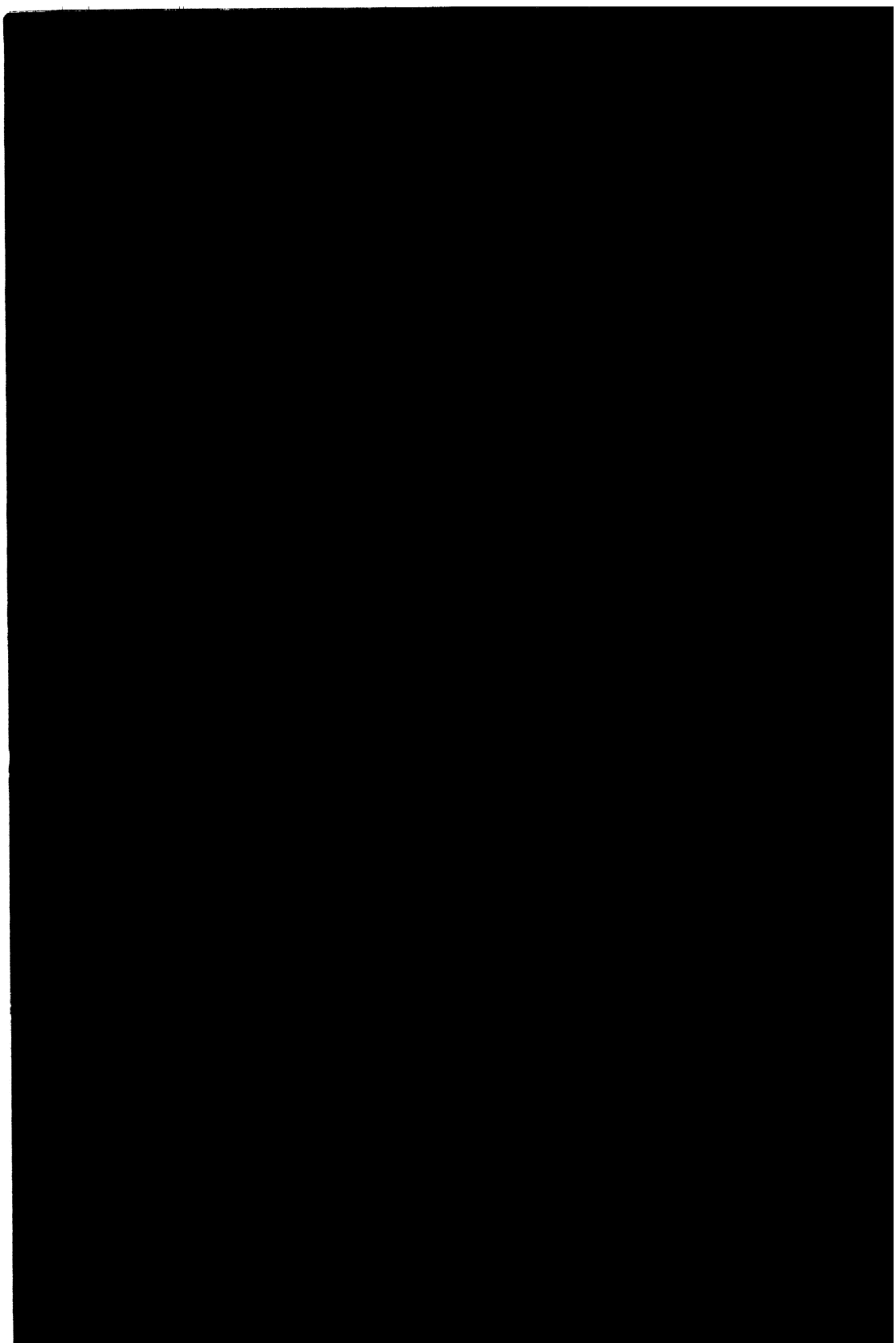


This blank page was inserted to preserve pagination.

INPUT/OUTPUT IN TIME-SHARED, SEGMENTED, MULTIPROCESSOR SYSTEMS

by

ARTHUR ANSHEL SMITH

S.B., Massachusetts Institute of Technology
(1964)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 1966

Signature of Author Arthur A. Smith
Department of Electrical Engineering, January 17, 1966

Certified by Jack B. Dennis
Thesis Supervisor

Accepted by Thomas J. Gray
Chairman, Departmental Committee on Graduate Students

INPUT/OUTPUT IN TIME-SHARED, SEGMENTED, MULTIPROCESSOR SYSTEMS

by

ARTHUR ANSHEL SMITH

Submitted to the Department of Electrical Engineering
on January 17, 1966 in partial fulfillment of the
requirements for the degree of Master of Science

ABSTRACT

After introducing and defining the concepts of time-sharing, segmentation, and multiprocessing, two classes of systems incorporating these are introduced. Both classes use associative memories, as 'look behind' devices to speed the operation of addressing the segmented memory, with the distinction between classes being the location of the associative memory. In one class, there is one associative memory for each processing element, no matter how many main memory units are connected to a processor; in the second class, there is one associative memory for each main memory unit, with the processors sharing the associative memory. After introducing two criteria for input/output systems, that the overhead associated with their use be small and that they may be physically and logically simple, and describing further operations of the systems, it is concluded that members of the second class, having shared associative memories, best meet these criteria.

Thesis Supervisor: Jack B. Dennis
Title: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENT

Appreciation to Prof. J. B. Dennis, my thesis supervisor, and E. C. Van Horn, for their many hours of spent discussing the ideas contained herein; to P. J. Denning, and F. L. Luconi for their constructive and enlightening criticisms; and to Marsha Baker, my typist, for her patience with my handwriting.

"Work reported herein was supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government."

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION 6
- Time-sharing 6
- Memory protection and relocation 8
- Segmentation 12
- Multiprocessors 13

CHAPTER 2. DETAILS OF TWO CLASSES OF CONFIGURATIONS 18
- Class 1: Associative memory in the processor 18
- Class 2: Associative memories pooled in the arbiters 22
- Subclass 1: Sequential probing of associative memories 23
- Subclass 2: Broadcasting to multiport associative memories 24
- Further considerations for both classes 27

CHAPTER 3. INTRODUCTION TO INPUT/OUTPUT 31
- First criterion for input/output low-overhead 31
- Second criterion for input/output-simplicity 33
- Realizing the first criterion-addressing 36

CHAPTER 4. SCHEDULING 38

CHAPTER 5. DETAILS OF INPUT/OUTPUT 43
- Example 46

CHAPTER 6. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER RESEARCH . . . 49

APPENDIX 1. NOTATION 52

APPENDIX 2. FLOW CHARTS FOR THE ADDRESSING SCHEMES OF AMP, AMAS, AND MPAMA 54

APPENDIX 3. A CONFIGURATION FOR MINIMUM RESPONSE TIME 64

APPENDIX 4. MINIMUM EQUIPMENT INPUT/OUTPUT DEVICES 68

BIBLIOGRAPHY 72

LIST OF ILLUSTRATIONS

Figure 1.1	Simple state word storage	9
Figure 1.2	A simple segmentation scheme.	14
Figure 1.3	Overall system configurations	14
Figure 2.1	An associative memory being used a a 'look behind device.	20
Figure 2.2	Processor in AMP system	21
Figure 2.3	Diagram of an AMAS system	25
Figure 2.4	Diagram of an MPAMA system	26
Figure 2.5	State word organization	30
Figure 4.1	Typical ring of queues for a two processor type system	39
Figure A2.1	Flow chart for the addressing scheme of AMP	57,58
Figure A2.2	Flow chart for the addressing scheme of AMAS	59,60,61
Figure A2.3	Flow chart for the addressing scheme of MPAMA	62,63
Figure A3.1	Ring structure for the sample program	67
Figure A4.1	Sequence of signals betwen arbiter and punch	71

CHAPTER 1. INTRODUCTION.

With the rapid advances made recently in high speed digital computers, there has been a proliferation of ad hoc solutions to the various problems arising from ever larger configurations. This paper is a study of the effects of various requirements, especially those associated with input/output, on a specific type of computing system. The system under study is assumed to have three properties.

1) The computer operates in a time-sharing mode, with many users being serviced with apparent simultaneity.[2,13,17]
2) Memory is segmented, according to the scheme of Dennis, with individual words being referred to as elements of segments.[5,8,11,14]
3) There are several processing units using the main memory simultaneously.[5,8]

A brief discussion of these properties is now given.

Time-sharing

A time-shared computer is one in which any one of a number of processes (called user processes or user programs, or simply users) may be being executed at any instant of time. The time during which any given process is in execution, and the length of time during which it remains in execution, are both unpredictable as far as the user is concerned. The determination of which program should be in execution, and for what amount of time, is the job of the supervisor, a program which enjoys a certain amount of immunity from the randomness of the users. The supervisor may perform other tasks also, but it is, primarily the scheduler and hence, originator of all activity within the system.

In such a system those users which are not in execution must be stored someplace. If there are only a few users, and the length of their program and data is small then perhaps they can all be stored in main memory, i.e. the set of memory cells referenced by the address portion of a typical instruction, or by the program counter. If, however, there is too much information to be held in main memory all at once, then auxiliary memory must be used. By auxiliary memory is meant the set of all disks, drums, tapes, etc. which may contain information capable of being brought into main memory, or which may accept information from main memory.

It is important to note, at this point, that, if a user is to produce the proper results regardless of when he is executed, there must be more than his program and data saved away in memory when he is not being executed. There must be at least the location of the instruction being executed, and the values of various processor registers (accumulators, index registers, etc.). Perhaps more information such as the allocation of certain external devices, is also needed. The set of all information needed to successfully restart a user after another user has been executed is defined to be the state word of that user. Each user, from the time he is first in execution, until the time he is finished, must have some storage assigned to his state word. Since, furthermore, the state word will spend most of its time in storage, rather than in a processor, there is no need to continually shift it about in memory. In fact, we shall

assume that the location of the state word of a given process remains the same throughout the period from the beginning of execution to the termination of that process. Figure 1.1 shows state words as they are stored in memory.

Memory protection and relocation

Since part of one users program may be in main memory while another user's process is being executed, there must be some scheme to protect the dormant (i.e. not being executed) process from the one being executed. There are two rather simple ways to provide this protection. First, if each process occupies one contiguous piece of memory (at least while it is being executed), then any address generated can be compared to an upper and lower bound. If the address is not within these bounds, an error condition exists. Second, each address generated may be compared to a table of valid addresses for this process. In either event it is more practical to consider not the individual words of memory, but larger pieces, of size 2^{λ} words, called blocks. In the first case, this permits the checking apparatus to be smaller by λ bits, while in the second case, it permits the reduction of the search to a table lookup. In both cases, the effect is to partition main memory into blocks. To lend concreteness to this argument, assume that main memory is 2^{α} words large, and that each block of 2^{λ} words long; then there are $2^{\alpha-\lambda}$ blocks. For a reasonable size memory, 64k words(α=16),

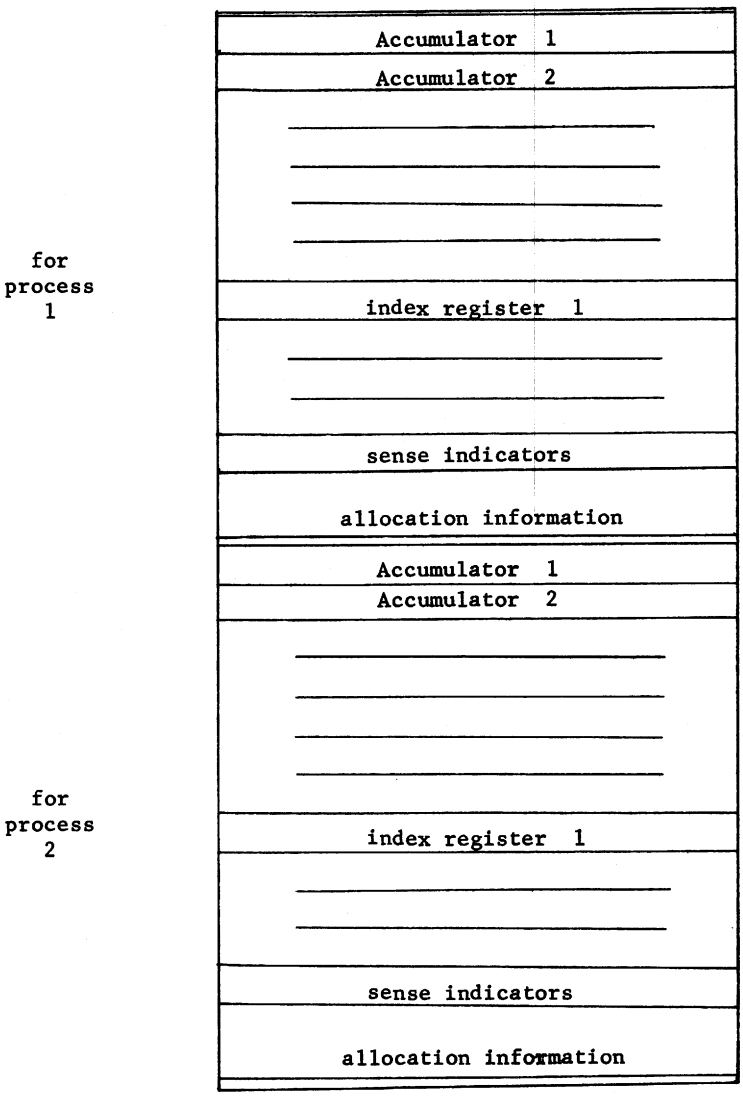


Figure 1.1. Simple state word storage

and reasonable size blocks, 1000_8 words (λ=9), then there are 128 blocks, and a table lookup to determine validity is certainly possible.

Since main memory is divided into blocks, the allocation of it must also be in terms of blocks, and hence, the auxilary storage devices may be designed to perform data transfers in multiples of one block. This is a further argument for dividing main memory into blocks. For convenience, we shall refer to the λ low order bits of the address as the line number, and the (α-λ) high order bits as the block number. See also appendix 1.

Since there may be many users, many blocks of a given user may be placed on auxiliary storage between periods of execution, and it may be more convenient to return these blocks to main memory locations other than those in which they were initially stored. It this is to be done, some facility must exist for forcing the program to perform as it would if it had not been moved about in main memory. This facility is called relocation. There are many possible alternatives for this. One is to provide enough information to permit the supervisor to modify the words of the program whenever it is moved. This is not particularly desirable since it may involve considerable overhead expense. A second alternative is to provide a mapping from the addresses which the user generates into addresses which represent the proper locations in main memory. At this point, we introduce some terminology to facilitate the discussion. First, the block number, line number pair which is actually used to perform

the memory access is termed a location. The configuration of bits which the program generates which correspond to a location is termed an address. Since protection is provided on the basis of blocks, it is not unreasonable to provide relocation on the basis of blocks also. The line number of the location and of the address are then the same, while the block number of the location corresponds to the page number of the address. All that is needed for automatic relocation, therefore, is a table, for each user, which gives the correspondence between page numbers and block numbers.

Note that there exists the possibility of more addressses than locations. This situation is entirely acceptable if no more words are actually needed than there are locations; the other words may be stored on auxiliary storage. In fact this situation is actually desirable; for example, if a user has two arrays which may grow exceedingly large, but of which only the last few entries are relevant at any given time, then the user may start one array at L/2, and the other array at $\frac{3L}{4}$, where L is the largest possible address. Either array may then grow to occupy one quarter of the total possible addresses, and the programmer need not concern himself with the problems of dynamic arrays, if there are enough addresses. A large number of addresses implies a large number of bits in the address portion of an instruction, most of which will be zero in most cases. To avoid the expense inherent in many bits in the address, we simply abbreviate some of the bits by using a scheme called segmentation.[8,11]

Segmentation

In the segmentation scheme, we assume there exist several attachment registers which can be loaded, in essence, with the particular high order address bits desired. Then, in the instruction, instead of providing large page numbers, a few high order bits, called the attachment tag or segment tag, are used as the number of the attachment register to be used with this address. The total address is formed by concatenating the contents of the proper attachment register with the page number and line number from the instruction. Everything but the line number is then transformed to a block number which, with the line number, forms a location. Any configuration of bits which can be loaded into an attachment register determines a segment, the particular configuration being called the name of the segment. The number of bits provided in an instruction for the page number and line number and the size of index registers, determine the maximum possible size of a segment; since this maximum size is very large (presumably) the following fiat will be made to simplify the resulting system: there is no relation between the addresses of two different segments. That is, whereas if 1 is added to the address of the last word of page 1 of a given segment, the resulting address is that of the first word of page 2 of that segment, if 1 is added to the address of the last word of the last page of segment 1, the resulting address is meaningless; it is not the address of the first word in segment 2! Repeating, the words of any one segment are contiguous if their page number, line number pairs

are contiguous; the words of two different segments are unrelated.[5]

This simple restriction permits much simplification of the hardware. In particular, it means that relocation for each segment may be done without regard to other segments. Namely, for each segment, there exists a segment page table, or simply page table, the i^{th} entry of which gives the block number corresponding to the i^{th} page of the segment. The page table may also give protection information for each block, although this, in general, will be the same for each page of a given segment. Note that in this way several users may use the same segment without establishing too elaborate conventions. Note also, that there is implied a relationship between a segment tag and the page table for the associated page table; this relationship is established by the supervisor. Figure 1.2 shows schematically the operation of a segmentation scheme.

Multiprocessors

Up to this point, it has been assumed that there was only one processor utilizing memory. In a large time-sharing system, however, it is unlikely that this will be the case, especially since it is, after a certain point, less expensive to build two processors than to build one processor which is twice as fast as either of the two. All of the previous discussion is valid, but a few more points may be made which arise from the fact that there is more than one processor.

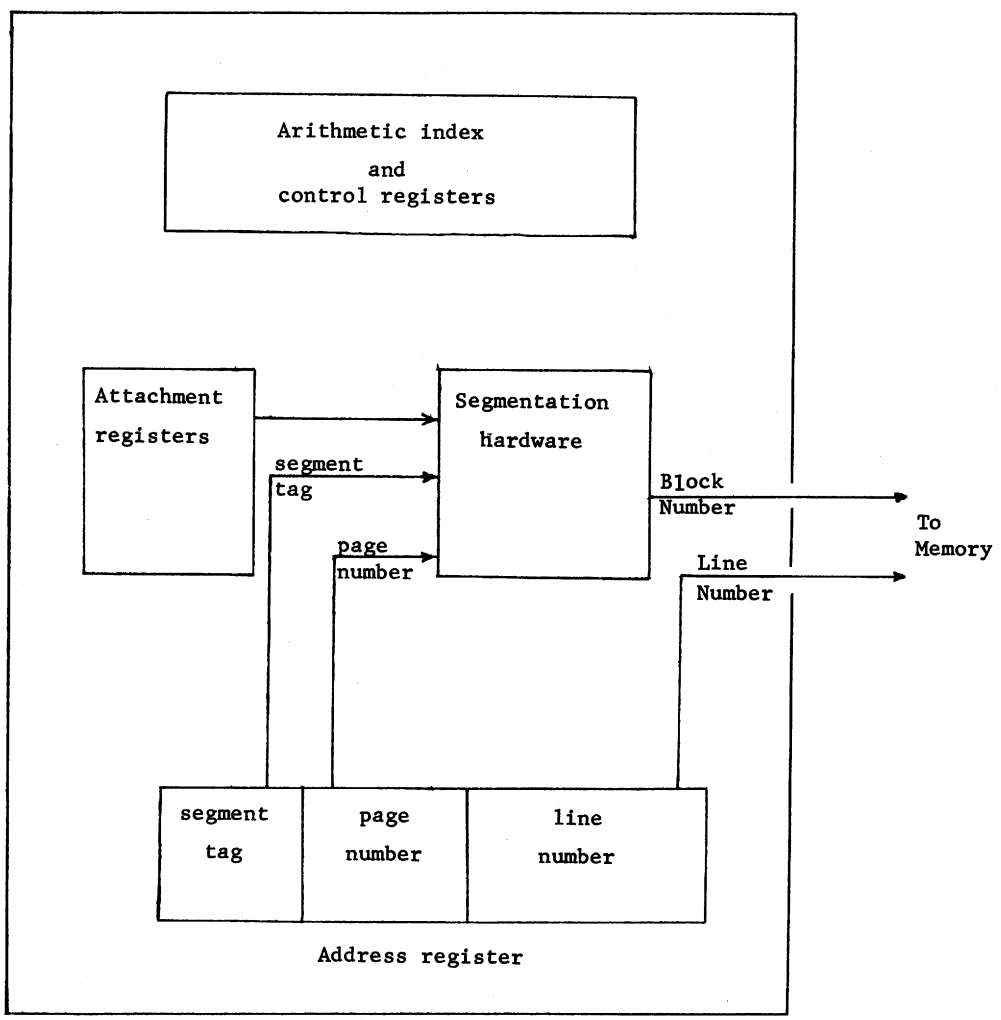


Figure 1.2. A simple segmentation scheme

The most important point is that there must be a device called an arbiter, in the system which resolves conflicts in the use of memory, i.e. it decides who goes first if two processors desire access to the same block of main memory. This is necessary, since, by the way they are constructed, core memories are capable of accessing only one word at a time. This consideration also dictates that instead of one large core memory for a main memory, there should be a multiplicity of smaller core memories. Of course, each of the smaller core memories must have an arbiter for its own, to take advantage of the multiple memories.

If input/output devices, furthermore, are treated as processors i.e. they are capable of executing a certain limited set of instructions which must be fetched from main memory, then there must also be a device in the system which routes I/O instructions fetched by a main processor to the proper I/O device. This function is most easily made part of the arbiter, which must already be connected to each processor and I/O device. Clearly, this function of the arbiter need not be duplicated for multiple memories.

Having added this much to the arbiter, it is not unreasonable to consider adding slightly more, and making the arbiter the controlling element in the system. In particular, the arbiter should have the ability to store a processors state word, load a processor with a state word, and, perhaps, automatically decide what state word should be loaded into what processor.

The first two abilities are needed to handle for example, malfunctions of the I/O system where the process manipulating the I/O device should be halted, and an error routine initiated. The last ability is not needed, but makes a large portion of the scheduling problem quite quickly solvable. More will be said about the arbiter in later chapters.

Finally, reliability dictates that, for every device in the system, there will be at least one duplicate. This permits any element of the system to be disconnected for maintainence purposes without shutting down the system. This then leads to an overall system arrangement as shown in Figure 1.3.

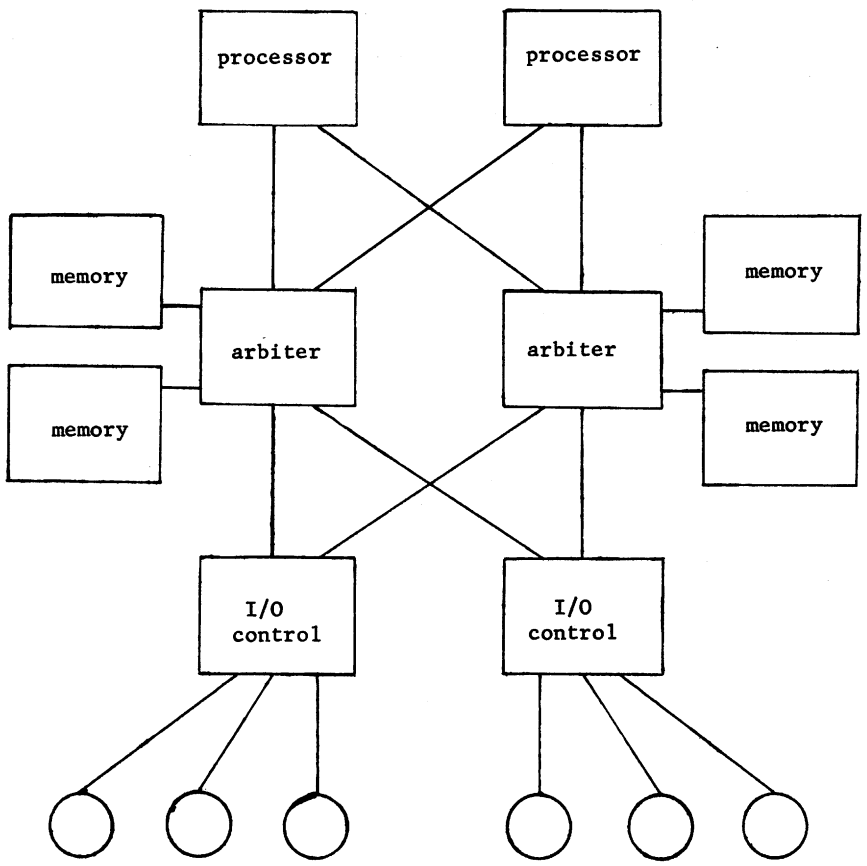


Figure 1.3. Overall system configuration

CHAPTER 2. DETAILS OF TWO CLASSES OF CONFIGURATIONS.

Given the properties of a time-shared, segmented, multiprocessor system, there are, of course, several possible ways to realize it. That is, there are several differing system designs which meet all of the criteria laid down in the previous chapter. At this point, two classes of such systems, differing only in the way attachment registers are implemented, are presented and examined.

Class 1: Associative memory in the processor

The first system to be studied resembles both the one currently being constructed at Project MAC[5,7,14,18,20] and the one originally proposed by Dennis.[8,11,12] As far as the user can tell, this system differs from a standard computer only in the addressing scheme, which uses the segmented form of addressing described in Chapter 1. The address consists of a segment tag, a page number, and a line number; as mentioned before, the supervisor establishes a relationship between the contents of a attachment register and the page table for the appropriate segment. In particular, corresponding to each attachment register, there is a segment page table register which gives the location in main memory of the initial word of the page table for the segment named in that attachment register. The appropriate block number for any segment tag, page number pair, therefore, is determined from the contents of the location formed by adding the page number to the contents

of the appropriate segment page table register. As described, this system requires two memory references for each word of memory actually desired; the first reference is to determine the correct block number, the second to actually fetch the word.

The effective speed of the system is, therefore, halved by using segmentation. Since this is a rather heavy penalty, the speed of the system is improved by adding, to each processor, an associative memory to be used as a 'look behind' device for determining the proper block number, given the segment tag, page number pair; i.e. an associative memory of n words holds the n most recently used segment tag, page number pairs and the corresponding block numbers, for this user. (See Figure 2.1) Instead of using the segment page table registers to determine the proper block number, the associative memory is consulted first, and only when the segment tag, page number pair is not found, are the segment page table registers used. In this way, much of the time spent in determining block numbers can be eliminated, and/or overlapped with the cycle of main memory. Figure 2.2 shows a typical processor for this system; since the associative memory is located in the processor, this system will be called AMP.

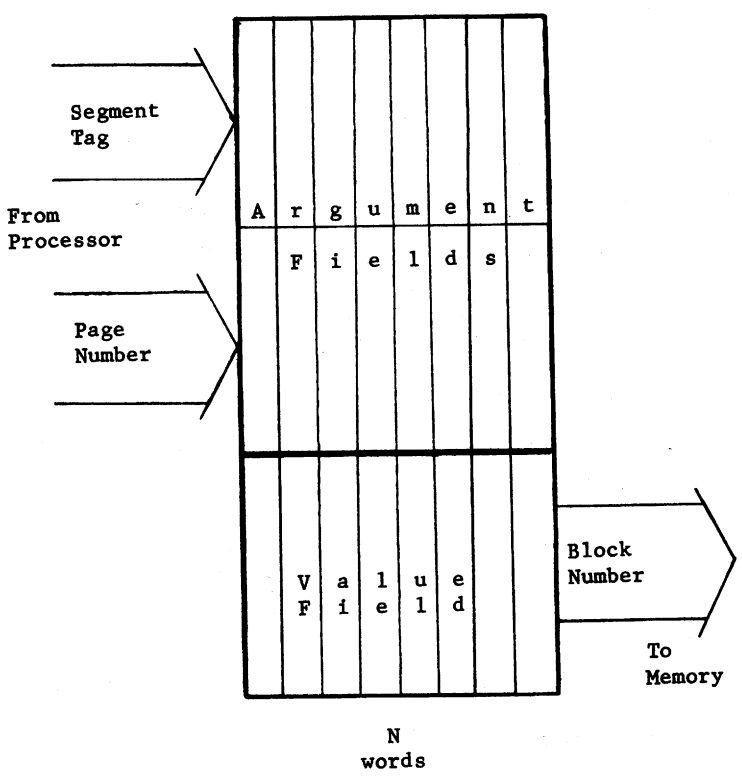


Figure 2.1. An associative memory being used as a 'look behind' device

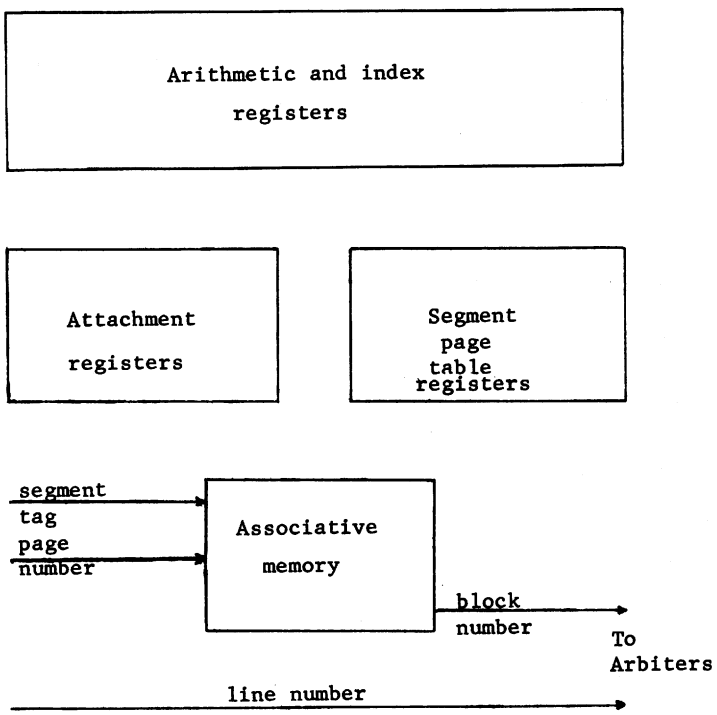


Figure 2.2. Processor in AMP system

Class 2: Associative memories pooled in the arbiters

Since AMP is assumed to be a multiprocessor system an alternative method of organization immediately presents itself. In particular since the processors all share a common main memory, there exists the possibility of their sharing associative memory also. Just as sharing permits more efficient use of main memory, the same statistical properties should permit more efficient utilization of the associative memory. Since associative memories are currently rather expensive, this may in fact be an important economic factor. The issue is not one sided, however, since if the associative memory is pooled among several devices, its entries must contain information telling to which processor it belongs. This means that each word of the associative memory must contain more bits, and consequently be more expensive.

There are other problems involving the pooling of the associative memory, however, which are not simply economic. The problem of many devices communicating with the same associatve memory is the same as that which arose with regard to many devices and the main memory; and the place to solve both problems is the same, in the arbiter. A distinction between the problems exists, however, in that the block number of a word of memory uniquely determines one arbiter which has control over that word, whereas, a segment tag, page number pair does not determine an associative memory with a matching entry. In particular,

the only way of determining the presence of a segment tag, page number pair somewhere in the associative memory, is to probe the associative memory in each arbiter. This can be done in two ways, either by interrogating the arbiters in sequence, or by broadcasting the interrogation to all arbiters at once.

Subclass 1: Sequential probing of associative memories

In this system, called AMAS since the associative memories in the arbiters are sequentially probed, the device attempting to access main memory directs a request to some particular arbiter, 'A', for the word corresponding to some segment tag, page number pair, for this user. 'A' may respond either with the desired word, or with an error flag indicating that the associative memory associated with 'A' does not contain a match for this segment tag, page number pair. If the error response occurs, the process is repeated with arbiter 'B', and so on, until either a match is found, or some predetermined number of arbiters has been unsuccessfully probed. If a match is found, then the block number is known, and the desired word may be fetched from main memory. If no match is found, the segment page table registers are used to determine the block number. Note that at each arbiter, there may be some time wasted in queues, and that, if no match exists in any associative memory, this time may be more than that which would have been wasted by using the segment page table registers in the first place.

A diagram of processors and arbiters in AMAS is shown in Figure 2.3.

Subclass 2: Broadcasting to multiport associative memories

In this system, the device requesting access to main memory "broadcasts" to several arbiters at once, the segment tag, the page number pair, and the user's identification bits. If a match is found, and it is assumed that the associative memories are loaded in such a way that only one match will be found, then the output of the associative memory is the proper block number, and a fetch from main memory is performed by the appropriate arbiter. It is clear in this scheme that the several associative memories will be rather busy looking for matches, since each arbiter must probe its associative memory for every request of every processor connected to it. In comparison with AMAS, assuming that the interconnections between arbiters and processors is the same, this system generates twice as many requests on the average [assuming uniform distribution of requests to arbiters]. This could present a problem of time being spent waiting in line to use the associative memory; using current, integrated circuit technology, however, it is possible to build an associative memory with many input and output pairs, so that each processor could have its own private input and output buffers, and no waiting would ever need to occur in normal operation. Since this system uses a multiport associative memory in the arbiter, it is called MPAMA. A diagram of the processors and arbiters is shown in Figure 2.4

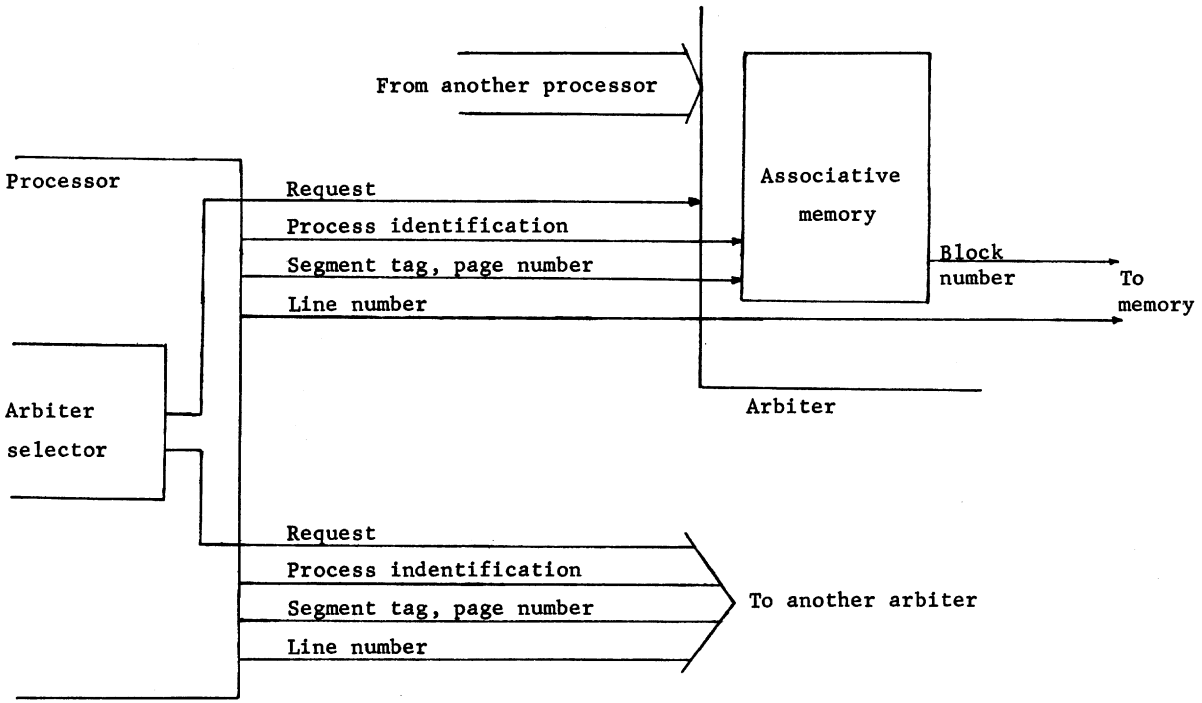


Figure 2.3. Diagram of an AMAS system

The arbiter selector in the processor turns on a request to one arbiter at a time.

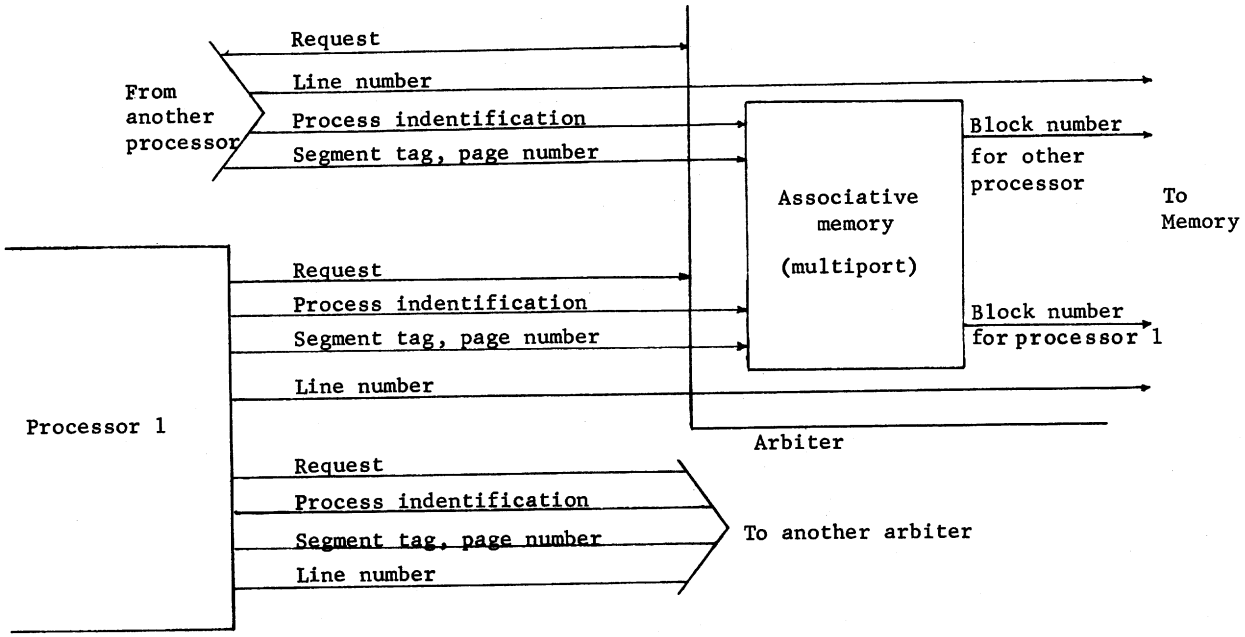


Figure 2.4. Diagram of a MPAMA system

The processor turns on all requests simultaneously. The multiport associative memory services all processors simultaneously.

Further considerations for both classes

Having sketched the basic principles of the two classes of systems, the operation will be examined more closely. Consider first the problem of loading an attachment register so that references involving that register are defined. The most obvious instruction for performing this operation is

Load Attachment Register from Y

where Y contains the segment name. Although obvious this method is undesirable since the formation of an association between the segment name and the location of its page table, which will probably involve much overhead, must be repeated each time an attachment register is loaded. A more devious system, involving two instructions, is therefore assumed; the first involves all the overhead present in the above scheme, but is used less frequently; the second requires very little overhead, and is executed whenever it is desired to load an attachment register.

The first instruction is really a call to the supervisor. The notation for this instruction shall be

j := acquire Y

where Y contains a segment name. The notation means that the supervisor should perform all the bookeeping needed to permit the user to reference the segment, and should place the address of the page table for this segment [along with protection information] into a special segment called the user's segment directory. The location within the segment directory at which this information is stored, called Y's directory index, is placed in j.

The second instruction is a true machine instruction:

LAR n,j Load Attachment Register n from j

where j contains a directory index. Notice that it should be no more difficult to implement the LAR instruction that it is to implement a load index register instruction. If the user, furthermore, cannot alter his segment directory, except by way of the supervisor, then it is impossible for the user to load an attachment register with invalid information. In actual operation, the LAR instruction places the address on the page table of a segment into the segment page table register.

Next consider the use of the associative memory. As first described, the associative memory was merely a look behind device, and the attachment registers and segment page table registers were actually distinct hardware registers. A little thought will reveal that the attachment registers are no longer needed, since only the address of the page table is really used, and it exists in the segment page table register. Since all systems are committed to an associative memory, furthermore, it is possible to eliminate the page table registers as distinct hardware items, and implement then as specially flaged entries in the associative memory. This should, in addition to eliminating registers, improve the statistical properties of the pooled associative memory.

If the entries in the associative memory correspond to page table registers are not used frequently, it is quite likely that these entries will be purged from the memory. Since this would make it impossible to determine the block number of a page of some segment if the segment tag, page number pair were not already in the associative memory, some trickery is needed. Observe first that someplace in the state word of a user, there must be storage provided for the attachment registers so that a process, once stopped, can be restarted. All that is really needed then is the directory index for each attachment register, corresponding to the most recent LAR n,j instruction. One way to insure this information being always present, is to have the LAR n,j instruction copy the contents of j into a location, say k_2+n within the state word. The contents of a segment page table register can always be reconstructed, in this case, by effectively executing an LAR n, k_2+n instruction. The organization of the state word for this system is shown in Figure 2.5. Details of all three addressing schemes are given in Appendix 2.

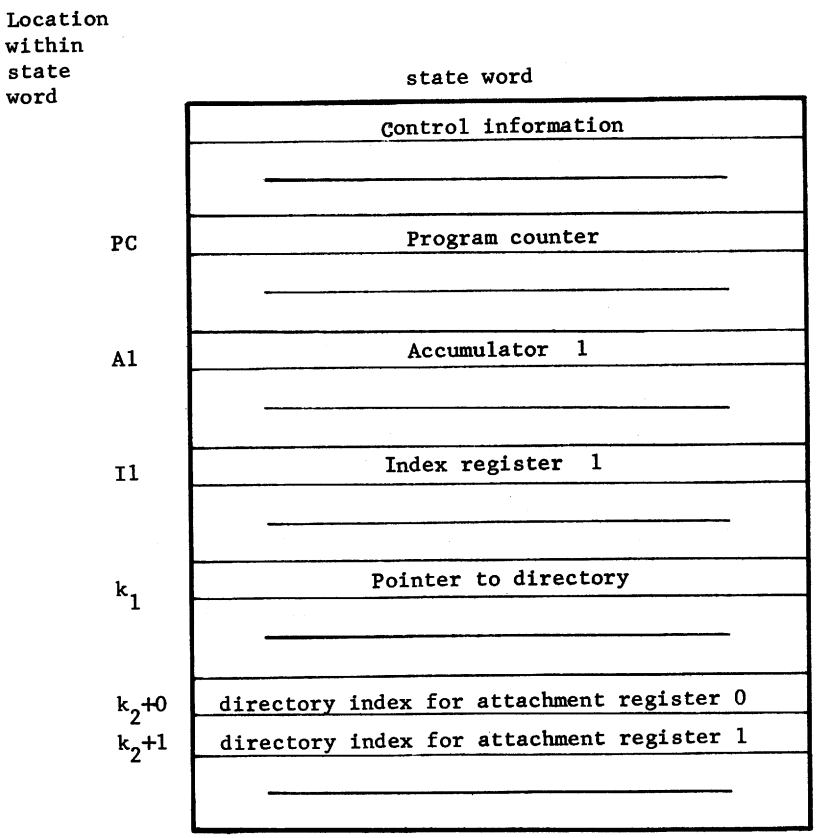


Figure 2.5. State word organization

CHAPTER 3. INTRODUCTION TO INPUT/OUTPUT.

At this point it is appropriate to consider the processes by which programs acquire data and demonstrate results. Although there are many aspects of these processes, the general term for the processes as a whole is input/output; these input/output processes are characterized by the fact that they interact with elements of the system other than the main memory and processors, and hence may involve relatively long times for completion.

First criterion for input/output-low overhead

As a preface, two criteria for any input/output system are introduced. First and foremost is the requirement that whatever input/output is performed, the overhead shall be minimal. By overhead is meant that expense, in terms of time and money, which would not be incurred had the equivalent operation been done on a non time-shared, segmented, multiprocessor system. This goal is, of course, reasonable, but it has been made a primary objective as a simple economic consideration; namely, most of the computing being done at the present time is limited, not by the speed of processing, but by the speed of input/output devices. This is entriely understandable, since most input/output devices, such as tape drives, line printers, and punched card equipment, are electromechanical by nature, rather than electronic.

This limitation means, however, that if there is overhead associated with input/output, then certain programs will never be reasonably run in a time-shared, segmented, multiprocessor system. In particular, if a program is limited by input/output time, then, although the computations can be performed more efficiently, and even less expensively, by a time-shared, segmented, multiprocessor system, the expense of the overhead in doing input/output will overshadow the other savings, and such a job will always be best performed on a conventional machine. More concretely, consider a typical business type data processing problem, such as the production of a weekly payroll. Such a job is typically done on a small, slow, business oriented machine [such as the ubiquitous IBM 1401], even though it might be performed more efficiently, and with less programmer effort, on a large machine. This reason for doing it on a slow machine is simply that, since most of the processor time will be spent waiting for input/output to be completed, it is less expensive to have a small processor wait than a large one. It is clear then that there might be some slight advantage for the individual with an input/output limited problem to using a time-shared segmented multiprocessor system, if there is no overhead in performing input/output.

The important reasons for wanting to be able to accomodate such users, however, are not the advantages to the user, but the advantages to the system. It is important to recall at this point, that, classically, the reason for wanting to time-share a computer system is that, in this way, the load upon the various elements of the system would be averaged over several users, and hence would be more nearly constant. Inherent in this philosophy is the idea that some tasks will be limited by computation speed [these are called scientific jobs] and others will be limited by input/output speeds [these are called 'business' jobs]. In fact, however, if there is any overhead in input/output, then the business jobs will never enter the system, and the efficiency of the system as a whole will decrease.

Second criterion for input/output-simplicity

The second criterion for input/output system is that there must be an interface to which rather simple data acquisition systems may attach. This is to attract more potential users to the system, and hence to improve the statistics of usage. In particular, it is desired that some inefficiency of processor time may be exchanged for simplicity of hardware, and the resulting increase in the number of problems actually being solved on the system will compensate in overall efficiency for this local inefficiency. This goal is really an attempt to prevent small conventional system from

flourishing because certain small data acquisition devices, such as analog-digital converters, and counters, are more easily connected to small systems than to large ones. This second goal suggests the idea that the solutions of some problems associated with input/output are really no more than special cases of solutions to general problems associated with multiple processors. The specific problem of allocating input/output devices ought really to be considered part of the problem of allocating processors to parallel paths within one program. If, indeed, this approach is taken, then there immediately arises the question of whether all processors are equivalent; that is, can any processor perform any task? If the processor is an input/output channel [in the IBM sense of channel] then the response to such a question must be in the negative; one should not expect a data-channel to perform floating point division. Closer examination, however, reveals that such an answer is really avoiding the question, since it already assumes the existence of a data channel. The real question is whether or not such a channel should be built. Without attempting to answer this question, it is simply noted that data channels were added to the IBM 704, to form the IBM 709; it was easier to add them than to redesign the entire processor. Recently, however, the DEC PDP-6 has appeared, aimed at the same market as the IBM 7090/7094, without any data channels, using instead, a highly flexible system of priority interrupts. The philosophy is simply that, using this interrupt system, input/output activity requires a sufficiently small percentage of the processor time that the cost

of building a data channel would be greater than the savings which would result. If, in fact, in a large system, some processors are used as data channels, the prime benefit is in reliability, since now not only may processors replace processors, and data channels replace data channels, but data channels can replace processors and vice versa.

It is impossible to resolve the questions of philosophy involved in one paragraph, however, it is worthwhile to note that no program can keep all parts of a modern sophisticated processor equally busy, without extremely careful (and unlikely) planning. [The CDC 6600 provides the biggest challenge in this respect]. It is not unreasonable, therefore, to consider any large system to be built from many blocks of similar but not identical nature; something similar to the IBM system 360 will probably exist, with the exact mix of models depending on the expected type of job for each particular installation. A user may then specify which processor he wanted to execute his program, chosing on the basis of what he needs, and how much he is willing to pay. A user with little or no floating point calculation should not have to pay for expensive floating point hardware; he may even be willing to let all floating point calculations be done interpretively, by software. A user doing no variable length comparisons, similarly, should not pay for hardware to do them. Each user will normally request the minimum hardware to do his job efficiently. If, however, the exact item he wants is not available,

no matter; the supervisor can assign a different processor, and adjust his charges accordingly. This scheme tacitly assumes that all processors have a common order code; with the exception of specialized I/O processors, this is not an unreasonable assumption, and will in fact be made.

Just as a variety of processors was postulated, so also a broad spectrum of input/output processors will be assumed. These will vary from "standard" units, such as tape drives and controllers, to "non-standard" units such as on-line mass spectrometers. Of course, the supervisor can only be expected to "know" about standard units; its knowledge of non-standard units will be limited to their existance and a list of authorized users. As will be seen shortly, this is really all that is needed.

Realizing the first criterion-addressing

One of the most obvious methods of reducing overhead in input/output, or in anything else, is to reduce the work the supervisor program must perform. When an input/output operation is begun, the location of some data must generally be communicated to the input/output device; since true locations are known only to the supervisor, whereas segmented addresses only are known to the user programs, locations must be transmitted to input/output devices as segmented addresses, to avoid calling the supervisor every time input/output is desired. It thus follows as an immediate consequence of the first criterion, low overhead in input/output, that all processors and input/output devices must use segmented addresses, and must, accordingly,

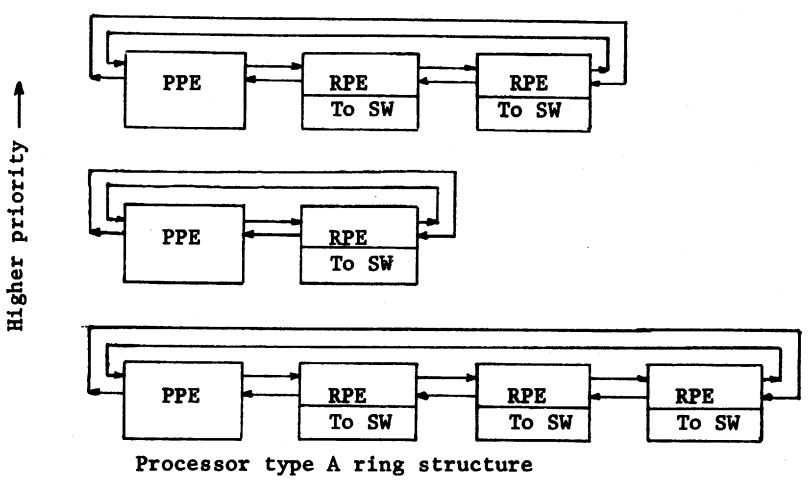
have attachment and segment page table registers. In fact, the need for calling the supervisor to provide locations for each input/output operation generates so high a percentage of the total input/output overhead, that the requirement of segmented addressing for input/output devices will be considered equivalent to the first criterion.

CHAPTER 4: SCHEDULING.

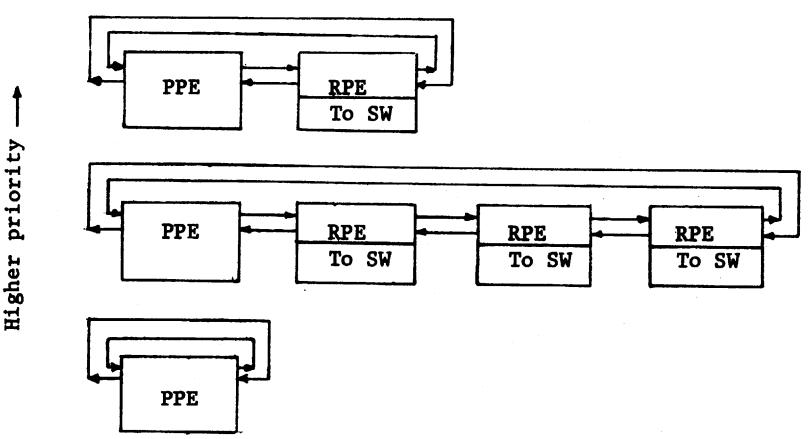
Before proceeding, it is necessary to make some assumptions about the method by which jobs are scheduled on the microscopic level. The question of macroscopic scheduling, of which programs enter core, or of how users are allotted the various resources of the system, is not relevant; such schemes exist in infinite variety. What is relevant is the scheme by which the system's resources are kept maximally busy on a millisecond to millisecond basis, by reallocating processors to the various programs as events occur within the system. The most important event, for our purposes, is the requesting, by a program, of a different processor, when the program requires different facilities for its execution. One important instance of such an event occurs when a program desires to perform input/output activity.

For the purpose of performing this scheduling, a system of queues is assumed, one for each processor type, with these queues arranged into rings of processes, one ring for each priority level. A diagram of this structure is shown in Figure 4.1

The Permanent Priority Entries, labelled PPE, one for each processor type, for each priority, always exist; the locations for them are recognized by the hardware. Each entry contains two pointers; one is



Processor type A ring structure



Processor type B ring structure

Figure 4.1. Typical ring of queues for a two processor type system

PPE = Permanent Priority Entry
RPE = Ready Process Entry

the first pointer in a list running forward through all the Ready Process Entries for this processor type and priority; the other is the first entry in a list running backward through these entries. At any instant, therefore, the first(head) and last(tail) entries for any given processor type and priority are known. Since a Ready Process entry contains a pointer to the state word of the associated process, given the Ready Process entry, the processor can find all the information needed to execute the corresponding process.

Scheduling within this system is rather simple; each processor executes the job at the head of its highest priority non-empty ring, removing the corresponding Ready Process entry from the ring structure as it begins execution. As time progresses, the priority of the job under execution decreases. The highest priority in the ring structure is constantly compared with the priority of the jobs being executed, and, if the ring structure contains a job of higher priority than any job being executed by any processor of the corresponding type, a process swap occurs. During a process swap, the state word of the processor is stored in the appropriate place in memory, a Ready Process entry corresponding to the process being executed is placed at the appropriate point in the ring structure, another Ready Process entry is removed from the ring structure, and the corresponding state word is loaded into the processor.

A little reflection will reveal several interesting points. First, since each processor must have the ability to keep track of its ring structure and to execute a process swap, and since these abilities will not be in constant use, it is reasonable to make the hardware for these abilities shareable by incorporating it into the arbiter, and, in effect, providing the processor with instructions, which can be issued to the arbiter, for using the hardware to modify the ring structure and load and store the elements of the state word. Second, loading and storing of the state word of a process ought to be under control of the processor; although performed by the arbiter in fact, the first thing to be loaded should be an indication of what else must be loaded. This lets the programmer control more precisely the overhead involved in his being activated; thus if a user is using only one of sixteen accumulators, he can avoid storing and loading fifteen assumulators every time he is activated. Furthermore, different processors may have different elements in their hardware; one processor may have hardware index registers, while another may use memory locations instead. A third point is that Ready Process entries, instead of having pointers to state words, may be made part of the state word, i.e. given the location of a Ready Process entry, the location of the rest of the state word may be implicitly rather than explicitly determined. This point is mentioned simply to indicate that there is nothing scared about the exact organization described.[9]

More interesting is the observation that, in this system, changing from one processor to another is simply a special type of process swap; the appropriate point, mentioned above for the Ready Process entry corresponding to the process being executed, lies, not in the ring structure for this processor type, but in the ring structure for another processor type. More on this later.

It is important, also, to note that processors in the sense used above need not correspond to different physical entities. For example, one physical magnetic disk storage unit may correspond to as many processors as there are disk access mechanisms.

CHAPTER 5. DETAILS OF INPUT/OUTPUT.

Allocation of input/output devices or other processors, and protection of these from tampering by other users, is very similar to allocation and protection of core memory. It is desirable in both cases to provide the protection in hardware, so that devices or memory may be used with minimum overhead. In both cases, also, it is desirable to lump all the overhead into one call to the supervisor, which is executed once, all necessary allocation and protection thereafter being handled by hardware. More precisely, since a system of memory allocation has been designed, a slight extension makes it suitable for allocation of processors. This extension is simply that in

```
j := acquire(Y)
```

Y may be, not only a segment name, as before, but also a processor name. Since the same instruction is being used, the distinction, if it need be made, must lie in the names; segment names must be distinguishable from processor names so that the supervisor can tell what is being allocated. The only things the supervisor needs to know about the processor being allocated is its existence, and whether or not a particular user is permitted to acquire it. This last restriction is not really meaningful if the processor is a standard arithmetic processor; however, if the processor is a line printer control, then

this restriction is reasonable. In any case, the supervisor allocates the processor, and places an entry at location j in the user's segment directory. [Incidentally, the term segment directory is no longer appropriate since it will contain entries other than those for segments. The name capability directory will therefore be used].

Having created an entry in the capability directory, the only operation which can be performed is placing the entry in an attachment register. As before, this is a hardware operation which places the directory index in an appropriate location in the state word of the process. Now, in order to force execution of the current process by the processor named in attachment register n, a special instruction is executed:

GP n Get Processor from attachment register n

This instruction causes the processor to instruct the arbiter to perform a special process swap, described earlier, in which the Ready Process Entry for the current process is placed in the ring structure for the processor named in attachment register n. The exact point in which to place the Ready Process Entry is determined by the equivalent of the contents of the segment page table register for attachment register n. In particular, this register contains, instead of the location of a page table, the location of the Permanent Process Entry for the processor and priority desired. Thus when the 'GP n' instruction is decoded by the processor, the processor sends

instructions to the arbiter to store its current state word [exactly what is stored is determined by the processor], insert a Ready Process entry into the ring whose Permanent Priority Entry is specified by the contents of the segment page table register for attachment register n, pick the highest priority Ready Process ntry from its own ring structure, and finally, pick up the corresponding state word.

Since the number of Permanent Process Entries is small, there will be unused bits in the segment page table register when it is being used in this manner. These bits should be made part of the state word of the process and should be available to the processor if desired. The particular purpose of this is to provide for many different protected entities with entries in the same ring structure. Consider the problem of two processes wanting to use the same tape controller, but different drives. The supervisor must protect the two users from one another, but the controller should have a simple method of determining to which unit the instructions issued by a process apply. By placing the unit number in the unused bits in the page table register, it is protected from alteration by the user, and at the same time is readily available to the processor.

Example

In order to clarify and illuminate some of these ideas, a program for simulating a card reproducer is shown. The program is written in MAD except that machine code instructions are sprinkled in where needed. These instructions are:

LAR	n,Y	Load Attachment Register n from location Y
GP	n	Get the Processor named in attachment register n
READ	Y	Read a card into location Y, Y+1, etc. until the entire card has been read.
WRITE	Y	Write a card from locations Y, Y+1, etc. until an entire card has been filled.
STATUS	Y	Place the status information for the processor into location Y.

The program follows:

```
      VECTOR VALUES CR = $10, CARD READER, 1$
R     FOR ACQUIRING CARD READER NUMBER 1 FOR THIS PROCESS
      VECTOR VALUES CP = $10, CARD PUNCH 1$
R     FOR PUNCH
      VECTOR VALUES  DSEG = $ SEGMENT, READ, WRITE, CREATE$
R     FOR ACQUIRING A NEW SEGMENT WITH READ AND WRITE
R     CAPABILITIES
      VECTOR VALUES  ZERO = 0
R     THE NAME OF THE CURRENT PROCESSOR IS ASSUMED
```

```
        R   TO BE STORED AT INDEX 0 IN THE CAPABILITY
        R   DIRECTORY
START       JCR = ACQUIRE (CR)
            JCP = ACQUIRE (CP)
            JDSEG = ACQUIRE (DSEG)
        R   GET ALL NEEDED DEVICES AND STORAGE FROM THE SUPERVISOR
            LAR  3,JCR
            LAR  4,JCP
            LAR  5,JDSEG
            LAR  6,ZERO
        R   AT THIS POINT, THERE IS NO ENTRY IN ANY RING STRUCTURE
        R   CORRESPONDING TO THIS PROCESS, SINCE IT IS ACTIVE
LOOP        GP   3
        R   THIS CREATES AN ENTRY IN THE RING STRUCTURE FOR THE
        R   CARD READER.  THE ARITHMETIC PROCESSOR IS NOW WORKING
        R   ON  ANOTHER JOB.  THE CURRENT PROCESS WAITS UNTIL
        R   THE CARD READER WILL SERVICE IT.
            READ [6,2]
            STATUS [6,0]
        R   THE ABOVE TWO INSTRUCTIONS ARE EXECUTED BY
        R   THE CARD READER AFTER THE READY PROCESS ENTRY
        R   HAS BEEN REMOVED FROM THE RING STRUCTURE
            GP   6
        R   PUT READY PROCESS ENTRY INTO ARITHMETIC PROCESSOR'S
        R   RING STRUCTURE, AND WAIT
            WHENEVER ([6,10] .AND. EMASK .NE. 0), EXECUTE ERROR. ([6,0])
```

```
R   EMASK MARKS OFF ALL BUT THE ERROR INDICATIONS
    WHENEVER ([6,0] .AND. EOFMSK .NE. 0), EXECUTE EKIT.
R   TEST FOR NO MORE CARDS.  EOFMSK MARKS ALL BUT END OF
R   FILE FLAG
    GP   4
R   PLACE READY PROCESS ENTRY INTO RING FOR CARD PUNCH, AND
R   WAIT
    WRITE [6,2]
    STATUS [6,1]
R   EXECUTED BY THE CARD PUNCH AFTER REMOVING READY
R   PROCESS ENTRY FROM ITS RINGS
    GP   6
R   WAIT IN ARITHMETIC PROCESSOR RING
    WHENEVER ([6,1] .AND. ERMSK .NE. 0), EXECUTE ERROR. ([6,1])
    TRANSFER TO LOOP
    END OF PROGRAM
```

CHAPTER 6. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER RESEARCH.

In Chapter 2, two classes of systems, which appeared only trivially different then, were described. If the proposals made in the preceeding chapters are valid, however, there is only one reasonable choice for the organization of a time-shared, segmented, multiprocessor system. This choice is MPAMA, systems using multiport associative memories located in the arbiters.

MPAMA is chosen instead of AMAS, systems with associative memories in the arbiters sequentially probed, simply because of speed. By probing all associative memories simultaneously rather than sequentially, MPAMA should require less time than AMAS to determine the proper block number for a given page number, segment tag pair, assuming that the size and time for a single probe of the associative memory is the same in both systems. MPAMA should, therefore, require less time for a memory reference, on the average, and correspondingly be faster than AMAS, although the two systems are otherwise identical.

AMP, with the associative memories located in the processors, is discarded because of the difficulty in meeting the two criteria for input/output systems, low overhead and hardware simplicity, as discussed in Chapter 3. The second criterion, hardware simplicity, is violated by the fact that each input/output device must have special hardware, perhaps an associative memory, for dealing with segmentation. Since this argues strongly for input/output devices dispensing with segmented

addressing, and referencing memory by location, it is necessary to recall from Chapter 3 that this plan would require supervisory intervention for each input/output operation, and hence violate the first criterion for input/output, low overhead.

The first criterion, furthermore, is violated by the need for duplicating entries in the associative memories of the arithmetic processor and an input/output device, if both are used by the same process. In particular, since processors in MPAMA use the same associative memory, when a process is transferred from one processor to another, memory cycles are not necessarily spent in reloading the associative memory. Further details of how MAPMA meets both criteria of Chapter 3, are shown in Appendices III and IV.

It is acknowledged that the ideas in this paper are not a complete solution to the problems of input/output. There are those who will say that these proposals are too extreme, and that the current method of program interrupt is sufficient; there are those who will say that these proposals are too conservative, that in a few years, processors will consist of small elements, with highly complex interconnections, so that getting a tape drive to operate and performing a floating point addition will be equally simple. It is recommended that the former study the arguments in this paper, and question their validity, and that the later study this paper as a starting point.

If this paper is taken as a starting point, then several projects should be undertaken as soon as possible. First, all three systems should be extensively simulated, so that any non-obvious properties of the system will be made apparent. Second, a detailed design and cost analysis of associative memories of the conventional type, and of the proposed multiport type, should be undertaken both for its own sake, and for the sake of comparing the actual cost of the three systems. Finally, as many computation centers as possible should collect statistics on the exact instruction mixes actually being run, so that more exact planning for optimizing response will be possible.

It is impossible to determine the future. If this paper in any way affects that future for the better, that is a sufficient justification for its being written. The only possible strategy is to wait and see.

APPENDIX 1. NOTATION.

Let α be a memory word.

Then α contains an address portion. (i.e. a section of the word which is normally interpreted by the hardware as an address.)

Then PN (α) represents the page number of the address

ST (α) represents the segment tag

LN (α) represents the line number

A (α) represents the address

e.g., α = $\underbrace{625113\ 6\ 73\ 2551}_{\text{address portion}}$

LN (α) = 2551

PN (α) = 73

ST (α) = 6

Furthermore, if there are λ bits used to indicate the line number,

π bits used to indicate the page number,

and σ bits used to indicate the segment tag,

then the entire address A(α) is only determined insofar as the contents of attachment register ST(α) are known. At any instant, however, there are exactly 3 possibilities. Either 1) A(α) is an invalid address, in the sense that attachment register ST(α) has never been loaded, or 2) A(α) corresponds to some location L(α), in main memory. In this case, $ST(\alpha)\cdot 2^{\pi} + PN(\alpha)$ corresponds by some mapping to BN(α), and A(α) corresponds to $L(\alpha) = BN(\alpha)\cdot 2^{\lambda} + LN(\alpha)$, or 3) A($\alpha$) corresponds to some location in secondary storage.

When addresses are used in examples, they will be written as [A, i] where A is an attachment tag, and i is an address without an attachment tag. That is, i is $\lambda + \pi$ bits long, and $i = LN([A, i]) + PN([A, i]) \cdot 2^{\lambda}$.

APPENDIX 2. Flow charts for the addressing schemes of AMP, AMAS, and MPAMA

The following flow charts describe the way in which the three classes of systems address main memory to produce, for process p, the word corresponding to attachment tag at, page number pn, and line number ln. The block number corresponding to the attachment tag, page number pair (at, pn) is bn, and the location of the desired word is $bn \cdot 2^{\lambda}+ln$ (see appendix 1). It is assumed that the first element of the state word of process p is located in the first word of block sw; that location $sw \cdot 2^{\lambda}+k_1$ contains the block number for the first block of the capability directory; and that location $sw \cdot 2^{\lambda}+k_2+n$ gives the directory index for attachment register n (see figure 2.5)

In all three systems, the associative memory is probed, and if an entry is found corresponding to (at, pn), the value returned by the associative memory is the appropriate block number. If no such entry is found, the associative memory is probed to find the page table entry for segment tag at, and, if found, the value returned is the block number for the first word of the page table; this block number and the page number are used to get the block number of page pn for segment tag at; this information is loaded into the associative memory, and the cycle of probing the associative memory is begun again. If neither a segment tag, page number entry nor a page table entry is found, the location of the capability directory and the directory index for attachment tag at are found in the state word, and, from these, the block number of the first word in the page table for attachment tag at is found; the corresponding entry is made in the associative memory.

It is assumed that the associative memory has a hardware algorithm for determining which entry should be destroyed when a new entry is made.

The words of the associative memory are divided into fields, F1 through F4 for AMP, and F0 through F4 for AMAS and MPAMA. F0 contains the process identification, which is not needed in AMP; F1 contains the segment tag; F2 contains the page number for segment tag, page number entries and zero for page table entries; F3 contains a flag which is zero for segment tag, page number entries and one for page table entries; F4 contains the corresponding block number for segment tag, page number entries and the block number for the first word of the corresponding page table for page table entries (page tables are assumed to be no longer than one block). The following notation is used in the flow charts:

AMT(M0, M1, M2, M3, M4) is a boolean function which is true if associative memory M4 contains an entry with F0=M0, F1=M1, F2=M2, and F3=M3. For AMP, the first and last arguments are omitted; for MPAMA, the last argument is omitted.

AM(M0,M1,M2, M3,M4) is a function which gives the value of F4 for the cell in associative memory M4 with F0=M0, F1=M1, F2=M2, and F3=M3. For AMP, the first and last arguments are omitted; for MPAMA, the last argument is omitted.

MEM(B,L) is a function which gives the value of the L^{th} word of block B of main memory.

WRITAM(M0,M1,M2,M3,M4,M5) writes an entry into associative memory M5 with F0=M0,F1=M1,F2=M2,F3=M3, and F4=M4. For AMP, the first and last arguments are omitted.

ARB(LOC) is a function, for AMAS and MPAMA only, which has as its value the number of the arbiter which controls location LOC.

AMQ(M0,M1,M2,M3) is a function, for MPAMA only, with value equal to the number of the arbiter for which the associative memory contains an entry with F0=M0, F1=M1, and F2=M2, and F3=M3.

t1,t2,t3, and j are temporary storage registers.

word is the register where the contents of the desired address are finally placed.

Figure A2.1 is the flow chart for AMP; Figure A2.2 is the flow chart for AMAS; and Figure A2.3 is the flow chart for MPAMA.

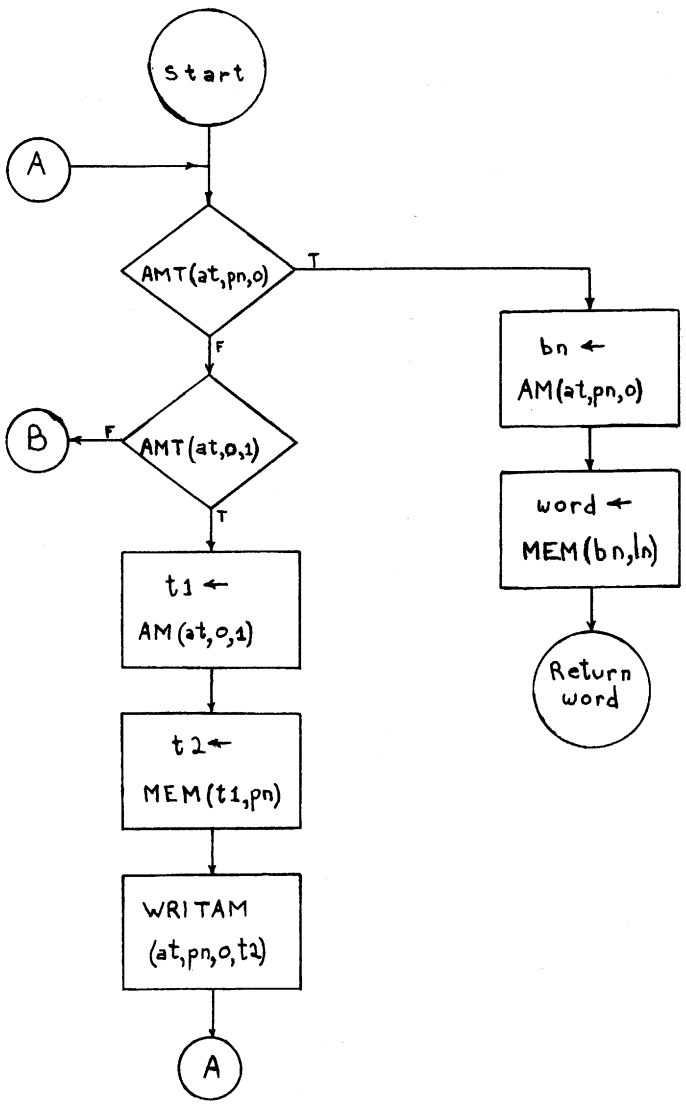


Figure A2.1. Flow chart for the addressing scheme of AMP

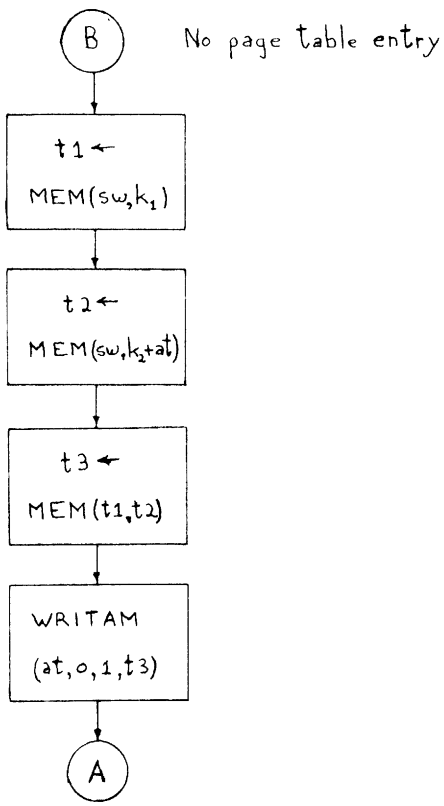


Figure A2.1 (con't)

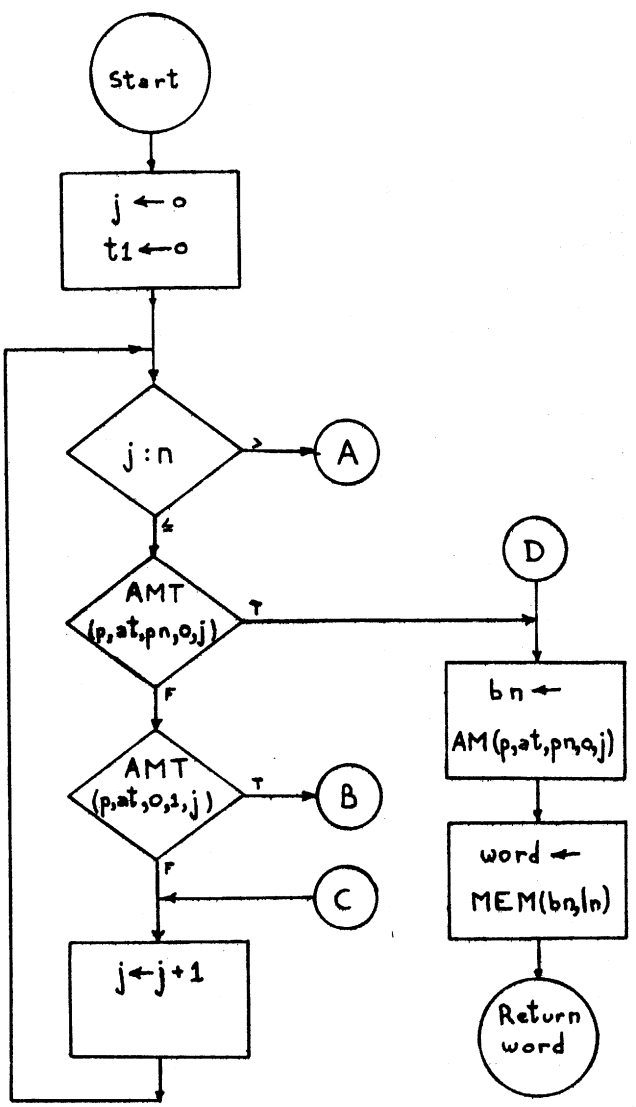


Figure A2.2. Flow chart for the addressing scheme of AMAS

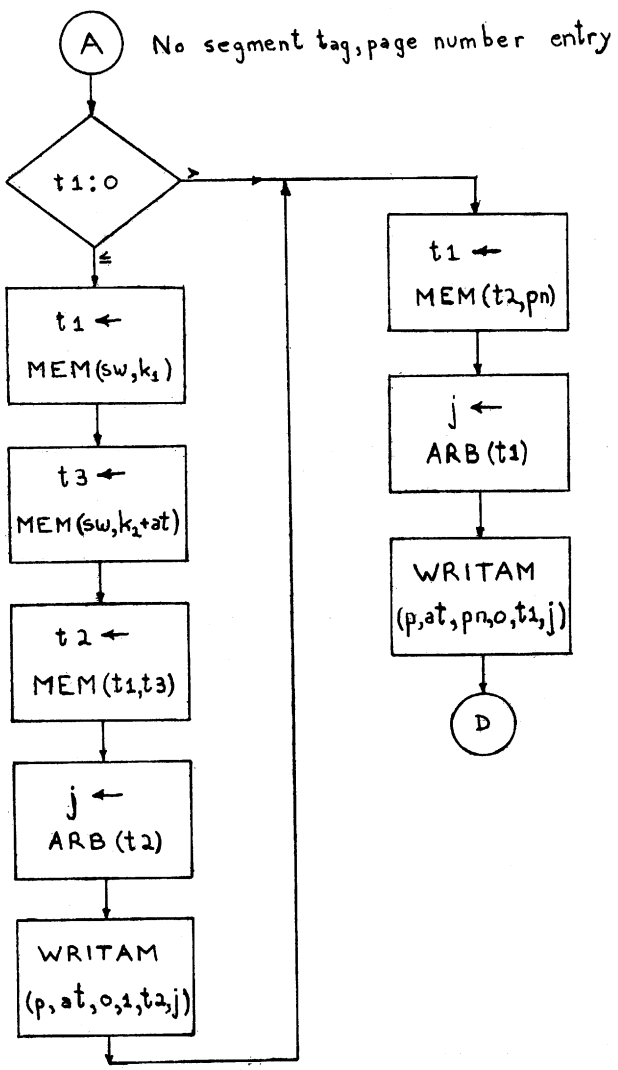


Figure A2.2 (con't)

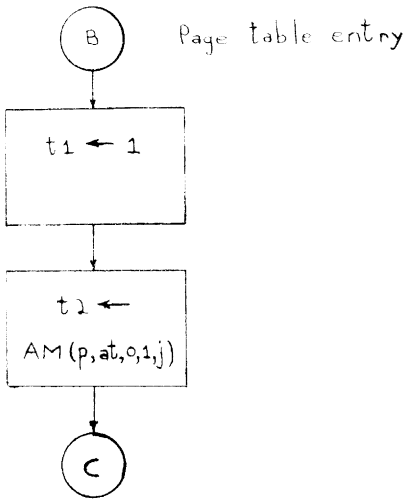


Figure A2.2 (con't)

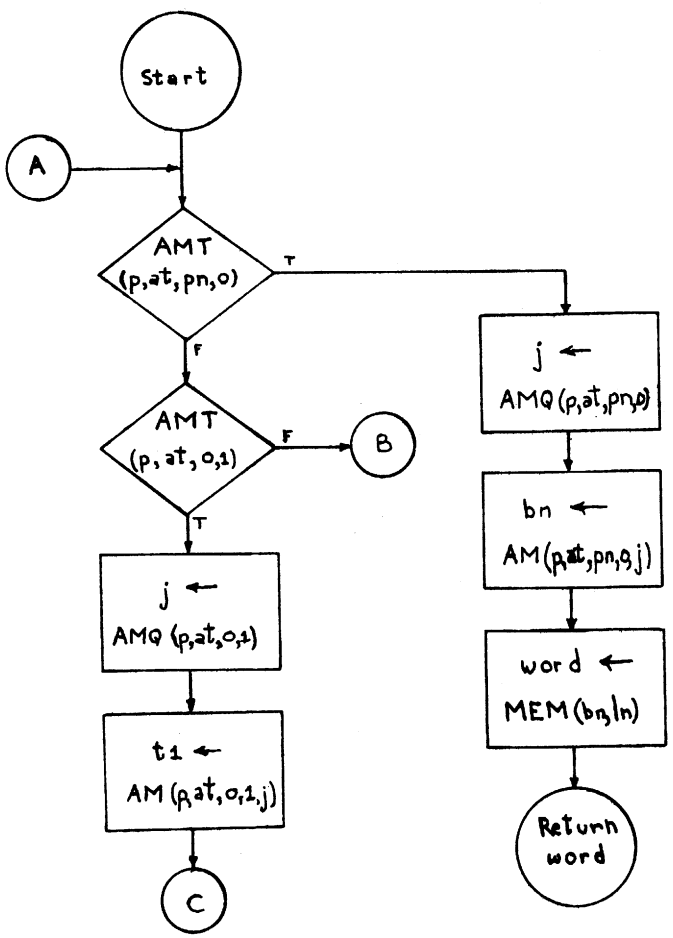


Figure A2.3. Flow chart for the addressing scheme of MPAMA

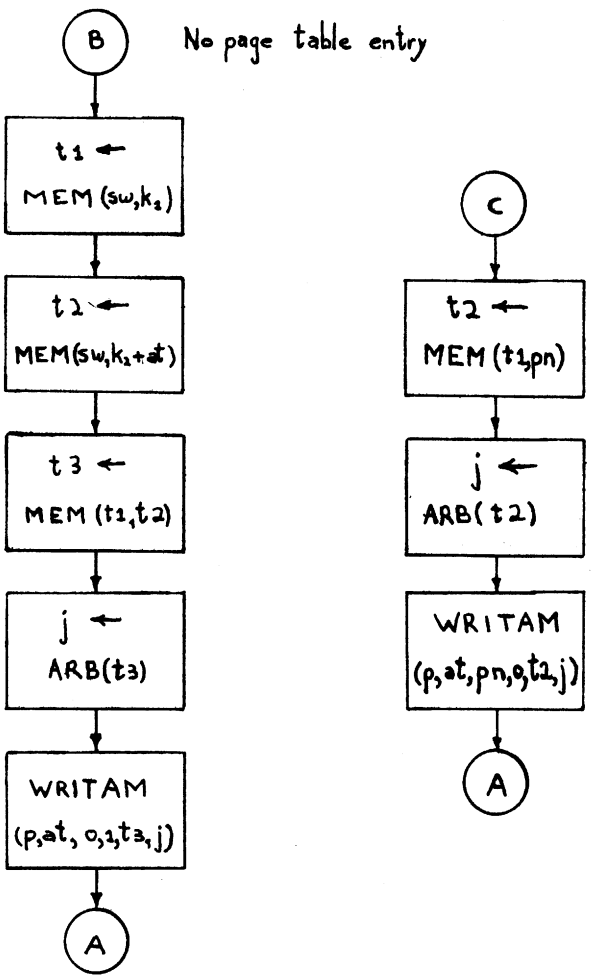


Figure A2.3 (con't)

APPENDIX 3. A CONFIGURATION FOR MINIMUM RESPONSE TIME

As discussed in Chapter 6, having the associative memory in the arbiter permits the saving of some time when a process changes from one processor to another. This appendix shows one possible system, not the ideal one, for obtaining this saving. This system preempts one entire processor for the control of a single program. It need not be a particularly expensive processor, and, considering the aim of the second criterion, it is not expected to be. The exact program to be described monitors incoming analog data, converts it to digital form to be stored for reference, and also activates the arithmetic processor whenever the input exceeds a certain level. The arithmetic processor then classifies the level of the input into one of three categories, and increments a counter corresponding to the category. Such a program might be used, for example, in monitoring a biological experiment, with appropriate warnings being printed when the counters overflow.

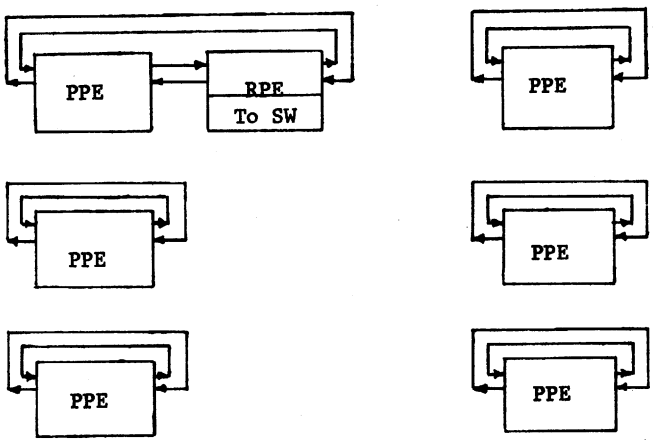
It is assumed that the analog-digital converter has an internal clock which determines the sampling rate, and an internal register for holding the critical level for activating the arithmetic processor. This register and the clock are set by certain instructions which are assumed to have been given. During normal operation, the base address for storing the digitized data is determined by the instruction, SAMPLE.

The exact location for the data is formed by modifying the SAMPLE address by index register 4. After a sample is stored, index register 4 is incremented and index register 5 is decremented. When index register 5 reaches zero, or when the critical level is exceeded, the instruction following the SAMPLE is executed; this will normally be a GP instruction.

Further, it is assumed that the arithmetic processor is named in attachment register 1, that the analog-digital converter is named in attachment register 2, and that the data segment is named in attachment register 3. The pseudo-7094 coding follows; and Figure A3.1 shows the structure of the scheduling rings during execution.

```
START       STZ     [3,1]       zero counters
            STZ     [3,2]       "       "
            STZ     [3,3]       "       "
            PAX     0,4         zero XR4
            PAX     5000,5      set maximum number of samples to 5000
LOOP        GP      2
            SAMPLE  [3,100]
LOOP 2      GP      1
            CLA     [3,100],4   Note that no reloading of the associative
                                memory is needed, since the converter
                                referenced this location during the last
                                data cycle.
            SUB     [3,5]       perform classification
```

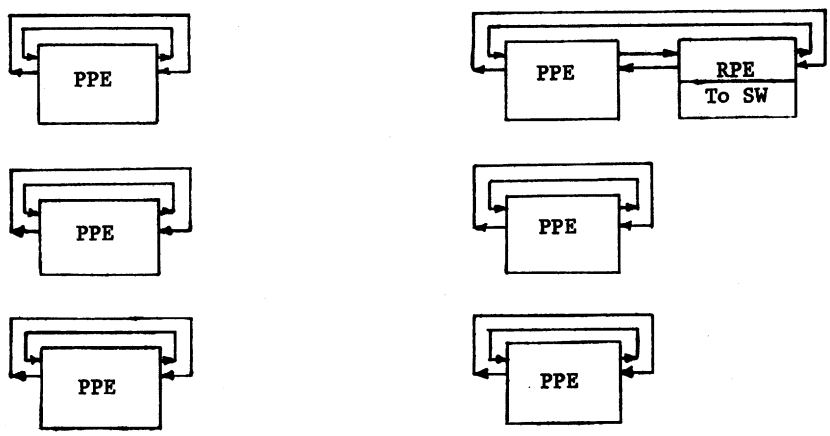
```
TPL     *+5
CLA     =1
ADD     [3,1]
STD     [3,1]
TRA     LOOP
SUB     [3,6]
TPL     *+5
CLA     =1
ADD     [3,2]
STO     [3,2]
TRA     LOOP
CLA     =1
ADD     [3,3]
STO     [3,3]
TRA     LOOP        the end
```



Analog-digital converter ring structure

Arithmetic processor ring structure

Just after execution of LOOP



Analog-digital converter ring structure

Arithmetic processor ring structure

Just after execution of LOOP2

Figure A3.1. Ring structures for the sample program.

Note that much system capacity is being wasted in order to obtain fast response and simplicity of input/output.

APPENDIX 4. MINIMUM EQUIPMENT INPUT/OUTPUT DEVICES

The second criterion specified for the input/output subsystem was that simple devices may be connected to it. An example of just how minimal this arrangement may be is given in this appendix.

A paper tape punch will be used to illustrate the simplicity of which an MPAMA system is capable. The procedure for operating the punch is to load the arithmetic processor's accumulator with the word to be punched [in the low order eight bits for example], and then give a GP instruction. This causes the state word of the process, and, in particular, the accumulator, to be stored by the arbiter and permits the arithmetic processor to be used for other processes. The arbiter places a Ready Process Entry into the ring structure for the punch, and,since it is assumed that there is only one process using the punch, the arbiter directs the punch to perform a process swap in order to accept the new, high priority process entry. This means simply that one control line ["Process Swap"] to the punch was turned on. Since the punch was not executing any other process, the first two steps of the process swap are ignored, and the punch immediately turns on a control line ["Process Accept"] to the arbiter telling it to remove a Ready Process Entry from the queue and begin tranferring the corresponding state word to the punch. The punch would then put

codes corresponding to the elements of the state word on the address lines, and turn on a "Data Request" control line, indicating to the arbiter that it should transmit the designated element on the data lines. In particular, the punch uses a code designating the accumulator. The arbiter examines the code, places the appropriate part of the state word [the accumulator] on the data lines and turns on a "Data Available" control line. The punch, detecting the Data Available line on, places the data in a buffer, starts the physical punching, and turns off all its control lines to the arbiter.

When the physical punching is completed, the punch simply puts an attachment tag number on the address lines, and turns on the GP control line to the arbiter. This causes the arbiter to place the Ready Process Entry back into the ring structure for the arithmetic processor and turn on the process swap control line, to store the state word for the process currently being executed by punch. Since the state word hasn't changed, the punch turns on the process accept signal, telling the arbiter that it is ready to accept a new state word. Since there isn't any other state word, this completes the procedure until the process wants to punch another character.

The number of lines connecting the punch with the arbiter then, is not impossibly large, and in fact the punch described is not much more complex than that for some existing systems. The lines, and the sequence of events occuring on them are now summarized.

Lines for communication between punch and arbiter:

Data lines (for this case, only 8 are needed)

Address lines (again, for this case not all are needed)

Control lines:

Process Swap

Process Accept

Data Request

Data Available

Get Processor

Sequence of signals between arbiter and punch (refer to Figure A4.1.)

(1) Process Swap turned on by arbiter

(2) Process Accept turned on as a response by punch

(3) Code for Accumlator placed on address lines, and Data Request turned on by punch

(4) Aocumulator contents placed on data lines, and Data Available turned on by arbiter. Punching may begin.

(5) Data Request turned off by punch in response to (4)

(6) Data Available turned off by arbiter in response to (5)

(7) Process Accept turned off by punch

(8) Process Swap turned off by arbiter in response to (7)

(9) Attachment register number placed on address lines, and Get Processor turned on by punch

(10) Process Swap turned on by arbiter in response to (9)

(11) Process Accept turned on by punch

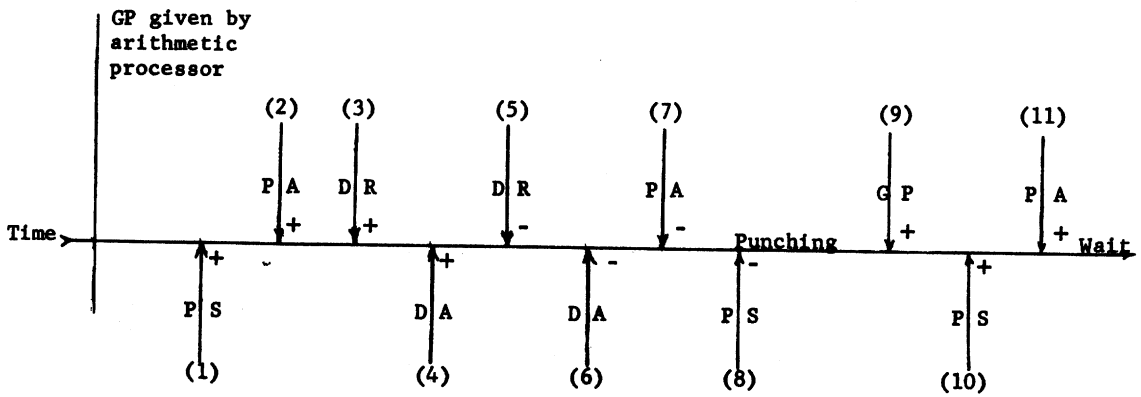


Figure A4.1. Sequence of signals between arbiter and punch

+ = Signal turned on

- = Signal turned off

BIBLIOGRAPHY

1. Buchholz, W., (Ed.), Planning a Computer System-Project Stretch, McGraw-Hill, New York, 1962.

2. Codd, E. F., Multiprogramming. Advances in Computers, Vol. 3, 1962, pp 77-153.

3. Codd, E. F.; E. S. Lowry; E. McDonough; C. A. Scalzi, Multiprogramming stretch: Feasibility considerations, Comm. of the ACM, Vol. 2, Nov. 1959, pp 13-17.

4. Conway, Melvin E., A multiprocessor system design, AFIPS Conference Proceedings, Vol. 24, 1963, pp 139-146.

5. Corbato', F. J., and V. A. Vyssotsky, Introduction and overview of the multics system, AFIPS Conference Proceedings, Vol. 27, 1965, pp 185-196.

6. Critchlow, A. J., Generalized multiprocessing and multiprogramming systems, AFIPS Conference Proceedings, Vol. 24, 1963, pp 107-126.

7. Daley, R. C., and P. G. Neumann, A general purpose file system for secondary storage, AFIPS Conference Proceedings, Vol. 27, pp 213-229.

8. Dennis, J. B., Program structure in a multi-access computer, Project MAC Technical Report, MAC-TR-11.

9. Dennis, J. B., Automatic scheduling of priority processes, Project MAC memorandum MAC-M-187, October, 1964.

10. Dennis, J. B., An example of intersphere communication and asynchronous parallel processing - typewriter console message handling by protected service routines, Project MAC memorandum MAC-M-189, September, 1964.

11. Dennis, J. B. , Segmentation and the design of multiprogrammed computer systems, Journal of the ACM, Vol. 12, No. 4, October, 1965, pp 589-602.

12. Dennis, J. B., and E. L. Glaser, The structure of on-line information processing systems, Project MAC memorandum, MAC-M-181, October, 1964.

13. Gill, S. Parallel programming, The Computer Journal, Vol. 1, No.1, April, 1958, pp 7-10.

14. Glaser, E. L.; Couleur; J. F., and G. A. Oliver, System design of a computer for time-sharing applications, AFIPS Conference Proceedings Vol. 27, pp 197-202.

15. Iliffe, J. K. The role of addressing in programming systems, Introduction to Systems Programming, Peter Wegner, Ed. Academic Press, New York, 1964, pp 256-275.

16. Kilburn, T.; Edwards, D. B. G.; Lanigan, M. J.; and F. H. Sumner, One level storage system, IRE Trans. on Electronic Computers, Vol. EC-11, No. 2, April 1962.

17. M.I.T. Computation Center, The Compatible Time-Sharing System: A Programmer's Guide, M.I.T. Press, Cambridge, Mass., 1963.

18. Ossanna, J. F.; Mikus, L. E., and S. D. Dunten, Communications and input/output switching in a multiplex computing system, AFIPS Conference Proceedings, Vol. 27, 1965, pp 231-241.

19. Scherr, A. L., A system for multiprocessing, I.B.M. Technical Report TR 00.900, October, 1962.

20. Vyssotsky, V. A.; Corbató, F. J.; and R. M. Graham, Structure of of the multics supervisor, AFIPS Conference Proceedings, Vol. 27, pp 203-212.

21. Witsenhausen, H., A note on asynchronous parallel processing, Project MAC memorandum MAC-M-163, July, 1964.

Also, manufacturer's descriptive literature and programming manuals for the following systems: IBM 704, IBM 7090, DEC PDP-1, DEC PDP-6, UNIVAC 1107, Burroughs D-825.

This empty page was substituted for a blank page in the original document.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-28

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 74 (81 - IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOW PAGE 73

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

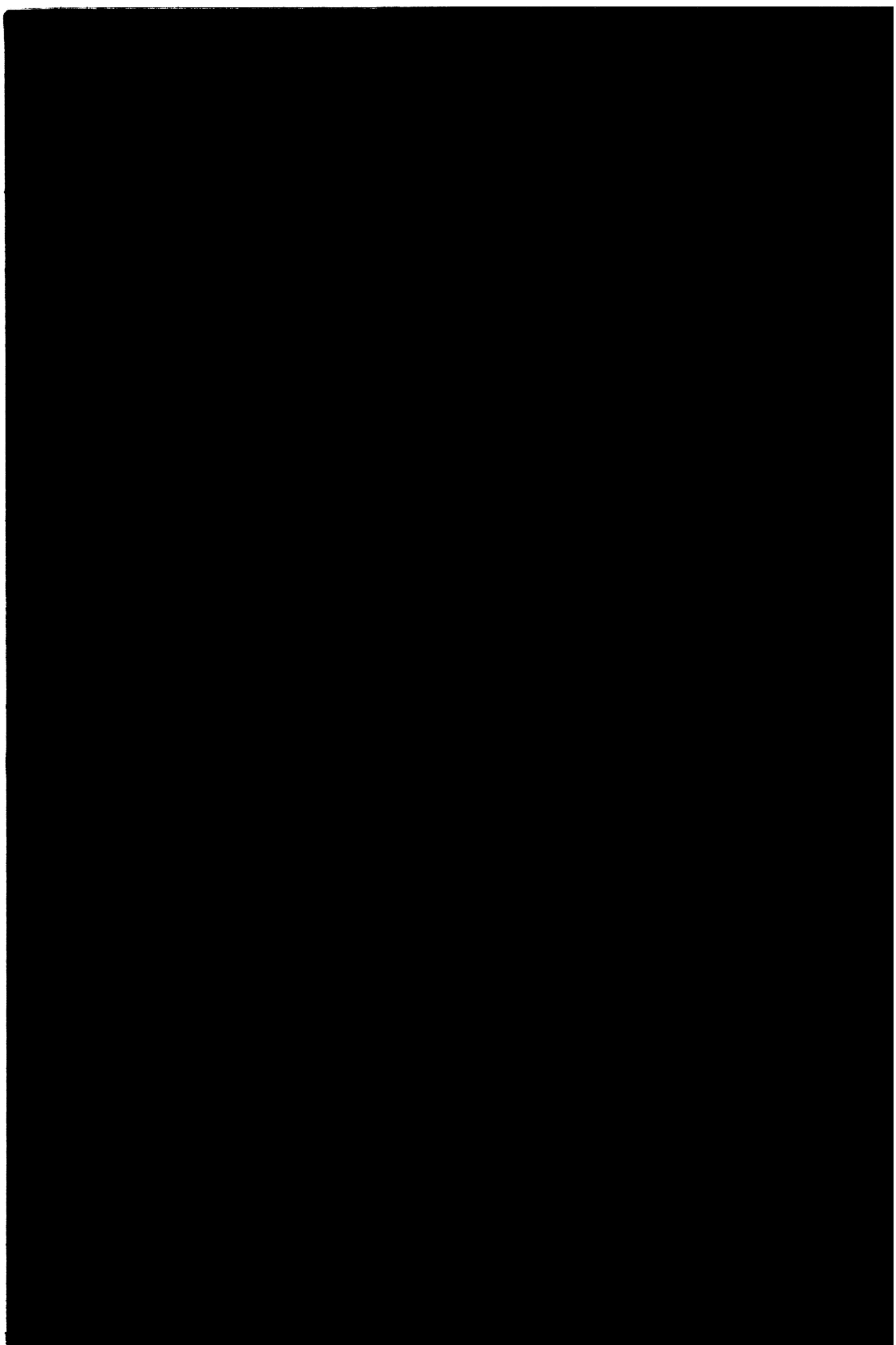
Description : Page Number:

IMAGE MAP: (1-74) UNM'D TITLE, 2-73, UNM'D BLK
(75-81) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)

Scanning Agent Signoff:

Date Received: 12/11/95 Date Scanned: 1/18/96 Date Returned: 1/19/96

Scanning Agent Signature: Michael W. Cook



Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 1/18/1996
M.I.T. Libraries
Document Services