MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AN EXECUTIVE PROGRAM FOR AN AEROSPACE MULTIPROCESSOR

by

Sumner Curtis Rosenberg

September 1971

Degree of Master of Science

T-552

PREPARED AT

CHARLES STARK DRAPER LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Cambridge, Massachusetts, 02139

T-552

AN EXECUTIVE PROGRAM FOR AN AEROSPACE MULTIPROCESSOR

by

Sumner Curtis Rosenberg
S.B., Massachusetts Institute of Technology
(1969)

Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September, 1971

Signature of Author _____
Department of Electrical Engineering, August 16, 1971

Certified by _____
Thesis Supervisor

Accepted by _____
Chairman, Departmental Committee on Graduate Students

AN EXECUTIVE PROGRAM FOR AN AEROSPACE MULTIPROCESSOR

by

Sumner Curtis Rosenberg

Submitted to the Department of Electrical Engineering on August 8, 1971 in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

In this thesis an organization for an aerospace multiprocessor computer control system is described, and an executive program for this multiprocessor is developed. The executive program consists of several routines which carry out specific executive functions. These routines are designed to be simple and as independent of each other as possible for the sake of system efficiency.

A simulation was written for the proposed multiprocessor system. A set of jobs based on the Lunar Landing programs of the Apollo Guidance Computer were run on this simulation, and a 5 processor system was found to be adequate for efficient performance of the job set. The jobs in this set were then divided into short segments to insure good system response. System performance was then studied using this job set as input to the simulation and increasing the system load by slowing down the instruction execution time. As the system load increased, so did the delays in scheduling jobs. The cause of excessive delays was attributed to the length of time that must be spent ordering a timed job queue.

The simulations showed that the performance of the proposed multiprocessor broke down when job computation loads were more than about 40%. Whether this is a general phenomenon for multiprocessors, and whether ways can be found to circumvent this problem, remain areas for further research.

Thesis Supervisor: Albert L. Hopkins, Jr.

Title: Associate Professor of Aeronautics and Astronautics

ACKNOWLEDGEMENT

The author would first like to express his gratitude to Dr. Albert L. Hopkins for the guidance and motivation he provided throughout the course of this thesis. Second, he would like to acknowledge the assistance of the Technical Documentation Group in the final preparation of this document. Finally, the author would like to express his gratitude to his wife, Pamela, for her understanding and encouragement during the course of this work, and for her assistance in typing the text of the thesis.

This document was prepared using the facilities of PUBLISHER.

This report was prepared under DSR Project 55-23890 sponsored by the Manned Spacecraft Center of the National Aeronautics and Space Administration through Contract NAS 9-4065.

TABLE OF CONTENTS

Page

CHAPTER 1

MULTIPROCESSORS

1.1 Introduction

The use of comuters for control and navigation has made possible space travel as we now know it. But as space missions have become more complex, so have their computer systems. The Apollo Guidance Computer is a single processor computer which is responsible for controlling all systems during a space flight. To be able to handle its varied tasks, this computer had to be multiprogrammed; that is, it had to do the computation for many tasks, but at any instant could be computing only one task. For the NASA Space Shuttle design, the M.I.T.-C.S. Draper Laboratory has proposed the use of a multiprocessor computer. The justification for this proposal is that a single processor computer has become too costly in terms of programming, testing, and lack of flexibility. The proposed multiprocessor design will enable the applications programmer to be much less concerned with the operating system design consequently, the testing of programs will be considerably simpler. Finally, the basic design allows for system changes simply and directly, due to its modularity.

1.2 Multiprocessing Defined

Basically, a multiprocessor computer system is a system where two or more processing units (or processors) share a central memory, where each processor may be executing a distinct task concurrent with other processors. Tasks may communicate with

each other, and receive input and send output through the central memory (see Figure 1.1).

The multiprocessor has an operating system which manages the input and the output (I/O), and schedules the work for each processor. This thesis is especially interested with the portion of the operating system which schedules and assigns the various tasks that each processor must execute. This "scheduler" is commonly referred to as an executive program, or executive. The executive usually consists of executive routines and executive data bases. Based on the information found in the data bases, the executive routines dispatch tasks to each processor; either directly, or indirectly, through the central memory.

1.3 Motivations for Multiprocessing

The use of multiprocessing is motivated by two distinct factors. One factor is the need for extremely fast computation. The other factor is a need for high reliability of the system. Both factors require more performance than an ordinary computer system can provide.

Speed requirements of computers are sometimes greater than some computer systems can provide. Real time simulations and control systems are prime examples of this problem. It may not be possible to complete required computation in the desired time. Computation speed is limited by the available electronic circuitry, and is rapidly approaching physical limitations. A new strategy is to divide the computation into semi-independent segments that can be run in parallel. The parallel processes may then be run concurrently on several processors in a multiprocessor system.

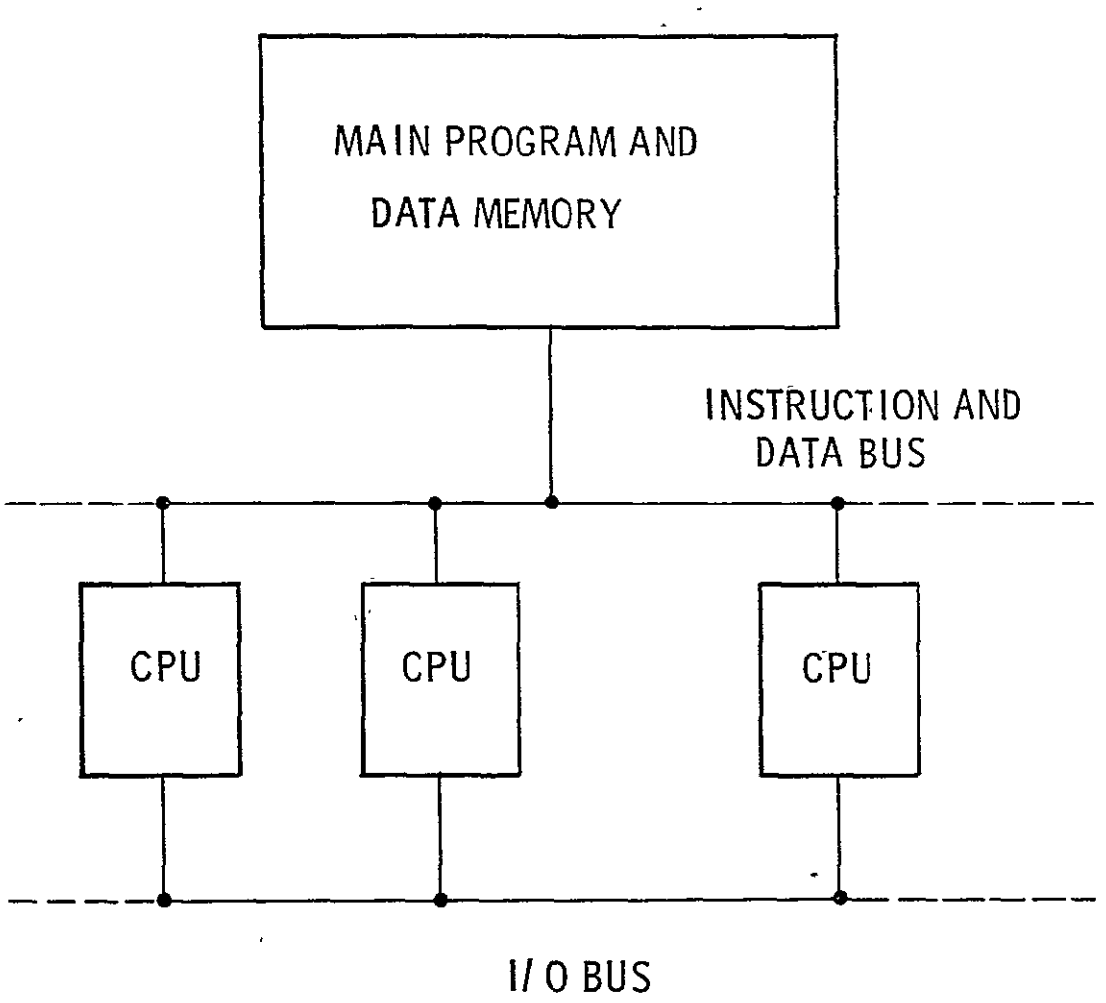

Figure 1.1 Multiprocessor Configuration

It may seem that no matter how fast we can make computers execute a job, the neeed will arise for an even faster system. The concept of multiprocessing opens new frontiers for achieving such goals. The new limiting factors are the number of processors in the system, and the ability of the programmer to divide a job into many parallel operations.

The need for high reliablity computer systems is also served by multiprocessing. In most computer systems the philosophy of reliability is that the components be quite reliable; aside from that, failures will be repaired after the fact. This philosphy is not adequate for vital systems where neither the materials nor the expertise to repair such a system is accessible; most notably on a manned space flight. In such situations individual component reliability will never be sufficient. Consequently, components are duplicated or triplicated, and many strategies may be used to ensure extreme reliablity.

A multiprocessor system lends itself very well to high reliability needs. First, duplication (or more) of computations at the program level can be used to verify that the system is healthy. Second, if there is a failure in one processing unit, the system can still operate on the remaining processing units.

This thesis is concerned with aerospace computing systems, and specifically considers a computer system which was at one time proposed for the NASA Space Shuttle Vehicle. Both of the above - speed and reliability - may be important considerations. For example, a monitoring cycle that must repeat in less time than it takes to finish its computation could possibly be implemented quite easily as parallel procecsses in a multiprocessor. Certainly, it is obvious that reliability is most important in such a vehicle, and thus, fault tolerance allowed by a multiprocessing system is quite valuable.

1.4 Organization of Multiprocessors

Although all multiprocessor systems share certain basic characteristics, the details of each system may differ according to the context in which it is used.

A large time sharing system, for instance, must make most of the decisions independent of the user. That is, the system cannot allow each user to tell it what to do; it must protect itself from selfish or malicious users. The system is designed to be fully loaded, and it must therefore use all of its resources efficiently. These considerations imply that the system must have a large interrupt structure; to handle I/O messages and time interruptions, for instance. Futhermore, to accomplish these objectives, the operating system must be running continuously, acting as a monitor of the rest of the system.

A small aerospace computer system, on the other hand, does not have to concern itself with the problems caused by unknown users. The programs in such a system must be specified and tested beforehand, to see that they have the desired effect. Thus, much of the operating system may be dependent upon the individual programs for control. Also, the system will not usually be designed to run at full load. Instead, spare processors will usually be availiable for I/O handling or as backups in case of processor failure. Consequently, it may be possible to eliminate many types of interrupts from the operating system.

As a result, such a system will be quite deterministic. This has become increasingly important in space applications as testing costs begin to surpass hardware and development costs. An engineer can tell more precisely at the programming phase exactly what effect his program will have on the rest of the system. Other aspects of multiprocessor systems may be independent of the nature

of the system and even of the fact that it is a multiprocessor. Such aspects include handling of hardware and software failures, handling of storage allocation, priority arrangements, and memory protection.

The operating system of a multiprocessor must be designed with regard to many considerations. Most design decisions must be made before a detailed operating system may be developed.

1.5 Executive Organizations

The purpose of this thesis is to develop a detailed executive program for an aerospace multiprocessor system. It is therefore valuable to consider alternative organizations of multiprocessor executives.

One straight-forward way to design an executive system is to have it be a fixed program which monitors all processors. When a change in status of a processor occurs, the executive carries out some action based on the state of the whole system. Such an executive program may reside in a processor specifically designed for executive use, or it may reside in one of the processors of the system. If one processor is specifically designated as the executive processor, the executive is said to be "dedicated." If the executive programs are able to reside in any processor, it is known as "semi-dedicated." The data bases associated with the executive may reside in the executive processor or in the common memory.

The advantage of this type of executive is that most decisions are made within a central operating system, and it can control all processors. Some disadvantages are that such an executive tends to be slow; and that it tends to underuse a processor unit,

since the programs must cycle on one processor even if they have no work.

One solution to the problem of slowness of the executive is discussed by Butler Lampson (Ref. 1). He suggests implementing the executive as a system of hardware modules "which would eliminate interrupts and drastically speed up the software schedule."

Other executive systems fall into the class of "floating" executives. A floating executive system allows any processor to call executive programs to run on that processor. The advantage of this organization is that the individual programs now call the executive as needed, and therefore the system control resides to a large degree within the jobs running on the system. Thus, computing time is spent on running the executive only when necessary. This organization assumes that the jobs being run act responsibly, and therfore lends itself to a closed computer system such as an aerospace control system.

Jack Pariser describes such an executive organization as used on a Hughes H-3118 multiprocessor (Ref. 2). This particular organization allows only one processor to have executive control at a time. That is, if other processors require use of the executive, they must be delayed until the processor having control releases that executive.

This is not a very sophisticated approch to the design of floating executives. The reason for such an approach is to prevent more than one processor from using the executive data base at a time, thereby preventing errors due to memory sharing overwrites. But, at the time that two or more processors are interested in using the executive, they may be concerned with different portions of the data base. Thus it would be advantageous if they are allowed to run simultaneously.

Consequently, the approach taken by M.I.T. in its proposal for a design of a space shuttle multiprocessor is to divide the executive into integrated parts. The executive is now composed of several subroutines to carry out the different functions of the executive. Data bases residing in common memory now reflect the total state of the system. Locks are supplied for independant portions of the data base, and subroutines use these locks to gain control of that portion of memory. This is the basis for the executive design presented in this thesis.

CHAPTER 2

PROPOSED MULTIPROCESSOR SYSTEM ORGANIZATION

2.1 Introduction

In order to develop the detailed executive, the system in which it will be used must first be roughly defined. The executive must reflect those characteristics of the system that affect it. These characteristics include hardware related aspects, such as: processor architecture, memory organization and allocation, communication between subsystems, available operations, and error detection. Also important are software considerations such as: different priorities and types of jobs, I/O requests, subroutine calls, and memory protection. The proposed NASA Space Shuttle multiprocessor organization is developed in References 3 and 4 and those areas which are related to the executive development are presented below.

2.2 Hardware Organization

Many areas of the hardware organization of a computer system affect the structure of the executive. The processors define the working area of the executive programs. The memory defines the size and limits for data transfer and storage allocation. The speed and methods of subsystem communication may indicate what possible tradeoffs may be made. The system operations also define the operation set allowable in the executive programs. Finally, hardware failures may necessitate certain executive actions.

Each processing unit of the multiprocessor under discussion is made up of three structures; a processing element, a scratchpad memory, and interface and error detection logic. The processing element contains an arithmetic unit and operating registers. The scratchpad memory is high speed memory, addressable only by the individual processor, and used to store intermediate results of separate tasks. Three scratchpads and two processing elements are redundantly tied together with the error detection logic. When an error is detected by this logic, the error handling mechanism is signaled. The operation and impact of this mechanism is described later on in this section.

In the design of the executive, we are certainly interested in the register capacity of the processing unit. Implementation of executive programs using only operating registers is desirable for both the speed and the size of the executive programs. For the executive described in the following chapter it was assumed that as many registers were available as were needed. This proved to be reasonable; eight registers in each processor were sufficient.

The programs and data of the system reside in main memory. A 64 K memory is felt to be more than adequate for future uses based on the experience gained from Apollo. Whatever space remains after memory is assigned to programs and data may be used as dynamically allocated erasable storage. Such areas are useful to store temporary variables or arguments to be sent by one job to another. This allows memory to be shared among different jobs as it becomes needed. The responsibility of allocating dynamic storage areas falls on the executive. It must provide the means to find an unused dynamic area, tell a process where that area is, and then return that area to a free pool when it is no longer needed.

Communication between the processors and main memory is here accomplished via a data bus. When an address in main memory is referenced by a processor, a microprogram in the processor signals that it wants to use the data bus. When the processor has been signaled that it has control, it then sends the required code along the bus to accomplish the operation on the memory. The processor retains control of the bus until it decides to release that control. The length of control is usually only for one operation, but it proves valuable for a processor to be able to "hog" the bus when correlated data must be updated as a set or when flagbits must be tested and set without interference from other processors. For the purpose of this study, it will be assumed that the bus is capable of handling all data traffic with negligible delay. A study of bus requirements and associated delays has been done by Efrem Mallach (Ref. 5).

As yet, no operation repertoire has been established for these processors. It would be advantageous, however, for the processing units to have a large set of operations to enable the programs to be written concisely and carried out quickly. Most important to the executive are the availability of flagbit operations that can test the condition of a particular flagbit and operate on it in the same instruction, without interruption. This is accomplished by means of a bus hog as described above. It will be assumed in the design of the executive that such operations exist.

To release the programmer from the concern of what might happen if there should be a hardware failure during execution of his program, a recent concept called Single Instruction Restart (SIR) was embodied in the design. Simply stated, the SIR is used to detect a hardware failure within the instruction that it first occurs. This prevents an error from propagating throughout the rest of the system. When an error is detected, the state of the processor and the contents of the registers and the scratchpad of

the failed processor are dumped into main memory, and a signal is set. The next freed processor will then respond to the signal by reading in the memory dump and acquiring the state of the freed processor. Computation is then continued at the point of failure.

That portion of the executive that is called when a task releases a processor has the responsibility to first check to see if any failure signals have occured. If so, the executive must transfer to a program that accomplishes the restart as described above. Since such a restart program is intimately linked to the details of the SIR, it will be considered part of a class of corrective programs and will not be developed in this study. Also included in this class will be those programs that decide what actions to take on the failed processor.

One interesting result of the SIR should be pointed out. If a processor suffers a hardware failure while it is operating an executive scheduling program the SIR causes the processor to halt. The next freed processor will restart the executive from the point of failure. This prevents system deadlocks due to permanently locked data sets.

2.3 Software Organization

The design of the executive programs is also closely tied to the software structure proposed for the system. The executive must handle different job types and priorities, it must be able to interact with the system's I/O, and it may allow for subroutine calls and parallel processing.

The Space Shuttle must be able to schedule events at a specific times and delays, and it must allow certain events to be contingent

upon the occurrence of other events or conditions. The executive must reflect these requirements, scheduling jobs contingent upon a certain time of arrival or a specific event occuring. As long as it has an acceptable response time, the executive is in the best position to decide that a timed job is due, and thus schedule it. On the other hand, if a job is waiting for some external event to occur, it would be inefficient and complicated for the executive to decide if such an event had occured. This task is handled better in I/O and is discussed below. It is also desirable for one job to be able to request another job to be run as soon as the system is able to schedule the new job. Consequently, the executive will schedule such a job run on the next freed processor, if it decides that the job should run next.

A problem arises over which job to schedule if more than one job is ready to run. A system of priorities must be implicit in the executive structure. Such decisions must be made considering both system and executive performance. One clear fact is that the priority system need not be extremely complex since there are many processors and the job structure will provide a high job turnover, as discussed at the end of this section. A simple way to implement a limited priority structure is to have one job queue of non-timed job requests for each priority, and a time queue for timed jobs. The executive scheduler checks the queues in a predetermined order to find a job to schedule. Jobs within the non-timed job queues are handled on a first-in, first-out basis. A job request is given an implicit priority on the basis of the queue into which the job request is inserted.

It is desirable, when one job schedules another, that data may be passed on as arguments or intermediate data. This is done through the use of dynamic areas, as follows: one job acquires a dynamic area, stores the data to be passed in the dynamic area, and then schedules the object job, associating with it the address

of the dynamic area. Each queue entry must therefore include the address of a dynamic area, if any, along with the job identification, and the time to be scheduled for wait jobs.

Consequently, a task may be distinguished from its corresponding program by the task's associated data. This allows much flexibility in the types of tasks allowed. For instance, one program may run concurrently on two processors, with each processor using a different set of data. Also a job may call itself recursively, passing the same dynamic area on each call, or continually acquiring new dynamic areas. But, perhaps most significant is the ease with which subroutine calls and parallel branching may be used. A subroutine call may be made by having the calling routine schedule the subroutine and pass a return address in the dynamic area. When the subroutine is finished it schedules the calling routine to begin at the return address that was passed. Parallel branching may be considered to be many subroutine calls, one for each branch. As each branch is completed, it tests and sets a flagword to see if all other branches are finished. The last branch to be completed then schedules the calling job.

The individual tasks request other tasks, and in the case of parallel branching they test for themselves whether the branches are ready to join. Such decisions are contained within the tasks, since they are in the position to make the decisions efficiently. On the other hand, such mechanical procedures as choosing the task to be run next or the handling and passing of dynamic areas, are handled best by the executive programs.

The organization of I/O handling is an especially important consideration in the design of the executive, since I/O handling is also an integral part of the operating system. Since input and output data for space missions are both important and voluminous, it is necessary that I/O be handled quickly. Therefore, it is

necessary that the I/O control act as a monitor of the I/O devices. Consequently, the I/O monitor will be continually running on one processor, but since the processors are identical it can run on any processor. Thus, the I/O monitor is a semi-dedicated program which communicates with I/O devices on a separate bus, and with the other processors via main memory.

The I/O processor maintains a list of events and associated jobs. When an event in the queue occurs, the I/O monitor inserts a request for the associated job in the executive queues. In this way, I/O interrupts can be eliminated, greatly simplifying the programmer's job. This method is valid only if the executive response time is sufficiently small to meet system requirements. The response time is the delay between the time the I/O monitor requests the job to be run and the time it is actually run. In the Apollo Guidance Computer, the I/O response time was 10 milliseconds. Therefore, it is proposed that tasks be divided into jobs of less than 10 milliseconds, to be certain that a processor becomes free every 10 milliseconds to schedule the new jobs. This idea will be considered in detail later in this thesis. The mechanics of dividing programs into jobs of the desired length is not considered herein, but will probably be accomplished by a compiler automatically inserting breakpoints in long programs.

2.4 Conclusion

The preceding discussion has been a brief summary of the proposed structure of an aerospace multiprocessor system, with emphasis on the requirements of the executive for that system. An executive design based upon these hypotheses is presented in detail in the next chapter.

CHAPTER 3

THE MULTIPROCESSOR EXECUTIVE

3.1 Introduction

In this chapter, a detailed presentation of an executive for the proposed multiprocessor is given. The functions of this executive are designed to carry out the requirements presented to it by the system organization proposed in Chapter 2. The executive consists of several routines which are presented here in detailed flow chart form. Many assumptions and arbitrary decisions had to be made in designing these routines, so that the executive presented cannot be precisely correct for all conditions. It is rather a model which can be studied and built upon. It is felt, however, that the model presented herein represents a reasonably optimal design under the assumptions made below.

3.2 Structure of the Executive

The executive program is actually a group of routines called by individual jobs to handle executive activities. These executive routines manage a set of memory locations, known as the executive data bases. The data bases contain the information presently needed by the executive subroutines to carry out the desired actions.

What are the actions that must be performed by the executive subroutines? As presented in the system proposal, the desirable actions are:

Request an immediate job

Request a timed job

Acquire a dynamic area

Return a dynamic area

Dispatch the next job to be run

Each of the above actions will be carried out by a distinct executive routine. A routine will be called by a job with necessary information in predescribed registers, and the routine will return information to registers, if necessary. It will be the responsibility of the programmer to see that the interface between jobs and executive routines is handled correctly, although some enforcement of interfacing rules can probably be accomplished in an assembly program prior to run time.

The data bases consist of job queues, pointers, and flagbits. One job queue is required for each priority level, and one job queue will be used for timed jobs (commonly called a waitlist). Three job priorities will be assumed in this model, which should prove to be sufficient in a multiprocessor environment. The entries in the job queues may be more than one word long, and will contain the starting address of a job to be run along with the address of the head of an associated dynamic memory area. In the wait queue the entry must also contain the time that the job is to be dispatched. A series of pointer locations is used to indicate which location contains the first job entry in each queue, to indicate the next free entry in a queue, and to indicate the next available dynamic area. Because the processors of the multiprocessor behave asynchronously, flagbits are necessary to lock portions of the data bases when they are in use. Thus, a processor must gain access to a data base through its flagbit, and then unlock the flagbit when it is finished.

Job queues should be long enough to insure that they will not be filled during normal operation. Although the precise length can only be decided upon by extensive simulation of the actual system programs, it is felt that a length of five times the number

of processors for the job queues and ten times the number of processors for the wait queue is more than sufficient under normal operating conditions, based on results of simulations in this thesis and in Mallach's thesis (Ref. 5).

Five executive routines have been written to perform the actions described above. To request an immediate job, the routine JOBIN is called as a subroutine with the address of the job to be scheduled, the address of the associated dynamic area, the priority of the new job, and the return address to the calling job in specified processor registers. JOBIN inserts the necessary information in a free entry in the specified queue, and then returns to the calling job.

Timed job requests will be handled by a routine called WTIN (for wait insert). The call will be similar to JOBIN, except an absolute time or time delay will replace the priority in a register. WTIN will have two calling points to distinguish between an absolute time and a delay. If the call is a delay time insert, WTIN will first compute the absolute time, since all wait queue entries contain absolute time. Typically, most wait inserts use delay times.

Two routines will manage dynamic area allocation. The names of the routines that allocate and return dynamic storage are GETDYN and FREDYN. They both require two processor registers to pass the dynamic area address and the return address to the calling job. Dynamic areas will be identical blocks of a specific length.

Finally, the routine that dispatches the next job to be run is called END OF JOB, since it must be called at the end of every job. END OF JOB is the most complex of the executive routines. Its sole purpose is to find the next job to be run, and then transfer processor control to the start of that job. END OF JOB first tests the restart flag to see if any jobs are to be restarted due to

processor failures. Otherwise, the program tests the job queues in order of priority to find a job to run. If no jobs are available to run, END OF JOB enters a delay loop and then starts over again on the same processor.

The flow charts for these routines along with a detailed discussion of their operation is presented in Section 3.4.

3.3 Assumptions

In this section some basic assumptions will be made which are necessary in order to develop the detailed executive routines.

As mentioned in Chapter 2, the size of main memory will be approximately 64 K words. The basic word size will be assumed to be 32 bits. This word size allows for much more precision than the 16 bit word used in the Apollo Guidance Computer. Conveniently, the word size also is twice the length necessary to specify any address absolutely. This fact indicates the desirability of half word operations. Finally, the word size makes it possible to use absolute time in wait job requests and wait queue entries. If the 32 bit word is used to represent milliseconds, the maximum time that can be represented by one word is about 50 days. This is certainly sufficient for present space missions, although future missions will probably be of longer duration. Handling of the wait queue is much simpler using absolute times, so it would be desirable to present a solution to the problems presented by clock overflow. This will be discussed in Section 3.10.

The processing units will have at least eight working registers of basic word length. The instruction set will consist of register-to-register and register- to-memory basic instructions. These will include the usual instructions to carry out load, store,

arithmetic, logical, shifting, testing, and conditional branching operations. Also, there will be flagbit operations, including an operation that tests and sets a flagbit in one bus hog. Finally, to save memory space and bus transmission time, halfword operations for loading and storing in main memory will be assumed to be available.

3.4 Strategies

General strategy and tradeoff decisions must be made before the detailed executive programs are designed. For example, it was previously stated that it is desirable to break up the executive data base into independent sections so that there is as much parallelism as possible in the executive routines. This division of the data base is accomplished by means of locks which are tested by the executive routines before accessing the associated portion of the data base. Strategy decisions must be made on how to divide up the data bases, how many locks to use, and where they should be tested in the executive routines. Certainly a waitlist insert can occur in parallel with a normal job insert and a dynamic area allocation. Thus, there should be seperate locks on each area of the data base, to allow WTIN, JOBIN, and GETDYN to run in parallel.

The question is how far we should carry out such strategies; and to answer this we must consider the tradeoffs. The three criteria for making these decisions are the total running length of the executive routines, the running length of instruction sequences during which locks are set in the executive routines, and the amount of memory taken by the executive programs and their data bases.

The last criterion will consistently be the least important in design decisions. Certainly, to allocate one bit in memory for an extra flag to allow two routines to run in parallel is a profitable choice. In fact, since executive programs are vital and run so often, it seems desirable to use any reasonable amount of memory to save even a small amount of running time in an executive routine.

On the other hand, the length of time that the data bases are locked is extremely important, especially as more processors are added to the system. Indeed, as is pointed out in a paper by Madnick (Ref. 6), this value is the limiting factor in the number of processors that can run on a multiprocessor system. Consequently, as more of the data base is divided into independent sections, less conflict will arise. However, to break up some portion of the data base, some additional instructions may be necessary. An example of this possibility is presented in the executive routines herein. Although END OF JOB and JOBIN use the job queues, the executive routines allow END OF JOB to remove an entry from a job queue in parallel with JOBIN inserting an entry in the same queue.

The decision in this example is whether the few additional instructions necessary are worth the additional lessening of conflict. Obviously, it would be good if the additional instructions do not add lock time to another data base. But perhaps more important is the addition to the total execution time of the executive routine. In a single processor computer this factor would certainly decrease the efficiency of the executive. But, in a multiprocessor, conflict can be more damaging to efficiency than the addition of total execution time, and can certainly lower executive response time. With this in mind, the executive routines presented try to attain maximum parallelism.

Another point in favor of this choice is that it is especially desirable to separate the locks associated with END OF JOB from the other routines as much as possible. Since many END OF JOBS may be cycling in idle loops, it is possible that they might monopolize a lock, causing great delay to a routine trying to do useful work.

Related to the minimization of lock time are the delays taken by other routines (which usually means another incarnation of the same routine) waiting for access to a lock. Considering only the executive efficiency it would be best if the routines had no delay and simply looped on the test instruction until gaining access. But this method may cause excessive data bus traffic, thereby slowing down other processors. It would therefore probably be desirable to have a delay of about 1/4 to 1/2 the average lock time. Since this problem is beyond the scope of this study, the positions of possible delays will be presented in the flow charts, but no actual delays will be recommended.

Detailed flow charts of the executive routines will be presented in the next five sections along with a narrative of what is actually being done and what decisions, tradeoffs, and methods are being used.

3.5 JOBIN

JOBIN is the routine called by a job to insert a job request into one of the immediate dispatch queues. The flow charts for JOBIN are presented in Figure 3.1. In the discussion below, parenthesized numbers refer to like numbered portions of the flow chart. The eight registers are signified by the notations R0 through R7. A register notation contained within square brackets in the flow charts signifies the contents of the location whose address is in that register.

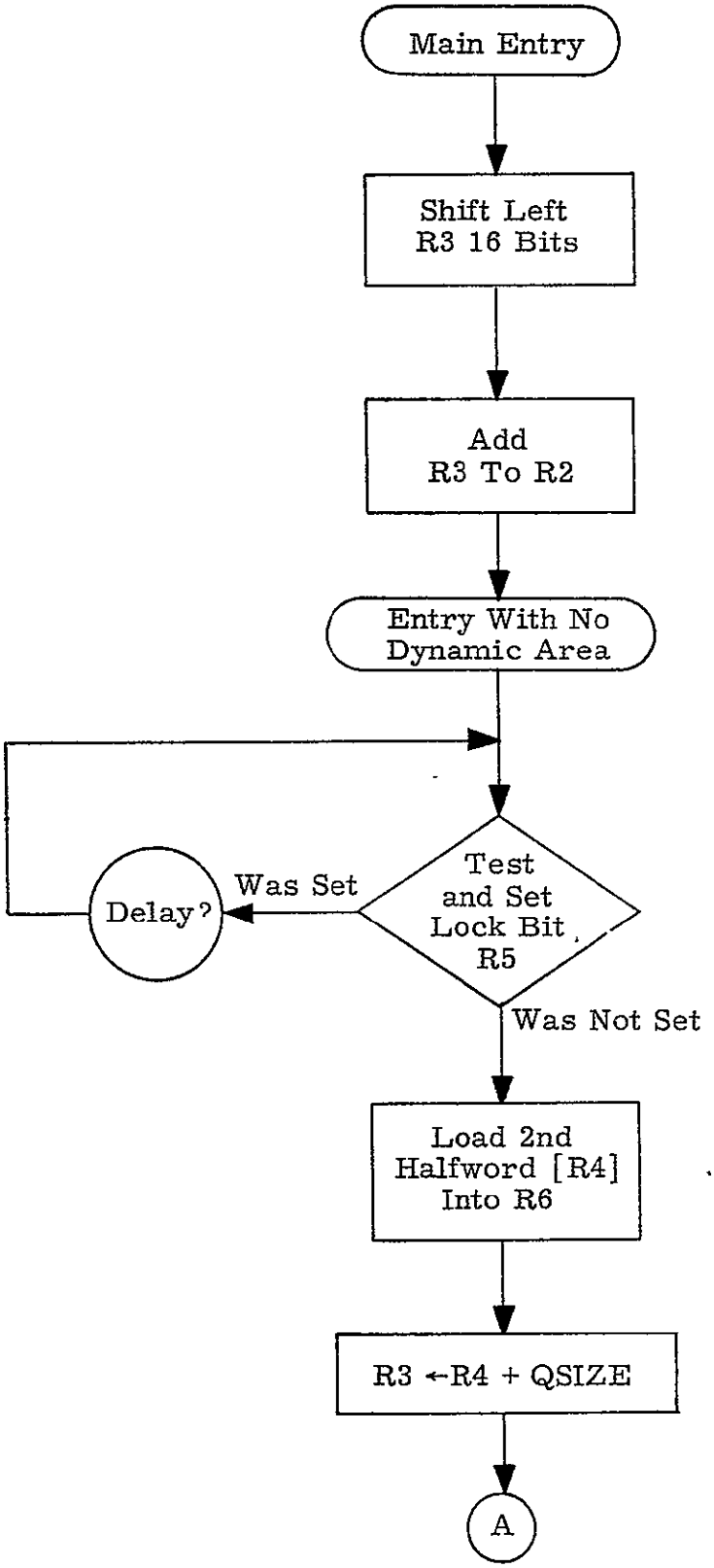

Figure 3.1 JOBIN

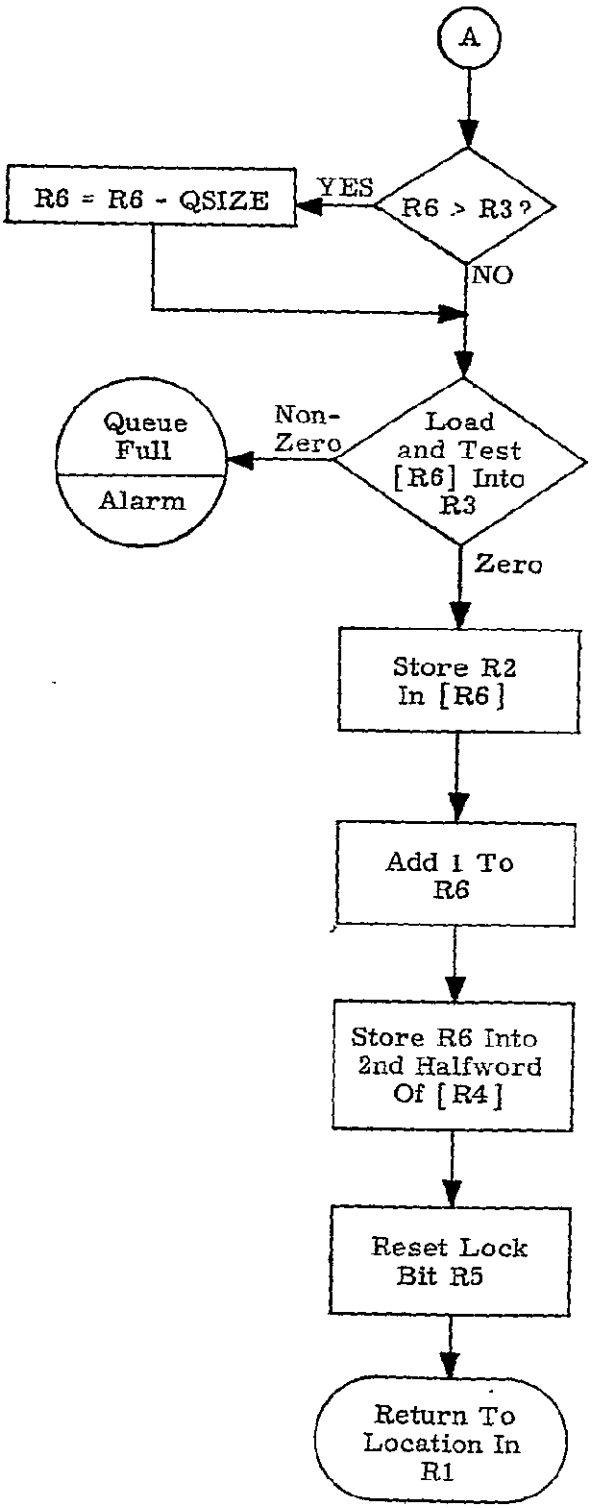

Figure 3.1 (Continued)

By convention, the routine will be called with the return address in R1, the address of the job to be scheduled in R2, the address of the associated dynamic area in R3, the address of the head of the correct priority queue in R4, and a flagbit code for the corresponding queue lock in R5. Thus, each priority queue has a lock to prevent another processor from entering a job in that queue at the same time. Actually, a few instructions could be eliminated in the execution time of any such call of JOBIN by duplicating the subroutine for each priority queue and eliminating a few decision points as well as input registers required. The information found in R4 and R5 would be implicit in the routine called. Such duplication would probably be worthwhile if the number of priorities is small and memory space is not at a premium. But, for ease of presentation, JOBIN is presented here as one routine. Finally, if the job insertion is the last action of a job, it is more efficient to load R1 with the address of END OF JOB rather than a return address.

The main entry of JOBIN is for jobs with associated dynamic areas (1). The routine then changes the contents of R2 into a dynamic/job address pair by shifting the dynamic area address in R3 left 16 bits, and adding it to the job address in R2 (2, 3). If there is to be no associated dynamic area, these actions can be bypassed by a secondary entrance (4). Then the flagbit indicated in R5 is tested and set (5). There is a lock bit for each queue which allows only one job to be entered in a queue at any time. Jobs may be entered in other queues concurrently. If the lock was already set, the instruction is repeated, possibly after a delay (as discussed above).

Each job queue will allow a specific number of entries, each one word long. This number will be called QSIZE. The length of the queue will then be QSIZE +1, with the first word containing a pair of addresses used as pointers (See Figure 3.2). The address

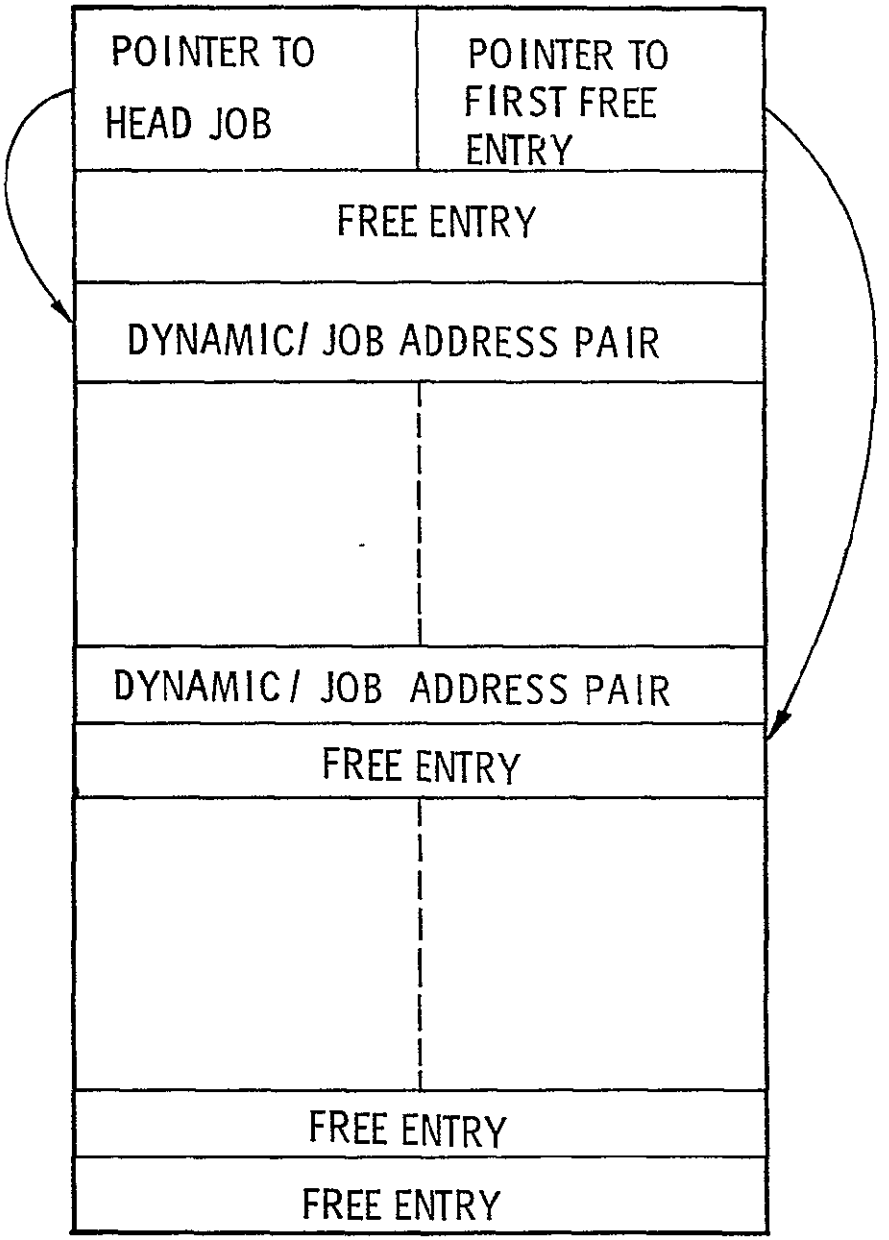

Figure 3.2 Job Queue

of this word will be the base address of the queue. The job queue will be handled as an unordered ring of entries. The first half of the base word will point to the head job in the queue, and the second half word will point to the first free entry in the queue. Therefore, the routine next loads into R6 the free pointer of the queue whose base address is in R4 (6). R4 + QSIZE is put in R6 to indicate the maximum address of the queue (7). If the free pointer (R6) is larger than the value in R3, the next free entry is the first address of the queue, and R6 should be decremented by QSIZE (8).

The routine must next be sure that the queue is not full. If the location pointed to by R6 is non-zero, there is still a job request in that entry - in fact it is the head entry of the queue (9). Hopefully, the queue will be long enough for this never to happen, but it must be tested to prevent overwriting of necessary data. A loop may be suggested here if the queue is full, but this would probably lead to deadly embrace and should not be done. A full queue probably indicates system trouble, and the program should therefore branch to an error routine and give an alarm. The details of such actions are beyond the scope of this thesis.

When a free entry is found, the dynamic/job address pair in R2 is stored in the entry pointed to by R6 (10). Next R6 is incremented by one word and stored in the free pointer in the second half of the word whose address is in R4 (11, 12). Finally, the lock bit is cleared and the routine transfers to the return address found in R1 (13, 14).

3.6 WTIN

Executive routine WTIN is called to insert a job request into the wait queue along with the time that it is scheduled to run. The flow charts for WTIN are presented in Figure 3.3.

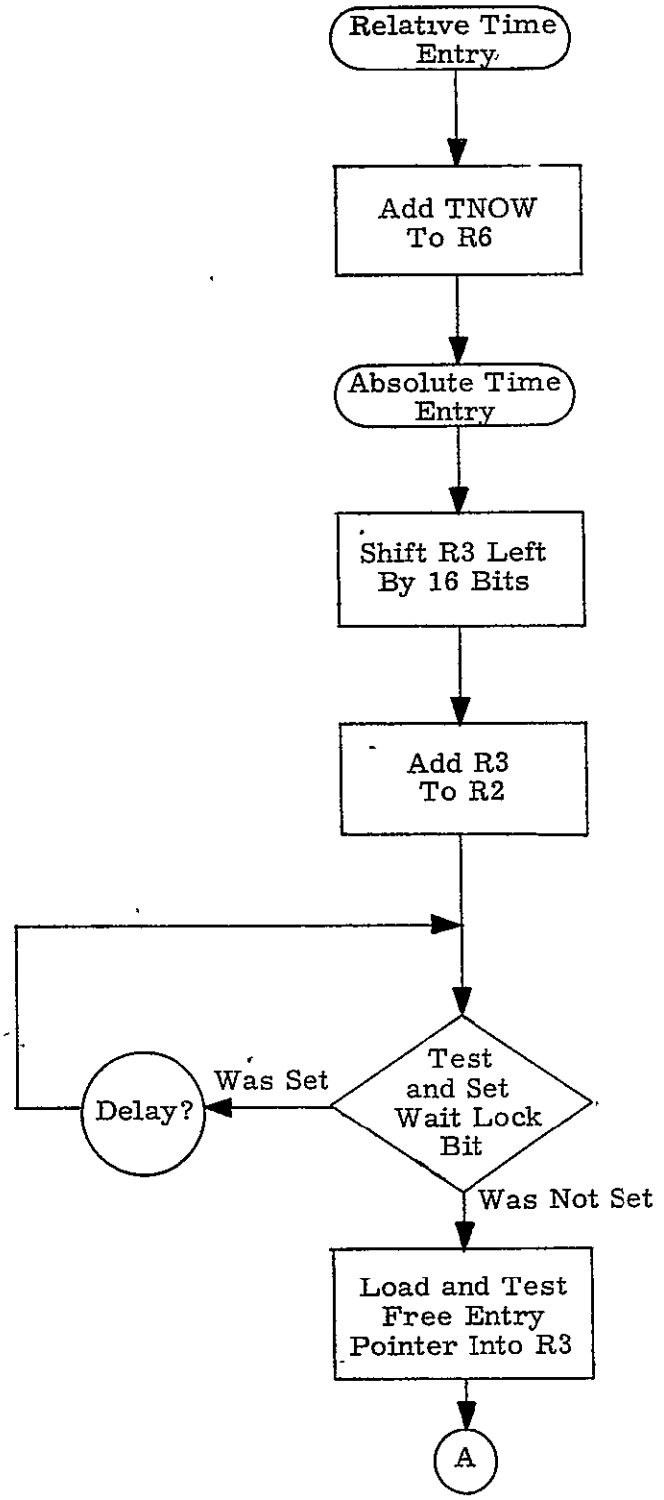

Figure 3.3 WTIN

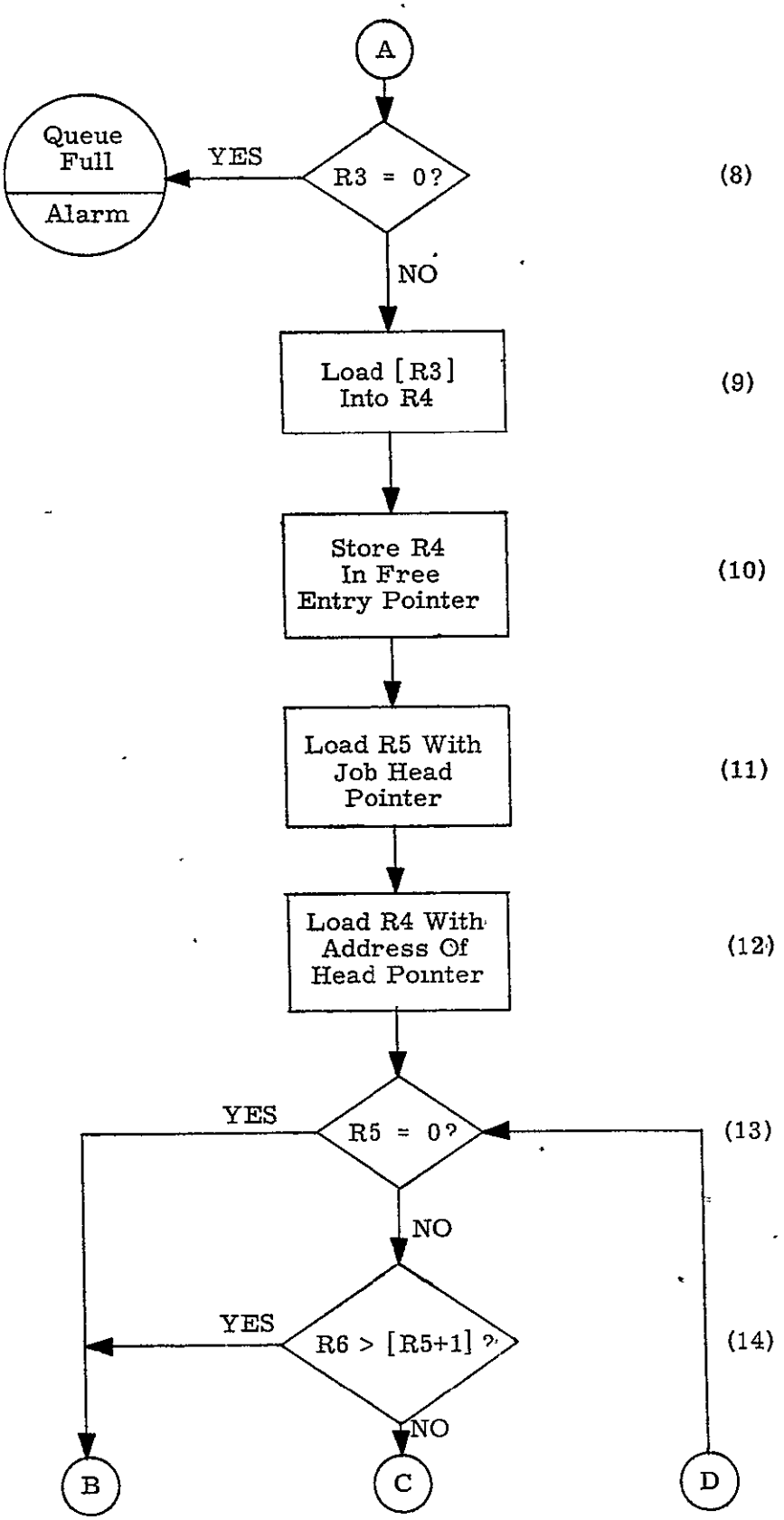

Figure 3.3 (Continued)

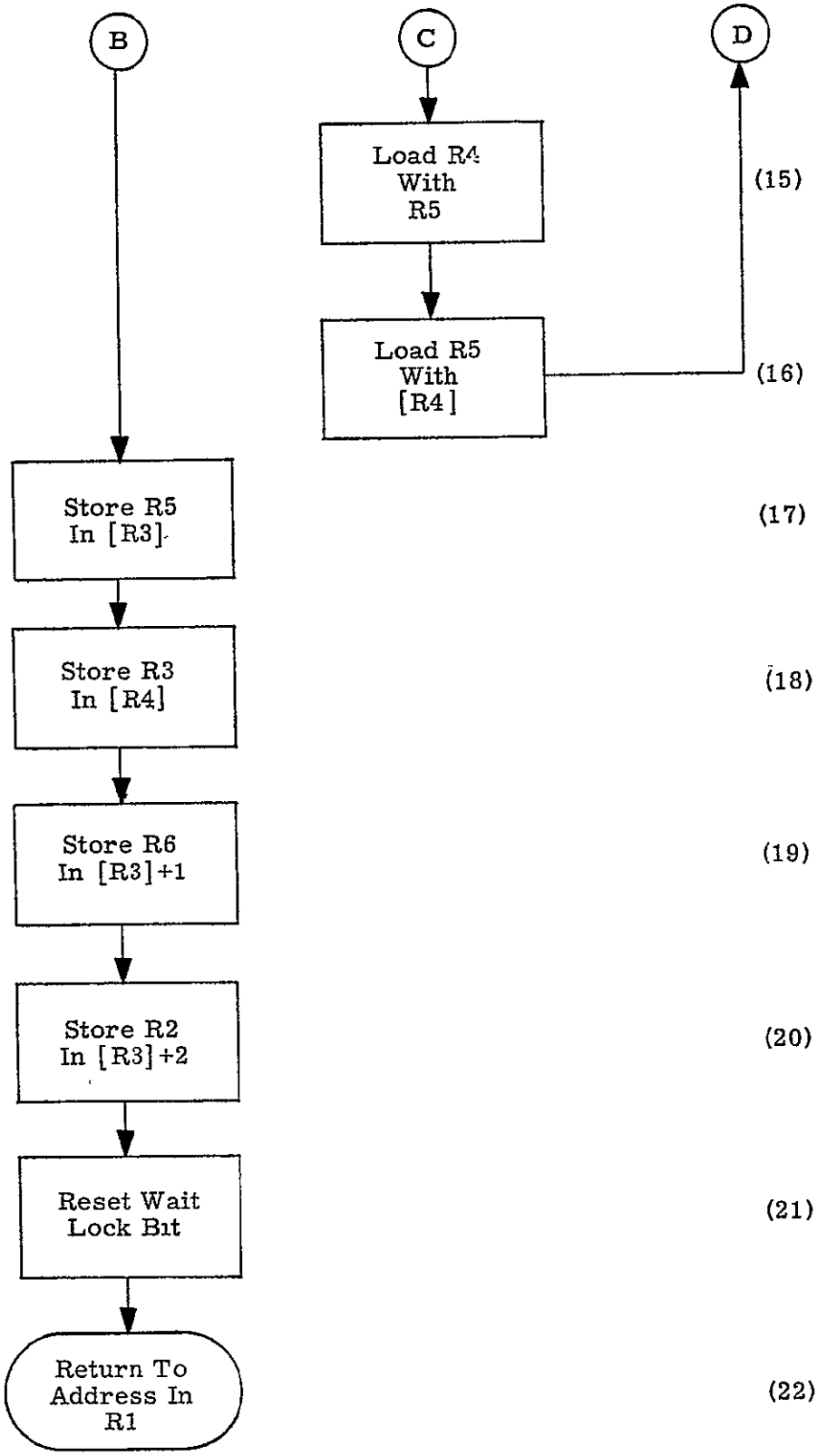

Figure 3.3 (Continued)

WTIN is called by a job with the return address in R1, the address of the job to be scheduled in R2, the address of the associated dynamic area in R3, and a value for time in R6. The time value is treated as either an absolute time or as a relative delay, depending upon where the routine is entered. The return value in R1 should be the address of END OF JOB if the wait insert is the last action taken by the calling job.

The main entry of WTIN is for jobs being requested after a delay relative to the present time (1). We will assume that a register called TNOW exists as a common read-only register to all processors, which acts as a clock. Thus, the routine will add TNOW to R6, so that R6 now contains the absolute time the job is to be scheduled (2). A job requested with absolute time already in R6 will enter after that point (3). Next, as was done in JOBIN, the routine will now put the dynamic/job address pair in R2 (4, 5).

The wait queue must be ordered in some manner so that the job dispatch routine need not go through the whole list on each cycle to find the entry with the smallest time. Therefore, the list will be ordered as each entry is added, thereby insuring the minimum number of sorts of the wait queue. The most efficient way to order such a list is by the use of "threaded lists." That is, each entry in the list includes a pointer to the entry that comes next in order. Also, since unused, or free, entries will be scattered throughout the memory area, a thread of free entries will also be used. Originally, this free list must be initialized so that all entries of the queue are in the free thread. Thus, the first half word of the head of the wait queue will contain a pointer address of the first entry of the ordered thread. The second half word will contain a pointer to a free entry which acts as the head of the free list. Consequently, each wait queue must contain three words: a pointer word, a dynamic/job address word, and an absolute time word (See Figure 3.4).

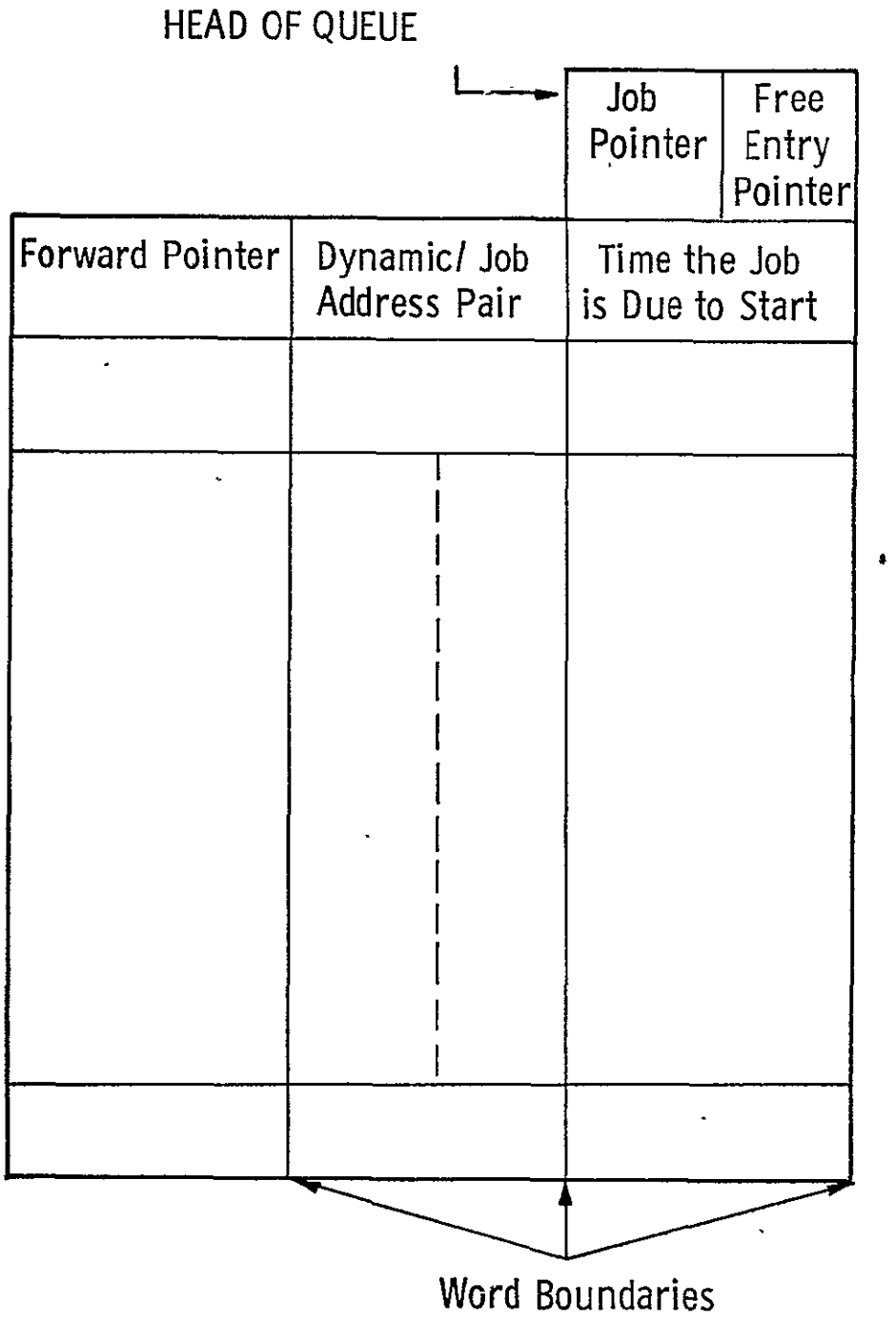

Figure 3.4 Wait Queue

Because the wait queue is an ordered list, its use is more complex than the use of the non-timed job queues. Therefore, the wait queue has one associated flagbit that locks out all other users of the queue, when one routine is using it. This flagbit is now tested and set by the routine (6). If the bit was previously set, WTIN loops back to try the test again, possibly after a delay. When WTIN gains access to the wait queue it first looks for a free entry in the queue in which to put data. It gets the address for an entry by loading the free entry pointer in R3 (7). By convention, if the value in R3 is zero, that queue is full; and a program alarm is sent out (8). Otherwise, the value in R3 points to the pointer word of the first free entry. The pointer word of this entry points to the next free entry, and must therfore be stored in the queue free pointer location (9, 10).

The next problem is to find where in the threaded list of requests the new entry must be placed. The routine loads R4 with the address of the queue head (11), and then loads R5 with the contents of the first halfword of the queue head, which is the pointer to the first member of the ordered list (12). It may be that no entries are present in the list, in which case the head pointer will have zero value (13). Then the routine can simply insert the new request as the first member of the list. On the other hand, the queue may have one or more entries. Then the value of the time of the job to be entered must be compared to the time of each entry in the ordered list until either the time of the new entry is less than that of an item in the list, or until the end of the ordered list is reached, in which case the forward pointer of the last word will be zero (14, 15, 16). The address of the entry that will immediately precede the new entry in the ordered list is left in R4, and the address of the entry that will immediately succeed the new entry is left in R5. The routine then stores the contents of R5 into the first word of the new entry, pointed to by R3 (17). Then the value in R3 is placed in the address pointed to

by R4; that is, in the forward pointer of the preceding entry (18). Thus, the list remains ordered by its time entries.

Finally, the time the new entry is to be run and the dynamic/job address pair are put into the second and third words of the new entry, whose first word is pointed to by R3 (19, 20). The wait queue lock bit is then reset (21), and WTIN transfers to the return address found in R1 (22).

3.7 GETDYN

The dynamic area handling routines are very simple. Basically, they manipulate a thread of free dynamic areas in much the same way as WTIN handles the free entry thread.

GETDYN is called with a return address in R1. The routine returns the address of the head of a dynamic storage area in R3. The flow chart for GETDYN is presented in Figure 3.5. GETDYN utilizes the first word of a dynamic area as the forward pointer to the next free dynamic area. These entries must be initialized so that all dynamic areas are in the free thread. Also there is a word in the executive data base which is used as the head of the free list pointer. Associated with this word is a flagbit, which allows only one processor at a time to work on the thread to get a dynamic area.

When called, GETDYN first tests and sets the associated flagbit (1). If the bit was previously set, the routine loops on the test instruction until the processor gets access to the head pointer. R3 is loaded with the value in the head pointer of a free dynamic area (2). R2 is then loaded and tested with the value in the first word of the new dynamic area pointed to by R3 (3). If this value is zero, there are no more dynamic areas left, and a program

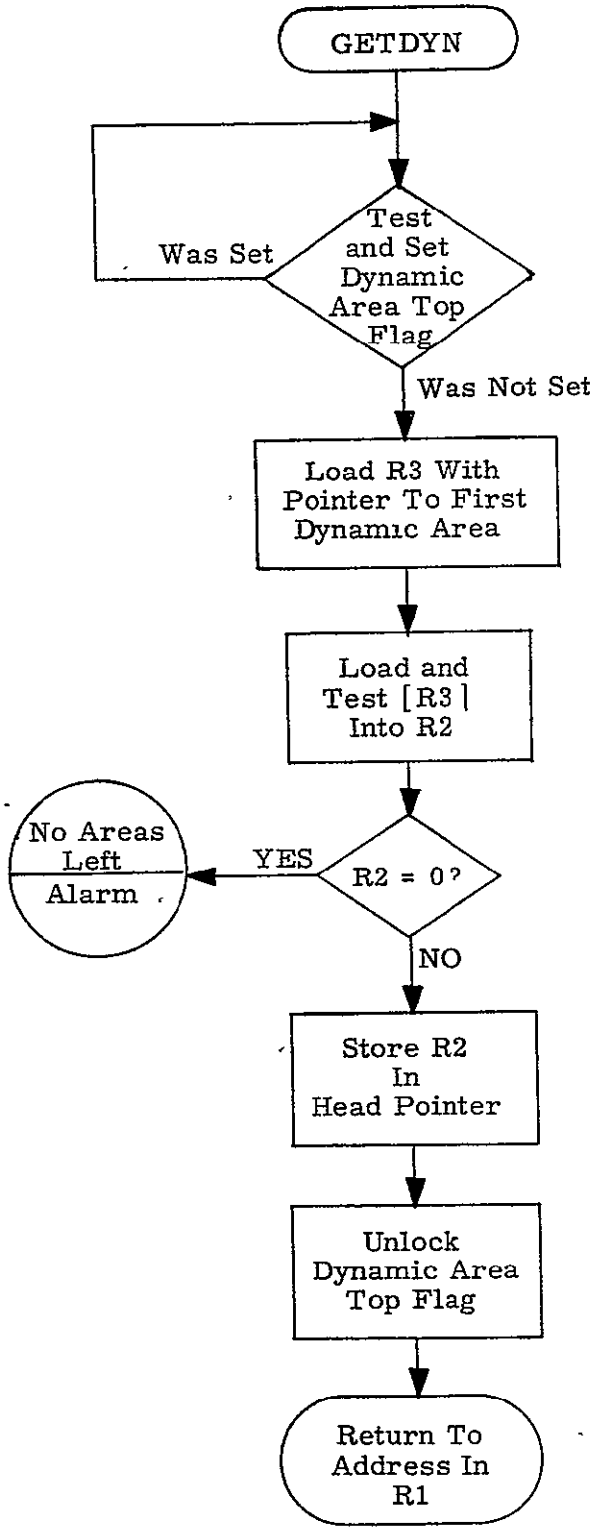

Figure 3.5 GETDYN

alarm is given (4). Otherwise, this value is the pointer to the next free area and is loaded into the head pointer (5). The flagbit is reset (6), and control is transfered to the return address found in R1 (7).

3.8 FREDYN

Freeing a dynamic area is almost the reverse of getting one, except that in FREDYN, the free area is added to the end of the threaded list. This method allows areas to be fetched and freed simultaneously. It requires one word to be used as a tail pointer, and an associated flagbit to lockout other users of FREDYN. The flow chart for FREDYN is presented in Figure 3.6.

FREDYN is called with the return address in R1, and the address of the dynamic area to be freed in R7. The first action of the routine is to zero the first word of the dynamic area pointed to by R7, since that word will be the forward pointer of the last entry of the list of free dynamic areas (1). Next, the flagbit is tested and set, looping back if the bit was previously set (2). The value of the tail pointer is then loaded in R2 (3). The address in R7 is now stored into both the first word of the area pointed to by R2 and the tail pointer (4, 5). Finally, the flagbit is reset and FREDYN returns control to the address in R1 (6, 7).

3.9 END OF JOB

END OF JOB is the longest and most complicated of the executive routines. It must make the decision of which job to run next from all the various possibilities. Implicit in the routine's design must be a priority structure, based upon the order by which END OF JOB considers the various queues. It was decided in the

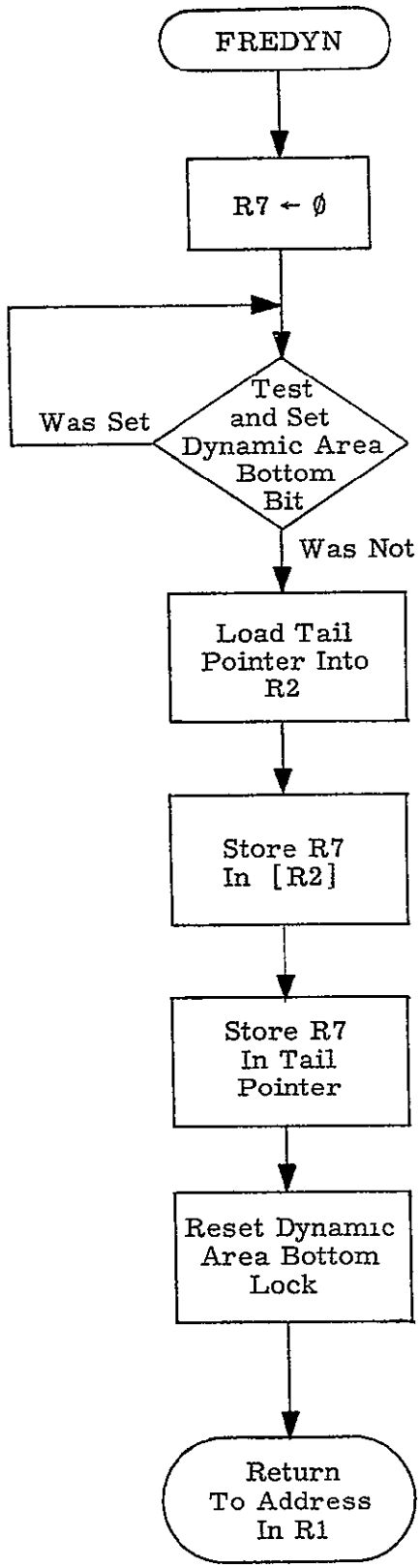

Figure 3.6 FREDYN

presented routine that the wait queue will be considered first, since by their nature, timed requests should have good response time. Then the remaining job queues are scanned in order of prescribed priority, if there was no job to be dispatched from the wait queue. This design is by no means hard and fast, but was chosen because of the simplicity of considering like queues as a group. Consequently the highest priority is accorded to jobs made via WTIN with a near zero absolute time. The flow charts for END OF JOB are presented in Figure 3.7.

END OF JOB is called at the end of every job, and when it finds the next job to run it transfers control to the start of that job, placing the address of an associated dynamic area in R7. When entered, the first thing END OF JOB does is to test and reset the restart flag (1). If this flagbit was set, there is a job on a failed processor that has signaled to be restarted by means of the Single Instruction Restart as discussed in Chapter 2. The routine branches to a special program that handles restarts. The details of such a program are beyond the scope of this study. If the flag was reset, END OF JOB then tests and sets the executive lock flagbit (2). The purpose of this lock is to prevent other END OF JOB routines from beginning until the flag is reset. If the flag was previously set, END OF JOB loops back to its entry point, after a possible delay (3).

When END OF JOB gains access to the queues, it first looks to see if there are any entries in the wait queue, by loading the value of the wait queue head pointer in R1 (4). This value will be zero if there are no entries, in which case END OF JOB branches to check the job queues (5). If there are entries in the wait queue, END OF JOB need only compare the time of the first entry with TNOW (6, 7). If TNOW is less than the first job time, no wait jobs are due to run, and END OF JOB branches to check the job queues for entries. Otherwise, TNOW will be greater than the

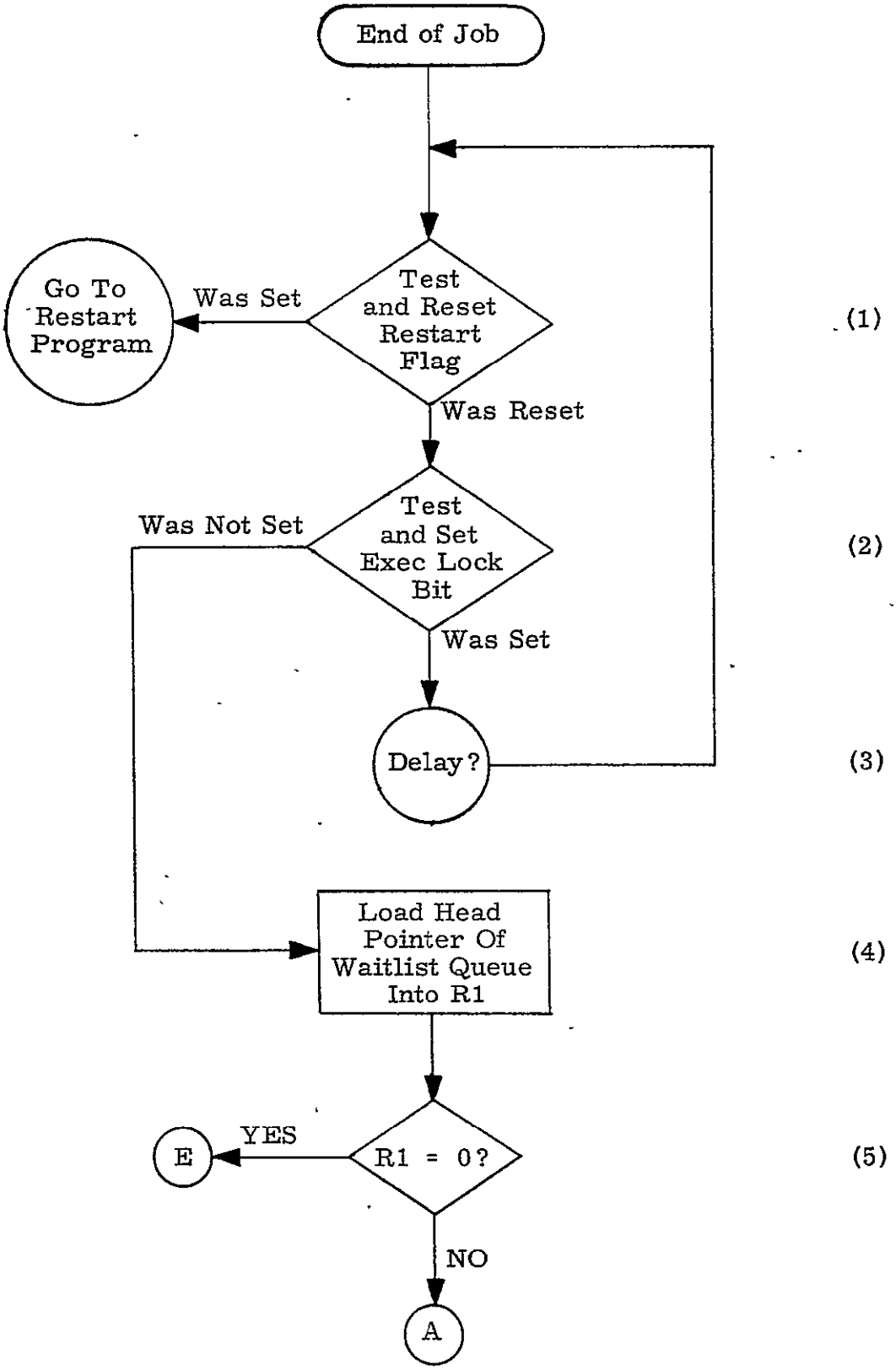

Figure 3.7 END OF JOB

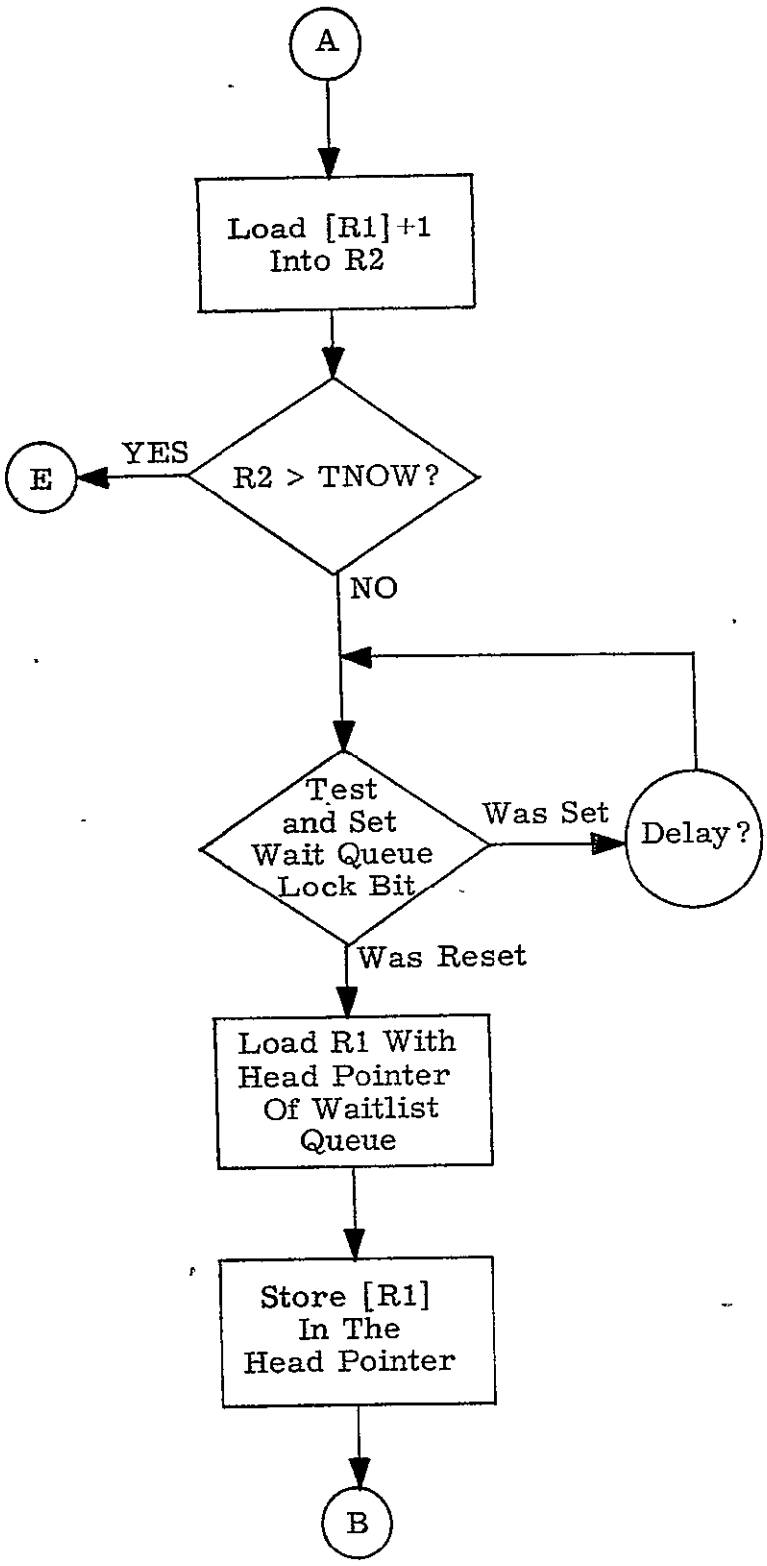

Figure 3.7 (Continued)

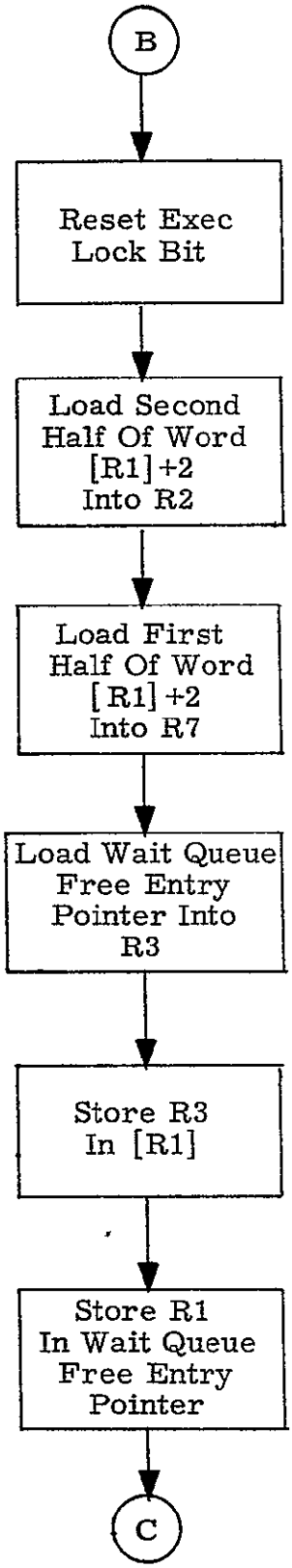

Figure 3.7 (Continued)

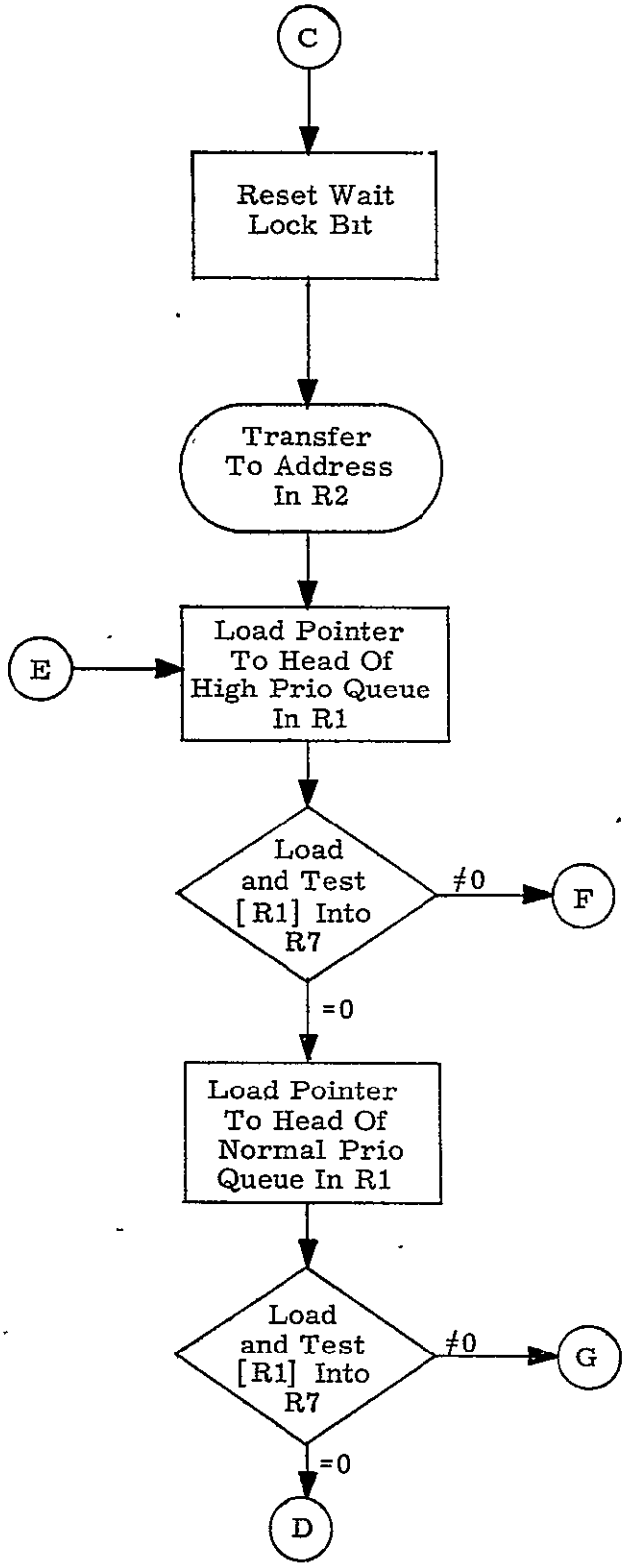

Figure 3.7 (Continued)

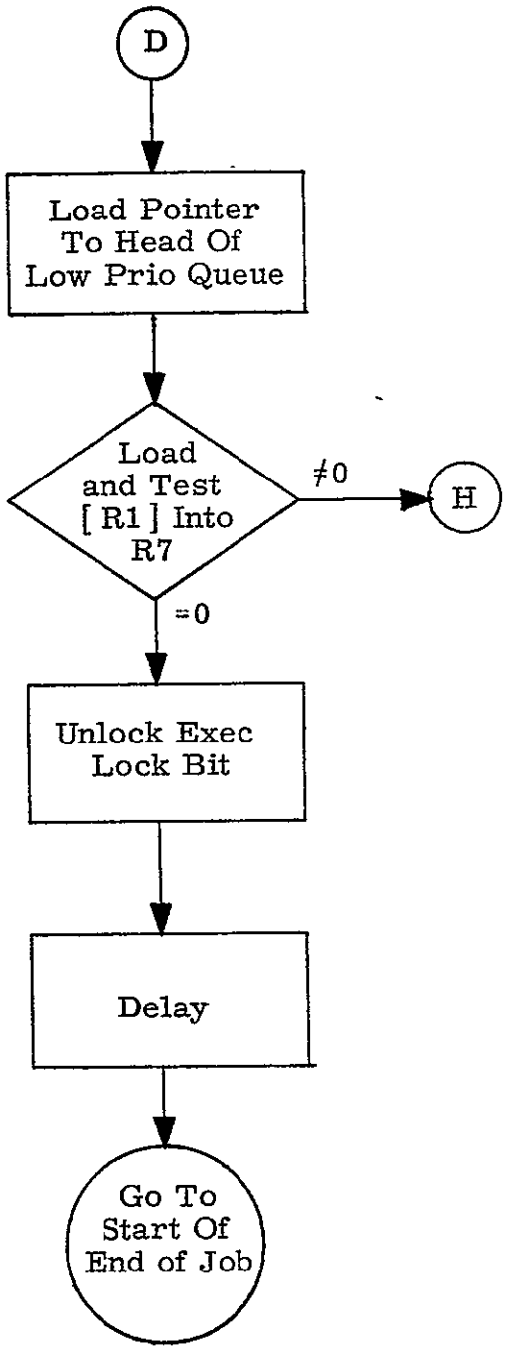

Figure 3.7 (Continued)

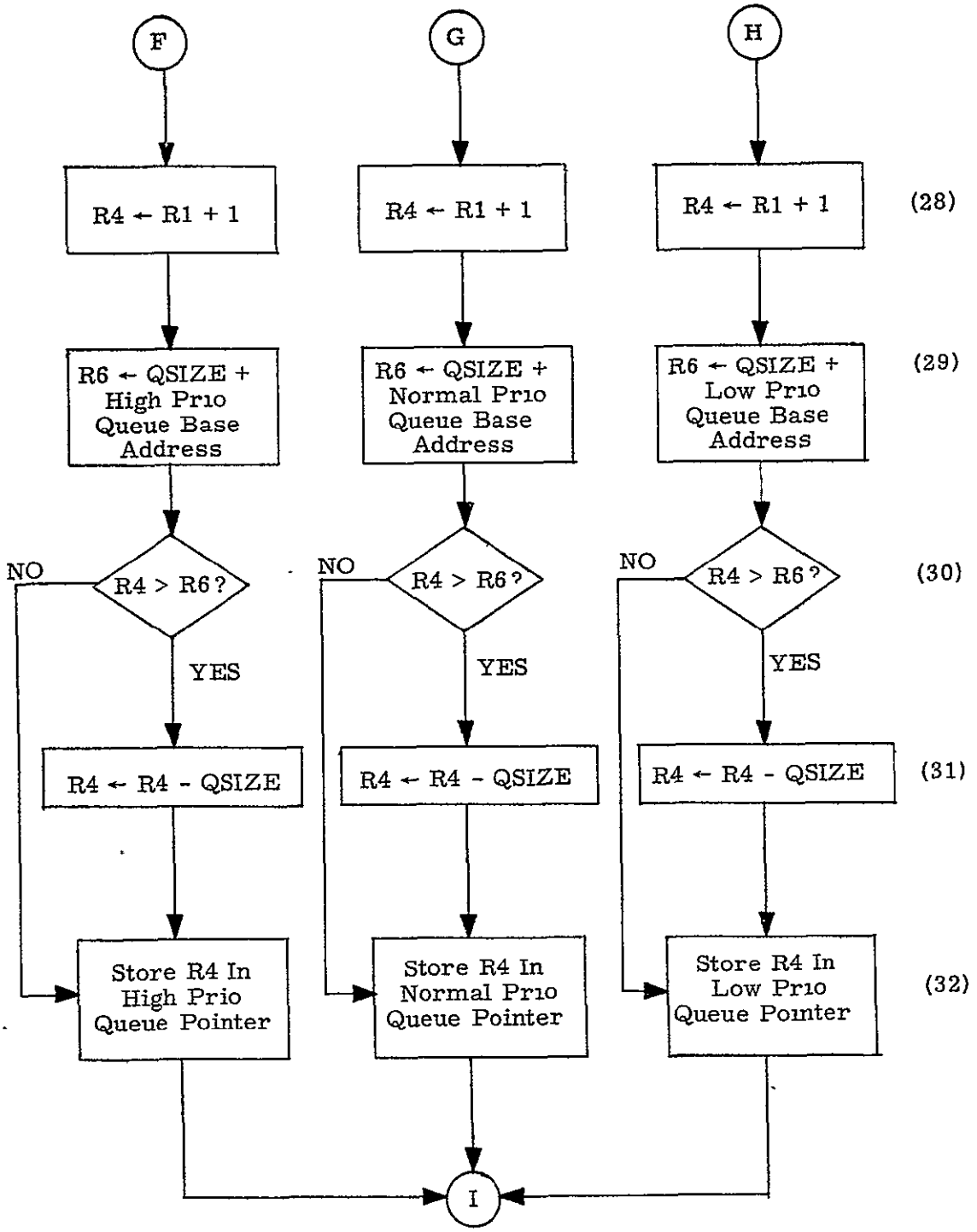

Figure 3.7 (Continued)

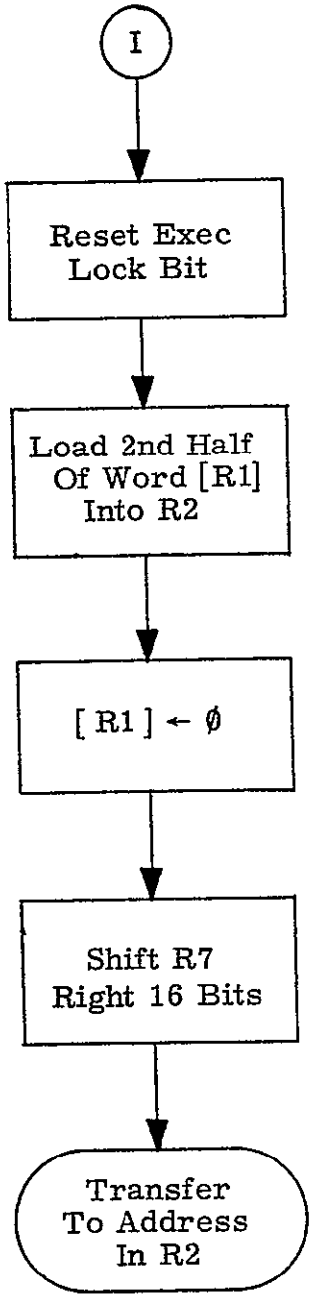

Figure 3.7 (Continued)

time of the first wait entry, and END OF JOB will therefore call a wait job to be run on its processor. But, before the routine is allowed to manipulate the wait queue, it must test and set the wait queue flagbit (8). This is done to prevent WTIN from also using the wait queue at the same time. If the flagbit was previously set, the routine loops on the test and set instruction, with a possible delay, until it gains access to the wait queue.

END OF JOB then reloads R1 with the contents of the head pointer in case a new head job was inserted into the wait queue before it was locked (9). Next, the forward pointer of the job to be run is stored in the head pointer (10). At this point the END OF JOB may allow other END OF JOBS to run without any danger, so it resets the executive lock bit (11).

Now END OF JOB is able to dispatch the job in the entry pointed to by R1. The second half of the entry's dynamic/job address pair is loaded in R2 (12), and the first half is loaded in R7 (13). Next the entry is returned to the free thread. The wait queue free pointer is loaded in R3 (14). The address in R3 is stored in the forward pointer of the newly-freed entry (15), and the address of this entry, which is still in R1, is stored in the wait queue free pointer (16). The wait lock bit is then reset (17), and the END OF JOB transfers control to the new job, whose starting address is in R2 (18).

If END OF JOB finds no timed jobs due to be run, it branches to the part of the routine that checks the regular job queues. This section tests each queue in order of queue priority for an entry. The pointer to the head of the high priority queue is loaded in R1 (19). Then the entry pointed to by R1 is loaded and tested in R7 (20). This value could be zero, in which case the queue is empty and the remaining queues are similarily tested (21-24). If the three job queues are empty, then there are no jobs to be dispatched. The routine will then unlock the executive lock bit

(25), and go into a delay loop (26). After this delay, it transfers to the beginning of END OF JOB (27), repeating the routine until a job is found for the processor.

If the value loaded into R7 above is non-zero for some queue, that value represents the dynamic/job address pair for the next job to be run. The forward pointer value in R1 is then incremented and put in R4 (28). END OF JOB then loads R6 with a constant representing the maximum address for any entry in the associated queue (29), and compares this with the value in R4 (30). If R4 is longer, it must be decrement by QSIZE, the size of the job queue, so that it will point to the top entry word in the queue (31). In either case, the value in R4 is then stored in the head pointer of the associated queue (32).

At this point, the executive lock bit can be reset, to allow other END OF JOB routines to run (33). The second half of the entry pointed to by R1 is then loaded into R2 (34). This is the starting address of the next job. Then the entry pointed to by R1 must be zeroed (35). Next, the contents of R7 are shifted right by half a word, leaving R7 with the address of the new job's dynamic area (36). Finally, the routine branches to the address in R2, and the processor begins the new job (37).

This completes the presentation of the five executive routines. The use of registers by these routines is summarized in Table 3.1.

3.10 A Simple Solution To The Clock Overflow Problem

It is desirable that all times be absolute in the waitlist queue so that a threaded list may be used. The proposed clock word will be 32 bits (or 31 bits, depending on whether a sign bit is

	JOBIN	WTIN	GETDYN	FREDYN	END OF JOB
Call With:					
Return Address In:	R1	R1	R1	R1	
New Job Address In:	R2	R2			
Dynamic Area Address In:	R3	R3		R7	
Prio Queue Base Address In:	R4				
Queue Lock Bit Code In:	R5				
Time Delay or Absolute Time In:		R6			
Routine Returns:					
Dynamic Area Address In:			R3		R7
Routine Also Uses These Registers:	R6	R4,R5	R2	R2	R1,R2,R3,R4,R6

Table 3.1 Fixed Register Conventions for Executive Routines

used). If the smallest bit indicates a millisecond, 32 bits will allow a 50 day clock. If the mission is longer than the maximum time of the clock word, overflow problems result when the clock passes the maximum time and entries still in waitlist queue have a lead bit equal to 1. These jobs will now not be run for up to 50 days!

A simple solution to this problem is to initialize the wait queue with a request for a "cleanup" job when TNOW is 1000...0, that is, one half the maximum expressible time. When this job is scheduled in 25 (or 12) days, we know that TNOW will be greater than 1000...0. A special instruction will be included for use by the cleanup job to kick the lead bit of the TNOW clock to zero, and increment a high order clock word. After the cleanup job carries out this instruction, it must make certain that all pending inserts via WTIN with leading bit values of one are completed. This can be accomplished in two ways.

The first way to make certain that all such jobs are inserted in the wait queue is to delay the cleanup routine after it sets the lead bit of TNOW to zero. To do this cleanup may test and set the executive bit to prevent interference from END OF JOB. Then a delay, possibly equal to the number of processors times the time it takes to insert a job via WTIN, will be taken. This delay could be made long enough so that all wait job inserts that have leading bits of value one in their time entry have a high statistical certainty of being inserted within the delay. But there is always a small possibility of an error using this method. Probably, a reasonable delay could make this possibility less likely than the possibility of the failure of the whole computer.

When the delay is finished, the cleanup job tests and sets the wait queue bit. The executive lock bit may now be reset. All time entries in the wait list now have their load bits "ANDED"

with zero. The wait list thread must now be reordered, since some entries with leading time bits of zero may have been inserted already. Finally, the cleaup job reschedules itself for absolute time 1000...0. It then unlocks the wait queue bit and calls END OF JOB.

Alternately, the solution following is completely safe. Let the wait queue bit be tested and set in WTIN before TNOW is added to R6 (See Figure 3.3 (2)). Then the cleanup job will also test and set the wait queue bit before setting the lead bit of TNOW to zero. No delay is necessary now, nor must the executive bit be used. All else remains as above. The cost of this method is that two or three instructions are added to the lock time of WTIN each time it is called. Thus, it must be decided whether this cost is worth the certainty it gives, or if it is better to allow a long enough delay every 25 days.

3.11 Conclusion

The executive routines presented here represent the main body of an executive system designed for the multiprocessor proposed in Chapter 2. Other routines may be added, or these routines may be altered, to achieve other objectives. For instance, it may be desirable to design a routine which will get more than one dynamic area in one call; or another routine that will release all dynamic areas assigned to a job. But these can be designed easily by building on the concepts presented in this chapter.

CHAPTER 4

DIGITAL SIMULATION OF THE MULTIPROCESSOR EXECUTIVE

4.1 Introduction

Once the executive proposed in Chapter 3 was developed, the next step was to design a digital simulation; first, to verify that the executive operated as desired, and second, to study the performance of a multiprocessor system incoporating the proposed executive. The simulation did indeed turn up some errors in the earlier executive routine designs before finally arriving at the design presented in Chapter 3. With the executive in final form, the simulation was then used to produce data on the performance of the proposed system under various conditions. The data and their interpretation are presented in Chapter 5.

This chapter presents the considerations that went into the development of the simulation. It was first necessary to program the executive routines to obtain a precise measure of the length of instructions for each of the executive routines. Second, a job set had to be described to represent the jobs running on the system. Finally, a FORTRAN simulation of the multiprocessor system was written based on the operation of the executive.

4.2 Model of the Executive Routines

Once the executive routines were designed, it was desirable to write actual programs that could implement these routines to give an indication of how large these programs would be. For the simulation, it was decided to use these programs to give an actual instruction count for various phases of the executive.

The executive routines were written in IBM 360 Basic Assembly Language (BAL). BAL was chosen because it is a machine language that largely satisfies the requirements of the proposed multiprocessor. BAL allows register-to-register and register-to-memory instructions; and it has a large set of desirable operations, including a test and set operation. The executive routines programmed in BAL are presented in the Appendix to this thesis.

The executive BAL programs were used to give an exact instruction count of various phases of each executive routine. For example, on the basis of the BAL programs, the executive routine JOBIN carries out two instructions before it sets the job queue lock, nine more instructions before the job information is stored in the queue, four more instructions until it resets the queue lock, and finally, one more instruction to return to the calling program.

The BAL programs were also used to give a measure of data bus use by the executive routines. It was assumed that any register-to-memory instruction included two bus calls; one to identify the memory location and one to send the desired value along the bus. Thus, JOBIN makes a total of fourteen bus calls each time it is called.

In the FORTRAN simulation presented in Section 4.4, instruction counts based upon the BAL programs are used to compute the running times between the various executive actions; such as the amount of time a job queue will be locked by a call to JOBIN.

4.3 Model of the Job Set

In order to simulate the multiprocessor it is necessary to define a set of jobs that will be running on the system. Only the

information pertinent to the simulation need be defined. Since the objective of the simulation is to study the performance of the executive, the information desired for any job is its instruction length, its bus use, its use of the executive routines, and what jobs it calls. The actual functions carried out by individual jobs are of no interest in this study.

The job set used in the simulation of the multiprocessor is based on analysis of the Lunar Landing Jobs carried out by Mallach (Ref. 5). The data used to describe this job set were based on the information presented in Appendix A of Mallach's thesis. A set of 43 jobs is described, with an explanation of the general purpose of each job. The description includes the jobs called by each job along with any time delay associated with the scheduling of a job. Finally, a table is given listing the number of basic instructions in each job, the bus calls by each job, and the interpretive time taken by each job. Interpretive time is that time the program spends doing interpretive instructions in the Apollo Guidance Computer (AGC). These instructions are basically calls to subroutines that carry out high powered instructions not included in the basic language, such as matrix operations. It was decided for the purpose of this simulation to divide interpretive time by four times the AGC basic instruction time to obtain the number of instructions that the interpretive time would represent. The factor of four was chosen rather arbitrarily to reflect that the savings on instructions that may result from a more powerful instruction set in the multiprocessor.

Each job is therefore represented by a set of numbers defining the number of instructions and the number of bus cycles it uses. Any calls to executive subroutines are made within the jobs, so, for the purpose of the simulation, each job is totally described by a table of numbers dividing each job into the number of steps equal to the number of executive subroutine calls it makes (including

END OF JOB). Each column of the table contains information defining the portion of a job from the last executive call until the next executive call. The rows of the table are: the number of basic instructions; the number of bus calls; a number specifing the next executive routine to be called; the number of the job being scheduled by this routine, if any; and the relative time delay if the next executive routine to be run by the job is WTIN. In many cases, the location of executive subroutine calls within a job and the distribution of bus calls among the steps was arbitrarily decided. In all jobs, the last step must terminate with a call to END OF JOB.

In Chapter 2 the desirability of dividing long jobs into shorter jobs was discussed. It was suggested that no segment of any job take longer than 10 milliseconds to execute. On the basis of the assumption of a basic instruction length of 25 microseconds in this simulation, no jobs would then be allowed to be more than 500 instructions long. Therefore, a job set was obtained from the set described above by dividing long jobs into segments of 500 instructions each. After initial control runs using the full-length job set, this short job set was used as input to most of the simulation studies.

4.4 FORTRAN Simulation of the Multiprocessor

The actual programming of the simulation was done in FORTRAN. The basic objectives of the simulation were to act as the operating system of the multiprocessor, to run the input job set on simulated processors, and to keep track of desired information about the system.

The main difficulty in designing the simulation was to simulate the activities of many processors on a single processor

computer. A state approach was taken to handle the activities of the processor. That is, a table is used by the simulation to describe the present state of each processor, including the next time that state is due to change. There is one column in the table for each processor, and the rows of the table include such information as when the processor will next change its state, what action will be carried out at the time of the next state change, what job is presently running on the processor, and what step of the job is the processor carrying out.

The control loop of the simulation searches the processor state table for the next time that any processor is due to change states, updates the system clock to that value, and then carries out the activity specified by that processor's table entries. After the activity is completed, the state of that processor is updated, including the time that it is next due to change state, and the cycle is repeated. Each activity must make certain to stop just before any change in system status that will affect other processors. The status change will then take place when that processor's next state change occurs. For example, if the beginning of JOBIN is the next activity to take place on a processor, it may proceed until the point where it has to test and lock the job queue flagbit. The time for the next state change is updated to the time when that operation will occur, and the test and set will be done the next time the processor becomes active.

The simulation must keep tables corresponding to the executive data bases and must simulate the executive routines when they are called. The FORTRAN simulation of the executive routines carries out operations on its tables corresponding to the operations the routines carry out on their data bases. The processor state time is then incremented based on the instruction and bus cycle count obtained fron the BAL program described in Section 4.2. The bus and instruction counts are multiplied by the single

instruction time and bus cycle time respectively, as input to the simulation, and their sum is the increment to the processor state time.

The major input to the simulation is the job set described in Section 4.3. Other direct inputs to the simulation are the single instruction time, the bus cycle time, and the delay taken by END OF JOB if it finds no jobs to schedule. The simulation tables and state table are initialized for the starting job assignments. Other system changes may be made by altering associated portions of the simulation routines. Such changes include varying the number of processors, changing the length of specific loop delays due to failure to access a data base, or alterations of the length of an executive delay.

Desired information can be collected for individual processors or for the whole system by inserting statements within the simulation to save and update specific values whenever the statements are reached. Data computed by the system is stored in tables which are updated as the simulation progresses. Examples of such data are the time each processor spends doing computation of a job, the time each processor spends running executive routines, the amount of time that all processors were busy, and the delay between the time jobs are requested and the time they are dispatched. System data can be output at specific time intervals, at the occurence of a specified event, or when the run is completed.

CHAPTER 5

RESULTS OF THE SIMULATION

5.1 Introduction

The simulation described in Chapter 4 was written for the purpose of measuring the performance of a multiprocessor system using the executive program described in Chapter 3. Simulations were first run using the full-length job set and then using the segmented job set described in Chapter 4. Finally, simulations were run with the basic instruction time lengthened, effectively loading the system, and the resulting system performance was studied.

To limit the scope of this study, certain system attributes were held constant throughout the simulations. First, the multiprocessor system studied consisted of five processors. Second, the three non-timed job queues were simplified to one queue of 40 entries, and the wait queue allowed 100 entries. Third, all artificial delay times in the executive due to locked queues were made zero. Thus, test and set instructions were repeated without delay until the queue in question could be accessed. This decision neglects the possibility that excessive bus use may result from the constant reading and writing of the flagbit. Finally, when the instruction time was changed, the bus cycle time was changed proportionately, eliminating considerations due to the relative importance of instruction time and bus time.

Certain concepts used throughout the simulation studies will now be defined. Job load is the fraction of the total processor running time that the processors of the system were actually doing

job related computations. A system is fully loaded when all processors have work to do; that is, when the job queues always have at least five jobs due. Thus, even when the system is fully loaded the job load will not be 100%, because some time must be spent on overhead due to the executive routines. Another system overhead is the processor lockout time. This is the time a processor spends waiting to access a locked data base. This time is not included in executive overhead because it is more closely related to the number of processors in the system. Finally, when a processor is not busy, it will cycle in END OF JOB until a job is found to be run on that processor. This time spent in END OF JOB without scheduling a job will be considered null time, rather than executive overhead time.

5.2 Simulation of the Full-Length Job Set

Simulations with the full-length job set were first run to test that the executive routines functioned as desired. Then a simulation was run assuming a 25 microsecond instruction time to see what the system performance with the long job set would be like. This performance is summarized in Table 5.1.

It can be seen from Table 5.1 that the system is lightly loaded when run with the full-length job set and with a 25 microsecond instruction time. The job load is 10.35%, and over half the time no processors are busy. Thus, the multiprocessor can easily handle the long job set. But, even though all processors are busy only .04% of the time, the longest single time span that all processors were busy was 4.07 milliseconds. This number suggests that at one time at least during this simulation, a job waiting to run might have to wait at least 4.07 milliseconds to be dispatched. This time falls within the 10 millisecond response time desired. But if the system becomes more loaded, the worst case response time could get quite large.

Basic Instruction Time:	25 microseconds
Job Load:	10.35%
Executive Overhead:	3.11%
Executive/Job Load Ratio:	30.1%

Processors Busy	Percentage of Time
0	54.55
1	31.93
2	10.50
3	2.65
4	.33
5	.04

Longest Time All Processors Were Busy: 4.07 milliseconds

Table 5.1 Simulation of Full-Length Job Set

5.3 Simulation of the Short Job Set

The remainder of the simulations of this chapter were made using the short job set. The short job set was devised from the long job set by breaking up long jobs into 500-instruction segments. The purpose of this segmentation was to insure good system response time by preventing a few long jobs from monopolizing the system. Therefore, included in the data output of the simulation is data on the response time; that is, the time it takes to dispatch a job once it is requested. The performance of the short job set including data on response time, when run for a multiprocessor with 25 microsecond instruction time, is summarized in Table 5.2.

The job load of this simulation is much less than in the full-length job set simulation. This is due to the reduction of the interpretive instruction impact, as discussed in Chapter 4. But the executive time has actually increased since more scheduling and dispatching must be done to accommodate many short jobs. The total number of jobs in the short job set is almost three times the number of jobs in the full length job set. Consequently, the ratio of the executive time to the job computation time is 55.7%. Thus, even if the system could be fully loaded without delays and lockouts, the job load cannot exceed 64% if the executive/job load ratio remains unchanged or becomes smaller. The simulation results, however, show that as the job load increases this ratio will increase, and the job load never in fact exceeds 51%.

The average delay in dispatching a job is 1.115 milliseconds. 99% of all jobs were dispatched within 3.1 milliseconds of the time they were scheduled. But the longest delay in scheduling a job was 7.87 milliseconds, which is nearly three times the maximum single time all processors were busy. Although this number is still within the 10 millisecond bound discussed earlier, the question

Basic Instruction Time.	25 microseconds
Job Load:	6.51%
Executive Overhead.	3.63%
Executive/Job Load Ratio:	55.7%

Processors Busy	Percentage of Time
0	66.53
1	24.50
2	7.20
3	1.42
4	.23
5	.02

Longest Time All Processors Busy: 2.822 milliseconds

Mean Average Job Delay: 1.115 milliseconds

Delay of 99%ile: 3.100 milliseconds

Maximum Job Delay: 7.870 milliseconds

Table 5.2 Simulation of Short Job Set

is: why should this delay be so much longer than the longest time that all processors are busy? This result certainly suggests that other factors are more important than just the fact that processors are free. Another question is how the job load affects the response time; especially the maximum delay. These are the subjects of the next section.

A histogram of job scheduling delays is presented in Figure 5.1. A point was plotted every half millisecond to represent the number of job delays which fall within the previous half millisecond. The points are connected by lines for clarity.

5.4 Loading the Multiprocessor

The question of what effect increasing the load of the multiprocessor would have on system response was posed in the last section. In order to examine this question, a way must be found to increase the job load. This was done by simply increasing the instruction and bus times. For instance, doubling the instruction and bus times should have nearly the same effect as having twice as many jobs on the system with the original instruction time. The truth of this statement is not investigated in this thesis since the real objective is to make the system busier. Slowing down the multiprocessor certainly accomplishes this objective. But doubling instruction time obviously means that it will take twice as long to run END OF JOB, and delay times cannot be compared directly. Thus, delay results will be normalized by expressing delays (and other time measured results) in terms of basic instruction times. For example, if a delay is 10 milliseconds and the basic instruction time is 25 microseconds, the delay is a 400-instruction delay.

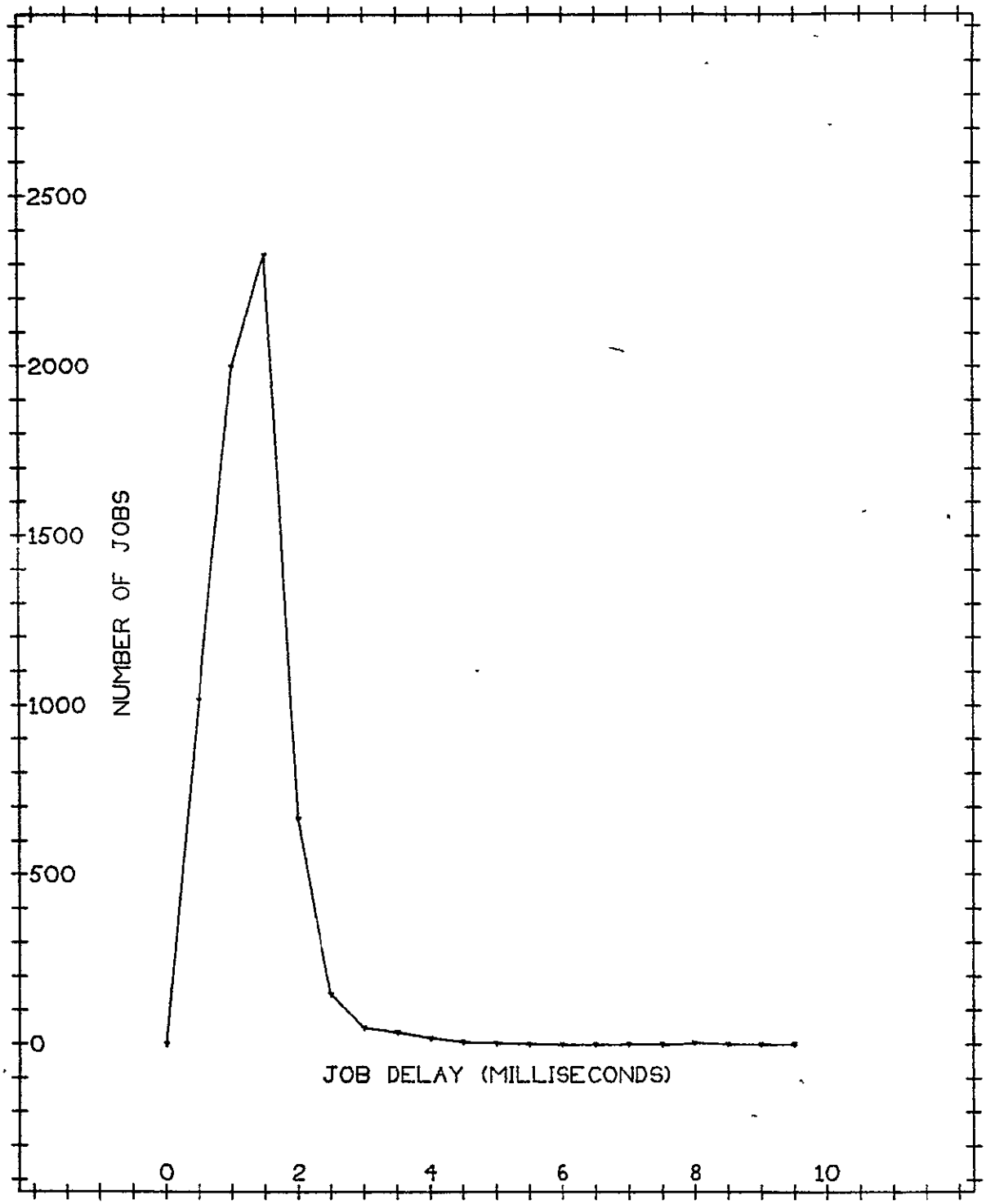

Figure 5.1 Histogram of Short Job Set

A simulation was then run with the original instruction time doubled. The job load went up to 12.84% and the executive/job load ratio remained nearly constant in comparison to the first run as expected. The longest time all processors were busy went up 66% to 200 instructions (using the normalized measure of time), which is reasonable, considering that the load has doubled. But now the maximum job dispatch delay was 520 instructions, a large increase from the maximum of 310 instructions of the first run. This delay also exceeds the 500 instruction maximum length of job segments, which is exactly what was to be avoided.

Consequently, it was decided to trace the history of this job dispatch delay to discover its origins. The sequence of events involved in the delay are presented in Table 5.3. From this table, it is seen that there are two major factors of the delay. The first factor is that a number of timed jobs are due before the job in question, which is a non-timed job, can be scheduled. While these jobs are all waiting to be scheduled, other timed jobs become due. The second factor is that while WTIN is running, it locks out END OF JOB from scheduling wait jobs. Since a non-timed job will not be dispatched until all timed jobs are dispatched, END OF JOB cycles until it can schedule a timed job.

The original decision to schedule timed jobs first failed to make provision for the present delay problems. One obvious suggestion is to allow END OF JOB to schedule a non-timed job if the wait queue is locked, although this solution does not solve the delays of timed jobs, some of which were also quite large.

The true problem stems from the fact that WTIN and END OF JOB must necessarily lock each other out from the wait queues. WTIN times can be substantial (124 instructions in one case of the present example), and if, as in this example, a number of processors gain accesses to the wait queue for WTIN before others

Relative Time (Instructions)	Events
0	Job 93 entered in normal job queue;
	Wait queue in use by a processor in WTIN;
	2 processors are free;
	Jobs 2 and 7 due in wait queue.
54	WTIN releases queues;
	Job 2 scheduled on a free processor.
81	Job 7 scheduled on last free processor;
	Jobs 1 and 26 now are due in wait queue.
191	A processor is freed;
	But a processor is in WTIN using wait queue;
	Thus, END OF JOB cannot run.
226	Processor in WTIN releases wait queue;
	Another WTIN gains access to queue.
350	Processor in WTIN releases wait queue;
	Another WTIN gains access to queue.
429	Now 4 processors are free;
	Job 1 is dispatched;
	Jobs 26 and 42 are due in wait queue.
457	Job 26 is dispatched.
483	Job 42 is dispatched.
520	Job 93 is dispatched.

FIGURE 5.3 Trace of a 520 Instruction Job Dispatch Delay

wanting to run END OF JOB, major delays occur. Thus, it can be seen that WTIN represents a bottleneck in the proposed executive scheme. This is one area where further investigation is desirable. One suggestion is to find some way to allow END OF JOB and WTIN to run in parallel. Such a solution is not immediately obvious; the wait queue must be ordered at some point, and when it is, other jobs can not be allowed to access the data base. Any possible solution would certainly add complexity to the executive. Another suggestion is to implement the wait queue in hardware, along with a device which automatically orders the queue. The methods of accomplishing this are beyond the bounds of this thesis, but such a device may do much to improve the proposed multiprocessor. WTIN time grows as the number of queue entries increases, causing not only the lockout discussed here; but also excessive executive time, which also lowers system performance.

The simulation was run with increasing loads until the multiprocessor's job queues began to overflow. Information derived from these runs is summarized in Table 5.4. Figure 5.2 shows the histrograms of the job delays associated with the job loads. It can be seen from the table that the highest job load attained was 39.93%. A further increase in system load caused the job queues to overflow. Actually, the simulation which resulted in the 39.93% job load had higher job loads during the final portion of the run. During the time the processors were constantly busy and no additions were made to null time, the job load reached 51%. This value may be considered an upper bound for the job load with the present system configuration.

The maximum job dispatch delay in the 39.93% load simulation jumped to 3967 instructions, which is almost eight times the segment size. Even the 99th percentile delay - that delay which is larger than 99% of all job dispatch delays - has increased to nearly 4000 instructions. Moreover, nearly 20% of all job dispatch

Job Load (per cent):	6.51	12.84	18.77	24.59	34.66	39.93
Percentage of Time N Processors Busy						
N=0	66.53	42.51	27.93	16.73	10.41	8.35
N=1	24.50	34.27	31.35	26.43	12.20	10.03
N=2	7.20	16.81	25.38	31.26	21.28	14.40
N=3	1.42	5.20	12.00	18.74	30.87	21.12
N=4	.23	1.06	2.94	6.06	20.23	25.75
N=5	.02	.15	.40	.79	5.02	20.36
Longest Time All Processors Busy (Instructions):	113	200	190	171	227	336
Mean Average Delay (Instructions):	45	44	49	54	75	389
99th Percentile Delay (Instructions):	124	146	200	213	371	2600
Maximum Job Delay (Instructions):	315	520	616	447	836	3967

Table 5.4 Results from Simulations of Increasing Loads

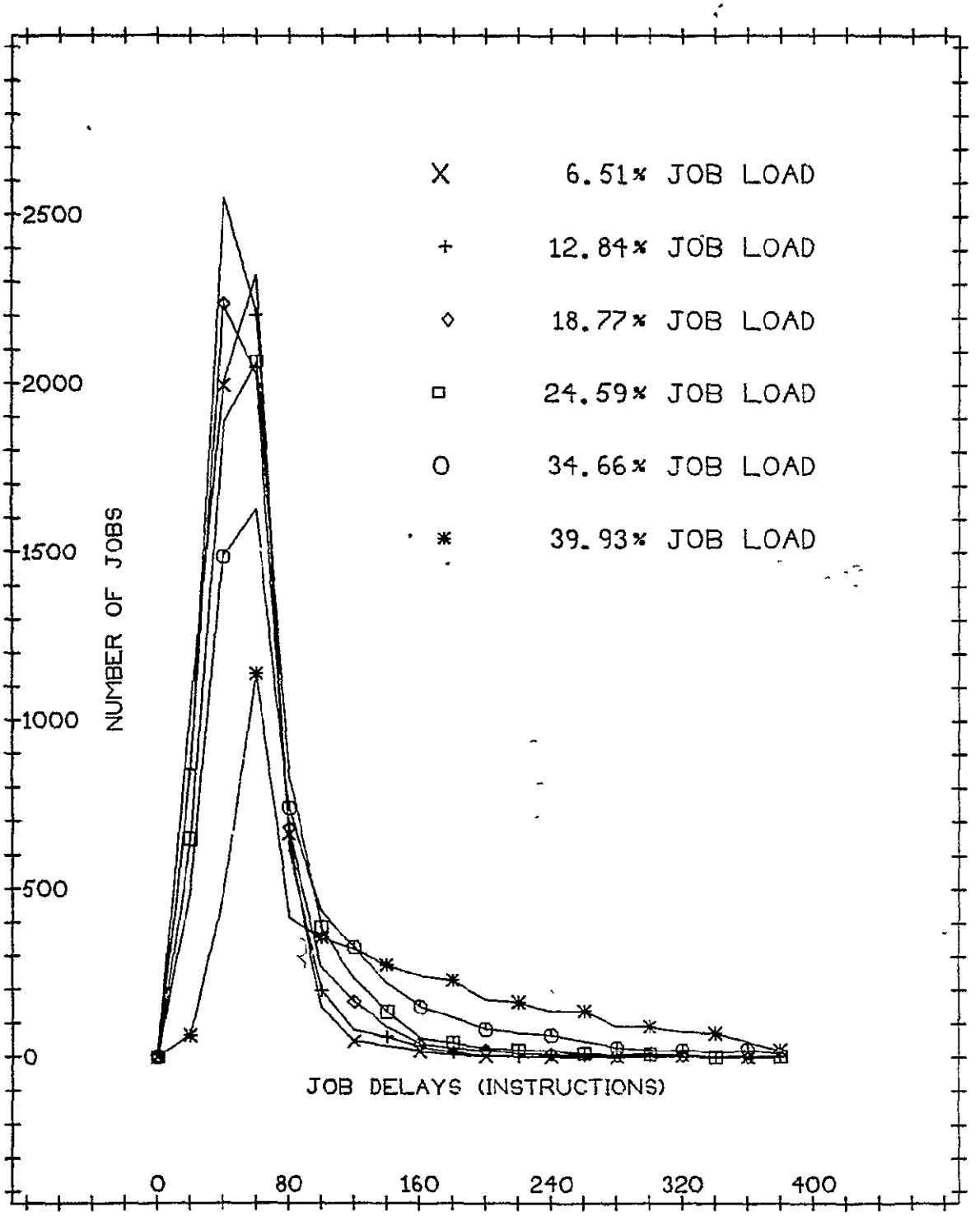

Figure 5.2 Histograms of Short Job Set for Various Loads

delays were larger than 500 instructions, so that substantial delays are quite common.

These simulations present an indication of the limits of the proposed multiprocessor system. They seem to demonstrate that the multiprocessor is limited in its ability to acheive greater throughput than a single processor computer. Of course, there are no best answers as to how the system should be used. Tradeoffs will be made on the basis of system requirements. Also, other steps may be taken to increase the system's capacity; such as increasing instruction speed or adding more processors.

5.5 Effect of Reducing WTIN Time

In the last section WTIN was found to be the major source of the large job delays. Therefore, it was decided to see what effect reducing the length of time of WTIN would have on job delays. The reason WTIN may run for such relatively long times is that it must insert a job in the correct position of the wait queue thread. Thus, if job delay times are random, WTIN will look at half the entries in the thread, on the average, before inserting the new entry.

The graph in Figure 5.3 presents some results of reducing the length of time of each cycle of the loop in WTIN that searches through the thread of wait queue entries. These simulations were run with the system load that resulted in a 39.93% job load when run with full WTIN time. It can be seen from the graph that the job load increases as WTIN time is decreased. This results from the fact that jobs are delayed less and, therefore, more jobs are run in the same period of time.

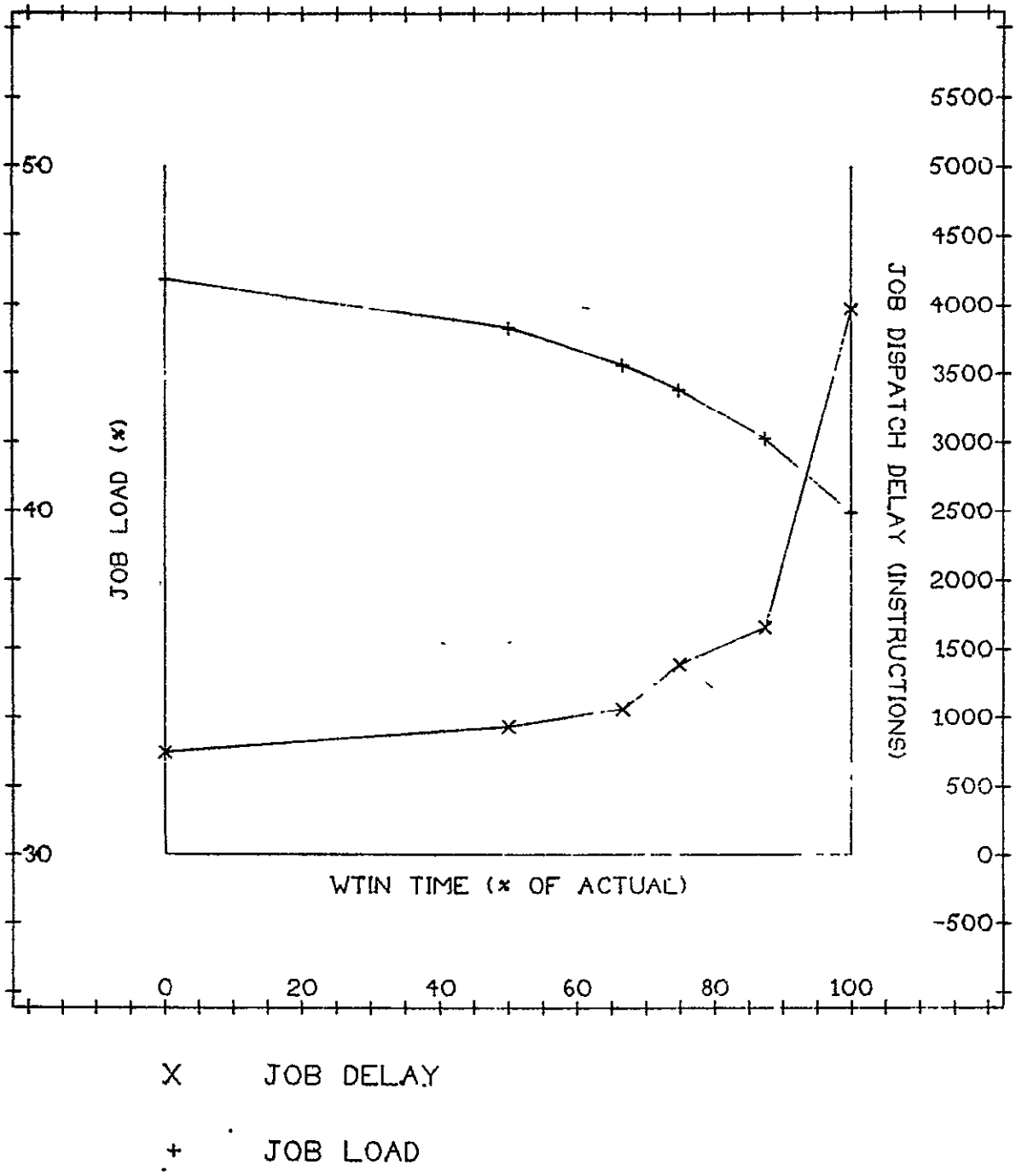

Figure 5.3 Effects of Reducing WTIN Time

The most striking result is that the maximum job delay decreases from 3967 instructions to 1648 instructions with only a 12 1/2% decrease in WTIN time. Even if WTIN took no time, the largest job delay would drop only to 741 instructions. This implies that the sharp rise in job delays at a certain point is due almost entirely to WTIN. WTIN seems to be able to handle its queue reasonably well until the system load becomes high enough that WTIN regularly delays the rest of the system. Because the system is delayed, there will be delays in dispatching remaining jobs from the wait queue, thus making the wait queue more full and slowing down WTIN even more. This vicious circle effect seems to account for the rapid rise in job delays.

With reduced WTIN time the system is running more efficiently. Therefore, the system load can again be increased to see what the limit of the job load will now be. This was done using a WTIN time 50% of the original. The simulations reached a job load of 48.4% for a run when the instruction execution time was increased 25%. But, during the time the system was fully loaded, it was running a 62% job load. This compares with the 51% job load with full WTIN time discussed in the last section. Consequently, one way to increase system capacity is to find a way to speed up WTIN.

5.6 Summation and Suggestions for Further Research

In this thesis an organization for an aerospace multiprocessor system was described, and an executive program for this multiprocessor was developed. The executive program consists of several routines which carry out specific executive functions. These routines were designed to be simple and as independent as possible for the sake of system efficiency.

A simulation was written for the proposed multiprocessor, and results of many simulations were presented in this chapter. Although no broad conclusions can be drawn from these results, some specific observations can be made for the specific configuration studied. First, a five processor system with a 25 microsecond instruction speed is capable of handling efficiently the job set based on the Lunar Landing program, either in long or segmented form. Second, as system load increases, job dispatch delays become larger. On the system studied, job delays became quite high at job loads higher than 35% of system time. The upper limit for the job load appeared to be 51%. The wait queue and its associated routine, WTIN, seemed to be the major cause of large delays. When WTIN running time was cut in half, the system response improved for equivalent job loads, and the upper limit for the job load reached 62%.

The results of this thesis indicate that a breakdown of system performance is inherent in the proposed multiprocessor as loads approach a certain limit. In this study, the limits of the job load for good performance appeared to be between 1/3 and 1/2 of system computation time. A similar limitation appears in Mallach's study of data bus allocation (Ref. 5). Whether such breakdowns are basic to multiprocessing structures is a subject which should be investigated further.

Some changes to the executive presented may show improvements in system performance. One such change is to allow END OF JOB to schedule a non-timed job if the wait queue is locked. Another suggestion is to allow segmented jobs to continue running uninterrupted if there are no jobs of higher priority waiting to be scheduled. Finally, a method of allowing WTIN and END OF JOB to run concurrently may improve job response. The design and effects of these suggestions are areas for further research.

These simulations studied one specific system configuration. Further studies may be made considering different numbers of processors, different priority structures of queues, and different job sets. Also, it may prove desirable to see how the system functions when other aspects of the system, such as bus use or I/O calls, are included in the simulation.

Finally, it was suggested in this chapter that an implementation in hardware of the wait queue and a device for ordering the queue would improve system performance. Hardware implementation of other aspects of the executive may also prove beneficial. The improvements, methods, and costs of hardware implementation represent other areas which should be studied.

APPENDIX

BAL PROGRAMS FOR THE EXECUTIVE ROUTINES

This appendix presents IBM Basic Assembly Language implementations of the executive routines developed in Chapter 3.

1. JOBIN

```
JOBIN    SLL    3,16(0)
         AR     2,3
ENT2     TS     0(,5)
         BC     4,ENT2
         LH     6,2(,4)
         LA     3,'QSIZE'(,4)
         CR     6,3
         BC     12,OK
         LA     6,4(,4)
OK       L      0,0(,6)
         LTR    0,0
         BC     8,FULARM
         ST     2,0(,6)
         LA     6,4(,6)
         STH    6,2(,4)
         NI     0(5),0
         BCR    15,1
```

2. WTIN

```
RELWTIN     A       6,TNOW
ABWTIN      SLL     3,16(0)
            AR      2,3
LOK         TS      WTFLG
            BC      4,LOK
            L       3,WAITFREE
            LTR     3,3
            BC      8,WAITALRM
            L       4,0(,3)
            ST      4,WAITFREE
            LA      4,WAITQUE
            L       5,0(,4)
LOOP        LT      5,5
            BC      8,BRANCH
            C       6,4(,5)
            BC      12,BRANCH
            LR      4,5
            L       5,0(,4)
            BC      15,LOOP
BRANCH      ST      5,0(,3)
            ST      3,0(,4)
            ST      6,4(,3)
            ST      2,8(,3)
            NI      WAITFLG,0
            BCR     15,1
```

3. GETDYN

```
GETDYN    TS      DYNFLG
          BC      4,GETDYN
          LH      3,DYNQUE
          L       2,0(,3)
          LTR     2,2
          BC      8,DYNLARM
          STH     2,DYNQUE
          NI      DYNFLG,0
          BCR     15,1
```

4. FREDYN

```
FREDYN    SR      0,0
          ST      0,0(,7)
          TS      DYNBOT
          BC      4,FREEDYN
          LH      2,DYNQUE+2
          ST      7,0(,2)
          STH     7,DYNQUE+2
          NI      DYNBOT,0
          BCR     15,1
```

5. ENDOFJOB

```
ENDOFJOB    TS      RESFLG
            BC      8,RESTART
BAK         TS      ENDFLG
            BC      4,BAK
            L       1,WAITQUE
            LT      1,1
            BC      8,SKIPWT
            L       2,4(,1)
            C       2,TNOW
            BC      10,SKIPWT
WAIT        TS      WTFLG
            BC      8,WAIT
            L       1,WAITQUE
            L       2,0(,1)
            ST      2,WAITQUE
            NI      ENDFLG,0
            LH      2,10(,1)
            LH      7,8(,1)
            L       3,WAITFREE
            ST      3,0(,1)
            ST      1,WAITFREE
            NI      WTFLG,0
            BCR     15,2
SKIPWT      L       1,PRIOHD
            L       7,0(,1)
            LTR     7,7
            BC      6,D1
            L       1,NORMHD
            A       7,0(,1)
            BC      6,02
            L       1,LOWHD
```

```
                A       7,0(,1)
                BC      6,03
DEL             NI      ENDFLG,0
                LA      7,N(,0)
                BCT     7,DEL
                BC      15,ENDOFJOB
D1              LA      4,4(,1)
                LA      6,QSIZE+PRIOHD
                CR      4,6
                BC      4,E1
                LA      6,PRIOHD+4
E1              ST      4,PRIOHD
                NI      ENDFLG,0
ORF             LH      2,2(,1)
                SR      3,3
                ST      3,0(,1)
                SRL     7,16(0)
                BCR     15,2
D2              LA      4,4(,1)
                LA      6,QSIZE+NORMHD
                CR      4,6
                BC      4,E2
                LA      6,NORMHD+4
E2              ST      4,NORMHD
                NI      ENDFLG,0
                BC      15,ORF
D3              LA      4,4(,1)
                LA      6,QSIZE+LOWHD
                CR      4,6
                BC      4,E3
                LA      6,LOWHD+4
E3              ST      4,LOWHD
                NI      ENDFLG,0
                BC      15,ORF
```

REFERENCES

1. Lampson, Butler W., "A Scheduling Philosophy for Multiprocessing Systems," Communications of the ACM, Vol. 11, No. 5, pp. 347-360, May, 1968.
2. Pariser, Jack J., "Multiprocessing with Floating Executive Control," 1965 IEEE International Convention Record, pp. 266-275.
3. MIT C.S. Draper Laboratory, STS Data Management System Design, Report E-2529, Cambridge, Mass. June, 1970.
4. MIT C.S. Draper Laboratory, STS Software Development, Report E-2519, Cambridge, Mass., July, 1970.
5. Mallach, Efrem G., Analysis of a Multiprocessor Guidance Computer, Ph.D. Thesis, M.I.T., Cambridge, Mass., June, 1969.
6. Madnick, Stuart E., "Multiprocessor Software Lockout," Proceedings - 1968 ACM National Conference, pp. 19-24.