1983

Microprocessor based modular support for an operating system

Ahmed Amin Elamawy
Iowa State University

Elamawy, Ahmed Amin

MICROPROCESSOR BASED MODULAR SUPPORT FOR AN OPERATING SYSTEM

Iowa State University PH.D. 1983

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages ✓
2. Colored illustrations, paper or print _____
3. Photographs with dark background ✓
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages _____
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Other _____

Microprocessor based modular support for an operating system

by

Ahmed Amin Elamawy

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Electrical Engineering
Major: Electrical Engineering (Computer Engineering)

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1983

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I. INTRODUCTION

It was estimated in 1972, that every 5% more facility put into a computer system cost 20% more to achieve, which represented a rapidly rising cost curve [1]. This was mainly due to the need for a more complex operating system.

The complexity of an operating system designed for a multiprocessor system is usually greater than that designed for a single processor system [2]. In the general-purpose multi-arithmetic logical unit configuration, the difficulty is mainly in the implementation of an integrated control within the operating system. For example, synchronization, task splitting, and scheduling are areas where the presence of more than one processing unit increases the supervisor's complexity.

As is well-known, most of the computer system's cost goes to the software design, especially the operating system. A strong, fast-rising relationship between the complexity of the operating system and the total system cost mandates the need for a more efficient utilization of current technology to support the operating system. It is important to notice that a more complex operating system not only means a higher system cost, but also means a larger percentage of the processing power devoted to executing the operating system code. Thus, the existence of powerful, fast, inexpensive microprocessors makes it worthwhile to study the possibility of utilizing current technology to support the execution of some operating system functions. This is especially attractive in the case of a modular, structured, operating system because a high

degree of parallel operating system processing should be achievable.

Extensive studies have shown that a computer system formed by interconnecting many small micro or miniprocessors achieves a better cost/performance ratio and higher reliability than a powerful, large, and complicated single processor system [3, 4, 5]. This concept is appealing, not only at the computer system level, but possibly at the operating system level. If we can look at the operating system by itself as a system, we may think about the possibility of multiprocessing some of its functions, especially those that lend themselves to parallel processing. Multiprocessing of operating system functions using inexpensive VLSI chips is the subject of this dissertation. To prove the point, a submodule for the exact implementation of the least recently used replacement policy in a demand paging system was designed, built, and tested. This work is described in the various chapters of the dissertation as noted below.

Chapter II has been devoted to reviewing some related literature. In Chapter III, an approach for supporting an operating system has been introduced. In Chapter IV, a specific example of a support module for a deadlock avoidance scheme has been described along with its related literature review. Chapter V contains a detailed description of the design of a module to implement the exact least recently used replacement policy. Also in Chapter V, the description of a submodule that was designed, built, and tested is given along with the least recently used routine, hardware circuit, and test circuit details. The data obtained

from testing the submodule along with some remarks and comments are given in Chapter VI. Chapter VII contains the conclusion.

CHAPTER II. LITERATURE REVIEW

The amount of literature dealing with the subject of microprocessor based support for an operating system is very limited. However, some publications can be related to this subject in a broad sense. These will be reviewed in this chapter. Since Chapters IV and V will provide specific application examples, it is more appropriate to review their related literature in those chapters.

A study in 1972 provided an example of operating system measurements and indicated the need for better hardware assistance in monitoring and adjusting the operating system performance [6]. Afterwards, multiprocessing systems and their operating systems were subjected to extensive research. It was found that multiprocessing systems achieved a better cost/performance ratio and better reliability than single processor systems [3, 4, 5]. One study used analytical and numerical techniques to compare job turnaround time and throughput rate of three multiprocessor system models with that of a single central processing model of equal processing rate [4]. The results indicated that multiple slow processors may sometimes be used to replace a fast central processor without significant performance degradation. The investigator concluded that this would be increasingly attractive as the cost of microprocessors continued to decrease. An interesting paper published in 1977 discussed different multiprocessor systems and envisioned two main types of control in multiple instruction multiple data (MIMD) architecture [2]. The first was fixed mode, in which one or more processors were dedicated to execute the operating system. When some

other processor terminated its task, or when all other processors were busy and a higher priority task had to be initiated, it was the responsibility of the dedicated processor(s) to schedule, terminate, and/or initiate processes. An advantage of such a scheme is that a special purpose hardware can be embedded in the design, hence decreasing the executive's overhead. The other type of control was the floating control mode. In this mode, each processor could have access to the operating system and could schedule itself. This mode had a reliability advantage over the fixed mode. The investigator concluded that despite the decreasing cost of hardware due to large scale integration, the increased complexity in communication and the overhead in the operating system should be taken seriously when thinking about distributed function systems. The investigator also suggested that the challenging problems in the design of coherent architectures of viable and efficient operating systems, and in the inclusion of evaluating tools both during the design process and in the completed system itself, would restrict for some time the range of useful systems.

Although it has become an established fact for many applications that multiprocessor systems are superior to single processor systems in terms of the cost/performance ratio despite the increased operating system complexity, it is not clear whether a homogeneous architecture is better to adopt. Apparently, homogeneous systems have some advantages in terms of reliability and design simplicity [7], whereas heterogeneous architectures have the merits of flexibility and performance improvement with appropriate load sharing [8].

A very interesting system with strong relationship between hardware architecture and the operating system architecture was described in 1977 [3, 9]. The system was called Poly-Processor System (PPS). The system was developed for time sharing services and consisted of a processor subsystem, a memory subsystem, and a connection subsystem. The processor subsystem consisted of a number of functionally specialized processors, which covered six functional classes. The set of functions for each processor class corresponded to the partitioning of conventional operating system functions. Furthermore, the functions of each processor class were divided into functionally specialized subclasses and each of six processor classes consisted of many subprocessors or modules. The memory subsystem consisted of six memory classes, which were categorized according to the behavior and characteristics of stored information. These classes were introduced to add changeability to the functions of processors, to prevent errors from spreading, and to reduce the memory access conflicts. Since the reliability of the system was affected by rigidly assigning the functions to the processor, a dynamic microprogramming technique was used to move a process from a failed processor to a processor that had at least as much connection as the failed processor.

The most important issue in the design of such a multiprocessor system as the PPS was to devise a connection subsystem between processors and memory modules that would be effective for highly parallel and closely cooperative processing. In order to achieve parallel processing, information, i.e. programs and data used in the system, was divided into

three categories: private information, command data, and shared data. Private information was stored in a memory provided exclusively for each processor. Command data which were used to initiate a program in another processor such as requests, inquiries, and answers, were transferred directly between processors. Shared data were stored in a memory shared by several processors. For command data and shared data, two different connection modules were provided; the interprocessor connection module used a common bus technique, and the processor-memory connection module used a crossbar switch technique.

Extensive studies of the PPS system pointed out the validity of relating the hardware architecture to the operating system structure [3, 9]. However, the system suffered some drawbacks which were noted by the authors who described and studied the system's performance. These drawbacks can be summarized as follows:

(1) The system was inflexible since it was hard to modify and expand.

(2) The system was tailored to fit a time-sharing service giving no potential for applicability in other system environments.

(3) Reliability was relatively limited by the small number of processors connected to main memory.

(4) System cost increased largely because of implementing the interprocessor class.

(5) System performance was degraded by the command data transmission overhead.

However, the PPS-related studies certainly established a good

background in searching for other approaches.

It is appropriate to conclude the literature review by remarking that there should be ways to support and multiprocess operating system functions regardless of the hardware architecture. It should be possible to apply many concepts whether the supported system is a single or multiprocessor system.

Additional literature will be reviewed in Chapters IV and V when specific operating system support examples are to be introduced.

CHAPTER III. MODULAR MULTI-MICROPROCESSOR BASED SUPPORT FOR AN OPERATING SYSTEM

On the average, 10-30% of a computer system's processing time is spent in executing operating system-related activities [6]. This not only represents an overhead in terms of central processing unit time, but can also be viewed as a load on other system resources such as main memory and shared buses, thus limiting the system performance. It can be said that in some way it is wasteful to execute some operating system functions on a main processor or processors. The data provided on the PPS system performance indicated that, on the average, 12 bytes of command data passed to an operating system module every 250 accesses by another processor [3]. The command data were used to initiate a program in another processor. It consisted of command code codes and parameter words which specified the program execution details. This was the case when only six modules were incorporated. If the number of modules has been increased such that every module became responsible for only one function, one would have expected a transaction size to be less than 12 bytes (no need for command code). Moreover, the inter-processor communication period could have been much longer.

An approach that might provide better cost/performance ratio will now be described. The approach is general enough so that it is applicable to different system architectures. The idea is to use as many support modules for the operating system as needed. Each module contains one or more microprocessors. The operating system may be viewed as composed of two parts: (1) A software part residing in main memory; and

(2) A microprocessor-based modular support part.

Communication between the two parts is carried out through dedicated, relatively small, reserved areas of main memory space. This is shown in Figure 1. The communication area of a module is also a part of the module's memory space and is accessible by both the main system's processor(s) and the module's microprocessor(s). The rest of the module's memory is private to the module and is only accessible by some or all the microprocessors (if more than one microprocessor is utilized in the module). This kind of architecture makes the idea applicable to a wide spectrum of system architectures. This occurs because common memory accessible by the main system processor(s) almost always exists in tightly coupled multiprocessor systems and certainly in single processor systems. Consider, for example, a single bus homogeneous multiprocessor system supported by a modular heterogeneous multimicroprocessor system as shown in Figure 2. Different aspects regarding the design and performance of such a system can now be pointed out.

Support Modules

Each support module consists basically of:

(1) One or more microprocessors;

(2) A private memory; and

(3) A communication memory accessible from the main system bus and a dedicated bus connected to the microprocessor(s).

The private memory is generally larger than the communication area and can be slower. The former need only match the microprocessor's speed, while the communication memory has to be fast enough to match main

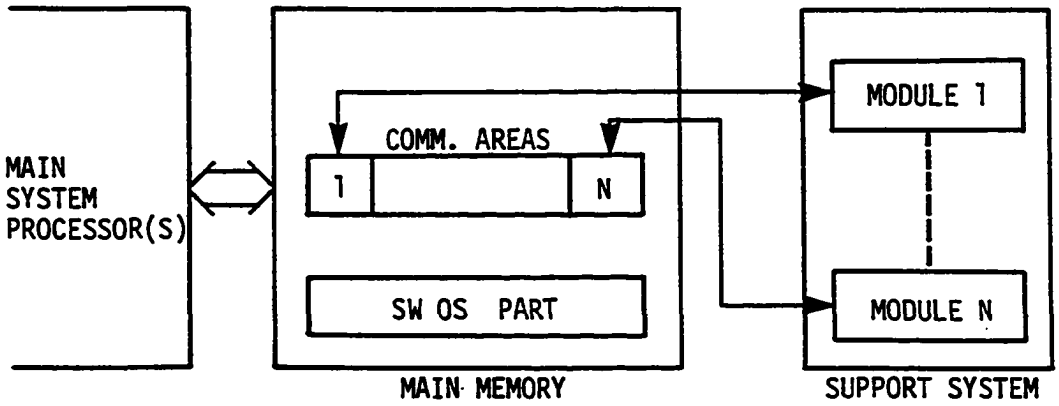


Figure 1. Communication technique between the support system and the main system

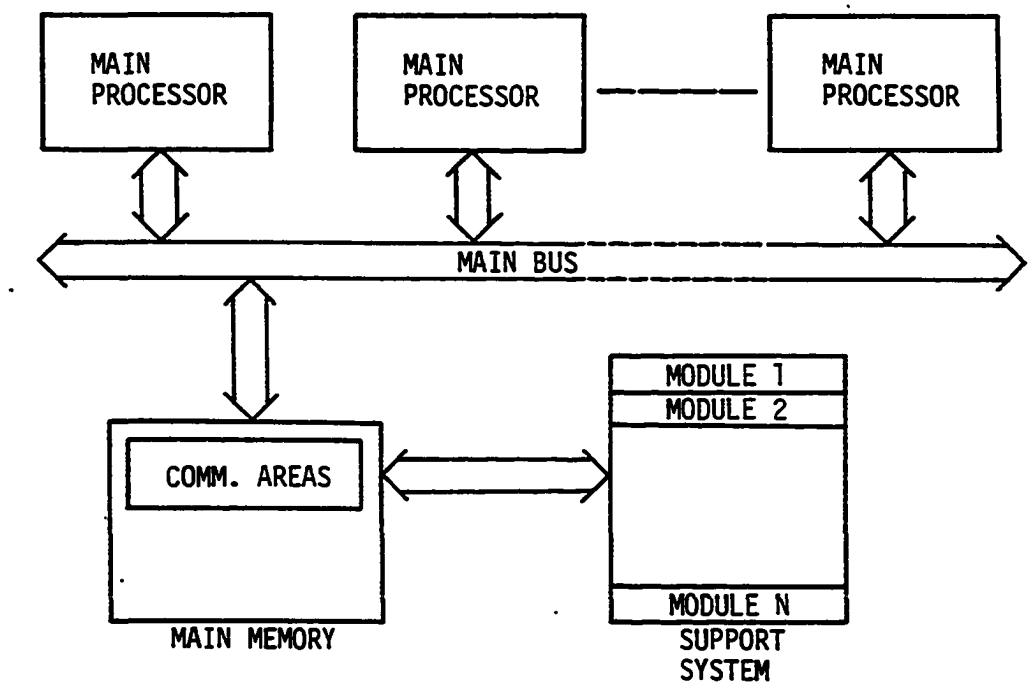


Figure 2. A homogeneous main system supported by a heterogeneous modular support system

system's processor(s) speed. Some advantages of such an organization are:

(1) Memory space, as seen by the main system, is much less than the actual memory space used by the module to process a function.

(2) Slower, hence cheaper, private memory is used to execute operating system functions. However, the overall system speed may improve because functions will be executed in parallel.

(3) Module demand on main system resources is minimum, leaving more resources such as central processing unit time, main bus, memory space, etc., for productive work.

Different kinds of microprocessors may be used in different modules. The selection of a certain kind for a module should be dependent on the function to be performed, as well as the characteristics of the microprocessor. The number of micros in a certain module should depend on the frequency the module is invoked, as well as the nature and length of the function. Some modules might have to perform jobs like monitoring functions or performance measurement. Such modules would have to submit a report to the main system which uses the report either to dynamically adjust some operating system parameters, or to help some operating system functions such as the implementation of the exact least recently used replacement policy in a demand paging system. In such cases, the number microprocessors would depend on the arrival or event rate as well as the code execution time of a microprocessor's routine.

Reliability

Because of the specialized nature of the modules, it might seem that there is a reliability problem. However, many techniques to improve the reliability of the support system are available; for instance, redundancy within a module, or redundancy at the module level, may be implemented at some insignificant cost. Besides, an extra microprocessor per module can be employed to perform supervisory and status checking functions of the other elements. In case of a microprocessor failure, the supervisor micro may take over its job and inform the main system about the problem. Redundancy at the module's level might also be feasible because of the low cost of microprocessors and of hardware in general.

Another interesting idea that could enhance both reliability and flexibility is to use a pool of microprocessors which are assigned functions dynamically. This idea was implemented in designing a file storage/retrieval system by Trans-A-File Co. [10]. The system used a set of microprogrammed microprocessors to perform a wide variety of tasks that were dynamically allocated to it. The control programs were stored on a tape and transferred upon allocation of a task to a microprocessor's memory. The trade-off in the design was mainly the response time required to reconfigure the system. However, in our case the idea may be utilized in the suggested support system to assign the function of a failed module to a stand-by module.

An idea that can result in a very reliable system is to store all module function codes in secondary memory. In case of module's failure,

its corresponding code is transferred to the main memory and is temporarily executed by the main system processor(s) without support.

Cost

With the constant decline in the cost of powerful microprocessors and hardware in general, it is feasible to use the brute-force approach which employs a large number of microprocessors to perform some function. For example, the author designed and built a part of a module (submodule) for implementing the exact least recently used replacement policy in a demand paging memory management system with a hardware cost around $300. Two MC68000 microprocessors at $104 each were employed for both testing and implementing the desired function. If a whole module is to be built of eight submodules, only nine microprocessors would be needed and the total cost of hardware should be less than $3,000. This figure is considered very small compared to the total cost of a multiprogramming computer system, and is nearly negligible.

Other Aspects

Some of the operating system functions may not lend themselves to parallel processing or to the idea of support. However, many functions do lend themselves very well to parallel or support processing. These include housekeeping work, scheduling, and monitoring functions. Some examples are sorting and maintaining lists, priority updating, deadlock avoidance-detection schemes, memory replacement algorithms, preparing compaction addresses in segmentation systems as well as finding enough space for incoming segments, job dispatching in multiprocessor systems,

and dynamic bus allocation in multi-bus architectures.

In the next chapter, we will give a specific example of a support module for a deadlock avoidance scheme. In Chapter V, the description of a module for exact implementation of the least recently used algorithm will be discussed, along with the detailed design of a submodule that was actually built and tested.

CHAPTER IV. SUPPORT MODULE FOR A DEADLOCK AVOIDANCE SCHEME

Introduction

Having introduced a general approach to support an operating system using multi-microprocessor based modules, we are now ready to give the first of two specific examples. The second will be given in the next chapter.

Before introducing our suggested support module for deadlock avoidance schemes, it is appropriate to review briefly some related literature.

One of the operating system functions is to allocate system resources to competing processes. The allocation scheme is usually designed to take care of the possibility of deadlocks. One way a deadlock occurs is when a process holding some resource has to wait for a resource held by another process, while the latter is also waiting for the former to release a resource it holds. Three possible methods to handle deadlocks are available. Each has its own merits and demerits. These methods are:

(1) Deadlock detection and removal;

(2) Deadlock prevention, and

(3) Deadlock avoidance.

Detection algorithms can detect a deadlock that has already occurred and they then try to find a minimum cost way to remove it by deallocating some resources [11, 12]. This approach has the disadvantage of a high time penalty if the resources to be deallocated are non-preemptive. Moreover, the cost of running a detection and removal algorithm in terms

of overhead is high, especially if the deadlocks occur frequently.

An algorithm is said to have time complexity O(f(n)) if the number of steps it needs to process data of "size" n is cf(n), where f(n) is some function of n and c is a constant [13]. The time complexity of the algorithm provides an approximate indication of the time required to execute it on some computer.

One of the well-known detection algorithms is $O(mn^2)$, where m is the number of resource types and n is the number of tasks [14]. Another algorithm that represents less overhead is O(mn), where m and n are as defined above. However, this latter algorithm requires two ordered lists which implies some extra overhead [11, 12].

A more general technique assigns a fixed cost c_i to the removal (forced preemption) of a resource of type r_i from a deadlocked task that is being aborted [12]. The algorithm finds a subset of resources that would remove a deadlock at minimum cost.

In general, all detection algorithms insure high supervisor overheads as well as swapping or I/O losses.

Prevention techniques are, in general, designed to exclude the possibility of a deadlock by removing one or more of the conditions necessary for a deadlock to occur. Three different approaches are suggested for the prevention of a deadlock [15, 16]. Nonetheless, each approach has a major disadvantage. The disadvantage of the first approach is poor utilization of system resources by allocating all resources a process needs all at once before it starts execution. The second approach suffers from the losses due to allowing preemption.

The last approach incurs supervisor overhead and poor utilization of resources.

All avoidance techniques use advance information about process resource requirements. Different models have been developed, each of which is different in the amount of information assumed available. Two extreme models will shortly be discussed. Intermediate models moderating the drawbacks of the extremes are available. The extreme models are usually simpler, less complicated than others but not necessarily better in terms of overhead. However, simple algorithms may be best suited for microprocessor based support which would take care of the overhead problem, such that main system resource demand could be less than that needed by a fairly complicated algorithm without support. This should be considered an advantage, since simple algorithms with high overhead tend to reduce software complexity, and hence overall system cost. The support hardware would eliminate the overhead penalty. In other words, we don't have to design more complicated algorithms to reduce the overhead; the support system will take care of that.

The first extreme model is the basic model. It assumes the availability of full information (which is impractical). The model consists of a sequence of process steps; during each step the resource usage remains constant. At the beginning of each step, an algorithm is invoked to determine whether the allocation of the requested resources is safe or not. The state of the system at time "t" relates requested and allocated resources. If it is possible to find a valid sequence of the uninitialized process steps such that all processes in the system

can run to completion, the state is safe; otherwise, the state is not safe. This model is clearly impractical and implies high overhead since the algorithm has to run before every process step can execute.

The second model is more practical [14, 17]. It assumes that only the maximum number of resources needed by each process at any time during its execution is known. In particular, each process has a resource vector. Each element in a resource vector represents the maximum number of a certain resource type that will be required by the process at any time during its execution. The algorithm utilizes an unordered list of vectors, each of which represents the rank of a process. The rank of process (i) is defined as the difference between the claim vector c_i and the allocation vector a_i (a_i represents the already allocated resources). The algorithm checks the safety of a request by trying to find a sequence in which a process can run to completion if the request is granted; otherwise, the request is denied. Fortunately, there is no need for backtracking with this algorithm [17]. However, the algorithm is $O(m^2)$, where "m" is the number of processes in the system. As "m" gets larger than five, the algorithm's overhead becomes unacceptable.

Another available algorithm is $O(m\ell g_2 m)$ [11]. It utilizes a heapsort of the list. However, the algorithm described in [17] will be considered just to prove that even simple algorithms can be supported to execute at a better speed than more sophisticated ones.

The Need for Hardware Support

As might have been already noticed, the overhead incurred in deadlock related algorithms is a major concern in designing this part of the operating system. It has been predicted that in future systems sharing an increasing number of individual users, the deadlock problems are likely to acquire greater significance [18]. It has been also predicted that systems which provide a common set of large files (or data bases), available for many users with different access rights, will consider an access to a small subset of records as a resource usage.

Based on the above, it seems appropriate to consider a hardware support module for this important part of the operating system.

The Support Module

The support module that will be presented is capable of reducing to a large degree the amount of overhead encountered in traditional deadlock avoidance schemes. The idea can also be extended to work with detection or prevention algorithms. For the sake of an example, Habermann's model for deadlock avoidance will be adopted [14, 17]. In order to understand the function and operation of the module, a brief description of the algorithm follows.

The vector $rank_i$ represents the state of process (i) according to the relation

$$rank_i = c_i - a_i$$

where c_i is the claim vector of process (i) and a_i is its current

allocation. Every element of the vector represents a resource type. A system vector "rem" (for the entire system) represents the remaining number of unallocated resources of all resource types. Upon a request by a process for resource allocation, the algorithm tries to find a sequence in which all processes can run to completion if the request is granted by searching an array that has all process state vectors [$rank_i$ $i\varepsilon 1, 2, \ldots, n$] as elements. The array is unordered and the search time is $O(m^2)$, where m is the number of processes currently holding or requesting resources. It can be proven that backtracking is unnecessary because if an nth process can be found to satisfy the relation

$$rank_i \geq rem + \sum_{j<i} a_j$$

while an (n+1)st process cannot be found to satisfy the (n+1)st relation; the allocation is not safe [17].

The support module employs a number of microprocessors plus some necessary hardware. The number of microprocessors is somewhat arbitrary and can be chosen to fit a desired speed. It is possible to execute Habermann's model as O(km) instead of $O(m^2)$, where k is an arbitrary speed factor that also defines the number of microprocessors needed and, consequently, the amount of hardware. It is worth mentioning that the support idea makes it feasible to apply Habermann's model for systems with $m > 5$ which were supposed to have a prohibitive overhead.

O(km) module, k = 1

This kind of module is feasible only when m is not very large. However, acceptable values of m can be much larger than 5 and m = 30 could be considered reasonable (m is directly related to the degree of multiprogramming). In any case, m is limited by the number of non-preemptive resources.

The module contains a total of (m+1) microprocessors of which m microprocessors are to serve the m processes in the system. One microprocessor will serve as the module's supervisor. A microprocessor that represents a process will be called a process micro. It must be mentioned that the number of active processes in the system will vary with time; however, the number of process micros can be selected as the maximum number of active processes allowed by the system at any given time. Another possibility is to select the number statistically, such that the probability that a process gets blocked because there is no process micro available is less than some small value. In such a case, the correspondence between a process and a microprocessor would be variable with time and it is the responsibility of the supervisor to assign processes to microprocessors. Figure 3 shows the organization of the module. A small bus is utilized to connect all the microprocessors within the module, while all communication between the main system and the module is done through a small area of main memory space accessible only by the supervisor.

Associated with process micro (j) are two flags 'ok_j' and 'out of $SRCH_j$'. Also, three external registers, R1j, R2j, R3j, are associated

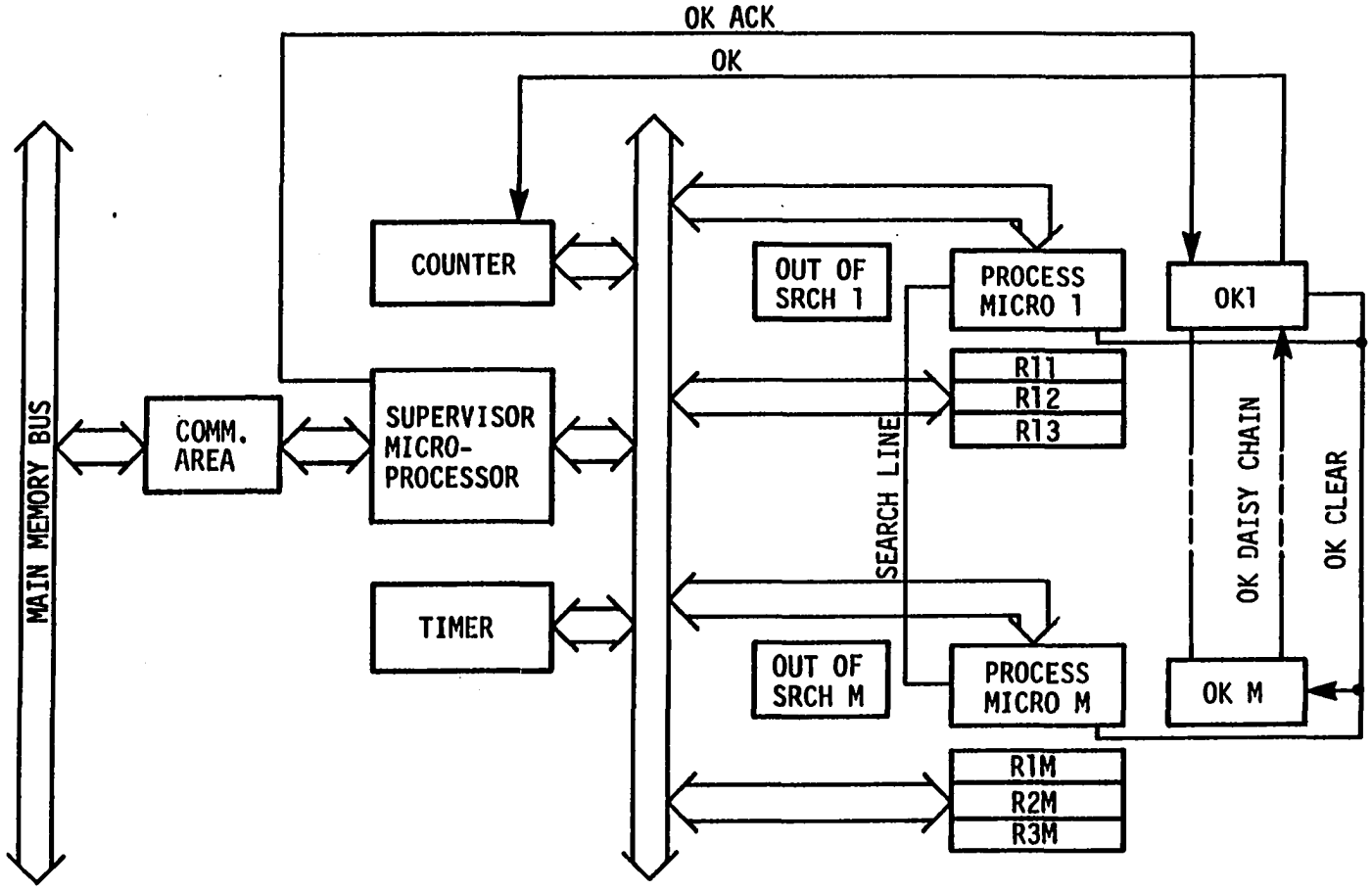


Figure 3. Organization of the O(m) module

with process micro (j) (jε1, 2, ..., m). R1j and R2j will be used as communication buffers between the microprocessors within the module; therefore, they should occupy the same locations in all microprocessors' address space. R3j will act as a status and control register. All registers are private to the module's memory space and are not accessible by the main system. The function of the two flags associated with each process micro will be discussed when the operation of the module is to be introduced.

Two modes of operation control the local bus. These modes are parallel write and addressed read/write. Part of the address on the bus defines the mode of the bus cycle. Registers R1j (jε1, 2, ..., m) will always be written into simultaneously through a parallel write cycle initiated by the supervisor and, hence, they should only occupy b1 bytes of its address space, where b1 is the number of bytes in R1. This also applies to registers R2j (jε1, 2, ..., m) except that any microprocessors in the module can initiate a parallel write into them. Registers R3j (jε1, 2, ..., m) are to be addressed separately only by the supervisor, and hence occupy mb3 bytes of its address space, where b3 is the number of bytes in R3. Registers R1j (jε1, 2, ..., m) will be used to receive supervisor messages, while registers R2j (jε1, 2, ..., m) will be used to store the rem vector after each search step. As mentioned before, registers R3j (jε1, 2, ..., m) are to represent control/status registers for the process micros.

Operation The supervisor microprocessor continuously monitors the communication area looking for a search message from the main system.

When one is found, it starts a search procedure by broadcasting the request and the requesting process number to all R1j registers. It also calculates a modified rem vector that corresponds to the rem vector if the request is granted. The modified rem vector is broadcasted to all R2 registers using a parallel write mode cycle. Each process micro compares the requesting process number to the number of the process it represents. The one that has a match becomes responsible for starting the first search step by activating the search line. Upon receiving the search signal, all active process micros start searching by comparing their rank vectors with the modified rem vector. If $rank_j$ is less than or equal to the rem vector, then process(j) is capable of running to completion if the request is granted, and hence, the process micro(j) sets its associated 'Ok_j' flag. Note that only micros that have active processes in the system should participate in the search; these will be called active micros. All Ok_j ($j \varepsilon 1, 2, \ldots, m$) are daisy chained, such that if any of them is set during a search step, a signal is generated to increment a counter 'COUNT'. The supervisor monitors the counter and whenever it detects an increment, it generates an acknowledge signal 'OKACK' that ripples through the daisy chain and stops at the first set 'Ok_n' flag, generating another signal that sets the corresponding 'OUT OF SRCH' flag. This tells process micro(N) that it has been accounted for, and that it should get out of the search after initiating the next search step. Process micro(N) calculates the new modified rem vector, stores it into all R2j ($j \varepsilon 1, 2, \ldots, m$) registers of active micros. A bit in R3j can

represent the status of processmicro(j) "active" or "inactive". This bit may control the acceptance of parallel write operations in R2j. An out of search process micro is inactive, and it is possible to include the 'OUT OF SRCH' flag in R3. Setting the 'out of SRCH' flag should also set the active/inactive bit to 'inactive' in R3. After broadcasting the new rem vector, process micro(N) clears all 'OK' flags by activating the 'OKCLEAR' line. Finally, it activates the 'SEARCH' line initiating a new search step. The search continues until either all search steps have been completed or an unsuccessful search step is encountered. A successful search means that the number in the counter is equal to the number of active processes, and in such a case, the supervisor has to pass a message to the main system indicating the safety of the request. However, any search step that ends without any 'OK' flag being set means that the result of the search is "not safe," and a message in that effect has to be passed to the main system. For simplicity, it is possible for the supervisor to set the "TIMER" to a certain value corresponding to the number of active processes before initiating the search. The "TIMER" interrupts the supervisor at the end of the expected search period, so that the supervisor can check the counter "COUNT" and determine the safety status of the request.

The support module makes the search time O(m), since only m search steps are needed at the most. The savings, as compared to Habermann's algorithm, come from the fact that up to m search steps in Habermann's model correspond to one search step in our case. In other words, up to

m search steps are parallel processed in one search step time in the support module.

O(km) module, k > 1

The idea and basic operation are the same as the O(m) module discussed above except that each process micro will be responsible for k processes instead of only one. In this case, k 'OK' flags and k 'OUT OF SRCH' flags will be needed per microprocessor. Also, 3k external registers per microprocessor are to be used. However, it is possible to use the same amount of hardware per process micro as the O(m) case giving the microprocessor more work to do internally, on in a private read/write memory such that a process micro can distinguish different process states.

It is clear that a trade-off between speed and the amount of hardware is needed. This is because hardware savings are at the expense of execution time. However, this kind of module (k > 1) might be necessitated by a large degree of multiprogramming.

O(km) module, k < 1

The idea and module organization are different in this case. It is feasible only when the number of non-preemptive resources in the system is relatively small. The idea is based on the following observations:

(1) Limited number of resources means limited number of competing processes.

(2) The state of the system at any given time is uniquely

determined by the process ranks.

(3) Not all process rank combinations are feasible [17]. Hence, only subset of all system states (determined by process ranks) are to be accounted for.

As discussed above, when all process ranks are known, it is possible to run an algorithm to determine the safety condition of the state. Thus, if we consider the ranks as inputs, the state as an output, it is possible to design some hardware to substitute the algorithm as shown in Figure 4.

The hardware is to consist mainly of a Read Only Memory (ROM) and some encoding logic. The ROM stores states corresponding to every feasible rank combination. Only one bit per combination is required. The value stored in a certain bit is determined during the design phase by running the algorithm for the corresponding rank combination.

Module Organization

Only one microprocessor is needed, as shown in Figure 5. It receives requests from the main system as discussed before. One register per process is used to hold the rank of the process. The combination of all register contents represents the state of the system. Thus, all we need to do is to encode all register contents to produce an address that addresses the ROM. This can be implemented by using a tree of Programmable Logic Arrays (PLAs). Each PLA at the first level combines and encodes two or three registers. At the same time, it suppresses redundant states that are not feasible before presenting its output to the next level. The process is repeated until final address

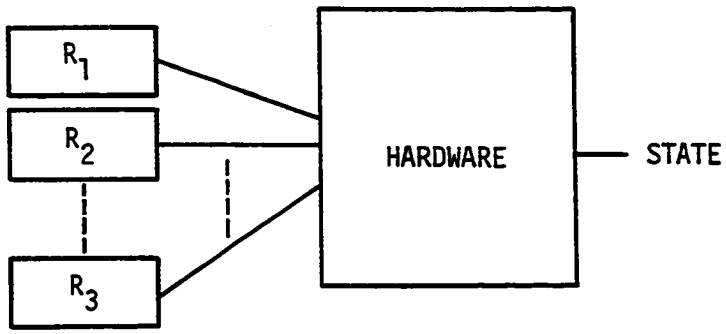


Figure 4. Hardware substitute for a deadlock avoidance algorithm

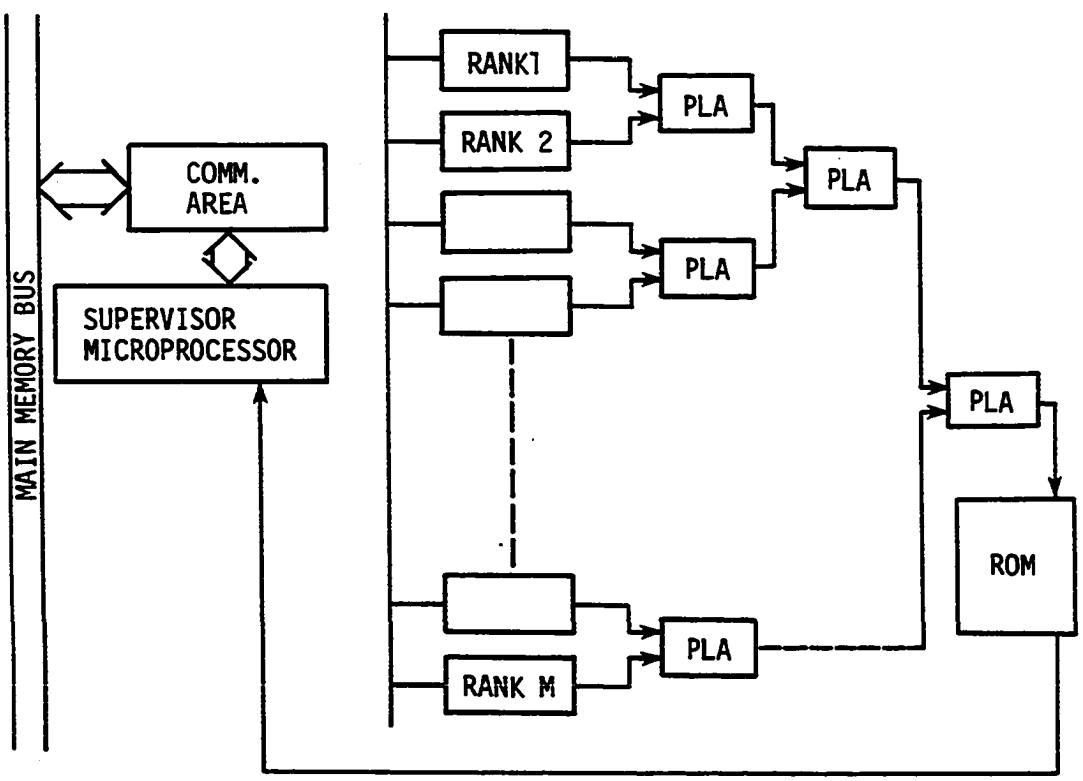


Figure 5. O(km), k < 1 module organization

(representing the state of the system) is presented to the ROM. The output of the ROM (one bit) is the safety state of the allocation request. The microprocessor receives the result and returns a message to the main system. The microprocessor has to return the state of the requesting process to its original state before the request if the request is denied. If the request is safe, the modified process state remains unchanged until the process makes another request or until it releases some or all the resources it holds.

Clearly, the module is very fast compared to the two modules previously described. However, it is only suitable when small numbers of non-preemptive resources are employed in the system.

CHAPTER V. A MODULE FOR THE EXACT IMPLEMENTATION OF THE LEAST RECENTLY USED REPLACEMENT POLICY

In our second example, we will not only provide the idea of how to implement the exact Least Recently Used (LRU) replacement policy in a demand paging system, but also give the detailed design of a submodule that was designed, built, and tested. The results proved the correctness of the idea and the feasibility of the technique. The next chapter will cover the results of the experiment, while in this chapter all design and implementation details will be given.

Introduction

A replacement policy in any memory management system has to find obsolete information in main memory. This is necessary before any new information can be loaded into the Main Memory (MM) if there is no free space available. Free space in MM is created either when information residing in MM is deleted or by purposely swapping out information which is of no immediate interest. If information is deleted, the cleared space can simply be added to the pool of free space. If free space must be created by swapping, we need a criterion by which obsolete information can be distinguished from active information. The right time to look for obsolete information is when memory space is requested while the free space is insufficient. This is why an algorithm which selects the information to be swapped out is called a replacement algorithm. A replacement algorithm applied by a demand paging system allows referencing inaccessible pages and interprets such a reference

as a request to make the page accessible. This is called a "page fault." If handling a page fault is left to the operating system, the operating system must find an obsolete page which can be exchanged for the requested page. The overall objective of a two-level storage management is to have those pages in main memory which have the highest probability of being referenced in the near future. Therefore, nearly all replacement algorithms have as their objective guessing which page in main memory currently has the lowest probability of being referenced. This page is then distinguished as the result of the replacement algorithm.

The simplest replacement policy selects a page at random. This algorithm is implemented by a designer who believes that it is not possible to make an intelligent guess as to which page is least likely to be referenced in the near future. Of course, implementing this policy is trivial, but its performance is poor [17].

Two other straightforward algorithms are the First-In-First-Out (FIFO), and Round Robin (RR). The FIFO replacement algorithm is based on the observation that the probability of referencing a page in the near future is likely to be a decreasing function of the time that the page resides in main memory. It seems, therefore, that the least harm is done if the oldest page is swapped out, that is, the page that was brought into main memory the longest ago. The FIFO replacement algorithm needs the support of a FIFO queue in which pages in the frames are ordered by arrival time. Implementing this algorithm is also trivial.

The Round Robin (RR) algorithm is based on the expectation that the

time intervals during which pages are referenced are reasonably close to an average time length. If this is true, the page with the lowest reference probability is the one in the frame least recently selected. Implementing the RR is very simple by the use of an "own" variable that points to the frame last cleared. When the algorithm is called, it cycles through the frame table starting from where it left off last time until it finds a frame that is in use.

Experiments and measurements have shown that the performance of the FIFO and the RR replacement policies are not good [17].

The Least Recently Used (LRU) algorithm recognizes the fact that some pages are used for longer periods of time than others. For example, a page containing part of a main program or the global data of a program usually has a longer lifetime than a procedure page or page of temporary data. Therefore, the LRU algorithm is based on the assumption that the probability of referencing a particular page is inversely proportional to the time interval between the last reference to the page and the present moment. The page selected by the LRU algorithm is then the least recently referenced page. Numerous studies pointed out the superiority of the LRU algorithm, one of which will now be reviewed.

An interesting study that was done at Princeton University in 1968 provides experimental data on the behavior of programs in a paging environment [19]. The study discussed the problems of paging systems in general and the problem of poor object program behavior in a multiprogramming environment in particular. Specifically, the frequency of page turning (transferring pages in and out of main memory) necessary

for the execution of a program never wholly in main memory, tends to degrade the system performance by introducing an excessive amount of input/output interference. Although the study dealt with different aspects that might affect paging system performance such as page size, number of pages kept in main memory at one time and page replacement algorithms, our concern is with the part that studied the effect of different replacement policies on program behavior. The experiment was designed for the study of programs written for the IBM system/360 model 50 computer and organized to operate under the operating system in use at Princeton University at that time. Each program studied was used as input to an interpreter written for the mentioned machine. The paging behavior of the interpreted program was traced by recording an identification of the new page, determining whether it was a data or an instruction page, and determining the number of instructions executed since the last page request. Simulations were carried out to determine the paging characteristics of the programs when run under different page replacement algorithms. Specifically, the study compared the page fault frequency introduced by the LRU algorithm and by the Belady Optimum Replacement (BOR) algorithm. The BOR is based on a prior knowledge of the entire sequence in which pages are used in the execution of a program [20]. The algorithm is considered the best possible replacement algorithm, but it is totally impractical. Thus, the study selected the BOR as a means of comparing the performance of various practical algorithms with the best possible one.

The study concluded that page turning is a substantial problem in

a demand paging system, and that a least recently used replacement algorithm yields a performance within about 30% of that optimum page replacement sequence. The authors also remarked that with sufficient main memory, the LRU algorithm is an appropriate replacement algorithm in most cases. The authors also mentioned that good agreement had been observed with the study made by Belady under different conditions [19, 20].

Two algorithms that approximate the LRU are the Least Frequently Used (LFU) and the MULTICS [17, 21]. The LFU algorithm counts the number of references to a page and selects the page that had been least frequently used. However, the overhead in the LFU case is very high, since the whole page table has to be searched every time a page fault occurs.

The MULTICS algorithm (also known as the second chance or the clock algorithm) is a much better approximation to the LRU than the LFU. Its overhead is much less and provides better performance than the LFU. However, the MULTICS is still an approximation and incurs an overhead that can be considered high [17, 19].

Despite the near full agreement that the LRU is the best practical replacement algorithm, it was believed that the exact implementation of the LRU is impractical. A common phrase in the literature was that exact implementation of the LRU is not feasible. For instance, in a 1978 book [17], the author said:

> An exact implementation of an LRU algorithm is not feasible because of its tremendous overhead on current hardware. It would be necessary to record the time of

> reference ever time a page is referenced because the operating system has no way of knowing which reference to a page is the last.

Well, we can say that this is no longer true. A hardware submodule was designed and built to challenge the above statement. The submodule had the objective of exact implementation of the LRU algorithm, and proved to be successful at a very low cost. The overhead in terms of CPU time is possibly less than that of any existing replacement algorithms.

There are two reasons that made possible the accomplishing of this:

(1) The availability of low cost, powerful microprocessors and hardware in general, and

(2) Parallel processing within a support module that works with minimum interference with the main system.

The design and operation of support module will now be given in detail.

Exact LRU Support Module Organization

The basic idea in the design is to use a number of submodules that work simultaneously within an LRU module. Each submodule contains a microprocessor and is responsible for finding the least recently used page frame in a certain main memory area. If we divide main memory into 'n' equal areas, then each submodule would be assigned one such area.

Only the part of the address (on main memory bus) that corresponds to the frame number is the concern of the module. The in-page address is of no importance and can be ignored because we are dealing with pages,

not locations. The frame address stream is first filtered out to suppress all consecutive references to the same page except the first one. This means that only the first in a sequence of references to a page is to be considered. It is only necessary to compare the relative reference times to different pages rather than the absolute reference times. This filtering is also important to reduce the arrival rate at the module and to minimize possible interference with the main system as will be explained later. An address distribution circuit, mainly a decoder, is to be used to route the filtered stream to submodules according to the main memory areas they service. A TIMER is incremented every time a filtered frame number is released from the filtering circuit. The TIMER serves all submodules, so only one timer is needed for the whole module. Figure 6 shows the block diagram of the support module. A frame address stream arriving at a submodule will be called an area stream.

A supervisor microprocessor for the whole module is responsible for finding the overall LRU page frame from among the LRU area pages produced by the submodules. The supervisor also has to communicate and present results to the main system. A more detailed discussion about the supervisor's functions will be given later in this chapter.

Submodule Organization

A submodule designed to handle an area stream contains basically a microprocessor, a read/write or Random Access Memory (RAM), and Read Only Memory (ROM). The organization of the submodule is shown in Figure 7. The RAM is used to record the time a page frame is referenced,

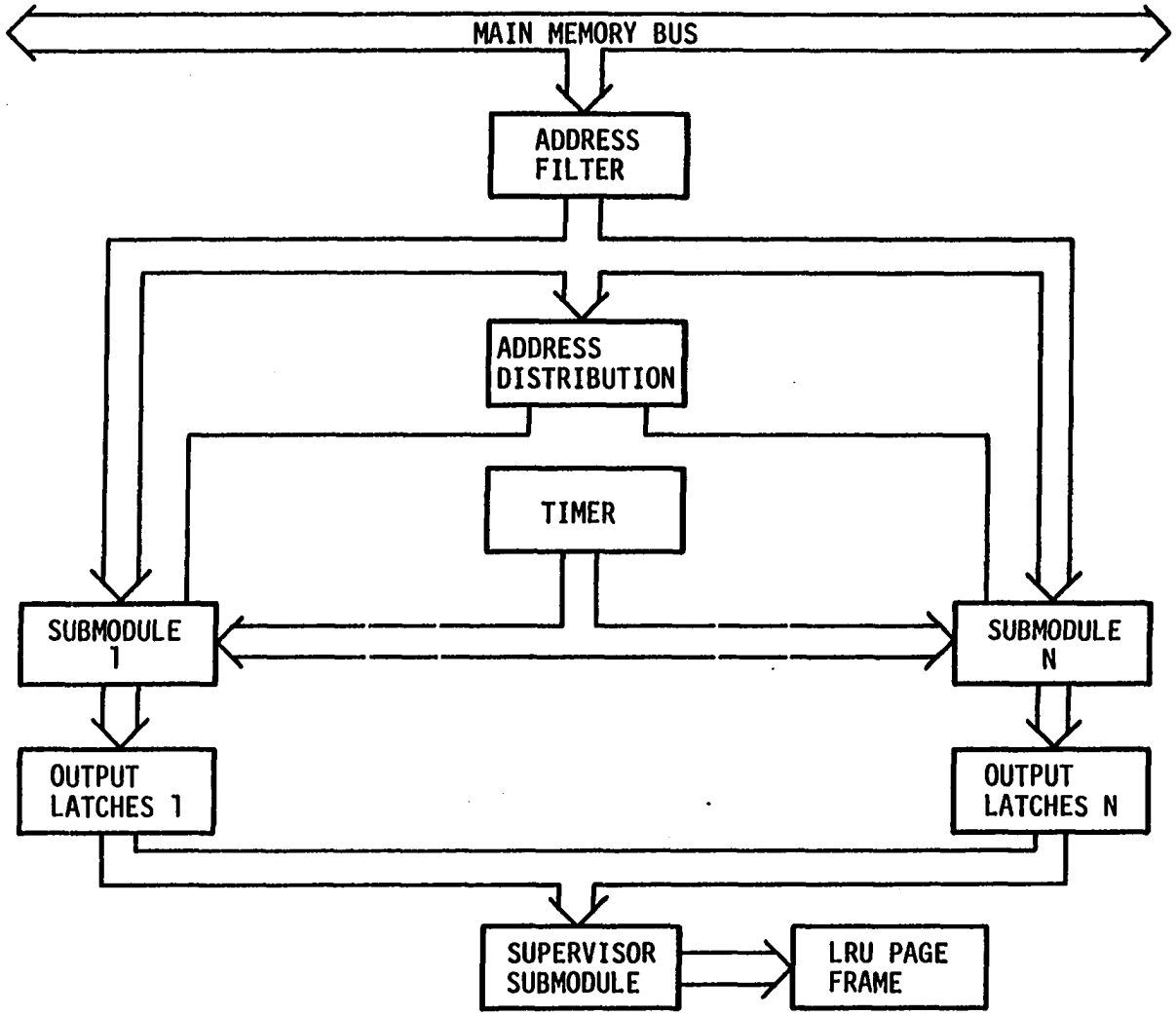


Figure 6. Block diagram of the LRU module

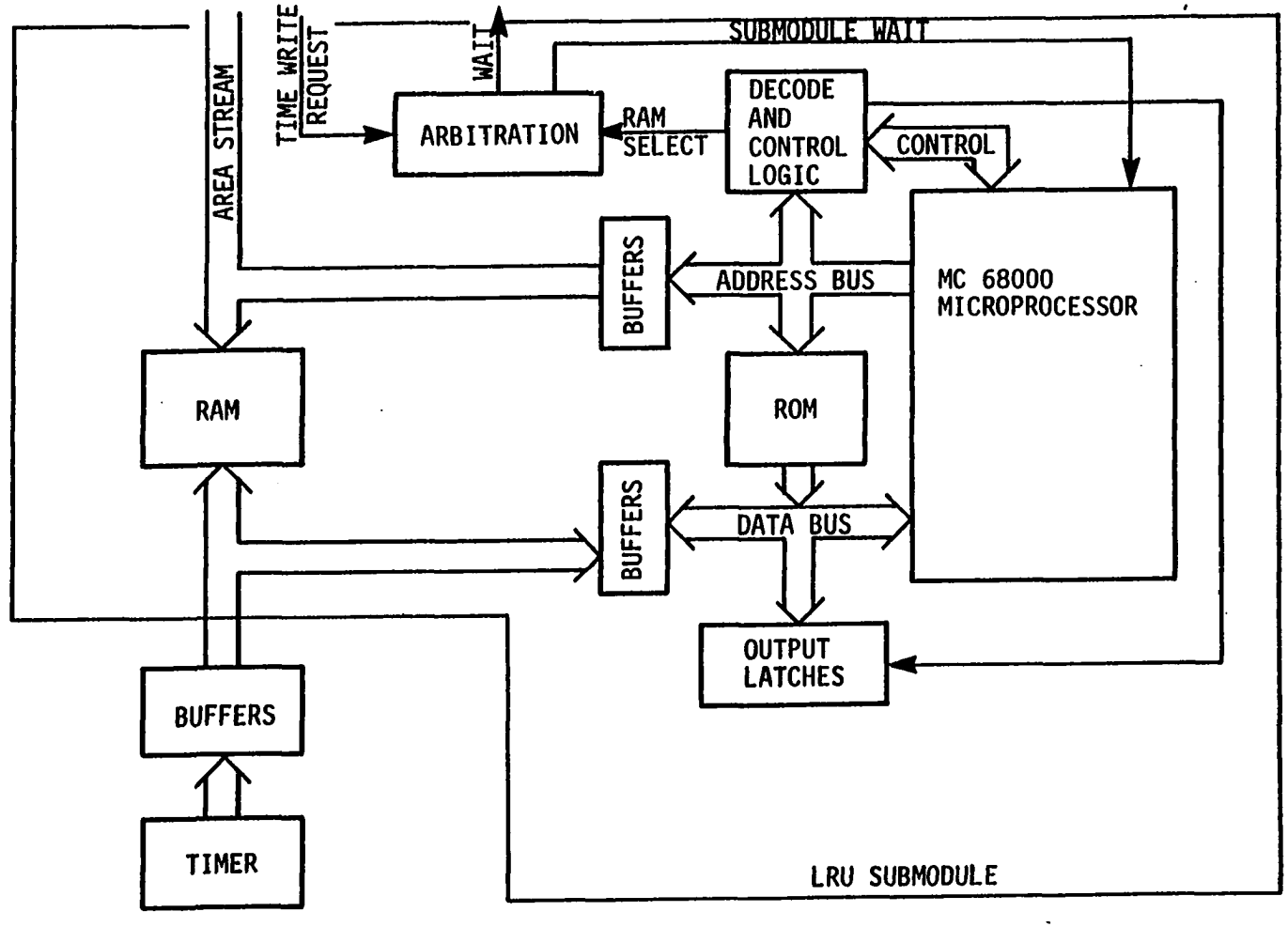


Figure 7. LRU submodule organization

while the ROM contains the routine to be executed by the microprocessor. For each page frame in the main memory assigned to the submodule, there is a certain set of locations in the RAM that contains its last time of reference. These locations will be called a Time Record (TR). Every time a frame address arrives at the submodule, the contents of the TIMER are copied into the TR corresponding to the frame address. The frame address itself is used to address the RAM directly to write the TIMER into the frame's TR in RAM.

The microprocessor reads its LRU routine from the ROM without interference from the main system. Three-state buffers are used such that the microprocessor can access its ROM freely at any time without having to interfere with the outside world. The only time interference has to be considered is when the microprocessor wants to access the RAM. An arbitration circuit is used to arbitrate between a time write operation and a microprocessor RAM read cycle.

The submodule is designed to handle 128 main memory page frames. If we assume that the main memory has 1024 page frames, then it can be seen as composed of eight equal areas of 128 frames each. However, the design can easily be modified to assign the submodule different number of frames other than 128. It is much more convenient to divide main memory into a number of areas that is a power of 2, and at the same time assign a submodule a number of frames that is also a power of 2. This could result in a much easier and more efficient design. For instance, a submodule can take care of 64, 128, 256, or 512 frames with different module response times. We chose to assign 128 frames to the

submodule and check the response time. As will be discussed later in the next chapter, the response time obtained with 128 frames/submodule is quite acceptable and doubling the number of frames should result also in an acceptable response time.

Detailed Submodule Design

As shown in Figure 7, both the data and address buses are buffered to control accesses to the RAM. Since the RAM is the source of potential conflicts between the main system (writing a time record) and the submodule's microprocessor (reading a time record), its access is controlled by a simple arbitration circuit. The only case a wait signal is generated by the arbitration circuit and sent to the main system is when the microprocessor is in the process of reading the RAM and a filtered address arrives at the submodule to initiate a time write cycle into the RAM. As discussed later, the probability of this event can be reduced to about 1%. Although our intention was to build and test a submodule, it was also necessary to design and build some extra hardware. For instance, an address generation module capable of producing some prespecified sequences of addresses is necessary to enable testing the submodule. Also, even though only one TIMER circuit is needed for the whole module, it is essential to have a TIMER circuit to test the submodule. In the following section, a detailed description of the major elements in the submodule is given.

The microprocessor

Motorola's MC68000 [22] was selected as the submodule's microprocessor for the following reasons:

(1) It is a fast microprocessor that can operate at high clock rate (up to 8 MHZ).

(2) Its address space is very large; in fact, it is much larger than is needed. This allows the use of some address lines for direct control with virtually no decoding, thus simplifying the design and reducing the cost.

(3) Most instructions can handle long words (32 bits). This results in a simpler and more efficient routine, especially since the TIMER is chosen to be 32 bits.

(4) Its data bus is 16 bits wide, which means fewer references to the RAM than would be the case if an 8 bit micro were selected. Thus, fewer potential conflicts with main system are to be expected.

(5) It is possible to utilize the bus error feature provided by the MC68000 to further reduce the possibility of interference with the main system activities. This will be explained in detail in the next chapter.

Although the MC68000 has many other areas of strength and superiority, only subsets of its capabilities were actually used in the design. For example, it has an advanced interrupt handling scheme that uses seven levels or priority; however, the whole interrupt system had not been utilized in the design. The interrupt system might be useful in

designing the supervisor submodule as a means of interaction with the main system.

Arbitration

A simple arbitration circuit is employed to organize and control accesses to the RAM. The arbitration circuit receives requests from the microprocessor to perform RAM read cycles and receives time write requests whenever a valid frame address arrives to the submodule. The request that arrives first is granted the access to the RAM. The arbitration circuit is a simple R-S latch built of fast NAND gates, namely SN74S00 integrated circuits. It must be mentioned that the main system does not actually make requests to access the RAM in the submodule but tries to reference a main memory frame that is assigned to the submodule. The request received by the arbitration circuit is generated within the module and can be interpreted as a request to record the reference time from the TIMER into the frames' TR in the submodule's RAM. Thus, the main system is not actually aware of what is taking place in the support module, but it sometimes may have to wait until the time recording process is completed. Thus, if main memory control logic is employed that is capable of causing a main CPU to wait until the addressed area is free, the resulting wait signal must be logically ORed with the wait signal generated by the support module.

Random Access Memory (RAM)

The read/write memory or Random Access Memory (RAM) represents a somewhat critical part of submodule design for the following reasons:

(1) The RAM is the only part of the submodule that can cause access conflicts between the microprocessor (trying to read) and the time recording process (trying to write).

(2) The addressing space as seen by the microprocessor is different from that seen by the time recording scheme. This will shortly be explained.

(3) Critical timing problems result because arbitration should be as fast as possible to allow both systems to work at their maximum speed. At the same time, timing specifications of the RAM chips must be met to ensure correct operation. Also, with the existence of two sets of address and data buffers, some other specifications had to be taken into account to enable and disable the buffers at appropriate times.

A set of 6116-4 RAM chips was selected for the read-write memory. They have an access time of 200 ns. Although the 6116 chips are internally organized as 2Kx8 bits, only 128 bytes/chip were actually used. The reason is the unavailability of wider word chips with fewer words.

The RAM is organized as 128 records, each having 32 bits. Hence, four 6116 chips are needed. This is consistent with assigning 128 main memory frames to the submodule.

Since the MC68000 is a 16-bit microprocessor, each Time Record (TR) has to be read from memory in two read cycles. This is not the case when a time write process is to be performed since it is possible to write a whole TR in only one write cycle (the microprocessor is not

involved here). This has been achieved in the design by using only 7 address lines from the main system, whereas 8 address lines from the MC68000 address have to be utilized to address the RAM. The least significant bit (A_1) of the MC68000 is used to select either the lower 16 bits ($A_1 = 0$) or the higher 16 bits ($A_1 = 1$) of a time record. Note that A_0 of the MC68000 doesn't appear on the address bus but is used internally in the case of byte instructions. All the instructions used are either word or long word instructions.

As mentioned earlier, an address stream generation module was built to allow testing the LRU submodule. The former employed another MC68000 microprocessor. To differentiate between the signal lines associated with the submodules MC68000 and those of the test module, we will affix letter P (for processor) to the submodule lines and a T (for test) to the test module lines.

Addressing the four RAM chips is either done using $A_{2P} - A_{8P}$ of the submodule or $A_{1T} - A_{7T}$ of the test module. Figure 8 shows the RAM addressing mechanism, as well as major control signals that control RAM operation. The control part is discussed in detail later in this chapter.

To meet the timing requirements, it was found essential to do the arbitration as early as possible in any RAM read or time write cycle. This is achieved by using A_{13P} as a RAM read request signal for the LRU submodule while A_{12T} is used as the time write request signal. These signals become valid $\frac{1}{2}$ clock period in advance of the actual read or write cycle starts since the address lines in the MC68000 are activated

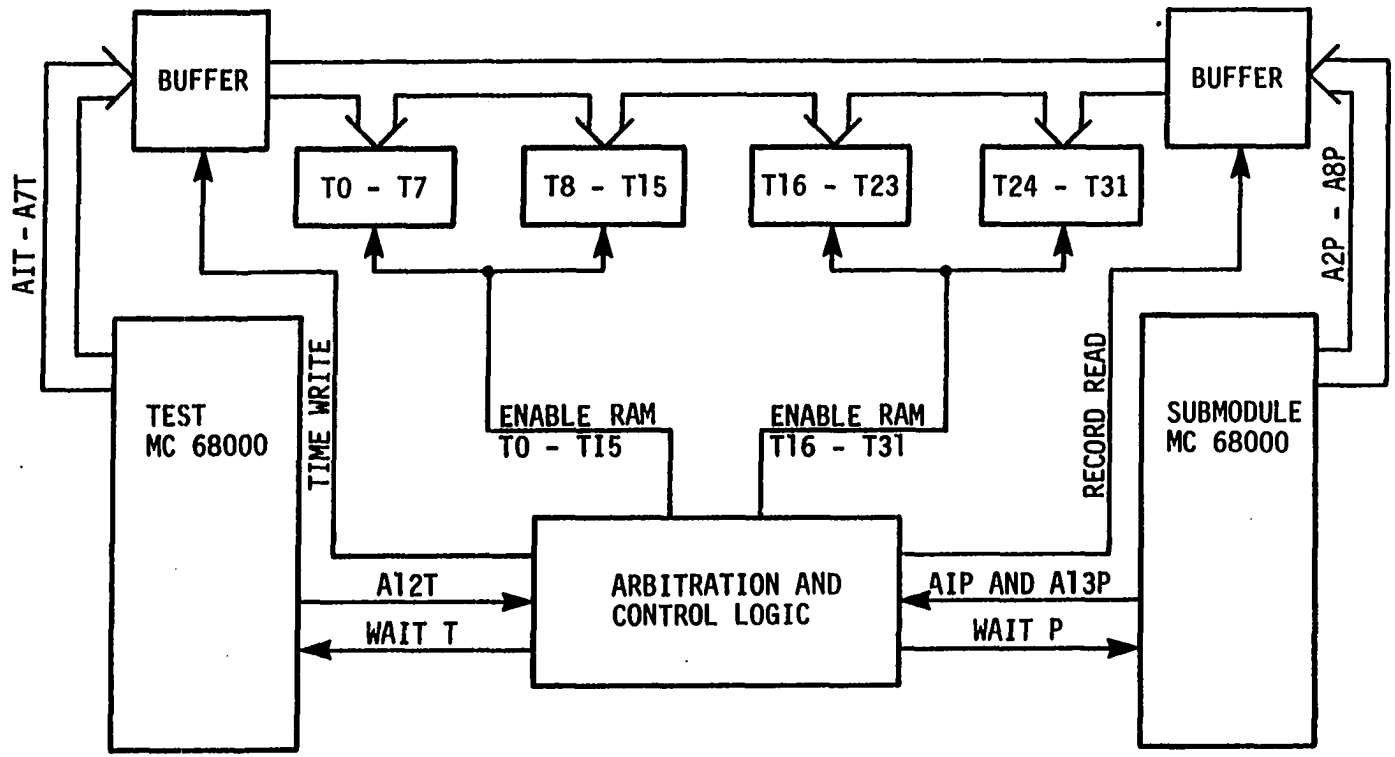


Figure 8. RAM addressing mechanism

$\frac{1}{2}$ clock period before the Address Strobe ($\overline{AS}$) signal is activated. Of course the use of A_{12T} and A_{13P} had to be taken into consideration when the programs were designed.

Since the RAM can be accessed from two different sources, a set of 3-state address and data buffers are needed on each side. However, it was essential to include two sets of data buffers (16 bits each) at the LRU submodule side because use of only one 16-bit buffer would short circuit some of the TIMER buffer output lines. This can be understood from Figure 9, which indicates that each TIMER buffer output line goes to one RAM chip and mandates that the same must apply to the other side to avoid short circuits.

The existence of two sets of buffers requires exclusive enabling, that is, one set is enabled at a time. This is taken care of by the arbitration circuit which always has one of its outputs active at a time. If no request is made to access the RAM, both buffers have to be disabled. It must be mentioned that bidirectional buffers are used at the submodule's side to allow the microprocessor to initialize the time records at the beginning of operation.

Read Only Memory (ROM)

The ROM stores the routine designed to implement the exact LRU algorithm and is considered private to the submodule. This part of the circuit is designed such that the microprocessor can access the ROM at any time freely without any kind of interference from the outside world. This means that a time record can be written into the RAM while the microprocessor is fetching or executing instructions that do not require

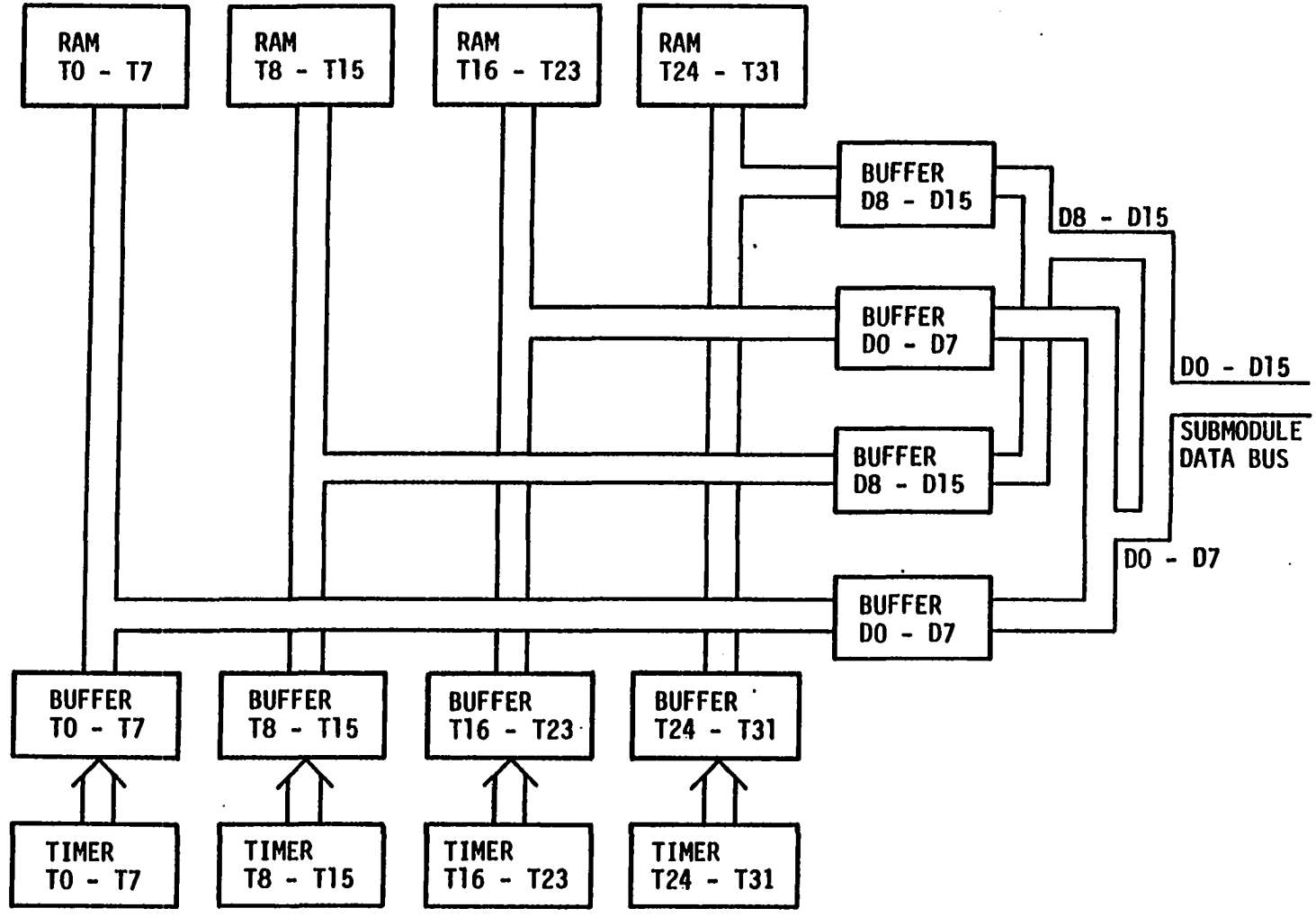


Figure 9. RAM data buffering

RAM accesses. Thus, the microprocessor can operate at full speed as long as it is not accessing the RAM while a time record is being written. The existence of data and address buffers permits direct connection of the address and data buses to the ROM. All control signals needed to control the ROM operation are generated directly from the microprocessor's address and control lines.

The TIMER

Although only one TIMER is required to serve all submodules, it was necessary to build one to allow testing the submodule. The TIMER was chosen to be a 32 binary counter that is incremented every time a valid address is released from the filtering circuit. The SN74393 chips are utilized to build the TIMER. Each chip can be configured to form an 8-bit binary counter; therefore, four chips are needed to build the 32-bit counter.

Since the arrival of a valid frame address to the module implies incrementing the TIMER and writing it into the RAM, the stability of all TIMER bits must be ensured during the write operation. This is accomplished by performing the increment operation immediately after the write operation is completed. The rising edge of the signal that enables the TIMER buffers (active low signal) is used to increment the TIMER. However, the TIMER needs a maximum of 240 ns to stabilize all 32 bits, which means that 240 ns must elapse between two consecutive time write operations to ensure correctness. It is possible to solve this problem by utilizing a single shot or monostable multivibrator that has a period of slightly over 240 ns (say 250 ns). The monostable

is driven by the TIMER increment signal and its output is used to inhibit any successive signal that arrives during the 250 ns period. Although this technique allows two time records to have the same value, it should not be considered a problem for two reasons:

(1) The probability of switching from one page frame to another after just one reference is very small (locality of reference) [17].

(2) If a page is to be selected as the LRU from two pages that have the same reference time, it does not make much difference which one is selected. This is because an output is produced by the module, say every 2-3 ms which makes 250 ns negligible. In fact, the 250 ns can be approximated to zero with an error of 1/800 at most.

It is also possible with faster chips than the SN74393 to avoid the whole stability issue, provided that the main system is not too fast. The SN74393 was good enough for the experiments since the address generator speed was not too fast for the selected chips and two consecutive increment signals were more than 250 ns apart. Thus, there was no need for the monostable in our circuit although it is trivial to employ it. Figure 10 shows the TIMER and its buffers while Figure 11 shows a timing diagram of a time recording and TIMER incrementing cycle. Notice that the buffers in Figure 10 are built of SN74LS244 chips which are unidirectional.

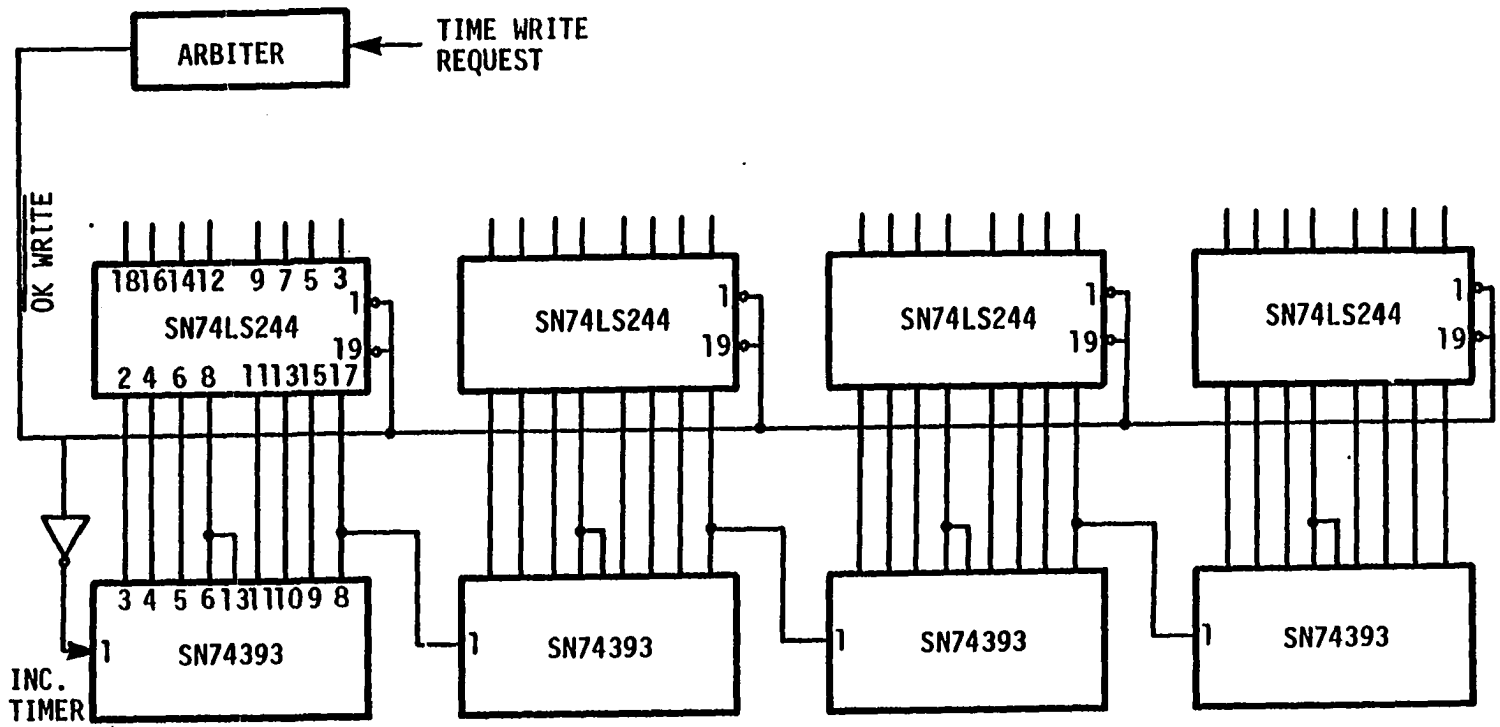


Figure 10. The TIMER and its buffering

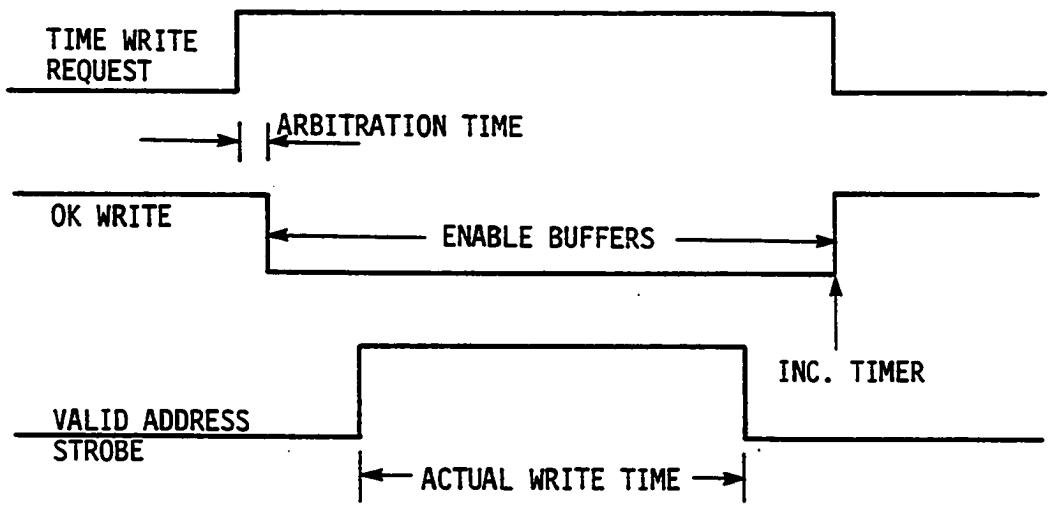


Figure 11. Time recording timing diagram

Decoding and control logic

Low power Schottky TTL chips are used throughout the submodule except for the arbitration circuit. This makes it possible to directly load the MC68000 microprocessor pins with more than one load. Actually, some pins are loaded with up to five loads directly without buffering. Moreover, as is well-known, Schottky logic is superior in handling unwanted noise signals because of the existence of a clamping diode at each input.

Since normal Schottky chips are faster than low power Schottky chips, the arbitration circuit which has to be very fast utilized normal Shottky chips.

Output latches

The output produced by a submodule is composed basically of two parts. The first part is the LRU page frame address within the main memory area assigned to the submodule. In our case, this part needs only 7 bits; therefore, an 8-bit output latch is enough to hold it. The other part of the output is the time of the last reference to the LRU frame; i.e., the time record of the output LRU frame. Since a time record is 32 bits wide, four 8-bit latches are needed to hold the time part. Thus, a total of five 8-bit latches is needed to hold a submodule's output. Intel's 8212 chips are used as output latches and are connected to the data bus through two SN74LS244 buffer chips. The reason why the time record is to be dispatched is that the supervisor microprocessor needs to compare the time records of different LRU frames

produced by different submodules in order to be able to find out the overall LRU page frame in the whole main memory.

Now, having described all major parts of the LRU submodule, a description of the address stream generation module will follow.

Address Stream Generation Module

The function of the address stream generation module is to simulate the main system address stream as well as the address filtering and distribution circuits. Thus, all that is needed is to generate a prespecified address stream that can be directed to the LRU submodule to facilitate testing and evaluating the submodule.

An MC68000 microprocessor is used to simulate the supported main system central processing unit(s) (CPUs). Two 2716 ROM chips, address buffering chip, and some LS chips are used along with the MC68000 to form the address generation module.

The basic idea is to have the microprocessor execute a very simple routine that is designed to produce some prespecified address sequence on the test module's address bus. Since only seven address lines are to be directed to the LRU submodule, it is necessary to differentiate between ROM references and addresses that should be directed to the submodule. This is done by assigning a lower address space to the ROM and higher one to the generated stream. Address line 12 (A_{12T}) is used to separate the two areas, hence $A_{12T} = 1$ means the address on the bus is to be directed to the submodule. Address lines A_{1T} through A_{7T} are used to represent a page frame address whenever A_{12T} is high. The block diagram of the test circuit is shown in Figure 12.

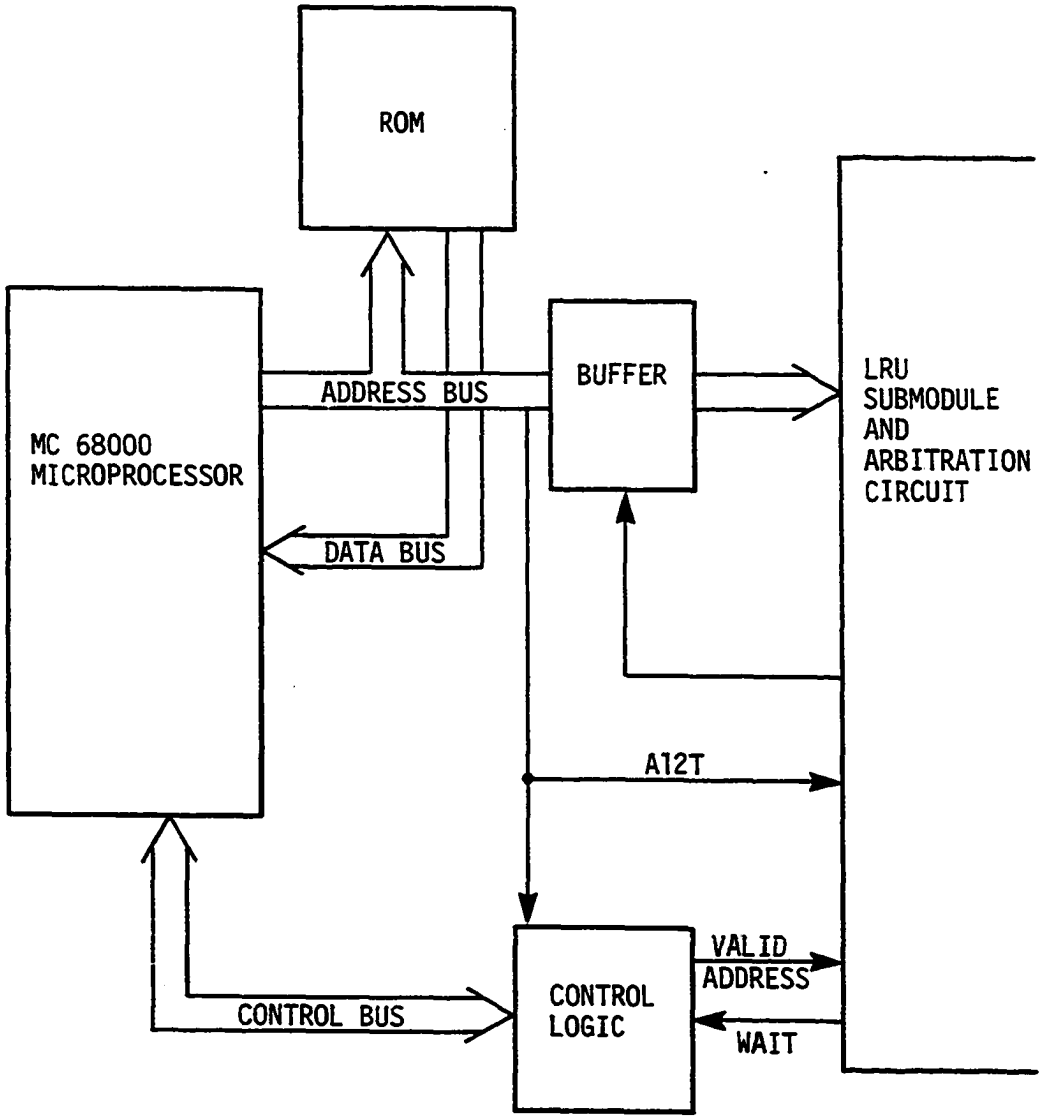


Figure 12. Address generation circuit block diagram

```
"68000"
;        ADDRESS GENERATION ROUTINE"ADDGEN4".
         ORG     000H
         HEX     0000,2300
         HEX     000,0400     ;ALL ADDRESSES ARE HEX
         ORG     400H
         MOVE    #0700H,SR    ;INITIALIZE STATUS REGISTER.
         MOVE.L  #1100H,A2
INIT     MOVE.L  #1000H,A0
         MOVE.W  #5,D2
         MOVE.W  #2,D3
         MOVE.W  #1,D4
         MOVE,W  #8,D5
SEGM1    MOVE.L  #1030H,A1    ;   "SEGMENT1".
LOOP1    MOVE.W  [A0]+,D0
         CMPA.W  A0,A1
         BNE     LOOP1
         ADD     #02H,A0      ;SKIP 1030.
LOOP2    MOVE.W  [A0]+,D0
         CMPA.W  A0,A2
         BNE     LOOP2
         MOVE.L  #1000H,A0
         DBNE    D2,LOOP1     ;EXECUTED 5 TIMES ?
SEGM2    MOVE.L  #1072H,A1    ;   "SEGMENT2"
LOOP3    MOVE.W  [A0]+,D0
         CMPA.W  A0,A1
         BNE     LOOP3
         ADD     #02H,A0      ;SKIP 1072.
LOOP4    MOVE.W  [A0]+,D0
         CMPA.W  A0,A2
         BNE     LOOP4
         MOVE.L  #1000H,A0
         DBNE    D3,LOOP3     ;EXECUTED 2 TIMES ?
DEGM3    MOVE.L  #10F8H,A4    ;   "SEGMENT3"
         MOVE.L  #1040H,A3
         MOVE.L  #1020H,A1
LOOP5    MOVE.W  [A0]+,D0
         CMPA.W  A0,A1
         BNE     LOOP5
         ADD     #02H,A0      ;SKIP 1020.
LOOP6    MOVE.W  [A0]+,D0
         CMPA.W  A0,A3
         BNE     LOOP6
         ADD     #02H,A0      ;SKIP 1040.
LOOP7    MOVE.W  [A0]+,D0
         CMPA.W  A0,A4
         BNE     LOOP7
         ADD     #02H,A0      ;SKIP 10F8
```

Figure 13. Address generation routine "ADDGEN4"

```
LOOP8   MOVE.W  [A0]+,D0
        CMPA.W  A0,A2
        BNE     LOOP8
        MOVE.L  #1000H,A0
        DBNE    D4,LOOP5      ;DO NOT REPEAT.
SEGM4   MOVE.L  #1016H,A1     ;   "SEGMENT4"
LOOP9   MOVE.W  [A0]+,D0
        CMPA.W  A0,A1
        BNE     LOOP9
        ADD     #2H,A0        ;SKIP 1016.
LOOP10  MOVE.W  [A0]+,D0
        CMPA.W  A0,A2
        BNE     LOOP10
        MOVE.L  #1000H,A0
        DBNE    D5,LOOP9      ;EXECUTED 8 TIMES ?
SEGM5   MOVE.L  #10C2H,A1     ;   "SEGMENT5".
LOOP11  MOVE.W  [A0]+,D0
        CMPA.W  A0,A1
        BNE     LOOP11
        ADD     #2H,A0        ;SKIP 10C2.
LOOP12  MOVE.W  [A0]+,D0
        CMPA.W  A0,A2
        BNE     LOOP12        ;DO NOT REPEAT SEGMENT5.
        MOVE.L  #1000H,A0
        BRA     INIT          ;REPEAT ALL SEGMENTS.
```

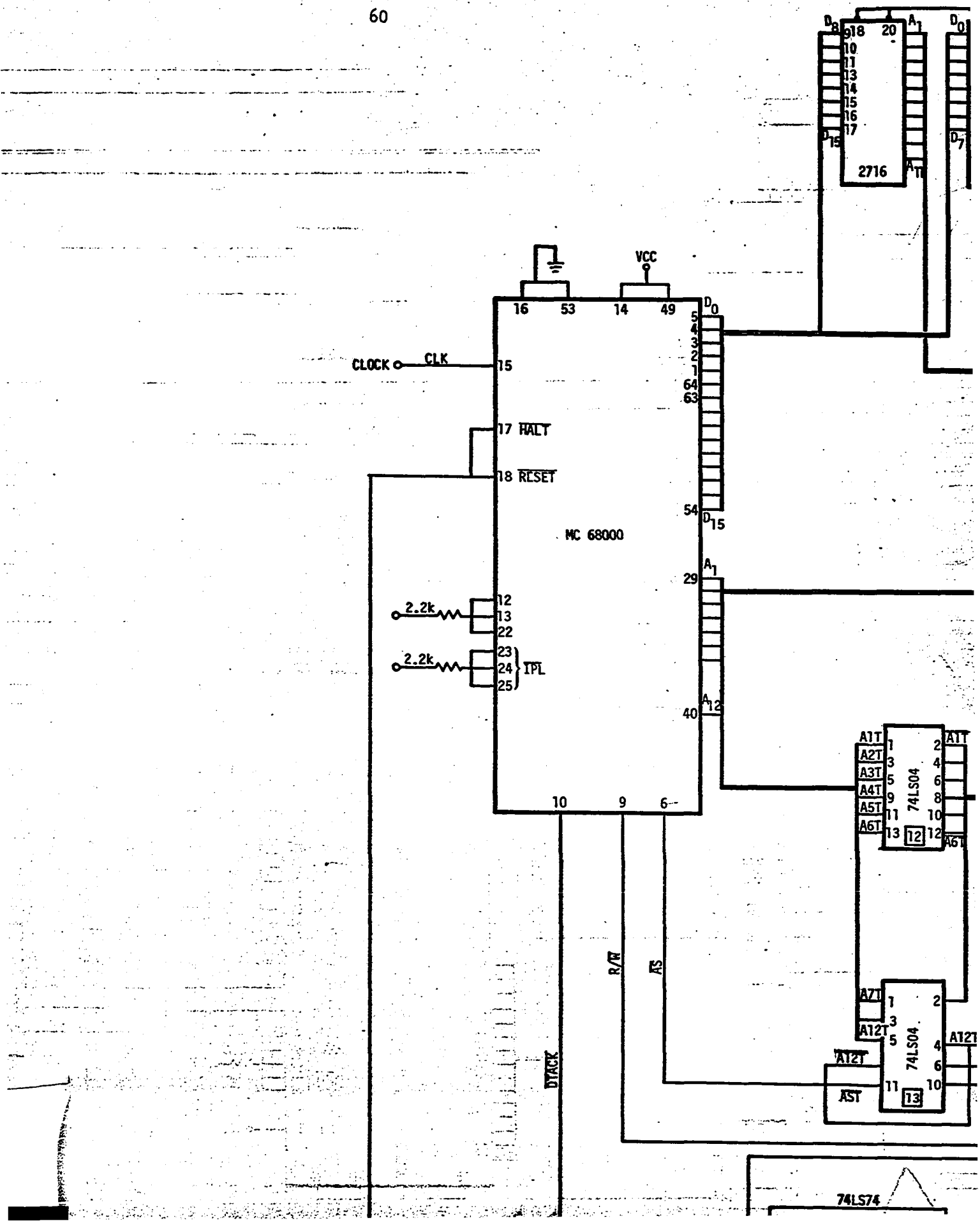
Figure 13. (Continued)

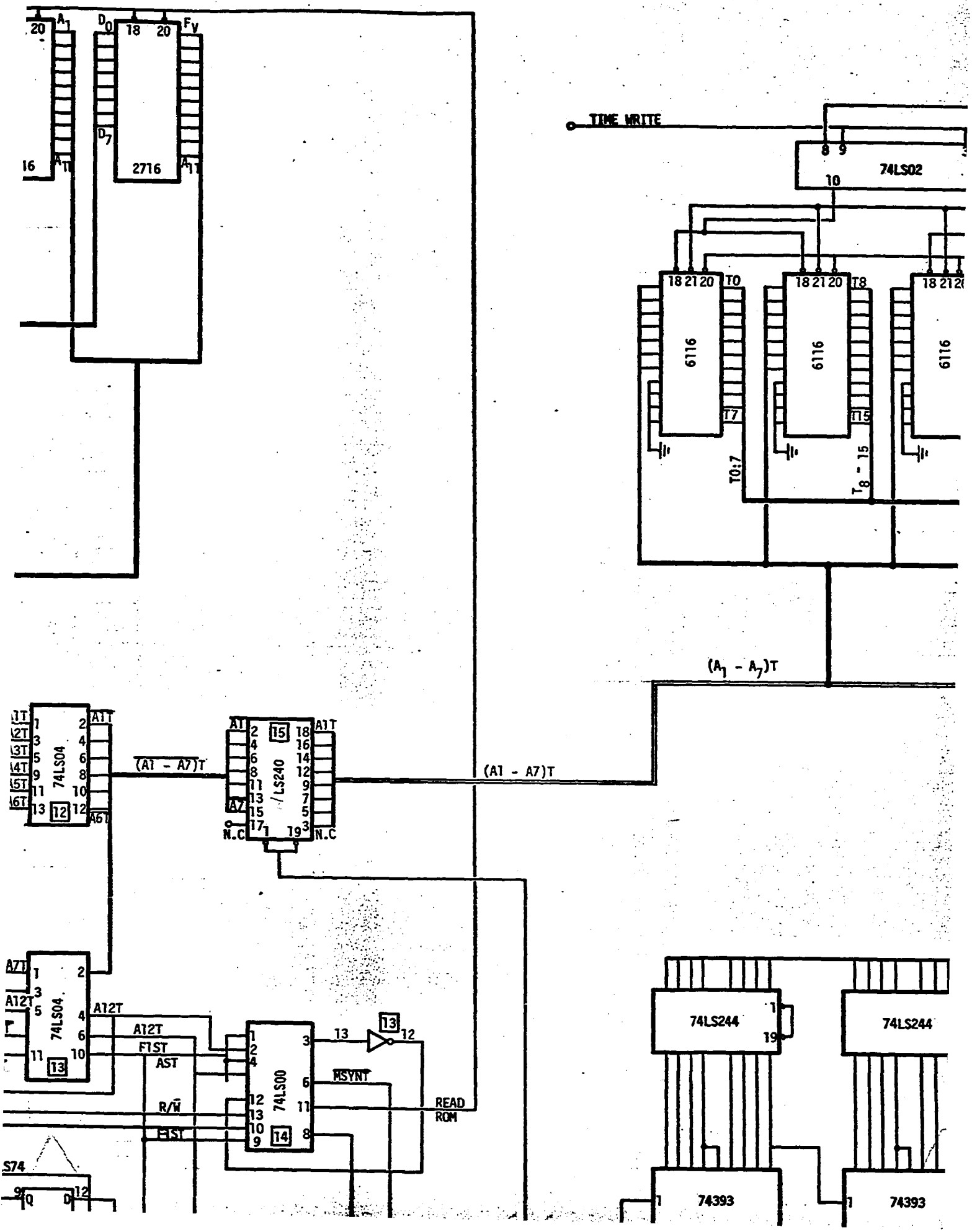
A set of simple programs were written to generate different sequences of addresses that were intended to test the response of the LRU submodule. A sample of these programs is shown in Figure 13. Since the address filtering and distribution circuits mentioned earlier have not really been built, it has been found necessary to simulate different arrival rates at the submodule in the design of the address generation circuit programs. The No Operation (NOP) instructions are used to substitute for filtered-out addresses. By controlling the number of the NOP instructions in an address generation loop, it is possible to control the arrival rate at the submodule. It is worth mentioning that in an actual situation, it is not expected that a valid address will arrive at the submodule every time an address is put on the main system's bus. The reason is that it is very unlikely that the main system will always switch from page to page after only one reference (locality of reference) [17]. Even if this happens, it is expected that, on the average, only 1/n of the references will be directed to a certain submodule, where n is the number of submodules in the LRU module. Thus, it is practical to assume that the arrival rate at any submodule will be, on the average, less than 1/n of the whole address stream rate on the main system's bus.

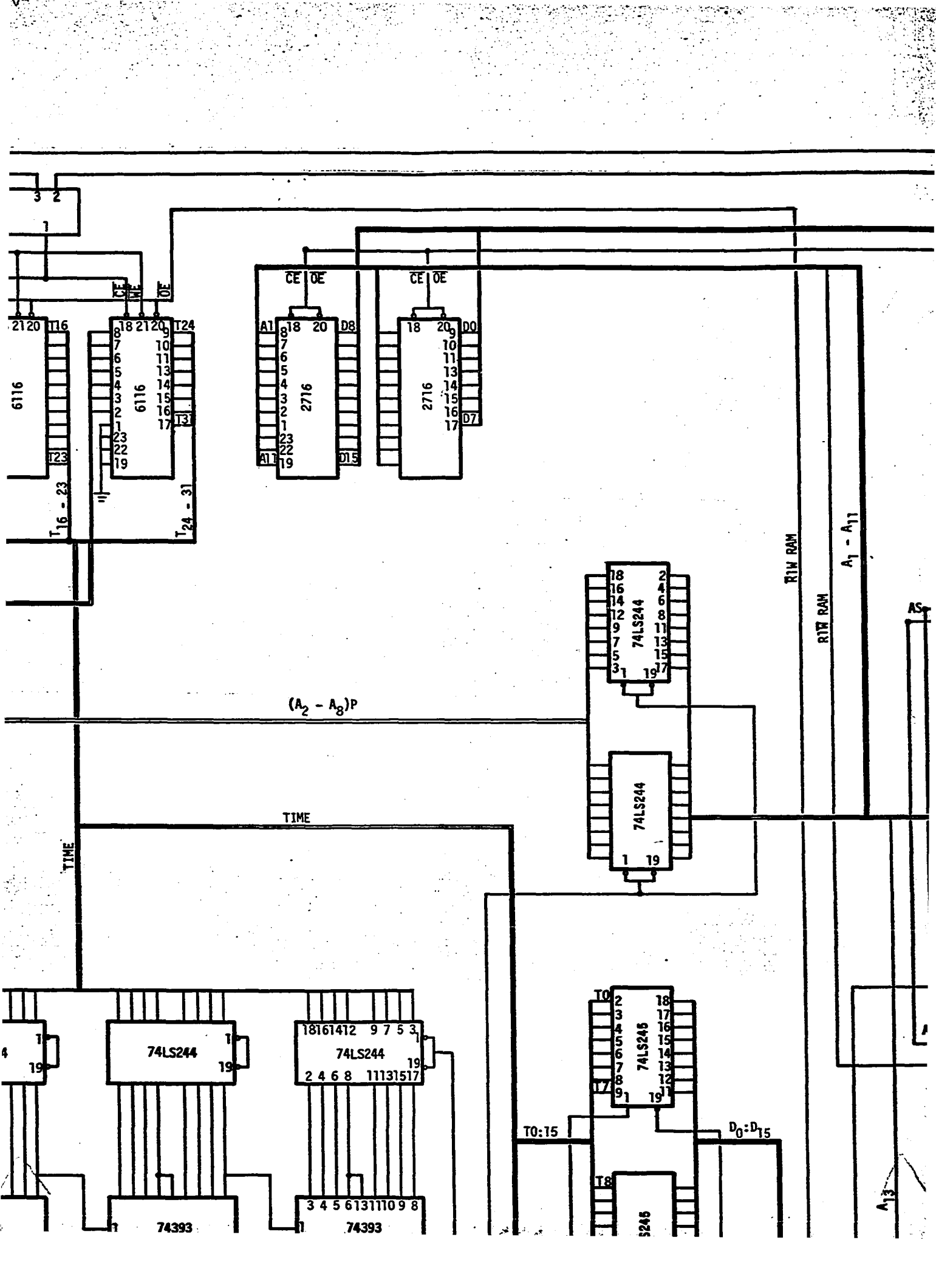
Detailed Circuit Diagram

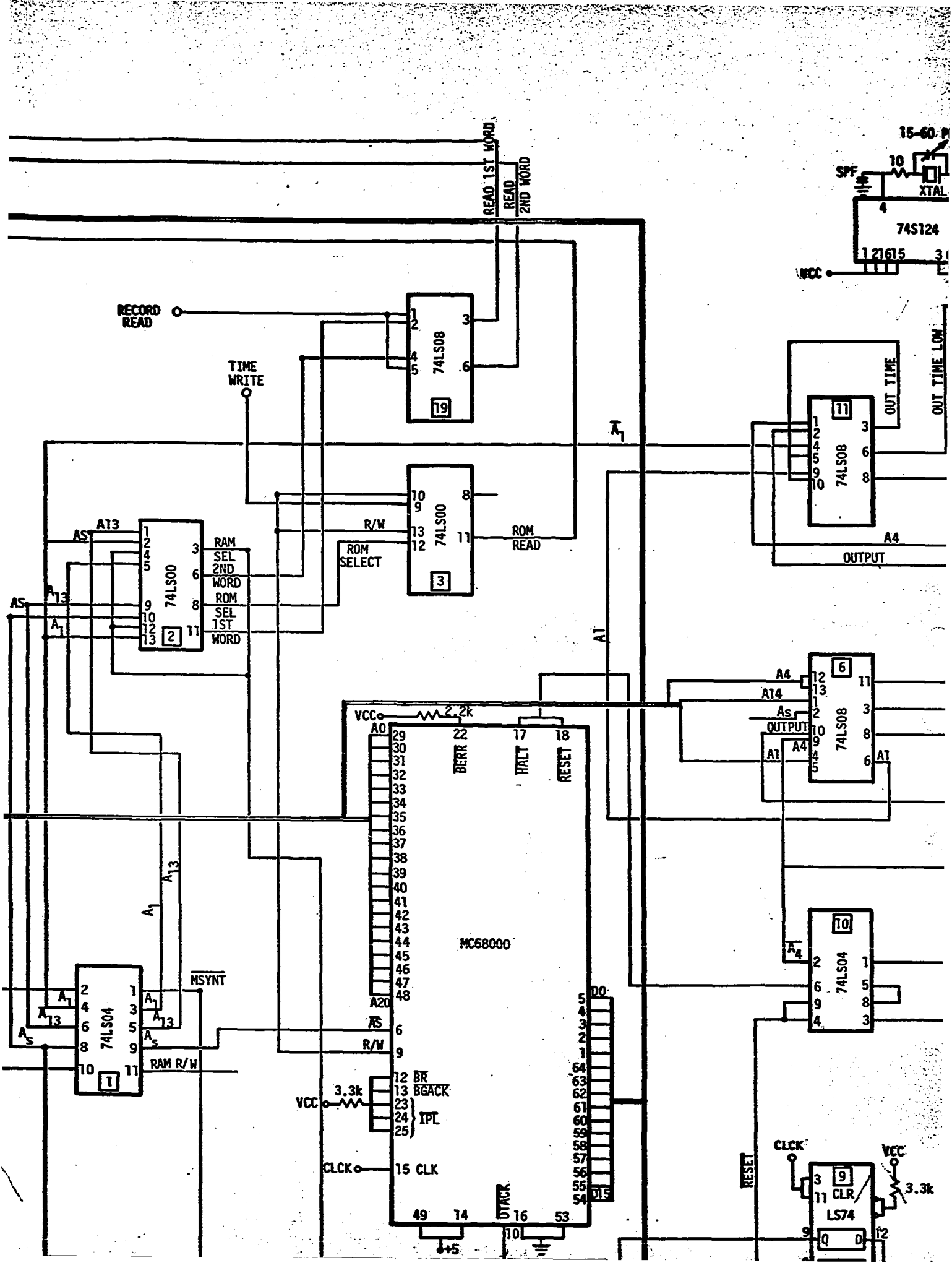
The detailed circuit diagram of the whole circuit including both the LRU submodule and the address generation circuit is shown in Figure 14, and a photograph of the built circuits is shown in Figure 15. Notice that the circuits shown in the photograph include the LRU submodule, the TIMER, the clock generator, the reset circuit, and the address generation

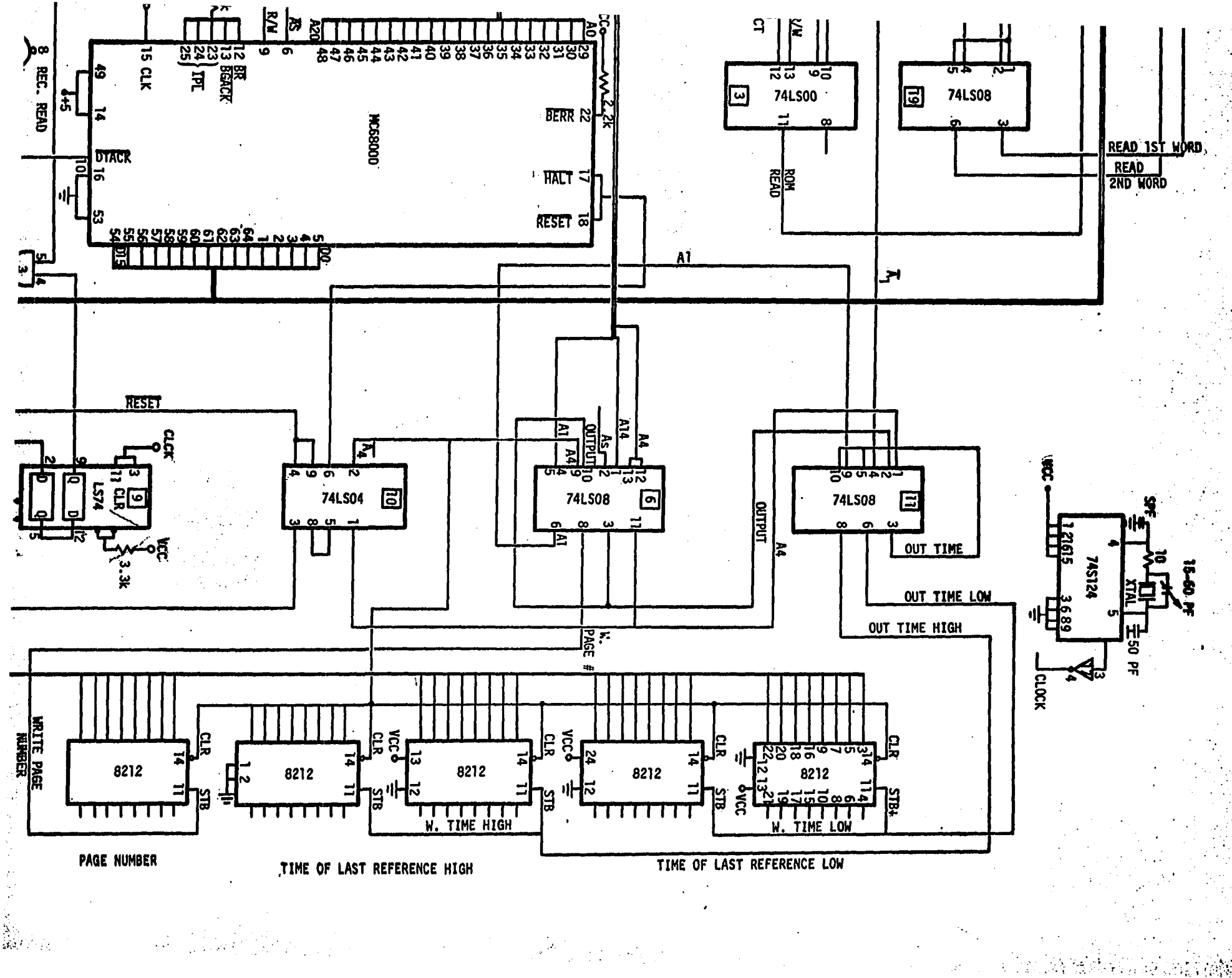
Figure 14. Detailed diagram of the LRU submodule and the address generation module

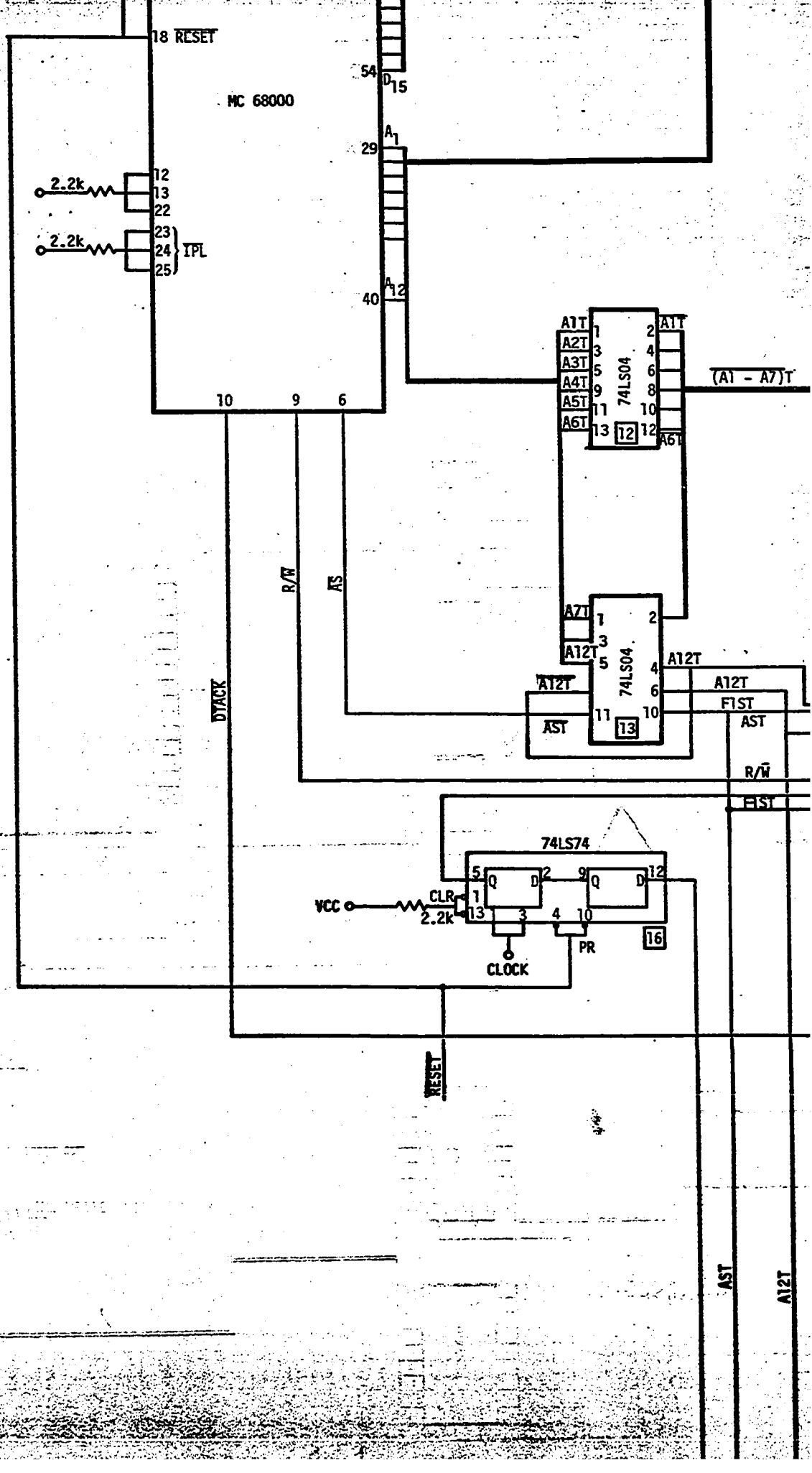

2716
VCC
16
53
14
49
CLOCK
CLK
15
17 HALT
18 RESET
MC 68000
2.2k
2.2k
IPL
12
13
22
23
24
25
10
9
6
40
29
54
R/W
AS
DTACK
74LS04
74LS04
74LS74
AST

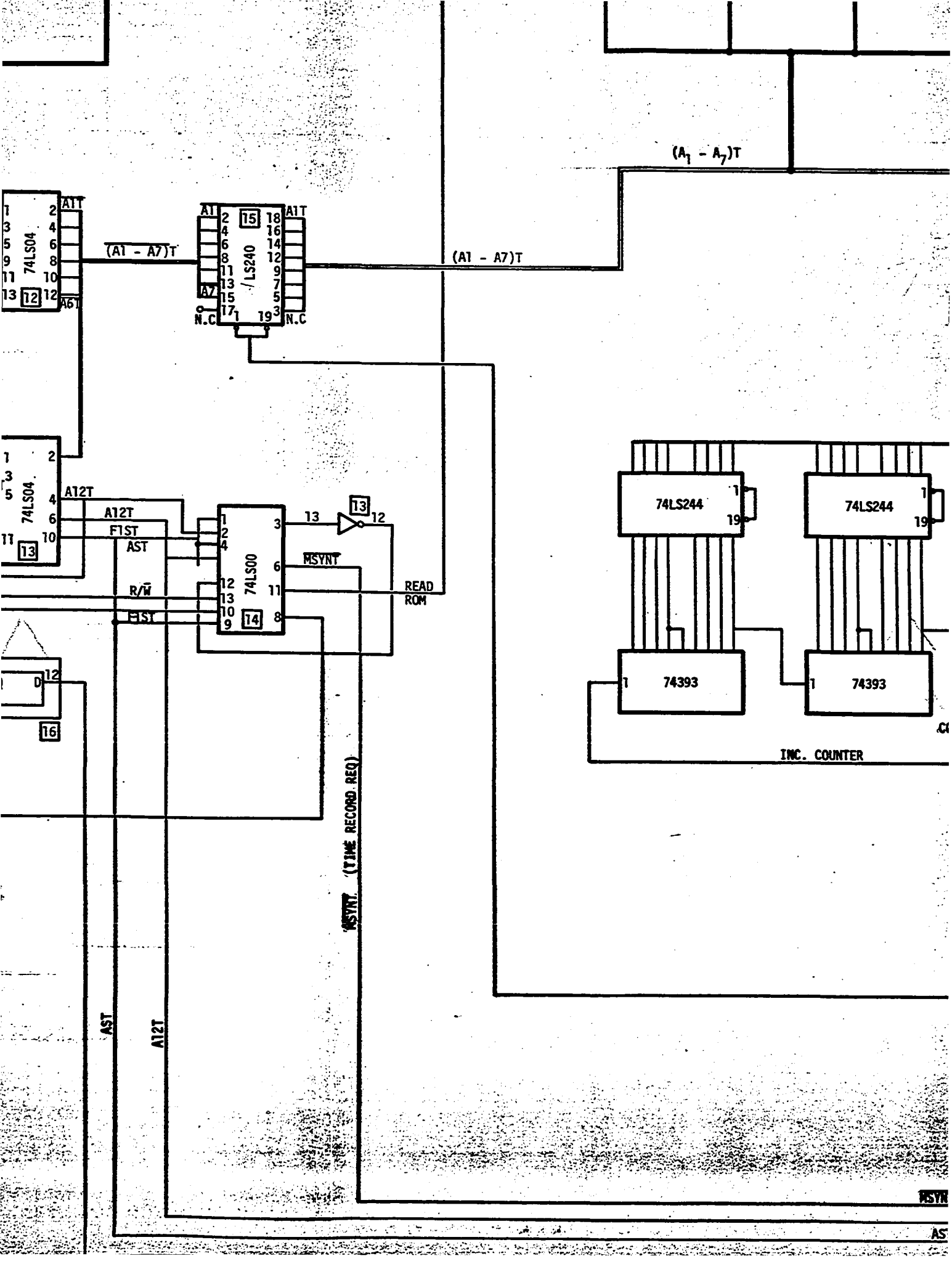

2716
TIME WRITE
74LS02
6116
6116
6116
T0:7
T8 - 15
(A1 - A7)T
74LS04
(A1 - A7)T
/LS240
(A1 - A7)T
74LS04
A12T
FIST
AST
R/W
FIST
74LS00
MSYNT
READ
ROM
74LS244
74LS244
74393
74393

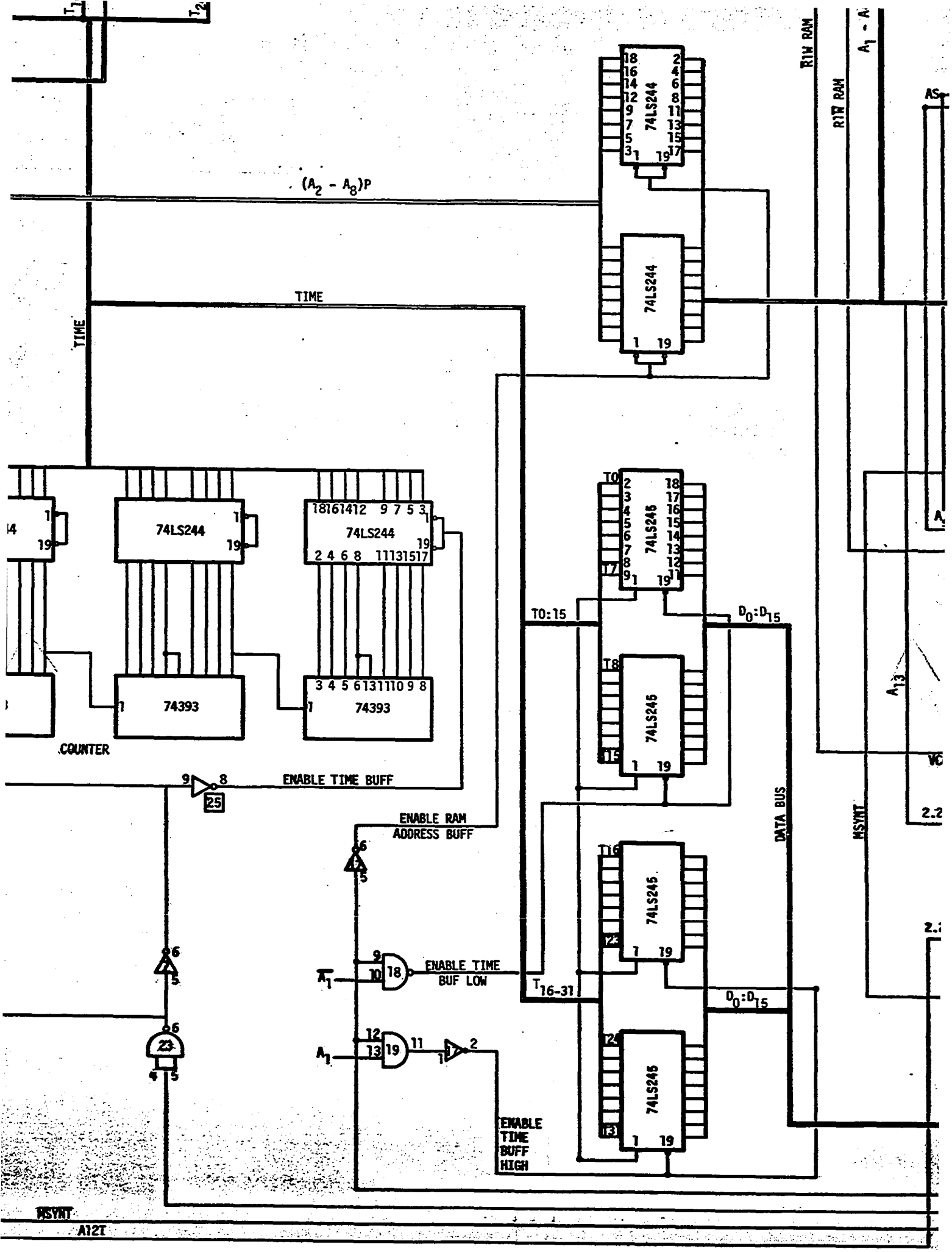

CE
WE
OE
6116
2716
T16 - 23
T24 - 31
R1W RAM
A1 - A11
AS
74LS244
(A2 - A8)P
TIME
74LS245
T0:15
D0:D15
74393
A13

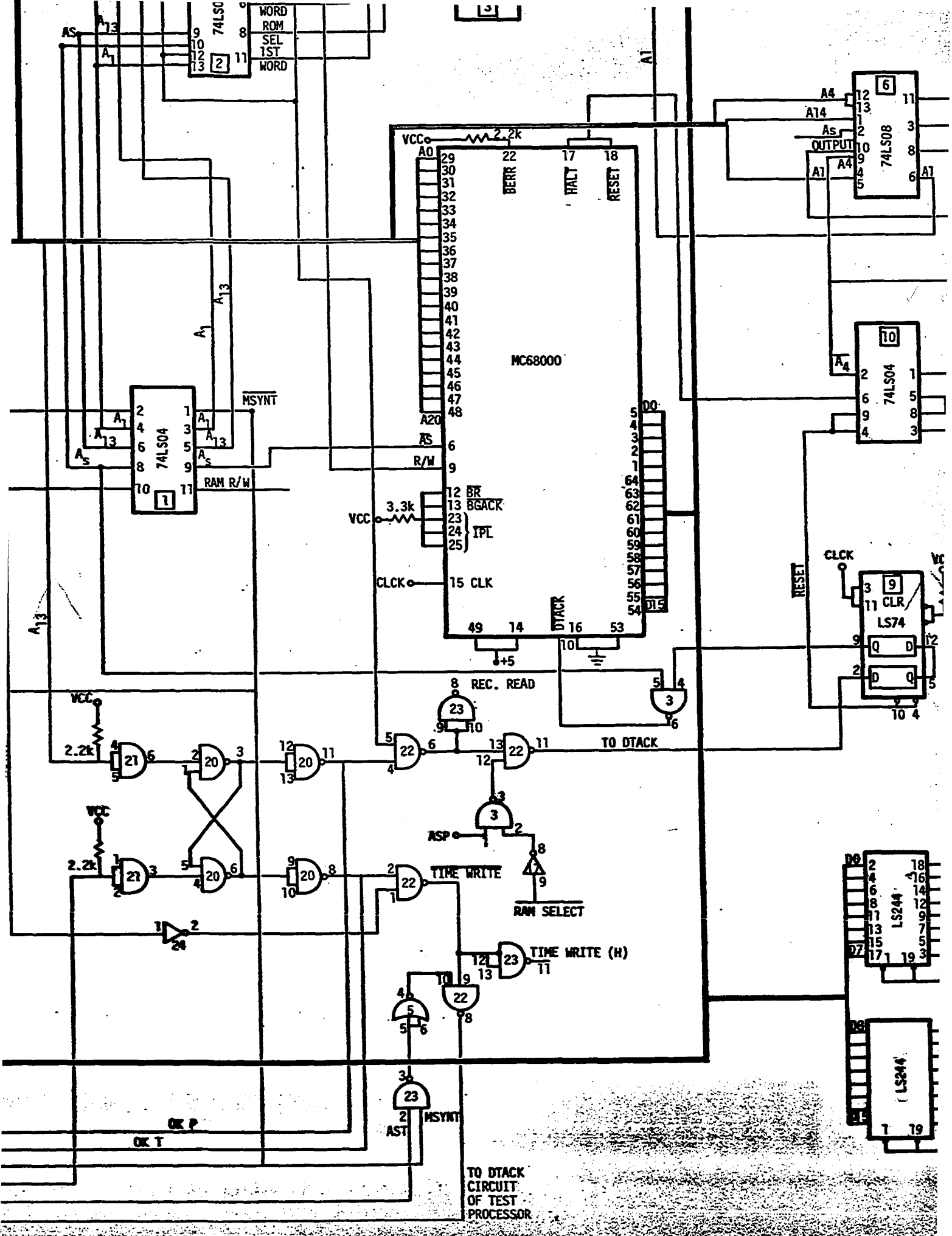

READ 1ST WORD
READ 2ND WORD
RECORD READ
TIME WRITE
74LS08
74LS00
74LS04
MC68000
RAM SEL 2ND WORD
ROM SEL 1ST WORD
ROM SELECT
ROM READ
R/W
AS
A13
A1
A4
A14
OUTPUT
OUT TIME
OUT TIME LOW
MSYNT
RAM R/W
VCC
2.2k
3.3k
CLCK
RESET
BERR
HALT
BR
BGACK
IPL
DTACK
74S124
SPF
XTAL
LS74
CLR


MC68000
74LS00
74LS08
READ 1ST WORD
READ
2ND WORD
ROM READ
RESET
74LS04
74LS08
74LS08
OUT TIME
OUT TIME LOW
OUT TIME HIGH
74S124
15-60 PF
XTAL
50 PF
CLOCK
8212
WRITE PAGE NUMBER
PAGE NUMBER
W. TIME HIGH
TIME OF LAST REFERENCE HIGH
W. TIME LOW
TIME OF LAST REFERENCE LOW


18 RESET
MC 68000
54 D15
29 A1
40 A12
2.2k
12
13
22
2.2k
23
24
25
IPL
10
9
6
A1T
A2T
A3T
A4T
A5T
A6T
74LS04
12
(A1 - A7)T
A7T
A12T
74LS04
13
A12T
AST
F1ST
R/W
74LS74
VCC
CLR
2.2k
CLOCK
PR
16
DTACK
R/W
AS
RESET
AST
A12T


(A1 - A7)T
A1T
74LS04
(A1 - A7)T
A1
A7
LS240
A1T
N.C
N.C
(A1 - A7)T
A12T
A12T
F1ST
AST
74LS00
MSYNT
READ ROM
R/W
F1ST
74LS244
74LS244
74393
74393
INC. COUNTER
MSYNT (TIME RECORD REQ)
AST
A12T
MSYR
AS


T1
T2
(A2 - A8)P
TIME
TIME
74LS244
74LS244
74LS244
74LS244
74393
74393
COUNTER
ENABLE TIME BUFF
ENABLE RAM
ADDRESS BUFF
ENABLE TIME
BUF LOW
ENABLE
TIME
BUFF
HIGH
T0:15
T16-31
74LS245
74LS245
74LS245
74LS245
D0:D15
D0:D15
DATA BUS
R1W RAM
R1W RAM
A1 - A
AS
MSYNT
MSYNT
A12T


WORD ROM SEL 1ST WORD
74LS08
MC68000
VCC
2.2k
3.3k
A0
A20
BERR
HALT
RESET
AS
R/W
BR
BGACK
IPL
CLK
DTACK
D0
D15
MSYNT
RAM R/W
74LS04
REC. READ
TO DTACK
ASP
TIME WRITE
RAM SELECT
TIME WRITE (H)
CLR
LS74
LS244
OK P
OK T
AST
TO DTACK CIRCUIT OF TEST PROCESSOR

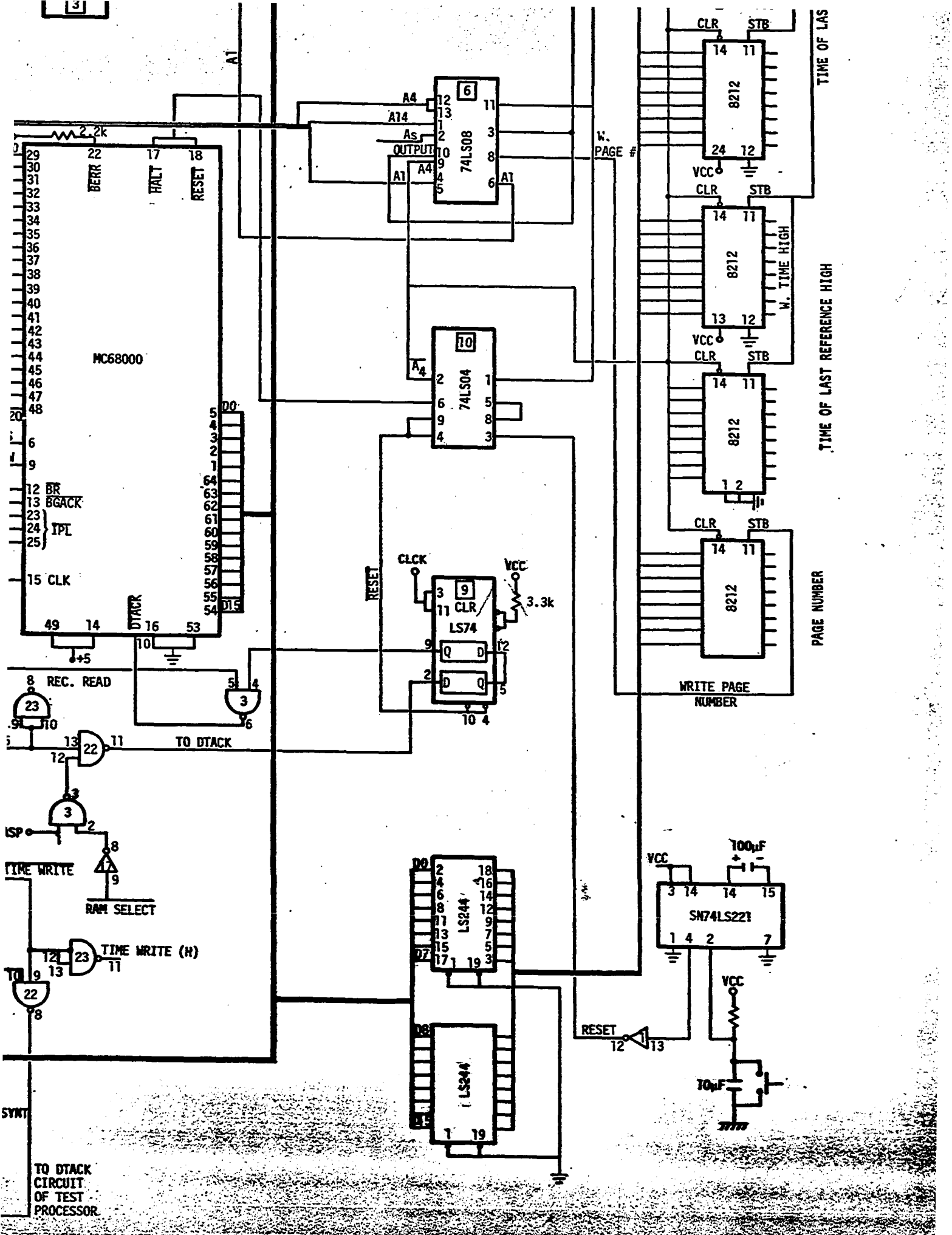
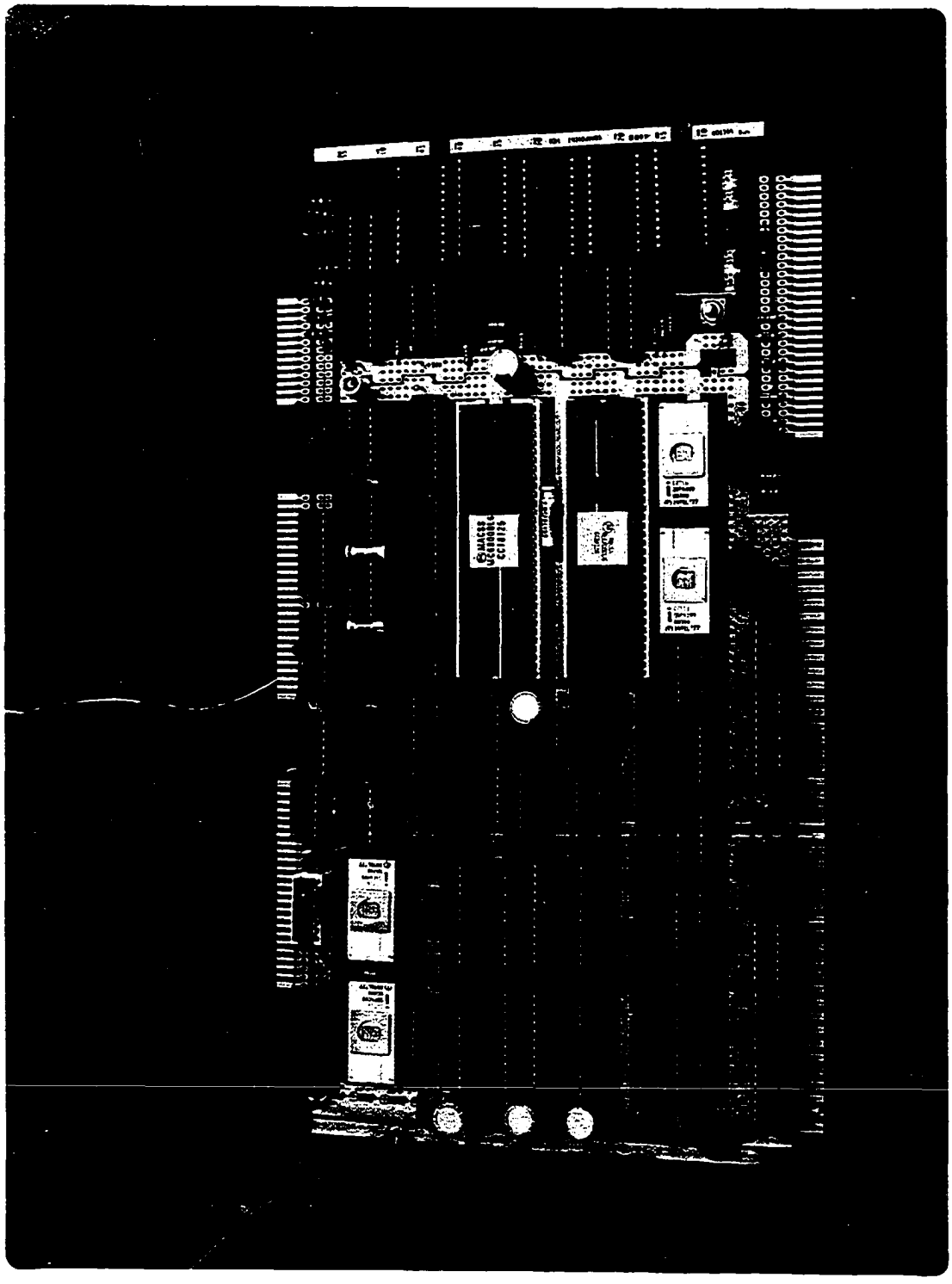

MC68000
2.2k
BERR
HALT
RESET
BR
BGACK
IPL
CLK
DTACK
+5
A1
A4
A14
As
OUTPUT
74LS08
74LS04
W. PAGE #
CLR
STB
8212
VCC
W. TIME HIGH
TIME OF LAST REFERENCE HIGH
PAGE NUMBER
WRITE PAGE NUMBER
CLCK
RESET
LS74
3.3k
REC. READ
TO DTACK
TIME WRITE
RAM SELECT
TIME WRITE (H)
LS244
SN74LS221
100µF
10µF
SYNT
TO DTACK CIRCUIT OF TEST PROCESSOR

Figure 15. A photograph of the built circuits



module. If the submodule were assembled on a separate printed circuit board, it should be possible to use a 5"x5" board. The whole LRU module could be assembled to occupy 6"x6"x10" at most. This means that it can nicely fit inside a modern disk drive assembly with no need for much larger space.

The LRU Routine

The LRU routine is designed to be executed by the submodule's MC68000 microprocessor. It has as an objective finding out the LRU page frame in the main memory area assigned to the submodule. The routine is written such that the number of references to the RAM is minimum. This is necessary to reduce the probability of a RAM access conflict as discussed before. By copying a time record to an internal register, it is easy to use the register in all comparison operations needed by the routine without having to reference the RAM again. A flow chart of the LRU routine is shown in Figure 16.

It has been found necessary to write more than one LRU routine to compare the performance of the submodule as the number of records stored inside the microprocessor varies. Therefore, three LRU routines, ROUTINE1, ROUTINE2, and ROUTINE3 were designed to work with different number of internally kept records. ROUTINE1 keeps only the record of the LRU page frame. ROUTINE2 keeps the LRU two records inside the microprocessor, while ROUTINE3 keeps the LRU three records. ROUTINE3 is shown in Figure 17, while ROUTINE1 and ROUTINE2 are given in Appendices A and B. The following discussion describes ROUTINE3; it also applies to ROUTINE1 and ROUTINE2 because of the similarity

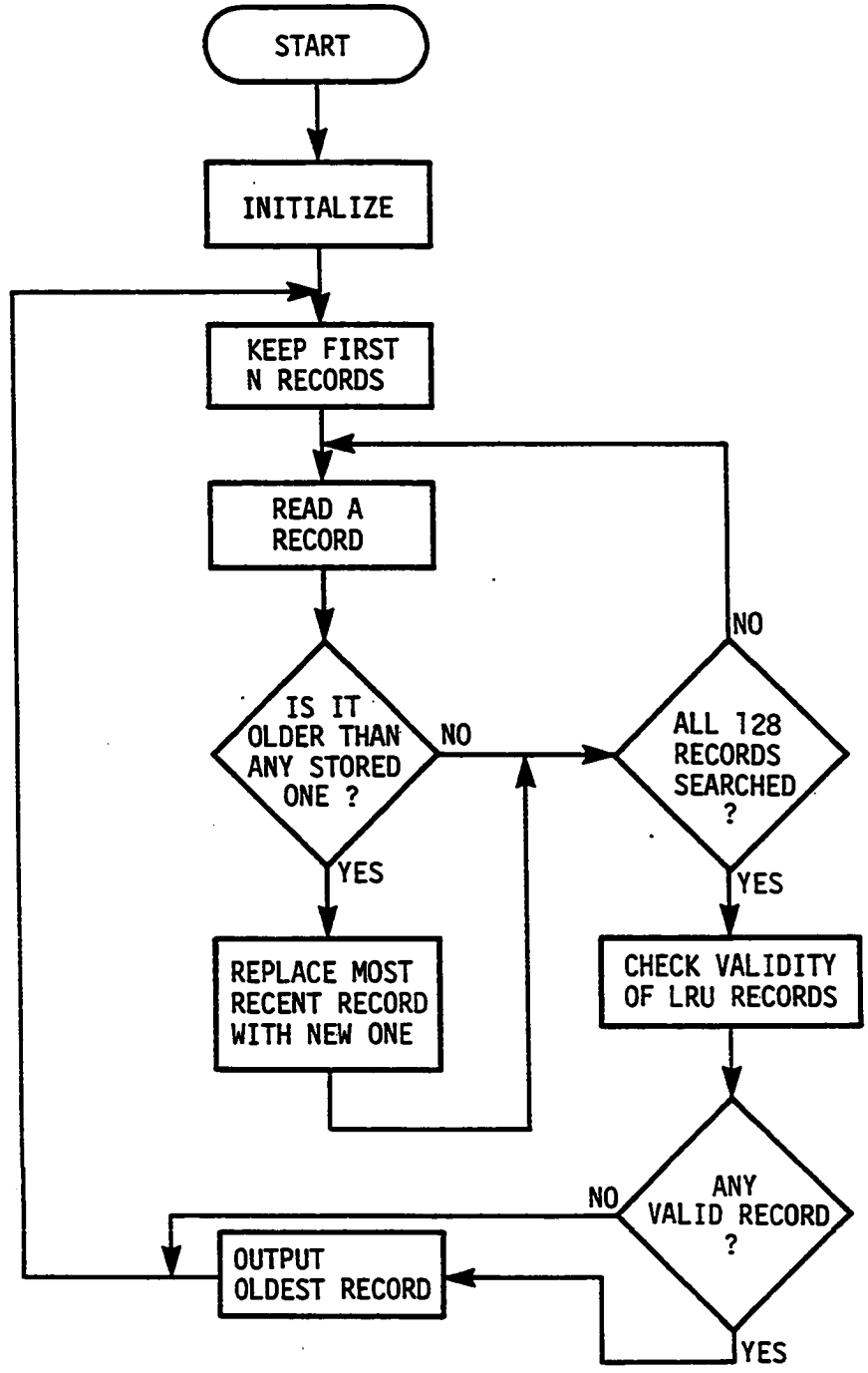


Figure 16. Flow chart of LRU routine

```
"68000"
;       EXACT LRU PROGRAM "ROUTINE3"
         ORG      0000H
         HEX      000,2300,000,0400
         HEX      000,0D00,000,0E00
         ORG      400H
INIT     MOVE.L   #700H,SR          ;INITIALIZE STATUS REGISTER.
         MOVE.L   #0,A0             ;CLEAR A0 THRU A2.
         MOVE.L   #0,A1
         MOVE.L   #0,A2
         MOVE.L   #22000H,A6        ;LOAD A6 WITH HIGHEST ADDRESS+4.
         MOVE.L   #22200H,A5        ;LOAD A5 WITH LOWEST ADDRESS.
         CLR.L    D7                ;D7 WILL BE USED TO CLEAR ALL
L1       MOVE.L   D7,-[A5]          ;128 RECORDS.
         CMPA.L   A6,A5             ;CLEAR ALL 128 RECORDS.
                  BHI               L1
         MOVE.L   #2200H,A5         ;REINITIALIZE A5&A6 FOR READ.
         MOVE.L   #2000H,A6
SEARCH   MOVE.L   -[A5],D0          ;READ FIRST THREE RECORDS AND
         MOVE.W   A5,A0             ;ORDER THEM.
         MOVE.L   -[A5],D7          ;D0 SHOULD HOLD THE OLDEST RECORD
         CMP.L    D0,D7             ;WITH ITS ADDRESS IN A0.
         BHI      L2                ;D1 &A1 SHOULD HOLD THE NEXT OLDEST
         MOVE.L   D0,D1             ;RECORD AND ITS ADDRESS RESPECTIVELY.
         MOVE.W   A0,A1             ;D2&A2 SHOULD HOLD THE LAST RECORD
         MOVE.L   D7,D0             ;AND ITS ADDRESS RESPECTIVELY.
         MOVE.W   A5,A0
         BRA      L3
L2       MOVE.L   D7,D1
         MOVE.W   A5,A1
L3       MOVE.L   -[A5],D7
         CMP.L    D0,D7
         BHI      L4
         MOVE.L   D1,D2
         MOVE.W   A1,A2
         MOVE.L   D0,D1
         MOVE.W   A0,A1
         MOVE.L   D7,D0
         MOVE.W   A5,A0
         BRA      NSRCH
L4       CMP.L    D1,D7
         BHI      L5
         MOVE.L   D1,D2
         MOVE.W   A1,A2
         MOVE.L   D7,D1
         MOVE.W   A5,A1
         BRA      NSRCH
```

Figure 17. Exact LRU program "ROUTINE3"

```
L5      MOVE.L  D7,D2
        MOVE.W  A5,A2
NSRCH   MOVE.L  -[A5],D7        ;READ A TIME RECORD INTO D7
        COMP.L  D2,D7           ;IF NOT OLDER THAN THE ONE IN D2
        BHI     TSTEND          ;IGNORE IT.
        CMP.L   D1,D7           ;IT IS OLDER
        BHI     L6              ;REORDER THE LIST AS ABOVE
        CMP.L   D0,D7           ;DISCARD D2&A2
        BHI     L7
        MOVE.L  D1,D2
        MOVE.W  A1,A2
        MOVE.L  D0,D1
        MOVE.W  A0,A1
        MOVE.L  D7,D0
        MOVE.W  A5,A0
        BRA     TSTEND
L6      MOVE.L  D7,D2
        MOVE.W  A5,A2
        BRA     TSTEND
L7      MOVE.L  D1,D2
        MOVE.W  A1,A2
        MOVE.L  D7,D1
        MOVE.W  A5,A1
TSTEND  COMPA   A5,A6           ;ALL 128 RECORDS SEARCHED?
        BNE     NSRCH           ;IF NOT GO BACK TO NSRCH.
OUTPUT  CMP.L   [A0],D0         ;CHECK VALIDITY OF D0.
        BNE     L8              ;IF NOT VALID GO TO L8
        MOVE.L  A0,D4           ;IT IS VALID.
        LSR.L   #1,D4           ;MAP ADDRESS BACK TO INPUT
        MOVE.W  D4,4000H        ;ADDRESS AREA AND OUTPUT IT.
        MOVE.L  D0,4010H        ;OUTPUT ITS TIME RECORD.
        BRA     ENDOUT
L8      CMP.L   [A1],D1         ;CHEK VALIDITY OF D1 AND OUTPUT IF
        BNE     L9              ;VALID.IF NOT GO TO L9.
        MOVE.L  A1,#4
        LSR.L   #1,D4
        MOVE.W  D4,4000H
        MOVE.L  D1,4010H
        BRA     ENDOUT
L9      CMP.L   [A2],D2         ;CHECK VALIDITY OF D0 AND OUTPUT IF
        BNE     ENDOUT          ;VALID.  IF NOT VALID NO OUTPUT IS
        MOVE.L  A2,D4           ;PRODUCED.
        LSR     #1,D4
        MOVE.W  D4,4000H
        MOVE.L  D2,4010H
ENDOUT  MOVE.W  #2200H,A5       ;REINITIALIZE A5.
        BRA     SEARCH          ;START A NEW SEARCH
```

Figure 17. (Continued)

```
        ORG     0D00H
BUSERR  MOVE.L  #0FFFFFFFFH,4010H ;BUS ERROR HANDLER.
        MOVE.L  #0400H,0CH[A7]
        RTE

        ORG     0E00H
ADDERR  MOVE.L  #0FFFFFFFFH,4010H ;ADDRESS ERROR HANDLER.
        MOVE.L  #0400H,0CH[A7]
        RTE
```

Figure 17. (Continued)

between all three routines except for the number of internally stored records.

The routine starts by initializing the status register to the user mode and an interrupt level of seven. A zero is moved to address registers A0, A1, and A2 in a long word instruction in order to clear the most significant bits in particular. This is necessary to avoid using long word instructions thereafter where word instructions can be used to reduce the execution time. Since only address lines A1P through A9T are used to address the RAM when A13P is asserted, the address space occupied by all 128 time records is 2000 through 21FF in hexadecimal. However, because of the availability of very large address space, address line A17P is used to differentiate between RAM read and RAM write operations such that when A17P is high, the operation is write; otherwise, it is a read operation. The only time the RAM is written into by the microprocessor is during initialization. Thus, address register A5 is initialized to 22200 hex which corresponds to the highest record address plus four. This is because A5 is used to step through the time records in a pre-decrement long word mode. A6 is initialized to 2200 hex which is the lowest time record address for a RAM write operation. Data register D7 is then cleared and used to clear all 128 records in a loop that starts at L1. After clearing all time records, registers A5 and A6 are reinitialized for RAM reads. A5 is loaded with 2200 hex, and A6 is loaded with 2000 hex. The SEARCH part is an initial step in the overall search process. It records the first three encountered time records (the highest address

time records), and at the same time sorts them in an ordered list. As a result, data register D0 holds the oldest referenced page frame record while address register A0 holds its address. D1 and A1 hold the next oldest time record and its frame address, respectively. D2 and A2 contain the last referenced time record and its address.

After initializing the search process, the NORMAL SEARCH (NSRCH) begins. It reads a time record into D7. It then compares it with the most recently referenced frame record stored in D2. If the new record has a higher value, then it has no importance since it corresponds to a page frame that has been referenced more recently than any of the three frames whose records are kept inside the microprocessor. In such a case, the time record and its address are ignored. If the time record is less than that in D2, then it must be considered. A comparison with the record in D0 and possibly D1 determines the new ordered list. The old contents of D2 and A2 are discarded and the new list is stored such that D0, D1, and D2 hold the time records, while A0, A1, and A2 hold the corresponding addresses in the same order discussed above. Address register A5 is used as a pointer that steps through the list in a pre-decrement mode. At TESTEND, the routine checks the completion of searching all 128 records by comparing A5 to A6 which holds the lowest address. If they match, then all 128 records have been searched; if A5 and A6 do not match, the routine branches back to NSRCH to continue the search. When all 128 records have been searched, the routine starts the output process.

It might happen that a time record gets changed after being

considered by the routine. Therefore, it is important to check the validity of a record before producing it as the output. The output process starts by checking the validity of the time record stored in D0 (the TR of the LRU page frame) by comparing D0 to the current value for the record in the RAM. If valid, the contents of A0 are outputted as the LRU page frame address, and the contents of D0 are outputted as the corresponding time record. If D0 is not valid, D1 is checked for validity and outputted along with A1, if it is still valid. If D1 is not valid, D2 is checked. If valid, A2 and D2 are outputted. If none of the records is valid, no output is produced and the routine jumps back to NSRCH to start a new search after loading A5 with 2200 hex as discussed before. It is worth mentioning that a page frame address is logically shifted right one bit before being outputted. This maps the output page frame address back to the original address space assigned to the submodule (1000 hex through 10FE hex) instead of 2000 hex through 21FC hex, as seen by the submodule. Addresses 4000 and 4010 are assigned to the output latches.

An LRU Module Overview

Although a whole LRU module has not been built, it is rather easy to build, especially since most of the module consists basically of copies of the designed and built submodule. The only part of the module that deserves more discussion is the supervisor submodule. As mentioned earlier, the primary responsibility of the supervisor submodule is to find out the overall page frame in main memory from among LRU area page frames produced by the other submodules. Thus, if we assume, as before,

that main memory is divided into eight equal areas of 128 frames each, then the supervisor has to search only the eight records produced by the area submodules. Therefore, it is clear that the amount of work that needs to be done by the supervisor is only 1/16 of that done by an area submodule. This implies the following:

(1) The output rate of the module is almost the same as that of a submodule.

(2) The potential access conflicts between an area submodule and the supervisor submodule for output latch access should be decided in favor of the area submodule.

The second item is very important in order not to slow down the overall speed of the module. Thus, in order to meet this demand, straight arbitration must be excluded and another way to solve potential conflicts must be considered. A technique that is simply, practical, and easy to implement will now be described.

For each submodule, a flip-flop that is automatically set whenever the submodule is writing an output latch is used. The latches used in the area submodule built are Intel's 8112s which have 3-state outputs. All latch outputs can then be directly connected to the supervisor's data bus. Since the supervisor microprocessor need only read the latches (and not write into them), it is possible to freely read any latch at any time by just enabling its outputs using proper addressing, as shown in Figure 18. It might happen that the submodule microprocessor is writing the latch while the supervisor is reading the latch at the same time. There are two simple ways to solve this problem. The first

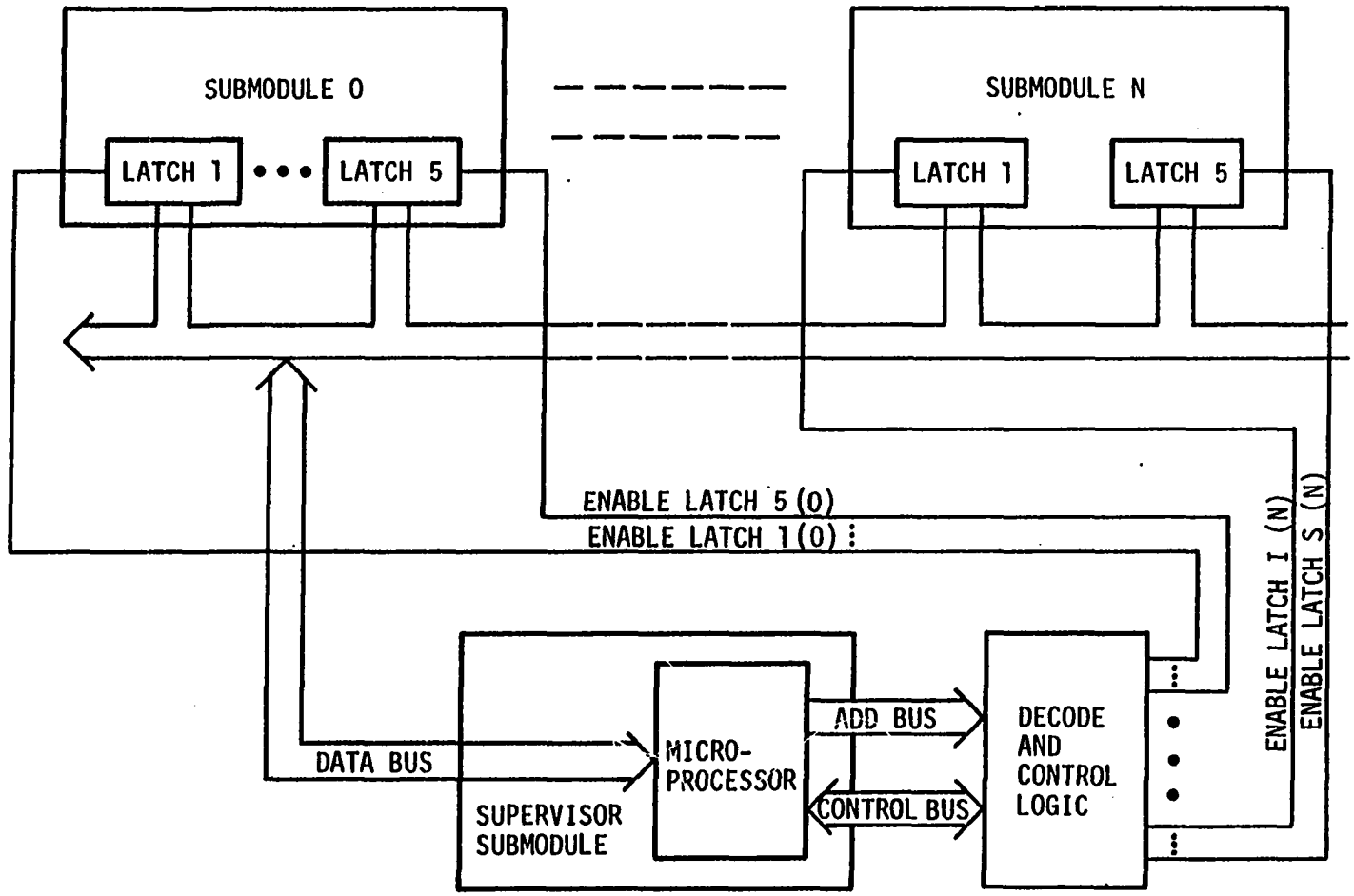


Figure 18. Possible connection between output latches and the supervisor submodule

utilizes the above-mentioned flip-flop. The supervisor has to check the flip-flop after each latch read operation. If the flip-flop is set, the supervisor clears the flip-flop and repeats the read cycle. The process is repeated until the flip-flop is found reset after the cycle. It might seem that the supervisor would be slowed down too much especially since it has to make 24 latch reads before producing an output. That is not exactly true, because in any given latch read cycle, the supervisor is dealing with only one submodule and the latter writes the latch less than 1% of the time. Therefore, it is very unlikely that the supervisor would have to repeat a latch read cycle more than one time before getting a correct read cycle. Moreover, slowing down the supervisor is not a problem even operating at 10% of its maximum rate, since the output production rate of the supervisor would still be faster than that of a submodule. Note that the work load of the supervisor is approximately 1/16 that of an area submodule for reasons mentioned above.

The second approach is even simpler than the first and can be implemented by just letting the supervisor read a certain latch two consecutive times and compare the values read. If the values agree, it goes on; if not, it reads the latch again until an agreement is found between the last two read cycles. It is worth mentioning, however, that at most it would have to read a latch a maximum of four times before a match is found. This happens when the submodule's micro writes the latch during the second read cycle by the supervisor. In such a case, the supervisor would have to do another two read cycles to the

same latch at most. This is because once a specific latch is written into by an area submodule, the next write cycle to the same latch is not less than 128 RAM references away. Thus, it seems that even the second simpler technique can, at most, slow down the supervisor submodule to work at no less than 1/4 of its maximum possible speed. Figure 18 shows a possible connection of the output latches to the supervisor submodule data bus.

CHAPTER VI. ADDRESS GENERATION ROUTINES, EXPERIMENTAL DATA, AND SOME REMARKS

In this chapter, the routines designed to run on the address generation microprocessor will be discussed first. The data obtained from testing the LRU submodule using different combinations of address generation routines and LRU routines will then be given. The experimental data provide very important performance figures, hence some remarks and observations will be introduced. We will discuss the possibility of designing an LRU module that represents less loading on the supported system than the one described earlier which provides output at a faster rate than actually needed.

Address Generation Routines

In order to test the performance of the LRU submodule, it is essential to simulate the address stream of the main system. This implies the need for different address sequences with different characteristics such as address generation rate and the time period during which some addresses are deliberately skipped to simulate unreferenced page frames. Therefore, four address generation routines have been designed, each of which produces a sequence that exhibits some specific characteristics. The routines are ADDGEN1, ADDGEN2, ADDGEN3, and ADDGEN4. The last one is shown in Figure 16, while the other three are given in Appendices C, D, and E, respectively. However, in order to enable easy understanding of the experimental data, the address sequences produced by the routines are shown in Figures 19 through 23. The skipped addresses are

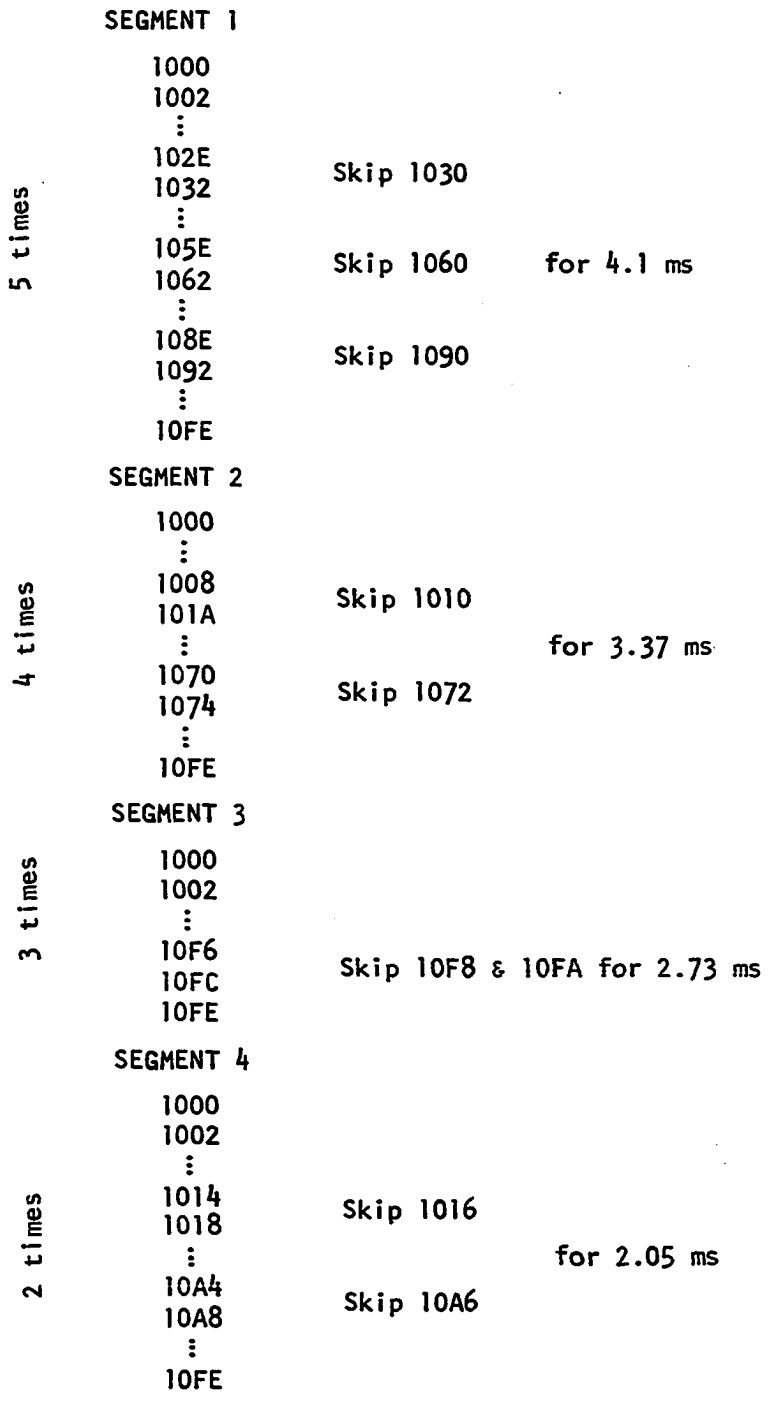


Figure 19. ADDGEN1 sequence (one NOP instruction in each loop)

SEGMENT 5

Don't repeat

1000		
1002		
⋮		
10C0		
10C4	Skip 10C2	for 1.4 ms
⋮		
10FE		

Figure 19. (Continued)

5 times	1000 1002 ⋮ 102E 1032 ⋮ 10FC 10FE	Skip 1030	for 5.4 ms
3 times	1000 1002 ⋮ 1070 1076 ⋮ 10FE	Skip 1072, 1074	for 3.7 ms
3 times	1000 1002 ⋮ 10F6 10FE	Skip 10F8, 10FA, and 10FC	for 2.7 ms
2 times	1000 1002 ⋮ 1014 1018 ⋮ 10FE	Skip 1016	for 2.7 ms
4 times	1000 1002 ⋮ 10C0 10C4 ⋮ 10FE	Skip 10C2	for 3.6 ms

Figure 20. Address sequence generated by ADDGEN2 (three NOP instructions are used)

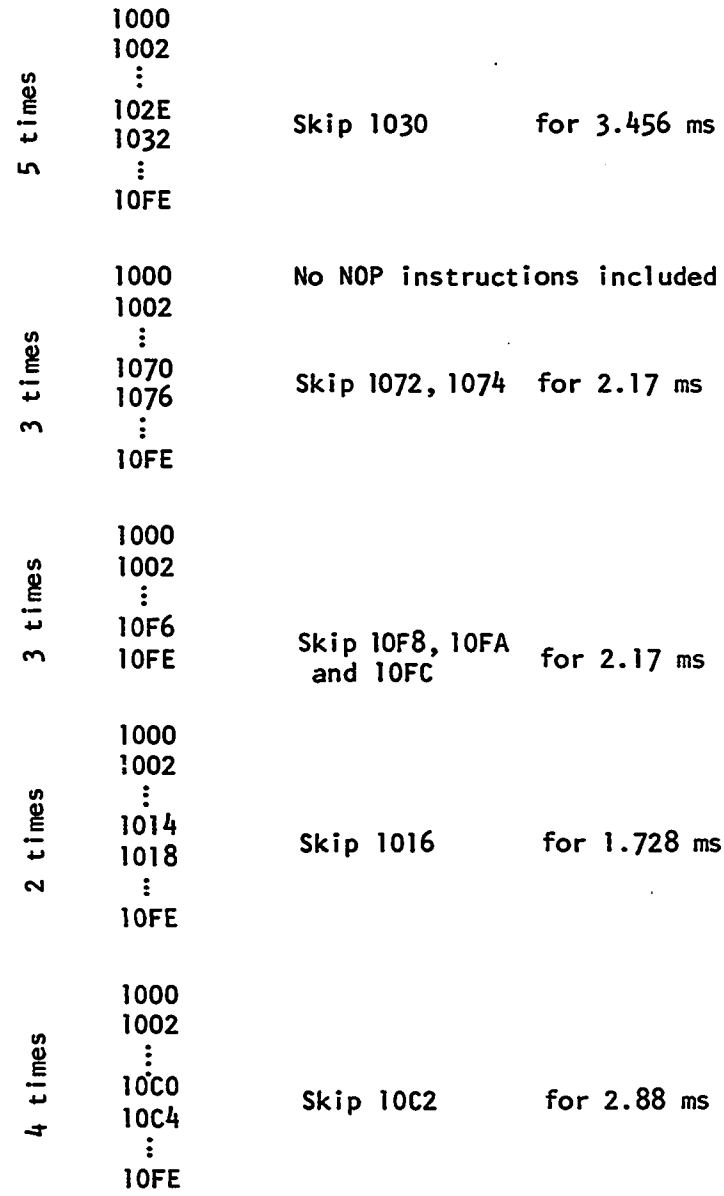


Figure 21. Address sequence generated by ADDGEN3 (no NOP instructions are used)

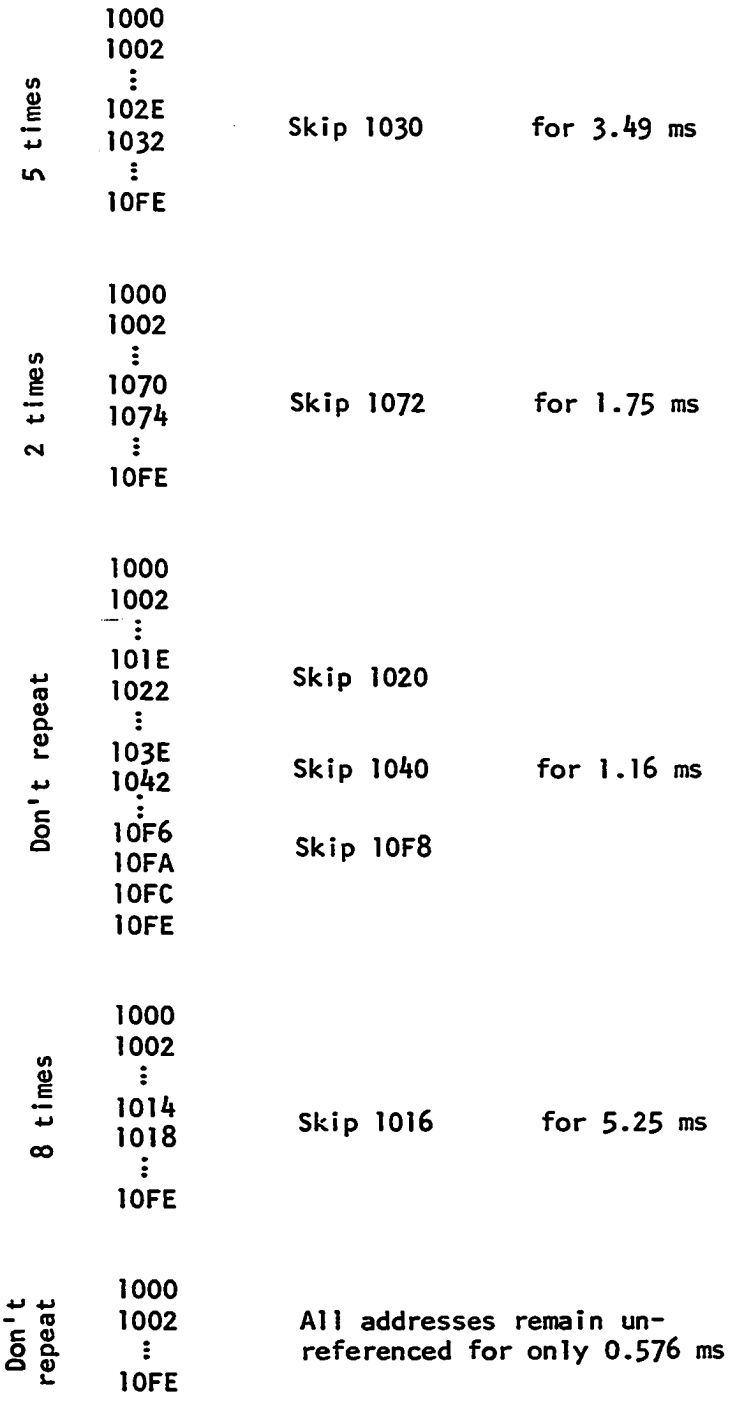


Figure 22. ADDGEN4 sequence (no NOP instructions included in the loops)

pointed out along with the time period during which these addresses remain unreferenced.

Each routine is composed of five segments, and each segment produces the hexadecimal addresses 1000 through 10FE in increments of two starting at 1000. It is to be pointed out again that the address line A_{0T} of the MC68000 microprocessor is used internally to select bytes. Therefore, address lines A_{1T} through A_{7T} are used to simulate the page frame address within the main memory area assigned to the LRU submodule. Thus, address lines A_{1T} through A_{7T} can change between 0000000 and 1111111 giving the desired 128 distinct addresses. Address line A_{12T} is used to differentiate ROM references within the address generation circuit ($A_{12T}=0$), and addresses that simulate main memory address stream ($A_{12T}=1$). Word instructions are used to produce the desired sequences, making A_{0T} insignificant to the operation. This explains the selected hexadecimal address range 1000 through 10FE.

To control the address generation rate and hence the arrival rate at the LRU submodule, the NO OPERATION (NOP) instructions have been utilized to add some deliberate delay between consecutive addresses. The number of NOP instructions is the same in all five segments of a certain routine. This means that the address generation rate of a certain routine is almost constant. However, the number of NOP instructions used in different routines is not constant and is as follows:

Routine	Number of NOP instructions
ADDGEN1	1
ADDGEN2	3
ADDGEN3	0
ADDGEN4	0

Within a certain segment, some addresses are skipped to simulate a page frame that is not referenced. The time period during which a certain address is skipped is controlled by the number of times a segment is repeated before moving to the next segment. Some segments are repeated up to eight times before moving to the next segment, while some segments are executed only one time followed immediately by the next segment.

After all five segments are executed, the routine jumps back to the first segment and the process is repeated indefinitely.

All that is required from the address generation circuit is to put the desired sequence of addresses on the address bus. This is done by making the address generation routines reference a non-existing list. Note that no RAM is employed in the address generation circuit. The routines use address register A_0 as a pointer to the list. Word instructions are used to read words between addresses 1000 and 10FE (hex) into data register D_0. This results in the address sequence to be put on the address bus and directed to the LRU submodule whenever A_{12T} is high.

Experimental Data

A total of twelve experiments have been executed. Each experiment corresponds to a different address generation routine and an LRU routine combination. The data recorded in each experiment represent the first 64 output records produced by the LRU submodule. A record consists of two parts, the LRU page frame address and the time of its last reference (its time record). It must be noted that although 8 bits can properly represent the page frame address, it is more convenient to record a 16-bit

Table 1. Key to different experimental data tables

	ROUTINE1	ROUTINE2	ROUTINE3
ADDGEN1	exp. 1 / Table 2	exp. 2 / Table 3	exp. 3 / Table 4
ADDGEN2	exp. 4 / Table 5	exp. 5 / Table 6	exp. 6 / Table 7
ADDGEN3	exp. 7 / Table 8	exp. 8 / Table 9	exp. 9 / Table 10
ADDGEN4	exp. 10 / Table 11	exp. 11 / Table 12	exp. 12 / Table 13

address word as produced by the LRU submodule to allow easy comparison with the address sequence produced by the address generation circuit. However, only the low order byte of the frame address is stored in an 8-bit latch which is adequate for the LRU module operation. It must also be noted that a time record is 32 bits long (because the TIMER is 32 bits long); however, only the low order 16 bits have been recorded since the high order 16 bits remain all zeros when the first 64 records are recorded. A time record represents a reference number rather than actual time. This is because the TIMER is incremented each time a valid address arrives to the submodule, and thus the TIMER contents represent the reference number.

The data obtained in the twelve experiments are recorded in twelve tables. To facilitate easy reference to the data of some experiment, Table 1 has the experiment number and its data table number as the entry, with the address generation routines and the LRU routines as ordinates. For instance, experiment 5 used ADDGEN2 as the address generation routine and ROUTINE2 as the LRU routine and the table that contains the output data is Table 6. All data are in hexadecimal format.

Two HP 1602 logic analyzers have been utilized to record the data. The data lines of both analyzers were connected to the data bus of the LRU submodule's microprocessor, while the signals that strobe the output latches were used for clocking the analyzers appropriately.

The first observation from the data is that the LRU submodule works properly. The output page address produced corresponds to some skipped addresses in the address sequence generated by the address generation

Table 2. ADDGEN1 and ROUTINE1 (one NOP instruction)

	Address (hex)	Time		Address	Time
1	1030	0000	33	10A6	24AD
2	1030	0000	34	1030	26E9
3	1030	0000	35	1030	26E9
4	1010	0279	36	1010	29C9
5	1010	0279	37	1010	24C9
6	1010	0279	38	1072	29F8
7	10F8	0560	39	10F8	2CB0
8	10F8	0560	40	1016	2E39
9	10C2	0867	41	1030	30BD
10	1030	096D	42	1030	30BD
11	1030	096D	43	1030	30BD
12	1030	096D	44	1010	339D
13	1010	0C4D	45	1010	339D
14	1010	0C4D	46	10F8	3684
15	10F8	0F34	47	10F8	3684
16	1016	10BD	48	10A6	3855
17	1030	1341	49	10C2	39DB
18	1030	1341	50	1030	3A91
19	1030	1341	51	1030	3A91
20	1010	1621	52	1010	3D71
21	1010	1621	53	1010	3D71
22	1010	1621	54	10F8	4058
23	10F8	1908	55	1016	41E1
24	10F8	1908	56	1030	4465
25	10C2	1C5F	57	1030	4465
26	1030	1D15	58	1030	4465
27	1030	1D15	59	1010	4745
28	1030	1D15	60	1010	4745
29	1010	1FF5	61	1010	4745
30	1010	1FF5	62	10F8	4A2C
31	10F8	22DC	63	10F8	4A2C
32	10F8	22DC	64	10C2	4D83

Table 3. ADDGEN1 and ROUTINE2

	Address (hex)	Time (hex)		Address	Time
1	1030	0000	33	1030	26E9
2	1030	0000	34	1030	26E9
3	1010	0279	35	1030	26E9
4	1010	0279	36	1010	29C9
5	10F8	0560	37	1010	29C9
6	10F8	0560	38	10F8	2CB0
7	10A6	0731	39	10F8	2CB0
8	10C2	0867	40	1016	2E39
9	1030	096D	41	1030	30BD
10	1030	096D	42	1030	30BD
11	1030	096D	43	1010	339D
12	1010	0C4D	44	1010	389D
13	1010	0C4D	45	10F8	3684
14	1072	0C7C	46	10F8	3684
15	10F8	0F34	47	10A6	3855
16	1016	10BD	48	10C2	39DB
17	10C2	128B	49	1030	3A91
18	1030	1341	50	1030	3A91
19	1030	1341	51	1030	3A91
20	1010	1621	52	1010	3D71
21	1010	1621	53	1010	3D71
22	10F8	1908	54	1072	3DA0
23	10F8	1908	55	10F8	4058
24	10C2	1C5F	56	1016	41E1
25	1030	1D15	57	10C2	43AF
26	1030	1D15	58	1030	4465
27	1030	1D15	59	1030	4465
28	1010	1FF5	60	1030	4465
29	1010	1FF5	61	1010	4745
30	10F8	22DC	62	1010	4745
31	10A6	24AD	63	10F8	4A2C
32	10C2	2633	64	10F8	4A2C

Table 4. ADDGEN1 and ROUTINE3

	Address (hex)	Time		Address	Time
1	1030	0000	33	1030	26E9
2	1030	0000	34	1030	26E9
3	1010	0279	35	1010	29C9
4	1072	02A8	36	1072	29F8
5	10F8	0560	37	10F8	2CB0
6	1016	06E9	38	10A6	2E81
7	10A6	0731	39	10C2	3007
8	10C2	0867	40	1030	30BD
9	1030	096D	41	1030	30BD
10	1030	096D	42	1030	30BD
11	1060	0985	43	1010	339D
12	1010	0C4D	44	1010	339D
13	1010	0C4D	45	10F8	3684
14	10F8	0F34	46	10F8	3684
15	10F8	0F34	47	1016	380D
16	1016	10BD	48	1030	3A91
17	1030	1341	49	1030	3A91
18	1030	1341	50	1010	3D71
19	1010	1621	51	1072	3DA1
20	1072	1650	52	10F8	4058
21	10F8	1908	53	10A6	4229
22	1016	1A91	54	10C2	43AF
23	10A6	1AD9	55	1030	4465
24	10C2	1C5F	56	1030	4465
25	1030	1D15	57	1030	4465
26	1030	1D15	58	1010	4745
27	1060	1D2D	59	1010	4745
28	1010	1FF5	60	10F8	4A2C
29	1010	1FF5	61	10F8	4A2C
30	10F8	22DC	62	1016	4BB5
31	10F8	22DC	63	1030	4E39
22	1016	2465	64	1030	4E39

Table 5. ADDGEN2 and ROUTINE1

	Address (hex)	Time		Address (hex)	Time
1	1030	0000	33	1016	1C30
2	1030	0000	34	10C2	1E00
3	1030	0000	35	10C2	1E00
4	1030	0000	36	10C2	1E00
5	1072	0263	37	1030	2033
6	1072	0263	38	1030	2033
7	1072	0263	39	1030	2033
8	10F8	04EE	40	1072	234D
9	10F8	04EE	41	1072	234D
10	1016	0674	42	10F8	2588
11	10C2	0844	43	10F8	2588
12	10C2	0844	44	1016	270E
13	10C2	0844	45	10C2	28DE
14	1030	0A77	46	10C2	28DE
15	1030	0A77	47	10C2	28DE
16	1030	0A77	48	1030	2B11
17	1072	0D91	49	1030	2B11
18	1072	0D91	50	1030	2B11
19	1072	0D91	51	1030	2B11
20	10F8	0FCC	52	1072	2E2B
21	10F8	0FCC	53	1072	2E2B
22	1016	1152	54	10F8	3066
23	10C2	1322	55	10F8	3066
24	10C2	1322	56	10F8	3066
25	10C2	1322	57	1016	31EC
26	1030	1555	58	10C2	33BC
27	1030	1555	59	10C2	33BC
28	1030	1555	60	10C2	33BC
29	1072	186F	61	1030	35EF
30	1072	186F	62	1030	35EF
31	10F8	1AAA	63	1030	35EF
32	10F8	1AAA	64	1030	35EF

Table 6. ADDGEN2 and ROUTINE2

	Address (hex)	Time		Address	Time
1	1030	0000	33	10C2	1E00
2	1030	0000	34	10C2	1E00
3	1030	0000	35	1030	2033
4	1072	0263	36	1030	2033
5	1072	0263	37	1030	2033
6	10F8	04EE	38	1072	234D
7	10F8	04EE	39	1072	234D
8	10F8	04EE	40	10F8	2588
9	1016	0674	41	10F8	2588
10	10C2	0844	42	1016	270E
11	10C2	0844	43	10C2	28DE
12	10C2	0844	44	10C2	28DE
13	1030	0A77	45	10C2	28DE
14	1030	0A77	46	1030	2B11
15	1030	0A77	47	1030	2B11
16	1030	0A77	48	1030	2B11
17	1072	0D91	49	1072	2E2B
18	1072	0D91	50	1072	2E2B
19	10F8	0FCC	51	10F8	3066
20	10F8	0FCC	52	10F8	3066
21	1016	1152	53	10F8	3066
22	10C2	1322	54	1016	31EC
23	10C2	1322	55	10C2	33BC
24	10C2	1322	56	10C2	33BC
25	1030	1555	57	10C2	33BC
26	1030	1555	50	1030	35EF
27	1030	1555	59	1030	35EF
28	1072	186F	60	1030	35EF
29	1072	186F	61	1030	35EF
30	10F8	1AAA	62	1072	3909
31	10F8	1AAA	63	1072	3909
32	1016	1C30	64	10F8	3B44

Table 7. ADDGEN2 and ROUTINE3

	Address (hex)	Time		Address	Time
1	1030	0000	33	1030	2033
2	1030	0000	34	1072	234D
3	1030	0000	35	1072	234D
4	1072	0263	36	10F8	2588
5	1072	0263	37	10F8	2588
6	10F8	04EE	38	1016	270E
7	10F8	04EE	39	10C2	28DE
8	1016	0674	40	10C2	28DE
9	10C2	0844	41	1030	2B11
10	10C2	0844	42	1030	2B11
11	1030	0A77	43	1030	2B11
12	1030	0A77	44	1072	2E2B
13	1030	0A77	45	1072	2E2B
14	1072	0D91	46	10F8	3066
15	1072	0D91	47	10F8	3066
16	10F8	0FCC	48	1016	31EC
17	10F8	0FCC	49	10C2	33BC
18	1016	1152	50	10C2	33BC
19	10C2	1322	51	1030	35EF
20	10C2	1322	52	1030	35EF
21	1030	1555	53	1030	35EF
22	1030	1555	54	1072	3909
23	1030	1555	55	1072	3909
24	1072	186F	56	10F8	3B44
25	1072	186F	57	10F8	3B44
26	10F8	1AAA	58	1016	3CCA
27	10F8	1AAA	59	10C2	3E9A
28	1016	1C30	60	10C2	3E9A
29	10C2	1E00	61	1030	40CD
30	10C2	1E00	62	1030	40CD
31	1030	2033	63	1030	40CD
32	1030	2033	64	1072	43E7

Table 8. ADDGEN3 and ROUTINE1

	Address	Time		Address	Time
1	1030	0000	33	10F8	3B44
2	1030	0000	34	10C2	3E9A
3	1072	0263	35	10C2	3E9A
4	10F8	04EE	36	1030	40CD
5	10C2	0844	37	1030	40CD
6	10C2	0844	38	1072	43E7
7	1030	0A77	39	10F8	4622
8	1030	0A77	40	10C2	4976
9	1072	0D91	41	1030	4BAB
10	10F8	0FCC	42	1030	4BAB
11	10C2	1322	43	1072	4EC5
12	10C2	1322	44	10F8	5100
13	1030	1555	45	10C2	5456
14	1030	1555	46	1030	5689
15	1072	186F	47	1030	5689
16	10F8	1AAA	48	1072	59A3
17	10C2	1E00	49	10F8	5BDE
18	10C2	1E00	50	1016	5D64
19	1030	2033	51	10C2	5F34
20	1030	2033	52	1030	6167
21	1072	234D	53	1030	6167
22	10F8	2588	54	1072	6481
23	10C2	28DE	55	10F8	66BC
24	10C2	28DE	56	1016	6842
25	1030	2B11	57	10C2	6A12
26	1030	2B11	58	1030	6C45
27	1072	3E2B	59	1030	6C45
28	10F8	3066	60	1072	6F5F
29	10C2	33BC	61	10F8	719A
30	10C2	33BC	62	10C2	74F0
31	1030	35EF	63	10C2	74F0
32	1072	3909	64		

Table 9. ADDGEN3 and ROUTINE2

	Address (hex)	Time (hex)		Address (hex)	Time (hex)
1	1030	0000	33	10C2	3E9A
2	1030	0000	34	10C2	3E9A
3	1072	0263	35	1030	40CD
4	10F8	04EE	36	1030	40CD
5	10C2	0844	37	1072	43E7
6	10C2	0844	38	10F8	4622
7	1030	0A77	39	10C2	4978
8	1030	0A77	40	1030	4BAB
9	1072	0D91	41	1030	4BAB
10	10F8	0FCC	42	1072	4EC5
11	10C2	1322	43	10F8	5100
12	10C2	1322	44	10C2	5456
13	1030	1555	45	10C2	5456
14	1030	1555	46	1030	5689
15	1072	186F	47	1030	5689
16	10F8	1AAA	48	1072	59A3
17	10C2	1E00	49	10F8	5BDE
18	1030	2033	50	10C2	5F34
19	1030	2033	51	1030	6167
20	1072	234D	52	1030	6167
21	10F8	2588	53	1072	6481
22	10C2	28DE	54	10F8	66BC
23	10C2	28DE	55	10C2	6A12
24	1030	2B11	56	10C2	6A12
25	1030	2B11	57	1030	6C45
26	1072	2E2B	58	1030	6C45
27	10F8	3066	59	1072	6F5F
28	10C2	33BC	60	10F8	719A
29	1030	35EF	61	10C2	74F0
30	1030	35EF	62	1030	7723
31	1072	3909	63	1030	7723
32	10F8	3B44	64	1072	7A3D

Table 10. ADDGEN3 and ROUTINE3

	Address (hex)	Time (hex)		Address	Time
1	1030	0000	33	1030	40CD
2	1030	0000	34	1072	43E7
3	1072	0263	35	10F8	4622
4	10F8	04EE	36	10C2	4978
5	10C2	0844	37	10C2	4978
6	1030	0A77	38	1030	4BAB
7	1030	0A77	39	1072	4EC5
8	1072	0D91	40	10F8	5100
9	10F8	0FCC	41	10C2	5456
10	10C2	1322	42	10C2	5456
11	10C2	1322	43	1030	5689
12	1030	1555	44	1072	59A3
13	1030	1555	45	10F8	5BDE
14	1072	186F	46	10C2	5F34
15	10F8	1AAA	47	10C2	5F34
16	10C2	1E00	48	1030	6167
17	10C2	1E00	49	1072	6481
18	1030	2033	50	10F8	66BC
19	1072	234D	51	10C2	6A12
20	10F8	2588	52	10C2	6A12
21	10C2	28DE	53	1030	6C45
22	10C2	28DE	54	1072	6F5F
23	1030	2B11	55	10F8	719A
24	1072	2E2B	56	10C2	74F0
25	10F8	3066	57	10C2	74F0
26	10C2	33BC	58	1030	7723
27	10C2	33BC	59	1072	7A3D
28	1030	35EF	60	10F8	7C78
29	1072	3909	61	10C2	7FCE
30	10F8	3B44	62	10C2	7FCE
31	10C2	3E9A	63	1030	8201
32	10C2	3E9A	64	1072	851B

Table 11. ADDGEN4 and ROUTINE1

	Address (hex)	Time (hex)		Address	Time
1	1030	0000	33	1016	3902
2	1030	0000	34	1016	3902
3	1072	02B3	35	1030	3E03
4	1016	04FF	36	1030	3E03
5	1016	04FF	37	1016	4369
6	1016	04FF	38	1016	4369
7	1030	0A00	39	1016	4369
8	1030	0A00	40	1030	486A
9	1072	0D1A	41	1030	486A
10	1016	0F66	42	1072	4B84
11	1016	0F66	43	1016	4DD0
12	1016	0F66	44	1016	4DD0
13	1030	1467	45	1016	4DD0
14	1030	1467	46	1030	52D1
15	1016	19CD	47	1030	52D1
16	1016	19CD	48	1016	5837
17	1016	19CD	49	1016	5837
18	1030	1ECE	50	1016	5837
19	1030	1ECE	51	1030	5D38
20	1072	21E8	52	1030	5D38
21	1016	2434	53	1072	6052
21	1016	2434	53	1072	6052
22	1016	2434	54	1016	629E
23	1016	2434	55	1016	629E
24	1030	2935	56	1016	629E
25	1030	2935	57	1030	679F
26	1016	2E9B	58	1030	679F
27	1016	2E9B	59	1016	6D05
28	1016	2E9B	60	1016	6D05
29	1030	339C	61	1016	6D05
30	1030	339C	62	1030	7206
31	1072	36B6	63	1030	7206
32	1016	3902	64	1072	7520

Table 12. ADDGEN4 and ROUTINE2

	Address (hex)	Time (hex)		Address	Time
1	1030	0000	33	1016	3902
2	1030	0000	34	1016	3902
3	1016	04FF	35	1016	3902
4	1016	04FF	36	1030	3E03
5	1016	04FF	37	1030	3E03
6	1030	0A00	38	10F8	42DD
7	1030	0A00	39	1016	4369
8	10F8	0EDA	40	1016	4369
9	1016	0F66	41	1016	4369
10	1016	0F66	42	1030	486A
11	1016	0F66	43	1030	486A
12	1030	1467	44	10F8	4D44
13	1030	1467	45	1016	4DD0
14	10F8	1941	46	1016	4DD0
15	1016	19CD	47	1016	4DD0
16	1016	19CD	48	1030	52D1
17	1016	19CD	49	1030	52D1
18	1030	1ECE	50	10F8	57AB
19	1030	1ECE	51	1016	5837
20	10F8	23A8	52	1016	5837
21	1016	2434	53	1016	5837
22	1016	2434	54	1030	5D38
23	1016	2434	55	1030	5D38
24	1030	2935	56	10F8	6212
25	1030	2935	57	1016	629E
26	10F8	2E0F	58	1016	629E
27	1016	2E9B	59	1016	629E
28	1016	w#9B	60	1030	679F
29	1016	2E9B	61	1030	679F
30	1030	339C	62	10F8	6C79
31	1030	339C	63	1016	6D05
32	10F8	3876	64	1016	6D05

Table 13. ADDGEN4 and ROUTINE3

	Address	Time		Address	Time
1	1030	0000	33	1016	3902
2	1030	0000	34	1016	3902
3	10F8	0473	35	1016	3902
4	1016	04FF	36	1016	3902
5	1016	04FF	37	1030	3E03
6	1016	04FF	38	1030	3E03
7	1030	0A00	39	10F8	42dd
8	1072	0D1A	40	1016	4369
9	1016	0F66	41	1016	4369
10	1016	0F66	42	1016	4369
11	1016	0F66	43	1030	486A
12	1030	1467	44	1072	4B84
13	1030	1467	45	1016	4DD0
14	1072	1781	46	1016	4DD0
15	1016	19CD	47	1016	4DD0
16	1016	19CD	48	1030	52D1
17	1016	19CD	49	1030	52D1
18	1016	19CD	50	1072	55EB
19	1030	1ECE	51	1016	5837
20	1030	1ECE	52	1016	5837
21	10F8	23A8	53	1016	5837
22	1016	2434	54	1016	5837
23	1016	2434	55	1030	5D38
24	1016	2434	56	1030	5D38
25	1030	2935	57	10F8	6212
26	1072	3C4F	58	1016	629E
27	1016	2E96	59	1016	629E
28	1016	2E9B	60	1016	629E
29	1016	2E9B	61	1030	679F
30	1030	339C	62	1072	6AB9
31	1030	339C	63	1016	6D05
32	1072	36B6	64	1016	6D05

circuit. As discussed in Chapter 5, the LRU routines check the validity of a record before producing it as an output. This can be noticed from the different experiments, since in no case has a page address been produced as an output without being one of the skipped addresses in the generated address sequences.

Because the LRU submodule works asynchronously with the address generation module, it can be observed that the output record sequence is not exactly repetitive although the address sequence directed to the submodule is repetitive.

The time interval between two consecutive addresses has been calculated for the four address generation routines. With a 6MHZ clock, the time intervals are:

ADDGEN1	5.3 μs
ADDGEN2	3.66μs
ADDGEN3	2.83μs
ADDGEN4	2.83μs

These time intervals are calculated rather than measured and the interference with the LRU submodule is not taken into consideration. Interference has also not been considered in calculating the time periods during which some addresses are skipped as shown in Figures 22 through 25. Although the delay due to the interference with the LRU submodule has not been considered, the numbers provide good ground for comparison. Also, it is true that for any system to be supported by an LRU module, the figures describing the system speed and address stream are likely to assume conflict free operation.

The data obtained from the twelve experiments provide a basis to

compare the performance of different LRU routines. Although 64 records per experiment are not large enough to draw any statistical conclusions, it has been observed that there is no major difference in the data when a relatively large number of records is recorded. The reason is that the whole address stream generated by a certain routine is repetitive and the only factor that might alter the output data is the relative arrival times for RAM access requests from the address generation circuit and the LRU submodule.

In the following section, some statistical data that relate to the performance of the three LRU routines described earlier under different arrival rates are introduced. The major comparison figure will be the probability that an output is produced by the LRU submodule under different arrival rates and different time periods, during which a simulated page frame is not referenced. Table 14 summarizes the performance of ROUTINE1 which keeps only one LRU record internally under different arrival rates. It can be noticed that it performs better with slower address arrival rates. It can also be seen that any page frame not referenced for 2.17 ms or more is outputted with a probability of one as long as it is the oldest referenced page frame. It is worth pointing out that with ADDGEN4 address 1030 hex is skipped for about 3.49 ms and has always been produced as an output two consecutive times. This emphasizes that 2.17 ms is enough time period for the LRU submodule when running ROUTINE1 to produce a correct output. It is interesting to note that when two or more addresses are skipped in a certain segment of an address generation routine, the lowest of these

Table 14. Performance of ROUTINE1

Address generation routine (arrival rate)	Time period address is skipped	Probability of being outputted
ADDGEN1	1.4 ms	0.5
(5.34 μs)	2.05 ms	0.625
	2.73 ms or more	1.0
ADDGEN2		
(6.7 μs)	2.7 ms or more	1.0
ADDGEN3		
(3.5 μs)	1.728 ms	0.09
	2.17 ms or more	1.0
ADDGEN4	1.16 ms	0.0
(3.5 μs)	1.75 ms	0.566
	3.49 ms or more	1.0

addresses is the LRU address because all four address generation routines reference lower addresses before higher ones. In no case has an address other than the lowest skipped address been produced as the output of the LRU submodule when running ROUTINE1. This is because ROUTINE1 keeps only one record inside the microprocessor. This implies that after the search and the validity check, if the record is not valid, no output is produced and the routine starts a new search. In such a case, it is certain that by the end of the new search, none of the higher order addresses would still be valid since the search time is longer than the time needed to execute an address generator segment. Moreover, since an invalid record means that the address generation routine has moved to a new segment, it is certain that the output is the lowest skipped address or none at all in a single ROUTINE1 execution.

The same kind of analysis can be applied to the performance statistics of ROUTINE2 shown in Table 15. The statistics shown for the ADDGEN4 and ROUTINE2 combination may look strange. An address skipped for 1.16 ms has a probability of 0.909 of being outputted, whereas another address skipped for 1.75 ms has a probability of zero of being outputted. To explain, it must be said that in the performed experiments, a page frame with higher address has a better chance of being produced as the output of the LRU submodule than a lower address provided that it is the actual LRU frame. The reason is that the address generation routines scan the addresses from low to high. Thus, if the search ends with two LRU records, the probability that higher address would

Table 15. Performance of ROUTINE2

Address generation routine (address/ time)	Time period address is skipped	Proba- bility of being outputted
ADDGEN1		
(address/5.34 μs)	1.4 ms	0.866
	2.05 ms	0.866
	2.73 ms or more	1.0
ADDGEN2		
(address/6.7 μs)	2.7 ms or more	1.0
ADDGEN3	1.728 ms	0.0
(address/3.5 μs)	2.17 ms or more	1.0
ADDGEN4		
(address/3.5 μs)	1.16 ms	0.909
	1.75 ms	0.0
	3.49 ms or more	1.0

still be valid is higher than the probability that the lower address would still be valid. This is particularly true if the address generation segment which skips these addresses is executed only one time. This is the case with segment 3 of ADDGEN4 in which the hex addresses 1020, 1040, and 10F8 are skipped. Since the segment is executed only one time, the mentioned addresses are not referenced for 1.16 ms. Address 10F8, thus, has the highest probability of being outputted for the mentioned reason. Thus, there is a strong relationship between the probability of outputting a certain page frame address and its location in main memory only if the LRU page frame is not referenced for less than 2.17 ms. This does not imply that the LRU submodule is producing a wrong result since at the time an output is produced address 10F8, in the above case, is the actual LRU frame address. It is to be noted that in an actual LRU support module situation, the order in which page frames are referenced is rather more random than in ascending order as in the test experiments.

It is worth noticing that address 1030 hex which remains unreferenced for 3.49 ms is outputted two consecutive times in all recorded cases when ROUTINE2 and ADDGEN4 are used. However, as with ROUTINE1, a skipping period of 2.17 ms of some page address makes it certain that the frame address would be the LRU submodule's output provided that it is the oldest referenced frame.

The performance statistics of ROUTINE3 are shown in Table 16.

There is no great difference in the performance of the three LRU routines considered, and any one of them is capable of producing any

Table 16. ROUTINE3 performance statistics

Address generation routine (address/ µs)	Time period address is skipped	Probability of being outputted
ADDGEN1		
(address/5.34 µs)	1.4 ms	0.5
	2.05 ms	0.5-0.75 (depending on address)
	2.73 ms or more	1.0
ADDGEN2		
(address/6.7 µs)	2.7 ms or more	1.0
ADDGEN3	1.728 ms	0.0
(address/3.5 µs)	2.17 ms or more	1.0
ADDGEN4	1.16 ms	0.40
(address/3.5 µs)	1.75 ms	0.60
	3.49 ms or more	1.0

page frame number as an output with a probability of one if it remains unreferenced for 2.17 ms or more, provided that it is the oldest referenced frame. However, it seems that ROUTINE2 is superior to both ROUTINE1 and ROUTINE3 because it executes faster than ROUTINE3 and it will output some page addresses that might be missed by ROUTINE1. For instance, it takes ROUTINE3 about eleven repetitions of ADDGEN1 to produce 64 output records, whereas it takes ROUTINE2 about $8\frac{1}{2}$ repetitions of ADDGEN1 to produce 64 output records. In comparison with ROUTINE1, it is clear that whenever two addresses are skipped in one address generation routine segment, the probability that the higher address will be outputted by ROUTINE2 is certainly higher than that with ROUTINE1. For example, if we compare the data obtained in experiments 1 and 2, it can be seen that the probability that address 1072, which is skipped in the second segment of ADDGEN1 along with address 1010, has a higher chance of being outputted with ROUTINE2 than with ROUTINE1.

Remarks and Observations

It has been observed experimentally that the time between two consecutive LRU outputs ranged between 1.5 ms and 2.75 ms. If we assume that the average time interval needed to produce an output is around 2.25 ms, it is possible to use the 8 MHZ version of the MC68000 microprocessor and have an output every approximately 1.69 ms. This output rate is actually faster than one would really need. This is because a computer system would not transfer pages to main memory at this rate, especially since it might take a modern disk 10 ms or more

to transfer a page [23]. To make use of this fact, one may suggest increasing the number of page frames assigned to a submodule to 256 or even 512 instead of 128 without jeopardizing the performance. This in effect means cutting the number of submodules to one-half or one-fourth the number when 128 pages are assigned to a submodule. A better idea is to try to reduce the amount of loading the LRU module represents on the main system. This idea will now be discussed in detail.

Reducing LRU module loading on main system

The LRU module is supposed to support the main system by performing the function of finding the least recently used page frame in main memory. In an ideal situation, the LRU module should work in total parallelism with the main system without any kind of interference. In our design, we used straight arbitration to solve the submodule's RAM access conflicts as discussed earlier. In our design, the probability that the main system will be forced to wait until a microprocessor completes a read cycle is estimated to be less than 1%. This is based on the fact that the microprocessor accesses the RAM on the average less than 20% of the time, and based also on the reasonable assumption that on the average a certain page will be accessed 20 consecutive times before switching to another page. Hence, only 1/20 of the addresses will actually reach the LRU module because of the filtering circuit effect. However, it is possible to use the designed circuit to support a main system that is two times faster. This is because with

an 8 MHZ microprocessor, a RAM cycle would take about 500 ns, while the TIMER-RAM combination can work at a speed that would allow access every 250 ns. Thus, a 1% interference would actually become a 2% delay since the main system would have to wait for a memory cycle that is two times longer than its memory cycle.

One possible technique to reduce the interference by about 50% is to use the BUS ERROR ($\overline{\text{BERR}}$) processing exhibited by the MC68000 microprocessor [22]. The MC68000 will repeat the bus cycle if the $\overline{\text{BERR}}$ and the $\overline{\text{HALT}}$ signals are activated at least 50 ns before the $\overline{\text{DTACK}}$ signal is received. This feature can be utilized to give the main system higher access priority to a submodule's RAM than the microprocessor. A timing diagram of an MC68000 word read cycle with two wait states (as is the case in our circuit) is shown in Figure 29. The time period X is the period during which any attempt to force the microprocessor to repeat the bus cycle would be too late and the read cycle has to be completed. However, any request to copy the TIMER into the RAM arriving during the time period marked Y can be granted immediately by utilizing the bus error feature. In such a case, the hardware logic must be designed to activate the $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ signals and at the same time put all the submodule's 3-state buffers in the high impedance state. This would cause the MC68000 to repeat the bus cycle immediately after the current bus cycle is completed. There is no limitation on repeating the bus cycle as long as the mentioned timing requirements are met. This allows a certain bus cycle to be repeated several times if the $\overline{\text{BERR}}$ and $\overline{\text{HALT}}$ signals are activated properly every

time.

It is clear that the added logic must also cause the main system to wait if the RAM access request is received during the period X. To simplify the extra hardware and guarantee proper operation, the 50 ns period mentioned can be increased to one-half a clock period (62.5 ns at 8 MHZ). This is also shown in Figure 29 and indicates that one-half of potential main system delays have been eliminated by deciding conflicts in the period marked X in favor of the main system, and thus reducing the interference load on the main system by 50%.

Fortunately the RAM cycles that have to be aborted with such a technique are read cycles and not write cycles. Thus, the technique is feasible and the extra hardware needed is expected to be simple.

It would be nice if a microprocessor were developed that had a bus error feature as the MC68000 but without timing constraints. In other words, if a bus error signal arrived anywhere during the bus cycle, it would still cause the processor to repeat the bus cycle. It might seem impractical to ask for such a microprocessor, but it is actually not. Such a microprocessor would open a new era in multi-microprocessor systems in general. It would simplify to a great extent the controlling of access to shared resources by simply granting the access to the first requester and causing all subsequent requesters during the cycle to try again. It would also simplify dynamic priority scheduling by allowing the highest priority processor free access to a resource, while forcing the others to repeat their bus cycles should a conflict arise. The priority can be changed dynamically by

rearranging the access rights. This should result in a very efficient utilization of a common resource since no arbitration time is needed every cycle as is the case in conventional systems.

A zero load LRU module

Thinking about a microprocessor with a bus error feature as described above leads to the idea of implementing the bus error feature external to the microprocessor with some software help. The idea is to allow the time recording process to start at any time without any constraints, and at the same time let the microprocessor check the correctness of its read cycle after its completion. Some hardware has to detect the arrival of a time recording signal and tri-state all RAM buffers on the microprocessor side. On the other hand, a special signal that starts with the start of microprocessor RAM cycle and ends $\frac{1}{2}$ clock period after the end of the cycle has to be generated and used to set a special flip-flop whenever a time write signal arrives while the mentioned signal is active. If the flip-flop is set, the microprocessor repeats the same cycle. If the flip-flop is found clear, the microprocessor proceeds without need to repeat the read cycle. It is possible that a certain cycle can be repeated several times before the flip-flop is found clear. Since RAM long word instructions require two consecutive word read cycles to the RAM, it is only possible to repeat booth read cycles. In such a case, the flip-flop would be set should a conflict occur during any of the two cycles.

Another alternative is to queue the filtered addresses at the sub-module and pass addresses in a way similar to Direct Memory Access

(DMA) operation.

An even simpler approach is to let the microprocessor perform each RAM read instruction two consecutive times and then compare the values read. If a match is found, it proceeds to the next instruction. If no match is found, the process is repeated until a match is found. One possible source of error in such a technique is that it is likely that the microprocessor would read a tri-stated buffer as an all-ones word. Thus, it is necessary to make sure that the matched words are not all-ones words. Of course, other sources of error such as critical timing almost always exist and must be taken into consideration.

It might still be acceptable to have LRU submodules that produce output at a four times slower rate than the one we built. In such a case, the last approach, although slow, may be acceptable.

CHAPTER VII. CONCLUSION

Since the operating system is a very costly part of a computer system in terms of both the initial cost and the operational cost, it has been found that an approach to support the operating system using modern microprocessors is worth studying.

An approach to support and parallel process some operating system functions has been introduced. The technique utilizes existing, inexpensive, and powerful microprocessors to support operating system functions that lend themselves to parallel processing. The support system consists of several modules, each of which performs some operating system function and communicates with the supported system through a small, dedicated main memory area called a communication area. The proposed technique is general and can be used to support systems under design as well as systems in operation. It also can be utilized in single processor, as well as multi-processor systems with shared memory. The cost, reliability, and other aspects have been studied. Some possible advantages of applying the proposed technique can be summarized as follows:

(1) It is possible to reduce the operating system CPU time requirements by parallel processing many of its functions. This is particularly attractive in the case of modularly structured operating systems as most current systems are. Reducing operating system CPU time requirements would mean that more CPU time is available for productive work.

(2) It is possible to reduce operating system complexity by adopting simpler functional algorithms. This is because in many cases the designer is forced to design more complicated algorithms merely to reduce the execution overhead. Since the support approach has as a major concern the reduction of the overhead, it is possible to go for simpler algorithms which might perform better with support than more complicated ones without support. This would reduce to some extent the overall software complexity, and hence the overall system initial cost.

(3) It is possible to perform some tasks that are currently considered impractical because their overhead is unacceptable. For instance, the exact implementation of the Least Recently Used (LRU) replacement policy in demand paging memory management systems is believed to be "not feasible" because of its overhead.

(4) Some support modules may be assigned monitoring and performance measurement functions. These modules may then submit reports to the main system which uses the reports to adjust dynamically or "fine tune" some operating system parameters. Some of these parameters might be:

(a) Working set size,

(b) Page/sector size,

(c) Bus allocation scheme.

This would enhance system performance since the parameters

are fine tuned to optimize the performance according to actual working conditions on line rather than being fixed at certain value during the design phase.

To prove, at least, some of the above-mentioned points, two specific applications have been invented as examples. The first is a support module for a deadlock avoidance scheme. In this application, theoretical study as well as a possible design of the module have been given. The second application is the exact implementation of the Least Recently Used replacement policy in a demand paging memory management system. In this case, a submodule has been designed, built, and tested.

Since deadlock avoidance schemes incur high overhead, it has been predicted that in the near future the deadlock problem will acquire greater attention. This is especially true in systems sharing an increasing number of individual users, and in systems which provide a large set of files or data bases for many users with different access rights. This is the motive behind considering deadlock avoidance schemes as an application example of the proposed approach. Many deadlock avoidance algorithms differing significantly in the degree of complexity and in the amount of overhead incurred are already available. However, in all cases the overhead gets unacceptable as the number of active processes in the system, say (m), gets larger than some value. Habermann's model is considered an extreme model because the amount of advance information about process resource requirements is very small compared to other algorithms [14, 17]. The algorithm is relatively simple but its execution time is $O(m^2)$, where m is the number of

processes. As m gets larger than five, the overhead becomes unacceptable. The algorithm, however, is much simpler than many other algorithms that trade simplicity for some overhead savings. Habermann's algorithm was selected for modular microprocessor-based support just to prove that simple algorithms can be supported to give better performance than many other more complicated algorithms without the support.

Design outlines for three support modules have been given. The first module contains m+1 microprocessors in which m microprocessors serve the m processes and one microprocessor serves as a supervisor. The module executes Habermann's algorithm with execution time O(m) instead of $O(m^2)$ without the support module. With support, the number of processes can be increased to 25 and the overhead may still be acceptable. Moreover, the algorithm is executed almost totally on the support module alleviating almost completely the whole overhead problem from the main CPU(s). If the amount of hardware in the support module is considered unacceptably high, a process microprocessor could be assigned say k processes instead of only one reducing the number of required microprocessors to $\frac{m}{k}+1$. However, the execution time in such a case is O(km), where k is an integer greater than one. This corresponds to the second support module presented. The third support module is applicable only in systems with small m and has an execution time O(km), where k is a positive fraction. The module uses only one microprocessor.

Thus, the first application example proves the practicality and

feasibility of the support approach even when simple algorithms are adopted. It also proves points 1 and 2 mentioned earlier in this chapter when the possible advantages of the approach were discussed.

The second application example deals with the exact implementation of the Least Recently Used (LRU) replacement policy. Theoretical and simulation studies demonstrated the superiority of the LRU replacement policy over all other practical replacement policies. However, it was believed that the exact implementation of the LRU was "not feasible" because of its tremendous overhead. Therefore, many systems tried to approximate the LRU. The Least Frequently Used (LFU), and the second chance or MULTICS, are two examples of such approximations. The LFU incurrs high overhead because the whole page table has to be searched every time a page fault occurs. The MULTICS exhibits less overhead than the LFU; however, it is still an approximation.

The author has described in detail the design of a microprocessor-based module for the exact implementation of the LRU replacement policy in a demand paging system. The idea is to divide main memory into n equal areas and assign each area to a submodule that runs an exact LRU routine to find out the LRU page frame address in its assigned area. One TIMER for the whole module is utilized in recording reference times to different page frames. Each submodule contains an MC68000 microprocessor, ROM, RAM, buffering chips, and some control logic. The microprocessor reads the routine from the ROM while the RAM stores the reference time records of pages assigned to the submodule. Each submodule outputs the area LRU frame address along with its time record

into some output latches. One supervisor submodule searches the outputs produced by other submodules and finds out the overall LRU frame address. Moreover, the supervisor has to communicate with the main system via a communication area. Thus, the LRU module is composed of n+1 submodules with one microprocessor in each submodule.

An LRU submodule has been designed, built, and tested. The submodule uses the MC68000 microprocessor to run the exact LRU routine. To test the submodule, it was essential to design an address stream generation module. Another MC68000 microprocessor has been utilized in building the address generation module. Three LRU routines and four address generation routines have been designed to allow extensive testing of the LRU submodule performance.

The LRU demonstrated good performance and produced correct outputs less than 3 ms apart. The submodule was assigned 128 page frames and ran at a frequency of 6 MHZ. It turned out that if the new 10 MHZ version of the MC68000 were used, the outputs would be about 1.5 ms apart. This would be a faster rate than most systems would need, and increasing the number of page frames to 512 would cause the submodule output rate to be about four times slower than the 128 page frames case. This would still be acceptable in most systems because a modern disk may take up to 10 ms to transfer a page to main memory [23].

The cost of a whole LRU module supporting a 1024 page frame memory would be less than $3,000, which is almost negligible compared to the cost of a multiprogramming computer system. The size of the LRU module should be small enough to fit nicely inside a modern disk drive, making

it a "smart disk."

The loading effect of the LRU module has also been discussed (main system's delay because of the LRU module). It has been found that any of several techniques could be used to reduce the loading effect to zero.

BIBLIOGRAPHY

1. Collins, S. D. In Infotch State of the Art Report on Operating Systems, No. 14. Maidenhead, England: Infotch International, England, 1972.

2. Baer, J. L. "Multiprocessing Systems." IEEE Trans. on Comp. C-25, No. 12 (December 1976):1271-1276.

3. Murakami, S., S. Nikishawa, and M. Sato. "Polyprocessor system, analysis and design." SIGARCH Newsletter, 5, No. 1 (March 1977): 44-56.

4. Chow, Y. C., and W. H. Kohler. "Performance of several queuing models for multiprocessor multiprogramming systems." Digest of Papers COMPCON, Washington D.C. 13 (Fall 1976):66-71.

5. Gonzalez, M. J., and C. V. Ramamoothy. "Parallel task execution in a decentralized system." IEEE Trans. on Comp. C-21, No. 12 (December 1972):1310-1322.

6. Hailstone, J. E. "Experience of operating system performance measurement." In Infotch State of the Art Report on Operating Systems, No. 14. Maidenhead, England: Infotch International, England, 1972.

7. Srodawa, R. J. "Positive experiences with a multiprocessing system." Computing Surveys, 10, No. 1 (March 1978):73-82.

8. Noguchi, K., I. Ohinishi, and H. Morita. "Design considerations for a heterogeneous tightly-coupled multiprocessor system." National Computer Conference, AFIPS Conf. Proc. 44 (1975):561-565.

9. Nishikawa, S., M. Sato, and E. Murakami. "Interconnection unit for poly-processor system: Analysis and design." SIGARCH Newsletter, 6, No. 7 (April 1978):216-222.

10. Anderson, G. L., and K. Bartlett. "Hardware allocation of data system resources." Computer Design 13 (July 1974):89-97.

11. Holt, R. C. "Some deadlock properties of computer systems." Computing Surveys, 4, No. 3 (September 1972):179-196.

12. Shoshami, A., and E. G. Coffman. "Prevention, detection, and recovery from system deadlocks." Proc. 4th Annual Princeton Conf. on Information Sciences and Systems, March 1970.

13. Hays, J. P. Computer Architecture and Organization. New York: McGraw Hill Book Co., 1978.

14. Habermann, A. N. "Prevention of system deadlocks." COMM. ACM 12, No. 7 (July 1969):373-377.

15. Russel, R. D. "A model for deadlock-free allocation primary version." Memo CGTM, No. 93. Dept. of Computer Science, Stanford Univ., June 1970.

16. Havender, J. W. "Avoiding deadlocks in multi-tasking systems." IBM System J. 7, No. 2 (1968):74-84.

17. Habermann, A. N. Introduction to Operating System Design. Chicago: SRA Inc., 1978.

18. Coffman, E. G. "Deadlocks in computer systems." Infotch State of Art Report on Operating Systems, No. 14. Maidenhead, England: Infotch International, 1972.

19. Coffman, E. G., and L. C. Varian. "Further experimental data on the behavior of programs in a paging environment." COMM. ACM 11, No. 7 (July 1968):471-474.

20. Belady, A. "A study of replacement algorithms for virtual storage computers." IBM Sys. J. 5, No. 2 (1966).

21. Bensoussan, A. "The MULTICS virtual memory: Concepts and design." COMM. ACM 15, No. 5 (May 1972):308-318.

22. MC68000 User's Manual. Austin, Texas: Motorola Semiconductor Products Inc., 1980.

23. Baer, J. L. Computer System Architecture. Potomac, Maryland: Computer Science Press, Inc., 1980.

ACKNOWLEDGMENTS

I would like to deeply thank my major professor, Dr. Arthur V. Pohm. His constant encouragement and personal kindness are deeply appreciated and acknowledged.

I am very thankful to Professor Terry A. Smay for his invaluable suggestions and discussions.

I also would like to thank Professors R. G. Brown, R. Lambert, and D. Grosvenor for agreeing to serve on my committee.

I wish to express my deep appreciation to Professor Julius O. Kopplin, Chairman of the Electrical and Computer Engineering Department, for providing the financial support during my study.

Finally, I would like to thank my mother and my wife for their support.

APPENDIX A. ROUTINE1

```
"68000"
;       EXACT LRU PROGRAM "ROUTINE1"
        ORG     0000
        HEX     000,2300,000,400  ;ALL ADDRESSES ARE IN HEX
        HEX     000,0D00,000,0E00
        ORG     400H
INIT    MOVE.L  #700H,SR          ;INITIALIZE STATUS REGISTER
        MOVE.L  #0,A0
        MOVE.L  #22000H,A6        ;INITIALIZE A5 & A6 FOR
        MOVE.L  #22200H,A5        ;CLEARING ALL RECORDS.
        CLR.L   D7
L1      MOVE.L  D7,-[A5]          ;CLEAR ALL 128 TIME RECORDS
        CMPA.L  A6,A5             ;IN THIS LOOP.
        BNE     L1
        MOVE.L  #2200H,A5         ;INITIALIZE AS TO HIGHEST ADD+4
        MOVE.L  #2000H,A6         ;AND A6 TO THE LOWEST ADDRESS.
SEARCH  MOVE.L  -[A5],D0          ;INITIALIZE THE SEARCH BY KEEPING
        MOVE.W  A5,A0             ;THE FIRST RECORD INTERNALLY.
NSRCH   MOVE.L  -[A5],D7          ;NORMAL SEARCH.
        CMP.L   D0,D7             ;IF NEW RECORD>OLD ONE: IGNORE IT
        BHI     TSTEND            ;BY BRANCHING TO TESTEND.
        MOVE.L  D7,D0             ;KEEP NEW RECORD AND ITS ADDRESS.
        MOVE.W  A5,A0
TSTEND  COMPA.L A5,A6             ;ALL 128 RECORDS SEARCHED ?
        BNE     NSRCH             ;IF NOT : GO BACK TO NORMAL SEARCH
OUTPUT  CMP.L   [A0],D0           ;CHECK VALIDITY OF LRU RECORD.
        BNE     ENDOUT            ;IF NOT VALID : DO NOT OUTPUT.
        MOVE.L  A0,D4
        LSR.L   #1,D4
        MOVE.W  D4,4000H          ;OUTPUT LRU FRAME ADDRESS.
        MOVE.L  D0,4010H          ;OUTPUT ITS TIME RECORD.
ENDOUT  MOVE.W  #2200H,A5         ;REINITIALIZE A5
        BRA     SEARCH            ;ALWAYS GO BACK TO THE NORMAL
                                  ;SEARCH.

        ORG     C000H             ;BUS ERROR HANDLER.
BUSERR  MOVE.L  #0FFFFFFFFH,4010H
        MOVE.L  #400H,0CH[A7]
        RTE

        ORG     0E00H             ;ADDRESS ERROR HANDLER.
ADDERR  MOVE.L  #0FFFFFFFFH,4010H
        MOVE.L  #400H,0CH[A7]
        RTE
```

APPENDIX B. ROUTINE2

```
"68000"
;      EXACT LRU PROGRAM  "ROUTINE2"
         ORG      000H
         HEX      000,2300,000,400
         HEX      000,0D)),000,0E00
         ORG      400H
INIT     MOVE.W   #700H,SR          ;INITIALIZE STATUS REGISTER.
         MOVE.L   #0,A0             ;CLEAR A0 THRU A2.
         MOVE.L   #0,A1
         MOVE.L   #0,A2
         MOVE.L   #22200H,A5        ;LOAD A5 WITH HIGHEST ADDR+4.
         MOVE.L   #22000H,A6        ;LOAD A6 WITH LOWEST ADDRESS.
         CLR.L    D7                ;NOTE THAT A17 IS ACTIVE DURING
L1       MOVE.L   D7,-[A5]          ;RECORD INITIALIZATION.
         CMPA.L   A6,A5             ;INITIALIZE ALL RECORDS TO ALL
         BHI      L1                ;ZEROS.
         MOVE.L   #2200H,A5         ;INITIALIZE A5 &A6 FOR READ.
         MOVE.L   #2000H,A6
SEARCH   MOVE.L   -[A5],D0          ;READ FIRST TWO RECORDS AND
         MOVE.W   A5,A0             ;ORDER THEM SUCH THAT D0 HOLDS
         MOVE.L   -[A5],D7          ;THE OLDER TIME RECORD WITH A0
         CMP.L    D0,D7             ;HOLDING ITS ADDRESS.
         BHI      L2                ;D1&A1 SHOULD HOLD THE OTHER
         MOVE.L   D0,D1             ;RECORD AND ITS ADDRESS.
         MOVE.W   A0,A1
         MOVE.L   D7,D0
         MOVE.W   A5,A0
         BRA      NSRCH
L2       MOVE.L   D7,D1
         MOVE.W   A5,A1
NSRCH    MOVE.L   -[A5],D7          ;"NORMAL SEARCH"
         CMP.L    D1,D7             ;READ RECORD INTO D7 AND COMPARE
         BHI      TSTEND            ;IT WITH D1 IF IT IS HIGHER;
         CMP.L    D0,D7             ;DISCARD IT.  IF NOT HIGHER :
         BHI      L3                ;DISCARD D1 &A1 AND REORDER THE
         MOVE.L   D0,D1             ;LIST IN THE SAME WAY AS BEFORE.
         MOVE.W   A0,A1
         MOVE.L   D7,D0
         MOVE.W   A5,A0
         BRA      TSTEND
L3       MOVE.L   D7,D1
         MOVE.W   A5,A1
TSTEND   CMPA.L   A5,A6             ;ALL 128 RECORDS SEARCHED?
         BNE      NSRCH             ;IF NOT GO BACK TO NSRCH.
```

```
OUTPUT  CMP.L   [A0],D0          ;CHECK THE VALIDITY OF THE
        BNE     L4               ;LRU RECORD. IF NOT VALID GO TO
        MOVE.L  A0,D4            ;L4 TO CHECK THE OTHER RECORD.
        LSR.L   #1,D4            ;ONE BIT SHIFT RIGHT MAPS ADDR
        MOVE.W  D4,4000H         ;TO INPUT STREAM AREA. OUTPUT
        MOVE.L  D0,4010H         ;LRU ADDRESS AND TIME RECORD.
        BRA     ENDOUT           ;GO TO ENDOUT.
L4      CMP.L   [A1],D1          ;CHECK THE VALIDITY OF THE 2nd
        BNE     ENDOUT           ;LRU RECORD :OUTPUT IF VALID.
        MOVE.L  A1,D4
        LSR.L   #1,D4
        MOVE.W  D4,4000H
        MOVE.L  D1,4010H
ENDOUT  MOVE.W  #2200H,A5        ;REINITIALIZE A5 AND GO BACK TO
        BRA     SEARCH           ;START A NEW SEARCH

        ORG     0D00H
BUSERR  MOVE.L  #0FFFFFFFFH,4010H ;BUS ERROR HANDLER.
        MOVE.L  #0400H,0CH[A7]
        RTE

        ORG     0E00H            ;ADDRESS ERROR HANDLER.
ADERR   MOVE.L  #0FFFFFFFFH,4010H
        MOVE.L  #0400H,0CH[A7]
        RTE
```

APPENDIX C. ADDGEN1

```
"6800"
;        ADDRESS GENERATION ROUTINE "ADDGEN1"
         ORG     000H
         HEX     0000,2300
         HEX     0000,0400          ;NOTE:ALL ADDRESSES ARE
         ORG     400H               ;IN HEXADECIMAL.
         MOVE    #0700H,SR          ;INITIALIZE STATUS REGISTER.
         MOVE.L  #1100H,A2          ;INITIALIZE A2 TO HIGHEST
INIT     MOVE.L  #1000H,A0          ;ADDRESS IN THE LIST+2,
         MOVE.W  #5,D2              ;AND A0 TO LOWEST ADDRESS.
         MOVE.W  #4,D3              ;D2 THROUGH D6 HOLD THE
         MOVE.W  #3,D4              ;No OF TIMES DIFFERENT
         MOVE.W  #2,D5              ;SEGMENTS ARE REPEATED.
         MOVE.W  #1,D6
SEGM1    MOVE.L  #1030H,A1          ;   "SEGMENT1"
         MOVE.L  #1060H,A3          ;ADDRESSES 1030,1060,AND
         MOVE.L  #1090H,A4          ;1090 WILL BE SKIPPED.
LOOP1    MOVE.W  [A0]+,D0
         NOP
         CMPA.W  A0,A1
         BNE     LOOP1
         ADD     #02H,A0            ;SKIP 1030.
LOOP2    MOVE.W  [A0]+,D0
         NOP
         CMPA.W  A0,A3
         BNE     LOOP2
         ADD     #02H,A0            ;SKIP 1060.
LOOP3    MOVE.W  [A0]+,D0
         NOP
         CMPA.W  A0,A4
         BNE     LOOP3
         ADD     #02H,A0            ;SKIP 1090.
LOOP4    MOVE.W  [A0]+,D0
         NOP
         CMPA.W  A0,A2
         BNE     LOOP4
         MOVE.L  #1000H,A0
         DBNE    D2,LOOP1           ;EXECUTED 5 TIMES ?
SEGM2    MOVE.L  #1072H,A3          ;   "SEGMENT2"
         MOVE.L  #1010H,A1          ;ADDRESS 1010, AND 1072
LOOP5    MOVE.W  [A0]+,D0           ;WILL BE SKIPPED IN THIS
         NOP                        ;SEGMENT.
         CMPA.W  A0,A1
         BNE     LOOP5
         ADD     #02H,A0            ;SKIP 1010.
```

```
LOOP6   MOVE.W  [A0]+,D0
        NOP
        CMPA.W  A0,A3
        BNE     LOOP6
        ADD     #02H,A0          ;SKIP 1072.
LOOP7   MOVE.W  [A0]+,D0
        NOP
        CMPA.W  A0,A2
        BNE     LOOP7
        MOVE.L  #1000H,A0
        DBNE    D3,LOOP5         ;EXECUTED 4 TIMES ?
SEGM3   MOVE.L  #10F8H,A1        ;   "SEGMENT 3"
LOOP8   MOVE.W  [A0]+,D0         ;IN THIS SEGMENT 10F8 AND 10FA
        NOP                      ;WILL BE SKIPPED.
        CMPA.W  A0,A1
        BNE     LOOP8
        add     #04H,A0          ;SKIP 10F8 & 10FA.
LOOP9   MOVE.W  [A0]+,D0
        NOP
        CMPA.W  A0,A2
        BNE     LOOP9
        MOVE.L  #1000H,A0
        DBNE    D4,LOOP8         ;EXECUTED 3 TIMES ?
SEGM4   MOVE.L  #1016H,A1        ;SEGMENT4.
        MOVE.L  #10A6H,A3        ;HERE 1016 & 10A6 WILL BE
LOOP10  MOVE.W  [A0]+,D0         ;SKIPPED.
        NOP
        CMPA.W  A0,A1
        BNE     LOOP10
        ADD     #2H,A0           ;SKIP 1016.
LOOP11  MOVE.W  [A0]+,D0
        NOP
        CMPA.W  A0,A3
        BNE     LOOP11
        ADD     #02H,A0          ;SKIP 10A6.
LOOP12  MOVE.W  [A0]+,D0
        NOP
        CMPA.W  A0,A2
        BNE     LOOP12
        MOVE.L  #1000H,A0
        DBNE    D5,LOOP10        ;EXECUTED 2 TIMES?
SEGM5   MOVE.L  #10c2H,A1        ;SEGMENTS.
LOOP13  MOVE.W  [A0]+,D0         ;HERE 10C2 WILL BE SKIPPED.
        NOP
        CMPA.W  A0,A1
        BNE     LOOP13
        ADD     #2H,A0           ;SKIP 10C2
```

```
LOOP14  MOVE.W  [A0]+,D0
        NOP
        CMPA.W  A0,A2
        BNE     LOOP14
        MOVE.L  #1000H,A0
        DBNE    D6,LOOP13       ;DO NOT REPEAT SEGMENT5.
        BRA     INIT            ;GO TO INIT TO REPEAT ALL OVER.
```

APPENDIX D. ADDGEN2

```
"6800"
;        ADDRESS GENERATION ROUTINE  "ADDGEN2" .
         ORG      000H
         HEX      0000,2300          ;ALL ADDRESSES ARE IN HEX.
         HEX      0000,0400
         ORG      400H
         MOVE     #0700H,SR          ;INITIALIZE STATUS REGISTER.
         MOVE.L   #1100H,A2
INIT     MOVE.L   #1000H,A0
         MOVE.W   #5,D2              ;D2 THRU D6 CONTROL THE No. OF
         MOVE.W   #3,D3              ;TIMES SEGMENTS 1 THRU 5 ARE
         MOVE.W   #3,D4              ;EXECUTED.
         MOVE.W   #2,D5
         MOVE.W   #4,D6
SEGM1    MOVE.L   #1030H,A0          ;   "SEGMENT1".
LOOP1    MOVE.W   [A0]+,D0           ;IN THIS SEGMENT ADDRESS 1030
         NOP                         ;WILL BE SKIPPED.
         NOP                         ;NOTE THAT 3 NOP INSTRUCTIONS
         NOP                         ;ARE USED TO SIMULATE SLOWER
         CMPA.W   A0,A1              ;ADDRESS ARRIVAL RATE AT THE
         BNE      LOOP1              ;LRU SUBMODULE.
         ADD      #02H,A0            ;SKIP 1030.
LOOP2    MOVE.W   [A0]+,D0
         NOP
         NOP
         NOP
         CMPA.W   A0,A2
         BNE      LOOP2
         MOVE.L   #1000H,A0
         DBNE     D2,LOOP1           ;EXECUTED 5 TIMES?
SEGM2    MOVE.L   #1072H,A1          ;   "SEGMENT2".
LOOP3    MOVE.W   [A0]+,D0           ;HERE 1072 &1074 WILL BE
         NOP                         ;SKIPPED.
         NOP
         NOP
         CMPA.W   A0,A1
         BNE      LOOP3
         ADD      #04H,A0            ;SKIP 1072 &1074.
LOOP4    MOVE.W   [A0]+,D0
         NOP
         NOP
         NOP
         CMPA.W   A0,A2
         BNE      LOOP4
         MOVE.L   #1000H,A0
         DBNE     D3,LOOP3           ;EXECUTED 3 TIMES ?
SEGM3    MOVE.L   #10F8H,A1          ;   "SEGMENT3".
```

```
LOOP5   MOVE.W  [A0]+,D0        ;HERE 10F8,10FA, AND 10FC WILL
        NOP                     ;BE SKIPPED. THE SEGMENT WILL
        NOP                     ;BE REPEATED THREE TIMES.
        NOP
        CMPA.W  A0,A1
        BNE     LOOP5
        ADD     #06H,A0         ;SKIP 10F8,10FA&10FC.
LOOP6   MOVE.W  [A0]+,D0
        NOP
        NOP
        NOP
        CMPA.W  A0,A2
        BNE     LOOP6
        MOVE.L  #1000H,A0
        DBNE    D4,LOOP5        ;EXECUTED 3 TIMES ?
SEGM4   MOVE.L  #1016H,A1       ;   "SEGMENT 4".
LOOP7   MOVE.W  [A0]+,D0        ;HERE ONLY 1016 WILL BE SKIPPED
        NOP                     ;AND THE SEGMENT WILL BE REPEATED
        NOP                     ;TWO TIMES.
        NOP
        CMPA.W  A0,A1
        BNE     LOOP7
        ADD     #2H,A0
LOOP8   MOVE.W  [A0]+,D0
        NOP
        NOP
        NOP
        CMPA.W  A0,A2
        BNE     Loop8
        MOVE.L  #1000H,A0
        DBNE    D5,LOOP7        ;EXECUTED 2 TIMES?
SEGM5   MOVE.L  #10C2H,A1       ;   "SEGMENT5".
LOOP9   MOVE.W  [A0]+,D0        ;ONLY 10C2 WILL BE SKIPPED,
        NOP                     ;AND THE SEGMENT WILL BE
        NOP                     ;REPEATED 4 TIMES.
        NOP
        CMPA.W  A0,A1
        BNE     LOOP9
        ADD     #2H,A0
LOOP10  MOVE.W  [A0]+,D0
        NOP
        NOP
        NOP
        CMPA.W  A0,A2
        BNE     LOOP10
        MOVE.L  #1000H,A0
        DBNE    D6,LOOP9        ;EXECUTED 4 TIMES?
        BRA     INIT            ;REPEAT ALL 5 SEGMENTS.
```

APPENDIX E. ADDGEN3

```
"68000"
;          ADDRESS GENERATION ROUTINE "ADDGEN3".
          ORG     000H
          HEX     0000,2300     ;ALL ADDRESSES ARE IN HEX.
          HEX     0000,0400
          ORG     400H
          MOVE    #0700H,SR     ;INITIALIZE STATUS REGISTER,
          MOVE.L  #1100H,A2     ;SET A2 TO HIGHEST ADDRESS+2.
INIT      MOVE.L  #1000H,A0     ;SET A0 TO LOWEST ADDRESS.
          MOVE.W  #5,D2         ;D2 THRU D5 WILL CONTROL THE
          MOVE.W  #3,D3         ;No OF EXECUTIONS OF SEGMENT1
          MOVE.W  #3,D4         ;THRU SEGMENTS RESPECTIVELY.
          MOVE.W  #2,D5
          MOVE.W  #4,D6
SEGM1     MOVE.L  #1030H,A1     ;   "SEGMENT1".
LOOP1     MOVE.W  [A0]+,D0      ;IN THIS SEGMENT ADDRESS 1030
          CMPA.W  A0,A1         ;ONLY WILL BE SKIPPED.
          BNE     LOOP1
          ADD     #02H,A0
LOOP2     MOVE.W  [A0]+,D0
          CMPA.W  A0,A2
          BNE     LOOP2
          MOVE.L  #1000H,A0
          DBNE    D2,LOOP1      ;EXECUTED 5 TIMES ?
SEGM2     MOVE.L  #1072H,A1     ;   "SEGMENT2".
LOOP3     MOVE.W  [A0]+,D0      ;HERE 1072 AND 1074 WILL BE
          CMPA.W  A0,A1         ;SKIPPED.
          BNE     LOOP3
          ADD     #04H,A0       ;SKIP 1072&1074.
LOOP4     MOVE.W  [A0]+,D0
          CMPA.W  A0,A2
          BNE     LOOP4
          MOVE.L  #1000H,A0
          DBNE    D3,LOOP3      ;EXECUTED 3 TIMES ?
SEGM3     MOVE.L  #10F8H,A1     ;   "SEGMENT3"
LOOP5     MOVE.W  [A0]+,D0
          CMPA.W  A0,A1
          BNE     LOOP5
          ADD     #06H,A0       ;SKIP 10F8,10FA,10FC.
LOOP6     MOVE.W  [A0]+,D0
          CMPA.W  A0,A2
          BNE     LOOP6
          MOVE.L  #1000H,A0
          DBNE    D4,LOOP5      ;EXECUTED 3 TIMES?
SEGM4     MOVE.L  #1016H,A1     ;   "SEGMENT4".
```

```
LOOP7   MOVE.W  [A0]+,D0
        CMPA.W  A0,A1
        BNE     LOOP7
        ADD     #2H,A0          ;SKIP 1016.
LOOP8   MOVE.W  [A0]+,D0
        CMPA.W  A0,A2
        BNE     LOOP8
        MOVE.L  #1000H,A0
        DBNE    D5,LOOP7        ;EXECUTED 2 TIMES ?
SEGM5   MOVE.L  #10C2H,A1       ;   "SEGMENT5"
LOOP9   MOVE.W  [A0]+,D0
        CMPA.W  A0,A1
        BNE     LOOP9
        ADD     #2H,A0          ;SKIP 10C2.
LOOP10  MOVE.W  [A0]+,D0
        CMPA.W  A0,A2
        BNE     LOOP10
        MOVE.L  #100H,A0
        DBNE    D6,LOOP9        ;EXECUTED 4 TIMES ?
        BRA     INIT            ;REPEAT ALL SEGMENTS.
```