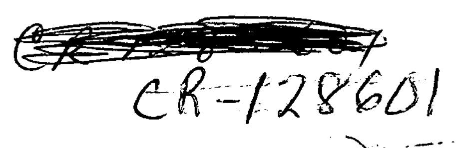
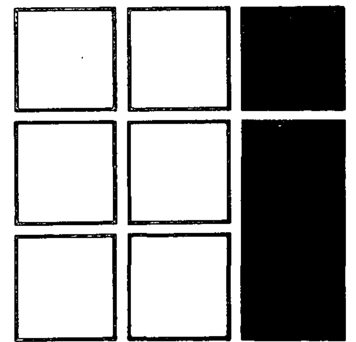


(NASA-CR-128601) ENGINEERING STUDY FOR THE FUNCTIONAL DESIGN OF A MULTIPROCESSOR SYSTEM Final Report J.S. Miller, et al (Intermetrics, Inc.) Sep. 1972 474 p CSCL 09B G3/08

N73-10235

Unclas 45209

Reproduced by
NATIONAL TECHNICAL INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151



INTERMETRICS

474 p.

Final Report

ENGINEERING STUDY FOR THE FUNCTIONAL DESIGN OF A MULTIPROCESSOR SYSTEM

by:

James S. Miller, Woodrow H. Vandever,
Saul F. Stanten, Arra E. Avakian, Alex L. Kosmala

September 1972

Prepared under contract NAS9-11745 by:

Intermetrics, Inc.
701 Concord Avenue
Cambridge, Massachusetts 02138

Intermetrics Technical Report #14-72

FOREWORD

This document is the Final Report of an engineering study for the functional design of a multiprocessor computer system to provide the computational capability of the Data Management System for a Space Station. The study was sponsored by the NASA Manned Spacecraft Center, Houston, Texas, under Contract NAS 9-11745. It was performed by Intermetrics, Inc., Cambridge, Massachusetts, over the period May 1971 to August 1972, under the direction of Dr. James S. Miller and Alex L. Kosmala. The Technical Monitor for the Manned Spacecraft Center was Mr. James P. Ledet. EB-6 4865

Publication of this report does not constitute approval by NASA of the findings or conclusions contained therein.

ii

ACKNOWLEDGEMENTS

The authors wish to acknowledge the strong influence of the Burroughs B6700 computer design on the multiprocessor architecture described in this report. In addition to the references cited from the open literature, the Burroughs Corporation has generously supplied additional information and consultation. We particularly thank Mr. H. Norton Riley for his help.

Our special gratitude is extended to Miss Kester D. Whitney for her endurance in typing the many drafts of this report, and for drawing all the figures and flowcharts.

TABLE OF CONTENTS

		Page
1.0	OBJECTIVES AND SUMMARY	1
	1.1 Introduction	1
	1.2 Multiprocessor Efficiency	2
	1.3 System Reliability	7
	1.4 Summary of Results	12
2.0	INSTRUCTION ARCHITECTURE	21
	2.1 Design Rationale and Method for a HOLM	21
	2.2 Instruction Functions	50
	2.3 MP Instruction Design Factors	63
	2.4 MP Instruction Architecture	88
3.0	MULTIPROCESSOR OPERATING SYSTEMS	157
	3.1 Introduction	157
	3.2 The Process State Controller	163
	3.3 Interrupt Handling	191
	3.4 Memory Management	207
	3.5 I/O Management	251
	3.6 Timing and Synchronization	271
	3.7 Fault Recovery Methodology	309
4.0	FAULT TOLERANT ASPECTS OF THE MULTIPROCESSOR	317
	4.1 General Philosophy and Requirements	317
	4.2 Major Phases of Fault Tolerant Operation	319
	4.3 Error Detection	323
	4.4 Recovery	338

TABLE OF CONTENTS

(continued)

		Page
5.0	IMPLEMENTATIONAL ASPECTS	385
	5.1 Design of Multiprocessor Components	385
	5.2 System Performance	449
	5.3 Laboratory Model	454

1.0

OBJECTIVES AND SUMMARY

1.1 Introduction

The major requirement of this study was to generate a functional system design of a multiprocessing computer system capable of satisfying the computational needs of the Data Management System of a Space Station. These were specified to include:

a) Real time control.

b) Data processing and storage.

c) Data retrieval.

d) Remote terminal servicing.

In accordance with the expected long lived nature of an operational space station, the computer was required, in addition to possessing the necessary processing capability, to be sufficiently fault-tolerant to permit uninterrupted control of Space Station activities, and to be easily expandable in performance and capabilities to provide adequate support for a planned program of Space Station development.

The multiprocessor computer configuration appears to be an attractive candidate for this application, since it has the greatest potential for achieving solutions to these requirements. However, the several attempts to realize operational MP systems have, to date, been plagued with the problem of high executive overhead which has prevented the actual performance from attaining its anticipated potential. A major objective of the study has been to examine the cause and effect of high executive activity in a MP system, and to investigate ways of alleviating processing bottlenecks by judicious tradeoffs between the software and hardware implementations of critical executive functions.

A second major objective of the effort has been to ensure a high degree of reliability in the operation of the MP computer system. This objective was considered in a very broad sense: the study was purposely not confined to just achieving

a computer configuration which could provide a very high expectation of continued operations in the event of a hardware failure. It also addressed itself to the question of the reliability of the necessary operational software. This latter consideration prompted an early assumption that only a modern, structured high order language (HOL) would be used in all application programming for the MP. This assumption led to the feasibility of optimal trade offs of diagnostic and checking functions between the operating system and the compiler. It also led to the major decision to investigate the advantages of executing the statements of an HOL directly, rather than as the output of a compiler tailored to a more conventional machine instruction architecture. The study was required to establish a multiprocessor computer configuration of hardware modules, which were to be specified to the functional level, with interface definitions. Whenever possible the use of available off-the-shelf equipment was to be specified.

The next two sections of this chapter will present in some detail the background and motivation for the achievement during the course of this study of the above objectives. The chapter closes with a brief summary of the remainder of the report.

1.2 Multiprocessor Efficiency

The development of executive and operating systems has pursued two, primarily different, goals. One has been to raise the level of the user interface: to remove the necessity for an application programmer to become familiar with the exact mechanization of, for example, the I/O; to enable the application programmer to specify his problem in a problem oriented way, e.g., via a HOL, or utility routines. This line of development has established a methodological approach to total system design, with an orientation towards the user's needs. The second goal has been to obtain efficiency in the use of such system resources as, for example, I/O devices, channels, operating memory, and most importantly, the processor(s). While the gain that can be realized by allowing the competing activities in the system to mutually share the system's resources seems very attractive, a new set of problems is introduced by this concept.

The question of efficiency is very nebulous, since its quantization depends heavily upon the level of computer operation being examined. For example, at one extreme it is possible to consider as a measure of efficiency the percentage of time that the computer is "in use". At a lower level it is possible

to consider the percentage of time that the processor, (or an independently addressed memory module, or an I/O unit, or the internal bus) is in actual "use". At an even lower level, it might be the percentage of time the different logic elements (registers, combinational circuits, or even individual gates) are in use. With the first definition, almost any computer in a busy, 24 hour per day facility could merit a 100% efficiency rating. At the lowest level, any computer system would have a low efficiency rating, since few gates are ever acting simulantaneously. It is at the inbetween level of definition (the processor, memory, and bus level of the system), which is the preserve of the operating system, that most attempts to increase efficiency are made. The historical development of executive systems with regard to efficiency can be viewed as the pursuit of maximum time utilization of the elements of the network of processors, memory, I/O and their interconnections, to realize an overall effective increase in computational throughput.

Very early computers were in general I/O bound. The "processor" was idle most of the time, while the I/O was executing continuously. Since most I/O devices were time independent of the processor, double buffering techniques were developed to allow I/O access to memory in parallel with the processor. This then is concurrent processing; two (or more) active elements are in "use" simultaneously. The early executive systems were principally concerned with the I/O area. Not only did it make I/O handling easier for the user, but the concurrency in the I/O area increased "efficiency".

If a job is so I/O bound that a significant fraction of time elapses before the processor is used again, the next logical development would be to allow another job to use the processor in the meantime. The concept of multiprogramming is built upon the desire for more efficiency. Given a particular configuration, the hope is that by the device of multiprogramming, more computational power can be obtained at the expense of only a little added overhead. The two concepts of multiplexing (i.e., the sharing of system resources), and the concurrent execution by active elements, are fundamental to the understanding of executive and operating systems. The price of multiplexing is the overhead needed for its control. This overhead is felt both in execution time and in memory space.

Concurrency is the basis for gain in efficiency. How to encourage concurrency, and how to take advantage of it is the basic problem of the multiprocessor designer. Potential parallelism may exist at several different levels:

a) Jobs run on a computer system are generally completely independent of each other, even if they share resources.

b) Within a given job, there may be a structure of independent tasks, which if executed in parallel, could improve the total throughput.

c) Within a routine most statements are independent of each other.

d) Within a single arithmetic statement some computations can be done in parallel.

The price to be paid at each of these levels of potential parallelism to create concurrency are the penalties associated with resource sharing and the synchronization of the independent activities into a whole. The greatest potential for concurrency is at the lower levels of parallelism, but here the overhead of synchronization rises sharply.

These different levels of potential parallelism are of degree rather than kind. From the execution point of view they represent similar characteristics, differing only in the frequency of their occurrence. Most multiprocessing systems to date process independent jobs, or independent parts of a job. However, just as concurrency is found at different levels, so different computer architectures have been implemented exploiting varying degrees of concurrency. The CDC 6600 has several execution elements working on the same instruction stream simultaneously, while the IBM 360/91 pipelines an instruction stream in order to improve efficiency. A multi-computer configuration is generally defined as a multiple processor in which each computer has its own operating memory but shares secondary memory. Multiprocessors proper are generally differentiated from multicomputers by the sharing of operating memory. A multiprocessor represents a complication over a single processor by the introduction of new active elements which must have the capability of sharing all the system resources. Since memory is the outstanding contributor to the cost of the hardware in most computer systems, much work has been done in the area of its efficient use. The multiplexing of memory for both multiprocessing and for virtual memory, to increase the apparent size of memory, has resulted in the concepts of paging, segmentation, and the use of "cache" memories. The additional processing and memory space for executive functions required to control the multiplexing of memory have usually been offset by the gains in throughput for a given system cost.

The use of multiprocessors has been proposed for a number of reasons:

a) reliability and graceful degradation,

b) modularization with the ability to incrementally add computational power to meet an expansion of processing requirements,

c) to achieve computational power not possible with one processor.

The problem of reliability and graceful degradation is not simply a matter of providing multiple copies of all computer elements. The jobs which are critical can never demand more computational power than the degraded state will allow. Therefore, the computational power of any extra processors can only be used by non-critical jobs. The total mission program, therefore, cannot be allowed to make full reliable use of the potential computational power.

There are several reasons why the addition of an extra processor does not add an extra unit of computational power:

a) Two systems, one comprised of a single processor with a unit computation time of T, and the other of n processors each with unit computation time of T/n, are not, in general, of equivalent computational power. Even if the overhead of resource allocation and synchronization in the n processor case is ignored, the n(T/n) processing power will be less than the single T, if ever a situation occurs in which fewer than n processors are executing. Once the number of processors in use has fallen below n, some computation power has been lost, and being measured in time, it can never be made up again.

b) The introduction of multiple processors also introduces a new resource which must be manipulated, namely processors. This entails the creation of a data structure for the management of the processors, and requires time to execute the actual allocation of this resource. Since there is now the possibility of truly concurrent process execution, synchronization of the various processors during allocation also becomes a time overhead.

c) When unique system data has to be accessed by concurrent processes, the problem becomes more difficult than in the simple multiprogramming case. Both the use of Compools and of system data must be carefully managed to prevent erroneous conditions arising from incomplete changes in the data base. The use of the LOCK mechanism commonly provided for this function, creates an intentional bottleneck, and hence incurs a loss of computation by the locked-out processor(s).

d) With the introduction of multiple processors, conflict over operating memory arises, resulting again in one or more of the processors losing some computational power. If the number of separately addressable memory modules is held constant while the number of processors increases, the conflict becomes worse than simply additive, since queues begin to develop. Interleaving is generally used in an attempt to minimize conflict by randomizing memory module usage, and thus obtaining stationary behavior. But since memory conflict can be a major problem, allocation policies to separate memory usage for the different processors are often developed. This, of course, represents another overhead function which degrades the computation power of the multiprocessor system as compared to a single processor.

e) The use of a "cache" memory with a single processor has been found to yield (an apparent) fast response from a relatively slow operating memory. In a multiprocessor system, the use of a cache becomes much less advantageous, because:

1) If a copy of shared data in the cache is updated, the same information must be written through to main memory because other processors must have access to it.

2) Similarly, if shared data is written into, all the other cache memories must be informed of the event in order to invalidate the old value.

3) In the single processor system a cache increases the actual percentage of bus traffic. In a multiprocessor system this increases in bus traffic causes conflict, which again is subject to the worsening effect of queues as the number of processors increases.

f) If a form of virtual memory system is specified, a problem occurs between main and secondary storage which is analogous to that between a cache and main memory with regard to bus traffic. The larger the number of procesosrs in concurrent execution, the more often a page fault occurs and the more conflict over secondary to main memory traffic. Also, the policy controlling the use of main memory becomes complicated if it is designed to minimize such conflict. It is instructive to remember that it is the simple and less elegant solution to a resource management problem that is usually the better, since the

overhead for the elegant solution can easily offset its savings.

g) The policy of interrupt handling is highlighted in a multiprocessor environment. The question as to which processor handles non-process oriented interrupts must be decided.

The above problems of a multiprocessing environment show that the overhead involved is not in general additive, i.e., it does not increase linearly with the number of processors. Since these are basically queing problems, when conflict occurs it tends to increase the probability of further conflict, and this non-linear behavior becomes more marked as the number of processors is increased. It is very difficult to establish, in the general case, an upper limit to the number of processors which can cost effectively be justified to achieve an increased throughput. The behavior of the elements of a multiprocessor and the characteristics of their interactions with each other depend very strongly on the nature of the processing load. Only a specific implementation under known conditions can yield this information, and this was clearly not within the scope of the present study. Except for highly specialized applications, implementations of multiprocessors to date have not exceeded four or five processors. For the purpose of the effort to be described in this report, a minimum of three computational processors and one I/O processor was chosen as the basis of the multiprocessor design.

1.3 System Reliability

1.3.1 Introduction

The concept of system reliability involves the various properties which combine to assure that a program may be run with a very high degree of confidence that it will perform successfully (provided, of course, that the program itself is error-free). The corollary must be also true: no program, even one with errors, should be capable of disrupting others. While the logical design of the hardware is presumed to be error free, failures of components or connections must be detected before damage results, and the capability to circumvent such faults must be provided.

There are four specific areas in which the hardware/operating system designer must pursue system reliability:

a) Elimination of errors in hardware design

b) Elimination of errors in software

c) Detection of and recovery from hardware faults

d) Detection and containment of manifested software faults

Category a) should be self-explanatory. The programmers of a system have a difficult enough time of it without having to worry, for example, about the possibility that an addition operation with certain operands may give the wrong result. While the problem of ensuring a correct hardware design has a fundamental resemblance to its software counterpart, it is more limited, and lends itself well to systematic design and testing.

Because so very much software is written for a given machine, the possibilities for error are high, and the resource of time and money available to check software, on a per function basis, is drastically lower than that for hardware. Because there is currently no means to prove software to be error-free with an adequately high confidence, software difficulties have been raised to the topmost level of importance in almost every application of computers to problems. The fundamental solution appears to lie not in providing better testing and debugging aids, but rather in the improvement of languages and programming techniques to make the occurrence errors much less likely, and to make the detection of the remainder more inevitable.

In the design resulting from this study, the problem has been attacked by providing an environment in which high-order language is the only means of programming; the use of "smart" compilers is then tantamount to a tireless validation processor. Only the limitations in thoroughness of the compilers themselves, which can be incrementally improved during their life times, allow flaws to remain undetected. By supporting comprehensive validation with hardware capabilities provided explicitly for the purpose, the system efficiency has been kept high, even with the extra functional workload imposed.

The detection of hardware failures cannot reasonably be placed into any other domain than that of the hardware itself. First, because of the variety of ways in which a single failure can manifest itself and second, because of the logical difficulty of using the suspect operations of the machine itself for diagnosis. The current design relies heavily on the provision of hardware dedicated expressly to concurrent error detection. Viewing the problem in terms of Dijkstra's concepts of structured programming, the hierarchy of abstract machines is distorted if, at the bottom level, the hardware itself cannot be relied upon to obtain repeatable results. Thus, the

hardware-level abstraction has been regarded as yielding only two outcomes for each operation: the correct result, or an error indication. This is not to say that the recovery problem is insignificant, but at least the problem can be constrained into a relationship between the hardware and a bounded recovery software package; this is a vast improvement over the situation where each application program must operate in fear (or ignorance) of the bad things which may happen to it, and is required to provide such survival aids as it can.

The fourth area of design for reliability, detection and containment of manifested software faults, arises only when previous prevention attempts have failed. Yet this problem will inevitably remain a key focus for system designers, because the consequences of such faults remain so severe, even though their likelihood has been decreased by application of the previously described concepts. Again, this is where the combination of special hardware features and a high order language implementation can be of great value. It is the route chosen for this multiprocessor design, and some of the consequences are summarized below.

1.3.2 Software Faults and Reliability

In this section, the key types of software faults are reviewed, and the system design features incorporated to counter them are described.

1.3.2.1 Bounds Violations: Attempts of processes to read or write inappropriate locations in their address spaces have long been a source of serious operational problems. Although some high-level languages, such as FORTRAN, tacitly or expressly foster such a programming style, it is well within the grasp of high-level language implementations to limit this behavior. The bounds violations which may occur include:

a) invalid array indexing (index value does not select a member of the named array)

b) addressing program code instead of data, and vice versa

c) addressing a value not described by the name-component of the address

d) erroneous program branches (entry at other than entry-points, indexed branches with bad index, etc.)

e) I/O operations which read or write beyond the correct area of memory.

The adoption of addressing via descriptors provides the means for efficient validation of indexing which complements the name validation performed by the compilers. Thus, the validity of accesses to named operands is checked at compile-time: that a selected part of an operand array is truly a subset of that array is ensured at execution time. The explicit manipulation of indexing registers by programs whose intention is not stated, and whose validity cannot therefore be checked, is completely avoided.

1.3.2.2 Operand Checking: The variety of applications performed in a computer of any size dictates that, for efficiency, a number of different data types be supported in the hardware. Although often implemented via software, it is desirable to perform certain conversions between data types automatically. If data types are inadvertently mixed, and if no detection-aid is provided, invalid results may arise with no notification of any kind. In the current system, both of these key capabilities have been implemented in the hardware: automatic data-type conversion, and signalling of inappropriate operands. This is achieved by:

a) making different data-types have maximum familial resemblence

b) including type-identification with operands

c) converting to single type, where possible, when operands are brought to the stack prior to their participation in arithmetic or logical operations.

While additional storage is required to denote operand types, offsetting economies result from the elimination of explicit conversion instructions, and the instruction repertoire is simplified by the elimination of varieties of instructions required to operate on varieties of data.

A different kind of data verification is performed by software: validation of Compool versions. When a program is compiled, its use of one or more Compools is signified by directives in the source program. To prevent the logical chaos which would result if the program in execution believed the Compool structure to be different from that of the version actually provided at execution time, a unique identifier is associated with each revision of a Compool (and with every other named segment or file as well). A program compiled with a Compool contains the id of the version utilized; when run-time binding is performed between the physical Compool then present and the requesting process, the version id is compared with that of the present Compool, and an error is signalled if there is disagreement.

By these means, the desired relations between types and templates may be explicitly enforced. This problem is thereby removed from the province of management. While it is clearly possible to achieve correct results via management practices alone, the effort required and the consequences of failure can be circumvented by the outlined method.

1.3.2.3 Process Execution: Software may be incapable of producing correct results if scheduled or executed under inappropriate conditions. For instance, if a process is designed to be performed once per second, but is executed more or less often, its behavior may be unpredictable. Hence the system design must reflect the importance of imposing space and time budgets upon application software, and fulfilling the "contractual commitment" of making the budgeted amount available even if errant processes are present.

The fundamental basis which has been adopted for scheduling is priority. Three general categories are recognized: of highest priority are the processes with real-time response requirements and responsibilities; next are relatively short-execution-time processes related more loosely to real-time; and finally, data-reduction and information management processes of longer duration, having almost no timing constraints. It is anticipated that the high-priority real-time processes will represent a small fraction of the computational load, so that priority alone suffices to signify their urgency; if they formed a larger fraction, interference among these processes could be a significant factor.

At the next level, execution-times are longer, but bounded. The lowest group has the least urgency, and tends to contain processes with extended execution requirements. The fulfillment of budget commitments is tailored to these three levels; at the top, response is paramount, requiring rigid enforcement of time-limits in order that inter-process interference does not exceed expectation for any reason. Such enforcement is provided through the use of a process timer which is part of each processor. Used as an alarm clock, this timer may be set to the budgeted value when a process is assigned a processor, thereby signalling a violation if the value runs out before the process terminates. This device may also be used for time-slicing and accounting.

The management of storage among competing processes in the current MP design is accomplished through a simple, general purpose algorithm which implements virtual memory address space by variable-size segment multiplexing. Primitive calls to executive procedures enable segments needed by processes with real-time response constraints to be loaded into operating memory

in advance of their need, to avoid fetch-delays. Language restrictions normally prevent processes from using storage in excess of predictable amounts. Protection against violations in this area can be implemented in the operating system if it proves necessary, although it presently does not seem to be.

In both storage and execution-time budgeting, it is clearly the responsibility of the managers of the software to recognize the need of avoiding overloads, and to budget accordingly. No load-leveling functions are deliberately specified in the current operating system design, since there is no reason to believe that an algorithm can successfully adjudicate load-clashes resulting from mis-management. While the system does include watch-dog timing of such estimatable durations as the maximum time to execute one instruction and the time-interval over which a process may inhibit interrupts, system participation in the higher-level issues is inappropriate. Only the tools for measuring and enforcement are provided.

At this point, the issue of "graceful degradation" must be raised: one reason for adopting the multiprocessor organization is because it can continue operation at a reduced performance level when a processor fails. The successful utilization of this property requires more than hardware and OS error recovery procedures; as indicated in the previous section, the application workload must be designed to function satisfactorily under the reduced-performance conditions which may be experienced. Flexibility in scheduling must be provided; the rescheduling interval for a repetitive process must be calculated as a function of the number of processors, or the fraction of initial capacity surviving. Otherwise, the inter-process load clash referred to above is likely to result from excessive demands on a partially-functioning system. The management of dynamic memory multiplexing adapts itself exceedingly well to incremental loss of memory capacity. Because the memory management algorithm, under any conditions, is a mechanization of the principle of keeping the most active segments in M2, loss of memory just tends to expel segments somewhat sooner. While this can lead to thrashing if overload is allowed, comparison of process storage estimates with available capacity permits this condition to be sensed, and relieved.

1.3.2.4 <u>The Role of the Compiler</u>: In addition to its primary function of providing a source language in which programs may be easily and lucidly expressed, the compiler plays a key part in the strategy for system dependability. First of all, it forces certain programming conventions to be observed despite forgetfulness or contrary motivation from the programmer.

It achieves this since the compiler designer may dictate precisely how every operation in the source language is to be mapped into the executable instruction level. Wherever it is deemed desirable, additional code can be inserted into the object program, or system-routines may be called, to implement checks of almost arbitrary variety. Because system managers can be assured that certain checks are inexorably performed, the associated problems dealing with undetected violations are eliminated.

Not only is compiler-enforced checking beneficial to security, but it has the added property that the OS/compiler combination is always in effect: thus, checks which have been performed at compile-time can be avoided at run-time, for increased efficiency.

Another aspect of high-order language programming is that the compilers, and not the hardware, can be used to grant selective access to files and even instructions in the machine. Thus, the current system does not have a class of instructions known as "priviledged"; instead, source programs are simply denied the use of the language features that employ them. For example, since it would be undesirable for an application program to execute the instruction which assigns a processor to another process, the compiler would never produce such an instruction for an application program. The compiler is thus seen to insure that permissible operations are performed only under specified constraints, and that non-permissible operations are not performed at all.

1.4 Summary of Report

The remainder of this report has been organized into four chapters, covering the Instruction Architecture, Operating System Design, Fault Tolerant Design, and Hardware Implementation Aspects. Before embarking on brief synopses of these chapters, we present an overview of the basic multiprocessor configuration that was chosen as the subject of development for this study, and provide definitions of some of the more common terms used throughout the report.

1.4.1 Choice of Multiprocessor Configuration

The choice of a basic MP configuration for development to the level of a functional specification was influenced by a number of factors. Perhaps the most obvious initial decisions to be made in any MP design concern the number and character of the processors.

1.4.1.1 Number of Processors: In order to provide a maximum degree of fault tolerance at the least cost of redundant hardware, it was decided to keep the amount of non-redundant special purpose processing hardware to a minimum. As a consequence the MP was specified to consist of a number of identical, interchangeable processing elements which would execute the major processing workload, and a single, more specialized processor to handle I/O processing and a number of other unique functions. These functions include interrupt handling, interprocessor communication control, and the central timer. The executive was specified to be non-dedicated (to any given processor), and its functions are performed by any of the processors. The choice of which processor is made on the basis of status (e.g., by having completed its current assignment), or by reason of its greatest interruptability as determined by the priority of its current process.

The maximum number of processors that the chosen MP should consist of was left somewhat undetermined. It transpired that at the level of design required of this study this information was not of importance: at no stage in the design were decisions made on the basis of "so many" processors. The generality of an n-processor system was always maintained. A value for "n" must eventually be selected in order to proceed to an implementational level, and this value will then place an upper limit on the degree of expandability. The discussions of Chapters 4 and 5 concerning the hardware aspects of design required the number of processors in the configuration to be established in order to demonstrate certain design principles. For this reason a basic MP configuration was chosen to consist of three processors plus one I/O controller. This configuration was chosen, since it represents the simplest which possesses completely all the characteristics (and problems) of the n-processor case. The two processor system which has received the greatest amount of development and operational experience of all configurations, represents a degenerate form of multiprocessor: while certainly exhibiting true concurrency of processes, nevertheless the dual processor allows certain simplifications of executive functions to be made because of the binary number of active elements in the system.

1.4.1.2 Internal Bus: The next most obvious aspect of a multiprocessor configuration is the type of network interconnecting the various modular elements. An initial study of multiprocessor systems [1] had led to a proposal for a single multiplexed data path as the internal bus. The apparent advantages of a single bus were: an easy expandability for adding more elements to the system (e.g., processors, memories, etc.) without major electrical re-design, and as an adjunct in combatting

conflict among the concurrent processes over access to the systems' resources, by allowing only one communication at a time. It was proposed to alleviate the performance required of the bus by lowering the density of data traffic with the use of cache memories local to each processor. Further examination of this proposal showed the cache to be not effective in lowering bus traffic [2]. Furthermore, it was estimated that the behavior of a cache system when executing a HOL instruction stream of much lower anticipated addressing locality than a conventional architecture, would render the cache of even less value to the MP.

The choice of bus was narrowed to either the cross-bar switch, or the dedicated bus with multiple ports into the elements of the system. From an implementational point of view, these two were shown to be of equivalent complexity and parts count. The latter was chosen for its marginal advantage over a cross-bar switch in providing for failure recovery. These factors are fully discussed in Chapters 4 and 5.

1.4.1.3 Role of Secondary Storage: The MP design has been impacted, in both the hardware and software, by the strong effect exerted by the choice of secondary storage technology. The very concept of a memory system hierarchy stems from the inability of any one known technology to meet the speed and capacity requirements of a high-performance computer within the physical constraints of size, weight, and power consumption imposed by a Space Station environment. The management of information transfer between the levels of the hierarchy constitutes a major activity in the multiprocessor, and careful design is necessary to prevent this function from dominating the computational resources of the computer. The current design has, in response to the requirement for feasible implementation with off-the-shelf equipment, chosen to specify the established rotating magnetic drum or disk as the secondary storage medium. Accordingly, the MP system has been designed to accomodate the significant access delays experienced with this kind of device, delays which have historically been the motivation for the concept of multiprogramming, as discussed in Section 1.3.2. Multiprogramming of the individual processors has been adopted as a consequence of this decision, and this has been another cause of executive inefficiency due to the overhead associated with task switching. The adoption of a technology such as the experimental solid state, ferroacoustic, block oriented memory [3] may obviate the necessity for multiprogramming as an expedient to higher throughput, because of its inherent faster access response. However, this and other technologies were eliminated due to commercial unavailability.

1.4.1.4 MP Configuration: The resulting multiprocessor configuration which was adopted as a basis for the work reported in the next four chapters is illustrated in Figure 1. The memory terminology in the figure is used throughout the report, and is defined as follows:

a) M1: Local memory, dedicated to, and only for use by a processor. This is a general term and refers to all aspects of buffer, scratchpad, control and associative memory, required by a processing element. The contents of any M1 storage cell are available only to the processor of which M1 is an intimate component. Only in case of recovery after a P and/or M1 failure are these contents made available to another processor. In this MP design M1 is not, strictly, a member of the memory hierarchy.

b) M2: Operating memory (main memory, or, in popular terms, "core"). Since the discussions in this report are fairly independent of memory technology the terms M2 and M3 were coined in order to refer unambiguously to the specific hierarchical levels they represent. M2 consists of several individual memory modules, all of which are accessible to all processors, including the I/O controller. Each access takes place via a data path dedicated to each processing element, through a port in each M2 module. The basic MP configuration, therefore, requires four ports per M2 module, which, as described in Chapter 4, are actually dually redundant to allow fault recovery. Each module is fourway interleaved, for purposes of speed, access conflict resolution, and fault recovery.

c) M3: Secondary storage (backup or Mass Memory). In this design M3 was not quite technology-independent, as explained above. Being a conventional drum or disk, it was decided to interface this level of the memory hierarchy with the rest of the computer system in the more conventional manner, via an I/O channel. The use of M3 to implement the concept of virtual memory then places the heaviest requirement on the design of the I/O controller and the I/O executive routines, as described in Chapter 3.

As mentioned earlier, several unique functions are gathered together into one, unique module, which is (for convenience) termed the I/O controller (IOC). All interfaces to the outside

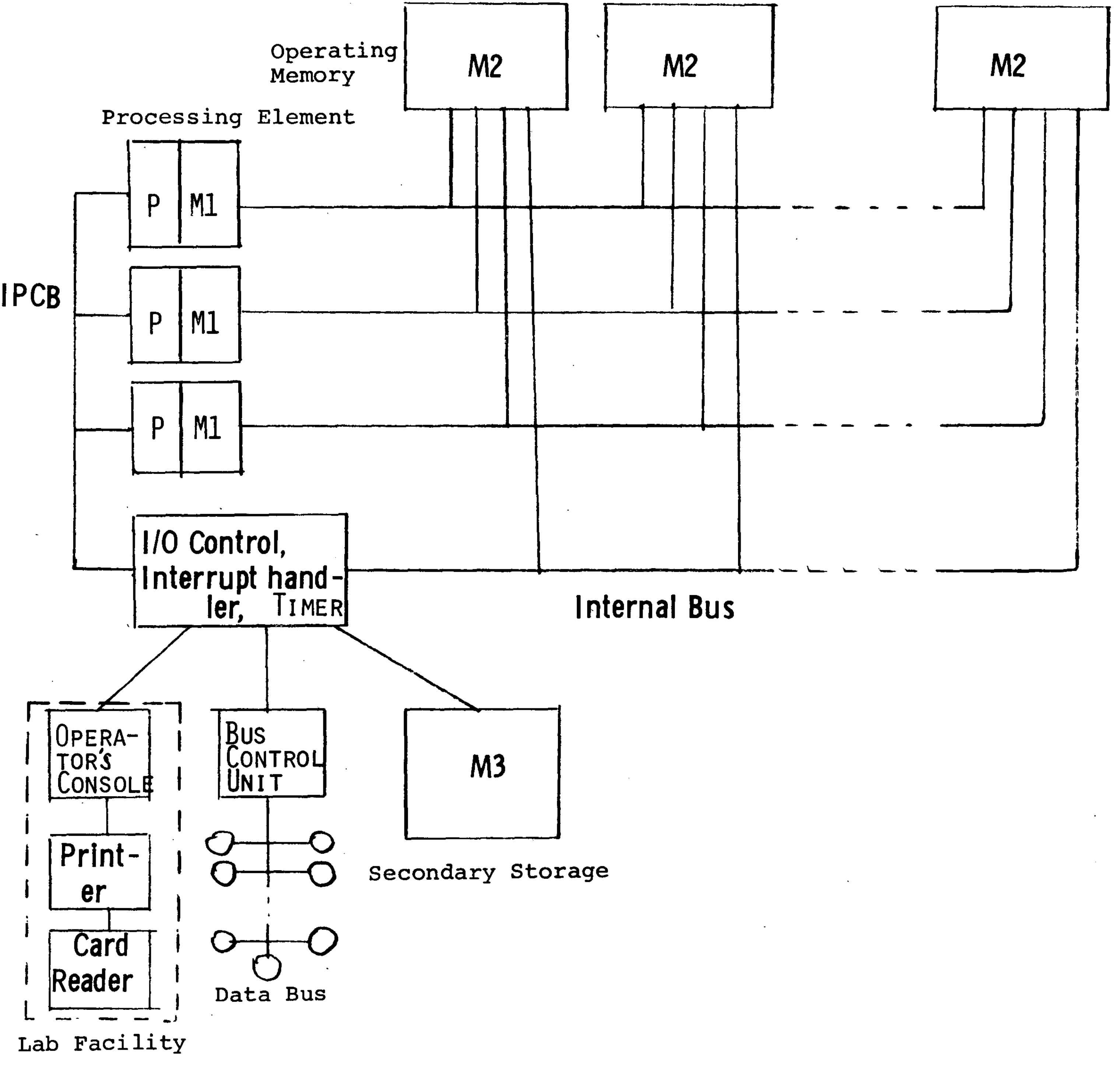


Figure 1: MULTIPROCESSOR BASIC CONFIGURATION

world are handled via the IOC; the current design can accommodate a minimal laboratory set of peripherals, a high speed avionics data bus, and the secondary storage device M3.

Communication between the processing elements of the MP system (the P's and IOC) is handled by a separate interprocessor communication bus (IPCB), for reasons developed in Chapter 5.

(It should be emphasized that the basic configuration of Figure 1 does not indicate the levels of redundancy specified for fault detection and/or recovery. For these details, refer to Chapter 4.)

1.4.2 Summary of Report

The remainder of this report consists of four chapters whose contents are now briefly summarized.

Chapter 2 - Instruction Architecture

The rationale and design methodology for the proposed HOL instruction set are detailed. The inherent advantages of the use of HOL's at the application program level are extended when the machine instruction design complements the HOL. The suitability of a sequentially ordered HOL to the single instruction, single data stream type of processor, and its inherent compactness of expression is stressed. The ability of a HOL to recognize higher level data types results in elimination of type conversion overhead, and allows easy application of dynamic checking for the validity of operator and operand. The various functions of an instruction set are categorized into Data Management (addressing, arithmetic manipulation, logical operations, data field manipulations, name operations), Flow Control, and System Functions (register manipulation, stack control, the multiprocessor environment, interrupts and I/O). The specific requirements of the proposed MP architecture are addressed: data types and widths, the size of address fields and stack depths are established. Finally, the complete instruction set is defined and described.

Chapter 3 - Multiprocessor Executive Design

The basic data structures recognized by the executive are first defined. Then follows a description of the Process State Controller concerning the allowable states and transitions of processes, the activities which influence them, and the executive procedures which are invoked by those activities. This is followed by a description of the interrupt structure, a categorization of the interrupt conditions, and the proposed techniques of interrupt response selection. A large proportion

of Chapter 3 is devoted to the subject of Memory Management. The technique of M2 multiplexing with variable size blocks is described. M2 space is administered as "available" or "in-use", and "in-use" space is further divided into "resident" and "transient". The procedures designed to release, find and make M2 space are described. The response to the absent segment trap, and the special interface with the I/O routines for segment transfers are described. Next follow two sections which explain the handling of names and named entities: the topics covered are name scope, file directories and the Compool structure. Then the functions of I/O are summarized, with details of the data structures and control procedures required to manage the three types of I/O interfaces to the outside world, namely the avionics data bus, the laboratory peripherals, and the disk (or drum). Next is a section describing timing, synchronization and the handling of events. The chapter closes by describing the role of the executive in providing system recovery after the detection of a failure in a major element of the system (i.e., processor, memory or I/O controller).

Chapter 4: Fault Tolerant Aspects of the Multiprocessor

This chapter contains the assumptions, philosophy and the proposed techniques for fault detection and recovery, from a hardware implementational aspect. The basic goal is the instantaneous detection and containment of errors at the source, by hardware, followed by a more leisurely recovery process under the control of the operating system. Basically, this chapter proposes dual redundancy with comparison for the detection of errors within a processor, address and word parity with write verification for error detection in the memory, and triple modular redundancy for the critical elements of the single I/O controller. Techniques for the redundant storage of variable data in M2 to enable system recovery in the event of memory failure are described.

Chapter 5: Implementational Aspects

This chapter discusses the implementational aspects of many of the functional level decisions that were described in chapters 2 through 4. The internal structure of a processing element is described with special regard to the mechanization of the instruction set, the handling of the process stacks, and the interprocessor communication bus. The internal structure of an M2 module is described, including interleaving, data redundancy, timing, and probability for access conflict. Some assessment of overall system performance is given. Finally

the problems of an actual implementation of the MP design are reviewed. Several available computer components are evaluated, and a compromise MP configuration utilizing building blocks from the Burroughs Aerospace Multiprocessor (D-machine) is described.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

References for Chapter 1

1) Miller, J.S., et. al., "Multiprocesosr Computer System Study", Contract NAS9-9763, NASA/Intermetrics, Inc., March 1970.

2) Miller, J.S., "Probability Model for Memory Conflict", Internal Memo #06-71, Contract NAS9-11745, NASA/Intermetrics, Inc., September 1971.

3) Kosmala, A.L., et. al., "Engineering Study for a Mass Memory System", Contract NAS9-9763, NASA/Intermetrics, Inc., August 1970.

2.0

INSTRUCTION ARCHITECTURE

This chapter is composed of four sections. The first three present the rationale and design methodology for an instruction set oriented toward executing more directly the statements of higher order, problem-oriented programming languages. The fourth presents a proposed set of instructions and data structures for the subject multiprocessor.

2.1 Design Rationale and Method for a HOLM

The design of an instruction architecture for a higher order language machine (HOLM) takes several steps which are logically separable. Each of these steps will be discussed in turn.

2.1.1 Desire for a HOL

The consideration of a HOLM presupposes the desirability of requiring programmers to write in a higher order language (HOL). There is a standard set of arguments in favor of HOLs, particularly for large systems projects and applications [1-3]. These include:

a) Ease of communcation

1) The program becomes self-documenting, which reduces the need for separate documentation at different levels of management (e.g., mission description, analytical equations, program description).

2) The ability to communicate allows the analyst who develops the equations to program them. This avoids the inter-personnel misunderstandings inherent between groups of people of differing disciplines.

3) In any large project, a turnover of personnel must be assumed, and maintainability becomes paramount. Not only must different people be able to maintain the program, but they must also be able to easily redesign whole sections as necessary.

b) The HOL is chosen because it is oriented to the problem being solved, resulting in:

1) Fewer errors due to conceptual difficulties and the interaction of different ways of stating a problem.

2) Shortened programming time.

c) The programmer becomes less concerned with the following common machine features and problems:

1) Scaling and precision.

2) Base register allocations.

3) General register considerations.

4) Initialization problems, particularly in loops.

d) The time required to learn the language and the experience required of the programmers is minimized. Transferability, ease of debugging, ease of checkout due to problem oriented modularity and separation from hardware problems are all improved.

2.1.2 Desire to Execute the HOL Efficiently

Efficiency in a computer is measured in terms of time and space. The time of execution or the memory space needed in executing a HOL must not exceed that of the equivalent machine oriented language machine.

There are two different aspects to be considered: compile time and run time. Given a particular machine and a set of user programs, it is the compiler designer who decides to what degree the compiler should try to optimize the user's code for either space or time. Of course, the optimization of run time would cause a corresponding increase in compiler time and compiler memory space.

If, however, we consider two different machines, a HOLM and a MOLM, then the job of compilation on the HOLM is equivalent to assembling on the MOLM. The difference between the compiler and any other user program becomes negligible since the problem of optimizing the translation from HOL to MOL is eliminated (Figure 2.1-1). We therefore need concern ourselves more with how to save space and time in execution than with the compilation process.

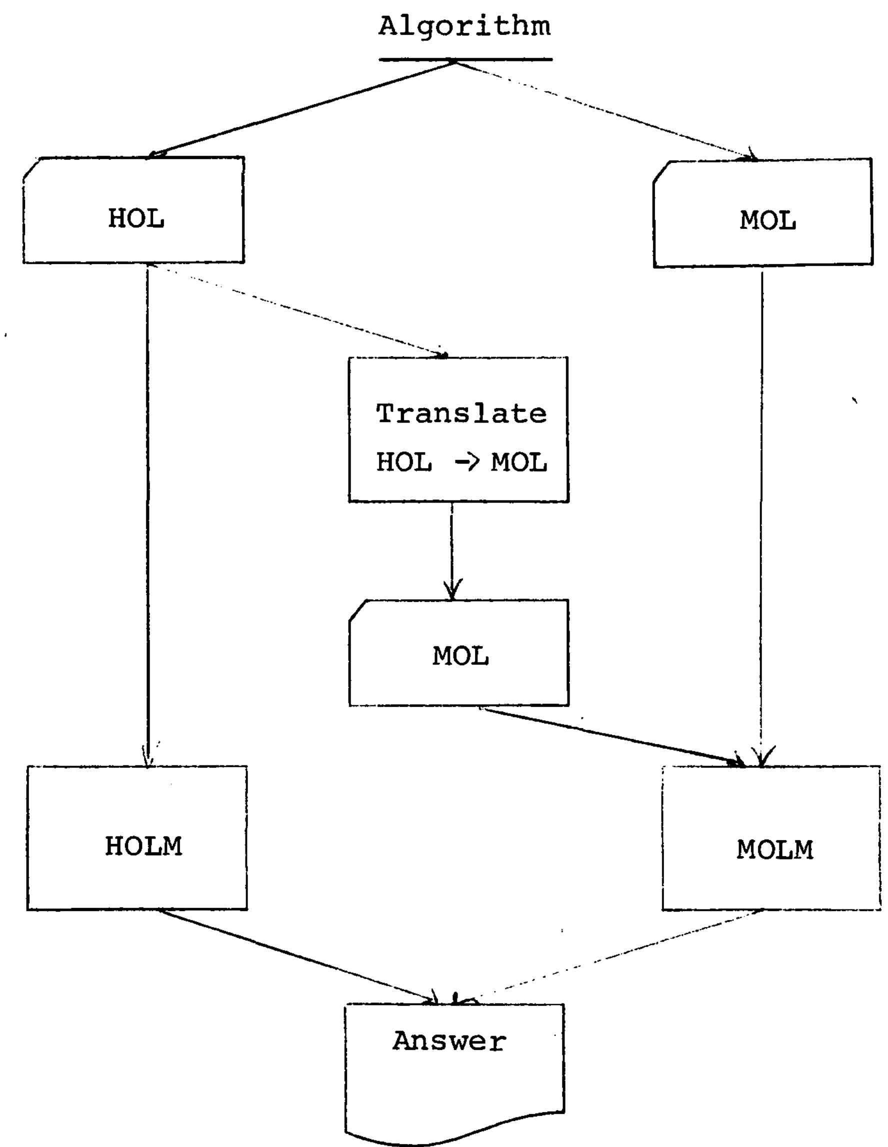


Figure 2.1-1: Translation Steps from Problem to Machine

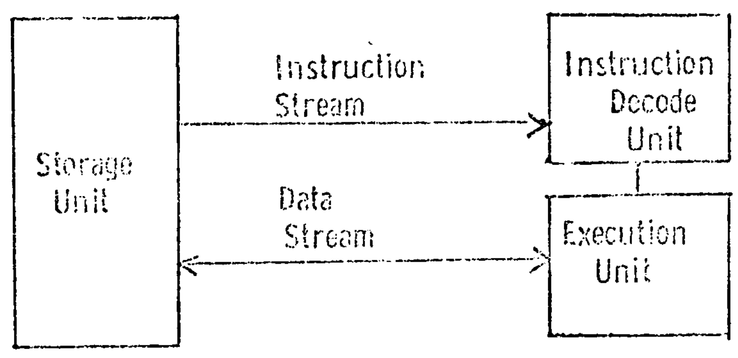
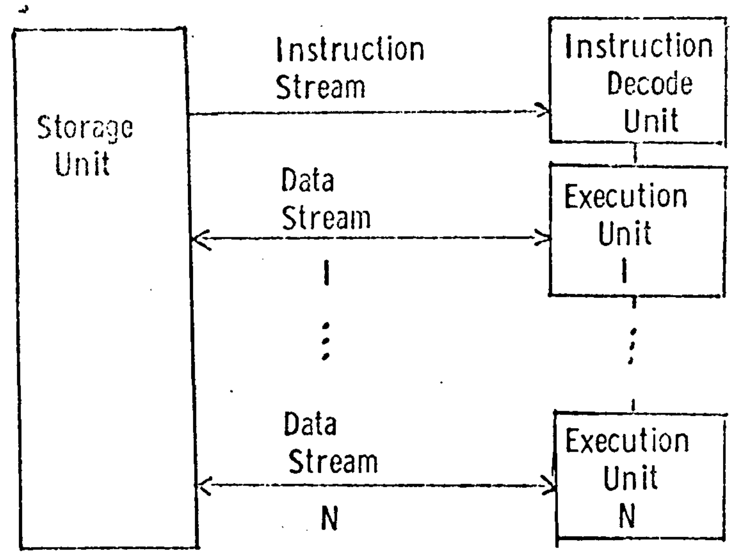


Figure 2.1-2: SISD Organization



Examples of SIMD: ILLIAC IV
Goodyear Array Processor
Bell Lab PEPE

Figure 2.1-3: SIMD Organization

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Most studies which compare a HOL and MOL require the HOL to be translated into the MOL for execution. This, of course, leads to the inefficient coding for the HOL and therefore increases its memory space and execution time over programming in the MOL directly. However, having postulated the execution of the HOL without this translation, the method of time and space comparison must be made between two different machines: a MOLM and a HOLM. These must be programmed with the same algorithm (program) in their respective languages. Unfortunately, this type of comparison has not been made on a large scale, and it is not clear, given the current set of machines in existence with their different timings, memory resources, and other variables, that it could be made meaningfully.

2.1.3 Machine Constraints

Current general computers perform basically a single instruction stream execution of a single data stream (SISD) [5]. Other forms of computers, such as single instruction stream execution of multiple data streams (SIMD) have been developed. These SIMD include the ILLIAC IV, the Goodyear Associative processor and the Bell Laboratories PEPE [6-8]. SIMD is oriented towards a particular type of problem (e.g., radar signal processing) that is not commonly found in most scientific or commercial applications.

In SISD types of machines various hardware mechanisms have been implemented with some form of local reorganization to obtain instruction execution overlap [9]. These methods are limited in their scope by the presence of branching. Such optimization by the hardware is independent of the form of language, and can be considered whenever instructions possessing potential parallelism are presented in a sequential manner. If one were to postulate a certain degree and type of paralelism, then of course, the instructions generated could be optimized before hardware execution. This is in fact done in Sugimoto's PL/I direct processor [10], where parallelism is flagged by the compiler. The hardware then is easily able to take advantage of the inherent parallelism. (Executing beyond a conditional branch is of course limited by the sophistication of the hardware. It is not a static parallelism, but a dynamic, time varying one.)

This leads us to consider a processor that would be able to execute HOL arithmetic statements in parallel. Given an arithmetic statement such as:

A = (B+C)M + 38 D;

it would be able to execute it as a tree structure, as shown in Figure 2.1-4. The actual implementation of such a machine would be, of course, complicated and in general building complicated hardware to take advantage of this would not be cost effective. Examples can be constructed which sequentially take time N to execute, yet in parallel would take only $\log_2 N$*. These are not very common in practice.

If an execution unit were distributive, as in the CDC 6600 [27], then the possibility of executing in parallel becomes more attractive. If the parallelism is across statement borders, the limited parallelism mentioned above is of course increased. Chen [28,29] has proposed even crossing program boundaries to obtain parallelism.

When we talk of sequential machines we should note that not only is the execution of a single stream implied, but also a particular ordering is given to the instructions. Different orderings can be conceived. It is usually implicit when speaking of instruction ordering to treat it, as in English, symbolically from left to right, from the top to the bottom of a page, and physically from the low to the high numbered locations of core (e.g., when non-branching, this ordering is realized simply by incrementing the program counter by one).

Given that a SISD machine must execute its instructions sequentially, the question arises as to what then is the best form in which to present the instruction stream for execution by the HOLM.

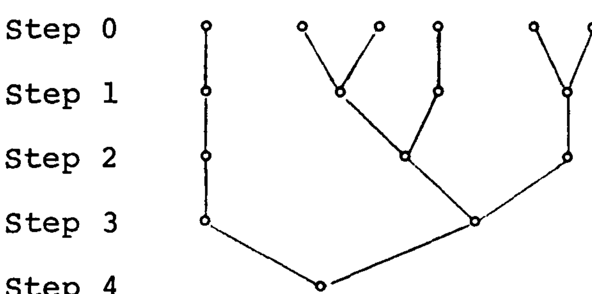
Verbally, the answer is simply to say that the best form is to have each piece of information present as it is needed to be executed, and not to have any more information than is necessary.

In practice this means changing the form of a parenthetical language (which HOLs, patterned after English, usually are) into Polish notation; i.e., parenthesis-free notation. Each of the operands (as many as necessary) is attached to the nearest operator (in a right to left manner) which then operates on the operand(s) and reduces the result to a new operand. Execution of a Polish string takes place from right to left. However, since most Indo-European languages read from left to right, compiler designers usually talk of reverse Polish notation, which is the mirror image of Polish notation, but gives the convenience of left to right execution. Figure

---

* Consider the branches of a tree with each lobe doubling -- hence, n steps implies 2^n operands = N. This implies parallel time of n or $\log_2 N$.

A = (B + C) M + 38 D;



The execution of this statement has an inherent depth of five steps, assuming each operation which can be done in parallel, is done in parallel.

Figure 2.1-4: Example of Statement Structure

2.1-5 shows our tree example in Polish notation. (Hamblin [30] has a full discussion of various Polish forms and implications).

Our use of reverse Polish notation to meet the needs of sequential execution, will be called sequential execution form (SEF) in order to avoid confusion with the use of the term "Polish notation" elsewhere.

In order to put a parenthetical HOL (hereafter it will be assumed any HOL discussed is of parenthetical form) into SEF, it is necessary to re-order the parenthetical input. This reordering of a parenthetical language to a SEF is called parsing. This is the work of the first phase of any compiler. (In parsing a HOL the method of the re-ordering is important. The parse must be able to match the execution tree of Figure 2.1-4 in order to generate code which interprets the SEF correctly.) In summary, current ordered sequential machines require the parsing of a HOL to a SEF.

2.1.4 Semantic Conciseness of the HOL

In this section the meanings and semantics of the SEF operators and operands shall be considered. What does the SEF consist of? How is it possible to save on execution time and memory space of a HOL program?

A simple answer is that if the HOL has been designed as a problem oriented language, then it is descriptive of the "work" to be done. Each statement in the HOL and each syntactical part (except for noise words such as TO of GO TO) of a statement has a meaning in relation to the "work" being done. (The mapping of each syntactical particle into a series of instructions is called the "semantics" in a compiler.) Hence, each statement and/or syntactical unit has precisely the functional meaning that the programmer intended it to have. We will now consider a few of them in detail.

a) Matrix, Vector Operations

If the language contains matrix and vector types, and if it allows standard linear operations such as inner and outer products, cross-products, transposes, etc., then each operation is considered as an entity, and not as if each data type were an "array" of scalars. For example, a matrix transpose changes all non-diagonal elements and must be done in effective "parallel". If it were done sequentially, without temporaries, the operation could not be done, since some element would have a new value before its old value has been read.

Parenthetical Notation:

A = (B + C) M + 38 D;

Polish notation (read right to left)

= A + x 38 D x M + B C

Reverse polish notation (for the convenience of English language speakers)

C B+ M x D 38 x + A =

Execution of reverse polish:

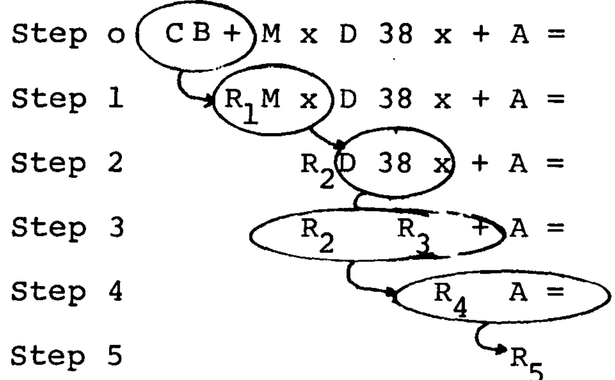


Figure 2.1-5: Conversion of Notations

Analogously an integer type is an integer, and may not be considered a collection of bits that can be manipulated individually.

In general, the semantical particle is the "tightest" unit of information describing the "work" being done (see Figure 2.1-6). It is the most global form of information as presented by the programmer. Any transformation into another language (e.g., MOL) will lose the global context, which in general, is not recoverable.

The semantical particle contains the smallest number of logical operands and operators. This follows from the problem orientation of the HOL and from its globalness, because it is the language in which the problem solution was programmed. If the HOL is problem oriented enough, we see that its semantical units form an upper bound on the number of logical units needed to describe the "work" and hence, under a reasonable mapping to bit space, an upper bound for memory allocation. Identically, it forms the upper bound for execution time since it contains exactly the necessary global information needed for execution.

b) FORTRAN DO loops and HAL DO FORs

The DO FOR statement is an operation that can be considered to contain six operands: the iteration variable, initial value, increment, limit operand, and the loop start and finish address. It should be considered as a whole, containing all the information needed for its execution.

Knuth [12] in his empirical study of FORTRAN programs has found that statically 39% of the DOs contain only one statement and 18.5% involve only two statements. These statistics suggest that it is worthwhile to consider how FORTRAN DO loops are used.

1) Array processing: The manipulation of vectors matrices in FORTRAN makes necessary the use of DO loops. Many languages such as HAL overcome this by having vectors and matrices as arithmetic types. HAL and APL both allow array processing on a statement level for arbitrary arrays of the various data types. For HAL and APL it would seem that the number of DO loops necessary for array processing has been minimized if not completely eliminated. This is the result of their containing more global information [13, p. 292] than FORTRAN, since for arrays, HAL and APL are closely related to the "work".

Assume Vectors of length 3

		Number of Logical Operators	Number of Logical Operands
HAL			
	$X = \overline{Y} \cdot \overline{Z}$	2	3
FORTRAN			
	X = 0.0		
	DO 1 J = 1,3		
1	X = X + Y (J) * Z (J)	5	9
BAL			
	LA R1,3		
	SR R2,R2		
	SER RD0,RD0		
Loop	LE RD2,Y(R2)		
	ME RD2,Z(R2)		
	AER RD0,RD2		
	LA R2,4(R2)		
	BCT R1,Loop		
	STE RD0,X	9	21

Figure 2.1-6: Examples of Semantical Conciseness

2) Table Searching: A very common use of DO loops is to place information into a table and then by using an index of the location in the table, to obtain information in associated tables. For example, storing information into an array under a person's name and then using the relative address of its table location as the index into the associated information tables. This is an example of inefficiency within the HOL itself; the HOL has not been able to specify concisely what it is that it wishes to do.

   With the advent of content addressable memories, a linear search whose time would have been directly proportional to the number of table entries, can now be done in a constant time [14]. DO loops are written to perform linear, binary (logarithmic) or exponential searches on a table with or without hash coding. This use of a DO loop is a function which is not "work" oriented and should be described in the HOL; the programmer should not have to worry about the type of search to be done but only that a search is being done. An example of the type of syntax that could be added to a HOL is given in Figure 2.1-7.

3) Anything else: In general, this could be classified as an in-depth re-iterative arithmetic calculation which would not fall into the array pattern. This, in our sense, is the proper use of DO loops, since it cannot in general be abstracted to a higher level of meaning.

c) Indexing and Subarraying

Data type considerations also lead to HOL concepts not generally expressed in a MOL. Operands have a "type" such as integer, bit, character, or scalar. If a fixed point scalar is considered, then "scaling" becomes a further "type" attribute. With all arithmetic types, consideration of precision becomes important. With arrays of operands, the attributes of rank and dimension are introduced. Usually compilers generate machine code which takes into consideration all of this information. Since the HOL instructions are not oriented to these considerations, especially when protection is involved, inefficiency in code generation develops. In order to make sure an index does not exceed array length, inefficient, continually executed code must be produced for this test. Type attributes become fixed into the generated MOL code.

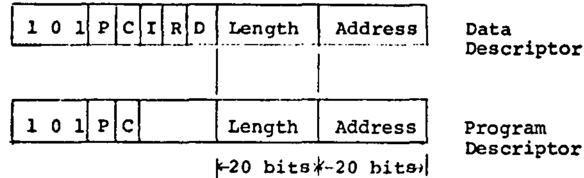
PLACE value INTO table SAVE INDEX IN variable

The "value" is placed into the array "table" with its entry index stored into the "variable". If the table overflows the "variable" could be set to some unattainable value such as zero.

FIND value IN table SAVE INDEX IN variable

This could be used to locate a given "value" in a "table". As above, if the "value" is not present the index could be set to some unattainable value.

Figure 2.1-7: Example of HOL Syntax



P Presence Bit, = 0 → Interrupt to fetch

C Copy Bit, = 1 points to "main" descriptor

I Indexed Bit, = 0 → not yet indexed

R Read-only Bit, = 1 → interrupt if try to write

D Double Precision, = 1 → double precision

Figure 2.1-8: Basic Descriptor words for Burroughs B6500

Several forms of "descriptor" have been developed to help solve these inefficiencies and to allow attributes to become dynamic. Descriptors range from simply insuring that an array's index does not exceed the array dimension, as in early Burroughs machines [15], to specifying "type", each rank's dimension, and initialization information as in the SPLM [17], Sugimoto's PL/I direct processor [18], and a Russian machine [31].

Besides having arithmetic precision implicitly handled in the code generated by a compiler, it is possible to have it done dynamically. For example, the SYMBOL machine [19,30] maintains arithmetic precision on a dynamic basis to a maximum limit specified in a LIMIT register, which is under programmer control. It also has an empirical mode which is only as accurate as the maximum length of the operands involved.

There are many problems of indexing. One is to make sure that the index does not exceed the array dimension. This can be accomplished by including the array dimension in the descriptor (see Figures 2.1-8, 2.1-9, 2.1-10), which value can be considered an operand of an indexed array. It is present when access to the array occurs and the necessary protection test can be accomplished as part of the indexing instruction. A second problem of indexing is the mapping of multiple subscripts into an array. APLM [16] accomplishes this adroitly by considering the mapping as a polynomial with the indices as the arguments (Figure 2.1-9). This leads to both an efficient execution of the process of multiple indexing as a single instruction and the necessary array limit testing.

HAL presents a third problem in indexing since it has the ability to make sub-arrays. HAL must have some mechanism to denote sub-arrays; that is, to specify a collection of variables which do not necessarily have a single contiguous storage location but are a series of contiguous blocks whose location and sizes are a function of the rank and dimensions of the initial array. This problem can be solved by a generalization of Abrams' APLM polynomial mapping, which would not only map into an array, but would give for each rank, starting element address, last element address, and spacing between elements for each rank.

In summary, three semantic areas have been shown in which current MOLs fail to effectively interpret the desires of a problem oriented HOL:

a) Vector and Matrix operations

b) DO loops

c) Data attributes including indexing and precision maintenance.

"Although arrays are stored in the memory, M, of the machine, scalars are not. They appear only in the machine registers, in particular the value stack, and as immediate operands in a code string. An array is divided into two parts. The first is the value array which is a row-major order linearization of the elements of the array. The second part is a descriptor array (DA) for an array, which contains the rank, dimension, and storage mapping function for the array."

@ARR	RC=2	LEN=05
+01	VB=VARR	AB=000
+02	RANK=2	
+03	R(1)=003	D(1)=02
+04	R(2)=002	D(2)=01

LEN : = Length of array

VB : = array base address

AB : = constant used in polynomial mapping

RANK: = rank of the array

R(I): = dimension of rank I

D(I): = value used in polynomial mapping

"The DA contains the coefficients of the storage mapping polynomial, DEL (labelled D(I) here). Recall that for an array ARR, the element ARR[;/L] is located at

VBASE + ABASE ++/DEL x (L-IORG);

This formula is the storage mapping function for any array."

Figure 2.1-9: Descriptor for Abrams' APLM

The following descriptor examples are shown as they appear in the program string and the effect of their execution in the stacks:

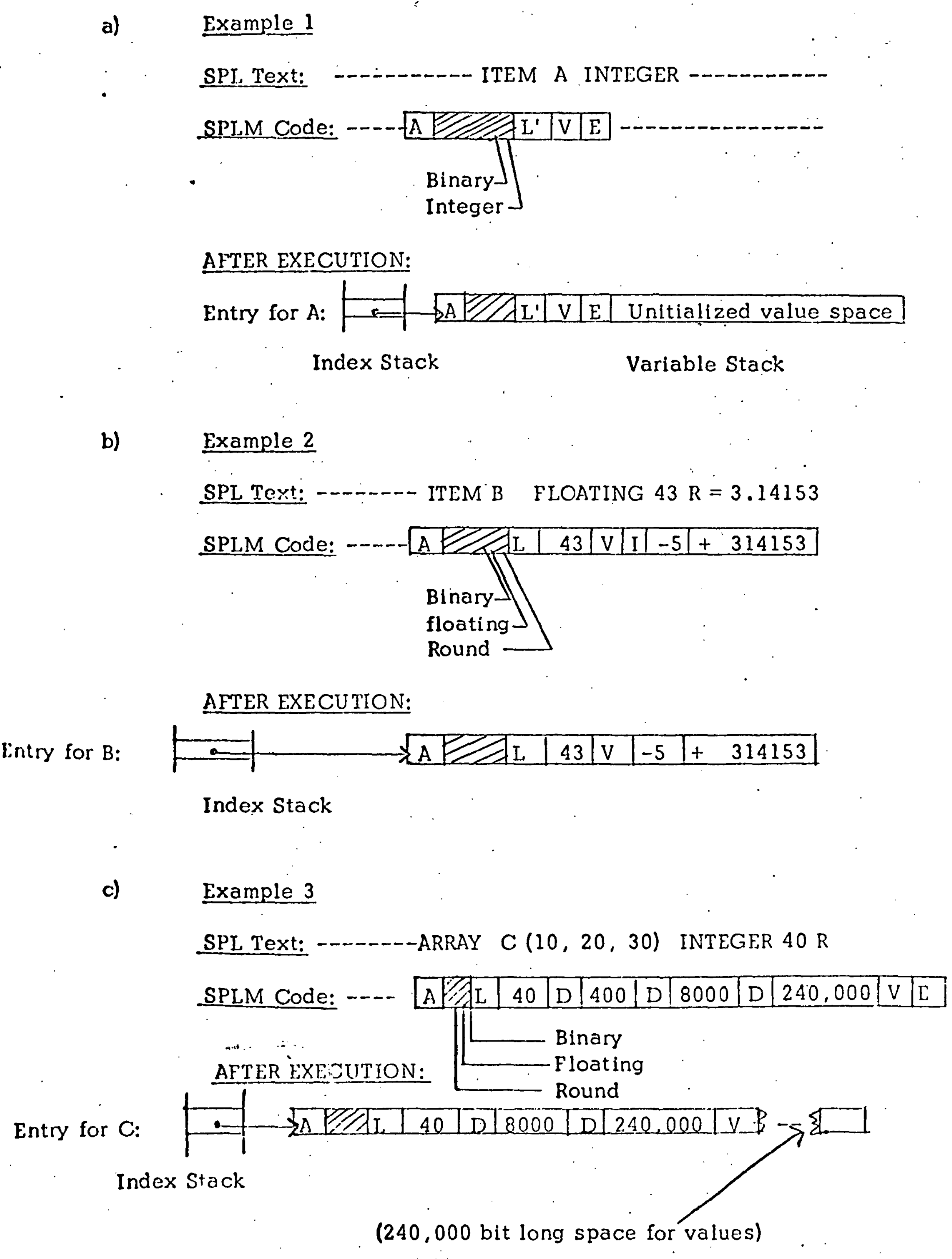


Figure 2.1-10: Descriptor for Keeler's SPLM

In examining these, it is seen that one feature which is desirable but not included in the current HOLs would be a table lookup statement. This can be considered as a failure of the HOL, and not its implementation.

Other HOL statements must also be similarly considered. These include the generation of flow control in IF's and DO CASEs (FORTRAN computed GO TO's). From a HOL viewpoint, the difference between data types as seen in a MOL is not necessarily clear (integer, scalar, bit) and as shall be seen, type conversion is one of the major MOL overheads.

2.1.5 Logical Construction for Execution

This section considers what is meant by executing the HOL's semantic particles on a machine in Sequential Execution Form.

Lawson's paper [21] considers five logically equivalent instruction streams. These are instruction streams with three operands, two operands, one operand, an implicit operand, and finally a tree-structure where each operator node points to its operand nodes. While it should be noted that the interpretation of these operands in Lawson's paper take on a very particular meaning where it would be possible to interpret them differently, they are truly representative. For example, consider the two operand case:

> "There is no explicit indication of where to store the result of an arithmetic operation. This is handled by placing the results in an operand push-down stack(s) for temporary values, which will be illustrated in this section. The use of a value stored in the operand push-down stack is simply denoted in the instruction stream by the presence of a T'. This could actually be represented in the instruction stream by a zero value or some nonaddress value."

The two operand case could have been interpreted differently. For example, always store the result into the indicated location of the first operand and use the top of a stack (or single accumulator) along with the second operand as the arguments of the operator.

Figure 2.1-11 shows an arithmetic statement X = A * B/(C + D), written in each of these five forms. Lawson analyzes these results with four considerations:

$$X = A * B/(C + D) \Rightarrow$$
$$*,\ p \rightarrow A,\ p \rightarrow B,\ p \rightarrow T_0$$
$$+,\ p \rightarrow C,\ p \rightarrow D,\ p \rightarrow T_1$$
$$/,\ p \rightarrow T_0,\ p \rightarrow T_1,\ p \rightarrow T_0$$
$$=,\ p \rightarrow X,\ p \rightarrow T_0,\ \text{null.}$$

THE THREE-ADDRESS INSTRUCTION STREAM

$$X = A * B/(C + D) \Rightarrow$$
$$*,\ p \rightarrow A,\ p \rightarrow B$$
$$+,\ p \rightarrow C,\ p \rightarrow D$$
$$/,\ T',\ T'$$
$$=,\ p \rightarrow X,\ T'.$$

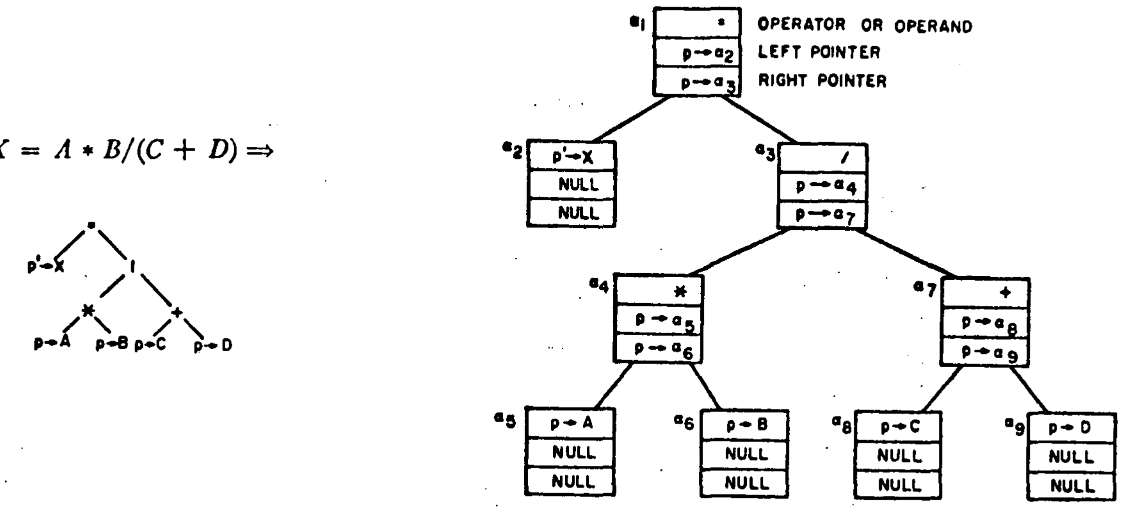
THE TWO-ADDRESS INSTRUCTION STREAM

$$X = A * B/(C + D) \Rightarrow$$
$$F,\ p \rightarrow A$$
$$*,\ p \rightarrow B$$
$$F,\ p \rightarrow C$$
$$+,\ p \rightarrow D$$
$$/,\ T''$$
$$=,\ p \rightarrow X.$$

THE ONE-ADDRESS INSTRUCTION STREAM

$$X = A * B/(C + D) \Rightarrow$$
$$p' \rightarrow X,\ p \rightarrow A,\ p \rightarrow B,\ *,\ p \rightarrow C,\ p \rightarrow D,\ +,\ /,\ =.$$

THE POLISH-NOTATION INSTRUCTION STREAM



A PROGRAM-TREE AS AN INSTRUCTION STREAM

Figure 2.1-11: Various Forms of Instruction Streams

a) <u>Conciseness</u> is defined to be the number of instruction elements required to represent the instruction streams.

Polish (Implicit)	9 x 1 = 9
One-address	12 x 2 = 24
Two-address	12 x 3 = 36
Three-address	16 x 4 = 48
Program-tree	27 x 3 = 81

b) <u>Complexity of interpretation</u> is a measure of both the instruction incrementing size and the number of instruction units directly examined.

He orders the forms in increasing complexity as follows for this attribute:

three-address, one-address,

two-address, Polish, program-tree.

c) <u>Dynamic capability</u> is defined to be a measure of the capability inherent in the machine and its instruction stream for coping with recursive and reentrant programs.

This is found only in the Polish and program-tree forms since they are the only ones which do not depend on any absolute address but only use push-down stacks or pointers.

d) <u>Flexibility</u> is defined to be the ability to easily manipulate the instruction stream form in order to change the order or to delete parts. This is extremely valuable for code optimization. Only the program tree form of instruction stream has this property.

Each of these types of instruction stream is logically equivalent. Each of them is capable of doing exactly the same job: they execute the HOL statement X = A*B/(C+D). Many papers in discussing a HOLM emphasize arithmetic expressions and their execution without considering other features of the HOL. Having considered DO loops as an operator with six operands lends an insight into arithmetic statements which is usually overlooked. Arithmetic statements of the form:

A+B+C

when translated into reverse Polish look like:

AB+C+ or ABC++

yet they could be considered:

$ABC+_{(3)}$

where $+_{(3)}$ is a triadic operator [18]. It was previously stated that the semantic particle was the most global information available -- but of course, as in table searches, this is true only to the degree that the HOL reflects the "work". The addition of three operands is truly one event, even if the notation makes it appear as two dyadic operations. Similarly, n-array operators could be made explicit.

These results indicate that of all the logically equivalent forms, the Polish notation is the most desirable for execution efficiency (it is the most concise and logically non-redundant) and also enjoys dynamic capabilities which are easily implemented by a machine.

This Polish form which is desired for execution may not be the form which is desired as the intermediate language of a compiler. The program tree form helps with optimization since each operator node points to its operands and therefore allows easy manipulation [21]. With the Polish form it is difficult to recognize the operator's relation to its operands if these operands are intermediate results; therefore it is difficult to use the Polish form during optimization.

HOL program control statements, such as DO's, IF's, and GO TO's, can be imbedded into the execution sequence of the HOL instructions. This is normally done in ordered sequential machines. It is also possible to consider that these <u>are</u> "control" functions on the normal instruction stream and therefore are to be executed as a separate set of information. This is what Sugimoto's PL/I machine does.

2.1.6 Mapping into Physical Space

The final step in designing a HOLM is the process of mapping the logically constructed semantic particles into real

bits, and to fully design the flow within the hardware. This mapping into bits must realize the space compactness that the logical conciseness would indicate.

In considering the physical mapping, the static and dynamic behaviors of instruction sets become important. The most complete statistic of a HOL (FORTRAN) to be found is the one by Knuth [12]. Quite a few published papers exist on MOL statistics [24,33,36-38]. The comparison of a HOLM and a MOLM execution of the same problem* simply does not exist in any meaningful way. The reason is simply that HOLMs do not exist in any abundance and therefore the number of variables is too great to allow intelligent comparison.

Figure 2.1-12 is indicative of MOL statistics. It shows the MOL instructions of the IBM 360 generated as the output of a program written in the HOL of XPL. It is interesting to note that there were 8352 L (loads) and 3074 LH (load half-words), yet there was only one LPR (load positive register) and one OI (OR immediate) and none of quite a few instructions. These statistics are, of course, biased in that they do not show the distributions of a normal assembly language programmer, but are the output of a compiler. Also the addressing scheme used in XPL is a very particular one, not general, and hence influences the instruction statistics. Instructions used for maintaining addresses play a major role in the IBM 360.

Yet Figure 2.1-12 does show several interesting facts in common with most MOL statistics:

a) The number of SLL (Shift left logical) instructions is high due to the need to adjust indexing register to appropriate byte boundaries depending on the size of the operand (double word, word, half word, byte).

b) The number of loads is enormous for two reasons:

   1) The addressing scheme for XPL demands a load of an address constant before a branch can occur (in the general case), and

   2) Due to the IBM 360 having 16 general registers and the desire to stay within them in order to "save execution time", the registers must be manipulated; in particular, they must be loaded from memory.

---

* This should not be confused with the execution of a given problem written in a HOL, where the HOL has been then translated to a MOL.

INSTRUCTION FREQUENCIES:

BALR	81	BAL	1084
BCTR	3	BC	3143
BCR	299	LH	3094
LPR	1	ST	2457
LTR	55	N	363
LCR	33	O	43
NR	102	X	51
OR	165	L	8352
XR	8	C	837
LR	100	A	427
CR	360	S	252
AR	2194	M	8
SR	2132	D	17
MR	7	AL	2
DR	2	SL	5
ALR	15	SRL	117
SLR	11	SLL	546
STH	1600	SRA	8
LA	1511	SRDA	18
STC	950	STM	79
IC	1254	TM	54
EX	11	DI	1
		LM	80

HAL PASS 1, SOURCE.77
28 April 1971

Figure 2.1-12: MOL Instruction Frequency

c) The absence of BXLE (Branch on index low or equal) and BXH (Branch on index high) instructions which are very powerful looping instructions. Although BXH and BXLE are "powerful" instructions for looping, the overhead for their use, both in system demands for registers, and in a MOL programmer being able to be set them up, is generally too high.

d) The large number of AR (add register) instructions comes about through the particular indexing scheme XPL used. This scheme insures that one register is needed per HOL operand.

e) As in any machine, many common functions are not necessarily thought of when instruction sets are designed. Byte manipulation in a general register, requires a SR (substract register) before an IC (insert character). Similarly LH (load half-word) is not logical, but propagates its sign which many times has to be "anded" off.

Church [24] states (see Figure 2.1-13):

> "In instruction occurrence we found arithmetic 8.3 percent and jumps 12 percent. What are we doing with the rest of the commands? Obviously, we need the "Data Moves" function, but do flow charts call for anything near 40 percent? And what of the transfers? My flow charts do not call for anything near 23 percent of the problem to be involved in transferring."

His solutions to the problems that he proposes deal with a) excessive editing, and b) fixed length instructions. Excessive editing is felt in both the boundary problem (causing shifting) and in the overhead of loading and storing registers. All of this overhead is only used to please the computer; it is completely unrelated to solving the problems. Church recommends addressing the bit level, ignoring all address boundaries in order to lessen excessive editing.

This could be expanded to avoid any type conversions. By having integers as unnormalized floating point variables, the overhead of conversions both in speed and concern with over/under flow is removed (e.g., Burroughs floating point [39, p. 2-10]).

Variable length commands also allow space compactification. This "variable length" can be realized in several fashions.

Instructions	Percent	Total
Arithmetic	8.3	1,146
Data Moves	39.4	5,437
Logic	2.7	372
Shifts	2.3	312
Transfers	23.9	3,303
Jumps	12.0	1,655
I/O	0.7	99
Miscellaneous	10.7	1,480
		13,804

TABLE I—Instruction Occurrence by Instruction Class

Instructions	Percent	Total	Program Monitor	Navigation Program	Track Correlation Program
Arithmetic	10.5	100,206	177	12,370	87,659
Data Moves	38.4	369,645	6,035	21,228	342,382
Logic	1.4	14,064	66	3,160	10,838
Shifts	1.3	12,642	1	1,490	11,151
Transfer	26.6	256,589	4,172	18,601	233,816
Jumps	17.1	165,121	3,893	8,668	152,560
I/O	0.1	803	97	706	0
Miscellaneous	4.6	44,337	410	4,780	39,197
Total	100.0	963,457	14,851	71,003	877,603

TABLE II—Instruction Execution by Instruction Class

From Church [24]

Figure 2.1-13: Examples of Instruction Execution Frequency

a) Variable length operands depending on their addressing properties. (e.g., relative addressing, local addressing with reference to an implied stack, or system wide addressing)

b) Variable number of operands (e.g., A+B+C considered as $CBA+_{(3)}$)

c) Variable length operation codes depending on frequency of use (static and/or dynamic frequency).

Church presents several benchmarks showing the saving on the Litton computers involving these concepts. There is a dramatic saving in a number of instructions, memory size, and in the actual execution time.

The experience of branching and jumping was even higher on the Apollo Guidance Computer. It was considered to be involved in one out of every five instructions [32, p 154; 33].

The importance of branching frequency in the design of large systems should be emphasized. If a paging environment is to be efficient, it is necessary that instruction sequences be executed within a page and without branching to another page. This points out that not only the branching frequency, but also the distribution of the branching distances is important. Burnett [34] has shown how harmful branching is to the expected locality of such paged systems. The physical length of a HOL instruction sequence may or may not exceed that of the physical length of an equivalent MOL instruction sequence; however, the HOL sequence certainly does not introduce any branching except that which is called for in the HOL procedure. It is an open question if the HOL sequence would have more or less locality, but the total system execution in either case would be better than that of the MOLM since the HOLM is semantically compact.

The basic problem behind all MOL statistics is that the MOL is simply not oriented towards the problem being solved, but is concerned about the "machine" and its needs.

Even with a small rise towards problem orientation, there can be considerable savings. A small program [26] was written for the AGC in both the AGC basic language and in its interpretive language. The MOL program took 250 words while the interpreter took 85 words. Admittedly, it was a biased program which was not MOL oriented. But this is precisely the point: the problems to be solved are HOL oriented, not MOL oriented.

The statistics of a HOL (FORTRAN) indicate some other interesting facts ([12] and Figure 2.1-14).

Assignment statements were 51% of the executable statements statically and 67% dynamically. Sixty-eight percent of these static assignment statements were of the form A=B. Twelve and a half percent of the assignment statements were of the form A=A op α, that is, removing the A=B forms, about 35% were A=A op α. Furthermore, 40% of the additions were of the form of A=A+1.

These statistics would suggest that there should be feedback to the logical design from the consideration of "real" programs.

Even though it was previously maintained that the best information possible was given by the user, this can be bettered since the HOL is never completely oriented towards problems. (This is why theorems are proved: a theorem provides a handy summation of arguments and results which can be applied to problems without reverting back to the basics.) If Polish form were used, A=B would be implemented as BA= where B and the address of A would be "stacked" before the operator = was examined. Due to the common occurence of A=B it might be more efficient to have an: =AB, where the discovery of the operator "=" indicates that it will use the two operands immediately following. Similarly, it might seem reasonable to have specific unary operators to increment and decrement. Polish form would demand that A+1 be put into the form A1+, but due to its common occurrence, it could save space and execution time to have: A incr. (Similarly, of course, depending on cost considerations and frequency of occurrence, 2*X, X/2, etc.)

DO loops are just as interesting (Figure 2.1-15). Ninety-five percent of the DO loop statements have the default increment of 1. Thirty-seven percent of all DO loops enclose only one statement. Fifty-three and a half percent of all DO loops are imbedded only to a depth of 1.

Kerner's FORTRAN machine [11] generated DO loop instructions as shown in Figure 2.1-15. Knuth's statistics suggest that only three bits may be necessary (less than eight word distant) instead of such a big "Last Address" field.

Indices also generate expected, but interesting, statistics. In the 166,599 appearances of variables, Knuth found:

Table 1. Distribution of statement types.

	Lockheed		Stanford	
	Number	Percent *	Number	Percent *
Assignment	78435	41	4869	51
IF	27967 **	14.5 **	816 **	8.5 **
GOTO	24942	13	777	8
CALL	15125	8	339	4
CONTINUE	9165	5	309	3
WRITE	7795	4	508	5
FORMAT	7685	4	380	4
DO	7476	4	457	5
DATA	4468	2	28	.3
RETURN	3639	2	186	2
DIMENSION	3492	2	141	1.5
COMMON	2908	1.5	263	3
END	2565	1	121	1
BUFFER	2501	1	0	0
SUBROUTINE	2001	1	93	1
REWIND	1724	1	6	-
EQUIVALENCE	1382	.7	113	1
ENDFILE	765	.4	2	-
INTEGER	657	.3	34	.3
READ	586	.3	92	1
ENCODE	583	.3	0	-
DECODE	557	.3	0	-
PRINT	345	.2	5	-
ENTRY	279	.1	15	.2
STOP	190	.1	11	.1
LOGICAL	170	.1	9	.1
REAL	147	.1	3	-
IDENT	106	.1	0	-
DOUBLE	3	-	99	1
OVERLAY	82	-	0	-
PAUSE	57	-	6	.1
ASSIGN	57	-	4	-
PUNCH	52	-	5	.1
EXTERNAL	23	-	1	-
IMPLICIT	0	-	16	1.5
COMPLEX	6	-	0	-
NAMELIST	5	-	0	-
BLOCKDATA	1	-	2	-
INPUT	0	-	0	-
OUTPUT	0	-	0	-
COMMENT	52924	(28)	1090	(11)
CONTINUATION	13709	(7)	636	(7)

* Percent of total number of statements excluding comments and continuation cards.

** The construction 'IF ( ) statement' counts as an IF as well as a statement, so the total is more than 100%.

Table 2. Distribution of executable statements.

	Static	(percent)	Dynamic
Assignment	51		67
IF	10		11
GO TO	9		9
DO	9		3
CALL	5		3
WRITE	5		1
CONTINUE	4		7
RETURN	4		3
READ	2		0
STOP	1		0

Knuth

Figure 2.1-14: Statistics of FORTRAN Usage

58.0% were unindexed

30.5% had one index

9.8% had two indices

1.0% had three indices

0.2% had four indices

The percentage of unindexed and single indexed variables indicate how important it is, in order to save space, to have variable length descriptors dependent on the size of the array's rank.

These Fortran statistics allow good quantitative judgements on qualitatively held feelings. But each HOL is different and its statistics will undoubtedly vary significantly. Due to the fact that HAL is able to directly perform linear algebra (vector and matrix operations) the usage of subscripts with these operations will be less than in FORTRAN. Similarly simple array processing is directly handled in HAL; this too will lessen the use of subscripts. These two effects also lessen the number of DO FOR's needed in a HAL program.

On the other hand, subscripts are used in HAL for both BIT and CHARACTER string extractions. This will add to the appearance of subscripts. Therefore, it is apparent that an optimal bit mapping for HAL cannot be quantitatively done from Knuth's FORTRAN statistics.

The 7933 DO loops were further investigated to determine their length and depth of nesting; about 95% of the DO statements used the default increment of 1 . Most DO loops were quite short, involving only one or two statements:

Length	1	2	3	4	5	> 5
Number	3046	1467	758	576	1043	1043
Percent	39	18.5	9.5	7	13	13

The depth of DO nesting was subject to considerable variation; the following totals were obtained:

Depth	1	2	3	4	5	> 5
Number	4211	1853	1194	437	118	120
Percent	53.5	23	15	5.5	1.5	1.5

Knuth

	3	2	1	1	5	6
DO	0 1 0	S_1	M_3	M_1	M_2	LAST ADDRESS

M_1 = index start value of 1 or 2

M_2 = end of loop value less than 32

S_1 → use M1 and/or M2 or address that follows to get value

M_3 = increment value of 1, 0 → address follows to get value

Last Address → relative address, 0 → end address follows.

Kerner, et al.

Figure 2.1-15: FORTRAN DO Loop Statistics

2.2 Instruction Functions

In any form of instruction architecture there are certain categories of functions that are essential to the operation of the computer, although within each category the particular instructions may vary both in form and meaning. In order to gain both an insight and an appreciation into the variability of design within each category, a comparison of several standard machines with different instruction architectures is given.

2.2.1 Data Management

The ultimate goal of any algorithmic process is computation upon a data base. The description of a problem involves the explicit operations that must be performed on the data which represents the quantized parameters of the problem. These manipulations usually take the following forms:

a) Addressing the data base. Current general machines such as the IBM 360 and Univac 1108 address operands either explicitly, in the "operand" field of an operator instruction, or use a "load" instruction which obtains the desired operand and then holds it in one of a small group of registers. This enables the manipulation of the data without further main memory references.

The Burroughs B6500, which is a Polish-oriented machine, in effect does the same thing. But instead of allowing explicit operand addresses with each operator, operators implicitly reference the top of a push-down stack for the necessary operand. In order to place explicit operands into the stack, an instruction is used which indicates whether the program string entity is an operand. But since all operands are indicated in this way, only two bits are used to indicate the "load stack" operation. Figure 2.2-1 illustrates and defines the meaning of these two bits.

When a manipulated value is to be placed into an operand location, a general machine uses a "store" instruction with an address operand field. In the B6500 the address for a "store" operation is supplied from the stack. It is interesting to note that not all Polish machines operate in this fashion. Some use an explicit address field with the store operator [40]. Regarded from a Polish string point of view, the store operator has become an infix operator whose (second) operand immediately follows. This has the advantage

a	0	entity is an "operand"
	1	entity is any other "operator"

If a = 0, then

b	0	place value of "operand" on top of stack
	1	place address of "operand" on top of stack

In order for the machine to discover what kind of "entity" is being processed, there must be a unique bit representation. The B6500 uses the first bit of each program string "entity" to be a meta-operator indicating if the "entity" is an "operator" or an "operand".

Figure 2.2-1: Burroughs Operand Identification

of not requiring the intermediate stacking of an address into the pushdown stack. It also eliminates the need for two bits for a Polish "operand load" operator.

The advantage of a pushdown stack in addressing is realized in two ways:

1) A savings in memory utilization, since addresses become implicit. Although this does not necessarily save the length of the instruction string when all operands are reflected with an explicit data definition, it does save the need for temporary entities.

2) A savings resulting from the fact that the pushdown stack reflects the dynamic characteristics of the algorithms. A general register scheme reflects only the static knowledge of a compiler, which is of necessity, faulty. It is interesting to note that Stone [41] proposed "A Pipeline Pushdown Stack Computer" which in effect reassigns the general registers statically assigned by a compiler to a dynamic set during execution.

b) Arithmetic Manipulation: Most general machines recognize two classes of arithmetic data: integers and floating point. Often the floating point can be specified in two different precisions, and similarly the integers can be of different sizes. (For example, the IBM 360 recognizes single and double precision floating points, and full and half word sized integers.) A standard set of operations can be performed on these arithmetic entities: addition, subtraction, multiplication, division, negate operand. Other operations are sometimes incorporated such as: absolute, "SGN" function, exponential, maximum, and minimum; and currently various matrix and vector operations are being considered to be implemented along with general array operations [40,48].

In instruction architecture design there is a continual tradeoff as to the amount of information which is provided by the "operators", and that which is provided by the "operand". Descriptors can be considered either as part of the operator (a "sub-operator"), or as the operand which is being acted upon [49]. The descriptor then becomes the primary focus for the tradeoff as to whether the operator or the data should indicate where the "explicit" data information is to be found.

Descriptors (see Figure 2.2-2) can be thought of as op-code modifiers indicating, for example, that an addition is to be of scalar, array, or matrix types. One could also consider a descriptor to be the actual data object being manipulated by the operator. The descriptor then becomes a mapping from the logical domain into the reality of the physical machine, which is built with linearly addressed storage.

The IBM 360 has entirely different sets of instructions for processing integers, floating point and decimal variables. Similarly, the size of the integer involved or the precision of the floating point is explicit in the operator (for example, multiply can take any of the following forms: M, MR, MH, MD, MDR, ME, MER, MP, MXR, MXD, MXDR.) The B6500 design on the other hand offers generalized arithmetic operations, but interestingly enough retains the explicit MULX (multiply extended) instruction to change single precision operands into a double precision result. For integer operands the B6500 also possesses the operator, IDIV and RDIV, which give the integer or fractional result of an integer division. The generality comes about because of two different properties of the B6500:

1) Integers are a subset of floating point. There is thus no need for explicit type conversion between the two.

2) The descriptor contains the precision of the data. One bit of the descriptor indicates if the data is single or double precision.

In addition to this information in the descriptor, every memory cell within the B6500 contains a three bit tag field which indicates what kind of entity it contains (e.g., descriptor, operand, program control word, indirect reference word, etc.). This tag field also indicates if the operand is single or double precision, which enables the appropriate pushdown stack operation to be automatically performed.

This generality can be carried farther with an identical operator, for array processing. That is, if the descriptor indicates that the operand is an array, the operation can be done on each element of the array (e.g., array addition, element by element).

c) Logical Operators. Binary operations occur in several entirely different ways. Operators involving binary

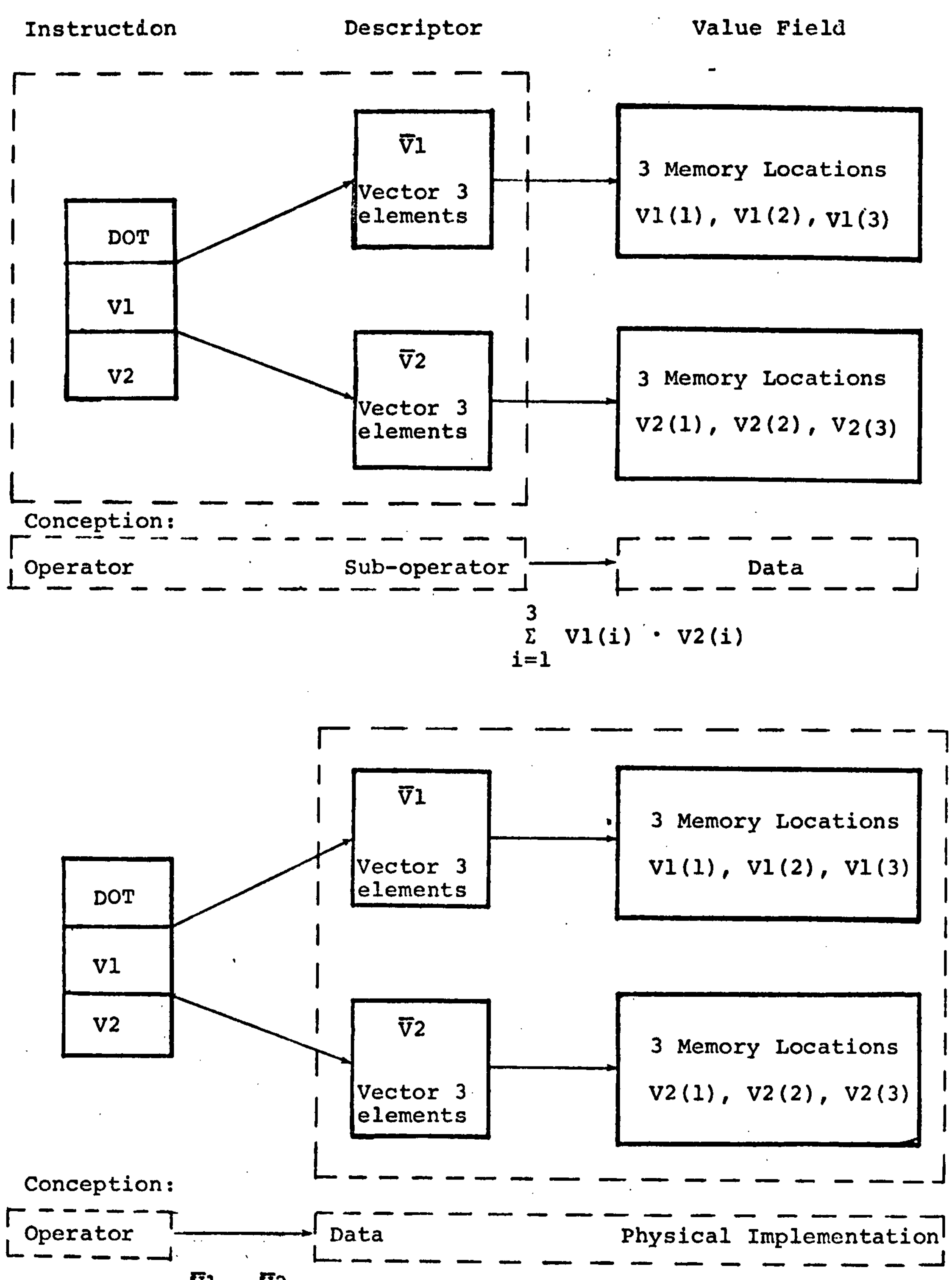


Figure 2.2-2: Two Concepts of Descriptors

data usually are some subset of the standard: and, or, not, exclusive or, equivalence.

Closely connected to these are those operations which, instead of working on a word, affect an individual bit. These include the B6500 bit set and reset instructions.

Once the binary form of data is recognized, a whole set of instructions is usually formulated to take advantage of this knowledge and its relationship to the computer's architecture. These include the standard logical shift, algebraic shift, cycle, and field isolate instructions.

Only explicit knowledge of the bit construction of the manipulated data makes these instructions useful in themselves. An algebraic shift is indeed a multiply or divide, positive or negative, by factors of two. When writing in a HOL such as HAL, bit string manipulations are carried out with respect to a logical entity regardless of the true physical implementation. To extract a series of bits from binary variables in HAL one merely subscripts and does not rely (or care) about physical implementation. Realistically the best compromise is probably to be found in those instructions which are able to extract binary fields from within a word but which do not depend upon algebraic characteristics. Keeler's SPLM [17] relies totally upon descriptor knowledge in order to manipulate binary fields; this is, of course, an extreme.

The final use for binary information is found in conditional information. These include the six dyadic relations: less than, greater than, less than or equal, greater than or equal, equal, not equal; and various monadic functions such as: positive, negative, zero. As with arithmetic manipulations, information can be embedded either in the operator or the operand. In the IBM 360, the compare instructions are: C, CR, CH, CL, CLR, CLI, CP, CD, CDR, CE, CER. These enable comparisons of: character strings, integers (algebraically), integers (logically), decimal, single precision floating point, double precision floating point.

The B6500 has no need for explicit integer or floating point comparisons, since integers are a subset of floating point. It does, however, implement a special "equal" to provide the logical equal in a bit by bit fashion in addition to algebraic equals. Character information is treated separately, and has its own set of instructions.

It is interesting to consider that these comparisons result in a truth function with a value of "true" or "false". Most implementations have identified true with one, and false with zero. This facilitates the logical operations "and", "or", etc., as commonly defined, and usually leads to the equivalence of "true" and the integer "one". This is mere convention, and should not be implied as an absolute identity.

d) Data "Field" Manipulation. The existence of different "types" of data structures implies the need for conversion and manipulation. It is not necessary to differentiate arithmetic data from character data. These could conceivably be the same. Character data is used for communication between the outside world and the computer system. The I/O of a system which has a human interface must be intelligible to the human. In order to provide enough symbols for a ready comprehension, 4, 6, or 8 bits are used to encode a character. When numeric problems are scheduled, this form of encoding is extremely wasteful and leads to an "internal" numeric form for the computer. When great range in value is called for, floating point forms are generally considered.

Because algorithms commonly involve integer computation, and because integer computation in the past was faster and took less memory, separate integer types are normally defined. "State" information (e.g., true and false) requires binary representations to be introduced.

Very few machine implementations contain a complete set of operators for conversion from each type to the other. The IBM 360 has been considered inadequate because it has basically only CVB (convert to binary) and CVD (convert to decimal) instruction, although there are further instructions for character to character conversion (PACK, UNPK, EDIT, EDMK, TR, TRT).

Besides conversions between types, there is need for conversions of varying forms within a type. For example, creating single precision form double precision, either by rounding or truncating. On the B6500 since integer is a subset of floating point, one needs an "integerize" operator.

If the types of data in the machine are limited to character, floating point, integer and bit, then a complete repertoire of conversions would be provided by: character to scalar, scalar to character, scalar to integer, bit to integer. This limited set would suffice because

1) integer is a subset of scalar

2) there exists an explicit rule for mapping bit to integer

3) integers can be considered as bit strings.

The SPLM [17] allows automatic type conversion in the process of assignment. The types of the respective operands are, of course, kept in the descriptor.

If a machine is built for much character string handling, then explcit operators for performing these functions (e.g., concatenate and substring) become important. The B6500 descriptor cannot encode the operand as a "character string". This information is explicit in the use of an operator. Here is a good example of a tradeoff between using bit information in the descriptor to encode "type", or to allowing the operator to be the indicator. Since the overlap of "operators" between arithmetic type and character type is small, there is little ambiguity in what is to be done. What ambiguity does exist is resolved by separate operators. The gain is a reduction in the size of the descriptor field and the only real loss is less protection against the misuse of an operator.

e) Name Operations. Inherent in the manipulation of data values is the process of indexing, of choosing an element of an array. This is one of the prime reasons for the existence of integer arithmetic in algorithms. Most general machines accomplish the process of selecting an element of an array by use of "index" registers. This in turn leads to the necessity for explicit operations for the sole purpose of manipulating the index registers (one must load, store, possibly increment or decrement, test, or jump with respect to them). The allocation of index registers raises the same problem as that of the allocation of general registers: the static allocation at compile time, in general, does not correspond to optimal usage at execution time. With Polish-oriented machines, the indexing of arrays is usually done off the pushdown stack, as are the other operations. The IBM 360 general registers are regarded equivalently as index registers and accumulators for reasons of implementational convenience; besides, the corresponding computations do not demand a differentiation. The use of the stack for indexing can be an explicit one index-rank instruction at a time, as in the B6500, or a completely automatic process initiated when an operand is stacked, as in the SPLM.

Sub-arrays can be handled implicitly by the "operator", or if an infrequent occurrence, more properly by an explicit combination of element operators and automatic loop control.

If arrays are to be handled explicitly, common "small" entities such as 3-vector and 3x3-matrices would probably be the only choice. The temporary storage for arrays (even in stacks) becomes a new problem since, if the top part of the stack is implemented in "fast" registers, it would be unwise to force the bottom of the stack into slow core, only to bring it back again after a short array calculation. These cases indicate that separate vector and matrix stacks could be potentially useful [40].

2.2.2 Flow Control

Of importance in writing algorithms for a computer is the specification of the dynamic flow control through the algorithms. Computers became more than adding machines when they developed the ability to take alternative courses of action dependent on the dynamic state of a variable. From a machine point of view this change of sequence can take the simple form of a conditional branch. From a HOL point of view it is embodied in the basic notions:

a) alternate choice: IF...THEN...ELSE; DO CASE;

b) iteration: DO loop; DO WHILE;

c) modularization: GO TO; CALL; function; RETURN; EXIT;

General machines tend to reflect alternate choices with conditional branches, iterations with some form of indexing operation and testing (since one of its main uses is for array processing), and modularization with branch and link type instructions.

Conditional branches do correspond rather directly with IF...THEN...ELSE and indexed branches with DO CASE. The trouble with branch and link in general machines is that it does not have any effect upon the environment of the algorithm. Depending upon the addressing structure of the machine, a new address environment must be put into effect with each subroutine. Similarly, current values of the operand registers must be saved or protected. The B6500, corresponding closer to a HOLM, does save current values via its stack (since it is a pushdown stack) and does the necessary housekeeping in order

to set up the desired address environment for the subroutine called. It must be realized that if the "operator" does not set up the environment, it still must be set. Since modularization is a feature of an HOL (subroutines, procedures, programs) this is done quite often, and should therefore be done efficiently.

Just as entering subroutines requires the saving of registers and the setting of the environment, returning requires resetting the environment and registers to previous usage. Depending on the HOL being designed, there may be the need for either an abnormal exit or transfers to a label or procedure which has been passed as parameter. These, of course, involve careful housekeeping in order to ensure the proper environment and the resetting of and purging of registers.

DO loops owe their occurrence to several different factors as was seen above. Assuming the HOL is semantically able to reflect array processing, it becomes necessary for the HOLM to implement array operations. If implemented explicitly (although array processing, in the general sense, is not likely to be common), array operations become relatively simple since no explicit iteration variable is to be found; instead only a current value, increment value and final value. DO loops in the general case not only depend upon these three values, but also contain an iteration variable which can be changed within the iteration. The increment and final value could also be expressions which must, in general, be recalculated since they could also have changed within the iteration. This reasoning leads to the conclusion that except in the very simple loop case (which is the most frequent [12]), the statement sequence: expression calculation, test, conditional branch, (statement code), absolute branch, will probably be the most general and optimal sequence of instructions. It is of course possible to maintain loop information in special registers. The IBM 360 BXLE, BXH instructions are particularly designed for loop control, but are seldom found in actual use since they make large demands on the resources of the general registers.

The B6500 has a special instruction for those cases of simple loops, and the Keeler SPLM uses an INCR operator especially designed to perform and test for the increment at the end of a loop.

2.2.3 System Functions

The need for special instructions for manipulating the registers or data paths, and for easing the implementation

of algorithms depends upon the instruction set. There is also a need for instructions which reflect the network architecture of the system, the paths of interconnections and communications. Multiprocessors introduce new instruction needs because processors become a multiple resource, and, of course, there is the need to communicate between the computer system and the outside world (I/O instructions).

a) Instruction architecture needs. On the instruction level the IBM 360 has such simple needs as an NOP; Load PSW to set the "control information"; ISK and SSK to set the memory protection information; and SVC to implement operating systems function.

The B6500 besides having similar executive functions contains explicit instructions to handle the instruction architecture. In order to facilitate the use of the pushdown stack there are instructions which manipulate it: PUSH, DLET, EXCH, DUPL, RSUP, RSDN. (In a similar vein, the IBM 360 does load register, LR.) Since the B6500 associates tag bits with memory cells, it must be able to manipulate these as an entity; hence, such instructions as STAG (store tag bits) and RTAG (read tag bits). On the other hand, storage protection is performed as a descriptor function.

The actual frequency of use of stack manipulation instructions depends on the compiler, both from an algorithm point of view and the compiler writer's convenience.

For example, consider:

A(I+1) = B(I+1)

When a subscript is repeated within a statement, the statement can be optimized by only having the subscript calculated once. With a stack mechanism there is no need to create a temporary to store the results but rather it is possible to

1) duplicate the subscript value before indexing, so that a value is available as the next indexing value, and

2) exchange the top two positions of the stack if it occurs in the wrong order.

For example, a B6500 sequence could look like:

ONE	put a literal one into the stack
VALC I	put the value of I into the stack
ADD	add, leaving result on top of stack
DUP	duplicate the top of the stack
NAMC B	place address of B in top of stack
NXLV	load value of B(I+1) in top of stack
XCH	exchange this value with previously duplicated I+1 value
NAMC A	load address of A
INDX	index this address with I+1
STOD	store value B(I+1) into A(I+1)

b) Network Architecture Needs. The system must be able to control its interaction with the outside world. This includes such instructions as HALT, IDLE and the enabling/disabling and masking of the interrupt system. As long as asynchronous events are allowed to occur and timing is important, interrupts will be present. The particular nature of the system will depend upon the critical nature of the required processing time, and the degree of asynchronous complexity.

c) Multiprocessor Environment. In a multiprocessor environment the "system" may not be identified with the "processor". Instead the processors become a commodity to be allocated along with the other resources (I/O channels and devices, memory, time). In order to identify how the system is functioning, it is necessary that the processor be able to interrupt another processor (any one but himself). These functions are represented in the B6500 as the WHOI and the HEYU instructions, and in the RCA 215 as the "Load CPU Identity" and "Interrupt CPU" instructions.

Depending upon the design of the executive, and the network available, a set of system configuration instructions could be mandatory.

There must be the equivalent of a Test and Set instruction to make sure only one process may use specific data or program at a given time, when this is logically necessary.

d) I/O Instructions. I/O is dependent on the device for which the interface exists. If I/O is carried out asynchronously with respect to a processor, then a process requires simply to start, test and halt the device with an instruction to test the channel if it is a separate unit. The particular information needed to control a device is then given in memory and these instructions are designed for the particular device and are executed by that device once it is "started". The IBM 360 also includes read and write direct instructions.

2.2.4 Conclusion

In the above analysis the instructions have ignored any particular function which would be useful to the executive. These instructions would in particular have to do with the data manipulations which are carried out by an operating system. These data structures are involved with resource allocation, and, in particular, include queues (FIFO), push-down stacks (LIFO) and linked lists. However, to take advantage of these data structures, it would be necessary for the operating systems to be written in a language in which they are semantically recognizable. An algebraic language does not contain queues, stacks or linked lists, and it would be impossible for the compiler to create such a semantic unit out of user code which did not explicitly indicate their existence. A DO loop is a DO loop, and it is impossible to discover that it is really a table search.

The extremes of descriptor usage can be seen in comparing the IBM 360, B6500, and the SPLM. The 360, as has been seen, has all type and precision information explicit in the operator. The SPLM learns everything from the descriptor. When an assignment operator occurs, all type, precision, rounding, truncating, conversion then take place. The B6500 is mid-way between. Arithmetic type versus character type information is explicit in the operator, while precision is found in the descriptor.

One further complication occurs in considering instruction sets. If the processor has a large degree of parallelism, and is capable of obtaining information from memory asynchronously to its use of its arithmetic and logical

unit, then there are possible advantages to having more information in the descriptor. If an operator does not contain all the information needed to transform the operands, then the descriptor must be consulted. If this must be done sequentially with the operator indicated transformation, the operator syllable would become the bottleneck in the program string execution, while the "load (address) of operand" would become a single immediate operation. That is, memory would not have to be consulted, since the information would be in the program string itself. If the descriptor is of variable length (e.g., SPLM) this would, in general, mean that even the descriptor would not be found in the stack. The stack would contain only an indirect pointer to the descriptor, which would reside in some other location of memory. Since processors (due to memory costs) tend to be "memory bound" the interaction of separate descriptor fetches to memory must be carefully weighed against the gain in generality of the operators and the reduction in program string length, or a mechanism to circumvent the extra fetch must be implemented.

2.3 MP Instruction Design Factors

It has been decided that the design of the instruction set of the MP computer should be tailored to a "higher order language". This not only simplifies the implementation of the system software and removes large classes of potential errors, but it also reduces the amount of memory needed for operation. The reduction in the number of bits needed for the storage of a program in a HOLM versus a MOLM is due to several complementary factors. These include the use of descriptors, implicit addressing of the stack, dynamic two dimensional addressing, minimization of different data types, and semantic conciseness of the operations.

2.3.1 Polish String Compactness

"Polish" machines require fewer bits in the program string than do standard machines. There are at least five separable reasons for this phenomenon. In the following it is convenient to consider the Burroughs B6500 and the IBM 360 as representatives of two possible extremes of instruction architectures.

a) Two Dimensional Addressing (Static and Dynamic)

Both the B6500 and IBM 360 have two dimensional addressing and exhibit a savings over the first and second generations. In order to process large computational jobs a large amount of addressable space

is needed. With a second generation machine such as the 7090 all of this space (and hence the limit of the memory size) must be addressable In this case then, it was necessary to use 15 bits in every operand address. The IBM 360 and B6500 both have two dimensional addressing. The IBM 360 uses a 12 bit displacement which is to be added to one of fifteen base registers. This allows for a full 24 bit addressing (of bytes) scheme. Here 24 bits of address space has been compressed into 16 bits of information. The B6500 scheme uses only 14 bits with its operands, where the "base" (display) registers contain only the number of bits needed to indicate the current lexical level ($\ell\ell$) (i.e., $\ell\ell = 1$ implies 13 bit displacement, $\ell\ell = 2$ implies 12 bit displacement) and the B6500 displacements refer to "words". Since program segments in the B6500 are described via a "descriptor", the actual size of memory which could be addressed is only limited by the numbers of bits so used in the descriptor. In point of fact, Burroughs uses a 20 bit word address field in the descriptor.

It is easy to see that if the memory of a computing system is large compared to the modular size of "programs" (or perhaps even procedures and routines), program string savings are to be found by using a two dimensional address.

There is a great difference, however, between the IBM 360's and B6500's two dimensional addressing schemes. The IBM 360 base registers are assigned "statically" at compile time, and it is up to the compiler to try and optimize base register usage. This optimization is minimal if only one base register is needed within a segment. This becomes difficult in large segments since the dynamic characteristics of the segment modularization must be considered.

The static two dimensional addressing of the IBM 360 has several aspects:

1) By using 4 bits everywhere for base registers the displacement range is reduced, since seldom are that many registers required.

2) If a program is "one big" segment, then several base registers are needed and segment boundaries must be carefully watched.

3) If the base registers are set upon entering and upon returning to each module then i) there must be code to do this in the program string, and ii) name scope problems arise when variables in a previous level are to be addressed, since their base registers are in general no longer properly set.

The B6500 optimizes upon the two dimensional address idea by

1) using only the number of bits necessary for the current lexical level to indicate the number of bits for the "base" register. This leaves the rest of the bits for displacement. (There is also the fortuitous circumstance experienced by all, that the more "inner" a subroutine the "smaller" it is, i.e., less displacement is needed to fully address it.)

2) The base registers point to the beginning of each dynamic module, hence allowing the displacement to reach its most extreme logical dynamic range.

3) Since the usage of the "base" (display) register is unique and well defined, (versus general, e.g., base register/accumulator/index register) the initialization and resetting of them can be accomplished automatically, requires no explicit code in the program string, but does maintain the current dynamic name scope.

b) Implicit Addressing

It is interesting to note that in one sense most computers are Polish machines. That is, they all execute their instructions in the "sequential" form as presented to them by the output of a compiler.

A = B + C;

on the B6500 versus the IBM 360:

B6500	360
VALC C	L RO, A
VALC B	A RO, B

```
ADD         ST R0, C
NAMC A
STOD
```

In each case: (fetch C), (add B to this value) and (store value into A). In effect it is the only sequential form possible (ie., ADD before STORE) for this expression.

However, when temporary locations become necessary a difference appears in the code, although the total effect, must of course, remain the same. Consider A = (B + C) * (D + E);

```
B6500       360
VALC C      L R0, C
VALC B      A R0, B
ADD         ST R0, TEMP
VALC E      L R0, E
VALC D      A R0, D
ADD         M R0, TEMP
MULT        ST R0, A
NAMC A
STOD
```

Assuming that there are only a few (in our case exactly one) accumulators being used, during the expression evaluation it becomes necessary to create a temporary.

The creation of a temporary indicates an increase in the program size for two reasons.

1) In general, the use of temporaries is a static decision and hence cannot behave better than the dynamic usage of the stack, therefore, in general, one needs more "temporary storage" locations than stack storage.

2) More importantly, in the IBM 360 type of machine, every instruction has an operand, therefore the temporary requires an address which in turn takes space. The B6500 uses implicit addressing; the needed number of operands coming from the appropriate number of locations on top of the stack.

When temporaries are needed, most often (at least much of the time) an implicit address scheme allows for the savings of "temporary" operand addresses.

c) Descriptors

As was seen above, descriptors can be considered either as sub-operators or as the ideal data structure which is being manipulated. When considered in the first manner, it is seen that the descriptor saves on the program string length since "fewer" operators need be specified since the "sub" part of the operator is found in the descriptor of the data structure. For example, the IBM 360 has for "add":

AR, A, AH, ALR, AL, AP, ADR, AD,

AER, AE, AWR, AW, AUR, AU, AXR,

while the B6500 has simply "ADD". This of course requires fewer opcodes, and in turn fewer numbers of bits to represent the necessary operators.

When the descriptor is regarded as the "data structure", it exhibits at least two virtues. One is that by being "semantically concise" (further discussed below) it places into one location the complicated description of the data structure, which thereby need not be repeated in multiple references in the program. The other is the observation that the number of entities which are manipulated by a program are few. The reason that a large addressing space is normally necessary is that if the machine does not have descriptors, then each "memory cell" of the data structure must be directly addressable. The example of an array of 100 scalars on the IBM 360 is in fact 100 memory locations. On the B6500 it is one entity: a descriptor, which indicates the dimensions of 100 and where it is to be found in physical core. This very important phenomenon reduces the addressing requirement of a program

string, since the full physical memory address need only appear in the descriptor. The descriptor becomes one of the "few" entities which must be addressed, and hence only a small address field is needed in the program string proper.

d) Type Differences

Descriptors allow any information which can be "bottlenecked" to be placed in the descriptor once, instead of having the information repeated throughout the program string.

In addition to character data (for I/O) and an internal arithmetic form, most machines have other internal forms. The difference between the "character" and "internal arithmetic" comes largely from the savings yielded by compactly storing and manipulating them in the internal form. The various internal forms come from precision considerations and also from speed considerations (integer arithmetic versus floating point).

Types can be optimized by:

1) Making one a proper subset of another (e.g., integer is a subset of single precision floating point on the B6500) the difference between the operators disappears (except for an explicit operator to recover the proper subset; such as INTEGERIZE).

2) The need for multiple forms of the same operator disappears (e.g., IC, LH, L, LD, LE).

3) The need for explicit type conversion operations is reduced. The program string can be further minimized by providing an explicit operator for each type conversion when needed (e.g., scalar to character, while integer to scalar would be implicit by the integer definition as a subset of scalar).

e) Semantic Conciseness

Probably the most significant impact resulting from the use of semantically compact operators is the decrease in program string length. When the operators correspond to the operations in the problem being executed, a minimal amount of translation is needed and hence the minimum amount of expansion in the program string.

When a procedure indicates a vector cross product, it means a "vector cross product", not some strange manipulation upon six scalars resulting in three scalar results. This form of semantic compactness is rare to find, perhaps currently only in those machines designed to execute APL.

The Kerner and Gellman machine is "semantically concise" in the execution of FORTRAN since its operators are those that FORTRAN indicates. It is also interesting to note that this machine does not have "two dimensional addressing", "implied addressing", "descriptors", or "type optimization", yet it is a semantically concise FORTRAN machine.

The Burroughs B6500 is similarly an "ALGOL" machine. But it does have the other program string size-saving features, and is semantically concise with regard to "ALGOL".

But neither of these machines are necessarily "semantically concise" with respect to a linear algebra problem, since their respective languages (FORTRAN and ALGOL) are not. Of course the IBM 360 is semantically concise only to "BAL" which is merely a tautology. In fact, then, the IBM 360 is not semantically concise to any real "problem oriented language".

Besides being semantically concise with respect to the operations expressed in a problem, the operators can be "semantically concise" in the way in which they are constructed. Branching occurs locally within a program under execution and not with respect to all of physical memory. The IBM 360, as most machines, allows the branch address to be any address of physical memory. The B6500 uses relative addressing (that is, relative to the program under execution). This of course reduces the address space necessary, since it corresponds to the dynamic space involved at execution time.

Along the same lines of reasoning is the question of indexing. As has been discussed elsewhere two of the main reasons for indexing are array processing (e.g., linear algebra) and table searches. If the problem oriented language has such operations, and the machine is semantically concise, such indexing need no longer be explicit. Single references of data via indexing, can be accomplished through the descriptor, with or without descriptor modification.

In the IBM 360 each memory reference instruction generally carries 4 bits of indexing information. The B6500 indexes only when needed, and since a stack is used (hence implicit addressing) only an 8 bit operator is needed (which can also load the resultant indicated entity). Assuming that not every memory reference is indexed (the indices themselves must be fetched from memory) the use of indices only when needed (and semantically concise operations make the need less) will minimize the program string length.

The use of short literals also compactifies the program string since the constants used are usually small integral values. Recognition of this fact allows for their representations in the amount of space needed and not the amount for the worst (largest) case possible.

There are then at least five different aspects to the conciseness of "Polish" machine program strings which contrast with the more conventional machine:

1) Dynamic two dimensional addressing

2) Implicit addressing of operands

3) Descriptors

4) Type optimization

5) Semantic conciseness.

Each of these could be implemented separately or in combination and with varying degrees of usage.

2.3.2 Information Bit Width: Data and Descriptors

The basic flow of information can be considered as the gathering of the necessary operand information into the processor "pushdown stack", followed by an operation upon those operands, leaving the result in the stack.

In the laboratory model design there is the need to be compatible with current industry peripherals in regards to data size. This forces the basic data structure of the instruction architecture to be built upon multiples of the standard eight bit byte. Further consideration of the needed accuracy of arithmetic computations, and of the size of addressable memory within the descriptor leads to the question what multiple of the basic byte the data word should be formed.

2.3.2.1 Arithmetic Precision: The binary basis of computer design has predicated data structure sizes as multiples of the base 2. The most efficient addressing, with regard to bookkeeping is when the number of entities manipulated is a power of two. The eight bit byte has become the basic industry unit of memory. As an example, when the IBM large computers developed from the second generation 7094 to the third generation 360, the arithmetic types switched from 36 bits to 32 bits.

If one postulates the use of 32 bit single precision floating point, the decision must be made as to how many bits are used for the mantissa and how many for the characteristic.

The most accurate input precision needed with guidance and navigation is one part in a million. In these extremely rare cases, $10^6 \approx 2^{20}$ implies at least 20 bits of mantissa. Most digital to analog input has a maximum of one part in a thousand or $10^3 \approx 2^{10}$ i.e., between eight and twelve bits of information.

A standard way of breaking a 32 bit floating point word is shown in Figure 2.3-1a. (For the present, the actual location of the bits within a word is not important, but the number of bits used for the various purposes is. In fact, since arithmetic types are truly "internal" forms, this exact mapping of bits is a decision for optimal usage of the hardware.) The precision of 6.9 digits is just acceptable and the range of $10^{\pm 38.4}$ is usually acceptable. When double precision is used, the second word again can be divided between mantissa and characteristic. There are two different reasons to make a double precision data form. One is to increase the precision of the calculation, and the other is to allow an increase in the range. By solely adopting either, the other remains a potential problem. Figure 2.3-1b shows the increase in range and precision by going to double precision. The precision is now 15.3 digits while the characteristic has been increased to $10^{\pm 614.4}$. Since the arithmetic form is internal, the second word's characteristic has been concatenated to form the higher part of the joint characteristic, while the second word's mantissa has been concatenated as the low part of the joint mantissa. This has been done in order to make the single precision value a subset of the double precision. Similarly it has been assumed that the radix point of the single precision word is to the right of its mantissa when the characteristic is zero. This was done under the normal HOL convention of minimizing data types by making the "integer" form a subset of the single precision floating point.

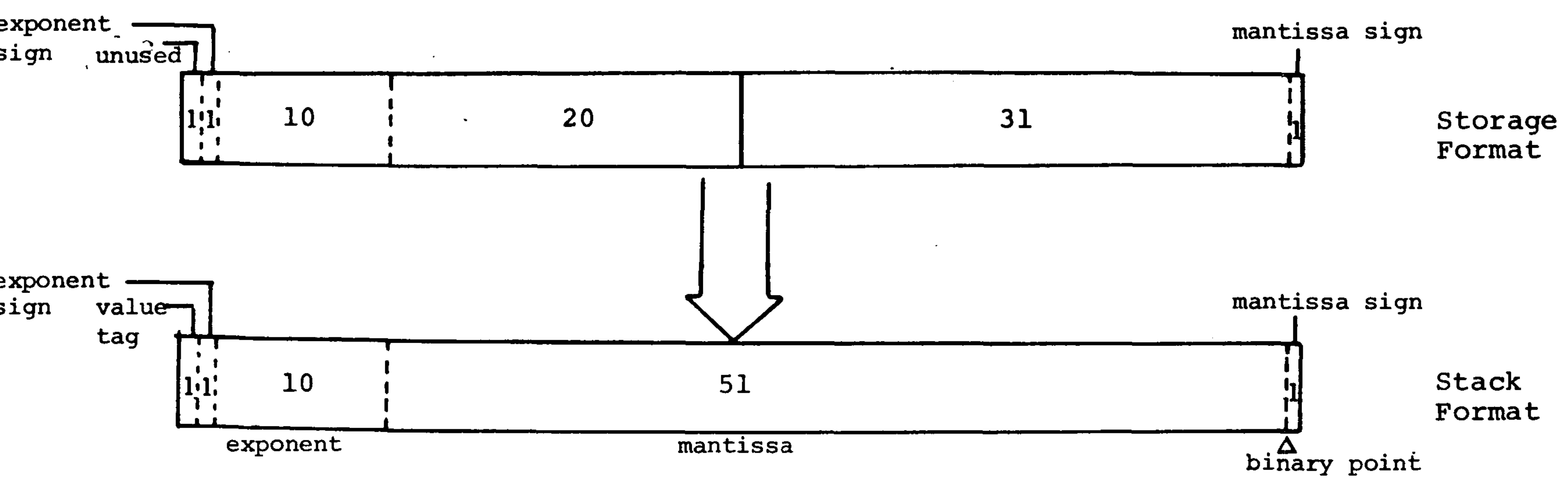


Figure 2.3-1: Mapping of Single- and Double Precision Floating Point Data From Storage to Stack

In this scheme operators either refer implicitly to the stack or to control information. The actual moving of information into the stack is indicated by (several) operand meta-operators. These meta-operators must be able to address the "values" which are to be manipulated. In modern block oriented languages the "name scopes" of a routine follow the static embedding of the routine at compile times. The only other values which can be manipulated by the routines are those that are passed as formal parameters. This block oriented name scope property allows for dynamic two-dimensional addressing, where the lexical level of the routine, plus an indicator as to which entity at that lexical level is being referred to, uniquely identifies the variable in question. HOLs which contain COMMONS or COMPOOLS pose a further addressing problem since COMPOOLS are at the outermost level of name scope, yet if they are "named" COMPOOLS, this name would be a third dimension of addressing. The special problems of COMPOOL addressing in the MP instruction architecture will be discussed later, in section 3.4.9.

In actual execution it has been seldom found that more than eight lexical levels are reached, and hence growth potential to 16 or 32 levels seems more than adequate. Since the basic data building block is the byte these (however many) meta-operators must consist of some multiple of bytes. Statistically the operand fetches are in great preponderence in the system and outnumber all other operators, except literals or immediate information. Since at least two bits are needed to indicate the operand meta-operator (to distinguish value from name), if they were packed into one byte, then at most six bits could be used for the "address" of the entity. This comprises only 64 possible addressable entities and is too small to be truly useful, since the "lexical level" of the entity must also be specified. (If one were to argue that one byte meta-operators could be used, at least for current lexical level entities, then one more bit would be needed to differentiate them from the general meta-operands which would still be needed. The number of entities which then could be indicated would be at most 32.)

The two dimensions of the address refer to the current stack at the appropriate lexical level, and the entity offset. This type of addressing does not place any real restrictions upon the architecture of the machine, but only limits the number of individual entities which are locally addressable in any given routine. However, the number is large, at least 10 bits, implying over 1000 addressable entities.

Besides the operand meta-operators, addresses appear in the course of program execution, either through formal

parameter usage or in a descriptor (of data or a program segment). If a quantity is a formal parameter, then it is defined and exists within the routines which called the current routine. It, therefore, has a stack entry location and an associated two dimensional address. Unfortunately, it might not be in the current static (compile time) name scope, and care must be taken to indicate which environment it is in. If the "name" is a descriptor, then it points to where the value field is to be found. Since the MP is to support a dynamic memory management policy, the descriptor must have a large enough address field to contain the address of any M2 or M3 location. This then partially predicates the bit lengths of a descriptor without considering its other functions. M3 is to be about one million words and M2 is considered to be one hundred thousand words.

M3: $10^6 \simeq 2^{20}$ implies 20 bits

M2: $10^5 \simeq 2^{16.7}$ implies 17 bits

If every word of M3 is to be addressed, the 20 bits for the M3 address must be provided along with one extra bit to distinguish M2 and M3 addresses. If the M3 were block oriented of at least eight words, then the 17 bits for the M2 address (plus one to distinguish the M2 from the M3 address) would be sufficient.

The remaining problem to be considered is: how does the execution distinguish between operations, names, and values? Since the operators are indeed the program stream, and flow through the program is either normally implicit (or explicit when branching or transferring to another program module), the only real problem is to distinquish names from values. Operand values along with descriptors are entities which are addressed by the operand two dimensional address. The question arises as how to distinguish the "values" of the operands, from the "name" of the descriptor. One possible answer would be to force all operands to have a descriptor: this is in fact what Keeler's SPLM postulates. The other solution, which is commonly found, is to include "tag" bits on all entities. These "tag" bits are used primarily to distinguish "value" from "name". (Burroughs B6500 also distinguishes "single" precision from "double" precision in order to facilitate the transfer of operands from the hardware part of the stack into the M2 memory part of the stack.) The solution of "tag" bits on all operands is thought to be expensive since all data must have them, even though they do have the virtue of maintaining the integrity of the difference between "name" and "value".

There are two logically different "names". One is created as a formal parameter which is analogous to the pronoun

"he", and the other gives a description of a complex data structure, which is a proper name such as "John". The formal parameter name refers to an entity which is not known to a routine at compile time. While tag bits could have been bypassed in the differentiation of "value" and "descriptor" by means of separate operators for each, the use of formal parameters means that a descriptor and value must somehow be differentiated.

The question of the differentiation of name from value, besides being a necessity in the case of formal parameter passage, is also involved with the policy of segment management within a dynamic environment where there is the desire to prevent long chains of indirection during execution.

When a formal parameter linkage is created, if the instruction simply created an "indirect" reference to another two-dimensional address within its name scope, then this could in turn be a formal parameter, and so on. Hence when the value was to be fetched, this indirect linkage chain would have to be followed each time. It is obviously desirable to have at most one level of indirection. (The value itself can not be passed since the value might change during execution, either directly or via side effects.) The presence of a descriptor is in itself a level of indirection to the values, but if it is passed as a formal parameter then multiple copies of the same descriptor begin to appear, which is very detrimental to dynamic memory management. (If a segment is to be removed from core, all descriptors referring to it must be found and updated.) Therefore, assuming one level of indirection for formal parameters, how can this indirection indicate whether it points to a value or a descriptor name?

In most HOL machines a given instruction indicates with an "indirect" bit that the entity referred to is to be indirection to the desired value. In some cases the entity referred to, if indirect, can in turn indicate that what it refers to should be further indirected. Our case of the differentiation of "names" and "values" is somewhat different in that the entity itself indicates if it is a "value" or a "name" and hence if the value is to be found elsewhere. The primary advantage of having the entity indicate whether it is a name or value is that the program code then is identical for either case.

Consider that the internal specification of arithmetic types is the concern of only the machine, and the user communicates via character strings through the limited interface called the I/O. It can be seen that for execution purposes, to an operator the information is either in the stack or available through a descriptor. The entry of information into the stack is via the operand meta-operators. The form that the information

takes within the stack is truly independent of how it appears in the rest of M2 or M3.

In order to avoid all data having a tag bit(s), the MP instruction architecture proposes a solution which has several interesting ramifications. This is to maintain all information within the execution part of the stack in double precision. By considering all execution stack entries as 64 bit quantities, 64 bits are made available for either the value or for any necessary tag bits. A floating point entry in the stack is to be a 63 double precision floating point quantity with one bit of tag information. This bit is to indicate that this is a "value" versus a "name".

Single precision has now become a proper subset of double precision. All execution takes place as if it indeed was double precision. This has the added advantage of greater precision in expression calculations. Usually it is only in intermediate calculations that one finds extreme ranges in a variable. (For example, a term such as μ/r^3 in celestial navigation has a great range, while the final result has a much smaller exponent.) The greatest loss of significance in calculation is to be found when subtraction takes place between variables having similar initial values. By using double precision operands, more significance is available in intermediate calculations. Finally, a major source of error normally introduced into calculations when the number being represented is either transcendental, or does not have a finite base 2 representation (such as e or .1, respectively) is minimized. The calculation is correspondingly enhanced with more precision.

The main objection to this approach of maintaining double precision within the execution part of the stack is both in the cost and the time needed to continually perform double precision calculations (versus single precision floating point and/or integer). The benefit to be gained from uniformity in arithmetic data types, along with enhanced precision in calculation, is thought to outweigh the extra cost of a 63 bit floating point ALU that is capable of performing within the specified time constraints.

2.3.2.2 Descriptors: The potential power of descriptor usage has already been discussed rather fully. From an implementation point of view the main constaint is the size of the descriptor and the need for the descriptor to handle linear arrays. In order to handle linear arrays the descriptor must be able to address any secondary storage (M3) location in which the array may be stored. Similarly, it must provide bounds checking.

a) Bounds Checking and Structure Mapping

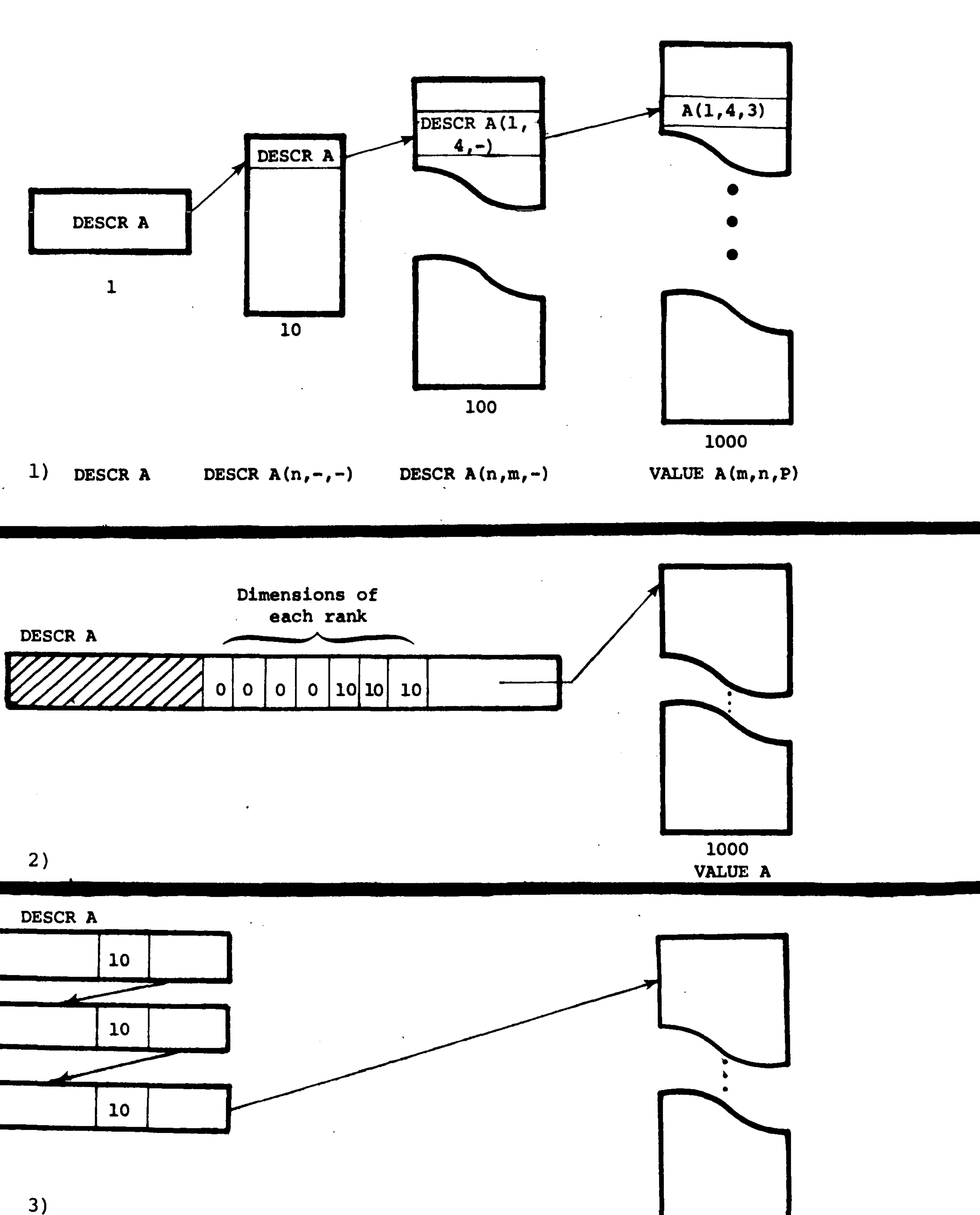
Arrays of one rank are often manipulated by the changing of the length of the array, either by changing the memory starting address or by modifying the allowable length. In the MP design, however, the memory management has been taken into consideration. It has been decided not to allow more than one descriptor to actually point to a segment in M2 or M3. Any other descriptors which wish to point to the segment must instead be COPY descriptors, and point to the main (Mom) descriptor. This means that array partitioning cannot be manipulated by changing the M2 address; rather the descriptor must have an explicit field to indicate the array offset starting point. This tradeoff between reference usage (the extra level of indirection imposed by memory management with exactly one descriptor per segment) was made because of the possible gains to be made in reducing memory management overhead by the use of a content addressable memory to offset the execution penalty of the extra level of logical indirection.

b) When multirank arrays appear, it becomes necessary to maintain knowledge of the spacing between the elements of each rank, since the multirank array is physically stored into a single rank linear array memory. This added complication has been attacked in several ways, none of which is satisfactory from all viewpoints (Figure 2.3-2).

1) Burroughs has selected to provide descriptors of but a single rank. If multirank entities are encountered then the higher ranks are implemented by a descriptor of descriptors, each of which describe a rank. This is unsatisfactory in at least two ways. Consider a three rank entity of dimensions 10 x 10 x 10. There is one descriptor for the whole entity which points to 10 descriptors each of which in turn points to 10 descriptors or a total of 111 descriptors for the 1000 element array. In order to get the first element from the lowest rank for each of the higher ranks involves inordinate overhead in code and time. If the lower ranks are generated dynamically the savings found in space are paid for in the considerable extra time needed for their dynamic creation.

2) At the other extreme the descriptor provides the mapping information for each rank. This is indeed what both Keeler's "paper" SPL [17] and Abrams'

Figure 2.3-2: Representation of Multirank Arrays



"paper" APL [16] machines do (though in detail differently). The difficulty with this is, of course, that either each descriptor is provided with the ability to handle the most extreme case allowable and hence the general (normal, small) user pays continually, or else there is a scheme for variations in the descriptor size. If the descriptors are chained, then this can either be done by pointers, if non-contiguous, or by an escape indicating more of the descriptor follows. Davis and Zucker [51] have proposed one method in which each rank points to the next. The disadvantage of this appears most readily in the problem of the parameter passage since now a variable length quantity must be handled, and hence can only be referred to indirectly. (It might be pointed out that when multirank information is being processed, any of these hardware aided schemes provide great savings over the general software approach of brute force.)

3) A third method would be not to modify the descriptor of the array, but to carry along with it a separate entity which would correspond to the mapping within the descriptor array of the virtual multiranked data. This is similar to the above linked descriptor approach, but allows for ease of redefinition of what the arrays look like, along with an easier hardware implementation. The details as to whether the mapping quantities are to be kept as spacings between each rank, or whether each lower rank should appear as a proper subset of the higher is not entirely clear.

4) Currently, the approach which is most favored is a compromise. It is necessary, if the HOLM is going to allow complex data structures, that descriptors of descriptors be permitted. This allows for the structuring of information within the HOL, of combining data types and in effect creating data types not forseen by the HOLM, or even the HOL. Multirank information is handled by a special rank descriptor describing each succeeding rank. An MP instruction will handle these entities automatically when it is possible to index them down to a single element. The more general case is provided for by explicit operators.

From the arithmetic point of view another design tack would have been to try and use 48 bit floating point numbers. This improves the precision while allowing an increase in range to an acceptable limit. However, the precision is not enough for all calculations and hence some form of double precision is still necessary (Figure 2.3-3).

2.3.2.3 Addressing Scheme: Computation is concerned with the manipulation of data. This manipulation has three different components: the operations to be performed (operators), the indicators of those things which are to be manipulated (names or address), and the data which is actually manipulated (values). These three entities: operations, names, and values are merged into bit patterns in the general von Neumann type of computer, even though a higher order language implicitly differentiates them. The operations themselves are of two forms: they either manipulate data or control the flow through the program.

It was seen above that a HOLM saves on memory by

a) implicitly addressing a stack

b) having a dynamic two dimensional address mechanism that corresponds to the logical program module.

If one considers the addressing needs of each of the above three entities, the restrictions on word size will be seen. Since the basic unit of data available in the proposed MP is the eight bit byte, the smallest separable particle that can be called an "operator" will then be assumed to be of this "byte" size. Operations either manipulate "values" via the stack, or they refer to program control information. If they refer to the stack, then their addressing is "implicit", in that operands needed by the operator are taken from a number of positions near the top of the stack. If the operator is referring to control information, it can either change the program sequence within a given program module, or transfer to another program module. If the operator is transferring control within the same program module (branching), all of the module is known at compile time, and a simple self relative branch can occur. The actual size of a branch operator is constrained to be a multiple of the byte. When transferring to another module, if a return is indicated, the stack already contains the information as to how the current module can be entered, thus enabling the return. When another module is called the process of locating it in a dynamic environment predicates a call to an address which has a descriptor of the described program segment. This will bring the module into memory if necessary.

sign		sign	
1	n	1	46 - n

characteristic (sign bit + n bits) — mantissa (sign bit + 46 - n bits)

n = 9

$2^{(2^9)} = 2^{512} = 10^{153.6}$

characteristic base 10 = ±153.6

$2^{37} = 10^{11.1}$

mantissa = 11.1 digits

n = 8

$2^{(2^8)} = 2^{256} = 10^{76.8}$

characteristic base 10 = ±76.8

$2^{38} = 10^{11.4}$

mantissa = 11.4 digits

Figure 2.3-3: 48 Bit Single Precision Floating Point

2.3.3 Hardware Stack Depth

One other main design criterion has highly influenced the MP instruction architecture. This is the consideration as to how deep the "hardware" part of the stack mechanism should be.

No matter how the final data flow between machine registers is implemented, it is extremely advantageous to postpone placing an operand upon the stack until it is absolutely necessary. Consider the HOL expressions:

A = B + C;

The best Polish form from a stack depth point of view would be:

CB + A =

and hence a "normal Polish machine" implementation would be

load value of C

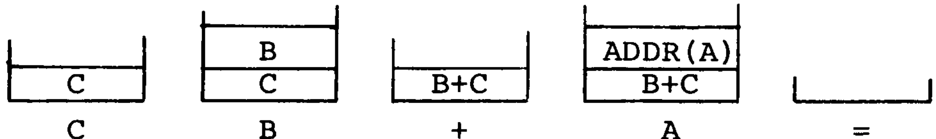
load value of B

add

load address of A

assign

If the flow in the stack is considered it is seen to be



If the stack were implemented with only one hardware register at the top of the stack, then every time the depth was over one, an entity would have to be pushed into memory, consuming a memory cycle. (This argument is valid for any small finite depth for the hardware part of the pushdown stack.) In looking

at the action upon the stack in the above example, it is seen that both "B" and "ADDR(A)" are only present until the operator (which immediately follows) is acted upon.

It would be possible to make this explicit in the program string by using the standard Polish postfix operators with a combination of infix operators. For example, the code could be rewritten as:

B(+C) (=A)

and then the stack action would be:

B B+C

B (+C) (=A)

On the other hand, postfix operators can not disappear completely. Consider

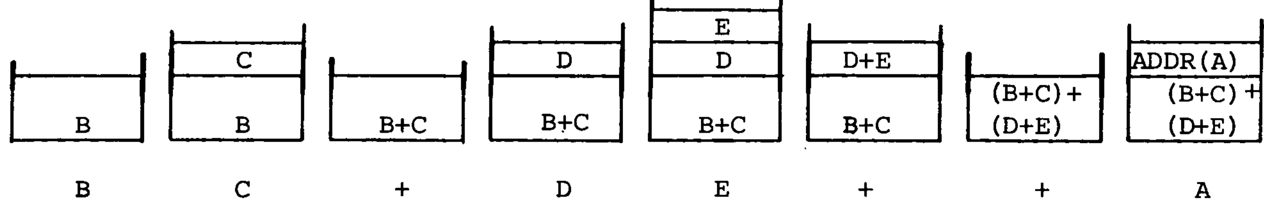
HOL: A = (B+C) + (D+E)

Polish: BC + DE ++ A =

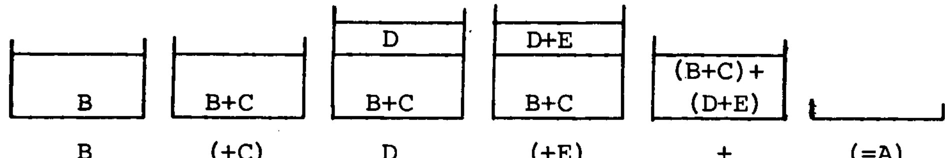
Combination: B(+C) D(+E) + (=A)

The stacks in the polish and combination forms would look like:

Polish:



Combination:



Instead of "pushing" into memory implicitly four times, this occurs with the combination only once. The reason for this phenomenon of course is that most dyadic operators ignore their two input variables once the single resultant value is calculated. Hence the presence of the second operand on top of the stack is only until the next "entity" is decoded (the operator).

This understanding suggested a number of different approaches.

a) Since most data manipulations come about because of dyadic operators in which one operand is extremely temporary, the two top registers of the stack must be hardware for speed and should be considered logically as only one entry of the stack as far as "push up" is concerned.

b) The need for explicit infix operators can be circumvented if a "load value" or "load address" meta-operator does not complete its action of placing the memory fetch into the stack until the next "entity" is examined to see if the data is to be immediately used. If it is to be used as input for the operator there is (in general) no need to transfer the memory fetched data to the stack and hence chance the undesired possibility of a "push" upon the stack and, therefore, cause an extra memory write (i.e., to take the low hardware stack register and continue it into the memory part of the stack). If the "entity" does not use the value or address obtained, (e.g., it may be another "load value" or "load address") then the machine would need the ability to simultaneously place the operand at the top of the stack, possibly causing a memory write.

c) Although the above "look ahead" mechanism is conceptually the easiest to implement, since it leaves the Polish string intact, it might be desirable to actually implement the infix operators in the program string itself. This would save on possible hardware complexity. As was shown in the second example, the presence of infix operators does not remove the necessity for identical operators in postfix form. This stems from the fact that intermediate calculations take place upon quantities which are not explicitly addressed, but rather occur as the required number of stack entries. (On a non-stack machine where all addresses are explicit, temporaries must be created for intermediate results.) One form of implementation then would be to save a bit in all operators which could be both postfix and infix, to indicate if the operand(s) are implicit in the stack or that one operand immediately follows in the program string. For example:

+ operator: | + 0 |: use two stack entries for operands

| + 1 |: use top of stack and operand

In order to save memory cycles in the switch between the bottom of the "hardware" stack and the part of the stack in main memory, it is necessary to consider carefully the appropriate depth of the hardware stack.

The pushdown stack is in reality being used for several conceptually different purposes. Control information (procedure and/or program linkage and state information), parameter passing (make passed parameters addressable in the current environment), local data definitions (address storage for currently defined data), and expression evaluation are all embedded into one stack. Since all computation is done in a sequential manner this embedding of four different phenomena is natural.

Just as it is advantageous not to stack the second operand of a dyadic operator, it is desirable not to push into and read from memory too often, but rather to stay in the "hardware" part of the stack.

In examining the four different uses for the stack as indicated above, it is seen that the most dynamic usage is the expression evaluation. The life time of a value of an expression during evaluation is not very long compared to control information. (This, in general, follows from the fact that the expression is "within" the routine.)

If the "hardware" part of the stack is long enough to contain some significant percentage of all expression evaluations, then these calculations could be done without the extra memory cycles needed to push the stack into memory and, therefore, a significant gain in processing time should be expected. An optimal depth of the hardware stack would have to be obtained by an analysis of representative programs. Although this optimal value is not known, it is certainly greater than the two found in most Polish machines, but probably less than ten.

The policy of "pushing" into memory is relatively easy. It is obvious that a "push" must take place when the hardware part of the stack overflows. It is also obvious that the hardware part must be put into main memory if the "stack" is switched as when changing programs. (Note that even though this might entail a multiple store to "clear" the hardware stack, it is no worse than if there were fewer hardware stack registers since the

entries would then have been put into main memory previously.) However, the policy of "pushing up" must be decided. One of the reasons for increasing the hardware stack size was to prevent expression evaluation from writing into memory too often. If, however, every time an empty hardware register appears, it is automatically filled from the memory part of the stack, the hardware stack registers would continually be overflowing. The solution to the problem is in noting the parallel between the short lifetime (temporary-"ness") of stack usage for expression evaluation. This is indeed parallel to the above discussion on the use of infix operators. That is, if stacking was held off until it could be seen if the operator immediately used the operand, an extraneous memory write could be avoided. Similarly, if the depth of the hardware stack is considered just as a "single top of the stack register", but with an indepth look ahead, then it is seen that the policy of "push up" is just as if there were but one hardware register at the top of the stack. Therefore, a "push" into memory occurs only if the depth of "look ahead" is exceeded. (Or, of course, if the stack base is changed while changing programs.) A "push up" occurs only if the "top of stack" (i.e., all the hardware stack) is empty. (Or, perhaps, if there is a resetting when returning from another program or routine, or there is a call for "n" operands.)

Considered from this point of view the need for an explicit "look ahead" for the stack of a dyadic operator is not needed, but rather the "top of stack" operation merely consists of two hardware registers: one to be consistent for the use as a one depth "look ahead".

Considering the hardware part of the stack in this manner, it is seen that since it is in reality a look ahead mechanism and provides a method for removing the timing differences between postfix and infix operators, it can be used also to remove the difference between most dynamic and static instructions of multiple operands. For example, in order to set a bit in a variable, the difference of code between dynamic and static execution would normally be the difference in operators in order to be efficient.

$B_I = 1$	$B_3 = 1$
LOAD VALUE B	LOAD VALUE B
LOAD VALUE I	BIT SET 3
BIT SET	

If the hardware stack is simply an operand temporary store, then the second example above during execution, is equivalent to:

LOAD VALUE B

LITERAL 3

BIT SET

The MP instruction architecture, in general, is careful not to differentiate operators with dynamic operands from those with static operands which can be specified by literals. This has the advantage of reducing the number of operators which then need be implemented.

2.4 MP Instruction Architecture

The MP instruction architecture will be described in three sections. The first discusses the control structures of the architecture, its addressing mechanism and the usage of a stack. The second section describes the data structures defined for the MP instruction architecture. These include the various data types, the descriptors and other ancillary special words. The third section describes the actual set of MP instructions.

2.4.1 Control Structures

The addressing and data flow within the MP instruction architecture reflects the assumed Polish nature of the program code and its execution. The various forms of addresing found within the MP and their relationship to stack usage are now described.

2.4.1.1 Addressing. There are four types of addressing which appear in the execution of programs within the MP architecture. These are: addressing relative to the instruction location, lexical level and displacement ($\ell\ell$,d) of an operand, stack number and offset (SNO) of an operand relative to the system environment, and finally a physical M2 or M3 address.

a) The first form of addressing, relative to the current instruction, is used for both control flow within a program segment and for literals within a program segment. This form of addressing allows compactification of the program code since not all of M2 or M3 memory need be addressed, but rather only a relatively small amount of information for the short distance involved need be provided. Figure 2.4-1 shows an example of this addressing form.

b) The second form of addressing, lexical level-displacement, is used to address operands corresponding to name scope rules within the implemented higher order language. In languages with a name scope property modeled after ALGOL, each procedure is defined by a block of code. Within this block of code local variables are definable. Besides these local variables, variables which are defined within blocks which contain the procedure are also able to be referenced. Hence, at any given procedure level there is a linear sequence

IF A = B THEN C = A - 1 ELSE C = A;

		Byte Length
	GET A	2
	GET B	2
	EQUL	1
	LTS4 9 (= else - then)	1
	JOF	1
then:	GET A	2
	LTS4 1	1
	SUB	1
	ADR C	2
	STD	1
	LTS4 5 (= end - else)	1
	JMP	1
else:	GET A	2
	ADR C	2
	STD	1
end:		

Figure 2.4-1a: Relative Addressing

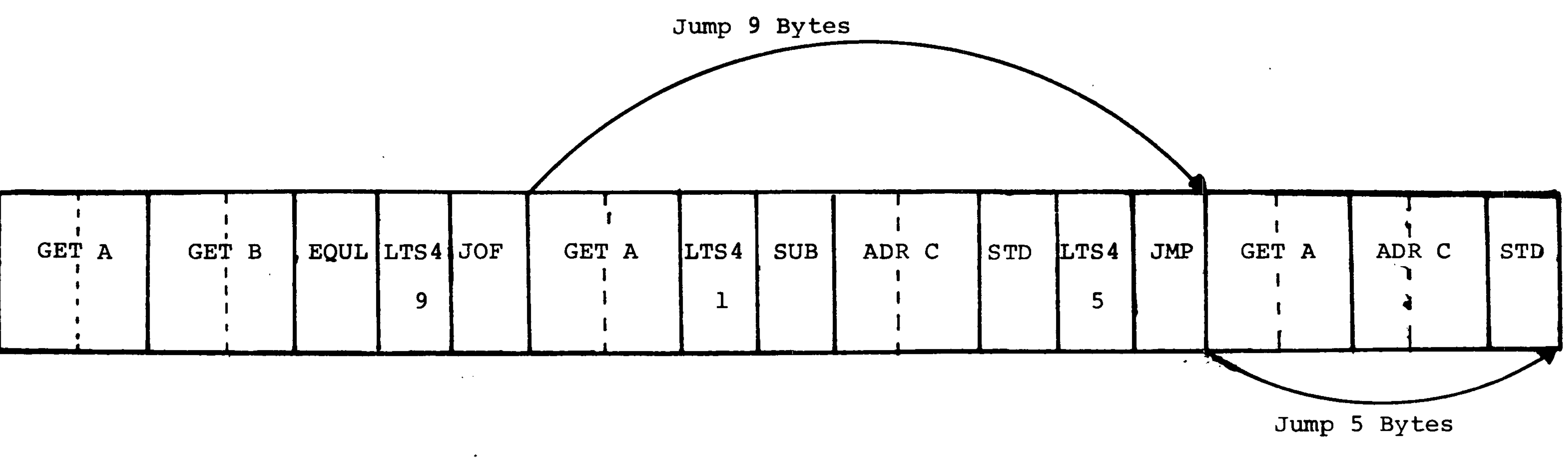


Figure 2.4-1b: Branching Relative to Instruction

of addressable variables: local variables, variables in the next procedure level containing the local level, ... to the outermost global level. Each procedure is therefore given a lexical level corresponding to how global it is. If a procedure level is of lexical level n, then it is contained in lexical level n-1, and contains procedures of lexical level n+1. Figure 2.4-2 shows an example.

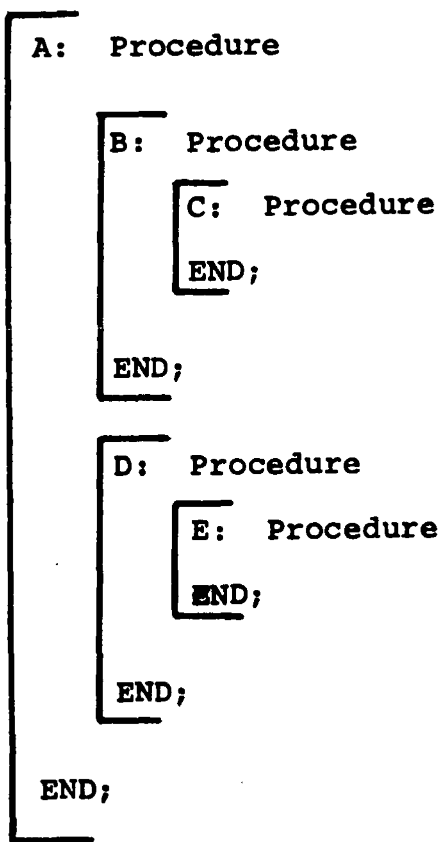
The lexical level-displacement form of addressing therefore consists of two parts, one specifying the lexical level where the operand is found, and the other a displacement relative to this lexical level to identify the particular entity. This form of addressing appears within the MP instruction architecture only in the operand meta-operators as compiled into the program segment. Upon execution of an operand meta-operator, this form of address is changed into the stack number-offset form of address relative to the system's dynamic environment. This converted address is then either used in locating the variable concerned, or is placed directly into the stack in the form of an address word (ADW). In order to obtain the maximum range of addressing possible using the lexical level-displacement form of addressing, the operand meta-operator address is interpreted differently depending upon the current lexical level during execution. Figure 2.4-3 shows the interpretation of these addresses.

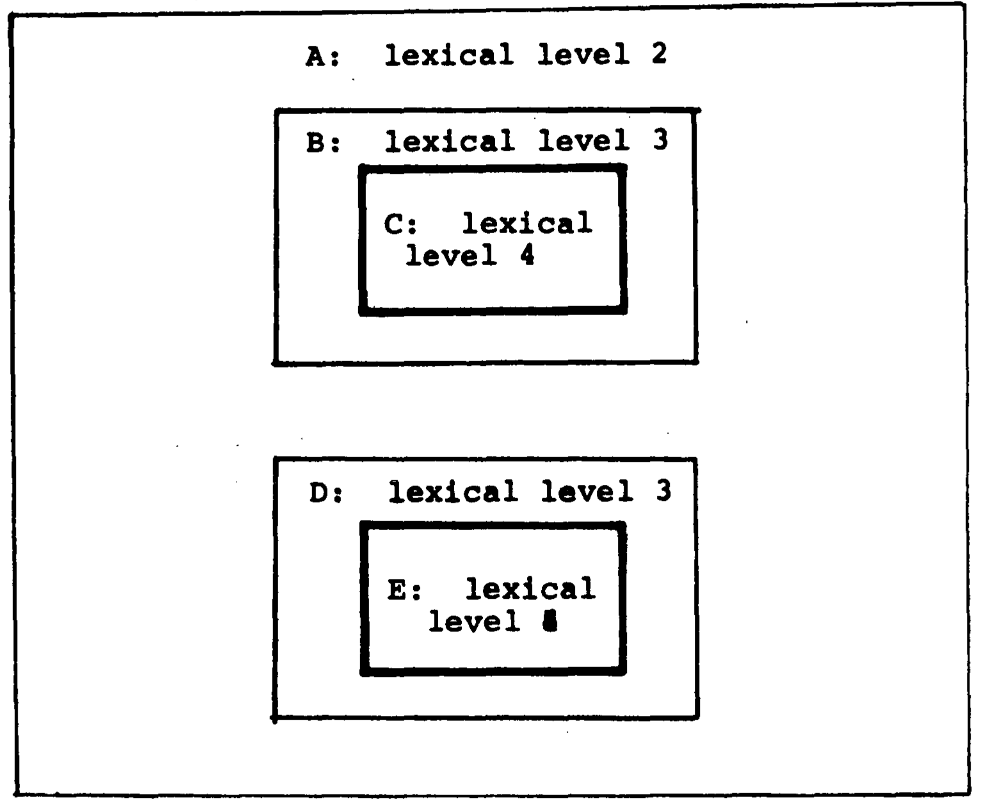
c) The third form of addressing is used to maintain the addressability of objects within the system's dynamic environment. This form of addressing makes use of a stack number and an offset from the base of the stack. This avoids having to change physical M2 addresses when segments are either moved or removed. The stack numbers are dynamically assigned by the system during the execution of a process. Addressing information within the dynamic environment, other than in a Mom descriptor, is maintained in this stack number-offset form (Figure 2.4-4).

The stack number-offset address form is 20 bits long. The first 8 bits are used to indicate the stack number and the remaining 12 bits contain the offset from the base of the stack. In order to implement descriptors of descriptors (e.g. to construct complex data entities such as STRUCTURES in PL/I or HAL) one form of the stack number-offset addressing has been implemented for

C.2

Figure 2.4-2: Example of Lexical Level Definitions





Addressing containment of procedures by definitional level in name scope languages

Statements within E may address variables in E or D or A: lexical levels 4 or 3 or 2. Statements within C may address variables in C or B or A: lexical levels 4 or 3 or 2.

1xx	$a_{12}a_{11}a_{10}a_9a_8$

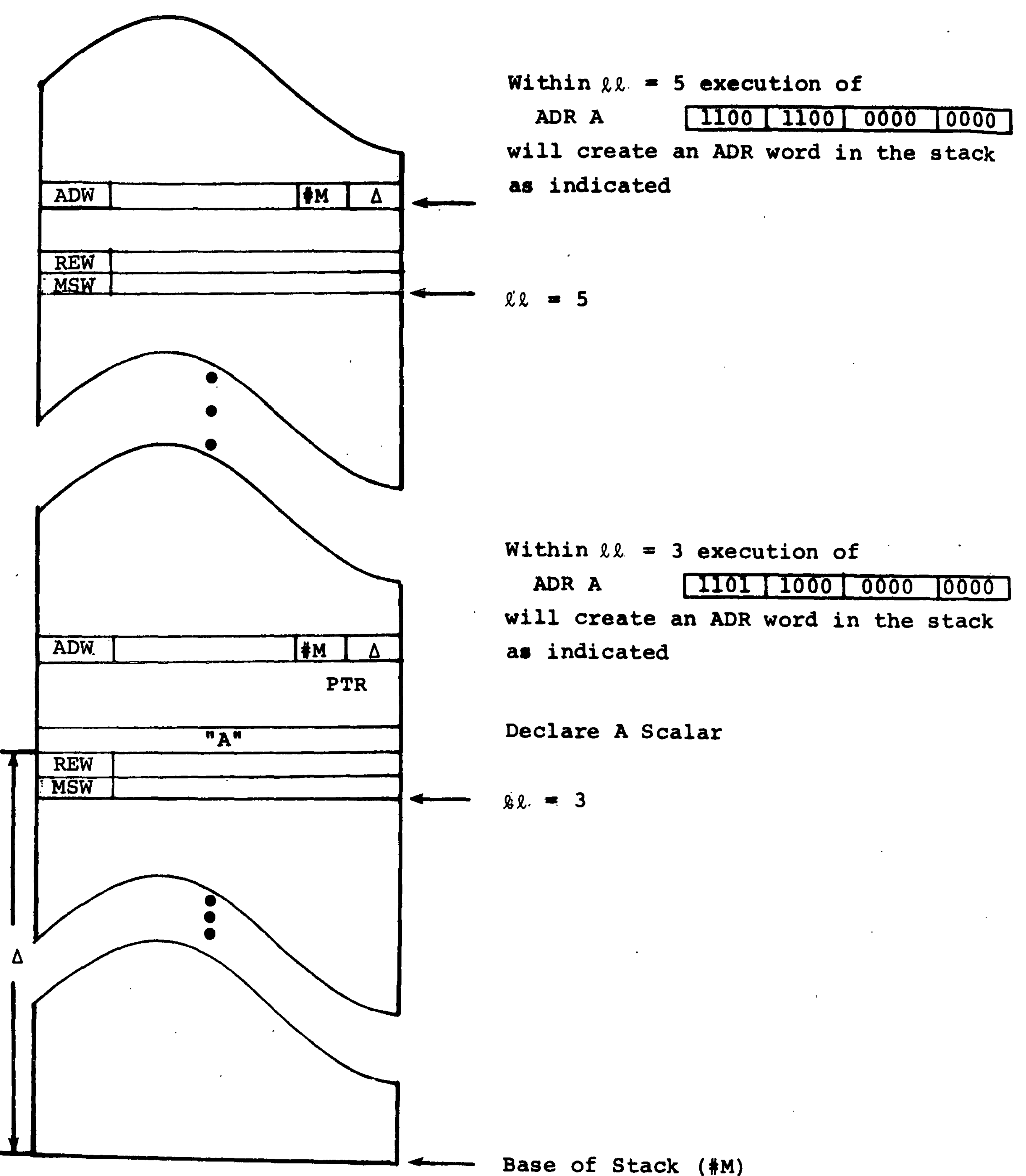
$a_7a_6a_5a_4$	$a_3a_2a_1a_0$

xx indicates which of the four operand meta-operators this is

a...a The 13 bits "a" are interpreted as the lexical level displacement addresses in the following manner:

Current Lexical Level	Lexical Level	Displacement
0-1	a_{12}	$a_{11} \cdots a_0$
2-3	$a_{12}a_{11}$	$a_{10} \cdots a_0$
4-7	$a_{12}a_{11}a_{10}$	$a_9 \cdots a_0$
8-15	$a_{12} \cdots a_9$	$a_8 \cdots a_9$

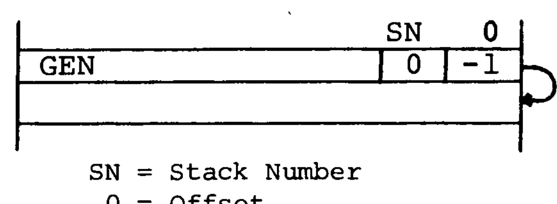
Figure 2.4-3: Lexical Level Displacement Addressing



Lexical level-displacement addressing depends upon the ℓℓ of the compiled procedure. Stack number-offset addressing is invariant within the system.

Figure 2.4-4: Stack Number - Offset Addressing

self relative addressing. When the stack number is zero, the offset field is considered to be a signed integer with eleven bits of numeric value. This offset field is then used as a relative pointer address with respect to its own location.



SN = Stack Number
0 = Offset

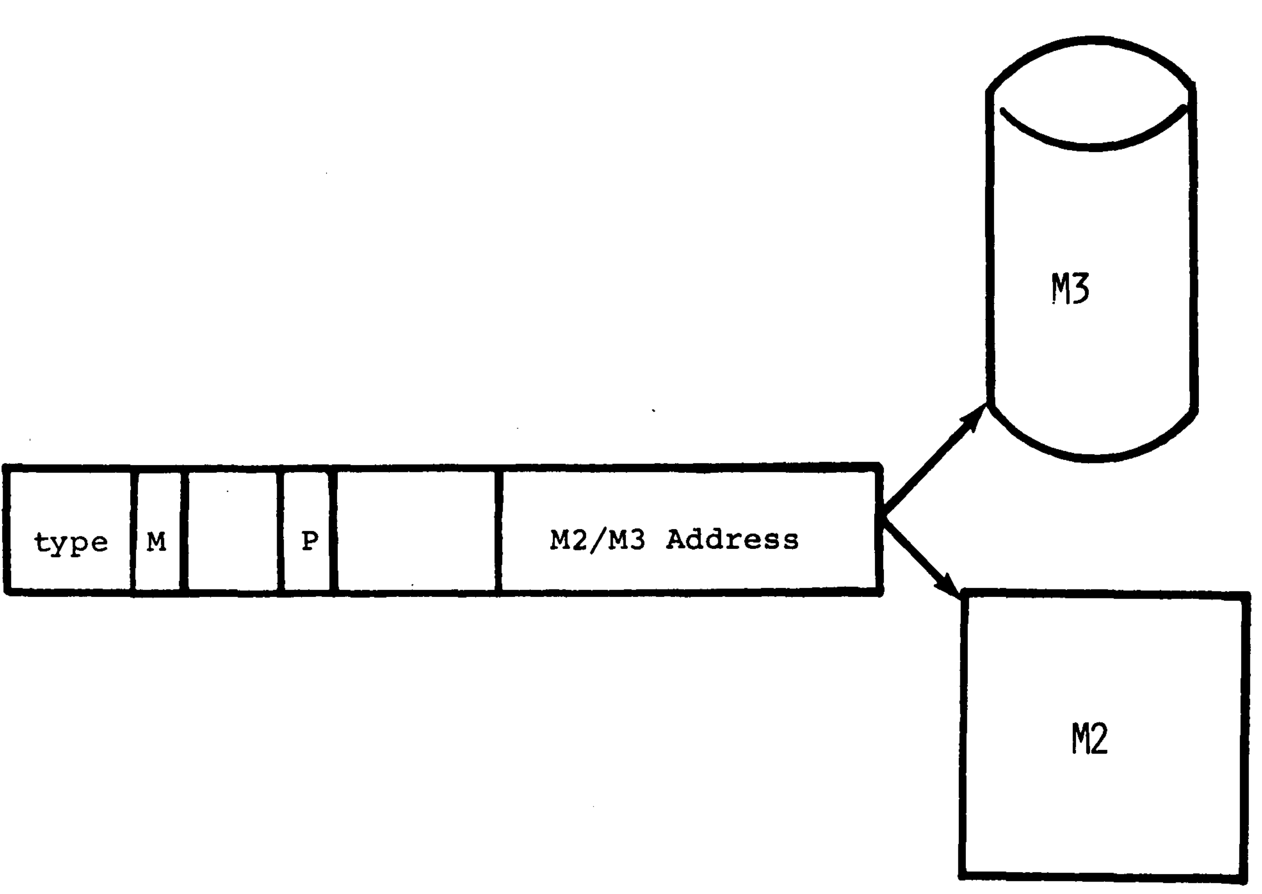
When this form of descriptor pointer field is referenced, it is changed to the equivalent stack number - offset value for usage.

d) The fourth form of addressing found within the MP instruction architecture is a physical M2 or M3 address. In order to implement a well balanced memory management system, these physical addresses have been restricted solely to Mom descriptors. There is exactly one Mom descriptor for any segment of code or data.

If a segment is within M2, the presence bit (P) of the relevant Mom descriptor is set to 1. The M2/M3 address field then contains an M2 address. If the segment is within M3, the presence bit is set to 0 and the M2/M3 address field then contains an M3 address. (Figure 2.4-5). The M2/M3 address field is 20 bits in length and hence can address one mega-unit. As envisioned within the MP design, the address field will address 32-bit units.

Figure 2.4-6 shows the addressing of an array element by a Copy descriptor. Since the Mom descriptor may not be modified, the index field of a Copy descriptor representing the element is added to the M2 address field of the Mom descriptor to obtain the element's physical address.

2.4.1.2 Stack Usage. The MP instruction architecture has been designed for Polish string, stack oriented execution. The stack in the MP design is used for control information,



M = Mom. descriptor

P = presence bit { 0 M2 address; 1 M3 address }

Figure 2.4-5: Presence Indication

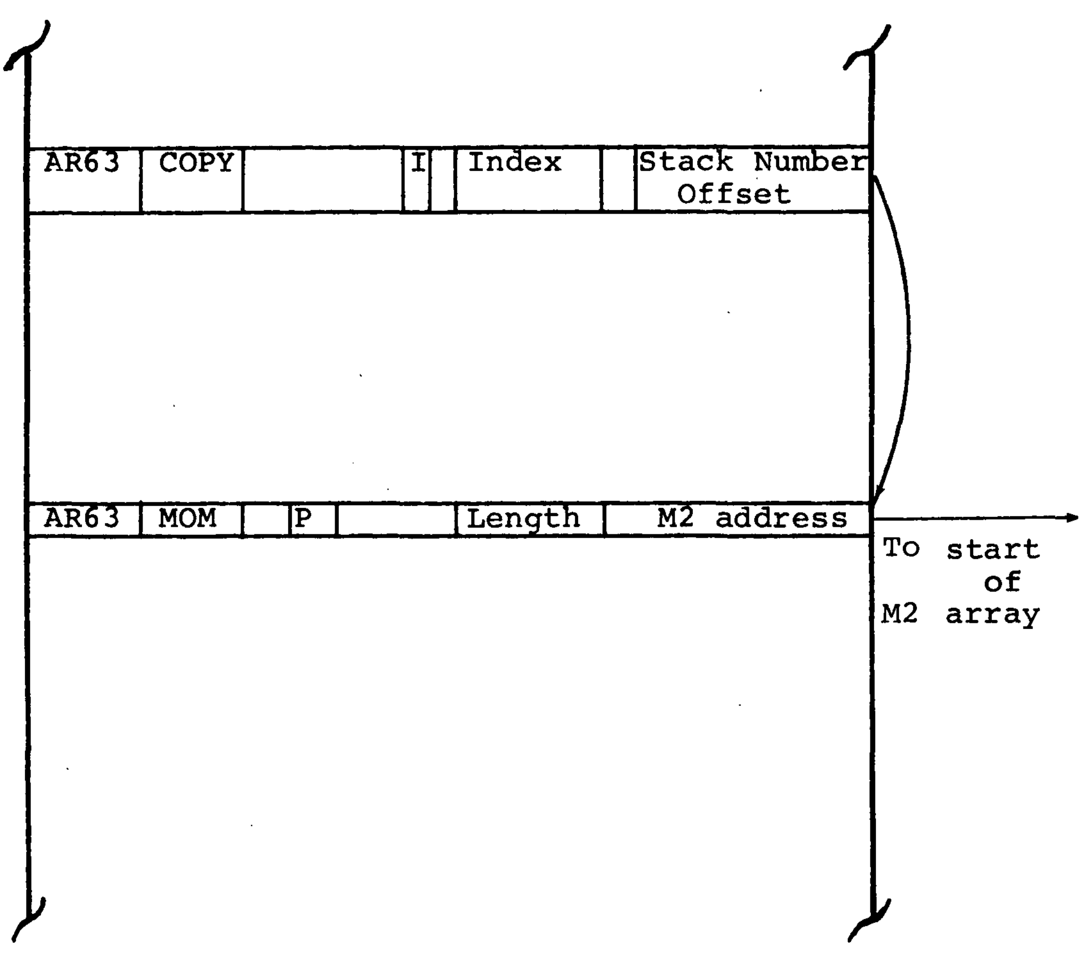


Figure 2.4-6: Addressing via a Copy Descriptor

formal parameter passage and variable definition, besides being an execution stack. The addressing of information was seen above to have been implemented within the framework of contemporary block oriented languages with name scopes in conjunction with the usage of the stack.

a) When a process is executing it must have access to the dynamic environment of its variables. But it must also have access to its procedures and systems routines (both utilities and explicit executive functions). The MP architecture implements these needs within the name scope framework of addressing. The system routines have the most global name scope and are permanently assigned to lexical level zero. System information then permanently becomes defined and primitive to the whole MP system by means of the addressing of lexical level zero with the appropriate displacement for a given primitive. These displacement values need only be known to the HOL compiler since all code must be generated by the compiler. If for any reason there were to be a redefinition of system wide primitives, addresses (i.e., changes in displacement) of all procedures would then be affected and might have to be recompiled. However, this would not occur in a working environment where the system has been defined and implemented.

   In order to address the various code segments within a compiled program, a dictionary of the various separate segments is generated by the compiler. This is then addressed by use of lexical level one, and a given code segment is referenced by an appropriate displacement with respect to lexical level one.

   The outermost level of data definition therefore begins on lexical level two. Procedures are compiled on the outermost level as lexical level two while each succeeding block containment is compiled on the succeeding lexical level. The MP architecture has provided for 16 lexical levels: lexical level 0 to 15.

b) Control Information

   Upon entry to a procedure, it is necessary to save:

   1) the return procedure linkage

   2) the return environment

3) the current name scope information.

Figure 2.4-7 shows an example of how this linkage information is contained within the stack in the MP design.

It is seen from this figure that the three necessary links are maintained in the two control words: the MSW and the REW. The STACKLINK PTR of the MSW points back in each case to the previous MSW. The ℓℓ LINK PTR of the MSW points back to the previous lexical level within the current name scope of the executing procedure. The SEGMENT PTR of the REW in each case points to the program segment of the procedure which is to be returned to. A detailed description of the MSW and REW special words is given in the section on special words (Section 2.4.2).

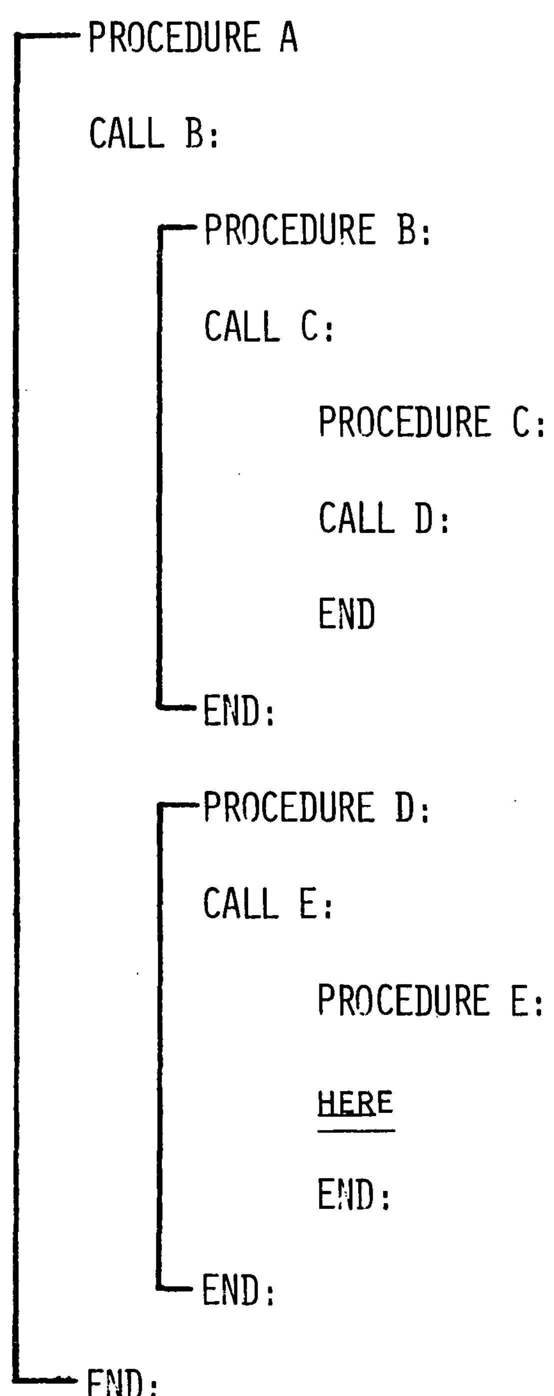
c) Formal Parameter and Variable Definition

Formal parameters are addressed within a procedure by use of the procedure's current lexical level, in conjunction with an appropriate displacement. Variables, by definition, can only be addressed if they are within the name scope of the procedure referencing them. In this case they are addressed by use of their lexical level of definition and an appropriate displacement. Figure 2.4-8 shows an example of this mechanism.

d) Expression Evaluation

The stack is also used for the evaluation of expressions. Operators (in the general case) obtain the number of operands they need from locations at the top of the stack; in turn the operator leaves the resultant value as the top of stack. Figure 2.4-9 gives an example of operator execution and its interaction with the stack.

Figure 2.4-7a: Code Example



The control information linkages within the stack are shown in the figure when control is at the statement "here".

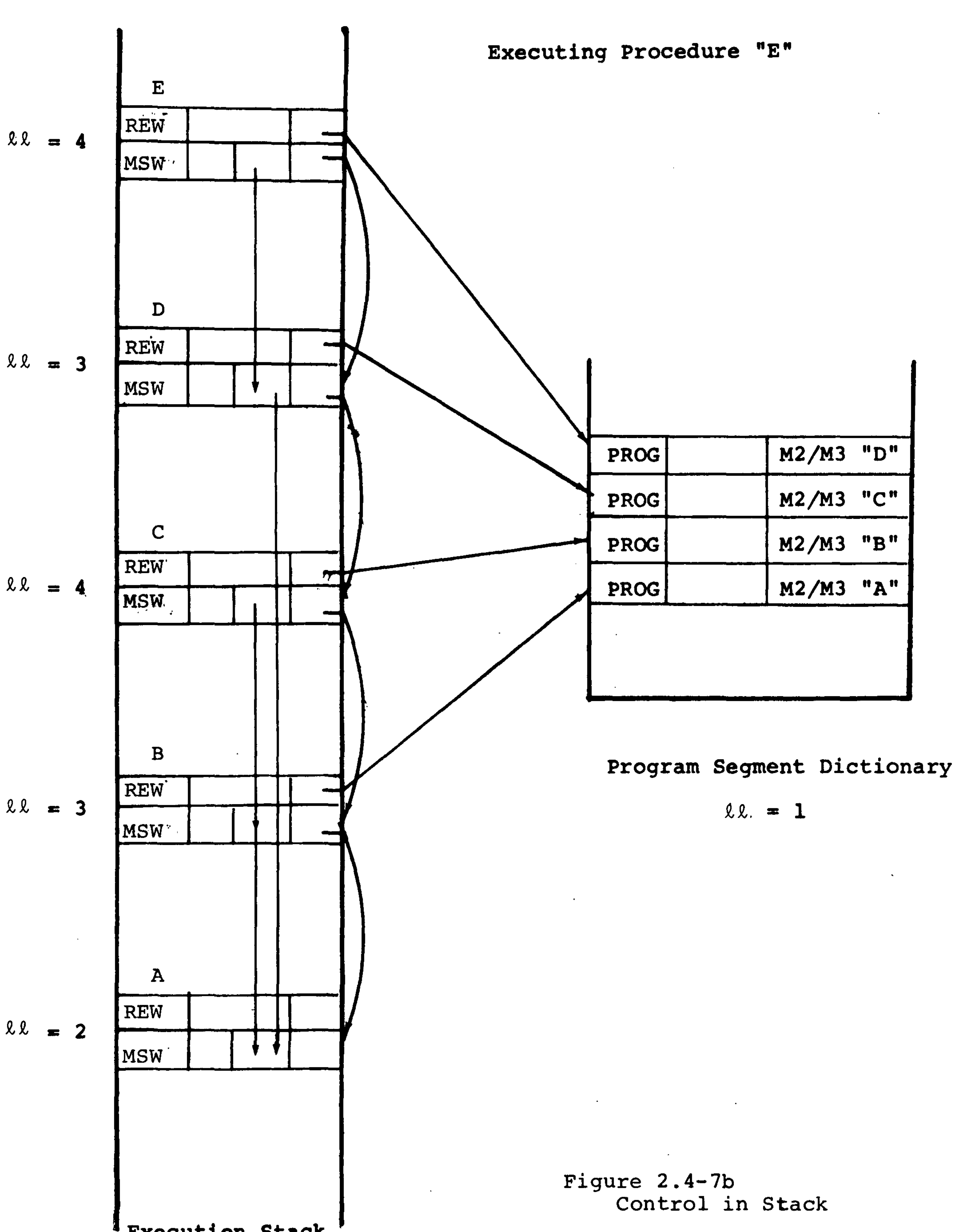


Figure 2.4-7b
Control in Stack

```
PROCEDURE A (ARG1, ARG2);
   Declare ARG1 Scalar;
   Declare ARG2 Array (5);
   Declare VAR1 Scalar;
   Declare VAR2 Array (3);
        .
        .
        .
END;
```


Figure 2.4-8: Parameter Passage

HOL Statement

A = B + C

MP Instructions

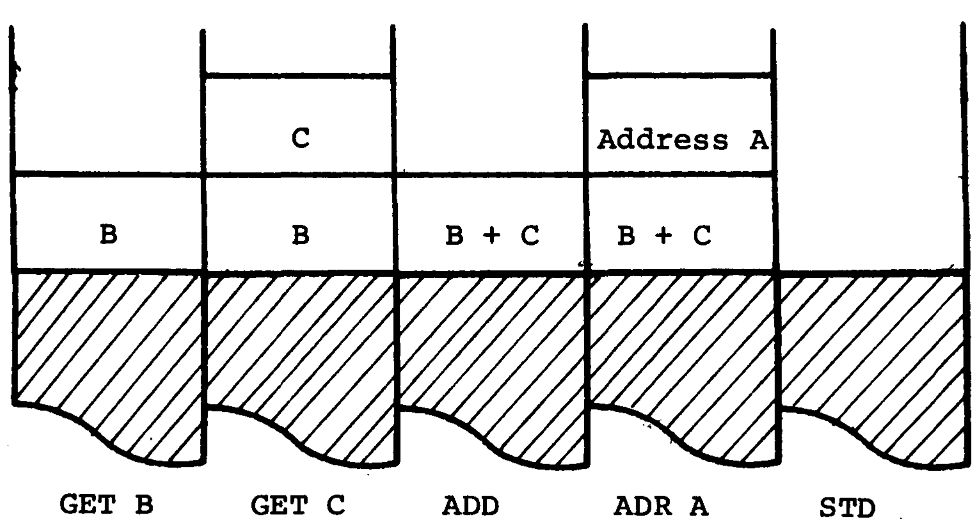
GET B

GET C

ADD

ADR A

STD



Result of Instruction Execution

Figure 2.4-9

2.4.2 Data Structures

The data structures of the MP instruction architecture contain arithmetic and character values, descriptors and special control words. The MP instruction architecture has been based upon an information width of 64 bits within the stack mechanism. Since the MP instructions are Polish operators with implied operands, they operate upon operands which are located within the stack. The use of an information width of 64 bits has made mandatory a reasonable compromise between a fully tagged architecture [52,53] with tag bits being associated with each memory word, and an untagged architecture with operand distinctions being made solely by implication of the operator.

Within the stack, values are distinguished from both descriptors and special words by use of a single bit. This bit is the value/name bit and allows the use of a maximum of 63 bits for value information within the stack. When the value/name bit does not indicate a value, the next bit then distinguishes descriptors from the special words. The succeeding three bit fields in both descriptors and special words provides a further differentiation for their special usages (see Figure 2.4-10).

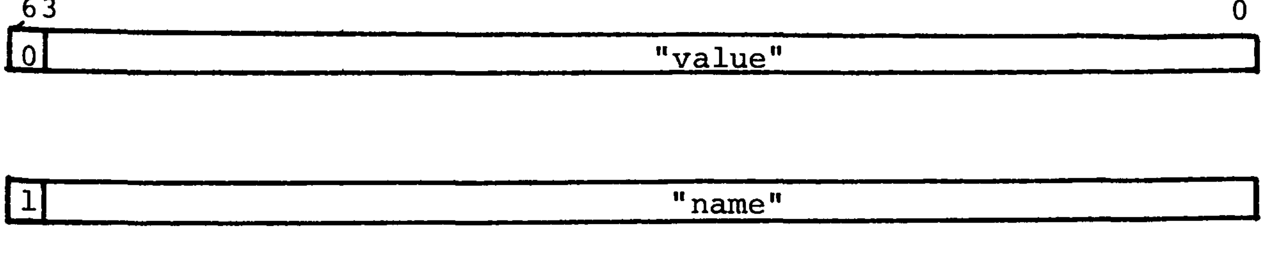
When referring to bit positions within words in the MP architecture, bit 63 is the most significant and bit 0 is the least significant bit position.

2.4.2.1 <u>Descriptors</u>. Descriptors are used primarily for the purpose of addressing arrays. These arrays may be of the various arithmetic forms, of character strings, of program segments, or of an unspecified type which must be explicitly manipulated by the instruction stream.

The use of descriptors allows the compactification of the addressing of entities within the instruction stream; they aid in the dynamic management of memory; they are able to specify detailed information about arrays which permits their usage to be dynamically verified (e.g. bounds checking); and they prevent the repetition of redundant information (precision, array length) from appearing within the instruction stream.

a) Mom Descriptors

The usage of descriptors within the MP is intimately related to the memory management and fault



Bit 63   0 indicates quantity is a value
1 indicates quantity is a descriptor of special word

63 61 58   0

10	ID	"descriptor"
11	ID	"special word"

Bits 63 & 62   10 indicates a descriptor
11 indicates a special word

Bits 61,60,59   ID identify particular descriptor or special word

Figure 2.4-10: Descriptor Identification

tolerance design. For any segment, either a program segment or a data segment, there is to be but one descriptor which contains a physical M2/M3 address. This descriptor is called the Mom descriptor and it must be referenced whenever data which it describes is to be referenced. This restriction allows for the relocation or the removal of any segment in M2 or M3 with the change affecting only the Mom descriptor M2/M3 address field.

There are seven types of descriptors. Four of them are arithmetic. These four are the:

1) AR63 descriptors used to indicate arrays of the 63 bit double precision floating point values,

2) AR32 descriptor used to indicate arrays of the 32 bit single precision floating point values,

3) AR16 descriptors used to indicate arrays of the signed 15 bit integers,

4) AR8 descriptor used to indicate arrays of signed 7 bit integers.

The CHAR descriptor is used to describe arrays of the eight bit character type. The PROG descriptor is used to indicate segments of code which are only to be executed. The GEN descriptor is used to indicate segments of untyped data. This form of segment must be explicitly manipulated by the instruction stream since type information, and hence the possibility of verification, is lacking. This descriptor is used in particular for system functions where there is a mixture of information types written within the array, (e.g., a process stack itself is a segment and must be described by a GEN descriptor.

Finally, descriptors must also be used to provide multirank information about the array indicated by the Mom. If an array is of rank one, then implied within the Mom descriptor is the fact that the first element of the array begins with an offset of zero from the M2/M3 address; the total number of elements is given by the length field; the spacing between each element is implied by the type of the descriptor (Figure 2.4-11). However, in the general multirank case, this information is not implicit and must be separately provided.

Figure 2.4-11: Descriptors

Type	M	A	D	X	▨	I	Delta	Array Offset	Limit	MOM PTR	Copy
5	1	1	1	2		1	8	12	12	20	

Type	M	A	▨	X	P	R	C	CR	▨	Length	M2/M3 Address	Mom
5	1	1	1	1	2	1	1	1	2	12	12	20

MOM PTR = Stack number - offset of associated mom descriptor

P = Presence bit: either M2 or M3 address

R = Refer bit: segment has been referred to either by reading or writing into it

C = Changed bit: Segment has been written into

CR = Critical information:

- 00: Normal, one copy stored
- XX: Critical, duplicate copies interleaved
  - 11: Both copies good
  - 01: "One" copy good, use this one
  - 10: "Other" copy good, use this one

Length = Length of segment in units of that array type ("critical" data segment twice as long as length indicates)

M2/M3 address = Physical address of the segment

Type = Data Type Form:

- AR8: eight bit arithmetic array
- AR16: sixteen bit arithmetic array
- AR32: 32 bit single precision floating point array
- AR63: 63 bit double precision floating point array
- CHAR: character array
- PROG: code segment
- GEN: untyped descriptor

M = Copy Descriptor (uses stack number-offset pointers)
Mom Descriptor (uses M2/M3 address pointers)

A = Read/write access allowed from array
Read only allowed from array

D = Single rank array, no additional rank information present
Multiple rank array, more rank information follows

X = Compool bits:

- 00: Normal non-Compool
- 10: Compool unreferenced
- 11: Compool referenced

I = Delta, array offset, limit fields refer to (sub) arrays
Delta = 0; Limit = 0; Array offset = single element index

Delta= Distance between elements in this rank
Distance in units of elements

Array offset = index into array of first element of this rank;
In units of elements starting at ϕ

Limit= Maximum limit for index into this rank in units of elements

Figure 2.4-12 defines the multirank descriptor format and shows an example of its use both when the data is declared (and hence associated with a Mom descriptor) and also when the data has been partitioned (and hence associated with a Copy descriptor).

When multirank arrays are to be used, the Length field of the Mom descriptor is still used to give the total length of the array. This implies that a separate descriptor is then needed for each rank in the multirank description. The multirank bit (D) of a Copy descriptor is used to indicate that further rank information follows. Since a Mom descriptor must be the final descriptor reference in a sequence describing a multiranked entity, it needs no bit (D). In a Copy descriptor the D bit indicates that the NEXT RNK/MOM PTR field refers to either the next Copy descriptor, if D is set, or else that there is no further rank information and NEXT RANK/MOM PTR field refers to the associated Mom descriptor.

The manipulation of the multirank information can be handled automatically by the operators of the MP in standard cases. However, there are several important points to note about multirank descriptor usage. First, "type" must be the same in all the descriptors in the sequence. Secondly, there should always be exactly enough descriptive information for the number of dimensions which currently exist. In Figure 2.4-12, M was declared to be a matrix; i.e., rank 2. Since the Mom descriptor does not contain rank information, two further Copy descriptors are needed in order to fully describe the matrix. When, as in this example, the matrix is partitioned, it still has rank 2, and therefore information for two dimensions must be given. Since the Copy descriptor does contain rank information, only one additional Copy descriptor is needed to describe the modified matrix. Therefore, the number of descriptors needed for any given rank are seen to be:

Type	00	D	(unused)	I	RNK Delta	RNK Offset	RNK Limit	Next RNK/MOM PTR
5	2	1	3	1	8	12	12	20

Type: Multi-rank information descriptor, type as in Figure 2.4-11

D = Last rank information: PTR is A MOM PTR
More rank information: PTR is A NEXT RNK PTR

I = RNK Delta, RNK Offset, RNK limit fields refer to (sub) arrays
Indexed: RNK Delta = 0, RNK Limit = 0; RNK Offset = single element index

RNK Delta = Distance between elements in this rank;
Distance in units of elements.

RNK Offset = Index into array of first element of this rank;
in units of elements starting at 0.

RNK Limit = maximum limit for index into this rank in units of elements

---

Example of Multi-rank Declaration and Usage

M*2 to 3, 3 to 5

AR63	0	0	0		0	1	2	3	MOM PTR
AR63	0	0	1		0	6	6	2	Next RNK PTR

Declare M matrix (3,6);

AR63	0	0	0		0	1	0	6	MOM PTR
AR63	0	0	1		0	6	0	3	Next RNK PTR
AR63	1	0			P R C CR			18	M2/M3 Address

Figure 2.4-12: Multirank Descriptor Format

Rank	# of Mom Desc.	# of Copy Desc.
0	1	0
1	1	0
2	1	2
.		
.		
.		
n	1	n

Mom Addressed if Possible

Rank	# of Copy Desc.
0	1
1	1
2	2
.	
.	
.	
n	n

Copy Created and Addressed

The access bit (A) is used to indicate that the array elements are for either read or write usage, or are restricted to read only. While this mechanization is valuable for the implementation of user program protection, it must be capable of being overridden by a system program. During execution, any data referred to via a Copy descriptor will have the access privileges of the most restrictive form found in its path to the data.

The Mom descriptor is the only object used in execution that contains an actual reference to an M2 or M3 address. The presence bit (P) of the Mom descriptor indicates whether the M2/M3 address field refers to M2 or to M3. A data descriptor which has not yet been referenced which also has an M3 address of zero is considered not yet to have had storage allocated for it. Upon this first reference, Memory Management will allocate to it an appropriate area.

The referred-to bit (R) is used by Memory Management. Whenever an element of the segment referred to by the Mom descriptor is referenced, this bit is set. The changed bit (C) is similarly used to indicate that an element within the segment has been changed; that is, the segment has been written into. With the use of these two bits it is possible both to discover occurrence of use and change within the segments, making possible an efficient yet intelligent Memory Management policy.

Descriptor	Element	Byte Size of Addressed Element	
AR63	63 bit floating point	8	double word
AR32	32 bit floating point	4	word
AR16	signed 15 bit integer	2	half word
AR8	signed 7 bit integer	1	byte
CHAR	character	1	byte
PROG	instructions	8	double word
GEN	anything	8	double word

Figure 2.4-13: Descriptor Types

One of the prime concerns of the MP architecture is with fault tolerance. While all program segments which reside within M2 have a duplicate copy which can be found in M3, this is not true of the data segments. It would be possible to provide check points so that all variable data at the instant of the check point were saved in M3 (in case of a failure before the next check point). However, this is costly in execution of a check point with M3 storage, in the recovery time, and in the design and implementation of the procedures necessary to minimize the amount of M3 usage. (See Chapter 4 for a more detailed discussion.)

The MP architecture uses a different technique to save redundant data. Whenever data is judged to be critical (which must be so specified in the compiled program) it will be dynamically stored in a dual redundant form. The actual storing in this mode depends to some degree on the width of the interleaved M2 memory modules implemented. Figure 2.4-14 shows an example based upon an access width of 32 bits in the M2 memories. The data is stored into the interleaved modules, alternately one copy then the other until all the array has been stored. This method of critical data storage requires:

1) that there are at least two interleaved modules,

2) that the number of interleaved modules is even,

3) that the failure modes of each of the interleaved modules are independent.

The CR bits of the Mom descriptor are used to indicate critical data. If CR = 00 then the information is non-critical and is saved in the normal fashion with only one copy. If the data is critical, CR = 11, and the data is stored in the dual redundant mode. If one of the memory modules actually fail, the operating system is able to turn off one of the CR bits, leaving either CR = 01 or CR = 10. In either of these two cases, the processor will then use only the single copy which is indicated by the CR bits to be good. It is then but a policy matter, depending on the convenience and real time needs of the process, when to move the data into good memory, or when to reconfigure the memory system.

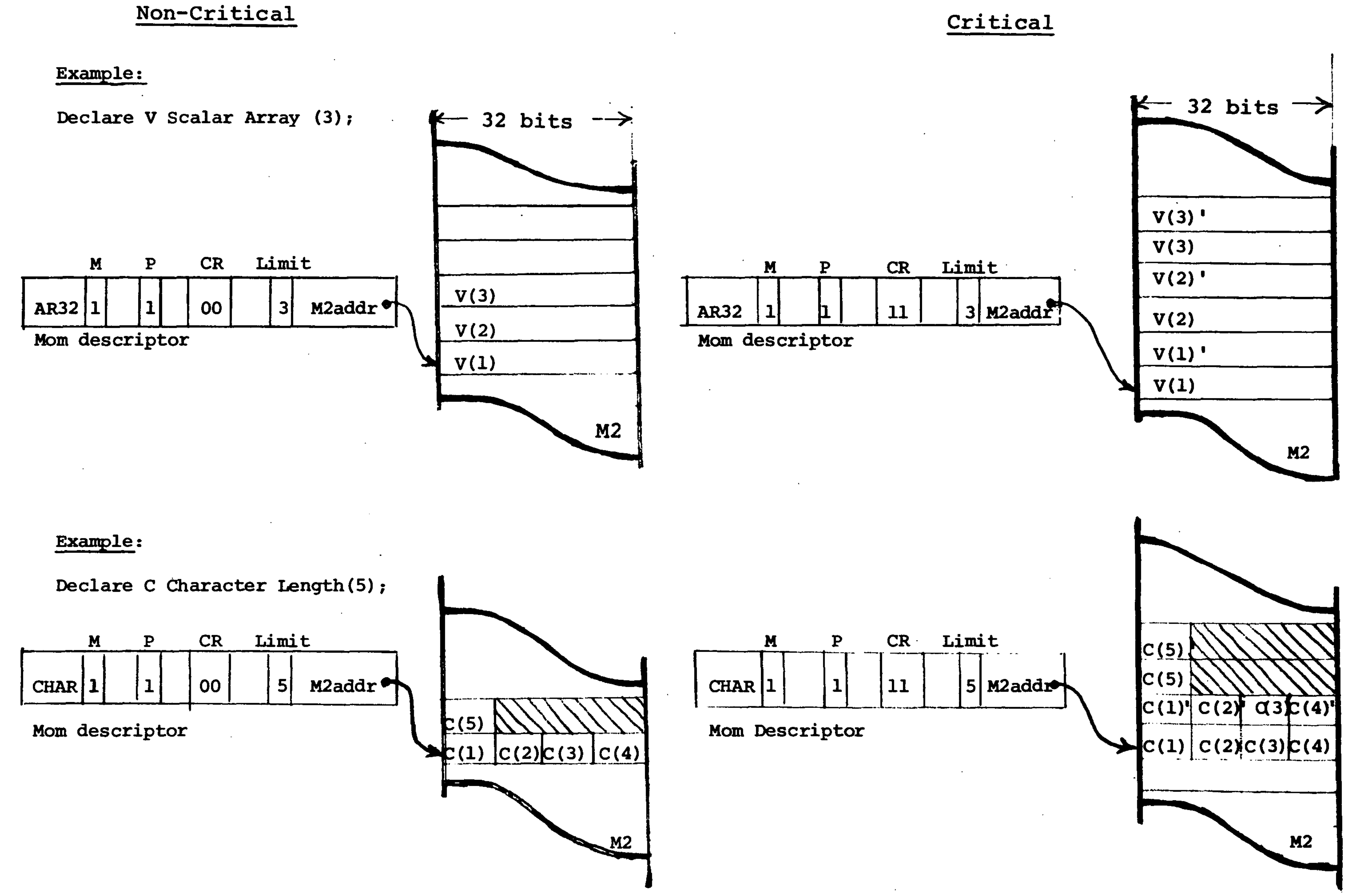


Figure 2.4-14: Redundant Storage of Critical Variables

In addition to the presence of the critical bits within any data descriptor, each process should itself be able to be declared critical or non-critical. If the whole process has been declared critical, then all dynamic storage will be created critical when first referenced and its critical bits set within the descriptor. This explicit declaration of a process as critical allows for the use of a single system routine (e.g., the SIN function) compiled in the non-critical mode, while still guaranteeing the critical storage of data during its execution by a critical process.

Compools in the MP architecture are bound to a program upon being first referenced. This necessitates the identification of Compool descriptors (or the descriptors of any unresolved entities) within the stack. The X bits are used for this. When there is no need for dynamic binding, these bits are set to 00. When the Compool has yet to be referenced, the X bits have a setting of 10. When the Compool reference is resolved by the executive the bits are set to 11, and the pointer field then directly points to the start of the proper Compool stack.

b) Copy Descriptors

Whenever a descriptor has to be changed other than by Memory Management; e.g., to obtain a sub-array or to index an element of the array, a Copy descriptor must be made of the Mom descriptor. The initial difference between Copy and Mom descriptors is in their addressing field: while the Mom descriptor points (by definition) to the M2 or M3 physical location where its segment is located, the Copy descriptor must point to the Mom descriptor of which it is a copy. The Copy descriptor's address-field is then in the stack number-offset form and points to the Mom descriptor.

Since the Mom descriptor is truly a descriptor of a segment and cannot be changed, any manipulation which changes a descriptor must be performed upon a Copy descriptor. The MP instructions have been designed to allow a relative ease in the manipulation of arrays, both of a single and multiple rank. In a multiranked entity, it is necessary to maintain at least three quantities of particular interest to each rank. These are the number of elements in the given rank, the offset of where in the whole array the first element of this

rank is located, and the spacing between each succeeding element within the given rank. These three quantities have been implemented in all Copy descriptors by the LIMIT, ARRAY OFFSET, and DELTA fields respectively. The LIMIT field length has been chosen to be twelve bits, which corresponds to the Mom descriptor's LENGTH field and allows for the addressing of 4095 array elements. The ARRAY OFFSET field must, of course, be of the same size in order to allow for array partitioning (sub-arrays). The DELTA field is eight bits in length and allows for a separation of 255 elements between the elements of any given rank. This is more than sufficient for any reasonable form of multiranked entity. If the array has but a single rank, then the spacing would be one element and the DELTA field would be one. In a three-by-three matrix, column vector elements are separated by a spacing of three elements and therefore DELTA would be three.

The most common array manipulation within current HOLs is to index the array to obtain a single element. Since all changes in descriptor information must occur within a Copy descriptor, an index bit (I) is used to indicate that indexing of the array has occurred. When indexing has occurred, the ARRAY OFFSET has been appropriately set to the index of the indicated array element. Since further indexing or partitioning within this rank of the array cannot occur, the LIMIT and the DELTA fields are set to zero.

The Copy descriptor also contains the D bit, which indicates multirank entities. If D is set then the MOM PTR field points to the descriptor which gives the next rank information. This rank information has been initially obtained via the descriptor associated with multiranked Mom descriptor. However, these descriptors are Copy descriptors (address fields are stack number-offset), and can therefore be manipulated to reflect both indexing and sub-arraying.

Copy descriptors also contain the access bit (A). This allows copies to be more restrictive in their access privileges than the Mom descriptor.

2.4.2.2 Special Words. The special words used in the MP instruction architecture are for information needed to control the flow of program execution (see figure 2.4-15).

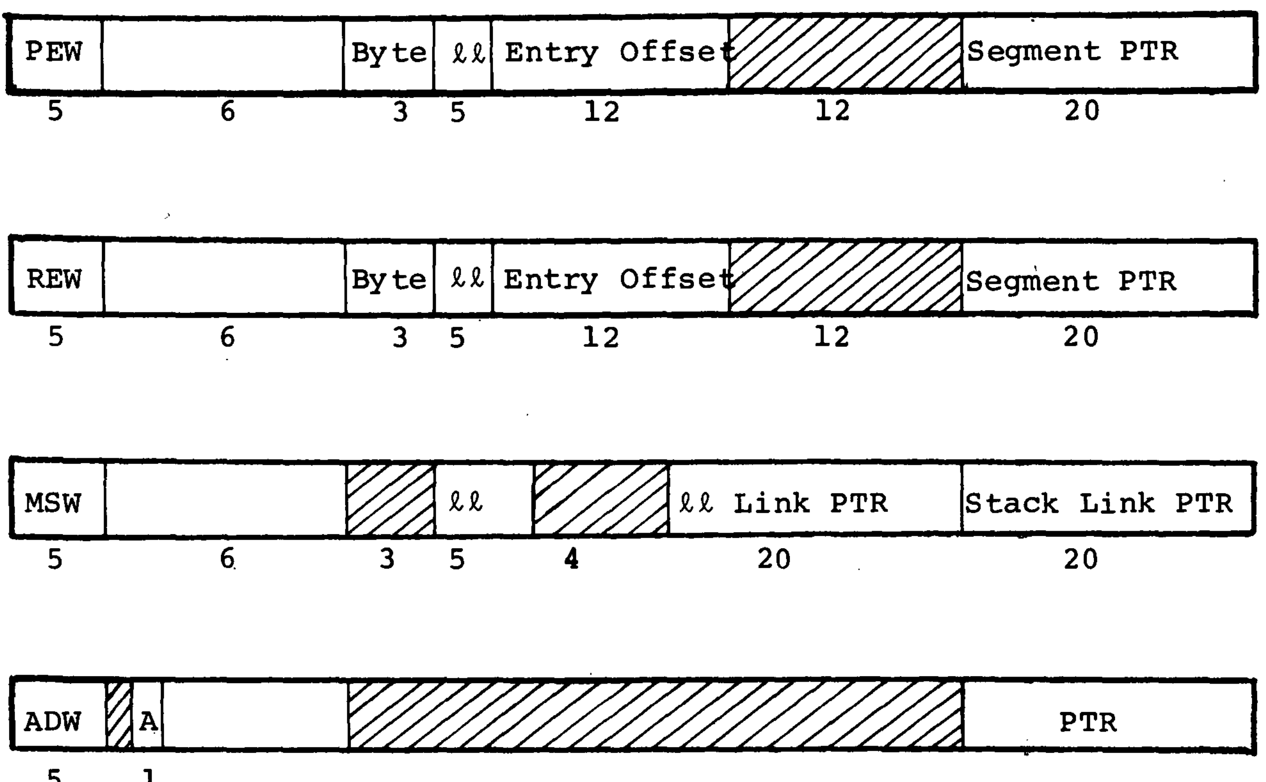
a) The program entry word (PEW) indicates the entry point into a program segment. This allows for the possibility of multiple entry points into a given program segment. Since the lexical level-displacement form of addressing found within a code segment is dependent upon the lexical level of compilation and the current name scope for proper interpretation, it is necessary that the lexical level of the procedure to be entered be identified. Since the offset into a program segment is given in double words, an extra three bits are necessary to identify the byte address within a double word of the first operator to be executed upon entry to a procedure. The last field in the PEW is a pointer to the appropriate program segment to be executed. This is given in the standard stack number-offset form.

b) The return entry word (REW) contains similar information to the PEW. It contains a pointer to the appropriate program segment, the double word and byte displacement for the return entry point, and the lexical level of execution upon return. This word, however, instead of being statistically compiled, is dynamically created whenever a procedure is entered to save the appropriate linkage information with which to return.

c) The mark stack word (MSW) maintains the static and dynamic environments of process execution. When a procedure is called, its lexical level is placed into the MSW and pointers are created to the previous MSW, in order to link the dynamic environment and to point to the previous MSW of the static name scope of the procedure. The linkage to the "previous" MSW is a simple matter of storing the address of the current MSW when the new MSW is created. The static name scope pointer is found to be that of the least name scope which the two procedures have in common. This is the lexical level at which the PEW for the new procedure was known by the calling procedure. The RUPT field is used by the software to designate interrupt response override (see Section 3.3).

d) Information may be addressed dynamically by use of an address word (ADW). This contains the stack number-offset form of address of the entity referred to. It

Figure 2.4-15: Special Words



PEW: Program Entry Control Word

ℓℓ: Lexical level of procedure to be entered
Segment PTR: Stack number-offset of program segment
Entry Offset: Double word offset within program segment to which to transfer control
Byte: Byte identification within double word entry offset

REW: Return Entry Control Word

ℓℓ: Lexical level of procedure when control is returned
Segment PTR: Stack number-offset of return program segment
Entry Offset: Double word offset within return program segment to which to return control
Byte: Byte identification double word return entry offset

MSW: Mark Stack Control Word

ℓℓ: Lexical level of indicated procedure
Stack Link PTR: Stack number-offset of previous MSW
ℓℓ Link PTR: Stack number-offset of previous lexical level MSW

ADW: Address Word

A: Access Bit: either read/write or read only allowed
PTR: Address pointer in stack number-offset representation

is also possible to set the access bit (A) in an ADW in order to make the referred-to information read only.

2.4.2.3 Data. The MP instruction architecture recognizes two basic forms of data: arithmetic and character.

a) Arithmetic Types

All quantities within the stack are maintained in a 64 bit form. Four basic arithmetic types exist within the MP architecture: a double precision floating point form of 63 bits, a single precision floating point form of 32 bits, a signed fifteen bit integer, and a signed seven bit integer. The forms which they take along with the transformation which occurs when they are loaded into a stack are shown in Figure 2.4-16.

Since all quantities within the stack must be maintained in a 64 bit form, the arithmetic types are always maintained as 63 bit floating point quantities. Bit 63 is used to indicate that the quantity in the stack is a value. The other arithmetic types can only be maintained by the use of the appropriate descriptor.

The exponent refers to the base two and is maintained in a sign and magnitude form. The mantissa is also of signed magnitude form and the radix point is considered to be to the right of bit 0. This form of arithmetic allows an integer to be a proper subset of floating point in such a way that when the exponent is all zeros the low order bits represent an integer in a straightforward manner. This immensely eases literal creation and usage.

b) Character Types

All characters in the MP instruction architecture are eight bits in length and must be addressed via a descriptor. The actual bit form is not specified but will vary depending upon actual implementation. For example, the code may be either ASCII or EBCDIC. This will depend upon the convenience of the other computer systems with which the MP must interact. The only effect upon the MP instruction architecture is in the character to/from data convert instructions. This only need be specified at implementation time depending upon the constituent components then available.

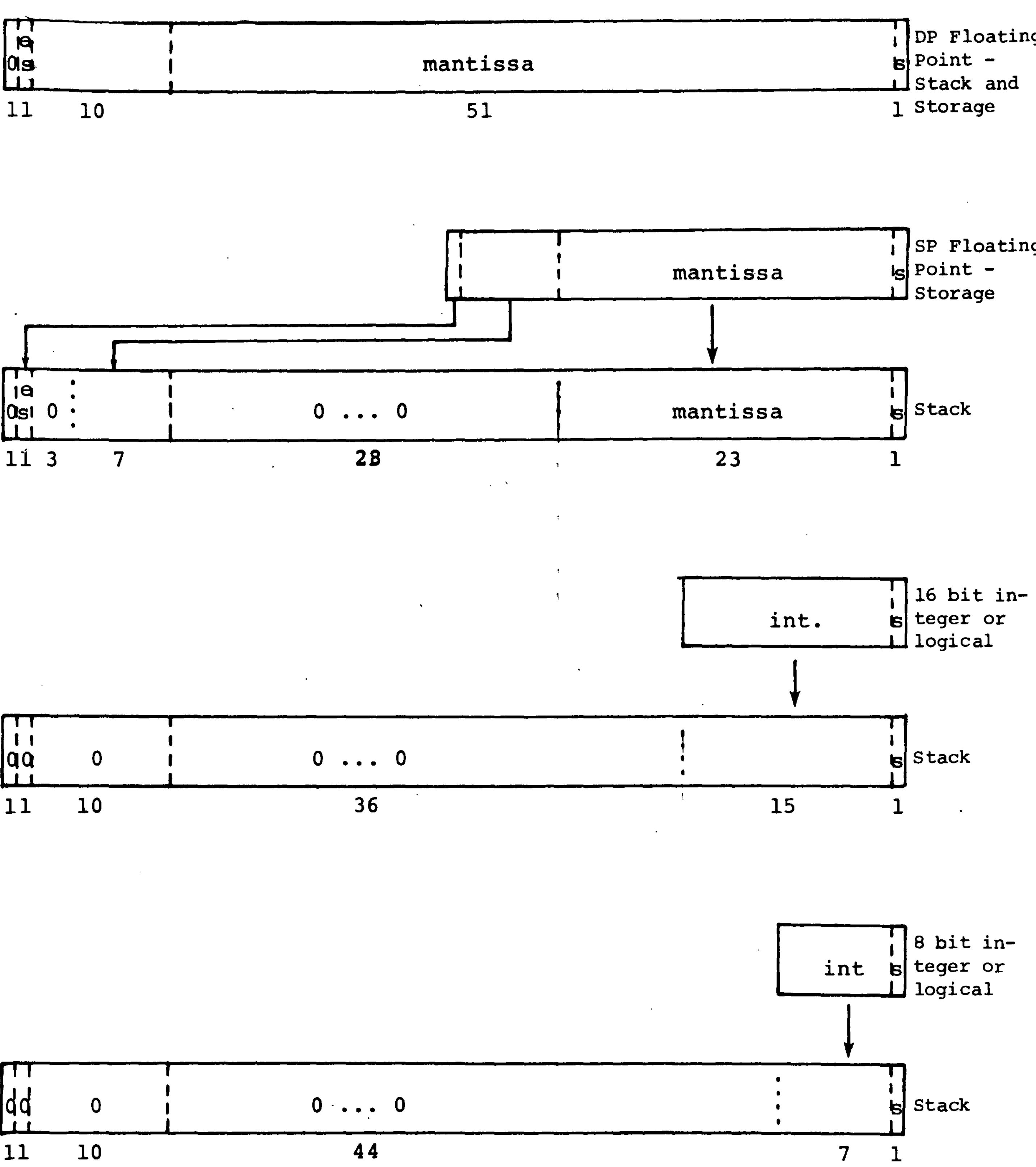


Figure 2.4-16: Arithmetic Type Formats and Mapping to Stack

	Exponent Range		Integer Precision	
	Bits	Base 10	Bits	Digits
63 bit floating point	11	±307.2	51	15.3
32 bit floating point	8	± 38.4	23	6.9
16 bit integer (incl. sign)	--	0	15	4.5
8 bit integer (incl. sign)	--	0	7	2.1

Integers and mantissas are represented in signed-magnitude form

Exponents are all sign-magnitude to the base two

Figure 2.4-17: Arithmetic Type Precision and Range

2.4.3 Instruction Set Description

The design of the MP instruction architecture has been based upon Polish instructions with implicit operands This basic principle has been modified where experience and judgement have indicated some moderation can enhance performance. In the Polish string there are two basic entities to separate within the execution sequence: operators and operands.

Operands, as implemented, take two forms: either they refer to variables or else they refer to literals. Literals have been implemented in a way to most compact the code according to currently known statistics of their usage, and are found completely within the program segment itself. Variable operands have been designated as operand meta-operators since they are logically operands, but actually manipulate the stack.

The operators in general obtain their operands from the appropriate number of locations at the top of the stack. The stack locations in the description of the operators and operands will be called T for the top of stack, T2 for next to top of stack, T3 for the one below T2, and so forth. The design of the particular set of operators designated depends on the fact that the top registers of the stack are implemented in hardware, and that this number of locations is at least eight (T, T2, T3 ... T8). Unless otherwise stated, the operand used by an operator in the following descriptions is purged from the stack upon execution of the operator, e.g., the ADD operator adds together the values of T and T2, purging both, and places the resultant value into T.

2.4.3.1 Automatic Fetch and Automatic Store. The differentiation between names and values is maintained within the stack. This allows the execution of operators to perform automatic fetching and/or automatic storing if the operand so indicates. While it is possible for the execution of an operator to follow a chain of addresses for an indeterminate length, this would have very deleterious effect upon the execution time characteristics of the MP. With the appropriate implementation of a HOL compiler the indirection should contain no more indirection than one ADW with a possible following descriptor. The main cause of indirection when a HOL is used is in the passage of formal parameters. When a parameter in a called routine is in turn a "call by reference" parameter in the calling routine, many HOLMs would implement

this by the generation of an extra level of indirection. In the MP instruction set the Copy meta-operator has been explicitly provided for this purpose. If the parameter to be passed to a called routine is a parameter to the calling routine, then the Copy meta-operator will pass this information, whether value, descriptor or ADW without any extra indirection (Figure 2.4-18).

While the automatic fetch function and the equivalent automatic storage function are extremely useful in the implementation of a HOL, it is essential that the Operating System be able directly to manipulate any entity as an object irrespective of its value/name function. For example, in the overwriting of information pertinent to memory management, and in the initialization of information which may otherwise have extraneous tag bits in the given memory cell. To this end the STDI and STNI operators have been provided for overriding the automatic store feature of the architecture. These instructions will store destructively or non-destructively immediately to the address indicated without any further level of indirection.

2.4.3.2 Operand Meta-Operators. There are four operand meta-operators used for the fetching of operands, or for creating an address pointing to them.

COPY The lexical level-displacement address of the operator is changed into a stack number-offset address. The contents of the location thus addressed is placed into T.

GET The lexical level-displacement address of the operator is changed into a stack number-offset address. The indicated value is fetched. If the entity addressed is not of rank zero, its descriptor is reduced to rank zero by using stack-top entries as index values until the rank is zero. The value is then placed into T.

ADR The lexical level-displacement address of the operator is changed into a stack number-offset address. An ADW with this address is placed into T.

ADRE The lexical level-displacement address is changed into a stack number-offset address. If the

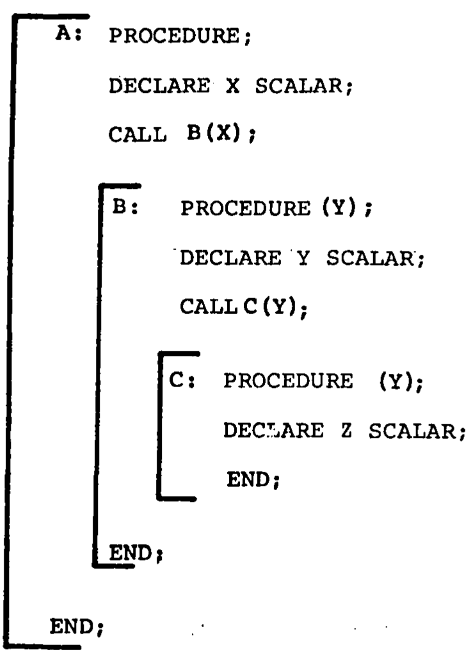


Figure 2.4-18a: Example of Code

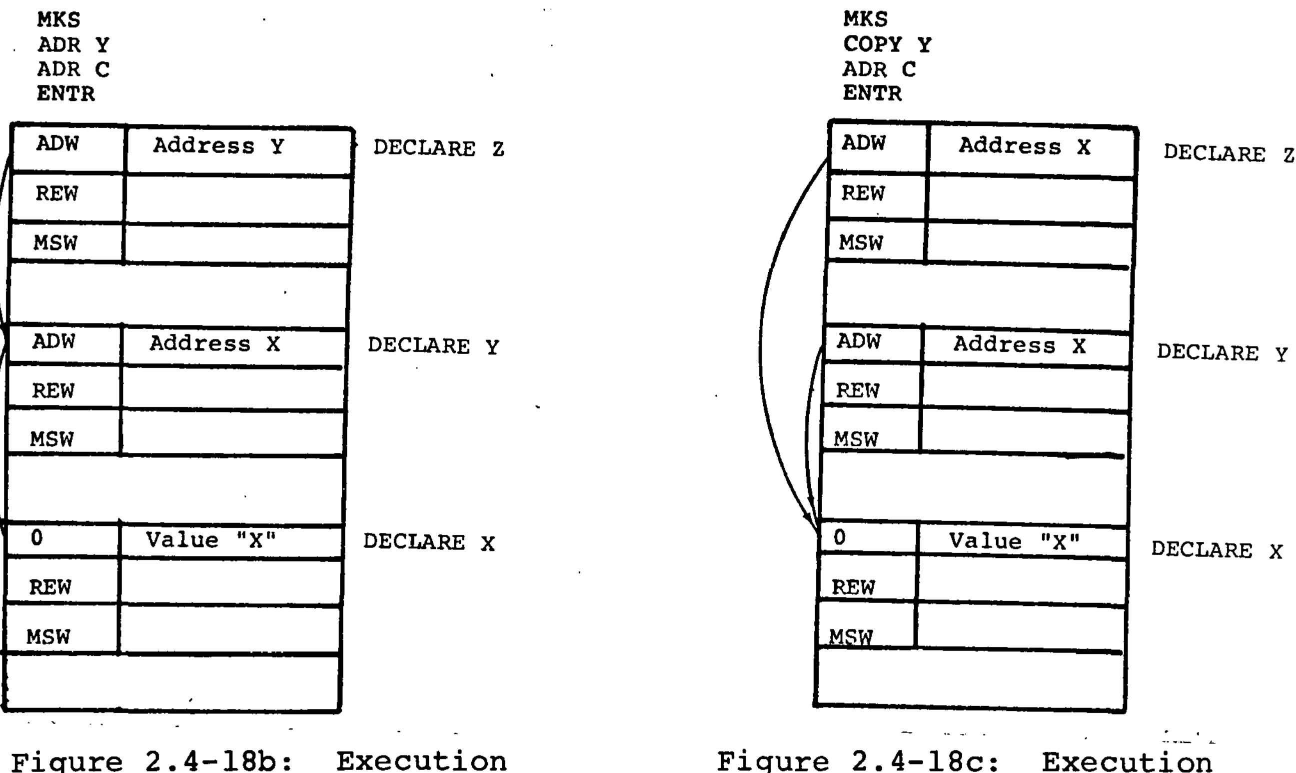


Figure 2.4-18b: Execution Sequence for CALLC(Y) Without COPY

Figure 2.4-18c: Execution Sequence for CALL C(Y) With COPY

addressed entity is a descriptor, the descriptor is fetched. The descriptor is reduced to rank zero by using stack-top entries. The final indexed descriptor of an element is then placed into T.

2.4.3.3 Array Operations and Memory Operations. Current HOL usages cannot accurately reflect the need or desire for array operations. It is not possible to obtain probable array usage statistics since current HOLs do not, in general, have array operations of any significance. The MP has been provided with several basic storage, arithmetic, and logical operators which can be used in array manipulation.

The main difficulty in the implementation of array operators is the need for large temporaries, with corresponding allocation and de-allocation requirements. In the case of character string manipulation, these basic operations can be successfully handled with low overhead by the automatic maintenance of temporaries. The method employed is fully discussed below in the section on Data Field Manipulations (2.4.3.10). General array operators, such as the addition of two arrays, are not amenable to as easy a solution, particularly when fault tolerance must be maintained.

In order to avoid complexities in the implementation of temporaries, all operators which work upon arrays must necessarily indicate a receiver for the result. Therefore, all of the store operations are able to work with arrays. The MP dyadic arithmetic operators also have been provided with a triadic form, where the third operand is the receiver. Triadic operators also exist for the logical "and", "or", and "exclusive or" operators.

The array operators may operate on various combinations of operands. The operands may all be single valued, as is the normal case. The operands may all be arrays, in which case the rank and dimensions of each of the operands must match. There may be a combination of arrayed and single element operands. In this case, the receiver operand must be arrayed and either one, both, or neither of the input operands may have the same rank and dimension.

All of the store operators (2.4.3.6) may act as array operators. Those arithmetic and logical operators which are allowed to be array operators are implemented by having a

prefix syllable to the appropriate operator. These prefix syllables are of two forms.

STT — T2 and T3 provide the operands for the operator immediately following STT. The result of that operation is stored into the location indicated by T.

STT3 — T and T2 provide the operands for the operator immediately following STT3. The result of that operation is stored into the location indicated by T3.

Figure 2.4-30 provides a list of those operators allowed to appear after STT or STT3. The usage of STT versus STT3 depends upon the parse algorithm and temporary usage policy of a given HOL compiler.

Besides providing the possibility for array operations, STT and STT3 are defined to lock out of memory the other processors until the completion of the operation, when the input operands have single element values. This allows the contents of a memory location to be obtained, manipulated and restored as an integral operation without the interference of other processors. This is in effect a lock with a memory operator and is extremely useful in a MP system.

2.4.3.4 Literals. Literals are used for loading into the stack information which is known at compile time. This information can be either for numeric values, bit constructions or for control flow. There are two basic forms of literals: one loads the literal information immediately into T from the program string, while the other has an address relative to the operator from where the literal is to be loaded into T.

LTS4 — load the indicated signed four bit integer value into T

LTS10 — load the indicated signed ten bit integer value into T

LTS15 — load the indicated signed fifteen bit integer value into T

LT32 — load bits 0 to 31 of T with the indicated 32 bit value. Set bits 32 to 63 of T to zero.

LT32F — load the indicated 32 bit single precision floating point scalar into T

LT64 — load the indicated 64 bit value into T

LTS7M — load the indicated signed seven bit integer values into each succeeding T. The number of operands may be two, three, four, or five.

LTLD — This instruction loads a literal from the program segment as indicated by a program segment relative address. The literal address may be either specified by a signed fifteen bit integer operand or, if this is not present, the value of T. The instruction must specify if the literal to be loaded is either a signed seven bit integer value, a signed fifteen bit integer value, a 32 bit floating point value, or a 64 bit quantity. This instruction can be used for the efficient pooling of large literals.

LTLDX — This instruction is similar to LTLD but allows an index to be added to the literal load address, and checks the index value with a specified limit. The T and T2 (or if the program segment literal address is found at T, then T2 and T3 respectively) contain the index limit and the index value respectively. T and T2 are integerized. If T2 is greater than zero and less than T, then T2 is to be considered the index. If T2 is less than or equal to zero, zero is to be considered the index. If T2 is greater than or equal to T, then T is considered the index.

The index is multiplied by the byte width of the literal to be loaded. This value is added to the literal address, and the specified literal is then placed into T.

2.4.3.5 Name Manipulation. The MP instruction architecture has been designed so that single operand elements may be either fetched or addressed by the use of one of the operand meta-operators. This is accomplished by a process of automatic indexing (ADRE and GET). However, it is necessary to be able to manipulate general data structures not envisioned as

multirank arrays of one of the five basic data types, and to be able to partition the arrays of these basic forms. Therefore, several operators have been included so that it is possible to load the next entity of an address chain, index a descriptor, and to change the array limit or offset (Figure 2.4-19).

LOAD — Using the address field of T place that to which it is pointing into T. If this is a Mom descriptor change it into a Copy descriptor.

LDRK — In order to maintain the appropriate number of descriptors for a multiranked entity, this operator will not only load the next piece of information, but if the current rank has been indexed, its offset value will be combined with the offset value of the descriptor of the next rank.

1) If T contains an ADW, fetch that to which it points. If this is an ADW or a Copy descriptor, place it into T. If this is a Mom descriptor place the descriptor into T and make it into a Copy descriptor.

2) If T contains a Copy descriptor which has been indexed, and if that which is pointed to is a descriptor, then use the ARRAY OFFSET field of the indexed descriptor and add to the ARRAY OFFSET field of the new descriptor. Place this new descriptor into T.

3) If T contains a descriptor which has not been indexed, and if that which is pointed to is a descriptor, then place T into T2 and place the new descriptor into T.

INDX — T2 must indicate* a descriptor. Index this descriptor by the integerized value in T, set the index bit (I) of the descriptor, and set the DELTA field and LIMIT field to 0. If the value of T exceeds the LIMIT field this causes an index error. The modified descriptor is placed into T.

LIM — T2 must indicate a descriptor. Change the LIMIT field of the descriptor by the inte-

*"indicate" is used in this context to mean "contain or point to".

Example: M_2 TO 3, 3 TO 5

```
ADR  MRD      Matrix M row descriptor
LDRK          Load first rank information
LTS4 2        Change offset of first rank
AROF
LTS4 2        Set new rank limit
LIM
LDRK          Load next rank information
LTS4 3        Change offset of this rank
AROF
LTS4 3        Set new rank limit
LIM
  .
  .
  .
```

Figure 2.4-19: Manipulating Multiple Ranks

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

gerized value of T. If the value of T exceeds the LIMIT field, this causes an index error. The modified descriptor is placed into T.

AROF — T2 must indicate a descriptor. The integerized value of T must be less than the LIMIT field of T2. If the value field of T exceeds the LIMIT field this causes an index error. The integerized value of T minus 1 multiplied by the DELTA field is added to the ARRAY OFFSET field of T2 and placed back into the ARRAY OFFSET field of the descriptor. The modified descriptor is placed into T.

2.4.3.6 Store. The ability to place information into a variable is accomplished by the store instructions. In order to implement HOLs with multiple receivers two forms of stores are provided: one has the normal stack purge effect with its operands; the other saves the sending value and only purges the receiver address.

Provision has been made for store instructions which use the receiver address as the storage address, ignoring any tag bits located there, overriding the automatic store feature. Similarly they send the entity located in the stack, overriding the automatic fetch mechanism.

STD — Store the value indicated by T into the location indicated by T2.

STN — Store the value indicated by T into the location indicated by T2. Leave T2 value in T. (Do not purge receiver address.)

STDI — Store the 64 bits of T2 into the location indicated by T and override the autostore mechanisms.

STNI — Store the 64 bits of T2 into the location indicated by T and override the autostore mechanism. Place T2 into T.

BSR m,n — Store the m low order bits of the value indicated by T2 into the value indicated by T starting at bit position n.

All of the Store operators may be used for array operations.

2.4.3.7 Arithmetic Manipulation. The MP has a basic set of mathematical operations which correspond to those normally needed. They act upon 63 bit double precision floating point operands taken from the stack.

ADD	T and T2 are added. Result is placed into T.
SUB	T is subtracted from T2. Result is placed into T.
MUL	T and T2 are multiplied. Result is placed into T.
DIV	T2 is divided by T. Result is placed into T.
CHSN	The mantissa sign of T is changed and the new value is placed into T.

The ADD, SUB, MUL and DIV operations may be used with the STT and STT3 array prefix operators.

2.4.3.8 Logical Manipulations. There are three basic classes of logical manipulations: relational operations, logical operations, and bit manipulation

a) Relational Operators

There are usually six basic arithmetic relationals: equal, not equal, greater than, less than or equal (i.e., not greater than), greater than or equal, and less than (i.e., not greater than or equal). In the MP instruction architecture the result of a relational operator is considered to be a boolean which has a truth function value of true or false. This is implemented by placing a 1 or 0 respectively into T. This mechanization then allows for the immediate manipulation of the truth function value in compound relational statements by means of the logical operators. Since code is to be produced by a compiler, it has been felt advantageous to implement only three of the arithmetic relationals, and to implement both a branch on true and on false. The compiler is then able to implement the correct conditional or branch condition while reducing the number of operators needed. (Even in the worst case of a non-judicious choice of compound conditionals, the use of the single byte logical "not" operator is no penalty compared to the savings realized with these operators in the general case.)

Besides the arithmetic relational operators, a logical identity operator has been implemented for bit patterns. Character data will be able to use the three arithmetic operators since all character data must be referenced via a CHAR descriptor and is therefore easily identified. The comparison in this case is of normal dictionary form, where the exact ordering of the alphanumeric and special characters will depend on the character code used.

EQUL — T and T2 are compared. If they have the same algebraic value, a one is placed into T, otherwise a zero is placed into T.

GREQ — T2 is compared to T. If T2 is greater than or equal to T algebraically, then a one is placed into T, otherwise a zero is placed into T.

LSEQ — T2 is compared to T. If T2 is less than or equal to T algebraically, then a one is placed into T, otherwise a zero is placed into T.

SAME — T and T2 are compared. If they have the same logical bit value a one is placed into T, otherwise a zero is placed into T.

b) Logical Operators

These operators are used to perform logic operations upon the two top of stack entries, on a bit-by-bit basis.

LAND — T and T2 are logically "and"-ed together. The result is placed into T.

LOR — T and T2 are logically "or"-ed together. The result is placed into T.

LXOR — T and T2 are "exclusively or"-ed together. The result is placed into T.

LNOT — T is complemented on a bit-by-bit basis and the result is placed into T.

c) Bit Manipulation

There are three basic reasons for bit manipulation within a word. One is to be able to set, reset, or change a particular bit; the second is to be able to move (shift) a field within a word; and the third is to be able to move a bit field from one word, into another bit field in another word.

When the hardware part of the "top of stack" is large enough to contain multiple operands (e.g., an 8 deep hardware stack), then the difference between prefix, postfix and infix operations becomes negligible as far as the execution characteristics of the hardware is concerned. As illustrated in Section 2.3.3 the hardware part of the stack can be considered a "look ahead" mechanism.

BSETL	n	Set bit position n of T to 1. Place result into T.
BRSTL	n	Reset bit position n of T to 0. Place result into T.
BCHGL	n	Complement bit position n of T. Place result into T.
BTSTL	n	Test bit position n of T. Leave value of T. Set condition code in status register.
BSET		Set bit position indicated by T in T2 to 1. Place result into T.
BRST		Reset bit position indicated by T in T2 to 0. Place result into T.
BTRN		Transfer the number of bits indicated by T, starting at bit position indicated by T2 from value indicated by T4 into the value indicated by T5 starting at bit position indicated by T3. Place result into T.
BLD	m,n	Place m bits starting from bit position n of the value indicated by T into T.

BOUT m,n — Place m bits starting from bit position n of the contents of T into the low order bits of T.

BIN m,n — Place the n low order bits of the value indicated by T into the m bit positions starting at position n in T2. Place this new value into T.

2.4.3.9 Flow Control. Flow control within the MP instruction architecture has been designed to satisfy the requirements of modern HOLs: alternate choice, modularization, and interaction control.

a) The primary method of flow control within a computer is the conditional branch. Branches within a given program segment can be accomplished by relative addressing with respect to the current operator location.

JOT — If T2 has bit 1 set to one, add the value of T to the instruction location counter. In either case T and T2 are purged.

JOF — If T2 has bit 1 set to zero, add the value of T to the instruction location counter. In either case T and T2 are purged.

JMP — Add the value of T to the instruction location counter.

JCC m — The low order four bits (bits 3-0) of m are associated with the value of the two-bit condition code. If the bit which corresponds to the condition code value is a one, then add the value of T to the instruction counter.

b) The modularization of programs is accomplished by a standard enter and return mechanism. Since formal parameters are passed as the first entries of the called procedure, there is of necessity a separate instruction used to create the control words (MSW, REW) within the stack, distinct from the actual instruction which causes entry into the called procedure.

PRCS — The process operator is used to create a new stack and therefore a new dynamic environment. T indicates a GEN descriptor

for the new stack. T2 indicates an integer value to be used as the stack number. Any values left in the hardware part of the stack are pushed into the memory portions of the old stack.

MKS — Place a MSW and REW into T2 and T respectively.

ENTR — T should indicate a PEW. Enter the called segment via this PEW. The previous MSW and REW in the stack now have their linkage fields appropriately set.

RTRN — Return via the current REW, reinitializing the stack to the previous procedure level and preserving the contents of T.

EXIT — Return via the current REW, reinitializing the stack to the previous procedure level.

c) The looping control for DO FOR statements has been implemented by the use of a single operator with appropriate operands. The actual implementation of the FOR operator in the laboratory model can be modified depending upon the semantics of the loop control in the implemented HOL. It has been assumed here that the semantics correspond to that of the DO FOR statement of HAL.

FOR — This operator uses either four or five operands from the stack. The number of operands corresponds to the initial executions of loop control. These two types of execution are differentiated by requiring T3 to be a value operand of the initial value for the initial execution phase, while T3 must be the address of the loop variable for the successive execution phases. Figure 2.4-20 and 2.4-21 show the state of the stack upon execution of the FOR operator, a flow chart of its execution and an example of its usage.

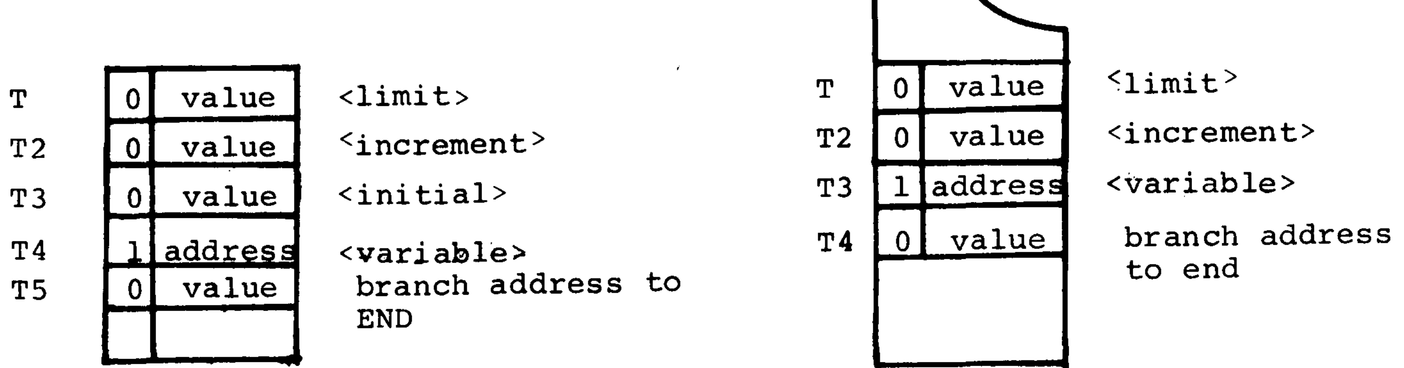
d) The implementation of computed GO TO or DO CASE statements is provided by the use of a combination of branch instructions and the literal load and index

a) Syntax of FOR Statement:

DO FOR <variable> = <initial> BY <increment> TO <limit>;

b) Appearance of Stack:

i) Prior to first iteration   ii) Subsequent iterations



c) Example of Compiled Code:

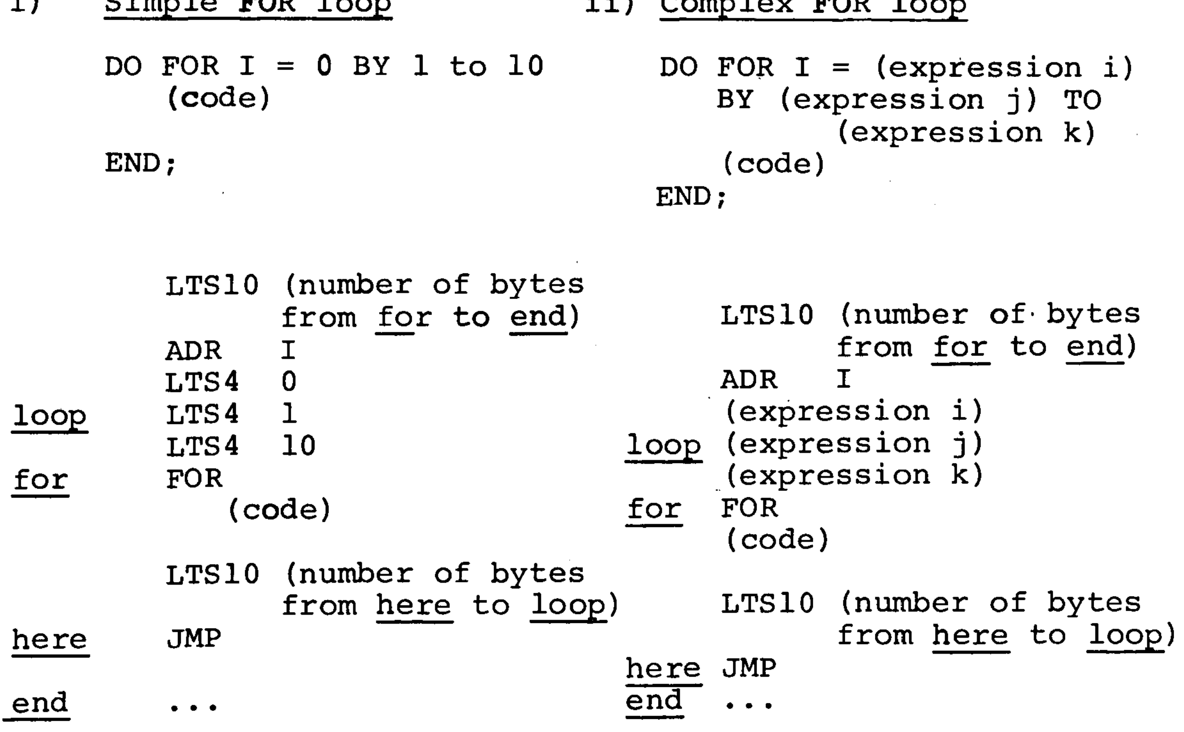


Figure 2.4-20: Example of FOR Instruction Execution

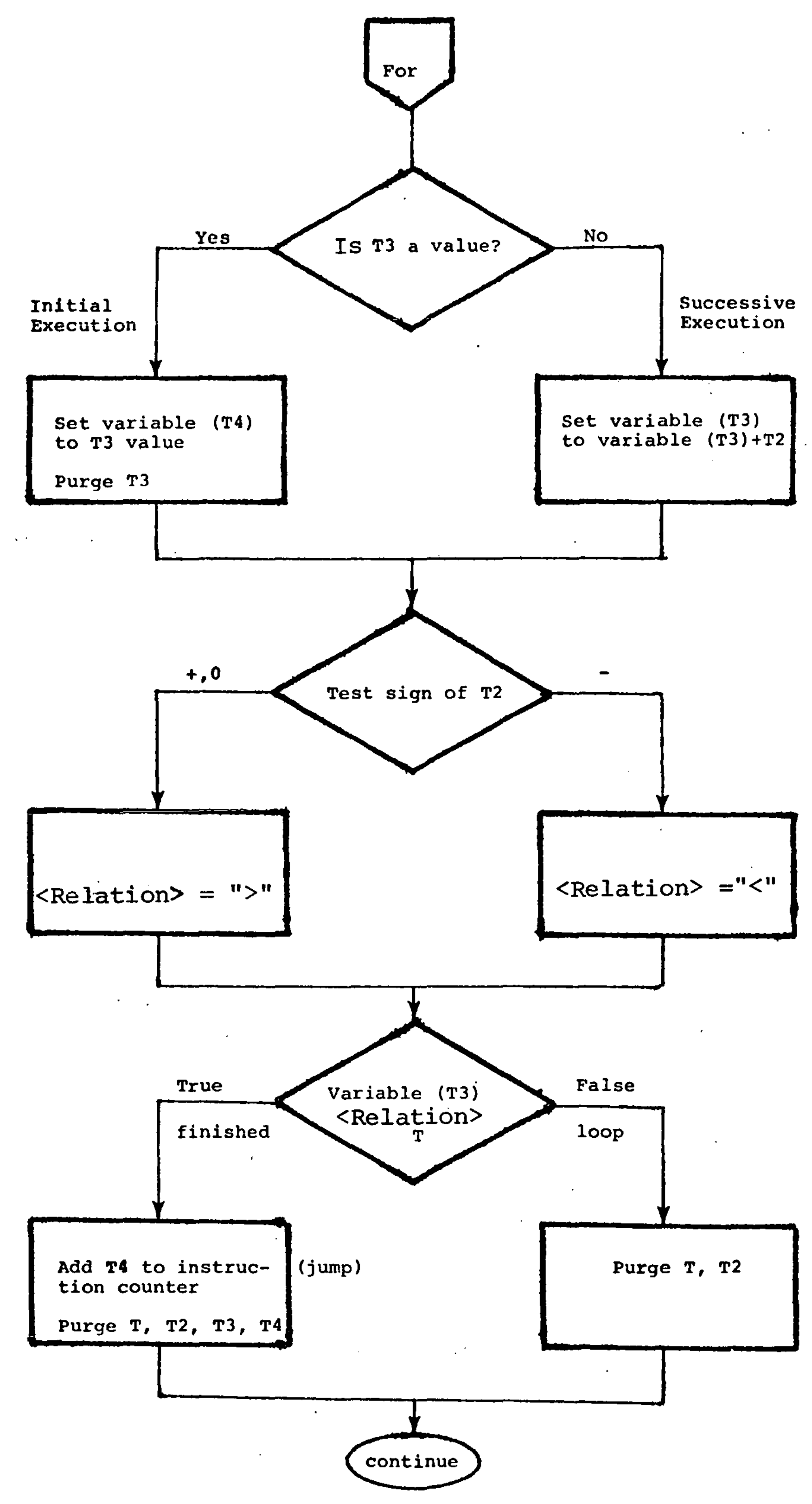


Figure 2.4-21: FOR Flow Chart

(LTLDX) instruction. Figure 2.4-22 shows an example of a DO CASE implementation where the semantics of the DO CASE are such that if the case variable is out of range, the ELSE part of the statement will be executed.

2.4.3.10 Data Field Manipulation. The basic data types in the MP instruction architecture are character, floating point and integer. Integers are an implicit subset of floating point. The conversion between the different precision floating points and integers is implicit within their references through descriptors. There is also the facility for explicit conversion between different data types and for the recovery of data types which are the proper subset of another type.

INTT	T is integerized by truncation. The result is placed into T.
INTR	T is integerized by rounding. The result is placed into T.
FRAC	The integer part of T is purged and the fractional part is placed into T.

Arithmetic-to-character and character-to-arithmetic conversions are standardized within the MP instruction architecture. The exact details of the conversion may be varied with a particular HOL implementation, but the basic instruction structure is identical. The manipulation of character strings within the MP can be effected by sub-string (sub-arraying) and concatenation operations.

CCAT	T and T2 indicate CHAR descriptors. The character string indicated by T is concatenated to that indicated by T2. The result of the concatenation is placed into the process string area and a descriptor of this concatenated string is created and placed into T.

An examination of the definition of CCAT illustrates the general problem of string manipulation: there must be a temporary work area available to the processor for the manipulation of the temporarily created concatenated character strings until such time as they are used and then purged.

a) Example of DO CASE

```
DO CASE X;

     ELSE (case out of range);

     A:  (code for case A);

     B:  (code for case B);

     C:  (code for case C);

END;
```

b) Compiled Code for above DO CASE

1) Instruction

```
        GET    X
        LTS4   (number of entries in lit. table)
        LTLDX  (rel. pos. of lit. table), (byte width of
                    lit. table)
        JMP
   else        (code for case out of range)

        LTS4,10(bytes to end from here)
        JMP

   A           (code for case A)

        LTS4,10 (bytes to end from here)
        JMP

   B           (code for case B)

        LTS4,10(bytes to end from here)
        JMP

   C           (code for case C)

        LTS4,10 (bytes to end from here)
        JMP
         .
         .
         .
   end         . . .
```

2) Table of Integers

```
   lit. table     0
                  (bytes from A to else)
                  (bytes from B to else)
                  (bytes from C to else)
                  0
```

Figure 2.4-22: Example of DO CASE Execution

There are two basic choices available for this temporary manipulation; either the compiler can generate temporaries statically, and hence simply provide an extra operand for those character (or array) operators that need temporaries, or else the allocation of the temporaries is done dynamically by the processor. It is this latter method that the MP instruction architecture proposes.

Figure 2.4-23 shows the initial usage of the character string temporary work area. The current stack has within it, at a system-wide defined displacement, a descriptor of the character string work area. In the figure the offset relative to the stack number is indicated as Δ (delta). Besides this Mom CHAR descriptor, a Copy descriptor is located at Δ + 1 for use of the operators which need the temporary work area. The character string work area is assigned as needed, alternating from the bottom and the top of the array. Within the Copy descriptor "a" indicates the offset at the bottom of the work area from which the next temporary area may be assigned. Similarly, "b" represents the top location currently in use; consequently the area assigned from the top end cannot exceed this offset value. It will be noted that the first entry of the temporary stack is an initial Copy descriptor.

In Figure 2.4-24 the manipulation of the work area needed in the calling of procedures is shown. When a procedure specifying a character temporary area is entered, the Copy descriptor is stored into the temporary area. The ARRAY OFFSET field of this stored descriptor is set to the ARRAY OFFSET field of the Mom descriptor. The stored descriptor is thereby effectively linked back to the previous stored descriptor. The Mom descriptor then has its ARRAY OFFSET field set to the Copy descriptor's ARRAY OFFSET field and hence is an effective pointer to the just stored descriptor. The Copy descriptor then has its ARRAY OFFSET field incremented to point to the first free area after the stored descriptor.

When a monadic character operator is executed and the operand itself is not a temporary, then either end of the temporary array area may be used for the needed temporary storage. If the operand is a temporary, then the other end of the temporary area provides for temporary storage. Similarly, with dyadic operators, the only problem that could arise is when both operands are temporaries. In this case, since the only character dyadic operator is concatenate, the first operand will remain with the second operand concatenated to it if the first operand is at the bottom.

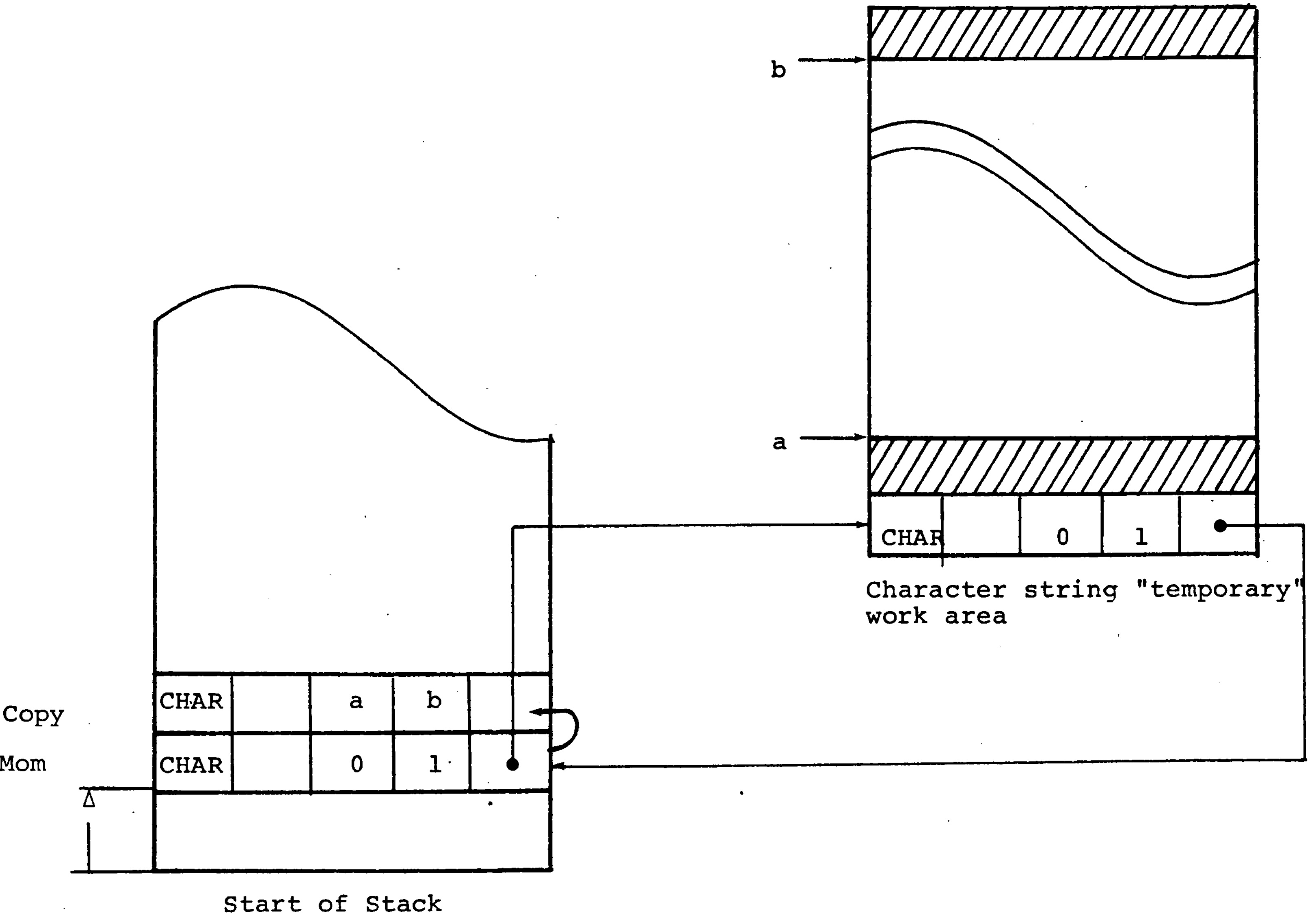


Figure 2.4-23

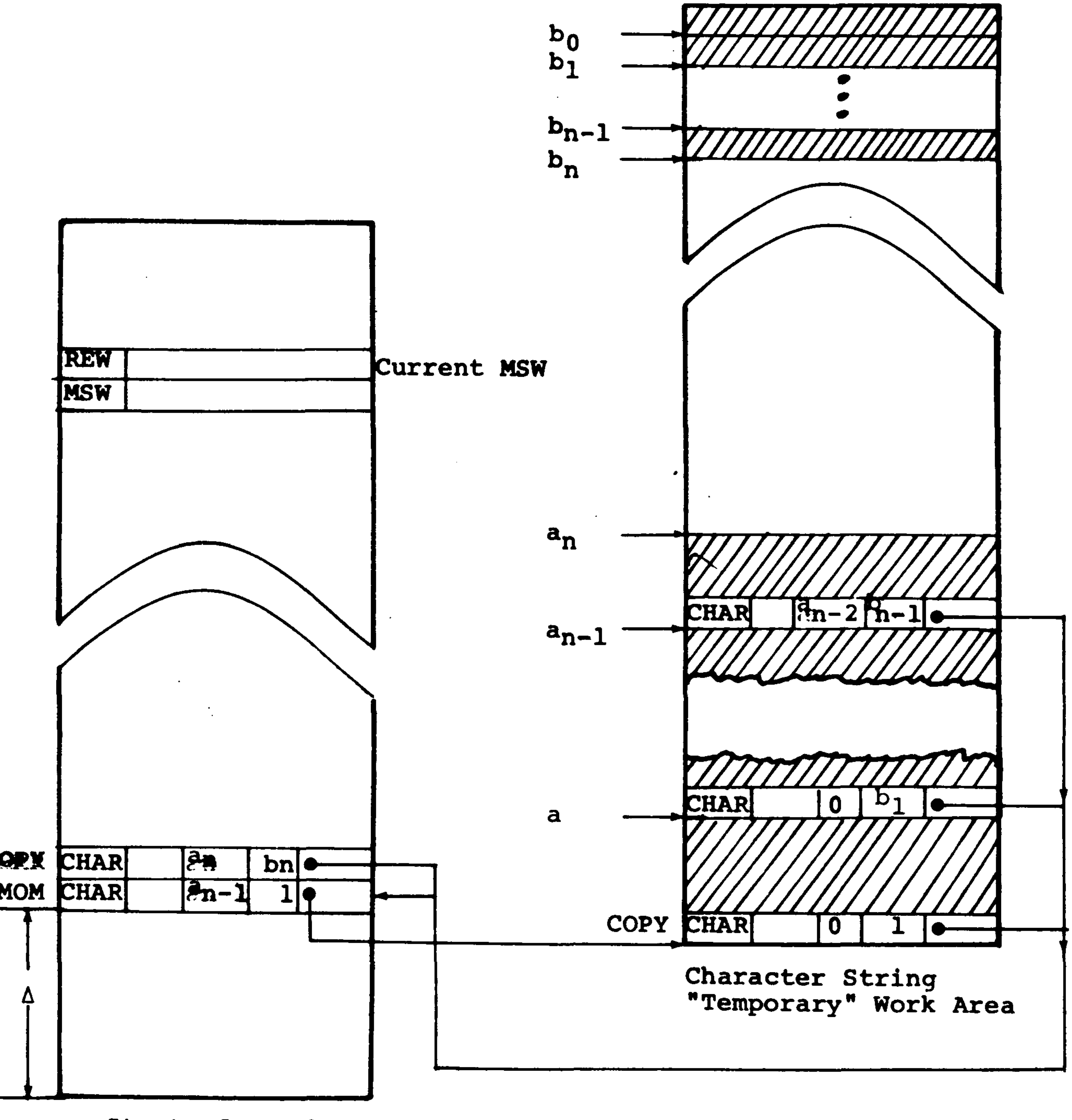


Figure 2.4-24

Otherwise, the final operand will be moved to its new location at the top, and the second operand becomes concatenated.

ITOC	T is integerized and then converted to the normal character string equivalence. It is stored in the character string temporary area, the descriptor pointer to this character string is created and placed into T.
STOC	T is converted to the normal character string equivalence and stored in the character string temporary area. A descriptor pointing to this character string is created and placed into T.
CTOI	T indicates a CHAR descriptor. The indicated character string is converted to an integer and is placed into T.
CTOS	T indicates a CHAR descriptor. The indicated character string is converted to a scalar and is placed into T.

The standardized form of character scalar or integer input to the character conversion instructions is as follows.

STANDARD ARITHMETIC INPUT FORM :=

b̸ {[+]| -}b̸ scalar b̸ [E b̸ [+]| -} b̸ integer b̸]

scalar: ={integer.[integer]|[integer].integer}

integer: ={digit}[digit]...

digit: = 1|2|3|4|5|6|7|8|9|0

In the above notation, the braces { } indicate that which is contained must be present. The brackets [ ] indicate that which is contained is optional. The line | indicates an option "or". The dots ... indicate a repeat of the proceeding item of any length. b̸ denotes zero or more spares or blanks.

This syntax then defines the form of arithmetic character string which will be converted into the internal arithmetic form by the character conversion operators. Figure 2.4-25 gives several examples of the allowable forms.

STANDARD ARITHMETIC INPUT FORM: =

```
ƀ {[+]|-} ƀ scalar ƀ [E ƀ {[+]|-} ƀ integer ƀ]

scalar: =   {integer[·integer]|[integer]·integer}

integer:=   {digit}[digit]...

digit:  =    {1|2|3|4|5|6|7|8|9|0}
```

```
 - 2.3      E  10
   2
 +  .01     E - 8
 +100.3
  12E5
```

Figure 2.4-25: Examples of Standard Arithmetic Input Form

The standard arithmetic output form is of two kinds depending on whether the conversion is from an integer or scalar.

STANDARD INTEGER OUTPUT FORM :=

{+|-} <u>integer</u>

The length of the character string output is that of the number of digits needed to represent the integer plus the sign character. If the integer has value of zero the form is simply +0 or -0 depending upon sign.

STANDARD FLOATING POINT OUTPUT FORM :=

{+|-}. <u>integer</u> E{+|-} <u>integer</u>

The exponent <u>integer</u> always contains 3 digits with leading zeros as necessary. The mantissa <u>integer</u> contains the number of digits needed for representation, but in no case exceeds 16 digits. Figure 2.4-26 shows an example of these forms.

2.4.3.11 <u>System Considerations</u>. This group of instructions meets several requirements associated with MP instruction architecture, MP hardware implementation, MP module interconnections, and MP I/O communications.

a) Instruction Architecture Requirements

The manipulation of the stack is required to reverse the wrong order of entries within the stack, or to override the normal self purging feature of the stack upon usage.

XCH	T and T2 exchange places.
DLET	T is purged.
DUPL	T is placed into both T and T2.
NOP	No operation. This is sometimes needed as a filler for compiler convenience.

STANDARD INTEGER OUTPUT FORM: = {+|-}<u>integer</u>

STANDARD FLOATING POINT OUTPUT FORM: = {+|-}·<u>integer</u> E{+|-}<u>integer</u>

VALUE	INTEGER OUTPUT FORM CHARACTER	INTEGER OUTPUT STRING LENGTH	FLOATING POINT OUTPUT FORM CHARACTER	FLOATING POINT OUTPUT STRING LENGTH
10	+10	3	+.1E+002	8
1	+ 1	2	+.1E+001	8
0	+ 0	2	+.0E+000	8
.023	——	——	+.23E-001	9
-15,983	-15983	6	-.15983E+005	12
-1/3	——	——	-.3333333333333333E+000	23

Figure 2.4-26

b) Hardware Implementation

These instructions control the actual running of the MP system, and allow hardware functions at a lower level than the MP instruction architecture to be handled.

IDLE		A processor idles until such a time as it is assigned a process, either directly by the operating system, or because of an interrupt.
HALT		A processor is brought to a non-execution phase until it is explicitly restarted, either by the operating system or by operator intervention.
LDPR		T indicates one of the processor hardware register. Place the 64 bit value of the hardware register into T.
STPR		T indicates one of the processor hardware registers. T2 is a 64 bit quantity. Store the 64 bit quantity in T2 into the processor hardware register indicated by T.

c) Network Interconnections

These instructions allow communication and interactions among the different active modules of the MP system. Several of these instructions are needed to handle the concurrent execution of two or more processes in the MP system.

STLD		Store the value indicated by T2 into the location indicated by T. During the same memory cycle the contents of the memory location indicated by T are read and placed into T.
LDID		The unique identification of the processor is placed into T.
IPC	n	Transmit over the IPCB the first n bytes contained in the stack. This is used to enable interprocessor communication. This instruction can be used to communicate all the commands necessary to the I/O and other processors.

OPERAND META-OPERATORS

COPY
GET
ADR
ADRE

LITERALS

LTS4
LTS10
LTS15
LT32
LT32F
LT64
LTS7M
LTLD
LTLDX

NAME MANIPULATORS

LOAD
LDRK
INDX
LIM
AROF

STORE

STD
STN
STDI
STNI
BST m,n

ARITHMETIC MANIPULATORS

ADD
SUB
MUL
DIV
CHSN

LOGICAL MANIPULATIONS

EQUL
GREQ
LSEQ
SAME
LAND
LOR
LNOT
BSETL n
BRSTL n
BCHGL n
BTSTL n
BSET
BRST
BTRN
BLD m,n
BOUT m,n
BIN m,n

ARRAY AND MEMORY

STT
STT3

FLOW CONTROL

JOT
JOF
JMP
JCC m
PRCS
MKS
ENTR
RTRN
EXIT
FOR

DATA FIELD MANIPULATION

INTT
INTR
FRAC
CCAT
ITOC
STOC
CTOI
CTOS

SYSTEMS CONSIDERATION

XCH
DLET
DUPL
NOP
IDLE
HALT
LDPR
STPR
STLD
LDID
IPC n

Figure 2.4-27: Instruction Repertoire

OPERATOR	BYTE LENGTH	FORM	
a) All Instructions (except those below)	1	0 0 v v v v v v	v..v provides codes
b) Exceptional Instructions			
BST m,n	3	0 0 p p p p p p | 0 0 m m m m m m | 0 0 n n n n n n	
BLD m,n	3	0 0 p p p p p p | 1 1 m m m m m m | 0 0 n n n n n n	
BOUT m,n	3	0 0 p p p p p p | 1 0 m m m m m m | 0 0 n n n n n n	
BIN m,n	3	0 0 p p p p p p | 0 1 m m m m m m | 0 0 n n n n n n	
		m ... m bit field length	
		n ... n starting bit position	
BSETL n	2	0 0 q q q q q q | 0 0 n n n n n n	
BRSTL n	2	0 0 q q q q q q | 0 1 n n n n n n	
BCHGL n	2	0 0 q q q q q q | 1 0 n n n n n n	
BTSTL n	2	0 0 q q q q q q | 1 1 n n n n n n	
		n ... n bit position	
JCC m	2	0 0 r r r r r r | 0 0 0 0 m m m m	
		m ... m mask bits	

Figure 2.4-28: Standard Operators

Figure 2.4-28 (continued)

OPERATOR	BYTE LENGTH	FORM
c) Operand Meta-Operators		
COPY	2	1 0 0 a a a a a | a a a a a a a a
GET	2	1 0 1 a a a a a | a a a a a a a a
ADR	2	1 1 0 a a a a a | a a a a a a a a
ADRE	2	1 1 1 a a a a a | a a a a a a a a
		a...a lexical level, displacement
d) Literals		
LTS4	1	0 1 0 S x x x x
LTS10	2	0 1 1 0 0 S x x | x x x x x x x x
LTS15	3	0 1 1 0 1 0 0 0 ... x x x x x x x x (2 bytes)
LT32	5	0 1 1 0 1 0 0 1 ... x x x x x x x x (4 bytes)
LT32F	5	0 1 1 0 1 0 1 0 ... x x x x x x x x (4 bytes)
LT64	9	0 1 1 0 1 0 1 1 ... x x x x x x x x (8 bytes)
LTS7M	3	0 1 1 0 1 1 0 0 ... s x x x x x x x (2 bytes)
	4	0 1 1 0 1 1 0 1 ... s x x x x x x x (3 bytes)
	5	0 1 1 0 1 1 1 0 ... s x x x x x x x (4 bytes)

Legend: s - sign bit
x...x - numerical value

Figure 2.4-28 (continued)

OPERATOR	BYTE LENGTH	FORM
LTLD	1	0 1 1 1 0 0 N N
	or 3	0 1 1 1 0 1 N N ... i i i i i i i i (2 bytes)
LTLDX	1	0 1 1 1 1 0 N N
	or 3	0 1 1 1 1 1 N N ... i i i i i i i i (2 bytes)

NN: literal to be loaded

- 00 signed 7 bit
- 01 signed 15 bit
- 10 32 bit flt. pt.
- 11 64 bit value

i...i: literal table address

Operator	Code Value 8	4	2	1
a) Arithmetic Manipulators				
ADD	overflow	<zero	>zero	zero
SUB	overflow	<zero	>zero	zero
MUL	overflow	<zero	>zero	zero
DIV	overflow	<zero	>zero	zero
CHSN	---	<zero	>zero	zero
b) Logical Manipulators				
LAND	---	all ones	mixed	all zeros
LOR	---	all ones	mixed	all zeros
LNOT	---	all ones	mixed	all zeros
BSETL n	old one	old zero	new one	new zero
BRSTL n	old one	old zero	new one	new zero
BCHGL n	old one	old zero	---	---
BTSTL n	old one	old zero	---	---
BSET	old one	old zero	new one	new zero
BRST	old one	old zero	new one	new zero

Figure 2.4-29: Condition Code Setting

a) Array Operations

Arithmetic	Logical
ADD	LAND
SUB	LOR
MUL	LXOR
DIV	

b) Example of Usage

A = B + C;

i) Normal Usage	ii) STT Usage	iii) STT3 Usage
ADRE A	ADR B	ADR A
GET B	ADR C	ADR B
GET C	ADR A	ADR C
ADD	STT	STT3
STD	ADD	ADD

Figure 2.4-30: Allowable Array Operations and Memory Operations

References for Chapter 2

1. Graham, R.M., "Use of Higher Level Languages for Systems Programming", Technical Memorandum 13, Project MAC, September 1970, AD 711 965

2. Corbato, F.J., "Sensitive Issues in the Design of Multi-Use Systems", MAC-M-383, Project MAC, December 12, 1968.

3. Corbato, F.J., "PL/I as a Tool for System Programming", Datamation, May 1969.

4. McFarland, C., "A Language-Oriented Computer Design", FJCC 1970, pp. 629-640.

5. Flynn, M.J., "Very High-Speed Computing Systems", Proceedings of the IEEE, Vol. 59, No. 12, December 1966, pp. 1901-1909.

6. Fulmer, L.C., and Meilander, W.C., "A Modular Plated-Wire Associative Processor", GER-14727, Goodyear Aerospace Corporation, Akron, Ohio, March 11, 1970.

7. Githens, J.A., "A Fully Parallel Computer for Radar Data Processing", NAECON '70 Record, pp. 290-297.

8. McIntyre, D.C., "An Introduction to the Illiac IV Computer", Datamation, April 1970.

9. Chen, T.C., "The Overlap Design of the IBM System/360 Model 92 Central Processing Unit", FJCC, Part 2, 1964.

10. Patzer, W.M. and Vandling, G.C., "Aerospace Systems Implications of Microprogramming", in Air and Spaceborne Computers edited by E. Keonjians, Technivision Services, Slough, England, April 1970.

11. Kerner, H., and Gellman, L., "Memory Reduction Through Higher Level Language Hardware", AIAA Journal, Vol. 8, No. 12, December 1970, pp. 2258-2264.

12. Knuth, D.E., "An Empirical Study of Fortran Programs", Stanford University, Computer Science Department, Report No. CS-186, 1970, AD 715 513.

13. Thurber, K.M., and Myrna, J.W., "System Design of a Cellular APL Computer", IEEE Transactions on Computers, Vol. C-19, No. 1, April 1970.

14. Stone, H.S., "A Logic-In-Memory Computer", IEEE Transactions on Computers, January 1970, pp. 73-78.

15. Hauck, E.A. and Dent, B.A., "Burroughs' B6500/B7500 Stack Mechanism", SJCC, 1968.

16. Abrams, P.S., "An APL Machine", Doctoral Dissertation, Stanford University, February 1970, AD 706 741.

17. Keeler, F.S., et al, "Computer Architecture Study", SAMSO-TR-420, October 1970, AD 720 798. (This is the SPL machine.)

18. Sugimoto, M., "PL/I Reducer and Direct Processor", Proceedings of 24th National Conference ACM, 1969.

19. Chesley, G.D., and Smith, W.R., "The Hardware-Implemented High-Level Machine Language for SYMBOL", SJCC 1971.

20. Rice, R. and Smith, W.R., "SYMBOL - A Major Departure from Classic Software Dominated von Neumann Computing Systems", SJCC 1971.

21. Lawson, H.W., Jr., "Programming-Language-Oriented Instruction Streams", IEEE Transactions in Computers, Vol. C-17, No. 5, May 1968.

22. Elson, M. and Rake, S.T., "Code-Generation Techniques for Large-Language Compilers", IBM System Journal, Vol. 9, No. 3, 1970.

23. HAL PASS 1, SOURCE. 77, Intermetrics, April 28, 1971.

24. Church, C.C., "Computer Instruction Repertoire - Time for a Change", SJCC 1970.

25. McKeeman, W.M., "Language Directed Computer Design", FJCC 1967.

26. Lawton, T.J., "AGC Programming Comparison", MIT/IL, Space Guidance Memo #8, Apollo Distribution, July 5, 1962.

27. "Control Data 6600 Computer Systems Reference Manual", Control Data Corporation, August 1963.

28. Chen, T.C., "Parallelism, Pipelining and Computer Efficiency", Computer Design, January 1971.

29. Chen, T.C., "Unconventional Superspeed Computer Systems", SJCC 1971.

30. Hamblin, C.L., "Translation to and from Polish Notation", The Computer Journal, October, 1962.

31. Myamlin, A.N., and Smirnov, V.K., "Computer with Stack Memory", in Information Processing 68, North-Holland Publishing Company, Amsterdam, 1969.

32. "STS Data Management System Design, Task 2", MIT Draper Laboratory, June 1970.

33. "STS Data Management System Design, Task 5", MIT Draper Laboratory, July 1970.

34. Burnett, G.J. and Coffman, E.G., Jr., "A Study of Interleaved Memory Systems", SJCC 1970.

35. Tucker, A.B. and Flynn, M.J., "Dynamic Microprogramming Processor Organization and Programming", CACM, Vol. 14, No. 4, April 1971.

36. Bryan, G.E., "JOSS: 20,000 Hours at a Console - A Statistical Summary", FJCC, 1967.

37. Freibergs, I.F., "The Dynamic Behavior of Programs", FJCC, 1968.

38. Ashley, D.W., "A Methodology for Large Systems Performance Prediction", Technical Report: TR 00.1773, IBM System Development Division, Poughkeepsie Laboratory, September 10, 1968.

39. "Burroughs B6500 Information Processing Systems Reference Manual", Burroughs Corporation, 1969.

40. Wersan S.J., et al, "Architectural Study for Advanced Guidance Computers, Part 2", February 5, 1971, AD 723 669.

41. Stone, H.S., "A Pipeline Pushdown Stack Computer", Parallel Processor Systems, Technologies and Applications, edited by L.C. Hobbs, et al, Spartan Books, 1970.

42. "Burroughs B6700 Handbook Preplanning Edition", Burroughs Corporation, 1971.

43. Vandever, W.H., "Designing a Higher Order Language Machine", Multiprocessor Memo #05-71, Intermetrics, 19 July 1971.

44. "Univac Data Processing Division 1108 Multiprocessor Systems, System Description", Univac, No date.

45. "IBM System/360 Principles of Operation", IBM Form A22-6821-5.

46. "Introducing the RCA 215 Military Computer", RCA, No date.

47. "HALMAT, An Intermediate Language of the first HAL Compiler", Intermetrics, Inc., Cambridge, Mass., 22 October 1971, Revised.

48. Hassitt, A., et al, "Implementation of a High Level Language Machine", 4th Annual Workshop on Microprogramming, University of California, Santa Cruz, California, 13-14 September 1971.

49. "Functional Design of a Multiprocessor", Monthly Progress Report, October 1971, Contract NAS 9-11745, Intermetrics, Inc., Cambridge, Mass., 15 November 1971.

50. Hansen, P. Brinch, "RC 4000 Computer Reference Manual", 2nd Edition, A/S Regnecentralen, Copenhagen, June 1969.

51. Davis, R.L., and Zucker, S., "Structure of a Multiprocessor Using Microprogrammable Building Blocks", Proceedings NAECON, 1971.

52. Iliffe, J.K., "Basic Machine Principles", American Elsevier Publishing Co., New York, N.Y., 1968.

53. Iliffe, J.K., "Elements of BLM", Rice University Computer Project, 7 November 1968.

3.0

MULTIPROCESSOR OPERATING SYSTEM DESIGN

3.1 Introduction

This section presents the functional design of a basic operating system for the multiprocessor which is the subject of this study. The design has been directed toward the computer hardware configuration described in Chapter 1, and makes much use of the instruction architecture and data structures that were proposed in Chapter 2. The operating system design presented covers the broad spectrum of functions necessary to provide adequate and manageable throughput for the proposed computer configuration. It does not just address those problems of management unique to multi- (as differentiated from single) processors. The attitude taken throughout the design of the operating system has been to make no assumption of single processing, or serial processing in order to implement an operating system feature. At all times during the evolution of the design the presence of many, concurrent activities in the system was an assumed likelihood, so that at no stage was a solution to the single processor problem arrived at first and then modified to the more general case of several processors.

3.1.1 Design Philosophy

The operating system design philosophy reflects the emphasis on the achievement of reliable operation of both hardware and software that has already been stated in the Introduction to this report. Some of the specific assumptions and decisions that contribute to the overall operating system design philosophy illustrate this emphasis.

It was assumed that only higher order language(s) would be used in the programming of application software. In addition to providing the well established advantages for the production of the software itself, the exclusive use of HOLs allows secure system operation to be realized without exhaustive runtime verification of each request for OS functions. An intimate and well-defined interface beween OS and the compiler(s) was assumed achievable, so that an optimal division between static (pre-run) and dynamic (runtime) diagnosis could be made. The

impact of certain OS decisions on the design of a given compiler was not, however, a subject of this study although due consideration for reasonableness and implementability of any assumed interface was given. Another outcome of HOL assumption was the elimination of the need for special privileged (or supervisor) instructions and modes of operation, since any attempted variations of OS groundrules by "non-privileged" users become automatically screened out during the compilation phase.

An additional assumption was that the programming language should possess multi-tasking capabilities (like PL/I, or HAL). The problem of maximizing the theoretical throughput of several processors requires user assistance to identify task or process parallelism; the exploitation of implicit parallelism at the algorithmic or instruction level has proved difficult, expensive, and of limited effectiveness (especially in the type of general purpose application expected in the Space Station).

Finally, it was assumed that the language/compiler to be used in programming the Space Station application software would possess the facility of handling common data pools (Compools). The MP design provides a Compool implementation.

The traditional problems of multiprocessing are associated with the conflict between several simultaneous processes over access to certain common resources, including processors, memory, shared data and procedures. Although most of these problems exist in some form in a multiprogrammed, single-processor system, the additional degree of difficulty in a multiprocessor environment is contributed by the actual time concurrence of the several processes. This fact eliminates the effectiveness of the usual device of serializing the execution of the several processes when a common response or critical activity is involved, because if any or all processors in an n-processor system are made to wait, there is an immediate and drastic reduction in the potential throughput. The resolution of this conflict has been acknowledged throughout the OS design effort, and has even received special hardware attention in the form of the proposed associative memory incorporated as part of each processor, to assist in the addressing of shared data segments (see Chapter 5).

The major task that the proposed OS design tackles is the management of operating memory (M2) multiplexing. The decision to implement a virtual memory system was influenced by factors of economy, ease of application programming, and fault-tolerance. The choice of variable rather than fixed

size memory blocks as the quanta to be multiplexed aggravates the task of making virtual memory work efficiently. With limited M2 space the preservation of the "working set" of the current process and, of even more importance, that of other processes, becomes the major concern for OS. All this must be accomplished without excessive consumption of overhead.

3.1.2 Definitions

A word on the meaning of the terms used throughout this (and other) chapters is in order. The key terms and their assumed definition are as follows:

a) Program: This is an independently compilable section of code containing pure procedures and/or data.

b) Procedure: A section of code to which execution control can be passed, with or without the passage of parameters. There are two kinds of parameters:

   1) Internal, not known outside of process (see below)

   2) External, known to name management and declared in the Process Information Area (see below)

c) Segment: A contiguous block of words defined by a descriptor, which is the unit of memory management.

d) Process: The unit of work as recognized by the operating system. A process is represented by a stack.

e) Stack: Although strictly a LIFO list, the definition of a stack is less rigorous when used to represent a process.

f) Level: A demarcation in the addressing hierarchy. Derived from the concept of lexicographical level in block structured language (such as ALGOL or HAL), but extended to provide convenient addressing by the operating system.

Figure 3.1-1 illustrates the relationship and use of some of these terms, and will be used as a basis for the material of the succeeding sections of this chapter.

Each process is represented by an execution stack. The initial hierarchical level for process execution, and therefore the lowest numerical level for any process stack, is level 2. Subsequent procedure nesting varies the lexical level of each

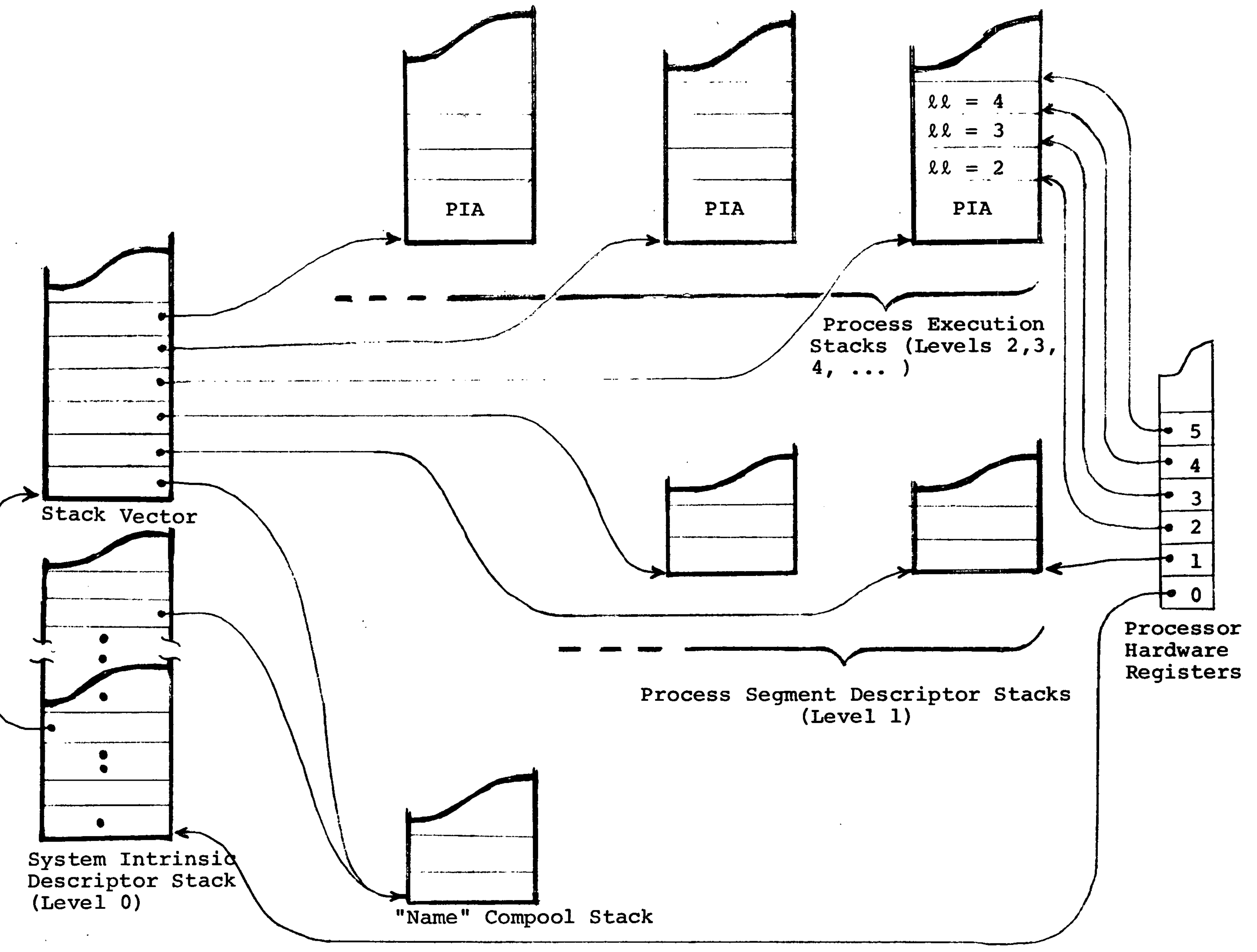


Figure 3.1-1: Operating System Data Structures

process stack to 3, 4, 5, etc. The portion of a process stack that is below level 2 contains a collection of data termed the Process Information Area (PIA) containing names, priorities, counters, for bookeeping, etc., specific to each process (see section 3.2). Above the PIA the stack behaves more strictly as a LIFO list, in accordance with the conventions of the HOL.

Each process has associated with it a vector of descriptors defining the segments containing the procedures to be executed by the process. These descriptors are addressed as if the vector were a stack: by stack number and offset from the base of the stack. For convenience, this collection of segment descriptors is termed Level 1, since it exists at a more global level than the individual processes, and each such vector will be referred to as a stack (even though, strictly, it is not).

At the most fundamental level there is a single collection of basic system descriptors, variables, etc., which is termed the Level 0 stack, again for convenience of addressing. One descriptor at level 0 points to the stack vector, which contains descriptors of all the stacks in the system including the "pseudo-stacks" of levels 1 and 0.

Each processor contains a set of hardware registers which indicates the actual M2 addresses of the start of each of the system levels, i.e., the base address of the corresponding stack. Figure 3.1-1 also shows the linkages that tie the Compool mechanism into the system. These are discussed in greater detail in section 3.4.9.

3.1.3 Synopsis

The remainder of this chapter is devoted to a detailed description of the major functions of the operating system:

Section 3.2 Process State Controller. This section defines the states and state transitions that a process can experience in the MP system, and defines the rules that govern the allowable paths that a process may take. An indication of the system initialization requirements is also given, although these are not developed in such detail, being somewhat implementation dependent.

Section 3.3 Interrupt Handling. This section describes the techniques chosen to handle the effect of unexpected events within a processor (such as traps, failures, etc.), within the system (segment faults,

timer interrupt, etc.), and from outside the system (I/O complete, operator console input, etc.). A very flexible mechanism for defining the desired response to any of these interrupts is described, making use of the pushdown properties of stacks.

Section 3.4 Memory Management. This presents in great detail the implementation, by the operating system, of the segment fault handling, space allocation, and segment replacement functions associated with multiplexing the use of M2.

Section 3.5 I/O Management. This section presents a summary of the operating system I/O data structures and procedures, with emphasis on the interface with Memory Management.

Section 3.6 Timing and Synchronization. The problem of interprocessor communication and synchronization and real time operation are addressed in this section.

Section 3.7 Fault Recovery. Finally, the role of the operating system in assisting the system to resume an operational state, after the detection (by the hardware) of a failure, is described.

3.2 The Process State Controller

3.2.1 Introduction

The process is the unit of work in the multiprocessor system. Each process is a sequence of actions, directed by the instructions in programs. Each process at any moment in time has a single locus of control, i.e., the current point in its sequence. For a process which is running on a processor, this locus of control advances as the processor executes operators. For a process which is not running, the locus of control is temporarily halted at some point. This point will always be a location in the process state controller procedure, since as will be seen, a process always enters this procedure when it is about to give up its processor.

What precisely is meant by a process running on a processor, besides that its locus of control is advancing? The locus of control is really just the program counter, which indicates what operator is being executed by the processor. But the status of the program counter is insufficient to define the locus, since the hardware reacts to an interrupt by simulating a procedure call, thus unexpectedly altering the program counter. The unexpected interrupt procedure is not in itself a new process, even though it may decide to cause a change to a new process. More fundamental to the definition of a process is the stack. The stack has been described in greater detail in Chapter 2; it is sufficient to say here that it contains all the changing computational and control information used by a processor executing instructions. The stack is the impure data half of the "pure procedure/impure data" and is necessary for recursive, reentrant, and clean programming. It is the stack which describes the process. A running process is one whose stack is active, i.e., being used by a processor as its working stack. With this definition, the processor may be executing an interrupt routine which has nothing to do with the useful work of a process, yet that process is still a running process. Interrupt routines run parasitically on the stack of the process running on the interrupted processor, because the processor must always have a stack.

The Process State Controller is a collection of operating system procedures which alter the state of processes and cause processors to be assigned to them. Since there are more processes than processors, and since only one processor can be assigned to a process at any moment, each processor is multi-programmed, i.e., switched from process to process as processes change state. A process that is assigned to a

processor is called a "running" process. One that is capable of being assigned to a processor, but is not because all processors are already assigned, is called "ready". A process which has reached a point where it temporarily cannot use a processor is called "waiting". A running, ready, or waiting process can be stopped by another running process. The stopped state exists independently of the ready or waiting states; a process can be stopped-ready or stopped-waiting. The state transition diagram illustrated in Figure 3.2-1 should make this clear.

Only a running process can get to the waiting state, i.e., a process becomes waiting only by its own action. All transitions except 2 and 3 (refer to Figure 3.2-1 for identification) can be caused only by another process. Assuming every processor is executing a process, process state transition 2 or 3 is simultaneous with another process undergoing state transition 1. A process in the stopped and waiting state must be both readied and resumed to become eligible for running. The creation and termination of a process is not shown on this diagram because the topic is dealt with elsewhere.

3.2.2 Process State Controller Interfaces

A process may call on the services of the Process State Controller through a number of entry points, depending on the specific service. The main services are:

WAIT

READY(P)

STOP(P)

RESUME(P)

EXCHANGE

The (P) is a reference to a process, composed of two items:

a) stack number

b) process identification (processid)

The stack number immediately identifies the process stack, and the processid guarantees that the correct process is referenced. When a process is created, it is assigned a stack number. When it terminates, the stack number is free

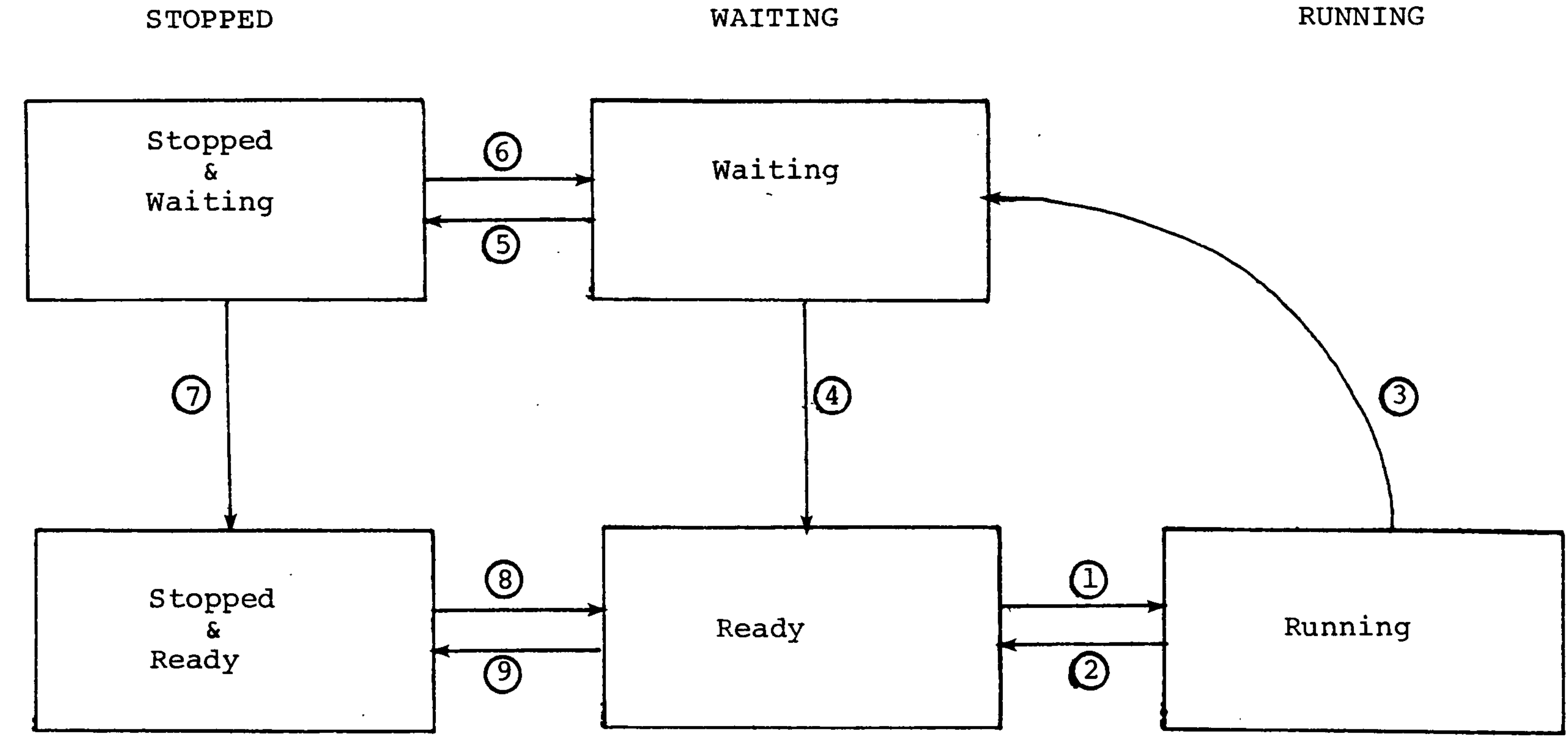


Figure 3 .2-1 Process State Transitions

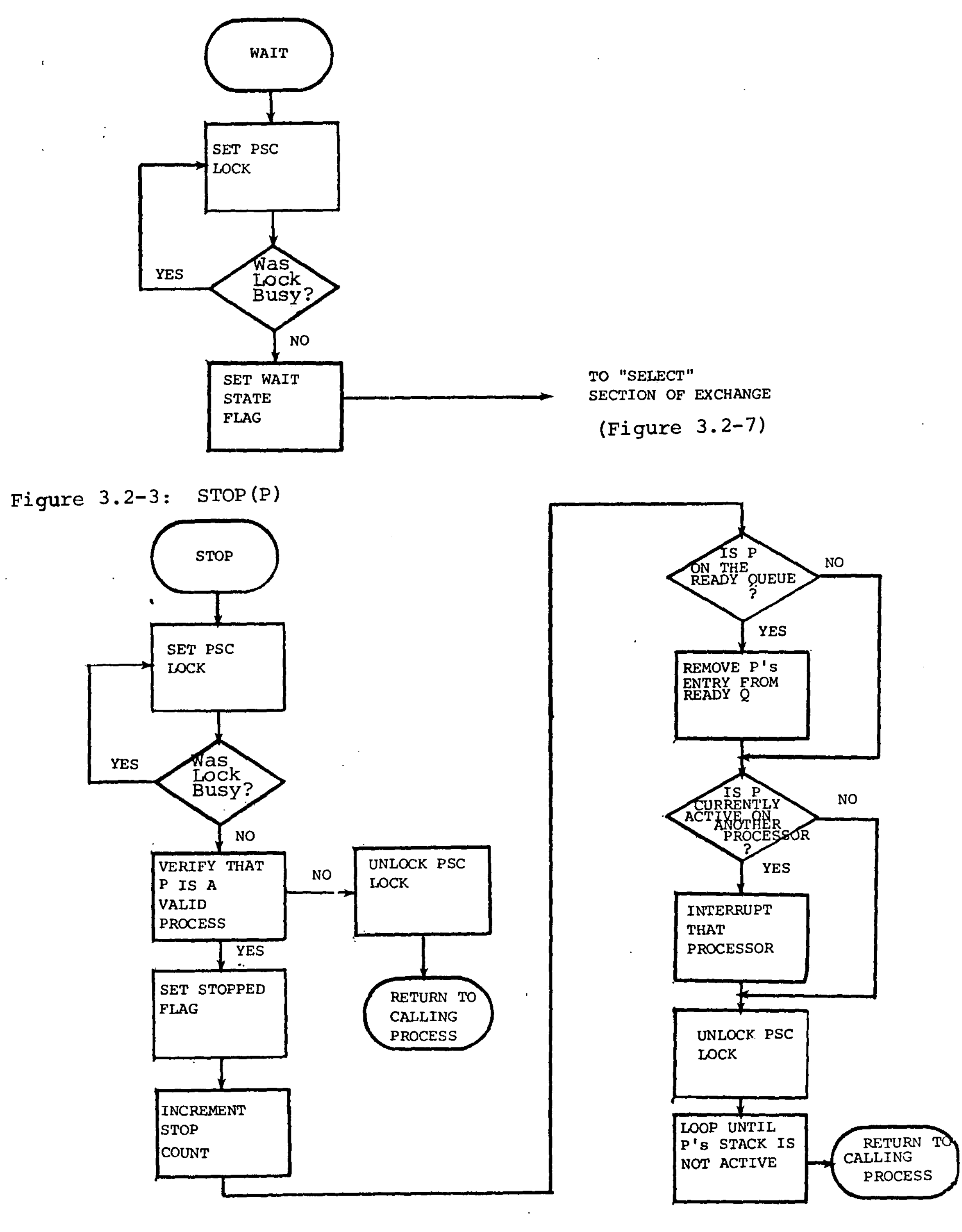
to be re-assigned to a new process. The processid, however, is unique to each process. Each interface which is passed P as a parameter verifies that it is a valid process reference. If it is not, the interface takes no action.

The action and typical use of each interface is described below. The transition numbers again refer to the process state transition diagram, Figure 3.2-1.

3.2.2.1 WAIT: This procedure places the running process into the waiting state (transition 3). Note that it requires no argument, since only a running process may place itself into the WAIT state. The highest priority ready process (priority and the ready queue are detailed later) is made running (transition 1). Procedure WAIT is called by the event handling procedure WAIT(E) after the event list structure is set up. It is also called by the timer management routine which puts a process to sleep for a given time after it places an entry in the timer queue. It is called when a process has requested exclusive use of a resource already in use. Note that the Process State Controller does not itself set up the structure to remember to ready the process. It is the responsibility of the higher level system routine to ensure that some process will invoke the READY(P) interface at the proper time. The same is true for the STOP(P) and RESUME(P) interfaces.

3.2.2.2 STOP(P): STOP(P) has no effect, other than generating diagnostic conditions, if P refers to the process executing the STOP(P); i.e., no process may stop itself. If P is in the ready state, STOP(P) places it in the stopped and ready state (transition 9). If P is in the waiting state, it is placed into the stopped and waiting state (transition 5). If P is running, the stopped indicator is set and the processor running P is interrupted, causing P to execute the EXCHANGE interface (see 3.2.2.5) which places P in the stopped and ready state (transitions 2 and 9 together). The processor executing the STOP(P) does not return to the calling process until P has actually been stopped by EXCHANGE. On return from STOP(P), P is guaranteed to be in the stopped state. In each case (P ready, waiting, or running) the stop count for P is set to 1. However, if P is already in the stopped and ready or stopped and waiting state, the stop count is just incremented by 1. STOP(P) can be used to correct a memory overcommitment or thrashing condition (see section 3.4) by stopping several low priority processes. It is used before terminating a process, or when debugging a process to freeze its action.

Figure 3.2-2: WAIT



INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

3.2.2.3 READY(P): If P is in the stopped and waiting state the procedure READY places it in the stopped and ready state (transition 7), and returns control to the executing process. If P is in the waiting state, it is placed into the ready state (transition 4), and its priority is tested. If P's priority is less than or equal to that of the lowest priority running process (call it P_ℓ), P is left in the ready state, and placed on a queue of ready processes. If P's priority is greater than $P\ell$'s, it is assigned $P\ell$'s processor (transition 1) and $P\ell$ is pre-empted to the ready state (transition 2). The pre-empted process $P\ell$ could be the one which executed the READY. Indeed this is very probable, since a READY is often executed by interrupt procedures, and system interrupts are usually assigned to the processor running the lowest priority process (see section 3.3). If the process executing the READY is not the one to be pre-empted, READY takes the following actions. It places P on the ready queue, interrupts the processor running Pe, then returns control to the executing process. In responce to the interrupt, the processor running Pe executes an EXCHANGE interface (see below), which swaps P and Pe. P becomes running (transition 1) and Pe becomes ready (transition 2). READY is called by the SET(E) event handling routine if a WAIT(E) or WAIT(n,E1,...,Em) condition is satisfied (see section 3.6), or by the timer interrupt routine if a time wait condition is satisfied.

3.2.2.4 RESUME(P): RESUME is very similar to READY. RESUME takes no action if P is not in the stopped and waiting or stopped and ready state. If it is, the stop count is decremented by 1. If the count is still positive, no further action is taken. If the count is zero, RESUME takes the same action as READY, substituting transition 6 for 7, and 8 for 4. Pre-emption and transitions 1 and 2 occur as in READY.

3.2.2.5 EXCHANGE: The purpose of EXCHANGE is to swap the current process with the highest priority ready process (transitions 1 and 2), if that ready process has an equal or higher priority. When a process executes an EXCHANGE because of an interrupt from a READY or RESUME, a higher priority process has already been placed on the ready queue. An EXCHANGE executed when there is no ready process of equal or higher priority has no effect, returning immediately to the process. If the EXCHANGE is executed by a process P in response to an interrupt coming from a processor executing a STOP(P), an indication has already been made of the pending transition 9 so EXCHANGE causes process P to enter the stopped and ready state (transitions 2 and 9). In this case, the highest priority ready process is always selected for transition 1, even if it is less than the

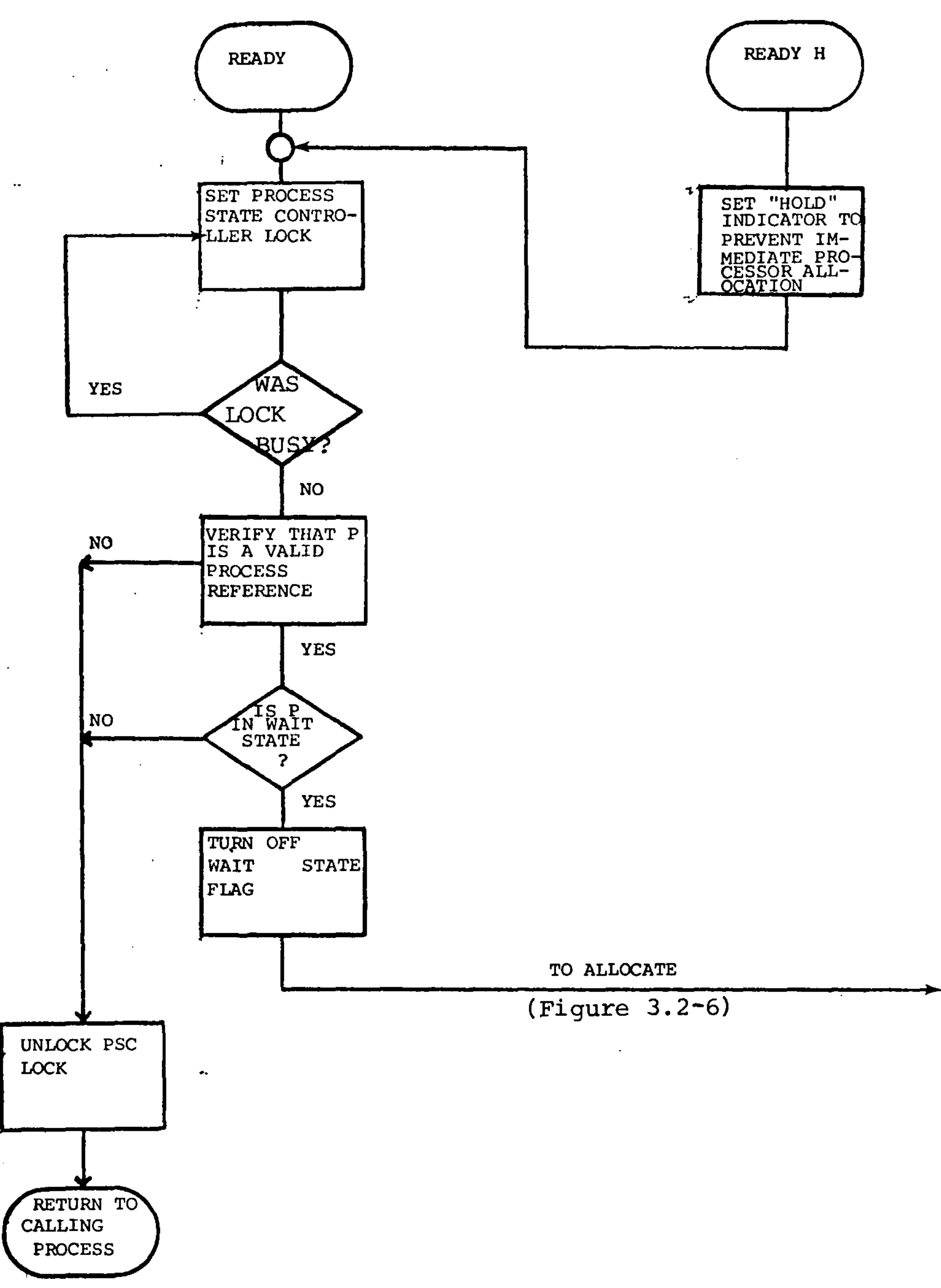


Figure 3.2-4 READY(P) and READY(H)

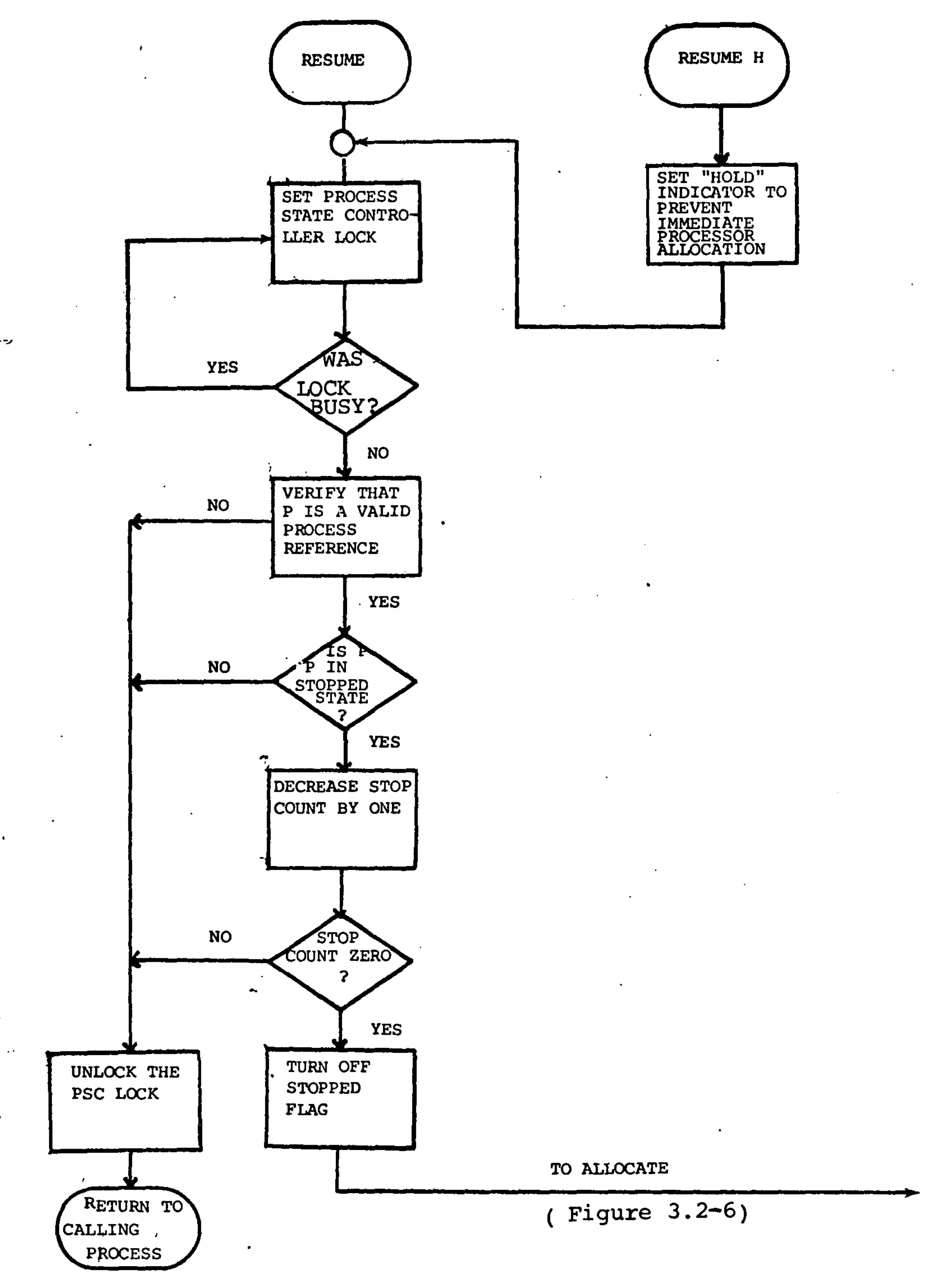


Figure 3.2-5 RESUME(P) and RESUME(H)

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

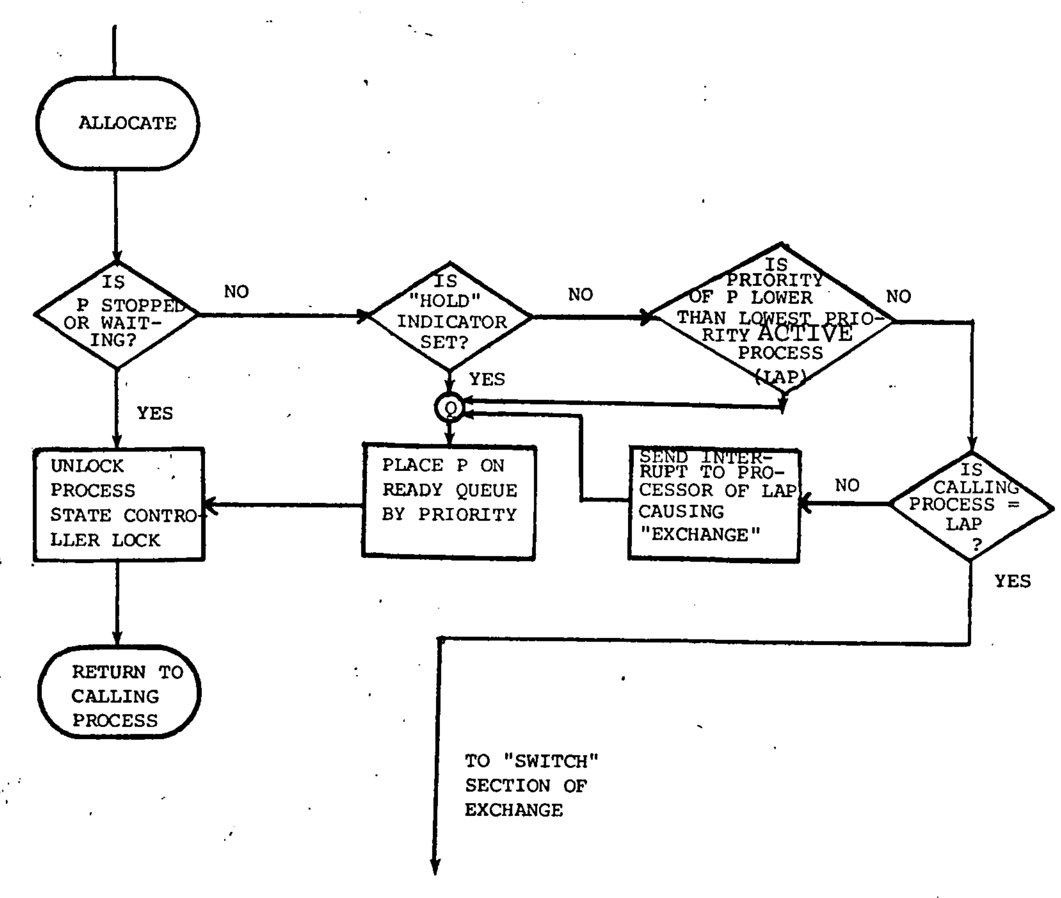


Figure 3.2-6 Common ALLOCATE Section

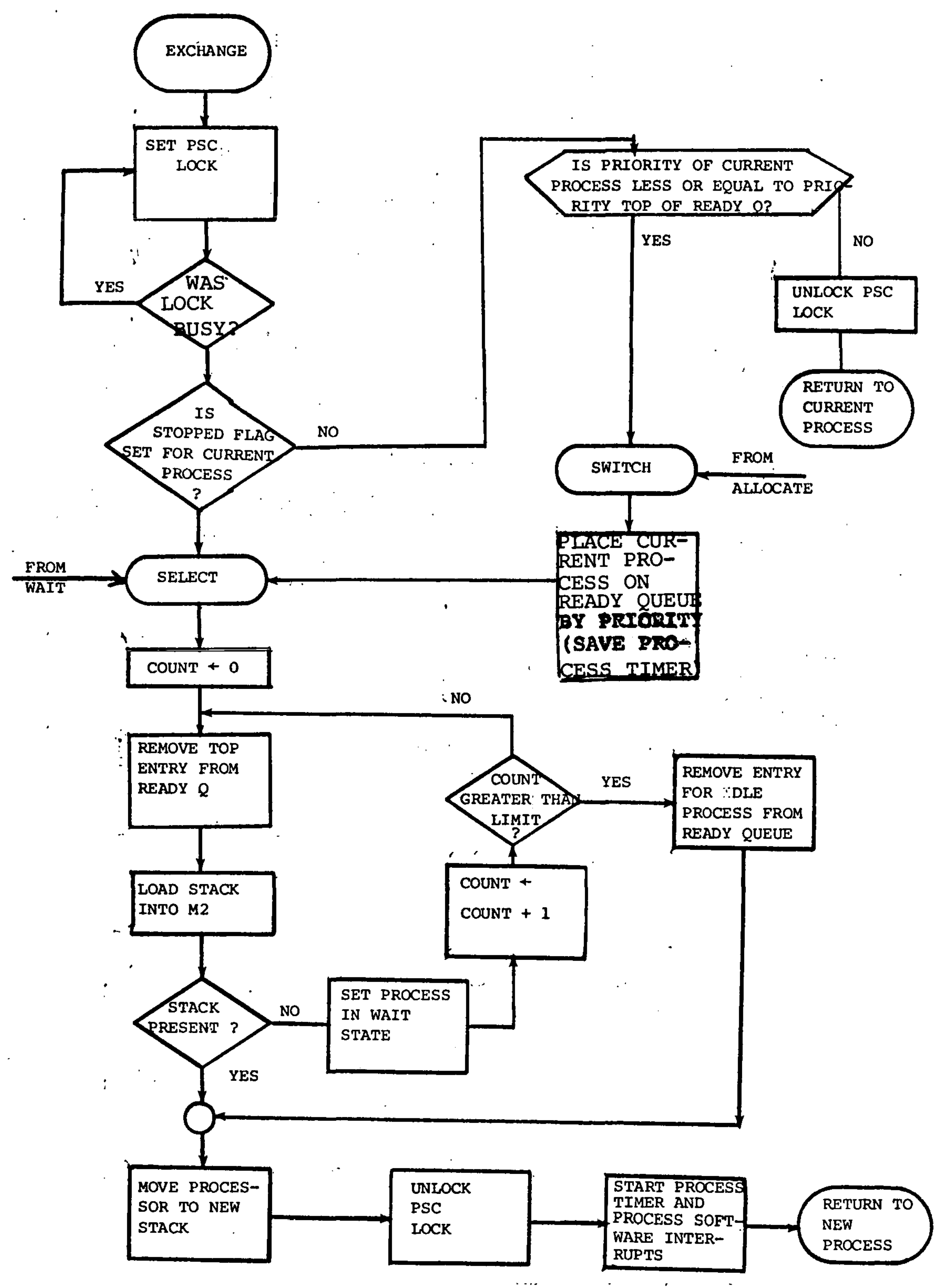


Figure 3.2-7 EXCHANGE

priority of P. EXCHANGE can be used to time-share processes of equal priority (see the discussion on priority and the ready queue). EXCHANGE can also be used for pre-planned multiprogramming for applications involving lengthy use of the processor by high priority processes.

3.2.3 The Process State Controller Data Base

The Process State Controller is concerned with processes as objects to manipulate. The information describing these objects is called the Process State Data Base, and includes stack and process identification, process state, priority, and ready queues. The Process State Data Base does not include the actual process stacks. This distinction is made so that the Process State Controller can make decisions about processes whose stacks are not present in M2. Stacks are relocatable objects, whereas the Process State Data Base is permanently resident system global data. The specific items in the data base are:

3.2.3.1 PROCESSID (Stack Number): An array indexed by stack number containing the unique processid for each stack. The value is non-positive if the stack is not a process (inactive or other use). The array is used to verify that the pair (stack number, processid) is a valid process reference.

3.2.3.2 LASTPROCESSID: An integer variable, initially zero, which is incremented each time a process is created. The new value becomes the processid of the new process. It is also the count of processes created since system initialization. This scheme provides a unique identifier for each process.

3.2.3.3 STATE (Stack Number): An array indexed by stack number which contains the state of every process stack. Possible bit assignment and state values are (numbering bits from right to left):

		Value	State
bit 1	0 - not running	0	ready
	1 - running		
bit 2	0 - not waiting	1	running
	1 - waiting	2	waiting
bit 3	0 - not stopped	4	stopped and ready
	1 - stopped	6	stopped and waiting

3.2.3.4 <u>PRIORITY (Stack Number)</u>: An array indexed by stack number which contains the priority of each process.

3.2.3.5 <u>STOPCOUNT (Stack Number)</u>: An array which contains the stop count of stopped processes. (STATE, PRIORITY and STOPCOUNT may all be merged into a single array for efficiency)

3.2.3.6 <u>PRIORITYOFPL</u>: A variable indicating the priority of the lowest priority running process, used by READY(P) and RESUME(P) to determine if pre-emption is necessary.

3.2.3.7 <u>PROCESSOROFPL</u>: A variable indicating the processor executing the lowest priority running process, used by READY(P) and RESUME(P) when pre-emption is necessary.

3.2.3.8 <u>PROCESS CONTROLLER LOCK</u>: A lock set by a processor to keep out other processors while executing the Process State Controller. A processor which tries to enter the Process State Controller while this lock is set must stall until it is unlocked. This guarantees integrity of the Process State Data Base in a multiprocessor environment.

3.2.3.9 <u>READY QUEUE (Priority Group)</u>: An array of ready queues indexed by priority group.

3.2.3.10 <u>QUEUELEMENTS</u>: An array whose elements are used for the READYQUEUE.

3.2.4 Priority and the Ready Queue

The concept of priority allows us to assign values to processes defining their relative importance. A process with a higher priority value is more important in some way and should be given preference by the system. In some general purpose operating systems the priority is often partially or completely determined by the operating system according to the type of load the process presents to the system (CPU, I/O and/or Memory requirements). Since this multiprocessing system is specified for a space station purpose, its operating workload will be generally more definable. Process priority can, therefore, be pre-determined and will not be automatically altered. Even the concept of processes changing their own priorities is not

considered, although if it becomes necessary a Process State Controller interface can easily be constructed to handle such added functions.

Priority may be used by any part of the operating system which must choose between servicing two or more processes if other policies such as first-come, first-served are not sufficient or desirable. The Process State Controller uses priority to decide whether a ready process should be made running. Given n processors and m processes in the ready or running state, it is the responsibility of the Process State Controller to see that the n highest priority processes are in the running state, with the other $m-n$ processes in the ready state. To guarantee that $m \geq n$, the system has pre-defined n "idle" system processes that are always ready (or running). These processes have the lowest possible priority, and do nothing but cause the processor to idle until it receives an interrupt. An idle process is made running only when there are no other ready processes.

Three categories of processes have been defined according to their required response characteristics -- real-time, interactive, and batch. These categories serve to divide the entire range of priority values into groups. If priority is assigned a 4-bit field with the range 0-15, four priority groups can be established using the two high order bits as selector:

a) 0-3, extra low (for the idle processes or distant background work),

b) 4-7, batch (response time not important),

c) 8-11, interactive (response time on the order of a second),

d) 12-15, real-time (response time on the order of milliseconds or less).

Perhaps the most important reason for establishing priority groups can be seen when we examine the need for a ready queue. The ready queue is an operating system structure which enables the processes in need of a processor to be selected quickly without testing the state of every process. (A process that is readied is not placed on the ready queue if it immediately becomes running, that is, if its priority and the readying process's priority are such that readied process pre-empts the readying process.) Processes are normally placed on the ready queue in order of priority so that the highest priority ready process is always first in the queue, and can be selected for running with the

minimum possible overhead. Placing a ready process in the queue entails more overhead than its selection, however, since the queue must be searched until the proper place is found. Therefore, to reduce this overhead, one ready queue for each priority group is defined. The real-time ready queue is kept free of lower priority processes, and is always checked first. If it is empty, the next lower priority ready queue is examined. The extra expense of checking more than one queue is spent only when it can be afforded, increasing appropriately for decreasing priority groups. The overhead of placing a process on the ready queue is reduced for all priority groups, since the two high order bits of the priority immediately select the correct queue. The two (or more, if desired, to increase priority discrimination within a priority group) low-order bits determine placement within the selected queue. A process is placed behind other processes of equal priority. Hence for processes of equal priority, the ready queue operates as a first-in, first out (FIFO) queue. This property can be exploited in conjunction with the EXCHANGE interface to effect time-sharing of equal priority processes. An entire priority group may be time-shared by not using or ignoring the low-order bits. The time sharing can be voluntary and pre-planned, i.e., the process willingly issues an EXCHANGE at pre-determined points, or involuntary. An example of the latter is a process which is forced to execute the EXCHANGE through the timer interrupt routine every time slice.

3.2.5 Process Creation and Termination

The process state transition diagram in section 3.2.1 gives no indication of how a process is established in one of the described system states. Most processes are created dynamically by other processes, have a definite beginning and are usually definitely terminated. There are a few system processes which exist for the life of the system. These are the idle processes, which "run" when there is nothing to do, and one important system master process, which will be described later. The remainder are created and terminated as work is generated and finished.

Additional states, transitions, and process state controller interfaces must be defined before the process state diagram can be enlarged to include process creation and termination. The new states illustrated in Figure 3.2-8 are

a) Stack number available,

b) Non-process stack

c) Dead process stack

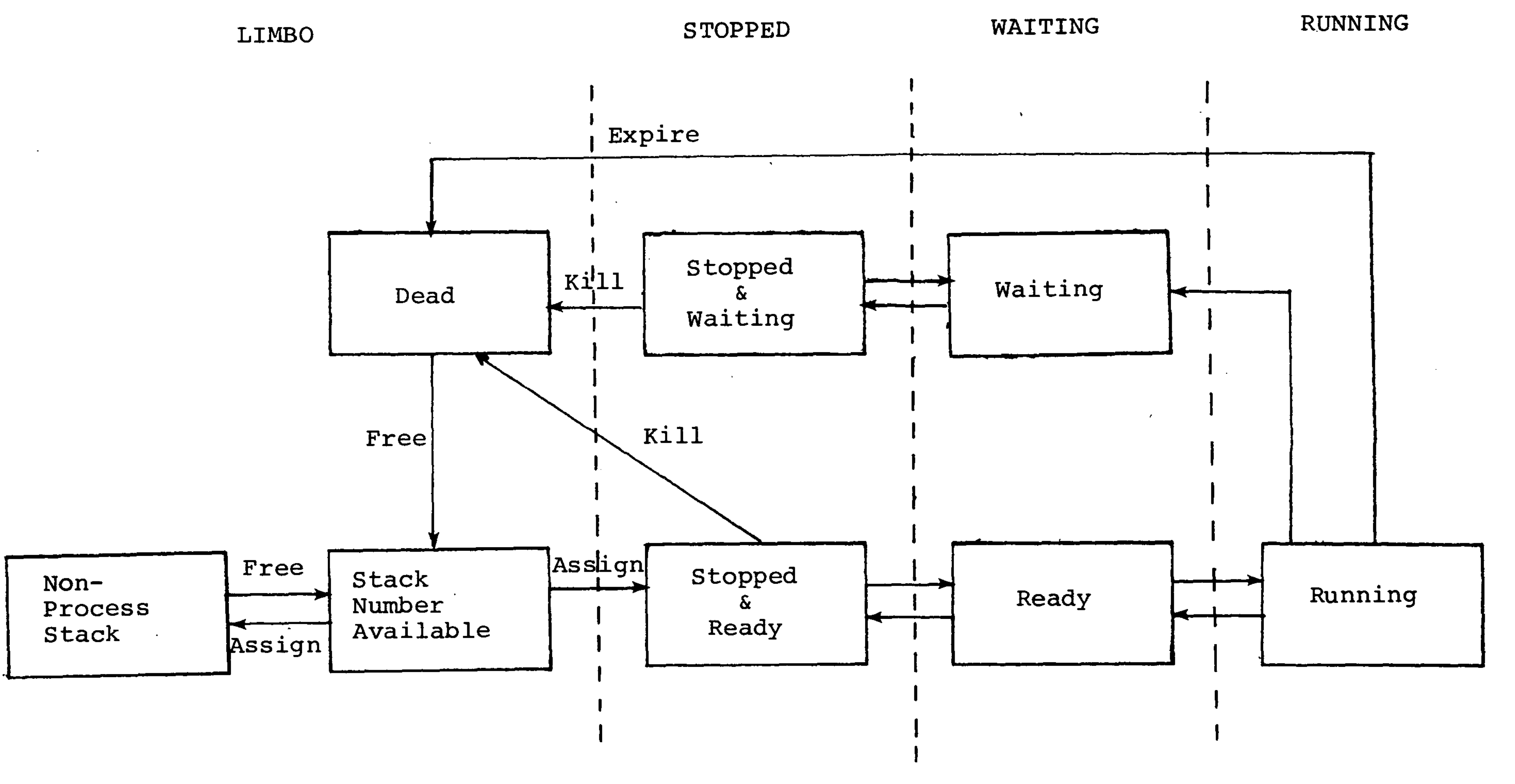


Fig. 3 .2-8 Complete State Transition Diagram

Stack number available means that the previous use, if any, of that stack number has ended, and the actual memory area for the stack has been freed. The stack number is an index to an array of descriptors termed the stack vector (see section 3.1.2). An available stack number is one which indexes to an empty slot in the descriptor segment free to accept a descriptor for a new stack. Stacks may be used for Compools and segment dictionaries as well as processes, and since the Process State Controller data base carries state information for every assignable stack number, the use of a stack for non-process purposes must be indicated. The third state, the "dead" state, is one which all terminating processes enter. A terminating process cannot be expected to free its own stack. Therefore, the stack cleanup and release function is handled by the system master process. The master process takes a process from the dead state, performs any necessary cleanup functions, and finally frees the stack so the stack number will be available again. Four new process state controller interfaces carry out the transitions. They are ASSIGN, FREE, KILL, and EXPIRE, and the transitions are identified in Figure 3.2-2.

3.2.5.1 <u>ASSIGN(P, Stacksize, Stacktype)</u>: ASSIGN finds an available stack number, obtains a memory area of the correct size, and enters the descriptor in the stack descriptor array. Stacktype is used to distinguish between process and non-process use, and P is set on return to contain the stack number and unique processid. If Stacktype indicates a process stack, the state on return from ASSIGN is stopped and ready.

3.2.5.2 <u>FREE(Stack Number)</u>: FREE is the reverse of ASSIGN. If the stack to be freed is a process stack, it must be in the dead state or no action is taken.

3.2.5.3 <u>KILL(P)</u>: This interface takes a process in the stopped state and places it in the dead state. No action is taken if P is not in the stopped state. No cleanup is performed. This is a simple bookkeeping function designed to isolate the irreversible step in termination.

3.2.5.4 <u>EXPIRE</u>: EXPIRE is similar to KILL except that it allows a process to place itself into the dead state. Obviously the process must be in the running state.

3.2.6 Dependent and Independent Processes

3.2.6.1 Introduction: There is more to process creation and termination than the simplified interfaces described above. The higher level system procedures INITIATE, SPROUT, and TERMINATE have responsibility for the births and deaths of processes. They carry out all the detailed work, calling on ASSIGN, KILL, and EXPIRE only at the critical transition points. FREE is executed by the system master process after the cleanup action on a dead process. INITIATE and SPROUT create independent and dependent process respectively. The difference between the two types of process is based on the concept of owning and sharing data. System global data, including Compools (to be described in section 3.4.) are defined and addressed at lexical level 0. At level 1 are defined the procedure segments, shared by processes using the same procedures. These two levels are not owned by any one process exclusively, but by the system. At level 2 and above exists the dynamic data, distinct for each process. Dynamic data (or its descriptor) is located in the process stack, and is therefore owned by that process. The data exists only while the process exists. A process can share the data owned by another process, creating a dependency on that process. The sharing of data owned by other processes can occur in two ways. The following example, which is coded in a mythical (but typical) block structured language, illustrates process dependencies:

```
2  ┌ABLE: PROCEDURE;
   │DECLARE A:
   │
   │3  ┌BAKER: PROCEDURE;
   │   │DECLARE B;
   │   │
   │   │4  ┌CHARLIE: PROCEDURE;
   │   │   │DECLARE C;
   │   │   │
   │   │   │5  ┌DONNA: PROCEDURE;
   │   │   │   │DECLARE D;
   │   │   │   │      .
   │   │   │   │      .
   │   │   │   │      .
   │   │   │   └CLOSE;
   │   │   │
   │   │   │SCHEDULE DONNA AS PROCESS 2;
   │   │   │CALL DONNA;
   │   │   │SCHEDULE BERYL(C) AS PROCESS 3;
   │   │   └CLOSE ;
   │   │
   │   │CALL CHARLIE;
   │   └CLOSE;
   │
   │3  ┌BERYL: PROCEDURE(X);
   │   │DECLARE X;
   │   │X = X+1
   │   └CLOSE;
   │
   │CALL BAKER;
   └CLOSE ;
```

The first dependency to be described is due to a common lexical level. Referring to Figure 3.2.9, Process 1, while executing the procedure CHARLIE, schedules Process 2, which starts to execute the procedure DONNA, declared at lexical level 4 in Process 1. DONNA may reference data declared within the containing lexical levels 2 through 4 in Process 1's stack. References to data declared at level 5 and above are local to Process 2. The procedure DONNA, and therefore the continued execution of Process 2, is dependent on the existence of levels 2 through 4 in Process 1.

The second type of data sharing occurs when a new process is passed a call-by-reference parameter. An illustration of this is the scheduling of Process 3 from level 4 of Process 1, with passage of parameter C. The procedure BERYL in the new process shares the common lexical level 2, but a reference to the dummy parameter X is actually a reference to the passed parameter C owned by process 1, and declared at level 4. This level therefore is a critical level (see section 3.2.6.3) to process 3. That is, Process 1 may not exit level 4 before Process 3 has terminated.

A process can create more than one dependent process, which in turn can create other dependent processes. Thus a family tree structure of processes is possible, with a process having exactly one parent (on which it is dependent), any number of sibling processes with the same parent, and any number of offspring processes (which are dependent on it). At the root of the structure is always an independent process. An independent process is one which owns all of the data at and above the lexical level 2 that it references. The global data and Compools at level 0, and the procedure segments at level 1 are owned by the system, not by any process. Independent processes are thus dependent only on the existence of the system itself.

3.2.6.2 SPROUT and INITIATE: These similar system procedures each have three parameters:

a) Procedure reference and parameters. This defines the initial procedure to get control in the new process, as well as any parameters for the procedure.

b) Process start-up variable. This is a structural variable which defines certain characteristics of the new process, such as priority, stack size, total time limit, initial time conditions, file requirements, etc.

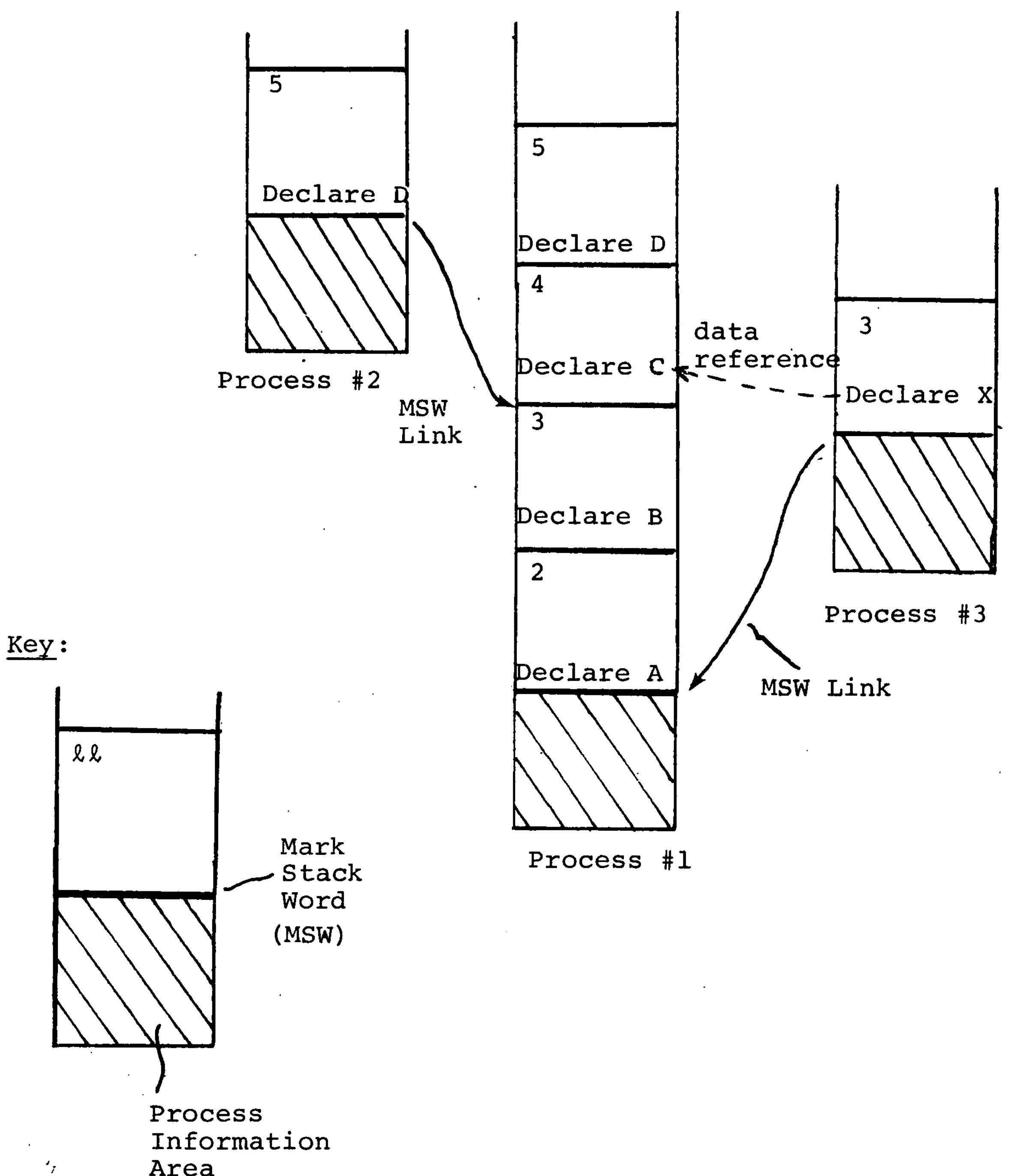


Figure 3.2-9
Process Dependencies

c) Process reference. This is the pair (Stack Number, Processid) which will be passed back to the caller of SPROUT or INITIATE so the originating process can refer to the new process.

The first item is restricted to INITIATE. For an independent process, the procedure must be defined at an outer level. It cannot be imbedded in any other procedure, else the new process would share data owned by another process. Any parameters for an independent process must be passed by copied value only to prevent dependency.

3.2.6.3 CRITICALEXIT: The above restriction, of course, does not apply to SPROUT. The important thing for dependent processes is that the data on which they depend remain available until they have terminated. To guarantee this, the compiler must determine the critical level, which is the innermost lexical level which defines the procedure or parameter specified in SPROUT. The creating process must be prevented from leaving this level and destroying the shared data until the new process has ended. The compiler inserts a call to the system procedure CRITICALEXIT just before exit from a critical level. CRITICALEXIT when executed checks the count of processes dependent on that level. If it is greater than zero the process enters the waiting state. The count is incremented by SPROUT and decremented by the system procedure TERMINATE when a process ends. TERMINATE also issues a READY(P) when the count reaches zero and the process is waiting for its dependents to complete.

It is possible that the initial procedure supplied to SPROUT is at the outermost level and potentially could have been initiated as an independent process. In such a case, the new process is dependent only because it was created by SPROUT, and the critical level is the outermost level of the creating process.

The steps SPROUT and INITIATE take to create a process are as follows:

Step	Action
1.	Call ASSIGN to obtain a process stack
2.	Create the procedure segment dictionary, if necessary.
3.	Initialize the process information area of the stack.

Step	Action
4.	Force the stack to look like an interrupted process just before the first instruction to be executed.
5.	Set the priority of the process.
6.	RESUME(P) to make the transition from stopped-and-ready to ready. If there is an initial time condition, call on time management to issue the RESUME(P) at the proper time.

Step 1) results in a process in the stopped and ready state with undefined priority, and whose stack is bare of structure. The structure is added in steps 3) and 4). Step 2) obtains a stack as in step 1), but for use as the segment dictionary. This step can be by-passed if a process using the same procedures already exists, which is the usual situation for dependent processes. The segment dictionary was constructed by the compiler and is loaded from M3. It contains the Mom-descriptors for all the code segments to be used by the process.

Step 3) sets up the process information area, a fixed-length section at the bottom of every process stack. It contains the status and bookkeeping information about a process, and is accessible by both the system and the process. Some of the expected items to be found here are:

a) Process type: independent/dependent, system/application

b) Wait state type: cause of entering waiting state, e.g., event, time, I/O, busy resource

c) Process family links: a process reference to parent, first offspring, previous and next siblings.

d) Parent's critical level: for a dependent process, the critical level of its parent process

e) Interrupt response vector: a descriptor for each category of interrupt which specifies what interrupt action is to be taken (see section 3.3 for further explanation)

f) Resource list: a bookkeeping list identifying what resource names are assigned to this process

g) Processor timer limit: amount of allocated processing time left for this process

h) The time the process was last in the running state

i) Software Interrupt Queue: used to record occurrences of software interrupts while process is not running, so action can be taken when the process next becomes running

j) Process reference of this process, the pair (Stack Number, Processid)

Step 4) does the final stack initialization. The mark stack word (MSW) which identifies a new lexical level and records the containing lexical level (addressing environment), and the return entry word (REW) which records procedure return information are artificially entered in the stack. The result is a stack which looks as if it had been interrupted just before executing the first instruction of the first procedure. The return entry word for the outermost level cannot specify any procedure to which to return, since there is none. The action taken by the exit operator when attempting to exit through this REW is to cause an interrupt to the system TERMINATE routine.

Step 5) simply sets the specified priority in the Process State Controller Data Base. It is safe to do this because no other process will be accessing the priority field for the new process, even if another process has entered the Process State Controller on another processor. The final step depends on whether the process is to be executed immediately, or at some later time. In the former case, a RESUME(P) is issued, leaving the new process in the ready state. If there is to be an initial time condition, SPROUT or INITIATE calls on timer management routines to enter a request for a RESUME(P) at the proper time. It then exits, leaving the new process in the stopped and ready state.

3.2.6.4 TERMINATE: The system procedure TERMINATE is called explicitly when a process wishes to terminate itself or another process, and implicitly via an interrupt when the outermost procedure of a process is exited. TERMINATE has three major functions:

a) It leaves the process to be terminated and all of its dependents in the dead state via KILL(P) or EXPIRE.

b) It decrements the count of dependent processes in the critical level of the parent process, making it ready if appropriate.

c) It notifies the system master process via SET DEATH (DEATH is a global event) that one or more processes have entered the dead state. It also enter all the dead processes in a system dead queue for easy access by the system master process, which must perform any cleanup functions and free the stack.

The TERMINATE procedure is designed to terminate a process and all its dependents without recursion. This makes it safe for an offspring process to terminate a parent, which indirectly causes self-termination. A local list is kept of all processes being terminated, so the process and its dependents are all added to the dead queue at one time, and the event DEATH is set only once.

If the executing process is being directly or indirectly terminated, it is placed into the dead state via EXPIRE only as the last action of TERMINATE, to allow completion of the entire operation. Otherwise, TERMINATE returns to its caller.

3.2.6.5 <u>The System Master Process</u>: The master process has three functions:

a) It interacts with the operator to respond to commands and supply information about the state of the system.

b) It starts jobs by creating independent processes via INITIATE. It would do this as a response to an operator command, or to any parameter in the system to which it is sensitive.

c) It responds to the system event DEATH by performing stack cleanup operations and finally freeing the stack. Stack cleanup involves:

   1) freeing any data segments pointed to by Mom descriptors in the stack

   2) freeing the segment dictionary and all procedure segments if they are not in use by another process

   3) freeing any system resources still allocated to the process

   4) issuing a FREE (Stack Number) to release the stack area and make the stack number available.

3.2.7 Minimum System State

A system which functions only when a complex supporting structure exists always provokes the question: how is the supporting structure itself created? The answer is to find the absolute minimum structure from which the system can function sufficiently to create the full structure. The minimum structure is then pre-defined, pre-created, and entered into the system using only the most rudimentary abilities of the machine. The method by which this minimum state is loaded is an implementational detail, but the definition of the minimum system state is properly a part of the system's design.

Lexical level zero, the system global level, is a permanently resident stack with the following minimal structure:

a) a descriptor for the stack descriptor array

b) descriptors for all permanently resident system procedures such as:

1) Segment fault handler

2) Process State Controller Data Base and procedures

3) Memory Management procedures

4) Process creation

5) System resource allocation

3.2.8 Access Control Mechanisms

3.2.8.1 Introduction: One generalized system function can handle the following similar types of access control needed to guarantee integrity in a multiprocessor system:

a) Locking of shared data so that it can be safely read and modified by several simultaneous processes

b) Control over both sharable and non-sharable hardware and software resources

c) A mutual exclusion function so that a section of code can be bracketed with interlocking turnstyles, guaranteeing that only one process at a time may be executing it.

These functions are all provided with the pair of system procedures called ACQUIRE and RELEASE. The items acquired and released are actually just names, which by common, compiler-enforced agreement are associated with whatever data, resource, or code is to be controlled.

ACQUIRE and RELEASE have a variety of options and modes of action:

3.2.8.2 ACQUIRE(Mode, Request List):

a) Mode: Wait, or Nowait

Wait specifies that the calling process be placed into the wait state until all of the items in the request list are available. Nowait means that the calling process wishes immediate return with an error indication if not all of the items are available.

b) Request List:

A list of the item names and the type of control desired for each. Each entry in the list has the following structure:

1) Major Name: A name which assigns the item to a category group

2) Minor Name: A name which uniquely identifies the item within the category group

3) Type of Control: Shared, Update or Exclusive

The type shared is used if the item under control is unmodified by its use, e.g., read-only access to shared data. Any number of processes can ACQUIRE the same item with shared control. The type update is used for items which are to be modified by using a temporary copy of the data. This allows one process to have update control while any number still have shared control. At the end of successful computation involving the modification of the temporary copy, the process with update control requests exclusive control. When exclusive control is granted, the process may safely copy the modified temporary back to the original data.

Exclusive control guarantees that no other process has any type of access to an item. This is used with items which are changed by use, e.g., the writing of shared data, or in the control of a non-sharable resource.

3.2.8.3 RELEASE(Item List): Each entry in the Item List specifies an item for which acquired control is to be released. The item is specified by its Major and Minor names. The release of an item can result in the waking of another process waiting for the availability of that item.

Items are designated by Major and Minor names to allow the system to determine efficiently whether or not an item is available. The system must keep a list of all items acquired, and the major names provide a means of grouping items into categories, and hence limiting the list searching that needs to be done. For example:

a) A data structure potentially used and modified by a number of processes is to be read by a process

```
all ref-        ┌ACQUIRE(WAIT, DATA, ABC, SHARED) waits until no other
erences         │process has ABC exclusive
to ABC          │
are iso-        │- - - )
lated to        │- - - }      references to data structure ABC
this block      │- - - )
                │
                └RELEASE(DATA, ABC) releases control of ABC
```

Note: The major name is DATA, identifying the item category. This could be any name, as long as the compiler ensured that all programs use the same name for the same category. The minor name, ABC, is actually the location of ABC's Mom descriptor. The "name" is then uniquely determined by this bit string. If any other process is modifying ABC, this process would wait at ACQUIRE until the data is modified.

b) The same data structure is to be modified.

```
┌ACQUIRE(WAIT, DATA, ABC, UPDATE) waits until no other process
│has update or exclusive control
│
│TEMPABC ← ABC makes copy of data to be modified
│
│- - -     references to ABC translated to references to TEM-
│- - -     PABC. Modifications are made only to TEMPABC.
│- - -     ABC is left unchanged.
│
│ACQUIRE(WAIT, DATA, ABC, EXCLUSIVE) waits until only this pro-
│cess has control. Any shared control by other process has been
│released at this point.
│
│ABC ← TEMPABC copy is used to update original
│
└RELEASE(DATA, ABC)
```


C.3

c) A section of code must be made exclusive (one process at a time)

```
      ┌ ACQUIRE(WAIT, CODE, XYZ, EXCLUSIVE)
      │
XYZ:  │
      │
      │      section of code to made exclusive
      │
      │
      └ RELEASE(CODE, XYZ)
```

d) A non-sharable device (e.g. PRINTERA) is to be allocated to a process

```
┌ ACQUIRE(NOWAIT, DEVICE, PRINTERA)
│
│
│ Continue with use of device
│ .
│ .
│ .
└ RELEASE(DEVICE, PRINTERA)
```

If the device is busy a "PRINTERA IS BUSY" is generated.

3.2.9 Process Timing

There is a "watchdog" timer for each process, to keep track of and optionally limit the time spent by a process in the running state. When a process is made running, the limit value is placed in the timer, which is a countdown device and causes an interrupt when the count reaches zero. Whenever a process is taken out of the running state, the value left in the timer is used to update the current process time limit. The occurrence of the watchdog timer interrupt can be ignored, used to terminate the process, or cause operator notification, depending on the type of process.

Other types of timing provide information useful for testing and evaluating system performance. The Process State Controller needs the real time clock to record the time when a process is taken out of the running state. This can give information about the length of time a process is waiting or stopped. It also records the time a process is placed on the ready queue, so if necessary the delay between when it becomes ready and when it is made running can be determined.

3.2.10 A Note on Process Stack Handling

To maintain the integrity of the system the processor executing a Process State Controller procedure must be disabled against asynchronous interrupts to avoid the loss of control that would otherwise be caused by interrupt handling. However, since it is impossible to disable the presence-bit interrupt, any attempt to run a process whose stack has been made not present must be avoided. The Process State Controller calls on the memory management procedure LOADSEGMENT (See section 3.4.7.2) to insure that the stack it is about to make running is in fact present in M2. If the stack is already present, the return from LOADSEGMENT is immediate, and the process is made running. If LOADSEGMENT returns an indication that the stack is not present, the Process State Controller can do one of two things. It can

a) make the next highest priority process running, or

b) idle until the stack is made present.

The first choice involves some probability that the next lower priority ready process could also be absent from M2. Loading this and any further stacks when only one of them (the one of highest priority) will eventually run, uses up M2 space. In situations such as these it is highly likely that memory is in short supply, so that needlessly allocating even more could encourage thrashing (see section 3.4). A decision to idle the processor at this point is not unwarranted, since very shortly the right candidate for processor allocation will be present. The wait for segment I/O to complete may be less deleterious to overall throughput than a collapse caused by overcommitted memory.

The ideal solution is to have a counter in the Process State Controller, whose limit value is a parameter which can be adjusted during system testing and evaluation. The counter is set to zero when the Process State Controller first examines the ready queue for a ready process. If the stack is not present, it is loaded into memory and the counter is incremented. If the counter limit is exceeded, the processor makes one of the idle processes running. If it is not exceeded, it proceeds to select the next higher priority process. The process whose stack is being loaded is left in the waiting state so it can be made running when memory management issues the READY(P).

3.3 Interrupt Handling

When a condition requiring the interruption of a processor occurs, several steps must take place:

a) An eligible processor must be selected to handle the interrupt;

b) the status of the interrupted process must be recorded, to enable its resumption at a subsequent time;

c) a description of the interrupt condition must be made available to the interrupt-handling procedure;

d) the interrupt-handler procedure must be selected and entered.

3.3.1 Processor Selection

The rationale for selection of a processor is illuminated by considering the list of interrupts which can occur. There are two major classes of interrupts, distinguished by the scope of their effect. The first class concerns conditions directly related to a specific process; the second class includes all the remaining conditions of system-wide effect. Each class may be subdivided further according to whether the interrupt is synchronous, i.e., a direct consequence of execution of a particular instruction, or whether the interrupt is asynchronous to the instruction sequence. In the latter case, the condition results from some other mechanism, such as time, completion of an I/O operation, etc. The interrupts defined for the current multiprocessor design are described in detail as follows.

a) Process oriented

  1) Synchronous

    a) Arithmetic traps

      1) DP Floating Point Overflow. The result of an add, substract, multiply, or divide operator exceeds the capacity of the double precision floating point word format; therefore, the result cannot be expressed.

      2) DP Floating Point Underflow. The result of an add, subtract, multiply, or divide operation is too small (nonzero) to be expressed in the double precision floating point format.

3) SP Floating Point Overflow. In a single precision store operator, the exponent field of the word to be stored is larger than that which can be contained in the exponent field of a single precision word. Thus, the result cannot be expressed.

4) SP Floating Point Underflow. In a single precision store operation, the exponent field cannot be expressed in a single precision word format because the value is too small (nonzero).

5) Division by Zero. The divisor in a divide operator has a zero value. The result is undefined.

6) Integer Overflow. In a floating-point to integer conversion, the number of integer bits exceeds the length of the mantissa of the floating point word; the result cannot be exactly expressed as an integer.

7) Conversion Fault. A character to integer conversion operator encountered a character other than 0 - 9.

b) Control Traps

1) Illegal Index. The integerized index of an array was found to be less than zero or larger than the length of the array.

2) Illegal Operation Code. The operator selected by the processor represents an unimplemented operation.

3) Illegal Operand. Many types are possible; for example, use of a value-operand in a context where a descriptor, PEW, or other non-value operand is required.

4) Apparent Software Fault. Two cases are presently defined: the absent segment trap is sensed by a processor for which interrupts are inhibited, and interrupts are inhibited by a processor for an excessive length of time.

5) Illegal Address. The address generated for an M2 operation does not correspond to an M2 memory location.

6) Invalid Stack Number. The stack vector entry corresponding to a specified stack number is not assigned to a stack.

7) Invalid Stack Offset. The offset specified for a stack exceeds the current length of that stack.

8) Conformability Error. The arrayness of the operands of an instruction are not compatible; for example, the attempted addition of two linear arrays of different lengths.

2) Asynchronous

a) Stack Bounds Violation. A stack has grown in length beyond the upper limit of the available space for that stack, or a stack link is peeled back below the stack marker for the link.

b) Watchdog Timer Runout (process). The processor timer for process execution timing has reached the limit to which it was set.

c) Watchdog Timer Runout (instruction). The processor timer which times execution of each instruction has run out, indicating that an instruction completion has not occurred for too long a time.

b) System Oriented

1) Synchronous

a) Absent Segment Trap. A load or store operation has addressed a segment not currently resident in M2 memory.

b) Processor Error. The error control logic of a processor has signalled a disagreement in a result generated by the processor.

c) Memory Error (process). The execution of an instruction by a processor has encountered an M2-memory error.

2) Asynchronous

a) I/O Completion. A requested I/O operation has been completed without error.

b) I/O Data Error. A requested I/O operation has been completed with a signal of data error.

c) I/O Controller Error. A fault has been signalled by the I/O controller error detection circuitry.

d) Timer. A match has occurred between the system clock and the clock interrupt register.

e) External Signal. One of a specified set of external signals has occurred.

f) Processor Interrupt (IPC). One processor has directed that another processor be interrupted.

g) Memory Error (I/O). The execution of an I/O operation by the I/O controller has encountered an M2-memory error.

The placement of the stack bounds-violation and instruction-timer runout interrupts into the asynchronous category of process oriented interrupts should be explained. Both are, in a sense, related to execution of an identifiable operator by a processor. However, in the stack bounds-violation instance, the operator involved may have been fabricated by the processor in response to an interrupt signal; the operator thus would have no real physical existence in a code segment. In the timer-runout case, the point or progress of execution is unpredictable, since the signal from the timer is truly asynchronous relative to the execution sequence. Hence, both of these interruptions fail to fully satisfy the "synchronous" conditions, and are excluded from synchronous category for this reason.

Process oriented interrupts may readily be handled by the processor on which the related process is being executed. The same is true of the absent-segment trap, which is specifically caused on behalf of a process: however, it is listed among the system interrupts for two reasons:

a) It is handled by a procedure which is considered to be part of the operating system, since management of a global resource (memory) is involved.

b) The time-duration of the anticipated response may make it desirable, from an efficiency viewpoint, to reassign the processor, rather than to make it idle until the segment becomes present.

That the second of these points may be invalidated by by use of a high-performance secondary storage device is not particularly germane to the current discussion.

The Processor Interrupt also implies a preference for a particular processor. The Processor Error and Memory Error interrupts are synchronous, and seem to have an affinity for the processor which was involved. However, the Processor Error interrupt may be an indication that the processor cannot execute instructions satisfactorily, in which case its attempt to handle the interrupt could cause an endless loop. The Memory Error interrupt could have arisen at the M2 stack-top during a process's attempt to enter a procedure; if so, an attempt to enter the Interrupt-Handler procedure will probably result in a loop. Thus, in both cases, a processor other than the one involved is selected.

Consequently, all system wide conditions, except the Absent-Segment Trap and Processor Interrupt require a means for selection of a processor. This function is provided in the I/O controller, in accordance with an interruptibility index maintained there for each processor. It suffices here to say that after "disqualification" of certain processor (e.g., as just described for the Processor Error condition), the processor chosen has the highest probability of immediate assignment to a process readied by the logical consequence of the interrupt. Loosely speaking the eligible processor executing the lowest priority process is designated.

3.3.2 Status Recording

Occasionally, the interruption of a process is not followed subsequently by its resumption. Much more frequently, however, resumption does indeed take place, albeit at a later time, perhaps on a different processor, and perhaps in an entirely different area of memory. It is a natural consequence of the real-time response requirements which the system must satisfy, and the choice of an interrupt-driven priority scheduling algorithm for assigning processors to processes, that running processes can be temporarily suspended to allow performance of some urgent function. Thus it is necessary to record all information essential to the restarting of the process when the interrupt procedure is entered.

Interrupt response is treated as the involuntary entry of a procedure. Whereas in the normal entry to a procedure segment, the steps required to mark the stack, supply arguments (pass parameters), note the return location, and achieve entry, are all accomplished by operations explicit in the calling procedure, these must be performed automatically by the processor accepting an interrupt. This difference is more readily dealt with than is the problem which results from the occurrence of a disabling failure in a processor during the execution of an instruction. Yet, in this case, the design philosophy of the system dictates that the victim-process must survive. Hence, not only must the status information be allocated sufficient space in the execution stack area between caller and called procedure, but each processor must maintain its essential status record in an externally accessible memory. Thus, the processor error-detection components need only signal the occurrence (and type) of failure, and render the processor dormant; the task of rescuing the victim process and continuing its execution at an appropriate time is given to another processor.

The status which must be retained includes the instruction pointer to the return location in the interrupted procedure, an operator-dependent completion indicator, and the dynamic status information contained in special processor registers during execution, such as current lexical level, condition code, interrupt mask, and current stack-top register. The instruction-completion indication mentioned above is used to specify the degree of progress which had been made in the execution of the operation designated by the return pointer. Normally, the indication will reflect the usual circumstance of interrupts being accepted between instructions; the operator being returned to will thus not have been initiated. Three cases cause other behavior:

a) Interruptions which are integral to the execution of the operator, and require resumption rather than restart upon return;

b) Interruptions allowed in the middle of array operations, for enhancement of response time;

c) Interruptions due to faults, for which resumption rather than re-execution of the operator is logically necessary.

In all cases, the completion indicator is treated in conjunction with the operator at the return point as a more specific description of the operation to be performed upon return. Viewed in this way, it is loosely analogous to the fractional part of a number adding a more detailed description of the value than provided by just the integer part.

An additional item of status information is the processor timer. This register is used as a combination watchdog and accounting tool. When a processor is assigned to a process, the timer is loaded with a limit value, and as the process execution occurs, the timer is counted down. Should it reach zero, the Watchdog Timer Runout interrupt is signalled; depending upon the policy desired, this event can be used to denote an error (process-time overrun), or to implement process time-slicing. In any event, the process timer value must be stored in the stack of its process when the processor is taken away, and restored when the process is subsequently made to run again.

The preservation of the process time is complicated by the fact that unknown and unnumbered system interrupt handlers preempt the processor and make use of the stack of the interrupted process. Several philosophical approaches are reasonable: First, the processors may be provided with two timers, one for the application process and one for the interrupt handlers. A bit in the PEW on entry (or REW on exit) would specify which of these was to be incremented. A second approach would retain the single timer, but the interrupt handler would be required by convention to exchange the process's value with its own timer load-value at the point of entry to the interrupt procedure; exit would perform the inverse action. Still a third approach resulting from pragmatic considerations, is reasonable. A determination is made by some means for each interrupt handler: If the execution is thereby deemed "short", the interrupted process would be charged for the time spent in the interrupt handler; if the time were judged "long", then a timer contents exchange would be performed, as in the second approach.

The third method is preferred, since it is believed equitable (the boundary between "short" and "long" can be set to a value commensurate with the desired precision of accounting), and only one timer is required in each processor for this functional purpose. The saving of the timer value upon reassignment of a processor is left to software.

3.3.3 Interrupt Description

At the time when the stack is marked in preparation for entry into an interrupt-handler procedure, the processor places two words into the stack as though transmitting them as arguments to the called code segment. The first of these words indicates the nature of the condition causing the interrupt, while the second is used to provide supplementary data. The format of the second parameter word depends upon the interrupt condition specified by the first parameter. For example, in the Absent Segment Trap interrupt, the second parameter is the location (stack number and offset) of the Mom descriptor for the absent segment.

3.3 Interrupt-Handler Selection

Because of the wide variety of applications in which the multiprocessor is expected to be utilized, it is desirable to allow the largest degree of flexibility in interrupt handling. It is therefore planned to implement a mechanism for specification of interrupt responses which is quite general. This mechanism will be made available to a system user only through high-order language facilities authorized for his use; consequently, without the need for hardware-imposed surveillance, a wide range of interrupt-handling capabilities may safely be provided. Required protection can be imposed by the compilers, rather than by the hardware.

Each user may specify an action to be performed when an interrupt which he is authorized to field occurs. The interpretation of such specifications must follow the customary stack model for flow of control. That is, a dynamic specification of response in a given lexical level (or program block) should override the specification for that interrupt which was in effect at block entry. Upon exit from the block, the preceding value must be restored.

Space is provided for this purpose in the base of the stack segment for each process. (The "base" is the fixed area below the first active data in a stack segment; it is where the process information is stored. See Figure 3.3-1) Four descriptors are placed there, in a fixed position relative to the first word of the stack segment. Each of these descriptors describes a pseudo-stack segment, referred to as an interrupt response vector (IRV) stack. The four IRV stacks correspond to Arithmetic Traps, Control Traps, Asynchronous Process Oriented, and System Oriented interrupts respectively. When a process is created, the operating system initializes each of the four descriptors as the copy of a system Mom, which is located in the level 0 stack.

The formats of the IRV sections correspond directly with the interrupt categories they represent. The length of a section is the number of distinct interrupts in that category (viz., 7,8,3, and 10), and each word is a PEW for the interrupt-handling procedure for that interrupt. The IRV stacks consist of one or more groups of PEW's, each group being a section as described above. A description of how these stacks are accessed and manipulated will illustrate the detail of their structure.

When an interrupt is accepted by a processor, the interrupt category, which is part of the interrupt signal, is used to select one of the four descriptors in the base of the process's stack. The index number of the interrupt within that category is

used to select an element of the interrupt response vector adadressed by that descriptor; the PEW found there determines the code segment to be entered.

When a process executes a statement which specifies a replacement of the response for a particular interrupt, (for example, in PL/I, one might write ON ZERODIVIDE...), a system procedure is invoked which consults a four-bit group in the current Mark Stack Word (MSW). The category number of the specified interrupt is used to select one of these four bits; if that bit is a zero, the current block has not specified any overrides for interrupts in that category. In this case, two courses of action may follow, depending upon whether the process has or has not created a local IRV stack for the category. Assume, for now, that it has. The system procedure then makes a copy of the top group of the appropriate IRV stack, and places this group-copy at the top of the IRV stack. It then writes the PEW for the newly specified response over the entry for that interrupt in the new copy. Finally, it sets to one the bit for the category in the MSW, and updates the descriptor for the category in the process's stack base.

This action is illustrated in Figures 3.3-1 a) and b). The related information just before the execution of the override statement for division by zero is shown in a), and b) shows the conditions just after. In a), note that the appropriate bit in the current MSW is set to zero, and that the Arithemetic Trap category descriptor in the stack base points to the bottom of the second IRV-group, which is then the top stack group. After the override, b) shows the MSW bit set to 1, that a third IRV group now occupies the top of the IRV for this category as indicated by the descriptor in the stack base, and that the PEW for division by zero in the newly-formed top group of the IRV specifies the response directed by the override statement just executed.

If the process were now to execute an override for this category of interrupt, the response would be different, since the bit in the MSW is currently set to one, indicating that an IRV group already exists. Whether division by zero or other interrupt in the same category is overridden, the action in this case is the same: the PEW for that response is just overwritten. Hence, any previous override specified in the current block for that interrupt is not stacked, but destroyed by a subsequent override in the same block.

When a block is exited, the system procedure which administers stack cut-back examines the four interrupt-response bits in the MSW. For each which is a one, the top group is peeled from the IRV stack by updating the descriptor in the

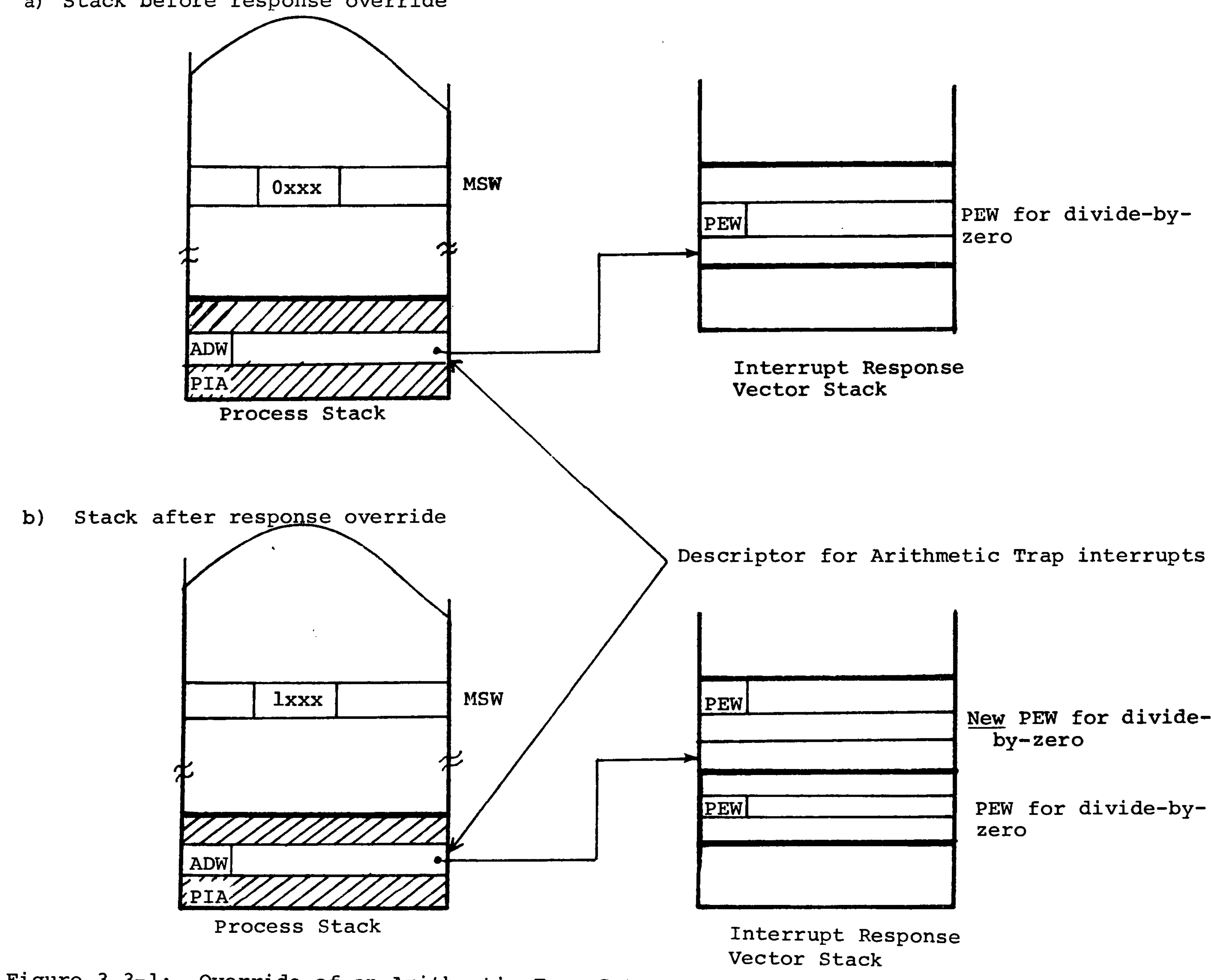


Figure 3.3-1: Override of an Arithmetic Trap Category of Interrupt Response (divide-by-zero)

stack base. Thus, the desired stacking action for interrupt response is achieved. Specification of interrupt responses in different blocks are stacked, in order of dynamic flow of control.

The subject postponed above may now be treated; namely, the action taken when a process specifies the interrupt response, but does not possess an IRV of its own. This condition arises because the operating system initializes a newly-created process with interrupt response descriptors which are copies (specifying subarrays: see below) of a single Mom descriptor located in the level 0 stack. This Mom descriptor defines a combined IRV which is shared by all processes which do not specify or have not yet specified responses to interrupts of a given category. The initial IRV situation for a process is shown in Figure 3.3-2. The response PEWs for all interrupt categories are placed into a single array as shown, since the system responses are fixed. The use of the sub-array designators in the copy descriptors allows separate access to each of the four sections of the system IRV, using only the single Mom descriptor.

The above configuration is that encountered when a process first specifies its own interrupt response. The procedure invoked to perform the function can distinguish this case by discovering that the descriptor for the specified category is a copy-descriptor, not a Mom; it then obtains space for a local IRV for the process, and copies the corresponding section from the system IRV to the local IRV. It then rewrites the descriptor in the stack base, making it into a Mom for the new segment. Further processing is similar to the case already described. The process is reversed when stack cut-back is performed, and the IRV is to be cut back but contains only a single group; the block-exit procedure must then release the space occupied by the IRV, and set the descriptor back to being a copy of the system Mom.

The amount of space required to store IRV's is determined by the size of the "entry" (that is, the length of the IRV section for that category), and the anticipated nesting depth. The categorization of interrupts corresponds roughly to relative frequency of override; Arithmetic Traps are most probable, Control Traps next, and so on. In the absence of other information, space for more entries should be provided for an Arithmetic Trap IRV than for a Control Trap IRV, etc. However, the language compiler may provide additional information which can steer the space-allocator. For example, if only one override statement appears, only one IRV entry will be needed in the absence of recursion.

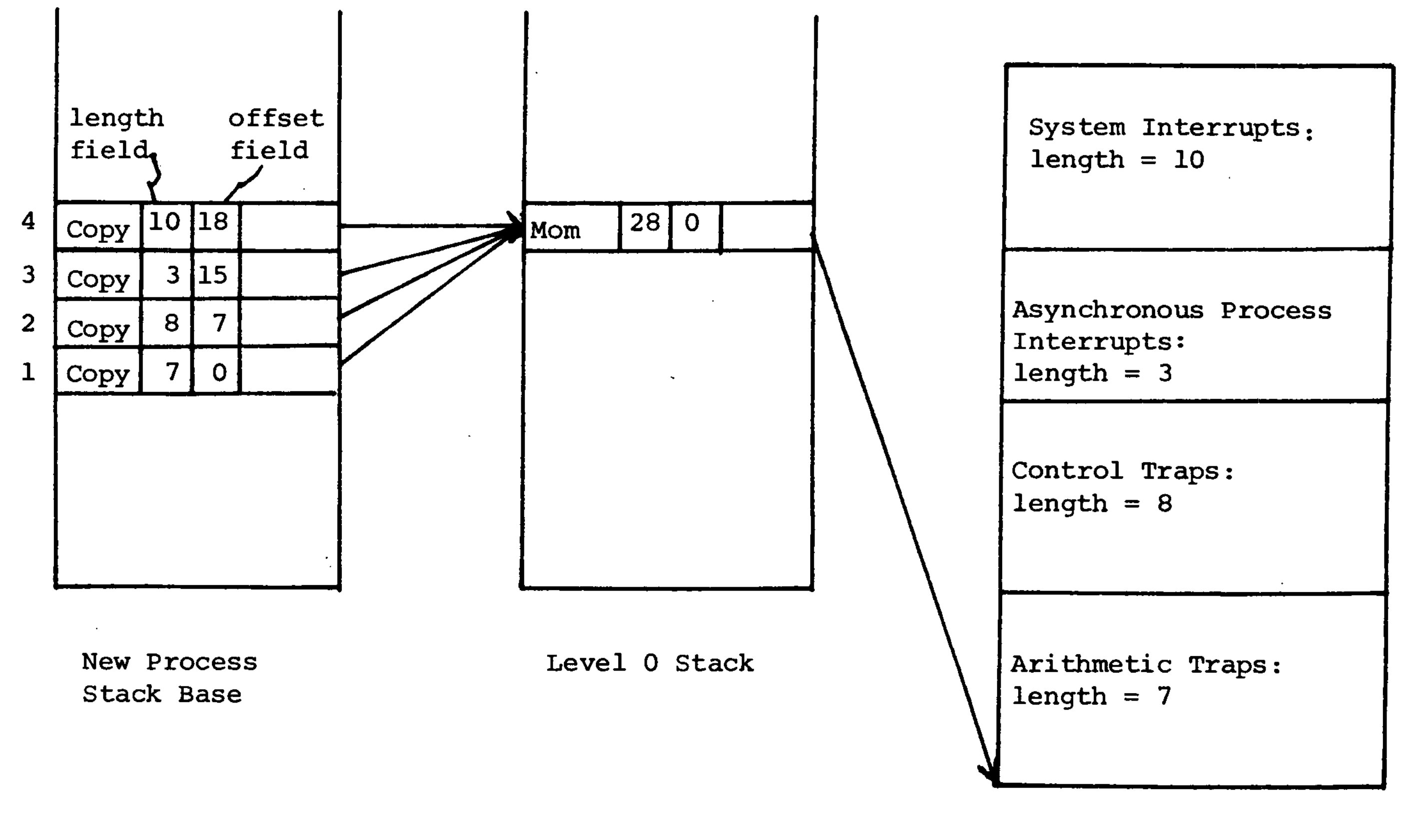


Figure 3.3-2 Initial Configuration of a New Process Relative to Interrupt Response

A final observation deals with the implementation of a function like the PL/I REVERT xxx, where xxx is an interrupt condition. This statement is used to set the response to the interrupt xxx back to the value it had upon entry to the present lexical level. In the current system, this is easily implemented: REVERT invokes a procedure which examines the category-bit in the MSW. If the bit is zero, there is no local response active for the category, and control is returned to the caller. Otherwise, the IRV is located, and the appropriate PEW in the top group is replaced by the contents of the PEW in the next-to-top group (which must be accessed from the system IRV if the process's IRV contains only one group). Similar selection from the system IRV is performed to implement a specification such as ON ZERODIVIDE SYSTEM, which directs that "system" response to the division-by-zero interrupt be invoked.

3.3.5 Interrupt Masking

The ability to interrupt a processor in the current design achieves two objectives:

a) it allows a processor to respond to conditions not anticipated by the process in execution, and

b) it provides rapid response to the occurrence of certain conditions without the overhead required for high-frequency polling to determine whether the conditions have occurred.

The first of these is necessitated by the real concurrency of operations in a multiprocessor system. For example, an input/output operation started by one process may complete when some other process is in execution. Thus, while the executing process may be totally unrelated to the I/O operation which just completed, it is often desirable to respond immediately to the occurrence of the condition, to initiate a subsequent I/O request to the device, and to determine whether the completion readies a high priority process for execution.

The second objective is exemplified by the validity-testing of operands and addresses in conjunction with execution of instructions by the processor; for example, the testing for invalid memory address or arithmetic overflow. Such tests are effectively performed in a parallel with the execution of each instruction, making it unnecessary for processes to provide checks (necessarily sequential, rather than parallel) to ensure that all conditions are met for satisfactory completion of the instruction.

At times, the interruption of a process may be undesirable. In a few cases, the desired response to a condition is standard, and may be directly implemented by the processor, avoiding the overhead of entry to and exit from an interrupt-handler procedure. The best example of this type of condition is floating point underflow, in which the customary response is simply to set the result to zero and continue. Examination of the interrupt classification list, given in section 3.2, reveals that the interrupt causes may be categorized as to whether they do or do not represent states in which the processor may continue to perform the instruction during which the condition occurred. For example, all of the arithmetic and control traps, the stack bounds violation interrupt, the absent segment trap, and the synchronous error indications are situations in which the processor must take action other than that implied by the current instruction. If the index to an array is invalid, the processor cannot select an array element; if the operation code is invalid, the processor cannot execute that instruction; if a floating point overflow occurs, the result cannot be expressed within the data format of the machine; if the stack bounds are violated, the processor must seek an alternate path of execution. In the other category are those conditions in which the response may be postponed: timer runout, I/O completion, I/O data error, timer signals, and so forth. Considering the first "cul-de-sac" operations (for which an escape route must be provided), it is seen that only two conditions, for which the normal response is standard, arise with sufficient frequency to make the specification of a standard response desirable. These are the DP and SP floating point underflow conditions. The DP underflow represents a case in which the exponent of the result of an operation is too small to be exactly represented within the exponent field. The SP floating point underflow occurs when a single precision store-operation finds that although the double precision exponent field is adequate to contain the exponent, the single precision store cannot be completed as specified. In both these cases, it is almost always desired merely to replace the result by zero, and continue normally. Hence the processor is implemented with a 2-bit mask field which may be set to determine whether occurrence of each underflow condition is to cause entry to an interrupt procedure, or conversion of the result to a zero value. At every procedure entry, the processor stores the values of these two mask-bits as part of the status of the calling process, and restores the mask setting from these bits when the return is executed.

The "cul-de-sac" interrupts described above represent conditions which could logically be anticipated in terms of the execution of particular instructions (e.g., one knows that a multiply operation may overflow), but for which validity checking is more effectively provided by concurrent operations in

the hardware than by sequential operations in the code itself. The asynchronous conditions such as timer runout, I/O completion, etc., are philosophically different since they cannot, in general, be anticipated at a particular point either in the context of a given instruction, or even by the process being executed. Responding to these conditions via an interrupt represents a change in context from the process which was interrupted to that of the interrupt-handler, since the interrupt handling is logically unrelated to the process which was interrupted. Because of the absence of a logical imperative, delays in response to such conditions can usually be tolerated.

There are situations in which one or both of the above objectives are not appropriate: for example, the interruption of a process to respond to an unanticipated condition may disrupt a computation in a "critical section". Under other conditions, the overhead involved in interrupt handling may actually be higher than the polling overhead that would be necessary if the condition were not to cause an interrupt. For these reasons, additional control of interruptibility is provided in the system.

The manipulation of certain data bases must be restricted to one process at a time. For example, the dispatching of queues of ready and running processes must prevent such occurrences as the simultaneous allocation of two processors to a single process. In a multiprocessing environment, two conditions must be fulfilled to handle such data bases safely and efficiently:

a) Once a process has gained access to such a data base, other processes must be denied access to that data base;

b) The process must be allowed to perform its manipulations without its processor being pre-empted for some other purpose.

Explicit data-locking techniques may be used to enforce an upper bound to the number of processes granted access. However, to dynamically impose a lower bound, it is necessary to prevent the processor from being assigned to an interrupt handler. This is achieved by providing an interrupt-inhibiting capability to a process. This is a signal transmitted to the I/O controller via the IPC bus, which specifies that the issuing processor has become uninterruptible for those interrupts for which the processor is selected by the I/O controller. As noted above, these interrupts are ones whose response may be postponed without disturbing the logical correctness of the ensuing operations. Therefore, even if all processors should be in the inhibited state for a short time, the conditions may be adequately handled later.

In addition to making the processor uninterruptible, the I/O controller initializes an inhibit-state interval timer for that processor, with a limit value. Should no subsequent interrupt-enable command from that processor be received before runout occurs, the I/O controller will signal the processor with the Apparent Software Fault interrupt. This mechanism insures that no process remains in the interrupt-inhibited state excessively, to limit the effect on interrupt response. An appropriate value for this time limit appears to be about one millisecond.

The use of a one millisecond time limit on inhibited interrupts precludes the handling of Absent Segment Traps triggered by processes having interrupts inhibited. Consequently, if an absent segment trap should occur when the processor is in the interrupt inhibited state, the Apparent Software Fault interrupt will be triggered rather than the Absent Segment Trap.

One further form of interrupt inhibiting is mentioned to complete the treatment of this subject. When the multiprocessor system workload is heavy, the frequency of Absent Segment Traps can be expected to be relatively high. Conventional processing of an Absent Segment Trap requires entry to an interrupt handler, initiation of an M3 operation, and placement of the process into the wait state. Upon completion of the M3 operation, an I/O Completion interrupt is signalled. The handler for this interrupt is then entered, the process waiting for the segment is readied, another I/O operation to M3 is initiated if one is queued, and the processor allocation routine is called to see if it is appropriate to assign a processor to the newly readied process. An alternate implementation has been chosen to avoid, at least in most cases, the necessity for entering the I/O Completion interrupt handler when the segment transfer is concluded. The details of this mechanization are described in conjunction with I/O control in section 3.5; however, it is appropriate to point out here that this is achieved by providing a capability in the I/O controller which causes it to make a choice of whether or not to signal I/O Completion. Thus a dynamic decision is made as to whether the interrupt should be suppressed or signalled, depending upon the existence of a queue of operations waiting for the device. If the interrupt is suppressed, the condition is made known to the system by the setting of a bit field in a location accessible to the absent segment trap handler. After initiating an M3 operation required to make an absent segment present, this handler checks the completion-states of M3 segment transfers previously issued. The processes whose segments are found to have completed their transfers are readied; thus the need to enter the I/O Completion Interrupt handler is avoided. This diminishes the overhead for absent segment handling, especially under heavy load, when computational overhead is most detrimental to the system throughput.

3.4 Memory Management

3.4.1 Introduction

It is assumed that all software written for the system will be processed by compilers to produce code and data segments in accordance with specifications determined by the memory management procedures. Each compiler thus produces and stores in M3:

a) Code segments, which normally correspond to procedure blocks in the source program, and include additional code, supplied by the compiler, for establishing and initializing the stack link from which the segment executes.

b) Initialized-data-array segments contain the initialization values to be applied to a given array when that array is "created" and when the procedure in which the array was declared is entered during execution. Uninitialized data-arrays are created at run-time by the absent-segment interrupt handler (see section 3.4.7.1),and initialized and uninitialized scalars are allocated and set by the stack-link initializing code mentioned previously.

c) The segment dictionary, which contains a descriptor for each code segment forming a part of the program to be executed, whether it is internal, i.e., a compiled part of the program, or an external segment to be dynamically bound at execution time.

All references to segments occur via an access path that includes the Mom descriptor for the segment. This special descriptor, of which there is only one for each segment, indicates whether the referenced segment is currently to be found in M2 memory or not. When the segment is present in M2, the Mom descriptor contains its M2 address; if it is not in M2, the Mom contains its M3 address (several exceptions to this are described below). When a reference to a segment occurs, the processor checks the present/absent state by examining the "presence" bit in the Mom. If the segment is present, the access is completed; however, if it is absent, the processor signals the Absent Segment Trap, and enters the system procedure which handles this condition. The absent-segment-trap handler makes the referenced segment present, after finding or creating space for it in M2, by reading the segment from M3, if it resides there, or making it up, under certain circumstances to be described later. If adequate M2 space is available, an area of

appropriate size is obtained from a free area; otherwise, the distinction between physical and virtual storage is exploited by pre-empting space occupied by some one or more other segments to make room for the one currently demanded. Thus, a process may address logically more procedure and data segments than may conveniently be stored simultaneously in M2. The observed patterns of addressing of typical programs, namely, the temporal and spatial locality which they exhibit, allows many segments to be absent without seriously impacting the throughput of a processor, relative to its capability given much larger M2.

Spatial locality [1] refers to the concept that if a procedure accesses a logical location L at time T, then it is likely to refer to an address in the range L - dL to L + dL at the next opportunity. On the other hand, temporal locality is the property that if the set {Li} of logical addresses are referred to in the interval T - dT to T, there is a high likelihood that addresses from this set will occur in the interval T to T + dT. Programs are found, on the average, to have an exploitable degree of both types of locality; hence, the operating system attempts to maintain in M2 those segments which have recently been referred to. Because the segments which have not recently been referred to typically will not soon be referred to again, it is possible to achieve almost full utilization of the processor even when M2 space for only a fraction of the segments is made available to a process. Hence, the overall size of M2 may be smaller (or alternatively, the number of processes may be made larger) compared with a system in which M2 was not dynamically multiplexed.

The description of how M2 multiplexing is implemented forms the remainder of this section.

3.4.2 Administration of M2 Space

At any instant of time, M2 is logically divided into a number of areas, which are of three types: available, in-use, and unusable. Available areas or parts are those not currently occupied by a segment; unusable areas are similarly unoccupied, but are not eligible for use because of a fault or other reason. In-use areas are classified as shown in Figure 3.4-1. A discussion of some of the operations which are provided to deal with memory parts will provide motivation for the form selected for administrative data associated with memory parts.

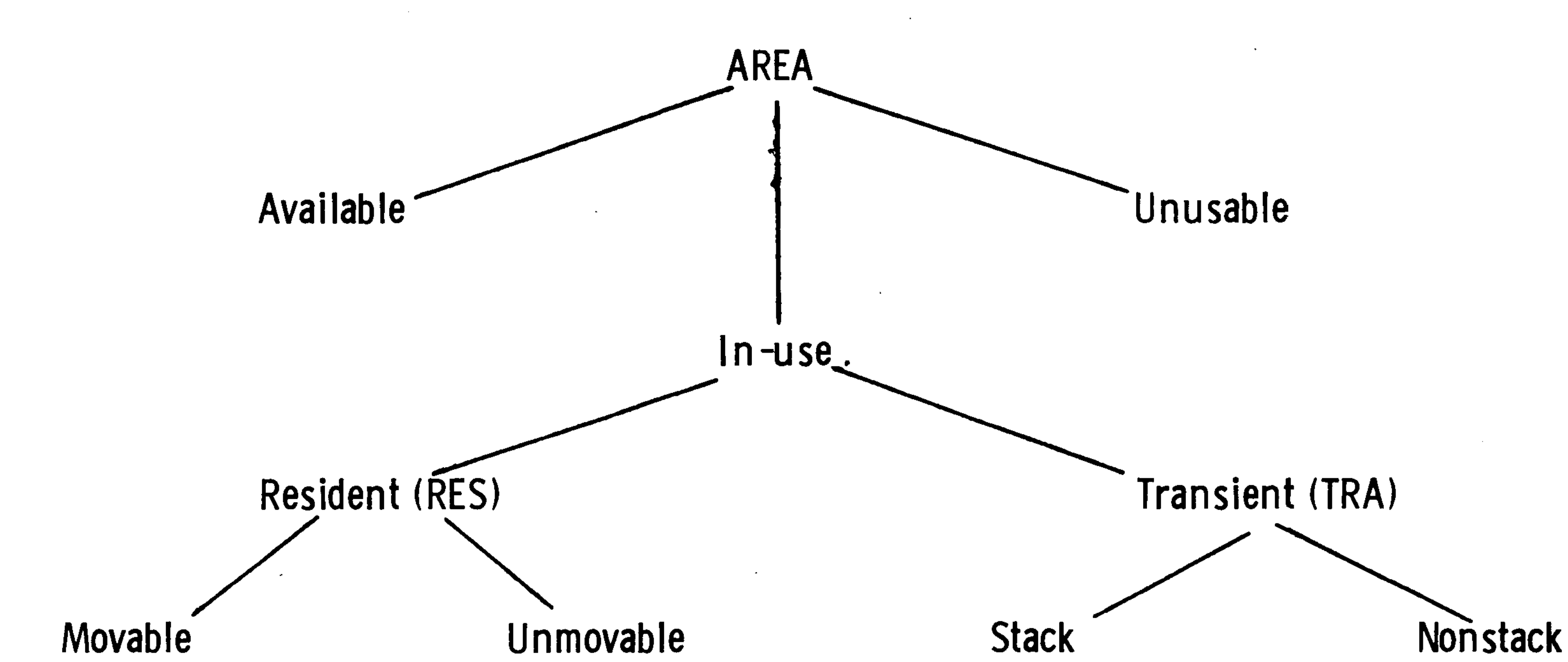


Figure 3.4-1 Classification of M3 Memory Areas

3.4.2.1 Space Allocation

Space allocated to a segment may be treated as "resident" or "transient". That space which is transient is subject to dynamic displacement from M2 in connection with the practice of memory multiplexing; resident space is not. Because the area available for multiplexing is reduced when resident space is increased, the use of resident space is to be minimized. Normally, it is used:

a) To improve real-time response, since the time penalty incurred from absent segment traps can be avoided;

b) To improve the performance of the operating system, by avoiding the delay and overhead of segment loading for heavily used functions.

c) To avoid logical difficulties such as might arise if, say, a segment of the Absent Segment Trap Handler caused an absent segment trap.

d) To insure that the key (top) stack sections of ready processes are present in M2, so that the processor allocation procedure may assign a processor to them without delay.

The operating system handles all requests from processes for M2 space. Such requests most often arise from absent segment traps, but other types of space-requests are defined. The response of the OS to space requests is a compromise between maximum utilization of M2 and minimum computational overhead. Among the phenomena which can diminish the effectiveness of a storage-management algorithm is fragmentation. In a scheme employing fixed-size pages, the fragmentation is "internal"[2]; it results from the fact that the unit of space allocated for a given request is likely to be bigger than logically necessary. For example, if a 1024-word page is fetched because a 100-word array is needed, the extra words may not be used by the computation before the page is displaced in response to a demand for some other page. Fragmentation makes demands for other pages more frequent, since the memory consumed by non-required words reduces the remaining available space.

External fragmentation occurs only when multiplexing with variable-sized areas. Unlike internal fragmentation, where the lost or unnecessary area is <u>within</u> the allocated memory part, the unused areas with external fragmentation occur <u>between</u> in-use memory parts, whenever variable-sized areas are obtained or released. For example, suppose an algorithm is used which allocates space in a 10-word memory from the smallest area big enough, and consider the sequence $R_1 = 5$,

$R_2 = 1$, F_1, $R_3 = 2$, $R_4 = 3$ ($R_1 = 5$ denotes "request number one is for 5 words"; F_1 means "free the area obtained by R_1"). The availability of the memory (initially all free) after each request is given by

$\underline{5}$, 5

$\underline{5}$, $\underline{1}$, 4

5, $\underline{1}$, 4

5, $\underline{1}$, $\underline{2}$, 2

$\underline{3}$, 2, $\underline{1}$, $\underline{2}$, 2

where the underlined digits represent in-use blocks, and the non-underlined digits refer to free areas. The two 2-word free areas represent fragmentation. A request $R_5 = 3$ or 4 would not be satisfiable, since the four words of available space are fragmented.

The above example, showing the ill effects of fragmentation, may also be used to demonstrate the behavior of an alternative space-selection strategy. The above algorithm is called "best fit" [3], since a request is satisfied from the area whose size is both sufficiently large and most nearly equal to the size specified in the request. Another algorithm is termed "first fit", since the first area whose size is adequate is chosen. For the sequence of requests described above, the memory map resulting from the "first fit" strategy would be

$\underline{5}$, 5

$\underline{5}$, $\underline{1}$, 4

5, $\underline{1}$, 4

$\underline{2}$, 3, $\underline{1}$, 4

$\underline{2}$, $\underline{3}$, $\underline{1}$, 4

This case has the property that all available space is left in a single area; a request for three or four words would be granted at this point, even though with "best fit", failure would occur.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Admittedly, cases may be constructed in which "first fit" fails and "best fit" succeeds (e.g., $R_1 = 5$, $R_2 = 2$, F_1, $R_3 = 3$ gives

First Fit	Best Fit
$\underline{5}$, 5	$\underline{5}$, 5
$\underline{5}$, $\underline{2}$, 3	$\underline{5}$, $\underline{2}$, 3
5, $\underline{2}$, 3	5, $\underline{2}$, 3
$\underline{3}$, 2, $\underline{2}$, 3	5, $\underline{2}$, $\underline{3}$

in which requests for four or five words can be satisfied by "best fit" but not "first fit"). Still, "best fit" has two distinct disadvantages, although it initially might seem more appealing: First, it requires examination of more storage areas, on the average. Only when an exact fit is found may the search be stopped before all available blocks have been examined*, with "first fit", the search terminates as soon as any adequate area is found. The second disadvantage is that a by-product of "best fit" is the creation of a number of very small blocks, since the difference between the sizes of the chosen and requested blocks deliberately is minimized. Inasmuch as this increases the total number of available areas, the search time for "best fit", already longer, may be increased further. To reduce the deleterious effect on the "best fit" search, it is beneficial to maintain a list of blocks ordered by size; the search then should be conducted from the largest to smallest, stopping when the first block not large enough is found; the previous block is then used. If the search is conducted from the small end of the size range, the added small blocks are likely to be often searched but not accepted, thus increasing the time required to find a solution.

We have selected the first-fit algorithm for these reasons. While this represents a design decision, it does not imply a commitment; a laboratory model or a simulation may be used as a test-bed to explore the alternate choices.

A second issue relative to storage fragmentation is the differentiation of requests for resident space from those for transient space. While the distinction between certain operations on these types is strict, it is nevertheless true that the distinction between their retention times in M2 is more vague. This is simply because some resident segments may soon be released, and some transient segments may stay in M2 for relatively long intervals without being displaced.

---

* If available blocks are ordered by size, not all blocks need be examined; however, the overhead of maintaining the order is added.

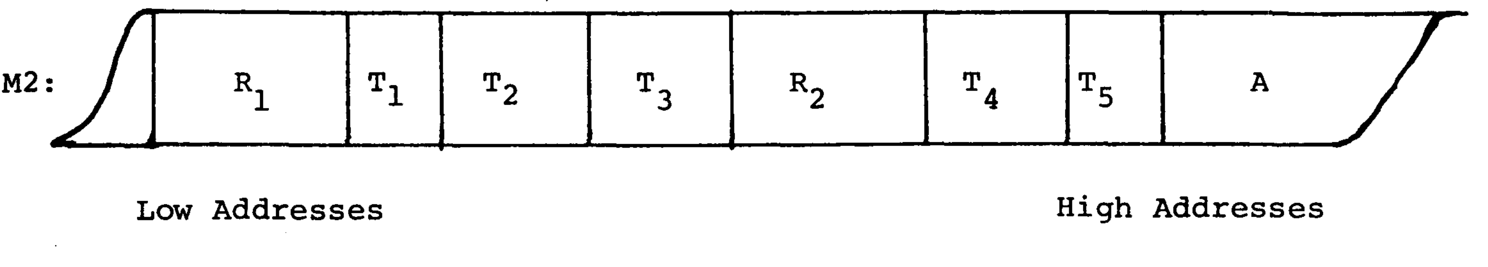
In the absence of empirical results, however, a design has been selected which presumes that at least on the average, the stay-times of resident segments substantially exceed that of transient segments. Consequently, it is advisable to attempt a partitioning of memory in such a way that fragmenting of the transient pool of space does not result from the scattered presence of long-term resident segments.

We treat requests for resident segments as though they had higher priority, in some sense, than transient requests, because requests for resident space may take space from transient blocks, but not vice versa. As a result, it is inappropriate to place a fixed partition between the two memory regions. A softer "boundary" is imposed instead, by adopting a strategy under which requests for resident space are more likely to be satisfied from lower M2 addresses than are transient space requests. This strategy, which resembles that of the B6700 operating system (MCP) [4], involves a list organization and a search procedure; these are now described.

3.4.2.2 Structure of Available-Memory Lists

Two lists of available memory parts are maintained: a resident (RES) list and the transient (TRA) list. When a request for space is received, the space allocator attempts to satisfy the request from the list which corresponds to the type of area requested. To reduce the chances of increasing memory-fragmentation when a request for RES space is granted, available-memory areas are added to the RES list only when they are bounded at their low end by an equally-fragmenting block: an in-use RES block, or an unusable block. For example, in Figure 3.4-2 if the in-use TRA area T_2 were to become available it would be linked into the TRA list, and not into the RES list. This is because if it were linked into the RES list, and the space then was assigned as an in-use RES area, it would create a new boundary between a TRA and RES area, and increase the degree of fragmentation. It would become a new RES-island in a sea of TRA areas. As another example, if in-use TRA area T_3 were to be freed, it would not be linked into the RES list, even though it is adjacent to the in-use RES area R_2. Although this would encourage no new TRA/RES boundaries, it would decrease the affinity between RES areas and the low address end of memory. Finally, when T_4 became free, the space would become linked into both the RES and TRA lists, since it is bounded at its lower end by an in-use RES block.

Available-memory blocks are included in the TRA list unless both upper and lower neighboring blocks are in-use of type RES or unusable. It is readily seen that available blocks may be members of both lists at once.



Key:

R In-use resident
T In-use transient
A Available

Figure 3.4-2 Partitioning of M2

When an in-use block is released (freed), two functions must be performed. First, if either or both adjacent blocks are already free (available), the newly freed block is coalesced with them. Second, the resulting free area is placed on one or both lists, depending upon its neighbors. If the best-fit policy was used in space allocation, the free area would be placed in the list in accordance with its size. Under first-fit, no such search for the correct location is logically necessary; the block may be placed at the head or tail of the list. However, to press RES areas towards low M2 addresses, the available-RES-list is ordered by M2 location. Consequently, when a block is added to the RES-list, a search may be required For the TRA-list, the use of first-fit eliminates not one, but two searches (one at space allocation time, and one when that space is released) relative to the best-fit approach.

The connections of an available memory area to the memory lists are accomplished by use of space within the block itself. Whether a block is available or in use, an indication of type and other characteristics are stored in the block header area, described below. To determine the nature of a block, the header may be examined. This examination is required, say, when an adjacent block is freed, to determine whether the block is available or in use, and if in use, whether RES or TRA. Locating the header of the successor block is straightforward when the location and size of the freed block are known. However, while the last word of the predecessor block is easily located, finding the header area is less easy, since the size of the block is variable. Thus it is desirable to store information at the end of each block as well as at its beginning. Various choices are available: for example, necessary characteristics may be stored at both ends, or the last word may contain the size of the block or a pointer to its first word. Exactly which of these forms is selected is relatively unimportant; the primary issue is whether such information is provided at all. The conflicting considerations to be traded off are the lost space in the block if the information is kept, versus the execution time penalty searching the lists to identify the predecessor blocks. Consistent with previous design choices which eliminated or minimized searches, we presume that loss of space is preferable to execution overhead. In the sequel, examination of predecessor areas will be discussed; it is postulated that this is done without the necessity of a search.

A similar line of reasoning leads to the use of both forward and backward linking for the lists of available blocks. This costs nothing in space, since the pointers are needed only when the block is not in use. The cost is the maintenance of two pointers, rather than one, for each list-entry. The saving occurs when a block is removed from the list, and the list must

be linked around it. Had the block been selected during a simple search for first-fit, the predecessor-block in the search could be remembered, obviating the need for a back pointer. However, blocks may be selected in other ways, described below. In these instances, the predecessor block has not necessarily been examined, and a search would be required to find it if the back pointer were not provided.

Figure 3.4-3 summarizes the administrative fields which are maintained in available-memory blocks, without regard to the number of words or bits which are required. Estimates may be based on M2 memory size (the pointers are physical M2 addresses); about twenty bits would suffice for size and pointer fields.

3.4.5 Space Administration Procedures

In this section, the procedures for obtaining and releasing areas of M2 storage are described.

3.4.5.1 RELEASESPACE Procedure: The memory management software contains a procedure which is called whenever a process wishes to free (release) a memory area. The only parameter which is needed by this procedure is the location of the beginning of the block. The nature of the manipulation which the releasespace procedure performs dictates that no more than one process at a time may execute the procedure; a lock is therefore provided.

The RELEASESPACE procedure examines the physically preceding and succeeding blocks. For each, it is determined whether the block is available; if so, it is delinked from the RES and/or TRA lists, the header area of the lower of the freed block and neighbor being examined is adjusted to reflect the combined size, and the trailer area is updated in the higher of the two. When both neighbors have been treated, there will be a single free block, a member of neither the TRA nor RES lists. The neighboring blocks of this block are examined to determine which of the lists should contain it. This decision is direct. Neighboring blocks must be 1) unusable, 2) in-use RES, or 3) in-use TRA. If the predecessor block is in-use TRA, the free block is placed only upon the TRA list. Otherwise, the successor block must be consulted: If the successor is in-use TRA, the free block is placed onto both available-lists; otherwise, it is added only to the RES list. This logic is illustrated in Figure 3.4-4 a), b) and c).

Lowest address	SIZE	
	TA	Header area
	FPR	
	BPR	
	FPT	
	BPT	
	• • •	Data or Program Area
Highest address	HL	Trailer area

TA represents type and attributes
FPx is a forward pointer
BPx is a backward pointer
xxR is a pointer for the RES (resident) list
xxT is a pointer for the TRA (transient) list
HL is a header locator field (size or pointer)

Figure 3.4—3 Logical Fields Utilized in Available Memory Parts

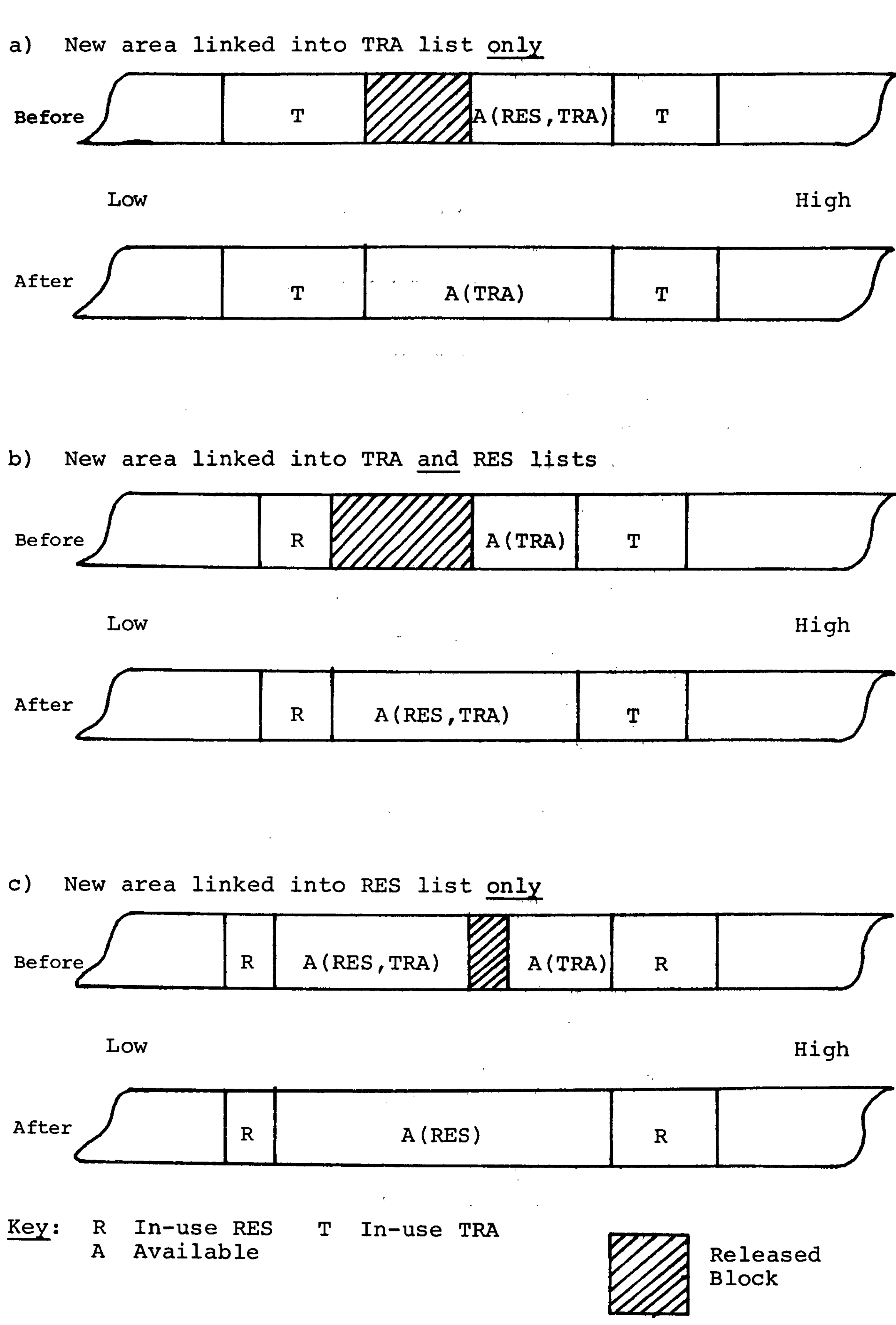


Figure 3.4-4 Release of M2 Blocks

When a block is added to the RES list, it must be placed into the list at a position corresponding to its M2 address. This may require a search. However, the search is not required if either neighbor was available and a member of the RES-list. When this situation exists, the pointer fields from the neighbor block are saved before the neighbor is de-linked, and are thus available to locate the nearest members of the list. (The fact that some of the delinking and re-linking implied by the description may be avoided in the implementation is obvious; such "efficiencies" would complicate the description if included, without adding to the clarity of the ideas. These details are thus omitted.) Should <u>both</u> neighbors have been available in the RES-list, the back pointer of the low or left neighbor and the forward pointer of the high or right neighbor are the two useful ones.

When neither neighbor provides usable pointers, it is necessary to search. Two alternatives occur as search strategies:

a) Physically adjacent blocks may be examined successively in one direction, until the first one which belongs to the RES-list is found, or

b) the list may be searched by following the pointer linkages.

The latter approach lends itself better to the use of build-in search instructions in the processor, and is therefore the one chosen. The search is stopped as soon as a block is found whose address is on one side of the new entry, and whose pointer contains null, or an address located on the other side of the new entry. Linking the new block into the list involves the ordinary pointer adjustments for doubly-linked lists: the forward pointer of the left neighbor, the back pointer of the right neighbor, and the two pointers of the new member are set.

A block to be added to the TRA-list is added at the tail-end of the list, since it is not ordered. This operation is facilitated by a pair of pointers for the list, which are kept in a known location. One of the pair points to the first entry in the TRA-list, while the other points to the last entry. To add an entry at the tail of the list, it is necessary to alter four locations: The new block's forward and backward pointers, the forward pointer of the previous last block, and finally, the second pointer of the list control pair. The new links are established in a sequence as illustrated in Figure 3. 4.5. The new pointer settings are shown as dotted lines and numbered in the order that the changes are made.

Further details of the RELEASESPACE procedure are described in subsequent sections.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

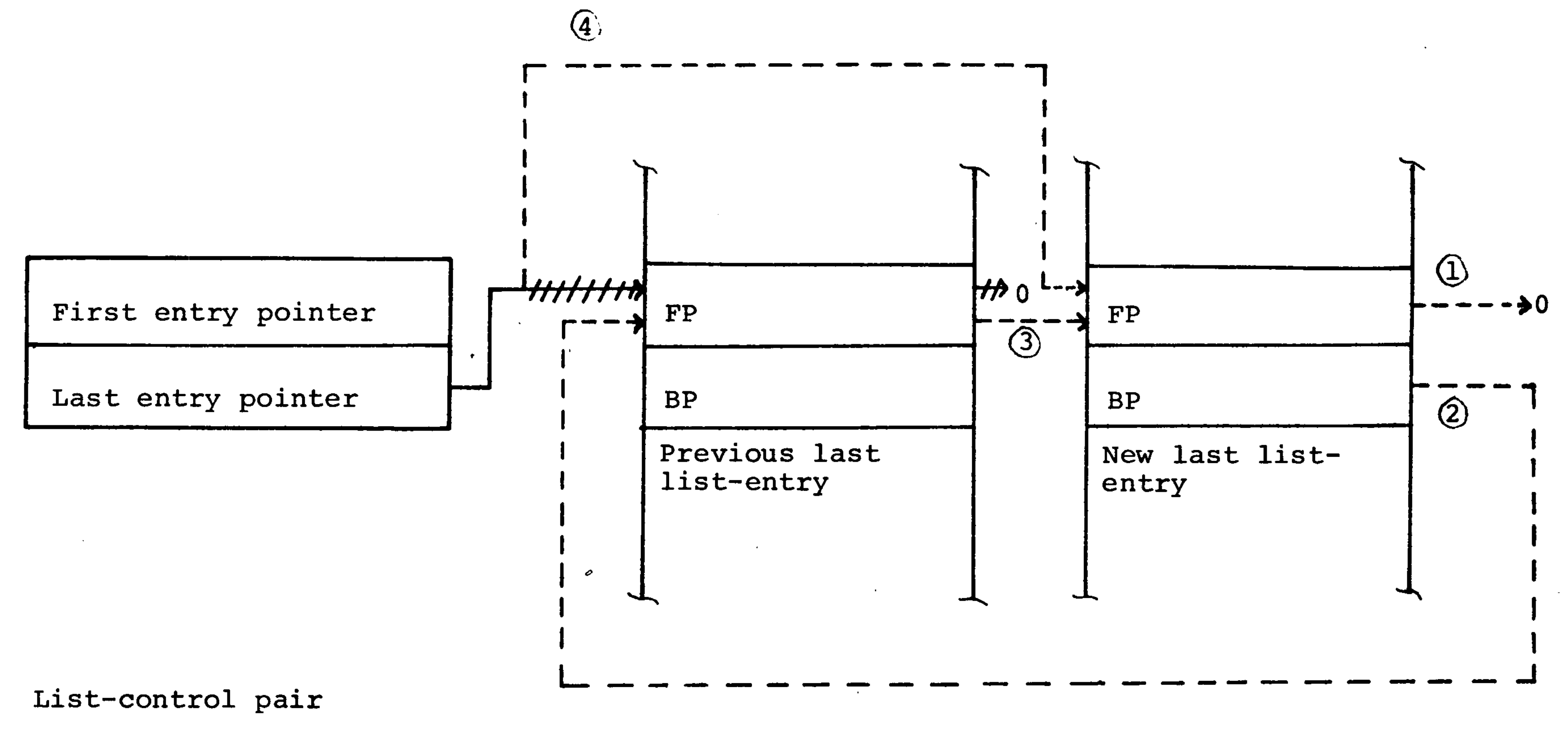


Figure 3.4-5 Linkage of a Free Block onto the Tail of the Available-Memory TRA-list. Original Pointer Values are Shown as Solid Lines. New Settings are Dotted.

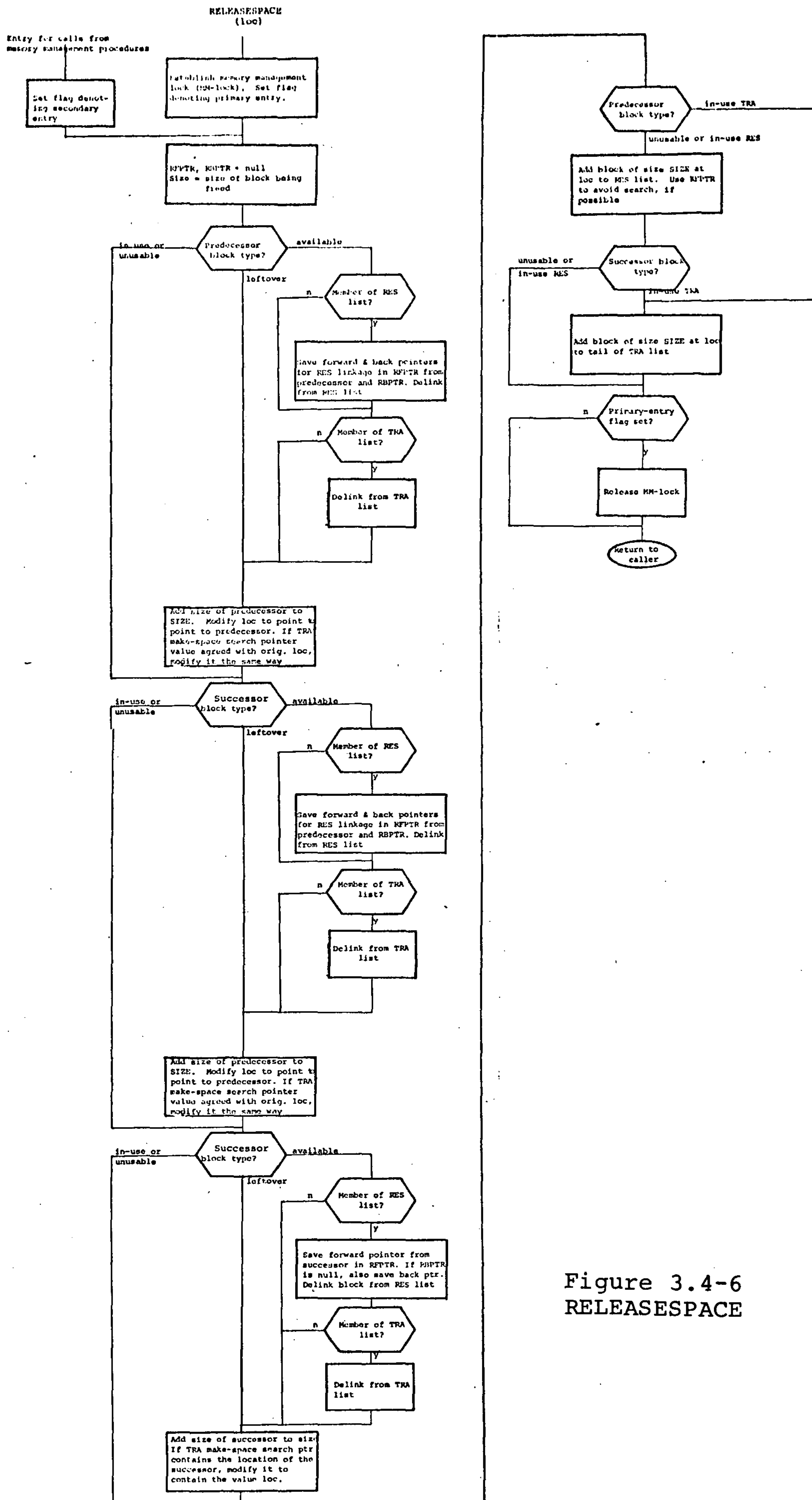


Figure 3.4-6
RELEASESPACE

3.4.5.2 FINDFREESPACE Procedure: This space allocation procedure is called by other procedures within Memory Management. The parameters associated with the call include the size of the requested area, and whether it is to be RES or TRA space. The function of the procedure is to attempt to fulfill the request from an available area, returning an indication of success with the M2-address of the block allocated, or failure. As dictated by the first-fit strategy, the appropriate list of available memory blocks is scanned. Should a block be found whose size is greater than or equal to the requested size, the search is successful. If the sizes are equal, the block is delinked from the list, and also from the other list if it is a member of both. If the sizes differ by more than a set amount, the required space is taken from the lower or higher end of the block, according to whether the request is for RES or TRA space respectively. The remaining area is linked into the lists the original block was in, which requires no searching since the original block pointers have correct values and are even in the right places if the request was for TRA space. In both cases, the trailer cell must be written with correct size information.

The situation where the size difference between the request and selected available block is small (less than the "set amount" mentioned above) is given special treatment. The threshold for this difference has not been chosen; it is readily altered when the system is in operation, and its tuning may therefore be postponed until then. A lower bound on the size difference between requested and available blocks value is determined by the combined size of the header and trailer cells, since a block must be at least this large to contain the administrative data. At the high end, it must not exceed a value above which the loss of the block is more harmful to system throughput than the overhead expense of reclaiming it. This level is related to the median request size; a size of a quarter of the median is suggested as a reasonable first approximation.

When the size difference is between zero and the threshold value, the breakage block is, by implication, too small to justify being treated normally. Accordingly, the original block is delinked from the available-memory lists as though an exact match had been found. The requested space is taken from the high or low end as above; the first and last words of the remainder are set to indicate that the block is a "leftover". The size is written as usual in the size cell, and the trailer cell is set normally. As a result the leftover available block is left stranded in memory: i.e., not linked into either available list. While in this condition, it is incapable of being selected for use; it must first be absorbed into an adjacent block which is freed.

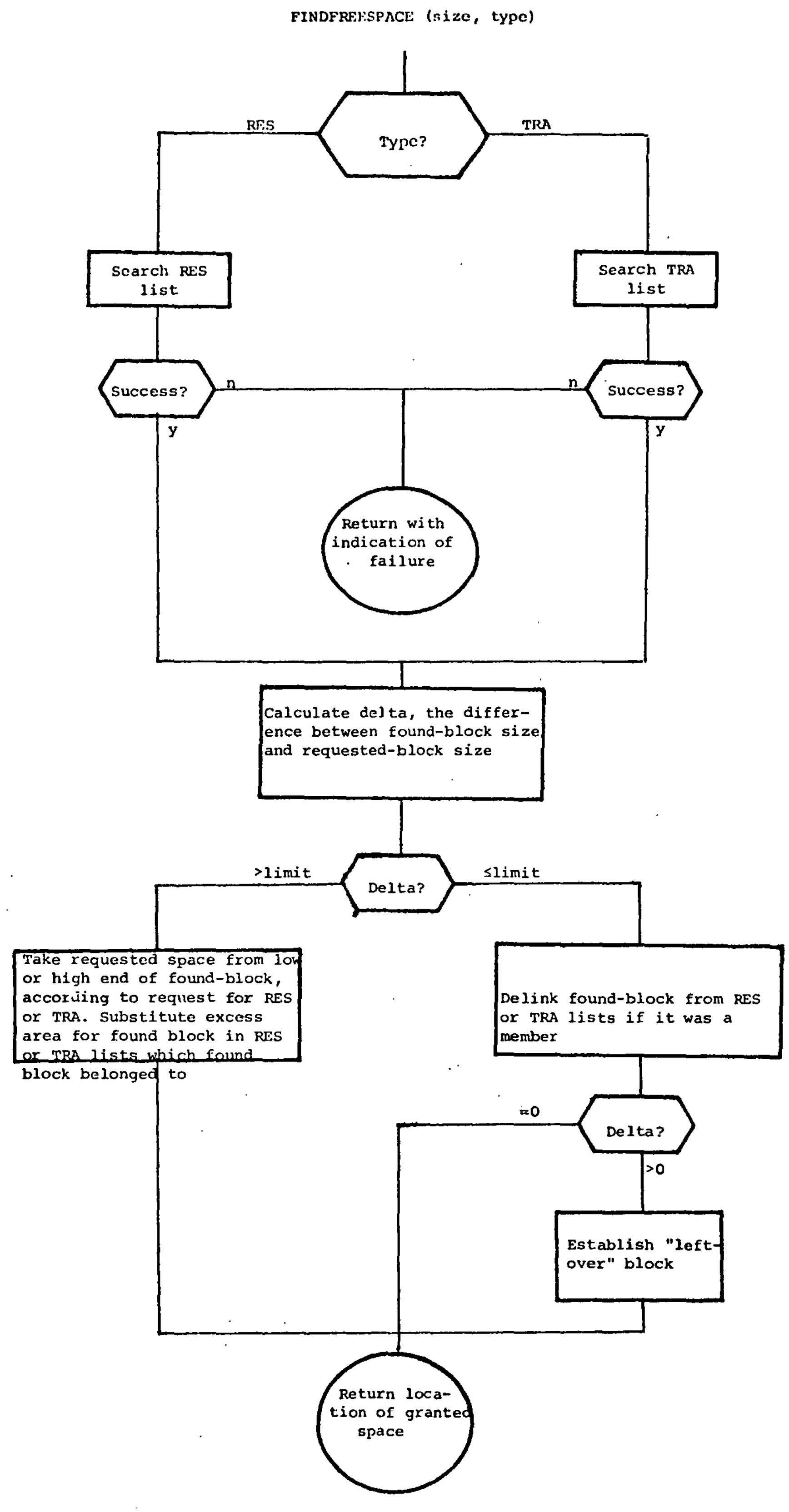


Figure 3.4-7: FINDFREESPACE

The RELEASESPACE procedure described previously is therefore modified so that when a neighbor of a released block is found to be a leftover, it is coalesced as though it were a normal available-block (except that no delinking is necessary).

3.4.5.3 MAKESPACE Procedure. If a block of sufficient size is found to be available, then the free-space allocation procedure returns its location. However, such a block may not be found, so it becomes necessary to make space.

Conceptually, space could be "created" in M2 in two ways: segments could be moved about in M2 so that fragmented available space is coalesced, or segments could be removed from M2, making their area available. MAKESPACE uses a combination of both approaches. By examining an indicator of the recency-of-use of candidate segments, a group of one or more areas is identified, whose total size is sufficient to satisfy the request. These segments may include a combination of in-use areas and available areas; the recency-of-use indication helps to select segments which have not been referred to recently by a process. In accordance with common practice [5], a block not referred to lately is assumed to be one not likely to be referred to soon, and thus its absence from M2 is likely not to be noticed for a reasonable period of time.

As described elsewhere, each segment in M2 has one and only one Mom descriptor. The Mom is the only descriptor which contains the M2 address of its segment; other descriptors access the segment via the Mom rather than directly. Because there is exactly one Mom per segment, and because all accesses are via the Mom, the processor can be caused to set a designated bit (the REF-bit described in section 2.4.2) in the Mom to a one whenever an access to the segment takes place. Additionally, the C bit (see Fig. 2.4-11) is set to one when an access occurs for the purpose of altering the segment.

The algorithm selected for segment displacement is based on one used by Multics [6], extended to handle variable-sized segments. In Multics, which uses fixed-size pages, a circular list is maintained, with an entry per main-memory page-frame. When space is needed, the list is consulted, starting at the location after the one examined last. If the usage bit is off, the page has not been referenced since the last time the space-finder examined the entry, and that page-frame is selected. If the usage bit is one, it is turned off, and the next entry is examined.

Actually, the algorithm is a special case of a more general one; the usage bit is used in conjunction with a k-bit

usage word. When a page frame is examined, the usage word is right-shifted one place, and the present value of the usage bit is placed into the vacated high-order position. Only when the word is zero is the frame selected. The case described above corresponds to k = 1; use of k = 0 makes the algorithm FIFO (first-in, first-out), and behavior with large k resembles least-recently-used (LRU) page selection. Because the search must examine a number of pages proportional to k, on the average, to find a solution, larger values of k mean larger overhead expense. Only if this overhead is more than offset by improved performance resulting from more judicious space assignment is a choice of large k appropriate. Multics has determined that k=1 is their best choice.

The corresponding problem for variable-size multiplexing (VSM) is more difficult, since it is not true that a given area is as useful as any other area; multiple-areas are frequently necessary to satisfy space requests. The same principle, however, may be applied. The concept of a k-bit shift register is equally valid in the VSM environment, and choosing the best value of k would be a valuable exercise when the system became operational. However, in subsequent discussion it is assumed that k = 1 had been chosen, and the shift-register concept is not referred to further.

Following an unsuccessful search of available areas, physically-adjacent blocks of storage are sequentially examined. When RES space is required, the search begins at the low end of M2, and proceeds toward high memory. When TRA space is needed, the search is from high to low memory, but begins at the point where the last TRA-search left off. Hence, the search normally begins somewhere in the middle, proceeds to the low end, jumps to the high end, and progresses back toward the middle. The searches are seen to be somewhat unsymmetrical; the motivation, as before, is to keep RES blocks concentrated at low addresses, to reduce fragmentation. Further it is desired that TRA-block lifetimes be independent of their location. In neither search is any possible memory area omitted from consideration, so that if a search for one type fails, no benefit can be gained by attempting a search for the other type before taking recovery action.

Each search progresses in "inchworm" fashion. When it begins a new iteration, two pointers are set to the end of the first block examined. If that block is "acceptable" (a term which will be defined below), the "front" pointer is advanced to the physically-adjacent block. If that one is also acceptable, the front pointer is advanced again. Each time a block is acceptable, the combined size of accepted blocks is compared with the size required, and the search is stopped when a

sufficient area has been accepted. When an "unacceptable" block is encountered, the front pointer is advanced to the next block, the rear pointer jumps forward and is also set to this block, inch-worm style, and a new iteration is begun.

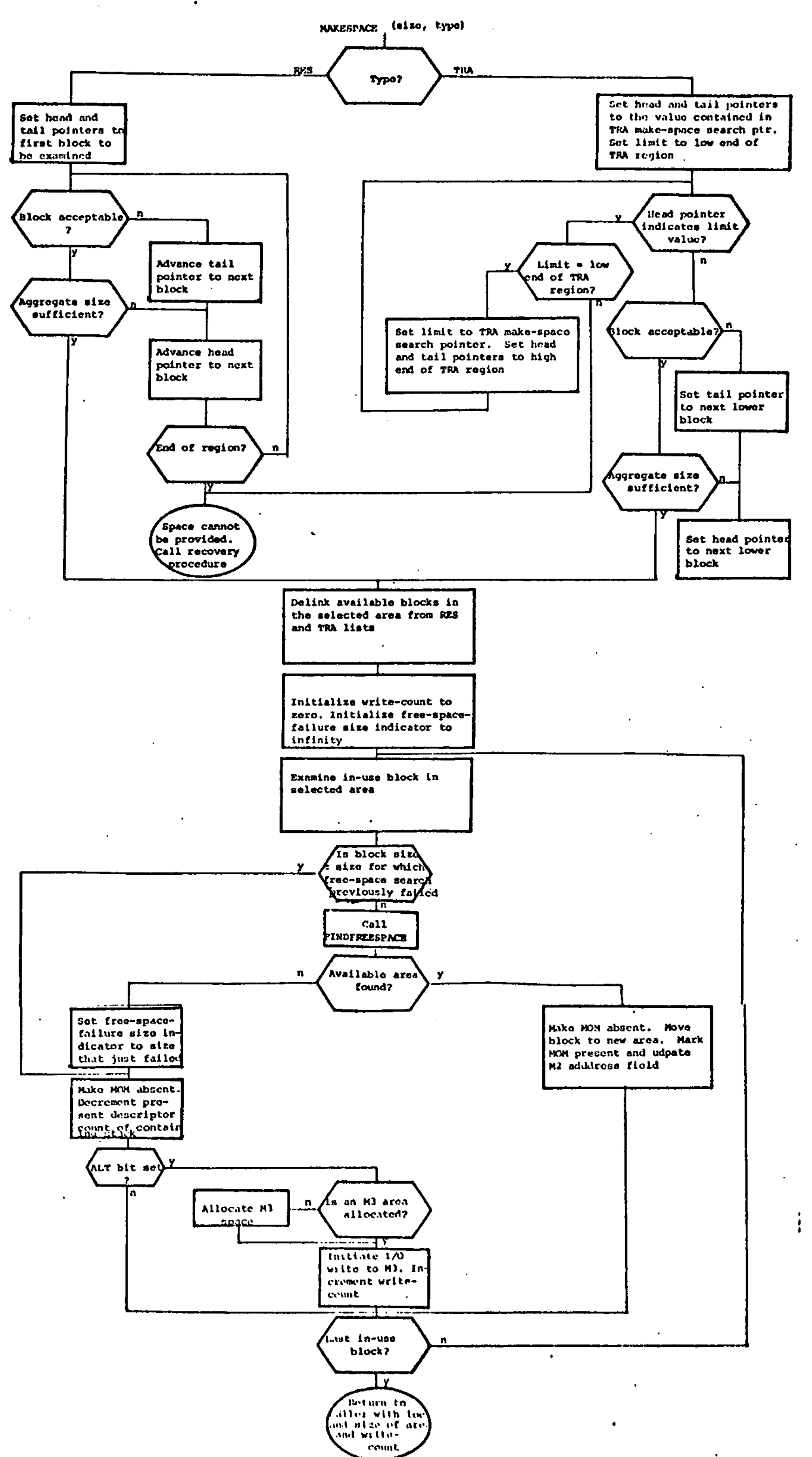
A block is defined to be "acceptable" in the above context if it is either available or in-use TRA with a reset REF-bit. Blocks which are RES or unusable are not acceptable, nor are in-use TRA blocks with REF-bits set. These blocks are passed by, but their REF-bits are turned off in the process; if they are not referenced before then they will be acceptable when they are next examined.

The acceptance criteria thus are seen to include available areas, and areas which are in-use TRA but not recently referenced. Such TRA blocks may be displaced from M2 if necessary; in most cases, an area exists for them in M3. If they have been altered, they must be written to M3; if they have not, the M3 copy is a duplicate of the M2 version, and no write to M3 is required. Depending upon the M3 characteristics, it may improve performance if a displaced segment is simply transferred to a new location in M2. The subsequent discussion is based upon the assumption that such an M2 to M2 transfer is preferred when it can be performed readily.

We resume consideration of the behavior of the MAKE-SPACE procedure at the point where an adequately large contiguous group of one or more acceptable segments has been identified. At this point, the blocks which are available are delinked from the available-memory lists. Next, it is necessary to handle the in-use segments either by moving them to a new location, or if this is not easily performed, by making them absent. For each segment, the FINDFREESPACE procedure is called to find an available TRA-list area big enough to contain the segment. If successful, the transfer is performed, and the Mom is appropriately modified.

If no available space is found, the Mom descriptor is marked absent, and the C-bit is examined to determine whether a write to M3 has to be initiated. If so, the I/O write-request is issued, and the C-bit turned off. This process is repeated for each in-use segment, although if an available-memory search failure occurs, the search-attempt is by-passed for subsequent blocks of equal or greater size. Although the multiprocessing environment appears to offer a chance that a subsequent search would be successful, it does not, in fact, offer this chance because of the necessity to restrict access to the procedures which manage global resources to one process at a time. Consequently, even if a running process had wished

Figure 3.4-8: MAKESPACE



to release storage, it would be prevented from doing so by the process (already executing) in the memory management procedures.

When all in-use segments have been moved or marked absent and related M3-writes have been issued, control is returned to the caller with a value indicating the size of the created space, and whether any writes were initiated.

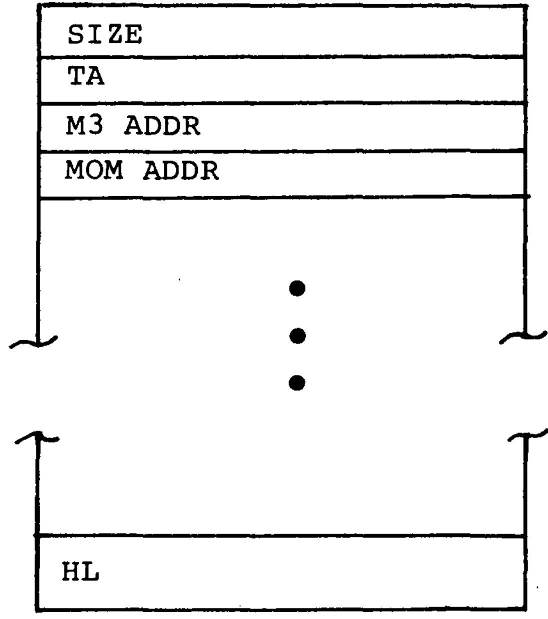
3.4.6 Addenda

At this point in the discussion it is appropriate to mention several small points which have influenced or do influence the way operations are done by the MAKESPACE procedure.

3.4.6.1 Administrative Data in In-Use Memory Blocks: Figure 3.4-9 shows the logical entities stored in each in-use memory block. This data includes the location of the Mom descriptor for the segment, and an M3 address field. When the segment is absent, the M3 address is contained in the Mom; when present, the Mom designates the M2 location.

3.4.6.2 M3 Writes: When it is desired to write a segment to M3, there is a possibility that no space has yet been allocated for it in M3. This case is recognized by a special M3 address value stored in the block, and is responded to by calling the procedure for M3 space allocation with size and class (TRA) as parameters. The circumstances in which this may arise include data arrays, initialized or not. An initialized data array is stored as a segment on M3 by the compiler. This segment is copied into M2 when the array is "created" at execution time. However, because the initialization values in M3 must be preserved for possible future use, the original M3 address may not be used as a target for M3-writing. Rather, a new location must be obtained when the first write (if any) to M3 is necessary. Uninitialized data-arrays have similar properties, except that not even an unusable M3 address exists.

3.4.6.3 Why M2-M2 Transfers of RES Blocks are Avoided: There is no logical necessity for avoiding M2-M2 transfers of RES in-use blocks to make space for a segment. The reason it isn't done is pragmatic: the MAKESPACE procedure identifies a set of blocks which is big enough to satisfy the request. If the set includes in-use blocks, they are TRA blocks, which means they may be moved to M3 should no hole for them be found



TA represents type and attributes

M3 ADDR is the location in M3 where this block is to be written or fetched

MOM ADDR is the stack number and offset for the Mom

HL is a header locator field

Figure 3.4-9 Logical Fields Utilized in In-use Memory Parts

in M2. The same cannot be said for RES blocks; if no space in M2 can be found, they have to be left where they are, since the meaning of RES is that the block must stay in M2. Thus the failure to find alternate M2 space for a RES block would spoil the acceptability of the set of blocks which were thought to be acceptable. Available blocks in the set, which would have been delinked from their lists in order to prevent their selection as a target for an M2 to M2 transfer, would therefore have to be re-linked, and the search re-initiated at the block "following" (higher or lower, depending on the search direction) the incubus-block.

A space-finding failure could be made less likely, if MAKESPACE could call itself recursively in the event no solution was found in the available-memory list. However, the ensuing overhead might become awesome, and we propose that such an approach be considered only if it appears capable of solving a performance problem encountered with the implemented system.

3.4.6.4 <u>The "Copy" Problem</u>: To reduce the length of the access path to a segment, it is desirable to avoid as often as possible the physical accesses of the level-0 stack (to locate the stack vector), the stack vector entry (to locate the appropriate stack), and finally the Mom descriptor (to locate the segment) when addressing a segment. For this purpose, an associative memory is provided within each processor, with the capability of delivering the contents of a Mom in response to the Mom's stack number and offset. While this makes a sizable reduction in the time required to traverse the access path, it introduces the characteristic problems which arise when copies of any alterable global data are made.

Specifically, when the memory management procedures make any change in a Mom descriptor, they must insure the absence of out-of-date copies by issuing a cancel instruction* to destroy such copies. Three different Mom-changes requiring such cancellation have been mentioned above:

a) turning the REF-bit off when a block is examined but found to have been recently used,

b) marking the descriptor absent,

c) modifying the descriptor when the segment has been moved.

Other cases exist in the sequel.

---

* The cancel instruction is a variant of the IPC operator described in Section 2.4.3.11.

3.4.6.5 Search-Pointer Invalidation: One of the four searches for space which have been described begins at a variable physical location. This is the search for TRA space by MAKESPACE which begins where it previously left off. The two available-memory searches involve linked lists. The search for RES space starts at the low end of M2 to deliberately encourage allocation of RES space near that end. These considerations do not apply to the search for TRA space; in fact, it is desired to make TRA-space survival probabilities uniform over the regions of memory where such space is allocated.

The RELEASESPACE procedure must account for the possibility that coalescing a freed block with a neighbor-block may eliminate a block to which the TRA search pointer is set by MAKESPACE. It therefore checks for this eventuality when blocks are coalesced. If it occurs, the pointer is simply adjusted to point to the block into which the previously pointed-at block was merged.

3.4.6.6 Stack Displacement from M2: Each process in the system is associated uniquely with some stack. It may use other stack sections as well, and other processes may use part of its stack, but there is a one-to-one association between a process and its "key" stack. While a process is running, its key stack is designated RES. At other times, key-stack sections may be selected for displacement by MAKESPACE. Because stacks may contain Mom descriptors, they represent a special case for memory management. Since it is necessary that all Moms in a stack be absent before the stack is displaced, the memory management routines maintain and observe a count of "present" Moms in the PIA (process information area) of each stack. When a Mom is accessed to change its present/absent indication, its stack number is always available; hence, the present-count is readily located.

Memory management thus records the usage of a memory area for a stack; when a segment is released or made absent or present, the present-count is adjusted. When MAKESPACE examines a memory part which contains a stack, the block is "accepted" only if it is TRA and its present-count is zero. This treatment will tend to preserve stacks in M2 longer than other segments, which is felt to be desirable.

3.4.7 Further Memory Management Procedures

This section describes procedures involved in the handling of memory segments.

3.4.7.1 <u>The Absent-Segment-Trap Handler:</u> When an Absent Segment Trap (AST) interrupt occurs, the ASTHANDLER procedure is called with a parameter which specifies the location (stack number and offset) of the absent descriptor. ASTHANDLER has three functions to perform:

a) obtain space for the absent segment,

b) initiate the transfer of the missing segment from M3 to M2 (sometimes not necessary), and

c) inspect the list of previously-initiated segment transfers to see if any completions have occurred, and take appropriate action if so.

The third of these functions might logically be performed by the I/O-completion handler. However, in the normal case, ASTs occur with such frequency that it is desirable to avoid the overhead of the I/O-completion handler for every segment arrival; therefore, the ASTHANDLER procedure issues I/O requests with a specification that completion is not to cause an interrupt. To prevent the arrival of the last segment from going unnoticed if ASTs became very infrequent for a period of time, the I/O controller signals I/O completion, even when suppresion was requested, if the completed request is not chained to a following request. (This sequence is explained in greater detail in section 3.5.)

The ASTHANDLER begins by establishing a lock to ensure that it is being executed by no more than one process. Then it examines the size of the required space. Next, it attempts to obtain space by calling the FINDFREESPACE procedure. If this fails, a call to the MAKESPACE procedure is executed to obtain space at the expense of a not-recently-referred-to segment. If the returned information from the MAKESPACE indicates that no writes to M3 were initiated, or following their completion if there were any, the amount of excess space granted by MAKESPACE is calculated. The size of the excess area may be zero, small, or not small. If it is zero, no action is required. If small, the area is marked "leftover"; otherwise, the RELEASESPACE procedure is called to link the block into the appropriate lists. Because the RELEASESPACE will examine the neighbors of the released block, and since the block of space just obtained is one of them, it is necessary to store the type information into the obtained area before this call is made.

Following the possible return of excess space, or immediately after space from the available list is found, ASTHANDLER may proceed directly to establish the absent segment. At this point, there are two possibilities: the segment does

or does not, exist on M3. Normally, it does; the other cases correspond to data segments declared without initialization, such as arrays and I/O buffers. When the segment is in M3, the M3 address is obtained from the Mom, and a read request is issued. When no initial contents are specified, the system designer has two alternatives: either let the system provide initialization, or make the segment present with whatever random contents it happens to contain. We adopt the safer choice, system initialization, since it is not possible at this time to assume that no accessing of the initial contents will take place. If the segment contained, for example, descriptors and other indirect-address words, the correct performance of store instructions would even be threatened. Hence, the ASTHANDLER procedure initializes the segment, marks it present, and sets the M2 address into the Mom.

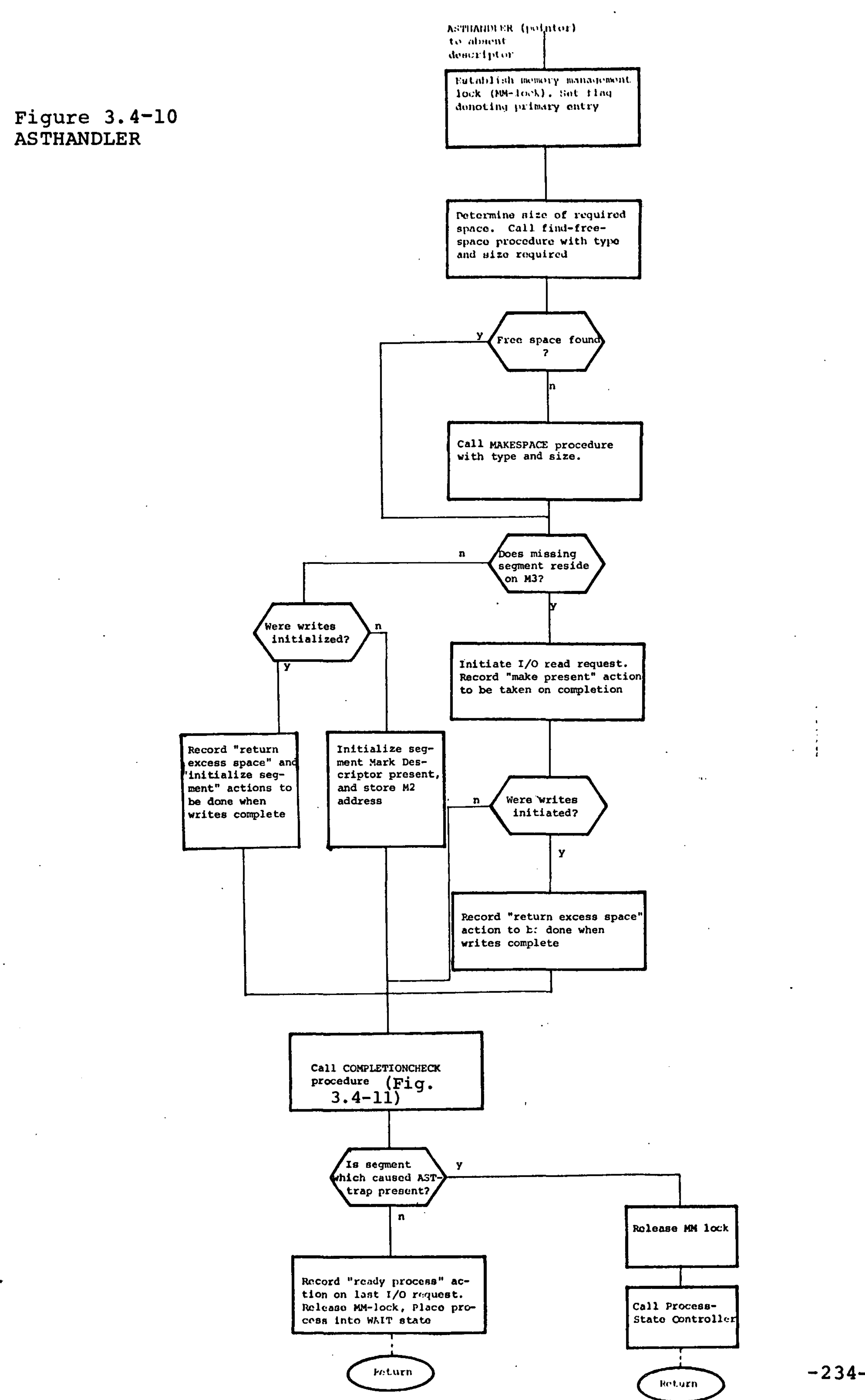
When the segment to be made present resides on M3, ASTHANDLER initiates a request for an M3 to M2 transfer, specifying that no completion interrupt be triggered. If the destination area in M2 required writes to M3 to make it available, the read request is chained onto the last write request.

ASTHANDLER performs its third function by examining the completion-status of previously-issued M3 transfers. Each of these keeps a record which specifies the action to be performed when completion occurs. Typically, whan an arbitrary I/O request is completed, two responses are required: first, a pending I/O request which has been delayed by a busy device or channel is initiated, and second, the process which issued the now-complete request is made aware of the completion. In the absent-segment case, the first of these responses occurs without software intervention by chaining I/O requests together in such a way that the I/O controller hardware itself initiates the next pending request. The second response, in most instances, is performed at the convenience of the ASTHANDLER to reduce overhead.

Thus, the table of outstanding requests is scanned; if any completions have occurred, the table entry is marked, so that it will not be re-interpreted; the actions specified within it are then performed. In the case of a write to M3, a space return call, a segment initialization, and the marking of the segment present may be necessary, perhaps followed by a call to the process-state controller to ready a waiting process. If the completion represents a segment arrival, the segment must be marked present, and a process may need to be readied.

The administrative functions could be performed by the process being readied after it is given a processor, except

Figure 3.4-10
ASTHANDLER



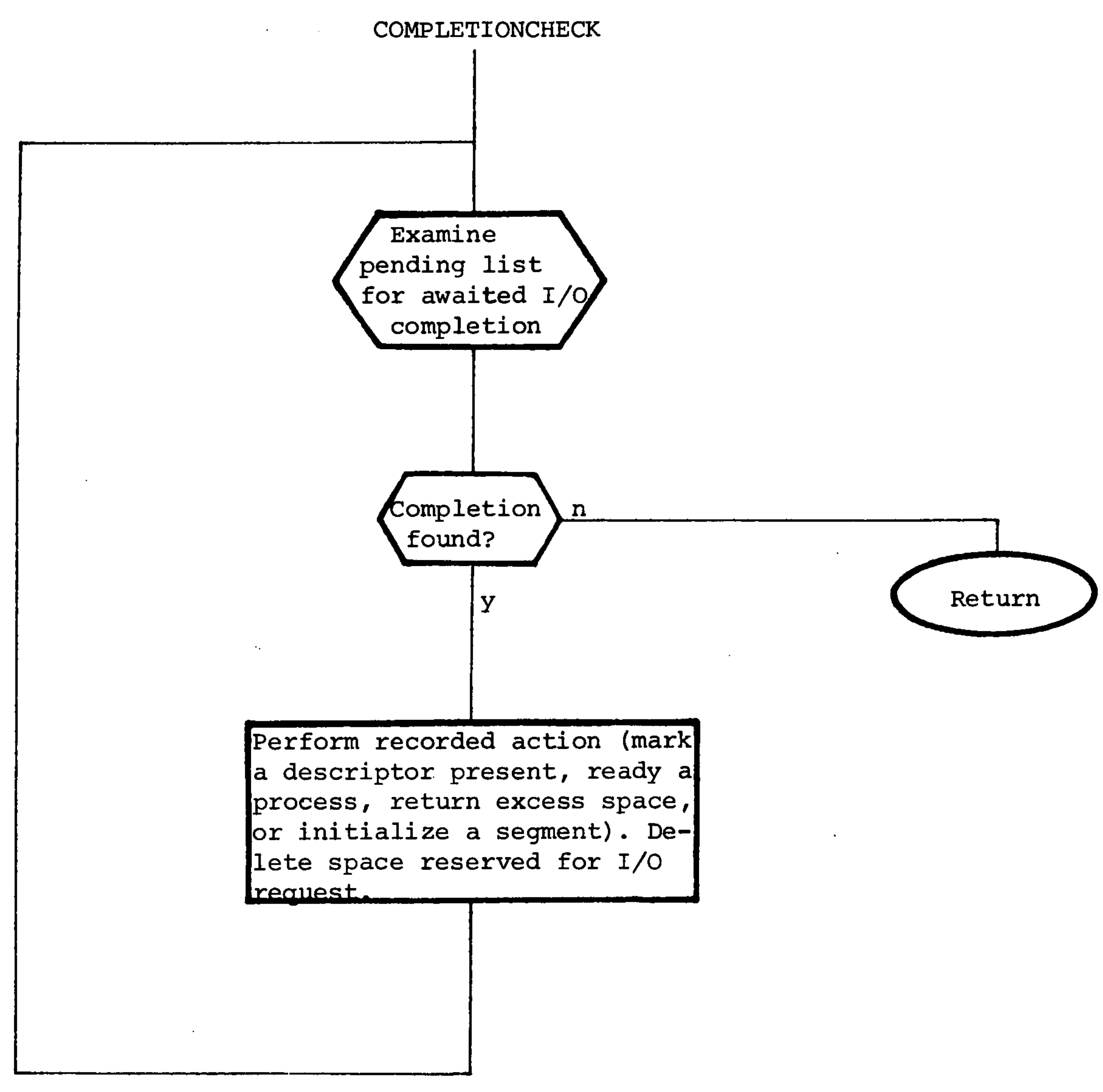


Figure 3.4-11 COMPLETION CHECK

for two considerations. First, a segment arrival may be of interest to more than one process; if the process which first attempted to make the segment present is of lower priority than one of the others, delegating to it the duty of marking the segment present may delay the resumption of the higher priority process. Second, there are cases where no process is readied by a segment arrival (a variant of the segment-load procedure, described below), and yet the functions must be performed anyway.

When all completions have been processed, the AST-HANDLER exits. The nature of its exit depends upon whether it has readied any processes, and whether its own process is able to continue immediately. If its own process has issued an uncompleted read or write request to M3, the ASTHANDLER issues a WAIT call on behalf of its process, after unlocking its lock. If its process does not need to wait, the Process State Controller is called to decide which process deserves the processor; otherwise the return is directly to the process which triggered the absent-segment trap.

To ensure that no other process attempts to make a segment which is already the subject of an M3 to M2 transfer present, the Mom for the segment designates the condition of "segment in transit". This is done by setting the M3 address to a distinguished value, which can be recognized by the AST-HANDLER upon entrance.

3.4.7.2 LOADSEGMENT: The LOADSEGMENT procedure may be called to make a segment present without stimulating the absent-segment trap. Two primary uses are forseen for this procedure:

a) pre-loading segments in anticipation of their use by real-time processes (to avoid the delay of absent-segment retrieval).

b) insuring the presence of key stack segments, to allow the Process State Controller to assign a processor to that process without encountering an AST.

The functioning of LOADSEGMENT closely follows that of the ASTHANDLER, except for the fact that the LOADSEGMENT procedure always returns to the caller. If requested to load a stack, the normal segment-arrival completion is set up, except that the process to be readied is not the caller of LOAD-SEGMENT, but the process associated with the loaded stack. When the loaded segment is not a stack, its arrival triggers only the action of making its Mom present; no Process State Controller call is made.

Before entering the functional sequence of ASTHANDLER, LOADSEGMENT checks the Mom to see if the segment is present. If so, the segment attributes are adjusted as specified by the call, and a return to the caller is performed with an indication that the segment is present. Thus, a call on the LOADSEGMENT to make a stack RES which happens to be present simply forces the type to RES, and returns.

3.4.7.3 CHANGESEGATTRIBUTES: The CHANGESEGATTRIBUTES procedure may be used to change the type and attributes as recorded in the Mom descriptor for the segment. This procedure requires two parameters:

a) The location of the specified Mom descriptor

b) The new type and attributes

If the segment is present, corresponding changes are made to the segment header and trailer areas.

As presently conceived, the CHANGESEGATTRIBUTES procedure will not accept requests which imply a size change for the specified segment, if a copy exists in M2 or M3. Although such a function is certainly implementable, this restriction is applied because no requirement for such a generalized capability has been identified, and the complexity of implementation does not warrant its creation merely to service some abstract need.

An important use of this procedure will be to relax the RES status of a stack segment to TRA when the logical necessity for retention of the stack in M2 has ceased to exist. Stacks will therefore become subject to displacement from M2 when they no longer contain "present" Moms. In this context, a further function was considered for the CHANGESEGATTRIBUTES procedure, but rejected: namely, when called to change a stack from RES to TRA, to search the stack either for 1) present Moms, or 2) present unaltered Moms, to call the RELEASESPACE procedure for unaltered segments, and mark them absent. For option 1), calls on a "make-segment-absent" procedure (not otherwise needed) would be required to write altered segments back to M3. Finally under option 1), since no present Moms would remain, the stack segment itself could be written to M3, making it absent. Performing the above actions might be defended on the basis that space in M2 would be made available expeditiously when immediate need for a process's stack subsided (such as when a WAIT request specifies a time for wake up a long time in the future). However, it is not clear that this would happen often, or that the ordinary workings of Memory Management would be

insufficient to prevent observable performance reduction in cases where it did occur. Hence, it has been decided to omit such a function, and leave the decision to displace the stack and its segments to the normal management operations.

3.4.8 Name Management

3.4.8.1 Introduction

The names used in a procedure are the symbols which identify the objects on which the procedure operates. HOLs all use names in this manner, and all presume certain rules of contextual interpretation which allow the same name to have different meanings in different contexts. Stated another way, names applied by a programmer need not be unique at the global level, but are required to be unique only in a given context.

The name scope rules introduced by ALGOL provide an excellent example of this concept. Figure 3.4-12 illustrates schematically several aspects of name-scope rules of the type used in ALGOL, PL/I, HAL, etc. Under these rules, a name declared in a block (denoted in the figure by the long left brackets) is recognized throughout that block, including blocks contained in that block, unless the name is also declared in a contained block. In Figure 3.4-12 block X contains a declaration for A; this identifier is recognized throughout blocks X and Y as a reference to the value of the variable declared in block X. Block W declares A and B; the variable thereby specified to be associated with the identifier "A" is completely distinct from the variable called A in blocks X and Y. References to A in block Z and the part of block W outside block X refer to the variable declared in block W. The variable B declared in block W is recognized everywhere except in block Z, where a declaration there defines B to be "external". The attribute "external" means that the B in the declaration is not defined within the section of program which will be compiled as a unit; hence, the "Call B" statement in block Z refers to a separately compiled (i.e., external to W) procedure.

The names associated with procedure blocks are treated as through they were declared in the block which contains the named block. Thus, X and Z are recognized in exactly the same contexts in which the A and B declared in block W are recognized. Thus, the only names known outside block W are the name W itself, and the B declared to be externally defined. The external B called in Z is likely to be the name (like W) of a compiled unit of code.

It is readily seen that a unit of source code treated as a separately compilable entity creates a division of context with respect to name recognition. Whereas the name A may be used with no concern whatever for uses of A within other compilable units, the names W and B (the external B) must be

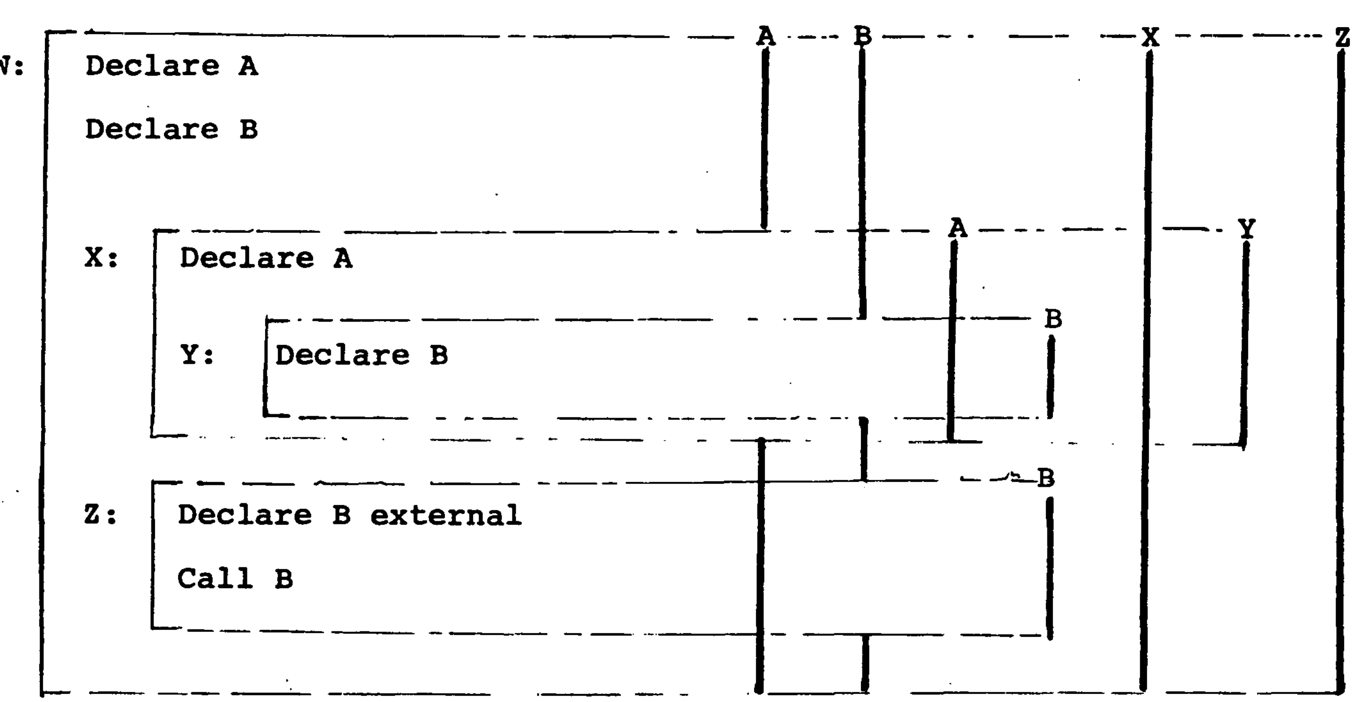


Figure 3.4-12 ALGOL - like name scope

unique at a more global level. For example, if two different procedures were compiled under the name B, there would be an ambiguity as to which of them was referred to in block Z, and additional information is required to resolve the ambiguity. This is the area to which name management applies.

3.4.8.2 Directory Structure: Name resolution may occur at a number of points in the life of a procedure. The discussion in this section is directed at the management of names relative to the M3 memory.

The objective of M3 name management is to establish the links between identifiers and areas of storage on M3. This is accomplished by the use of M3 directories. In conventional computer systems, the connection between location and name is direct. Directories are consulted with names as keys, and when the name is found, that entry in the directory supplies the storage address. The directory structure for the current system follows conventional practice with respect to name storage. Following a discussion of these characteristics, two approaches to the retrieval of information from M3 are described.

A hierarchy of directory files constitutes the M3 directory. Each file stored on M3 has a symbolic name. Hence, each directory file is named. At the apex of the directory structure is the root directory. Each directory entry contains the name of a file, which may be a directory file. No directory may contain more than one entry for the same name. Consequently, while a name may appear in more than one directory, the "pathname" for that name is unique. The pathname is defined as the concatenation of the names of the directories along the path from the root to the file name, each pair separated by a special character, for example ".", placed between names. Figure 3.4-13 shows a simple example of a directory tree. While the name A appears in several directories, it may be seen that the pathnames are all distinct: ROOT.A, ROOT.D1.A, ROOT.D1.D5.A, and ROOT.DIR·A. If no file is allowed to appear in more than one directory, the structure is truly tree-like, since no interconnections between branches occur. Hence, pathnames are unique, in addition to being distinct: each file has one and only one pathname.

If all files were linked back to the directory in which their entry appears, it would be possible to calculate the pathname of a file from knowledge of the file itself. Thus, it is possible to use only a part of a pathname (for example, its last component) if a directory pathname is specified implicitly or explicitly; furthermore, it is straightforward to define a set of search rules which may require the use of

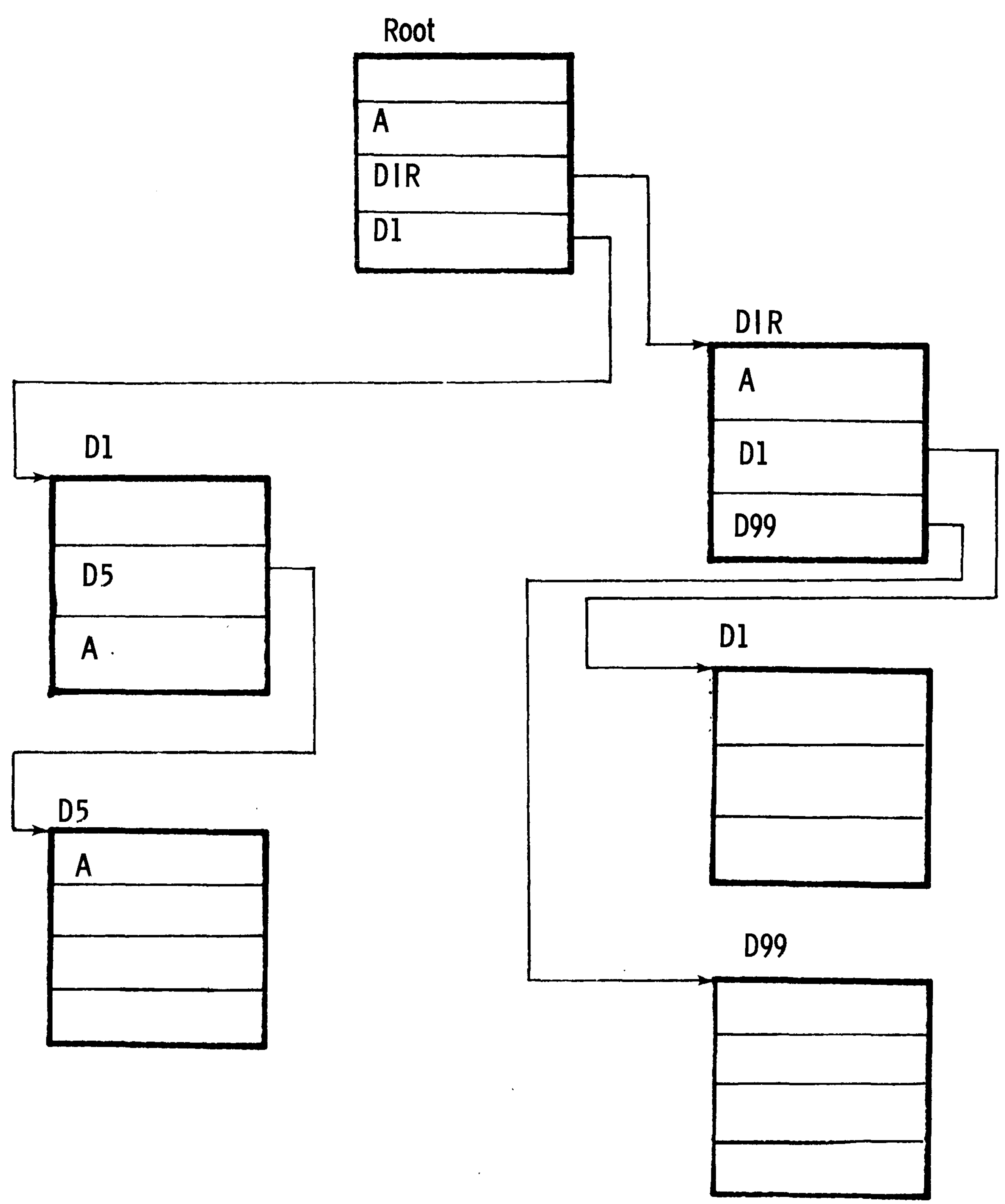


Figure 3.4-13
A Directory Tree

directories closer to the root directory than the one specified. One such search strategy might be to: 1) search for the specified name in the specified directory; if not found, 2) search directories whose entries appear in the same directory which contains the entry for the specified directory; if not found, then 3) search sibling directories at the next higher level, and so on.

While the provision of facilities through which search rules may be implemented is a responsibility of the operating system designer, the determination of one or more sets useful in particular applications is postponed. The discussion therefore passes to the "definition" aspects of the directory: that information located with the name which enables the file to be located in M3. This subject, in the present system, must be considered in conjunction with fault-tolerance in M3, since it may be necessary to move a file from one location to another at an arbitrary time to recover from failure.

Whereas in conventional systems it is sufficient to place the storage location of a file into the directory entry, an alternate scheme is preferred in a fault-tolerant system design. We choose to make the association from name to M3 address a two-step procedure rather than a single step. This is achieved by providing each segment with a globally-unique identifier of manageable size. The pathname is not usable for two reasons: first, the pathname may be altered without affecting the segment itself, and second, the pathname is of variable length and too bulky. Therefore, a name is given to a file when it is created; this name may be obtained from the system clock (refer to section 3.2.5.1 on timer handling). This name corresponds one-for-one with a file. If the file is not modified, the name is not changed; if the file is modified or deleted, the name is no longer applicable, and furthermore, will never be used again. This proposed technique imposes a requirement on the M3 device. This is simply that to obtain segment j from a file whose global name is i, an I/O request is issued which specifies only the two values i and j to the device to locate the desired record. The subsequent response of M3 then resembles that of a content-addressable device: the name ij is used <u>by the device</u> to locate the segment, the physical location not being specified or revealed. With an added capability to relocate segments, or at least to rename them, this form of accessing provides for straightforward fault-tolerance in M3, without requiring participation of the I/O software, except perhaps for the recovery-procedure's action in copying information from a faulty area to a healthy one.

The reason that this formulation has the desired behavior is that each file is referred to in terms of its name

and logical structure; details of mapping from name and segment to M3 address are left solely to the device itself. This prevents the following logical problem from arising. If several processes had looked up a name in a directory, and made a copy of the related M3 location, and then, for efficiency, simply specified I/O operations in terms of the copied location, no means would be available to discover which processes owned such copies, or to advise them that the locations had been changed or were no longer valid.

An M3 device with this capability is not known to exist at the performance levels required in the present application, although the facilities provided in the IBM 360 direct-access devices for the handling of indexed data sets are quite good. In any case, to allow for the possibility that the system may be implemented without such a device, a work-around scheme is also presented. This alternative scheme has as its objective the creation of a section of an access path to a file which 1) provides the physical location of the file, and 2) which is used in all references to the file.

The operating system implements this by requiring that every user of a file declare his intention by calling the OS for access permission. If this file is not already in use, the OS marks the directory to show that it currently resides in M2, and where, and copies (moves) the information from the directory to the M2 location. For each subsequent user-request for access to the same file the OS will note that the directory entry is in M2, and will increment a count which records the current number of users. For each reference to the file, the OS will join the record (or segment) number to the copy of the file location it keeps, and issue the I/O request accordingly. As users indicate that they have finished with a file, the OS diminishes the user count; when it reaches zero, the directory entry is copied back to the directory in M3, and the M2 space is freed.

Should a fault occur in any record of a given in-use file, its directory entry will specify its M2 location. The recovery software may thus move the file, and update the M2 copy of the file's directory entry, locking out processes from access until the move is complete. As a result, all user processes, when allowed to continue, will refer to the file in its new location.

3.4.8.3 <u>Objects Named in Directories</u>: Although it would be possible to provide a directory entry for every segment and record in the M3 system, simplification and overhead reduction

occur if larger objects, on the average, are addressed in this way. Hence, directory entries are utilized to address:

a) Directories

b) Programs

c) Compools

d) Files

Directories have already been mentioned. By "programs" is meant the separately-compiled unit handled by a language translator: this normally includes procedure blocks and data arrays declared in the interior of the outer block. Compools are discussed in the following section (3.4.9). The term "files" refers to M3 data structures which may be accessed by a process through explicit I/O requests. This contrasts with the segments of a program, which are normally transferred between M3 and M2 under the automatic response of absent-segment traps and Memory Management software. Thus, programs must contain a segment which is a sub-directory for the remaining segments; this record is used to form the level 1 stack of a process in execution.

3.4.9 Compool Management

Many modern programming languages have the facility of sharing named blocks of data on a global level. Since these blocks are known on a level which is higher than that of a process, their management must be relegated to executive control. Within the MP system such Compools are identified by their character names.

When a process first references an element of a Compool, control is automatically transferred to the executive procedure which resolves the Compool linkage to the process. If the named Compool already exists within the system, it will be linked to the using process which will then resume. If the Compool does not yet exist, it will be entered into the current system list and then, as above, the process will be linked. After the first reference to a particular Compool, the linkage is established, and no further use of the executive linking function is made. When the process terminates and space is being de-allocated, any references to Compools must be deleted.

Figure 3.4-14 shows that a named Compool has the appearance of a user process's stack.

```
"name" COMPOOL:

     DECLARE S SCALAR;
     DECLARE V SCALAR ARRAY 3;
     DECLARE T SCALAR;
     DECLARE W SCALAR ARRAY 4;
               •
               •
               •
```


Figure 3.4-14: Compool Stack Structure

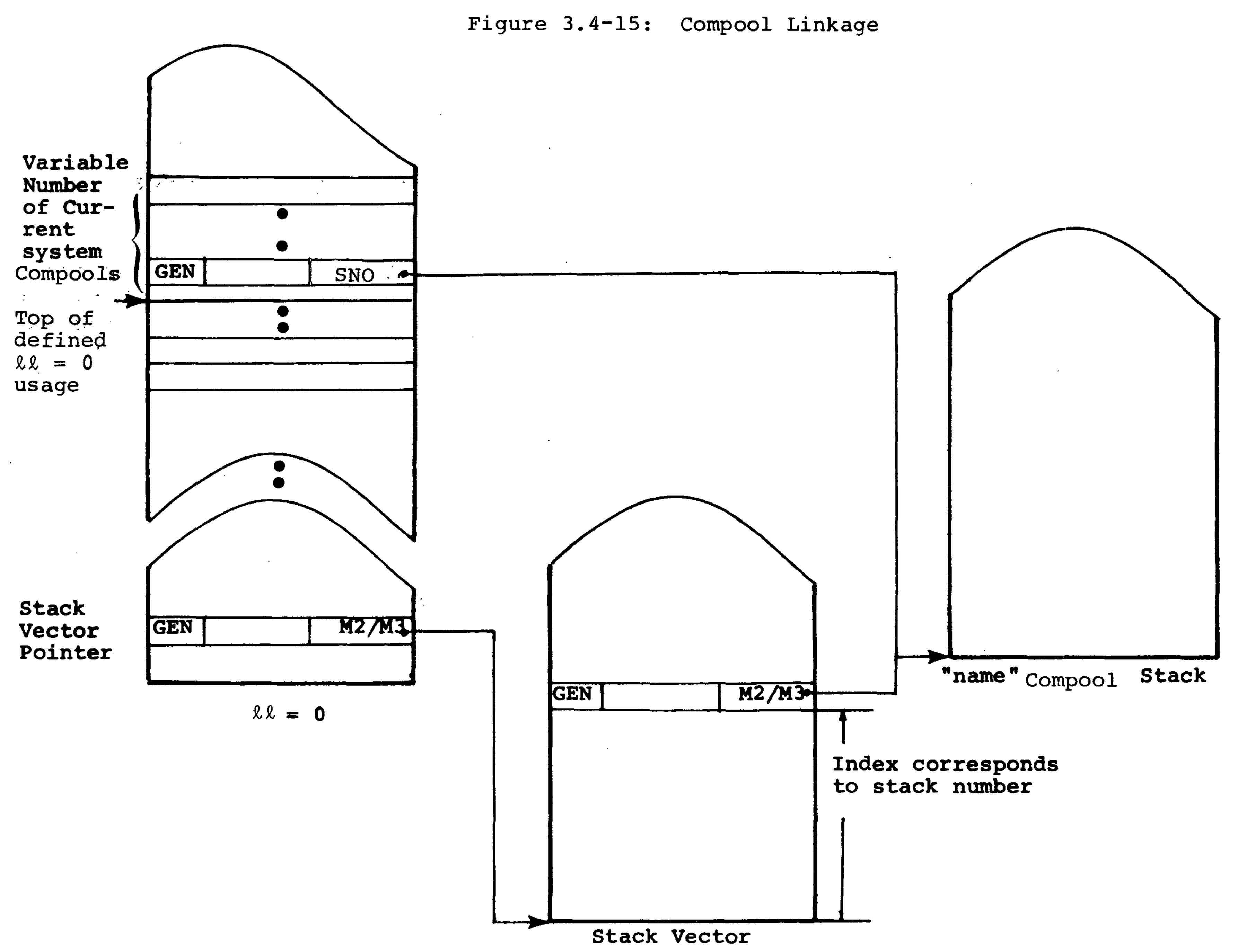
Scalars (integers) are kept within the Compool stack; arrays are indicated by descriptors within the stack. This formal identity to normal usage allows Memory Management to treat Compool segments in the same way as all other segments. It would have been possible either to create all of a given Compool storage in one array, or in arrays of all single elements, but this could create difficulty for Memory Management in the creation of extremely large arrays, which in turn might exceed addressing capabilities.

Besides the storage of the declared data, each Compool stack contains a back pointer (an ADW to the Mom descriptor which contains its physical address, and a "user count" value so that Memory Management may keep track of the number of current users (the count is zero if the last program using the Compool has finished). The base of the Compool stack also contains a character string descriptor which points to the Compool name, for the purpose of identification.

Figure 3.4-15 shows the system Compool linkages. Since the Compool is considered to be a stack, it must have its Mom descriptor in the system stack vector. In order to determine what Compools are currently known to the executive, the top of the lexical level zero array is used for Copy descriptors of the Compool stacks. A count of the number of named Compools currently in existence is maintained in the lexical level zero array. Therefore, whenever a Compool is referenced for the first time this group of descriptors is searched with the "name" character string within the actual Compool stack to discover the presence of the new used Compool. If it is found, then the user's Compool descriptor (figure 3.4-16) can be resolved. If it is not found, the Compool must be brought into the system and entered both into the stack vector and into this Compool search array. Then the user's Compool descriptor can be resolved.

Figure 3.4-16 shows the state of the user's Compool descriptor in both the not-yet-referenced, and the resolved states. When the Compool has not yet been referenced, the Compool descriptor has its X bits set to 10. This will cause the system to try and resolve the Compool descriptor to point to the relevant Compool. Before it is resolved, the Compool descriptor's pointer field points to a character descriptor, which in turn points to a character string giving the Compool's identification name. After resolution, the Compool's pointer field contains the relevant stack number-offset value pointing directly to the named Compool. The X bits in the Compool descriptor have therefore been changed to 11.

Figure 3.4-15: Compool Linkage



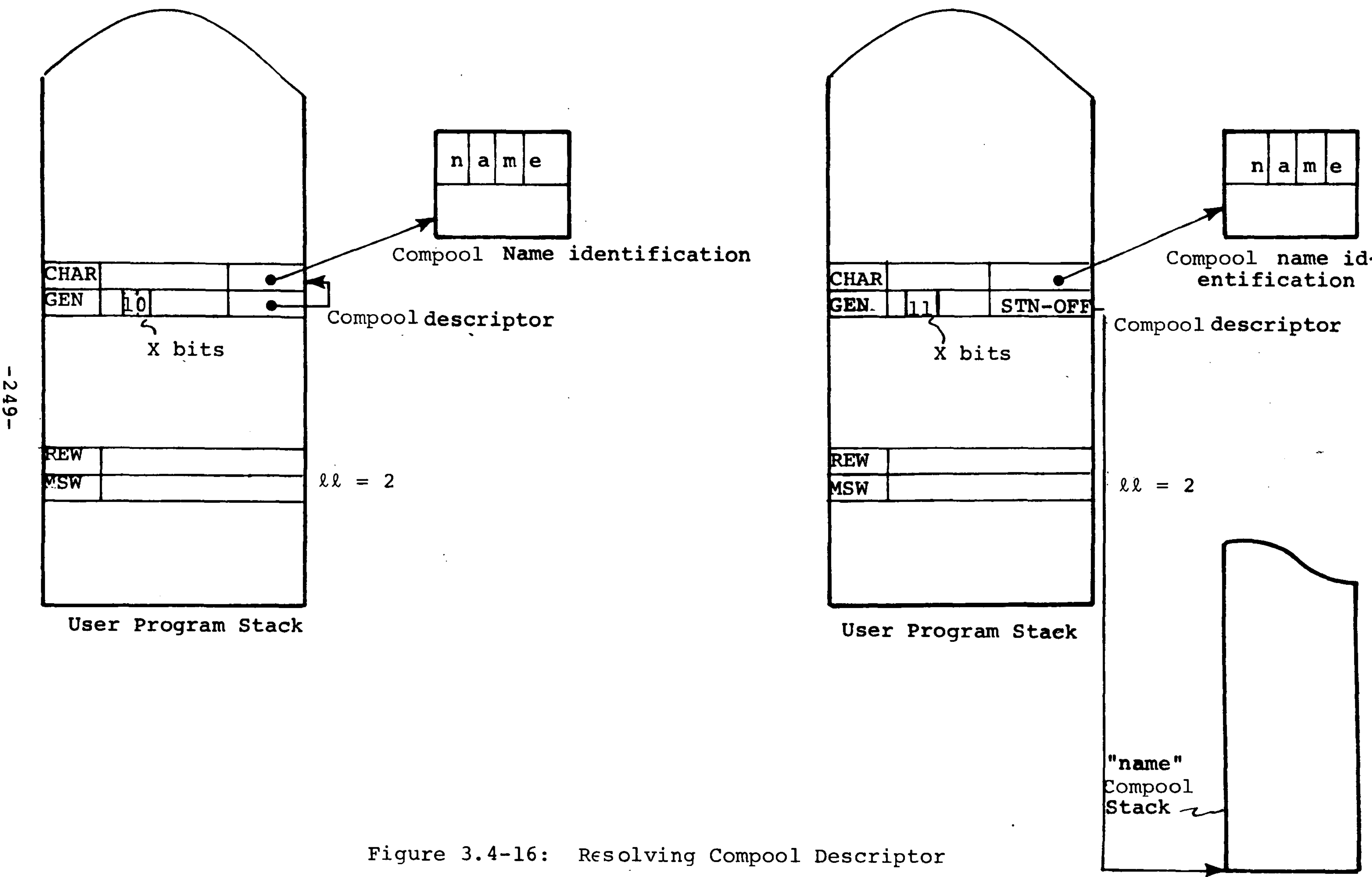


Figure 3.4-16: Resolving Compool Descriptor

Upon finishing a program, the executive decrements the "user counter values" within the appropriate Compools. (These are those for which the Compool descriptors in the process stack contained X bits set to 11.)

3.5 I/O Management

The functions to be performed by the I/O management procedures are determined: by the types of I/O devices with which the computer is to communicate, by the nature of the interface between computer and device, and by the purpose of the data transfer.

In order to be able to propose a method of I/O management, some assumptions must be made about the kinds of data transfers expected to be made with the world outside the multiprocessor. It will be assumed for the purpose of the current design that three basic types of I/O device interface will be serviced:

a) High speed block data transfer, as characterized by I/O discs, drums, tapes. Once a block transfer is initiated across such an interface, it continutes to completion. Block sizes are in excess of hundreds of words, and data rates are in millions of bits per second.

b) Slow speed, single data transfers, as typified by card readers, keyboards, and printers. Such transfers are often in less than tens of words at a time, and data rates typically thousands rather than millions of bits per second.

c) Short response time, high speed, short data transfers which are characteristic of a central avionics data bus.

Each of these types of interface will be briefly examined in the following section. After that the information structures required to control these device types, and details of a set of I/O procedures will be given.

3.5.1 I/O Interface Categories

3.5.1.1 High Speed Block Transfer Interface. The dynamics of this interface are most heavily impacted by its role in the transfer of memory segments between M3 and the multiplexed M2, as described in Section 3.4.

It is also affected by the choice of M3 storage technology. For reasons given in the Introduction to this report it was decided to adopt a conventional, rotating, magnetic device for M3, rather than one of the more advanced solid state mass memory technologies that are being developed today. A consequent decision was to interface the disk (or drum) through the I/O controller, and to manage I/O transactions by a set of I/O executive procedures, basically as has become standard practice in the larger ground-based computing facilities of today. File I/O and memory segment I/O are thus handled, with the exception of some specific differences which are detailed in Section 3.4.7.1, by the same executive procedures.

A fixed head drum can exhibit an average access time in the 5 ms to 10 ms range, corresponding to rotational speeds of 6000 to 3000 rpm respectively. This performance is expected to be adequate in supporting the expected processing capability of the multiprocessor, although an accurate assessment of the necessary M3 performance will only be obtained by observing the computer under near-realistic processing loads.

Once a desired file has been located on a drum-like M3, the data may be transferred at rates on the order of 1 to 3 x 10^5 words per second. The transfer must be allowed to complete, else a further access delay of 5 to 10 milliseconds will be incurred. For memory segment transfers of tens to hundreds of words, the actual transfer occupies a few milliseconds; i.e., an interval of the same order as access time. It will be assumed that only one data transfer at a time can be accommodated by a single M3 device. This is not a fundamental restriction, but is usually dictated by the cost of providing multiple read and write electronics, buffers, control circuits, etc.

A high speed port into M2 can accommodate a higher data rate than required to sustain a single M3 transfer: without interleaving, up to 10^6 words per second; with the four-way interleaving described in Section 5, nearly 4 x 10^6 words per second are possible. An I/O controller channel servicing an M3 device of the type assumed could therefore be parallelled by several others, if the need to provide multiple simultaneous data transfers to high speed, block organized devices necessary. For example, a high speed tape unit could be connected and used simultaneously with an M3 device, perhaps through the same, suitably designed, channel. The description of I/O procedures given later in this section does not differentiate between I/O requests to disks, drums, or tapes.

3.5.1.2 Slow Speed, Unit Record Transfer Interface. This interface is considered separately here only because of its role in a laboratory implementation of the multiprocessor. It serves the basic unit record devices: card reader/punch, operator console/keyboard, and line printer. These devices are of negligible importance (in terms of performance) in an operational space station implementation (and may not even be required), but become the principle I/O devices in an experimental environment. From an I/O control point of view their main characteristics are:

a) Long delays in accessing, due to mechanical and/or human response times, of up to fractions of a second.

b) Data lengths of only a few bytes or words, determined by physical limitations such as punched card capacity, keyboard encoding, printer character set, etc.

The current design will follow conventional practice, which is to time multiplex I/O to this type of device by servicing all devices through a single channel and a single electrical interface. As a reflection of the lesser importance of unit record devices no specific I/O data structures or control procedures will be devised to suit their particular characteristics. Unit record I/O will be handled by the techniques developed to control M3 I/O, even though this may be a non-optimal approach.

3.5.1.3 Avionics Data Bus Interface. The interface between the computer and a multiplexed central avionics data bus shares certain characteristics with the interfaces described above, but in addition has its own peculiar properties. Data buses can be configured in a number of ways, with widely differing performance, and control requirements. In order to make more than sweeping generalizations, some specific assumption of data bus characteristics must be made. Studies to date [7], have shown that an initial Earth Orbital Space Station can be serviced by a data bus which has the following typical characteristics:

Multiplexing:	Time division (TDM)
Frequency:	10 MHz
Number of devices (stations):	256
Command structure:	Command/response

These are the important control characteristics, from the point of view of I/O communication.

Command/response implies central computer control. Bus I/O takes place only at the behest of the computer; no device may volunteer information. It is our opinion, however, that although a strict C/R control policy may be shown to be quite adequate at this stage of Space Station development, it will be advantageous to provide a bus interrupt capability. This is not so much to provide the devices with control authority, but rather to accommodate bus control units (BCU) designed to perform off-line chores such as error monitoring, detection of unusual conditions, response to unsolicited communication from Station subsystems, etc. These would otherwise be programmed into the I/O software and executed with repetitive I/O requests to the BCU, which obviously increase traffic at the interface between I/O controller and data bus.

It is expected that bus communication between computer and a device on the bus will consist of short blocks of data, typically from one to 128 bytes in length. This conclusion is based on the assumption that high speed repetitive functions, such as the evaluation of a strapdown inertial algorithm would be performed locally and not over the data bus by the central computer. Data transfers of blocks of data larger than 1 to 128 bytes are usually not time critical; e.g. CRT display frame transmission. These can be broken into smaller blocks, and transmitted separately.

A typical data bus communication will start out with a command word from the BCU containing the address of a specific device and an operation code. (See reference [8] for a more detailed treatment.) The response to this is an echo of the BCU command by the device, indicating correct receipt of the message. Following this preamble to establish the communication link there is the transfer of the data dictated by the operation code, to or from the device. The number of bits of data required for one complete communication depends on the data length. Totals are shown below for a "short" message of 4 bytes and a "long" message of 128 bytes (1 byte = 8 bits).

Number of Bytes

Message Type	Control		Data	Total
	Command	Echo		
Short	4	4	4	12
Long	4	4	128	136

To obtain an approximate idea of the data bus bits rates involved in servicing all devices, it will be assumed that 80% of the 256 devices are commanded with short messages, and 20% with long:

Message Type	Total	Bytes	Bits
Short	205	2460	2×10^4
Long	51	6936	5.5×10^4
Total			7.5×10^4

A complete service cycle of all devices on a maximally configured bus thus generates 75K bits. For a 10 MHz transmission frequency this cycle can be repeated every 7.5 milliseconds. In practice, delays due to finite transmission speeds will increase the cycle time. However, a 10 ms to 20 ms bus service cycle seems to be entirely achievable. A 20 ms cycle with the preponderance of long communications assumed will generate about 300K bytes/sec of actual data; i.e., a data rate comparable to that of the higher speed storage devices such as drums, disks, and tapes. However, a data bus differs significantly in the manner in which this data is addressed and controlled. The type of bus described is essentially a table-driven device: in practice communication between the computer and the avionics devices will occur as follows:

a) Number of device interfaces will require to be accessed for a real time data at the highest service cycle frequency; i.e., every 10 ms to 20 ms.

b) Others will require accessing periodically, but at lower frequencies than the maximum.

c) Some will require occasional sampling at random intervals.

The mix of devices in each category is a function of mission phase and/or Station operations. It is a delicate design problem to ensure that all the highest frequency requests are completed without exceeding the basic bus service cycle, and without losing some of the less frequent requests. Since these constraints are known only to the system implementor, a specific bus confi-

guration should not be assumed by the hardware (or implicitly designed into the system software).

3.5.2 Peripheral I/O Data Structures and Control Procedures

This section will define the basic data structures and control procedures involved in peripheral I/O operations, as differentiated from those associated with data bus I/O. I/O to devices serviced by both the high speed selector-type channel and the slower multiplexer channel are handled in a basically similar fashion.

3.5.2.1 Basic Peripheral I/O Sequence. Before a more detailed description of each I/O data structure and control procedure is undertaken, the following summary of a typical peripheral I/O operation from initiation to completion is given to illustrate the process:

a) An I/O request is initiated whenever a running process executes a READ or a WRITE statement. This transfers control to the executive procedure REQUESTIO.

b) REQUESTIO creates an I/O Control Block or IOCB, which contains all pertinent information concerning the I/O request, by calling procedure FORMIOCB.

c) The IOCBs for a particular device are formed into a list, termed the DEVICEQ, by procedure QUEUPIO. Entries in this list are removed when the associated I/O operation is completed, or is for some reason deleted before being accepted by the I/O controller.

d) A component of each IOCB in the DEVICEQ is an I/O control word, or IOCW. This is the element that is directly accessed by the I/O controller, and into which it writes the status of each request (see later sections).

e) The IOC will access IOCWs sequentially in all DEVICEQs. At any given time there will only be one IOCW from each DEVICEQ active within the IOC. As each IOCW is executed the requested data is transferred to (or from) a specified M2 location, and at the completion of the transfer an I/O complete interrupt is normally triggered. However, some I/O requests will indicate

that no such interrupt is to be given. At the completion of these the I/O controller will set an appropriate field in the IOCW, and proceed to the next IOCW in the DEVICEQ. When all IOCWs in a DEVICEQ have been serviced, the IOC will issue an interrupt to indicate the DEVICEQ is empty.

f) The "I/O complete" and "DEVICEQ empty" interrupts cause procedure IOFINISHD to be entered. This procedure will do any of three things:

1) It will cause a specified process to be transferred from the WAIT to the READY state.

2) It will set a specified Event and call the EVENTHANDLER routine.

3) In the case of responses to absent segment requests, IOFINISHED transfers control to the memory management procedures.

One or more of these three cases is always specified by the initiator of an I/O request.

3.5.2.2 I/O Data Structures. It was evident from the above brief summary of an I/O transaction sequence that the basic element of data involved in an I/O operation is a collection of control information called an I/O Control Block, or IOCB. (It should be noted that in a given implementation of the I/O data structures and control procedures, the IOCB need not take the form of a contiguous group of uniformly organized words. The following description of its logical components does not necessarily, therefore, imply a physical structure.) The IOCB may be functionally divided into the component fields illustrated in Figure 3.5-1.

<table>
<tr><td colspan="2">Initiator</td></tr>
<tr><td colspan="2">Action</td></tr>
<tr><td colspan="2">Priority</td></tr>
<tr><td rowspan="4">IOCW</td><td>M2</td></tr>
<tr><td>Operation</td></tr>
<tr><td>Control</td></tr>
<tr><td>Link</td></tr>
</table>

Figure 3.5-1 IOCB Structure

These fields are provided to instruct both the I/O executive procedures, and the IOC in the execution of a given I/O request. The first three fields contain information primarily for the benefit of the I/O control procedures:

a) Initiator. This field identifies the initiator of the I/O request. It may or may not be the beneficiary of the complete operation.

b) Action. Information in this field determines largely how the I/O request is handled. It contains two subfields providing direction at request time and at completion time:

   1) At request time ACTION indicates whether control should be returned to the initiating process immediately the request has been placed, or whether the initiating process should be put into the WAIT state until the required I/O operation has been completed.

   2) ACTION instructs the I/O procedure IOFINISHD (which is entered whenever "I/O complete" or "DEVICEQ empty" interrupts are signalled by the IOC, and which scans all active requests for status) what to do in the case of each completed I/O request:

      a) Whether to place a process into the READY state. If so, the appropriate stack number, processid is furnished.

      b) Whether to set an event, in which case the event is identified by the stack number and offset of its descriptor.

      c) Whether the request was from the Absent Segment Handler, in which case Memory Management is called with appropriate parameters (see Section 3.4).

c) Priority. Normally requests for I/O to a given device are placed in the order they arrive. That is, DEVICEQ's are FIFO lists of IOCBs. However, if a more optimal ordering is desired, it may be established by indicating the relative priority of an I/O request in this field.

The remaining fields in the ICOB are used primarily to control and communicate with the IOC. The following four subfields are grouped and collectively termed the I/O control word (IOCW):

a) M2. This subfield defines the space in M2 allocated to receive the I/O data transfer, or from which I/O output is to be taken. It consists of the actual M2 address of the beginning of the M2 space, and the size of the space:

M2:

Address	Size

b) Operation. This subfield defines the path of control to the specified device location where the desired data either resides or is to be directed. Its several component fields define channel number and command, device number and command, and device address and size of the physical data file to be transferred:

Operation:

Channel		Device		File	
#	Command	#	Command	#	Command

c) Control. Communication between the executive I/O procedures and the IOC occurs in this subfield. Its various component fields are illustrated and described below.

Control:

Active	Quiet	Accepted	Complete	Error	Status

Active: The I/O request initiated by REQUESTIO, may indeed have been completed by the IOC, but has not yet been acknowledged by the intended recipient. This indication is necessary because the Absent Segment Handler may not be capable of responding immediately to the presence of a new segment, or it may wish to respond to those I/O completes for which the interrupt was suppressed (see Section 3.4 for details). When the completed I/O operation is finally acknowledged by the intended recipient, this field is reset, and the space occupied by the IOCB may be overwritten by another IOCB.

Quiet: This field indicates whether I/O completion is to be signalled by an I/O complete interrupt, or just by setting the complete field.

Accepted: The IOC sets this field to indicate that it is now in process of responding to the request.

Complete: The IOC sets this field at the completion of the I/O request.

Error: If the I/O operation failed to complete without errors, this field is set. It instructs the procedure IOFINISHD to analyze the status field for the cause of the failure.

Status: A field containing encoded device and channel status and/or error conditions set by the IOC at I/O completion.

d) Link:

Previous: This subfield is set by the procedure SETUPQ to insert the current IOCB into a DEVICEQ when requests are not to be arranged as a FIFO list. It also indicates when the current IOCB is the only one in the queue.

Next: This subfield is read by the IOC to determine where its next IOCW should be taken from. "Next" also indicates if there are no further IOCBs in the DEVICEQ: the IOC interprets this as an instruction to issue the "DEVICEQ empty" interrupt.

As stated previously, all IOCB's for a given device are linked into a DEVICEQ by setting the link field in the IOCW appropriately: Figure 3.5-2 illustrates three DEVICEQs.

Figure 3.5-2 DEVICEQ Linkages

The IOC takes one IOCW from each DEVICEQ associated with a given channel, and initiates the corresponding parallel I/O sequences with the appropriate devices. This implies that the IOC contains the necessary internal storage to maintain parallel I/O processing of multiple devices. To start the IOC at the first entry of a given DEVICEQ requires a hard-wired command to the IOC from a processor establishing a new DEVICEQ.

3.5.2.3 I/O Control Procedures. An I/O request to the IOC is required to specify the channel, device and file address. At compile time this data is generally not availalbe: file declarations and file READ and WRITE statements are made with "local" names assigned by the programmer. An important step that must be taken before these statements can be executed is to associate the local names with real file names and addresses. This is done at process creation, and/or at READ, WRITE execution time. Two procedures may be invoked to establish and link a file:

a) CREATEFCB: This procedure is called at process creation time. It requires as input a list of all local file names referenced by the program that will be executed by the process

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

(these are supplied by the compiler as part of the program's code stream), and the actual names of the associated files (which must be supplied either in a form of JCL (as in OS/360), by operator input, or by referring to a directory set up at system initiation or major mission phase time). CREATEFCB inserts a File Control Block (FCB) into the process' PIA for each referenced file. The FCB becomes the object of all file references made by the process. It contains the following fields:

LOCAL NAME
ACTUAL NAME
OPENED
FILE ADDRESS
RECORD POINTER

Figure 3.5-3 File Control Block

CREATEFCB writes the local and actual file names into the appropriate fields, allocates space for the remaining parts of the FCB, and leaves them with a null indication. Their significance is discussed next.

b) OPENFILE: This procedure provides the link between the actual file name and its physical location on an I/O device. OPENFILE is performed as part of the name management function (see Section 3.4.8). OPENFILE then marks the field "opened" appropriately, fills in the actual file address, and zeros the Record Pointer. This last field is set whenever a specific portion of the file is accessed by a READ or WRITE statement, since files are usually accessed one logical record at a time, and a logical record does not, as a rule, coincide with the physical record being transferred.

   Provided a file is certain of being accessed it may be desirable to open it just as soon as its FCB is created, in order to save overhead at READ time. In this case CREATEFCB and OPENFILE are called sequentially by the process creation procedure (see Section 3.2.). However, a file will normally be opened when it is first read so that if the program never progresses.

to the READ and WRITE statements, time will not be consumed in searching directories and space establishing links. Subsequent "READS" after the first will find the FCB marked "opened", and will ignore the call to OPENFILE.

There are two basic I/O statements: READ and WRITE. These are of the form READ/WRITE (local file name, logical record number, logical record length, action, return), and both invoke the major I/O procedure:

c) REQUESTIO: This procedure references the FCB in the process' stack to check whether the file has been opened. If not, it calls OPENFILE. It then calls Memory Management to allocate space of sufficient size in M2 to accommodate the desired record. REQUESTIO now calls FORMIOCB, which constructs the IOCB appropriate to the I/O request (see Section 3.5.2.2). Subsequently, procedure SETUPQ is called to insert the IOCB into the DEVICEQ by setting the link field in the IOCW. If the specified DEVICEQ is found to contain only completed and/or inactive I/O requests, as signified by the control field of the IOCW, the procedure STARTIO is called. STARTIO is the procedure that instructs the IOC to start executing a new sequence of I/O requests. It accomplishes this by initiating a processor to IOC communication over the Inter Processor Communication bus (see Section 5.1.4). It forms the M2 address of the first IOCW and sends it to the IOC together with an accompanying command to interpret the IOCW as the start of a new DEVICEQ. This done, STARTIO returns to REQUESTIO. If SETUPQ found the current DEVICEQ to be still not completely serviced by the IOC, it returns to REQUESTIO. REQUESTIO now performs the final phase of I/O request initiation. The READ (WRITE) statement indicates (via the "action" item in its argument) how the ACTION field of the IOCB is to be set, and REQUESTIO sets it accordingly to inform the IOFINISHD procedure what to do at I/O completion. REQUESTIO then, depending on the presence of the default argument "return" in the READ (WRITE) statement, returns control to the initiating process. If no "return" is indicated, REQUESTIO places the process into the WAIT state, puts its ID into the ACTION field of the IOCB, and passes control to the scheduler.

The IOC now reads the first IOCW of a new DEVICEQ as instructed, marks its control field "accepted, decodes it, and initiates its own microprogrammed sequences to accomplish the intention of channel and device commands. The IOC maintains the M2 address of the current IOCW so

that it may set the appropriate Control Fields in the IOCW at completion and read the "next" subfield for the next IOCW in sequence. If "next" indicates there is none, the IOC issues a "DEVICEQ empty" interrupt, and eliminates the current DEVICEQ from its internal control mechanism. New requests for this device must perform STARTIO before the IOC will acknowledge them. At the completion of requests that do not suppress interrupts, and at "DEVICEQ empty", the IOC selects a processor to enter the other major I/O procedure, IOFINISHD.

d) <u>IOFINISHD</u>: This procedure identifies the IOCB corresponding to a given "I/O complete" interrupt from the M2 address of the IOCW that the IOC sends as part of the interrupt signal via the interprocessor bus. IOFINISHD first checks whether the completion was without errors, as indicated by a null "error" field in the IOCW. If an error is detected, IOFINISHD calls the IOERROR procedure, which interrogates the status field to isolate the source and type of error. Subsequent actions depend strongly on a specific implementation, and will not be discussed further. IOFINISHD may then do one or both of the following:

   a) Transfer the process identified in the ACTION field of the IOCB from the WAIT state to the READY state.

   b) Set the event specified in the ACTION field, and call the EVENT HANDLER procedure.

   Once these are accomplished, IOFINISHD declares the I/O request terminated by setting the Active subfield of its IOCW to inactive. This frees the space occupied by the IOCB. IOFINISHD next removes the IOCB from the DEVICEQ by re-linking the remaining IOCB's. Finally, IOFINISHD calls the Absent Segment Trap Handler. This procedure has the special capability of scanning the DEVICEQ at any time, for completed "quiet" segment transfers. (These are identified in the ACTION field of their corresponding IOCBs.) The ASTHANDLER indicates its response by setting the Active field of the corresponding IOCWs to inactive. In the special case of a "DEVICEQ empty" interrupt (which may coincide with a regular I/O complete interrupt), IOFINISHD itself performs

a scan of the DEVICEQ to search for "quietly completed" I/O requests that were not initiated by the ASTHANDLER. It expedites these as instructed in the ACTION fields, and sets their IOCWs to indicate inactive. It is up to the initiator of such requests to ensure adequate response to "quiet completions" when the "DEVICEQ empty" interrupt occurs infrequently.

3.5.3 Data Bus I/O

Data transfers to and from the avionics data bus differ in character from those to the other, more conventional, I/O peripherals as follows:

a) It is expected that a significant portion of the data will be sampled at fairly high frequencies: from 50 to 100 times per second.

b) Response times for the data requests are short. Depending on the type and frequency of the bus, delays of only tens of microseconds can be achieved. This makes the bus look more like a slow operating memory module than a rotating magnetic peripheral device.

c) Relatively small amounts of data are expected to be transferred for each bus I/O request: the typical example of Section 3.5.1 defined a range of from 1 to 128 bytes.

These characteristics can, of course, change considerably with the degree of local processing that is done at the data bus device terminal. If much of the high frequency computation is performed at the device, it is likely that the bus traffic resulting from communications between device and control computer will become less periodic and of longer duration when it does occur. Such a trend would make the need to accommodate the above capabilities a) through c) in the design of the multiprocessor executive less essential. However, the following discussions is based on the assumption that a) through c) above do represent a significant proportion of data bus I/O.

3.5.3.1 <u>Data Bus I/O Handling</u>. The requirements for data bus I/O will be divided into two categories:

a) Repetitive

b) Single requests

This is a simplification, because the kinds of I/O communication on the bus, as pointed out in Section 3.5.1, span the gammut from highly repetitive to once only. However, only in a specific implementation can it be judged worthwhile to consider more than the above two classes of bus I/O requests. Even so, it cannot be established uniquely where the line between the two categories may be drawn. At some frequency repetitive requests must be deemed to be infrequent enough to quality as single events: perhaps once a second is a reasonable frequency.

For repetitive I/O, the IOC will execute a set of requests continuously, at a repetition rate that is specified by the application. It will continuously update (or write out) the contents of M2 locations specified for each request. It will continue to do this until instructed otherwise over the IPCB.

Single requests to the bus will be handled very much like the conventional I/O requests described in the previous sections. The executive I/O procedures will assemble a data bus I/O control block and inform the IOC over the IPCB of where in M2 to pick up the single request. The IOC then makes a temporary insertion into one of its repetitive request tables, and, once it has been executed, removes it. The IOC signals the completion of the single event with a hardware "I/O complete" interrupt, as described previously. This then is the basic policy for controlling data bus I/O operations.

3.5.3.2 Data Bus I/O Structures. A data bus I/O command word, or DBCW, takes the form illustrated below:

Control		BCU	Bus		Device	M2	
Status	Next	Op Code	Address	Command	Operand	Address	Size

Control: A field similar in fashion to that described in the previous sections on peripheral I/O. "Status" indicates the status of the current request, and "next" where the following DBCW is to be found.

Bus: This field specifies the device address, and bus command (e.g., read, write, set mode, etc.).

Device: Device operand field specifying operation to be performed by specific avionics subsystem (interpretation known only to device).

M2: Destination field - address and length of M2 area in which result of bus I/O is to be placed, or from which output is to be taken.

A number of DBCW's are assembled into a table of requests and the whole table is executed sequentially every ΔT seconds, where ΔT is a parameter passed to the IOC by the initial IPC instruction to start executing the table.

Other tables correspond to groups of requests to be serviced at different periods are also assembled. The structure of a complete set of bus requests to be performed at predetermined intervals $\Delta T1$ through $\Delta T6$ is illustrated in Figure 3.5-4.

ΔT1 ΔT2 ΔT3 ΔT4 ΔT5 ΔT6

DBCW1

DBCW2

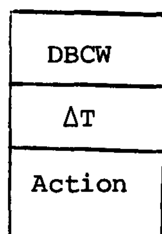
DBCWn

Figure 3.5-4 Data Bus Control Table

The basic difference between the structure of these tables and the DEVICEQ's is that they are associated with a given execution interval ΔT, and not with a particular device. The relationship between the ΔTs may very well (but not necessarily) be powers of two times the basic interval for a given "minor cycle". The largest ΔT corresponds to the period of the "major cycle" during which all requests in the set of all tables have been executed at least once. The establishment of the ΔTs, and the population density of each table is a non-trivial problem of data bus system design. It is important not to exceed the maximum data bus duty cycle. Single data bus I/O requests are based on the same DBCW format, but are handled in a different manner, as described below.

3.5.3.3 Data Bus I/O Procedures. The principal task of the executive software is to construct DBCWs and to assemble them into tables for IOC execution. Most of this must be accomplished at the time a process that expects to perform data bus access is created, so that the frequent BUSREAD or BUSWRITE operation can remain fairly straightforward. At process creation time a Data Bus Control Block or DBCB, is established in the process' PIA for every declared data bus access, by procedure SETUPBUS.

a) SETUPBUS: This procedure constructs a DBCB. The DBCB contains the following instruction fields:



DBCW: This is as described in the previous section.

ΔT: This is the expected update cycle period.

<u>Action</u>: Whether the request is repetitive or single.

SETUPBUS, which is called by all procedures that declare intention to perform bus I/O, then assembles all DBCW's into tables corresponding to the specific execution cycle periods, ΔT. It then enters a procedure to instruct the IOC:

a) Where each table resides in M2.

b) The ΔT associated with each table.

c) An instruction to start servicing the table.

The IOC then proceeds to access DBCWs sequentially and to execute them, placing the results in the specified M2 locations. It continues to do this until instructed to stop either

a) immediately,

b) at the end of the next minor cycle,

or

c) at the end of the next major cycle

by means of an IPC instruction. A reason for halting the IOC may be

a) An error condition has been detected by data bus application software.

b) A DBCW is to be changed or inserted.

c) A set of tables is to be deleted or added due to a major mission phase change.

b) <u>READBUS</u>: This procedure is entered at bus access time, via the BUSREAD(WRITE) statement. Its main function is to check on the validity of the data, which is accessed by reference through the corresponding DBCB in the process' stack base. In the asynchronous access technique here described, the "updatedness" of the data

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

is not known implicitly in cases where several pieces of data that have been separately requested are to be read as a group. It is important, therefore, to ascertain that all members of the group are updated during the same ΔT cycle. This is achieved by associating each data item with an "update cycle" field. Only when all members of a group contain the same value for this field is the group homogeneous. For a given update table this field can be as simple as a single bit.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

3.6 Timing and Synchronization

This section presents two topics that both have to do with the real-time control aspects of the multiprocessor operations: the organization and use of the central system timer mechanism, and the synchronization of concurrent processes.

3.6.1 Timer Management

The system timer incorporates a register which contains system time scaled in microseconds (approximately: see below) and measured from a fixed epoch. The fifty-one mantissa bits in the floating-point format of the machine double word can exactly represent, to a microsecond, any instant of time over a seventy year period.

The timer register is not modified under program control except for initialization. Consequently, it can be used as a source of numbers which are guaranteed to be unique throughout the lifetime of the system, inasmuch as accesses of its contents cannot occur more closely spaced than a microsecond. At a slight increase in circuitry, accesses could be limited to a larger minimum spacing (by delaying, when necessary, the response to a particular form of timer-value request), thereby reducing the field required to assure uniqueness of accessed values. For example, if values were delivered no more often than once per millisecond, a pattern of 38 bits from the middle of the clock would not recur for almost nine years.

(An alternate implementation of unique-name generation can be provided by devising a unique-name register. When accessed, the value from this register is delivered to the requestor, and that value plus one is stored back in the register. No command would be capable of modifying the value in any other way. The number of bits needed to express N unique id's is obviously $\log_2 N$ if N is a power of two.)

In addition to its use as a clock and as a possible source of unique bit-patterns carrying a time-tag, the timer is instrumented so that it may be used to signal an interrupt at a specified time. For this purpose, a clock interrupt register is provided which parallels the timer, bit for bit, in its middle range (refer to figure 3.6-1). Each time bit 10 of the clock register receives a carry-in during incrementation (which happens every millisecond), the content of the clock interrupt register is compared with the corresponding bits of the clock register. When a match is found, a timer interrupt is triggered, causing invocation of the system procedure which administers usage of the timer.

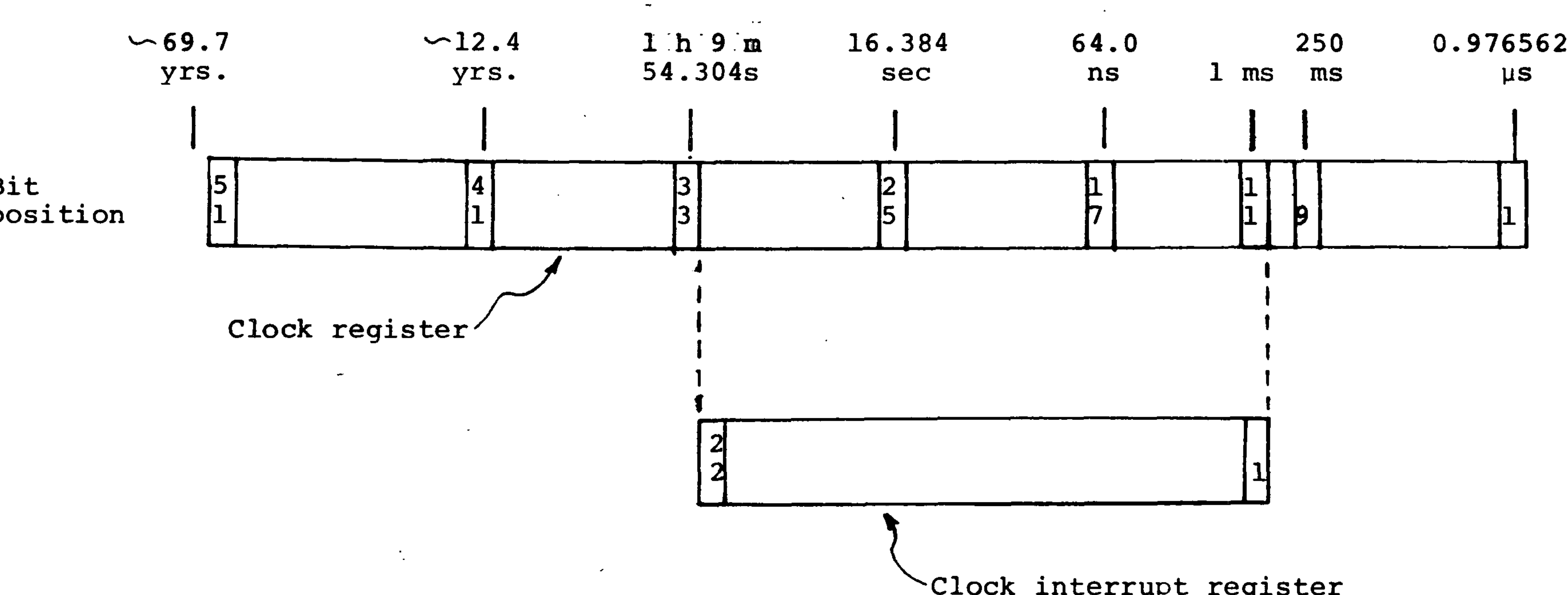


Figure 3.6-1: Organization of Real-time Clock Register and Clock Interrupt Register

The choice of mid-range bits to be matched is somewhat arbitrary. On one hand, there is a desire to minimize the hardware; on the other, generality and precision questions arise. The twenty-two bit span of the illustrated configuration was chosen on the basis that: 1) timer-interrupt spacing less than a millisecond was very likely to exceed the response capability of the timer-handling software, and 2) it appears relatively unlikely that the earliest time awaited by any process in the system will be farther into the future than an hour from present time.

3.6.1.1 Timer Primitive Operations: To schedule an action to be performed at a specified time, one of the operating system primitives

WAIT(t)

SET(E, t)

RESET(E, t)

RESUME (processid, t)

is executed by a process. The WAIT operation places the process into the wait state until the specified time. SET and RESET cause the specified event E to be placed into the set or reset state at the given time (refer to section 3.6.2 on event handling for a description of SET and RESET actions). RESUME is used to remove the stopped-state indication from the designated process at a specified time.

The SET, RESET, and RESUME operators return a value containing a unique identification (id) associated with that execution of the primitive, and a pointer to the cell created in the timer queue area. This value (pair) can be used as the argument of the

PURGE(value)

operation, which cancels the effect of a pending SET, RESET, or RESUME.

3.6.1.2 Timer Management Data Structure: Pending actions to be performed under timer control may be stored in a queue, arranged in order by due-time. Because requests for timer-controlled actions are not necessarily submitted in the same sequence as their actions will be performed, a threaded or linked list structure is preferred to a physically-ordered one. This queue is

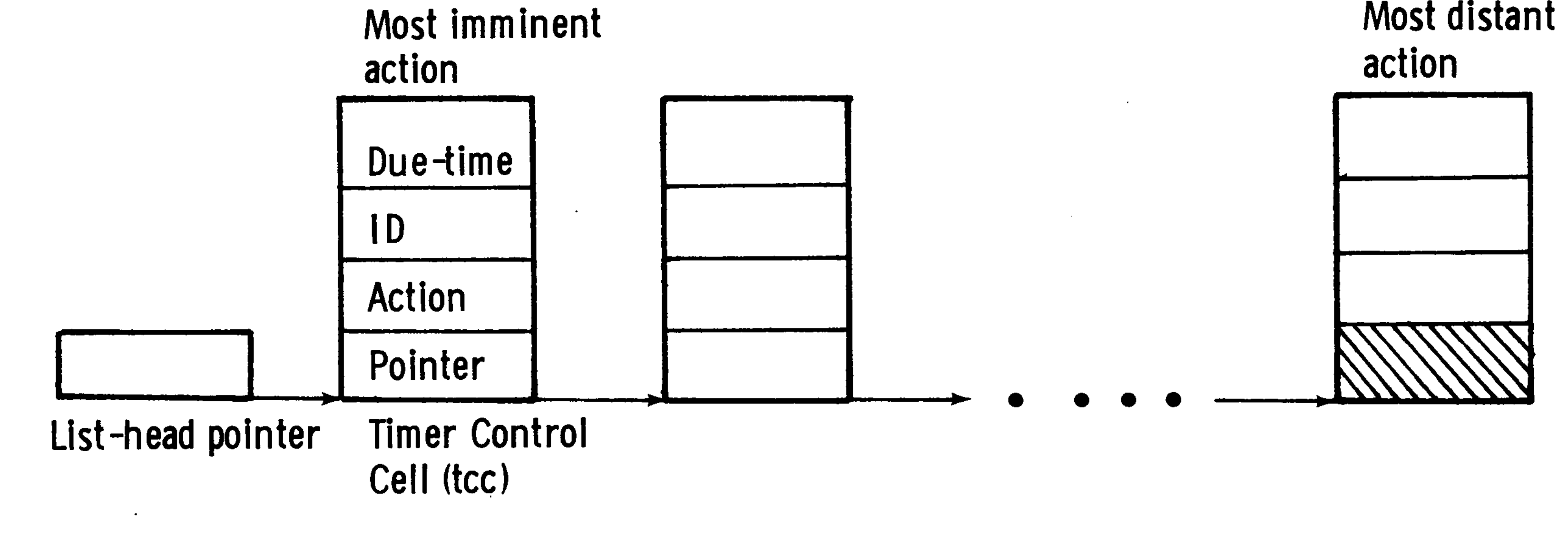
stored in a data segment whose size determines the maximum number of active requests which can be handled. An overflow segment may be used to store the least imminent requests if the primary area content approaches its capacity. The use of these segments is described below in the section on implementation.

The operation of the timer-request handler is briefly described here to motivate the introduction of an alternate data structure. When a request is received, the request handler obtains space for a queue element from the queue data segment. (The format of a queue element, called a timer control cell or tcc, and the structure of the queue are shown in Figure 3.6-2.) Next, the pair of tcc's whose due-times bracket the due-time of the new request must be located. Because the physical storage locations of the tcc's are essentially random, only the pointer field contains information related to logical order. (Any scheme which reorders tcc's physically when a new tcc is added is rejected for reasons of efficiency.) Thus, it is necessary to search the list from its head (found via a pointer stored in a fixed location) until a due-time later than the newly requested one is encountered. If the list contains an average of p tcc's, this number of accesses may be deemed undesirable if p is large, and an alternate scheme is proposed which alleviates this problem. Which form of implementation is appropriate depends almost exclusively upon the nature of the application workload on the computer.

The alternate data structure is a group of m queues, rather than one. If timer requests are uniformly distributed over the finite interval spanned by the queue-group (mT; see below), then the average search-time is reduced by a factor of m.

The m-queue structure is a generalization of the single queue described above. Rather than a single list-head pointer, the alternate structure uses m such words, referred to as index words. The value of m is chosen to be an integral power of two, for efficiency on a binary machine.

The list of pending actions is correspondingly divided into m sections, each of which contains actions scheduled for performance within a particular interval of time of length T. (The value of T is measured in timer units of one millisecond each, and is also an integral power of two.) For purposes of illustration, suppose m has the value 8, and T is 512. Then if the current time is t and the integer part of t/T (represented by [t/T]) is called t_0, figure 3.6-3 shows the time intervals represented by the 8 list sections, where $t' = t - t_0$. The nature of the index array assignment strategy may be visualized by observing what happens as t' grows until it exceeds T, assuming t_0 is held fixed. The index word, which represented the interval



Timer Primitives

WAIT(t)

RESUME(processid, t)

SET(E t)

RESET(E, t)

PURGE(value)

Figure 3.6-2 Single Timer Queue Structure

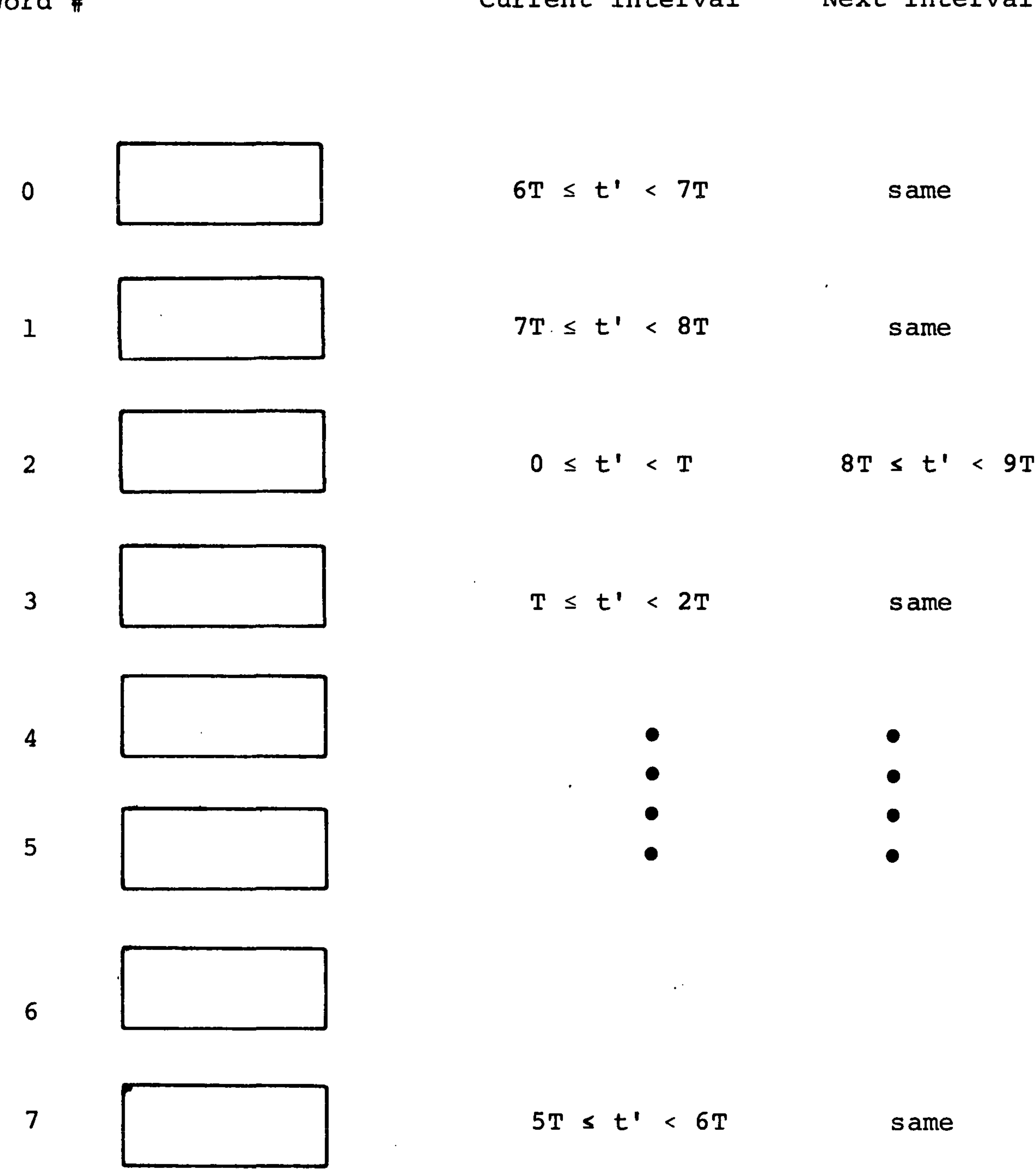


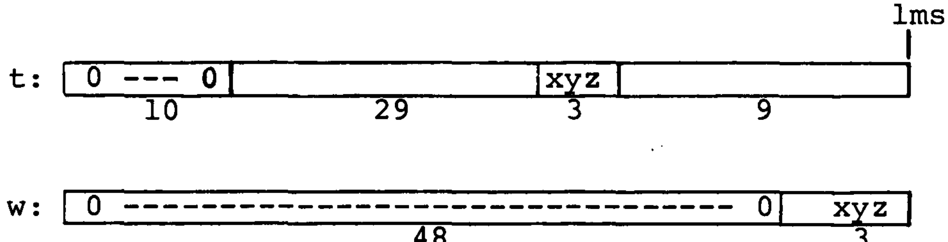
Figure 3.6-3: m-Queue Timer Data Structure

$0 \le t' < T$ is now no longer needed, since current time has advanced to a value beyond the end of the interval; the freed index word is therefore assigned to the first interval not represented: namely, $8T \le t' < 9T$. This assignment is implicit, not explicit, in that no software or hardware operations are required. Description of the list-operations will make this clear.

Another illustration of the assignments for index words is provided in figure 3.6-4. The horizontal lines represent time; three current times are shown by the arrows. The numbers of the index words assigned to the intervals are shown at the tick marks denoting the beginnings of the intervals. As real time increases and passes a tick mark, the index word assigned to the interval just completed is implicitly reassigned to the earliest unassigned future interval. Thus, intervals represented by the same index word at different times begin at points which are mT units apart in time (in the example case, 8 x 512 = 4096). If the initial assignment of index words caused word 0 to be associated with the interval which began at time zero, then the word number associated with any time t may be computed from:

$$w = [t/T] - [[t/T]/m]\ m$$

where the notation [X] means "the largest integer not larger than X". For the numbers chosen for illustration, this calculation is merely the selection of three bits ($m = 2^3$) and shifting them right by 9 places ($T = 2^9$):



Associated with each index word is a chained sub-list of scheduled actions, ordered by due-time. A pointer field in the index word points at the timer control cell (tcc) for the earliest time of interest in the interval. This tcc points to the next one, and so on. The pointer field of the last tcc in

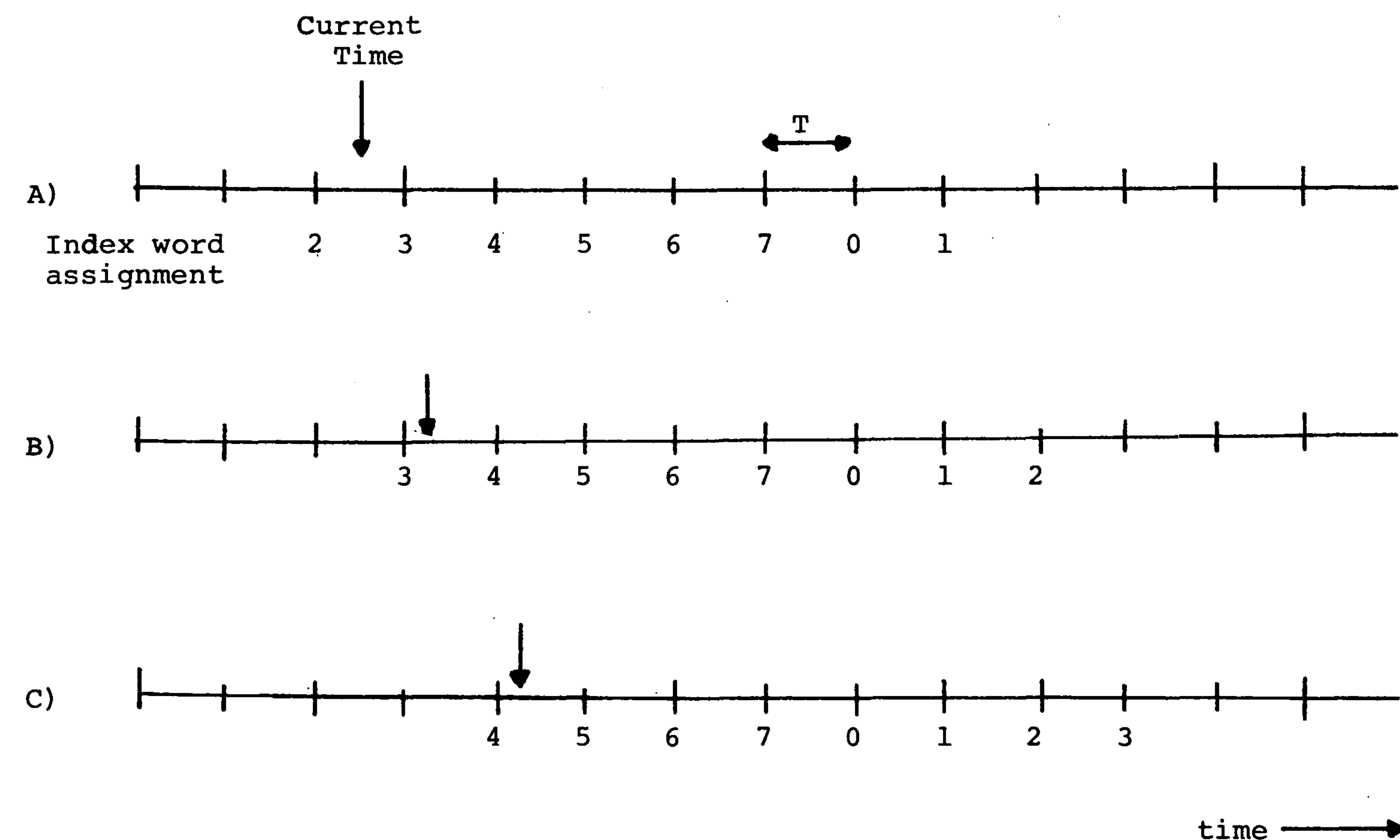


Figure 3.6-4 Assignment of Timer Index Words to Intervals of Time
(eight words are assumed)

the interval contains a null value. The formats of index words and tcc's are shown in figure 3.6-5. Because the absolute time of the represented interval is implicitly known (namely, the beginning of the interval for word w at time t is given by $([(t/T - w)/m] m + w]T)$ only as many bits as required to specify the location within the interval need be retained, (T units, representable by $\log_2 T$ bits; in the example, 9), making packing possible for conservation of space.

3.6.1.3 Implementation of Timer Primitives: The implementation of the timer primitives depends somewhat upon whether one or several timer queues are maintained; the considerations which determine the choice are mentioned above. The description in this section is couched in the terms of the more intricate structure, since it is slightly subtle. However, the single-queue may be treated as a special case of the multi-queue structure, with m=1 and $T = 2^{41}$. Certain actions, such as calculation of the index word number, would obviously be omitted for the single-queue implementation.

Invocation of one of the timer primitives causes the operating system timer-administrator procedure to be entered. For requests other than PURGE, space is obtained for a timer control cell from the timer list free space area, and the unique id is obtained and stored in the tcc along with the encoded action specification (e.g., SET-action on event E) and the due-time. The index word w is calculated as above, and the pointer which it contains is followed until either the last link in the sub-list is encountered, or the due-time of a link is found to exceed the due-time of the present request. The pointers in the new tcc and its logical predecessor are set to link the new tcc into the chain at that point, so that the sub-list remains ordered by due-time, from earliest to latest. Should the due-time of the request have caused it to be placed at the head of the list for the earliest interval, then it is necessary for the timer procedure to set this new due-time into the clock interrupt register, since the new request is for the most imminent action. Finally, the id and a pointer to the tcc are returned to the caller.

At this point, a special case for the multi-queue implementation is described. The division of time into m intervals, and the use of sub-lists associated with each interval, has been chosen to reduce, by a factor of m, the average search time required to find the correct spot in the queue for a new request. A side-effect of this implementation is that requests for response farther into the future than the end of the mth interval must be specially processed. Hence, a subsidiary list of "distant future" tcc's is kept. When the timer procedure makes this list not empty, the timer procedure also enters a

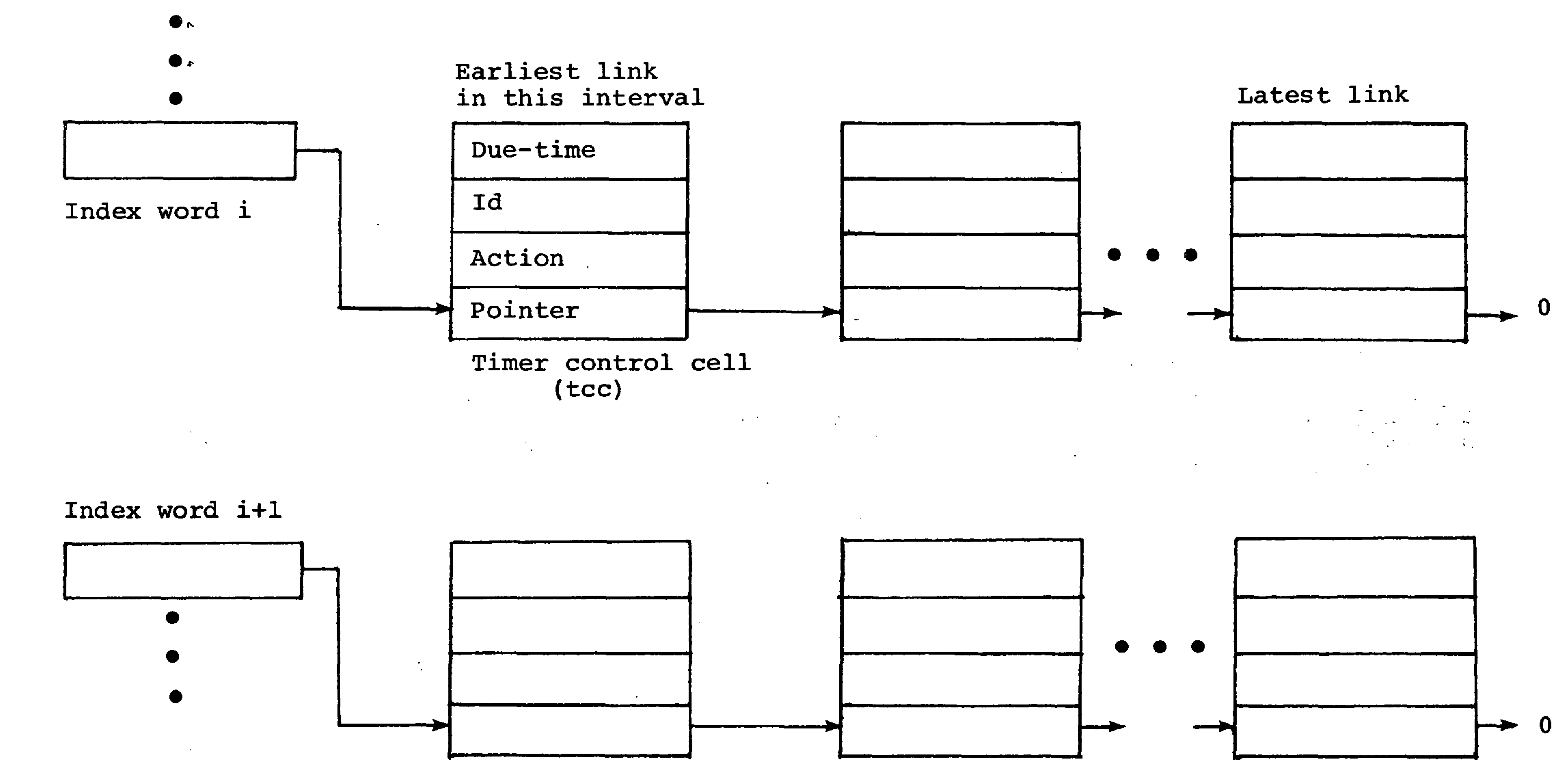


Fig. 3.6-5 Timer sub-list structure (fields are portrayed; their widths are not implied)

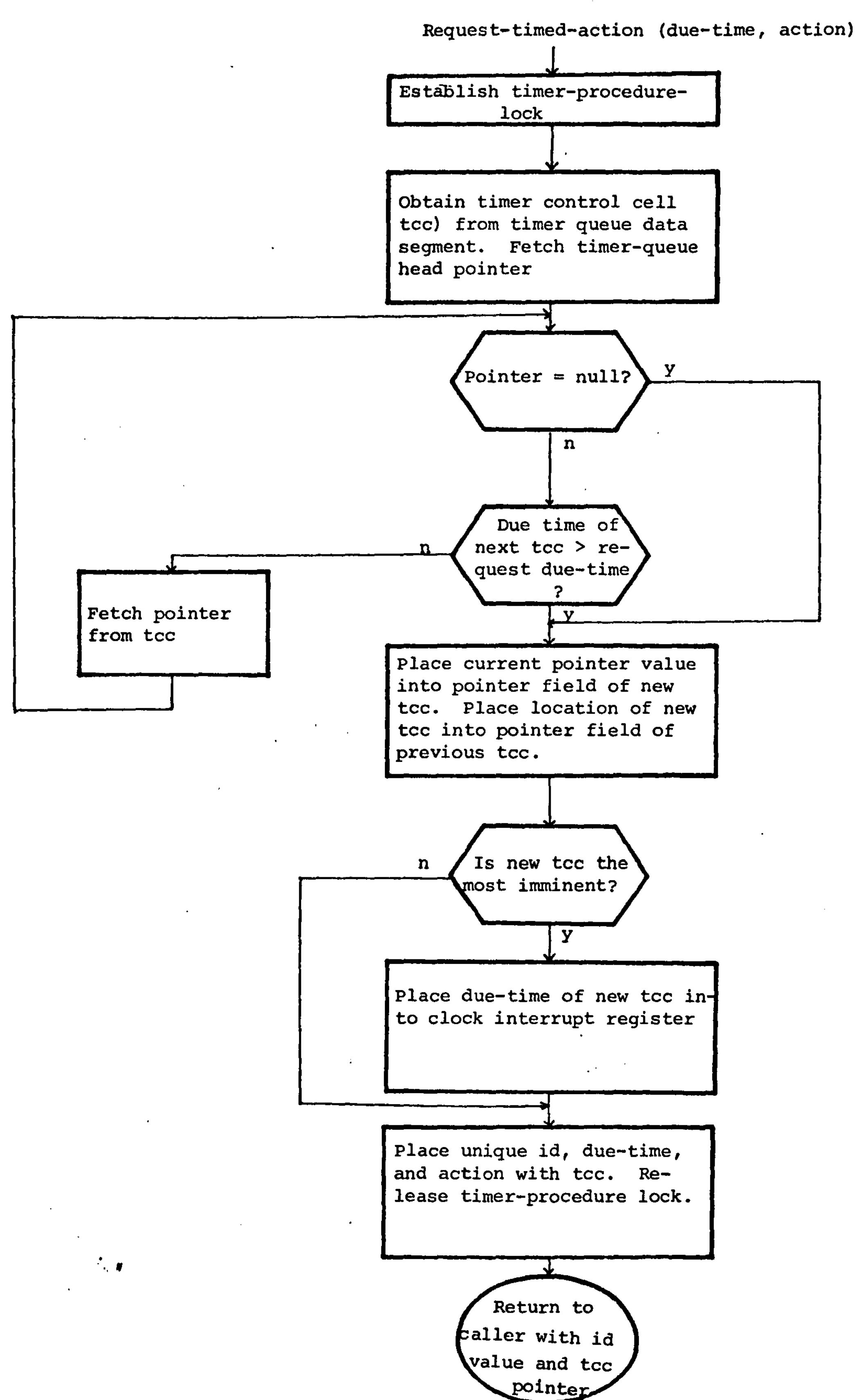


Figure 3.6-6: Timed-Action Request (Single Queue Structure)

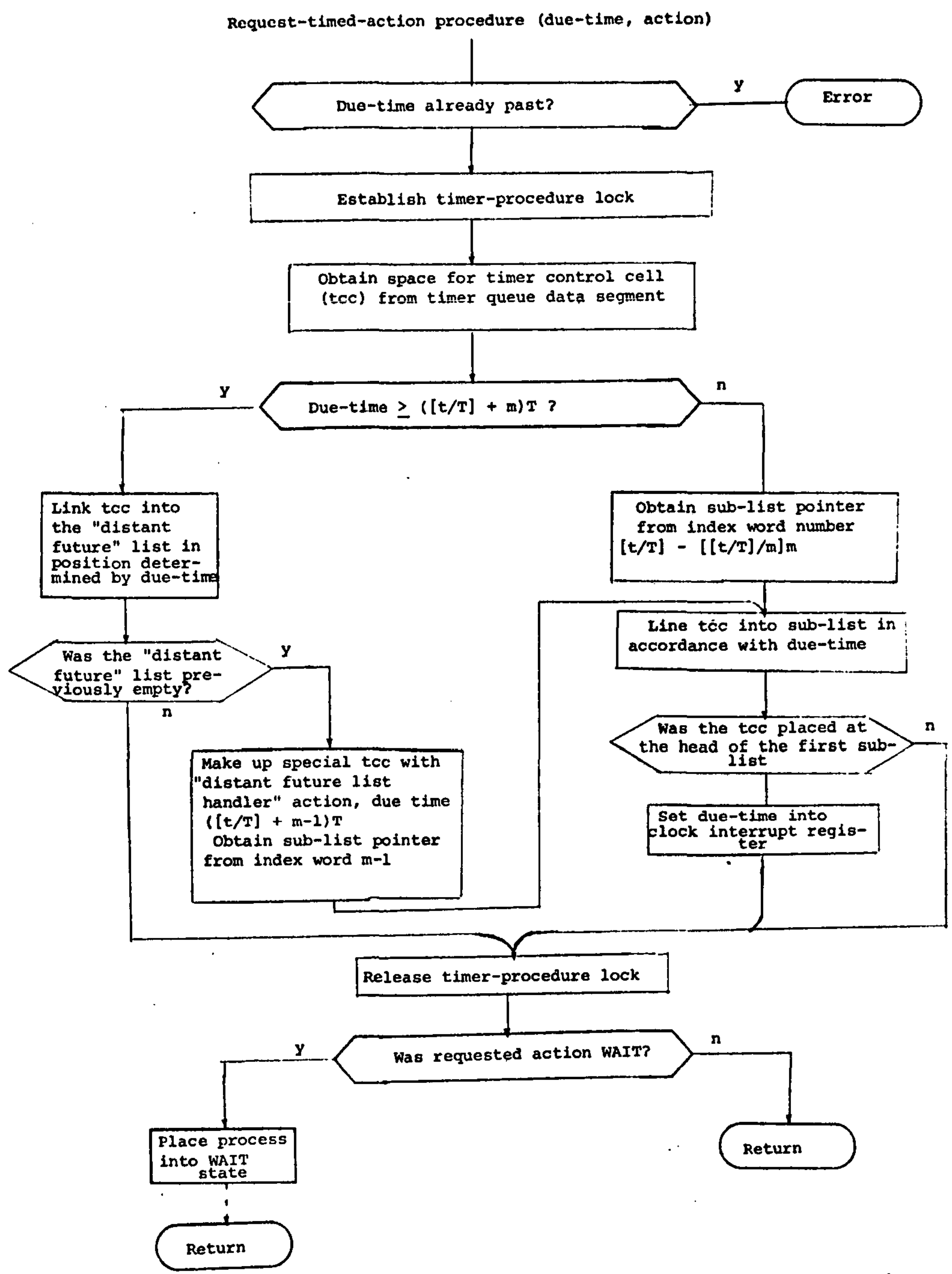


Figure 3.6-7 Timed Action Request (m-Queue Structure)

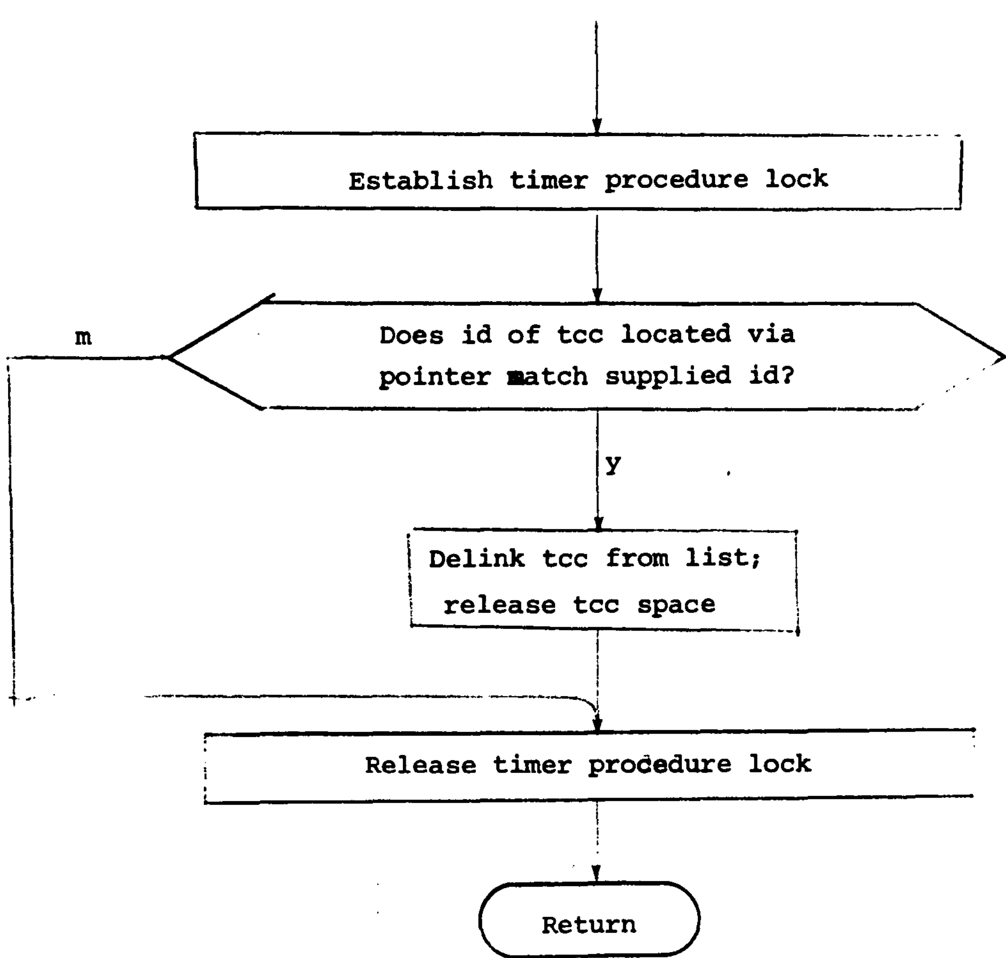


Figure 3.6-8 Timer Request Purge

request in the normal list, to come due at time ([t/T] + m - 1)T, the beginning of the latest active interval. When this request is triggered, a procedure is entered which moves those items into position which can be linked into the normal lists. If the "distant future" list remains non-empty, the procedure reschedules itself for (m - 1)T milliseconds later.

The PURGE operation consists of examining the tcc indicated by the pointer part of the argument to determine if the id in the tcc matches the id part of the argument. If not, control is returned to the caller. If so, the tcc is delinked from its list. If it occupied the head-position of the earliest sublist, the clock interrupt register is reset to correspond to the due-time of the new head tcc. Control is then returned to the caller.

The operations performed when a timer interrupt occurs are now described: The timer interrupt signifies that the due-time of the action in the head tcc has arrived; thus the timer interrupt handler begins its activity by performing the action specified in the head tcc. It then releases that tcc's space, destroys its id field, and links the zeroth index word to the subsequent tcc; next, it examines the due-time of the following tcc; if it is also currently due, that action is performed, and the next tcc is examined. This continues until a not-yet-due tcc is found; its due-time is placed into the clock interrupt register, unless the action is due farther into the future than can be expressed within the capacity of the register (4194.304 seconds, or about seventy minutes). In this unlikely case, the timer interrupt handler procedure makes a dummy action become the head of the list, and sets its due time to be 4190 seconds earlier than the due-time of the (now) second list element, or 4190 seconds from the current time, whichever is sooner.

If any of the tcc actions performed by the interrupt handler readied a waiting process, the interrupt handler calls the Process State Controller; otherwise, it exits directly.

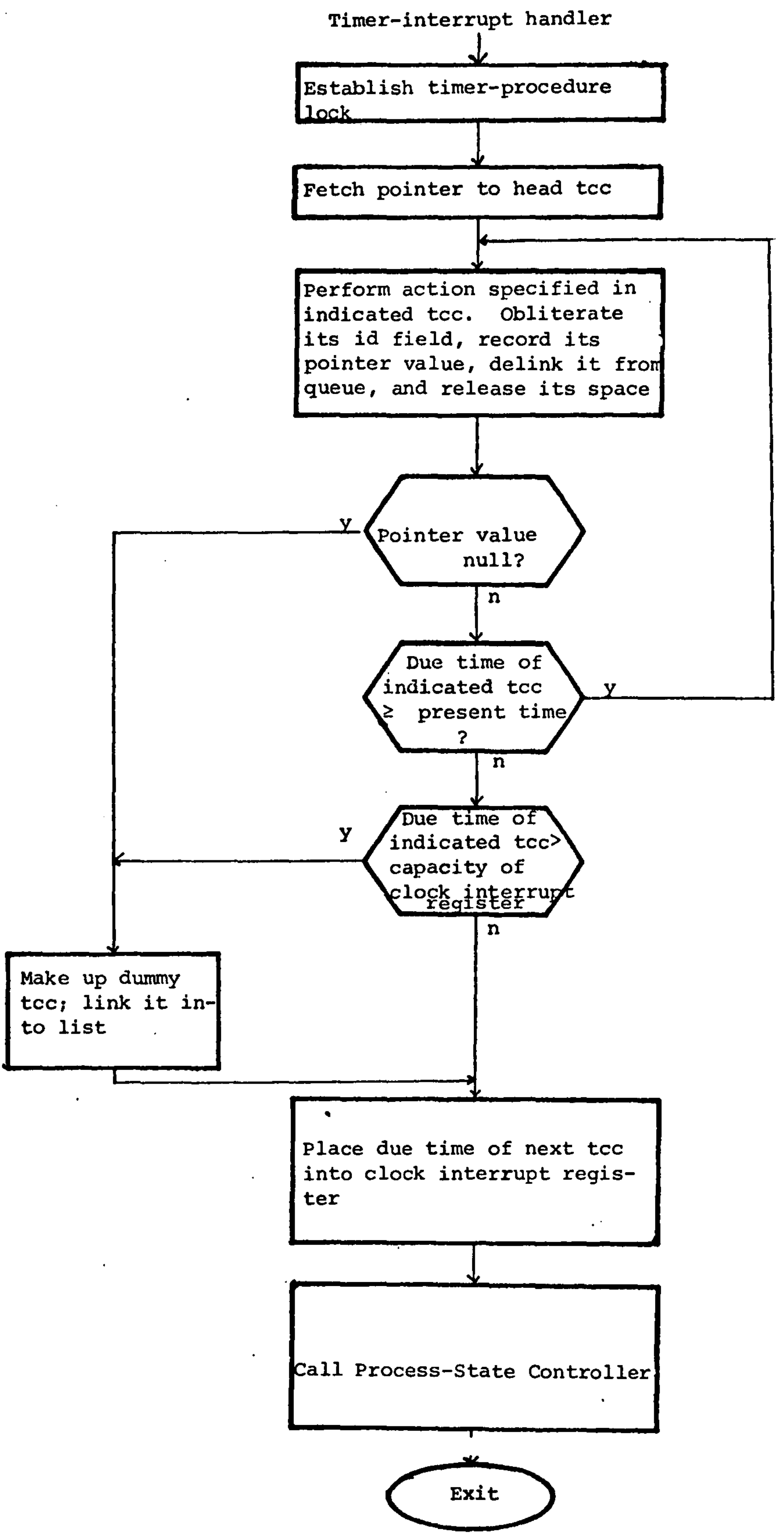


Figure 3.6-9: Timer Interrupt Handler (Single Queue Structure)

C. 4

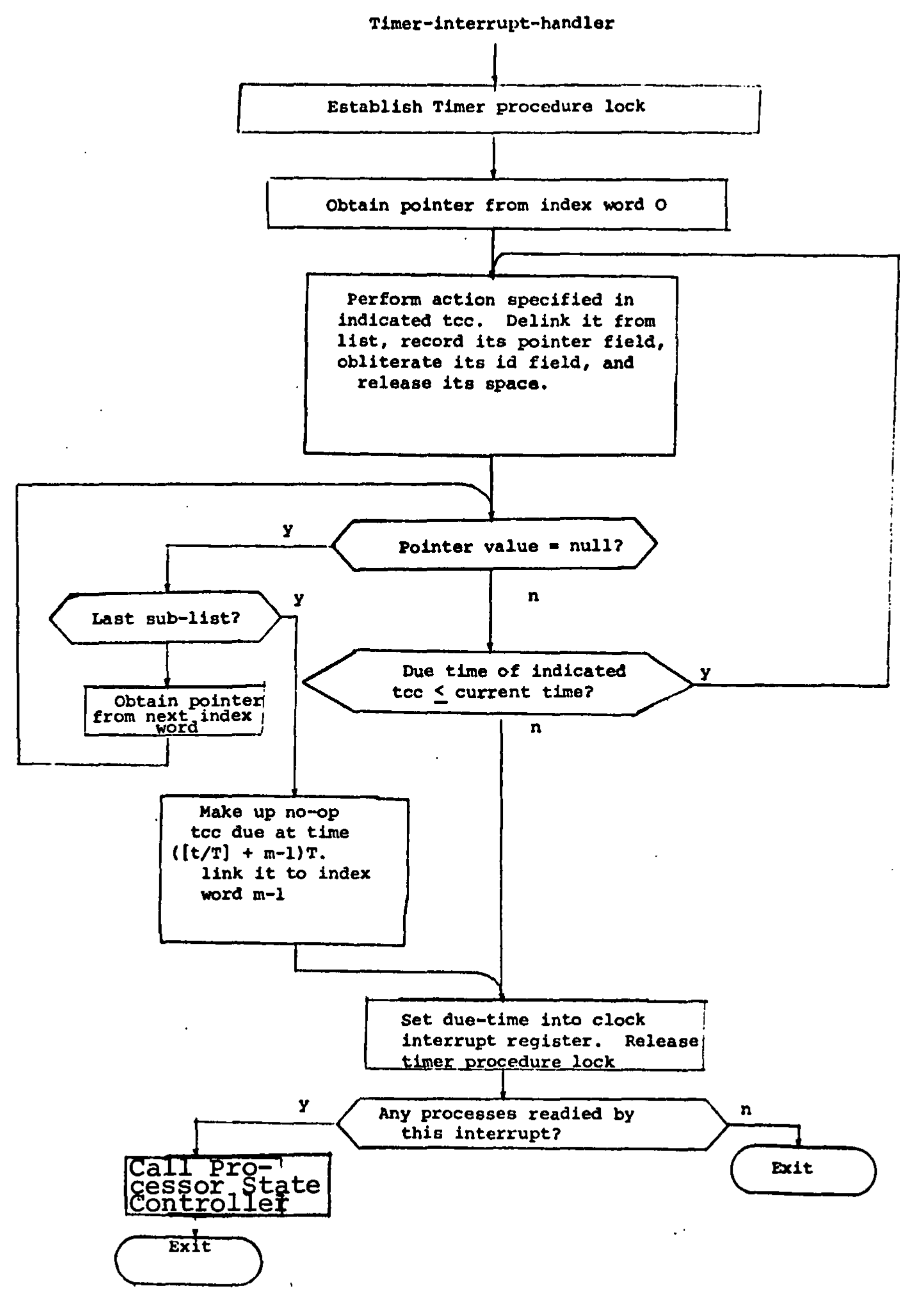


Figure 3.6-10: Timer Interrupt Handler (m-Queue Structure)

3.6.2 Events

Communication between parallel operations in the multiprocessor is facilitated by a data structure based on the concept of "occurrences" or "events", along with a set of primitive operations defined around these objects.

Consider a data element called an event-cell which contains a field indicating the state of the event-cell, i.e., whether the associated event has happened or not. Strictly speaking, the state is a two-valued variable. A second field, the event-type, describes whether the event is a "normal" event or a "pulsed" event. A normal event is one whose state field retains its set (or reset) value until altered by an appropriate primitive; a pulsed event assumes the "set" state for only a brief interval, and then returns to the "reset" state automatically, without requiring explicit application of a primitive. The usefulness of the distinction will become clear in the following discussion.

An event-cell has a third field, which is a pointer to a data element of a related type, called an event-control-cell The final data-element related to event handling is the action-cell. The components of these elements (shown in figure 3.6-11) their use is described in conjunction with the primitives associated with events.

3.6.2.1 Event-related Primitives: As mentioned, event-cells have two states. The value of the state field may be obtained by means of the primitive function

STATE(E)

where E is the name of the event cell. The value returned is a boolean true or false according to whether E is set or reset, respectively. If E is a pulsed event, STATE(E) always returns reset.

The state of an event call may be assigned by

SET(E)

RESET(E)

These operations leave the state of a normal event SET or RESET, respectively, regardless of the previous state. The RESET

State (Set/reset)
Type (Normal/pulsed)
Pointer (to event-control cell)

a) EVENT CELL

Operator pointer (to next ecc involved in this primitive, or action cell)
Event pointer (to next ecc in event chain, or event cell)

b) EVENT CONTROL CELL (ecc)

Count
Action type (WAIT, SET, etc.)
Action location (stack number, event pointer)
Operator pointer (to first ecc in the chain for this action)
Identification (id)

c) ACTION CELL

Figure 3.6-11 Event Data Structures

operation has no effect on a pulsed event; the SET operation toggles the state to the SET value and back to the RESET value immediately.

To take advantage of a system's capability for concurrent operations, it is often necessary to force synchronization at particular points. One means of accomplishing this is by use of the operating system primitive

WAIT(E)

whose effect depends upon whether the state of E has the set or reset value. If STATE(E) is set, WAIT(E) has no effect; execution of the process continues normally. If STATE(E) is reset, the process is caused to enter the wait state. It returns to the ready state only when some other process executes a SET(E), or when E becomes set by the alternate form of SET described below. An alternate form of WAIT is also provided:

WAIT(n, E1, E2, ... , Em)

where m and n are positive integers such that $1 \le n \le m$, and the Ei are names of distinct events (i.e., no event name may appear more than once in the list). The effect of execution of this operation is to place the process into the wait state unless or until any n of the m events are SET. A description of the implementation of this primitive will provide insight into its exact behavior.

Upon the execution of the WAIT operation, an operating system procedure is entered. This procedure examines the states of the members of the event list specified in the call, sequentially. Those that are in the RESET state are noted, and those which are in the SET state are counted. If the set count (call it c) reaches n, WAIT performs no further action, and control is returned immediately to the user process, which continues execution normally. If less than n set-states are found, a chain of (m-c) event-control cells is created: Each event-cell noted as above (reset state) is accessed to obtain its pointer field. For each, a new event control cell (ecc) is created, and linked into its event chain. This is accomplished by following the event chain from the event cell to the last ecc currently in the list; its event pointer (or the pointer in the event cell itself, if its list is empty) is set to point to the new ecc, and the event pointer of the new ecc is set to point to the event cell, completing a circular chain. The operator pointer in the

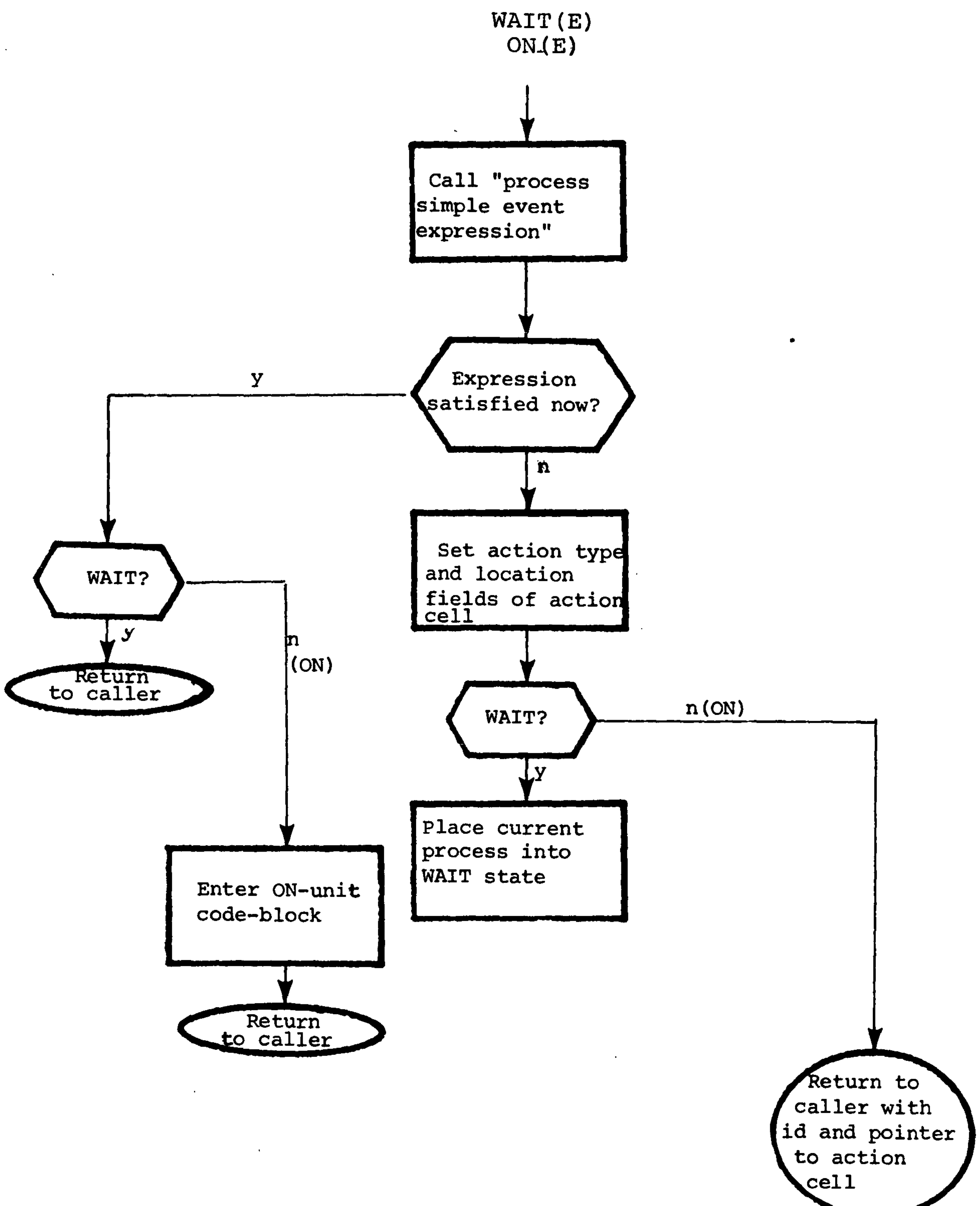


Figure 3.6-12 WAIT(E); ON(E)

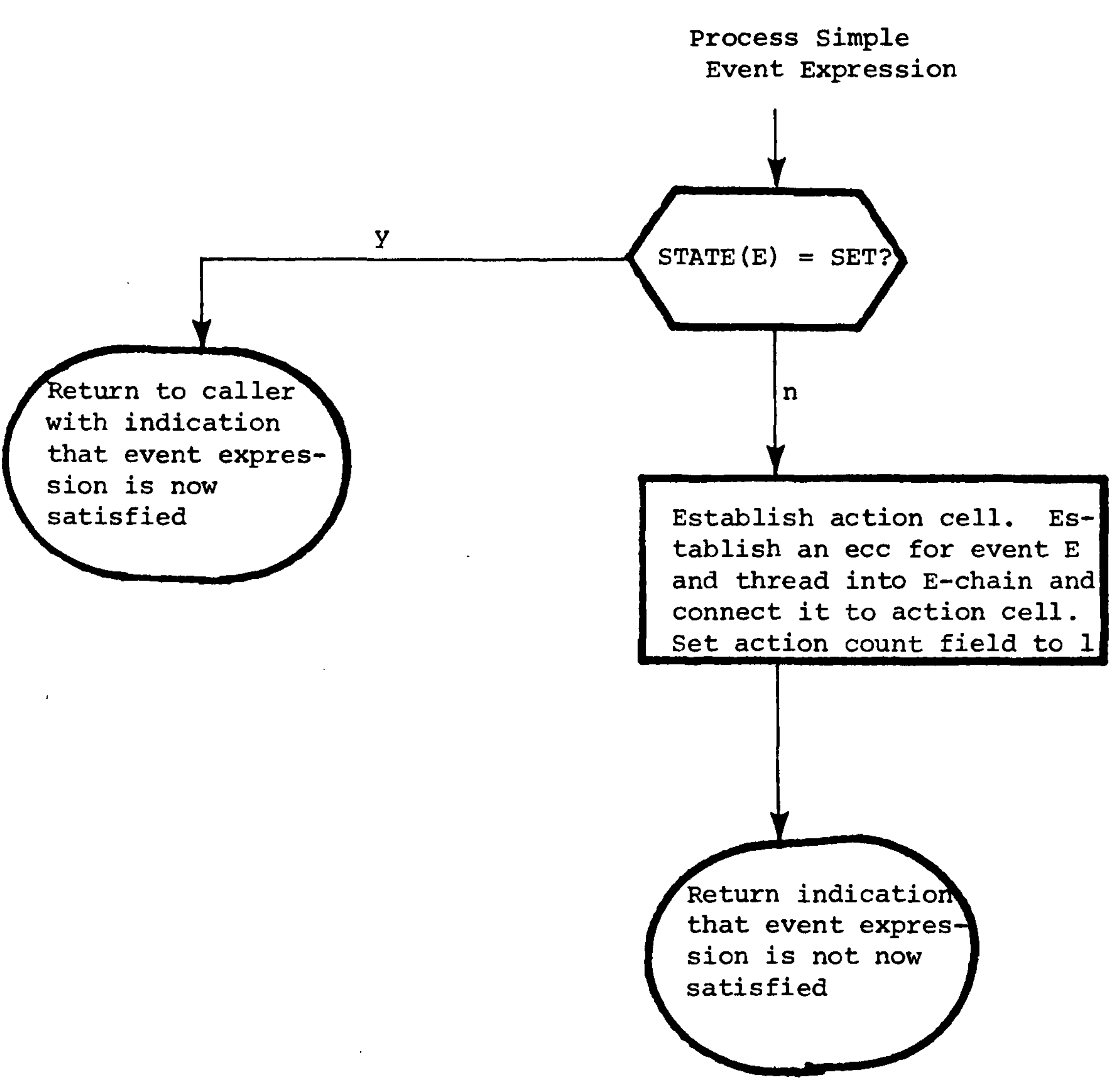


Figure 3.6-13 Simple Event Handling

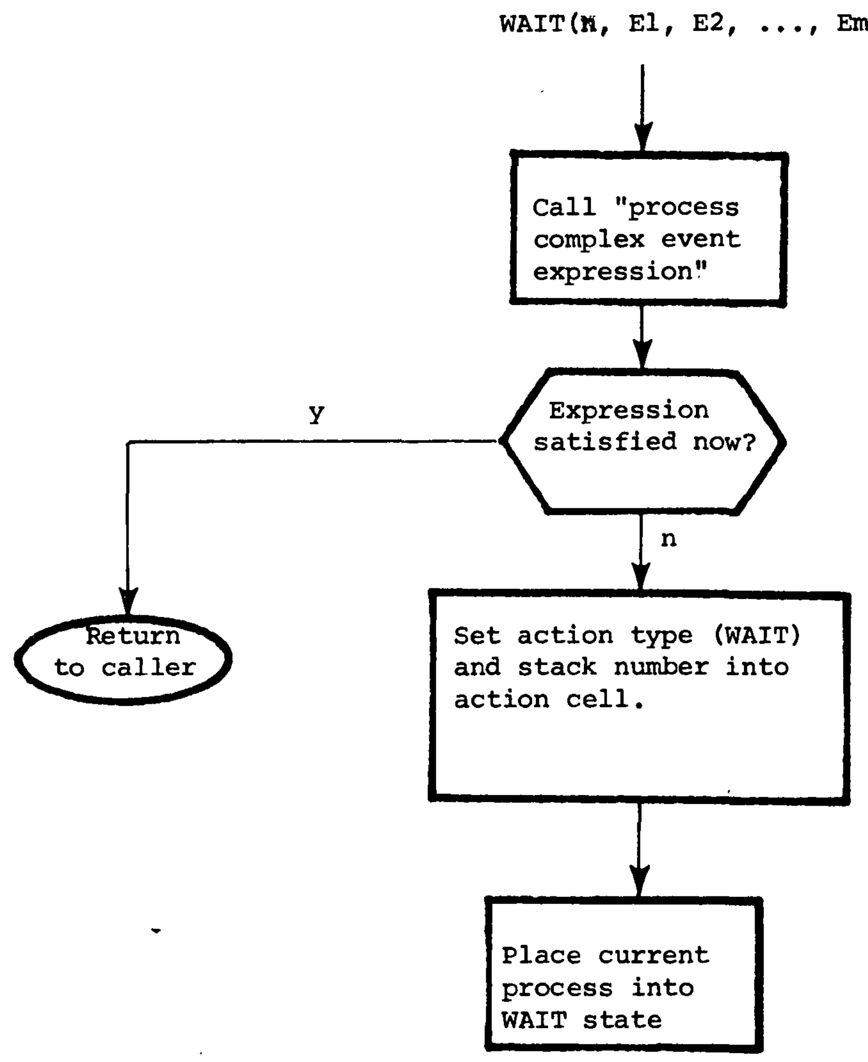


Figure 3.6-14 WAIT(n,E1,E2,...,Em)

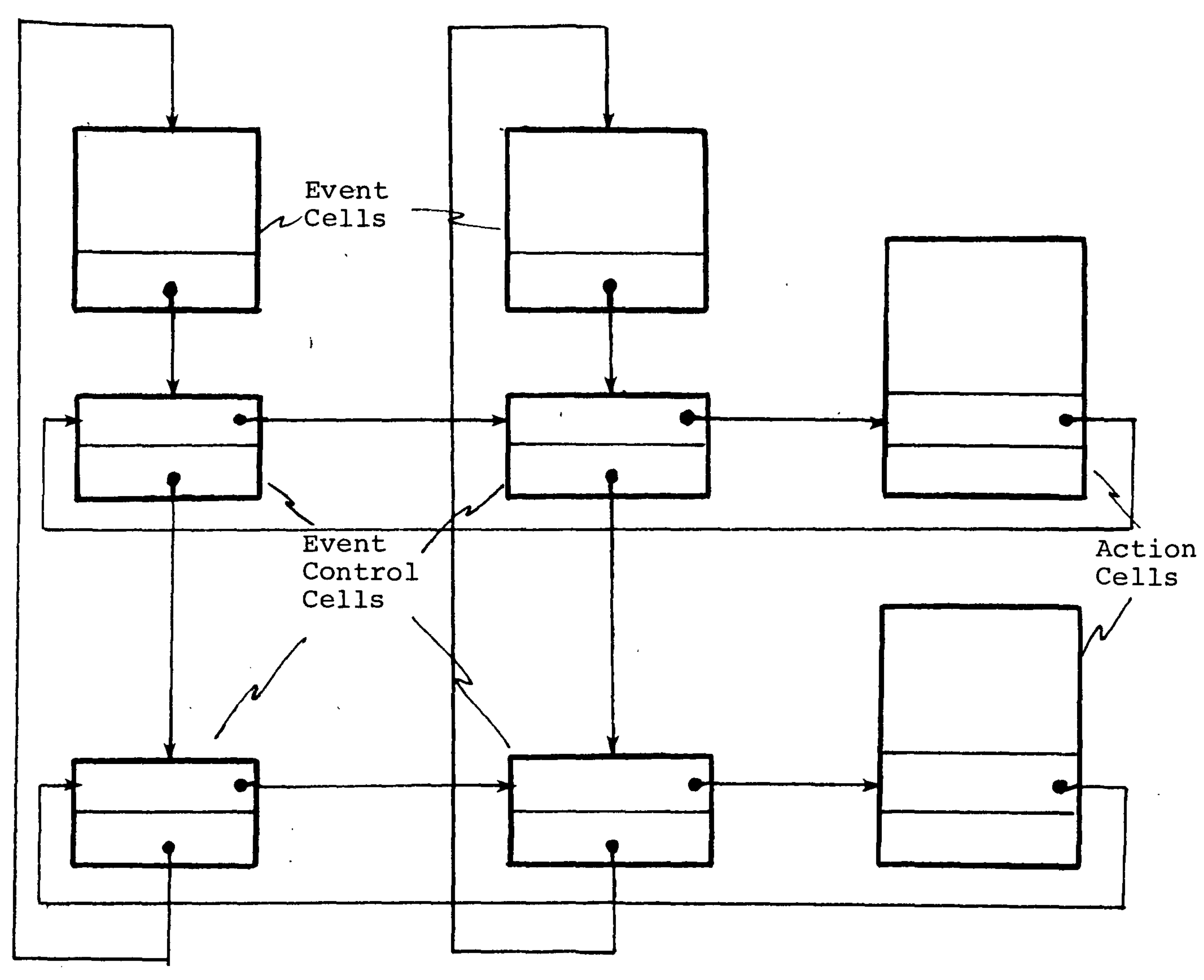


Figure 3.6-15 Organization of Event Links

new ecc is set to point to the ecc for the next member of the subset list of events which was found to be in the reset state. The operation pointer of the ecc for the last such event is set to point to the action cell created for the operation, and the operator pointer of the action cell is set to point at the first ecc for that operation. Thus a circular chain is created which links all the ecc's and the action cell for reset events related to the particular operation invocation. Figure 3.6-15 illustrates the arrangement of circular links for the case of two events and two action cells which happen to be related via four event control cells. The action cell contains a count-to-go (whose initial value is n-c, where n was specified in the operator, and c was the number of events found in the set state), the stack number for the process placed into the wait state (the field for the stack number is used differently by operators other than WAIT), and the operator pointer mentioned previously.

The WAIT procedure now places the calling process into the wait state, and transfers control to the processor assignment routine to allow a ready process to be placed into the running state.

When a process executes the SET operation, an operating system procedure is invoked. This SET procedure returns control immediately if the specified event is already in the set state. Otherwise, it examines the event type. If it is a normal event, its state is assigned the set value. If it is a pulsed event, no change is made to its state. Thus, a pulsed event is always found in the reset state, but a SET operation on a pulsed event always performs the subsequent processing functions; they are performed for a normal event only if SET causes a state transition. Pulsed events are thus useful in connection with repetitive or recurring signals; a WAIT on a pulsed event, for example, is always a WAIT for its next occurrence.

The SET procedure now operates upon the chain of ecc's associated with the event. These are accessed via the pointer in the event cell and the event pointers in the ecc's. For each such ecc, the action cell is accessed, and its count decremented by one. If the resulting count value is nonzero, it is stored back into the action cell, the ecc is released, and the next ecc is processed. (If the operator pointer in the action cell points to an ecc about to be released, its value is replaced by the value of the operator pointer from that ecc.) If the count value is zero, then enough events have been SET to cause the designated action to be performed. For example, if the operator had been WAIT, the waiting process is readied. The action cell is released, as are any ecc's belonging to the same operator which are still attached to (other) events. These are

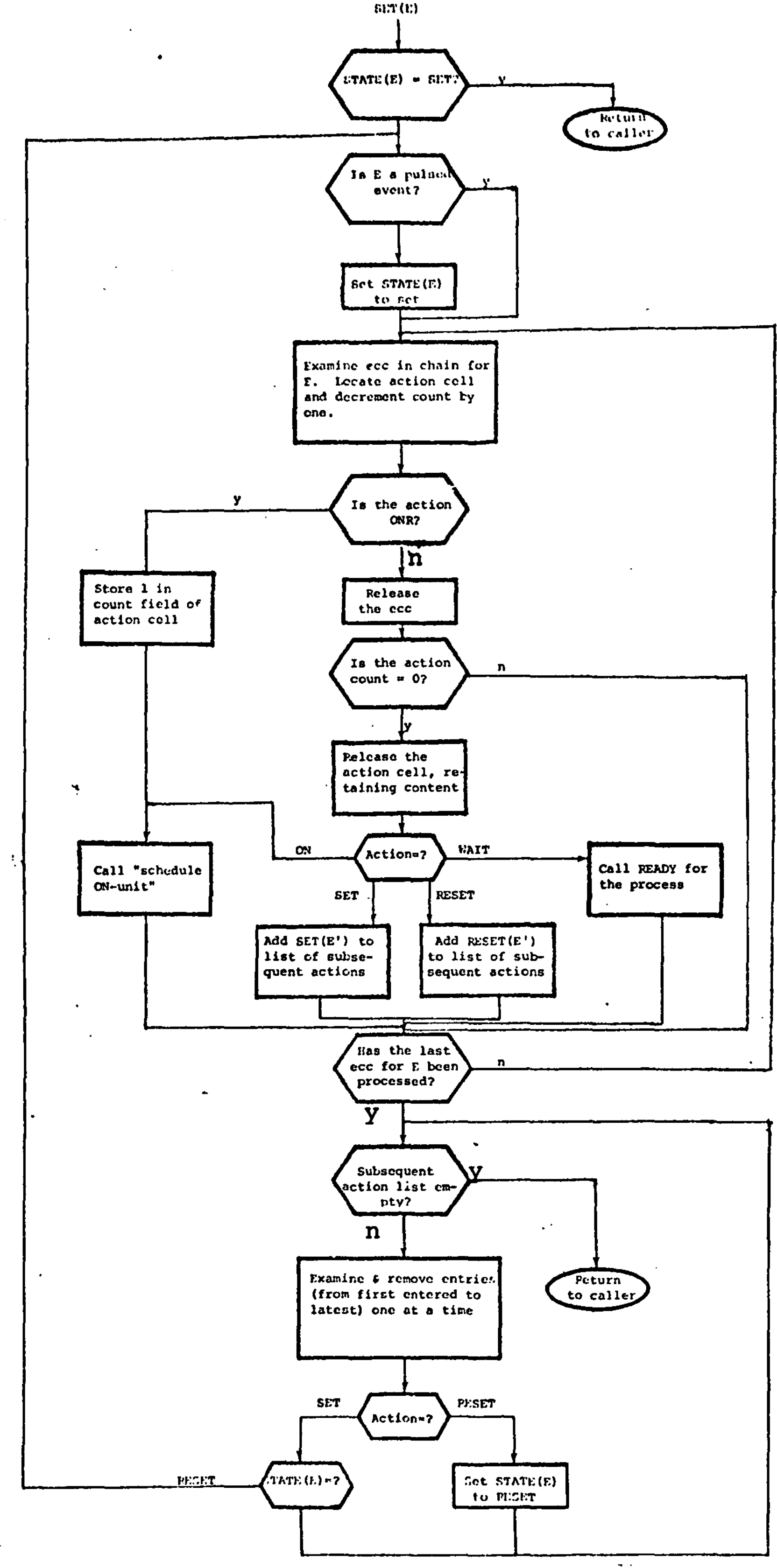


Figure 3.6-16: SET(E)

found via the ecc operator pointer. As each ecc is released, the event chain to which it belonged is linked around it.

When these operations have been performed for all ecc's for the SET event, the result is that the entire ecc chain for the event has been released (see ONR for an exception). Furthermore, if any pending operations were completed by the SET, all ecc's and the action cells for these operations have been released as well, and the actions have been performed. That is, except for subsidiary actions, such as ecc processing induced by the effect of the complex-SET primitive described below. The SET procedure returns control to the calling process directly, or via the processor assignment procedure if performed-actions readied any process. In the latter case, it may be that the priority of one or more just-readied processes entitles them to bump lower-priority processes from the running to the ready state.

The remaining primitives may now be described in terms of the above implementation. The structure of the ecc chains for

SET(E', n, E1, E2, ..., Em) $1 \le n \le m$

RESET(E', n, E1, E2, ..., Em) $1 \le n \le m$

is similar to that for the complex WAIT. Only the "action" is different; when the condition (n of m) is satisfied the set or reset value is assigned to the event E'. If this changes the state of E' from reset to set (which implies that the operating system SET procedure was in control), a record is kept. When the original call on SET completes, the further operations are sequentially performed before control is returned to the caller.

Two primitives are provided which can cause interruption of a process as a result of an event becoming SET:

ON(E) <code block>

ON(n, E1, E2, ... , Em) <code block>

Execution of the ON operator causes a structure like that for WAIT to be established, but with an action cell which points to a PEW for the code block. This code block, referred to as the "ON-unit", is not executed at the time the ON is executed; rather, it is entered at the first opportunity once the

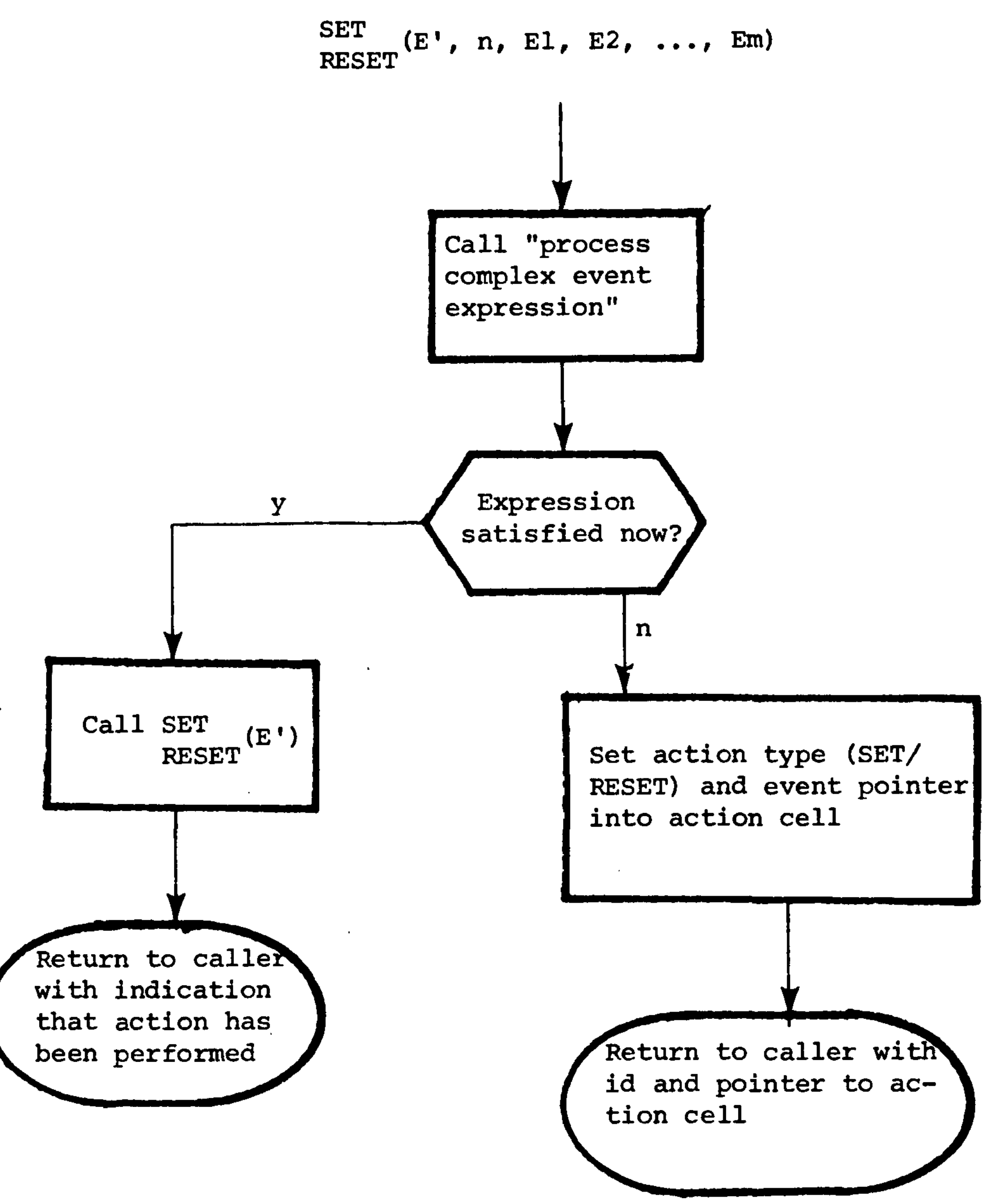


Figure 3.6-17 SET(E',n,E1,E2,...,Em) and RESET(E',n,E1,E2,...,Em)

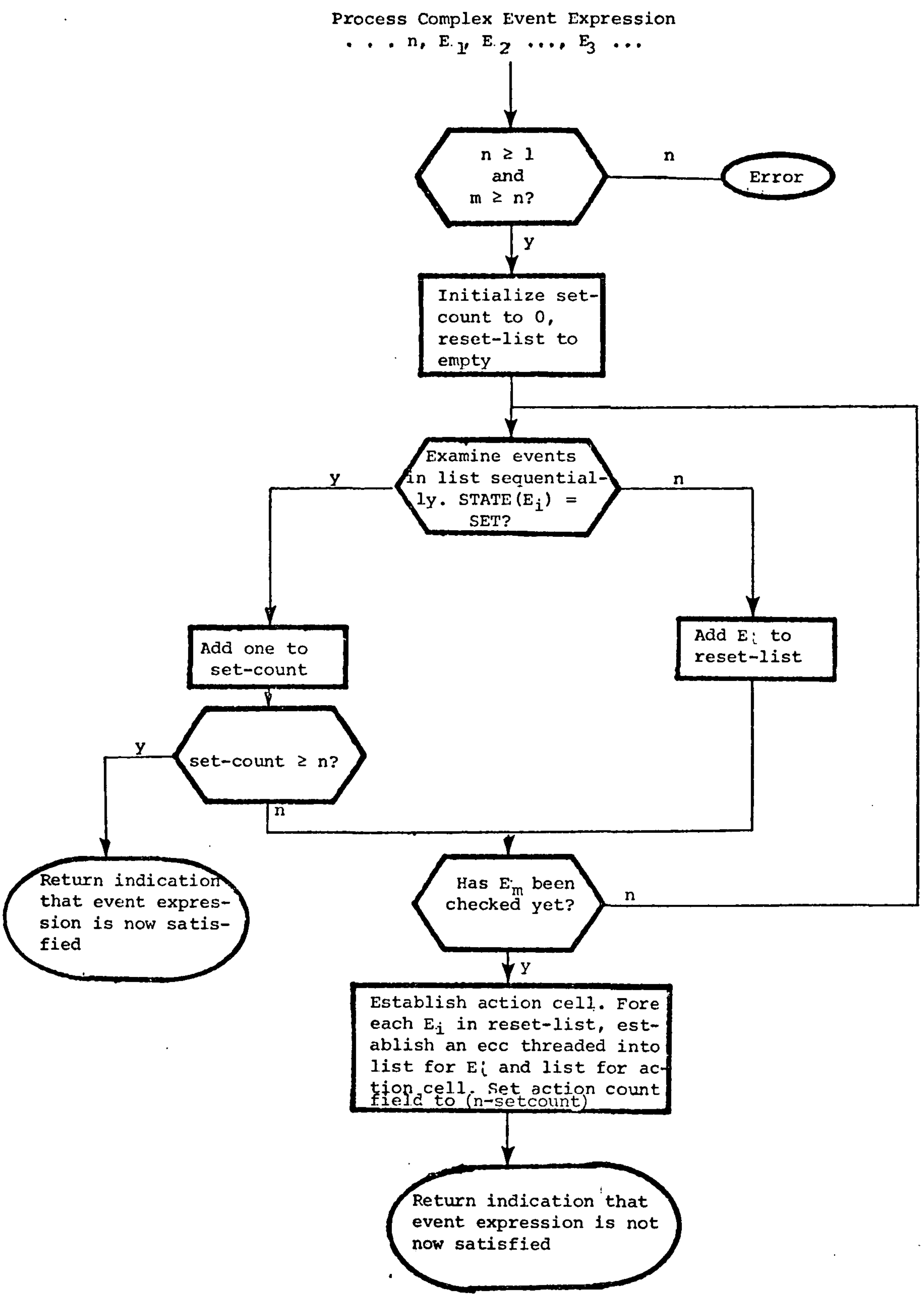


Figure 3.6-18: Complex Event Handling

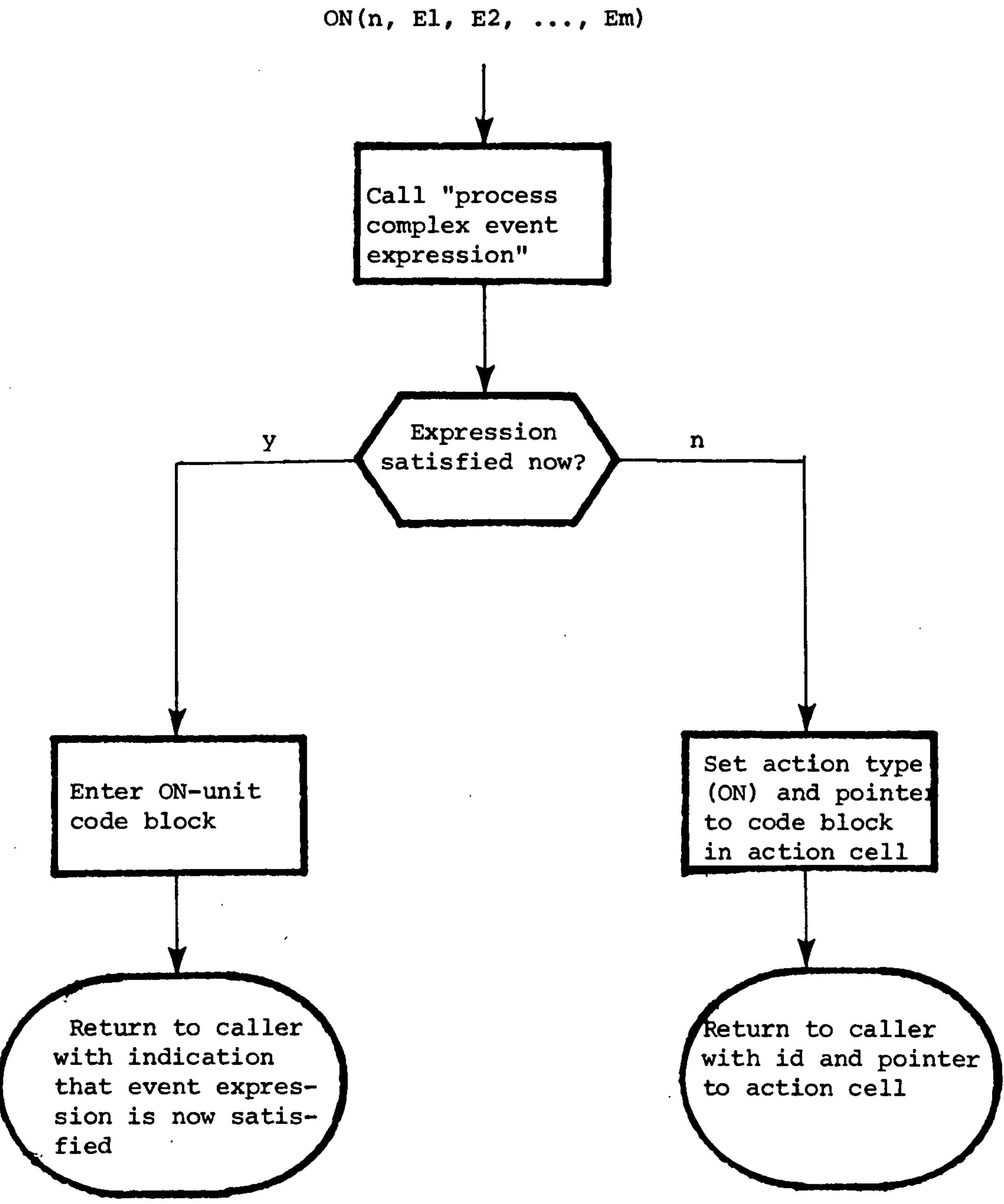


Figure 3.6-19 ON(n,E1,E2,...,Em)

condition is satisfied. In this context, the "first opportunity" occurs as soon as the process which executed the ON is found to be in the running state. Thus, if the process is running at the time, the entry of the code block will be immediate. Otherwise, the process must reach the running state before the entry occurs. Specifically, the satisfaction of the condition does not ready the process if it is in the wait or stopped state (unless, of course, the wait condition happens to be satisfied as well). In any case, the execution of the ON-unit's code block occurs at the same priority as the process to which it belongs.

Execution of ON-unit code may be interrupted by other ON-units. If two or more ON-units are found pending at the moment a process is returned to the running state, entry of the oldest is initiated, followed immediately by the next oldest, and so on, until the queue of pending ON-units is emptied and the stack build-up has been accomplished. This action is analogous to that which would have occurred if the ON-units had been triggered serially, but close together in time. This is functionally desired, to permit handling of faults or traps which arise during the execution of ON-unit code in a standard fashion.

One further variation of the ON primitive is provided to deal with repetitive events:

$$ONR(E_p) \text{ <code block>}$$

where E_p is the name of a pulsed event. The only difference between the effect of ONR and ON is that ONR marks the ecc so that its release and the action block's release are inhibited when the event is set. The consequence of retaining these blocks, instead of releasing them, is that the ON-unit remains active, and will respond to subsequent SETs of the pulsed event without explicit re-execution of an ON operation. This provides improved efficiency when repetitive action is desired, since the overhead of release and re-establishment is avoided.

3.6.2.3 Example of Event Mechanization: Figure 3.6-21 shows, in abbreviated form, the structure for events E1, E2, and E3 (all of which are in the reset state) following the execution of

ON(E1) <code>

SET(E1, 2, E2, E3)

WAIT(1, E1, E2)

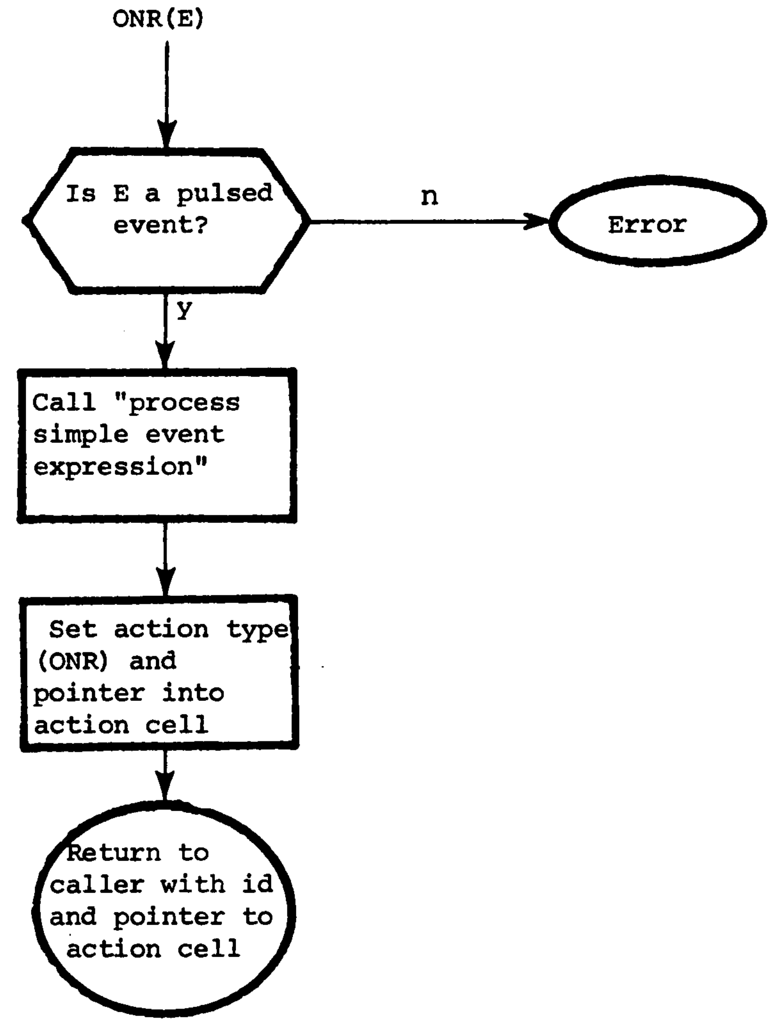


Figure 3.6-20 ONR(E)

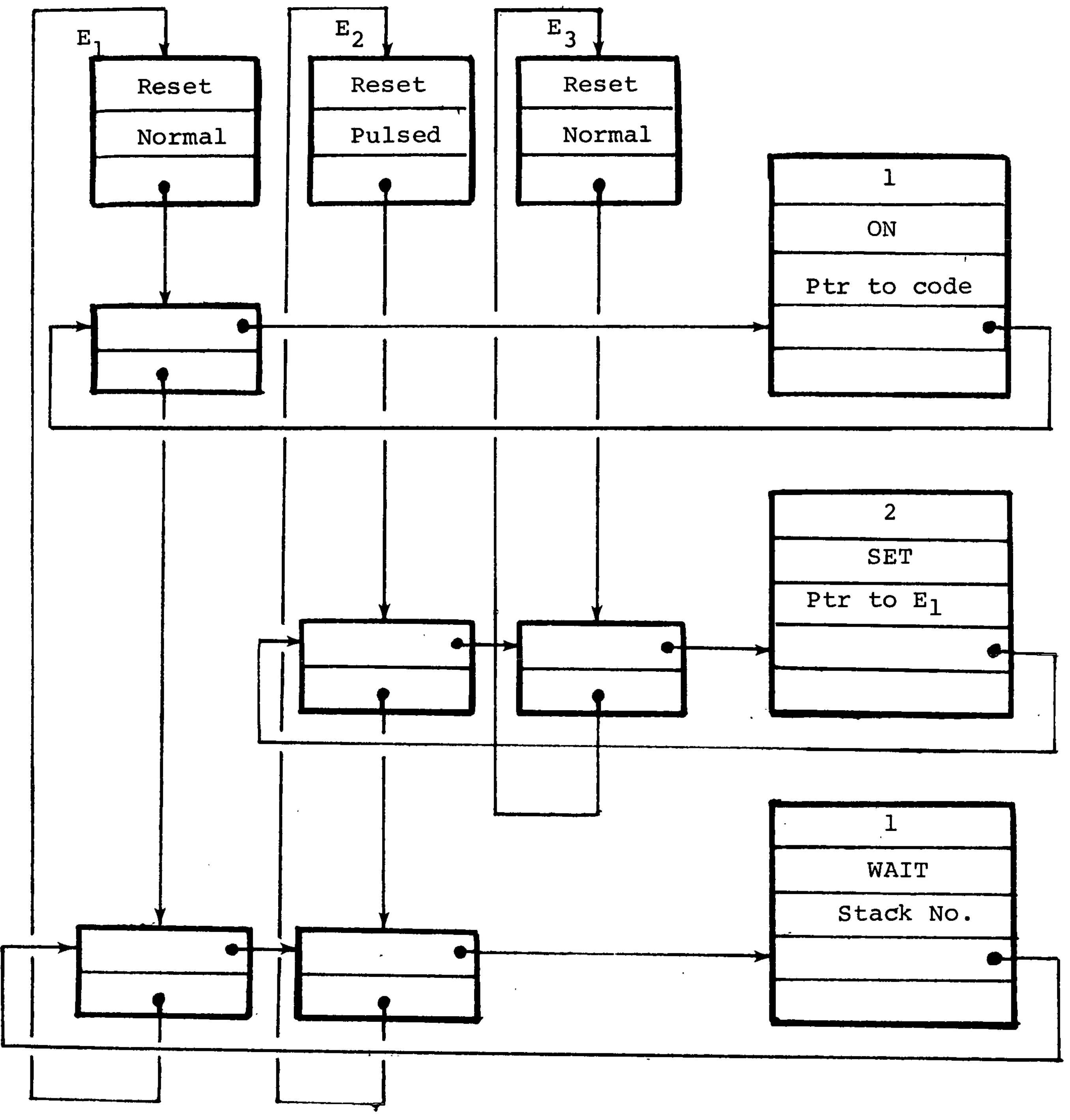


Figure 3.6-21 Event Links After Execution of:

ON(E_1) <code>
SET(E_1,2,E_2,E_3)
WAIT(1,E_1,E_2)

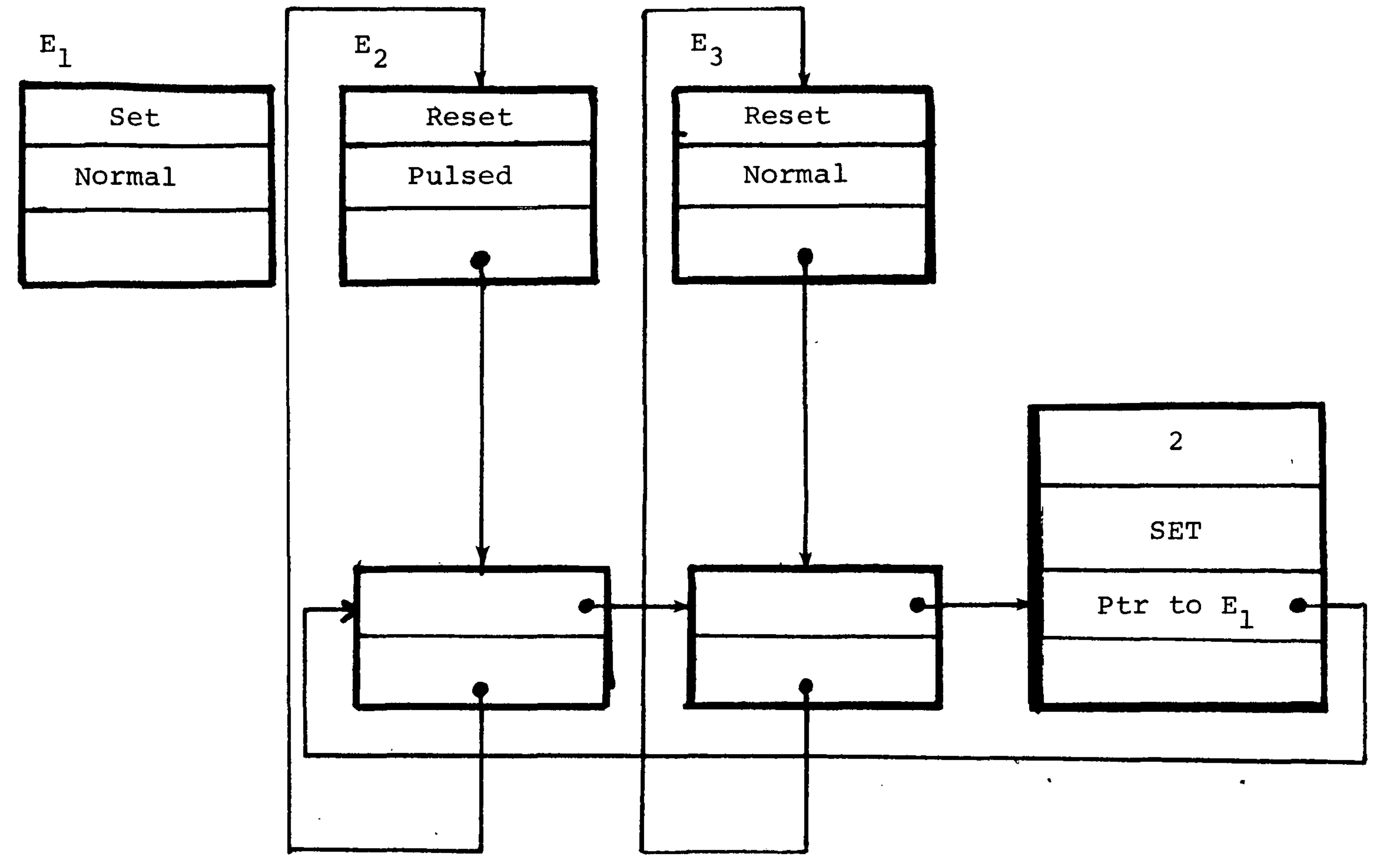


Figure 3.6-22 Event links of figure 3.6-21 after execution of: SET(E_1)

If E1 should be SET by some process, two consequences would result. First, the "ON" action would be triggered, since the count-to-go is one, and E1 is connected to the action cell. Second, the other ecc for E1 would cause the waiting process to be readied, since its count is also one and E1 is part of its action list. The E2 ecc linked to the WAIT action cell would be released, since the action to which it is connected was performed. Figure 3.6-22 shows the structure of Figure 3.6-21 as modified by the SET(E1).

Suppose instead that a process had issued SET(E2). The result is shown in Figure 3.6-23. The count for the "SET" action cell has been reduced by one, the waiting process has been readied, and the E1 extra ecc has been released. Note that because E2 is a pulsed event, its state remains reset. Now suppose that SET(E3) is subsequently performed. This completes the condition for the "SET" cell, and therefore causes E1 to be set. This, in turn, triggers the "ON" unit, leaving the structure empty, with E1 and E3 in the set state.

3.6.2.4 Complex Event Expressions: If an action is desired as a consequence of any n of m events, the primitives are applicable with no elaboration.

WAIT(1, E1, E2, ... , Em)

has the effect of waiting until the "OR" of the events happens, while

WAIT(m, E1, E2, ... , Em)

waits until the "AND" happens. (Note, however, the looseness of the interpretation of AND; because of the implementation, there is no guarantee that all events listed will be simultaneously in the set state when the wait ends. Rather, it is true only that each event has been at least briefly in the set state since the WAIT was issued; it is easy to visualize a case where a process which tests the states of all listed events just after a WAIT on a list will find every one in the reset state.)

The SET primitive can be used to obtain the effect of a WAIT which ends either when event C happens or when both A and B happen:

SET(X, 2, A, B)

WAIT(1, X, C)

Thus, the dummy event X is set upon the "AND" of A and B, and the WAIT waits for the first occurrence of X or C.

3.6.2.5 Scope or Lifetime of Event Operations: For an event primitive to be meaningful, all components in the primitive must have an existence. For example, if a program block is exited in which a member of an event list (or an E') is declared or an active ON is located, or if a WAITing process is terminated by another process, the related event structure must be purged. Furthermore, if the process which executed a still-active complex SET is terminated, the SET structure must be purged. Although more complicated strategies are easily constructed, the additional sophistication does not seem to justify the additional overhead.

On occasion, it is desirable to purge an event-structure explicitly. This may be achieved by performing

PURGE(key)

where the key is the value which was returned as part of the execution of the ON operator, or by the complex form of SET or RESET. It contains an id part and a pointer part. The id is unique, and is also stored in the action cell. The pointer part points to the action cell, from which the row chain of ecc's may be entered. If the id-field of the cell indicated by the pointer part does not match the id specified in the PURGE key, PURGE returns without effect.

3.6.2.6 Connection to External Signals: Because of the real-time nature of the applications for which the multiprocessor is intended, it is useful to provide a correspondence between event cells and actual events manifested in signals which trigger the "external signal" interrupt. This interrupt is categorized as system-oriented, and it is handled by the operating system external signal interrupt procedure. This procedure is compiled with a Compool segment which contains a declaration for each event cell to be connected to an external signal. The names of these cells are determined and known at the operating system level, so they may be expected to remain reasonably fixed. However, the Compool and the interrupt handler may readily be modified when necessary.

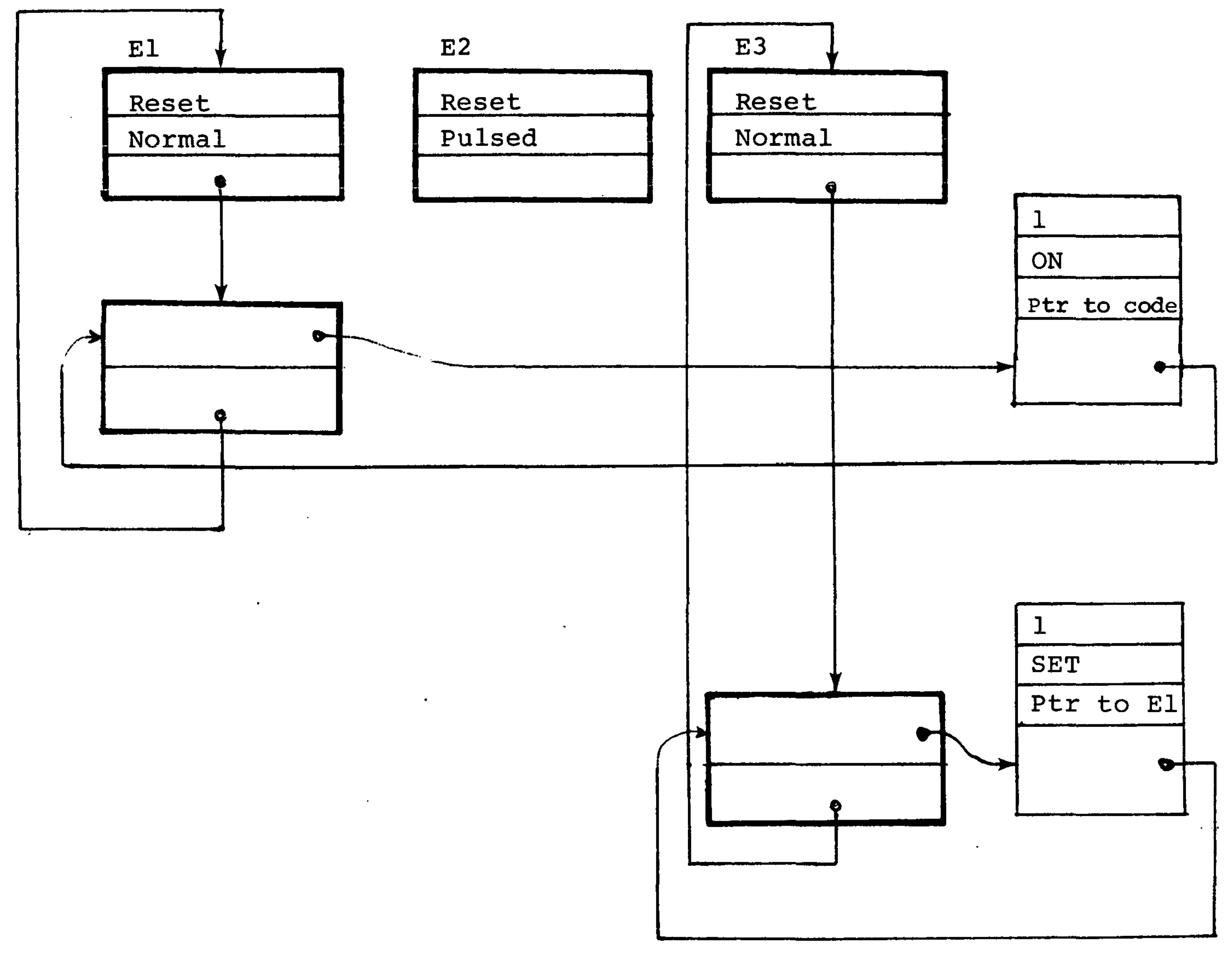


Figure 3.6-22: Event Links of Figure 3.6-21 After Execution of: SET(E_1)

The occurrence of an external signal interrupt will trigger a SET operation on the appropriate event cell. Any number of processes may build event structures involving these cells.

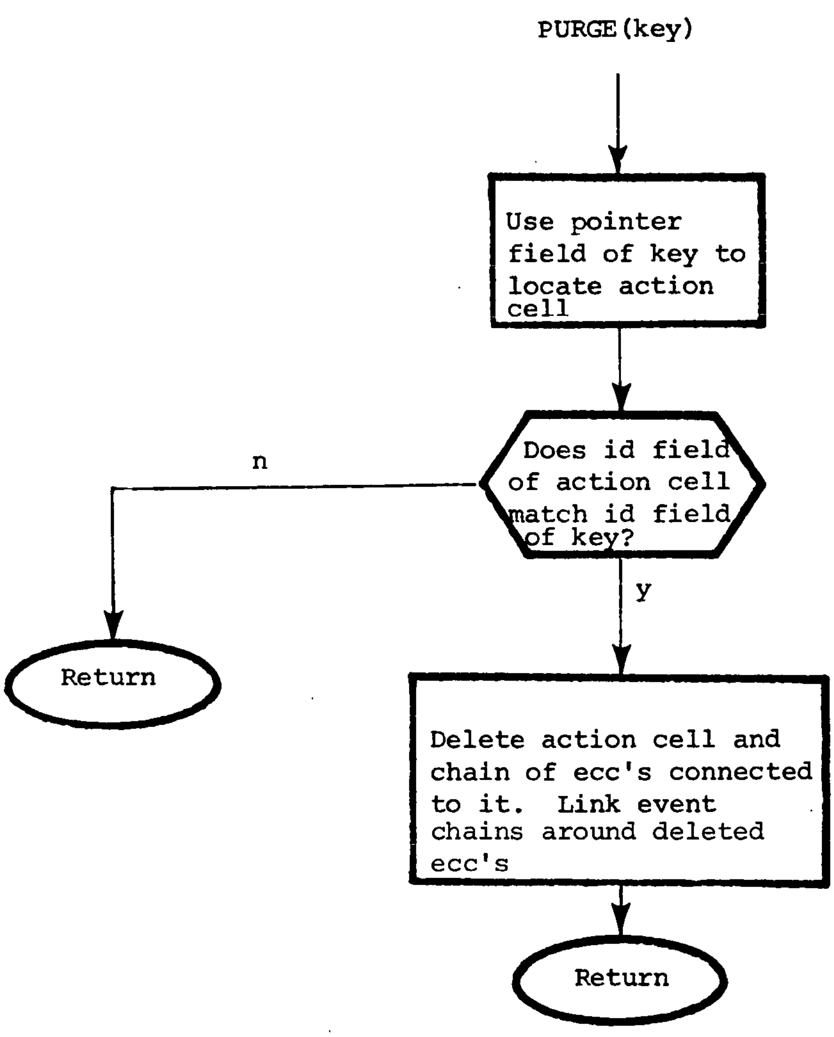


Figure 3.6-24 PURGE (key)

3.7 Fault Recovery Methodology

As indicated in the introduction to this report, the emphasis in this multiprocessor design has been to provide instantaneous, all-hardware error detection to prevent the random propagation of the consequences of an error. Because of limitations of speed and complexity, no software is involved in the detection of errors, and only the operating system has been involved with the task of managing system recovery. The overriding requirement that has been observed throughout the system design is that the applications programmer should be kept oblivious of the error detection and recovery mechanisms. This section discusses the functions of the operating system in the recovery process. Some duplication of the more detailed descriptions to be found in Chapter 4 is made for the sake of completeness.

The failures of four kinds of units are considered in this section: processor, M1, M2, and I/O controller. The design of the system is such that the processor failures do not incur loss of information; thus, the recovery from failure of a processor differs from the recovery from a failure of the M1 with which it is directly associated. Faults in M2 memory involve information loss. Faults in the I/O controller must be masked, since there is only a single copy of this device.

3.7.1 Recovery From Processor Failure

Processing units are comprised of two identical units which are coupled with comparison circuitry for error detection purposes. This design has been chosen so that errors committed by a processor are detected immediately before propagation occurs. The instruction execution sequence is partitioned into two phases such that in no case does processor failure render the instruction incapable of being restarted at an immediately preceding point. This is achieved by insuring that no instruction phase destroys the inputs to that phase. Therefore, the information is available, and it is merely necessary that the processor correctly signal the occurrence of an error, and that the system provide control for instruction retry. Because retry may prove unsuccessful, it is also necessary to provide access to information stored in M1, in order that the process interrupted by the failure may be resumed on another processor.

Upon the occurrence of an error in a processor, the error control logic for the processor transmits a signal to the I/O controller, by means of the IPC bus. The error

control logic also renders the processor dormant (a condition in which the processor ceases instruction execution and is not responsive to interrupt signals). The response of the I/O controller occurs in two sequential parts. During the first part signals are sent directly from the I/O controller to the faulty processor requesting the processor to re-execute the instruction phase during which the error was detected. Because the inputs have been preserved, this phase may be retried as often as desired. Only if it completes successfully will the next phase be enabled, and not until then will inputs to the previous phase be destroyed. The faulty processor responds to the request by attempting the specified execution; whether successful or unsuccessful, it responds to the I/O controller with a signal indicating whether the result was correct. If the result was incorrect, the I/O controller will command another attempt, and so on, until a predetermined number of retries has been attempted without success, or retry succeeds. In either case, the I/O controller prepares an interrupt command to some other processor (the "rescuing" processor), chosen on the basis of interruptibility index. The purpose of this interrupt depends partly upon the success of the retry attempt. The fact that the error occurred, together with the outcome of the retry attempt, are recorded regardless. If the retry attempt was successful, it is presumed that the processor which detected the error can continue the process it was executing at the time, i.e., the error is diagnosed as having been temporary. The rescuing processor sends a resume command to the faulty processor over the IPC bus, and returns to normal operation itself.

If repeated retry attempts prove unsuccessful, the error is classified as permanent, and the process interrupted by the fault must be resumed on a different processor. The rescuing processor calls the M2 space allocation routine to obtain an area into which the contents of the faulty processor's M1 memory may be unloaded. Next, it sends to the faulty processor (in fact, to its M1) a command to unload M1 into the designated M2 area. This command and its address information are sent via the IPC bus. Two forms of this command are available; the form used in this case is to unload M1 with full error detection enabled. That is, should a fault occur during the unloading of M1 into M2, an appropriate error signal will be sent, causing normal M1-failure action to be followed. (The second form of the unload command is described below in connection with M1 failures.) Because the processor which failed did not normally terminate its activity with respect to the stack of the interrupted process, the rescuing processor must manipulate the M1 information and the abandoned stack in such a way that the abandoned stack is made normal with respect to resumption of operation. This implies packing and

storing of status bits, return control information, and mark stack indicators, and fabrication of stack values for a normal interrupt procedure entry. When this action is completed, a call to the operating system is performed, to switch the status of the interrupted task from the running to the ready state. At this point, the process interrupted by the fault is in the same condition which results from a normal interrupt, and it may be rescheduled by the operating system in accordance with its priority on any available processor. Before transferring control to the dispatcher, the rescuing processor schedules a self-test diagnostic process for the faulty processor. The discussion of such a maintenance program is beyond the scope of the report; however, it is noted that the initial conditions for the operation ot this routine are 1) the faulty processor's M1 memory requires loading, and 2) the faulty processor is in a dormant state, and requires a special command before it will attempt execution of instructions. Thus, it appears reasonable to assign the self-test package to a healthy processor, and to allow that processor to administer the operation of the self-test package.

The final actions of the rescuing processor are to release the M2 area obtained to hold the M1 contents, and to transfer control to the operating system dispatcher, to allow the process interrupted by the fault to be assigned a processor if its priority warrants. Of course, if this happens, some other process will of necessity be switched from the running to the ready state, since there is now one less processor available in the system.

3.7.2 Recovery From M1 Memory Failure

Three independent tests are made in M1 for error detection purposes: parity tests on each copy of the duplexed M1, and an agreement-comparison between the two copies. The various combinations of good and bad parity with agreement and disagreement result in eight different possibilities for the outcome of each comparison. Only one of these represents satisfactory operation: the copies agree, and both have correct parity. Any condition in which disagreement or bad parity is detected results in a signal over the IPC bus generated by the error detection logic in the M1 system. A 3-bit error configuration indicator accompanies the error signal. The I/O controller responds as with processor failure: it commands a number of retries. If this proves unsuccessful, it selects a suitable processor, and interrupts it. The rescuing processor calls the M2-space allocation procedure of the operating system to obtain two areas to receive the M1 contents. The process then uses the second form of the M1 unloading command, which specifies unloading of one member of the duplexed pair into the

designated M2 location, inhibiting transmission of an interrupt signal in the event that a fault (parity failure) is detected during the operation. Instead of an interrupt, a detected fault sets a condition code in the rescuing processor, which is examined upon completion of the unload operation. Thus, the rescuing processor commands, sequentially, that both copies of the contents of the failed unit are unloaded to M2 memory. From this point, subsequent action depends upon the 3-bit error code generated by the failure in M1. If neither copy of M1, or both copies of M1, signal bad parity, it is impossible to tell whether either copy is correct. In this unlikely event, the recovery software is unable to continue, and appeals for assistance from the system operator.

In the more probable failure case, the 3-bit error code will indicate disagreement between the values, and one bad parity. If the copy of M1 not indicating bad parity was unloaded into M2 without indication of failure, then this copy is loaded into the faulty M1, and execution of a single phase of the interrupted instruction is commanded. If unsuccessful, the retry is repeated, without reloading M1, a limited number of times. Should this prove unsuccessful, the process interrupted by the fault must be switched to the ready state, its stack adjusted to resemble normal operation, and the rescuing processor exits via the operating system dispatcher as described above under processor failure. In case of success, the M2 areas may be released, and operation can be resumed on the original processor.

3.7.3 Recovery from M2 Failure

There are two cases of M2 fault detection: the fault may be encountered during execution of some operation in 1) a processor, or 2) in the I/O controller. In the first case, the processor enters the dormant state, as in the processor-failure case, and transmits a signal to the I/O controller via the IPC bus. The I/O controller commands a single phase retry to determine whether the fault was transient. If after a limited number of retries the result is unsuccessful, the I/O controller sends an interrupt signal to another processor, to obtain assistance. Because there is a chance that the processor which encountered the fault might have been performing those operations required to enter a procedure just when the fault was encountered, it is undesirable to select that processor to handle the interrupt, since an attempt to respond to an interrupt may cause recurrence of the failure. However, any processor may be selected if the M2 fault was signalled during an operation under control of the I/O unit.

Recovery from M2 failures depends upon the presence of doubly stored information. In the case of procedure and other read-only segments, double storage consists of a copy in M2 and a copy in M3. For critical variable information double storage takes place through maintenance of two copies in different M2 units. The decision as to which variable data is sufficiently critical to warrant multiple storage in M2 is partly up to the application program, and partly under control of the system. When the application program specifies double storage for a given segment, that fact is recorded in the Mom descriptor (see section 2.4.2); the hardware then automatically performs the multiple stores without requiring additional code to be provided.

Singly stored read-only and variable segments still require double storage, however, of the following three quantities:

a) the length of the M2 storage area where the segment is found,

b) the location of the Mom descriptor, and

c) the location in M3 from which the copy was obtained.

Thus, even in the event of total unit failure, it is possible to discover the identity of the contents of that unit from the information that survives, to locate the descriptors for the information, and to locate the copy in M3. The action taken with respect to memory isolation following a failure indication depends upon the scope of the failure. Should a group of contiguous words in a module be affected, the recovery software constructs an unusable-memory block, also marked non-relocatable, so that the area of memory known to be faulty is removed from further use without disabling the entire module. To determine which segments are located in that area, a search of physical memory, in the order of (and by use of) the memory links is carried out. The Mom descriptors pointed to by those links are accessed. If the Mom descriptor indicates a read-only segment, it will be refreshed from M3 merely by marking its descriptor absent. If, on the other hand, the descriptor indicates doubly stored information, an M2 to M2 transfer can be performed to re-create two correct copies. Finally, if singly stored variable data is involved, the Mom descriptor must be marked unusable by the recovery procedure. This is accomplished by marking the segment absent, and by using a pseudo M3 address designator which is interpreted by the absent segment interrupt handler as signifying a destroyed segment. Should one or more processes attempt to

make this segment present, the interrupt handler will recognize the situation, and terminate these processes with an appropriate diagnostic indication.

On completing these actions, the rescuing processor logs the fact of failure and the recovery action taken, frees the dormant processor, and resumes operation normally.

3.7.4 Recovery from I/O Controller Failure

Since there is only one copy of the I/O controller, errors in this device must be masked by redundancy. The recovery operation is then done automatically in the device. A processor is selected to be interrupted for the purpose of logging the fault and the recovery action. Mask bits are provided so that these interruptions can be held below a certain frequency, so that the system is not too frequently called upon to log a masked fault. Error counts may be maintained in the I/O controller itself, because of its modular redundancy. Further details of the IOC internal modular redundancy are given in the next chapter.

References for Chapter 3

1) Madnick, S.E., "An Analysis of the Page Size Anomaly", MIT Project MAC, 13 December 1971.

2) Randell, B., "A Note on Storage Fragmentation and Program Segmentation", Comm. of ACM, Vol. 12, No. 7, July 1969.

3) Knuth, D.E., "The Art of Computer Programming", Vol. 1, Fundamental Algorithms, Addison-Wesley Publishing Co., pp. 435-455.

4a) Crighton, R.P., "B6700 Core Memory Allocation", August 1971, privately obtained memorandum.

4b) Burroughs Corporation, "B6700 Master Control Program - Information Manual", Document # 5000086, 1970.

5) Mattson, R.L., "Evaluation Techniques for Storage Hierarchies", IBM Systems Journal, Vol. 9, No. 2, 1970, pp. 78-117.

6) Corbató, F.J., "A Paging Experiment with the Multics System", MIT Project MAC, MAC-M-384, 8 July 1968.

7) North American Rockwell/Space Division, "Modular Space Station Phase B Extension - Information Management Advanced Development Final Report", 31 July 1972, Contract NAS9-9953, MSC02471, DRL No. MSC-7-575, Line Item 72.

8) Intermetrics, Inc., "Final Report -- Standard Interface Definition for Avionics Data Bus Systems", May 1971, prepared under contract NAS9-11477.

PRECEDING PAGE BLANK NOT FILMED

4.0

FAULT TOLERANT ASPECTS OF THE MULTIPROCESSOR

This chapter deals with the problem of failure detection and the recovery process which re-establishes the processing activity. For the purpose of this discussion a failure is defined as a malfunction of a system component, while an error is the manifestation of the failure, usually detected as bad data or incorrect control sequencing. A failure may be categorized as either transient or permanent. A permanent failure is defined as one which will repeatedly cause the same error condition. A transient failure cannot be repeated consistently. It is presumed that if a transient failure occurs the hardware will function properly if its "state" information is restored. "State" information includes values of all control variables and initial data.

4.1 General Philosophy and Requirements

In order to develop a specific approach to fault tolerance a number of requirements were generated and a philosophy was established, against which specific design considerations could be judged. The requirements were based upon those of the anticipated Space Station [1,2,3], in the absence of operational requirements specified by the contract. The following basic premises of error detection were adopted:

a) The system must be designed so that almost all failures can be detected. The desirability of detecting and recoverying from every possible single failure is clear. However, in practice this proves impossible. Only with an assumption of statistically independent failures of all components, may all failures will be detected. The failure of any integrated circuit, the open circuiting at any interconnection and the shorting of any circuit or wire to a logical "1" or "0" state will be considered. Cases in which a shorting bar is placed across a number of circuits simultaneously is considered a violation of the statistical independence principle. In the case of a dual redundant subsystem electromagnetic coupling which causes failures in both units is also a case of statistically dependent failures.

Preceding page blank

b) Errors must be detected as soon as they occur. The error should not be allowed to propagate so as to incapacitate the system's ability to recover. A failure which causes incorrect information to be written into M2 or improper I/O commands to be issued could leave the system in an unrecoverable situation. Incorrect M2 write operations must be prevented or made to occur at such a low probability so as to be practically impossible.

c) A basic assumption is that repair by replacement of the multiprocessor communicating elements will be undertaken after a permanent hardware failure has been detected, the system reconfigured and the malfunctioning element isolated. For this reason the system will be designed to guarantee single component failure recovery. Recovery from the single failure might leave the system in a degraded performance mode. However after repair is accomplished the maximum potential capability can be reestablished. This does not imply that the affects of the failure will not be felt by the system. A number of low priority functions may not have been executed. This loss can never be made up, and must be a factor acknowledged by the system designers. Future operations will, however, eventually recover to be within specification. The failure criteria for the multiprocessor will then be Fail Degrade (FD) after the first failure, and fail safe (FS) after the second failure. If a third failure occurs before repair is accomplished the operation of the system cannot be guaranteed. In case of a permanent failure the system will provide degraded performance. That is, when all elements are in a non-failed state their total resource will be allocated to optimizing the computer system throughput. There will be no on-line spares. In order to minimize the effect of the FD mode upon real time computations all processes are allocated a priority. After a failure high priority tasks will be guaranteed resources to recover and resume operation in preference over low priority or background tasks, which may therefore be delayed or even locked out due to temporary overcommitment of resources. Real time computations are considered to be the highest priority processes.

d) If a component failure occurs, a second failure is assumed impossible during the reconfiguration process. For component failure rates on the order of 10^{-3} to 10^{-4} $(\text{hours})^{-1}$ and reconfiguration times measured in milliseconds, the probability of the second failure occuring during reconfiguration is less than 10^{-8}.

e) Because of real time control functions the recovery time must be deterministic, and of the order of 10 or 100 milliseconds. Recovery time is measured from the time the error is detected to the time the highest priority active process resumes its calculations. In order to meet this requirement an analytical and experimental series of tests must be developed to demonstrate that under all processing situations the system can recover in less than the maximum recovery time.

f) The experience of the Apollo program demonstrates that it is essential for the applications programmer to be ignorant of the recovery process. The entire error detection, isolation and recovery process must be handled by the hardware and the operating system.

g) A false indication of a failure is not catastrophic to system performance. At worst it can cause an internal reconfiguration and put the system in an FD mode of operation. Fault isolation logic will indicate a failure in the error detection mechanism. However, if the failure detecting mechanism fails in a mode which prevents the reporting of a failure, then a serious situation can occur. The second failure, if in an operational unit (P, M1, M2, I/O, bus) would not be reported and the error condition created by this second failure could be propagated. This non-indication situation must be designed out of the equipment by appropriate application of redundancy techniques.

4.2 Major Phases of Fault Tolerant Operation

The choice of an effective fault tolerant design requires an investigation of error detection mechanisms, fault isolation logic, and recovery philosophy. These items will be discussed in general and their application to the operating units of the multiprocessor (P, M1, M2, I/O, bus) will then be presented.

4.2.1 Error Detection and/or Correction

Errors due to transients and errors due to permanent hardware failures must be considered. Transient errors are generally not caused by hardware failure but rather by a source of external noise. They occur for a short duration and infrequently. It is therefore impossible to isolate the source of the error by testing since the hardware operates satisfactorily after the failure. Consequently, it is not necessary for a spare unit to be switched in order to perform recovery. Only the state of the failed unit prior to the failure

must be restored. Any successful error detecting technique must detect both transient and hardware failures. The following sections review a number of methodologies that may be applied to provide the error detection capability.

4.2.1.1 Periodic Diagnosis. This method relies on software to periodically initiate test sequences and compare the results with predetermined values. For a number of reasons it is unsatisfactory for error detection within the multiprocessor: First, there is no guarantee that the error is detected before damage has been caused in terms of bad write operations into M2 or incorrect I/O commands. Second, there are failure modes which can prevent a processor from sequencing, requiring, therefore, some form of hardware time-out. Third, the categorization of all the possible failure modes and the execution of the tests to interrogate the possible failures can be a significant software effort. Fourth, there is a large probability that transient errors won't be detected [4].

4.2.1.2 Error Detecting Codes. Coding techniques have been studied for many years [5]. Codes may be classified as either transmission codes or arithmetic codes. Both may be used for error detection on the internal bus and in memory. Transmission codes do not retain their error detection characteristics under arithmetic operations. As a matter of fact neither transmission nor arithmetic codes retain their characteristics under non-linear binary operations.

There are a number of points to consider before relying solely on coding for error and failure detection.

a) Not all component failures result in error patterns which can be detected by a given code.

b) Extensive error coding imposes higher bit rates, and may degrade performance.

c) The commonly applied parity coding is an effective method of detecting single bit errors, but is ineffective in protecting against transients which affect more than a single bit. This is the statistical independence requirement.

In the processor and memory sections these points will be addressed when codes are considered.

4.2.1.3 Component Level Redundancy. This methodology applies redundancy at the circuit or gate level. It is possible to

design fail safe logical systems, in the sense that the system continues satisfactory operation even if a component fails or a single bit transient error occurs. By incorporating redundancy as an inherent part of the initial design procedure rather than after the design is accomplished, logic can be synthesized which is tolerant of single component failures. The method applied by Russo [6] and refined by Beister [7] involves utilization of a minimum distance 3 state assignment, and is analogous to a single error correcting Hamming code. Russo's design for implementing a decimal counter came to more than eight times the amount of hardware than for the nonredundant version. The implementation involved AND/OR diode logic. Beister's implementation, exploiting the properties of threshold logic, required 3.5 times as much logic.

These results are for the detection and recovery from single component failures. The amount of extra hardware required to achieve this capability is, indeed, disappointing.

Quadded logic [8,9] allows a number of logic gate failures within a digital computer without disturbing its capacity to perform the function for which it was designed. This techniques can, in theory, increase reliability by orders of magnitude.

Another form of logic level redundancy is known as interwoven redundant logic [10]. With this technique, each gate receives a number of versions of each input and forms its output from the redundant input information. Certain redundant gates while performing logic can correct errors from the previous stage in one layer of a multilayer structure. Other redundant gates correct errors in two alternating layers.

All of the above techniques suffer from three major drawbacks:

a) On the basis of gate count alone, the application of component level redundancy is expensive (at least four times the simplex version);

b) The large increase in the number of interconnections between the redundant gates can itself produce unreliability;

c) It is very difficult to maintain statistical independence between failures when one considers problems associated with mechanical packaging and power supplies.

4.2.1.4 Functional Level Redundancy. Functional level error detection is at the level of the arithmetic unit, operating

registers, shifter, etc. If dual functional units are synchronized properly then the output modes of the dual units can be compared and errors detected. Even at this level single failures may escape error detection; for example, a power supply transient. However, careful engineering can anticipate these situations.

Majority voting is a technique [11,12] where each functional unit or system is triplicated and the output is chosen to agree with the majority of the individual outputs from the triplicated elements. An example of an application of majority vote techniques is the Saturn V Computer [13]. Adaptive vote taking [14] is a modification and extension of the majority vote technique. This method employs more than three versions of a functional unit output. When one unit fails, it is automatically switched out of the majority vote network, allowing a greater increase in reliability over conventional majority vote techniques.

4.2.1.5 Modular Redundancy. A question arises as to whether functional level redundancy with comparators at each functional intersection or modular redundancy (a complete duplicate P, M1, M2, or I/O unit), provides better error detection properties. From a system performance point of view it is only necessary to detect the inability of a subsystem to communicate correctly with its environment. A duplicate module operating independently of the first, with a comparator to detect any difference in outputs, is just as effective in detecting errors as functional redundancy within the module, provided that error sources are statistically independent in the two modules. In fact, modular redundancy may very well be more cost effective, since the total component count will not, depending of course on the complexity of the comparator, approach that of the functional redundancy examples in the previous section.

In a sense the definition of a multiprocessor implies that errors are to be detected at the module level and degradation in performance is to occur when a module fails. A multiprocessor should be capable of operating with less P, M1, M2, or I/O capability. It is the modularity and the degradability which are the main reasons for investigating multiprocessors.

4.2.2 Fault Isolation

Fault isolation is the process of differentiation between the bad and good units. In an arrangement using dual operating units with a comparator it requires a lower level of error detection. If all errors were caused by solid hardware failures, then

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

software or microdiagnostics could be successful in isolating the failures. However, because of the transient nature of many failures, anything less than hardware fault isolation logic would be unsatisfactory.

4.2.3 Recovery

Recovery is defined as the continuation of system operation, with data integrity, within real time constraints, after an error has occurred. If a computer system is modeled as a very large finite state machine then the recovery process can step the system from an illegal state (determined by the error detection and isolation logic) to the proper legal state from which normal processing can resume. The problem of recovery is in exactly determining and restoring the proper legal state in an efficient manner.

The return state could be the last legal state of the system prior to the failure. In this situation the state can probably be determined quite precisely. Single instruction restart is an example of returning to the last legal state. However, if a more complex sequence of operations is chosen as a restartable entity then it might become more difficult to determine the exact state. In this case, an equivalent state, which will ultimately arrive at the same final state, is chosen.

4.3 Error Detection

The following sections will present various techniques for detecting errors within the various communicating elements of the multiprocessor.

4.3.1 Error Detection within P

In order to gain an appreciation of the complexity of error detection within P a number of problem areas are discussed.

4.3.1.1 Fan Out. In the process of logical design, a single gate is often used to drive many different gates. This is called fan out. The failure of the driving gate not only causes its output to be incorrect, it can also propagate to the output of all the gates to which it is attached. An example where this problem directly affects the data content within a P element is in data bussing.

Internal to P are a number of busses or common points where many different data registers may be gated. The failure

of a logic element or connection in the data bit logic usually manifests itself as a single bit error. This type of error can be easily detected by conventional parity techniques. However, the failure of a control line used to gate information onto the bus can manifest itself in a multiple error situation. In the worst case, all the bits on the bus could be in error.

If two registers are gated onto the bus at the same time, a logical OR between the two registers occurs. To the author's knowledge, no mechanism exists which can, in general, detect this situation short of a complete duplicate bus with redundant control lines.

In terms of coding theory if there are N information bits a minimum of N check bits must be used to detect a burst of N errors. This bussing example shows a situation in which a single component failure caused a potential N bit burst error.

4.3.1.2 Arithmetic and Logic Unit (ALU). In discussing the failure tolerance aspects of the ALU, one must distinguish between the coded form of the data inputs and the actual information which the coded form represents. The coded information may possess redundant information which can be exploited to provide error detection.

An ALU may be defined as having two inputs, an operation and an output. A and B are the information inputs, C is the information output and OP is the operation defined between A and B:

$$A \text{ OP } B = C,$$

OP is typically ADD, SUB, Logical AND, Logical OR, Exclusive OR, MULT, DIV, etc.

Now, if F(A), F(B), and F(C) are the coded words, there must exist an operator * such that,

$$F(A) * F(B) = F(C)$$

Also, to be useful, in error detection, the function F must contain enought redundant states to detect all single errors. One may consider two types of F functions. Separate codes enable F(A) to be the juxtaposition of A and some redundant check bits G(A) with a check bit operation *:

$$F(A) = A, G(A)$$

$$A \text{ OP } B = C \quad G(A) * G(B) = G(C)$$

Non-separate codes do not lend themselves to this partitioning.

Literature [15,16,17,18] exists which deals with the problem of arithmetic codes, e.g., OP = ADD or SUB. Both separate and non-separate codes exist. If OP = Logical AND or Logical OR, no efficient separate code exists. To be efficient, the number of bits necessary to represent F(A), while still retaining a single error detection capability, must be less than twice the number of bits necessary to represent A.

Peterson [19] has shown that all the arithmetic codes provide error detection by performing a residue check on the result of the arithmetic operation. Rao [20] has shown that for a modulo(3) check upon a 25 bit word approximately 40 percent more hardware (in terms of a NAND gate implementation) was required.

One of the most efficient separate codes is where G(A) is the parity function of A. However, the logic necessary to compute the parity function of the sum of two numbers requires the duplication of the carry logic of the adder. In addition only half of the possible single component failures are detected. In conclusion, any coding method used for error detection within an ALU is quite complex in terms of the number of logic gates.

4.3.1.3 Encoding and Decoding. Within the logical structure of P are areas in which bit patterns must be decoded or encoded into different formats. The decoding may or may not produce mutually exclusive outputs. These processes may, in many instances, allow a single component failure at the input to the combinational structure to produce multiple errors on the output of the combinational circuit.

4.3.1.4 Control Unit. The utilization of a microprogrammed control memory greatly reduces the problem of control unit error detection. Parity associated with each control word is very effective. However, accessing the wrong control word because of a failure in the addressing logic can cause catastrophic results.

Incorrect addressing can be caused by incorrect address decoding, which could result in the selection of the incorrect control word or the logical OR of two control words.

4.3.1.5 Packaging and New Technology. Before recommending solutions to the problem of error detection a few points will be made considering the technology with which the next generation of systems are expected to be built.

a) The cost of a system will be a function of the number of different types of circuit packages, and the external connection complexity (pin count) of each package, and be more or less independent of the complexity within a given package.

b) The interconnections between LSI circuits must be minimized to achieve a high level of reliability.

c) Functions such as arithmetic units will be integrated on one LSI chip.

An example of the cost of an error detecting code is the case of a register to register transfer. Information transfer from one register to another can be protected by the attachment of a parity bit. Considering present day MSI technology, an 8 bit register is packaged on one chip. Similarly, an 8 bit parity generator or checker is also packaged on the chip. Therefore, the parity logic constitutes just as many circuits as the register itself. This implies twice the logic to provide for error detection.

To obtain independent failures the error detection logic and the processing logic must be packaged separately. This implies that they cannot be manufactured on the same LSI chip. If error detection is accomplished at the level of the register, or arithmetic unit, then the error detection logic will consist of at least the same number of LSI packages as the functional logic.

One may consider the use of redundant registers with a comparator. This would require three times the logic.

4.3.1.6 Processor Configuration for Error Detection. From the discussions in the previous sections it is concluded that redundancy at the processor level is the most practical method of error detection. A comparator placed across the dual P elements provides the error indication. Some of the reasons for this choice are listed below:

a) Periodic software self-test cannot catch all failures before they propagate to multiple errors.

b) Error detecting codes cannot detect all possible errors.

c) It will require at least twice the logic, and subsequently at least twice the cost, to detect all possible single component failures in P. Therefore the cost of a dual P unit is reasonable.

d) The redundant processors can be packaged separately with independent power distribution. This will more closely meet the failure independence assumption.

e) Redundancy with a comparator at only one interface will reduce the number of interconnections between the redundant processors.

f) Errors are detected before bad outputs may propagate from the P. The comparator placed at the output of P might allow an error to propagate within P, but no bad information leaves P.

4.3.2 Error Detection in Memory

The choice of error detection techniques to be applied to memory is very dependent on the details of the memory hardware configuration and the technology. The purpose of this discussion is to present the failure modes which can occur in most "state-of-the-art" memory systems. Some generalized error detection mechanisms are suggested. Details particular to certain technologies or configurations (such as linear select plated wire memory) are not emphasized. Figure 4.3-1 depicts a generic memory system. Each of the major subsystems in the memory may fail in a number of ways. Transients may be induced at any of the indicated interfaces. One of the major tasks in designing an error detection system is to understand the results of the various failure modes.

4.3.2.1 <u>Control and Timing</u>. A failure within the control and timing elements of the memory system can manifest itself in a number of ways.

a) A read operation may be changed into a write operation, and vice-versa. Most dangerous is the destruction of the addressed word by an inadvertent write.

b) The memory may fail to sequence, thus preventing access or storage operations from continuing.

A failure in the control and timing logic can be detected by applying three principles to the design:

a) Separate the read and write control logic. This includes the control word. A simultaneous read and write operation

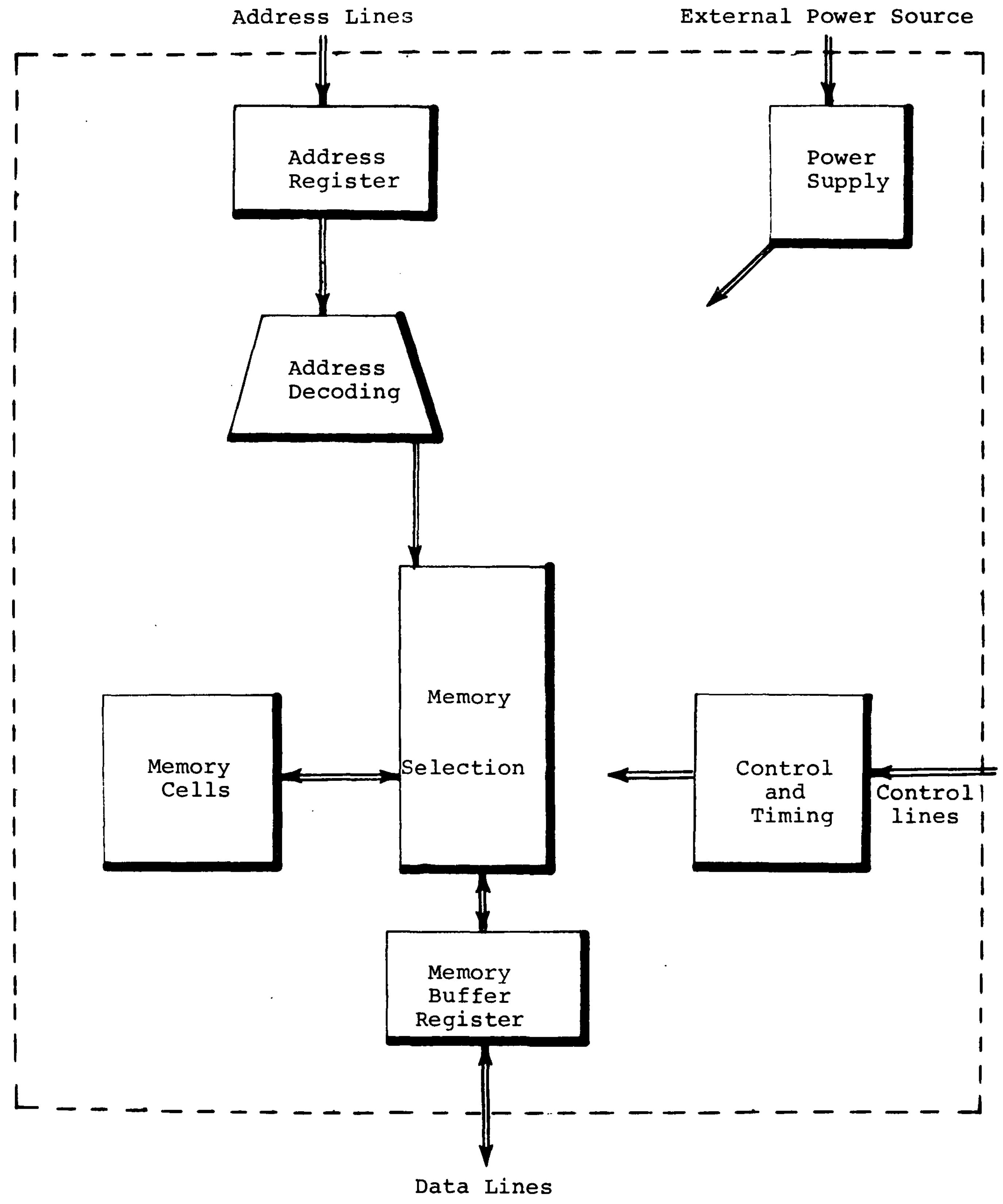


Figure 4.3-1: Generic Memory System

or a failure to read or write is detected as a control inconsistency. In order for a read to change to a write a specific simultaneous double failure most occur. This is of such low probability as to eliminate the situation from consideration.

b) Generate an end-of-read or end-of-write operation when the sequence has terminated.

c) Detect the nonsequencing of the timing element (a time-out error indicaiton must be generated).

These principles are illustrated in Figure 4.3-2.

4.3.2.2 Address Register and Decoding. A failure in this logic can cause:

a) the wrong word to be read.

b) a word to be written in the wrong location.

c) a word to be written simultaneously in multiple locations.

A number of detection methods, each applicable to different memory structures, are presented.

Method 1: Address Feedback

In a coincident current core memory a 3D selection scheme is employed. If the results of the address decoder are tapped at the end of the selection chain, they can be re-encoded into the original address and compared against the address stored in the address register. This is illustrated in Figure 4.3-3.

This technique essentially detects addressing failures in the decoding and selection network. If a flip flop in the address register failed in a "stuck at 1 or 0" state this technique would not work. Parity checks upon the address word stored in the address register would be required to detect the failure of a single address bit.

For a 3D memory with N words, $2\sqrt{N}$ lines would be fed back into the encoder. For a 2D linear select memory all N lines would be fed back. A 1024 word memory module would require a total of 64 feedback wires for 3D and 1024 feedback wires for 2D.

This principle has been applied in the design of the SIRU computer developed by M.I.T. [26]

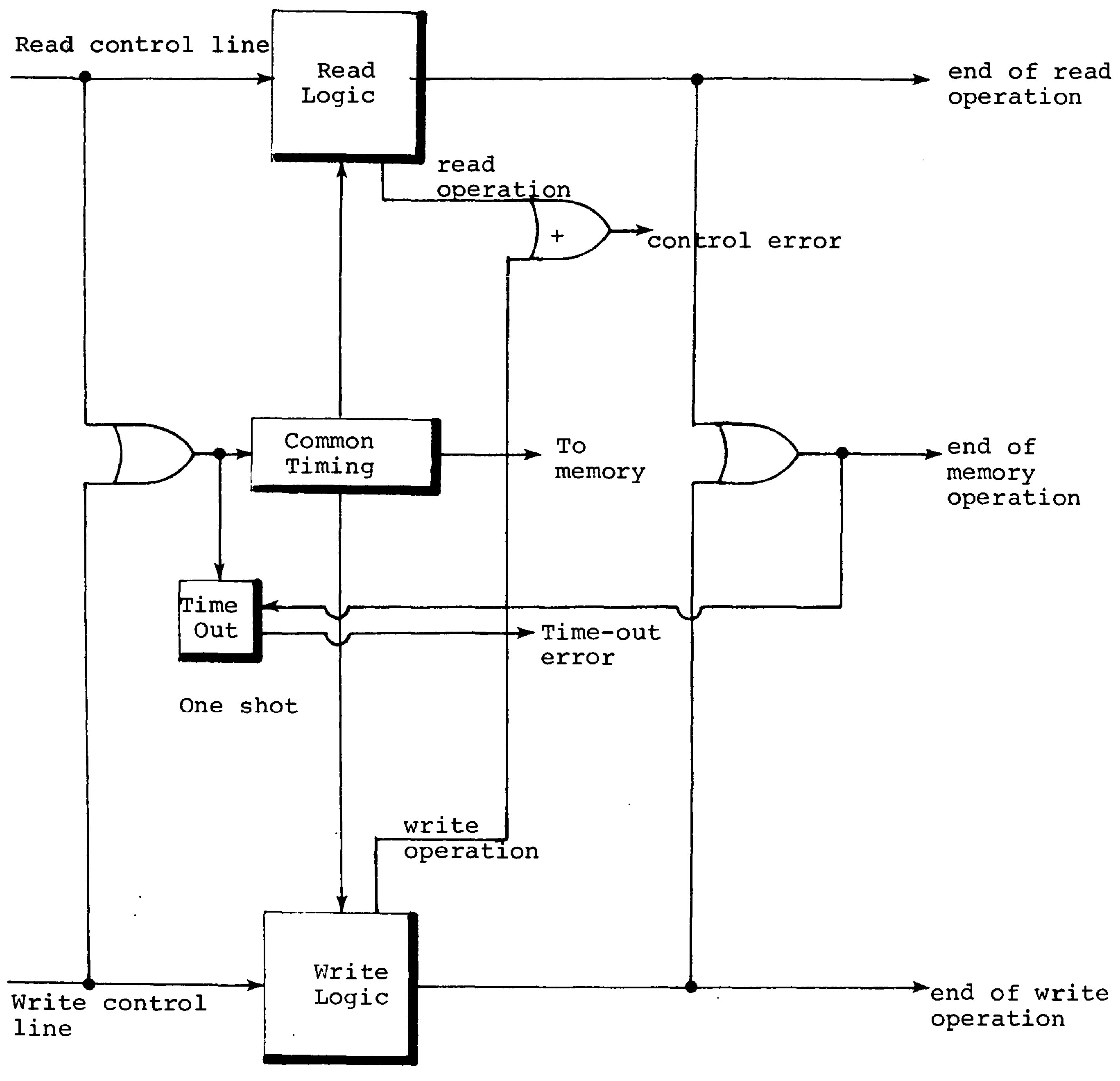


Figure 4.3-2: Memory Timing and Control

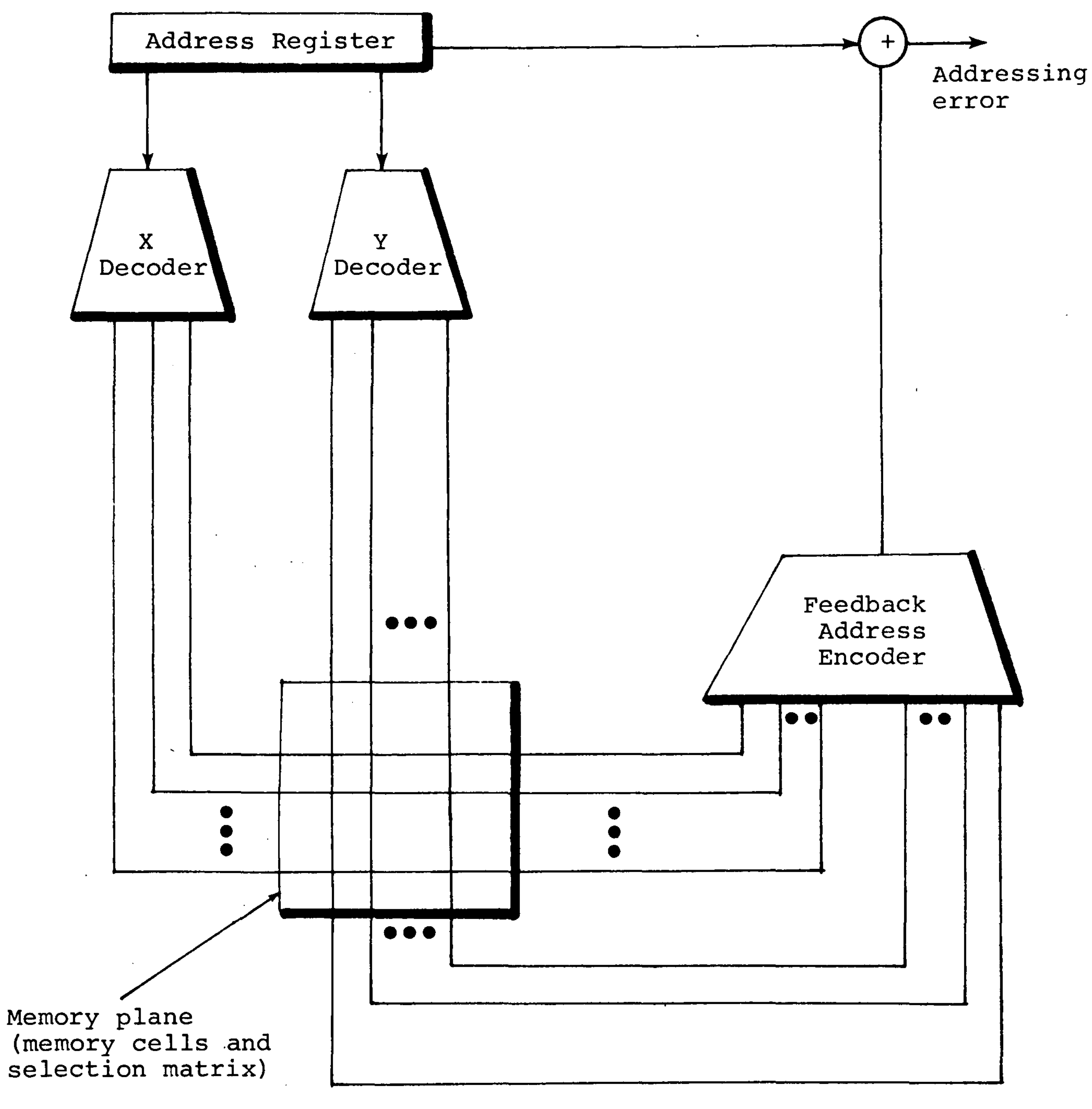


Figure 4.3-3: Address Decoder Feedback

Method 2: Bit plane partitioning with self-contained address decoding

This technique is most applicable to memory systems which exploit the newer LSI memory chips. These memories are bit oriented and possess internal address decoding within each chip. For example, a 1024-bit memory chip may contain either 1024 1-bit words or 256 4-bit words. The former configuration best suits our needs although the principle can be applied to the latter configuration.

Figure 4.3-4 illustrates a memory system consisting of 1024 x 1-bit chips in which address errors occurring after the address register are detected by simple word parity. The failure of the address decoding and selection logic within the memory chip can only manifest itself as an error in a single bit. This desired result is achieved essentially by an N fold redundancy of the addressing logic. This redundancy, however, is an integral part of the integrated circuit logic and is, therefore, not a cost factor.

The principle of bit partitioning with separate address logic for each bit can be easily applied to magnetic memories. However, the cost is the fabrication of N sets of addressing logic.

If 256 x 4-bit memory units are used then the failure of the address logic within the memory unit can result in the loss of four consecutive bits. Simple parity won't detect this failure. However, four parity bits can be employed.

Method 3: Storage of Address with Data

This method relies upon the fact that for many applications an addressable entity consists of a large number of bits. The number of bits addressed must be much larger than the number of bits needed to perform the addressing. This situation clearly exists in the drum storage device where 512 or 1024 words comprise an addressable entity. In this case one of the stored words will contain the physical address.

When information is read from memory the address contained in the information will detect addressing errors. When a word is written into memory it must be re-read to verify that it was stored properly. However, a permanent failure in the addressing logic could have caused the information to be written and retrieved from the wrong area of memory. Only when the destroyed information is requested will the error be detected. This is probably satisfactory since the destroyed data was not required until this time.

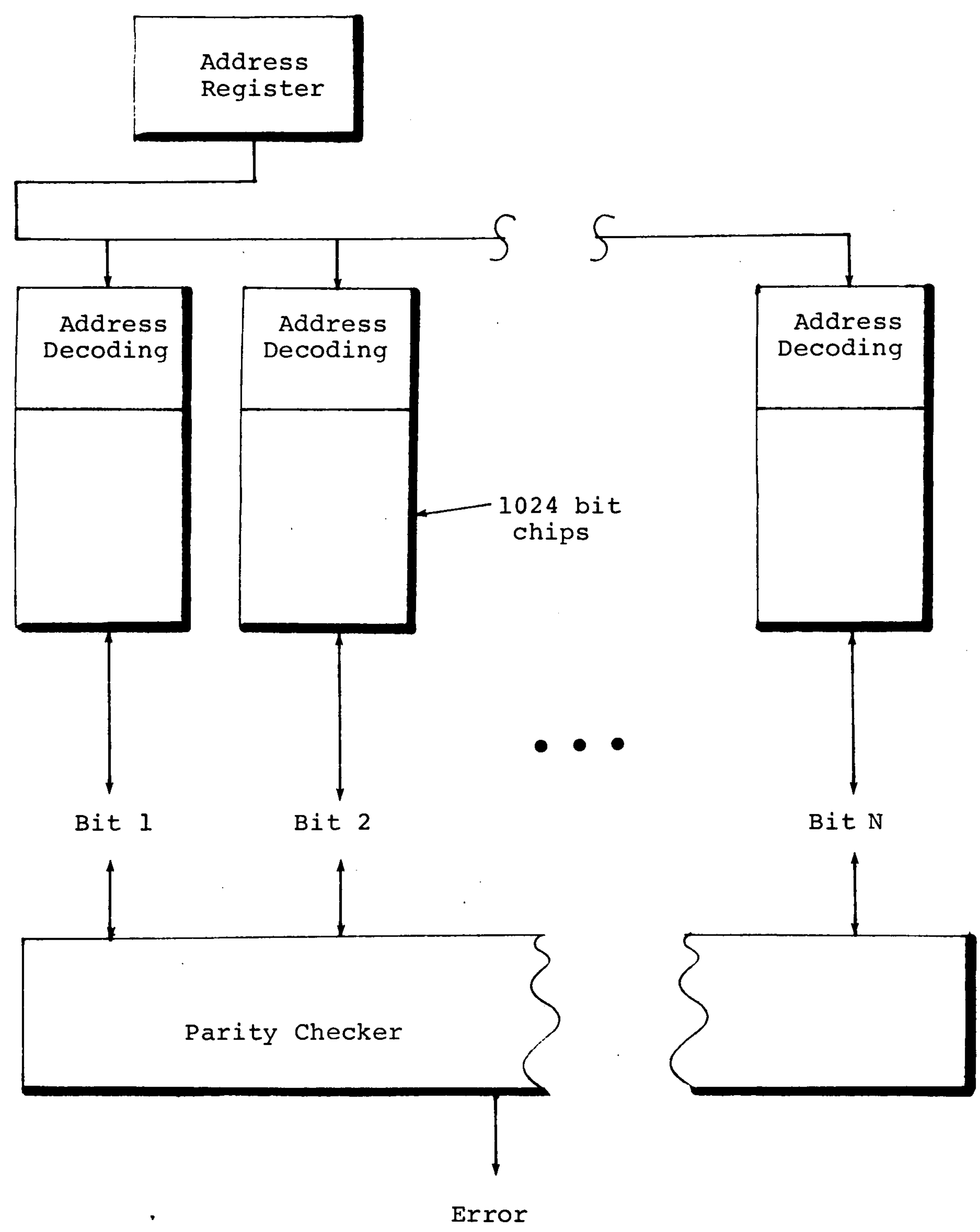


Figure 4.3-4: Bit Partition 1024 Word Memory

Method 4: Current Summing and Address Parity

Szygenda and Flynn [21] have suggested two interesting techniques involving error detection and recovery in a complete memory system. The following two error detection techniques relate to addressing failures. To protect against addressing failures, which could possibly cause nonrecoverable multiple data errors, two conditions are sensed:

a) More than one of the address decoder outputs becoming active during a memory operation. This is detected with a special current summing circuit on the address drive lines. This circuit indicates when more than one address line is selected at a time. See Figure 4.3-5.

b) The failure of a single bit in the address register. This is detected by a parity indication of the address word included in the storage of each data word. The address parity check detects the failure during the read check operation. If the bits of the address word are denoted by Ai (i = 1 ... n), then the address parity bit Ap is given by

$$Ap = \overline{A1 \oplus A2 \oplus \ldots \oplus An}$$

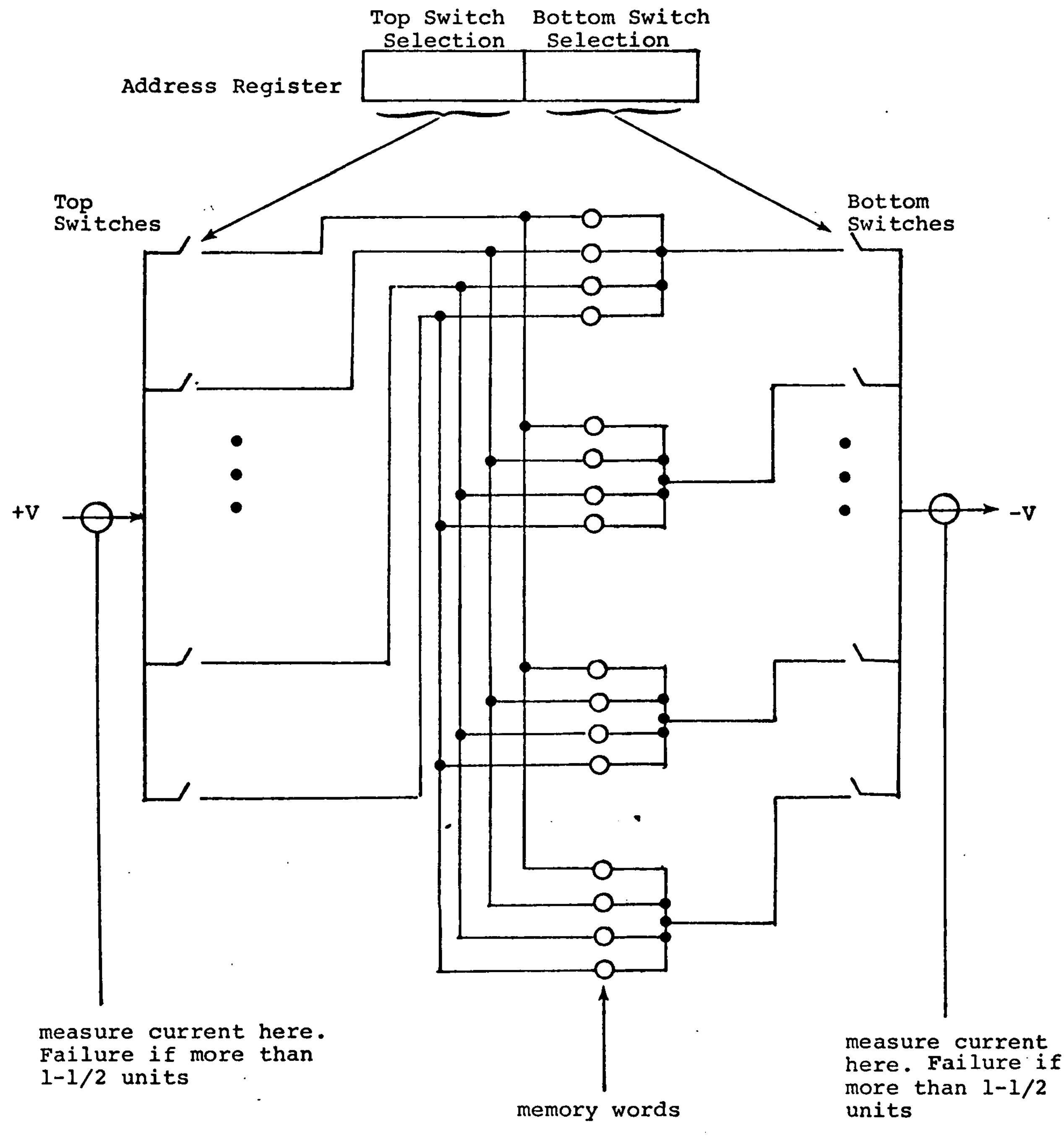
$\oplus$ = exclusive or

A single bit addressing errors in bit i causes Ai to be complemented ($Ai \rightarrow \overline{Ai}$). Since Ap is assumed to be unaffected by the error, address error detection is accomplished by an address parity bit check.

4.3.2.3 Memory Cells. Memory systems are prone to single bit errors due to component failures. However, electrical transients during a write operation will usually destroy the word being written.

Single bit errors are detectable by simple parity. The solution to the generalized burst error is more complex. However, if a few assumptions are made the solution becomes more tractable.

a) Once information is stored in the memory cells, the most common manifestation of failure will be the single bit error.



Notes
1) Only one top and one bottom switch are selected at a time.

2) Each top switch and each bottom switch draw one unit of current when they are selected and zero units when they are not selected

Figure 4.3-5: Current Summing for Linear Select Addressing

b) Burst errors are caused mostly by transients during the read or write operations.

c) A less likely cause of burst errors is the failure of a control line which will cause the entire memory word to go into the all "0" or all "1" state.

Two features incorporated into a memory system will enable most of these error modes to be detected.

a) Simple parity with one parity bit per word.

b) Write verification which reads each new word after a write operation. It is estimated that at worst a 10% degradation in memory speed would ensue due to write verification.

4.3.2.4 Memory Buffer Register. The Memory Buffer Register is logically bit oriented and a failure is detected by parity. Depending upon the technology employed, single bit errors will be more or less independent.

If the register bits are designed into the memory bit planes, then independence of failures is made more realistic. However, if an 8-bit MSI register chip is employed, a failure can make 8 bits bad and, therefore, independence is not achieved. In this case 8 parity bits would be required. One of the major functions of the logic designer of a fault tolerant system is to partition his logic design so as to guarantee that the inputs to a particular error detection mechanism possess statistically independent failure modes.

4.3.2.5 Memory Configurations for Error Detection. The previous discussion has dealt with memory failure and error detection in general. Specific recommendations for the three levels of memory within the multiprocessor are now presented.

4.3.2.5.1 M1 Local Memory: The following features are suggested for M1 error detection:

a) Bit plane partitioning with either 96 chips of 64 bits each or 48 chips of 128 bits each. The latter requires two word parity bits to guarantee independence of failures.

b) Address parity incorporated into each word.

c) Since each P-M1 processing unit is duplexed for recovery (this is discussed in Section 4.4), the inputs to the two M1's are compared. So is the read/write control line signaling.

4.3.2.5.2 M2 Operating Memory: The M2 memory will include the following features:

a) Current summing for a double address selection

b) Address parity bits

c) Word parity bits

d) A read verification of each write operation

e) Special control error detection logic

4.3.2.5.3 M3 Mass Memory: The following error detection features are proposed for the M3 level of memory:

a) Since accesses to M3 are usually in large numbers of words the physical M3 address can be stored along with each block of data without increasing the storage overhead appreciably.

b) Word and block parity can be applied to each accessed block.

c) Special control error detection specific to the particular M3 device failure modes is advisable.

4.3.3 Error Detection within the I/O Controller

As far as error detection is concerned there is enough non-memory like logic contained within the I/O controller so that the discussion concerning the processing unit also applies to the I/O controller. Therefore, as far as detection is concerned, the I/O controller should be dually redundant with comparators used for error detection. The M2 interface will be the same as for a processing unit. The IPCB is dual, although the function of P and I/O in relationship to control of the IPCB is quite different.

4.3.4 Error Detection on the Internal Bus

The internal bus consists of a data transmission path and a control structure. Parity is very effective in detecting errors incurred in the data path, especially in addition to any M2 parity bits contained within a word.

The failure of the bus control logic can result in either a non-transmission of information or a transmission to an incorrect module. A non-transmission can be effectively detected by a hardware time out error indication. The steering to the wrong module is detected by incorporating the M2 module number into each transmission, whether to or from memory. The processing unit is aware of the M2 module it is communicating with.

4.4 Recovery

This section presents an analysis of the problems associated with recovery from a failure, after it has been successfully detected. Major emphasis is placed upon the processing unit and the operating memory.

4.4.1 Recovery from a Processing Unit Failure

For the purpose of the following discussion a processing unit of the multiprocessor consists of the processor, P, and the local memory, M1. The P element includes the arithmetic, logical and control mechanisms. The M1 contains the top of the stack, base registers, status and other locally stored information.

4.4.1.1 Alternative Recovery Techniques. Five possible recovery techniques are described in this section. The basic differentiating factor is the recovery time. That is the amount of recomputation which must be performed in order to bring the system back to the proper operating state. All five recovery techniques, to be successful, must possess the following properties.

a) A restart point, constituting a valid state of operation, must be established either by hardware or software. None of the information needed to restart can be modified during the execution of the restartable entity. Only temporary information, local to the restartable entity, may be destroyed.

b) The restartable entity must be divided into two distinct restartable phases: Phase 1 involves input of data,

calculation and the utilization of a temporary memory area for storage of results. Phase 2 involves the updating of data in memory by means of a copy cycle from temporary to final storage. Only after the updates are completed and a correctness verification is performed will the temporary storage be released.

c) The mechanism used to generate restart points must be absolutely independent of the applications program, so that testing of the restart point generation mechanism can be performed, independently of the application.

d) Whatever the recovery mechanism, it must be designed to be reentrant. That is, it must be capable of being repeated, from the beginning, without loss of information. If one assumes that the processing unit is incapable of computing properly during a transient, then a restart attempt during a long duration transient will fail. In some sense one may think of a long duration transient as a permanent failure. A second failure indication during a restart attempt should not eliminate the possibility of a third or even fourth restart attempt after an automatic reconfiguration of the computing elements.

e) As a general principle the I/O controllers must be designed to assume an appropriate share of the recovery process. Such functions as being able to determine the state of an I/O device as well as the process which last commanded the device must be incorporated into the I/O controller.

4.4.1.1.1 Fresh Start: The concept of a system Fresh Start assumes all information stored in memory may be destroyed, new programs and their initial condition data loaded in, and the system started at some fixed point. This is a satisfactory approach for a batch processing system. The restart point is always the same, namely the reset state of the system.

The properties discussed previously are guaranteed as follows:

a) The restart point, namely the reset state of the processing unit, is fixed. The only information needed for restart are the programs and certain initial data. All other data is created during execution.

b) The two phases of operation are satisfied trivially since all information created is considered temporary.

c) The recovery mechanism is completely determined before run time.

d) The mechanism of Fresh Start can clearly be re-executed an arbitrary number of times.

e) Fresh Start implicitly assumes that all I/O can be reset to its starting state, or can be recycled with impunity.

The real time aspects of the space station environment make the concept of system Fresh Start impossible. Either the applications programmer must consider the impact of fresh starting at any arbitrary point in time, or the system must be capable of sustaining very long recovery times.

4.4.1.1.2 Checkpoint Restart: The recovery time may be made more deterministic by employing a checkpoint restart mechanism. In large computational facilities, restart points are often established by taking complete core and register dumps at fixed times. The time between checkpoints can be of the order of a few minutes up to an hour. This system dump philosophy requires time for the dump and a backup storage device such as a tape or disc.

Real time is usually not important in a checkpoint restart system: the average system throughput is being safeguarded.

The recovery properties are achieved as follows:

a) Restart points are established at periodic times when complete memory and status snapshots are taken. The most recent snapshot is not destroyed during subsequent processing.

b) The two phases of operation basically are implemented as follows:

   1) Phase 1 uses the information just dumped and modifies the M2 copy of the information during its computational processes.

   2) Phase 2 is actually the snapshot. The snapshot, being generated during phase 2, must utilize a different buffer area than was used for the last snapshot. Only after the present snapshot is complete can the buffer area associated with the previous snapshot be released.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

c) The restart point generation and recovery mechanisms clearly are independent of the applications program. But, the implication of a long recovery period impacts the systems analyst.

d) Restart may be tried an arbitrary number of times from the last checkpoint.

e) I/O interaction problems are not solved by this technique. However, I/ O is a separable question.

Although well defined, checkpoint restart will not, in practice, be able to meet a recovery time requirement of 10 to 100 milliseconds.

The checkpointing of M1 and the status of P will require less than 200 microseconds per P-M1 unit. However, one can not in general checkpoint a processing unit without a simultaneous checkpoint of the entire M2 memory. The validity of this statement becomes clear when one realizes that the state of P-M1 can not be returned to some time in the past and be expected to operate with the present state of M2. The two states are, in general, inconsistent. The major time factor is, therefore, associated with dumping M2.

If we assume 16 - 8K M2 modules and allow a snapshot rate of one 32-bit word per microsecond, an M2 dump would require 128 milliseconds. Thirty-two megabits per second is a higher rate than a standard drum, disc or tape can handle. However, let us assume for a moment a satisfactory backup device is available. During the dumping process of 128 milliseconds, no other execution can take place.

If a 10% degradation factor is arbitrarily assumed then one can not snapshot more often than every 1.28 seconds. Using these numbers, one sees that in order to get the system back to the state at which the failure occurred could require 1.28 seconds of processing.

4.4.1.1.3 Software Restart: The software system may be conceived of as a grouping of algorithms, each executed sequentially in time. Each algorithm receives data from its calling sequences and returns the results. In order for an algorithm to be restartable the following information must be available:

a) The algorithm in process and the starting point address of the algorithmic code.

b) All the information initially required by the algorithm.

The recovery properties are achieved as follows:

a) The restart point must be determined by the compiler and/or the operating system dynamically during execution in order for the recovery process to remain independent of the applications programmer. Whether this can be accomplished practically is a major question which will be touched upon in discussion of problem areas.

b) The two phases of operation, calculation and update, must be inserted into the machine code by the compiler.

c) Points a) and b) imply that the compiler and the operating system can relieve the application programmer of concern with the recovery process.

d) There does not seem to be a reason why a restartable process can not be restarted a number of times.

e) Although the I/O problems are not specifically handled by this solution it is felt that a better understanding of the I/O problem can be achieved because the restartable process is a logical executable entity and is, therefore, aware of its own I/O requirements.

In the past real time systems have relied upon software restart as the basic recovery mechanism. Software restart was employed in the Apollo Guidance system [25]. It was a major contributer to the magnitude and expense of the software checkout effort. A number of problem areas which are the basic causes of software restart complexity are discussed next. It should be noted that they exist whether the application programmer is involved or not.

a) Problem Area 1: Data Sharing

1) Self Dead-Lock

When two or more processes are allowed access to a "critical section" of code, a locking mechanism must be provided. A "critical section" as defined by Dijkstra [23] is a shared section of code that may be executed by only one process at a time.

An example of a critical section is the updating of a data segment in a COMPOOL. Suppose the following static program segment is constructed:

Statement Number	Statement Function
1	Input
2	Calculate
3	Wait till lock is not set
4	Set the lock
5	Update data (critical section)
6	Release the lock
7	Continue
8	End

The locking mechanisms, implemented properly, will provide the means for controlling the proper update of the common data when two asynchronous processes desire access to the same critical section.

Suppose that the code described above is excuting in the middle of step 5, a failure occurs, and restart is attempted from step 1. The dynamic execution of statement steps will proceed in the order [1,2,3,4,5, failure, restart, 1,2,3, dead lock]. After the restart execution the system finds itself locked out of the critical section because the lock was not released before the restart was initiated.

One may, of course, invent various mechanisms to release all locks before a restart is initiated. This can become quite complex. One may also store the I.D. of the process with the lock so the lock applies to all processes except the process which set the lock in the first place.

2) Externally caused Deadlock

One may conceive of situations where it is desired not to restart a particular process. The system design specification requires that the process that failed be terminated but other processes continue. In this situation the terminated process can still leave a lock set which causes a different process to enter a deadlocked

situation. One may consider this situation bad design practice or a bad specification, but nevertheless possible.

b) Problem Area 2: Preservation of All Starting Data

It should be clear that in order to restart an execution sequence, its input data must not be destroyed or modified. That is, all information that is "read" from memory during execution must not be allowed to be written into by the restartable sequence. This requirement gives rise to a number of restrictions or problems that the system designer must face.

1) The N = N + 1 Problem

Assume the following program:

```
N = N_old
Call Procedure A (with parameter N)
Continue
Step
  1          Procedure A         Calculate
  2                              N = N + 1
  3                              Calculate
  4                              Return to caller
```

If procedure A is made a restartable entity then a failure during step 3 will cause the following dynamic sequence to be executed:

1, 2, 3, failure, restart, 1, 2, 3, 4

At the end of Procedure A (step 4), $N = N_{old} + 2$. However, the proper value at this point should have been $N = N_{old} + 1$.

One may argue that the beginning of Procedure A is not the correct restart point. This is true. Restart should have commenced at the $N = N_{old}$ statement. However, without the application programmer specifying the restart point it is not clear that the compiler or dynamic execution situation has enough information to determine the restart point.

Clearly many software routines may call Procedure A. Therefore, a static determination of the restart point is impossible.

One could, of course, program the same logic without using N = N + 1 as a legal statement. For example:

```
 N = N_old
 Call Procedure A (with parameter N)
Procedure A      Calculate
                 N_new = N + 1
                 Calculate
                 Return to caller with answer N_new

N = N_new        (This is a copy cycle)

Continue
```

As far as N is concerned Procedure A is now restartable. Restartability has been achieved by passing the parameter N to Procedure A, having Procedure A return the parameter N_{new}, and finally executing a copy cycle $N = N_{new}$.

2) Very Large Data Elements

The basic mechanism involved in preserving information required for recovery is the use of a copy cycle. All storage must occur in a temporary area of memory and only the calling routine can change the starting values; namely, perform the copy cycle. If very large arrays are involved in a particular calculation a practical problem might arise as far as memory size, and copy cycle time are concerned.

3) Test and Set

The Test and Set instruction often used for entering "critical section" violates the principal of not destroying the starting information. As a matter of fact, any locking mechanism does. The solution to the locking problem presented in section 4.4.1.3.2 involves a hardware special mechanism to allow Test and Set (or other locking instructions) to be separate restartable entities. Test and Set must be executed in two phases as discussed in Section 4.4.1.1.

c) Problem Area 3: Interacting Processes

1) Spawning a Parallel Process

Consider the following software sequence

Step		
1	Start	Input
2		Calculate
3		Schedule Procedure A
4		Continue

Step 3 schedules Procedure A. It is assumed that the multiprocessor executive will handle the details of the scheduling. Procedure A will eventually be bound to a processing unit for execution. If a failure occurred during step 4, a restart is assumed to begin at step 1. Notice that Procedure A will be scheduled every time step 3 is executed.

In this situation the schedule command actually alters the state of the system between the initial execution of step 1 and the step 1 execution triggered by the restart mechanism. Procedure A has been scheduled more than once.

One of three machanisms must be invoked to solve this problem.

i) Realize that step 1 is the incorrect restart point for a failure during step 4.

ii) If step 1 is the restart point than all schedule commands issued during the procedure should be voided. In general, this is very difficult since Procedure A might be in various stages of execution. It might have been completed. This is a problem introduced by multiprocessing.

iii) The schedule statement could be skipped over during the restart attempt.

2) Forking and Joining

Quite often many processing units are employed in the execution of a single calculation, so as to exploit intrinsic parallelism. An example would be a large correlation or statistical analysis calculation. In this situation the initial process has forked into a number of parallel processes. These processes join at various points so communication can take place. A failure occuring during the execution of a forked process will, in general, interact with the other branches of the fork. The dynamic

interactions can make it very difficult, in general, to undo certain partial results before restart.

d) Problem Area 4: Real Time

The ability of a process to recover within a specified "recovery time" is of importance. One may argue that it would be very difficult to determine the worst case recovery time because it is dependent upon the process being executed at the moment of failure detection. This is true. However, the Apollo program employed the concept of software restart and although the solution and testing of the solution was complicated, real time requirements could be met. For this reason the concept is not rejected on the basis of real time requirements.

4.4.1.1.4 Single Instruction Restart: M.I.T. [22] has proposed the concept of a Single Instruction Restart (SIR). The concept was developed to make recovery from hardware failures and transients transparent to the programmer. A fundamental tenet in the SIR concept is that all errors are detected within the instruction in which they first occur, so there is no propagation of errors to subsequent instructions. Each instruction is divided into two parts; a compute phase and a store phase. During the compute phase results are stored in a temporary buffer, and during the store phase they are transferred to their final storage location. Each phase is made restartable. When an error is detected the instruction phase in which the error occurred can be automatically re-executed, before potentially erroneous data overwrites good data.

The recovery properties discussed previously are achieved as follows:

a) Restart points are automatically determined as either the compute phase or store phase of each instruction. The use of a temporary buffer prevents the destruction of restartable information.

b) The basic two phases of execution are built into the implementation of each instruction.

c) Not only is the applications programmer not involved in determining restart points, he is not even concerned with the delay due to recovery time. SIR should impose a recovery time that will be small

compared to the iteration rate of any real time process.

d) Clearly each instruction phase may attempt restart any number of times.

e) Although not a solution to the I/O restart problem, SIR does indicate the error within the instruction execution. If I/O is handled by a separate I/O controller then this controller could buffer the I/O command and reject redundant commands during a restart attempt.

The proposed solution presented in section 4.4.1.2 does utilize a number of the SIR concepts.

4.4.1.1.5 Fault Masking -- Triple Modular Redundancy: The last techniques to be discussed imposes the least recovery time. In a sense it is the other extreme from Fresh Start. TMR utilizes three identical processing units and votes upon their outputs. The majority output is assumed to be correct and the faulty processing unit is masked.

TMR meets the recovery requirements in a trivial way:

a) Restart points are not used. In a degenerate sense the restart point is the present time. Because of the masking effect of TMR, restart information, as such, is not required.

b) Retry is not required and therefore the phasing and buffering of restartable entities are not an issue.

c) TMR is completely transparent to the programmer.

d) TMR masks transients.

e) Since the voter outputs are assumed to be valid, TMR does not result in problems associated with I/O.

Three configurations which employ majority voting to mask failures will now be discussed. A number of groundrules and assumptions are used as a basis of discussion.

a) Even though three P's and/or M1's are used, the interface to M2, the interprocessor communication bus are dual redundant and bidirectional buses, implying an interface between different levels of redundancy.

b) The main purpose in considering majority voting is to eliminate the need for single instruction restart.

Figure 4.4-1 presents a number of alternative configurations which employ majroity voting. Configuration 1 utilizes 3 P-M1 units and dual interfaces to M2 and to the IPCB's. The EC logic is used to automatically interpret the error signals and to reconfigure the system according to a preplanned strategy. The EC logic sets the position of the S's, shuts off V units, and generates fault interrupts.

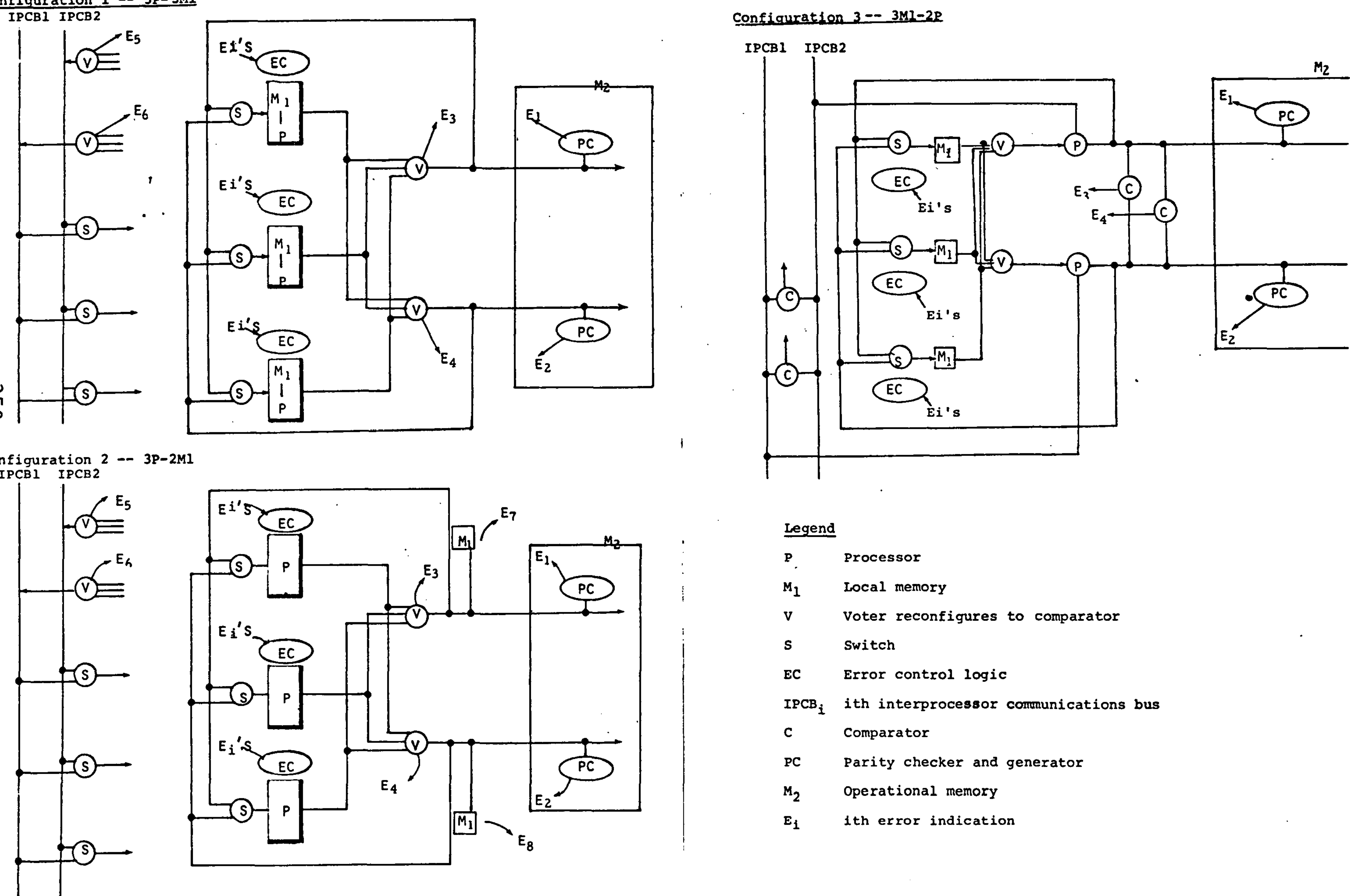
Configuration 2 differs from configuration 1 in that 3P's are used but only 2 M1's are employed. In this configuration the internal error detecting ability of M1 is utilized to aid in fault isolation. The EC unit in configuration 2 is more complex since it can control switching of M1 and M2 inputs.

Configuration 3 is presented for completeness and employs 2P's and 3M1's. When 2P's are employed voting and switching logic on the IPCB's are not required. However, 3 EC's are necessary to independently control the 3 S's.

The following discussion will demonstration that configuration 1 is the only one worthy of further consideration.

Configuration 2 has been proposed because it initially requires less hardware by the absence of the third M1 module. A more detailed investigation demonstrates that this is not so (see Figure 4.4-2). The voting and switching logic is identical and independent for each bit on an interface. In configuration 1 the V elements interface with M2, which is 32 bits wide. Therefore, each V element is 32 bits wide. In configuration 2 each V element is 64 bits wide because M1 is specified to be 64 bits wide. Therefore, V and S elements for the M2 interface in configuration 2 contain twice the hardware of configuration 1. In addition, self-error detection within M1 exists for configuration 2, and is not necessary for configuration 1.

Figure 4.4-1: Voting Configurations



Legend

P	Processor
M_1	Local memory
V	Voter reconfigures to comparator
S	Switch
EC	Error control logic
$IPCB_i$	ith interprocessor communications bus
C	Comparator
PC	Parity checker and generator
M_2	Operational memory
E_i	ith error indication

Factor	Configuration 1	Configuration 2
IPCB interfaces	3	3
No. of P's	3	3
No. of M's	3	2 + error det
Size of V	32 bits wide	64 bits wide
Size of S	32 bits wide	64 bits wide
EC complexity	a little less	a little more

Figure 4.4-2: Configuration 1 and 2 Comparison

Finally, estimates indicate that a 96 word 64 bit wide M1 is sufficient to support the instruction set proposed in section 2. Present day technology can easily produce 256 bits of M1 on one integrated circuit. Next year's technology will expand this to 1024 bits of M1 on one integrated circuit. The extra M1 module in configuration 1 consists of 6144 additional bits. Clearly the size and cost of the extra M1 module is small.

Let us look at configuration 3 for a moment. If the C elements indicate a P failure, then it is not possible to determine which P is at fault. It is presumed that information in the M1 modules is valid. In order for M1 information to be valid an error indication during an M1 write operation must terminate the write before execution. The only restart policy in this situation is to read all the M1 information, place it in another processing unit (2P-3M1 complex), and re-execute the instruction which was terminated by the error detection. This is exactly the SIR philosophy. The next section will show that the SIR philosophy can be implemented without voters, and with 2P's and 2M1's with internal error detection.

This discussion should demonstrate that configurations 2 and 3 only offer theoretical interest. If voting elements are to be utilized then configuration 1 consisting of three P-M1 units is the candidate.

4.4.1.2 <u>Proposed Configuration for the Processing Unit.</u> On the basis of the presentations made in section 4.4.1.1, certain configuration possibilities can be eliminated. Fresh Start and Checkpoint Restart are eliminated on the basis of real time requirements.

If software restart were to be implemented one would still have to provide hardware error detection and the generation of a processing unit fault interrupt. Figure 4.4-3 indicates the processing unit configuration required for hardware error detection and software restart.

If software restart were practical, in light of all the problem areas discussed, then it is assumed that enough information exists in M2 to restart the process on another processing unit. A need would still exist for some error control (EC) hardware to collect the error signals and communicate to the executive the failed state of the processing unit. This can be accomplished by initiating special M2 write sequences and/or communicating over the IPCB's.

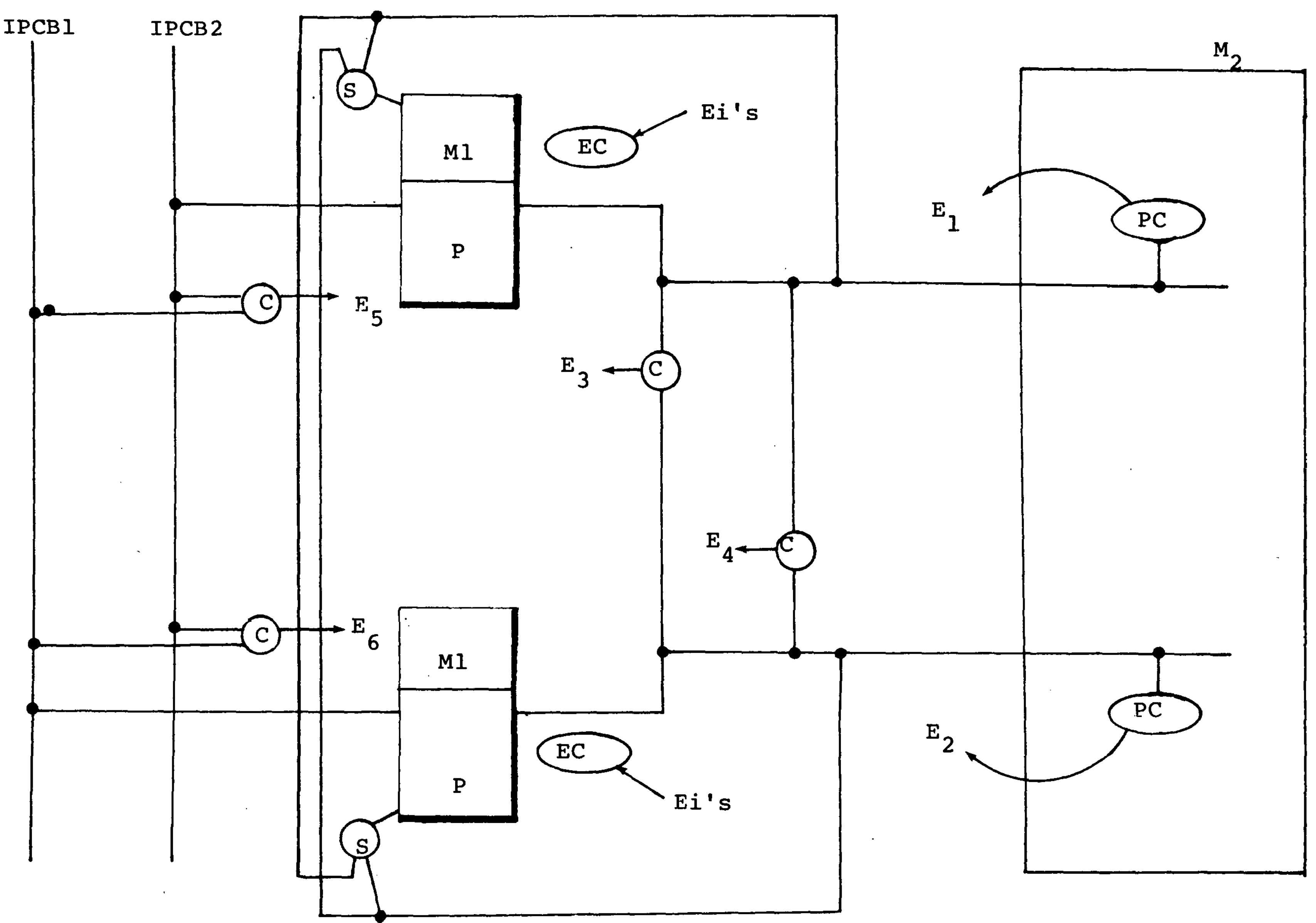


Figure 4.4-3: Configuration for Software Restart

For a little additional hardware many of the problems associated with software restart can be eliminated. Figure 4.4-4 presents the proposed configuration which employs SIR.

The cost of implementing any fault tolerant configuration requires an analysis of both the hardware and software aspects. It should be clear that software restart involves quite a complex set of software in order to automatically generate restart points. This cost could be quite comparable to the applications software itself. The hardware cost is not significantly less than the proposed technique employing SIR. See figure 4.4-5.

On the other hand, the additional software required by the SIR technique is not a substantial increment over the TMR approach whereas TMR hardware costs significantly more.

The proposed configuration employing SIR is the proper compromise between software restart with its very complex structure, and TMR with its complex hardware structure.

The proposed configuration shown in Figure 4.4-4 relies upon the following recovery philosophy.

a) Error detection is accomplished completely by hardware

b) After every instruction the complete state of the processing unit is contained within M1. This means that M1 contains an image of the program counter, as well as all the P element status which is used to communicate information across the instruction boundaries. The status information includes, among other things, overflow, information, execution phase information, interrupt enable information and the contents of any processor flipflop which is not in the reset state after the completion of an instruction. All the information necessary to restart is contained in M1.

c) The comparators and error detection logic is located and sequenced in such a manner that an error is detected before incorrect information is written into M1 and M2. Any failure occuring within M1 which results in bad information is detected during a subsequent M1 read operation.

d) All the error signals from both threads of the processing unit are sent to both EC's, which in turn, generate an interrupt over the dual IPCB's. The EC logic also causes the processors to stop execution of instructions.

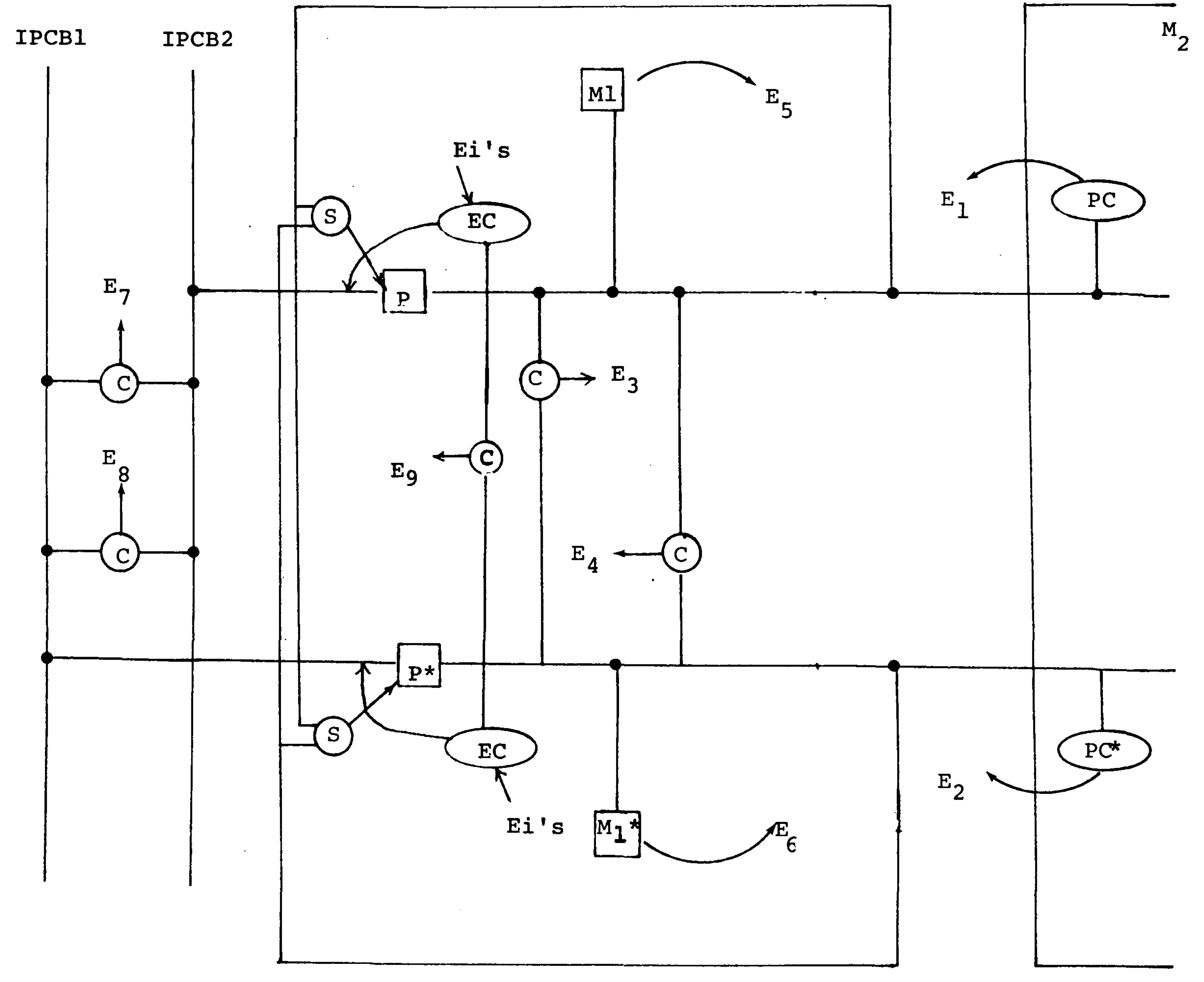


Figure 4.4-4: Proposed Configuration

	Voting	SIR (Proposed Configuration)	Software Restart
a) Hardware Aspects			
No. of P's	3	2	2
No. of Ml's	3	2	2
No. of S's	3-32input+int.	2-32input	2-32input
No. of C's	---	2-32input+EC**+int.**	2-32input+int.
No. of V's	2-32input+int.		---
E C Complexity	6 error signals	9 error signals	6 error signals
Int. Interface Complexity	2V's-3S's	2C's	2C's
b) Software Aspects			
Restart Point Generation	---	hardware	hardware
Software Execution Control to guarantee recovery	---	micro program	complex software
Transient Determination	hardware	software	software
Degraded Mode Determination	software	software	software
Restart Procedure	---	software	software
Failure Reporting	software	software	software
Difficulty in achieving independence from applications programmer	easy	easy	very difficult

Figure 4.4-5: Relative Cost Trade-off

e) Any other processor can respond to the interrupt. The E_5 and E_6 signals (see figure 4.4-4) are sufficient to indicate whether M1 or M1′ contains valid information. The interrupted processor issues a command to dump the contents of the good M1 into a fixed area of M2. During the dumping process the comparators used for P error detection are inhibited. In a sense, this dump is like taking a checkpoint upon the processing unit's state. The big difference compared with normal checkpointing is that, due to the redundant M1 and EC logic, the checkpoint can be taken after an error indication. Therefore, the overhead of continuous checkpointing is avoided.

f) A determination must next be made as to whether the error indication was caused by a transient or a permanent failure. After the M1 dump a command is issued to reload both M1's from the M2 dump area. Details of how this function is performed were given in the discussion of system recovery following a fault indication in section 3.7.

The hardware configuration shown in Figure 4.4-4 possesses the following properties.

a) All the elements run in a synchronized manner

b) Error indicators E1 and E2 indicate errors caused by transmission of information from P to M2.

c) Error indicators E3 and E4 indicate errors caused by faulty P element which manifest themselves as inconsistent address or data information being transmitted to M1 or M2.

d) Failures caused within M1 are detected internal to M1 and are indicated by E5 and E6. E5 and E6 also are used to provide fault isolation and indicate which M1 contains valid information.

e) The EC logic receives all the error indications, controls the M1 dump and signals the fault interrupt

f) Comparators across the IPCB's are used to indicate inconsistent interprocessor communication

g) Finally signal E9 compares the action of the EC. It is possible for an EC to fail in such a mode so as not to perform the M1 dump function as required. When this occurs the system functions properly. Only when a P or M1 fails is the EC required to execute properly. In this

case E9 indicates inconsistent EC action. This is a case of a double failure. Recovery is not required in this low probability situation. E9 is just used to generate a fault interrupt. If P or M1 fails, EC fails and E9 fails to function properly, a triple failure exists and no guarantee is made as to system performance.

h) A failure in an S element will ultimately result in an E3 or E4 error indication.

4.4.1.3 Technical Aspects of the Proposed Configuration. The following discussion attempts to provide solutions to various technical questions which might arise in implementing the proposed recovery scheme.

4.4.1.3.1 What is a restartable instruction? In order to design a processing unit so that SIR can be employed a number of groundrules must be applied during the design implementation of each instruction.

Each instruction must be partitioned into two phases. During phase 1 the instruction is fetched, data is read, computations are made, and results are stored in a temporary buffer area. In the proposed design this buffer area is contained within M1. The buffer area contains information to be written into both M1 and M2.

During phase 2 the buffered information is copied into its final destination in M1 and M2. The contents of the buffer area are not destroyed until all the copy cycles are completed and verified. Each phase is designed to be separately restartable. Figure 4.4-6 schematically represents the execution of a generic restartable instruction.

If a failure indication occurs during phase 1 then the old copy of the program counter indicates which instruction was being executed. None of the information needed to execute the instruction has changed, so phase 1 can be re-initiated. If a failure occurs in the middle of phase 2, then, even though some information might have been copied, the information temporarily buffered in M1 is still valid, and a complete re-initialization of phase 2 will be indicated.

Interrupt testing can either occur at the end of phase 2 or at the beginning of phase 1. It is assumed that all

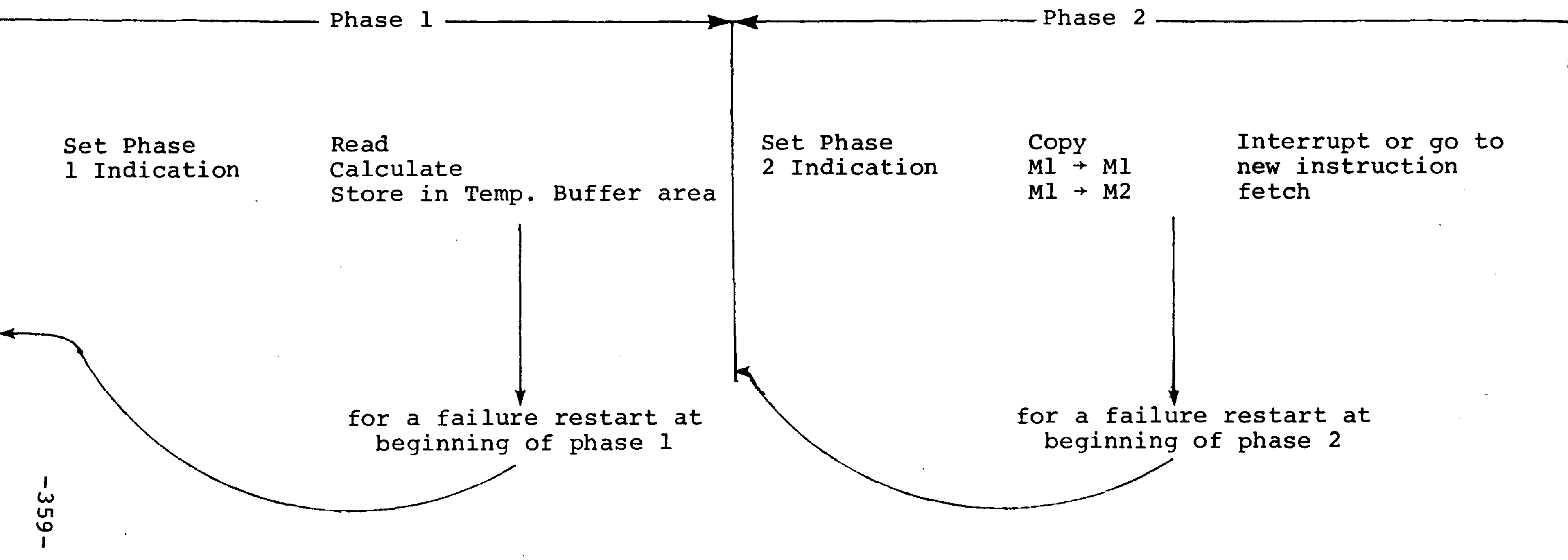


Figure 4.4-6: A Generic Restartable Instruction

interrupt conditions are caught in latches, so that the interrupt test is just a matter of reading these latches, and determining whether to fetch the next instruction in the instruction stream or to enter the interrupt control microroutine. The interrupt control microroutine is designed to be restartable and incorporates the concepts of a double phase operation with a buffer area. That is, the interrupt control microroutine can be considered to be a restartable instruction.

4.4.1.3.2 Locking: How does the proposed scheme resolve the problem of locking of shared data and the possibility of either self or external deadlock? These problems were described in section 4.4.1.1.3.

Two options for implementing a restartable-non-deadlocking-lock instruction are given below.

a) Option 1 -- Let M2 Execute It

This technique allows the addressed M2 module to do the physical execution of the lock instruction, upon receipt of a command. Basically the M2 module would be executing a read-modify-write instruction. The execution sequence proceeds as follows:

P	M2
1) Instruction Fetch	
2) Generate M2 address, command, value, and send to M2	
3) Wait for M2 response	
	4) Read M2 location, put old value in M2 buffer. Write new value into M2.
	5) Send value extracted from M2 back to the P unit
	6) The P unit which is waiting will respond
7) Input value from M2 and put on top of stack for testing	
8) Copy cycle	

This instruction implemented with M2 processing presents a problem in restartability.

If the processor fails during step 7 restart can not begin at step 1 since M2 information has already been changed. One would have to implement a double buffer technique with a copy cycle within M2 itself. However, if this is resolved then the instruction must be designed with three separate restartable sections. Phase 1 would consist of steps 1 and 2. Phase 2 would consist of step 3 (M2 step 4,5,6, with an M2 copy cycle) and step 7. Finally, phase 3 would be step 8.

Recall that we are not considering M2 failures in our discussion. However, in the final configuration, areas of M2 which can be modified will be dual redundant and hardware error detection will exist. If an M2 module fails during its phase of execution then the redundant module is assumed to be reliable.

During the M2 execution phase no other processing unit can access the memory module. The addressed module will be in a "busy" state.

If a failure occurs during execution of this lock instruction then quite possibly the lock will remain on the data (and module) until after the recovery process. It is anticipated that this wait will not exceed 10 - 100 milliseconds.

b) Option 2 -- Let the processing unit do it

This implementation does not employ a read-modify-write option for M2. It does however introduce four M2 primitive commands. The commands are

1) Read M2 -- This is a normal read

2) Write M2 -- This is a normal write

3) Read M2 and lock -- This command performs a normal read and locks the selected module so as to prevent access by all processors except the one that locked it.

4) Write and Unlock -- This command performs a normal write and releases any module lock.

The lock created by command 3 is very temporary in nature. It will be released after execution of the instruction. It is required to prevent another process from accessing the module.

The sequence for the execution is as follows:

	P		M2
1)	Instruction Fetch		
2)	Generate M2 address, read and lock command, and send to M2		
3)	Wait for M2 response		
		4)	M2 performs a read
		5)	Sends word to P
		6)	After P accepts word it locks module
7)	Processor reads in word, modifies it, puts it on top of stack and generates a write command with the modified word		
8)	Copy cycle for M1 and M2 writes		
9)	Issue unlock command during last write operation		

When a failure occurs during phase 1 (steps 1-7) the EC must perform a special function. It must signal the M2 module to release the lock which was set in step 6. This allows the instruction to be restarted at step 1 and eliminates the possibility of a self-deadlock.

Unlike option 1 a failure releases the lock and temporary external deadlock, possible during the recovery time, is impossible.

It is felt that option 2 is the superior implementation for a lock instruction, since the SIR implementation is more consistent.

4.4.1.3.3 Real Time: It should be possible after a detailed design to generate a worst case estimate of the maximum recovery time for the highest priority process. At present M1 is, at most, 96 words of 64 bits each. If M2 can accept 32 bits per microsecond then the dumping of M1 into M2 would require 192 microseconds. The restoration of the dumped data into another M1 would also require 192 microseconds. The software required to determine when to restart should not exceed five milliseconds. Therefore, a high priority process should be able to recover in less than 10 milliseconds.

4.4.1.3.4 Initiating a Restart: Since the initiation of a restart, from a permanent failure, has been forced to be the same as a return from an interrupt, the steps to be followed include:

a) Restoration of M1 and M1′

b) Take status from M1 and place in SR1 and SR2

c) Reconstitute the base registers, program counter, and stack control words.

d) Clear the contents of the CAM by making all the locations vacant. (The contents of the CAM are built up dynamically during execution. They are not needed for restart.)

e) Execute the next instruction fetch and continue processing.

4.4.2 Recovery from M2 Failures

This section addresses the problem of recovery from an M2 failure. The mechanisms to be employed in detecting M2 errors were presented in section 4.3.2. After error detection by M2 and the S interfaces of the processing units certain actions must take place in order to provide a satisfactory recovery situation. The following discussion is divided into three parts. The first deals with the action required for the various failure modes. The second presents various alternative configurations which could be employed to achieve recovery from an M2 failure. The third part presents a more detailed investigation of the proposed recovery technique.

4.4.2.1 Modes of M2 Failures. The various failure mechanisms presented in section 4.3.2 create errors which can be segregated according to the action which must occur in order to recover from the failure.

Type 1. The type 1 error is the result of a transient during a memory operation which does not create incorrect memory state information. After the transient has subsided a re-execution of the particular memory command will produce satisfactory results. No damage to the hardware has occurred, and the contents of all the memory cells (the state of the memory) is valid. An example of a type 1 error is electromagnetic noise coupling into a memory sense amplifier or even into a write drive line. A re-read or re-write will correct the error.

Type 2. The type 2 error results from a failure which causes incorrect memory state information. The hardware still performs satisfactorily but the memory state is altered. Exact information concerning the cause of the error is either unavailable or has been destroyed. An example of such an error is an electrical transient in the addressing structure during a write command. Information is written into an incorrect location. However, after the transient it is impossible to determine which location has been destroyed except by doing a complete memory search, and even that may not be entirely successful.

Recovery from type 2 transients requires back up information stored in a physically different device. The state of the memory module which suffered the error could be restored, or the back up information could be used directly.

Type 3. The type 3 error results from a permanent hardware failure. The contents of memory cannot be accessed successfully. Back up information must be used and the bad memory module configured off line and repaired. The system must then enter a degraded mode of operation.

4.4.2.2 Alternative M2 Recovery Configurations. This section will describe briefly a number of different configurations, which provide failure recovery within M2. As a general principle recovery from a memory failure requires the redundant storage of information, which may either be contained in M2 or M3. The different configurations to be discussed differ basically in the back up media, and the complexity and overhead involved in anticipation of recovery from an M2 failure.

1) Configuration 1 -- Completely Dual M2 Complexes

This configuration assumes that the entire grouping of M2 modules possess a redundant image, denoted by M2'

See Figure 4.4-7a. As discussed previously, each processing unit consists of dual P's with a redundant internal bus interface. Configuration 1 allows each half of the dual P to communicate simultaneously with independent redundant M2 modules. Recovery is accomplished by just using the correct M2 module after the faulty module is isolated. Recovery is very quick, almost instantaneous. The main drawback to this configuration is that twice as much M2 is required.

None of the existing mechanisms of the virtual memory system are exploited. All programs and data possess redundant copies in M2. In addition, the virtual memory system possesses copies in M3. The initial impression of this configuration is that of hardware overkill. Namely, it is possible to achieve the same goal with a less expensive system.

2) Configuration 2 -- M3 Backup

This configuration utilizes M3 as the location of all backup information. Periodically, the system is stopped and M2 is dumped. After the dump, all modifications to M2, (write operations) are entered into a synchronizing buffer and finally stored in M3 (See figure 4.4-7b.

Recovery is a matter of reloading the M2 dump from M3 and re-executing all the recorded write operations.

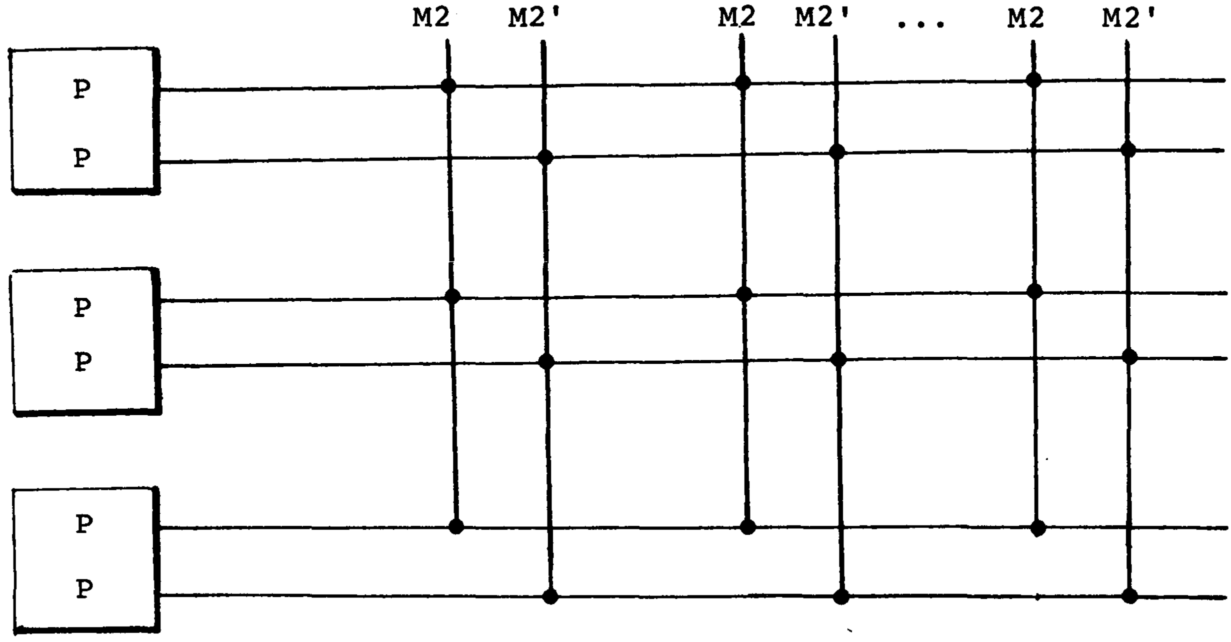
Although the hardware is less costly there is overhead in terms of M2 dumping and storing of all write operations into M3. Dumping of M2 could require 100 ms. The restoration of M2 after a failure would require the spare M2 module to be switched in and the failed module to be switched out. The reload would require 100 ms plus the execution of all the write commands since the time of the snap shot.

3) Configuration 3 -- Duplex Addressing

This scheme allows both M2 and M3 to be used as backup media. M3 is used to backup most program segments. M2 is used as the backup for data and possibly certain critical program segments. Program and Data Segments can be stored anywhere in M2. When space is assigned to a data segment two "holes" must be found in

Figure 4.4-7: M2 Recovery Configuration

a) Configuration 1 -- Completely Redundant M2 Module



b) Configuration 2 -- M3 Backup

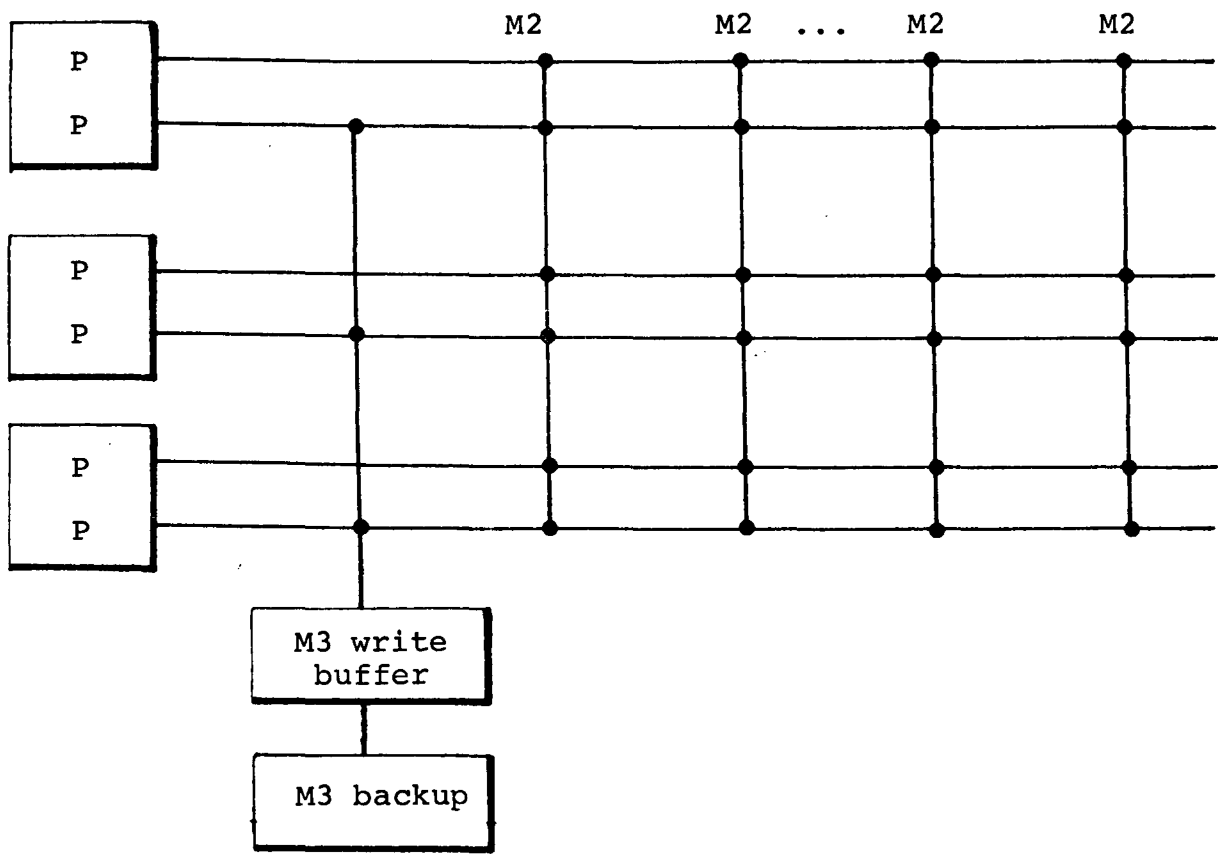
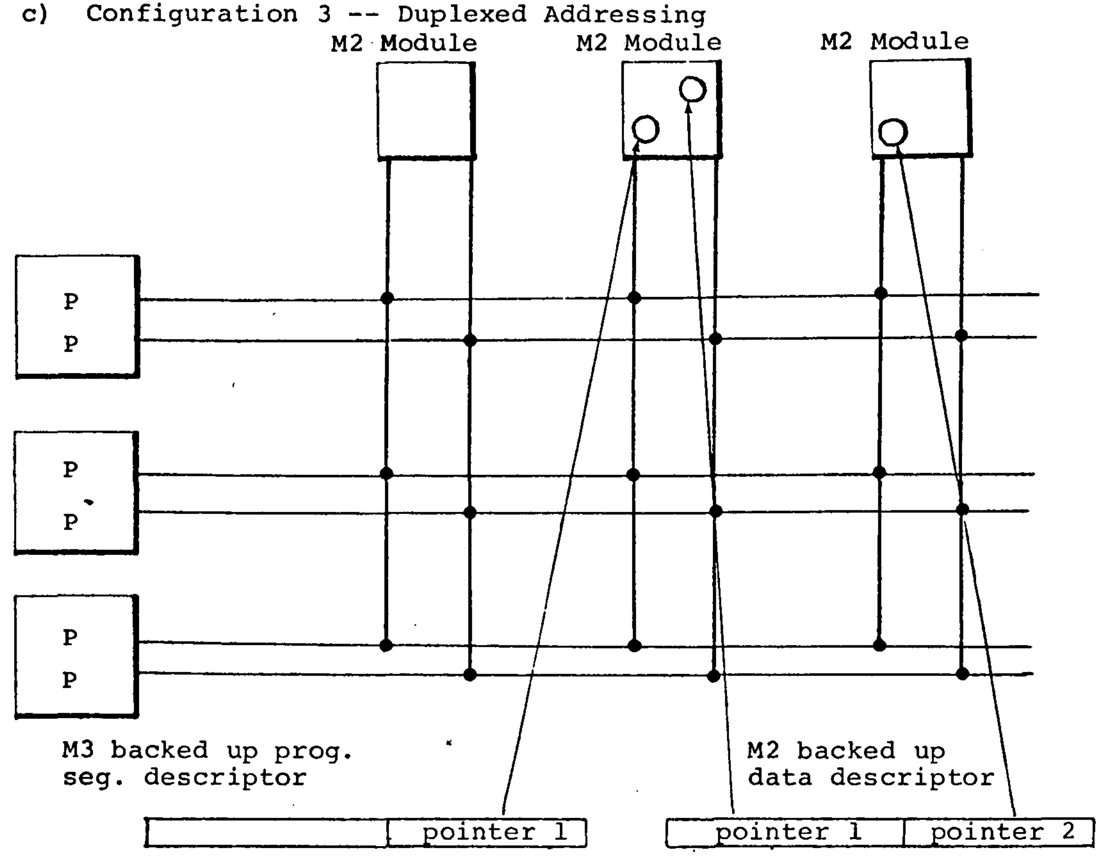


Figure 4.4-7 (continued)



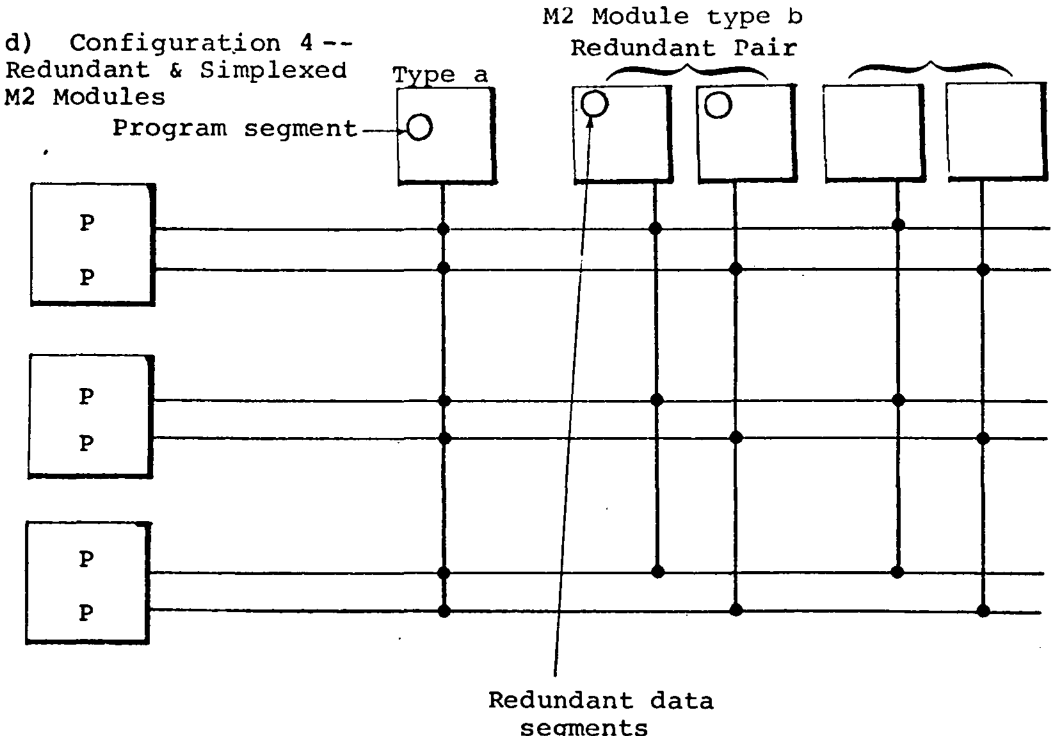
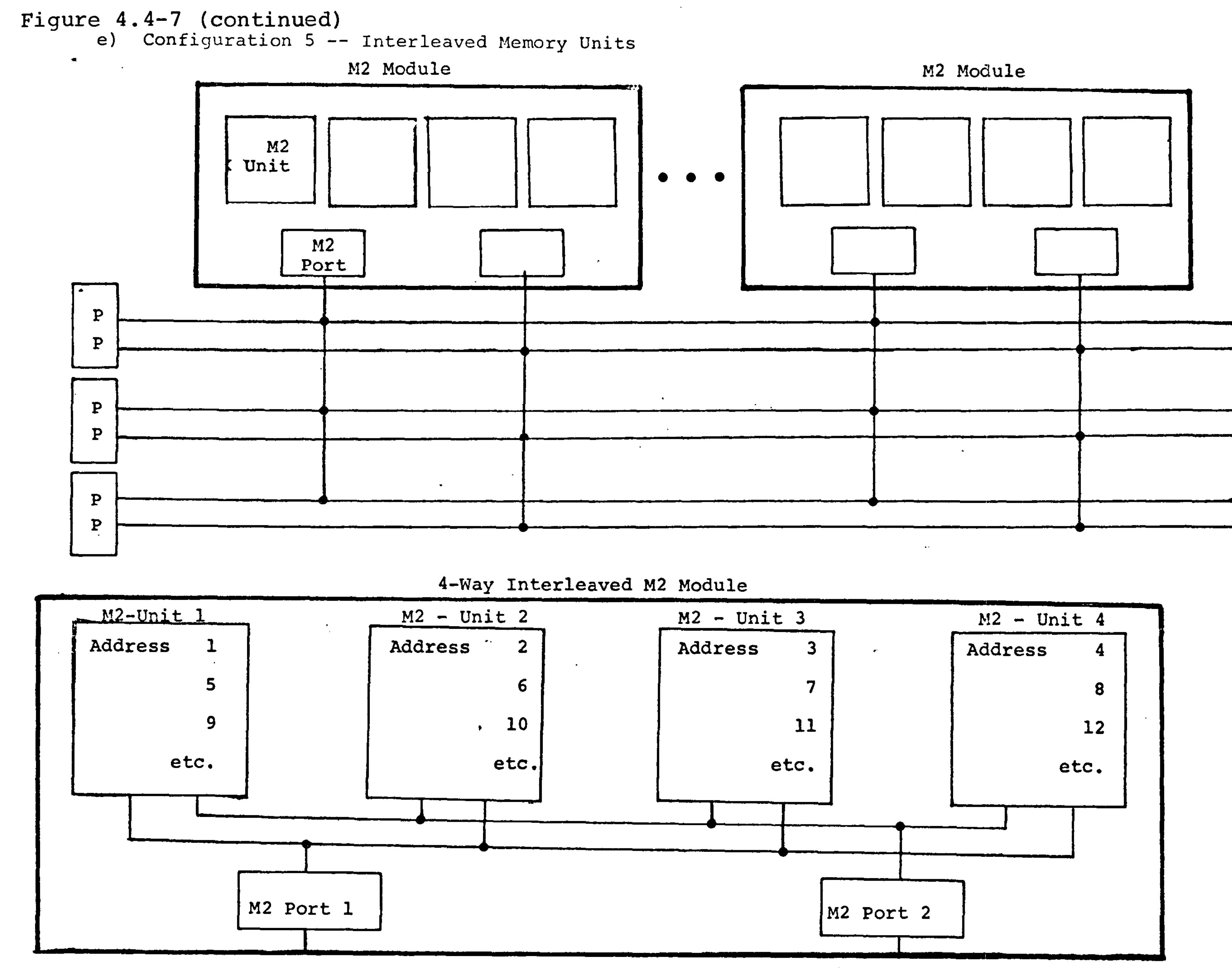


Figure 4.4-7 (continued)
e) Configuration 5 -- Interleaved Memory Units



M2. These holes can be anywhere in different modules. The data segment descriptor will then contain two pointers. One to the data storage location and one to the redundant copy. Programs redundantly stored would also contain two pointers. The majority of program segments, however, will be backed up on M3 and would only require a single pointer (see figure 4.4-7c).

All the addressing modes of the system would, in general, require two addresses. These include descriptor addressing, indirect addressing, stack addressing through base registers, as well as the various stack control address mechanisms.

Redundant writes into independent modules of M2 back up segments is accomplished automatically via the dual redundant processing unit bus links. Recovery of M3 backed up information requires making the segment not present. The memory management routine which handles segment faults will automatically reload the M3 segments when required on demand.

The above description is clearly, very cursory. However, it serves to show that allowing the redundant M2 copy to be located arbitrarily in M2 does entail overhead in dual address generation. Memory management is also made more complex.

4) Configuration 4 -- Redundant M2 Data Modules -- Simplex M2 Program Modules

This scheme is a more constrained version of Configuration 3. The M2 modules will be separated into two operational types. Module type (a) will contain non-critical simplex program segments which are backed up in M3 only. The nature of the segment can be specified in the segment descriptor.

Module type (b) will contain the more critical data segments and some critical program segments (e.g., the absent segment fault handler) which require redundant storage in M2. The back up will occur in an adjacent M2 module. Only M2 backed-up segments can be placed in a type (b) module. (See figure 4.4-7d.) The address of the redundant copy is implicit because it is fixed relative to the prime copy's address. The role of M2 modules can be changed by special privileged instructions. However, this can only occur during system initialization time. Statistics can be kept as to the number of type (a) module segment faults, and the number of type (b) module segment faults.

Recovery is very similar to Configuration 3. However, memory management is not made more complex and the redundant M2 addressing is handled automatically by the hardware. The major penalty of this approach over Configuration 3 is the pre-assignment of memory module types at initialization time, with the possibility of some wasted M2 space.

5) Configuration 5 -- Interleaved M2 Memory Modules

This configuration defines an M2 module as four M2 units which are interleaved on the low order address bits. (See figure 4.4-7e.)

Information segments may either be stored in a simplex a duplex mode. The mode is specified within the descriptor. Most program code would be stored simplexed and interleaved across the four memory units. Most data segments would be stored duplexed. In the duplexed storage mode address i and address i + 1 contain identical information. That is, two adjacent memory units contain identical copies of the redundant words.

The two memory ports connect to the redundant P interfaces. Communication with any M2 unit occurs through either port. This is under control of the command issued from the processing units.

Recovery is similar to configurations 3 and 4. The advantage of this scheme over configuration 4 is that the physical memory modules need not be assigned to either simplex or redundant storage a priori. This is done dynamically, without incurring any overhead. More efficient use is made of the available memory space. Low criticality data need not be stored redundantly. It is merely reconstituted by recomputation or by re-reading from M3 or tape. In case of a failure the data would be made not present and fetched again from M3.

4.4.2.3 Proposed Configuration. The previous section presented five possible approaches to M2 recovery. Figure 4.4-8 presents a summary of various trade-off factors which resulted in the further investigation of Configuration 5 and its final choice. Configuration 5 is actually a compromise between hardware and software complexity, overhead, recovery time, flexibility, and reliability. The following discussion will present various aspects of the chosen design.

Configuration 1 - Complete M2 redundancy
Configuration 2 - M3 backup
Configuration 3 - Duplexed Addressing
Configuration 4 - Redundant and Simplexed M2 Modules
Configuration 5 - Interleaved Memory Units (chosen configuration)

	Configuration Number				
	1	2	3	4	5
Hardware complexity	4	1	2	2	2
Software complexity	1	3	4	2	2
Overhead	1	4	4	2	2
Generality/Flexibility Balance M2/M3 backup	4	4	1	1	1
Reliability	4	1	2	3	2

Rating	1	2	3	4
	best	good	fair	worst

Figure 4.4-8: M2 Configuration Trade-Off

4.4.2.3.1 Philosophy Behind the Configuration Choice: The following factors affect the choice of the M2 recovery philosophy.

1) The difference between read only information (program segments) and read-write information (data segments) is exploited. Read-write information requires a constant update of a redundant copy. Read-only information does not, and therefore can reside on the slower, more economical M3. Read-write information requires a redundant copy in M2 to minimize the overhead involved in updating.

2) If an M2 module which contained program segments failed, it is desirable to exploit the virtual memory mechanism already implemented within the system to aid in the recovery process. All program segments can be considered to reside in M3. They are brought into M2, on demand, for execution. If the program segments contained within the failed module were, as the result of the failure, made "not present", then the M3 to M2 transfer mechanisms will allocate space and transfer the required segments automatically. The "not present" segment indication is contained within the program segment descriptor. Descriptors are considered to be data and are in turn stored redundantly.

3) It is desirable to allow the system to degrade after an M2 failure. Although thrashing may become more probable, this is considered acceptable.

4) The hardware error detection mechanisms are sufficient for isolation, and to indicate the action to be initiated by the software.

5) Consistent with the processing unit failure recovery philosophy, the application programmer is not involved with M2 recovery. Hardware and the operating system control the entire process.

6) Recovery time is in the 10-100 millisecond range, for the high priority real time tasks.

4.4.2.3.2 Features of the Proposed Configuration: The proposed configuration possesses the following properties:

1) There are four interleaved memory modules. Each memory module consists of four 8K by 34 bit memory

units. The four units are interleaved on the lower order address bits. The 34 bits per word include 32 information bits, one word parity bit, and one address parity bit.

2) Each M2 module possess a unique identification. The ID will be fixed and unique for each M2 module. Access to the ID (and module selection via the ID) will be limited to special operating system procedures.

3) Each M2 module reponds when it is addressed by a specific module address. This module address is changeable, since the ID and module address are not the same: The ID is fixed and the module address is variable. The selectability of the M2 module address eliminates the possibility of voids in the address space.

4) There are 2 bus links from each processing unit. Each link interfaces with one of the two M2 module ports. Part of each command issued to M2 from P will be an indication of which port to use for transmitting the accessed data.

4.4.2.3.3 M2 Access Primitives: In order to support M2 fault tolerance a number of basic M2 primitive commands are required.

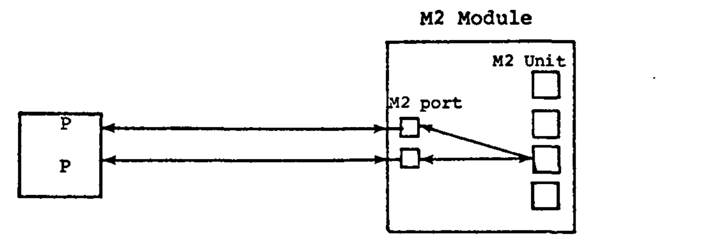
a) Read Single Simplexed -- This command, or sequence of operations, is used to read information from one M2 unit. In this situation the M2 unit will place one word of information on both bus links of the requesting processing unit. The S unit (see figure 4.4-9a) will check data and address parity, provide a time out indication, and receive the various error indications from the M2 module.

b) Read Single Redundant -- This command is used to read M2 information (program or data) stored redundantly in two adjacent M2 units. Both M2 ports operate independently; they accept commands and generate responses on different bus links. The bus link (see figure 4.4-9a) is specified within the command field issued by P.

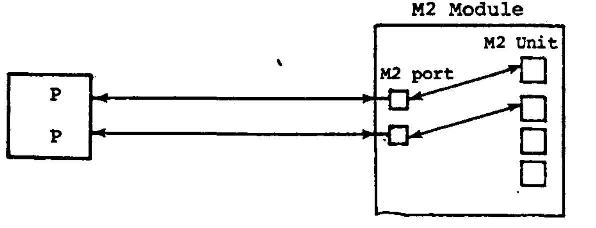
c) Write Redundant and Verify -- When updating information in M2 a simultaneous redundant write operation

Figure 4.4-9: M2 Primitive Operations

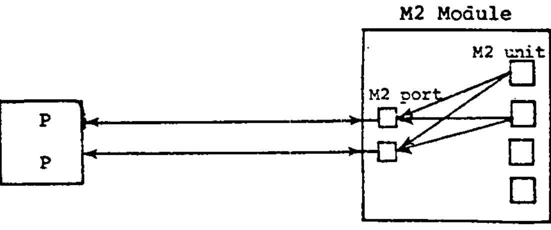
a) Read and Write Single Simplexed



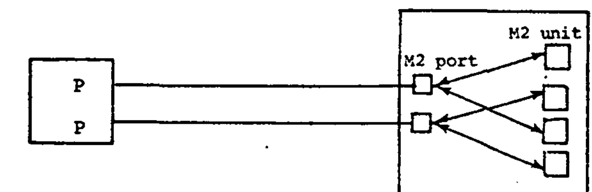
b) Read and Write Single Redundant



c) Read and Write Double Simplexed



d) Read and Write Double Redundant



is used (see figure 4.4-9b). Part of a normal write operation is to echo-check the information back to the S unit for comparison. This is performed independently through the dual M2 ports and dual S interfaces.

The write and verify operation requires two M2 cycles for each word written. However, the system performance degradation is much less than a factor of two since most of the M2 to processor operations are expected to be Read and not Write operations. In a more conventional architecture, such as the IBM 360, if every instruction were a store instruction, only half of the M2 operations would be write, since one needs the instruction read cycle before the memory write cycle. If every instruction pair were Load and Store, then only 25% of the M2 operations would be writes. If one considers the stack-oriented instruction set proposed for the multiprocessor, one concludes that a 10% estimate for the number of write operations would be a realistic worst case. That is 10% of the M2 operations require two cycles, while 90% only require one cycle.

d) Write Single Simplexed and Verify -- In those situations for which recovery is not mandatory, variable information may be written in a simplex mode. The command is issued over both bus links but only one M2 unit is modified (see figure 4.14a). Error detection still requires a verification cycle similar to command c.

e) Read Double Simplexed -- This command allows two 32 bit words to be accessed from the interleaved M2 units as part of the same command. The information is transmitted over the bus links 32 bits at a time. This command is very useful when a double precision number (64 bits) is required from M2. The 64 bit access would only require 100 ns more time than a 32 bit access due to the interleaving of the M2 units (see figure 4.4-9c).

f) Write Double Simplexed and Verify -- This command is the same as command d except that two different words are stored in adjacent M2 units.

g) Read Double Redundant -- This command allows a two word access from redundant storage (see figure 4.4-9d).

Each M2 port accesses its own data from the redundant copies stored in consecutive M2 addresses.

h) Write Double Redundant and Verify -- This command allows the redundant storage and verification, through echo-check, of a 64 bit double word. This command is used when the M2 part of the stack overflows and 64 bits must be pushed into M2.

4.4.3 Recovery from an I/O Unit Failure

Major emphasis of this multiprocessor contract has been placed upon the processing unit and the operational memory, M2. This is due, in part, to the lack of definition of I/O requirements in the contract. The I/O interfaces assumed for the purpose of operating system design were detailed in section 3.5, and the operating system recovery philosophy for I/O was briefly outlined in section 3.7.

4.4.3.1 I/O Recovery Problem Areas. The interaction between software and the I/O system presents problems which are similar to those of communicating internal processes. A number of these are delineated below:

a) I/O Locks

When a software process requires access to an I/O device, the device may require to be locked to the process. That is, no other process can access the selected device until the I/O request is finished. Problems of deadlock exist when the initiating process fails. These are no different from those encountered in the use of "critical sections" of code.

b) Interaction between the Process and I/O

A situation analogous to spawning a parallel process exists when a process initiates an I/O command requiring an "I/O complete" interrupt. A restart could cause a repetition of the I/O request, with a subsequent double access and double interrupt response. The I/O design must anticipate this situation or circumvent it.

c) I/O is like a Write Command

In a sense an I/O operation is like a write command. That is, it changes the state of the system. In case of a

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

failure during the I/O command all the information required to restart the I/O command must be retained. That is, the I/O command as well as the I/O data must be completely double buffered, so that a restart can be initiated.

d) Cancelling I/O

There is, however, one major aspect of I/O which is different from a normal memory write command. While a Memory Segment is being updated it is locked out from other users. A failure during this update may leave the segment in question in an indeterminate state.

The entire segment must be rewritten. The updating information was destroyed. In the case of I/O the reissuing of an I/O command may not be satisfactory. For example, if a process had issued a command to a tape unit to "skip to next file mark" just before a failure occurred, the restart procedure would cause the "skip to next file mark" command to be issued again. This situation is similar to the N = N + 1 problem discussed earlier. In certain cases it is desirable to abort an I/O command in the event of a failure. The I/O controller must be capable of being reset to its previous state. This is, in general, not a trivial task.

4.4.3.2 <u>I/O Configuration organized for Recovery</u>. Two options will be considered to provide a recoverable configuration.

a) Dual I/O Units

If two or more I/O units are required for system operation then the recovery aspects of the I/O can be made very similar to those of a processing unit. Each of the I/O units would be configured like a processing unit with an M2 interface, a special interface to the Processors via dual redundant IPCB's, an M3 interface and a multiplexer interface (probably a data bus) to the outside world. SIR would be employed.

b) Triple Modular Redundancy (TMR)

Since only a single I/O unit is proposed to meet the performance requirements, a triple redundant I/O unit with voting logic is a candidate. Transients are completely masked in this configuration. If a permanent failure occurs then the voting elements can be reconfigured to comparators and the bad I/O unit taken off line for repair. Two options are available as far as recovery from a failure in the I/O unit.

The details of the specific I/O configuration depends upon the nature of the I/O devices and the communication and synchronization between the devices and the I/O controller.

Figure 4.4-10 shows the proposed redundant I/O configuration. The basic structure incorporates a TMR central I/O control, and a level of peripheral interface redundancy consistent with the device redundancy. The characteristics of the configuration are presented below.

a) The triple redundant I/O hard core contains the central control, timers and the interrupt control. A failure in this critical area will allow the system to keep running without propagating the error.

b) In order to interface the TMR section with other dual redundant interfaces voters and switches are provided. The S elements, which are controlled by their associated I/O elements, are used to select which of the dual redundant interfaces to receive. The V elements vote upon the triple redundant I/O outputs and produce dual redundant outputs. The voters will automatically reconfigure to comparators and switch out a faulty I/O where required.

c) The IPCB, M2, M3, and data bus are all postulated to be dual redundant. For this reason their interfaces are shown to be dual and they interface to the I/O via the S's and V's. The multiplexer channel which contains peripherals necessary to operate the lab model is only shown as a simplex subsystem, with a corresponding single interface.

d) It is assumed that all the peripheral devices attached to the data buses and the M3 cotnroller possess characteristics which will aid in the recovery process. These characteristics include:

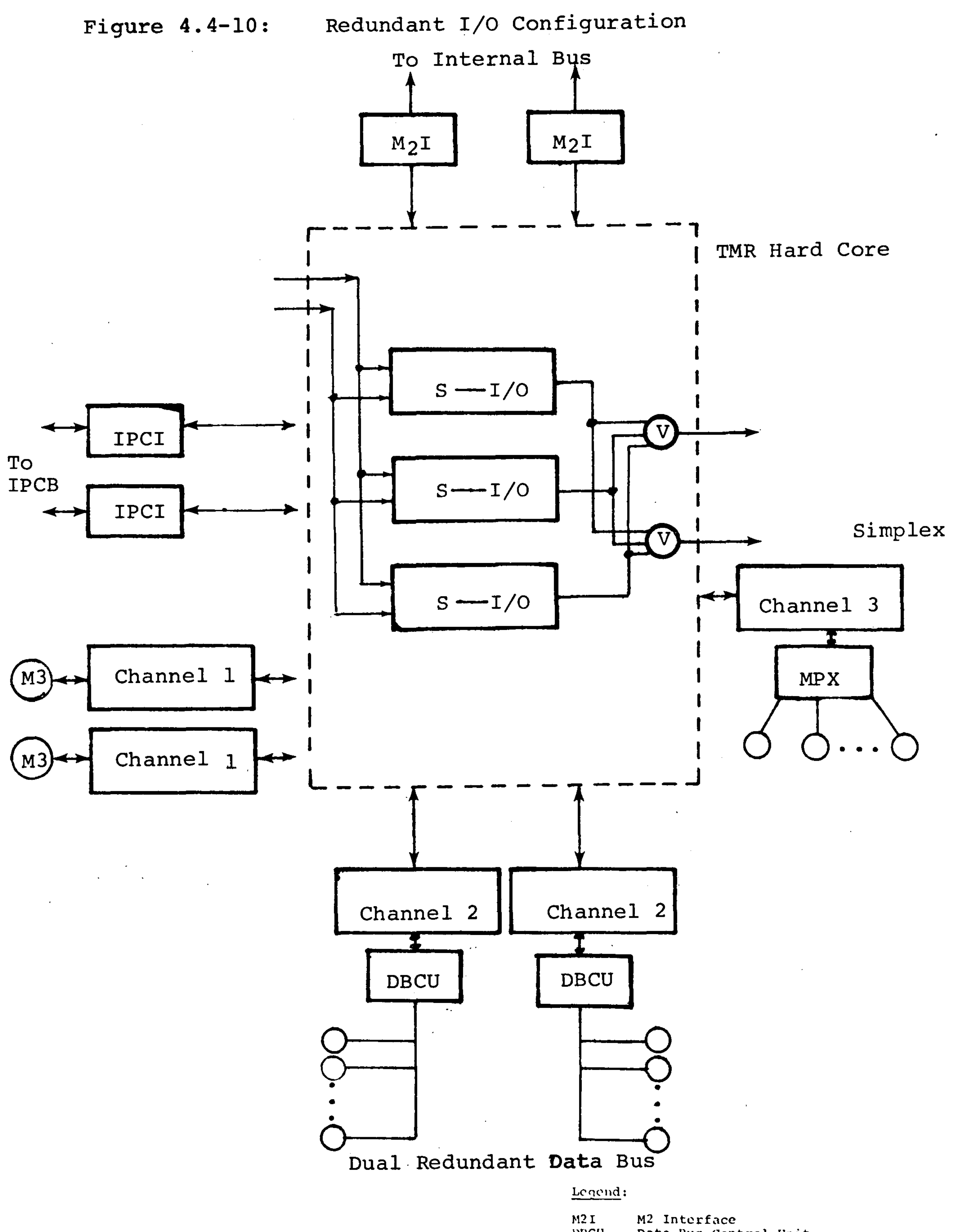
   1) hardware to aid in fault isolation between dual redundant threads

   2) sufficient buffering so that aborted commands cannot hang up a subsystem

   3) the ability to be reset and to indicate upon request the status of the I/O device

e) Certain problems caused by locking of processes to I/O devices must be resolved by the operating system. This requires the capability to selectively delete the IOCB created by a process which is cancelled (either purposely

Figure 4.4-10: Redundant I/O Configuration



Legend:

M2I — M2 Interface
DBCU — Data Bus Control Unit
IPCB — InterProcessor Communication Bus
IPCI — InterProcessor Communication Interface
I/O — I/O contains central control, interrupt control, timers
V — Voter
S — Switch

or as the result of a failure) from the appropriate DEVICEQ, and to relieve the M2 space allocated as the I/O buffer area (refer to section 3.5.2).

One of the main motivations for a triple redundant I/O central core is to reduce this problem as far as I/O failures are concerned. A failure within the central TMR IOC cannot propagate past the voters. However, a voter or channel failure can cause a temporary suspension of I/O or a re-issuing of an I/O command and the associated problem of releasing any I/O locks.

4.4.4 Recovery from an M3 Failure

In a sense the discussion concerning M2 recovery is applicable to M3 recovery. M3-M4 interaction is analagous to M2-M3 interaction. A few basic parameters are different, however.

M3 is required to provide in the order of 10^6 words of storage. This is small enough to be placed upon a single drum like device. M2 consists of many identical modules while M3 consists of only one.

M3 is probably 100 to 1000 times cheaper per bit than M2. For this reason a completely dual redundant M3 configuration with independent error detection is a very practical method of providing backup in case of an M3 failure.

4.4.4.1 Average M3 Access Time. If we assume that M3 is a drum like device, then a question arises as to what is the average access time when two drums are employed in a completely redundant configuration.

All write operations will take place onto both drums in the identical track and sector. Reads will occur from that drum which provides the quickest access. If it is assumed that both drums are not synchronized and that the data transfer time can be ignored compared to the drum latency time, then a number of interesting facts can be deduced:

a) The average read access time

$$\overline{TR} = \frac{P}{3}$$

where P is the period of rotation of the drum.

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

b) The average write access time

$$\overline{TW} = \frac{2P}{3}$$

c) On the assumption that there will be at least twice as many read operations as write operations, the average access time will be

$$\overline{TA} = \frac{2\ \overline{TR} + \overline{TW}}{3} = \frac{4P}{9}$$

which is slightly less than the average access time of a single drum, which is P/2. Therefore no degradation in performance occurs by using dual redundant drums.

References for Chapter 4

1) Space Division, North American Rockwell, "Modular Space Station Phase B Extension", Preliminary System Design, Volume IV Subsystems Analysis, January 1972, NASA Contract NAS9-9953, MSC 02471, DRL # MSC T575, Line Item 68.

2) Intermetrics, Inc., Task Report SA-101, "Central Processor Operational Analysis", WBS 94010-2, September 1971, prepared for NAR/SD.

3) Intermetrics, Inc., Task Report SA-102, "Central Processor Memory Organization and Internal Bus Design", WBS 94010-4, December 1971, prepared for NAR/SD.

4) Ball, M., and F. Hardie, "Effects and Detection of Intermittent Failures in Digital Systems", FJCC 1969, pp. 329-335.

5) Lin, Shu, "An Introduction to Error-Correcting Codes", Prentice-Hall, Inc., New Jersey, 1970.

6) Russo, R.L., "Synthesis of Error Tolerant Counters Using Minimum Distance Three State Assignments", IEEE Trans. E.C., Vol, EC14, June 1965.

7) Beister, J.C., "On the Implementation of Failure Tolerant Counters", IEEE Trans, E.C., September 1968.

8) J. G. Tryon, "Quadded Logic", in Redundancy Techniques for Computing Systems, R.H. Wilcox and W.C. Mann (eds.), Spartan Books, Washington, D.C. 1962.

9) Jensen, P.A., "Quadded Nor Logic", IEEE Trans. on Reliability, September 1963.

10) Pierce, W.H., "Interwoven Redundant Logic", Journal of the Franklin Institute, Vol. 277, No. 1, January 1964.

11) von Neumann, J., "Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components", In Annals of Mathematic Studies, Princeton, N.J., No. 34, pp. 43-98, 1956.

12) Lyons and Vanderkulk, "The Use of Triple Modular Redundancy to Improve Computer Reliability", IBM Journal of Research and Development, 6, No. 2, pp. 200-209, April 1962.

13) Dickson, Jackson, Randa, "Saturn V Launch Vehicle Digital Computer and Data Adapter", AFIPS Conference Proceedings, Vol. 26, FJCC, pp. 501-516, 1964.

14) Mann, W.C., "Restorative Processes for Redundant Computing Systems", In Redundancy Techniques for Computing Systems, R.H. Wilcox, and W.C. Mann (eds.), Spartan Books, Washington, D.C. , 1962.

15) Garner, H.L., "Generalized Parity Checking", IRE Transactions on Elect. Comp., September 1958.

16) Brown, D.T., "Error Detecting and Correcting Binary Codes for Arithmetic Operations", IRE Transactions on Elect. Comp., September 1960.

17) Garner, H.L., "Error Codes for Arithmetic Operations", IEEE Transactions on Elect. Comp., Vol. EC15, No. 5, October 1966.

18) Mandelbaum, D., "Arithmetic Codes with Large Distance", IEEE Transactions on Information Theory, Vol. IT13, No. 2, April 1967.

19) Peterson, W.W., Error Correcting Codes, New York, Wiley, 1961.

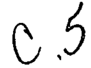
20) Rao, T.R.N., "Error Checking Logic for Arithmetic Type Operations of a Processor", Proc. First Annual Princeton Conference on Information Sciences and Systems, 1961.

21) Szygenda, S.A., Flynn, M.J., "Coding Techniques for Failure Recovery in a Distributive Modular Memory Organization", SJCC, 1971, pp. 459-466.

22) MIT, "STS Software Development", 7 July 1970, NAS9-4065.

23) Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control", Communications of the ACM, Vol. 8, No. 9, September 1965, p. 569.

24) Haberman, A.N., "Synchronization of Communicating Process", Communications of the ACM, Vol. 15, No. 3, March 1972, pp. 171-176.



25) Copps, E.M., Jr., "Recovery From Transient Failures of the Apollo Guidance Computer", AIAA Guidance, Control and Flight Dynamics Conference, Paper #68-932, August 1968.

26) Crisp, R., Intermetrics, Inc., (private communication).

5.0

IMPLEMENTATIONAL APSECTS

This chapter presents the hardware aspects of the proposed multiprocessor. The first section presents a discussion of the implementation of the various multiprocessor elements. The purpose of this discussion is to demonstrate the feasibility of implementing the functional design presented in the foregoing chapters. It is not intended to be a complete or detailed design. The second section presents an analysis of performance as well as a proposal for the laboratory model.

5.1 Design of Multiprocessor Components

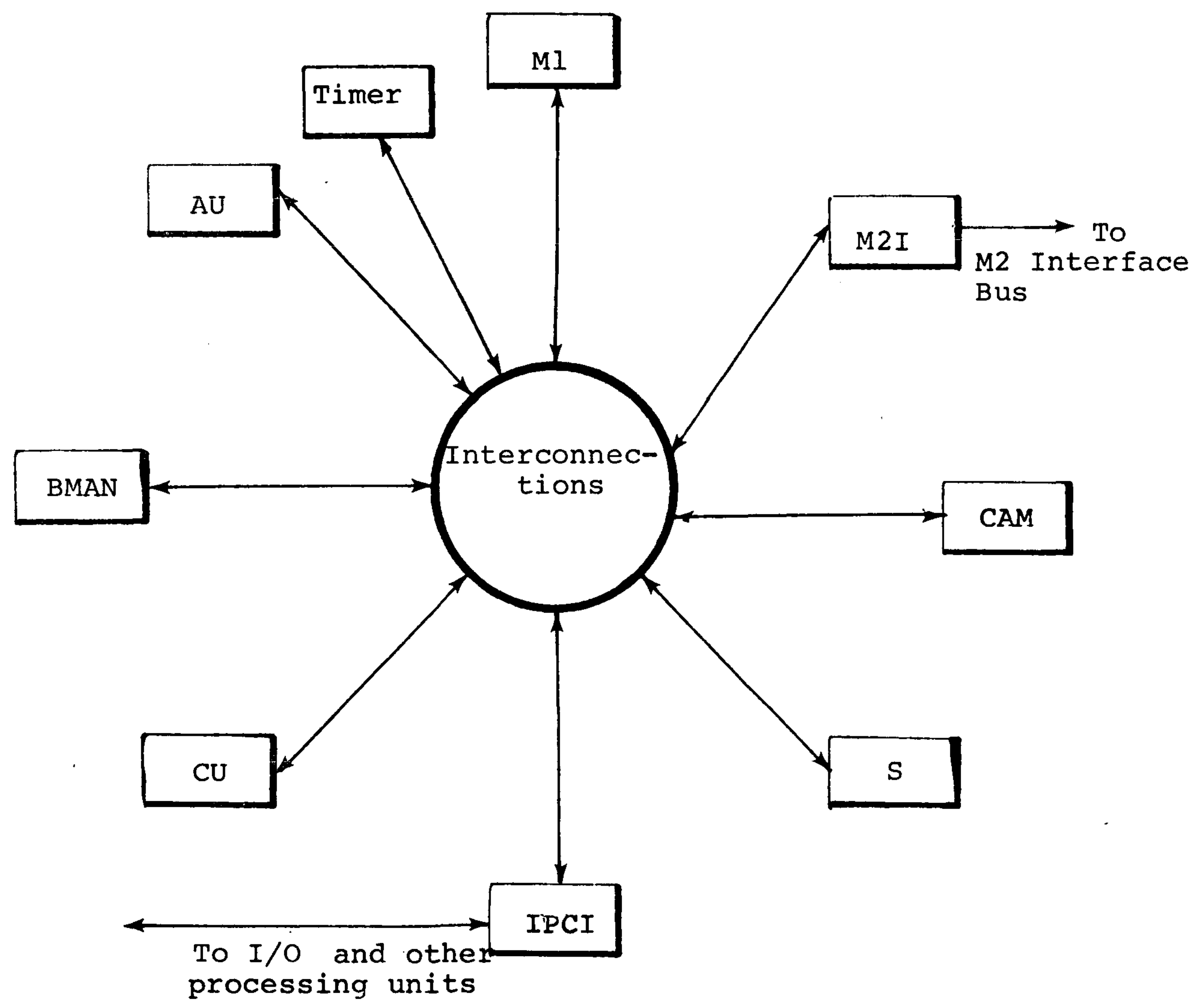
The implementation of the major elements of the multiprocessor are presented. These include the Processing Unit, Internal Bus, InterProcessor Communications Bus, I/O unit, Operating Memory, and the Mass Memory.

5.1.1 Processing Unit Implementation

Based upon the instruction set and philosophy described previously, this section will present certain details of an implementation of the processing unit.

5.1.1.1 Functional Elements of the Processing Unit: Figure 5.1-1 indicates the nine basic elements of the processing unit. A specific interconnection between these elements is intentionally left vague to indicate that the system designer may implement various degrees of parallelism as performance requirements dictate. As far as this discussion is concerned, it is only necessary to assume that all the elements can communicate with each other. The processor, P, is considered to contain all the interconnected elements of Figure 5.1-1 excluding the local memory, M1.

Figures 5.1-2 through 5.1-9 depict each element of Figure 5.1-1 in more detail. Each of these hardware elements is described in the following paragraphs.



AU - Arithmetic Unit
BMAN - Bit Manipulator
CU - Control Unit
IPCI - InterProcessor Communication Interface
S - Status
M2I - Operating Memory (M2) Interface
M1 - Local Memory
CAM - Content Addressable Memory
Timer- Instruction and Process Timers

Figure 5.1-1: Processing Unit Block Diagram

Only the logic involved in instruction execution is described. The error control logic, EC, the switch elements, S, etc., discussed in the fault tolerance section are not emphasized here, since they would tend to obscure the discussion of instruction implementation.

5.1.1.1.1 Arithmetic Unit (AU). The AU provides the facility for both executing floating point arithmetic and the limited length integer arithmetic required for detailed execution of an instruction. In this second category are such microsteps as add one to the program counter.

The double precision floating point number possesses a mantissa of 51 bits plus sign and an exponent of 10 bits plus sign. The adder, therefore, is split into a 51 bit fractional section and 10 bit exponent section.

Two input and one output buffer registers are provided. These registers may be used or bypassed depending upon the details of the microprogram used in the execution of a particular MP instruction. Only two commands can be executed by the AU. These are ADD and SUBTRACT. Logical operations are executed in the BMAN unit.

Quite often tests are required as to whether the result of an arithmetic operation is positive, negative, zero, overflow or underflow of mantissa or exponent. These conditions detected in the AU are sent to the status register for storage and subsequent testing.

The entire 63 bit add operation including the loading of SUMR can be accomplished in 100 ns.

5.1.1.1.2 Bit Manipulation Unit (BMAN). The BMAN consists of two sequentially executed functions of shifting and masking. The shift control register (SCR) controls the number of shift positions as well as the type of shift. A shift from 1 to 64 positions can be accomplished in a single processor cycle of 100 ns. The format and types of shifts are indicated below.

Shift Control Register,

3 bits	6 bits
type of shift	# of shifts

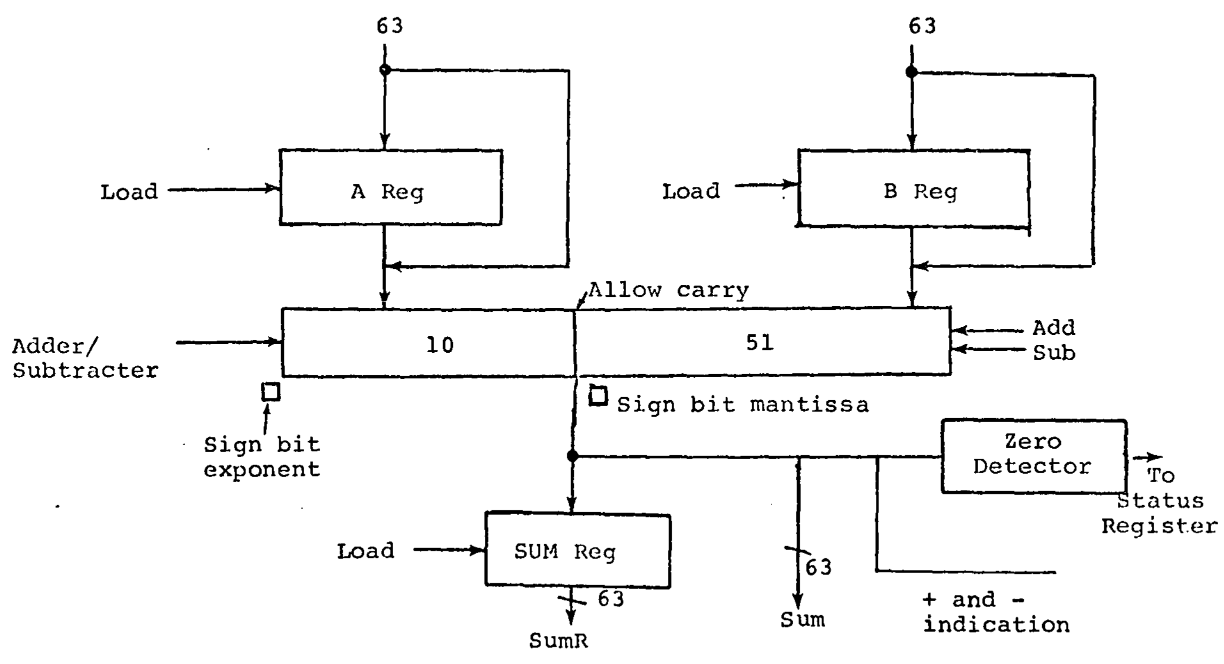


Figure 5.1-2: Arithmetic Unit (AU)

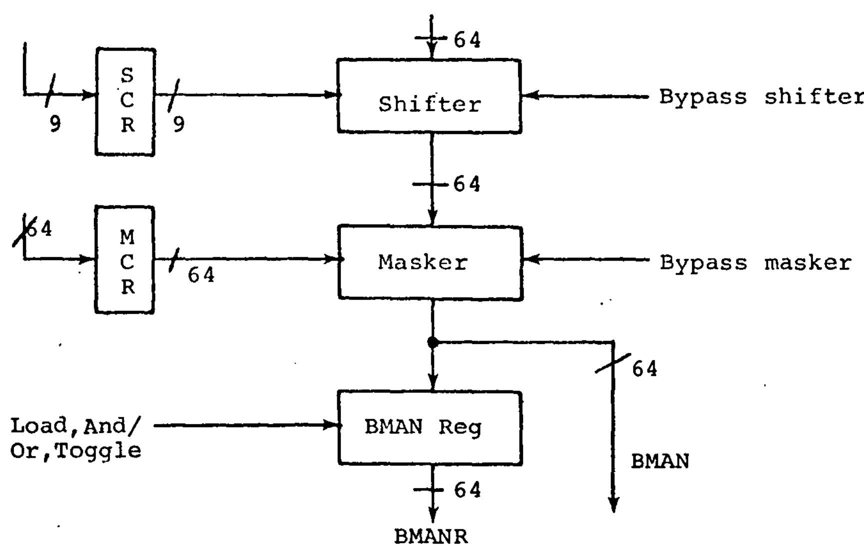


Figure 5.1-3: Bit Manipulation (BMAN)

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

Type of Shift

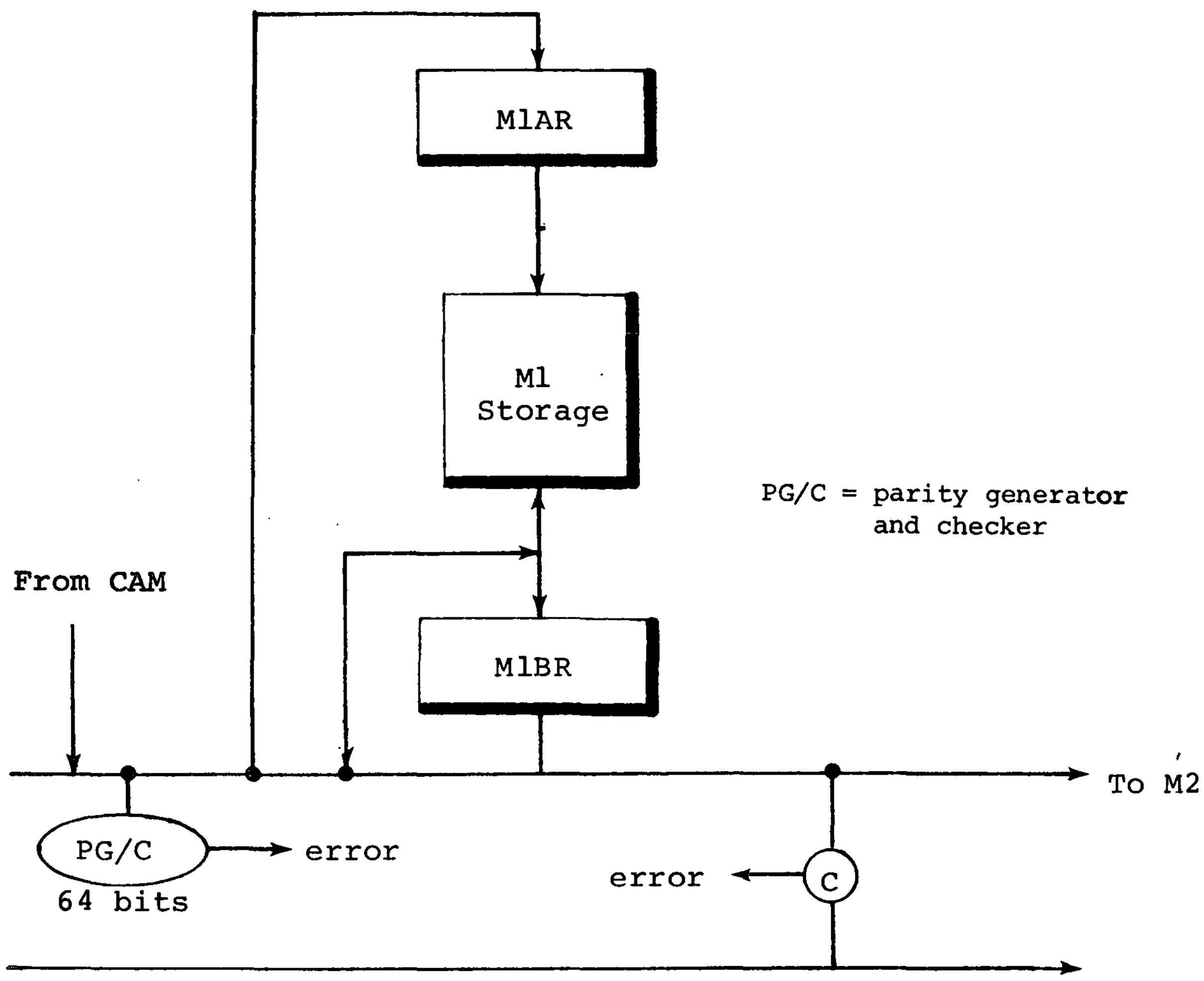
000	shift left	and fill in MSB with ϕ
001	shift right	and fill in LSB with ϕ
010	shift left	and fill in MSB with 1
011	shift right	and fill in LSB with 1
100	shift left	cycle
101	shift right	cycle

Often the SCR contents must be ignored since no shift is desired. For this reason a BYPASS SHIFTER control signal is provided.

The masker which basically consists of an AND gate per bit is controlled by a 64 bit mask control register (MCR). The output of the shifter and the MCR are ANDed to form the BMAN output. A BYPASS control is also provided. In addition a BMAN output buffer register is provided (BMANR). The register may be loaded, ANDed into, Inclusive ORed into, or Exclusive ORed into. The combination of shifter, masker and BMANR provides a powerful tool for not only executing the bit manipulation instructions, but also for extracting and inserting fields during detailed micro sequences.

The entire operation of shift, mask, and load BMANR can be accomplished in 100 ns.

5.1.1.1.3 Local Memory (M1). M1 consists of storage elements, an address register (M1AR) and a buffer register (M1BR) The contents of M1 are enumerated in Figure 5.1-4. A description of M1 cannot be undertaken without indicating its relationship to the fault tolerance aspect of the multiprocessor. A common bi-directional communications link is used to interface the Processor, P, with M1 and M2. Parity generation and checking logic (PG/C) is attached to this bus. Also the comparator, C, interconnecting the two halves of the processing unit is shown. At most 96 words and parity is required. This is 6144 plus parity bits. An M1 access including loading M1AR, reading M1, loading M1BR and checking parity can be accomplished in 100 ns.



Contents	# of 64 bit words
M1 portion of stack	8
16 - Base Registers (2/word)	8
Status words	2
M1 temporary write area	32
Descriptor cache accessed thru CAM	32
Instruction and Process counter images	1
Other temporary storage	13
Total	96

Figure 5.1-4: Local Memory M1

5.1.1.1.4 Content Addressable Memory (CAM). The CAM is used to provide a fast translation between the Stack Number and Offset (SNO) representation of an address and the actual physical M2 address. Figure 5.1-5 presents a block diagram and the format of the word contained within the CAM.

The CAM can operate in two distinct modes. In mode 1, data can be read or written through normal addressing. The CAM functions as a normal RAM with the address being placed in CAMAR and data being input or output to the CAMBR. The second mode of operation is an association of equality. A simultaneous comparison of all 32 internally stored CAM words is performed with the comparison data under control of the mask. The mask allows an equivalence association upon only the fields indicated by the mask. The result of an association is an indication of one or more M1 addresses that correspond to the correct association.

State of the art MSI technology produces associative memory circuits which can be configured into a CAM which possesses a cycle time of less than 100 ns.

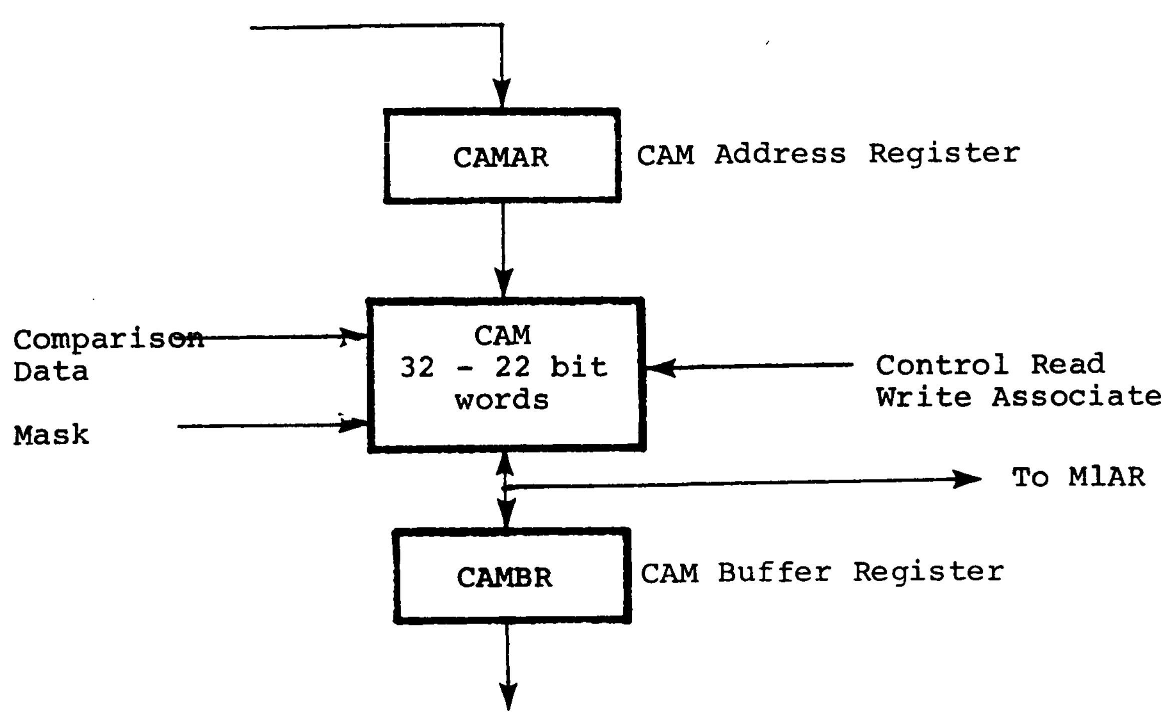
5.1.1.1.5 Control Unit (CU). The control mechanism for the processing unit employs the concept of microprogramming. M0 storage contains the control words which are used to activate the various control signals. Control signals are generated from three sources:

a) Directly from the M0 buffer register (M0BR1)

b) From a combination of status conditions and conditional control elements in M0BR1. The control signal decision matrix is stored in the conditional control memory (CCM).

c) Finally the micro control field can point to larger literal control fields or words (such as masks for BMAN) which are fetched from the literal ROM, LR.

Microsequencing is initiated by an instruction code which is placed into instruction register byte 0, IR0. This instruction code is decoded into an M0 address in the Address Decoder ROM (ADR). The resulting address is stored in M0AR and M0 is read. The two M0 buffer registers M0BR2 and M0BR1 as well as the temporary M0 address register, TM0AR, are used to provide microinstruction look-ahead.



CAM WORD FORMAT

20 bits	1 bit	1 bit
Stack Number and Offset (SNO)	Occupied Bit (OB)	Not Eligible bit (NEB)

Figure 5.1-5: Content Addressable Memory (CAM)

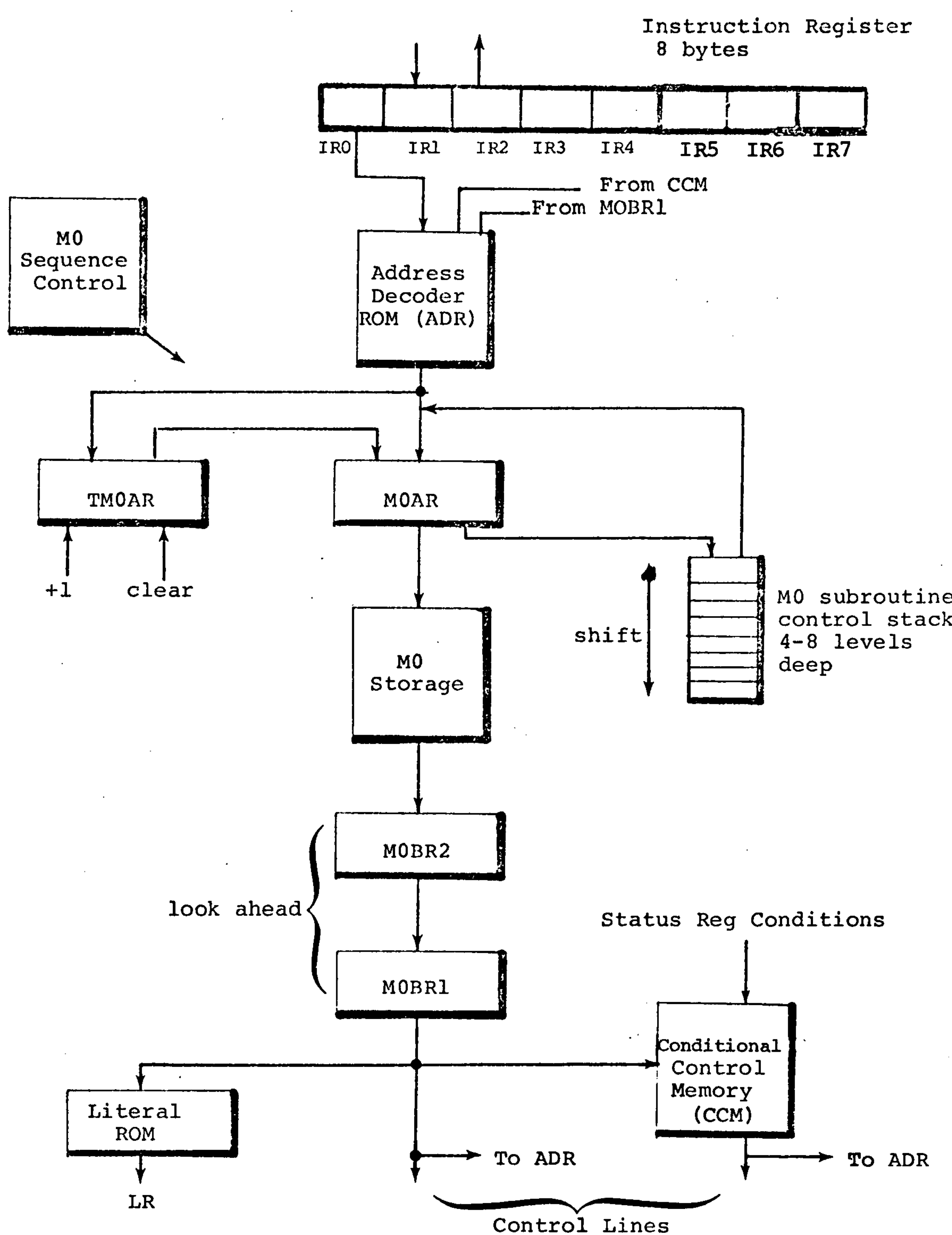


Figure 5.1-6: Control Unit

As the instruction set is implemented one finds that the concept of a micro code subroutine is quite useful in minimizing the size of M0. For this reason an M0 subroutine control stack is used to provide temporary storage of return information from M0 subroutines. At the beginning and end of every instruction the M0 control stack is empty. It is used only during the execution of the various micro-subroutines. The use of micro-subroutines will become clearer in the discussion of instruction sequencing.

Besides employing subroutines, unconditional and conditional branch addresses can be obtained from M0BR1 and the conditional control memory. These address indications, in general, are steered through the ADR.

5.1.1.1.6 Status. Two status registers are employed in the processing unit. Sixty four bits were not sufficient to represent all the states required. The contents of the two status registers SR1 and SR2 is determined on the basis of the probability of a status condition changing. SR1 contains the most frequently changed conditions and SR2 contains the more static conditions. Both SR1 and SR2 have images in M1 so that in case of failure the processor status can be restored.

Figure 5.1-7 presents a listing of the various status conditions which have been isolated. The entire state of a Processor at the beginning and end of an instruction is contained in SR1 and SR2. By possessing an image in M1, it is possible to re-establish the processor state for recovery purposes.

5.1.1.1.7 InterProcessor Communications Interface (IPCI). The InterProcessor Communications Bus (IPCB) provides a number of features which allow the processing units and I/O unit to communicate. The specific commands that are issued over the IPCB is discussed in section 5.1.4.

The IPCB is conceived to be time multiplexed with an 8 bit wide data path. Timing control is exercized by the I/O unit.

The IPCI in each processing unit contains a 4 byte shift register buffer so that it can possess a degree of asynchronous operation with the rest of the processing unit (see Figure 5.1-8). A processing unit or I/O unit issues a communication command by placing an

Status Register 1 (SR1)

Function	# of bits
Program Counter - PC	18
Syllable Counter - SC	2
Instruction Phase - ϕ	1
Lexical Level - $\ell\ell$	4
M1 Top of Stack - M1TOS	3
M1 Stack Limit - M1SL	3
M2 Top of Stack - M2TOS	18
M1 Stack Empty	1
Allow Interrupt	1
Interrupt Indication	1
Condition Codes (+,-,zero, etc)	8
InterProcessor Communication Pending	1
Others	3
TOTAL	64

Status Register 2 (SR2)

Function	# of bits
M2 Stack Limit - M2SL	18
M2 Bottom of Stack - M2BOS	18
Interrupt and Trap Information (partially encoded)	28
TOTAL	64

Figure 5.1-7: Status

Figure 5.1-8: InterProcessor Communications

Interprocessor Communication BUS (IPCB)

I/O

P-M_1

P-M_1

P-M_1

IPCB

Address Decode

shift

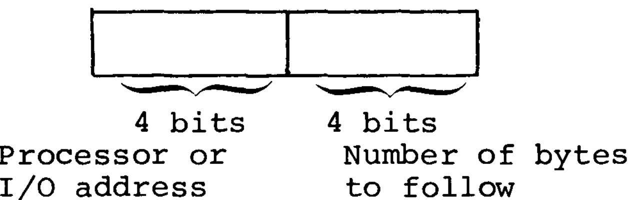
4 Byte buffer

Command Decode

To M_1
CAM
Status (Interrupt indication)
IPC pending
etc.

Control Unit

addressing byte upon the IPCB at a given time slot. The addressing byte processes the following format:

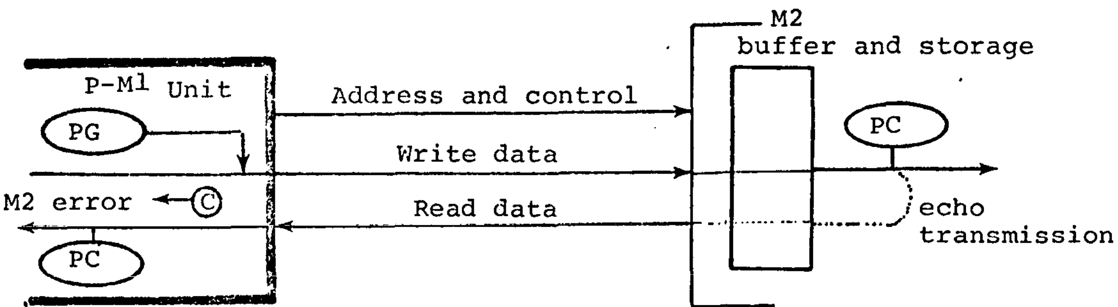


This addressing word is generated by firmware and can select one of the processors or I/O. One particular address indicates that all the processors are to be interrupted. Up to 16 bytes of information may follow the initial addressing word. These bytes are the top of stack word or the first two top of stack words, as needed.

5.1.1.1.8 M2 Interface - M2I. M2I contains no storage or buffer registers. Any that are required are assumed to be in M2 and dedicated to a P-M1 processing unit.

On read operations P generates an address, control word, and waits for an M2 response. It then checks for correct parity in PC to determine the possibility of an M2 error.

For write commands, which are executed only during phase 2 copy cycles ($\phi 2$) P will generate address and control information for M2, transmit it and immediately transmit the data to be written. It will then wait for an M2 response which provides a complete echo check on the data from the memory cells. M2I will then compare the echoed word with the word to be written. Any discrepancy indicates a transmission or M2 error. All this action is under CU control.



C = compares echo check of M2 write operation
M2 = operating memory
PG = parity generator
PC = parity checker

Figure 5.1-9: M2 Interface

An M2 read operation including internal bus delay can be accomplished in 1 microsecond. This time includes a 100 nsec addressing time, 800 nsec M2 cycle time and another 100 ns to transmit the read data back to P. A write will require 1.9 μsec. This includes 100 ns for address and command transmission, a data transmission of 100 ns, an M2 write of 800 ns, an M2 read of 800 ns, and a subsequent transmission back to P for echo-check of 100 ns.

The implications of the echo check on writes is discussed in the fault tolerance section.

5.1.1.1.9 Timers. Each processor contains two timers; an instruction timer and a process timer.

a) Instruction Timer: The instruction timer is used to insure that no MP instruction is able to enter into an infinite loop, and thus cause the lock up of a processing unit. Depending upon the micro step implementation, this might be possible for certain array type instructions. An indefinite degree of indirect addressing is another possible example. In either case an error exists with the

possibility of both halves of the processing unit entering into a loop of indefinite execution time.

The instruction timer will cause a trap if a MP instruction executes for an excessive time. Whenever a new MP instruction is executed the maximum MP instruction time limit is reloaded into the instruction counter and decremented automatically at a 1 μsec per pulse rate. If the timer reaches zero an instruction timer overrun trap will occur.

The contents of the instruction timer need not be saved in the case of failure. If a failure occurs, the instruction timer's contents will be re-established when the instruction retry is initiated.

A 24 bit timer would allow an instruction to take up to 16 seconds to execute. This is more than adequate.

b) <u>Process Timer</u>: The process timer is used to determine the amount of time a process has been in the running state. It is loaded with a value representing the maximum amount of processor execution time to be allowed. If a process is placed into a waiting state due to an I/O request, or an interrupt, its process counter is stored in the process stack so that it can be used when the process resumes on a processor.

If the process counter decrements to zero, then a process counter overrun trap is generated.

A process counter of 32 bits will allow a process to execute for 4000 seconds with a 1 μsec decrementation period.

Both the 24 bit instruction counter and the 32 bit process counter are separate counters external to M1. Periodically, (e.g., every M2 cycle), the counters are gated by each half of the processing unit to the comparators so that failure detection can occur. If the counters are consistent then they are written into a fixed location in M1 so that in case of a failure this old value (not more than a few microseconds old) can be used for recovery. By updating M1 with

counter information during an M2 cycle, the processing unit is not degraded due to extra M1 cycles, since the processor would be idle during an M2 read or write operation.

5.1.1.2 Instruction Execution: The functional elements described in section 5.1.1.1 are quite conventional. It is the manipulation of these elements by means of the micro sequence which enable the instruction set to be implemented. The discussion to follow indicates the basic sequences which must be followed to execute various micro subroutines and instructions. In practice a number of these may be executed in parallel in a single micro instruction. It would not be instructive at this time to define the micro word format, or indicate the parallelism involved. The main purpose is to indicate the sequencing involved and thus demonstrate the practicality of implementing the instruction set.

5.1.1.2.1 Generic Instruction Execution. Figure 5.1-10 depicts the sequential flow of a generalized instruction. Included in this description are indications of various micro subroutines, the two phases of a restartable instruction, as well as interrupt and trap interfaces to the normal instruction flow.

The first action is to test the interprocessor communication indicator (IPC bit). If it is set, then the indicated command will be executed. After this execution control is returned and a test is made of the interrupt indicator bit. If an interrupt is present a micro-subroutine is called to enter the interrupt routine. If no interrupt is present the next instruction, indicated by the program counter and syllable counter in SR1, is fetched. The instruction is executed using an M1 temporary storage area for M1 and M2 write operations. During the execution of an instruction various trap indications may be generated. Type 1 traps cause an entrance to a software routine. Type 2 traps, set an indicator bit and continue with the instruction execution.

An example of a type 1 trap is a segment fault, or an inconsistant or illegal descriptor. Processing can not continue. After executing the given trap routine, the original instruction will be re-executed.

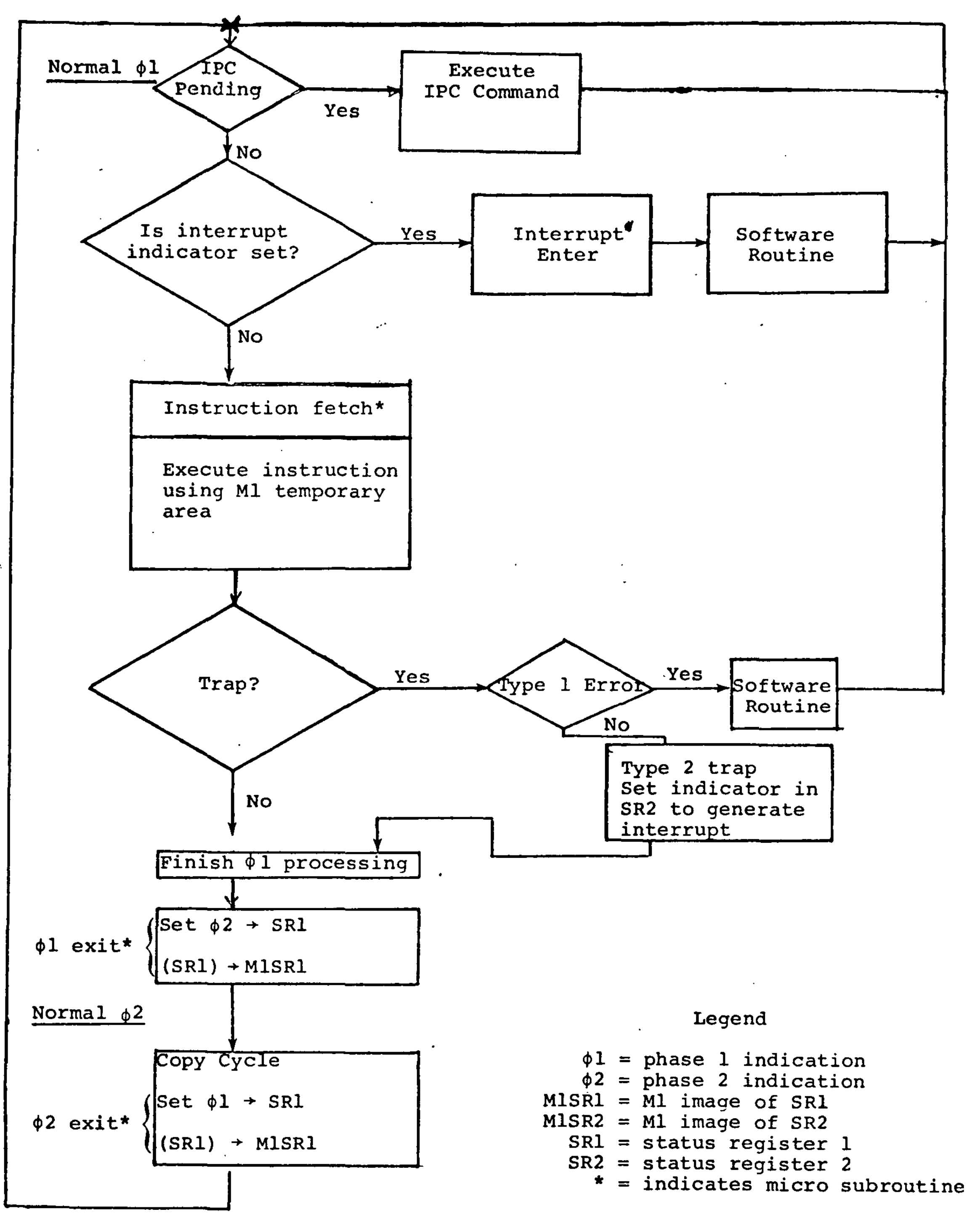


Figure 5.1-10: Generic Instruction Execution

An example of a type 2 trap is overflow or divide by zero. In these cases an indication in SR2 is set and the instruction processing continues. At the beginning of the subsequent normal $\phi 1$ cycle these conditions will be tested and an interrupt linkage established if the trap was allowed by the trap mask in SR2.

The last step of $\phi 1$ processing is called the $\phi 1$ exit sequence. $\phi 2$ indication is set into SR1 and SR1 is written into its image area in M1. The normal $\phi 2$ involves a copy cycle whereby M1 and M2 locations are updated from the M1 temporary storage area. $\phi 2$ exit involves setting $\phi 1$ into SR1 and writing SR1 into M1SR1.

If SR2 was changed during normal $\phi 1$ an M1 write of SR2 into M1SR2 would be executed through the M1 temporary storage area. Only the update of SR1 into M1SR1 need not utilize the M1 temporary buffer area.

5.1.1.2.2 Some Micro Subroutines and Sequences. To gain more insight into the instruction execution a number of illustrative micro subroutines and sequences are presented.

Many of the flow charts, to be presented, possess execution time indications. Little effort has been expended in trying to exploit any parallelism contained within the logic to increase speed. The purpose of the time indications is to give the reader a "ball-park" estimate of what one might expect in implementing the proposed instruction set.

Similarly the flow charts are meant only to be illustrative and not necessarily complete. A design worked out in fine detail would not serve our purpose of demonstrating the implementability of the proposed functional design.

5.1.1.2.2.1 Interrupt Enter: This routine is entered when the $\phi 1$ indication is set. (See Figure 5.1-11) Like any other instruction the Interrupt Enter subroutine is composed of a $\phi 1$ and $\phi 2$. During $\phi 1$ the stack is marked by the placement of a special MSW on the top of stack and the stack is PUSHed. PUSH is described in section 5.1.1.2.2.3. Next, SR2 and SR1 are pushed into the stack. Finally new status is set up for the linkage to the interrupt service routine.

ϕ_1 is set from Normal ϕ_1

<u>Interrupt ϕ_1</u>

↓

Change PC to SNO

Representation for relative addressing

Calculate offset Δ = M2TOS - M2BOS

Store stack no and offset in MSW

Generate remainder of interrpt MSW → M1BR

Push*

Reset interrupt indicator

(SR2) → M1BR

Push*

(SR1) → M1BR

Push*

Fetch the appropriate interrupt response descriptor, depending upon interrupt category from base of local stack

Set status INT serv routine address → SR (PC and SC)

LL indication → SR1

ϕ_1 exit

<u>Legend</u>

MSW - Mark Stack Word
M1BR - M1 Buffer Register
PC - Program Counter
LL - Lexical Level
SNO - Stack No. and Offset
* - indicates micro-subroutine

Figure 5.1-11: Interrupt Enter Micro Subroutine

SR1 is entered into M1SR1 during $\phi 1$ exit. SR2 is entered into M1SR2 during the normal copy cycle. The $\phi 1$ exit routine always is followed by Normal $\phi 2$ as indicated.

After the interrupt service routine is executed a RTRN instruction is executed, which reestablishes the processor's internal status.

5.1.1.2.2.2 Instruction Fetch: The actions to be followed by the Instruction Fetch micro subroutine are indicated in Figure 5.1-12. All instructions are either of a one or two syllable categorization. A LTS4 is clearly one syllable. LTS10 is a two syllable instruction. LTS15, 32 and 64 are categorized as one syllable instructions followed by 2, 4 or 8 byte literal data fields.

Two registers control the sequencing of the instruction fetch. The syllable pointer (SP) indicates the number of bytes contained in the instruction. The syllable counter (SC) indicates the byte position within the 8 byte M2 instruction fetch at which the instruction starts.

Upon entrance into the instruction fetch sequence the SC is updated to point to the beginning of the next instruction. If SC $\geq$ 8 then the next instruction is not in IR and must be fetched from M2. In this case the program counter (PC) is incremented and used as an address for an M2 read double (8 bytes) operation. The word received from M2 is loaded into IR byte positions 0 thru 7 (IR0,1,2,3,4,5,6,7).

If the first byte of the instruction is in IR, it is shifted into IR byte position 0. The instruction is then decoded. A one syllable instruction causes 1 $\rightarrow$ SP and the instruction execution phase is entered. If the instruction is two syllables then a determination must be made as to whether the second syllable is in IR1 or must be fetched from M2. If it is fetched from M2, the four byte M2 word is loaded into IR1,2,3,4, 2 $\rightarrow$ SP, 4 $\rightarrow$ SC, and the instruction execution phase is entered.

5.1.1.2.2.3 PUSH: The PUSH micro subroutine, whose flow chart appears in Figure 5.1-13, involves both the M1 and M2 portion of the stack. Figure 5.1-14 depicts these two portions.

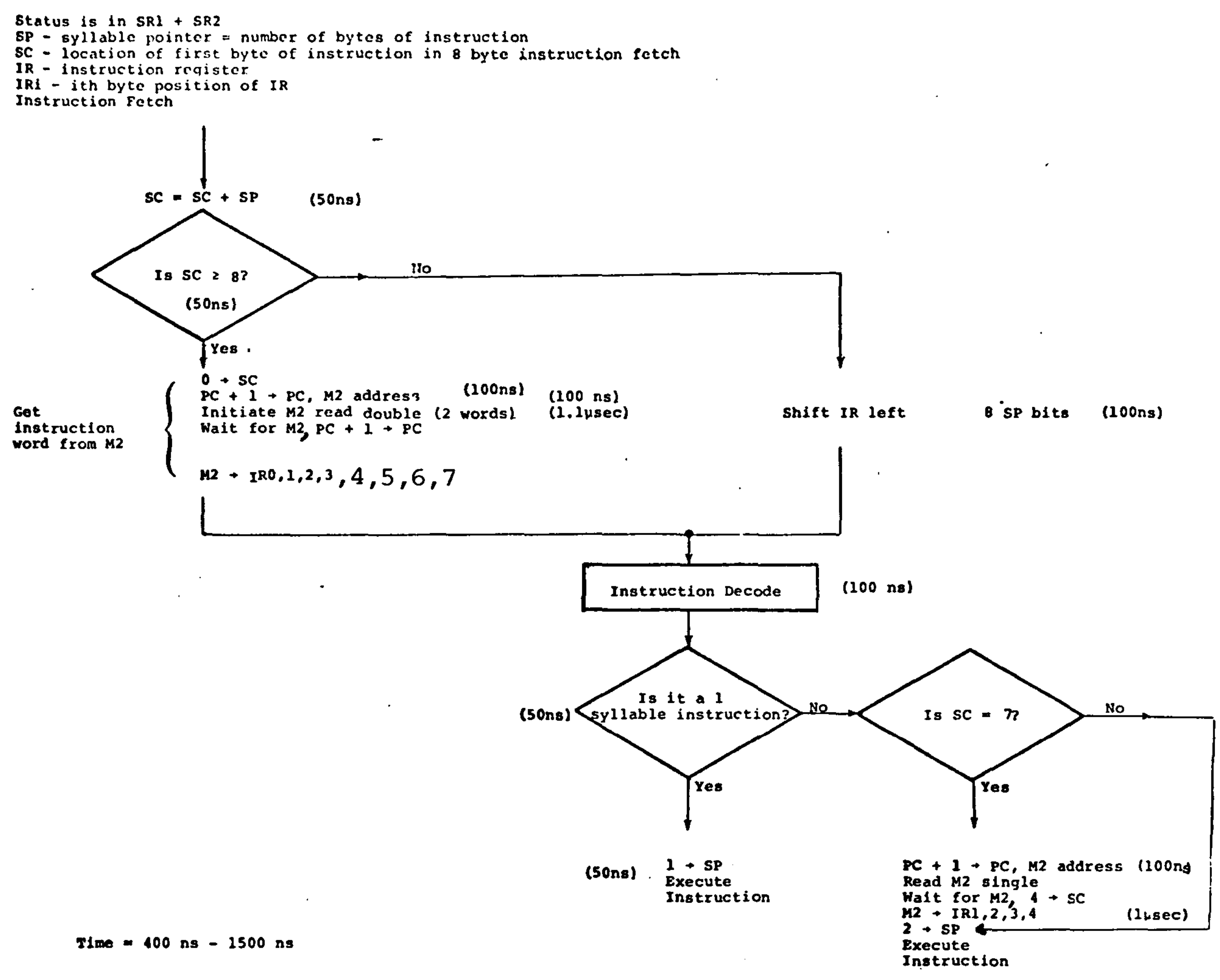


Figure 5.1-12: Instruction Fetch

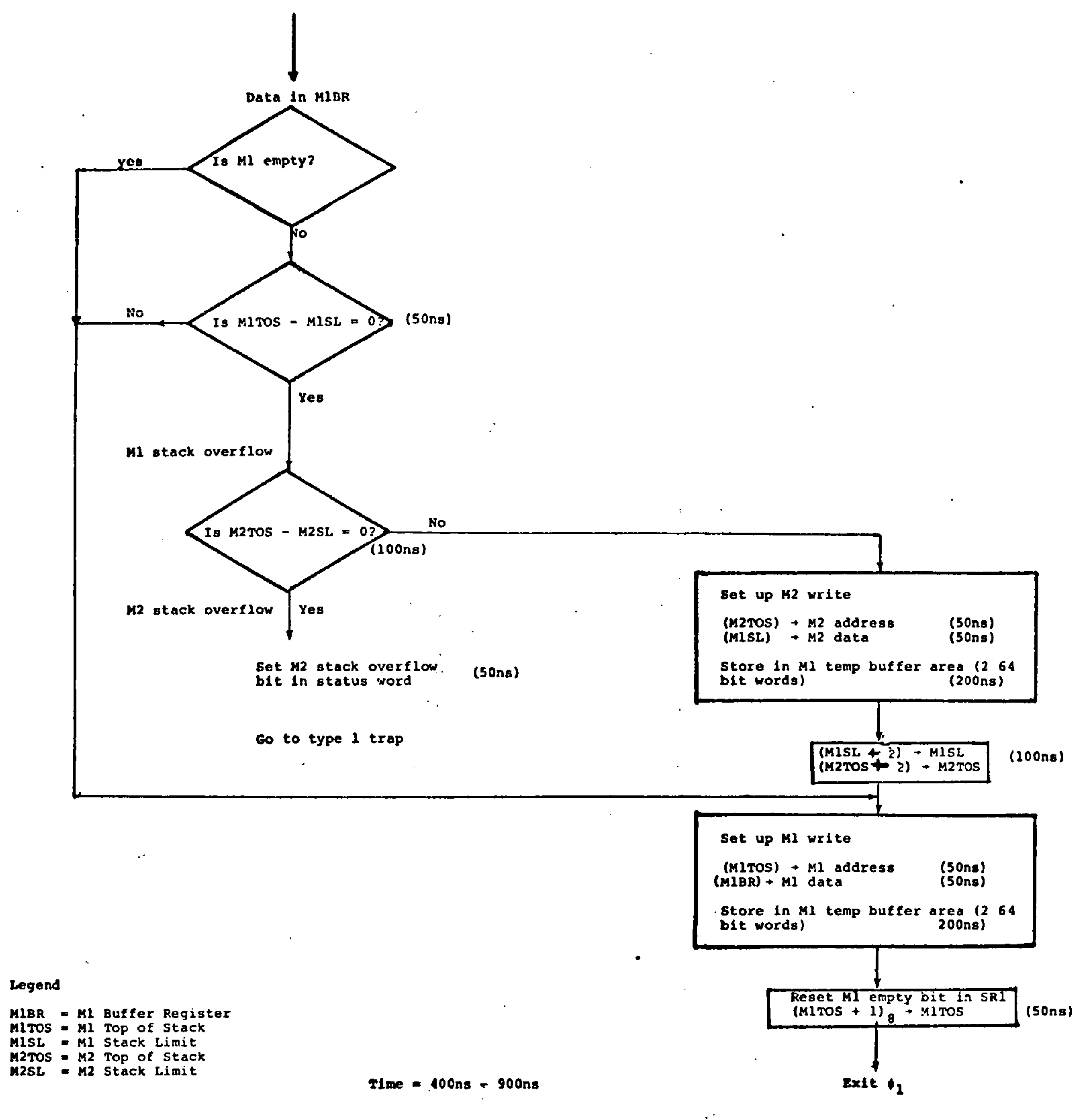


Figure 5.1-13: PUSH

M1 portion of stack is contained in M1 locations 0 through 7
M1SL points to bottom of M1 portion of stack
M1TOS points to first empty location on top of stack
M2TOS points to the first empty location in M2
M2SL indicates the maximum limit of M2 stack
M2BOS points to the oldest location in the M2 part of the stack

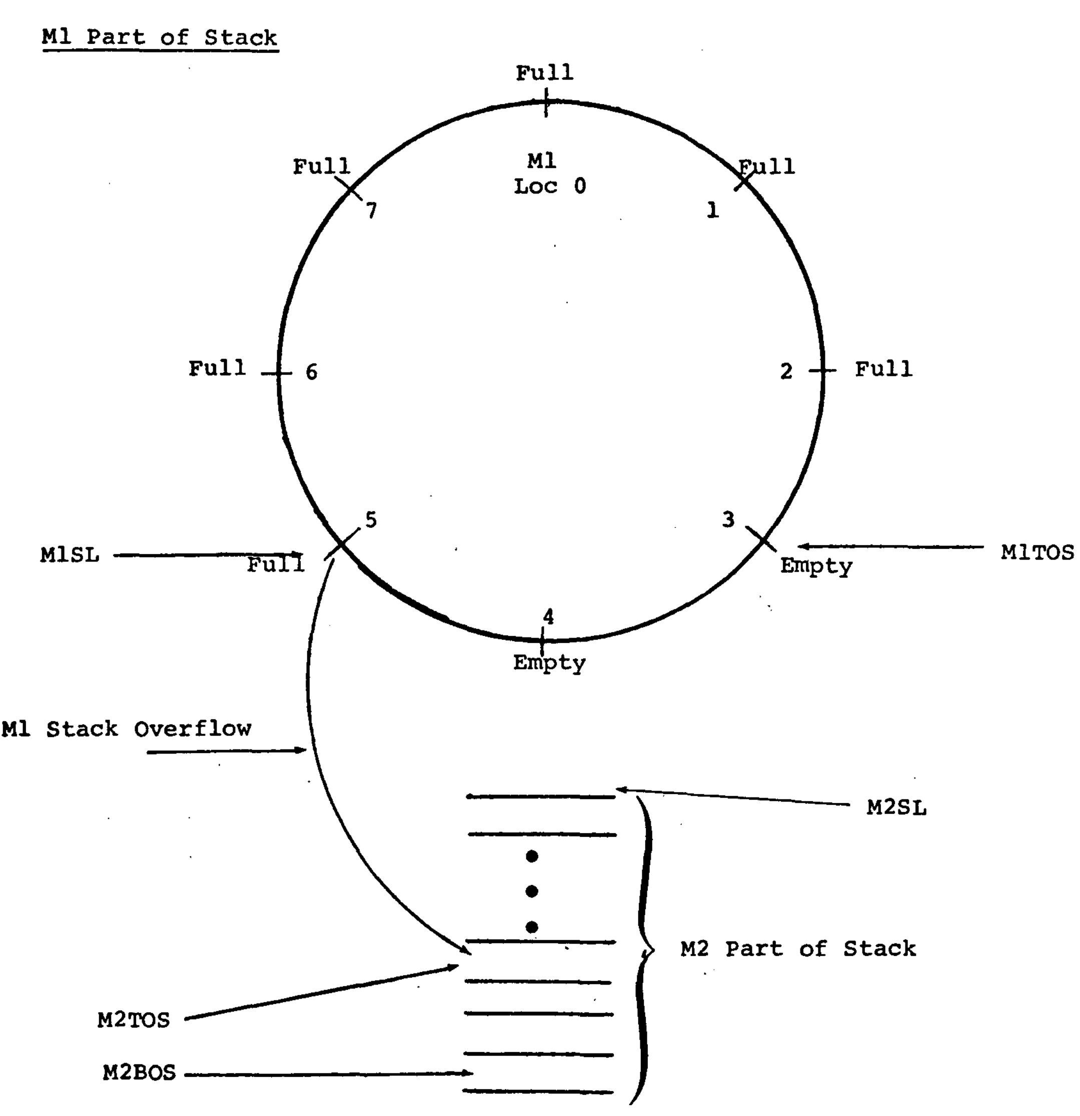


Figure 5.1-14: The Stack

The M1 portion of the stack can be pictured as a wraparound shift register. The oldest data is pointed to by M1SL. The first empty location is pointed to by M1TOS. Whenever M1TOS = M1SL, namely the M1 portion of the stack overflows, the contents of (M1SL) is moved into M2 location indication by M2TOS. If M2TOS ever equals M2SL then the M2 part of the stack has overflowed and a trap is generated. The stack overflow trap routine could then, depending upon conditions, allocate more storage for stack use and change M2SL.

The data to be entered into M1TOS is contained in M1BR. Upon entrance to the routine M1TOS is compared with M1SL to see if the M1 portion of the stack has overflowed. If it has not an M1 write is set up. The M1 address is M1TOS and the data is contained in M1BR. This write is executed into the M1 temporary buffer area for the subsequent copy cycle during $\phi 2$. Finally M1TOS is incremented, modulo8, before the exit $\phi 1$ cycle.

If M1 stack overflows a determination is made as to whether the M2 part of the stack will overflow. If so a trap is entered. If not an M2 write is set up in the M1 temporary buffer area. The M2 address is (M2TOS) and the data is pointed to by (M1SL). M1SL and M2TOS are incremented, followed by the M1 write set up.

5.1.1.2.2.4 <u>POP</u>: The POP micro subroutine is shown in Figure 5.1-15. If the M1 part of stack is empty then an M1 stack underflow exists and a 64 bit read from M2 must be initiated with an M2 address of (M2TOS)-2. The M2 words are placed into M1BR. On the other hand, if the M1 stack is not empty, the contents of ((M1TOS)-1) is read and placed into M1BR.

If the M1 part of the stack becomes empty the condition is set into SR1 for input into the next POP sequence. Every PUSH sequence will reset this empty condition.

5.1.1.2.2.5 <u>Effective Address Generation (EA)</u> (Lexical Level Offset Addressing): When ever an operand-meta-operator is encountered an effective address (EA) must be calculated. The format of this class of instruction is:

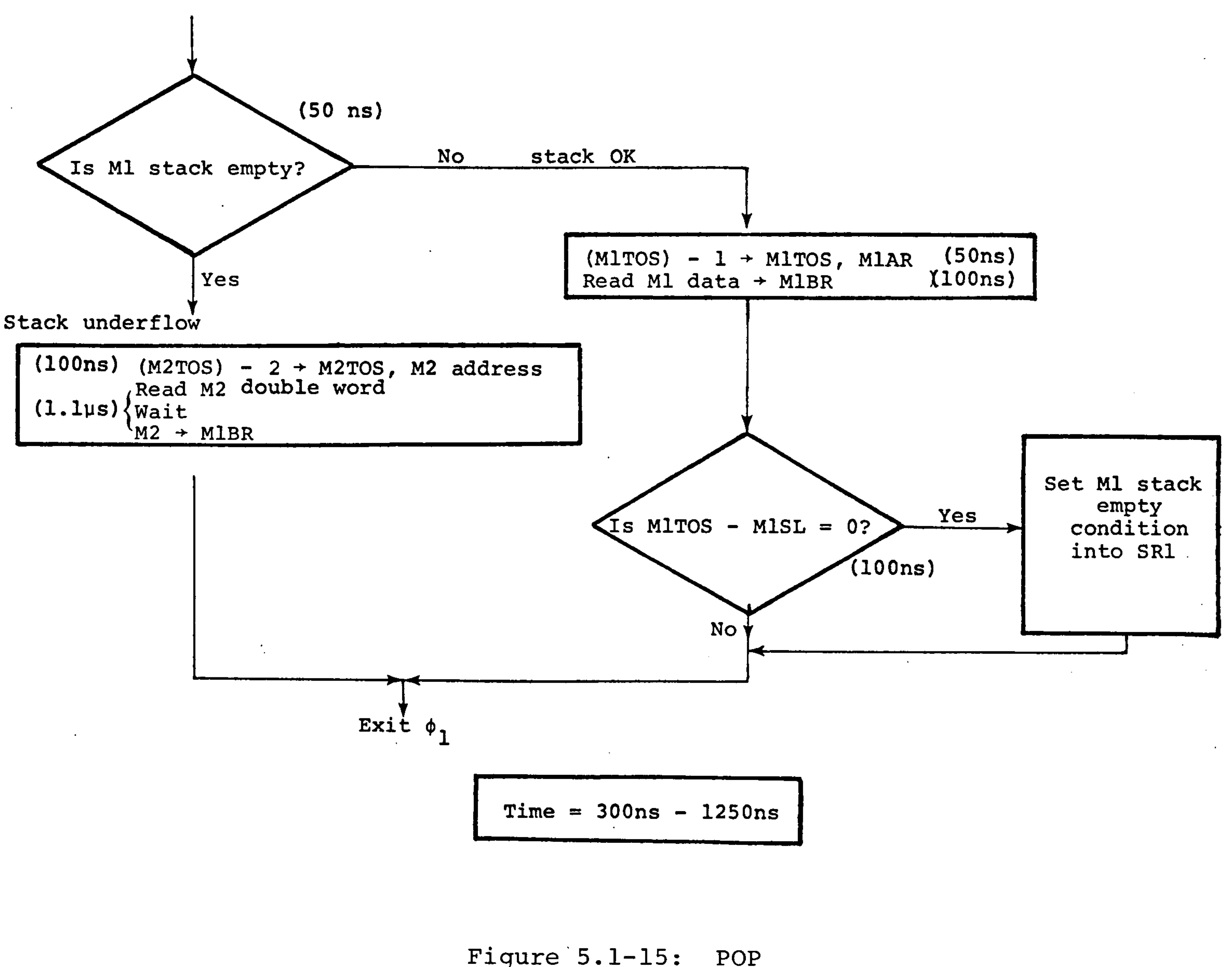
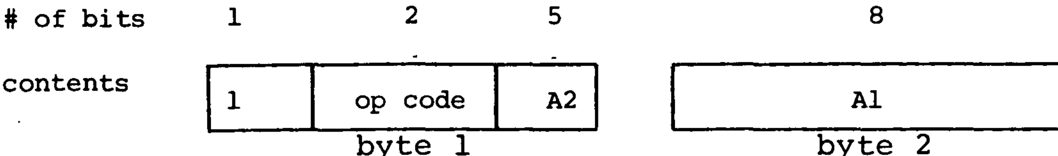


Figure 5.1-15: POP



The address couple A2||A1 forms a 13 bit field, a_{12}, a_{11}, a_{10}, ... , a_0 which is interpreted as follows:

a) The lexical level indicator, ℓℓ, is the key to the interpretation of A2||A1. The first step is to find the positive integer m where:

$$2^{m-1} < \ell\ell \leq 2^m \qquad (50ns)$$

b) Form field 1 where

$$\text{Field 1} = a_{12}, \ldots , a_{13-m} \qquad (50ns)$$

c) Fetch from M1 the base register specified by field 1. Denote this base register by BRm (100ns)

d) BRm is in SNO representation. The absolute M2 location must be determined by fetching it thru the CAM. See section 5.1.1.2.2.7 for details.

e) Next field 2 is formed

$$\text{Field 2} = a_{12-m}, a_{11-m}, \ldots , a_0 \qquad (50ns)$$

f) Finally the effective address (EA) is formed where

$$EA = (BRm) + \text{Field 2} \qquad (100ns)$$

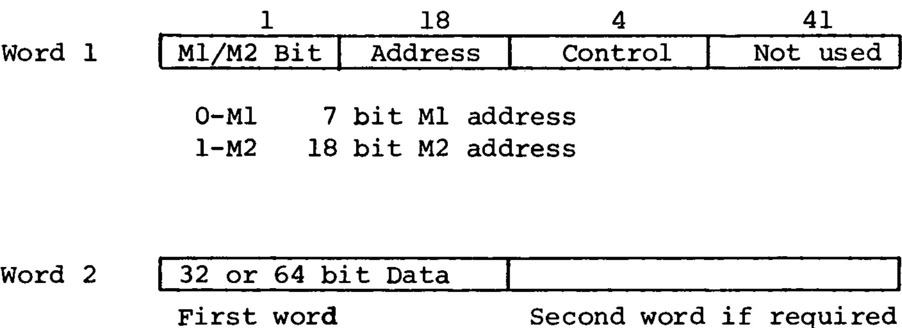
This addition only occurs to the offset portion of (BRm)

g) Total execution time: 350ns

5.1.1.2.2.6 Stack Fetch: When information is required from any location except the top of stack, a stack fetch micro subroutine must be executed. A top of stack fetch is accomplished by the POP routine. Figure 5.1-16 shows the sequencing of the stack fetch micro-subroutine.

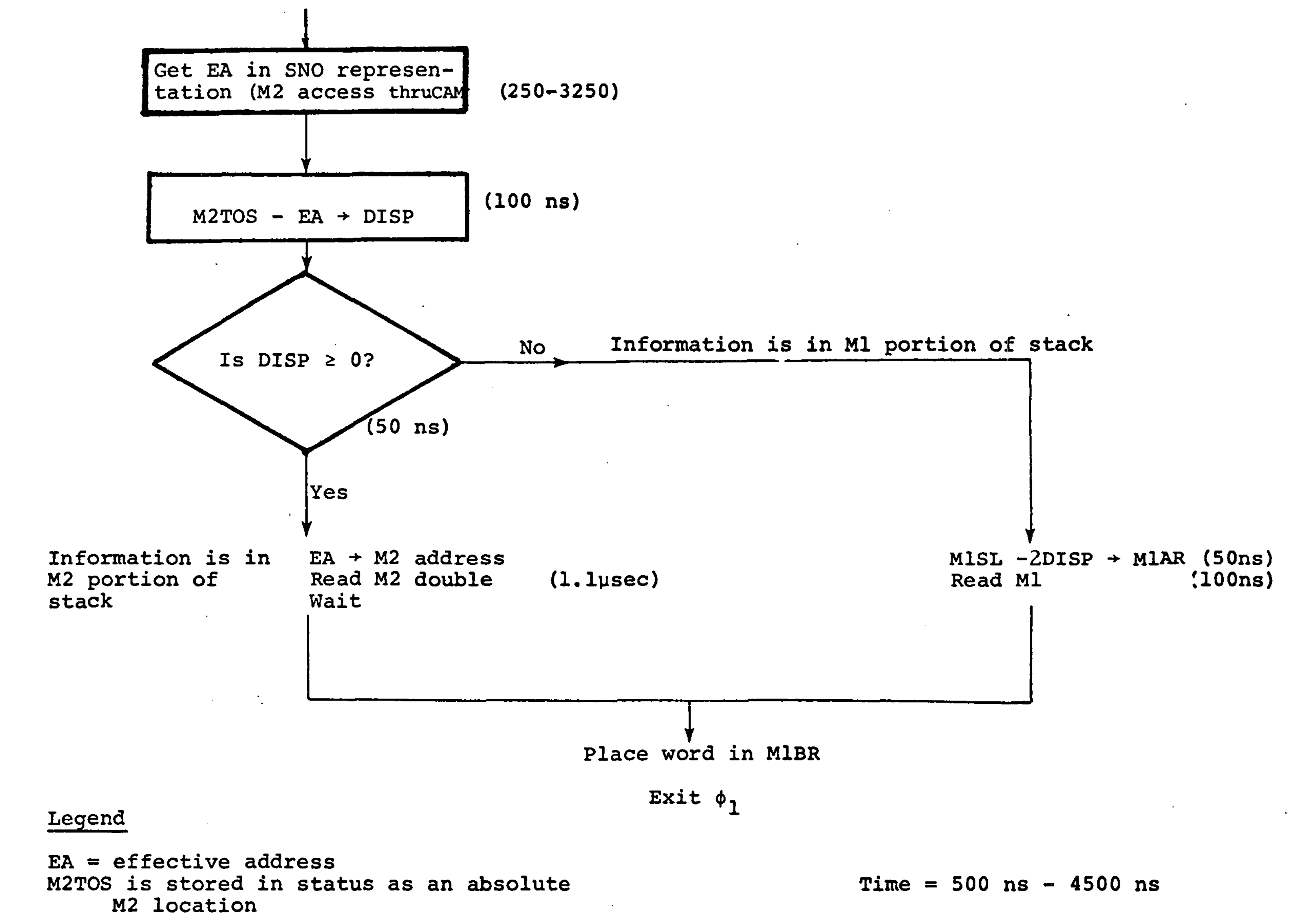
The main test to be performed is to determine whether the information to be fetched is in the M1 or M2 part of the stack. This is accomplished by the calculation of the displacement DISP. Information is then read from either M1 or M2 and placed in the M1BR.

5.1.1.2.2.7 M1 Temporary Storage, Exit ϕ1 and Normal ϕ2: The following format is used in placing information into the M1 temporary buffer:



The first bit indicates whether the write is to take place into M1 or M2. This bit also indicates how to interpret the 18 bit address field. M1 write operations are always 64 bits wide. M2 write operations may be either single (32 bits) or double words (64 bits) and can require simplexed or redundant storage. This information is contained within the control field of word 1.

One might comment as to the number of unused bits in this format and suspect an inefficient utilization of M1 space. However, realize that the M1 temporary buffer is small and its contents are very dynamic. Packing the words would introduce a large overhead of execution time to assemble and disassemble the packed format.



Legend

EA = effective address
M2TOS is stored in status as an absolute M2 location

Time = 500 ns - 4500 ns

Figure 5.1-16: Stack Fetch

Figure 5.1-17 shows the complete normal $\phi 2$ sequence including the copy cycle and the lead in sequence of exit $\phi 1$. This represents more detail than was indicated in Figure 5.1-2. Although not indicated previously the temporary storage counter, TSC, is incremented every time an M1 or M2 write operation is entered into the M1 temporary buffer area. TSC is used to control the number of copy cycles. If TSC = 0 then the copy cycle is over, the phase indication is changed to $\phi 1$ and M1SR1 is updated.

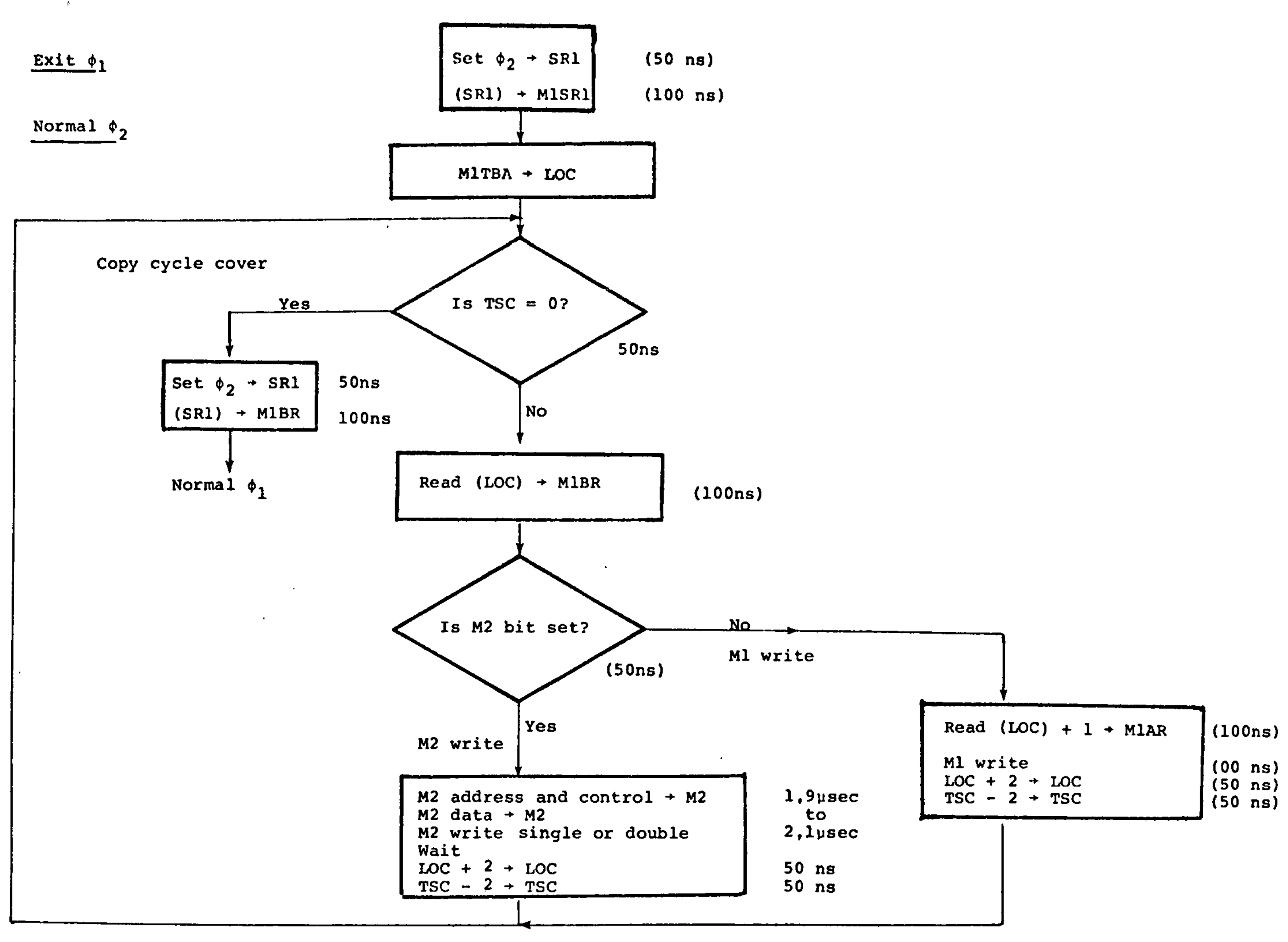
The exit $\phi 1$ and normal $\phi 2$ micro sequences are used by every instruction, as well as the interrupt and trap enter and exit sequences.

5.1.1.2.2.8 M2 Access through the CAM: The CAM was introduced into the processor design in order to save a number of levels of indirection in fetching a descriptor. When a descriptor is desired the SNO is used as the key for an associative search through the CAM. If an association is found, the address obtained from the CAM is used to fetch the descriptor from the descriptor cache in M1.

If there is no association, two parallel functions must be performed. The desired descriptor is fetched from M2 by indirection through the stack vector. Simultaneously space is found in the descriptor cache for the new descriptor, and a CAM entry is associated with it.

If there is a vacant (not occupied) location, it is used. A vacant location is found by an association on OB = 0. Any vacant location can be used. If, on the other hand, all the locations in the descriptor cache are occupied (all OB's = 1), then a modified LRU algorithm is used to displace one of the entries.

The CAM control counter (CC) is used to cycle through the CAM. If the location to which CC points contains a NAB = 0 then this location is used for storage of the descriptor. If, however, the NAB = 1, it is set equal to 0, CC is incremented and the search continues. The maximum search time for this algorithm is 32 iterations of 200 ns each which is 6.4 μsec. However, this is the worst case situation. A more typical number would probably be 3.2 μsec which is about the same time required for fetching the descriptor through the stack vector.



Legend

TSC = temporary storage counter
M1TBA = location in M1 of beginning of temporary buffer area

Time = 200 ns + (2,350 (2,150) x number of M2 writes + 450 x number of M1 writes

Figure 5.1-17: Exit ϕ1 and Normal ϕ2

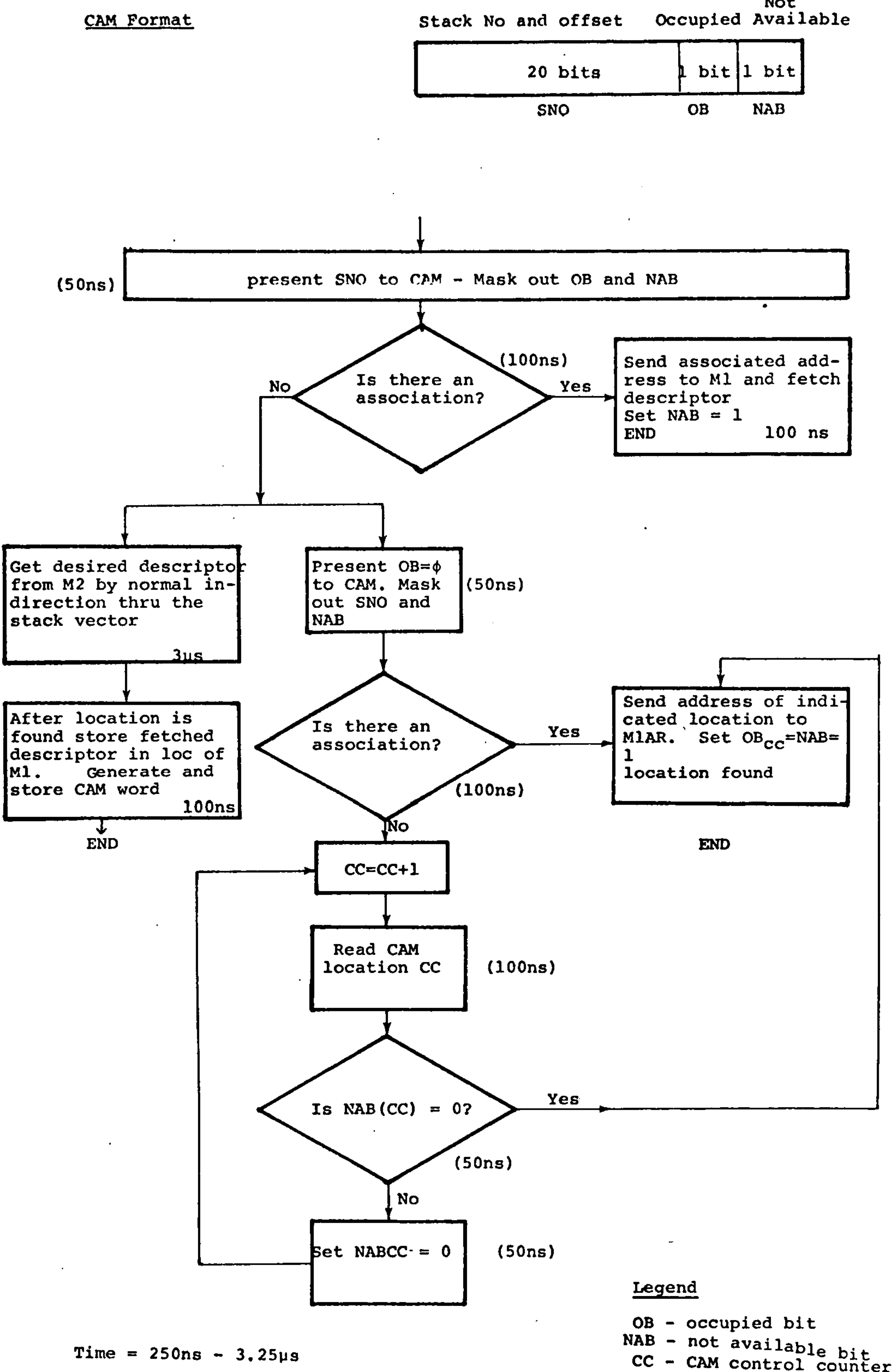


Figure 5.1-18: M2 Access Thru CAM

5.1.1.2.3 Instruction Execution Flow Charts. The micro subroutines, presented in section 5.1.1.2.2 are used as building blocks for the execution of instructions. Not all the useful subroutines were presented. This is a matter of detailed design. Similarly only some typical instruction flows are presented in this section. However a sufficient diversity of examples is presented to give the reader a flavor of the detailed design, and to demonstrate the practicality of implementing the instruction set. The definition of the instructions has previously been presented in Chapter 2. Detailed explanations of these flow charts are not necessary.

The execution time estimates are based upon the micro subroutines presented previously. The wide dispersion in execution times occurs because information needed for execution can be found in M1 or M2. If all information is read or written into M1 then the minimum time will occur. When all information is read or written into M2 then the maximum time situation will occur. In a sense the maximum times would represent an implementation where M1 is too small or nonexistent.

Copy Instruction

Effective Address Generation (EA)	350
M2 access thru CAM	250 - 3250
Stack Fetch	(300 - 1150)
Push	(400 - 900)
Exit ϕ_1	(650 - 3000)
	1.95μs - 8.35μs

Figure 5.1-19: COPY

BSET

POP to get bit number → SCR		(300 - 1250)
Read 00 ... 1 from LR		50
LR → shifter → BMANR		50
POP to get operand;		(300 - 1250)
M1BR → ORed into BMANR	(This sets bit)	(100)
BMANR → M1BR		50
Push		(400 - 900)
Exit ϕ_1		(650)
		1.9μs - 3.8μs

Figure 5.1-20: BSET

LTS4

LIT = 5 bit field in IR0,1	
Convert LIT to DPFP → M1BR	200
Push	400 - 900
Exit ϕ_1	650 - 3000
DPFP - Double Precision Floating Point	Total 1.250μs - 4.1μs

Figure 5.1-21: LTS4

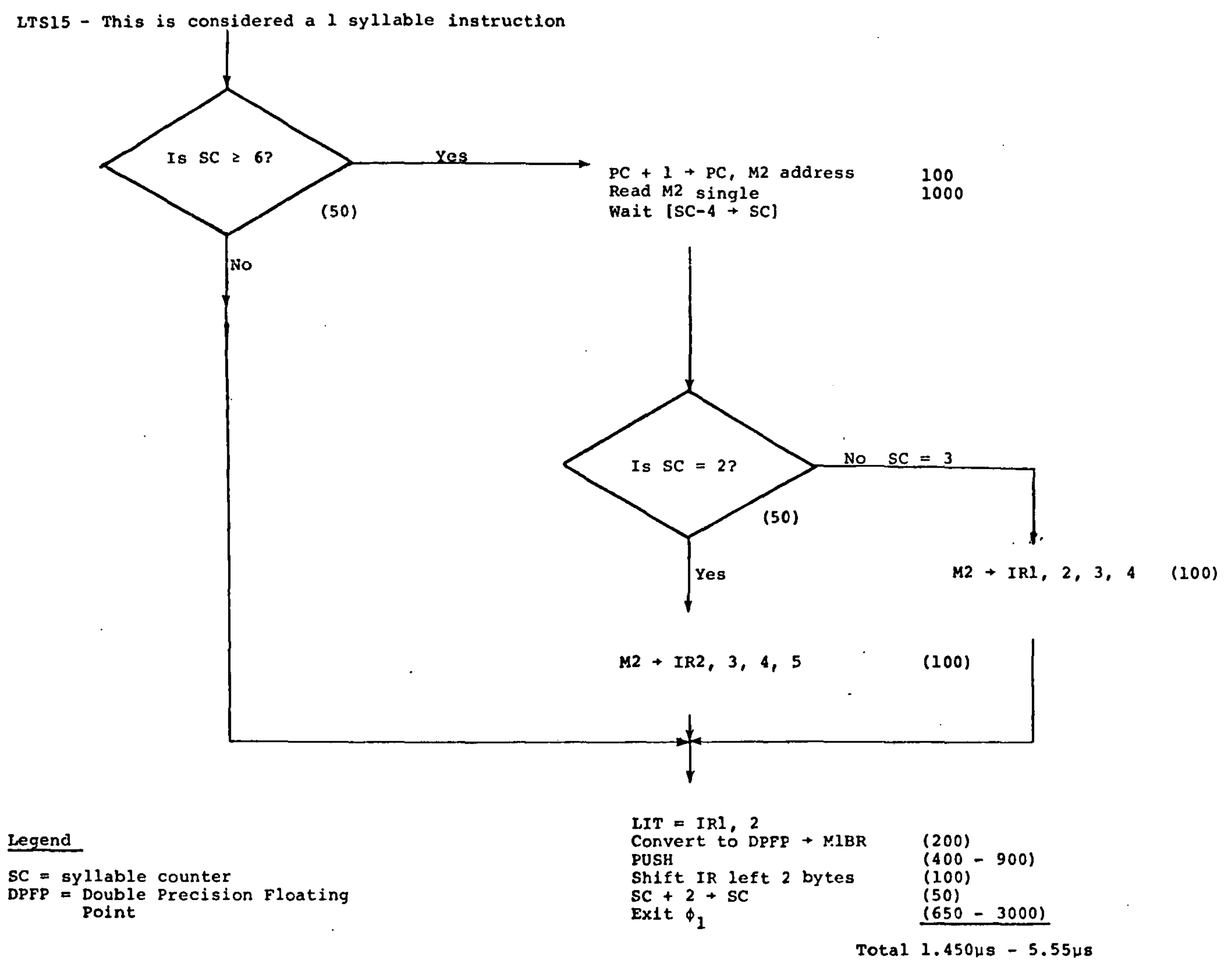


Figure 5.1-22: LTS15

ADR

Effective Address Generation	(350)
Format ADW with read/write access	(200)
Place in M1BR	50
PUSH	400-900
Φ1 exit	650-3000
	Total 1.650 - 4.5 μ sec

Figure 5.1-23: ADR

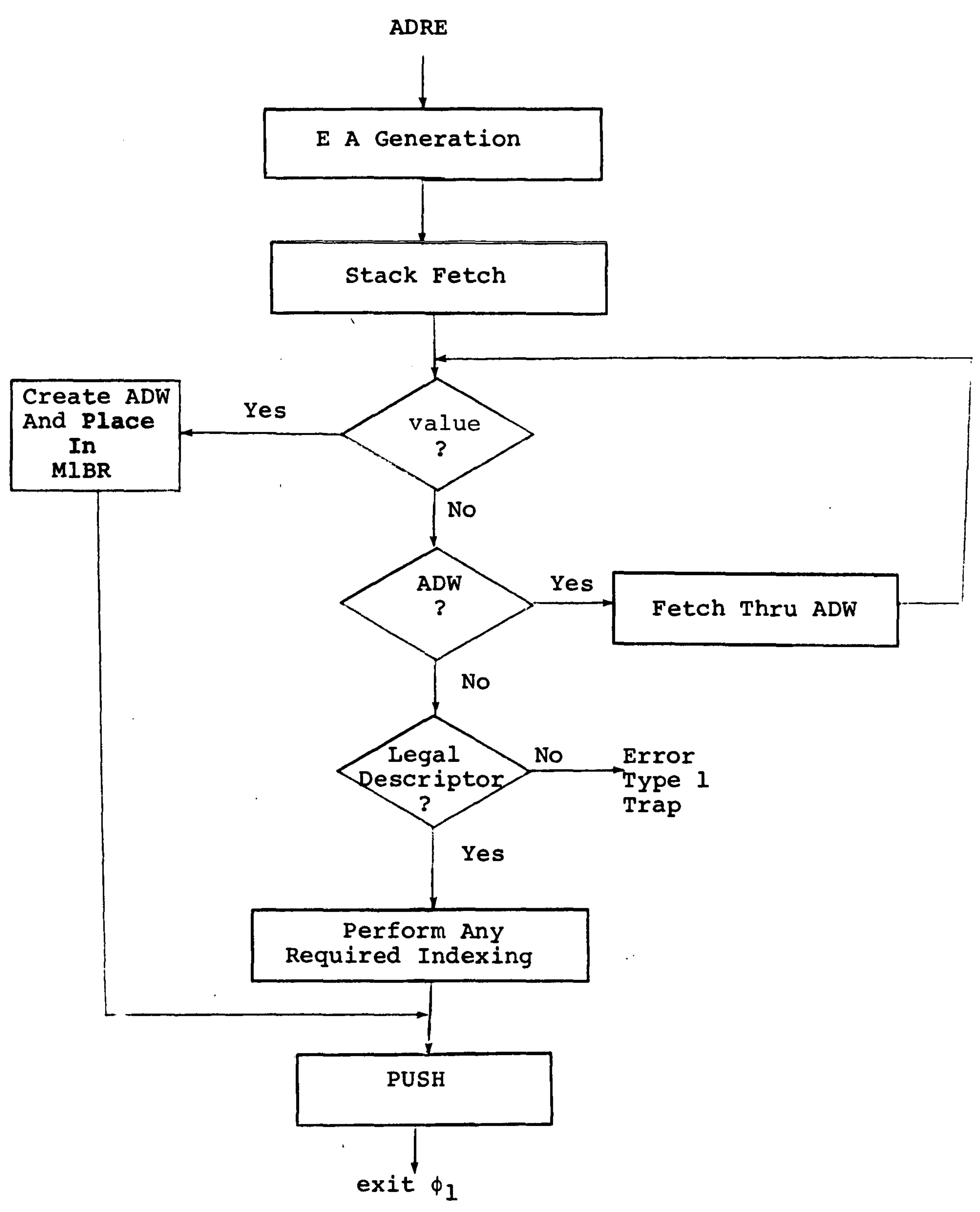


Figure 5.1-24 ADRE

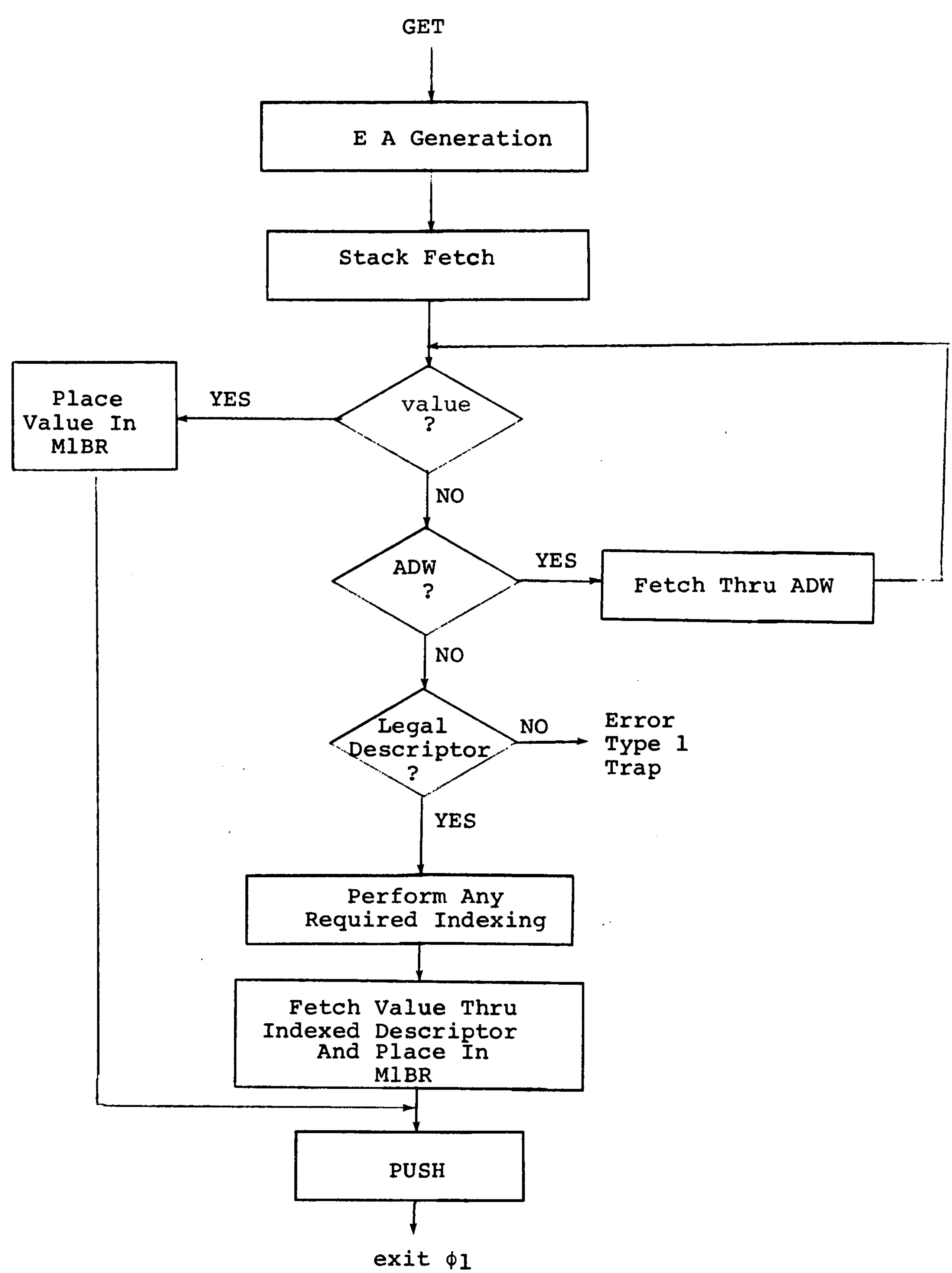
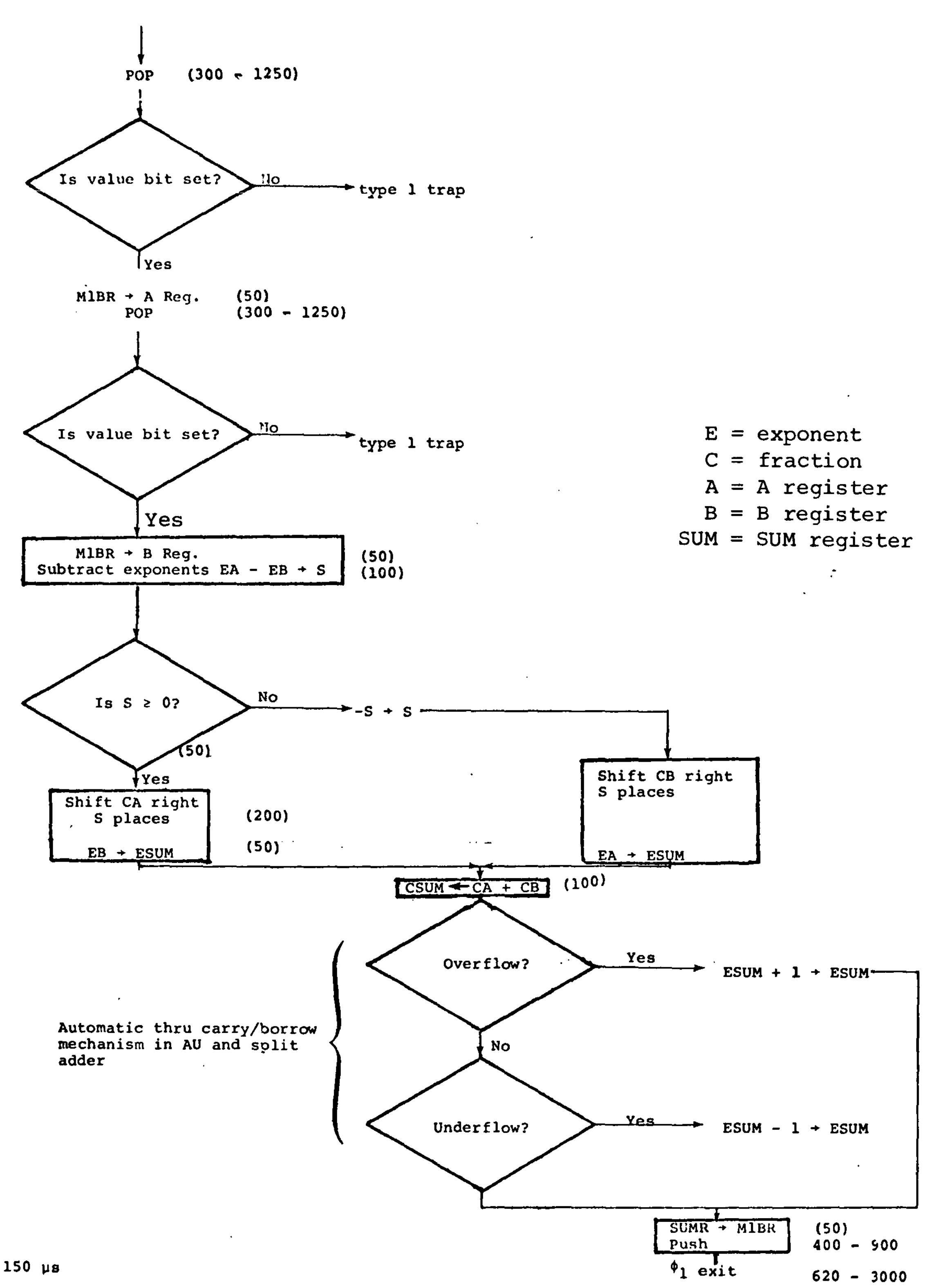


Figure 5.1-25 GET

Floating Point Add Instruction



Total = 2300 - 5.150 μs

Figure 5.1-26: Floating Point ADD

Stack Organization

Initial Cycle		Each Subsequent Cycle
LV	Limit value	LV
IV	Increment value	IV
INIT	Initial value	LVA
LVA	Loop variable address	BA
BA	Branch address	

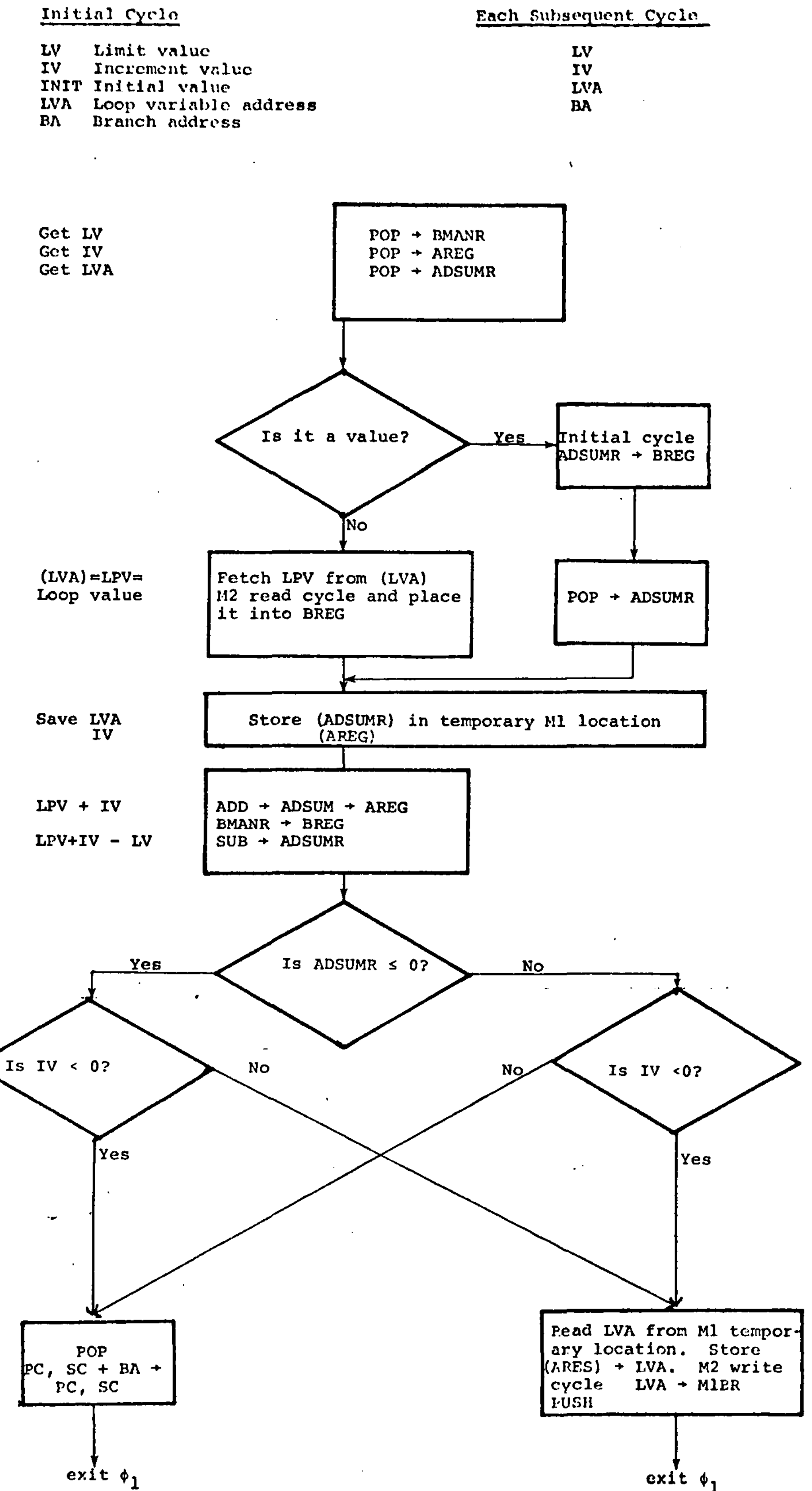
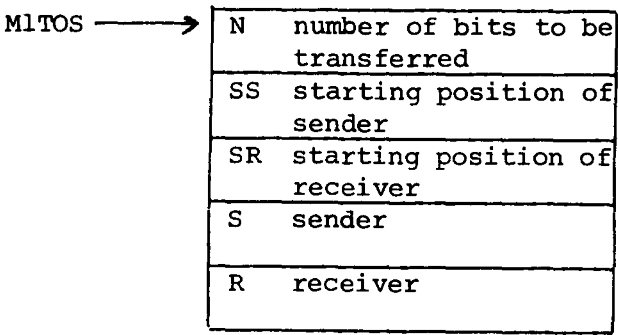
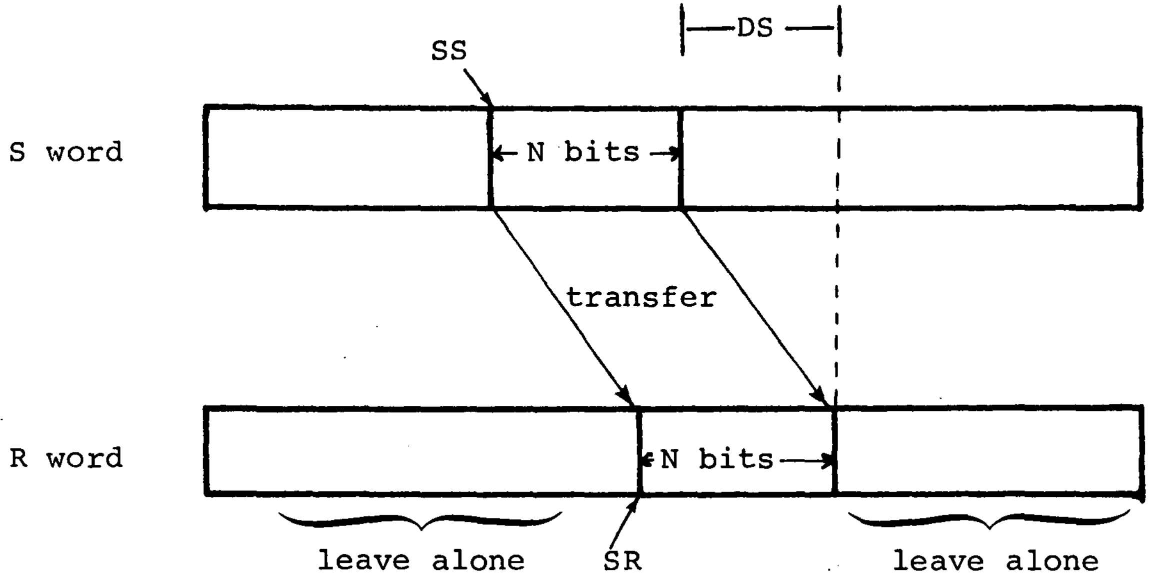


Figure 5.1-27: FOR Instruction

BTRN explanation:

Stack organization





DS = differential shift

Figure 5.1-28: BTRN

Get N	POP → SCR fill with 1's, shift right
	Read "10 ... 0" from LR
(BMANR) = $\overbrace{111 \cdot 1}^{N\ bits}$ 00 ... 0	LR → shifter, shift → BMANR
Get SS	POP
	M1BR → A Reg
Get SR	POP
	M1BR → B Reg
Calculate DS	SUB → ADSUM
<u>Set up MCR and SCR</u>	
	Set cycle left shift into SCR, B Reg → SCR count
(MCR) = $\underbrace{00}_{SR\ bits}$ $\overbrace{1111}^{N\ bits}$ 0000	BMANR → shifter → MCR
	B Reg → BMANR ADSUM → SCR
	A Reg → shifter → masker → OR into BMANR
	BMANR → M1BR
	PUSH
	Exit ϕ_1

Figure 5.1-29: BTRN Sequence

5.1.2 Internal Bus and Operating Memory Implementation

The internal bus provides the interconnection between the P's and I/O's with the operating memory M2 modules. The structure of the internal bus is intimately connected with the manner in which the M2 modules are to be utilized. For this reason both topics will be covered in this single section.

5.1.2.1 An M2 Module: The fault tolerant discussion concerning M2 failure recovery introduced the concept of two interleaved M2 units to provide redundant storage in consecutive addresses within different M2 units (see section 4.4.2). The processor implementation discussion (Section 5.1.1) indicates that double words (64 bits) are often accessed from M2. A redundant write of a double word would require four M2 accesses. Since this is a common occurrence the four accesses should be executed almost simultaneously for performance maximization.

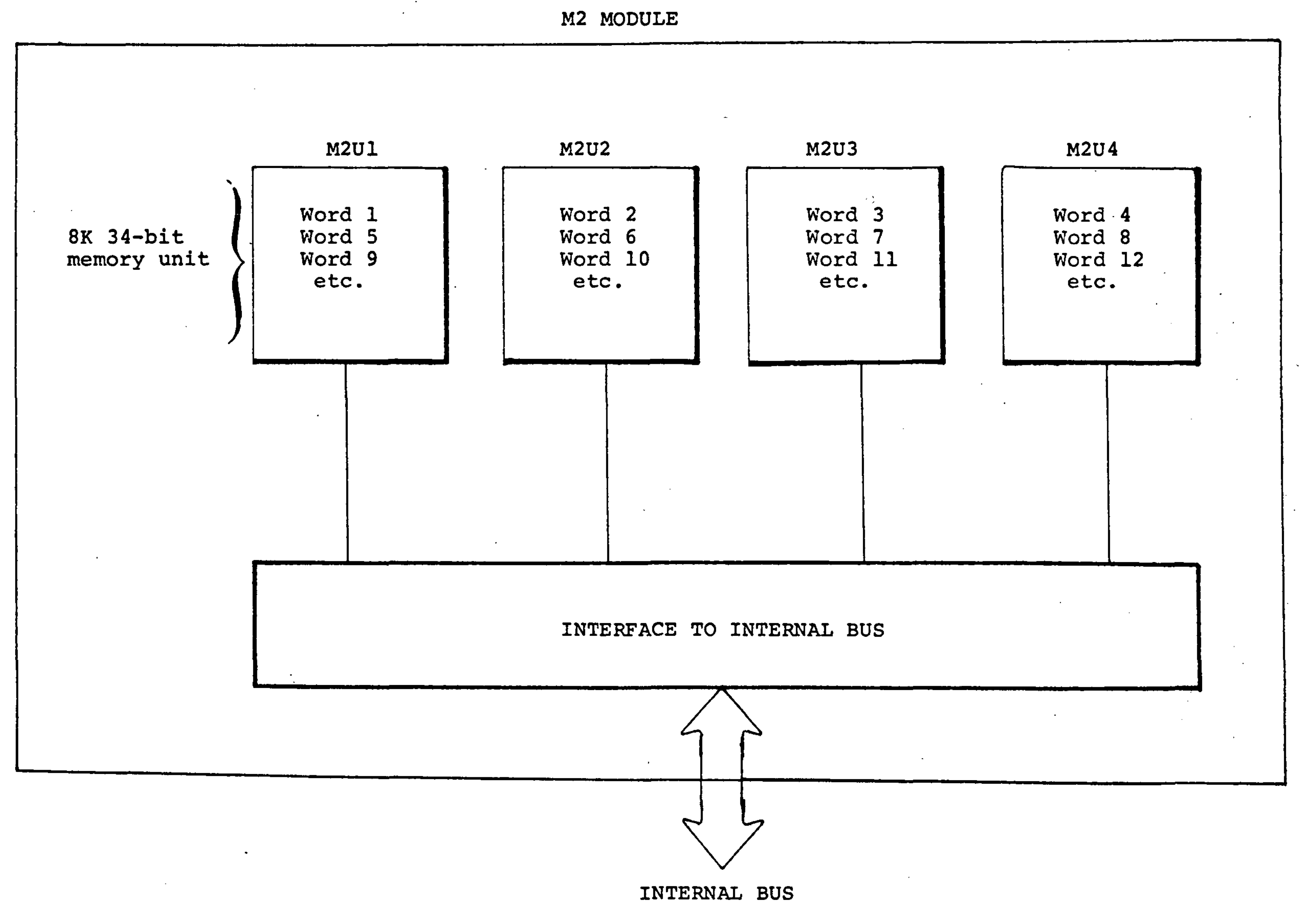
This leads us to define an M2 module as consisting of four M2 units, which are interleaved on the lower order address bits (see figure 5.1-30). The entire M2 system then consists of four M2 mdoules. Each module contains four 8K interleaved units (32K words of memory). The entire M2 system possesses 128K of memory.

5.1.2.2 Internal Bus Design Goals: The interface to the internal bus and the internal bus structure is a trade off between performance and hardware. The following design goals are suggested:

a) The one way delay between a P and M2 unit, including the P's M2 interface, bus delay, and the internal bus interface of M2 should be less than 100ns. A 10 MHz bit rate on a wire is well within the present state of the art technology.

b) The internal bus should not degrade system performance. This means that if a P element desires access to an M2 unit that is not being cycled, the internal bus should not introduce contention.

5.1.2.3 Processor and I/O Impact: The fault tolerant discussion indicates the necessity for each processing and I/O unit to be internally dual with two separate links to the internal bus. For a three dual processing unit, dual M2 interfaced I/O system, which is our nominal design, a total of eight communicating elements can possibly be interfaced to each M2



M2Ui = M2 module Unit i   i = 1,2,3,4

Figure 5.1-30: M2 Module Interleaving

module. An expanded system would require two additional interfaces per processing unit. It is exactly the control of these interfaces which provides the speed/cost (hardware) tradeoff of the internal bus and memory interface design.

5.1.2.4 How Many Interfaces to an M2 Module: At least two interfaces or ports to each M2 module must exist to provide the redundant interface to each P unit. This would allow a maximum of only two simultaneous accesses. In some sense four ports into each four way interleaved M2 module is sufficient to sustain the four possible simultaneous conversations. If four port M2 modules were employed two possible internal bus configurations, shown in Figure 5.1-31 would be possible.

Each M2 access initiated by a P or I/O uses two ports simultaneously. For this reason both configurations show the redundant P or I/O horizontal busses attached to different ports of the M2 modules (vertical bus links).

In configuration 1 a processor pair can access an M2 module only through one pair of ports. If these ports are being used by another processor pair then bus contention arises even though the two requesting units are addressing different M2 units. Configuration 2 eliminates this cause of bus contention by allowing a processor pair to access an M2 module through either pair of ports. Even this configuration presents the possibility of bus contention under the condition that two different processor pairs are accessing the same M2 unit and therefore the four ports are occupied. Another processor or I/O could not access a different M2 unit within the same M2 module. This source of bus contention is only eliminated by utilizing eight ports in each M2 module. This is depicted in Figure 5.1-32.

Although the eight port configuration seems initially to make each M2 module more complex, it completely eliminates any switches in the internal bus. The internal bus structure has degenerated into just an interconnection matrix consisting of wiring and no circuits. The switches have actually been incorporated into each M2 module. Figure 5.1-33 indicates the components of an eight port four way interleaved M2 module.

In the past the hardware designer had to be concerned about the complexity of the cross bar switch and the memory ports. Present day MSI and LSI technology, however, tend to reduce this concern due to the regular structure of the switch elements and ports. Each intersection of the switch must switch 34 horizontal inputs to the vertical bus. All of these switches are identical. (Burroughs in their Interpreter-based aerospace multiprocessor have actually built cross bar switch elements using

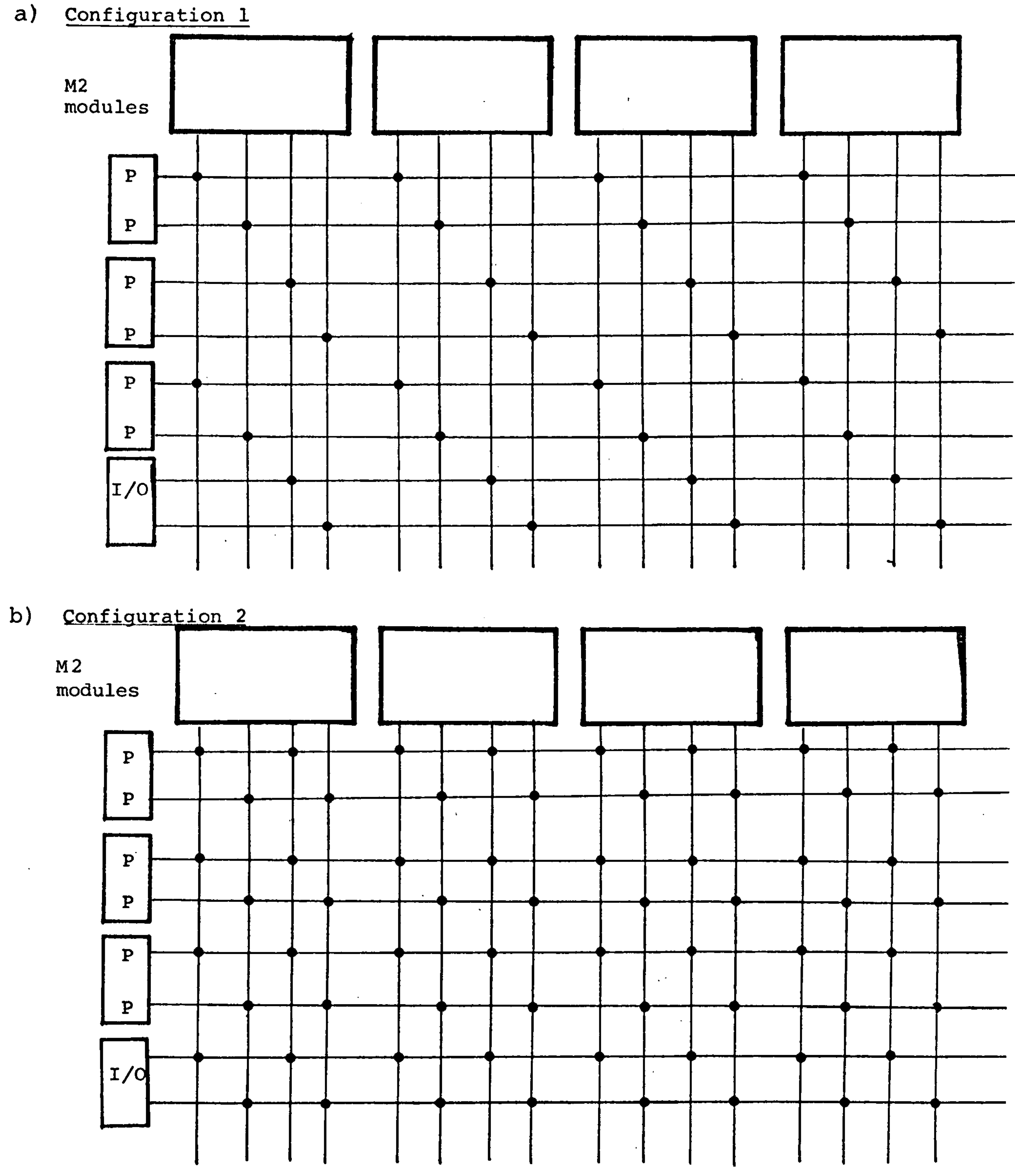


Figure 5.1-31: Four Port M2 Module Internal Bus Configurations

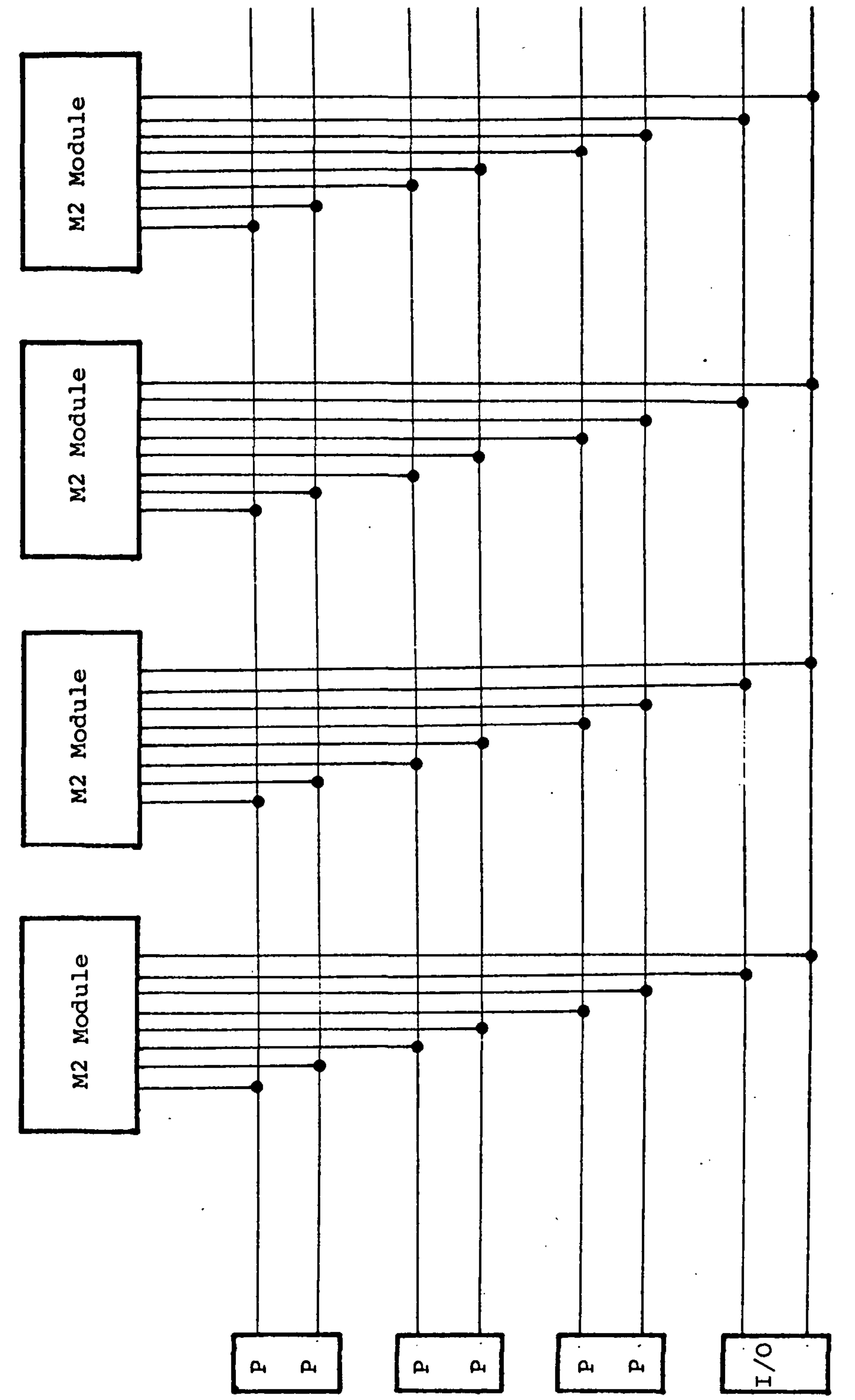


Figure 5.1-32: Eight Port M2 Modules and Dedicated Internal Buses

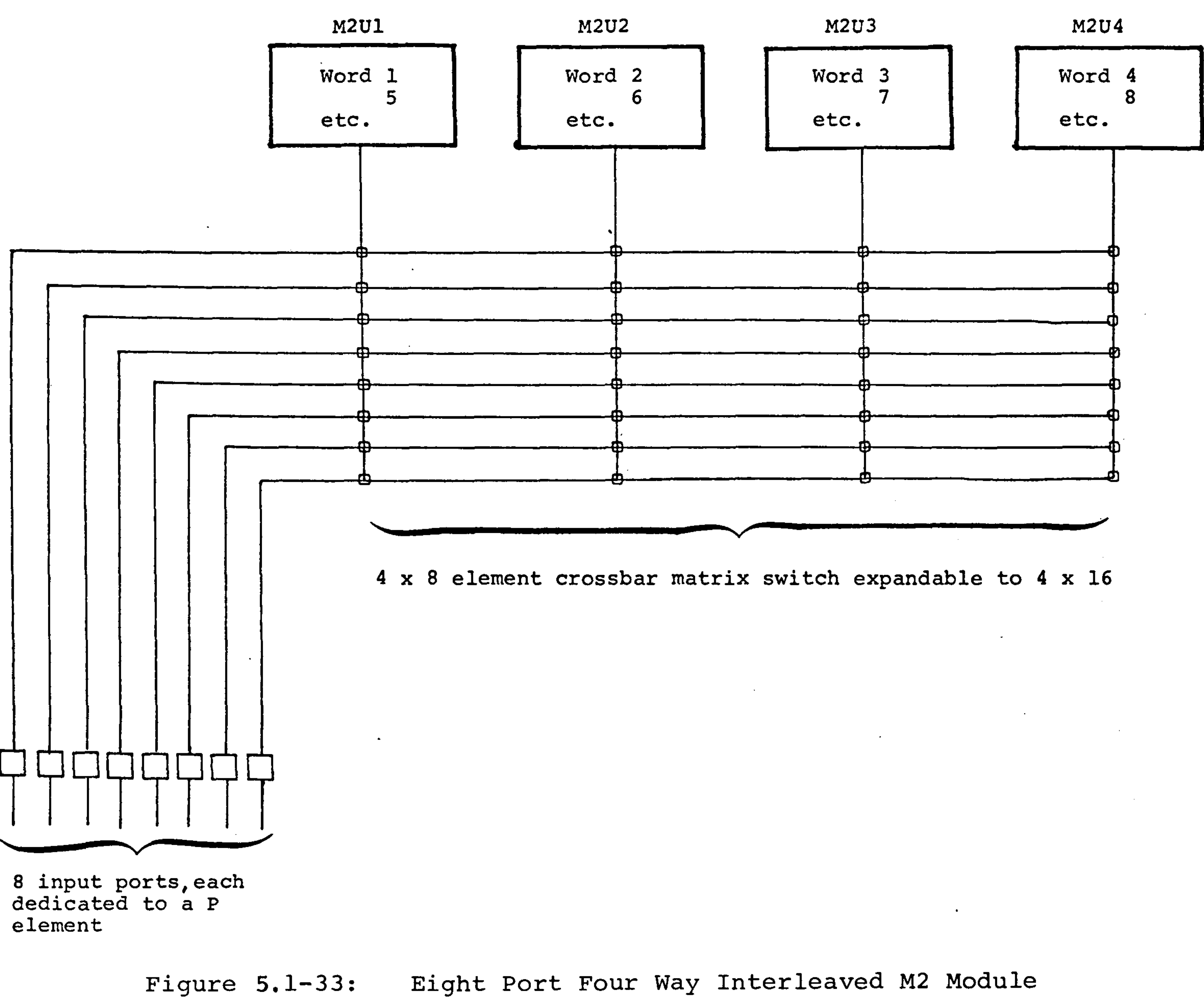


Figure 5.1-33: Eight Port Four Way Interleaved M2 Module

LSI technology). The memory ports also form identical regular structures, in that they contain only one or two 34 bit buffer registers for interface synchronization.

Figure 5.1-34 presents a tradeoff between the three configurations. For configurations 1 and 2 the four ports must be switched, internal to the M2 module, to the four M2 units. This switching involves a 4 x 4 cross bar swtich. Figure 5.1-34 indicates the total number of crossbar switch modes both internal and external to the M2 modules as well as an indication of the external wiring complexity. Configuration 3 possess the best performance (zero bus contention) at a cost of more complex wiring. Whether this is a real factor or not is a function of the packaging of the entire multiprocessor system. For the post-1980 time frame with anticipated packaging technology, all the communicating elements of the multiprocessor could probably be packaged in a single desk sized unit. The interconnections between the P's and M2 modules would probably not be wires at all, but just an interconnecting multilayered signal board. The low complexity of the M2 module ports of configuration 3 over configuration 1 tends to cancel the effect of the difference in crossbar mode switches.

On balance configuration 3, employing 8 input four way interleaved M2 modules is proposed.

5.1.2.5 M2 Commands and Execution Times: Each request generated by a P or I/O and sent to an M2 module possesses a four bit field which is depicted below.

Memory Command Bits

Read = 0	Single = 0	Simplex = 0	Unlock = 0
Write = 1	Double = 1	Redundant = 1	Lock = 1

With this command word a total of 16 different memory commands are possible. For example a 0110 command reads four consecutively stored words, one from each M2 unit, and leaves the M2 module in the unlocked state. The four words consist of two double length words (64 bits) redundantly stored.

The major cycle time of an M2 unit was chosen to be 800 ns. Faster units were not specified because of the substantial power increase. One of the major points of this multiprocessor design study was to achieve performance through architectural innovations rather than pushing the state of the art in component technology. An 800 ns aerospace memory unit is clearly very reasonable for a post-1980 space station.

	Configuration 1	2	3
Total number of crossbar switch modes	96	128	128
External M2 interfaces to M2 modules (wiring complexity)	16	16	32
Contention due to internal bus	high	medium	zero
M2 post complexity	high	high	low

Figure 5.1-34: Memory Port Trade Off Matrix

Considering an internal bus transfer time of 100 ns and an echo check on all write operations, the following execution times for the memory commands can be deduced:

Command	Function	Time
00ϕϕ*	read single	1.0 μsec
01ϕϕ	read double	1.1 μsec
10ϕϕ	write single	1.9 μsec
11ϕϕ	write double	2.1 μsec

5.1.2.6 Memory Conflict: The selected M2 module configuration allows a number of memory commands to be executed simultaneously out of different M2 units. Conflict between the two commands will only take place if both commands require access to the same internal memory unit simultaneously. Figure 5.1-35 illustrates a case where two processing units command an M2 module to execute a read double redundant command. The two commands arrive simultaneously at the four ports. Both processors require one cycle of each of the M2 units. If Processor 1 is the first to be serviced, as is illustrated, processor 2 is only delayed for 1 memory cycle (800 ns) before its command execution is started.

Memory conflict only occurs when there is contention for a memory unit. A model to calculate the effect of memory conflict can be generated under the following conditions:

a) Memory requests are uniformly distributed across the address space and no correlation exists between one memory access and the next. This assumption can be questioned since both program and data do possess locality. Also quite a few double word requests will be made. However, using this assumption, an order of magnitude estimate may be made.

b) Each processor makes a request to memory with probability Pr. Pr is the ratio of the time spent accessing M2 divided by the total processing time of the processor. If the M2 cycle time is very slow compared to the processor cycle time then Pr tends toward 1. However, if many processing unit cycles are used between each M2 cycle then Pr tends to be less than 1.

---

* ϕ = don't care conditions -- either simplex or redundant, lock or unlock.

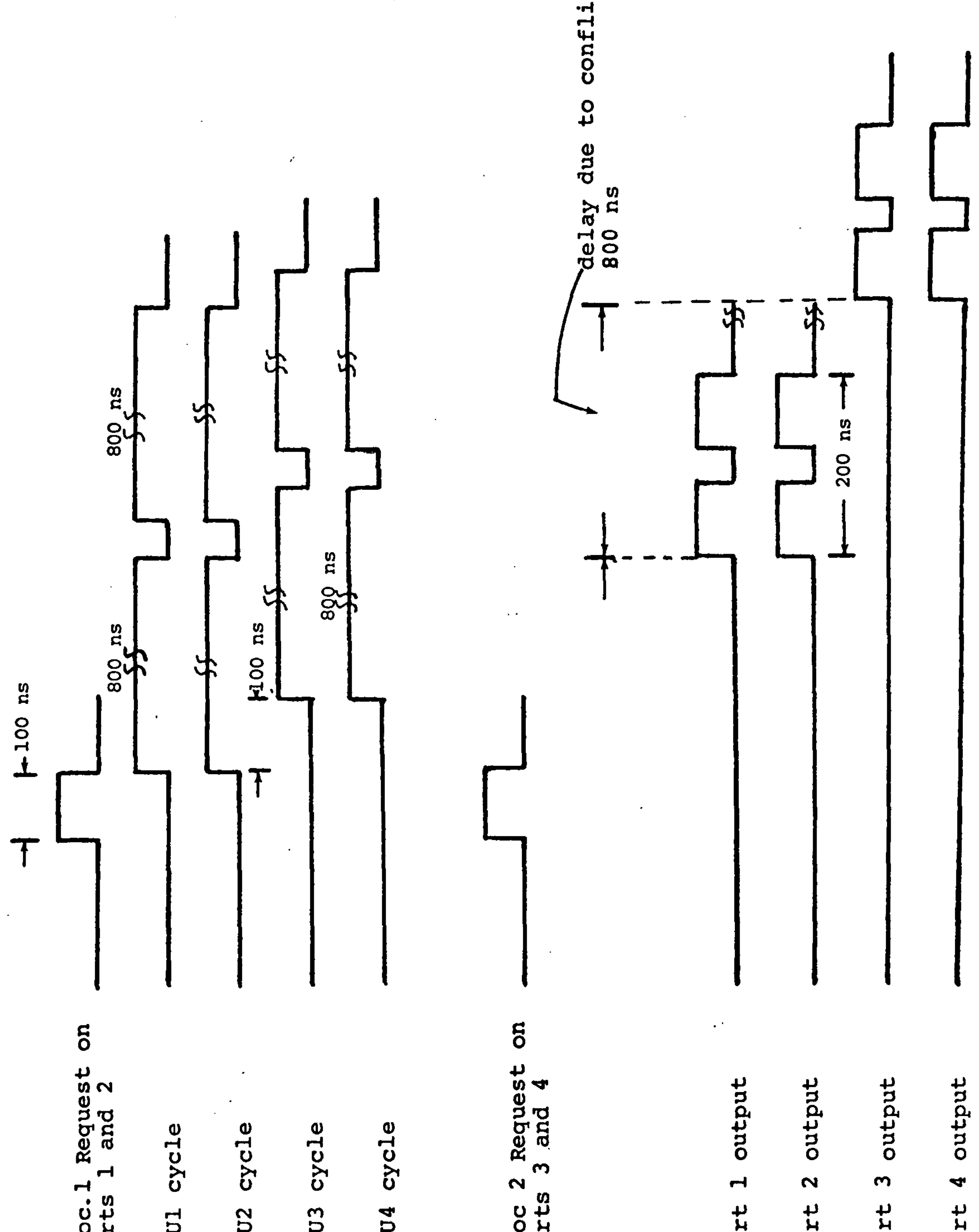


Figure 5.1-35: M2 Module Timing

Pr is a measure of the time allocation between M2 and P. The higher order language instruction set employing M1 tends to create intervals in which the processor is busy and the memory is idle. Therefore, Pr should be less than 1. An evaluation of execution times indicates that a Pr of 3/4 can be achieved with our instruction set; a 1 μsec memory and bus delay, and a 50 - 100 ns processor cycle.

c) There are M memory units which can be accessed independently and simultaneously.

d) There are a total of R processing units and I/O units which can issue requests to the memory unit.

The probability that a given processing unit will request access to any specific unit is Pr/M.

What is the effect of memory contention? The result is to produce an effective M2 cycle time, t_{2eff}, which is larger than the individual M2 unit cycle time $t2$ = 800 ns. The effective M2 cycle time is defined to be the time interval from the inital request for an M2 access to the final completion of the M2 command. It includes the delays due to memory contention. As far as our analysis is concerned all memory commands whether read or write are assumed to require the same time. The difference between memory access and cycle time is ignored. The purpose is to present an order of magnitude estimate of the effect of memory contention and not obscure the result with second order effects.

Given a processor is requesting access to a particular M2 unit, the probability that none of the R-1 other processors are requesting access to the same M2 unit is

$$P_0 = \left(1 - \frac{Pr}{M}\right)^{R-1}$$

The probability that one of the R-1 other processors are requesting access to the particular M2 unit is

$$P_1 = (R-1)\left(1 - \frac{Pr}{M}\right)^{R-2}\left(\frac{Pr}{M}\right)^1$$

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840

In general the probability that i out of the R-1 other processors are requesting access to the particular M2 units is

$$P_i = \binom{R-1}{i} \left[1 - \frac{Pr}{M}\right]^{R-1-i} \left[\frac{Pr}{M}\right]^{i}$$

If there is no contention the M2 access time is t_2. If one other processor requires access then the worst case access time is $2t_2$.

In general the worst case access time encountered when there are i other processors in contention for a given M2 unit is $(1 + i)t_2$.

The effective worst case access time averaged over all contention possibilities is therefore

$$t_{2eff} = \sum_{i=0}^{R-1} (1 + i)t_2 P_i = t_2 \sum_{i=0}^{R-1} P_i + t_2 \sum_{i=0}^{R-1} i\, P_i$$

Since P_i is a binomial distribution

$$\sum_{i=0}^{R-1} P_i = 1 \qquad \text{and} \qquad \sum_{i=0}^{R-1} i\, P_i = (R-1)\frac{Pr}{M}$$

Therefore

$$t_{2eff} = t_2 \left[1 + \frac{(R-1)Pr}{M}\right]$$

For the nominal system assuming Pr = 1, R = 4, M = 16, $t_{2eff} = (1.188)t_2$. That is M2 seems about 18.8% slower due to memory contention.

For the higher order language oriented instruction set utilizing M1 for temporary storage a value of Pr = 3/4 is reasonable. Figure 5.1-36 shows $t_2$eff for various values of M, R and Pr.

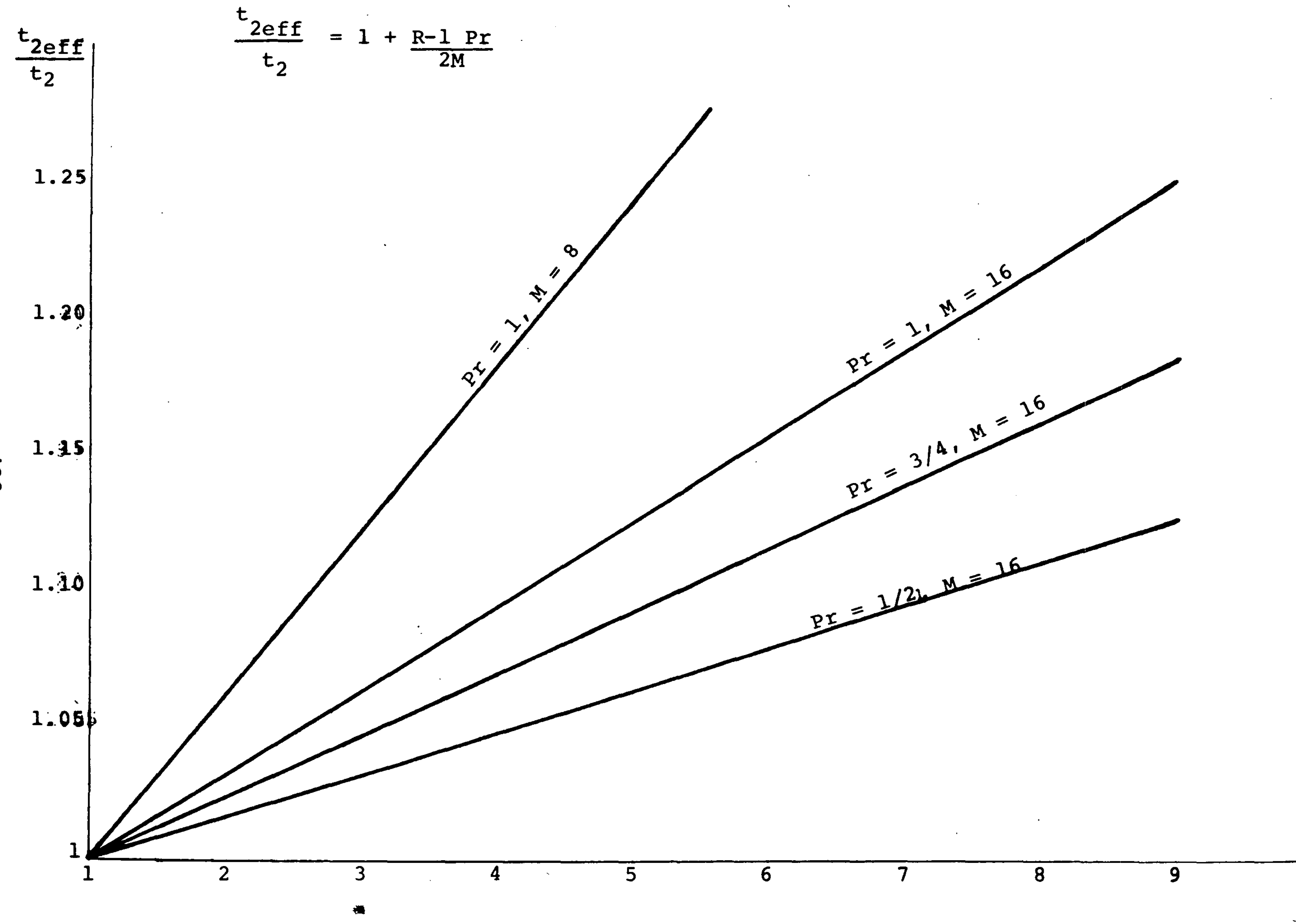


Figure 5.1-36: t_{2eff} as a Function of R, Pr, and M

5.1.3 I/O Unit Implementation

Figure 5.1-37 depicts a block diagram of the I/O unit which consists of the eight basic elements described below.

5.1.3.1 Central Control (CC): The central control unit provides the decoding of the I/O operations, for the initiation and synchronization of commands, and for data transfers between the units. The CC contains an arithmetic unit and the logic required to perform conditional decisions. The sequences issued by CC are stored in a micro control memory and are initiated via commands from the various interfaces.

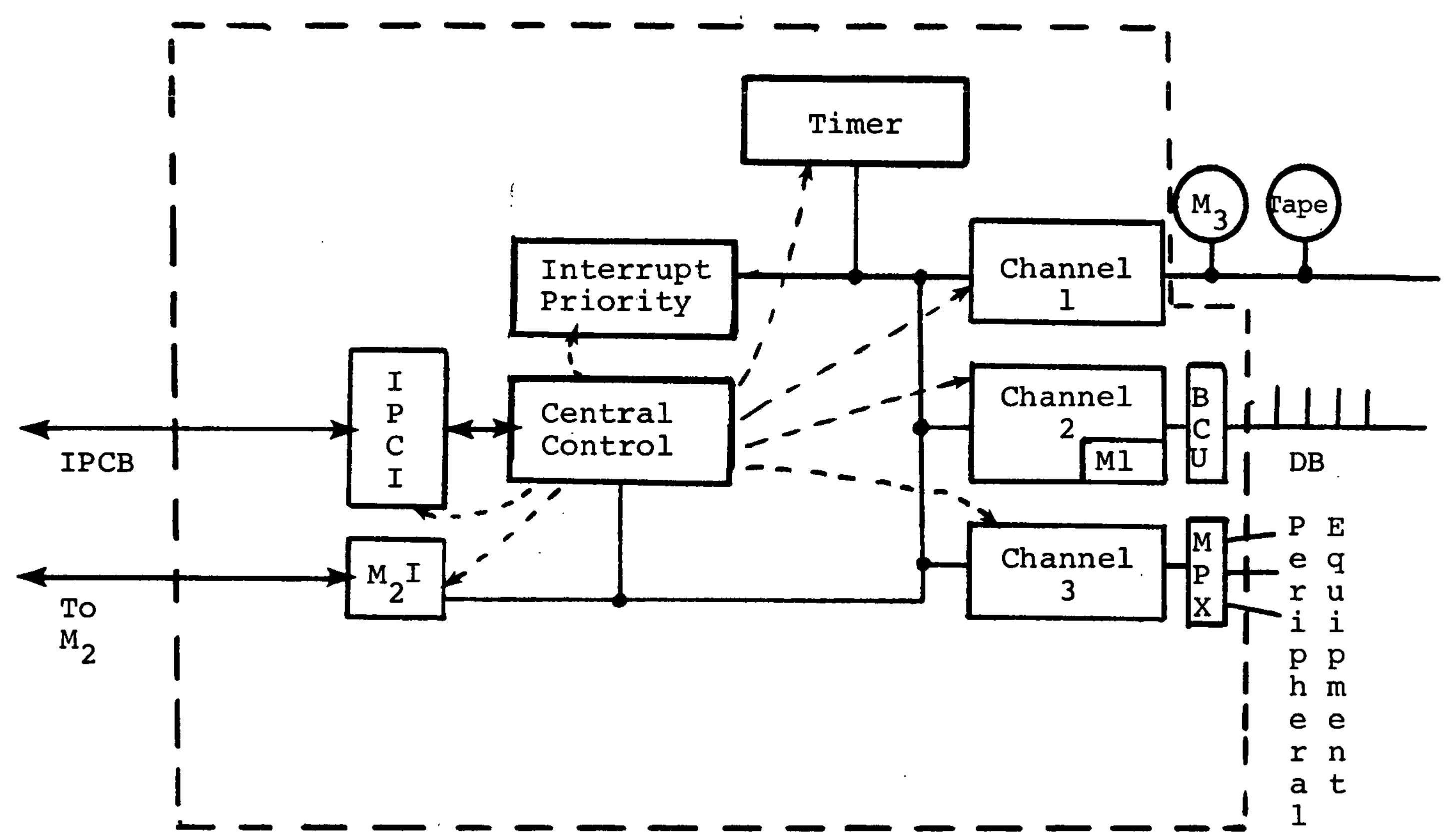
5.1.3.2 InterProcessor Communications Interface (IPCI): The IPCI corresponds to the similar element in the processor, but contains additional logic for the control of the IPCB. The IPCI receives and sends commands to the processors. These commands will be discussed in section 5.1.4 dealing with the IPCB. A four byte buffer is contained within IPCI in order to provide synchronization.

5.1.3.3 Interrupt Priority: This element is used to store the priorities of the processes currently being executed on each of the processors. The inhibit or allow state of interrupts for each processor is also contained in this element.

When an interrupt signal is to be sent, the most interruptible processor is determined on the basis of information contained within the interrupt priority storage. The sequence of execution for this determination is under CC control.

5.1.3.4 Timer: The timer consists of two counters. A 51 bit counter which contains absolute time and a 22 bit counter which is used to generate an interrupt at a specific time. Each counter may be written into or read from by a command over the IPCB. The operation of this timer has been explained in Chapter 3.

5.1.3.5 Channel 1: Channel 1 of the I/O unit is used to communicate with devices which must operate in a burst mode. Discs and tapes fit into this category of devices. Once a data transfer is initiated on this channel, it cannot be interrupted or multiplexed due to the high bit transfer rate. M3, the mass memory, and file or archival storage are shown attached to this channel.



IPCI - Inter Processor Communication Interface
IPCB - Inter Processor Communication Bus
M_2I - M_2 Interface
BCU - Bus Control Unit
DB - Data Bus for Avionics Subsystem
MPX - Multiplexor
——— - Data Path
---- - Control Path

Figure 5.1-37: I/O Unit Block Diagram

5.1.3.6 Channel 2: Channel 2 is used to control the Data Bus Control Unit which in turn drives the devices of the avionics subsystem. Contained in local storage within channel 2 is a table of commands which drive the Data Bus in a periodic fashion. This local storage may be addressed and written into in order to change the Data Bus sequences, or it may be read from to determine the state of the sequencing.

Once channel 2 has been initiated by an I/O command fetched from M2 by the CC, it will proceed to operate in an asynchronous fashion, transfering information between M2 and the various devices of the avionics subsystem.

5.1.3.7 Channel 3: Channel 3 is used for the slow peripheral devices such as card readers, printers, keyboards, control panel, slow speed short record cassetts tapes, and any other devices which may be required for operation and testing of the system.

All the devices attached to channel 3 can be multiplexed on a four byte basis. This imposes a requirement for a certain degree of buffering at each device attached to the channel 3 multiplexer. In addition, the channel itself must possess a limited degree of local storage to buffer device commands, data, and information to aid in forming the M2 address.

5.1.3.8 M2 Interface, M_2I. The M_2I can transfer information between M2 and the other elements of the I/O controller. The priority as to which I/O interface has access when contention exists is fixed.

Priority 1 (highest) Channel 1: The devices which operate in the burst mode must be serviced at a rate consistent with their data rate. M3 can possess a data rate of up to 10 MBPS, which is three to six times less than the M2 data rate. However, channel 1 devices cannot sustain a large delay between a request for an M2 transfer and the final servicing of the request since the addressed record is usually not fully buffered and M2 and the auxilliary device must be synchronized during a data transfer.

Priority 2 Channel 2: The devices which are driven by tables in the local memory of channel 2 present to M2 a data rate three to six times less than that of channel 1. Yet if too much delay is introduced in each M2 transfer the minor and major cycle times might be exceeded.

Priority 3 Central Control: When the CC receives a command over the IPCB it often has to fetch an I/O control word from M2. While this fetch can be delayed a reasonable amount of time, queueing of too many IPC commands before execution must be avoided.

Priority 4 Channel 3: The devices attached to channel 3 are all slow speed and involve only a few bytes per transaction. A delay of ten to even one hundred M2 cycles will not appreciably affect the performance of these devices.

Priority 5 (lowest): Since the interrupt priority and timer elements of the I/O unit do not use M2 to a significant extent, these elements are placed into the lowest priority category.

5.1.3.9 <u>I/O Unit Operational Functions</u>: Throughout the discussion in the various chapters a number of requirements have been placed upon the I/O unit. This section will summarize these requirements by presenting a deliniation of the various commands which the I/O unit must execute. These commands are categorized according to which of the elements of the I/O unit is involved.

a) Interrupt Control Commands for Each Processor:

    Set Priority
    Allow Interrupts
    Inhibit Interrupts

b) General Interrupt Functions:

    Determine the most interruptible Processor
    Generate the interrupt command word and transmit over the IPCB
    Format the Interrupt Descriptor

c) Channel 1 Commands and Functions:

    Initiate data transfer to device
    Initiate data transfer from device
    Stop or Reset Channel
    Test channel (channel status transfer)
    Stop or Reset Device
    Test device (device status transfer)
    Commands associated with specialized devices such as:

        Read sector clock for drum
        Rewind tape
        Skip next tape file
        Skip next record
        etc...

M2 address generation
Interrupt generation

d) Channel 2 Commands and Functions:

Write M1 }
Read M1 } Each table entity is to be addressable
Execute command table continuosuly
Execute command table for one major cycle
Stop execution at end of major cycle
Stop execution immediately
Test channel (status transfer)
Test devices on data bus (status transfer)
M2 address generation
Interrupt generation

e) Channel 3 Commands and Functions:

Read device
Write device
Test channel
Test device
Stop or reset device
Stop or reset channel
M2 address generation
Interrupt generation

f) IPCI Functions:

Decode I/O address
Control the transfer of information across IPCB

g) CC Functions:

Decode IPC command and initiate the appropriate micro-sequence
Read I/O descriptor from M2
Control all the attached channels and functional elements

h) Timer Commands and Functions:

Set absolute time register
Read absolute time register
Set time interrupt register
Read time interrupt register
Inhibit interrupt from time interrupt register
Allow interrupt from time interrupt register
Generate timer interrupt

5.1.4 <u>InterProcessor Communications Bus (IPCB)</u>: The IPCB does not have any severe time constraints placed upon it and for this reason an eight bit wide time multiplexed communicating link interfacing all the communicating elements (P's and I/O) is proposed. If the IPCB is not busy then a communicating element shall be guaranteed access to the bus within 1 μsec. When access is granted, each command or response byte shall be transmitted within 1/4 μsec period.

Once the initial addressing byte has been placed upon the IPCB communication is established and the IPCB becomes dedicated to transferring the indicated number of bytes at 1/4 μsec per byte. Thus an eight byte transmission would require 2 μsecs for completion. The time period in which the receiving element decodes its address is sufficient to determine the identity of the sending element.

A wide variety of communications take place over the IPCB. These are:

a) Cancel SNO: This command sends a stack number and offset (SNO) address to all other processors. The receiving processor looks into its CAM to see if the given SNO address is present. If it is, the CAM entry is made vacant and the processor returns an acknowledge signal. The cancel SNO command requires the initial IPCB command byte to be followed by three more bytes specifying the given SNO address. This command is executed by the receiving processor(s) and normally requires a 500 ns pause during MP instruction execution.

b) Dump M1: This command causes the receiving processor to dump the state of M1 into a given M2 area. Whether the prime or backup M1 is to be dumped is indicated in the command.

The command requires a three byte transmission after the initial address byte in order to specify the M2 address for the M1 dump. Part of the second byte is used to indicate the status of the error detection logic during the dump and also which of the duplicate M1's is involved.

c) Load M1: This command causes the given area of M2 to be loaded into the selected M1. This load M1 command is similar to unload M1. Part of the second byte will indicate whether both M1's or which M1 of the processing unit pair is to be loaded.

d) I/O Commands: An I/O command is sent from a P unit to an I/O unit. After the addressing byte, three bytes are transferred to the I/O controller. These bytes address an I/O descriptor. The I/O unit fetches this descriptor from M2, interprets its command bits and executes the appropriate I/O command.

e) Processor interrupt: The I/O unit determines which process can be interrupted. The interrupt is then directed from the I/O unit to the appropriate P unit. The initial addressing byte is followed by an interrupt descriptor which is placed on top of the interrupted process's stack by the receiving processor.

Interrupts may also be directed to a particular P from another P. However, all inter-P interrupts must be sent through the I/O unit in order that the inhibit interrupt condition may be tested and the interrupt appropriately queued when necessary.

f) Inhibit/Allow Interrupts: When a processor inhibits interrupts, an indication is sent to the I/O unit as well as being retained within the processor's own status word. Only one byte follows the initial address byte. This second byte is used to indicate whether the interrupts are to be allowed or inhibited.

g) Process Priority (Interruptibility Index): Since the process priority is used in determining processor interruptibility, the I/O unit must receive this command to indicate the priority of an executing process when it is initially assigned to a processor. Only one byte is required to indicate the priority based upon priority values of 1 to 16.

h) Response: Important long transmissions required a response, so that the sending element can be informed about the successful receipt of the message. This is a necessary feature for fault tolerance.

i) Processor and Memory Failure: These commands are generated by a P and directed to the I/O unit. They indicate a failure, and provide the I/O unit with the error bits associated with the failure. This command requires two bytes following the address byte.

j) Instruction Retry: When this command is issued to a processor, the phase of the instruction which was being executed at the time of failure is reexecuted.

k) Timer Control: The timer element of the I/O unit actually contains two distinct counters. One is a 51 bit counter which is incremented at a once per 1 μsec rate. This counter provides an indication of absolute time for the system, and is also a source of unique values for (time) tagging (see section 3.6). The other is a register which is compared with the middle 21 bits of the 51 bit counter. When an equality occurs, an interrupt is generated. Both counters in the timer may be addressed separately for reading and for loading of new or updated values.

5.1.5 Mass Memory (M3) Implementation

This presentation discusses M3 from several points of view: requirements, technology and design issues.

5.1.5.1 Functional Requirements: The mass memory services two major functions for the space station.

a) Memory Multiplexing. As the repository for most program and some data segments, the M3 functions as an extension of the operating memory, M2. A segment is the basic unit of memory multiplexing and consists of a logical program unit or data unit which may vary in size from a few words to many thousands of words. The M3 address within a descriptor specifies the starting address of the relevant segment. Issues concerning the physical allocation of space to segments is discussed in section 5.1.5.3.

b) File Storage. The mass memory also serves the function of temporary storage for active files. A file can be addressed by name through an I/O command. However, its program or data cannot be used until it is broken down into segments. Files should reside on M3 while being used, but after they are updated and become inactive, the files may be moved to archival longtime storage.

Files may be dynamically created and destroyed. For example, a file may be created on M3 to serve as the repository for information to be sent to a printer. After the printer (via the I/O) empties the contents of the file, the file would be destroyed and the physical M3 space it occupied can be recovered for other uses.

Another example of file creation is the recording of experimental data for future processing. As data is accumulated it is stored in a file on M3. When the experiment is over, the file may be moved to tape and at some future time the experimental data may be processed.

5.1.5.2 Technological Aspects: Mass memories, with a capacity in excess of 10^6 words, have been constructed from discs, drums, tapes and even large core storage. Proposals have been made to construct mass memories utilizing plated wire technology, MOS integrated circuit technology and the new magnetic bubble technology.

The characteristics of these technologies differ significantly in terms of latency time and data transfer rates. A plated wire or MOS M3 can be categorized by little, if any, latency of the order of hundreds of microseconds, and a data transfer rate of .5 to 2 MBPS. Drums and fixed head discs often possess latency times of five to tens of milliseconds and a higher transfer rate of 2 - 10 MBPS. Tapes generally have even a longer latency time which is dependent upon the location of the desired data on the tape.

The choice of which M3 technology to employ has large interaction with the memory management design. One of the major design decisions made at the beginning of the contract was to assume that M3 was to be implemented with a head per track drum or disk like device. The discussion of memory management was oriented in this direction. The reasons for this design decision were:

a) The laboratory model of the space station multiprocessor, will most assuredly utilized a disc or drum for M3. Within the 1972 to 1975 time period solid state mass memories present both a technological risk factor as well as a large cost factor.

b) It seems reasonable that at least early space stations, circa 1980, will require technology which has been proved by 1975. A conservative point of view would indicate that one should not rely upon a plated wire or a solid state mass memory technological breakthrough within the next three years.

In order to place some constraints on M3, the following characteristics are proposed:

a) Head/track disc or drum with electronic switching of tracks. There will not be any mechanical parts except for the drive motor.

b) A storage capacity in excess of 10^6 words. The M3 address in the descriptor contains 20 bits which can address 2^{20} or 10^6 different entities. When M3 is used in memory management the addressable entity is a segment. If one assumes an average of 2^6 = 32 words per segment then the address field of the descriptor can possibly access 32×10^6 words on disc.

c) The disc latency time is of the order of 5 - 10 milliseconds. The bit transfer rate is of the order of 5 - 10 MBPS.

d) Tracks can be electronically switched in 10 - 20 μsecs.

5.1.5.3 Design Issues: This section will discuss some of the design problems which must be resolved during the detailed design and implementation phases of the multiprocessor project.

a) Physical M3 Storage Allocation. A question arises as to whether fixed size blocks or variable sized records should be used on the disc. There are reasons for and against the choice of either approaches:

   1) Since the unit of memory management is a variable sized segment it would seem that variable sized blocks should be stored on the M3 device. Variable sized records are used by IBM on many of their disc file devices. Records are separated by record gaps, as on a tape, and record identification are contained within the record. The record gaps and control information stored with each record tend to reduce the utilization efficiency of the storage medium.

   2) With regard to M3 memory management, variable sized records create more software problems than do fixed sized blocks. M3 management includes the allocation and deletion and compaction of M3 space. Variable sized records can create an external fragmentation problem on M3 similar to that which occurs on M2.

   3) If fixed sized blocks are employed then the mapping of variable sized segments into fixed sized blocks becomes a problem. If a segment is smaller

than a block, internal fragmentation becomes an issue. If a segment is larger than a block then one must consider issues such as whether to link many blocks together to form a segment, and whether they should be contiguous or be placed randomly.

b) M3 Request Optimization. When M3 is a disk or drum and a queue exists for M3 requests there is an optimum ordering in which the requests can be serviced to minimize the average access time. If a first come first served (FCFS) algorithm is used for disc access then the average access time increases as the disc queue length increases. On the other hand, if a shortest search time first (SSTF) algorithm is employed then most of the disc requests can be serviced in a single M3 rotation. This can significantly decrease the average disc access time when the disc queue contains a larger number of entries.

For devices with very high latency times such as moveable arm discs the SSTF algorithm could possibly prevent certain M3 accesses from being executed because new accesses arise continuously with shorter search time. An M3 constructed of plated wire technology would possess a random access addressing scheme with each access requiring the same time. In this case all choices would be equally optimal and hence an FCFS algorithm would be most satisfactory. It is seen that as latency time decreases, the necessity for disc optimization also decreases.

5.2 System Performance

This section will examine the impact of a number of design areas upon the multiprocessor's internal performance. The design areas to be considered are:

a) Processor Design. This includes the overall effect of M1 and the higher order language oriented instruction set.

b) Fault Tolerance. The restartable instruction design and redundant writes with echo checks tend to reduce performance as compared to a non-fault tolerant design.

c) Internal Bus. The necessity for a generalized switching arrangement between processing units and M2 units introduces a delay to each memory cycle which reduces performance.

d) Memory contention. The effect of memory contention yields an apparent increased M2 cycle time.

The following definitions will be used in the analysis. For the purpose of this analysis an I/O unit is considered a processing unit.

n_1 = number of processing unit cycles per unit time for a single processing unit

n_2 = number of M2 cycles per unit time associated with a single processing unit

t_1 = processing unit cycle time

t_2 = M2 unit cycle time

t_{2eff} = effective M2 cycle time as seen by a processing unit. This is measured from the initial M2 request to the completion of the command.

U = useful work per unit time for a processing unit. This is defined as the total number of processing unit cycles plus the number of M2 cycles initiated by the processing unit. Usually processor work is defined in terms of the number of instructions per second. For a conventional 360 type architecture an instruction usually corresponds to two M2 cycles.

In a sense the internal processor cycles should also be considered useful work; e.g., indexing which does not require an M2 access because it might use an internal register is a very useful function. Our multiprocessor makes very large use of its internal M1 storage and these cycles are just as "Useful" as M2 cycles.

R = number of processing units

M = number of M2 memory units

Pr = probability of a processor making an M2 request

From these definitions, it follows that:

$$U = n_1 + n_2$$

$$n_1 t_1 + n_2 t_{2eff} = 1 \text{ (one unit of time)}$$

$$Pr = \frac{n_2 t_{2eff}}{n_1 t_1 + n_2 t_{2eff}} = n_2 t_{2eff}$$

5.2.1 Performance and the Processor Design

The above definitions yield the following formulation:

$$U = \frac{1 - Pr}{t_1} + \frac{Pr}{t_{2eff}}$$

The effect of the higher order language oriented instruction set tends to keep the processor busier than a conventional instruction set. This is reflected in the parameter Pr. Semantic conciseness decreases the memory size needed for instruction storage which implies that fewer M2 accesses are requried as compared to processing unit cycles and therefore Pr tends to decrease. If the effect of Pr on t_{2eff} is ignored then Figure 5.2-1 shows U versus Pr for various values of t_1 and t_{2eff}. t_2 and t_{2eff} are assumed to be equivalent for purposes of this graph.

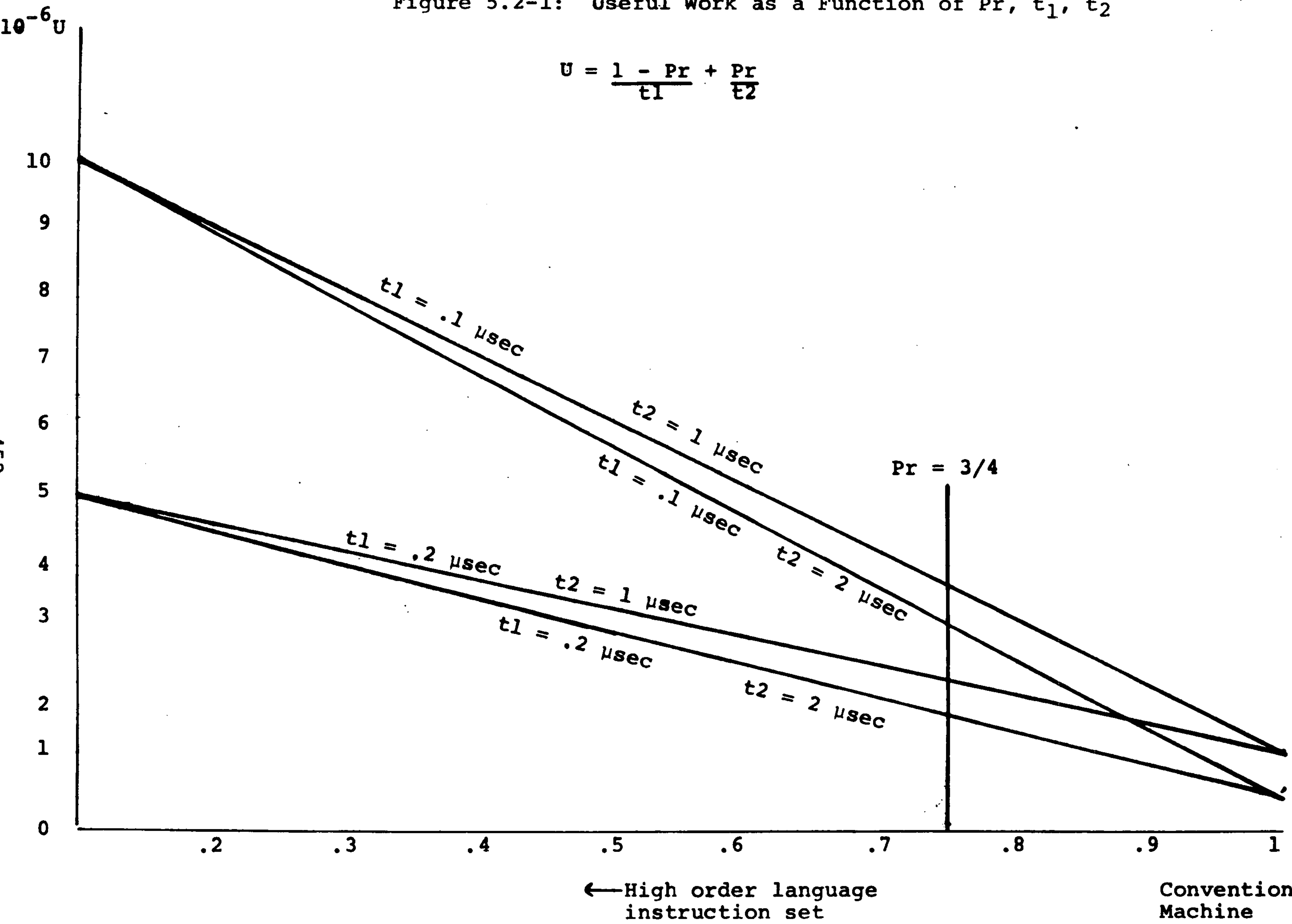
In general the smaller the value of Pr the greater the performance.

5.2.2 Effect of Fault Tolerance

The implementation of restartable instructions tends to increase the number of processor cycles, n_1. We can discount these extra M1 cycles used to buffer M1 and M2 writes and to perform the copy cycle by defining an effective t_1, t_{1eff} as follows.

Every time a M1 write operation is desired it is first written into the M1 temporary buffer area. It is subsequently copied into its final location. the entire temporary storage and copy including data and address cycles consists of five M1 cycles instead of one. However, the majority of M1 cycles are reads instead of writes. The exact ratio is difficult to determine without a representative instruction mix.

Figure 5.2-1: Useful Work as a Function of Pr, t_1, t_2



The following issues are involved in determining the ratio of M1 read to write operations.

a) The M1 part of the stack has an equal number of read and write operations. One can appreciate this by realizing that whatever is pushed (written) into the stack must ultimately be popped (read) from the stack.

b) The base register used for lexical level offset addressing tends to produce more reads than writes. Base registers are set up (written into) when a process is put into the running state or returned from an interrupt. Once set up they tend to be read only information.

c) The Descriptor Cache also tends to produce more read than write operations. A write operation occurs whenever a descriptor not contained in the cache is accessed. The most recently used 32 descriptors contained in the cache are read from M1 when SNO translation to absolute M2 address is used. Descriptors are modified (written into) whenever the status of the described segment changes.

d) Status register images tend to be write-only information. They are read whenever a process is initially set up or upon return from a fault condition.

The effective M1 cycle time can be expressed as

$$t_{1eff} = (1 - pw)t_1 + pw(5t_1) = (1 + 4\ pw)t_1$$

where pw = Percent of M1 Write operations.

If pw = 25% then the effective t_1 cycle time is doubled. If pw = 50%, that is an equal number of read and write operations, then $t_{1eff} = 3t_1$.

This effect is not quite as severe as it initially seems, because all processing unit cycles do not require M1 accesses. A processing unit cycle might just involve a register transfer.

The second effect of fault tolerance is the echo check of all write operations into M2. This makes a single word write require 1.9 μsec instead of 1 μsec. A read still requires 1 μsec. If M2 writes only constitute 10%

of all M2 accesses then the effective M2 cycle time can be calculated as:

$$t_{2eff} = .9(1) + .1(1.9)\ \mu sec = 1.09\ \mu sec$$

This indicates that M2 seems 9% slower due to fault tolerant actions.

5.2.3 Effect of the Internal Bus

The introduction of the internal bus adds delay between the P and M2 units. A M2 unit has an 800 ns cycle time. Addressing and data transfer add an additional 200 ns to each M2 cycle. A single P and single M2 unit configuration without any M1 would have a useful work factor of 1.25×10^6 units. The internal bus reduces this to 10^6 units. However, this is one of the costs one must endure for the flexibility of the multiprocessor configuration.

5.2.4 The Effect of Memory Contention

Memory contention manifests itself as an effectively slower M2 cycle time. This was derived previously as

$$t_{2eff} = t_2 \left[1 + \frac{(R-1)Pr}{M}\right]$$

Substitution of this value into the equation for U we find

$$U = \frac{1 - Pr}{t_1} + \frac{Pr}{t_2 \left[1 + \frac{(R-1)Pr}{M}\right]}$$

Figure 5.2-2 shows U as a function of R. We ignore the fact that as R increases and t_{2eff} increases, Pr will tend to get closer to 1. Therefore if a single P system has a Pr of 3/4, for a multiple P system Pr will approach one.
Therefore the upper and lower curves are to be considered to be assymptotes for small and larger R.

5.3 Laboratory Model

One of the requirements of this contract is to define a laboratory system. In order to arrive at a specific recommendation in this area, the basic purpose of the laboratory system was established.

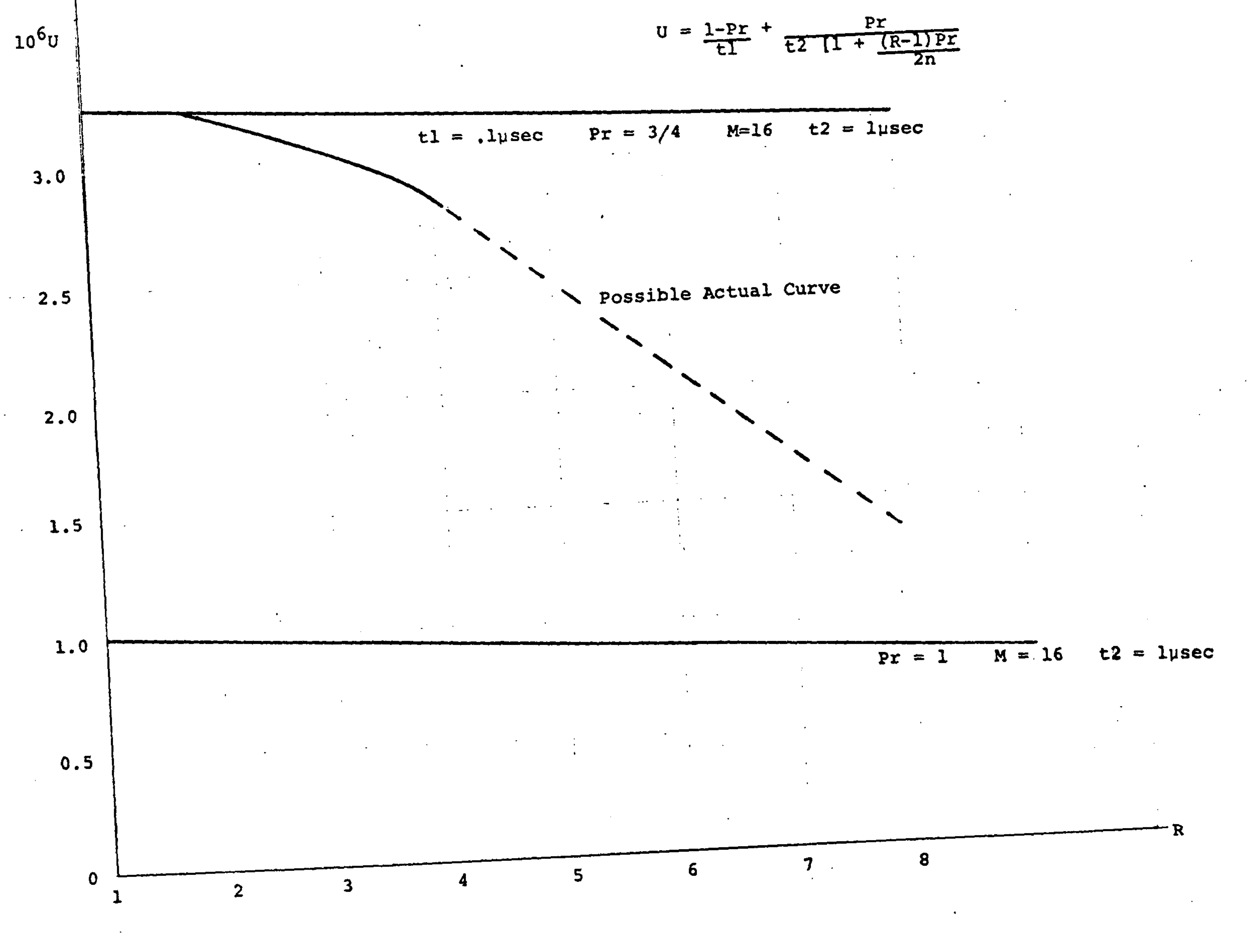


Figure 5.2-2: U as a Function of R

The main function of the laboratory system is to provide a learning tool. The system will be used to verify and modify the design concepts presented in the body of this report. For this reason flexibility of the proposed structure from both a hardware and software point of view was emphasized. This report presents concepts, most of which are to be implemented by software and firmware. For this reason the laboratory model should try to minimize the degree of hardware development.

It is clear that no existing system possesses all the characteristics of the proposed multiprocessor design. Many of the features will have to be simulated by firmware, and software in order to provide a realistic model.

The ultimate purpose of the multiprocessor is to support the operational, experimental and data management functions of a future space station. Design is an iterative process: The laboratory model will provide the means for observing bottlenecks in the design, and generating new design proposals for further testing. The main functions of the laboratory model are thus: design, test, and modify.

5.3.1 Features of the Laboratory Model

There are a number of desirable features which can be incorporated into the laboratory model which will aid in verifying the design concepts:

a) The processing unit should be microprogrammable. This will provide for implementation of the proposed instruction set along with the flexibility for change. Microprogramming will also allow the full testing of restartable instructions.

b) Floating point arithmetic must be employed. Microprogramming provides for its implementation.

c) The word length within the processing unit should be 64 bits. However, a 16 or 32 bit processor could be utilized by trading off time for word width.

d) The M2 memory modules should possess an information width of 32 bits. The internal bus should provide a mechanism for accumulating 32 bit words into 64 bit double words.

e) The local M1 memory should be 96 words by 64 bits, with an associated 32 x 22 bit CAM.

f) The structure of the system should be flexible enough to be efficient in implementing the needed descriptor mechanisms.

g) It would be desirable to run the system in real time. However, without the specific proposed equipment, it is realized that a simulation must degrade performance. In order to determine the system bottlenecks generated by the dynamic execution of a complex software, the system should run in a "proportional real time". The relation of execution time of the various instructions should be in the proper ratio even if artificial delays are required.

h) Both the hardware and software system should employ performance measuring mechanisms so as to make the job of evaluating different system features efficient.

i) Two specific types of I/O equipment should be able to be employed. The first category includes those facilities required to operate the laboratory model and record performance data. This includes tape equipment, possible card equipment, terminals, consoles, etc. The second category includes those equipments required in the space station environment: a data bus with its associated avionics equipment or simulators attached. Much of the equipment in both categories is similar, such as a mass memory.

Flexibility in the I/O structure of the proposed laboratory model is of major importance. The separate I/O unit of the design tends to isolate the I/O from the processing unit and therefore maximizes flexibility of the I/O interface.

5.3.2 Uses of the Laboratory Model

The laboratory model will provide a vehicle to determine the appropriateness of various algorithms suggested in this report. Many questions concerning the exact performance of the proposed mechanizations are unresolvable without an actual implementation.

Some of the measurements to be made and questions to be resolved by the lab model include:

5.3.2.1 Operating Memory (M2):

a) How does M2 access conflict vary with the number of M2 modules?

b) How does system performance change as the degree of interleaving within the M2 units is varied?

c) How is system performance affected by a change in M2 cycle time?

d) What is the proper relative size between M2 and M3? How small can M2 be made before thrashing occurs?

e) How much overhead is consumed by memory management?

5.3.2.2 Processing Unit (P and M1):

a) Measure the instruction frequency.

b) Determine the processor utilization factor (1 - Pr) by measuring the relative number of M1 and M2 cycles.

c) How much work does each processor perform? This measurement should be made for a varying number of P and M2 elements.

d) How is system performance affected as the size of the CAM varied? What is the minimum sized CAM that can be utilized without a significant performance degradation?

5.3.2.3 Programming:

a) Demonstrate the ease of using a HOLM for applications versus just a HOL.

b) Determine the effectiveness of the instruction set for implementing compilers and the operating system.

c) How efficient is the compiled code in terms of M2 space utilization?

5.3.2.4 Fault Tolerance:

a) Verify the restartable instruction design.

b) Verify the logical consistency of the recovery techniques.

c) Measure the relative recovery times.

5.3.2.5 Communications:

a) Measure the amount of traffic on the IPCB. Both average and peak bit rate, as well as the number of messages per second.

b) Measure the average bit rate on the internal bus.

c) How much traffic does the I/O contribute to the internal bus.

5.3.2.6 I/O:

a) How is system performance affected by placing the I/O command tables for each channel in local M1 memory associated with the I/O controller as opposed to M2?

b) What is the effect of the "Quiet Interrupt" concept? Is system performance increased?

c) How sensitive is system performance to a change in the interruptibility algorithm? Could a random processor be chosen to service system level interrupts? Is an algorithm based upon more than the process' static priority required?

d) What is the duty cycle of the I/O controller? Is it saturated or is it idle most of the time?

5.3.3 Off the Shelf Equipment

The major elements of the multiprocessor are the processing unit (P), local memory (M1), operating memory (M2), input/output unit (I/O) and mass memory (M3). The internal bus and I/O bus interconnect these elements and create the structure for the computer architecture.

Our study of off the shelf hardware started with an investigation of different manufacturers of hardware elements. Three major elements which required special attention were P, I/O and the internal bus. The memory elements could be implemented by a large number of suppliers.

A number of promising machines were investigated for possible use as the processing unit. Any machine which could not be microprogrammed was eliminated because of the lack of flexibility. Processing elements that were investigated included: INTERDATA 80, QM-1, META-4, and the Burroughs D-Machine.

A brief description of the characteristics of these machines is presented below.

a) INTERDATA Model 80 (Interdata Inc.)

The Interdata Model 80 is a high speed microprogrammed control processor which operates from a 60 ns control memory. The processing word width is essentially 16 bits. The model 80 possesses 16 general purpose registers which can be used as index registers or accumulators. Memory is available in 16K byte MOS modules with a cycle time of .25 μsec.

The 16 registers and its standard instruction set makes it look very similar to a 360 type architecture. However, a writeable control store is available to generate a specialized instruction set.

The memory bank controller allows two ports into memory. One port is dedicated to the processor while the other provides high speed direct memory access (DMA).

The Interdata 80 possesses many shortcomings as far as the laboratory model is concerned.

1) 16 bit words will create a slow system when 32 or 64 bit accesses are required. However, the high speed (250 ns) MOS memory tends to negate this criticism. This short length is typical of many other minicomputers that one might consider for use as the processing unit of the laboratory model.

2) The 16 general registers are not sufficient to implement an M1 of the size needed, and therefore main memory must be used.

3) The memory interface is not designed for multiprocessor applications when more than two processing units are required.

b) Meta-4 (Digital Scientific Corporation)

The Meta-4 processing unit consists of data registers, data processing logic, microprogrammed control utilizing a changeable ROM, and an integrated circuit scratchpad memory. Data Registers are 16 bits wide and up to 32 may be employed. In addition up to 256 words of 16 bits each of high speed scratchpad is available with an 80 ns cycle time. This would serve well to simulate the M1.

Memory modules may possess up to four ports thus allowing a three processor/one I/O unit configuration to be employed. If four memory units were contained within each module, the simultaneous access through these four ports would require four memory controllers.

As compared to an Interdata 80, the Meta-4 is superior in the ability to create a multiprocessor with local M1 storage. Its major drawbacks are:

1) The 16 bit word would create a slow emulation of the proposed multiprocessor.

2) The control store is a ROM and is therefore not writeable. However the user may substitute different ROM boards to create instruction changes. The creation of the ROM board is a simple matter of peeling off metalic strips from a program board. This can be done in any laboratory.

3) Expansion beyond four ports into memory would be very difficult.

c) QM-1 (Nano Data Corporation)

The QM-1 offers an exceptional degree of flexibility in a processor unit. Control is effected by double level emulation with a micro-control store driving a nano-control store. The micro memory is a writeable control store. The data width is 18 bits. One of the major features of the machine is the variety within the memory hierarchy. This includes main memory up to 512K bytes of 750 ns core, a local store of thirty-two 18 bit registers, external register consisting of thirty-two 13 bit registers, control store of up to 32K 18 bit words, and a nano store up to 1K 360 bit wide. This hierarchy of storage with the extremely wide nano memory, and potentially large degree of processing parallelism would certainly prove quite satisfactory for implementing the proposed instruction set.

Unfortunately there are two important shortcomings to the machine:

1) The word length is fixed to 18 bits.

2) The structure of the memory interface is not amiable to a multiprocessor configuration.

d) D-Machine (Burroughs Corporation)

The Burroughs D-machine is an unusually modular and flexible architectural design, which is capable of application to a wide variety of problem areas. In its basic multiprocessor configuration, it consists of three major building blocks: interpreters, switch interlock, and memories. The interpreter is a microprogrammed processor and is used to perform both arithmetic/logical computation and I/O device control. The switch interlock is the communication network which links interpreters, operating memory, and I/O devices.

The D-machine interpreter is constructed from five functional parts: memory control unit (MCU), control unit (CU), logic unit (LU), microprogram memory (MPM), and nanomemory (NM). (See Figure 5.3-1) The word length of the interpreter depends only upon the logic unit, which is modular in 8-bit blocks, from 16 bits to 64 bits. The use of microprogramming enables the control logic to be quite regular in structure, resulting in economy of manufacturing. Additionally, different microprograms may be used with the same hardware to implement different instruction sets for different applications. Furthermore, if a read-write rather than read-only MPM is attached, the system can reload its MPM dynamically to run programs written in different machine languages at different times.

To save storage, the microprogram structure of the interpreter has been divided into two logical sections: micro and nano. The control of functional operations within the interpreter is dictated by the contents of a location in nanomemory. Each of the 56 bits corresponds to a control line for the elements of the LU, CU, and MCU. A given nanoword is selected under control of a microword which specifies the nanoword's address in nanomemory. As a result, nanowords may be referred to by many microwords; hence, the bit saving.

The switch interlock (SWI) is a communication network which connects the interpreters, operating memories, and devices, The basic configuration is designed for four interpreters, eight memories, and eight devices. Hardware expansion beyond this is possible since the switch interlock is designed to be expandable. Because of the lesser logical complexity of the operations in the SWI per clock cycle compared with the interpreter, the SWI may be run at a higher rate than the interpreter. This permits introduction of a degree of

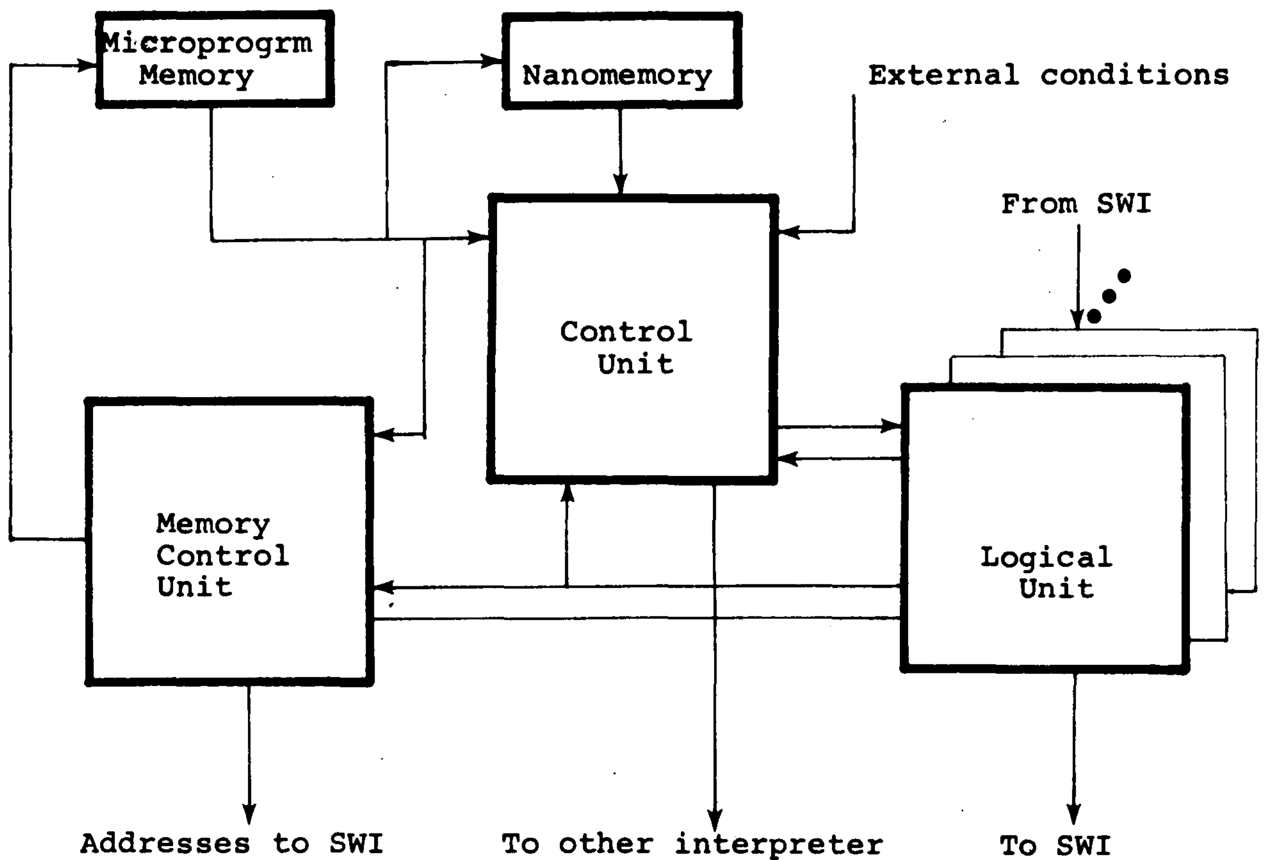


Figure 5.3-1: Interpreter Block Diagram

serialization which reduces the number of required parallel paths, diminishing complexity. Consequently, the building block of the SWI is a single bit wide. These may be combined to the degree necessary for high throughput.

The SWI can interconnect interpreters to one of two types of ports, either a memory port or a device port. Memory ports are characterized by the fact that an interpreter cannot be locked to a memory module for more than one memory cycle. Device ports allow a device to be locked to an interpreter for as long as desired. Devices can only be locked to one interpreter at a time while interpreters may be locked to many devices.

Burroughs is producing both a commercial and a military version of the interpreter-based system. The commercial version is being used for disk controllers and for other applications not yet announced. The military configuration is funded by the Avionics Lab at Wright-Patterson AFB and consists of a five-interpreter system.

Microprograms have been written which enable the machine to run D825 programs and B300 programs. A demonstration of these and of dynamic switching between them is in current operation at Burroughs', Paoli, Pennsylvania location. A full operating system for multiprocessor configurations is expected to be operational soon.

The major shortcoming of the D-machine is the fact that there is little local storage associated with an interpreter. However, Burroughs has assured that an Ml unit can be attached to a device port, which will serve the function.

e) Other Minicomputers

The majority of minicomputers are not satisfactory as the processing unit of the multiprocessor for one or more of the following reasons:

a) They are not microprogrammable with a writeable control store. This is a required feature in order to implement the HOL-oriented instruction set in a laboratory environment where the means for modification is necessary for development.

b) The word length is too short being 8, 16, or 24 bits in length, and therefore requiring a multiple memory access for 32 or 64 bit fetches. This would tend to slow the system performance.

c) The memories are not structured to be able to accept a variable number of inputs from the different P and I/O units. That is, they were not designed to be elements of a multiprocessor.

Realizing that the purpose of the laboratory model is to test concepts, in the total system design, it is apparent that the hardware development effort should be minimized. The Burroughs D-machine in contradistinction to the other candidates is more than just a processing unit. The D-machine also provides, besides its interpreters, M2 memory modules and I/O device ports all of which are interconnected through a switch interlock to allow a multiprocessor configuration. The choice of the D-machine is therefore to be preferred for the laboratory model since no other equipment could match its flexibility and minimize the hardware development cycle to the same degree.

5.3.4 Configuration Utilizing the D-Machine

Figure 5.3-2 shows the proposed laboratory model employing interpreters (I), a switch interlock (SWI), M2 modules, an M3 unit, a space station oriented peripheral interface, and an I/O equipment interface for controlling the laboratory model.

Three interpreters are to be used for the three processors while the fourth interpreter will be dedicated to be the I/O unit. The fifth interpreter will gather detailed operating statistics of the lab model and will control the commercial peripherals. By separating the statistics gathering function of the laboratory model from the operating aspects of the space station multiprocessor, it is thought that more realistic experimental situations may be generated.

Associated with each I is an M1-CAM combination attached to a device port of the switch interlock. A device port is utilized in order that the M1-CAM can be permanently dedicated to a particular I. Another device port interface contains a data bus control unit (DBCU). All of the space station peripheral equipments can be attached to this data bus. It would also be possible to have the actions of the specialized experiment simulated by another interpreter.

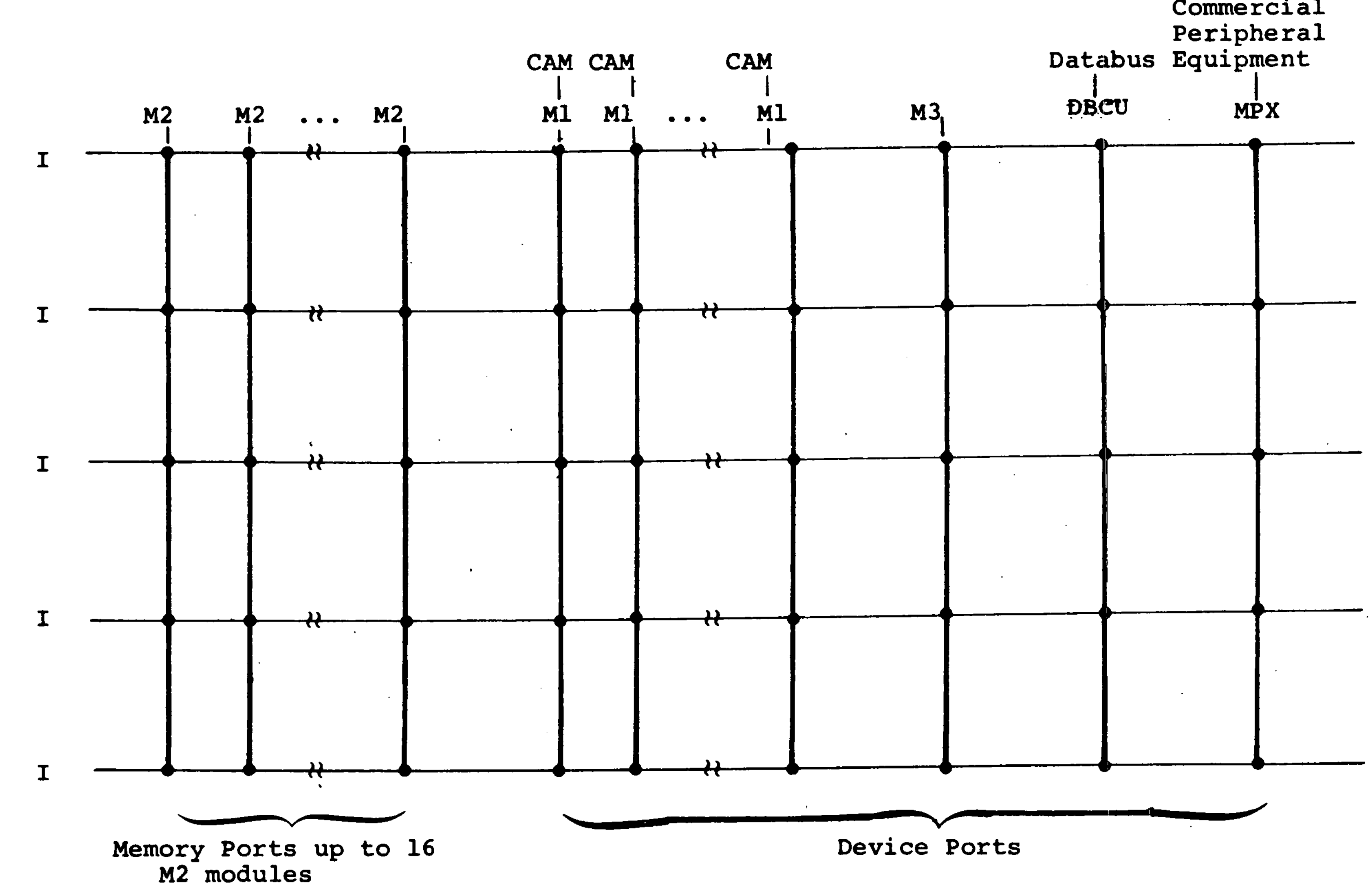


Figure 5.3-2: Proposed Laboratory Model

Since M2-M3 management is a critical high speed function, the M3 unit (disk) is given an independent device port. Finally all the commercial peripheral equipment such as card readers, printers, terminals, and magnetic tape are attached to another distinct device port through a multiplexer (MPX). It is anticipated that one interpreter interfaced through a peripheral multiplexer can handle all the equipment needed to start, stop, dump and monitor the multiprocessor system. Provisions should be made to vary the number of M2 modules from 1 to 16 in order to measure the effect of memory contention upon performance,

5.3.5 How to Use the Laboratory Model

This section will describe how various features of the proposed multiprocessor may be implemented on the D-machine.

5.3.5.1 M2 Interleaving and Addressing: The proposed design specifies that four M2 units should be interleaved on their lower order address bits. The addressable M2 entity is a 32 bit word. Both single and double words may be requested in a single M2 command.

One may conceive of requiring the SWI and specialized hardware to provide these functions. However, this would be a permanent change. A better alternative would be to allow the microprogram to generate the address appropriately. Since the interpreter must, in any case, generate the M2 address under micro-control, the interleaved mapping could be accomplished at this level with a microsequence. This technique allows the degree of interleaving and the details of the memory commands to become a parameter which may be varied to study and improve system performance.

5.3.5.2 Local Memory and the CAM: The CAM and M1 are not standard components of the D-machine. It is proposed to design, build and interface three of these units through device ports of the SWI. Present day technology allows 16 bits of associative memory to be purchased in one circuit chip. Our design specifies 640 bits. An important parameter to be measured will be system performance as a function of the number of words in the associative memory. The amount of CAM to use is conveniently a parameter which the microprograms could control.

The M1 storage can be purchased with from 64 to 1024 bits on a single circuit chip. Since the requirements indicate 6144 bits for each M1 unit, technology is of no problem in this area.

By constructing a specific M1-CAM combination tailored to our needs, it is anticipated that this emulated system will be able to perform without much degradation.

5.3.5.3 Fault Tolerance: It is realized that error detection is very specific to the particular hardware elements employed. A verification of the error detecting mechanisms discussed in this report cannot be realistically attempted with a simulation model. However, many of the principles of recovery from processor, M1 and M2 failure may be verified by simulating the error conditions through an interrupt.

The fault tolerant processing unit design specified in chapter 4 requires two processors and two M1-CAM units to be configured into one unit. This unit will provide the error detection mechanisms and part of the recovery mechanism. In order to simulate this design with D-machine components, two processors with their dedicated M1-CAM can be made to execute the same code. The question is whether they can be synchronized sufficiently to enable a restartable instruction to be interrupted by an error signal. Synchronization may be obtained by either hardware or firmware. A common clock may be used for two interpreters and their associated M1-CAM units. While this is not a difficult task, it is a hardware modification. Alternatively the firmware can synchronize the interpreters by testing global condition bits which may be set into all interpreters. The testing of these conditions at fixed times within the microprogram will enable a faster interpreter to stall and the slower one to catch up. While this type of synchronization costs time, it is flexible, and real time is of secondary importance as compared to the verification of the recovery principles.

The micro-control will allow instructions to be designed as restartable with a compute and a copy phase. The time differential between the restartable version of the instruction and the non-restartable version will also be an informative measurement.

The redundant storage in M2, and the 32 or 64 bit accesses will be under micro-control, and therefore the M2 access width can be varied to determine its effect upon system performance.

5.3.5.4 The Laboratory Model and Real Time: The D-machine was chosen to implement the laboratory model because it was felt that it provided an architecture which was closest to the proposed multiprocessor. For this reason it is hoped that a reasonable measurement of the MP's potential real time performance may be made. A number of factors will tend to make the lab model slower than the proposed design.

a) The M1-CAM combination is removed from the interpreter. It therefore will run slower than the proposed M1-CAM. A 64 bit access including address generation is reasonable in 200 ns instead of 100 ns.

b) The interpreter is not as powerful a computation device as the proposed processor. For this reason many of the hardware features of the proposed processor must be emulated through firmware. This will of course slow down the system performance.

c) The necessity to control memory interleaving in, and simulate the primitive memory commands with an interpreter instead of in the M2 module proper, will increase the apparent M2 cycle time.

When all these factors are considered it is reasonable to estimate that a performance degradation by factors of two to eight might be obtained in the laboratory model. This is really quite satisfactory when one considers that a software simulator can run 100 to 10,000 times slower than real time.