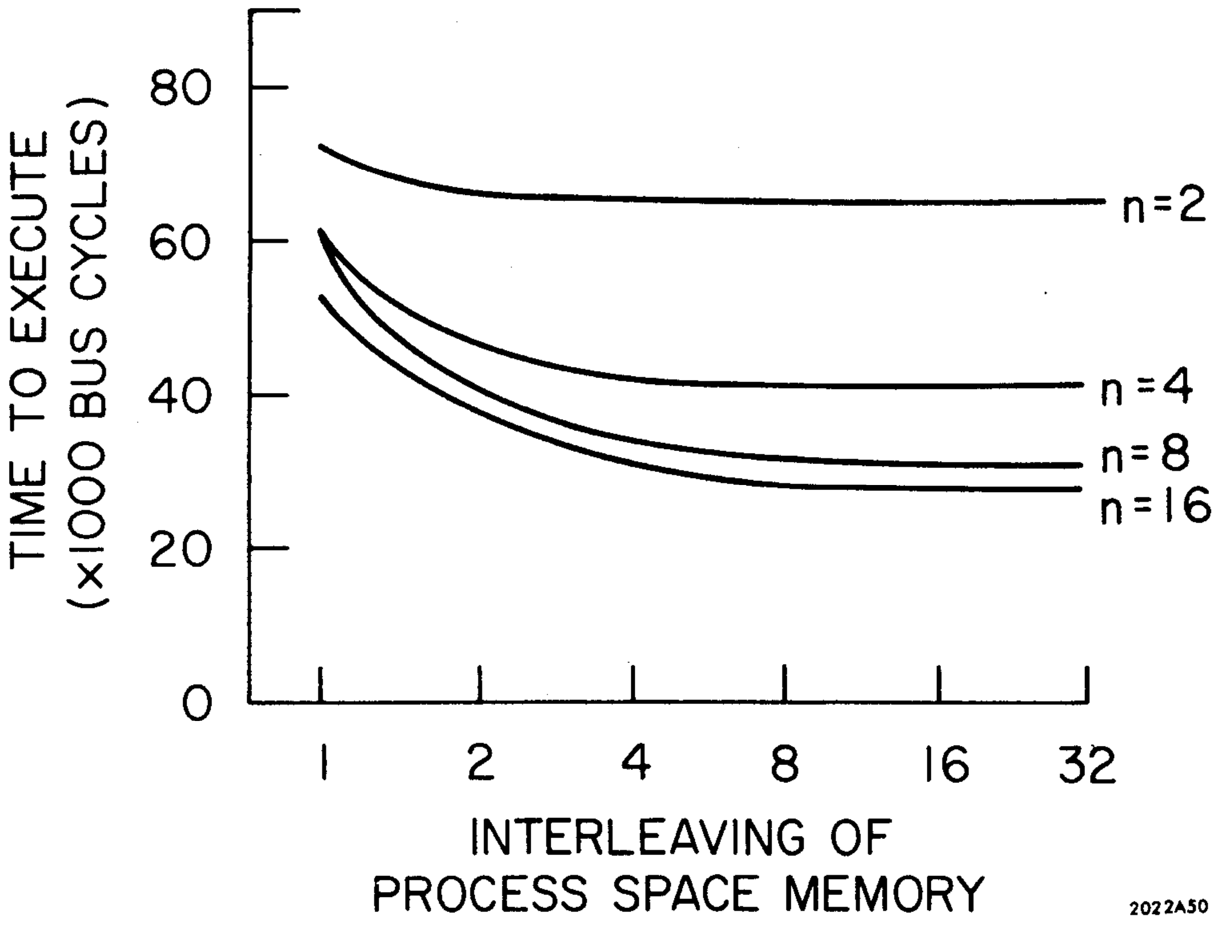


Figure 36b. Effect of Process Space Memory Interleaving on Throughput of Sum-Squared Graph Program(5)

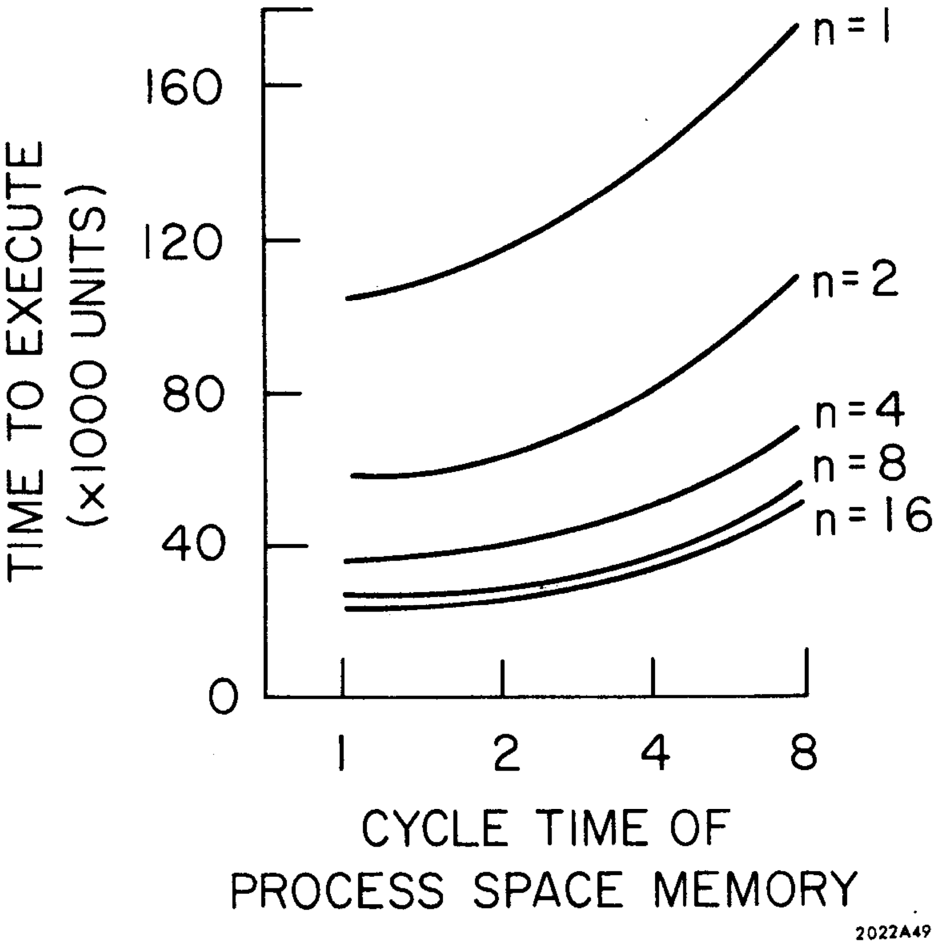


Figure 36c. Effect of Process Space Memory Cycle Time on Throughput of Sum-Squared Graph Program(5)

```
                  SYSTEM'S CONFIGURATION
                  ----------------------




CYCLE TIME OF MICROPROCESSOR= 1

NUMBER OF FUNCTIONAL UNITS= 0

NUMBER OF BUSES= 8

PSM READ/WRITE CYCLE TIME= 2
NUMBER OF PSM MODULES_INTERLEAVE=16
NUMBER OF INDEPENDENT PSM ACCESS PATHS=-1

MPM READ/WRITE CYCLE TIME= 2
NUMBER OF MPM MODULES_INTERLEAVE= 8
NUMBER OF INDEPENDENT MPM ACCESS PATHS=-1

MEM READ/WRITE CYCLE TIME= 2
NUMBER OF MEM MODULES_INTERLEAVE= 4
NUMBER OF INDEPENDENT MEM ACCESS PATHS= 1

CACHE READ/WRITE CYCLE TIME= 1
CACHE SIZE=32
LINE SIZE OF CACHE= 4
CACHE STORE THROUGH MODE=3
```

Table 5: PMS Configuration Used for Simulation Results

VI. Summary Comment and Conclusions

This thesis has described an architecture for a parallel microcomputer system that permits a systematic and flexible approach to the emulation of a wide variety of complex sequential and parallel intermediate machine languages in a dynamically varying Processor-Memory-Switch(PMS) environment. This architecture has been based on the view that complex emulators can be best structured in terms of a set of microprocessors that interact in a highly structured manner. Further, these highly structured interaction patterns are different for different types of emulators but for a particular emulator generally remain static. This view represents a modular, task oriented approach to managing the complexity of emulation.

These highly structured interaction patterns are dynamically defined through the concept of a virtual PMS environment. This concept embodies the capability for reconfiguring both the internal and the external environment of a microcomputer system: the varying of the number of internal working registers of each microprocessor; the varying of the structure of memory, e.g., its size and word length; and the varying of the number of microprocessors and functional units, and their interconnections and interaction patterns. This extra dimension of representational freedom provided by the concept of a virtual PMS environment allows:

1) The virtual state image of the microcomputer system, S(vm), to be structured so as to make the imbedding of the

state image of complex IML's, S(e), straightforward;

2) *The microinstructions to operate directly in the context* of an appropriate S(vm) so as to make the coding of an emulator compact and simple;

3) *The emulator to be coded so as to be independent of the* physical PMS environment but, at the same time, exploit physical resources when available.

In this way, the microcomputer architecture can be dynamically reconfigured so that it directly mirrors the structure of the IML to be emulated. A close mirroring between the structure of the microcomputer and that of the emulated machine is the key to efficient emulation.

A virtual PMS is implemented in the microcomputer architecture by adding a new global level of hardware control. By making the virtual PMS an integral part of the microcomputer architecture, the overhead in implementing highly structured parallel interaction patterns, where the parallel activity is of short duration, does not overwhelm the inherent parallelism of the interaction patterns. This new level of hardware control can be thought of as a simple, hardware operating system which controls the scheduling and interactions among microprocessors and functional units. A particular virtual PMS is dynamically defined by constructing an appropriate global control structure for the microcomputer system. An appropriate global control structure is constructed by dynamically modifying the syntax, i.e., the number of data elements and their relationships, of the control data structure (CDS) for this new global level of control. In a conventional computer or microcomputer system, the data structure for control contains a fixed set of data elements whose relationships are predefined. Thus, in a conventional system, control can be modified only by changing the value of data elements in the CDS, e.g., changing the program counter. The ability added here to modify the syntax of the data structure for control is the key to tailoring a virtual PMS environment for a particular emulated machine.

The CDS has been defined so as to allow the flexible structuring of a virtual PMS environment, while at the same time permitting the hardware algorithm for the mapping of virtual microprocessor activity to actual microprocessor activity to be straightforward. The CDS consists of an arbitrary number of microprocess state vectors (MSV); each MSV has a structure which has 13 components; different microprocess interaction patterns are defined by varying the number of state vectors and the values of their components which change the relationships among microprocesses. The components of the microprocess state vector can be broken into two overlapping classes: external-environment components and internal-environment components. Each of these classes can be further subdivided into control-environment components and data-environment components. The external control-environment components define the set of microprocesses that a microprocess can directly communicate with. The external data-environment components define how other microprocesses can transfer data to a microprocess. The internal control-environment components define the local CDS for the sequencing of microinstructions of a microprocess. The internal data environment components define the internal working registers of the microprocess. Thus, by dynamically altering the number of MSV's and their components, a virtual PMS can be tailored to a particular emulated machine.

There are two major tasks in emulation: performing address arithmetic computations (e.g., computing the address of an operand, decoding of an instruction format) and sequencing among the different tasks (e.g., control, decoding, semantic routine) required in performing an emulation. Corresponding to these two tasks areas, there are respectively two general classes of microinstructions in the microcomputer. One class, called the Integer Function Language (IFL), deals with internal registers of the microprocessor, and are like conventional vertical microinstructions. The other class, called the Structure Building Language (SBL), deals with the external environment of the microprocessor by modifying the CDS . The major emphasis in this thesis has been on the SBL because the flexibility of the control structure of the microcomputer is crucial to the effective emulation of sophisticated IML's. This flexibility of control structure is the major feature lacking in existing microcomputer architectures.

The SBL microinstructions are not oriented toward specifying any particular method of microprocess(or) interaction patterns, but rather are building blocks by which different interaction patterns can be defined. The key to the design of the SBL is to imbed, in a parameterized way, in a small number of microinstruction types the essential aspects of a wide variety of interaction patterns. In addition, SBL microinstructions are designed so as to provide information to the hardware mapping algorithm which allows the mapping algorithm to take advantage of similarities between the structure of the virtual PMS environment and that of the actual PMS environment.

The SBL, which consists of eight microinstruction types listed in Table 1, has two functions: a syntactic function and a semantic function. The syntactic function involves the dynamic construction of the CDS while the semantic function involves the dynamic invocation of microprocess interaction patterns defined in the CDS. In essence, the syntactic microinstructions dynamically define static, time-independent interrelationships among microprocesses. The semantic microinstructions use these syntactic interrelationships among microprocesses as a convenient representational framework within which to define dynamic, time-dependent interrelationships among microprocesses. The separation between the definition of interaction patterns and their invocation is possible because the execution of a microprocess is factored into three discrete, separable phases: a binding phase, an expansion phase, and an activation phase. The generation of a CDS caused by the binding and expansion phases can thus be separated from the sequencing of a CDS caused by the activation phase. This separation is extremely important because once the overhead cost has been incurred for defining the CDS whose structure generally remains static during an emulation, there is little overhead cost for each dynamic interaction pattern invoked.

A key aspect of microprocess interaction patterns is specifying at what time points in the activity life of a microprocess certain types of communications can be received. This aspect of microprocess interaction is accomplished through the concept of agreeable communication states: the semantics of SBL microinstructions are defined so that communication between two microprocesses is only consummated when the execution-state and type of communication(activation-type) are agreeable for communication. The set of agreeable states is designed so that a microprocess can 1) sequentially accept and process multiple communications, 2) selectively accept only certain types of communications, and 3) asynchronously accept requests for communication.

The feasibility of this microcomputer architecture has been demonstrated by examining its representational capabilities, its hardware organization, and its dynamic execution characteristics.

The representational capabilities of this architecture have been examined through the microprogramming of an emulator for a sophisticated parallel machine language, Adams' Graph Machine Language(AGML). The emulator of this machine language has demonstrated the versatility and usefulness of the SBL and the concept of a dynamically restructurable CDS in the following ways:

1) It has shown how an S(vm) can be constructed so as to make the embedding of the state image of a complex IML,

S(AGML), straightforward. In particular, it has indicated how a CDS can be tailored so that it directly mirrors the distributed control structure of AGML.

2) It has shown that an emulator can be compactly and simply coded when the microinstruction directly operates in the context of the appropriate S(vm). The microprogram memory required for the AGML emulator microprogram, including the storage for constants, is less than 600 microinstruction words.

3) It has shown how a CDS can be dynamically structured so as to easily represent a wide variety of different types of control structures, i.e., distributed control, semaphore processes, message queuing, broadcast control, etc. Further, it has indicated how these different types of control structures can be integrated together in a single CDS.

4) It has shown how a modular task approach to design of an emulator can be implemented naturally within the framework of a restructurable CDS.

The examination of a harware organization for this architecture has indicated that a coherent and simple logical design can be constructed which allows parallel activity on the virtual PMS level to be mapped correctly and without significant overhead onto parallel activity in the microcomputer system. The main design techniques used were

1) An Interprocessor Bus and its associated control which works like a common data bus so as to map directly, whenever possible, microprocess interaction patterns into microprocessor interaction patterns.

2) A built-in FIFO queue mechanism in the Virtual Interaction Controller for ordering access to a locked-out MSV so as to guarantee that no resource deadlock will be introduced in the mapping of virtual activity to physical activity.

3) A memory cache per microprocessor which allows the internal registers to be configured so as to match the particular microprocess being executed and to reduce the

memory interference among microprocessors and the time to context switch.

The dynamic execution characteristic of this architecture have been studied through the use of a detailed simulator of the logical hardware organization. This simulator has been used to quantitatively verify that the graph machine emulator is correct, and that parallel activity on the virtual PMS can be mapped without significant overhead onto the physical PMS. In particular, a major part of the computational activity of the AGML emulator can be performed in a parallel way as a function of 1/n: where sufficient parallel activity exists, the addition of microprocessors to the PMS configuration will reduce in a linear way the time it takes to execute the computation. The simulation results have also indicated that the logical hardware design, with the appropriate PMS configuration, can efficiently handle sustained parallel activity, involving highly structured interaction patterns, of greater than sixteen microprocessors. In addition, the phenomenon of the 'control working set' has been experimentally verified: the hardware overhead required to map virtual PMS activity onto physical PMS activity significantly decreases when the PMS configuration contains more than a certain number of microprocessors.

In summary, this thesis has demonstrated that a new type of microcomputer architecture, employing the concept of dynamic control structures, permits the effective (ease of representation, code density, and low hardware overhead) emulation of complex problem oriented computers whose architecture departs from a classical von Neumann architecture. This thesis has also verified that a modular task oriented approach to managing the complexity of emulation is feasible at both the hardware and software level.

REFERENCES

ABR70 P.S. Abrams, "An Apl Machine", Report SLAC-114, Stanford Linear Accelerator Center, Stanford University, February 1970.

ADA68 D.A. Adams, "A Computational Model with Data Flow Sequencing", (Ph.D Thesis), Report No. CS117, Computer Science Department, Stanford University, December 1968.

AND62 J.P. Anderson et al, "D825--A Multiple Computer System for Command and Control", in 1962 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 22. Washington, D.C.: Spartan Books, 1962, pp. 86-96.

AND67 D.W. Anderson et al, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling", IBM Systems Journal, Vol. 11, No. 3, January 1967, pp. 8-23.

BEL70 C.G. Bell and A. Newell, "The PMS and ISP Descriptive System for Computer Structure", in 1970 Spring Joint Comput. Conf.,. AFIPS Conf. Proc., Vol. 22. Montvale, N.J.: AFIPS Press, 1970, pp. 657-676.

BEL70A C.G. Bell et. al, "A New Architecture for Mini-Computer - The DEC PDP-11", in 1970 Spring Joint Comput. Conf., AFIPS Conf. Proc., Pt. II, Vol. 36. Montvale, N.Y.: AFIPS Press, 1970, pp. 657-676.

BEL71 C.G. Bell and P. Freeman, "C.ai: A Computing Environment for A.I. Research", Computer Science Department, Carnegie-Mellon University, May 1971.

BEL72 C.G. Bell et al, "Effect of Technology on Near Term Computer Structures", Computer, March/April 1972, pp. 29-38.

BIN68 H.W. Bingham, E.W. Reigel, and D.A. Fisher, "Control Mechanisms for Parallelism in Programs", Technical Report TR-68-3, Burroughs Corp., Paoli, Pennsylvania, October 1968.

CON68 M.E. Conway, "A Multi-processor System Design" in 1963 Fall Joint Computer Conf., AFIPS Conf. Proc., Vol. 24. Baltimore: Spartan, 1968, pp. 139-146.

COO70 R.W. Cook and M.J. Flynn, "System Design of a Dynamic Microprocessor", IEEE Trans. Comput., Vol. C-19, March 1970, pp. 213-222.

DAH68 O. Dahl, B. Myhrhang, and K. Nygaard, "Simula 67, Common Base Language", Norwegian Computing Center, Oslo, Norway, May 1968.

DEN66 J.B. Dennis and E.C. Van Horn, "Programming Semantics for Multiprogrammed Computation", Commun. Ass. Comput. Mach., Vol. 9, March 1966, pp. 143-155.

DEN68 P.J. Denning, "The Working Set Model for Program Behavior", Commun. Ass. Comput. Mach., Vol. 11, May 1968, pp. 323-333.

DEN71 S.F. Dennis and M.C. Smith, "LSI Implications for Future Design and Architecture", RC3598, IBM Research Lab., Yorktown Heights., N.Y., November 1971.

DER69 S. Derenzo, "MINF68: A General Minizing Routine", Group A-Programming Note no. P-190, Univ. of Calif., Lawrence Radiation Laboratory, Berekely, Calif., July 1969.

DIJ65 E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", Commun. Ass. Comput. Mach., Vol. 8, pp. 569-570.

ERS71 A.P. Ershov, "Parallel Programming", Artificial Intelligence Laboratory, Report No. AIM-146, Computer Science Department, Stanford University, July 1971.

FLY71 M.J. Flynn and R.F. Rosin, "Microprogramming: An Introduction and a Viewpoint", IEEE Trans. Comput., Vol. C-20, July 1971, pp. 727-731.

GIB67 D.H. Gibson, "Considerations in Block-oriented Systems Design", In 1967 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol. 30. Washington, D.C.: Thompson, 1967, pp. 75-80.

GRA70 W.R. Graham, "The Parallel and Pipeline Computers", Datamation 16,4, April 1970.

HAU68 E.A. Hauch and B.A. Dent, "Burroughs B6500/7500 Stack Mechanism", in 1968 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol. 32. Washington, D.C.:Thompson, 1968, pp. 245-252.

HOR69 J.J. Horning and B. Randell, "Structuring Complex Processes", RC2459, IBM Research Lab., Yorktown Heights, N.Y., 1969.

HUS70 S. Husson, "Microprogramming, Principles and Practices", Prentice-Hall, Inc., Publisher, 1970.

IBM66 "System 360 Model 40, 2040 Processing Unit", in IBM Field Engineering Diagrams Manual, Doc. 0223-2842, 1966.

KAR66 R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", SIAM J. Appl. Meth. 14, November 1966.

KNU66 D.E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control" (Letter to Editor), Commun. Ass. Comput. Mach., Vol. 9, May 1966, pp. 321-322.

KNU68 D.E. Knuth, "The Art of Computer Programming: Fundamental Algorithms (Vol-I)", Addison-Wesley Series in Computer Science and Information Processing, 1968, pages 406-455.

LAS68 S. Lass, "A Fourth Generation Computer Organization", in 1968 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol. 32. Washington, D.C.: Thompson, 1968, pp 435-442.

LAM70 B.W. Lampson, "The BCC-500 System", Berkeley Computer Corporation, Berkeley, Calif., 1970.

LAM69 B.W. Lampson, "Dynamic Protection Structures", in 1969 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 35. Montvale, N.J.: AFIPS Press, 1969, pp 27-38.

LAM68 B.W. Lampson, "A Scheduling Philosophy of Multiprocessing Systems", Commun. Ass. Comput. Mach., Vol. 11, May 1968, pp. 347-359.

LAW71 H.W. Lawson, Jr., and B.K. Smith, "Functional Characteristics of a Multilingual Processor", IEEE Trans. Comput., Vol. C-20., July 1971, pp. 732-743.

LEH66 M. Lehman, "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", Proc. of the IEEE, Vol. 34, December 1966, pp. 1889-1901.

LES68 V.R. Lesser, "A Multilevel Computer Organization Designed to Separate Data-accessing from the Computation", Report No. CS90, Computer Science Department, Stanford University, 1968

LES70 V.R. Lesser, "Direct Emulation of Control Structues by a Parallel Microcomputer", Report SLAC-127, Stanford Linear Accelerator Center, Stanford University, October 1970.

LES71 V.R. Lesser, "An Introduction to the Direct Emulation of Control Structures by a Parallel Microcomputer", IEEE Tran. Comput., Vol. C-20, July 1971, pp. 751-763.

LEV72 J.V. Levy, "Computing with Multiple Microprocessors", (Ph.D. thesis) Computer Science Department, Stanford University, July 1972.

LOR72 H. Lorin, "Parallelism in Hardware and Software: Real and Apparent Concurrency", Prentice-Hall Series in Automatic Computation, 1972.

MCK67 W. McKeeman, "Language Directed Computer Design", in 1967 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 31. Washington, D.C.:Thompson, 1967, pp. 413-418.

MEL65 A.J. Melbourne and J.M. Pugmire, "A Small Computer for the Direct Processing of FORTRAN Statements", Comput. J., Vol. 8, April 1965, pp. 24-27.

MIL70 J.S. Miller et al, "Multiprocessor Computer Systems Study, Final Report", NASA-CR-108654, March 1970.

NAR 67 R. Narasimham, "Programming Languages and Computer: A Unified Metatheory", Advances in Computers-1967, Vol. 8, Academic Press, 1967, pp. 189-244.

NEL70 E. Nelson, "Graph Program Simulation", Report No. CS185, Department of Computer Science, Stanford University, October 1970.

NEL72 E. Nelson, "Free Running and Resource Limited Graph Programs", (Ph.D. Thesis), Computer Science Department, Stanford University, September 1972.

RIC71 R. Rice and W.R. Smith, "SYMBOL: A Major Departure from Classic Software Dominated von Neumann Computing Systems", in 1971 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol. 38, Montvale, N.J.: AFIPS Press, 1971, pp. 601-616.

RID72 W.E. Riddle, "The Modeling and Analysis of Supervisory Systems", STAN-CS-72-271, Computer Science Department., Stanford University, March 1972.

ROD67 J.E. Rodriquez, "A Graph Model for Parallel Computations", (Ph.D. Thesis), Mass. Institute of Technology, September 1967.

ROS69 R.F. Rosin, "Contemporary Concepts of Microprogramming and Emulation", Comput. Surveys, Vol. 1, December 1969, pp. 197-212.

ROS71 R. Rosin, G.Frieder, and R. Eckhouse, "An Environment for Research in Microprogramming and Emulation", Dept. Report S-71-MU, Dept. of Computer Science, SUNY at Buffalo, May 1971.

ROSD69 J.L. Rosenfeld, "A Case Study in Programming for Parallel-Processors", Commun. Ass. Comput. Mach., Vol. 12, December 1969.

APPENDIX A

The Integer Function Language (IFL) Class of Microinstructions

The microcomputer contains, as previously discussed, two general classes of microinstruction: SBL and IFL microinstructions. A microprogram consists of an arbitrary inter-mixture of these two classes of microinstructions. In fact, the SBL and IFL microinstructions have the same format (see Figure 11 and Figure 37). The distinction between these two classes of microinstructions stems from a desire to clearly distinguish microperations which directly manipulate the external environment of the microprocess from those that manipulate the internal environment of the microprocess. This distinction leads to more efficient utilization of a microinstruction word and to microprograms which are more compact, more modular, and more easily understood and debugged. Thus, the SBL microinstructions, which deal with external environment, have a minimal set of internal sequencing rules (i.e., either execute the next microinstruction or terminate sequencing), and can only modify the internal environment of microprocessors by leaving a descriptor on the top of the value stack. On the other hand, the IFL microinstructions, which deal with internal environment, can modify the external environment of the microprocess only indirectly by storing data in the global data environment of the microprocess.

IFL microinstructions are specifically designed for the address arithmetic computations required in emulation: computing the effective address of an operand, decoding an instruction format, and emulating the action of an arithmetic unit, i.e., a floating-point adder. In order to accomplish these functions, the IFL has been designed to:

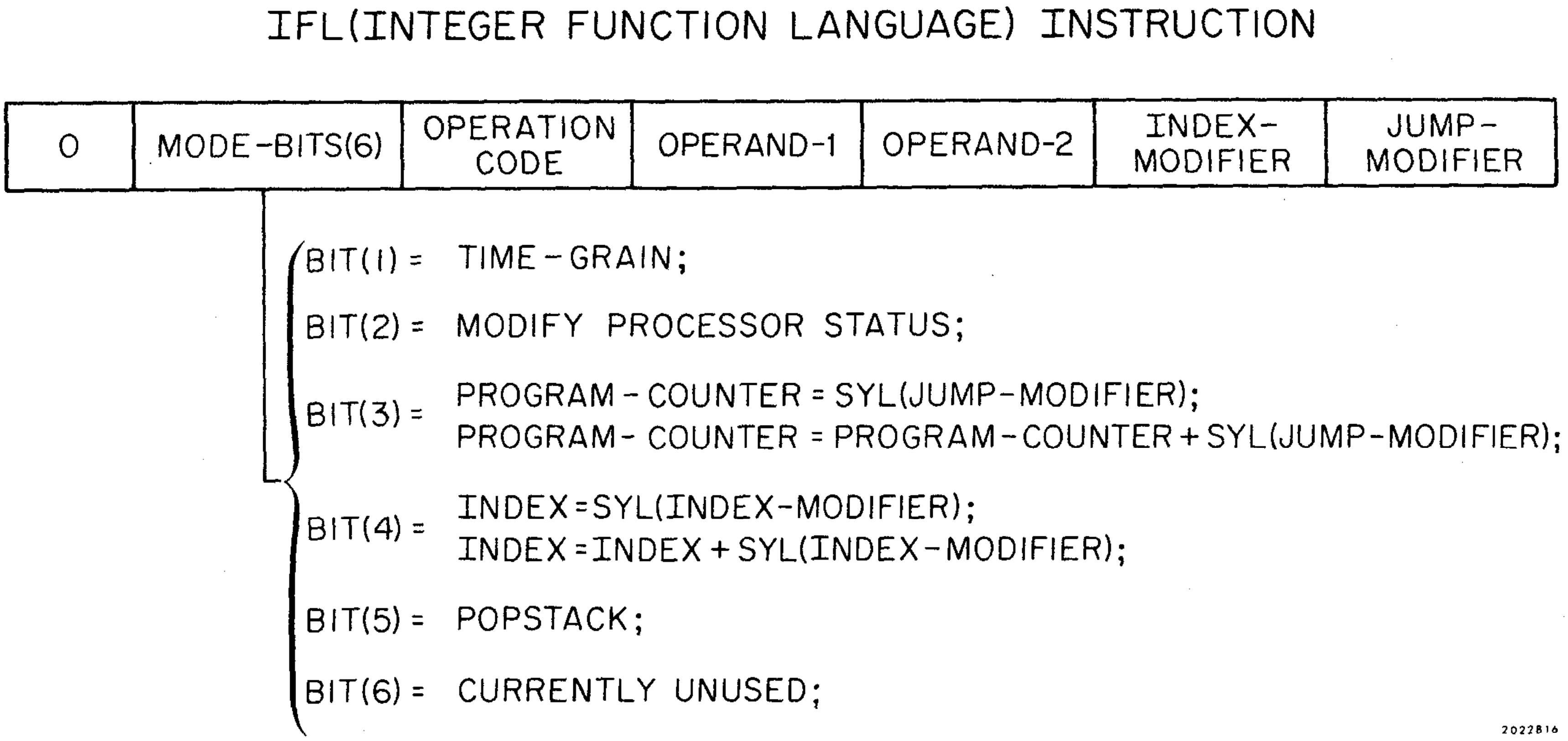
1) extract or store an arbitrary field of a 32 bit register in a single microinstruction.

2) directly perform arithmetic operations on aligned or unaligned fields of from 1 to 32 bits in length (same technique as used in MLP-900 (LAW71)).

3) test for an arbitrary condition and then jump appropriately in a single microinstruction.

4) evaluate indirectly fields of a microinstruction: an inline function call mechanism in which a microinstruction may invoke a microprogram in order to generate the value of a field.

The format of an IFL microinstruction (see Figure 37) resembles a vertical microinstruction format except that each field, instead of directly specifying a register or arithmetic operation, contains a syllable (See Figure 11 and B6700 (HAU68)). These five syllables fields, when evaluated, define five values which are used to specify an operation code, two operands, an index register modifier, and a program counter modifier*. This syllable approach introduces a level of indirection in the execution of a microinstruction which allows for very compact and modular code. In addition to the five syllables, there are five mode bits contained in the microinstruction



IFL(INTEGER FUNCTION LANGUAGE) INSTRUCTION

0	MODE-BITS(6)	OPERATION CODE	OPERAND-1	OPERAND-2	INDEX-MODIFIER	JUMP-MODIFIER

BIT(1) = TIME-GRAIN;

BIT(2) = MODIFY PROCESSOR STATUS;

BIT(3) = PROGRAM-COUNTER = SYL(JUMP-MODIFIER);
PROGRAM-COUNTER = PROGRAM-COUNTER + SYL(JUMP-MODIFIER);

BIT(4) = INDEX = SYL(INDEX-MODIFIER);
INDEX = INDEX + SYL(INDEX-MODIFIER);

BIT(5) = POPSTACK;

BIT(6) = CURRENTLY UNUSED;

Figure 37. Format of IFL Microinstruction Word

word. These mode bits specify whether the arithmetic operation will result in a modification to the "processor-status", whether execution will be suspended after the completion of this microinstruction if the process-state is execute-single-cycle, whether the value of the jump-modifier syllable will be added to the program counter (i.e., a relative jump) or used to replace its contents (i.e., an absolute jump), whether the value of index-modifier will be added to the index register or replace its contents, and whether the value stack will be popped at the completion of the microinstruction.

The execution of an IFL microinstruction is broken up into three steps:

1) evaluate all non-deferred syllables, and place their values on the program counter stack. If the index modifier syllable is not deferred, then the index register is immediately modified. The defer bit of index modifier syllable allows for either post- or pre-indexing operations. In addition, the defer bit of the jump modifier permits the new program counter address to be computed before or after the execution of an arithmetic operation.

2) evaluate all deferred syllables which involve a function call. The result of each function call is placed on the program counter stack.

3) perform the arithmetic operation specified by the operation-code syllable, and modify the index register and the program counter.

---

*The current program counter and index register are conceptually the top two registers of program counter stack; however, for implementation efficiency these two registers are held directly in internal registers in the microcomputer so as to avoid any cache accesses. The index register is used as a scratch pad register and is pushed down, together with the program counter, on the program counter stack when a function call is invoked. The index register is used also to hold a mask operand for masking operations.

The first two steps of execution are skipped if all syllables are deferred and there are no function calls. The results of an arithmetic operation is placed on the top of the value stack. If the program-counter modifier specifies a null value, then current program counter and index register are replaced by the top two elements of the program counter stack, and top of the value stack is transferred to the top of the program counter stack. Through this mechanism, the results of an inline function call is returned to the invoking microinstruction so that it may complete its execution.

The format of a syllable and its evaluation procedure are specified in Figure 11. A syllable is eleven bits long and contains three subfields: DEFER, DESCRIPTOR and ICONSTANT. The DEFER subfield is used to specify when a syllable will be evaluated, e.g., pre- or post-indexing. The DESCRIPTOR subfield indicates which one of the four possible evaluation modes is to be used to compute the value of the syllable. The ICONSTANT subfield specifies an 8 bit 2's complement integer which is used as an operand to the syllable evaluation procedure. These are four possible modes of evaluating a syllable:

1) Immediate operand -- return ICONSTANT subfield as value of syllable

2) Inline functional call -- use ICONSTANT plus the current program counter address to specify the starting address of a microprogram function. This microprogram function is invoked and the last result produced by this microprogram becomes the value of the syllable.

3) Register operand -- return the value of operand located in the local or global data environment of the microprocess. The index into the data array is specified by ICONSTANT and its sign indicate whether the local or global array will be used.

4) Long Constant or Internal State Register -- based on the sign of ICONSTANT return either a 32 bit constant stored in the Microprogram Memory or an internal state register (i.e., the program counter, the index register, the top two elements of the value stack, or any field or the Microprocess State Vector).

This four mode evaluation procedure provides a flexible but, at the same time, concise way of computing a syllable value.

The operation-codes fall into two classes: integer arithmetic operations and descriptor based operations. The integer arithmetic class of operations contains the conventional arithmetic, logical and shifting operations, whereas the descriptor based class of operations are used to access and store information from an array of data registers which are specified through a descriptor. The power of this set of operation codes for address arithmetic and bit extraction and manipulation, comes from the ability to augment these basic operations with a mask operand which is implicitly specified to reside in the index register. Table 6 contains a list of these operation-codes and their semantics.

The microassembler syntax for the IFL microinstructions is contained in appendix B. Appendix C contains numerous examples of how the IFL is used for address arithmetic computations, and how the IFL leads to compact microprograms.

TABLE 6. IFL Operation-Codes

<table>
<tr><th>Operation Code</th><th>Semantics</th><th>Comments</th></tr>
<tr><td></td><td>Integer Arithmetic Operations</td><td></td></tr>
<tr><td>SFO (Select First Operand)</td><td>MRESULT*=OPR(1)</td><td>These two opcodes are represented by values 1 & 2. They are used in conjunction with an opcode syllable which evaluates to a condition code which is either 1 or 2. This permits the loading of operand based on the value of the condition code.</td></tr>
<tr><td>SSO (Select Second Operand)</td><td>MRESULT=OPR(2)</td><td></td></tr>
<tr><td>ADD</td><td>MRESULT=(OPR(1)&MASK)+(OPR(2)& ¬ MASK)</td><td>These three opcodes permit arithmetic operations to be performed, in place, on unaligned register fields.</td></tr>
<tr><td>SUBTRACT</td><td>MRESULT=(OPR(1)& MASK)+((-OPR(2))& ¬ MASK)</td><td></td></tr>
<tr><td>MULTIPLY</td><td>MRESULT=(OPR(1)& MASK)x(OPR(2)& MASK)</td><td></td></tr>
<tr><td>SHIFT</td><td>RESULT=if OPR(2)≥ 0 then
O[1:OPR(2)]||(OPR(1)& MASK)[1:32-OPR(2)]
else
(OPR(1)& MASK)[1-OPR(2):32]||O[1:-OPR(2)]</td><td>Permits extraction and aligning of a field in a single microinstruction.</td></tr>
<tr><td>SHIFT_MASK</td><td>MRESULT= if OPR(2)≥ 0 then
O[1:OPR(2)]||OPR(1)[1:32-OPR(2)] else
OPR(1)[1-OPR(2):32]|| O[1:-OPR(2)]</td><td>Permits extraction and aligning of a field using an immediate constant mask field since masking done after aligning.</td></tr>
<tr><td>LOGICAL_AND</td><td>MRESULT=OPR(1)& OPR(2)</td><td></td></tr>
<tr><td>COMPARE</td><td>Condition Code produced by SUBTRACT operation</td><td></td></tr>
<tr><td>LCOMPARE</td><td>Condition code produced by LOGIC/L-AND operation.</td><td></td></tr>
<tr><td></td><td>Descriptor Operations</td><td></td></tr>
<tr><td>INDEX</td><td>RESULT=Pointer to ELEMENT(OPR(1),OPR(2))**</td><td></td></tr>
<tr><td>ACCESS</td><td>MRESULT=ELEMENT(OPR(1),OPR(2))</td><td></td></tr>
<tr><td>STORE</td><td>ELEMENT(OPR(1),OPR(2))=
(ELEMENT(OPR(1),OPR(2))& ¬ MASK) V
((TOP OF VALUE STACK)& MASK)</td><td>Permits in place modification of a register field - usually combined with an arithmetic operation to create two microinstruction sequence for updating a register field.</td></tr>
<tr><td>STORE_INDEX</td><td>ELEMENT(OPR(1),OPR(2))=INDEX REGISTER</td><td>Permits storing of syllable value into register in single microinstruction. This is accomplished by performing index modification operation before store operation.</td></tr>
<tr><td>LOAD_CONSTANT</td><td>MRESULT=first or second constant at location, OPR(1)+(OPR(2)-1)/2, in the Microprogram Memory dependent upon whether OPR(2) is odd or even.</td><td>Permits storing of constants in M.MPM rather than just M.PSM. This saves space by avoiding duplication of storage for constants. This is particularly important in multiprocessor configuration since there may be many microprocesses using the same microprogram.</td></tr>
</table>

* MRESULT indicates that if result is not a descriptor, then result will be masked. If mask option is not invoked, then mask is all one bits.

**ELEMENT(OPR(1), OPR(2)) specifies OPR(2) element of data array specified by descriptor OPR(1).

APPENDIX B

SYNTAX OF MICROASSEMBLER

```
<MICRO_PROGRAM>      :=   BEGIN <STMTS> END

<STMTS>              :=   <LSTMT> ;
                     :=   <LSTMT> ; <STMTS>

<LSTMT>              :=   <LABEL> <STMT>

<STMT>               :=   <IFL_STMT> ;
                     :=   <SBL_STMT> ;
                     :=   <CONS_STMT> ;

<SBL_STMT>           :=   <MEM_STMT>
                     :=   <FCP_STMT>
                     :=   <SEL_STMT>
                     :=   <ASP_STMT>
                     :=   <MSC_STMT>
                     :=   <GEN_PMSV>
                     :=   <GEN_EPSV>
                     :=   <GEN_REG>


<MEM_STMT>           :=   <ACCESS> ELEMENT ( <SYL> ) WITH FORMAT =
                          <SYL> AND LENGTH = <SYL> <DIRECTION>
                          MEMORY ARRAY ( DESCRIPTOR = <SYL> ,
                          OFFSET = <SYL> )

<ACCESS>             :=   READ
                     :=   STORE
                     :=   READ/STORE

<DIRECTION>          :=   FROM
                     :=   INTO
                     :=   FROM/INTO


<FCP_STMT>           :=   ACTIVATE FUNCTIONAL_UNIT ( <SYL> ) WITH
                          CONTROL_INFORMATION = <SYL> USING <SYL>
                          INPUT_GENERATORS <ACTIVATE> AND STORE STATUS
                          IN <SYL>
                     :=   ACTIVATE FUNCTIONAL_UNIT ( <SYL> ) WITH
                          CONTROL_INFORMATION = <SYL> USING <SYL>
                          INPUT_GENERATORS <ACTIVATE>
```

```
<SEL_STMT>              :=   <ACTIVATE> <SONS> WITH INPUT = <SYL>
                        :=   <ACTIVATE> <SONS> WITH INPUT = <SYL>
                             THEN WAIT FOR <SYL> TO SIGNAL RETURN

<SONS>                  :=   SON ( <SYL> )
                        :=   SONS ( <SYL> TO <SYL> )

<ASP_STMT>              :=   <ACTIVATE> NODE ( <SYL> ) WITH <ASP_PARAMS>

<ASP_PARAMS>            :=   <ASP_PARAM>
                        :=   <ASP_PARAM> , <ASP_PARAMS>

<ASP_PARAM>             :=   INPUT = <SYL>
                        :=   RETURN_ADDRESS = <SYL>
                        :=   EPSV = <SYL>

<MSC_STMT>              :=   INVOKE PROGRAM ( <SYL> ) WITH <MSC_PARMS>

<MSC_PARAMS>            :=   <MSC_PARAM>
                        :=   <MSC_PARAM> , <MSC_PARAMS>

<MSC_PARAM>             :=   STACK_TOP = <SYL>
                        :=   INDEX = <SYL>
                        :=   PROGRAM_STATUS = <SYL>
                        :=   INITALIZE_ROUTINE = <ADDRESS>


<GEN_PMSV>              :=   S = P ( DESCRIPTOR OF <GEN_PMSV1> )
                        :=   S = P ( <GEN_PMSV1> )
                        :=   <REBUILD> PROCESS WHOSE <GEN_PMSV1>
                             AND <CLOCK_PROCESS>

<REBUILD>               :=   STATIC
                        :=   DYNAMIC

<GEN_PMSV1>             :=   SUBSTRUCTURE CONTAINS <SYL> SONS
                             WITH <STATE_PARAMETERS>

<STATE_PARAMETERS>      :=   <STATE_PARAM>
                        :=   <STATE_PARAM> , <STATE_PARAMETERS>

<STATE_PARAM>           :=   PROGRAM = <SYL>
                        :=   PORT = <SYL>
                        :=   LOCAL_DATA = <SYL>
                        :=   EPSV = <SYL>

<CLOCK_PROCESS>         :=   CLOCKING PROCESS = <SBL_STMT>

<GEN_EPSV>              :=   S  = P ( EPSV WITH <EPSV_LIST> )
                        :=   RETURN ( P ( EPSV WITH <EPSV_LIST> ) )
```

```
<EPSV_LIST>           :=   <EPSV_PAR> = <SYL>
                      :=   <EPSV_PAR> = <SYL> , <EPSV_LIST>

<EPSV_PAR>            :=   GLOBAL_DATA
                      :=   VSTACK
                      :=   PSTACK
                      :=   GLOBAL_PROCESS
                      :=   EXTERNAL_ENV

<GEN_REG>             :=   S = <GEN_REG1>
                      :=   RETURN ( <GEN_REG1> )

<GEN_REG1>            :=   P ( DESCRIPTOR OF <GEN_REG2> )
                      :=   P ( <GEN_REG2> )

<GEN_REG2>            :=   <DESC_TYPE> DEFINED FROM (
                           DESCRIPTOR = <SYL> , OFFSET =
                           <SYL> ) WITH <DESC_LIST>
                      :=   <DESC_TYPE> WITH <DESC_LIST>

<DESC_TYPE>           :=   REGISTER_BLOCK
                      :=   STACK
                      :=   IO_BLOCK
                      :=   MEMORY_ARRAY

<DESC_LIST>           :=   <DESC> = <SYL>
                      :=   <DESC> = <SYL> , <DESC_LIST>
                      :=   ACCESS_CONTROL = <AC_MODE>
                      :=   ACCESS_CONTROL = <AC_MODE> , <DESC_LIST>

<DESC>                :=   DIMENSION
                      :=   WORD_LENGTH
                      :=   INITIAL_POSITION

<AC_MODE>             :=   LOCAL
                      :=   GLOBAL
                      :=   COROUTINE

<ACTIVATE>            :=   <TYPE_OF_ACTIVATION>
                      :=   <TYPE_OF_ACTIVATION> ( <ACT_MOD> )

<TYPE_OF_ACTIVATION>  :=   EXPAND
                      :=   EXECUTE
                      :=   EXECUTE_SINGLE_CYCLE
                      :=   SUSPEND
                      :=   TERMINATE
                      :=   RETRIEVE
                      :=   WAKEUP
                      :=   NULL_ACTIVATE
```

```
<ACT_MOD>                :=    <MODA>
                         :=    <MODA> , <ACT_MOD>

<MODA>                   :=    REFERENCE
                         :=    VALUE
                         :=    RETURN
                         :=    NO RETURN
                         :=    CONTINUE
                         :=    WAIT RESPONSE
                         :=    TRANSFER_DATA
                         :=    STORE_DATA
                         :=    TRANSFER_STATUS
                         :=    STORE_STATUS
                         :=    NO TRANSFER
                         :=    BUSY WAIT
                         :=    NO BUSY WAIT
                         :=    PARALLEL
                         :=    ACT_CODE = <INTEGER>

<IFL_STMT>               :=    <SEQ_STMT>
                         :=    <RET_STMT>

<LABEL>                  :=    <SYMBOL> :

<RET_STMT>               :=    <INDEX_STMT> , RETURN ( <OPERATION> )
                         :=    RETURN ( <OPERATION> )

<SEQ_STMT>               :=    <FUNC_INDEX> , <JUMP_STMT>
                         :=    <JUMP_STMT>

<FUNC_INDEX>             :=    <FUNC_STMT> , <INDEX_STMT>
                         :=    <FUNC_STMT>
                         :=    <INDEX_STMT>

<FUNC_STMT>              :=    S = <OPERATION>
                         :=    S = ( <OPERATION> , <MOD_LIST> )
                         :=    <OPERATION>
                         :=    ( <OPERATION> , <MOD_LIST> )

<OPERATION>              :=    <OP_CODES> ( <SYL> , <SYL> )

<MOD_LIST>               :=    <MODIFIER>
                         :=    <MODIFIER> , <MOD_LIST>

<MODIFIER>               :=    POP_STACK
                         :=    MODIFY_STATUS

<INDEX_STMT>             :=    I = I + <SYL>
                         :=    I = <SYL>
                         :=    I = I - <INTEGER>
```

```
<JUMP_STMT>          :=   GO TO <ADDRESS>

<ADDRESS>            :=   * + <SYL>
                     :=   <SYL>
                     :=   * - <INTEGER>
                     :=   <SYMBOL>

<SYL>                :=   _ <SX>
                     :=   <SX>

<SX>                 :=   <INTEGER>
                     :=   - <INTEGER>
                     :=   <BIT_STRING>
                     :=   L ( <INTEGER> )
                     :=   G ( <INTEGER> )
                     :=   L ( I )
                     :=   G ( I )
                     :=   F ( * + <SYL> )
                     :=   F ( <SYMBOL>  )
                     :=   CONS( <SYMBOL> )
                     :=   CONS( * + <INTEGER> )
                     :=   S
                     :=   S ( POP )
                     :=   S ( - 1 )
                     :=   P ( <POINTERS> )
                     :=   C ( <CONDITIONS> )
                     :=   I
                     :=   - I
                     :=   I.BEG
                     :=   I.END
                     :=   RESULT
                     :=   PC
                     :=   PC1
                     :=   ACT_CODE

<POINTERS>           :=   NULL
                     :=   LOCAL_DATA
                     :=   LOCAL_PROCESS
                     :=   GLOBAL_DATA
                     :=   GLOBAL_PROCESS
                     :=   RETURN
                     :=   EPSV
                     :=   SELF
                     :=   VSTACK
                     :=   PSTACK
                     :=   EXTERNAL_ENV
                     :=   PORT
```

```
<CONDITIONS>            :=    =
                        :=    ¬ =
                        :=    >
                        :=    ¬>
                        :=    <
                        :=    ¬<
                        :=    I=0
                        :=    I¬=0
                        :=    <INTEGER>

<OPCODES>               :=    <OPC>
                        :=    MASK_ <OPC>

<OPC>                   :=    SFO
                        :=    SSO
                        :=    INDEX
                        :=    ACCESS
                        :=    STORE
                        :=    STORE_INDEX
                        :=    SHIFT
                        :=    SHIFT_MASK
                        :=    LOAD_CONSTANT
                        :=    SET_STATE
                        :=    ADD
                        :=    SUBTRACT
                        :=    MULTIPLY
                        :=    SHIFT
                        :=    LOGICAL_AND
                        :=    COMPARE
                        :=    LCOMPARE

<BIT_STRING>            :=    ' <BITS> '

<BITS>                  :=    <BIT>
                        :=    <BIT> , <BITS>

<BIT>                   :=    0
                        :=    1
                        :=    0 ( <INTEGER> )
                        :=    1 ( <INTEGER> )

<CONS_STMT>             :=  CONSTANTS ( <CONSTANTS> )

<CONSTANTS>             :=  <LABEL> <CONSTANT>
                        :=   <LABEL> <CONSTANT> , <CONSTANTS>
                        :=  <CONSTANT>
                        :=  <CONSTANT> , <CONSTANTS>
```

```
<CONSTANT>            :=  <INTEGER>
                      :=  <SYMBOL>
                      :=  <BIT_STRING>
                      :=  * + <INTEGER>
                      :=  * - <INTEGER>
```

APPENDIX C

ADAM'S GRAPH MACHINE EMULATCR

```
/BEGIN/

/* GRAPH MACHINE(VERSION MAY 25,1972)   */

GRAPH_MACHINE:
  DEFINE STATIC PROCESS WHOSE SUBSTRUCTLRE CCNTAINS
   F(S3) SONS WITH
     PROGRAM=F(SONSGM),
     LOCAL_DATA=F(LOCAL_SONSGM),
     PORT=F(PORT_SONSGM),
     EPSV=F(EPSV_SONSGM)
                   AND
  CLOCKING PROCESS=INVOKE PROGRAM(GRAPH_CLOCKER);

 GRAPH_CLOCKER:
  S=INDEX(P(LOCAL_PROCESS),3);
  EXECUTE(WAIT_RESPONSE,NO_RETURN) NODE(S);
  /* SIGNAL GRAPH PROCEDURE TO -EGIN,EQLIVALENT TC FCP TRANSFER
     OF CONTROL INFORMATION */
  EXECUTE(WAIT_RESPONSF,NO_RETURN) NODE(S);
  /* GRAPH PROCEDURE RESPONDS AND THEN SUSPENDS ITSELF WHEN
     ALL INPUTS HAVE BEEN FETCHED,MUST REAWAKED AT THIS PCINT
     TO SIMULATE CONNECT TO FCP TO GENERATE CUPUT */
  NULL_ACTIVATE(WAIT_RESPONSE) NODE(P(SELF));
  /* THIS NULL ACTIVATE SIMULATES WAIT AFTER STATUS RECIEVED */
  WAKEUP(WAIT_RESPONSE,NO_RETURN,POP) NCDE(S) AND THEN RETURN;
  /* RESULTS OF COMPUTATION IS HELD IN PCRT */

 /* LOCAL  ENVIRONMENT OF GRAPH MACHENE

    1  DESCRIPTOR OF MAIN MEMORY
    2  WORKING REGISTER
  PORT-4 REGISTERS                               */

 S3:
  S=P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSICN=2,
       ACCESS_CONTROL=GLOBAL);
  /* DEFINE GLOBAL DATA ENVIRONMENT */
  S=DESCRIPTOR OF MEMORY_ARRAY DEFINED FROM(
     DESCRIPTOR=L(1),OFFSET=0) WITH DIMENSION=CCNS(PROGRAM_AREA),
     WORD_LENGTH=8;
   (STORE(S(1),1),POPSTACK);
 S=DESCRIPTOR OF MEMORY_ARRAY DEFINED FROM
    (DESCRIPTOR=L(1),OFFSET=CONS(PROGRAM_AREA)) WITH
    DIMENSION=CONS(DATA_AREA),WORD_LENGTH=8;
 (STORE(S(1),2),POPSTACK);
 /* INITIALIZED GLOBAL DATA ENVIRONMENT OF GRAPH_PROCEDURE */
 (STORE(P(LOCAL_DATA),2),POPSTACK);
 /* SAVE POINTER OF ENVIRONMENT IN L(2) */
 RETURN(SFO(3,NULL));
 CONSTANTS(PROGRAM_AREA:2048,DATA_AREA:30720);
  /* DATA AREA SHOULD BE 960*NUMBER_CF_LINKS */

SONSGM:
 RETURN(LOAD_CONSTANT(PC1,I.BEG));
 CONSTANTS(SPACE_MANAGER,PROCESSOR_SCHEDULER,
           GRAPH_PROCEDURE);

LOCAL_SONSGM:
 GO TO *+I.BEG;
 LOCAL_SM:
  RETURN(P(DESCRIPTOR OF REGISTER_BLCCK WITH DIMENSION=6));
 LOCAL_PS:
  RETURN(P(DESCRIPTOR OF REGISTER_BLCCK WITH DIMENSION=11));
 LOCAL_GP:
  S=P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSICN=10);
  STORE_INDEX(S,3),I=_0;
  /* INIT PROGRAM ADDRESS=0 */
  RETURN((SFO(S,NULL),POPSTACK));

PORT_SONSGM:
 S=ACCESS(P(PSTACK),-2), GO TO *+I.BEG;
 PORT_SM:
  RETURN(P(DESCRIPTOR OF REGISTER_BLOCK DEFINED FROM
          (DESCRIPTOR=S(POP),OFFSET=0) WITH DIMENSION=2));
 PORT_PS:
  RETURN((INDEX(S,11),POPSTACK));
 PORT_GP:
  RETURN(P(DESCRIPTOR OF REGISTER_BLCCK DEFINED FROM
          (DESCRIPTOR=S(POP),OFFSET=0) WITH DIMENSION=2));

EPSV_SONSGM:
 GO TO *+I.BEG;
 EPSV_SM:
  RETURN(P(EPSV WITH PSTACK=F(STACK4),VSTACK=F(STACK2),
          GLOBAL_DATA=L(2),GLOBAL_PRCCESS=S));
 EPSV_PS:
  RETURN(P(EPSV WITH PSTACK=F(STACK6),VSTACK=F(STACK4),
          GLOBAL_DATA=L(2),GLOBAL_PRCCESS=S));
 EPSV_GP:
  RETURN(P(EPSV WITH PSTACK=F(STACK10),VSTACK=F(STACK6),
          GLOBAL_DATA=L(2),GLOBAL_PRCCESS=S));
```

```
SPACE_MANAGER:

 /* LOCAL_ENVIRONMENT
     1-2 PORT
     3   CURRENT SPACE ALLOCATION INIT(C)
     4-6 WORKING REGISTERS

  ACTIVATION_CODE=
     8   REQUEST STORAGE-PORT(1)=NUMBER OF 64 BIT WORDS
     9   RELEASE STORAGE-PORT(1)=NUMBER OF 64 BIT WORDS
                             PORT(2)=BASE OF SPACE ALLOCATED */

 STORE_INDEX(P(LOCAL_DATA),3),I=_0;
REQUEST_LOOP:
 (COMPARE(ACT_CODE,8),MODIFY_STATUS),GO TO *+C(=);
 GO TO RELEASE_STORAGE;

REQUEST_STORAGE:
 S=SHIFT(L(1),-6);
 WAKEUP(CONTINUE) NODE(P(RETURN)) WITH INPUT=L(3);
 S=(ADD(S,L(3)),POPSTACK);
(TIME_GRAIN):
 (STORE(P(LOCAL_DATA),3),POPSTACK),GO TO REQUEST_LOOP;

RELEASE_STORAGE:
 (TIME_GRAIN): GO TO REQUEST_LOOP;
```

```
PROCESSOR_SCHEDULER:

/* LOCAL DATA ENVIRONMENT OF SCHEDULER_PROCESSOR
 1  A) HEAD (1:4) INIT(2),
    B) TAIL (5:8) INIT(1),
    C) CURRENT_SIZE (9:12) INIT(0),
    D) FREE_LIST (17:32)
       INIT('1(NUMBER OF PROCESSORS)'B)

 2-9   QUEUE OF PENDING REQUESTS

 10    QUEUE OF CORRESPONDING INITIATICN NUMBERS(8) BIT(4)

 11     PORT                                              */

 DEFINE STATIC PROCESS WHOSE SUBSTRUCTLRE CCNTAINS
  CONS(NUMBER_PROCESSORS) SONS WITH
       PROGRAM=CONS(SONSPS),
       LOCAL_DATA=F(LOCAL_SONSPS),
       PORT=F(PORT_SONSPS),
       EPSV=F(EPSV_SONSPS)
           AND
       CLOCKING PROCESS=INVOKE
       PROGRAM(SCHEDULER_CLOCKER) WITH INITIALIZE_ROUTINE=F(INIT_PS);

INIT_PS:
 I=_CONS(INIT_QUEUESPS),
 RETURN(STORE_INDEX(P(LOCAL_DATA),1));

LOCAL_SONSPS:
 S=P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSION=10);
 S=SHIFT(I.BEG,-4), I='1(4)0(4)'B;
 (MASK_STORE(S(1),5), POPSTACK);
 RETURN((SFO(S,NULL),POPSTACK));
 /*INITIALIZE PROCESSOR NUMBER*/

PORT_SONSPS:
 S=ACCESS(P(PSTACK),-2);
 RETURN(P(DESCRIPTOR OF IO_BLOCK DEFINED FRCM(DESCRIPTOR=S(POP),
 OFFSET=0) WITH DIMENSION=2));

EPSV_SONSPS:
 RETURN(P(EPSV WITH
        VSTACK=F(VS_PROCESSOR),PSTACK=
        F(PS_PROCESSOR),EXTERNAL_ENV= P(SELF)));
 (TIME_GRAIN):
    GO TO SCHEDULER_CLOCKER;
 /* SUSPEND UNTIL NEXT REQUEST */

 NULL_ACTIVATE(WAIT_RESPONSE) NODE(P(SELF));
 /* WAIT NEXT REQUEST SO WILL NOT GET ANCTHER PRCCESSOR REQUEST */
 GO TO PROCESSOR_COMPLETE;

 PROCESSOR_COMPLETE:
    (MASK_COMPARE(L(1),0), MODIFY_STATLS), I=_CCNS(MASK_C),
           GO TO *+C(=);
 GO TO ASSIGN_PROCESSOR;

 S=SUBTRACT(1,L(11));
 S=(SHIFT('1'B,S),POPSTACK), I=RESULT;
(TIME_GRAIN):
 (MASK_STORE(P(LOCAL_DATA),1),POPSTACK), GC TC SCHEDULER_CLOCKER;

ASSIGN_PROCESSOR:
 S=SUBTRACT(9,F(HEAD));
 S=(SHIFT(S,-2),POPSTACK),I='1(4)'B;
 S=(SHIFT_MASK(L(10),S),POPSTACK);
 /* COMPUTER ADDRESS OF PROCESSOR TC BE ALLCCATED */
 S=INDEX(P(LOCAL_PROCESS),L(11)),I=F(HEAD);
 /* INITIALIZE PROCESSOR WITH ADDRESS CF NCDE AND INIT NUMBER */
 EXPAND(CONTINUE,NO_RETURN) NODE(S) WITH INPUT=S(1),
  RETURN_ADDRESS=L(I);
 /* SIGNAL NODE WITH PROCESSOR */
 WAKEUP(ACT_CODE=9,REFERENCE,CONTINUE,PCP) NODE(L(I)) WITH INPUT=S;
/* UPDATE HEAD */
 (MASK_COMPARE(L(1),CONS(H9)),MODIFY_STATUS,PCPSTACK),
   I=_CONS(MASK_H),GO TO *+C(¬=);
 S=SHIFT(2,-28), GO TO *+2;
 S=MASK_ADD(L(1),CONS(H1));
 (MASK_STORE(P(LOCAL_DATA), 1), POPSTACK);
 S=MASK_SUBTRACT(L(1), CONS(C1)), I=_CCNS(MASK_C);

(TIME_GRAIN):
 (MASK_STORE(P(LOCAL_DATA),1),POPSTACK), GC TC SCHEDULER_CLOCKER;

CONSTANTS(SONSPS:
 PROCESSOR,INIT_QUEUESPS: '0010000l0(8)0(10)1(6)'B,
       NUMBER_PROCESSORS:6);
```

```
SCHEDULER_CLOCKER:
 (COMPARE(ACT_CODE,9), MODIFY_STATUS),
 GO TO *+C(=);
 GO TO PROCESSOR_COMPLETE;

REQUEST_PROCESSOR:
 S=(LOGICAL_AND(L(1),CONS(L16)), MODIFY_STATUS), I=1,GO TO *+C(¬=);
 (COMPARE(NULL,NULL),POPSTACK),GO TO NC_FREE_PRCCESSOR;
 (LCOMPARE(S,'1'B),MODIFY_STATUS),GO TC *+C(¬=);
 S=(SHIFT(S,1),POPSTACK),I=I+1,GO TO *-1;
FREE_PROCESSOR:
 S=(INDEX(P(LOCAL_PROCESS),I),POPSTACK),I=I-1;
 EXPAND(CONTINUE,NO_RETURN)
 NODE(S) WITH INPUT=L(11),
  RETURN_ADDRESS=P(RETURN);
 WAKEUP(ACT_CODE=9,REFERENCE,CONTINLE)
  NODE(P(RETURN)) WITH INPUT=S;
 S=(SHIFT('1'B,-I),POPSTACK), I=RESLLT;
 S=(SFO(0,NULL),POPSTACK);
(TIME_GRAIN):
 (MASK_STORE(P(LOCAL_DATA),1), POPSTACK), GC TC SCHEDULER_CLCCKER;


 NO_FREE_PROCESSOR:
    S=MASK_ADD(L(1),CONS(C1)), I=_CCNS(MASK_C);
 /*C=C+1*/
    (MASK_STORE(P(LOCAL_DATA),1), PCPSTACK);
    (MASK_COMPARE(L(1),CONS(T9)), MODIFY_STATLS), I=_CONS(MASK_T),
          GO TO *+C(¬=);
 /*T=END OF LIST*/
    S=SHIFT(2,-24), GO TO *+2;
    S=MASK_ADD(L(1),CONS(T1));
 /*STORE T*/
    MASK_STORE(P(LOCAL_DATA),1);
 S=(MASK_SHIFT(S,24),POPSTACK), I=P(RETLRN);
 STORE_INDEX(P(LOCAL_DATA),S);

 /* NEED TO QUEUE UP REQUEST */

 S=(SUBTRACT(S,9),POPSTACK);
 S=(SHIFT(S,-2),POPSTACK), I=RESULT;
 S=(SHIFT('1(4)'B,I),POPSTACK);
 S=SHIFT(L(11),I), I=S(POP);
 (MASK_STORE(P(LOCAL_DATA),10),POPSTACK);

 /* STORE AWAY INITIATION NUMBER */

 (MASK_COMPARE(L(1),CONS(C8)),MODIFY_STATUS), I=_CCNS(MASK_C),
  GO TO *+C(=);
```

```
PROCESSOR:

/* LOCAL DATA ENVIRONMENT OF PROCESSCR
 1-2    PORT
 3      ADDRESS OF CONNECTED NODE
 4      INPUT_LINK_STATUS(1:16),
        OUTPUT_LINK_STATUS(17:32)
 5      CONTROL_INFORMATION(1:16)
        PROCESSOR_NUMBER(25:28)
        INITIATION_NUMBER(29:32)
 6-10   WORKING REGISTERS
                                              */
 DEFINE DYNAMIC PROCESS WHOSE SUBSTRLCTURE CONTAINS 0 SONS
        AND
 CLOCKING PROCESS=
        INVOKE PROGRAM(PSEUDO_FUNCTION) WITH INITIALIZE_ROUTINE
        =_F(INIT_PROCESSOR);

 INIT_PROCESSOR:
 STORE_INDEX(P(LOCAL_DATA),3), I=_P(RETURN);
 S=SFO(L(1),NULL), I='1(4)'B;
 RETURN((MASK_STORE(P(LOCAL_DATA),5),PCPSTACK));
/*STORE AWAY CONNECTED_NODE ADDRESS AND INITIATICN NUMBER */

PSEUDO_FUNCTION:
 S=SHIFT(L(1),-16), I=CONS(U16);
 (MASK_STORE(P(LOCAL_DATA),5),POPSTACK);
 S=LOAD_CONSTANT(PC1,L(1)), I=RESULT, GC TC FTJUMP;
 /* STORE CONTROL_INFORMATION */

FUNCTION_TABLE:
 CONSTANTS(
        FADD,
        FSUBTRACT,
        FTWO_COPIES,
        FBRANCH_ROUTE,
        FNEGATION,
        FMINUS_1,
        FCOND_ROUTE,
        FMULTIPLY,
        FZERO_TEST,
        FLOOP_CONTROL
                 );

 FTJUMP:
  (COMPARE(NULL,NULL),POPSTACK), GO TC I;
```

```
FSUBTRACT:
FADD:
FBRANCH_ROUTE:
FMULTIPLY:
FCOND_ROUTE:
FETCH_TWO_OPERANDS:
 WAKEUP(FETCH_INPUT,WAIT_RESPONSE) NCDE(P(RETURN));
 STORE_INDEX(P(LOCAL_DATA),6), I=_L(2);
 /* STORE AWAY OPERAND (L) */

FTWO_COPIES:
FZERO_TEST:
FMINUS_1:
FNEGATION:
FETCH_ONE_OPERAND:
 WAKEUP(FETCH_INPUT,WAIT_RESPONSE) NCDE(P(RETURN));
 /*SIGNAL INPUT PHASE COMPLETE */
 WAKEUP(RET_TERM,CONTINUE) NODE(P(RETURN));
 S=SHIFT_MASK(L(5),16), I=_'1(5)'B;
 S=(LOAD_CONSTANT(PC1,S),POPSTACK), GC TO RESULT;
 CONSTANTS(ADD,
       SUBTRACT,
       TWO_COPIES,
       BRANCH_ROUTE,
       NEGATION,
       MINUS_1,
       COND_ROUTE,
       MULTIPLY,
       ZERO_TEST
            );

ADD:
 S=(ADD(L(6),L(2)),POPSTACK), GO TC STCRE_CPER1;

SUBTRACT:
 S=(SUBTRACT(L(6),L(2)),POPSTACK), GC TC STCRE_CPER1;

TWO_COPIES:
 S=(SFO(L(2),NULL),POPSTACK),GO TO STCRE_OPER2;

BRANCH_ROUTE:
 S=(SFO(L(2),NULL),POPSTACK);
 (COMPARE(L(6),0),MODIFY_STATUS),GO TC *+C(=);
 GO TO STORE_OPER1;
 S=SHIFT('01'B,-14),I=CONS(L16),GO TC STCRE_OLTPUT_STAT;

ZERO_TEST:
NEGATION:
 (COMPARE(L(2),0),POPSTACK,MODIFY_STATLS),GC TC *+C(=);
 S=SFO(0,NULL),GO TO STORE_OPER1;
 S=SFO(1,NULL),GO TO STORE_OPER1;

MINUS_1:
 S=(SUBTRACT(L(2),1),POPSTACK),GO TO STORE_OPER1;

COND_ROUTE:
 (COMPARE(L(6),0),POPSTACK,MODIFY_STATUS),GO TO *+C(=);
 S=SFO(L(2),NULL),GO TO STORE_OPER1;
 S=SFO(0,NULL),I=CONS(L16),GO TO STORE_OUTPUT_STAT;

MULTIPLY:
 S=(MULTIPLY(L(6),L(2)),POPSTACK),GO TO STORE_OPER1;

FLOOP_CONTROL:
 WAKEUP(FETCH_INPUT,WAIT_RESPONSE) NODE(P(RETURN));
 WAKEUP(STORE_STATUS,CONTINUE) NODE(P(RETURN))
    WITH INPUT=CONS(LUX);
 STORE_INDEX(P(LOCAL_DATA),4),I=_CONS(LUX),GO TO TRANSFER_RESULT;
 CONSTANTS(LUX: '010(14)10(15)'B);

STORE_OPER0:
 S=SFO(0,NULL),I=CONS(L16),GO TO STORE_OUTPUT_STAT;
STORE_OPER2:
 S=SHIFT('11'B,-14),I=CONS(L16),GO TO STORE_OUTPUT_STAT;
STORE_OPER1:
 S=SHIFT('1'B,-15), I=CONS(L16);
STORE_OUTPUT_STAT:
 (MASK_STORE(P(LOCAL_DATA),4),POPSTACK);
 /* SET OUTPUT LINK STATUS */
 /* FETCH INITIATION NUMBER */
 S= LOGICAL_AND(L(5),'1(4)'B);
 /* SIGNAL NODE OUTPUT READY */
 WAKEUP(ACT_CODE=10,SUSPEND,POP) NODE(L(3)) WITH INPUT=S;
/* STORE OUTPUT LINK STATUS */
 WAKEUP(STORE_STATUS,CONTINUE) NODE(P(RETURN)) WITH INPUT=L(4);
/* NOT SURE WHETHER LEAVE DATA IN PORT OR TRANSFER */
 (STORE(P(LOCAL_DATA),2),POPSTACK);
/* STORE OUTPUT IN PORT */
TRANSFER_RESULT:
 S=(SHIFT_MASK(L(4),14),MODIFY_STATUS),I=_'11'B,GO TO *+C(¬=);
 GO TO SEND_COMPLETE_SIGNAL;
 WAKEUP(STORE_OUTPUT,WAIT_RESPONSE) NODE(P(RETURN));
 (COMPARE(S,3),MODIFY_STATUS),GO TO *+C(¬=);
 WAKEUP(STORE_OUTPUT,WAIT_RESPONSE) NODE(P(RETURN));
SEND_COMPLETE_SIGNAL:
 WAKEUP(RET_TERM,CONTINUE,POP) NODE(P(RETURN));
/*SIGNAL PROCESSOR SCHEDULER COMPLETE */
 S=SHIFT_MASK(L(5),4), I=_'1(4)'B;
 WAKEUP(ACT_CODE=8,CONTINUE,POP) NODE(P(EXTERNAL_ENV)) WITH INPUT=S
        AND THEN RETURN;
```

```
GRAPH_PROCEDURE:

 /* LOCAL DATA ENVIRONMENT OF GRAPH PROCEDURE

    1-2  PORT
    3    ADDRESS OF PROLOG(16),NUM_LINKS(8),NLM_NCDES(8)
    4    BASE OF LINK AREA IN MAIN MEMCRY
    5    OUTPUT_LINK_STATUS(16),PROCESS_NLM(8),NEIL(4),NEOL(4)
    6    ADDRESS OF PROCEDURE NODE CLCCKER
    7-10 INSTRUCTION BUFFER AREA I(1)...I(NEIL),
                                 O(1)...C(NECL)
                                                          */

 /* GLOBAL DATA ENVIRONMENT

    1 DESCRIPTOR OF PROGRAM AREA IN MEMCRY SUBSYSTEM
    2 DESCRIPTOR OF DATA AREA IN MEMORY SUBSYSTEM

                                                    */


  DEFINE DYNAMIC PROCESS WHOSE SUBSTRLCTURE CONTAINS F(LINK_NODES)
       SONS WITH PROGRAM=F(SONS_GP),
  LOCAL_DATA=F(LOCAL_SONS_GP) PORT=F(PCRT_SCNS_GP),
  EPSV=F(EPSV_SONS_GP),
       AND
  CLOCKING PROCESS=INVOKE PROGRAM(PROCEDURE_CLCCKER) WITH
   INITIALIZE_ROUTINE=_F(INITIALIZE_LINKS);
```

```
LINK_NODES:
  READ ELEMENT(F(PADDRESS)) WITH LENGTH=16 FROM MEMORY ARRAY
       (DESCRIPTOR=G(1),OFFSET=0);
 /*STORE AWAY NUMBER OF LINKS AND READ IN L(3)/(16:0)*/
  S=SHIFT(L(1),16),I=CONS(L16);
  (MASK_STORE(P(LOCAL_DATA),3),POPSTACK), I=CONS(U16);
 /*ADD 2 TO PADDRESS*/
  S=MASK_ADD(L(3),CONS(C2U));
  (MASK_STORE(P(LOCAL_DATA),3),POPSTACK);
 /*CALL SPACE_MANAGER*/
  S=SHIFT(F(NUMBER_LINKS),-4);
  S=INDEX(P(GLOBAL_PROCESS),1);
  EXECUTE_SINGLE_CYCLE(WAIT_RESPONSE,NO_RETURN,ACT_CODE=8,POP)
    NODE(_S(POP)) WITH INPUT=S,RETURN_ADDRESS=F(SELF);
 /*CALL SPACE MANAGER, AND THEN STORE AWAY BASE OF LINK_AREA IN
  MEMORY_SUBSYSTEM*/
  STORE_INDEX(P(LOCAL_DATA),4), I=_L(1);
 /*SET PORT TOO*/
  STORE_INDEX(P(LOCAL_DATA),1), I=_0;
 /*COMPUTE NUMBER OF LINKS AND NODES AND THEN RETURN*/
  RETURN(ADD(F(NUMBER_LINKS), F(NUMBER_NODES)));

PADDRESS:
  RETURN(SHIFT(L(3),16));

NUMBER_LINKS:
  I=_'1(8)'B, RETURN(SHIFT_MASK(L(3),8));

NUMBER_NODES:
  RETURN(LOGICAL_AND(L(3),'1(8)'B));

NODE_TYPE:
  RETURN(SHIFT(L(1),30));
NODE_TYPE1:
  S=SHIFT(L(1),30);
  RETURN((ADD(S,1),POPSTACK));
 /*CREATE PROGRAM ADDRESS OF LINK OR PNODE, PROC_NODE, SNODE)*/

SONS_GP:
  (COMPARE(I.BEG,F(NUMBER_LINKS)),MODIFY_STATUS), GO TO *+C(>);
  RETURN(SFO(CONS(LINK_PROGRAM), NULL));
 /*IF NODE THEN READ FIRST TWO BYTES OF NODE DEFINITION*/
  READ ELEMENT(F(PADDRESS)) WITH LENGTH=16 FROM MEMORY ARRAY
       (DESCRIPTOR=G(1), OFFSET=0);
 /*LOAD APPROPRIATE PROGRAM ADDRESS*/
  RETURN(LOAD_CONSTANT(PC1,F(NODE_TYPE1)));
  CONSTANTS(LINK_PROGRAM:LINK_PROCESS,PARALLEL_NODE,
            PROCEDURE_NODE,SEQUENTIAL_NODE);
```

```
LOCAL_SONS_GP:
  GO TO *+F(NODE_TYPE1);

LOCAL_LINK:
     S=SUBTRACT(I.BEG,1), GO TO LOCAL_LINK_CCNT;

 LOCAL_PNODE:
     RETURN(P(DESCRIPTOR OF REGISTER_BLCCK WITH DIMENSION=11));

 LOCAL_PROC_NODE:
     RETURN(P(DESCRIPTOR OF REGISTER_BLCCK WITH DIMENSION=11));

  /*CREATE LOCAL_DATA_ENVIRONMENT FOR LINK AND INITIALIZE REGISTER
     TO DESCRIPTOR OF LINK AREA*/
 LOCAL_SNODE:
     RETURN(P(DESCRIPTOR OF REGISTER_BLCCK WITH DIMENSION=2));

 LOCAL_LINK_CONT:
     S=P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSION=5);
     (STORE(P(LOCAL_DATA),2),POPSTACK);
     S=(MULTIPLY(S,CONS(LENGTH_QUEUE)),PCPSTACK);
     S=(ADD(S,L(4)),POPSTACK);
     S=DESCRIPTOR OF MEMORY_ARRAY DEFINED FRCM(DESCRIPTOR=G(2),
      OFFSET=S(POP)) WITH DIMENSION=CCNS(LENGTH_CUEUE),WORD_LENGTH=64;
     (STORE(L(2),2),POPSTACK);
     RETURN(SFD(L(2),NULL));

PORT_SONS_GP:
  S=ACCESS(P(PSTACK),-2), GO TO *+F(NCDE_TYPE1);
 /*GET DESCRIPTOR OF LOCAL ENV*/

 PORT_LINK:
     RETURN(P(DESCRIPTOR OF IO_BLOCK DEFINED FRCM(DESCRIPTOR=S(POP),
            OFFSET=3) WITH DIMENSION=2, WCRD_LENGTH=64));

 PORT_PNODE:
     RETURN(INDEX(S(POP),11));

 PORT_PROC_NODE:
     RETURN(INDEX(S(POP),11));

 PORT_SNODE:
     RETURN(INDEX(S(POP),1));

 EPSV_SONS_GP:
   GO TO *+F(NODE_TYPE1);

  EPSV_LINK:
     RETURN(P(EPSV WITH PSTACK=F(PS_LINK), VSTACK=F(VS_LINK),
                GLOBAL_PROCESS=S));
```

```
EPSV_PNODE:
    RETURN(P(EPSV WITH PSTACK=F(STACK1C), VSTACK=F(VS_PNODE),
         GLOBAL_DATA=F(GLOBAL_PNODE), GLOBAL_PROCESS=S,
          EXTERNAL_ENV=P(SELF) ) );

EPSV_PROC_NODE:
    RETURN(P(EPSV WITH PSTACK=F(STACK1C), VSTACK=F(VS_PNODE),
         GLOBAL_DATA=F(GLOBAL_PNODE), GLOBAL_PROCESS=S,
          EXTERNAL_ENV=P(SELF) ) );

EPSV_SNODE:
    RETURN(P(EPSV WITH PSTACK=F(STACK1C), VSTACK=F(STACK4),
         GLOBAL_DATA=F(GLOBAL_PNODE), GLOBAL_PROCESS=S,
          EXTERNAL_ENV=P(SELF) ) );

PS_LINK:
PS_PNODE:
PS_FETCHOP:
PS_PROCESSOR:
PS_STOREOP:
STACK6:
    RETURN(P(DESCRIPTOR OF STACK WITH DIMENSION=6));
STACK8:
    RETURN(P(DESCRIPTOR OF STACK WITH DIMENSION=8));
STACK10:
    RETURN(P(DESCRIPTOR OF STACK WITH DIMENSION=10));
VS_LINK:
VS_PNODE:
VS_FETCHOP:
VS_IPNODE:
VS_PROCESSOR:
STACK4:
    RETURN(P(DESCRIPTOR OF STACK WITH DIMENSION=4));
STACK2:
    RETURN(P(DESCRIPTOR OF STACK WITH DIMENSION=2));

GLOBAL_PNODE:
  S=P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSION=6,
       ACCESS_CONTROL=GLOBAL);
 /*INITIALIZE REGISTER 2 TO PROLOG ADDRESS/FIRST 16 BIT OF INS*/
  (STORE(P(LOCAL_DATA),2),POPSTACK);
  S=SHIFT(L(1),16), I=CONS(L16);
  (MASK_STORE(L(2),2),POPSTACK);
  S=SFO(L(3),NULL), I=CONS(U16);
  (MASK_STORE(L(2),2),POPSTACK);
 /*UPDATE PADDRESS, : PADDRESS +IN+OUT+NODE_TYPE*/
  S=SHIFT_MASK(L(1),16),I=_'1(4)'B;
  S=SHIFT_MASK(L(1),20);
  S=ADD(S(POP),S(POP)), I=F(PADDRESS);
  S=ADD(F(NODE_TYPE1),S(POP)), I=I+RESULT;
  S=(SHIFT(I,-16),POPSTACK);
  (MASK_STORE(P(LOCAL_DATA),3),POPSTACK),I=_CONS(U16);
  RETURN(SFO(L(2),NULL));

INITIALIZE_LINKS:
  STORE_INDEX(P(LOCAL_DATA),6),I=_P(RETURN);
INTERNAL_INITIALIZE:
  READ ELEMENT(F(PADDRESS)) WITH LENGTH=16 FROM MEMORY
       ARRAY(DESCRIPTOR=G(1),OFFSET=0);
 /*INCREMENT PADDRESS BY 2*/
  S=ADD(L(3),CONS(C2U));
  (MASK_STORE(P(LOCAL_DATA),3), POPSTACK), I=_CONS(U16);
 /*CHECK TO SEE WHETHER ANY MORE LINKS TO INITIALIZE*/
  S=(SHIFT(L(1),24),MODIFY_STATUS), GO TO *+C(¬=);
  (COMPARE(NULL,NULL),POPSTACK),GO TO NULL; /* FINISH EXPANSION */
 /*GET NUM OF DATA ITEMS TO BE PLACED ON LINK*/
  S=SHIFT_MASK(L(1),16), I=_'1(8)'B;

DATA_TRANSFER_LOOP:
  READ ELEMENT(F(PADDRESS)) WITH LENGTH=64 FROM MEMORY
       ARRAY(DESCRIPTOR=G(1),OFFSET=0);
 /*CALL LINK*/
  EXECUTE_SINGLE_CYCLE(VALUE,NO_RETURN,STORE_OUTPUT) SON(S(1))
  WITH INPUT=P(PORT) THEN WAIT FOR 1 SONS TO SIGNAL RETURN;
 /*UPDATE PADDRESS BY 8*/
  S=ADD(L(3),CONS(C8U)),I=CONS(U16);
  (MASK_STORE(P(LOCAL_DATA),3),POPSTACK);
  S=(SUBTRACT(S,1),POPSTACK, MODIFY_STATUS), GO TO *+C(=);
  GO TO DATA_TRANSFER_LOOP;
  COMPARE(S(POP),S(POP)),GO TO INTERNAL_INITIALIZE;
 /* THIS SECTION OF CODE EXECUTED AFTER EXPANSION PHASE */
```

```
PROCEDURE_CLOCKER:
  S=SHIFT_MASK(L(1),16),I=_'1(8)'B;
 /* EXTRACT NEIL-NEOL */
  MASK_STORE(P(LOCAL_DATA),5);
  S=(SHIFT(S,4),POPSTACK,MODIFY_STATUS),GO TO *+C(¬=);
  (COMPARE(NULL,NULL),POPSTACK),GO TO BEGIN_GRAPH; /* NO INPUTS */
 /* READ IN I(1)...I(NEIL)         */
  S=SFO(P(PORT),NULL);
  S=P(DESCRIPTOR OF REGISTER_BLOCK DEFINED FROM(DESCRIPTOR=
      P(LOCAL_DATA),OFFSET=6) WITH DIMENSION=4);
  SET_STATE(1,S(POP));
 /* MULT NEIL BY 8 TO GET BIT LENGTH */
  S=(SHIFT(S,-3),POPSTACK);
  READ ELEMENT(F(PADDRESS)) WITH LENGTH=S FROM MEMORY ARRAY(
   DESCRIPTOR=G(1),OFFSET=0);
 /* RESET PORT TO LOCAL_DATA(1-2) */
  SET_STATE(1,S(POP));
 /* UPDATE PADDRESS */
  S=(SHIFT(S,-13),POPSTACK);
  S=(MASK_ADD(S,L(3)),POPSTACK),I=CONS(L16);
  MASK_STORE(P(LOCAL_DATA),3);

FETCH_INPUT_DATA:
  WAKEUP(FETCH_INPUT,CONTINUE) NODE(P(RETURN));
 /* OVERLAP FETCH OF DATA WITH COMPUTATION OF LINK ADDRESS */
 /* COMPUTE ADDRESS OF INPUT LINK TO BE INITIALIZED */
  S=SHIFT_MASK(L(5),6),I=_'1(2)'B;
  S=(ADD(S,7),POPSTACK);
 /* S=NEIL/2+7 */
  S=(ACCESS(P(LOCAL_DATA),S),POPSTACK);
  S=SHIFT_MASK(L(5),4),I=RESULT;
  S=(LOAD_CONSTANT(CONS(SHIFT_BYTE),I),POPSTACK),I=I+_1;
 (TIME_GRAIN):
  S=(SHIFT_MASK(S,_S(POP)),POPSTACK),I=_'1(8)'B;
 /* S=NUMBER OF LINK(I(J)) */
  EXECUTE_SINGLE_CYCLE(VALUE,NO_RETURN,STORE_OUTPUT,POP) SON(S)
  WITH INPUT=P(PORT) THEN WAIT FOR 1 SONS TO SIGNAL RETURN;
 /* UPDATE NEIL */
  S=(MASK_SUBTRACT(L(5),'10(4)'B),MODIFY_STATUS),I=_'1(4)0(4)'B,
   GO TO *+C(=);
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK),GO TO FETCH_INPUT_DATA;

 /* SEND INPUT TERMINATE SIGNAL */
  WAKEUP(RET_TERM) NODE(P(RETURN))  ;
```

```
BEGIN_GRAPH:

  S=LOGICAL_AND(L(3),'1(8)'B),I='1(8)'B;
  S=SHIFT_MASK(L(3),8);
  S=(ADD(S,1),POPSTACK);
 /* S(-1)=NUM_LINS+1,S=NUM_NODES */
 /* ACTIVATE ALL NODES */
  EXECUTE(NO_RETURN,POP) S SONS STARTING AT SON(_S(POP))
   THEN WAIT FOR 0 SONS TO SIGNAL RETURN;
 /* DETERMINE WHEN TO TERMINATE GRAPH */
RETURN_NODES:
  S=SFO(P(PORT),NULL);
  S=P(DESCRIPTOR OF REGISTER_BLOCK DEFINED FROM(DESCRIPTOR=
     P(LOCAL_DATA),OFFSET=6) WITH DIMENSION=4);
  SET_STATE(1,S(POP)),I='1(4)0(3)'B;
  S=SHIFT_MASK(L(5),-3);
  READ ELEMENT(F(PADDRESS)) WITH LENGTH=S(POP) FROM
   MEMORY ARRAY(DESCRIPTOR=G(1),OFFSET=C);
 /* RESET PORT */
  SET_STATE(1,S(POP));
 /* GET EXTERNAL OUPUT LINK STATUS */
 /* INITIALIZE OUTPUT LINK STATUS=0 */
  S=SFO(0,NULL),I=CONS(U16);
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK);
 /* COMPUTE ADDRESS OF OUTPUT LINK TO FIND STATUS */
FETCH_OUTPUT_STATUS:
  S=SHIFT_MASK(L(5),2),I=_'1(2)'B;
  S=(ADD(S,7),POPSTACK);
  S=(ACCESS(P(LOCAL_DATA),S),POPSTACK);
  S=LOGICAL_AND(L(5),'1(2)'B),I=RESULT;
  S=(LOAD_CONSTANT(CONS(SHIFT_BYTE),I),POPSTACK),I=I+_1;
  S=(SHIFT_MASK(S,_S(POP)),POPSTACK),I=_'1(8)'B;
 /* S=NUMBER OF LINK(O(K)) */
  EXECUTE_SINGLE_CYCLE(FETCH_INPUT,NO_RETURN,POP) SON(S)
    THEN WAIT  FOR 1 SONS TO SIGNAL RETURN;
  (COMPARE(L(2),0),MODIFY_STATUS), GO TO *+C(¬=);
  GO TO UPDATE_NEOL;
 /* MODIFY OUTPUT STATUS LINK */
  S=LOGICAL_AND(L(5),'1(4)'B),I=RESULT;
  S=(SHIFT('1'B,I),POPSTACK),I=I-_32;
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK),I=_S;
UPDATE_NEOL:
  S=(MASK_SUBTRACT(L(5),1),MODIFY_STATUS),I=_'1(4)'B,
     GO TO *+C(=);
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK),
     GO TO FETCH_OUTPUT_STATUS;
SIGNAL_OUTPUT_READY:
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK),I='1(8)'B;
  S=SHIFT_MASK(L(5),8);
 /* S=PROCESSOR NUMBER IN PARALLEL NODE INITIATION QUEUE */
```

```
  WAKEUP(ACT_CODE=10,SUSPEND,POP) NODE(L(6)) WITH INPUT=S;
 /* SEND OUTPUT LINK STATUS TO OUTPUT_PNCDE */
  S=SHIFT(L(5),16);
  WAKEUP(STORE_STATUS,CONTINUE,POP) NCDE(P(RETURN)) WITH INPUT=S;
FETCH_OUTPUT_DATA:
  S=(LOGICAL_AND(L(5),CONS(U16)),MODIFY_STATLS),I=0,GO TO *+C(¬=);
  (COMPARE(NULL,NULL),POPSTACK),GO TO TERMINATE_CUTPUT;
  S=(SHIFT(S,-1),POPSTACK,MODIFY_STATLS),I=I+1,GC TC *+_C(<);
  GO TO *-1;
  S=SFO(I,NULL),I='1(4)'B;
  S=(MASK_ADD(L(5),S),POPSTACK);
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK),I=CCNS(U16);
  (MASK_STORE(P(LOCAL_DATA),5),POPSTACK);
 /* COMPUTE ADDRESS OF OUTPUT LINK */
  S=SHIFT_MASK(L(5),2),I=_'1(2)'B;
  S=(ADD(S,7),POPSTACK);
  S=(ACCESS(P(LOCAL_DATA),S),POPSTACK);
  S=LOGICAL_AND(L(5),'1(2)'B),I=RESULT;
  S=(LOAD_CONSTANT(CONS(SHIFT_BYTE),I),PCPSTACK),I=I+_1;
  S=(SHIFT_MASK(S,_S(POP)),POPSTACK),I=_'1(8)'B;
  EXECUTE_SINGLE_CYCLE(FETCH_INPUT,NO_RETURN,PCP) SCN(S)
   THEN WAIT FOR 1 SONS TO SIGNAL RETLRN;
 /* STORE IN OUTPUT_PNODE DATA FETCHED FROM LINK */
  WAKEUP(STORE_OUTPUT,WAIT_RESPONSE) NCDE(P(RETURN)) ;
  GO TO FETCH_OUTPUT_DATA;

 /* HAVE COMPLETED TRANSFERING DATA, NCW MUST TERMINATE */
 TERMINATE_OUTPUT:
  S=SHIFT(F(NUMBER_LINKS),-4);
  (STORE(P(LOCAL_DATA),1),POPSTACK);
  STORE_INDEX(P(LOCAL_DATA),2),I=_L(4);
  S=INDEX(P(GLOBAL_PROCESS),1);
  WAKEUP(CONTINUE,NO_RETURN,ACT_CODE=9,\ALUE,PCP) NODE(S)
   WITH INPUT=P(PORT),RETURN_ADDRESS=P(SELF);
 /* NEED TO CALL SPACE MANAGER AT THIS TO DEALLCCATE SPACE */
  S=ADD(F(NUMBER_LINKS),F(NUMBER_NODES));
 /* SIGNAL ALL LINKS AND NODES TO TERMINATE  */
  EXECUTE_SINGLE_CYCLE(ACT_CODE=12,NO_RETURN) S SCNS STARTING AT
   SON(1) THEN WAIT FOR F(NUMBER_NODES) SONS TC SIGNAL RETURN;
  TERMINATE(POP) _S SONS STARTING AT SCN(1)
   THEN WAIT FOR S SONS TO SIGNAL RETLRN;
  WAKEUP(RET_TERM,CONTINUE) NODE(P(RETLRN)) AND THEN RETURN;

  CONSTANTS(C2U:'0(14)10(17)'B, U16:'1(16)0(16)'B,
            L16:'C(16)1(16)'B,C8U:'C(12)10(19)'B,
            LENGTH_QUEUE:1024          );
             /* LENGTH_QUEUE SHOULD BE 960 */
```

```
LINK_PROCESS:

/* LOCAL DATA ENVIRONMENT OF LINK_PROCESS
   1 QUEUE_STATUS
        A) HEAD_QUEUE(1:4) INIT(1),
        B) TAIL_QUEUE(5:8) INIT(0),
        C) CURRENT_QUEUE_SIZE(9:12) INIT(0),
        D) INPUT_REQUEST_READY(13:13) INIT(0),
        E) OUTPUT_REQUEST_PENDING(14:14) INIT(0),
        F) TERMINATE_CONDITION(22:22) INIT(0),
     2) DESCRIPTOR FOR QUEUE AREA
     3) PENDING REQUEST ADDRESS,
     4-5) PORT;

 ACTIVATION_CODE= 3,FETCH_OPERAND,
                = 4,STORE_OUTPUT,
                =   TERMINATE                                 */

 STORE_INDEX(P(LOCAL_DATA),1),I=_CONS(LINK_QUEUE);

LINK_WAKEUP:
 (COMPARE(3,ACT_CODE),MODIFY_STATUS),GO TO *+C(¬=);
 GO TO LINK_FETCH;
 (COMPARE(4,ACT_CODE),MODIFY_STATUS),GO TO *+C(¬=);
 GO TO LINK_STORE;

/* SET UP TERMINATE_CONDITION */
LINK_TERMINATE:
 S=SFO(CONS(TERMINATE_COND),NULL),I=RESULT;
 (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
 (MASK_COMPARE(L(1),0),MODIFY_STATUS),
  I=_CONS(MASK_IRP), GO TO *+C(=);
 GO TO IRP_DSET;
(TIME_GRAIN):
 GO TO LINK_WAKEUP;

LINK_FETCH:
 (MASK_COMPARE(L(1),0),MODIFY_STATUS),
  I=_CONS(TERMINATE_COND), GO TO *+C(=);
DUMMY_TRANSFER:
 GO TO TRANSFER_LINK_DATA;
/* CHECK WHETHER QUEUE IS EMPTY */
 (MASK_COMPARE(L(1),0),MODIFY_STATUS),
  I=_CONS(MASK_QUEUE_SIZE), GO TO *+C(¬=);
LINK_QUEUE_EMPTY:
 STORE_INDEX(P(LOCAL_DATA),3),I=_P(RETURN), GO TO LQE_CONT;
/* SAVE ADDRESS OF PERSON REQUESTING */
FETCH_QUEUE:
 S=SHIFT_MASK(L(1),28),I=_'1(4)'B;
 READ ELEMENT(S(POP)) WITH LENGTH=64 FROM MEMORY ARRAY(
   DESCRIPTOR=L(2),OFFSET=-64);
 /* UPDATE  QUEUE POINTERS */
  S=MASK_SUBTRACT(L(1),CONS(C1)),I=_CONS(MASK_C);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
  (MASK_COMPARE(L(1),I),MODIFY_STATUS),
   I=_CONS(MASK_H);
  S=SHIFT(1,-28),GO TO *+C(=);
  S=(MASK_ADD(L(1),S),POPSTACK);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
 TRANSFER_LINK_DATA:
  WAKEUP(STORE_OUTPUT,VALUE,CONTINUE) NODE(P(RETURN))
   WITH INPUT=P(PORT);
 /* CHECK FOR ORP SET */
  (MASK_COMPARE(L(1),0),MODIFY_STATUS),
   I=_CONS(MASK_ORP), GO TO *+C(¬=);
 (TIME_GRAIN):
  GO TO LINK_WAKEUP;

  WAKEUP(CONTINUE) NODE(L(3));
  S=SFO(0,NULL);
 (TIME_GRAIN):
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK),GO TO LINK_WAKEUP;

 LQE_CONT:
  S=SFO(CONS(MASK_IRP),NULL),I=RESULT;
 (TIME_GRAIN):
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK),GO TO LINK_WAKEUP;

 LINK_STORE:
  (MASK_COMPARE(L(1),0),MODIFY_STATUS),
   I=_CONS(TERMINATE_COND),GO TO *+C(=);
 DUMMY_STORE:
  GO TO STORE_COMPLETE;

  (MASK_COMPARE(L(1),0),MODIFY_STATUS),
   I=_CONS(MASK_IRP), GO TO *+C(=);
  GO TO IRP_SET;

 STORE_QUEUE:
  (MASK_COMPARE(L(1),I),MODIFY_STATUS),
   I=_CONS(MASK_T);
  S=SFO(CONS(T1),NULL),GO TO *+C(=);
  S=(MASK_ADD(L(1),S),POPSTACK);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
  /* TAIL OF LINK_QUEUE IS UPDATED */

  S=SHIFT_MASK(L(1),24),I=_'1(4)'B;
  STORE ELEMENT(S(POP)) WITH LENGTH=64 INTO MEMORY ARRAY(
   DESCRIPTOR=L(2),OFFSET=-64);
 /* UPDATE QUEUE SIZE */
 S=MASK_ADD(L(1),CONS(C1)),I=_CONS(MASK_C);
 MASK_STORE(P(LOCAL_DATA),1);
 /* CHECK WHETHER QUEUE IS FULL */
 (MASK_COMPARE(S,I),POPSTACK,MODIFY_STATUS),
  GO TO *+C(¬= );
 GO TO SET_ORP;

STORE_COMPLETE:
 WAKEUP(CONTINUE) NODE(P(RETURN));
(TIME_GRAIN):
 GO TO LINK_WAKEUP;

SET_ORP:
 (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
(TIME_GRAIN):
 STORE_INDEX(P(LOCAL_DATA),3),I=_P(RETLRN),
  GO TO LINK_WAKEUP;

IRP_SET:
 WAKEUP(CONTINUE) NODE(P(RETURN));
 /* TURN OFF IRP */
IRP_DSET:
 S=SFO(0,NULL),I=CONS(MASK_IRP);
 (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
 WAKEUP(STORE_OUTPUT,VALUE,CONTINUE) NCDE(L(3))
  WITH INPUT=P(PORT);
(TIME_GRAIN):
 GO TO LINK_WAKEUP;
 CONSTANTS( LINK_QUEUE:'0(3)10(28)'B,
            MASK_IRP:'0(12)10(19)'B,
            MASK_QUEUE_SIZE:'0(8)1(4)C(20)'B,
            MASK_ORP:'0(13)10(18)'B          );
```

```
PARALLEL_NODE:

 /*  GLOBAL DATA ENVIRONMENT

     1    NULL(16),OUPUT_LINK_STATUS(16)
     2    INSTRUCTION_ADDRESS(16),OPCODE(8),IN(4),CUT(4)
     3-6 INSTRUCTION BUFFER AREA I(1)...I(IN),
                                  O(1)...C(OUT);

   LOCAL DATA ENVIRONMENT
     1     QUEUE STATUS
            A) HEAD(1:4) INIT(2)
            B) TAIL(5:8) INIT(1)
            C) CURRENT_SIZE(9:12) INIT(0)
            D) OUTPUT_READY_LIST(13:20) INIT('0(8)'B)
            E) SCHEDULER_CALL_POSTPONED(21:21) INIT('0'B)
            F) TERMINATE_CONDITION(22:22) INIT('0'B)
            G) PREFETCH_COMPLETE(23:23) INIT('0'B);

     2-9   INITIATION QUEUE
     10    SCHEDULER ADDRESS
     11    PORT

    ACTIVATION_CODES FOR CALL PARALLEL NODE
      8    PREFETCH COMPLETE
      9    PROCESSOR ASSIGNED
      10   OUTPUT READY
      11   OUTPUT COMPLETE
      12   TERMINATE
                                                   */

DEFINE STATIC PROCESS WHOSE SUBSTRUCTURE CONTAINS 2 SONS WITH
  PROGRAM=F(SONSP),
LOCAL_DATA=F(LOCAL_SONSP), PORT=F(PORT_SONSP), EPSV=F(EPSV_SONSP)
                 AND
CLOCKING PROCESS=
  INVOKE PROGRAM(PNODE_CLOCKER) WITH
     INITIALIZE_ROUTINE=_F(INIT_PNODE);
```

```
SONSP:
  RETURN(LOAD_CONSTANT(PC1,I.BEG));
  CONSTANTS(INPUT_PNODE, OUTPUT_PNODE);

LOCAL_SONSP:
  RETURN(P(DESCRIPTOR OF REGISTER_BLCCK WITH DIMENSION=1));
 /*ALLOCATE SINGLE REGISTER*/

PORT_SONSP:
  S=ACCESS(P(PSTACK),-2);
  RETURN(INDEX(S(POP),1));
 /*SAME AS LOCAL ENVIRONMENT, ACCESS CFF PSTACK*/

EPSV_SONSP:
  GO TO *+I.BEG;
  RETURN(P(EPSV WITH VSTACK=F(VS_IPNCDE), PSTACK=F(STACK8)));
  RETURN(P(EPSV WITH VSTACK=F(STACK6), FSTACK=F(STACK8)));
 /*EXPAND SON NODES, INIT LOCAL ENV, EXECUTE INPUT_PNODE*/


INIT_PNODE:
  EXPAND(RETURN) SON(1)  THEN WAIT FOR 1 SCNS
  TO SIGNAL RETURN;
  EXPAND(RETURN) SON(2)  THEN WAIT FCR 1 SCNS
  TO SIGNAL RETURN;
 /*INIT QUE_STATUS*/
  STORE_INDEX(P(LOCAL_DATA),1), I=_CCNS(CUE_STAT);
 /*GET ADDRESS OF SCHEDULER*/
  S=MASK_SHIFT(G(2),8), I=_CONS(MASK_CP);
  S=(SUBTRACT(S,32),POPSTACK,MODIFY_STATUS),GC TC *+C(>=);
  S=(SFO(1,NULL), POPSTACK);
  S=(INDEX(P(EPSV),-2),POPSTACK),I=S;
  S=(INDEX(S,I),POPSTACK),I=I+_1;
  (STORE(P(LOCAL_DATA),10), POPSTACK);
  EXECUTE_SINGLE_CYCLE(CONTINUE,NO_RETURN) SCN(1)  AND THEN RETURN;
```

```
PNODE_CLOCKER:
  WAIT_WAKEUP:
       NULL_ACTIVATE(SUSPEND) NODE(P(SELF))  ;
       S=LOAD_CONSTANT(CONS(TABLE),ACT_CCDE), GC TO RESULT;

CONSTANTS(TABLE: *-3,
  PREFETCH_COMPLETE, DEVICE_ASSIGNED, CLTPUT_READY, OUTPUT_COMPLETE,
  TERMINATE_NODE);

PREFETCH_COMPLETE:
  S=(SFO(CONS(MASK_PREFETCH),NULL),PCPSTACK),I=RESULT;
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
  /* SET PREFETCH_COMPLETE */
  (MASK_COMPARE(L(1), CONS(C8)); MODIFY_STATLS),
       I=_CONS(MASK_C), GO TO *+C(¬=);

SCHEDULER_POSTPONED:
  S=SFO(CONS(MASK_SCP),NULL),I=RESULT,GC TC SP_CCNT;

UPDATE_QUEUE:
  (MASK_COMPARE(L(1),0),MODIFY_STATUS),
   I=_CONS(TERMINATE_COND),GO TO *+C(=);
  GO TO CHECK_TERMINATE;
  S=MASK_ADD(L(1),CONS(C1)), I=_CONS(MASK_C);
 /*C=C+1*/
  (MASK_STORE(P(LOCAL_DATA),1), POPSTACK);
  (MASK_COMPARE(L(1),CONS(T9)), MODIFY_STATLS), I=_CONS(MASK_T),
       GO TO *+C(¬=);
 /*T=END OF LIST*/
  S=SHIFT(2,-24), GO TO *+2;
  S=MASK_ADD(L(1),CONS(T1));
 /*STORE T*/
  MASK_STORE(P(LOCAL_DATA),1);

CALL_SCHEDULER:
  S=(MASK_SHIFT(S,24),POPSTACK);
  EXECUTE_SINGLE_CYCLE(VALUE,CONTINUE,NC_RETLRN,ACT_CODE=9,POP)
   NODE(L(10)) WITH INPUT=S, RETURN_ADDRESS=F(SELF);
  GO TO WAIT_WAKEUP;

 SP_CONT:
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK),GC TC WAIT_WAKEUP;

DEVICE_ASSIGNED:
  S=(MASK_SHIFT(L(1),24),POPSTACK),I=_CCNS(MASK_T);
  (STORE_INDEX(P(LOCAL_DATA),S),POPSTACK), I=_L(11);
  /* SET PREFETCH NOT COM-LETE */
  S=SFO(0,NULL),I=CONS(MASK_PREFETCH);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
  EXECUTE_SINGLE_CYCLE(CONTINUE,REFERENCE,NC_RETURN) SON(1)
   WITH INPUT=L(11);
  GO TO WAIT_WAKEUP;
 OUTPUT_READY:
  S=(SUBTRACT(L(11),21),POPSTACK);
  S=(SHIFT(1,S),POPSTACK),I=RESULT;
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK),GO TO AB;

OUTPUT_COMPLETE:
  (MASK_COMPARE(L(1),0),POPSTACK,MODIFY_STATUS),I=_CONS(MASK_SCP),
   GO TO *+C(=);
  GO TO CALL_SCH_POST;
 /*C=C-1*/
  S=MASK_SUBTRACT(L(1), CONS(C1)), I=_CONS(MASK_C);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK), GO TO UPDATE_HEAD;

  /* IF TERMINATE&PREFETCH_COMPLETE AND QUEUE=0
      THEN TERMINATE SON NODES */
 CHECK_TERMINATE:
  (MASK_COMPARE(L(1),I),MODIFY_STATUS),I=_CONS(MASK_TCPF),
   GO TO *+C(¬=);
  (MASK_COMPARE(L(1),0),MODIFY_STATUS),I=_CONS(MASK_C),
   GO TO *+C(=);
  GO TO WAIT_WAKEUP;
TERM_NODE:
  TERMINATE(RETURN) 2 SONS STARTING AT SON(1) THEN
   WAIT FOR 2 SONS TO SIGNAL RETURN;
  WAKEUP(RET_TERM) NODE(P(EXTERNAL_ENV)) AND THEN RETURN;

CALL_SCH_POST:
  S=SFO(0,NULL);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
  S=MASK_SHIFT(L(1),4);
  MASK_STORE(P(LOCAL_DATA),1), I=_CONS(MASK_T);
  S=(MASK_SHIFT(S, 24),POPSTACK);
  EXECUTE_SINGLE_CYCLE(VALUE,CONTINUE,NC_RETURN,ACT_CODE=9,POP)
   NODE(L(10)) WITH INPUT=S, RETURN_ADDRESS=F(SELF);

UPDATE_HEAD:
  (MASK_COMPARE(L(1), CONS(H9)), MODIFY_STATUS), I=_ CONS(MASK_H),
   GO TO *+C(¬=);
  S=SHIFT(2,-28), GO TO *+2;
  S=MASK_ADD(L(1),CONS(H1));
  (MASK_STORE(P(LOCAL_DATA), 1), POPSTACK);
 AB:
  S=SUBTRACT(F(HEAD),21);
  S=(SHIFT(1,S),POPSTACK),I=RESULT;
  (MASK_COMPARE(L(1),0),POPSTACK,MODIFY_STATUS),
   GO TO *+C(¬=);
  GO TO CHECK_TERMINATE;
  /* RESET ORL */
  S=SFO(0,NULL);
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK);
  S=ACCESS(P(LOCAL_DATA), F(HEAD));
  EXECUTE_SINGLE_CYCLE(REFERENCE,NO_RETLRN,PCP) SON(2) WITH INPUT=S;
  GO TO WAIT_WAKEUP;
HEAD: I=_CONS(MASK_H), RETURN(MASK_SHIFT(L(1), 28));

 TERMINATE_NODE:
  S=(SFO(CONS(TERMINATE_COND),NULL),PCPSTACK),I=RESULT;
  (MASK_STORE(P(LOCAL_DATA),1),POPSTACK),GC TO CHECK_TERMINATE;
  /* SET TERMINATE CONDITION */

  CONSTANTS(MASK_H:'1(4)0(28)'B,H1:'0001C(28)'B,
            H9:'10010(28)'B,MASK_T:'C(4)1(4)0(24)'B,
            T1:'0(4)00010(24)'B,T9:'C(4)10010(24)'B,
            MASK_C:'0(8)1(4)0(20)'B,C1:'C(8)00010(20)'B,
            C8:'0(8)10000(20)'B,MASK_SCF:'0(20)10(11)'B,
             TERMINATE_COND: '0(21)10(1C)'B,
            MASK_TCPF:'C(21)1(2)0(9)'B,MASK_PREFETCH:'0(22)10(9)'B);
```

```
INPUT_PNODE:

 /* LOCAL  DATA ENVIRONMENT
    1 PORT

   GLOBAL  DATA ENVIRONMENT SAME AS PARALLEL NODE */

   DEFINE STATIC PROCESS WHOSE SUBSTRUCTURE CONTAINS F(IN) SONS WITH
        PROGRAM=CONS(SONSIP),
        LOCAL_DATA=F(LOCAL_SONSIP),
        PORT=F(PORT_SONSIP),
        EPSV=F(EPSV_SONSIP),
             AND
   CLOCKING PROCESS=INVOKE PROGRAM(PINPUT_CLOCKER) WITH
     INITIALIZE_ROUTINE=_F(INIT_IPNODE);

IN:   I=_CONS(MASK_IN), RETURN(MASK_SHIFT(G(2), 4));

INX8:   I=_CONS(MASK_IN), RETURN(MASK_SHIFT(G(2), 1));
   CONSTANTS(SONSIP: FETCH_OPERAND, MASK_IN: '0(24)11110(4)'B);

LOCAL_SONSIP:
   S=P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSION=3);
   S=SFO(I.BEG, NULL);
   (STORE(S(1),1), POPSTACK);
   RETURN((SFO(S, NULL),POPSTACK));
 /*ALLOCATE 3 REGISTER SET FIRST REGISTER TO LINK NUMBER*/

PORT_SONSIP:
   S=ACCESS(P(PSTACK),-2);
   RETURN(P(DESCRIPTOR OF IO_BLOCK DEFINED FROM
        (DESCRIPTOR=S(POP),OFFSET=1) WITH DIMENSION=2));

EPSV_SONSIP:
   RETURN(P(EPSV WITH VSTACK =F(VS_FETCHOP), PSTACK =F(PS_FETCHOP)));
 /*FETCH I,...I,N AND EXPAND SONS*/


INIT_IPNODE:
   S=P(DESCRIPTOR OF IO_BLOCK DEFINED FROM(DESCRIPTOR=P(GLOBAL_DATA),
   OFFSET= 2) WITH DIMENSION=4);
   SET_STATE(1, S(POP));
 /*CHANGE PORT TO GLOBAL ENV. 3-6*/
   READ ELEMENT(2) WITH FORMAT=NULL AND LENGTH=F(INX8) FROM MEMORY
    ARRAY(DESCRIPTOR=F(MEM_DESC), OFFSET=F(INS_ADD));
   EXPAND(RETURN) F(IN) SONS STARTING AT SON(1)
   THEN WAIT FOR F(IN) SONS TO SIGNAL RETURN;
   RETURN(SET_STATE(1, P(LOCAL_DATA)));
 /*ACCESS EXT ENV TO GET ELEMENT OF GLOBAL DATA ENV*/
```

```
MEM_DESC:
  S=INDEX(P(EPSV),2); RETURN(ACCESS(S(PCP),1));

INS_ADD:
  I=_CONS(MASK_INS_ADD), RETURN(MASK_SHIFT(G(2), 13));

OP_CODE:
  I=_CONS(MASK_OPCODE), RETURN(MASK_SHIFT(G(2),8));
  CONSTANTS(MASK_INS_ADD: '1(16)0(16)'B,
  MASK_OPCODE: '0(19)1(5)0(8)'B);


P INPUT_CLOCKER:
  EXECUTE(WAIT_RESPONSE) F(IN) SONS STARTING AT SON(1)
  THEN WAIT FOR F(IN) SONS TO SIGNAL RETURN;
  WAKEUP(ACT_CODE=8,SUSPEND) NODE(P(RETURN)) ;
  ACTIVATE(NO_CONNECT) FUNCTIONAL_UNIT(L(1)) WITH CONTROL_INFORMATION=
   F(OP_CODE) USING F(IN) INPUT_GENERATORS INITIATED BY
    RETRIEVE(NO_RETURN) COMMAND;
  GO TO P INPUT_CLOCKER;


PROCEDURE_NODE:

 /* EXACTLY SAME CODE AS PARALLE_NODE, EXCEPT
  THAT INSTEAD OF CALLING SCHEDULER TO ASSIGN
  A PROCESSOR, AN MSV FOR A GRAPH_PROCEDURE
  IS GENERATED. THE ADDRESS OF THIS GRAPH_PROCEDURE
  IS THEN TREATED IN THE SAME WAY AS THE ADDRESS OF
  THE PROCESSOR RETURNED BY THE SCHEDULER.
  THIS MODIFICATION TO THE CODE OF THE PARALLEL_NODE
  WOULD BE AT LOCATION CALL_SCHEDULER. THE CODE FOR
  GENERATING THE MSV OF THE GRAPH PROCEDURE IS VERY
  SIMILAR TO THE CODE USED BY THE GRAPH_MACHINE TO
  GENERATE THE MSV OF THE MAIN GRAPH PROCEDURE.  */
```

```
SEQUENTIAL_NODE:

 /*  GLOBAL DATA ENVIRONMENT

     1   INPUT_LINK_STATUS(16),OUPUT_LINK_STATUS(16)
     2   INSTRUCTION_ADDRESS(16),OPCODE(8),IN(4),OUT(4)
     3-6 INSTRUCTION BUFFER AREA I(1)...I(IN),
                                 O(1)...O(OUT);

    LOCAL DATA ENVIRONMENT

     1   PORT
     2   SCHEDULER ADDRESS                       */

DEFINE STATIC PROCESS WHOSE SUBSTRUCTURE CONTAINS 2 SONS WITH
  PROGRAM=F(SONSS),LOCAL_DATA=F(LOCAL_SONSS),
  PORT=_F(PORT_SONSIP),EPSV=F(EPSV_SONSP)
                        AND
CLOCKING PROCESS=
  INVOKE PROGRAM(SNODE_CLOCKER) WITH
    INITIALIZE_ROUTINE=_F(INIT_SNODE);

SONSS:
  RETURN( LOAD_CONSTANT(PC1,I.BEG));
  CONSTANTS(INPUT_SNODE,STORE_OPERAND);

LOCAL_SONSS:
  GO TO *+I.BEG;
  RETURN(P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSION=1));
  RETURN(P(DESCRIPTOR OF REGISTER_BLOCK WITH DIMENSION=3));

INIT_SNODE:
  EXPAND(RETURN) SON(1)  THEN WAIT FOR 1 SONS
  TO SIGNAL RETURN;
 /*GET ADDRESS OF SCHEDULER*/
  S=MASK_SHIFT(G(2),8), I=_CONS(MASK_OP);
  S=(SUBTRACT(S,32),POPSTACK,MODIFY_STATUS),GO TO *+C(>=);
  S=(SFO(1,NULL), POPSTACK);
  S=(INDEX(P(EPSV),-2),POPSTACK),I=S;
  S=(INDEX(S,I),POPSTACK),I=I+_1;
  RETURN((STORE(P(LOCAL_DATA),2),POPSTACK));
CONSTANTS(
  QUE_STAT: '001000010(24)'B,
  MASK_OP: '0(18)1(6)0(8)'B);
```

```
SNODE_CLOCKER:
   EXECUTE(NO_RETURN) SON(1);NULL_ACTIVATE(SUSPEND) NODE(P(SELF));
   (COMPARE(ACT_CODE,8),MODIFY_STATUS), GO TC *+C(¬=);
   GO TO *+3;
   NULL_ACTIVATE(SUSPEND) NODE(P(SELF));
   GO TO TERM_NODE;
   /* CALL SCHEDULER */
   EXECUTE_SINGLE_CYCLE(VALUE,WAIT_RESPCNSE,NC_RETURN,ACT_CODE=9)
    NODE(L(2)) WITH INPUT=1,RETURN_ADDRESS=P(SELF);
   ACTIVATE(NO_CONNECT) FUNCTIONAL_UNIT(L(1)) WITH
   CONTROL_INFORMATION=F(OP_CODE) USING 1 INPLT_GENERATORS INITIATED BY
     EXECUTE_SINGLE_CYCLE(RETURN,VALLE) CCMMAND THEN STORE STATUS
      IN F(STATUS_ADDRESS);
   GO TO SNODE_CLOCKER;
```

```
INPUT_SNODE:

 /*  LOCAL  DATA  ENVIRONMENT

      1    WORKING REGISTERS

    GLOBAL  DATA ENVIRONMENT
          SAME AS  SEQUENTIAL  NODE                               */

  DEFINE STATIC PROCESS WHOSE SUBSTRUCTURE CONTAINS F(IN) SONS WITH
        PROGRAM=CONS(SONSSP),
        LOCAL_DATA=F(LOCAL_SONSIP),
        PORT=F(PORT_SONSIP),
        EPSV=F(EPSV_SONSIP),
             AND
  CLOCKING PROCESS=
       INVOKE PROGRAM(SINPUT_CLOCKER) WITH
        INITIALIZE_ROUTINE=_F(INIT_ISNODE);
 /REFILL/

INIT_ISNODE:
  S=P(DESCRIPTOR OF REGISTER_BLOCK DEFINED FROM
       (DESCRIPTOR=P(GLOBAL_DATA),OFFSET=0) WITH DIMENSION=1);
  SET_STATE(1,S(POP));
  READ ELEMENT(2) WITH FORMAT=NULL AND LENGTH=16 FROM  MEMORY
    ARRAY(DESCRIPTOR=F(MEM_DESC),OFFSET=F(INS_ADD));
  /* SET  INITIAL  STATE  OF  LOCKS */
  S=P(DESCRIPTOR OF IO_BLOCK DEFINED FROM(DESCRIPTOR=P(GLOBAL_DATA),
  OFFSET=2) WITH DIMENSION=4);
  SET_STATE(1, S(POP));
 /*CHANGE PORT TO GLOBAL ENV. 3-6*/
  READ ELEMENT(4) WITH FORMAT=NULL AND LENGTH=F(INX8) FROM MEMORY
   ARRAY(DESCRIPTOR=F(MEM_DESC), OFFSET=F(INS_ADD));
  EXPAND(RETURN) F(IN) SONS STARTING AT SON(1)
  THEN WAIT FOR F(IN) SONS TO SIGNAL RETURN;
  S=ADD(F(IN),4);
  READ ELEMENT(S(POP)) WITH FORMAT=NULL AND LENGTH=F(OUTX8)
  FROM MEMORY ARRAY
      (DESCRIPTOR=F(MEM_DESC),OFFSET=F(INS_ADD));
  RETURN(SET_STATE(1, P(LOCAL_DATA)));
```

```
SINPUT_CLOCKER:
  STORE_INDEX( P(LOCAL_DATA),1),I=_0;
  S=(LOGICAL_AND(G(1),CONS(UX16)),MODIFY_STATUS),GO TO *+C(¬=);
  (COMPARE(NULL,NULL),POPSTACK),GO TC NLLL;

FIND_UNLOCKED_IEDGE:
  S=(SHIFT(S,-1),POPSTACK,MODIFY_STATLS),
       I=I+1, GO TO *+_C(<);
  GO TO *-1;
  S=ADD(1,L(1));
  (STORE(P(LOCAL_DATA),1),POPSTACK);
  EXECUTE(CONTINUE) SON(I) THEN WAIT FCR 0 SCNS TC SIGNAL RETURN;
  GO TO *+C(=);
  GO TO FIND_UNLOCKED_IEDGE;

  NULL_ACTIVATE(WAIT_RESPONSE) SON(1) THEN WAIT FCR
   L(1) SONS TO SIGNAL RETURN;
  /* THEN SIGNAL PREFETCH COMPLETE */
  WAKEUP(ACT_CODE=8,SUSPEND) NODE(P(RETLRN));
  S=INDEX(P(LOCAL_DATA),1);
  SET_STATE(2,S(POP));
  S=(LOGICAL_AND(G(1),CONS(UX16)),MODIFY_STATUS,PCPSTACK),I=0;
FIND_UIE:
  S=(SHIFT(S,-1),POPSTACK,MODIFY_STATLS),I=I+1,GC TO *+_C(<);
  GO TO *-1;
  S=INDEX(P(LOCAL_PROCESS),I);
  (STORE(P(LOCAL_DATA),1),POPSTACK);
  GO TO *+C(=);
 (TIME_GRAIN):
  GO TO FIND_UIE;
(TIME_GRAIN):
  (COMPARE(NULL,NULL),POPSTACK),GO TO SINPUT_CLCCKER;
  CONSTANTS(UX16:'1(16)0(16)'B,SONSSP:FETCH_CPERAND);
```

```
FETCH_OPERAND:
  DEFINE STATIC PROCESS WHOSE SUBSTRLCTLRE CCNTAINS 0 SONS
            AND
  CLOCKING PROCESS=EXECUTE_SINGLE_CYCLE(NC_RETLRN,FETCH_INPUT)
  NODE(_F(LINK_ADD)) WITH RETURN_ADDRESS=P(SELF);

 /* LOCAL DATA ENVIRONMENT
   1    LINK ADDRESS I(J)
   2-3  PORT

  GLOBAL DATA ENVIRONMENT SAME AS PARALLEL NCDE */

LINK_ADD:
  S=SHIFT(L(1), 2), I=RESULT;
  S=(ACCESS(P(GLOBAL_DATA), I), POPSTACK), I=I +_3;
 /*EXTRACT G(3)+I MOD(4)) FIELD*/
  S=LOGICAL_AND(L(1),'1(2)'B),I=RESLLT;
  S=(LOAD_CONSTANT(CONS(SHIFT_BYTE),I),PCPSTACK),I=I+_1;
  S=(SHIFT_MASK(S,_S(POP)),POPSTACK),I=_'1(8)'B;
  RETURN((INDEX(P(GLOBAL_PROCESS), S), PCPSTACK));
  CONSTANTS(MASK_BYTE:MASK_8BITS,SHIFT_BYTE:SHIFT_8BITS);
MASK_8BITS:
  CONSTANTS('0(24)1(8)'B,'1(8)0(24)'B, 'C(8)1(8)0(16)'B,
            '0(16)1(8)0(8)'B);
 SHIFT_8BITS:
  CONSTANTS(0,24,16,8);
```

```
OUTPUT_PNODE:

 /* LOCAL DATA ENVIRONMENT
    1 PORT

   GLOBAL DATA ENVIRONMENT SAME AS PARALLEL NODE */

   DEFINE STATIC PROCESS WHOSE SUBSTRUCTURE CONTAINS 1 SONS WITH
          PROGRAM=CONS(SONOP),
          LOCAL_DATA=F(LOCAL_SONSIP),
          PORT=F(PORT_SONSIP),
          EPSV=F(EPSV_SONOP)
                 AND
   CLOCKING PROCESS
        =INVOKE PROGRAM(POUTPUT_CLOCKER) WITH INITIALIZE_ROUTINE=_F
        (INIT_OPNODE);

EPSV_SONOP:
   RETURN(P(EPSV WITH VSTACK=F(STACK6),PSTACK=F(STACK8)));

INIT_OPNODE:
   S=P(DESCRIPTOR OF IO_BLOCK DEFINED FROM(DESCRIPTOR=P(GLOBAL_DATA),
   OFFSET=2) WITH DIMENSION=4);
   SET_STATE(1, S(POP));
 /*CHANGE PORT TO GLOBAL ENV. 3-6*/
   S=ADD(F(IN),2);
   READ ELEMENT(S(POP)) WITH FORMAT=NULL AND LENGTH=F(OUTX8)
   FROM MEMORY ARRAY
        (DESCRIPTOR=F(MEM_DESC),OFFSET=F(INS_ADD));
   RETURN(SET_STATE(1, P(LOCAL_DATA)));

OUT: I=_CONS(MASK_OUT), RETURN(MASK_SHIFT(G(2),0));

OUTX8: I=_CONS(MASK_OUT), RETURN(MASK_SHIFT(G(2),-3));

    CONSTANTS(SONOP:STORE_OPERAND, MASK_OUT: '0(28)1(4)'B);


  POUTPUT_CLOCKER:
   ACTIVATE(NO_CONNECT) FUNCTIONAL_UNIT(L(1)) WITH CONTROL_INFORMATION=
    NULL USING 0 INPUT_GENERATORS INITIATED BY
     EXECUTE_SINGLE_CYCLE(VALUE,NO_RETURN) COMMAND THEN STORE STATUS
      IN F(STATUS_ADDRESS);
   WAKEUP(ACT_CODE=11,CONTINUE) NODE(P(RETURN));
  (TIME_GRAIN):
    GO TO POUTPUT_CLOCKER;

  STATUS_ADDRESS:
    RETURN(INDEX(P(GLOBAL_DATA),1));
```

```
STORE_OPERAND:

 /* LOCAL DATA ENVIRONMENT
   1    LINK ADDRESS I(J)
   2-3  PORT

  GLOBAL DATA ENVIRONMENT SAME AS PARALLEL NCDE */

INIT_SOPER:
 /*S=G(1), I=0 */
  S=(SHIFT(G(1),-16),MODIFY_STATUS), I=C ,GC TC *+C(¬=);
  RETURN((SFO(0,NULL), POPSTACK));
FIND_OEDGE:
  S=(SHIFT(S,-1),POPSTACK,MODIFY_STATLS),I=I+1,GC TC *+_C(<);
  GO TO *-1;
  STORE_INDEX(P(LOCAL_DATA),1);

 STORE_OUTPUT:
  EXECUTE_SINGLE_CYCLE(NO_RETURN, VALLE,STCRE_CUTPUT)
  NODE(F(LINK_ADD)) WITH INPUT=P(PORT),RETURN_ADDRESS=P(SELF);

  GO TO *+C(=);
 (TIME_GRAIN):
  GO TO FIND_OEDGE;
  RETURN((SFO(0,NULL), POPSTACK));

 /END/
```

APPENDIX D

The Internal Format of Sum-Squared Graph Program(5)
(Figure 27)

Memory Subsystem				Comments
(Start, End, Value)				
			Node Description Section	
1	8	10		nl
9	16	6		nn
17	18	1	Node 1	2 copies
19	24	3		
25	28	1		IN
29	32	2		OUT
33	40	1		I1
41	48	2		01
49	56	9		02
57	58	1	Node 2	2 copies
59	64	3		
65	68	1		IN
69	72	2		OUT
73	80	2		I1
81	88	3		01
89	96	4		02
97	98	1	Node 3	*
99	104	8		
105	108	2		IN
109	112	1		OUT
113	120	3		I1
121	128	4		I2
129	136	5		01
137	138	1	Node 4	+
139	144	1		
145	148	2		IN
149	152	1		OUT
153	160	5		I1
161	168	6		I2
169	176	7		01
177	178	1	Node 5	BR
179	184	4		

185	188	2		IN
189	192	2		OUT
193	200	10		I1
201	208	7		I2
209	216	8		01
217	224	6		02
225	226	1	Node 6	=0
227	232	9		
233	236	1		IN
237	240	1		OUT
241	248	9		I1
249	256	10		01

Link Initialization

257	264	8	Link 8
265	272	1	1 Data Item
273	304	0	
305	336	1	
337	344	6	Link 6
345	352	1	1 Data Item
353	384	0	
385	416	0	
417	424	1	Link 1
425	432	6	6 Data Items
433	464	0	
465	496	1	
497	528	0	
529	560	2	
561	592	0	
593	624	3	
625	656	0	
657	688	4	
689	720	0	
721	752	5	
753	784	0	
785	816	0	

External Link Specification

817	824	0
825	828	0
829	832	1
833	840	8