FAA-72-24

REPORT NO. FAA-RD-72-101

# ADVANCED COMPUTER ARCHITECTURE FOR LARGE-SCALE REAL-TIME APPLICATIONS

Gary Y. Wang


APRIL 1973

FINAL REPORT



Prepared for:
DEPARTMENT OF TRANSPORTATION
FEDERAL AVIATION ADMINISTRATION
Systems Research and Development Service
Washington, DC 20591

| 1. Report No. FAA-RD-72-101 | 2. Government Accession No. | 3. Recipient's Catalog No. |
|---|---|---|
| 4. Title and Subtitle<br>ADVANCED COMPUTER ARCHITECTURE FOR LARGE-SCALE REAL-TIME APPLICATIONS | | 5. Report Date<br>April 1973<br>6. Performing Organization Code |
| 7. Author(s)<br>Gary Y. Wang | | 8. Performing Organization Report No.<br>DOT-TSC-FAA-72- 21 |
| 9. Performing Organization Name and Address<br>Department of Transportation<br>Transportation Systems Center<br>Kendall Square, Cambridge, MA. 02142 | | 10. Work Unit No.<br>R-2101<br>11. Contract or Grant No.<br>FA203<br>13. Type of Report and Period Covered<br>Final Report |
| 12. Sponsoring Agency Name and Address<br>Department of Transportation<br>Federal Aviation Administration<br>Systems Research and Development Service<br>Washington, D.C. 20591 | | 14. Sponsoring Agency Code |
| 15. Supplementary Notes | | |

16. Abstract

In this study the air traffic control automation is identified as a crucial problem which provides a complex, real-time computer application environment. A novel computer architecture in the form of a pipeline associative processor is conceived to achieve greater performance improvement over the present air traffic control system by parallel processing. This new processor is structured into a multiprocessor configuration for reliability enhancement. Problems associated with multiprocessors are identified with special emphasis on execution time anomalies and memory conflict. A direct graph model is used for analysis from which simple heuristics are established for memory allocation and dynamic task scheduling to achieve optimal performance with minimal system overhead. These schemes are simulated and the results obtained follow closely the predicted system behavior

| 17. Key Words | 18. Distribution Statement |
|---|---|
| Parallel Processor, Associative Processor, Multiprocessing, Memory Allocation, Dynamic Task Scheduling, Graph Modelling, Air Traffic, Real-Time Systems | DOCUMENT IS AVAILABLE TO THE PUBLIC THROUGH THE NATIONAL TECHNICAL INFORMATION SERVICE, SPRINGFIELD, VIRGINIA 22151. |

| 19. Security Classif. (of this report) | 20. Security Classif. (of this page) | 21. No. of Pages | 22. Price |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | 186 | |

# PREFACE

This study has identified a large-scale real-time processing environment, proposed a computer architecture for the application, acknowledged a few key problem areas, and suggested an heuristic approach for their solutions.

The U.S. Continental air traffic control automation system was chosen to provide a real-time application environment. One of the major contributions of this study is the formulation of a novel computer architecture to achieve greater performance improvement over the present ATC automation system by parallel processing. Problems associated with multiprocessors are reviewed with particular emphasis on execution time anomalies and memory conflicts. A directed graph model is used from which simple heuristic rules are established for memory allocation and dynamic task scheduling so that near optimal performance can be achieved with minimal system overhead. The memory allocation and heuristic scheduling schemes are simulated. The results analyzed closely follow the predicted system behavior. In view of the complex nature and wide scope of the subject matter, a number of interesting aspects were intentionally left unanswered.

There are many areas of research still open with respect to the proposed associative pipeline multiprocessor structure, to say nothing of the many possible alternative parallel processor structures. To most of the questions, there are no pat "right" or "wrong" answers. There are, rather, trade-offs to be investigated and techniques to be developed. The analyses and methods proposed in this study are believed to be of direct usefulness in the design of the next generation of computers.

The author would like to acknowledge his appreciation to his superior, Dr. David Van Meter, for his encouragement, to his FAA/OSEM, SRDS colleagues, Mr. Leland Page and Mr. Donald Scheffler for their continued support. Sincere thanks are due to many colleagues at the Transportation Systems Center, to Mr. Jim Steinberg, and Mr. Jan Carlson for their assistance in running the H-516 computer, to Mrs.

Judy Gertler for her help in using the GASP simulation language, and to Mr. David Clapp, Mr. Charles Dancy and Mrs. Vivian Hobbs for many stimulating discussions. Great appreciations are due to Miss Kathleen M. McGann for struggling through the manuscript and to Mrs. Susan Bradbury for a fine editing job. The Technical Publications Group at TSC was extremely helpful at every step from art work to final printing.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

## LIST OF ILLUSTRATIONS

## LIST OF ILLUSTRATIONS (CONT.)

## LIST OF ILLUSTRATIONS (CONT.)

# LIST OF TABLES

# 1.0 INTRODUCTION

In general, the trend of computer development has always followed, in addition to the cost factor, three important objectives: performance, flexibility, and reliability. Of the three, emphasis has been placed on performance first. This is evidenced by various advancements in computer technology, such as shortened memory cycle time and increased logic circuit speed, and innovation of simultaneous CPU and I/O operations, etc. Gradually, attention has been shifting to flexibility. This demand has lead to the advent of the "family" concept, whereby the system tailors a user's needs at present and still provides easy expansion for him to grow into larger machines in the future. Of course, reliability has always been an important consideration in designing a computer, or for that matter, any electronic system. However, in the past, it mainly involved the hardware component reliability, e.g., transistors, connectors, etc. As long as hardware is carefully selected and tested, and there is an adequate diagnostic method for easy off-line maintenance, the system is considered satisfactory. When computers are used more in real-time application areas where users demand almost instantaneous responses, the reliability requirement is given a new meaning. Some of the critical real-time systems such as military command and control [1,2] space exploration [3,4,5], and air traffic control systems [6], etc., cannot tolerate any computer system down-time at all. Therefore, it becomes highly desirable and a great challenge to computer system designers to formulate a system which satisfies, to a large degree, all three requirements.

## 1.1 A REAL-TIME ENVIRONMENT

Real-time [7] is a term that is defined differently by different people. A real-time computer system may be defined as one which controls an environment by receiving data, processing it, and taking action or returning results quickly enough to affect the functioning of the environment at that time.

Implied in the definition is a requirement of the system "response time". Response time is the time which the system takes to react to a given input. In fact, any environment in which computers are applied can be classified into three distinctive, time-related categories. Typical commercial applications, such as payroll, inventory control, accounts receivable, etc., do not have stringent time requirements, but they do have to be performed at a certain time of, say, a month or a week. It is called a time-dependent environment. However, in a time-sharing system or an airline ticket reservation system, a user or a customer enters his request and expects a response in a reasonable time, perhaps a few seconds. This is called a time-sensitive environment since small variations in service are tolerable. The third category is typified by aerospace applications whereby the external work presents the computer with a set of rigid time constraints: certain times before which the system will not be able to make use of the processed data. When the external world is characterized this way, we call it a time-critical environment. Whether time-sensitive or time-critical, the system receives data, processes it, takes action and returns results quickly enough to affect the functioning of the environment as though the system responds to demands in real-time. Hence, systems of this sort are called "real-time".

In practice, however, not every function in a real-time system needs to be carried out quickly. Some functions require immediate attention more than others, and the distribution of this type of requirement differs from system to system. This real-time environment presents complex problems and yet offers great challenge. Hence, this is precisely the reason that we are interested in conducting an investigation in establishing novel computer architecture applicable in a real-time environment.

## 1.2 STATEMENT OF THE PROBLEM

The trend in the use of computers in the time-critical real-time applications is towards larger, high-speed, and more

integrated systems to handle the ever increasing computation load as well as automation functions. These large-scale systems typically cover a wide spectrum of activities ranging from bookkeeping data processing to highly sophisticated computations. Historically, these systems were realized by high performance general-purpose machines rather than special-purpose hardware to meet the flexibility requirements. Technologies have advanced at such an incredible pace that circuit and memory speeds have reached a point where further improvements will not make a great deal of impact on the overall system any longer. Yet, problems demanding great computation capability, particularly in military systems, are increasing. These increased application requirements have led to keen interests in seeking large-scale performance improvement through novel concepts in computer architecture and system organization.

Accepting the challenge, this study defines a sample large-scale real-time environment (air traffic control system), and attempts to formulate a novel computer architecture which shows great potential in satisfying all of the stringent requirements imposed by the real-time environment.

## 1.3 SCOPE OF THE STUDY

The first section provides an introduction, a broad definition of an environment in which we shall confine our studies, and a description of the problem and motives behind our investigation. The second section gives a brief description of the principles and techniques used in the air traffic control system which serves as our sample real-time system. It encompasses the review of the present air traffic control system, the projection of future demand, and three of the basic air traffic control automation functions. These basic techniques are again analyzed in Section 3 to focus on some of their processing characteristics which are adaptive to parallel processing. After a brief survey of existing and/or developing parallel processors, a novel computer architecture is formulated and recommended for such parallel processing activities with high

efficiency. To achieve the flexibility, modularity, and system availability, we integrate this computer architecture into a multiprocessor organization. Section 4 illustrates that such an effort presents no special system problem. Some simple and unique designs are introduced for the realization of some critical functions such as crossbar switching network and interrupt management subsystems. There is little doubt that a multiprocessor structure with a number of independent operational resources working concurrently could outperform a sequential machine with similar characteristics; however, the controlling of such a system is still not a very well understood problem. Section 5 performs a general survey on this subject describing briefly previous work done in multiprocessor scheduling with special attention on timing anomalies and memory conflict problems. Perhaps it is a safe conjecture to say that a finite, economic, formalized scheduling strategy could not be derived from a limited set of procedures to take care of all situations produced in a variety of real-time systems. Section 6 proposes an heuristic approach to both the memory allocation to reduce potential memory conflict and the dynamic scheduling to minimize program execution time. A computer simulation program was written to evaluate the feasibility of the heuristic approach. The analysis and evaluation are reported in Section 7. Lastly, Section 8 concludes this study by giving an assessment of technology for the implementation of the proposed computer architecture. Also included are areas which deserve further research. In addition, two appendices are attached. Appendix A gives a brief description of the GASP simulation language which offers simple and convenient facilities for bookkeeping and statistics collection. Appendix B documents the structure and the logic flow of the simulator itself which provides the intermediate level of flow charts down to the detailed program listings.

# 2.0 AIR TRAFFIC CONTROL APPLICATIONS

In recent years, the demand for air transportation has out-grown the provision of ground facilities to handle it. Forecasts prepared by the Federal Aviation Administration (FAA)[8], the Civil Aeronautics Board (CAB)[9], and the industry groups [10] indicate continued growth of demand. This is evidenced by the fact that departure jet liners frequently wait for take-off clearances in long queues on taxiways while others circle above in holding patterns. In the spring of 1970, the Professional Air Traffic Controller Organization stages a "sick-out" to demonstrate the need for great improvement of air traffic control (ATC) capabilities. It is evident that the improvement can only be obtained through system automation by which some of the basic but routine ATC functions could be carried out by computer systems automatically.

## 2.1 OVERVIEW OF AIR TRAFFIC CONTROL

The present air traffic control (ATC) system, although it uses some computers for target tracking and flight plan processing, is primarily a manual system with regard to the control and separation of air traffic. The major elements of the system include ATC facilities, designated airspace volumes, rules and procedures, airport and weather facilities, navigation and landing facilities, communication facilities, and the trained personnel who operate and maintain the system. The FAA operates three types of ATC facilities; airport traffic control towers, air route traffic control centers (ARTCC) and flight service stations (FSS). These, together with airport radars, beacons, and communication and navigation aids, comprise the major physical components of the National Airspace System (NAS).

The ATC philosopy is based on an airspace division concept which was established to define the services provided by the ATC system in various geographical regions, altitudes and stages of flight. Airspace is vertically divided into three major categories: positive control, controlled or mixed, and uncontrolled.

Positive controlled airspace currently exists above 18,000 feet in the Northeastern portion of the United States and above 24,000 feet in the remainder of the country. In this airspace, instrument flight rules (IFR) are in force at all times for all occupants. The controller accepts responsibility for separating aircraft. Controlled or mixed airspace, in general, starts at some altitude above the ground and extends upward to positive controlled regions. In terminal area control zones, it extends to the ground. This airspace is shared by controlled and uncontrolled aircraft obeying IFR and VFR (Visual Flight Rules) procedures. The controllers still are responsible for separation between IFR aircraft, but separation among VFR aircraft and between VFR and IFR aircraft is achieved by depending on the pilots' "see-and-avoid" capability. On a time-available basis, controllers provide radar advisories to identified aircraft of the presence of another aircraft. Uncontrolled airspace underlies mixed airspace in which separation is provided by procedures and "see-and-avoid" capability for all aircraft.

The monitoring and control of all aircraft is accomplished by having the airspace divided horizontally into many areas. As an aircraft flies across area boundaries, the status of the aircraft is "handed-off" from one controller to the adjacent controller consistent with their responsible areas. The controllers use radar and beacon transponders to detect aircraft, and use voice communications at VHF frequencies to relay control instructions. The control scheme is best illustrated in Figure 2-1.

## 2.2 ATC AUTOMATION PROGRAMS

The evolution process of ATC automation program properly begins with the release of the Project Beacon [11] report in 1962. At the recommendation of the report, FAA has embarked on two major ATC automation programs: enroute and terminal air traffic control.

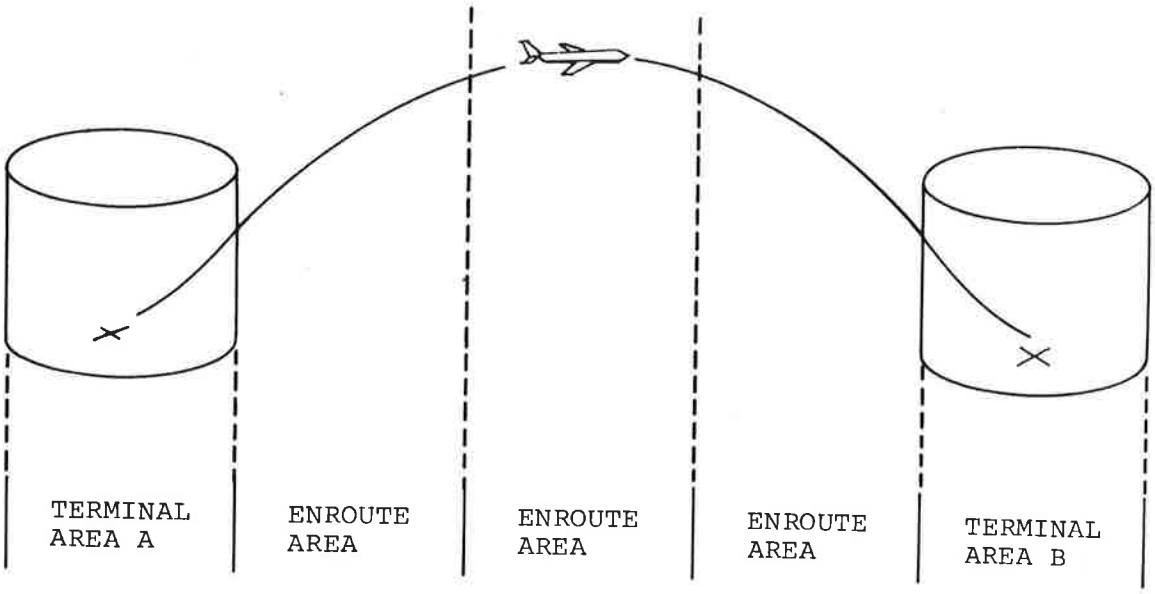



Figure 2-1 Simplified U.S. Air Space Structure

### 2.2.1 Air Routine Traffic Control Center (ARTCC)

A prime objective of the NAS air traffic control system is to increase system safety and efficiency through the application of automation techniques. Initially, the new level of automation is projected for Air Route Traffic Control Centers (ARTCC). There are twenty-seven ARTCC's in the U.S. controlling IFR flights enroute between points of take-off and landing within controlled airspace. The automation system handles many of the routine functions that burdened the air traffic controllers. The overall system configuration is depicted in Figure 2-2 in which the heart of the automation system is an IBM 9020 computer [12].

Each central computer will maintain the geographical position, altitude, and flight data for all controlled aircraft within its area of jurisdiction. Displays driven by the computer contain flight information which is automatically updated. Coordination between control positions, such as the "hand off" of

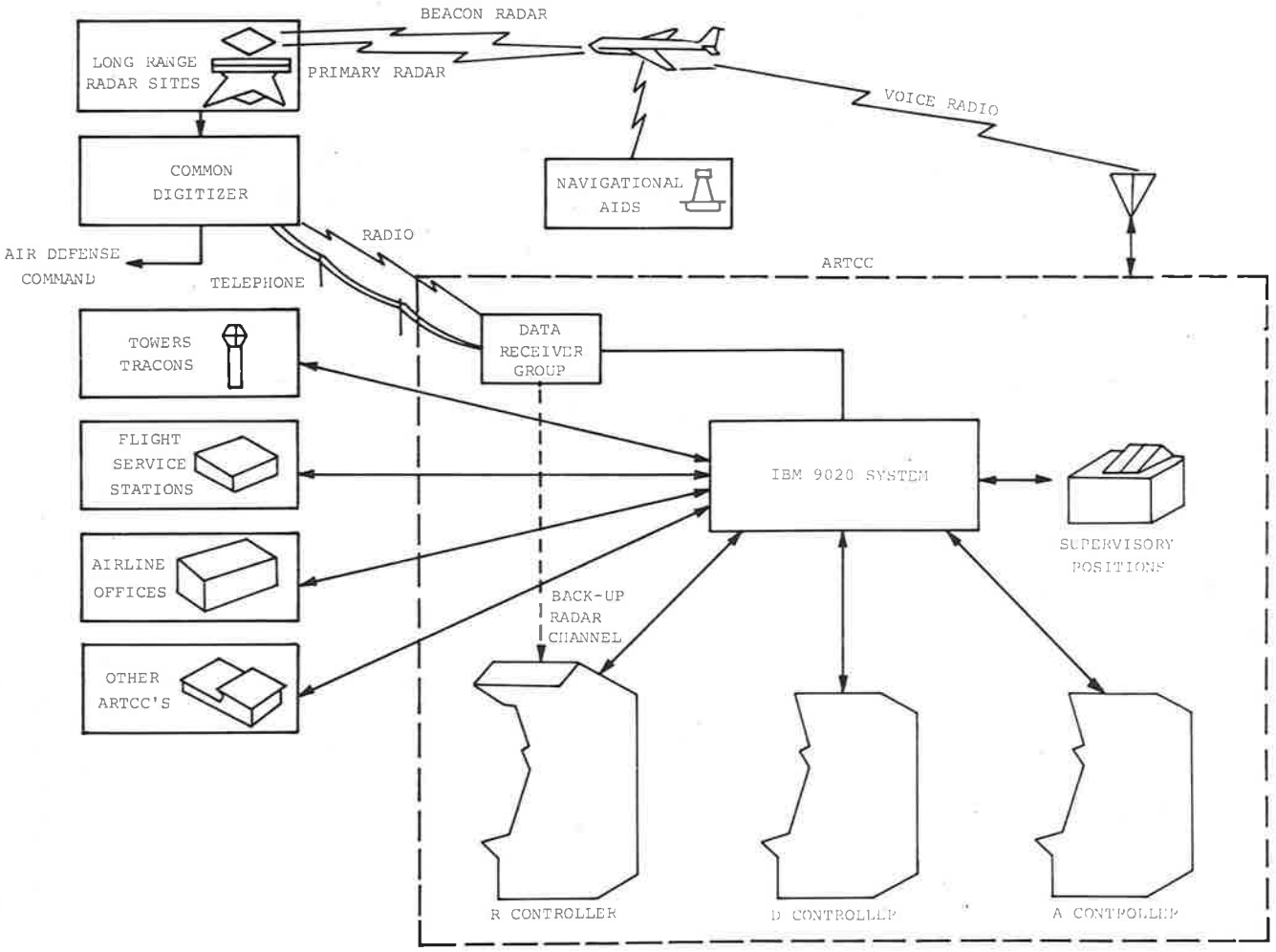



Figure 2-2 NAS Stage-A Enroute ATC System

aircraft while traveling from one control sector to another, will be computer-assisted. Upon request, the central computer will provide the requested flight control information in real-time. With the computer handling most of the bookkeeping, coordination, and "remembering" activities, the controllers will be able to devote more of their time to situation monitoring, and emergency handling functions.

The configuration of the central computer complex varies from site to site, however, Figure 2-3 represents a typical system which consists of three computing elements (CE), three I/O control elements, and nine storage elements (SE). This multiprocessor system provides high degrees of capacity "availability" at extra cost in redundant equipment. This capability of configuring and reconfiguring redundant equipment in and out of system is achieved by system software and hardware functions which are invisible on the block diagram. There are two system software programs and five operational programs. The control program [13] is the executive which oversees operations of the entire system. The Operational Error Analysis Program [14], upon detection of an error, analyzes and locates the failure, and effects system retry and/or reconfiguration. The five operational programs are: Radar Processing, Flight Plan Processing, Liaison Management, Input Processing, and Output Processing [15].

This system consists of pioneering hardware and software which have been designed to meet special requirements. It works well for the application in terms of modularity and availability. However, it ran into some difficulties during its software development in terms of meeting real-time response requirements. This is due partly to the fact that the general-purpose software concept was adopted for special purpose real-time applications, which created some unexpected timing delays. These timing anomalies will be discussed in more detail in Sections 4 and 5.

### 2.2.2 Advanced Radar Terminal System (ARTS)

Nelson and Sunderman [16] give an excellent summary on the history of the terminal ATC automation program. In brief, there

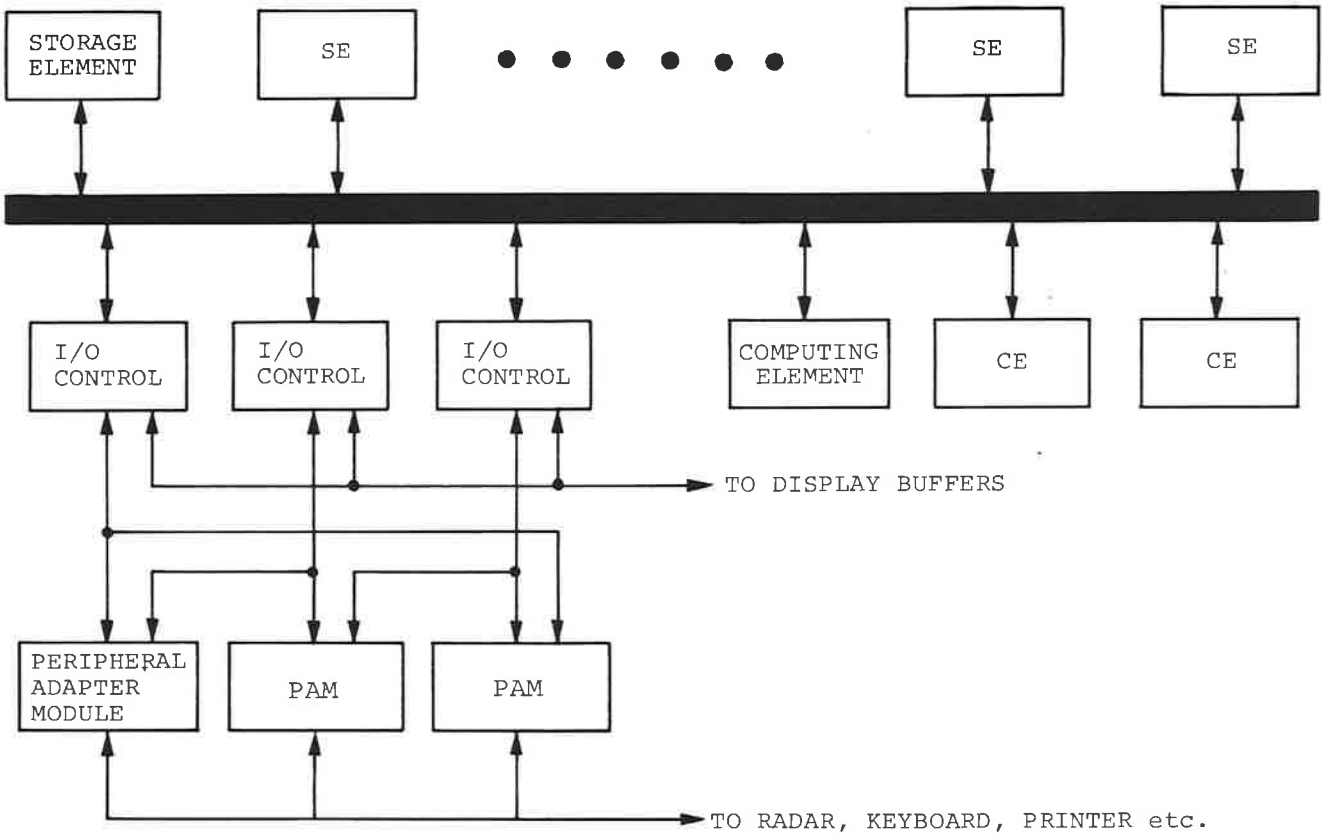



Figure 2-3 A Typical 9020 System Configuration

are three forerunners of the ARTS system. The first system was implemented in Atlanta, Georgia in 1963 and declared operational in 1966. The second system was installed in 1966 in the New York Common IFR Room and has been servicing Kennedy, La Guardia, and Newark airports since 1969. The third installation took place at Knoxville, Tennessee in 1969 and was operational in 1970. (A typical terminal ATC system configuration is illustrated in Figure 2-4). At present FAA is installing a newer and better version, ARTS III, at 64 terminal control centers throughout the United States. The basic functions that an ARTS system performs are: tagging the radar beacon targets on the display scopes with alphanumeric data for identification, keyboard hand-off function between controllers, and target detection and tracking.

The ARTS III system consists of three subsystems: Data Acquisition Subsystem (DAS), Data Processing Subsystem (DPS), and Data Entry and Display Subsystem (DEDS) [17]. A simplified information flow diagram for ARTS III system is shown in Figure 2-5.

DAS accepts broadband beacon replies and converts them into digital form suitable for further processing by DPS. Together with beacon replies, a set of associated azimuth, range, and timing information is also sent to DPS for further processing.

DPS accepts beacon replies from DAS, flight data from ARTCC and manual data entries from controllers via DEDS. Using this information, DPS detects targets and performs real-time tracking of the beacon-equipped aircraft in the terminal area and provides output data to control the dynamic display of alphanumeric formats on the DEDS.

DEDS provides the man/machine interface between the air traffic controllers and the ARTS equipment. It accepts the alphanumeric flight data (identification and altitude) from DPS and superimposes it over a broadband radar plan position indicator (PPI) display of video map, radar, and beacon signals. It permits controllers to enter and retrieve a variety of flight data via a combination of alphanumeric and functional keyboards.

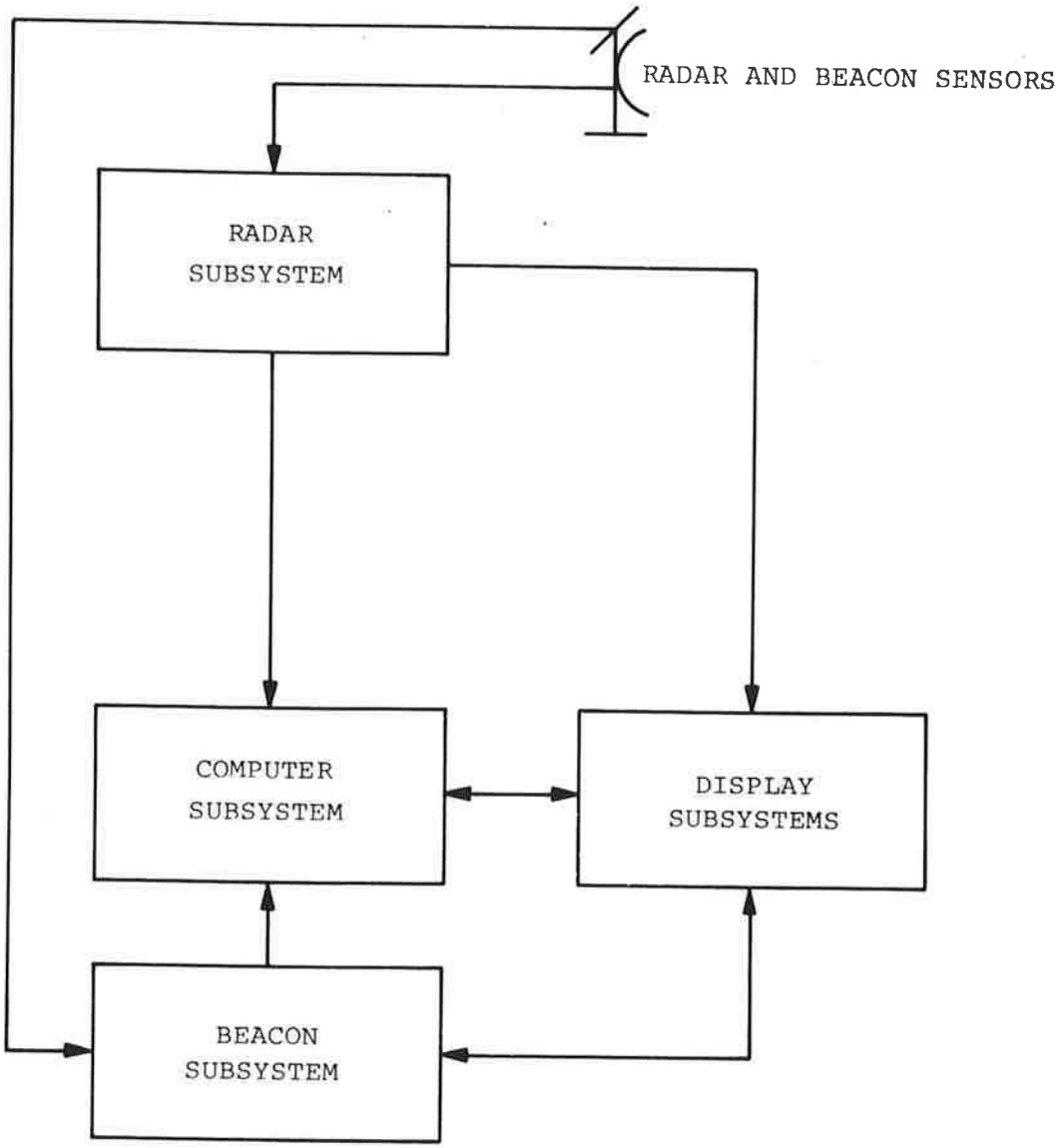



Figure 2-4 A Typical Terminal ATC System Configuration

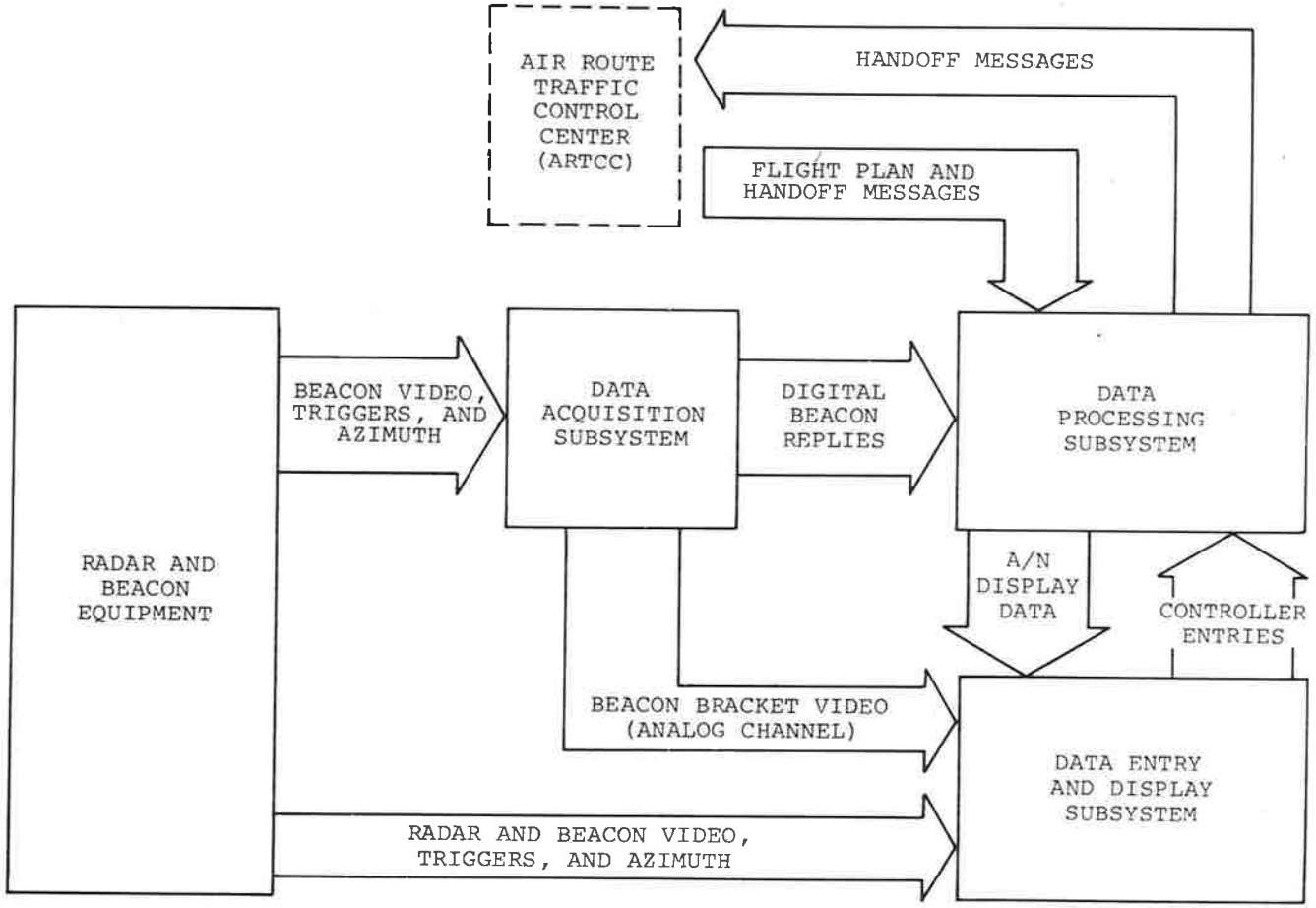



Figure 2-5 ARTC III Information Final Diagram

The heart of the ARTS III system is a UNIVAC 1230 computer which is modularly expandable into a multiprocessor structure. A well documented report [18] by UNIVAC describes the ARTS III system in great detail.

### 2.2.3 Future ATC Demands

In view of the continuing increase in air traffic in the summer of 1968, the U.S. Secretary of Transportation organized an Air Traffic Control Advisory Committee (ATCAC). Its report [19] submitted in September of 1969, has been accepted by the FAA and the Department of Transportation (DOT) as the basis of an additional development program.

In this report, Ashby [20] estimated that the total aircraft activity in the United States will almost double between 1968 and 1980, and more than double between 1980 and 1995, for a four-fold increase overall. The projection is made in terms of annual aircraft operations (take-offs and landings) and is shown in the following Table 2-1.

TABLE 2-1 ANNUAL OPERATIONS, 1968-1995 (IN MILLIONS)

| | 1968 | 1980 | 1995 |
|---|---|---|---|
| Air Carrier | 11 | 21 | 31 |
| General Aviation | 84 | 167 | 448 |
| Military | 33 | 34 | 40 |
| Total: | 128 | 222 | 519 |

From the safety viewpoint, Graham and Orr [21] analyze the near mid-air collision data gathered by FAA in 1969 and show that the mid-air collisions increase as the square of traffic and obey a random gas law in the mixed controlled portion of the airspace. Willis et.al. [22] proposed an Intermittent Positive Control (IPC) technique to help nominally uncontrolled aircraft to avoid collisions. All these analyses call for more automation aids to the current system. Given these projections and the premise of more automation aids, Blake and Nelson [23] analyze the computer power necessary to incorporate these ends into the current system. The demands are translated into storage require-

trated in Figure 2-7.

The most significant target tracking operation is track correlation. Correlation is the process of matching incoming target report information with established tracks. There are five correlation classes called initial, primary, normal trial, secondary, and turning correlation. Correlation may be performed on both target position and assigned beacon code.

In position correlation, a "bin" representing an assigned volume of airspace is constructed around a track's predicted position. The bin size is determined by the correlation class and track firmness. The configuration of the correlation bins is shown in Figure 2-8. As each target report is presented to the tracking function, the appropriate bin limits are placed around the predicted position of all tracks one by one and the entire track file is searched to determined if a target's reported position falls within the limits of any previously established tracks.

A newly activated track is given an initial track status. The initial track is correlated with only one bin per track in the initial correlation process. Unless only one track bin correlates with the report, the beacon code is also correlated. Table 2-4 shows that three sucessful correlations are sufficient to establish a normal track. Unsucessful correlation will change the track to a "coast" status.

Successful normal track primary correlation requires report correlation with a track's primary bin and with the assigned beacon code. If the primary correlation is successful, the report information is saved for track position correction and prediction. Secondary correlation is not required unless correlation with all primary bins or beacon codes has failed.

Secondary correlation bins are built around the same predicted position as the primary bin. If no correlation exists with the secondary bins, the reported position is correlated with left and right turning trial bins. The turning trial bins are built around predicted turn positions, and their sizes are the same as the primary bin.

the target tracking program.

### 2.3.2 Target Tracking Function

The primary purpose of the tracking function is to maintain the correct association between beacon target reports and the aircraft flight paths. The tracking technique employed in ARTS [24] may be generally characterized as a multi-state, alpha-beta tracker with logical turn detection. The multi-state characteristic permits adjustment of accumulated history for each track. The alpha-beta tracker has been most widely used in real-time tracking systems because it represents a reasonable trade-off between tracking accuracy, responsiveness, stability, computation time and computer storage requirement.

The primary operations performed in target tracking are: track initiation, track correlation, track firmness update, track correction, and track prediction. A "track" can be loosely defined as the path of a target or aircraft as seem by the computer. A central track file will consist of reported and predicted positions, firmness, velocity, beacon code, altitude, time of last report and other required data. Firmness is a measure of the stability of a track and an indication of the history of that track; the larger the firmness value, the more stable the track is in the system.

The FAA has established 31 firmness values divided among three track classes. The three track classes are initial, normal trial and normal. A new track introduced into the system is called an initial track, and is assigned a firmness value in the range of 0-7. A normal track is a track whose status has been assured and firmness value ranges from 12-31. A normal trial track is a track that is formed when a normal track begins to deviate from its expected position. Its firmness value ranges from 8-10.

The tracking control program is performed once every 125 ms on a sector basis (11.25° of radar scan is equal to a sector). The tracking operations sequence and their associated sector bases on which these operations are performed is illus-

4. Reflections - Some sites may be located so that surrounding buildings and terrain reflect replies. Targets are reported in false positions as well as in their true positions.
5. Fruits - A beacon may receive an erroneous reply from a target interrogated by another ground station.

Target detection process performed after each sweep (once every 2.5 ms) consists of two steps, reply correlation, and target declaration. The reply correlation process is to correlate new replies collected in a sweep with existing target records and to up-date these records. When a new reply finds a match with a target record, a "hit counter" is incremented. When no match is found, the new reply is entered into the target record as a possible new target. When all the new replies in a sweep are processed, the target record is examined. These targets that have not correlated with any new reply at all will have their "miss counter" incremented. The combinations of hits and misses of all targets will be examined next to see whether a set of criteria could be met. If so, those target records will be declared as true targets.

Target declaration process involves the computation of center azimuth of targets using a center of density technique. Let

AT = Azimuth at trailing edge of the run
R = Run length in sweeps
S = Sum of the sweep counts
H = Number of hits
F = A conversion factor

The center azimuth (AC) is then

$$AC = AT - F(R - \frac{S}{H})$$

The criteria used, such as number of consecutive hits for the establishment of leading edge of a run, number of consecutive misses for the establishment of trailing edge of a run, minimum number of hits requirement for a run, etc., are system parameters and they vary from site to site. The declared targets grouped together as beacon target reports are used as inputs to

mately 18 times (run length), although responses varying from 9 to 26 are common. This is illustrated in Figure 2-6.

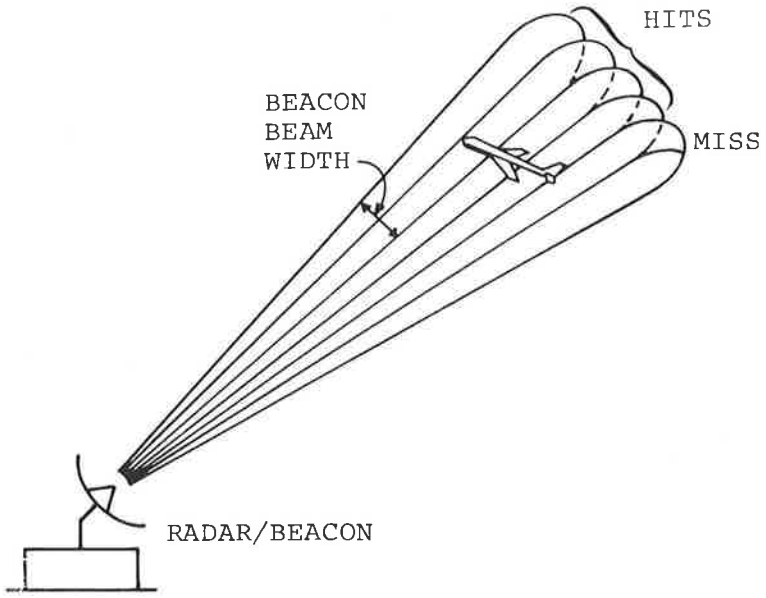



Figure 2-6 Hit/Miss Pattern of Beacon Responses

Some of the signal defects and their causes are given below:

1. Garbled Replies - Replies received from two or more aircraft in close proximity may be overlapped.
2. Target Splits - During the run length of a target a transponder may fail to respond due to over-interrogation from other ground stations or because the aircraft is maneuvering its transponder antenna out of position.
3. Ring-Around - Under certain conditions some transponders will respond to side lobes of the interrogation radiation pattern which causes excessive run lengths.

make sure that no aircraft is in a collision path with another aircraft at all times, Collision Prediction Function. The more accurate results that these functions could provide the system with, the more confident we are to give control commands to the aircraft, and the better the ATC system.

It is intuitive to note that whatever processes and/or computations are needed for these ATC functions, they have to be conducted for each and every aircraft under control, which means identical processes have to be repeated on a sequential machine. This undoubtedly suggests the possibility of parallel processing to increase the system performance. This portion of the study attempts to identify key processing characteristics of these three basic ATC functions so that parallel processing techniques could be introduced and novel computer architecture could be formulated to obtain performance increase.

### 2.3.1 Target Detection

The target detection function processes beacon reply messages received from the data acquisition system (DAS) and generates beacon target report for the subsequent use by the tracking function. Due to signal defects, some beacon replies are noise induced, and on the other hand, some true targets may not produce any reply momentarily at all. The target detection function will hence attempt to identify and minimize the effect of these defects.

Beacon antenna and radar antenna are co-located on the same rotating shaft scanning at a rate of 15 rpm's or one revolution in four seconds. While it is moving, the beacon antenna radiates a narrow beam of energy in the form of pulse train (called sweep) at 400 sweeps per second. However narrow, the sweep has a finite width which means that the resolution of beacon targets will vary from one to three degrees, depending upon variations in equipment sensitivity, aircraft position, atmospheric conditions, etc. At the interrogation rate stated above, a typical target could be expected to respond approxi-

ment and instruction execution rate which are tabulated in Tables 2-2 and 2-3.

TABLE 2-2 STORAGE REQUIREMENT (WORDS)

| | Enroute | Terminal | National |
|---|---|---|---|
| Data Acquisition | 74000 | 50000 | |
| Command & Control | 6300 | 80000 | |
| Present ATC Functions | 1150000 | 260000 | |
| Flow Control<br>Collision Avoidance System | 99400 | 75000 | 62200 |
| Monitor | 100000 | 75000 | |
| Total | 1486400 | 546000 | 62200 |

TABLE 2-3 MAXIMUM INSTRUCTION EXECUTION RATE (MILLION INSTRUCTIONS/SEC.)

| | Largest Enroute | Largest Terminal |
|---|---|---|
| 1980 | 22.5 | 19.9 |
| 1995 | 32.5 | 28.4 |
| Max. Sizing Model | 48.5 | 42.7 |

Comparing these estimates with the projected state-of-the-art, computers of sufficient capability will be available long before 1995 to meet the 1995 ATC requirements. The strong implication is that, while the computer hardware may be available, the complexity of the software in a more highly automated air traffic control system is a major challenge to the system designers and computer industry.

## 2.3 CHARACTERISTICS OF BASIC ATC FUNCTIONS

Although there are quite a number of ATC functions either being automated or under consideration to be automated, there are only three basic ATC functions on which other automation functions are to be built. First, we have to know accurately all the targets in the vicinity of the control area, Target Detection Function. Second, we have to know accurately at all times what and where every detected aircraft is, their altitude, direction, speed, etc., Tracking Function. Third, we have to

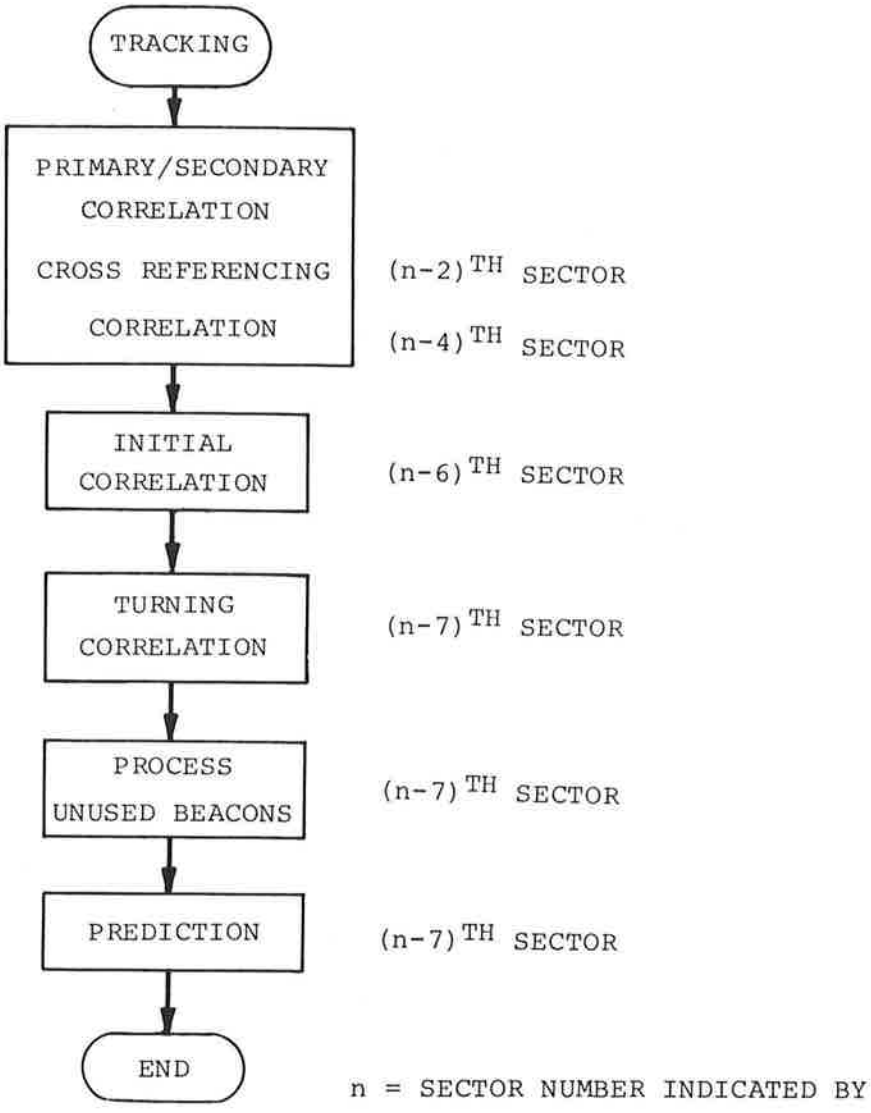



Figure 2-7 Tracking Program Sequence

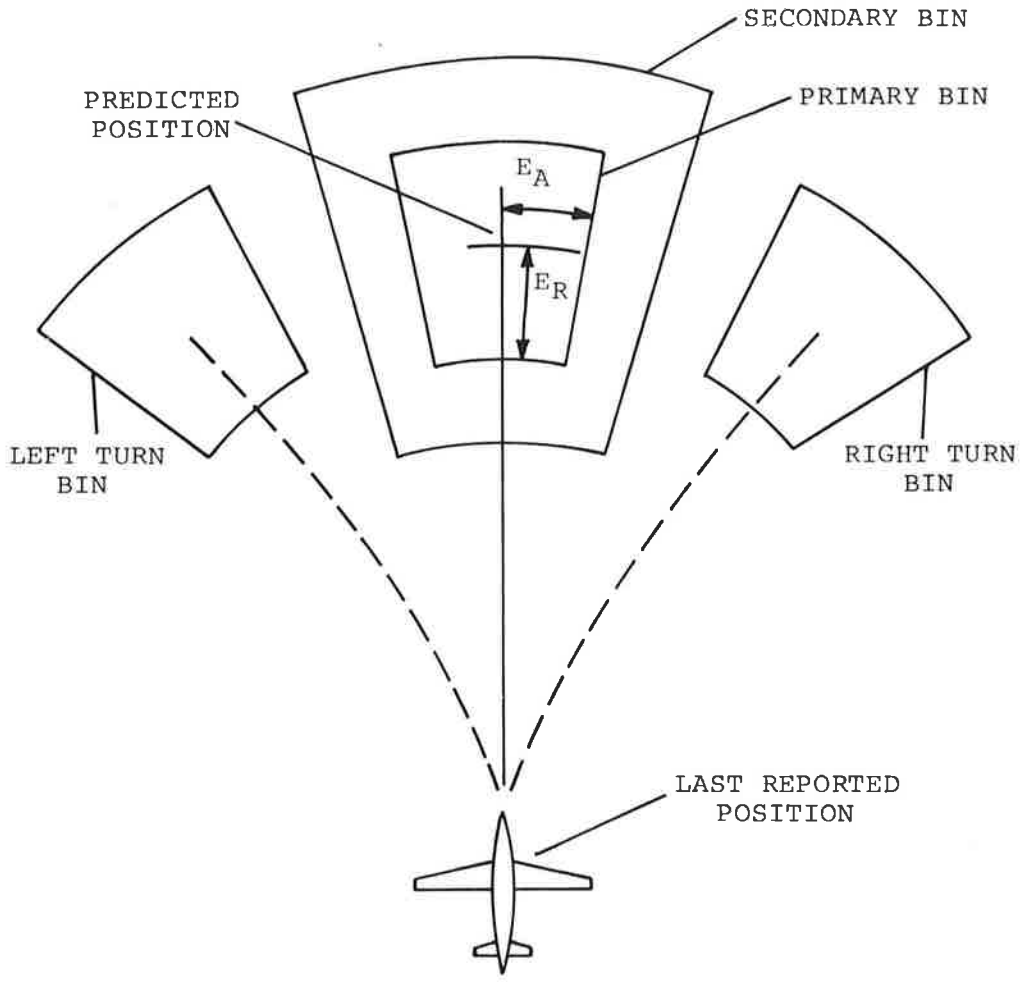



Figure 2-8 Bin Configuration in Track Correlation

If there exists a correlation with either the secondary or turning trial bins, a new normal trial track is established with a firmness of 10 (Table 2-4). The normal track has its firmness decreased and is coasted. Each time that neither the normal nor the normal trial tracks correlate, both firmness levels will be decreased and both tracks will be coasted. If the normal track correlates, its associated normal trial track will be dropped. If the normal trial track correlates and the normal track does not correlate, the normal trial track will be made a normal track and the previously established normal track will be dropped. No turning track correlation is performed on trial tracks or their associated normal tracks.

Should both secondary and turning correlation fail, the reported beacon code and track assigned beacon codes are correlated. If this correlation fails also, that target report is ignored. If beacon codes do correlate and the reported target and predicted track positions are within certain bounds, the report information is saved for track position correction and prediction [24].

After each successful correlation, the corresponding tracks correction is performed based on its reported position and its predicted position from the previous scan. The correction is performed on both X and Y coordinates but only the X coordinate is illustrated.

Position

$$(X_c)_n = (X_p)_n + \alpha(X_r - X_p)_n$$

$X_c$ - corrected position

$X_p$ - predicted position

$X_r$ - reported position

$\alpha$ - position smoothing parameter

$n$ - nth scan

TABLE 2-4 TRACK FIRMNESS UPDATE SCHEDULE

| Previous Firmness | Successful Correlation | Unsuccessful Correlation |
|---|---|---|
| 0 | 3 (Follow Only) | 0 Permanent Coast |
| 1 | 5 | 0 |
| 2 | 6 | 1 |
| 3 | 7 | 2 Initial Track |
| 4 | 13 | 0 |
| 5 | 14 | 4 |
| 6 | 15 | 5 |
| 7 | 16 | 6 |
| 8 | 14 | 0 |
| 9 | 15 | 8 Normal Trial Track |
| 10 | 16 | 9 |
| 11* | 10* | Unused |
| 12 | 14 | 0 |
| 13 | 15 | 12 |
| 14 | 16 | 13 |
| 15 | 17 | 14 |
| 16 | 18 | 15 |
| 17 | 19 | 16 |
| 18 | 20 | 17 |
| 19 | 21 | 18 |
| 20 | 22 | 19 |
| 21 | 23 | 20 Normal Track |
| 22 | 24 | 21 |
| 23 | 25 | 22 |
| 24 | 26 | 23 |
| 25 | 27 | 24 |
| 26 | 28 | 25 |
| 27 | 29 | 26 |
| 28 | 30 | 27 |
| 29 | 31 | 28 |
| 30 | 31 | 29 |
| 31 | 31 | 30 |

*Used to establish a trial track.

Velocity

$$(\dot{X}_c)_n = (\dot{X}_c)_n\text{-}1 + \frac{\beta}{t}(X_r - X_p)_n$$

$\dot{X}_c$ - corrected velocity

$\beta$ - velocity smoothing parameter

$t$ - time since last correction

The value of the smoothing parameters is a function of track firmness. The progression of firmness states and their corresponding alpha and beta were chosen in such a manner that they would approximate a least-square straight line fit of a succession of position reports. Since this method, a straight line fit, will not track turning aircraft efficiently, secondary and turning correlations are used to detect a turn and to adjust the associated firmness.

After each scan, every normal track will be given a predicted position to the time of the next radar/beacon scan following the equation below:

$$(X_p)_{n+1} = (X_c)_n + (\dot{X}_c)_n \cdot T_n$$

$$(Y_p)_{n+1} = (Y_c)_n + (\dot{Y}_c)_n \cdot T_n$$

$X_p, Y_p$ - predicted position

$X_c, Y_c$ - corrected position

$\dot{X}_c, Y_c$ - corrected velocity

$T_n$ - period of last radar/beacon scan

For a turning track the prediction equations are:

$$(X_p)_{n+1} = (X_c)_n + V \sin(\theta \pm R \cdot T_n) \cdot T_n$$

$$(Y_p)_{n+1} = (Y_c)_n + V \cos(\theta \pm R \cdot T_n) \cdot T_n$$

$(X_c)_n, (Y_c)_n$ - current position

$(X_p)_{n+1}, (Y_p)_{n+1}$ - predicted position

$V$ - speed of track

$\theta$ - heading of track measured clockwise from North

$R$ - turn rate of track

### 2.3.3 Conflict Detection Function

One of the most important functions that an air traffic control system must perform is the detection of potential collisions (conflict detection) and the implementation of steps to avoid them (conflict resolution) which comprise the overall collision avoidance system (CAS). At present, ARTS does not have any CAS system in operation. However, at Knoxville, Tennessee a prototype system is being tested and evaluated. Of the two functions, conflict detection is more basic, requires a great deal of computation, and is more adaptive to parallel processing. Hence, only conflict detection will be discussed here.

Willis, et.al.[22] further classifies conflict detection into two categories: distance filtering and conflict determination. Distance filtering is to filter from all possible aircraft pairs those that are within some predetermined distance from one another. Conflict determination determines which of the pairs filtered will present a real threat of collision when possible future positions are taken into account. At present, the distance filtering is performed by human controllers by simply viewing the separation of the aircraft on radar scope. For a computer which cannot "perceive" a pictorial pattern rapidly, this can be a difficult problem. In the simplest and straightforward filtering algorithm, distances between all aircraft pairs are computed in a radar/beacon scan and those falling below a particular threshold will be marked for conflict determination processing. If there were 5000 aircraft and each inter-aircraft distance comparison is assumed to require 10 instructions, the filtering process could require approximately $10^8$ operations ($1/2 \times 5000^2 \times 10$). Although some giant computers today may achieve this computation rate, they are certainly not cost effective. Clearly better ways of performing distance filtering is needed to reduce this computation load.

A generally accepted air space conflict model has two important parameters. These are the minimum acceptable miss

distance within which collision is imminent, and the minimum aircraft to maneuver out of conflicting situations. These two critical parameters can be projected into the volume of air space which surrounds each aircraft. This is illustrated in two dimensions in Figure 2-9.

With this conflict model, Willis, et.al., proposed an "Implicit Geometric Filtering" technique where air space is first divided into equal-volume cells, and each is given a unique, serial identification. As shown in Figure 2-10, each aircraft's identification number is entered into the cell that contains its current position as well as the cells intersected by its area of uncertainty. In this example, aircraft A and B produce a real conflict which must be resolved with a more detailed calculation. Aircraft B and C or A and C present potential conflict which will be ranked second in priority for the conflict determination process.

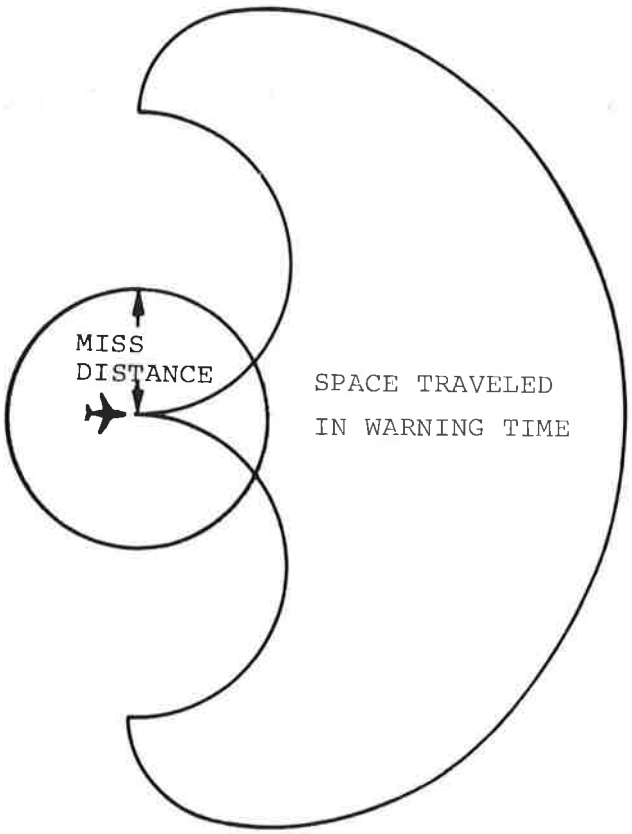



Figure 2-9 Conflict Space of a Single Aircraft

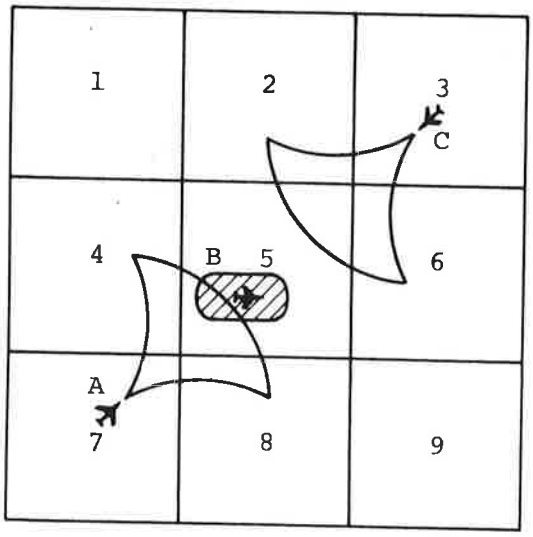



Figure 2-10 Implicit Geometric Filtering

# 3.0 AN APPROACH TO PARALLEL PROCESSING

In the second section, we have briefly described the principles and techniques in the realization of three basic ATC functions. In this chapter, we shall analyze these techniques and attempt to unveil some of their processing characteristics which are adaptive to parallel processing. After a brief survey of existing and/or developmental parallel processors, which may do the job, we formulate and recommend a novel architecture for such parallel processing activities with high efficiency.

## 3.1 ATC PARALLEL PROCESSING ANALYSIS

Both steps in the target detection function can be processed in parallel. The reply correlation is nothing but searching through the entire beacon record for a match with a specific reply. The search results can be marked at corresponding beacon record entries. At the end of each beacon sweep all hit and miss counters can be incremented in parallel. Hence, all updates, such as sensing for leading and trailing edges, etc., can also be done in parallel. When targets are ready to be declared, the center azimuth computation can certainly be carried out simultaneously. It is important to note that some processes (e.g., search) are logical and some (e.g., center of azimuth calculation) are arithmetic parallel processes.

In target tracking all types of correlations can be performed in parallel since they are carried out by repetitive between-the-limits search operations, which are logical processes. In track correction and position prediction the computations can be performed in parallel also provided that there are parallel arithmetic units.

It is in the conflict detection function, however, that the parallel processes have the greatest pay-off. As stated in Section 2, the implicit geometrical filtering technique could

indeed reduce computer operations from $N^2/2$ to N (N is the number of aircraft). However, we must take pains to compute, in addition to its present position, those critical positions covered by its area of uncertainty as well. With parallel processing capability these $N^2/2$ operations become tractable. The greatest advantage is that simple straightforward algorithms may be used without relying on complex computations.

## 3.2 PARALLEL PROCESSOR SURVEY

Among many good survey papers, Hobbs [25] conducted a comprehensive investigation in parallel processor systems, technologies, and applications, No attempt is made here to cover as much front and depth as he did, but rather to highlight some systems which are more relevant to the real-time problems specified; namely, Air Traffic Control.

Aside from the theoretical treatment by Holland [26] and Comfort[27], the first parallel processor under development is the ILLIAC IV[28] which was structured after Solomon computer[29]. ILLIAC IV consists of a two-dimensional array of 256 processing elements arranged in four sub-arrays or quadrant of 64 processing elements each. Each element, in addition to having its own fast local store and powerful arithmetic unit, has the capability to communicate with its four neighbors. Each quadrant has a control unit which controls the operations of its 64 elements. Driven by a host machine, the system can be operated in several configurations. For example, all control units can be executing the same instruction stream, or each could be executing a different instruction stream.

A somewhat different approach to parallel data computations is that of the associative processor. The classic associative processor utilizes associative or content-addressable memory techniques with the inclusion of additional processing operations on each word of data. Fuller[30] describes the associative memory processor with what he termed "sequential state transformation". Sequential state transformation is a computing technique for performing word-parallel, bit-serial

logical and arithmetic operations on data stored in an associative memory. This technique is based on the capability to perform an associative search, perform a Boolean operation simultaneously on data from all words responding to the associative match, and multi-write the results simultaneously back into the responding words.

Shore and Polkinghorn, Jr.[31] have proposed a detailed design of a sophisticated, and supposedly flexible, associative memory processor based on the techniques described by Fuller. In this processor, the total associative word length can be segmented on a software basis into any number of fields called syllables, each of arbitrary length. Associative memory search criteria (greater than, less than, greater than or equal to, less than or equal to, exact match, and don't care) can be specified independently for each of these syllables. If a particular word satisfies the criteria in an associative search, a bit is set in a response store (RS) section of that word. A record of previously satisfied searches may thereby be maintained in the RS of each word. Multi-add and multi-write parallel-bit operations (key register to associative memory word) may be performed simultaneously on all words responding to an exact match in a RS search. Complex arithmetic and logical operations on two syllables of the same word may be implemented using the associative search, the multi-add and multi-write operations, and Fuller's sequential state transformation technique. These operations would be executed bit-serially on each word, with the execution time dependent on the width of the largest syllable.

Recently, Meilander[32] and Eddy[33] proposed the application of an associative processor to perform a ground-based conflict prediction function. FAA has procured the use of a plated-wire associative processor for air traffic control evaluation at Knoxville, Tennessee[34].

Shooman[35] has presented a different approach to parallel processing based on an orthogonal memory. An orthogonal memory has two addressing directions: word-wise as in conventional

random access memory, and bite-slice as in associative memory. The orthogonal memory itself does not have any matching or processing capability. Computation and data processing are also done in two directions; word-wise by conventional arithmetic unit and bit-slice by a number of serial function generators (FG). With the help of LSI, he assumes 1000 FG's for a typical system, dividing memory into 1000-word blocks. Hence, parallel processing is done on a 1000-word basic by bit-slicing them out to the FG's.

A new parallel processing concept, reported by Githens[36], depicts an unstructured ensemble of any desired number of identical processing elements under a common control unit. This machine, called Parallel Element Processing Ensemble (PEPE) is intended as a supplement to conventional computers. The control unit exercises global control over PEPE elements via common input/output and control buses. All elements receive the same input and control signals and the ensemble as a whole performs on common operation at a time. Individual elements either participate or not, depending on their internal state which can be conditionally set by a global command. Each processing element has three functional units: the local memory unit, arithmetic unit and a correlation unit. The purpose of the correlator is to facilitate imput data to the corresponding element in a rapid pace.

Litzler[37] reported a similar development as PEPE in the class of unstructured parallel ensemble machines. This one, called Single Instruction and Multiple Data Ensemble (SIMDE), is modelled after PEPE with two modifications. SIMDE did away with the correlation unit in each element. Instead, each element has a condition code register. The global control unit could sense the condition of this register and change the status (active, inactive, hold, etc.) of the element accordingly. The second modification is in the output control. For its application, SIMDE is required to output data at a much faster pace than PEPE does. Hence, a priority output bus and its control are added. Both PEPE and SIMDE use micro-program memories for the

mechanization of the control logic.

One of the fastest super machines in operation at present is the STAR computer[38] which accomplishes parallel processing by serially streaming large data blocks through a high-speed "pipeline" arithmetic unit. The ultra high-speed performance is achieved through system organization rather than technology breakthrough. Eight 1.28 μs memory modules are interleaved at 160 ns apart. Each memory module has 2K words by 512 bits per word. Each word is divided into four 128-bit fields, which could be pumped through the pipeline at a 40 ns rate. If indeed pertinent data can be arranged such that consecutive 8-word fields (computer word is 64 bits, memory word is 512 bits) are stored in corresponding locations of separate memory modules, the system can keep the pipeline going at its full speed which amounts to 25 million operations per second.

## 3.3 A NOVEL ARCHITECTURE FOR PARALLEL PROCESSING

As indicated in the previous survey, a basic associative memory (also called content addressable memory) is best suited for logical parallel processing. However, its bit-sliced arithmetic operation, although parallel in word, is quite a limiting factor. With short operands and simple "counting" processes (either increment or decrement), the slow speed bit-sliced operation is still tolerable. When the operands are long or when there are multiplication or division involved in the arithmetic operation, the system advantage of associative memory is then diminished, simply because it takes a long time to do bit-serial multiplication and division. What we need then is a high performance arithmetic parallel processing capability superimposed on an associative memory. It is interesting to note that both PEPE and SIMDE class of machines achieve this goal of possessing both logical and arithmetic parallel processing capability in a completely reverse order. Multiple conventional processors are provided to form a base for arithmetic parallel processing. Logical parallelism is

achieved by carrying out sequential searches by various processors simultaneously.

We propose to integrate a "pipeline" processing capability with the associative memory to form a powerful parallel processing unit. Pipelining is the least expensive way to achieve parallel processing because the concept does not require additional processing units, but rather control of the streaming of input and output data in and out of a simgle processing unit. Before exploiting the "Pipelined Associative Processor" (PAP) operating principles, let us first turn our attention to see how an associative memory works.

### 3.3.1 Basic Associative Memory Operations

Associative memory is a unit capable of performing a storage function and also of performing a set of operations over the stored words. A significant feature is that the operations can be performed simultaneously over all stored words and/or over any selected field of all stored words. The operations can be divided into two categories: search operations and processing operations. Typical search operations include equality search, inequality search, maximum search and minimum search. Typical processing operations would be the logical combinations of these search operations plus consequent updates, i.e., read and write operations of certain storage locations. Parallel operations over selected fields are accomplished through the use of masking patterns which can be easily done by hardware logic and it will not be elaborated on here.

Figure 3-1 illustrates a generalized block diagram for the associative memory. A brief example will help demonstrate the parallel processing capability of the associative memory.

A segment of contiguous 1's in the Mask Register establishes a valid field which presents the argument 0101 of the Key Register to the entire associative array for parallel search over all words. Exact matches induce proper identification in the Match Result Register. In case of multiple

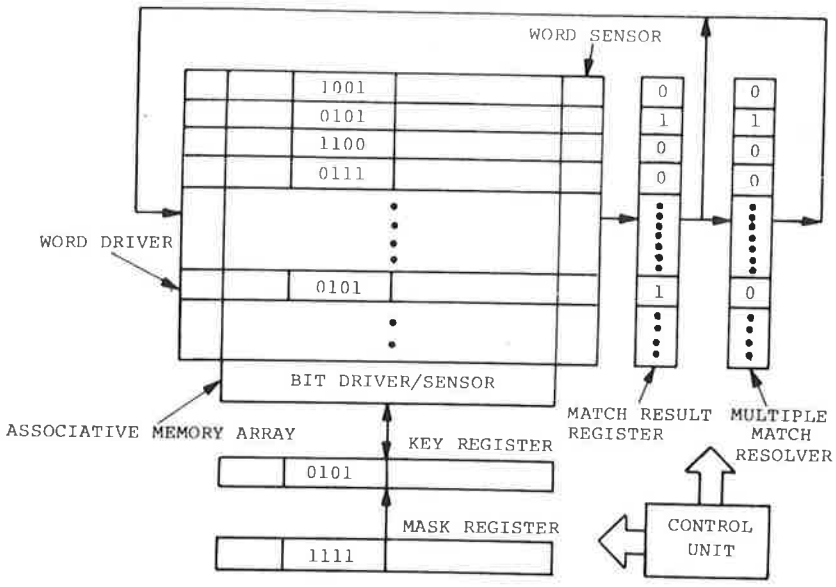



Figure 3-1 Operational Principle of Association Memory

matches as shown in the figure, the multiple match resolver will pick the first word following an arbitrary but pre-determined scan. The entire word may then be read out into the Key Register. Hence, it completes the basic equality search cycle. In early models, the basic equality search on a specified field is carried out on a "bit slice" basis. By bit slice, we mean a bit column of the entire AM array. The $i^{th}$ bit slice is therefore contains the $i^{th}$ bit of every word in the memory. Hence, for a field search, consecutive bit slice search cycles have to be performed repeatedly. Recently, however, a number of solid state LSI AM chips were successfully developed to carry out the basic equality field search in a true parallel manner, i.e., word parallel as well as bit parallel fashion. For our investigation here, we assume an AM with true parallel basic equality search capability.

Other basic operations include read the next word and write the next word functions which take the single output of MMR to select the next word for read or write. Parallel

write function, which stores a common data into all words, can also be considered a basic AM operation since it requires only one memory write cycle to complete its action. Parallel write is accomplished by feeding MRR output directly to the word drivers for multiple word selection. These basic operations are mechanized into the following instructions:

| | |
|---|---|
| READ NEXT WORD | (RW) |
| WRITE NEXT WORD | (WW) |
| PARALLEL WRITE | (WP) |
| EQUALITY SEARCH | (EQ) |

### 3.3.2 Compound Associative Memory Operations

The capabilities of an associative memory are affected to a great extent by the facilities provided for processing the contents of the Match Result Register. The Multiple Match Resolver is one example which pinpoints the first matched word. The Match Result Storage is used to temporarily store the configuration of MRR for later recall. A logic complement can be performed in MRR to achieve an inequality search by way of an equality search operation. By providing logic operation capability between MRR and MRS, compound associative memory operations can be achieved by combinations of consecutive two or more basic operations. A typical set of compound instructions carrying out compound associative memory operations is listed below:

| | |
|---|---|
| GREATER THAN | (GT) |
| LESS THAN | (LT) |
| MAXIMUM | (MX) |
| MINIMUM | (MN) |

No attempt is made to describe, in detail, how these compound operations are mechanized. In general, they are carried out on a bit slice basis, and the results of each bit slice comparison must be interpreted before the next step can take place. More compound operations can be created from the basic equality search operation and compound operations. For

example,

| | |
|---|---|
| GREATER THAN OR EQUAL TO | (GE) |
| LESS THAN OR EQUAL TO | (LE) |
| BETWEEN LIMITS | (BL) |

can be achieved by executing GT - EQ, LT - EQ, and GT - LT sequences respectively.

Please note that arithmetic operations are purposely omitted here, since they are to be addressed later in the pipeline processing section.

### 3.3.3 Parallel Processing - Logical

Let us define loosely "logical processing" as a set of operations containing criteria matching, record up-date, and incrementing or decrementing. Parallel processing in an associative memory follows these steps closely. In general, criteria matching is needed to locate items of interest in a file of some sort. These items are "tagged" for later identification. Record up-date involves simple insertion or deletion of certain field data in a record. The notion of "tagging" is quite important in carrying out parallel processing. Within each associative word, there is a tag field and a temporary storage field, which are used to keep track of the processing history and the temporary results.

An example would be appropriate at this point. The beacon reply correlation part of the target detection function is chosen to demonstrate how parallel processing is achieved. For clarity, only the simplified version is presented here. We assume that each associative memory word stores one target record which, for this application, has the field configuration as shown in Figure 3-2.

For each beacon sweep, we take one reply at a time and correlate it with all target records on "range" in parallel. Tag those records when correlations are successful. Create new records from reply data if these replies do not correlate. After all the replies in a sweep have been correlated, we have two types of records: new records and records in process.


Figure 3-2 An Associative Memory Word

For new records, we then look for a sequence of consecutive hits (system parameter) to determine the leading edge of a target. For those records in process, we up-date the azimuth, increment the hit or miss counter and look for a sequence of consecutive misses (system parameter) to determine the trailing edge of a target. New records with established leading edges will be changed to records in process status, whereas targets with established trailing edges will be passed on for target declaration.

Target declaration requires arithmetic parallel processing, which will not be elaborated on here. The rest of the correlation logic is illustrated in a flow diagram in Figure 3-3.

The "bin" correlation process in the tracking function is another perfect example for the logic parallel processing capability of associative memory. Assuming all correct bin sizes are set up for all established tracks in the central track file stored in an associative memory array, for each newly declared target there are only four comparison operations (two GE and two LE operations on two separate fields, range and azimuth) need to be performed to complete the correlation.

Logical parallel processing also applies perfectly on the geometric filtering process of the conflict prediction function. First of all, some aircraft may be screened out of conflict contention by altitude filtering. This is done by

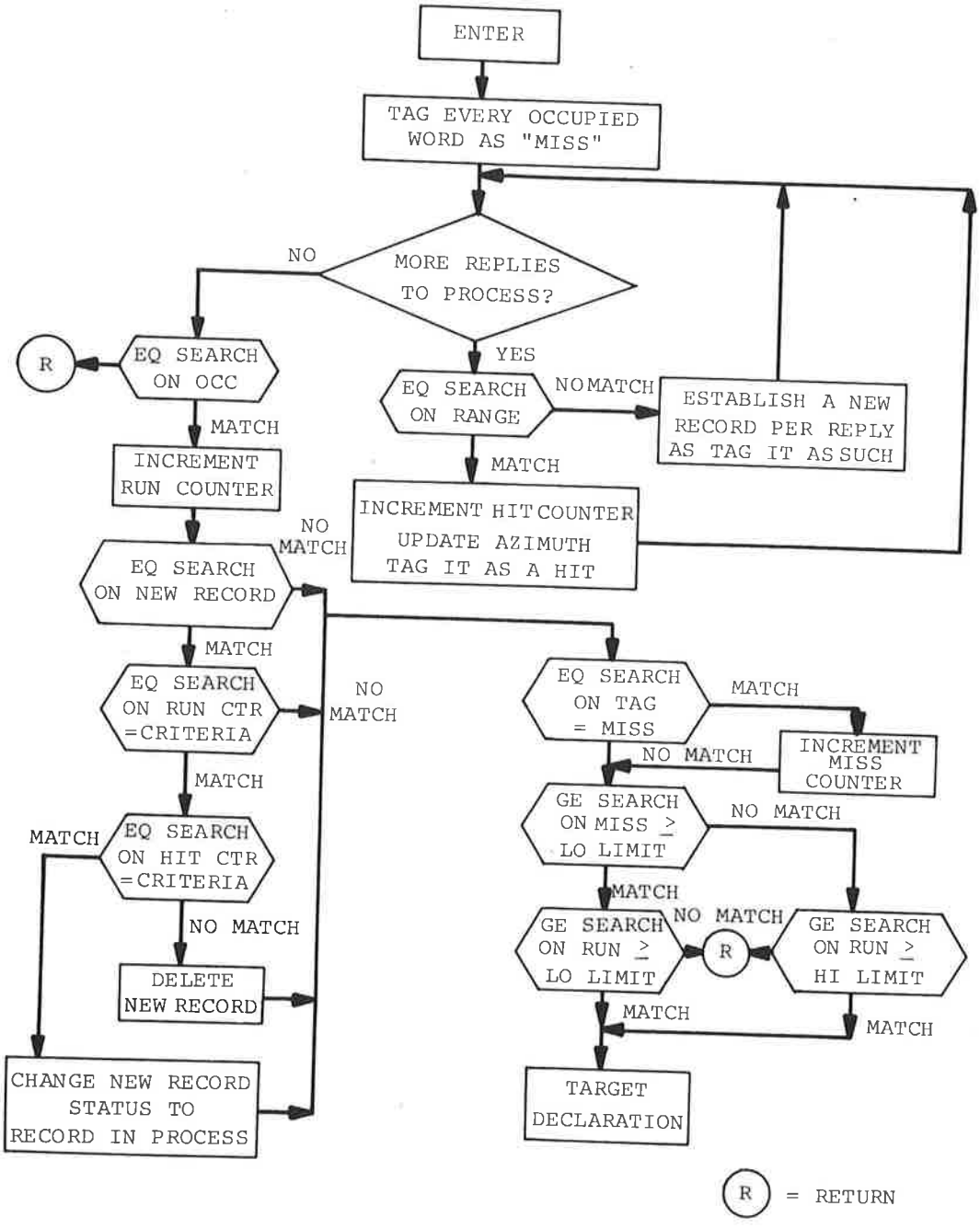



Figure 3-3 Target Detection Correlation Logic (AP Operation)

setting up the smallest altitude layers within which aircraft are considered to be unsafe, and are tagged for the second step planar filtering consideration. For each altitude layer there is a pair of limits (lower and upper). The whole process is carried out by setting up consecutive limit pairs and searching the entire central track file on "between limit" criteria. Aircraft found in between limits are then tagged for further processing.

For planar filtering, the terminal radar range of 60 miles may be sectionalized into 10 X 10 = 100 6-mile squares. For each square, two between-limits searches are performed to detect aircraft inside the square. For our discussion purposes, we may consider that aircraft pairs and/or groups within a 6-mile square need further processing; aircraft which do not share the same 6-mile square are out of conflict contention. Three more filtering paths with off-set grid patterns are necessary to accommodate those aircraft which happen to straddle across boundary lines. Figure 3-4 shows one possible set of off-set grid patterns.

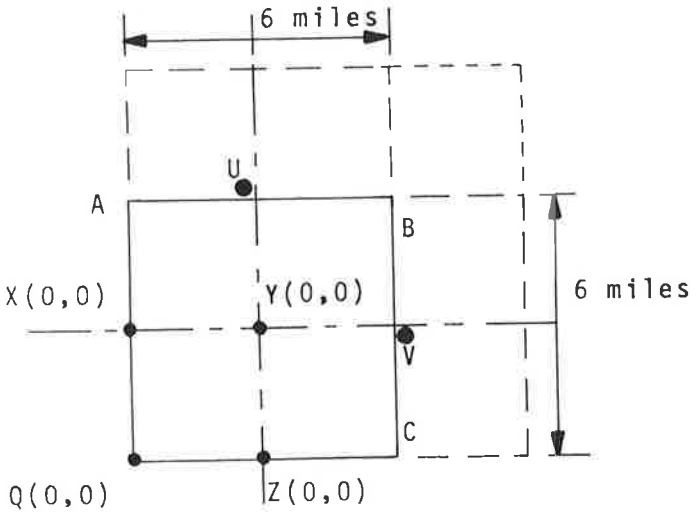



Figure 3-4 Off-Set Grid Patterns for Planar Filtering

The origin Q(0,0) of the 6-mile square may systematically move to positions X, Y, and Z for off-set. Aircraft straddle across lines AB and QC will be detected by square X(0,0); similarly, aircraft straddle across lines BC and AQ will be detected by square Z(0,0). Aircraft clustered around four corners Q, A, B, and C will be detected by square Y(0,0). The closest aircraft pairs which can go undetected by this method will be, for example, aircraft pairs U, V as indicated in Figure 3-4. For a 6-mile square the distance between U and V is over 4 miles. If a 4-mile distance is not enough for safely manuvering out of conflict, more filtering paths would be required. The main point here is, however, not to demonstrate any conflict detection algorithm, but rather to illustrate straightforward correlation methods carried out by an associative memory.

### 3.3.4 Pipeline Processing - Arithmetic

The performance comparison between associative processor and sequential machines can be typically demonstrated in Figure 3-5. A sequential machine is fast on unit computation and its total processing time increases linearly with the increase of number of compurations; whereas, an associative processor performing bit-arithmetic is rather slow, but its total processing time is independent of the number of objects to be processed. The crossover point is a function of the processors' operational characteristics. While retaining the bit-arithmetic capability for a large number of computations at the high end, a pipelined high-speed arithmetic unit is added to the associative processor to make its performance more attractive at the low end of the computing spectrun as well.

The concept of pumping large blocks of data streams through preset arithmetic pipelines to achieve greater increases in system throughput is quite well known. This technique has been used quite extensively in special-purpose applications, mostly defense and/or aerospace in nature.

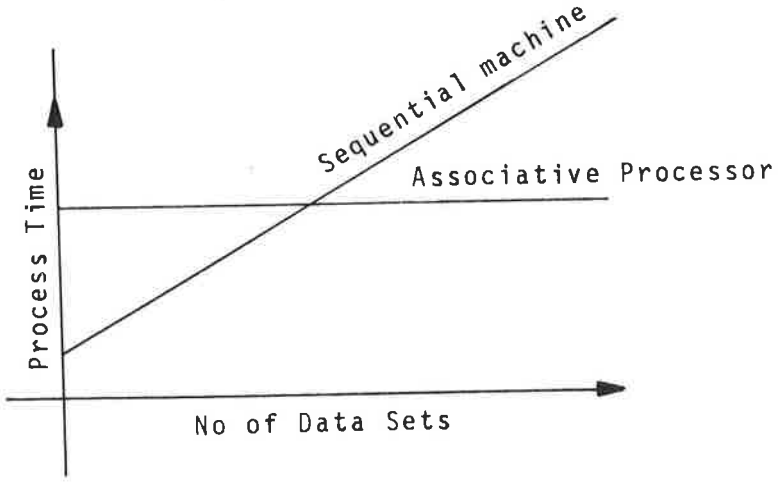



Figure 3-5 Typical Performance Comparison Between Sequential and Associative Processors

Only recently did this technique appear in the general-purpose large-scale computers. Two examples are Control Data Corporation's STAR series and Texas Instrument's ASC computer. The key to any pipeline operation is the ability to formulate the problem and organize the data block in such a way that once the arithmetic function unit is set to perform a specific operation the data is then streaming through the unit as fast as the hardware permits, and the results are stored away as fast as they become available. There are two aspects worth noting. First, the more data that can be pumped through the arithmetic unit without interruption, the higher the system throughput becomes. Second, the speed of the pipeline depends on how fast the memory can deliver and store data and on the processing speed of the arithmetic unit.

To superimpose a pipeline structure on an associative

memory, we need, of course, a high-speed arithmetic unit which takes data from the Key Register. In addition, we need a stepping mechanism which can be achieved by the combined operation of the Match Result Register and the Multiple Match Resolver.

Figure 3-6 shows the block diagram of a pipelined associative processor where a high-speed selection switch is necessary to select proper fields from the Key Register to interface with the arithmetic unit. The stepping mechanism operates as follows: after an equality search instruction, all the words which match with the criteria specified by the instruction are registered in the MRR. If a pipeline arithmetic instruction, e.g., Pipeline Add, is executed next, it will cause the single MMR output to drive the selected word for read-out into the Key Register. Two operands specified by the Pipeline Add instruction will be delivered to the A and B Registers of the arithmetic unit via the high-speed selection switch. After a short delay, the result of the addition is transferred back to the Key Register via the selection switch in an appropriate result field. The MMR output is activated again to drive the same word in the AM array for storing the result. While driving the selected word for writing, the MMR output is used simultaneously to reset the corresponding bit in MRR. Since the input logic of MMR is taken unclocked from MRR, the next "1" bit in MRR is reflected in MMR. This action accomplishes the "step down" and enables the MMR to select the next matched word automatically, which completes a basic cycle of the Pipeline Add instruction. The cycle then repeats itself in AM read-out, operands transfer, add, result transfer and AM write. The cycle (or stepping down) continues unitl there are no more "1" bits left in MRR. In other words, the "0" output of the MMR terminates the streaming action of the Pipeline Add instruction.

For convenience of looping, it is desirable to provide a Match Result Storage Register to temporarily store the contents of MRR. When MRR is cleared by the stepping function, it is possible to restart the stepping cycle again by loading

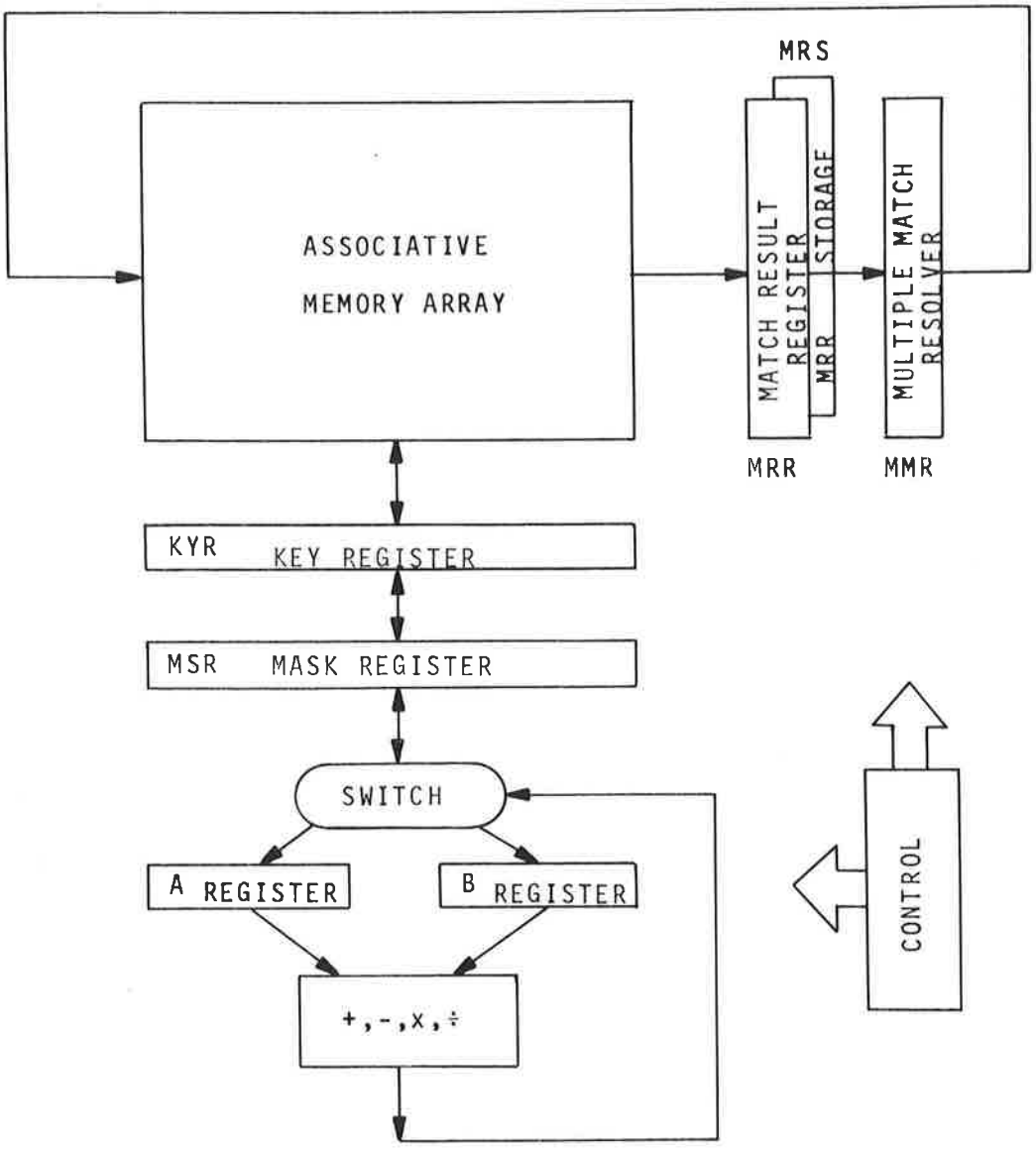



Figure 3-6 Associative Processor With Pipeline AU

MMR by MRS. For further clarity, let us define the two types of instruction formats as follows:

Search Instruction

| Op Code | Field Name | Key |
|---|---|---|

This instruction searches field for "Key" as specified by the operation code. For those words which satisfied the searching criteria, the corresponding positions of MRR and MRS will be set.

Pipeline Arithmetic Instruction

| Op Code | Field 1 | Field 2 | Field 3 |
|---|---|---|---|

This instruction loads MRS into MRR, reads-out the first word, operates on Field 1 against Field 2 as specified by Op Code and stores the results in Field 3. Except for the loading of MRR by MRS, the rest of operations repeat until MMR steps through the entire MRR.

The target azimuth determination of the target detection function involving the computation of equation

$$AC = AT\text{-}f(R - \frac{S}{H})$$

may simply be programmed as follows:

| Op Code | Field 1 | Field 2 | Field 3 | |
|---|---|---|---|---|
| EQS | TARGET | DECLARED | | (EQ and Store MRR) |
| PDV | S | H | Temp | (Pipeline Divide) |
| PSB | R | Temp | Temp | (Pipeline Subtract) |
| PML | F | Temp | Temp | (Pipeline Multiply) |
| PSB | AT | Temp | AC | (Pipeline Subtract) |

The first instruction (EQS) identifies those declared targets. The rest of the pipeline arithmetic instructions carry out the necessary computation steps on all of the identified targets.

## 3.4 SYSTEM VALIDATION CONSIDERATIONS

We have highlighted the special processing characteristics of the basic ATC functions. The bulk of the processing load turns out to be straightforward correlation and computation. These two requirements appear to be very closely matched by the capabilities of the proposed pipeline associative processor where correlations are easily carried out by the associative memory array and repetitive computations are conducted by the pipeline arithmetic unit. The arguments favoring the PAP organization for the ATC applications presented seem to be quite convincing; however, they are not sufficient to authoritatively validate this organization. What it takes for the system validation, then, is a complex simulation effort consisting of several major sub-tasks listed as follows:

- o Construction of a complex and versatile simulator capable of collecting statistics on critical system parameters,
- o System design and coding of ATC functions taking full advantage of the machine structure in parallel and streaming processing,
- o Construction of a complex and flexible scenario generator as the input to the simulation providing most, if not all, possible situations for exhaustive simulation.

The validation process described requires a major level-of effort which is beyond the scope of this study. However, one experiment being conducted at Knoxville, Tennessee by the FAA, Goodyear and UNIVAC can be referenced to partially reinforce the claims made on pipeline associative processors. Some of the preliminary evaluation results are reported as follows:

- o Associative processors and conventional processors can be combined to perform ATC functions in a real-time environment,
- o The overall tracking function including beacon, radar, turning, and altitude trackings performs within acceptable limits,

- The conflict detection algorithms performed by the associative processor appear to be sound, but anomalies do occur when tracks are flying parallel. The problem appears to be residing in the calculation of the track headings rather than in the detection logic,
- Performing the tracking and conflict detection functions at an estimated load of 1000 aircraft, the associative processor is active 61% of the time.

# 4.0 THE ASSOCIATIVE PIPELINE MULTIPROCESSOR ORGANIZATION (APMP)

The processing and computation power of an associative processor equipped with a pipeline arithmetic unit was clearly demonstrated in the previous chapter to be superior to sequential computers for specific classes of applications. However, conventional sequential machines undisputable excel in applications where numerous logical alternatives have to be tried and decisions made. It is apparent that for a large-scale real-time system covering a wide spectrum of data processing and computation activities, a computer with the combined characteristics of associative processor and sequential machines will be highly desirable. In such a system, the sequential machine is an organizer and controller; whereas, the associative processor is a doer, a number cruncher. Each contributes its best talent to achieve an overall optimum system.

This chapter attempts to describe such a computer organization, which, in this author's opinion, shows great potential for satisfying all stringent demands in general, as well as the air traffic control requirements specifically. This computer system, called Associative Pipeline Multi-processor (APMP), employs redundancy on a higher level, i.e., a pool of memory modules shared by a number of arithmetic and logic processor modules and a number of I/O processor modules, whereby all processors can operate independently and simultaneously.

## 4.1 SYSTEM ORGANIZATION AND MAJOR FEATURES

This chapter attempts to define the APMP system, delineate the system's functions and achieve the increased performance, flexibility and reliability that it claims to be capable of doing. Emphasis will be placed on those major features which differentiate this system from other multi-processors[1,12,39,40,41]. Design approaches to realize these features, as well as the rationales used, will also be explained.

### 4.1.1 Associative Pipeline Multiprocessor Structure

A multiprocessor system is defined, in general, as a number of processors sharing a number of commonly accessible memories by some means of interconnection between them. In the associative multiprocessor under discussion, we shall restrict it to a narrower definition as a number of identical arithmetic and logic processors and a number of identical input-output processors sharing a number of commonly accessible memory modules interconnected by a crossbar switch network illustrated by the block diagram in Figure 4-1. Note that the identical crossbar switch network is used to establish the communications between arithmetic/logic processors (ALP) and I/O processors (IOP). Two types of memory modules are employed: the conventional random access memory (RAM), and the associative memory or content addressable memory (CAM) with a pipeline arithmetic unit.

The A/L processors are identical and independent which means that they can execute independent tasks simultaneously. The I/O processors also perform data transfers and peripheral control functions independently and simultaneously. All processors may access memory modules simultaneously provided that no conflicts are present. Should conflicts occur, a conflict resolution logic built into the crossbar switch network, would determine the winner for the access. The reason why a crossbar switch network is selected is its design simplicity and modularity. This characteristic of the switch network provides easy additions and deletions of various modules. In essence, one can tailor the system according to the computation and data processing requirements by providing the appropriate number of processors and memory modules. The idea of restricting modules of the same type to be identical is to allow these modules to be interchangeable so that replacing modules are readily available if faulty modules are found.

In short, independent and simultaneous processor operations provide multiprocessing capability of independent tasks (these tasks may be parts of one job). The parallel search capability

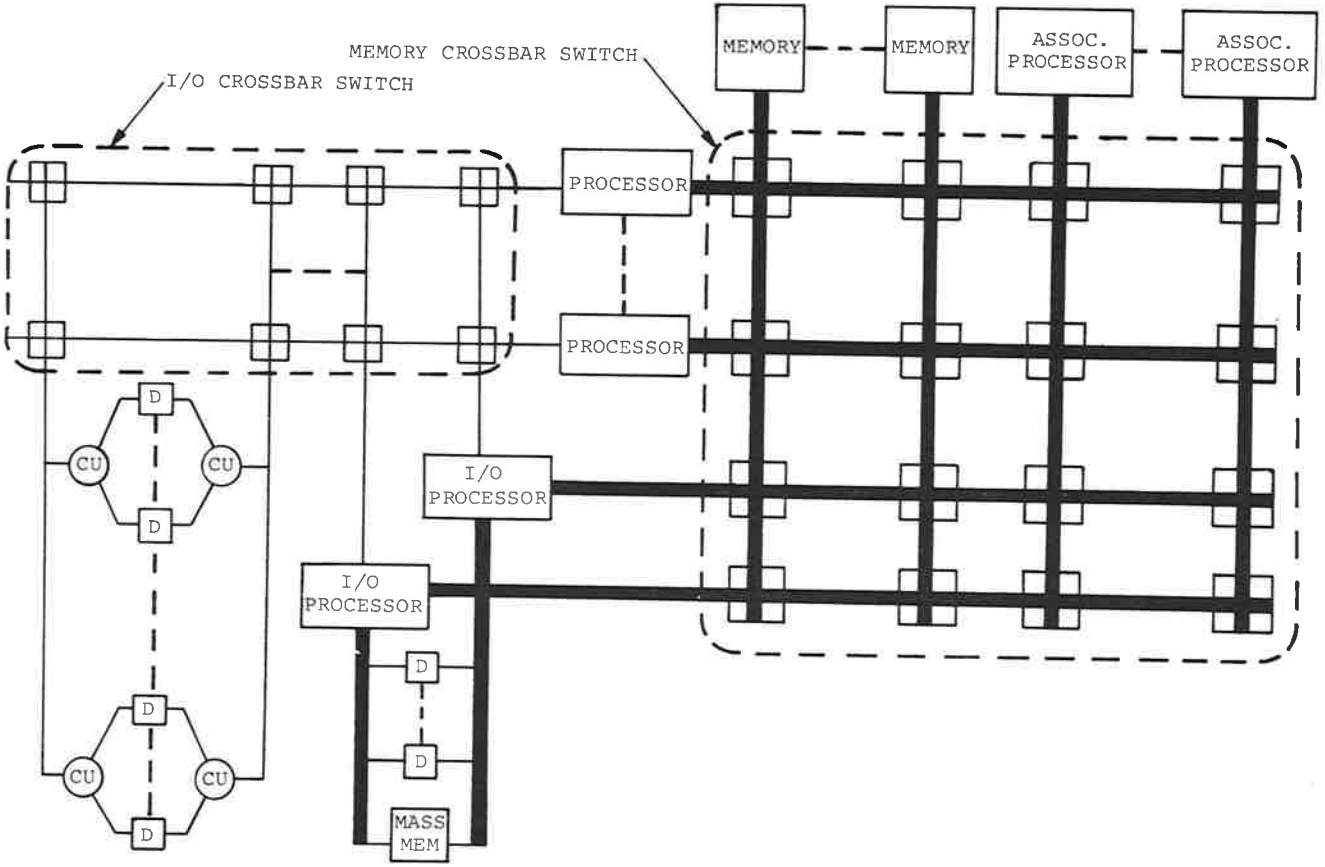



Figure 4-1 The Associative Pipeline Multiprocessor Organization

of an associative memory greatly enhances the system's overall performance. Modularity of the switch network provides easy system expansion which meets the flexibility requirement. Active redundancy applied on the major functional units level and TMR (Triple Modular Redundancy) voting applied on the system "hard core" crossbar switch network achieve the objective.

### 4.1.2 Autonomous Executive Control

In a multiprocessing environment, an executive control is essential to schedule all the processes, assign computer resources to these tasks, control the execution of programs, manage unexpected events, etc. There are several known alternatives for scheduling and assigning tasks to a multi-processor system. One such alternative is to establish one of the A/L processors as a master and all the rest as slaves. Such a scheme is feasible but inefficient, particularly when the system has a light load and the master processor (may be other processors as well) has long idling periods. Another technique suitable when the set of problems is static, is to use linear or dynamic programming techniques to establish the optimum assignment. Where the environment is dynamic with new problems being introduced at random, or where the system admits changing priority, these techniques may prove too costly and time consuming. Still another method is not to schedule at all but rather to assign problems on a first come first serve basis. When there is no deadline to meet, such a solution seems quite adequate.

However, none of these three alternatives can meet the real-time system requirement; hence, an autonomous executive concept is adopted whereby static tasks will be scheduled on a priority basis and dynamic requests will be handled via an interrupt system and serviced on a deadline basis. The early model of this concept was shown in Burrough's B-825 computer. Later, Pariser[42] and Gunderson[43] proposed ways of implementing a "floating executive". Autonomous executive concept is a refinement of those early models, where a simple flexible interrupt handling mechanism is incorporated. The autonomous

executive is considered to be dormant in the memory with its associate data, e.g., memory maps, assignment tables, etc. An A/L processor servicing a task is said to be in the problem state. When it finishes its present task, the processor changes its status into executive state where the autonomous executive routine (AER) updates the task assignment table and searches for the next task to be processed. AER locates the new task in memory, records the assignment, transfers control back to the problem state and starts processing the new task. In other words, the system adopts the convention whereby each physical processor takes its own initiative to request for executive control at the end of each task, to record the status of the completed task, and to find a new task for itself. This basic autonomous executive concept is no different from other multiprocessor executives. The difference lies in the interrupt handling techniques, and the use of associative memory for system table storage which will be discussed in a later section and in Section 6 respectively.

Since all A/L processors are operating independently and asynchronously, it is conceivable that more than one processor may request for executive control at the same time. Actually, the conflict presents no threat to the system since it will be resolved by the crossbar switch network automatically, provided that the first instruction the processor uses to transfer into executive state (i.e., supervisor call instruction as in some computers) has the "test and set" capability all done in one memory cycle. Supervisor Call instruction with Test and Set memory capability serves as an interlocking mechanism so that when a processor has successfully obtained executive status, it records a mark in a specific memory bit position to prevent other processors from seizing executive control. The interlock is essential to guarantee that no two processors can be in an executive state at the same time.

### 4.1.3 Memory-Oriented Logic Design

This system's feature is mainly a hardware consideration

which is influenced by the advances in the state-of-the-art circuit technology, LSI (Large Scale Integration). For reliability and producibility reasons, LSI technology favors regular, repetitive design of logic networks. A brief review of the internal structure of an A/L processor reveals that the structure of the arithmetic unit has a regular pattern whereas the control unit usually consists of various irregular gates, registers and counters randomly distributed throughout the processor. The memory-oriented design concept proposes to substitute the random gates by microprogrammed control memory of appropriate size and speed, registers and counters by small scratchpad memory (or memories).

There are four major advantages: (1) The design process is simplified, and the hardware checkout phase is reduced; (2) Microprogram control techniques transform hardware control logic design into a programming technique which enables logic designers to try out various schemes before finalizing the design; (3) Centralizing the control logic into a control memory, facilitates easy change of control logic via either re-programming or replacing the memory unit; and (4) Error detection schemes are easy to implement on such a system, e.g., residue number system for the checkout of arithmetic unit and simple parity scheme for the checkout of all the memory units. There are disadvantages of course. In general, microprogramming techniques are too costly for small, low-performance computers, and too slow for large, high-performance machines. However, both these disadvantages can be circumvented in the context of multiprocessing.

## 4.2 DESIGN CONSIDERATIONS OF KEY FUNCTIONAL ELEMENTS

In this section, we attempt to describe in some detail the structural and operational features of some key elements in the system. The detailed computer architectural design, such as word length, instruction format, addressing schemes, etc., is beyond the scope of this study and hence will not concern us here. Only those features that make this APMP

system outstandingly different from other systems will be highlighted. Hence, considerable attention is then given to the description of crossbar switch network, associative memory module and interrupt management logic.

There is no special requirement on a random access memory module to operate in a multiprocessor environment. A conventional memory module would be adequate if it is a self-contained system. By this we mean that a memory module has its own address register, data register, parity generator/checker and module number assignment register. Most of the memory systems, even the off-the-shelf memories, do contain all the items listed except the last one. This register containing the memory module number is needed for system reconfiguration purposes. This way the physical module number is divorced from the logical module number so that system software may assign or reassign any number to a memory module in case of reconfiguration due to memory module failure.

However, the memory module assignment register does influence the design and operation of the crossbar switch network control logic. The control logic, instead of responding to a single request line from a processor, has to incorporate a comparison circuit to compare a high-order position of an address field from a processor with the contents of a memory module assignment register. An equivalent match together with the processor request is the effective request signal to be evaluated by the conflict resolution logic. This is diagrammatically illustrated in Figure 4-2.

As noted previously, any A/L and I/O processor with a reasonably powerful instruction set can be structured into a multiprocessor organization. However, we would like to stress a design concept, "Regularity and Simplicity", to take great advantages of recent advancements in LSI technology. In traditional logic design concept, designers are highly motivated to save hardware and speed-up logical and arithmetic operations as fast as possible. This makes the system very difficult to check out and modify. With the advent of LSI, although hardware

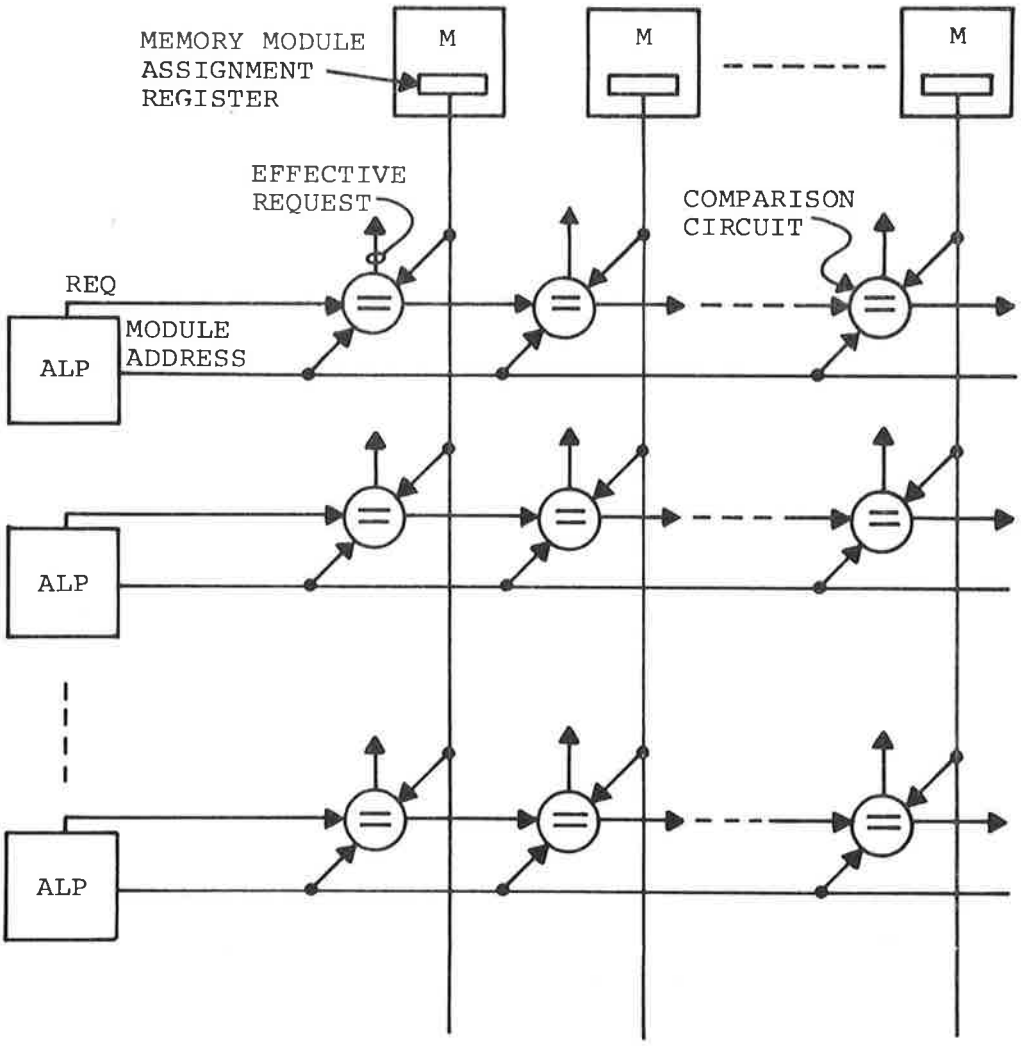



Figure 4-2 Generation of Effective Memory Request

saving and high-speed operation are still important design considerations, they are no longer the dominating factors. "Regularity and Simplicity" design concept on the other hand simplified the design and check-out processes, and sped up the design and implementation cycle. The slight loss in operating speed can be off-set by the overall increased systems performance of an associative multiprocessor system.

### 4.2.1 Crossbar Switch Network

The switching network is the vital link of the multiprocessor system as evidenced in Figure 4-1. It provides data connections and control interfaces among processor and memory modules. There are basically two types of switch networks which are realizable with the state-of-the-art technology. They are time division (multiplexing) and space-division (crossbar) techniques. The time-division scheme requires less hardware, and is relatively easy to control, but the system is quite rigid and less efficient since only one processor-memory pair can be connected at one time. On the other hand, the space-division which is capable of providing simultaneous connections of processor-memory pairs except in conflict situations, offers tremendous flexibility and efficiency. The disadvantages of this scheme seem to be the high hardware cost and complex control logic which increases with the number of processor and memory modules. Since the advent of LSI technology and its rapid progress, the hardware cost of a computer no longer influences the overall system cost a great deal. A sample construction of the crossbar switch network, as illustrated in the next paragraph, is a regular planar array of identical elements which is a perfect application for LSI fabrication.

In this paragraph, a simple design of the control logic is demonstrated for the versatile switching matrix. The logic of the switching matrix can be considered as having two parts, one for the data and the other for the control. The data paths are relatively simple as shown in Figure 4-3. At each inter-

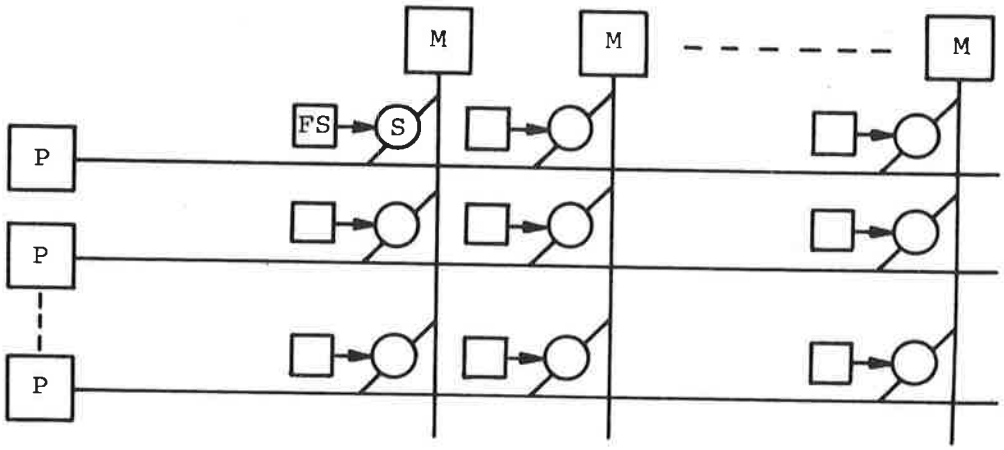



Figure 4-3 Crossbar Switch Network - Data Paths

section of the matrix there is a group of switches. These are represented by a single circle, S and controlled by a single flip-flop, FS. A flip-flop, being set or reset, closes or opens the corresponding switches which connect or disconnect the transmission path between a processor-memory pair. The control logic as to when to set which flip-flop is dependent on three logical entities, i.e., the status of a processor (requesting a memory or not), the status of the requested memory (busy or available) and the status of possible conflict requests by other processors. One could devise many methods to resolve the conflict from the competing processors to a single memory as demonstrated by Plummer[43]. However, none could yield a higher degree of simplicity and clarity in design and system flexibility than straightforward "round-robin" scheme.

The round-robin scheme simply states that after an access is made to memory M by processor P, the priority "baton" of M is passed on to the next processor in a predetermined sequence.

At any given time, if no request is issued to memory M by higher priority processors, the low priority processor will be able to access M, then shift priority to his next-door neighbor. Figure 4-4 illustrates one version of the detailed logic design, in which another flip-flop FP is added to each crosspoint to register processor priority for a particular memory. In Figure 4-4, only one column (corresponding to one memory) of control logic is shown since it is identical to all memories. In an environment of 4 processors by n memories, flip-flop FP21 being set denotes that with respect to memory M1 processor P2 has the highest priority, P3 next, P4 third, and P1 last. In a given column, only one FP flip-flop can be set at a time. The system is synchronized on clock pulse basis. By this, we mean that at every clock time, the effective requests from processors are examined, conflicts resolved and connections made between a memory and the winning processor.

Briefly tracing through the logic at this point may be helpful. Assume that at some point in time FP 21 is set, memory M1 is not busy and only P3 and P4 are requesting M1 simultaneously. Since P2 is not requesting, the priority enable line is passed by FP21 through P2 down to P3. At the end of the clock period, the priority enable signal and the P3 request set both FP41 and FS31 at the same time resetting FP21. Setting FS31 accomplishes interconnection between P3 and M1 while setting FP41 means that the priority baton has been passed on to P4. FS31 also starts up the memory operation and puts it in a busy state; in addition, it sends an acknowledge signal back to the requesting processor. At the end of the memory cycle, an end-of-cycle signal is activated by the memory which clears the switch flip-flop FS31 to zero and makes the memory available for further access. This time P4 is guaranteed to have the access to M1. Note that requests are examined and priority is passed on a clock pulse basis so that no single processor can tie up a memory module for more than one memory cycle at a time.

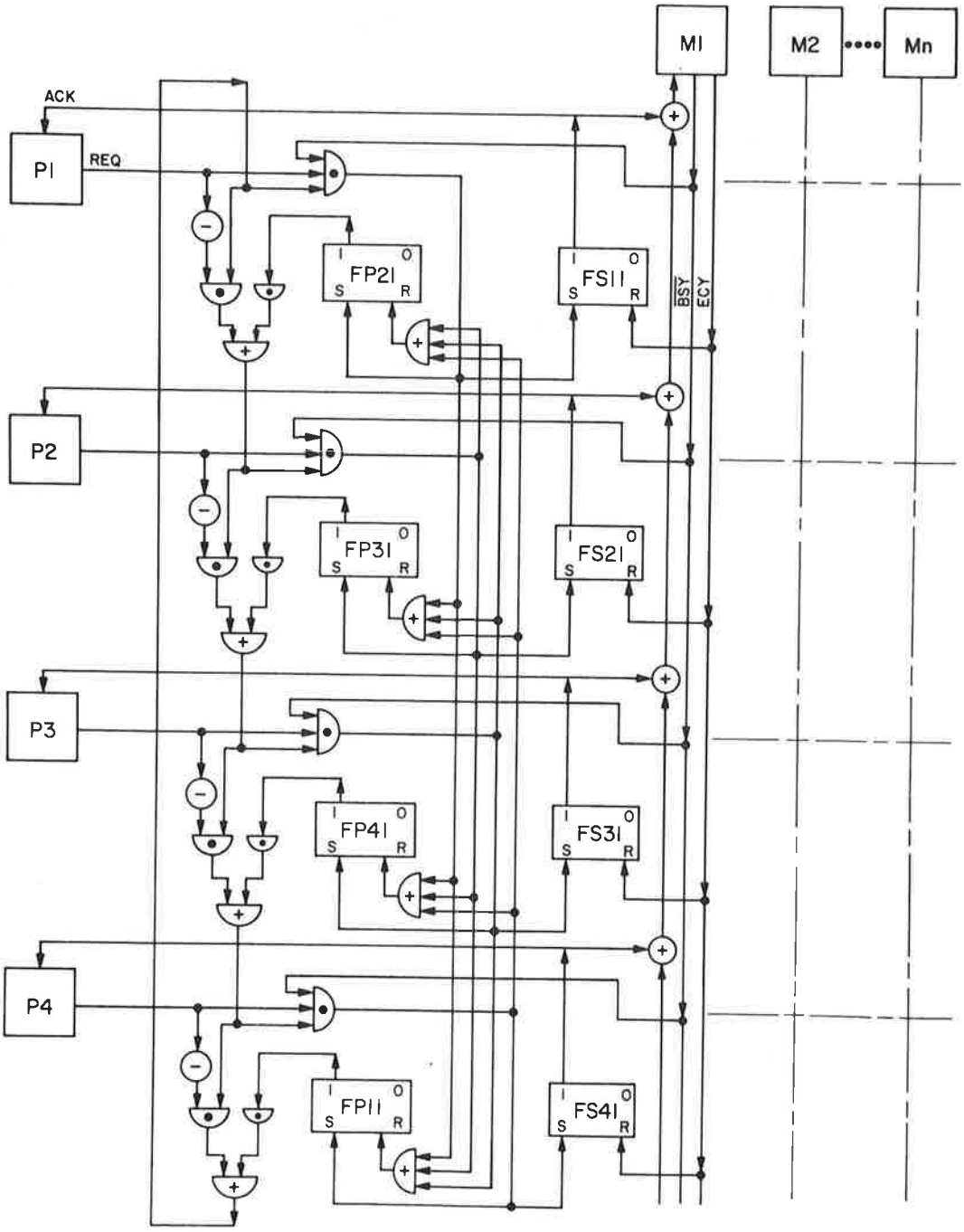



Figure 4-4 Crossbar Switch Network - Control Logic

### 4.2.2 Interrupt Management

Interrupt schemes in a conventional computer system are quite well understood. In a multiprocessor environment, however, the interrupt handling problem becomes more complex particularly when a system consists of identical processors each of which can be an executive autonomously. This is simply due to the fact that when an interrupt occurs what processor should be selected to take care of it. Before proposing any scheme, let us examine the characteristics of all interrupts. Basically, interrupts can be distinguished into four categories:

1. Program generated traps, e.g., sense switches time out clocks, etc.
2. Program errors, e.g., illegal op code, overflows, etc.
3. Machine errors, e.g., parity check, arithmetic checks etc.
4. External interrupts, e.g., I/O completion, terminal service requests, etc.

The first two types closely associate with a specific processor which should handle these interrupts directly. Machine error, however, can be considered either internal to a processor, e.g., arithmetic check or external, e.g., parity error during I/O transfer. The fourth type is completely external to ALP processors. Interrupts caused by internal machine errors should be handled as an external interrupt.

Interrupt handling presents common problems in different system types. An excellent summary of interrupt features and problems has been presented by Borgeus[44] and supplemented by Bennet[45] for single computer systems. A multiprocessor has the same problems plus additional ones arising mainly from having more than one processor in the system.

The external interrupts in a multiprocessor environment are characterized by occurences which are independent of jobs in execution. The occurrence of this type of interrupt has no dependency upon a specific processor but is related only to the

job which generated some I/O action which in turn generated the interrupt. Essentially, the external interrupt is a signal to either reactivate an old process or start a new one. However, it is not necessary to capture a busy processor to handle an external interrupt unless: (1) there are no idling processors to take it, and (2) the external interrupt is considered of higher priority than some of the jobs currently being processed by all the processors. What we are aiming for is to achieve an optimum interrupt response system by which all the processors in the multiprocessor system always execute jobs with highest priorities at all times. Goutains and Visc[46] suggested that interrupts be handled by a separate Interrupt Directory which assigns the most urgent interrupt to the processor executing the task with lowest priority. The Directory could be implemented by hardware consisting of two sets of selection circuit, two sets of priority evaluation logic an a comparator control logic.

We propose an alternate hardware approach to this problem. The unit is a simple iterative network applied to all I/O processors and central processors. Iterative circuit implies repetitive regular logic pattern which again lends itself for easy implementation by LSI. The construction is extremely simple. Figure 4-5 shows the network diagram as it goes from one processor to the next, whereas Figure 4-6 details the logic within each iterative element. In brief, each IOP collects and evaluates its own internal interrupts from those I/O devices connected to this particular IOP. Associated with each I/O operation, hence each active I/O device, there is a priority or urgency index which is equivalent to the priority of the job that initiated this I/O action. An IOP evaluates all interrupts with respect to their priorities and presents the interrupt with respect to their priorities and presents the interrupt with the highest priority as the input to the iteractive circuit to compete with other high priority interrupts from other I/O processors. The network, as indicated in Figure 4-5, arbitrarily feeds each IOP serially from left to right and each ALP from top down.

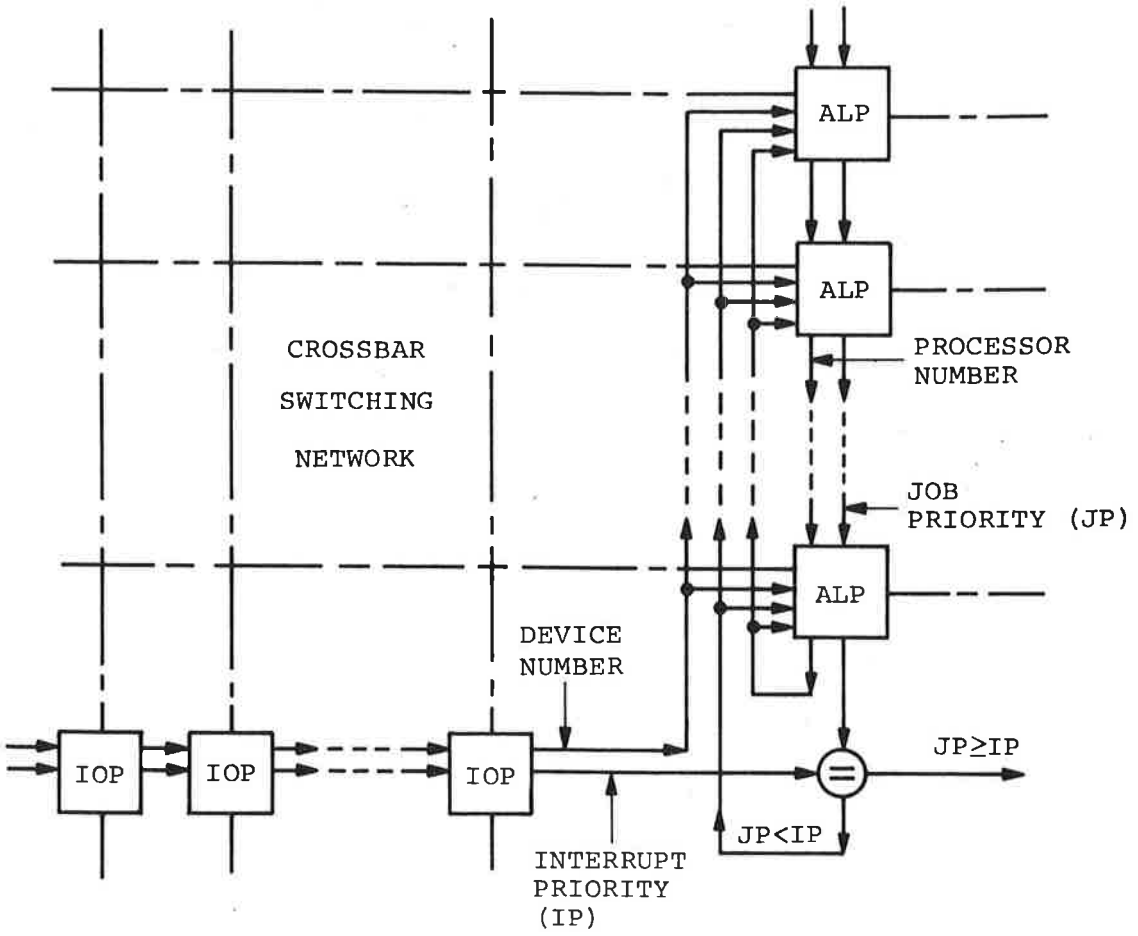



Figure 4-5 Interrupt Management Network

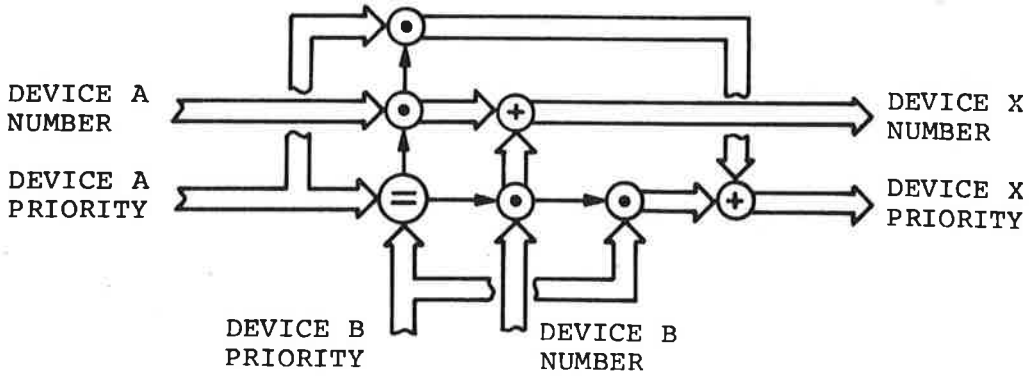



Figure 4-6 Iterative Comparision Logic

The essence is to pass these priorities into a series of comparisons so that the highest priority interrupt (IP) is trying to identify the lowest priority processor (JP). If IP$\leq$ JP, no interruption has taken place, and the interrupt simply has to wait. If, however, IP>JP, the processor number is fed back to the ALP for the initiation of an interrupt in addition to the I/O device number and its priority. Figure 4-6 illustrates in a symbolic way the simple logic in detail which consists of one 4-bit comparison circuit plus four AND gates and two OR gates. Using present $T^2L$ logic and a 4-IOP X 4-ALP system, within 500 ns from the first activation of an interrupt, a processor will be selected.

# 5.0 MULTIPROCESSOR CONTROL PHILOSOPHY

In Section 3, we have conceived a novel computer architecture to meet the parallel processing requirements imposed by the basic air traffic control functions. To further increase its power and system reliability, we let the Pipelined Associative Processor be driven by a generalized multiprocessor structure as described in Section 4. However, finding techniques for efficient control of the multiprocessor system with respect to the optimization of selected parameters is still an ill-understood problem. In this chapter, we attempt to survey the state-of-the-art, to study some scheduling techniques, to delineate some of the practical problems not yet covered by previous researchers, and to propose some feasible solutions.

## 5.1 GENERAL BACKGROUND SURVEY

There are numerous papers published discussing, in general terms, the characteristics and operations of multiprocessors. Two of them present good tutorial material. Witt[47] explains the multiprogramming and multiprocessing concepts by an excellent allegory. He considers a secretary working in a "multiprogrammed environment". When two or more secretaries are working in a pool, they are essentially working in a "multiprocessing environment". He further noted some of the special problem areas in multiprocessing which are related to certain forms of communications between the secretaries. Problems arise when, for example, two secretaries complete their present assignment at once and reach for the highest-priority work in the job queue at the same time, or when a secretary is rearranging the priority stack when the other desires the access to the job queue. Obviously, some kind of lock-out scheme is necessary. Critchlow[48] produced a comprehensive survey on multiprocessing and multiprogramming systems. It began with some background history of the growth of multiprocessing, continued with the detailed description of the memory access control, I/O switching

control and the priority interrupt control. It contains a good discussion on the supervisory control with further breakdown into memory allocator, scheduler, and I/O control. A good contribution is the discussion on some of the hardward aids to multiprogramming and multiprocessing, such as memory protection schemes using limit registers, hardware lockout, memory relocation schemes using base registers, page turning hardware, etc.

In the early 60's the U.S. Naval Research Laboratory and the Burroughs Corporation studied the computation requirement for a real-time, command and control system, which let to the development of the Burroughs' D-825 modular computer system. The overall architecture of the D-825 system was reported by both Anderson, et. al.[1] and Wald[2]. The executive program, called Automatic Operating and Scheduling Program (AOSP) has two important features: executive function is independent of processors, and more than one processor may execute AOSP simultaneously, each evoking programmed lockouts when modification of system table is required. In general, AOSP affects parallel processing of independent jobs automatically. These systems and software aspects are reported by Thompson, et.al[49].

In the mid-60's, the Federal Aviation Administration (FAA) embarked on a complex developmental program for increasing the capabilities and safety of its air traffic control system. Referred to as the National Airspace System (NAS), the program is evolving around a general-purpose, modular multiprocessor system, IBM 9020 consisting of 3 computing elements and 9 storage elements. The system organization and characteristics, reported by Blakeney, et. al[12], are highlighted by the configuration control scheme. Each element (computing element, storage element, etc.) contains a configuration control register (CCR). Set by special supervisor instructions, these CCR's set up the communication and control paths among various elements. Two methods are used for storage interlocking. When common tables or non-reentrant subroutines are used by all processing elements, TEST AND SET instruction provides a rapid way of determining the availabliity of such facilities. When two or more processing

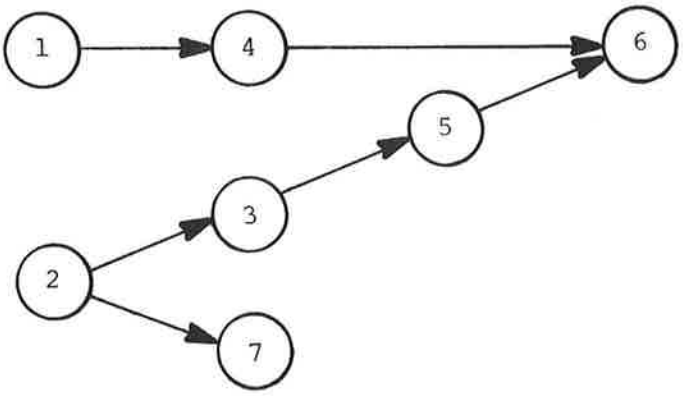
elements request common storage elements, a tie-breaking hardware priority-circuit in the storage element resolves the conflict.

The control program (or Executive routine) design is based on the concept that any computing element may execute control program according to the needs. Devereaux[13] gave a good description on the main features of the control program. Dynamic storage allocation is implemented by the supervisor call instruction, and storage resources in a storage element are classified into areas, blocks, and lines. The control program ensures that storage requests are made in a consistent manner, thereby guarding against the possibility of mutual "dead-lock". To avoid system dead-lock, lines are requested before blocks, which in turn must be requested before areas. If a particular type of resource is allocated to a program, an additional storage resource of the same type cannot be requested from the program until all of that type of storage is appropriately released.

## 5.2 MULTIPROCESSOR SCHEDULING TECHNIQUES

Scheduling independent processes on independent processors has created a great deal of interest and challenge; many researchers have made valuable contributions. In order to conduct parallel processing (we shall use "parallel processing" and "multiprocessing" interchangeably), we must be able to determine and organize parallel processable tasks. To this end, Ramamoorthy and Gonzales[50] conducted a comprehensive survey of techniques for recognizing parallel processable streams in computer programs, and in addition, proposed a new technique for detecting task parallelism. Other works including Hellerman[51], Stone[52] and Squire[53] are primarily concerned with detection parallelism within arithmetic expressions.

Ochsner[54], based on a task precedence structure, proposes a scheduling mechanization method by means of a universal list. Figure 5-1 shows a sample task precedence structure and an associated universal list with which these tasks are sequenced.




ABSOLUTE ENABLE BITS

CONDITIONAL ENABLE BITS

| A | C1 | C2 | SUCCESSOR TASKS | STARTING ADDRESSES |
|---|---|---|---|---|
| 1 | 1 | 1 | TASK 4<br>C1 | TASK 1 |
| 1 | 1 | 1 | TASK 3 7<br>C1, C2 | TASK 2 |
| 0 | 0 | 1 | TASK 5<br>C1 | TASK 3 |
| 0 | 0 | 1 | TASK 6<br>C1 | TASK 4 |
| 0 | 0 | 1 | TASK 6<br>C2 | TASK 5 |
| 0 | 0 | 0 | — | TASK 6 |
| 0 | 0 | 1 | — | TASK 7 |
| | | | | |
| | | | | |
| 1 | 1 | 1 | | |

Figure 5-1 Sample Program and Universal List Structure

The algorithm directs a free processor to scan the list from the top down until it reaches the first task in the ready-to-be executed state as indicated by the "absolute enable bit". In addition, each task has a set of "conditional enable bits" each one corresponding to a particular event, such as completion of a preceding task, or arrival of a real-time clock signal. The absolute bit is set to binary 1 when all conditional enable bits are set to binary 1's, which means all conditions are met and the task is ready to be executed.

Martin and Estrin[55,56], have investigated problems in the efficient operation of the Fixed-Plus-Variable Structure Computer developed at UCLA. In particular, they have studied the a priori assignment and sequencing aspect by means of both analytical and simulation modeling. The study uses a graph model in which vertices represent segments of computation in a program with estimates of execution time. Directed arcs between vertices establish precedence conditions in the organization of program segments. The function to minimize was the mean path length of the scheduled graph. The scheduling process is iterative in such a way that a trial change of assignment of a vertix to a processor is accepted only if a reduction is expected in path length.

Ramamoorthy et. al[57] conducted an extensive formal study of the scheduling problem. Based on a directed graph representation of task structure, a graph can be reduced by transforming maximal strongly connected (MSC) subgraphs into single vertices (getting rid of loops). From the reduced graph, two precedence partitions: earliest (E) and latest (L), can be generated. Thus, $E_i$ is the subset of tasks that can be processed in parallel at the earliest time corresponding to level i. Next, the upper and lower bounds of the number of processors needed to do the job in the least possible time can be obtained from these E and L partitions. Then a dominance relation between tasks is defined, and an important theorem proved which says if task i dominates task j, then there exists an optimal solution in which task i is started before or at the same time as task j. Based

on the E, L partitions and the dominance relation, three algorithms have developed such that given a task graph, the minimum number of processors required to process the graph in the smallest time can be determined.

Muntz and Coffman[58] have also investigated this very same problem but took a different approach. Two principal notions are introduced, pre-emptive scheduling (PS) where processors are allowed to be interrupted before a task is completed and reassigned to a new task, and general scheduling (GS) where any fraction of the whole capability of a processor can be abstractly assigned to tasks. For example, if 1/2 of a machine is assigned to a task which normally requires W time units to complete, it then will take 2W units of time. Then, they proved the equivalence of the GS and PS disciplines, which amounts to saying that time sharing and processor sharing are equivalent. Finally, they introduced an algorithm for constructing a minimal-length GS for any computation whose graph is a rooted tree, i.e., each node has one immediate successor except for a unique root node which has no successors.

## 5.3 MULTIPROCESSOR TIMING ANOMALIES

As early as 1960, Richards[59] in his study of an abstract machine with many identical work units (CPU's) and a single job queue capable of sharing a large memory, discovered some very interesting peculiarities of the system. Even though the system operates under a rather natural and reasonable set of rules, the system may exhibit certain somewhat unexpected anomalies. For example, it can happen that increasing the number of processors can increase the length of time required to execute a given set of tasks.

Later, Graham[60] gave a rigorous treatment on the basis of mathematical analysis and found theoretical bounds on these timing anomalies. The model used assumes n identical abstract processing units $P_i$, i = 1,2,---n, and a set of r tasks $T_j$, j = 1,2,---r. Also given is a partial ordering < on T and

a function $\mu:T\rightarrow(0,\infty)$. Once a processor $P_i$ begins to execute a task $T_j$, it works without interruption until the completion of that task taking $\mu(T_j)$ units of time. The partial ordering represents the precedence structure of the task set so that $T_i<T_j$, then $T_j$ cannot be started until $T_i$ has been completed. Finally, there is a sequence $L = (T_{s1}, T_{s2},---T_{sr})$ consisting of all the tasks of T and called a priority list. Initially, at time 0, all the processors simultaneously scan the list L from the beginning searching for tasks $T_j$ which are ready to be executed, i.e., which have no predecessor under < ordering. If two or more processors attempt to execute several tasks at the same time, the convention will assign the first ready task to the processor with the smallest index, the next ready task to the processor with the next smallest index and so on. In general, at any time a processor $P_i$ completes a task, it immediately scans L for the first available ready task to execute. If there are currently no such tasks, then $P_i$ becomes idle (or executes an empty task $\emptyset$). $P_i$ remains idle until some other processor completes its task, at which time all available processors including $P_i$ immediately scan L for ready tasks. In order to appreciate the timing anomaly problems, some examples would be helpful. The partial order < on T and the function $\mu$ can be represented by directed graph $G(<,\mu)$ whose vertices correspond to the $T_i$ and a directed edge (or arc) from $T_i$ to $T_j$ denotes $T_i<T_j$. The vertices are labeled with name $T_i$ and the execution time units $\mu(T_i)$. The activity of each $P_i$ is represented by a timing diagram.

Note that in the timing diagram the intervals are labeled by task names and their corresponding $\mu$'s. W is the length of time required to complete the set of tasks. In the following examples, variations in each of the four parameters are made and the effect these variations have on W are seen.

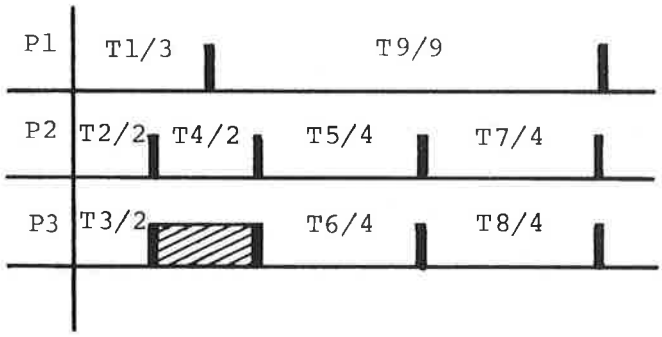
CASE 1:

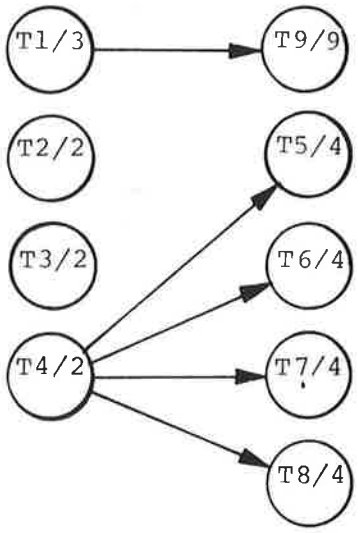
n = 3

L = $(T_1, T_2, T_3, ---, T_9)$

G(<,μ)

W = 12

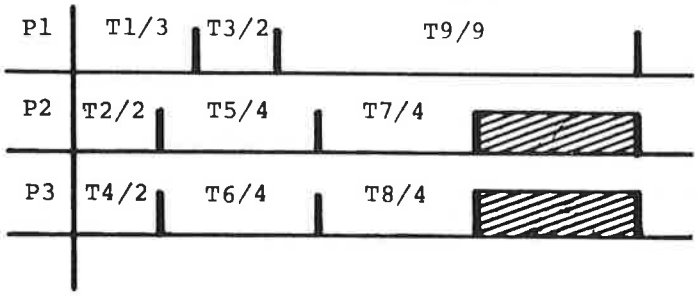



CASE 2:

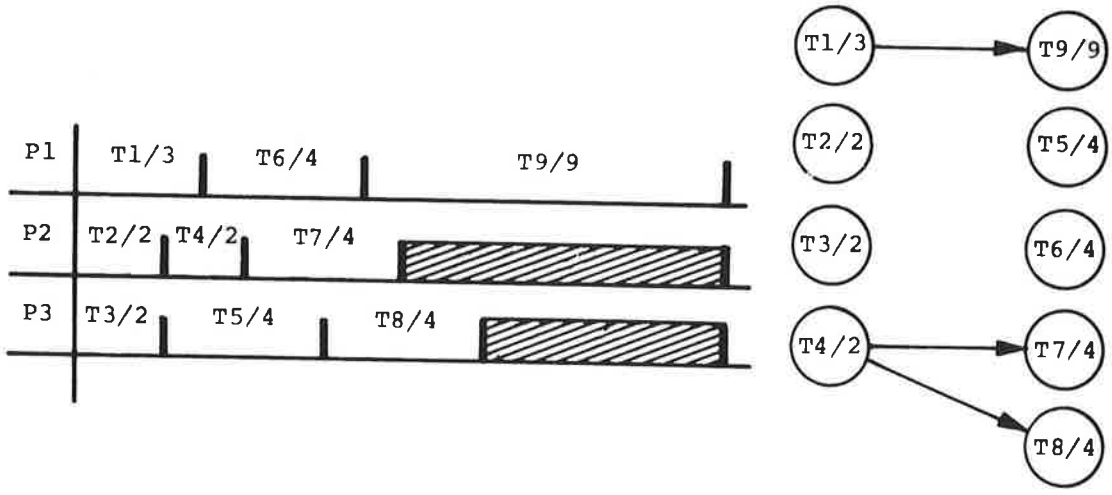
Replacing L by L' = $(T_1, T_2, T_4, T_5, T_6, T_3, T_9, T_7, T_8)$

W = 14



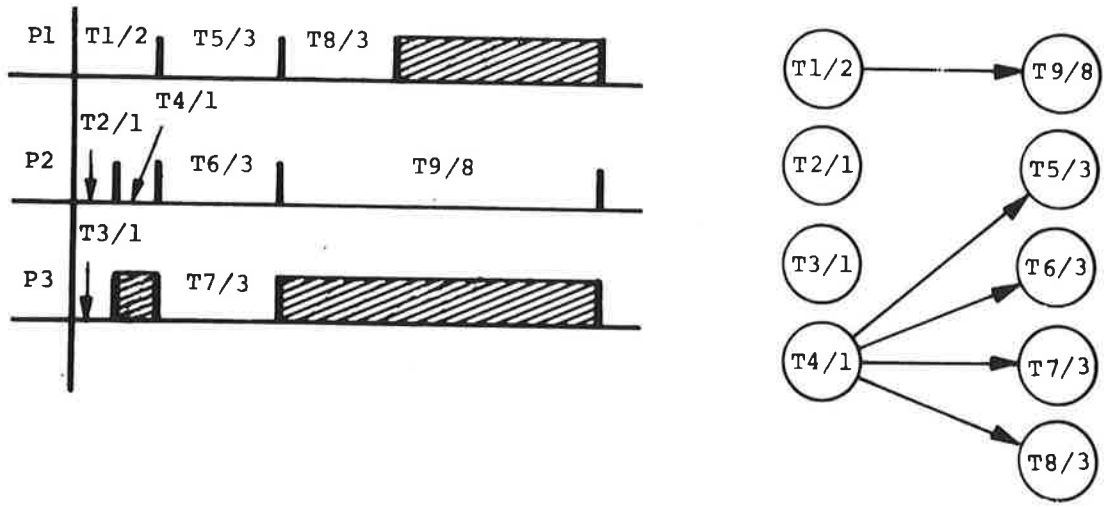
CASE 3: Replacing G by G'

W = 16



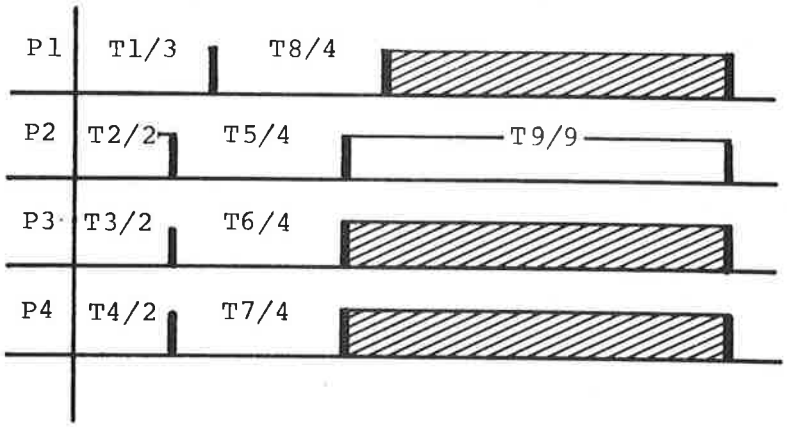

CASE 4: Decrease μ to μ' by 1 (μ'=μ-1)

W = 13



CASE 5: Increase n to n' by 1 (n' = n+1)

W = 15




The examples show that contrary to what might be generally expected, they can all cause W to increase. Furthermore, Graham gave a rigorous mathematical proof that given two runs R and R' where R' is related to R by the application of any combination of the four conditions above, the total run-times W and W' are related to one another by the following bound:

$$\frac{W'}{W} \leq 1 + \frac{n-1}{n'}$$

where n, n' are the number of processors in runs R and R', respectively. When the number of processors remain the same, this relation reduces to a special case,

$$\frac{W'}{W} \leq 2 - \frac{1}{n}$$

Addressing the problem of timing anomalies as noted by Richards and Graham, Manacher[61] uses a modified form of Ochsner's task assignment scheme to eliminate these timing anomalies, which he refers to as the "stablization of task schedules". A task set bounded by the precedence relation, has

a priority list associated with it and is represented by a program flow diagram (similar to a directed graph). Assuming all tasks are running at their maximum run-time, a timing diagram is constructed according to the scanning rules of free processors as directed by the priority list. This timing diagram is called the Standard Gantt Chart (SGC). In addition, the projective task list (PL) is defined as a list of all tasks ordered according to the time each task is picked up on the SGC. A stability algorithm is established by introducing additional precedence relations between independent tasks such that long critical paths of the task set will not be broken in by other unrelated tasks making those long paths even longer. Both a mathematical proof of the stability algorithm and a simulation report are included in this paper. Construction of relatively efficient SGC's by heuristic methods is also proposed.

## 5.4 SOME PRACTICAL AND CHALLENGING PROBLEMS

The literature surveyed thus far cannot be considered exhaustive; nevertheless, it brings out some perspective of the state-of-the-art of the multiprocessor technology. It does not seem to be difficult to physically construct such a system at present; however, controlling such a monster for efficient operation still remains much of an art. The multiprocessor timing anomalies under a set of seemingly reasonable operating rules, reveal a very interesting and challenging problem which may hold the key to the efficient operation of a multiprocessor system.

Let us re-examine the time anomaly study presented previously a little more deeply. In Graham's model, the role of the priority list, L, is not clearly defined, especially the relationship between the task graph (precedence structure) and L. However, in all examples, the priority list is consistent with the graph structure, which would not present any problem in our analysis. Using Case 1 as a base, let us focus on Cases 4 and 5. In Case 4, when the execution time of all tasks is reduced, the overall program execution time (or path length of

the directed graph) is, on the contrary, increased. Similar anomaly occurs in Case 5. The overall program execution time increases when one more processor is brought into the system. In practice, the task execution time can never be held constant since program loops, conditional branches, etc. are contained within tasks. In an operational multiprocessor environment, to a lesser degree, the number of processors in the system may change due to processor failure, maintenance schedules, etc.

The second important practical problem that theoretical modeling may find difficult to do is the memory allocation problem. This should not be confused with the general memory allocation problem in multiprogramming, but rather concerned with the allocation of independent tasks into separate memory modules in order to reduce potential memory conflict. When two or more processors try to execute independent tasks which happen to be located in a memory module, the accesses will be granted one-by-one in a cyclic fashion by system hardware. If these processors persistently try to access the same memory module, they all will experience time delays. The amount of delay is a function of the number of competing processors and the number of times that each processor would want to access the common memory. A very revealing case study was reported by Holland[62] on the memory conflict problem of the FAA-9020 multiprocessor system. Based on the system characteristics of the 9020 computer and its applications, a memory interference model was built in which computer programs were represented by a statistical distribution of instruction operating times which in turn were based on the instruction mix of the application program planned for carrying out ATC functions. This model was used to determine the "stretchout factor" or extension of program execution time that occurs because of the memory access delays due to competing memory requests from multiple CPU's and/or IOP's. The results of the simulation are plotted in Figure 5-2. The results are astonishing. For example, when two CPU's request the same memory module at the rate of 200,000 accesses per second (5 μsec per request to a 2.5 μs cycle time

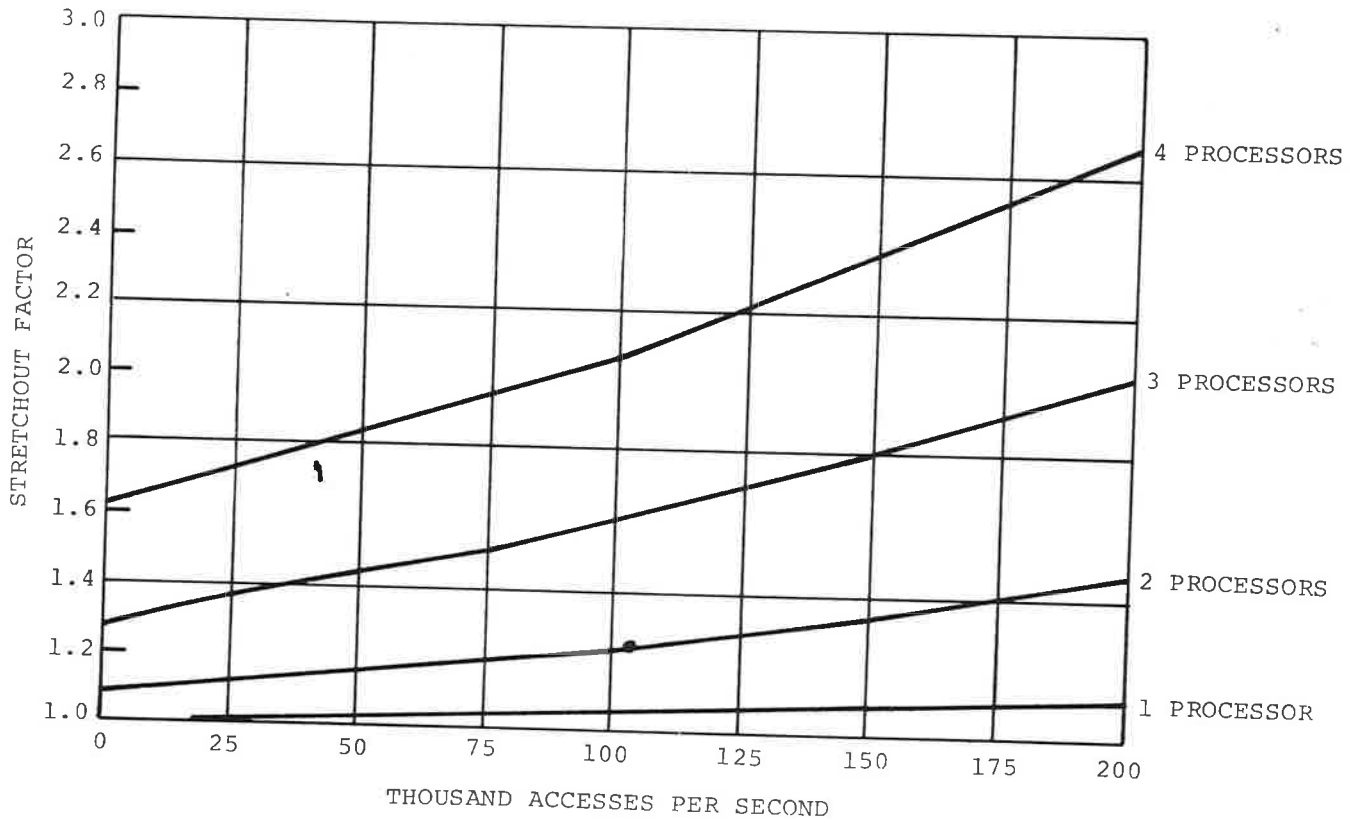



Figure 5-2 Execution time Delay as Function of Memory Conflict

memory), both tasks increase their running time by 50%. Note that program execution time stretchout with one processor is due to the simultaneous I/O operations. Careful analysis of the program structure design further reveals that good multiprogramming practices do not necessarily yield good results in a multiprocessing environment. In the design of the ARTCC Software System, reentrant code and common data base are used whenever possible to conserve storage. These practices naturally increase the possibility of memory conflict which weakens the performance of the multiprocessor system.

In short, because of the wide variation in task running time due to program loops, conditional branches, overlapped I/O and CPU operations, memory conflicts, and unpredictable external requests, the a priori assignment and theoretical scheduling schemes do not seem to provide a workable answer. What is needed is a flexible dynamic scheduling algorithm which attempts to minimize timing anomalies, meets deadlines for "hard" real-time tasks, honors tolerable response times for "not-so-urgent" tasks, and contributes tolerable bookkeeping overhead for the system. These requirements seem to point to the direction that a good heuristic approach may provide an answer.

In the next section, we shall report some of the findings of the studies following practical heuristic scheduling approaches.

# 6.0 AN HEURISTIC APPROACH TO MEMORY ALLOCATION AND DYNAMIC SCHEDULING

In this chapter we shall first establish a model on which our investigation into the memory conflict and timing anomaly problems will be based. Some definitions will be given and assumptions will be made. A general but practical way of structuring real-time programs will be discussed and its relation with respect to the directed graph representation of task set will be explained. We then proceed to describe some heuristics used for dynamic scheduling and memory allocations. The associated theoretical model from which these heuristics are derived will be presented. Finally, we propose a hardware aid to the scheduling activity so that system overhead is reduced to a minimum.

## 6.1 SYSTEM MODEL

The classical directed graph model will be used as the base for the study. The graph is reduced or acyclic, i.e., all data dependent program loops and conditional branches are within vertices. Vertices are used to represent tasks while directed arcs between vertices, representing partial ordering of the graph, establish the precedence relationship among tasks. Precedence relations specify the processing and computation sequences according to procedural and data "hand-off" constraints. Figure 6-1a shows that precedence relation A<B (A precedes B) satisfies both the procedural and data hand-off constraints. In Figure 6-1b however, the precedence relations A<C, B<C are needed only for data hand-off purposes, i.e., tasks A and B have to create proper T1 and T2 respectively before task C can use them to compute U. Figure 6-1c demonstrates the problem of independent tasks sharing common data base. Data X is needed by both tasks A and B. If processed in parallel, A, B will likely experience running time delays. Duplicate X seems to yield a solution; however, it creates other problems such as lock-out problems

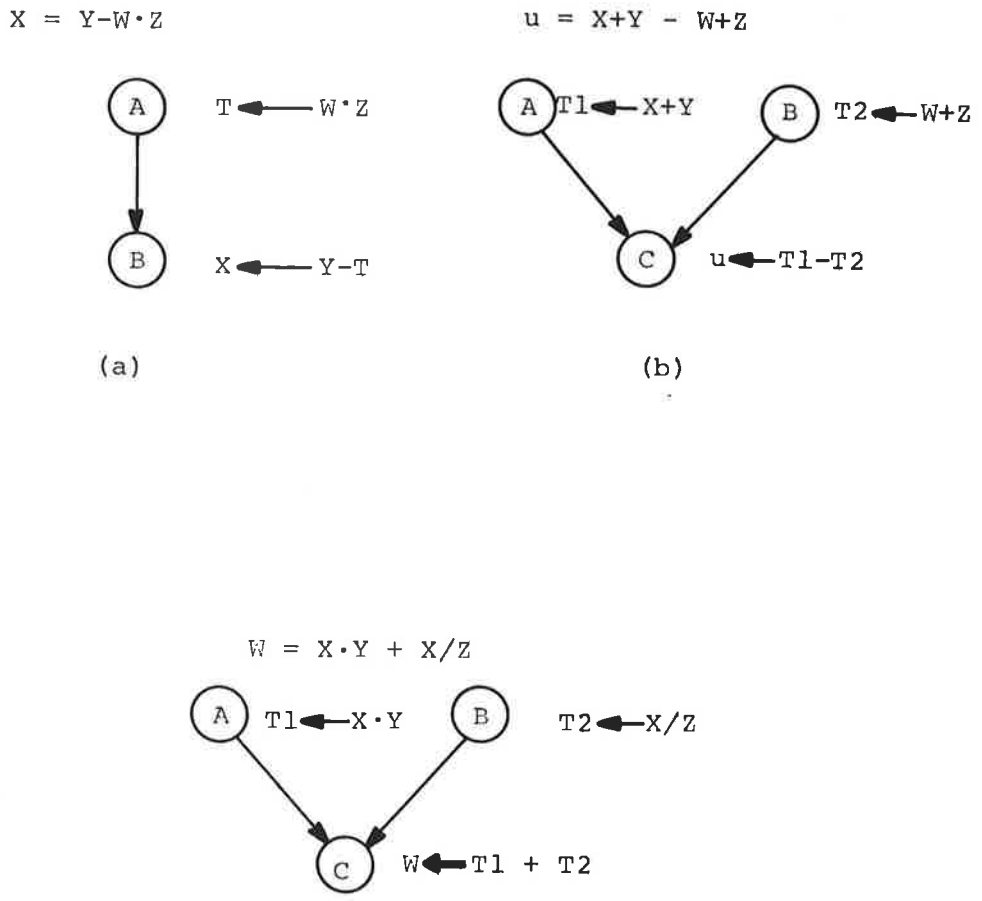



Figure 6-1 Procedural and Data Hand-Off Constraints

when X needs up-date, which copy should be accessed by what task, etc. In short, the use of a common data base to conserve storage, to enforce unified programming and to achieve the efficient operation of a multiprocessor system, cannot be obtained without any compromise. This complex problem is beyond the scope of our study, and we shall assume that no common data base exists among independent tasks for the rest of our study.

Program structures are traditionally hierarchical in nature. Parallel processes exhibit similar hierarchical structures. Parallelism can exist at instruction level, procedural statement level, and program level, etc. We shall assume that parallelism in our study occurs at the task and program level. Task is defined as a program segment chosen by the nature of the work it performs such as counting, searching record for up-dating, matrix computations, etc. Program is a piece of code to achieve certain major functions in an application, e.g., target tracking in an ATC system. A typical system represented by a hierarchical structure is illustrated in Figure 6-2.

All programs and data base are assumed to be core resident. File transfer, overlap, paging, etc., will not be considered here. To this end, memory allocation is static whereas task scheduling is dynamic in nature. Hence, the memory allocation process is carried out at program load time whereas task scheduling is carried out by the executive routine while the system is running.

## 6.2 DYNAMIC SCHEDULING HEURISTICS

Since one of the memory allocation heuristics depends on the results of scheduling, we will delay the discussion on memory allocation until scheduling schemes are introduced.

Let us review again Cases 1 and 4 of our timing anomaly study made previously in Section 5. In trying to find an answer for the increase in task set running time with the decrease of all individual task execution time, we found that the longest chain length (consisting of T1, and T9) was broken. In addition, the scan rule picks T2 and T3 before T4 is executed. While T2

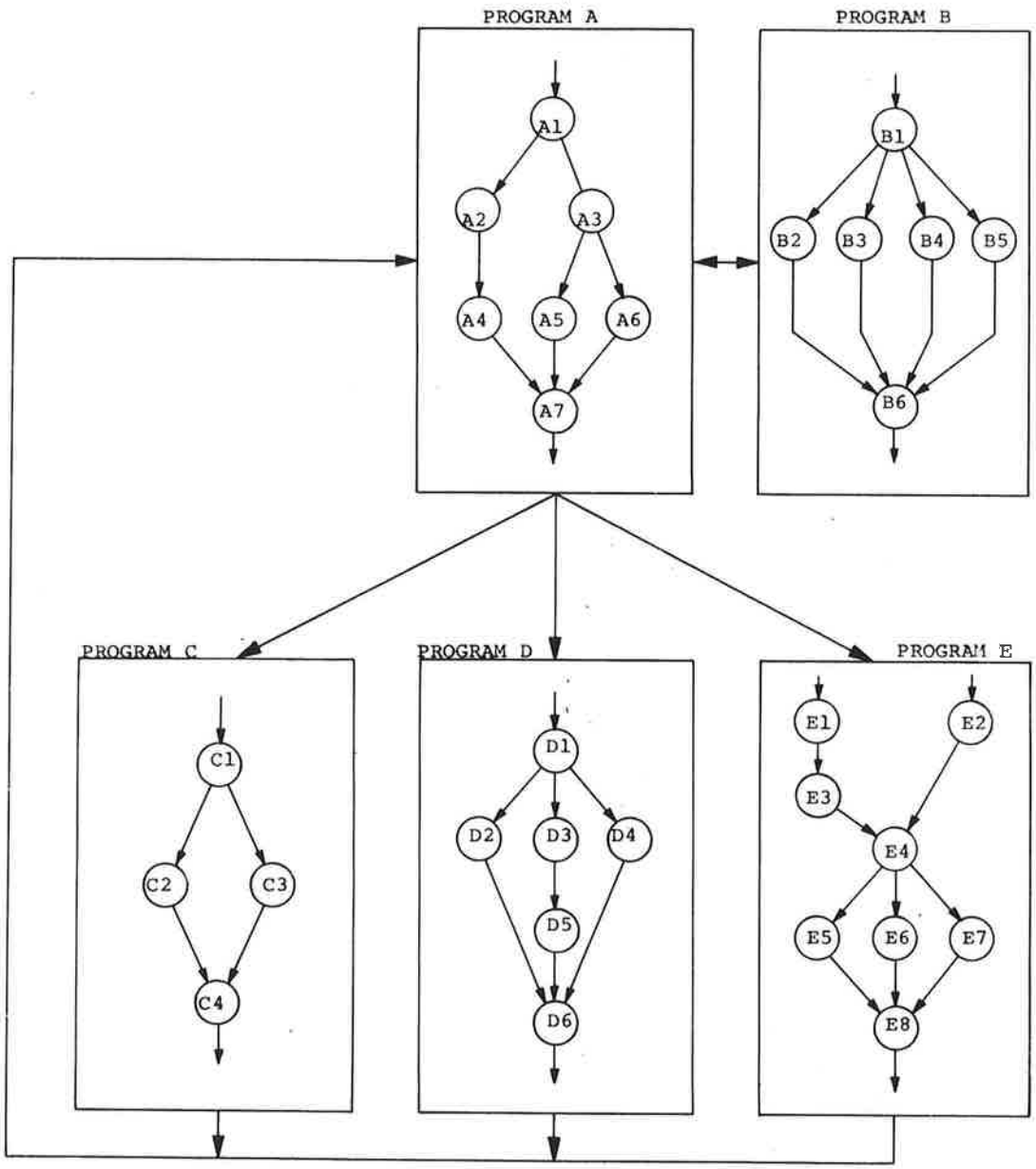



Figure 6-2 Hierarchical Program Structure

and T3 are independent, having no precedence relation with other tasks, T4 has four successors.

It is then immediately apparent that there are two heuristics at hand which may yield good schedules. They are:

1. Schedule task which currently leads the longest precedence path (LLP);
2. Scheduling task which currently leads the largest successor group (LSG).

The formal definitions of LPP and LSG are to be given in the following paragraphs:

Let i represent a task and $t_i$ be the processing time required by task i. For a given program graph, the processing times of the starting task s and the end task e are represented by $t_s$ and $t_e$, respectively. A path of task i is defined as a sequence of tasks traversing from starting task i to end task e. A Precedence Path is then the total processing time required to traverse from task i to task e.

Let $X_{ij}$ be the $j^{th}$ task sequence that makes a path from starting task i to end task e, j = 1,2,..., n where n is the number of distinctive paths between task i and task e.

Let $X_i$ be the set of paths between task i and task e. Then, the set of Precedence Paths between starting task i and end task e can be presented as

$$PP(ij) = \sum_{k \epsilon X_{ij}} (t_k)$$

and the Longest Precedence Path of task i is

$$LPP(i) = \operatorname{Max}_{\forall j} \left[ \sum_{k \epsilon X_{ij}} (t_k) \right]$$

Let the binary relation A<B be defined as task A immediately preceding task B, and binary relation A<<B be defined as task A preceding task B. It then follows that

1. If A<B, then A<<B,
2. If A<<B and B<<C, then A<<C

<< is a transistive binary relation.

The set of successor tasks of task i written as S(i) is defined as $S(i) = \{j \mid i<<j\}$. Given the relation i<j, a set of Successor Groups of task i is defined as the set of total processing times of task j and its successor tasks S(j). It is expressed as:

$$SG(i) = \sum_{k \varepsilon S(j)} (t_k) + t_j$$

and the Largest Successor Group of task i is

$$LSG(i) = \underset{\forall j}{Max}\left[\sum_{k \varepsilon S(j)} (t_k) + t_j\right]$$

These heuristics will certainly reduce timing anomalies if there are as many parallel paths as there are processors. When the number of processors is less than the number of parallel paths in a program (program and task-set will be used interchangeably), it creates a complicating yet important factor. As reported by Ramamoorthy, Chandy and Gonzalez[50] it is sometimes optimal to keep a processor idle even when there are tasks that can be executed immediately. An example program graph is shown in Figure 6-3 and its optimal schedule with two processors in the form of a Gantt Chart in Figure 6-4. However, timing anomaly occurs when we try to keep all processors busy as much as possible and this is illustrated in Figure 6-5. The reasons are: first, we have less processors (in this case 2) than parallel paths (3), and second, a task in a less critical path (task 6) happens to be available and assigned to processor 2 before tasks in critical path(s) can be made available. Critical paths are made of sequences of essential tasks which can be identified by examination of early and late partitions. In the example, the early (E) and late (L) partitions are expressed as

E = ( {1} {2,3} {4,5,6} {7,8} {9} )

L = ( {1} {2} {3,4,5}, {6,7,8} {9} )

The number of partitions (levels) in both E and L partitions are the same for a given program graph. Let i be the level index, then $E_i \Omega L_i$ are the essential tasks in level i. The essential tasks in this example are tabulated as follows:

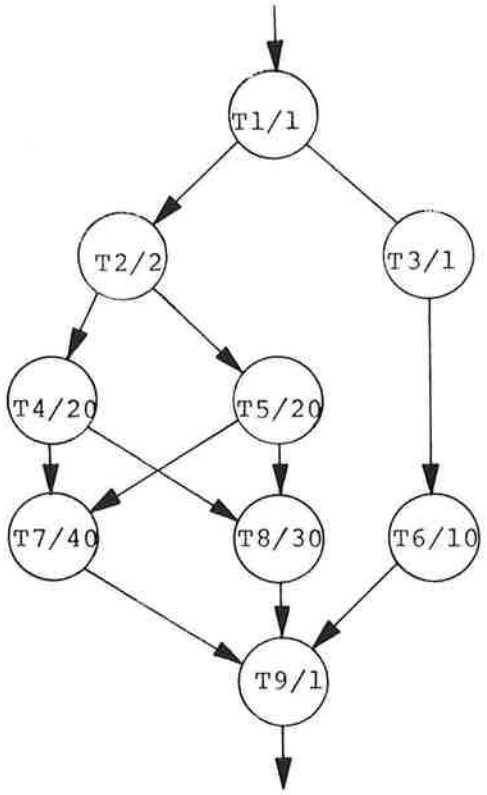



Figure 6-3 Sample Program Graph

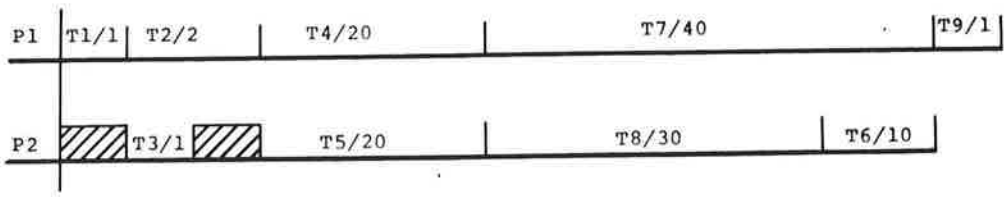



Figure 6-4 An Optimal Schedule for the Sample Graph

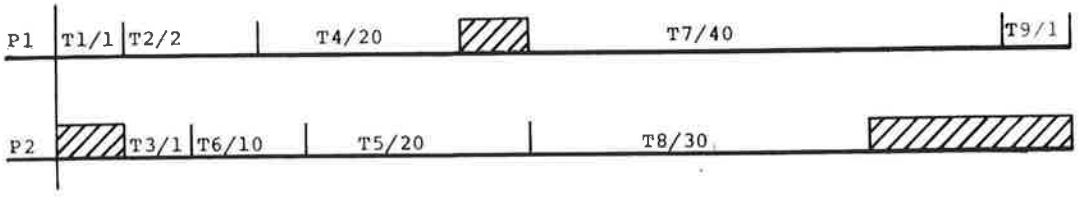



Figure 6-5 Schedule that Assigns Task-Processor Pair as Soon as They Become Available

| Level Index | Essential Tasks |
|---|---|
| 1 | 1 |

| Level Index | Essential Tasks |
|---|---|
| 2 | 2 |
| 3 | 4,5 |
| 4 | 7,8 |
| 5 | 9 |

Tasks 3 and 6 are non-essential tasks. Hence non-essential tasks can be delayed to make room for essential tasks to be scheduled for the purpose of timing improvement.

Given a program graph, when the first available task first served strategy and LPP followed by LSG heuristics are used for scheduling, the resulting Gantt chart is consistent with E - partition (Figure 6-5). If we reverse the directions of all arrows in the given graph, as indicated in Figure 6-6, and apply the same scheduling strategy starting from task 9 and ending at task 1, the resulting Gantt chart is consistent with L-partition which is shown in Figure 6-7. Comparing these two Gantt charts, it is apparent that L-partition based scheduling is an optimal approach. This means that in an actual program run, the original graph, as opposed to the reverse one, should be followed, and some of the non-essential tasks (task 6) should be kept as late as possible for its execution. Please note that the E or L-partition alternatives for the execution of task 3 does not affect the overall program execution time. One way to delay non-essential tasks is to impose "pseudo precedence relation (s)" on them based on the timing relations shown in the L-partition derived Gantt chart. Figure 6-7 shows that task 6 should be executed after tasks 4 and 5, hence it is recommended that pseudo precedence relations be established from tasks 4 and 5 to task 6.

In our example, we created a dummy task TD with execution time 0 to simplify the representation of precedence relations between parallel predecessors and successors. The set of relations

T4<T7, T4<T8

T5<T7, T5<T8

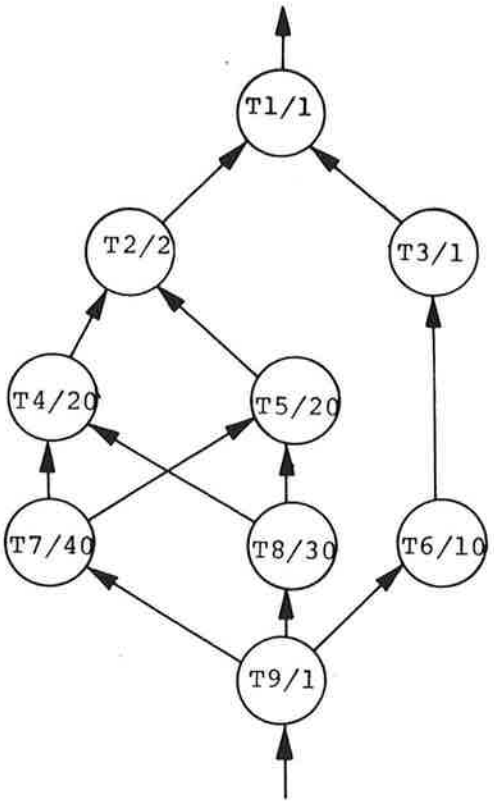



Figure 6-6 Reverse Program Graph for L-Partition Schedule

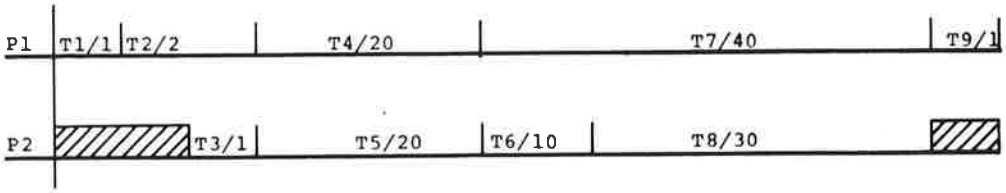



Figure 6-7 L-Partition Schedule

## 6.3 MEMORY ALLOCATION SCHEMES

By now, we realize, on the one hand, that tasks in a multiprocessor should be processed in parallel as much as possible. On the other hand, care should be given to the allocation of tasks in memory modules so that potential memory conflict during program execution is minimized, if not eliminated. For ease of discussion, let us assume that all memory modules are divided into an equal number of equal-sized pages. Furthermore, we assume that each task occupies only one page of memory space.

Given N tasks to be allocated into N pages, a feasible solution to the problem is any permutation of (1, 2, 3, ---, N) where a permutation ($T_1$, $T_2$, $T_3$, ---, $T_N$) corresponds to placing task $T_1$ in page 1, task $T_2$ in page 2, etc. An exact solution may be obtained by considering and evaluating all N! permutations. Assuming a 12-task problem being examined at a rate of 1 millisecond for one permutation, a 12! evaluation process would require about 100 hours which is not only costly but also impractical. Alternately, a dynamic programming approach, after Karp and Held[65], could also provide a feasible solution. Placing n tasks into n memory pages is a sequential decision process which is a finite automaton with a certain cost structure superimposed. The states of the automaton are represented by all the subsets of the unordered set {$T_1$, $T_2$, $T_3$, ---, $T_n$}, which have clearly $2^n$ states. The cost of a state {$T_1$, $T_2$, ---, $T_i$} is defined as the minimum cost of all permutations of $T_1$, $T_2$, ---, $T_i$. As a result, the cost of the "final state" {$T_1$, $T_2$, $T_3$, ---, $T_n$} is the cost of the minimum solution. Since all $2^n$ states must be examined and all permutations associated with each state evaluated, this algorithm possesses a run-time growth rate at least proportional to $2^n$. When n gets to be large, the computation commands a certain respect. Since both exact solution algorithms possess demanding computation requirements, a search for simple heuristic procedures is highly desirable.

The single important and dominating heuristic rule in memory allocation is to place sequential (or dependent) tasks in the same memory module and parallel (or independent) tasks in different

memory modules. Given a program graph G, tasks P and Q may be placed in the same memory module if and only if there is a path between P and Q. In addition, there are other practical factors to be considered such as a number of memory modules and number of processors in the system. We propose two memory allocation schemes of which one is memory-oriented and the other is processor-oriented.

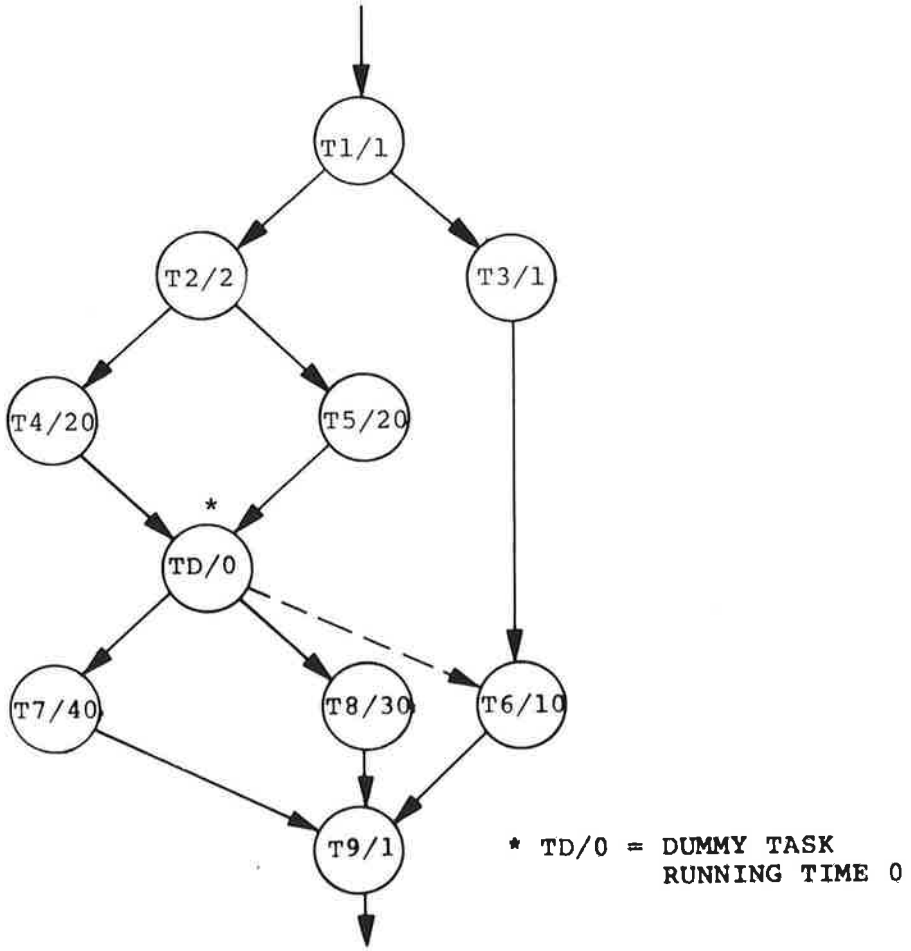



Figure 6-8 Program Graph with Added Pseudo Precedence Relation

## 6.4 MEMORY ALLOCATION BASED ON PRECEDENCE PARTITIONS (MAPP)

Given a program graph G, it can be segmented into precedence partitions. Whether they are E-partitions or L-partitions, the number of partitions in a given graph is fixed. An essential task at partition i is defined as a task in the set $(EP_i \cap LP_i)$. If a task appears in both $EP_i$ and $LP_j$, then the partition difference index (PDI) equals j-i.

Let us assume that all memory modules are divided into equal pages, and without losing generality, we assume that each task occupies only one page of memory. The MAPP memory allocation scheme consists of the following steps:

Step 1: Determine E and L partitions (EP and LP)

Step 2: Following EP level sequence ($EP_1$, $EP_2$,...,etc.) place each task in a partition level in separate memory modules, assuming unlimited memory modules and pages. The allocation processes should consider the essential tasks first. The non-essential tasks are then placed on a low "partition difference index" first basic.

Step 3: If the number of memory modules used exceeds the number of memory modules in real system, initiate "horizontal folding" process in which tasks placed in those excess memory modules are to be relocated into the empty pages of the actual memory modules. The fact that horizontal folding process is required implies more parallel paths than memory modules. Hence, we are forced to allocate parallel processable tasks in one memory module.

Step 4: If in some modules the number of pages used exceeds the memory pages per module in real system, "vertical folding" process is needed to relocate tasks of those excess pages into empty pages of other modules in

the system. The criteria to use here to find a module with empty page(s) is that all tasks occupied in that module have to be able to establish common paths with the relocating task.

Step 5: This is a clean-up step to see whether there are still some tasks left unprocessed. If so, find empty pages and allocate tasks in them unconditionally.

An example should be helpful at this time. Figure 6-9 gives a rather complicated program graph with 25 tasks. We would like to allocate these 25 tasks into 6 memory modules with 5 pages per module.

Step 1: E = ({1},{2,3,4,5,6,7,8,9,10,11,12},{13,14,15},{16,17,18},
{19,20,21,22},{23,24},{25})

L = ({1},{5,6,7,8,9,10},{11,12,13},{14,16,17,18},
{15,20,21,22},{2,3,4,19,23,24},{25})

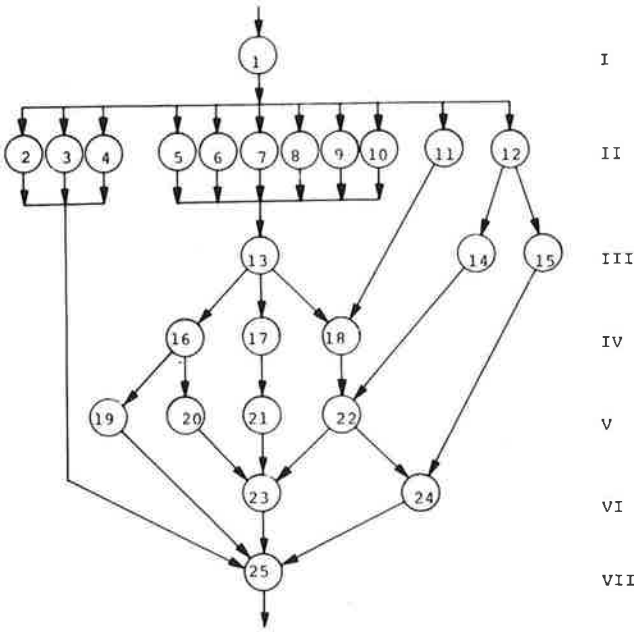



Figure 6-9 Sample Program Graph

| Partition Levels | Essential Tasks |
|---|---|
| 1 | 1 |
| 2 | 5,6,7,8,9,10 |
| 3 | 13 |
| 4 | 16,17,18 |
| 5 | 20,21,22 |
| 6 | 23,24 |
| 7 | 25 |

| Partition Difference Index | Non-Essential Tasks |
|---|---|
| 6-2 = 4 | 2,3,4 |
| 3-2 = 1 | 11,12 |
| 4-3 = 1 | 14 |
| 5-3 = 2 | 15 |
| 6-5 = 1 | 19 |

Step 2: Referring to the memory map below, at EP level I there is only one task 1, and it is allocated to memory module 1, page 1. At EP level II, tasks 5,6,7,8,9, and 10 are essential tasks which are allocated first, one per page per module. Tasks 11 and 12 have partition difference index 1 next, and tasks 2,3, and 4 have the highest partition difference index 4 and are allocated last.

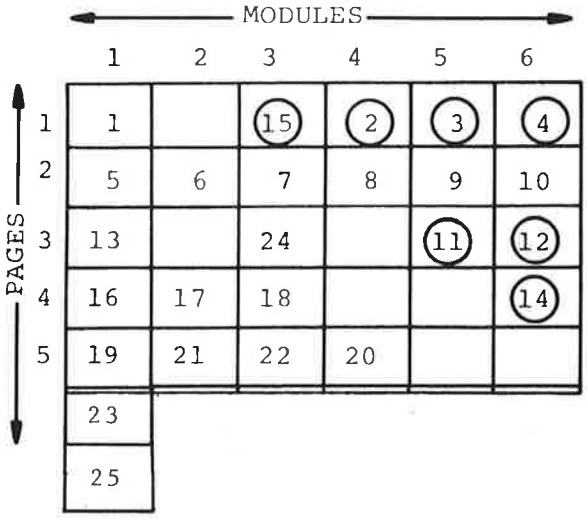
MODULES

| PAGES | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | | | | | |
| 2 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 2 | 3 | 4 | |
| 3 | 13 | | | | | | | 14 | | | | 15 |
| 4 | 16 | 17 | 18 | | | | | | | | | |
| 5 | 20 | 21 | 22 | 19 | | | | | | | | |
| 6 | 23 | | 24 | | | | | | | | | |
| 7 | 25 | | | | | | | | | | | |

At Level III, task 13 has a common path with tasks 1, and 5, so task 13 is allocated in module 1. Task 14 finds a common path with task 12, and is allocated with task 12 in module 8. Task 15 cannot find path with any, and is allocated a new module 12. At Levels IV and V, tasks easily find paths with allocated tasks hence they are correspondingly allocated. Same argument goes for tasks 23, 24, and 25.

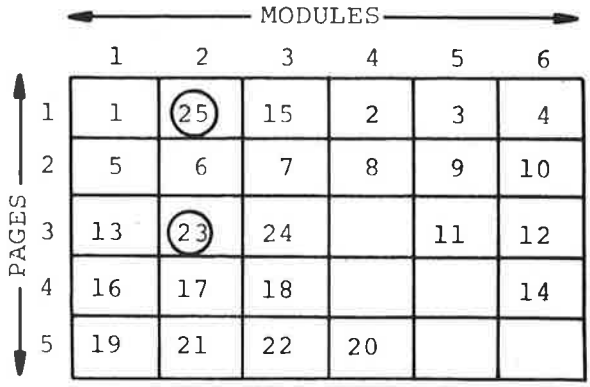
Step 3: Horizontal Folding

We have used 12 memory modules and the system has only 6 so the horizontal folding step is necessary which is taken from right to left. Task 15 found a path with task 24. Tasks 2, 3, and 4 could not find any paths and they were arbitrarily placed with tasks 8, 9, and 10 respectively; same argument goes for tasks 11, 12 and 14 and they are relocated in memory modules 5 and 6.




Step 4: Only tasks 23 and 25 are outside the page limit in this example. However, both tasks 23 and 25 found paths with all tasks in module 2 which has two

empty pages. The final memory map is thus obtained. No clean-up job is needed.




### 6.4.1 Memory Allocation Based on Static Scheduling (MASS)

While the MAPP memory allocation scheme is memory-oriented, MASS scheme is processor-oriented. As previously memtioned, the number of processors in a system is generally less than the number of memory modules. We then follow the scheduling algorithm previously discussed to establish a Gantt Chart showing schedules of all the processors. A transformation is then made from processor schedules into an overlap matrix whose entry $X_{ij}$ represents the concurrent execution time between tasks i and j. Referring back to Figure 6.4, for example, the overlap index $X_{67}$ has 10 time units. Tasks within a processor's schedule are allocated into one memory module. When one module's capacity overflows, the rest of the tasks in the schedule are allocated to a fresh memory module. Problems occur when separate modules are not enough to handle tasks of separate schedules which may result in having parallel tasks sharing the same memory module. A smoothing process is necessary to selectively swap tasks among memory modules until possible overlaps are eliminated or reduced.

The MASS memory allocation procedure consists of the following steps:

Step 1: Allocate tasks of a processor's schedule in a memory module. When there are more tasks in a schedule than pages in a memory module, continue to fill a new empty module with the remaining tasks.

Step 2: Start a fresh empty memory module with a new schedule and continue steps 1 and 2 until one of two conditions occur and then proceed to step 3.

Step 3: The allocation process stops when no more tasks are left to be allocated. It proceeds to step 4 when there are tasks unallocated and the fresh empty memory modules are exhausted.

Step 4: Select an unallocated task i and compute the sum of the overlap index $\sum_{j \epsilon M_k} X_{ij}$ with all tasks j in a memory module k which has empty pages. Allocate task i in the memory module k if $\sum_{j \epsilon M_k} X_{ij}$ equals zero or is a minimum.

Step 5: If $\sum_{j \epsilon M_k} X_{ij}$ is not zero, consider the possibility of swapping task i in $M_k$ with task $\ell$ in $M_n$ if and only if

$$\sum_{m \epsilon (M_n - \ell)} X_{im} = \sum_{j \epsilon (M_k - i)} X_{\ell j} = 0$$

The process repeats for all tasks in $M_n$ and for all memory modules until a swap is made.

Step 6: If a swap is not possible in step 5, relax the criteria as follows:

swap task i in $M_k$ with task $\ell$ in $M_n$ if and only if

$$\sum_{m\in(M_n-\ell)} X_{im} \quad < \quad \sum_{m\in(M_n-\ell)} X_{\ell m} \qquad \text{and}$$

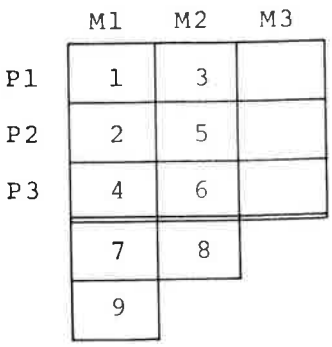
$$\sum_{j\in(M_k-i)} X_{\ell j} \quad < \quad \sum_{j\in(M_k-i)} X_{ij}$$

Repeat steps 4, 5, and 6 appropriately until all tasks are allocated.

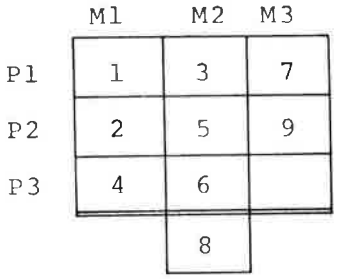
Again, this process is best illustrated by an example. Let us consider the program graph of Figure 6.8 and try to allocate the whole task set into three memory modules each of which has three pages. Referring to the corresponding Gantt Chart shown in Figure 6.7, an overlap matrix is established with the following sparsely distributed overlap indexes:

$$X_{ij} \begin{cases} 1 & i=2,j=3 \text{ or } i=3,j=2 \\ 20 & i=4,j=5 \text{ or } i=5,j=4 \\ 10 & i=7,j=6 \text{ or } i=6,j=7 \\ 30 & i=8,j=7 \text{ or } i=7,j=8 \\ 0 & \text{otherwise} \end{cases}$$

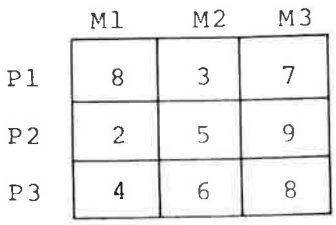
We assign the task schedule of processor P1 to module 1 and the task schedule of processor P2 to module 2 as shown in the following diagram 1. Take the longest remaining schedule, tasks 7 and 9, and assign them to memory module 3, diagram 2. After assigning the last task 8 to memory module 3, diagram 3, task 7 and task 8 have an overlap index 30. Then, task 8 is checked against all possible tasks in M1 to see whether a swap is possible. The first task found in M1 is task 1 and it is determined from the overlap matrix that task 1 does not interfere with tasks 7 and 9, and task 8 does not interfere with tasks 2 and 4. Hence, a swap between tasks 1 and 8 is made and the final memory map is obtained (diagram 4). Note that there are quite a number of choices of swapping candidates for task 8; all tasks in M1 and tasks 3 and 5 in M2 can all be used to ex-cahnge with task 8.



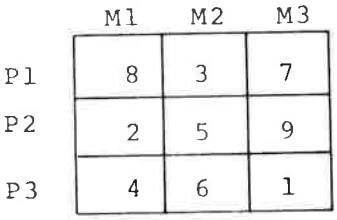

1




2




3




4

There are two aspects that are worth noting; first, the process always converges. Step 5 looks for the best possible condition for swapping, i.e., overlap indexes of both modules affected are zero. When optimal conditions can not be met, step 6 relaxes the criteria to permit swapping if there are improvements, such as reduction of overlap indexes, to be gained (in both modules). Second, there are two major heuristics employed in the process, i.e., the initial allocation heuristic and the swapping heuristic. Of the two, the swapping heuristic is the work horse while the initial allocation heuristic provides a good starting point that shortens the overall allocation process. If the initial allocation heuristic is not used, application of the swapping heuristic to different initial memory maps would yield different results. Referring to the previous example, a sequential assignment produced memory map 1. The sum of the overlap indexes of M1, M2 and M3 are 1,20, and 30 respectively. The swapping process is performed as follows:

1. Consider the memory module with the highest sum of overlap indexes, i.e., M3.

2. Find the task which produces the highest overlap index, i.e., task 7.

3. Following a natural numbering sequence on memory modules and pages, make attempts to swap tasks with task 7. Consider swapping task 1 with task 7. Although $\sum_{j\in(M3-7)} X_{1j} = 0$ but $\sum_{m\in(M1-1)} X_{7m} \neq 0$, no swapping is made.

4. Consider tasks 2 and 7. We have

$$\sum_{m\in(M1-2)} X_{7m} = \sum_{j\in(M3-7)} X_{2j} = 0$$

Hence, tasks 2 and 7 are swapped to produce memory map 2.

5. Consider M2 whose overlap index sum is 20, and consider task 4 which produced this overlap index.

6. Again following the natural numbering sequence, consider tasks 1 and 4. Since

$$\sum_{m\in(M1-1)} X_{4m} = \sum_{j\in(M2-4)} X_{1j} = 0$$

Then, tasks 1 and 4 are swapped to produce memory map 3.

| | M1 | M2 | M3 |
|---|---|---|---|
| P1 | 1 | 4 | 7 |
| P2 | 2 | 5 | 8 |
| P3 | 3 | 6 | 9 |
| Sum of overlap indexes | 1 | 20 | 30 |

Map 1

| | M1 | M2 | M3 |
|---|---|---|---|
| P1 | 1 | 4 | 2 |
| P2 | 7 | 5 | 8 |
| P3 | 3 | 6 | 9 |
| Sum of overlap indexes | 0 | 20 | 0 |

Map 2

| | M1 | M2 | M3 |
|---|---|---|---|
| P1 | 4 | 1 | 2 |
| P2 | 7 | 5 | 8 |
| P3 | 3 | 6 | 9 |
| Sum of overlap indexes | 0 | 0 | 0 |

Map 3

Note that this final result, map 3, is different from the previous result obtained with the initial allocation heuristics, although both results yield the optimal solution, i.e., the sum of the overlap index for all memory modules are zero.

## 6.5 MORE TIMING CONSIDERATIONS

The individual task running times used in the discussion so far are assumed to be fixed as either maximum run time or expected mean value. In reality, most task run times vary from run to run due to data-dependent loops, branch conditions, etc. This variation may not present any problem to the system if one and only one program (task set) is being executed. However, when two or more programs are active (Figure 6-2) and demand service, idling processor(s) of one schedule may be called to execute task(s) of another program which may not release the processor(s) back to the first schedule soon enough to finish it in the earliest possible time. In order to solve this problem, carefully planned system constraints may have to be established. In a non-real-time environment emphasis is placed on the efficient operation of the system, which would try to keep all processors busy at all times at the sacrifice of experiencing run time delays by some programs. In a real-time operation on the other hand, uninterrupted attention should be given to those "time-critical" programs such that timing requirements can be met. We do not propose any general solution to this problem, but some of our observations could lead to a workable control scheme in a real-time system.

### 6.5.1 Priority and Urgency Indexes

At present, most real-time systems are priority-driven, and the priorities are pre-assigned according to some predetermined guidelines. However, in a multiprocessor system with a complex program mix and system reconfiguration capability, the conventional role of priority needs to be reviewed. In order to remove some possible confusion, we will attempt to define the following notions:

Priority - as a measure of relative importance of a program with respect to other programs in an overall system. Hence, priority is considered to be global in nature.

Urgency - as a measure of attention for service required by a program in order to meet its deadline imposed by the system. Hence, urgency is a local criteria which varies with time.

We further clarify these two terms by saying that priority is, in a way, static in nature, and urgency is dynamic. As long as the system capacity is maintained, every program will have its share of the service. However, when a part of the system is at fault, and the remaining system resources may not be adequate to serve the entire system function, some low priority program (less important) may not be served at all and hence dropped off the system, or may be executed in a much lower rate in a degraded fashion. The urgency index on the other hand works on a "moment-to-moment" basis and reflects properly the degree of the pressing needs of attention of the individual programs. Each program has a priority index and an urgency index assigned to it. Once the assignment is made, all tasks within a program

have the same priority and urgency indexes. The study of how a system should behave in case of partial failure is a complex and separate problem which is beyond the scope of task scheduling. We are to limit our attention to the treatment of urgency in terms of task scheduling only.

### 6.5.2 Hierarchical Multiprocessing

The hierarchical program structure demonstrated in Figure 6-2 clearly suggests the possibility of hierarchical multiprocessing, i.e., simultaneous processing of tasks of two or more programs. The point is how to control such an operation so that system resources are efficiently utilized (keep all processors busy, as much as possible) and yet meet program timing requirements. Fortunately, not every program in a real-time system is "time-critical". Even if they were all time-critical, they do have variations in the degrees of urgency. ARTS air traffic control, for example, has five major functions: keyboard input processing, display output processing, interfacility processing, target detection and tracking. Of the five, only target detection and tracking functions are truely time-critical. Both keyboard and display require real-time response; however, the effect of slight delays in performing these functions is by no means catastrophic. As far as target detection and tracking are concerned, they perform in a different periodical basis on which the target detection program is run once every 2.5 ms, and the tracking program is run once every 125 ms. The higher urgency index is then given to the programs with a high execution rate. A workable scheme would be to allow non-time-critical but time-sensitive programs (e.g., keyboard and display processings) to share all system resources, whereas all system resources are dedicated to those time-critical programs such as target detection and tracking. The activation of the periodic programs are by means of external interrupts.

## 6.6 IMPLEMENTATION APPROACH

Several schemes of memory allocation and task scheduling were given in previous sections. Some discussion is in order to see how these schemes can be put together to establish a workable system. Key elements needed here are ways to describe available system resources and structures of the functional programs and tasks, which can be achieved through the use of tables. The System Resource Table specifies such items as number of memory modules, pages, processors and I/O channels, as well as various I/O devices. The Program Structure Table specifies the precedence relations, urgency index, and many other important attributes which are tabulated as follows:

- Program/Task Name
- Urgency Index
- Priority Index
- Task Execution Time
- Longest Precedence Path
- Largest Successor Group
- Predecessor Table Pointer
- Successor Table Pointer

} 1

Memory Assignment

- Memory Module Numbers
- Module Page Numbers
- Protection Keys
- Etc.

} 2

I/O Assignment

- I/O Channels
- Device Numbers
- Etc.

Status

- Inactive
- Ready for Process
- In Waiting
- In Process
- Etc.

} 3

This table is organized on a task basis. Part 1 is furnished by the system designer/programmer. Based on part 1 and the system resource table, the operating system at program load time generates a memory assignment map, part 2, using one of the memory allocation algorithms described previously. Part 3 of the program structure table is of cource dynamic in nature, and is assigned and updated by the Executive program during run time.

In a multiprocessor-multitask environment, especially when tasks have real-time demand, events occur at times which are unpredictable, and sequences in which programs are executed change constantly. The executive program of such a system finds itself constantly doing bookkeeping chores, such as set task X active, put task Y on a queue for an I/O channel, time to execute task Z because the record it requested from file has just been read in core memory, etc. The bookkeeping chores imply a lot of search operations going on by the Executive Routine, such as; Who has the highest priority? Who has the highest real-time urgency index and needs attention from the system immediately? Who has the maximum LPP and/or LSG. Conventional executives (or supervisor, master control, etc.) carry out this type of function by searching through a list in core item by item, which means a lot of repetitious time-consuming overhead computer time wasted.

This type of function is ideally performed in an associative processor. Although PAP is originally proposed for high-speed parallel processing of application programs, there is no reason why it could not be time-shared for executive control as well, especially the multiprocessor system which would have two or more PAP's connected. The whole program structure table can be stored in an associative memory. There are a number of ways of organizing such a table. A simple structure is used here to demonstrate the versatility of associative memory. Each entry of the program structure table, corresponding to a task, is stored in an associative memory word which is divided into a number of fields. Each field is corresponding to an attribute

which describes the task. The word format is shown in Figure 6-10.

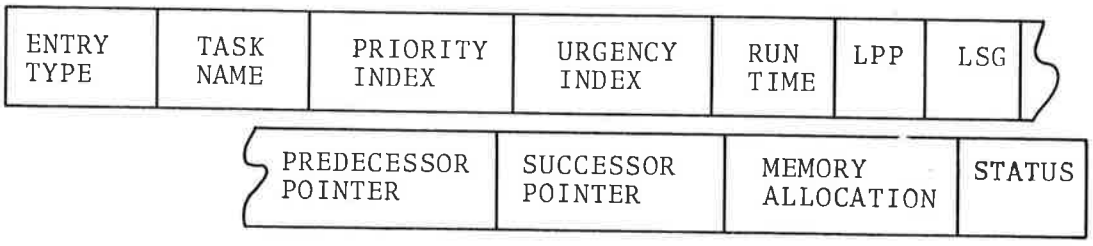



Figure 6-10 Associative Memory Word Format for Executive Control

The "ENTRY TYPE" field is used to enable the sharing of an associative memory for various types of storage purposes. It is this field that could distinguish, for instance, an aircraft track record from a program structure table entry. In the dynamic scheduling process, the executive program would first find out if there are some "Ready for Process" tasks by searching the STATUS field. Finding some, he may search for those with highest URGENCY INDEX followed by searching for the maximum LPP and again maximum LSG in order to select the next task to process. Since every search takes one or a few memory cycles only, system bookkeeping overhead is greatly reduced. The logic flow of the scheduling algorithm is illustrated in Figure 6-11.

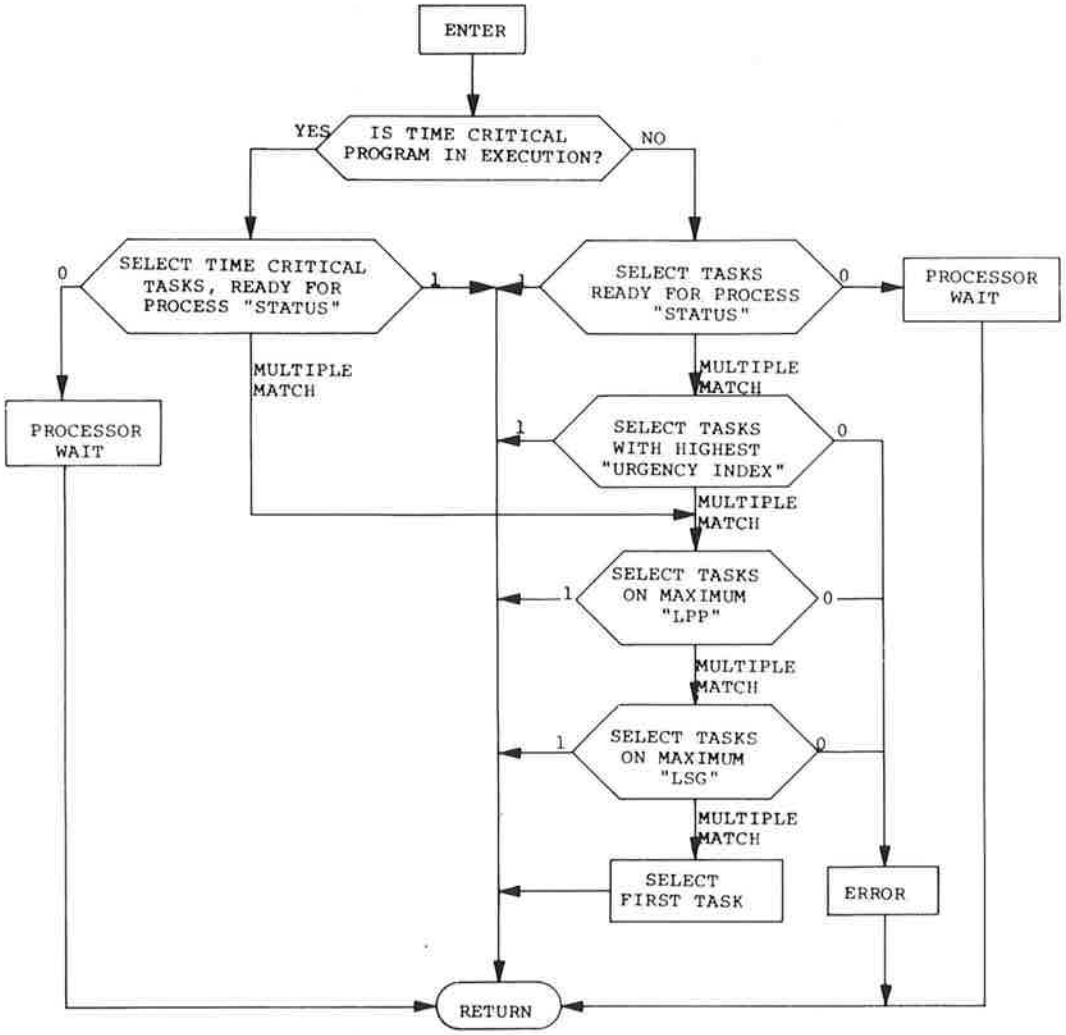



Figure 6-11 Dynamic Scheduling Logic Flow with Associative Memory Control

# 7.0 SIMULATION ANALYSIS AND EVALUATION

When a system of independent processes reaches a certain degree of complexity, it becomes exceedingly difficult, if not impossible, for humans to appreciate fully the interactions among parts of the system and to evaluate the effects of these interactions. Two systematic approaches are available for studying complex systems, which can provide some insights so that an appropriate evaluation of the system can be made. These two approaches are simulation and analytical modeling.

The differences between these approaches have been discussed fully in the leterature, and it is fair to state that each has its advantages. Simulation has the advantage of describing exactly what is happening in the system being analyzed, not an average of what might happen. A system being simulated can be examined during the simulation at any time. There are no restrictions on the analysis of simulation data, while the types of statistics available from analytical modeling, e.g., queuing theory studies are often limited. Disadvantages of simulation, as contrast with queuing theory, include lack of generality, the output describes the system behavior only for the given initial conditions, which may not be typical. The effect of this can often be offset only at the expense of large amounts of computer time. Programming requirements are also large, since each part of a complex system needs normally to be modeled separately. For the investigations of real-time systems conducted here, interests are focused on the examination of the "worst case" condition of program execution time due to timing anomalies and memory conflict. Hence, simulation is selected to exercise specific memory allocation and task scheduling heuristics and to observe effects on program execution times.

## 7.1 THE SIMULATION STRUCTURE

Most languages written expressly for the purpose of simulation tend to be too general and inefficient in their use

of computer time, inflexible in input-output or difficult to learn. For ease of use, programming simplicity, and easy adaptation to most machines, an event-driven simulation language GASP is used for our simulation. Strictly speaking, GASP is nothing more than a collection of FORTRAN subroutines and functions under the control of an executive which updates attributes of entities in files, advances time counters and collects statistics on specified parameters automatically. Its simulation concepts are quickly grasped because they are presented in the familiar framework of FORTRAN. Appendix A gives a comprehensive discription of the operation and the facilities of GASP.

### 7.1.1 Data Structure

Various states of the system under simulation are represented by a set of tables. There are three basis input tables to represent a program structure: a Task Description Table, an Immediate Successor Table and a Precedence Partition Table. Associated with each task of the Task Description Table are six attributes as follows:

Execution Time
Longest Precedence Path
Largest Successor Group
Predecessor Number
Predecessor Counter
Immediate Successor Pointer

Initially, the contents of the Predecessor Number and Predecessor Counter are identical. After completion of each of its predecessors, the Predecessor Counter is decremented by 1. This task is then ready for execution when its Predecessor Counter reaches zero. As soon as this task is being executed by a processor, the Predecessor Counter is reset to its original value by a Predecessor Number. The Immediate Successor Pointer is used to link to the Immediate Successor Table so that the precedence relationship among tasks is established. The Precedence Partition Table is indexed by the partition level number, whose entries are tasks within each partition level. There are

two output tables. One is the Memory Allocation Table which is indexed by the memory module, and page numbers, and the corresponding entries are task numbers assigned to these pages. The other is the Processor Activity Table which gives a profile of activities for each processor during the span of each simulation run. It records the actual start and end times of all tasks being executed by a specific processor including processor idle time. Finally, there is a Statistical Output Table which records such items as memory conflict occurrences, processor idle times, program run times, as well as a histogram showing the distribution of the memory conflict which has occurred. Important system parameters are entered into the system as individual variables. They are:

Number of memory modules in the system
Number of memory pages per module
Number of processors
Number of tasks
Parameter to select memory allocation heuristics
Parameter to select dynamic scheduling algorithms

### 7.1.2 Program Structure

The multiprocessor simulation program structure is represented in Figure 7-1. The input tables are used to derive memory allocations by three memory allocation heuristics. The Memory Allocation Table obtained is fed to the simulator together with all input tables. The simulator, selecting one out of three scheduling heuristic rules, conducts the dynamic scheduling simulation and collects appropriate statistics at proper time intervals. Since most of bookkeeping work has been taken care of by GASP facilities, the major coding for the simulation is to program the various events which would happen in the system. In our study, we have only one single event which is the "End-of-Task" event. This event coded as a subroutine (NDTASK) is called by the GASP executive program. Upon entering NDTASK, an active processor is released after taking statistics. All its successor's Predecessor Counters are decremented. NDTASK next tests for task readiness and processor availability. If there are

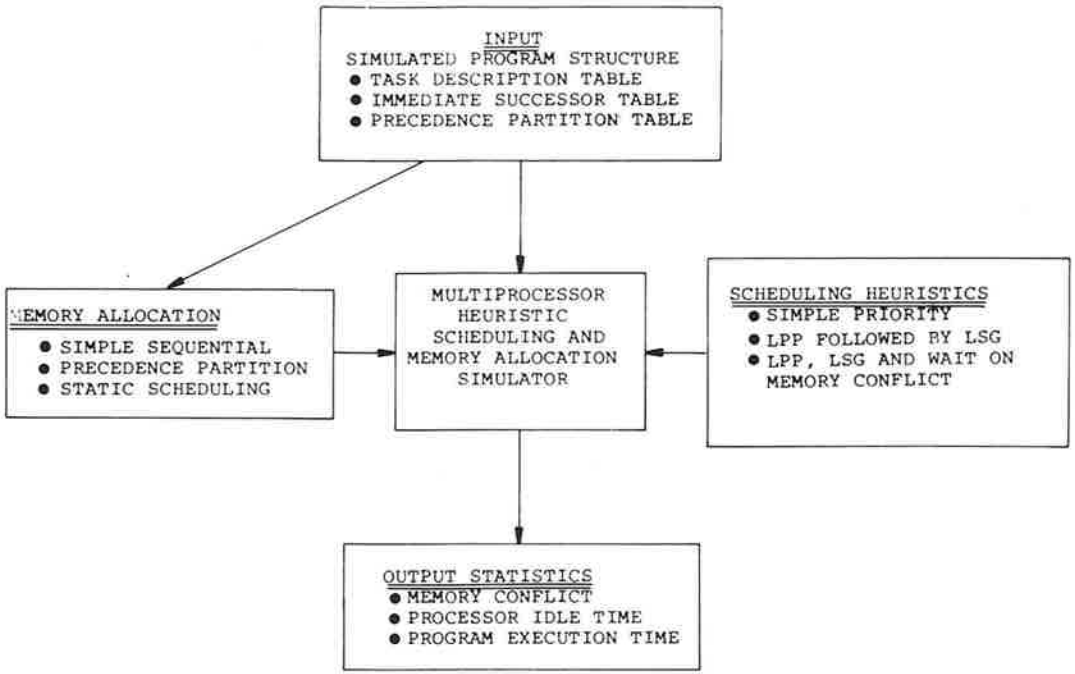



Figure 7-1 Multiprocess Simulation Program Structure

tasks ready to be processed, NDTASK schedules tasks by selected heuristic rules, determines task execution time by adding to the specified task execution time a generated random number to represent the variations in task run time, and updates the Processor Activity Table. It then proceeds to detect memory conflict, takes statistics on memory conflict time and revises the task completion time due to this conflict. Finally, it restores the Predecessor Counter by the contents of the Predecessor Number stored in the Task Description Table, updates GASP's Event File, establishes the next event and returns to GASP.

It should be noted that each memory allocation scheme is a routine reflecting one heuristic rule, which is executed only once in each simulation, and is called by the GASP main program. The three memory allocation routines are:

MASQ Memory Allocation via Sequential Assignment
MAPP Memory Allocation via Precedence Partition
MASS Memory Allocation via Static Scheduling

In addition, there are two subroutines supporting those memory allocation subroutines. Subroutine COPLAT computes the partition level and allocates task in MAPP, and subroutine OVRLAP detects and computes the memory overlap time in MASS.

A number of subroutines are also called by NDTASK. Each heuristic scheduling rule, except the simple sequential heuristic is carried out by a subroutine. They are:

PPSG = Select Longest Precedence Path first, IF multiple responses, select one with Largest Successor Group

WAIT = Same as PPSG except processor waits when possible memory conflict is detected.

NDTASK also has a supporting subroutine MCFLT which detects and takes statistics on memory conflict time, and it is used differently from subroutine OVRLAP. For detail program logic flow see the flow charts and program listings documented in Appendix B.

## 7.2 REPRESENTATION OF TASK STRUCTURE

At first, it seems highly desirable to use the actual set of ATC programs for the input of the simulator. Further examination of the system led to the belief that the program design is highly dependent on the computer architecture used in the system. Since a pipelined associative processor is introduced into a multiprocessor structure which has no resemblance to the present ATC computers, the use of ATC program structure as the simulator input is questionable. In addition, the interests of this study are focused on the interrelationship among program structures, scheduling algorithms, and memory conflicts with the goal of reducing or eliminating timing anomalies, rather than solving the timing anomaly problem of a particular program structure. It is felt that without losing generality, synthetic test models of program structures may be used. As illustrated in Figure 7-2, three test models are used. Model #1 is the most sequential, model #3 is the most parallel, and model #2 yields a somewhat in-between structure.

Each model is a 16-task program. For our investigation, we further assume a multiprocessor system of three processors sharing four memory modules. Without losing much generality, these memory modules shall have four pages per module, and each task shall occupy one page only. It is felt that a 3CPU X 4M configuration has all the ingredients of a multiprocessor system, and yet it is simple enough for analysis under simulation. No firm support is claimed in selecting the 16-task structure. However, one thing is apparent that more tasks offer more opportunity for multiprocessing which would tend to increase the system efficiency. On the other hand, more tasks require higher system overhead which reduces the system efficiency. These conflicting requirements suggest that the number of tasks partitioned out of a program would not be a very large value. Hence, 16 tasks are selected because they could produce enough multiprocessing activities for three processors and yet they would keep system overhead at a low level.

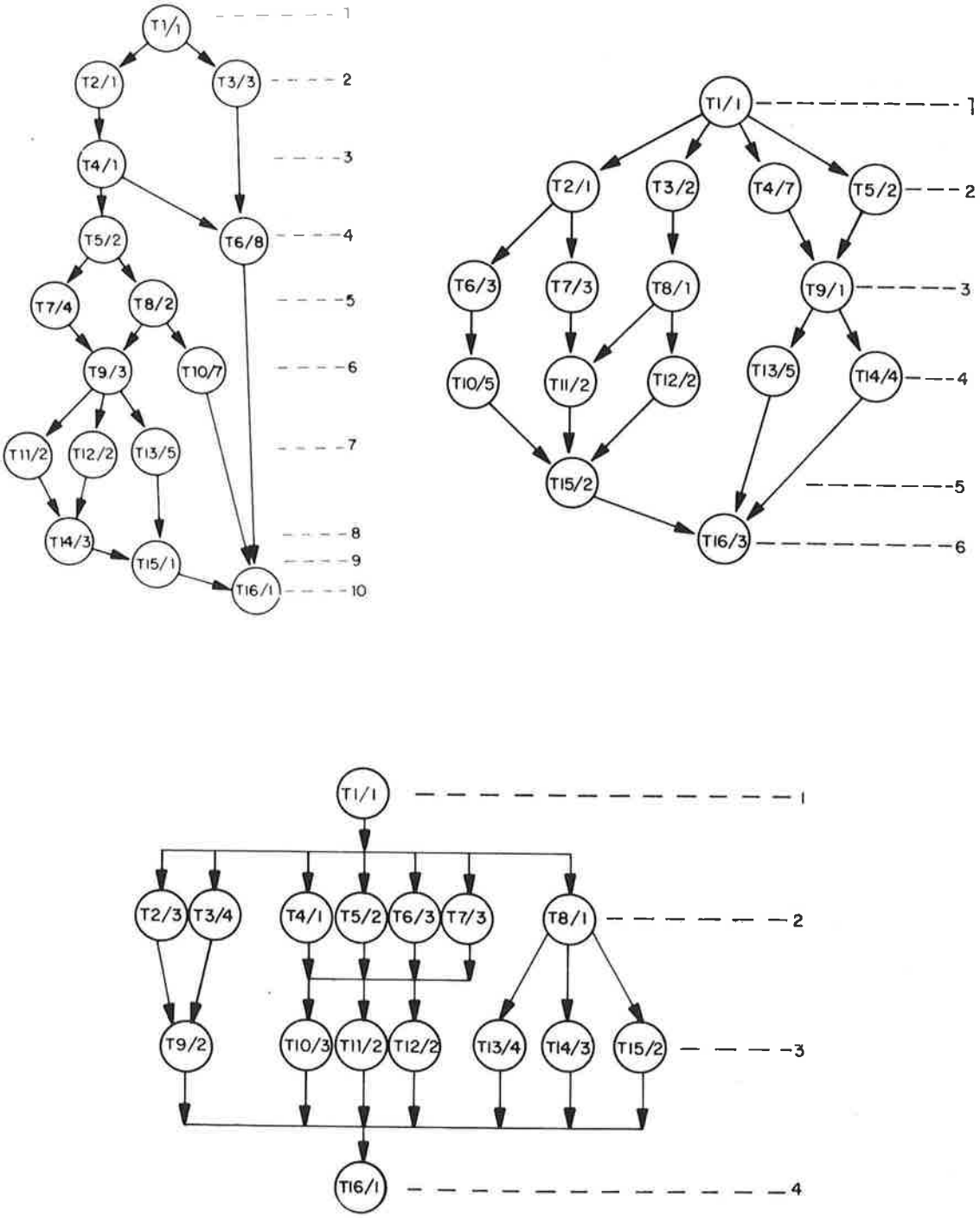



Figure 7-2 Test Models of Program Structure

## 7.3 ANALYSIS OF RESULTS

There are three models each of which will be allocated into memory modules by three memory allocation heuristics, which in turn, will be run by three dynamic scheduling heuristics through the simulator. In all, there will be 27 (3X3X3) simulation runs in which statistics on program execution time, memory conflict length and CPU activity time in percentage are collected, and tabulated in Tables 7-1, 7-2, and 7-3 respectively.

TABLE 7-1 AVERAGE PROGRAM EXECUTION TIME

| | | MODEL #1 | MODEL #2 | MODEL #3 |
|---|---|---|---|---|
| SEQUENTIAL SCHEDULING (RULE #1) | MASQ | 39.60 | 33.21 | 26.29 |
| | MAPP | 31.34 | 29.86 | 25.07 |
| | MASS | 31.64 | 31.30 | 24.35 |
| LPP FIRST THEN LSG SCHEDULING (RULE #2) | MASQ | 37.43 | 30.17 | 26.95 |
| | MAPP | 32.30 | 29.78 | 25.31 |
| | MASS | 31.46 | 28.37 | 24.02 |
| SELECT NEXT LPP WHEN IN CONFLICT (RULE #3) | MASQ | 45.12 | 31.84 | 25.52 |
| | MAPP | 37.07 | 29.41 | 22.70 |
| | MASS | 32.29 | 29.33 | 21.63 |

A number of observations can be made upon examination of this table. In general, programs with sequential or random memory allocation experience more memory conflict; hence, longer overall program execution time. Observe the results run on Model #1 whose program structure is more serial in nature; various scheduling rules do not seem to have much effect on the program execution time. This is reasonable because at any instance there will not be many tasks available to select by various scheduling rules. The availability of tasks is pretty much determined by the precedence relationship among tasks. When no specific attention is paid to memory conflict. This fact is properly reflected when scheduling Rule #3 is applied, and the system waits a long time to complete all tasks. The system improves equally well under either MAPP or MASS memory allocation heuristic. Since

the program structure is serial, it is relatively easy to allocate tasks out of potential conflict situations regardless of what allocation rule to use.

In running Model #3 of the parallel program structure, MASS allocation rule appears to be definitely superior to the MAPP and MASQ schemes. It is very interesting to note that regardless of memory allocation, program execution time is minimum when processors are instructed to wait as memory conflict situations are encountered. This is due to the fact that when a processor becomes free, the system has more parallel tasks ready to be executed. If memory conflict occurs in selecting the most desirable task by LPP and LSG rule, a processor simply goes on and tries to select the next desirable task to process. Since Model #3 is highly parallel, there is a high probability that a free processor will be able to select a task without memory conflict. When a MAPP or MASS memory allocator is used, the system has even better improvement.

The Model #2 runs seem to produce some random results, as indicated by short program processing times under Rule #1 with MAPP memory allocation and under Rules #2 and #3 with MASS memory allocation. Since the Model #2 program structure is neither highly serial nor highly parallel, it is understandable that its dynamic behavior would not be dominated by either characteristics. However, one interesting point revealed after some in-depth investigations, which may provide partial explanation of the system behavior, is the fact that the logic used in MASS is identical to the logic used in Rule #2; whereas, the logic used in MAPP is consistent with the sequential scanning logic used in Rule #1. We intuitively appreciate that the system would produce better results when the memory allocation scheme and the dynamic scheduling heuristic are consistent.

It is reasonable to assume that in order to take full advantage of multiprocessing, programs are to be segmented into highly parallel structures if at all possible. Under this premise, we select and recommend that Rule #3 be used as the scheduling rule and MASS be used to allocate tasks into memory modules.

To further compare the simulation results with a set of optimal schedules, Gantt Charts of the three models are shown in Figures 7-3, 7-4, and 7-5. Task execution times used in those Gantt Charts are minimum times. Since a uniformly distributed random number is added to these minimum task execution times, a factor of 1.5 should be adjusted to these optimal timing figures. Let $T_{pi}$ be the optimal execution time of program i, then

$$T_{p1} = 1.5 \times 21 = 31.5$$
$$T_{p2} = 1.5 \times 18 = 27$$
$$T_{p3} = 1.5 \times 14 = 21$$

The minimum program execution times produced by simulation for Models #1, 2, and 3 are 31.30, 28.37 and 21.63, respectively, which are very close to the extrapolated optimal value.

Table 7-2 illustrates the memory overlap times which are consistent with the program execution time. In almost all cases, sequential memory allocation produces the most severe memory overlap. Again, when memory allocation logic and scheduling heuristics are consistent, the least memory overlaps, both in mean and maximum values, are produced.

TABLE 7-2 MEMORY OVERLAP TIME

| | | MODEL #1 | | MODEL #2 | | MODEL #3 | |
|---|---|---|---|---|---|---|---|
| | | Mean | Max. | Mean | Max. | Mean | Max. |
| Sequential Scheduling (Rule #1) | MASQ | 1.15 | 7.68 | 0.87 | 7.43 | 0.84 | 5.70 |
| | MAPP | 0.03 | 2.70 | 0.35 | 3.83 | 0.52 | 5.78 |
| | MASS | 0.10 | 3.37 | 0.58 | 4.55 | 0.46 | 3.34 |
| LPP First Then LSG Scheduling (Rule #2) | MASQ | 0.84 | 6.09 | 0.38 | 5.11 | 0.97 | 5.43 |
| | MAPP | 0.25 | 4.44 | 0.24 | 3.40 | 0.70 | 6.43 |
| | MASS | 0.08 | 3.44 | 0.09 | 2.31 | 0.50 | 3.45 |
| Select Next LPP When in Conflict (Rule #3) | MASQ | - | - | - | - | - | - |
| | MAPP | - | - | - | - | - | - |
| | MASS | - | - | - | - | - | - |

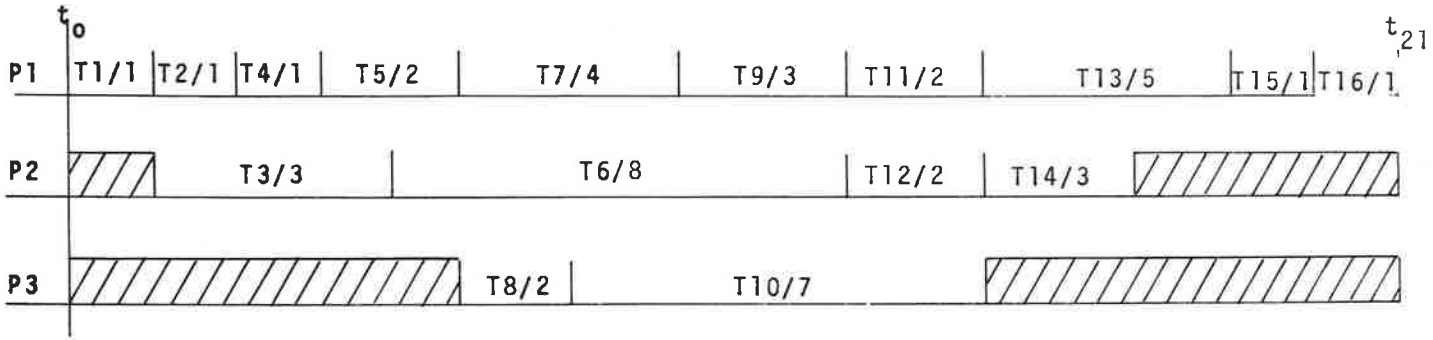



Figure 7.3 Model No. 1 Optimal Timing

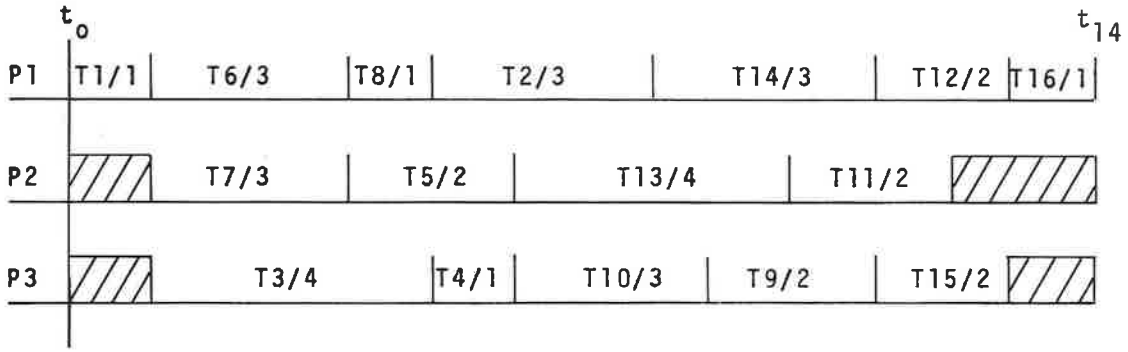



Figure 7.4 Model No. 2 Optimal Timing

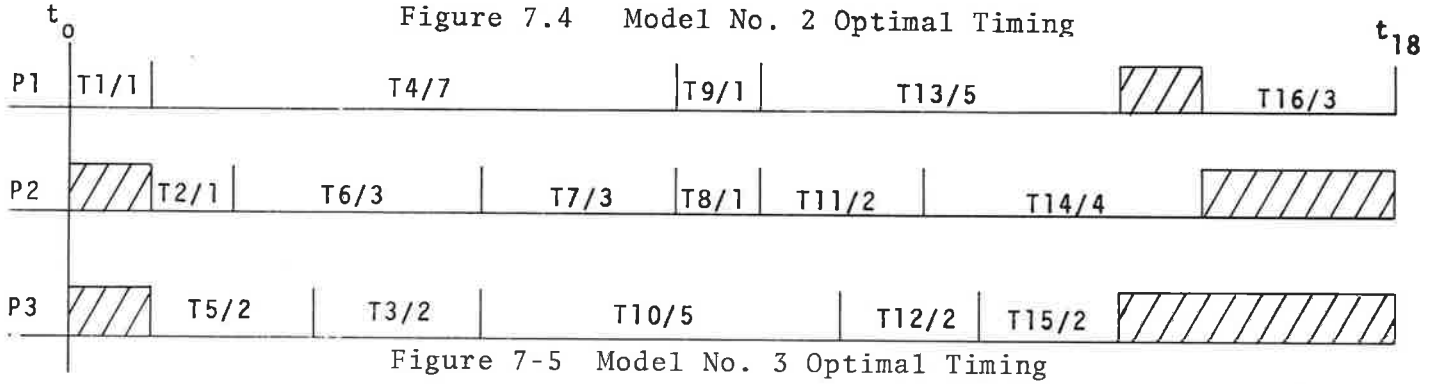



Figure 7-5 Model No. 3 Optimal Timing

TABLE 7-3 PROCESSOR ACTIVE TIME (%)

| | | MODEL #1 | | | MODEL #2 | | | MODEL #3 | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| Sequential Scheduling (Rule #1) | MASQ | 99 | 70 | 57 | 96 | 72 | 75 | 95 | 83 | 85 |
| | MAPP | 95 | 70 | 58 | 96 | 74 | 73 | 93 | 83 | 81 |
| | MASS | 96 | 72 | 57 | 98 | 72 | 75 | 97 | 80 | 82 |
| LPP and LSG Scheduling (Rule #2) | MASQ | 94 | 80 | 50 | 94 | 69 | 75 | 99 | 84 | 85 |
| | MAPP | 94 | 81 | 52 | 90 | 70 | 72 | 95 | 85 | 83 |
| | MASS | 95 | 72 | 56 | 93 | 70 | 70 | 98 | 83 | 81 |
| Select Next LPP When in Conflict (Rule #3) | MASQ | 87 | 52 | 19 | 90 | 61 | 55 | 92 | 72 | 51 |
| | MAPP | 97 | 66 | 29 | 94 | 63 | 64 | 95 | 76 | 70 |
| | MASS | 97 | 79 | 38 | 93 | 66 | 62 | 92 | 83 | 80 |

Study of Table 7-3 further confirms the observations made in previous paragraphs. It is interesting to note that most short program execution times accompany low processor utilizations which means efficient system performance. When more than one free processor is competing for available tasks, the scanning logic considers the low indexed processor first. Consequently, the utilization factor of processor #1 is always the highest.

In summary, we recommend that programs in a multiprocessor environment should always be segmented in a highly parallel structure. Individual tasks are to be loaded into memory modules by the static scheduling rule based on selecting the task with the longest precedence path first heuristic. If more than one task displays identical LPP's, then select the task with the largest successor group to be scheduled. This same LPP followed by LSG heuristic is to be applied to the dynamic scheduling rule also; however, in addition, in the case of memory conflict, the same rule should be applied again to select the next desirable task according to its LPP and LSG. If none is available, the processor should elect to wait. This combination yields a near optimal result with no memory conflict at all and a good portion of processor time free to spare. The processor spare time would

be used to handle interrupts so that external demands could be met.

# 8.0 SUMMARY AND CONCLUSION

## 8.1 TECHNOLOGY ASSESSMENT

The heart of the Associative Pipeline Multiprocessor System (APMP) is the associative memory array. The idea of a memory with content-addressable capability dates back to the early 50's and has always enjoyed a great deal of attention in the computer community through the years. Unfortunately, it has not yet reached the maturity to be widely accepted and implemented in computer systems; this is mainly because of the high cost. The basic associative memory array has been considered and reconsidered many times in various technologies such as cryogenics, multiaperature cores, tunnel diodes, and plate wires. The cost of these schemes was too high to allow widespread use of the associative memory.

With the advent of LSI technology and its rapid growth, the associative memory once more enjoys a renowned popularity since the number of a few additional circuit elements of a memory cell no longer influences the cost of the overall memory system. In fact, a few semiconductor manufacturers are offering associative memory array packages as their off-the-shelf items. Fairchild and Intel offer a 16 x 16 TTL AM package with a 30 ns access time. TI is marketing a MOS AM package which has a 16 eight-bit word organization and a 200 ns access cycle time. Honeywell developed a 256 word by 64 bit associative memory system using TI chips for the National Aeronautical and Space Administration. This MOS AM array requires complex driving and sensing circuitry to interface with outside TTL logic. The overall cycle time is about 350 ns. These AM arrays have "true" parallel search capability. By this we mean that the search is carried out in parallel within a field over all words. Goodyear, on the other hand, has developed a number of associative memory systems for the U.S. Air Force using plated wire technology. The search technique adopted is carried out on a bit-column basis over all words in parallel. To search on a field, the

technique has to perform bit-column searches sequentially until the entire field is covered. Recently, Goodyear has developed an ingenious method of realizing associative memory operation by means of random access memory arrays. Since it is still company proprietary information, the system implications and ramifications are not known at this time. General Electric recently reported their successful laboratory development of a 512-cell MOS associative array organized 32 words by 16 bits on a chip. The read/write as well as match cycle times are 200 ns; and the associative memory array interface is TTL compatible. This latest development has made a great stride in operating speed, interface flexibility and cell size reduction which could conceivably bring the cost down to an attractive level such that associative processing techniques could be economically incorporated into new computer systems.

## 8.2 FUTURE RESEARCH AREAS

This study has identified an environment, proposed a computer architecture for its application, acknowledged a few key problem areas and suggested an heuristic approach for their solutions. Due to the complex nature and wide scope of the subject matter, a number of interesting aspects were intentionally left unanswered. In order for the associative processor to have widespread acceptance, there is an urgent need to develop denser, less costly, and faster AM array chips in solid-state LSI technology. Unlike the random access memory, the building up of a workable associative memory model out of AM array chips presents a non-trivial interconnection and packaging problem.

The entire error handling philosophy deserves undivided attention. The question of optimal schemes for error detection, location and isolation, remains open. Once an error has been detected and isolated, the question of signaling the failure to the system is also pertinent so that the failed instruction may be retried. When unsuccessful, the task may be restarted from an appropriate roll-back point, automatically perhaps. If a failure persists, the system would automatically reconfigure the

overall resources into either a fail-safe (when spares are available) or a fail-soft (system performs at a reduced capacity) mode of operation.

Given a multiprocessor configuration, how should a program be segmented into task-set in order to achieve efficient operation? The results obtained by our simulation seem to indidate that the more parallel tasks a program can be segmented into, the more efficient this system is. However, with system overhead, associated with the execution of tasks, it is likely that there is a point of diminishing return beyond which further segmentation decreases the system efficiency. Working at a lower level, how should a task be structured in order to execute multiple data sets in parallel taking full advantage of the pipeline associative processor's capability?

When there are not enough processors to cope with active tasks, sharing multiprocessor resources among independent tasks of independent programs would still produce unpredictable timing anomalies. Although some conjectures were made in Chapter VI, further study in great depth is needed to provide better workable schemes than simple priority because this problem is particularly akin to real-time applications.

The least known subject in multiprocessing and parallel processing is in the area of programming language. At present, most parallel processor manufacturers are working on fundamental assembly languages. To this author's knowledge only two attempts in developing higher level languages for parallel machines are being made by Bell Telephone Laboratories in Parallel FORTRAN and by Sanders Associates in APL. In brief, the development in this area is still in the embryonic stage, and it certainly offers tremendous opportunity for further research.

## 8.3 CONCLUSIONS

The U.S. Continental air traffic control automation system was chosen to provide a real-time application environment. One of the major contributions of this study is the formulation of a novel computer architecture to achieve greater performance improvement over the present ATC automation system by parallel

processing. The heart of the computer structure is an associative memory capable of simultaneous search on selected fields over all storage words. To augment the associative memory with fast arithmetic capability, a high-speed arithmetic unit is integrated in with the associative memory module, and is capable of processing multiple data streams under the command of a single instruction. It is difficult to dispute the power of the associative memory in terms of logical parallel processing. However, in appraising the arithmetic parallel processing power of an associative memory, carried out in a bit-column basis, common opinions hold that associative memory works well only when the number of data sets in extremely large. The streaming of pipelineing technique proposed in this study would set this argument to rest when a data set of moderate size came streaming through the arithmetic unit at high speed.

For the reasons of system availability and modular expandability, this associative pipeline processor is proposed to connect with conventional processors and random access memories forming a multiprocessor organization. In a complex real-time application such as air traffic control, various functions can be categorized according to their processing characteristics. Regular and periodic parallel processing tasks with multiple data sets are assigned to PAP modules, whereas conventional processors take on those jobs arriving in random and of sequential nature. Problems associated with multiprocessors are reviewed with particular emphasis on execution time anomalies and memory conflicts. A directed graph model is used from which simple heuristic rules are established for memory allocation and dynamic task scheduling, so that near optimal performance can be achieved with minimal system overhead. The memory allocation and heuristic scheduling schemes are simulated. The results analyzed closely follow the predicted system behavior.

Finally, the dynamic scheduling algorithm can be implemented by means of storing essential system parameters associated with each task in an associative memory module. Taking full advantage of the search capability, the executive routine could

schedule tasks and update system table entries in the shortest possible time; hence, it further reduces system overhead. Technology risks in the fabrication of associative memories are assessed, and none are found, except that the cost is still a major hurdle.

There are many areas of research still open with respect to this associative pipeline multiprocessor structure, to say, nothing of the many possible alternative parallel processor structures. To most of the questions, there are no pat "right" or "wrong" answers. There are, rather, trade-offs to be investigated and techniques to be developed. The analyses and methods proposed in this study are believed to be of direct usefulness in the design of the next generation of computers.

# APPENDIX A

# GASP II - A FORTRAN-BASED SIMULATION LANGUAGE

INTRODUCTION

GASP II is a practical and useful simulation language particularly designed for discrete-event simulations. Being FORTRAN-based, it is relatively easy to learn due to the fact that GASP II is nothing more than a group of FORTRAN subroutines and functions. Its simulation concepts are quickly grasped because they are presented in the familiar framework of FORTRAN.

This language has developed over the past seven years at Arizona State University from the original GASP developed at U.S. Steel[64]. The versions of GASP introduced here are written in FORTRAN IV for the H-516 computer. GASP II has also been used extensively on IBM-1130, GE, and CDC series computers, and is being used at many universities, industries and government installations.

The GASP Program Structure

GASP depicts the system under simulation as made up of entities that are described by attributes, and are related through the use of files. The status of the simulated system can be changed if entities are created or destroyed, if attribute values change or if file contents are altered. When a change of state of any element causes a change of status in the overall system, it is called an Event. To further clarify these definitions, let us examine a customer-server queuing system. Customers and servers are entities. Arrival rates and service rates are the corresponding attributes. Arrival of a customer and end of service, which change the system status are considered to be Events.

GASP, which consists of twenty-four FORTRAN subprograms, is organized to provide six specific functional capabilities required by every simulation:

1. Event Control
2. Information Storage and Retrieval
3. System State Initialization
4. Program Monitoring and Error Reporting

5. Statistical Computations and Report Generation
6. Random Variable Generation

and a timing mechanism which sequences automatically the simulated system from event to event. In a real sense, GASP should not be called a simulation language but rather a set of facilities which make a simulation job much easier.

Figure A-1 illustrates a typical GASP program high-lighting the interaction between those programmer written program segments and those facilities provided by GASP.

## THE FILING ARRAY-NSET

The heart of the GASP program structure is a two-dimensional filing array called NSET. The term entry, represented by columns of NSET, is used to describe events and entities whereas rows are used to store the associated attributes. The last two rows, however, are used to store two positional pointers which indicate the relative position of the entry with respect to other entries in the same file. These pointers are called predecessor and successor pointers.

Figure A-2 illustrates an example of an NSET (6,9) array. In this figure we see that column 3 "points" to column 9; i.e., the value in the successor row of column 3 is 9, which means that the entry in column 9 comes after the entry in column 3. The predecessor value for column 3 is 9999. This is a code that indicates there is no entry before the entry in column 3. When an entry has a predecessor value of 9999, we say it is the <u>first entry</u> in the file. Each file has a first entry, which is defined by the variable MFE. MFE is a one-dimensional array with as many elements as there are files in the filing array.

The code 7777 is used to indicate an entry that has no successor and, hence, is the last entry in the file. We see that the entry stored in column 5 is the last entry. Variable, MLE, is used to define the last entry of a file. MLE is a dimensioned variable, and there is a last entry associated with each file.

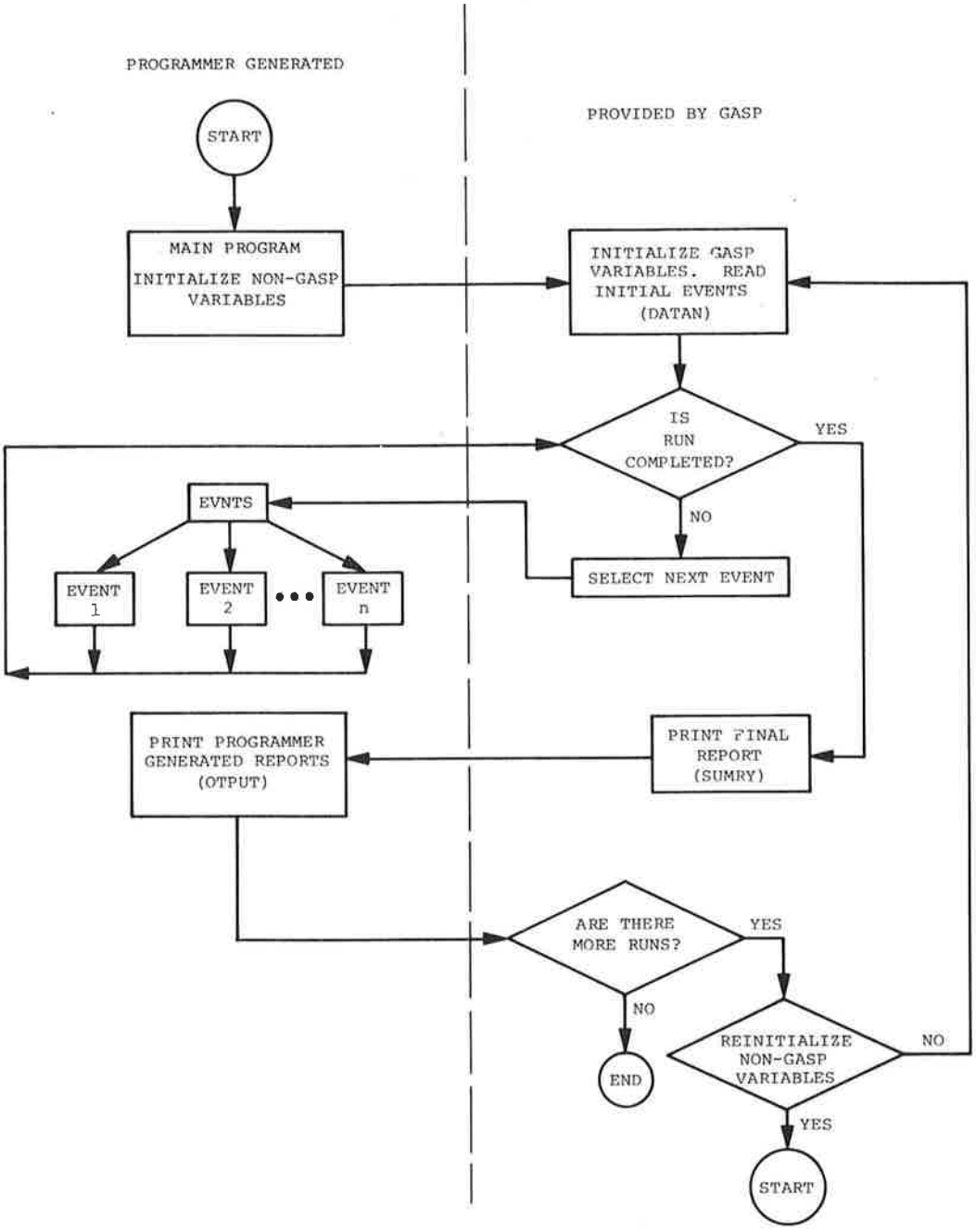



Figure A-1 A Typical GASP Program

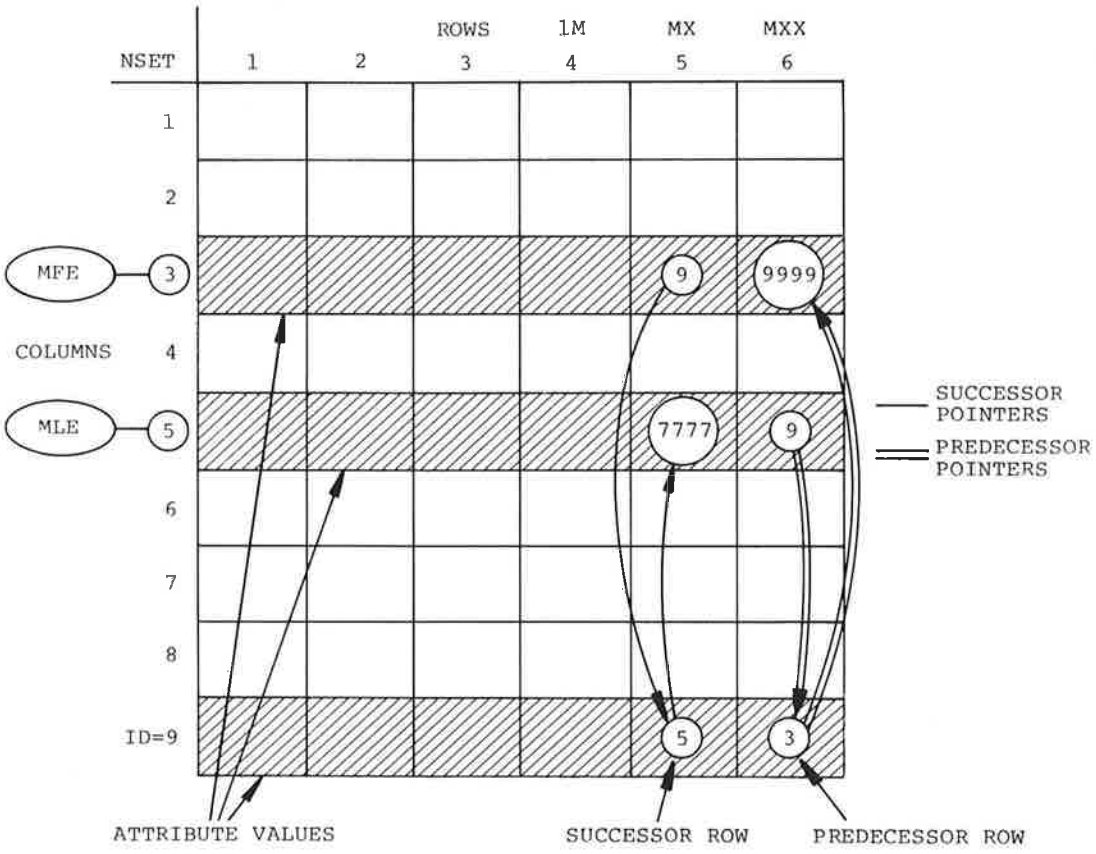



Figure A-2 GASP File, NSET (6,9) Structure

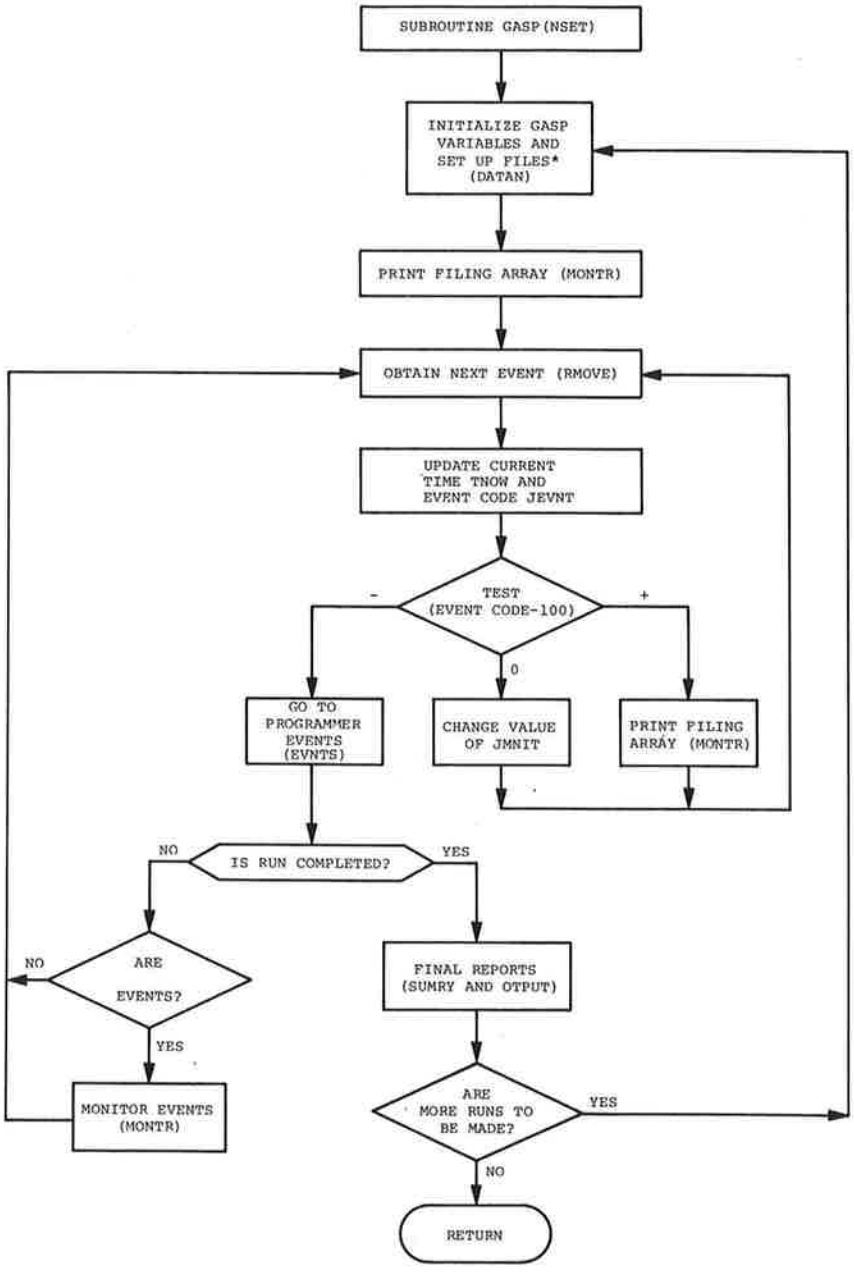
GASP EXECUTIVE

Of the twenty-four FORTRAN subprograms, one is SUB-ROUTINE GASP(NSET) which is the master control routine. It starts the simulation, selects events, sequences time, controls the monitoring of intermediate simulation results and initiates the printout of the final output when the simulation has been completed. Subroutine GASP is called only by the main program, which is written by the programmer. After control is turned over to the GASP for a simulation run, it is not returned to the "main" program until the run is completed.

Referring to Figure A-3 GASP first calls subroutine DATAN, which initializes all GASP COMMON variables. DATAN also reads in initial events and initial file entries. The initial file status is printed by GASP, using subroutine MONTR. GASP begins a simulation by using subroutine RMOVE to remove the next (first) event from the event file; thus, at least one initial event must be inserted into the event file in subroutine DATAN. The event file is always file number 1. The statement CALL RMOVE(MFE(1), 1,NSET) removes the next event and stores its attributes in ATRIB. The time of an event and the event code must be the first two attributes of an event stored in the event file, Therefore, ATRIB(1) and ATRIB(2) are given these values, which GASP transfers to the variables used to describe current time TNOW, and the event code JEVNT.

Next, a test is made on the event code to determine if the event selected is a monitor event or a programmer's event. GASP II defines two event codes for monitoring. Event code 100 causes JMNIT, a 0 or 1 variable, to change value. If JMNIT is 1, each event is printed as it occurs. Event code 101 causes a call to be made to subroutine MONTR, which is used for debugging and to obtain intermediate results. A programmer can modify MONTR to obtain additional information during a simulation.

If the event code is less than 100, an event written by the programmer is to take place. GASP calls the event subroutine through the subroutine EVNTS, which is also written by the programmer. The event code is transferred to the subroutine




*THE GASP SUBPROGRAMS EMPLOYED TO HELP ACCOMPLISH THESE FUNCTIONS ARE GIVEN IN PARENTHESIS.

Figure A-3 General Flow Chart of Subroutine GASP

EVNTS as an argument. Subroutine EVNTS calls the appropriate event. After the functions associated with the event are completed, control is returned to GASP through the subroutine EVNTS.

At this point, codes are checked to determine whether the simulation is completed. If MSTOP (recall GASP constants are initialized in DATAN) is zero, a special event must be written to end the simulation. This event should set MSTOP to 1 to end the simulation run. If MSTOP is greater than zero, the simulation is completed when TNOW is greater than or equal to TFIN. TFIN is a GASP variable defining the ending time of the simulation. (TFIN is not used at all when MSTOP is zero.) When TFIN is used as the stopping condition, subroutine SUMRY is automatically called to compute and print the final GASP reports. Subroutine OTPUT is also called. OTPUT must be written by the programmer; it is used to allow the programmer to print information not produced by subroutine SUMRY. If it is desired to bypass SUMRY and OTPUT, the control variable, NORPT, should be set greater than zero. If SUMRY and OTPUT are desired, set NORPT equal to zero.

When the final reports have been printed by SUMRY and OTPUT, one simulation run has been made. The variables NRUN and NRUNS are used to indicate the simulation run number and the number of runs remaining (including the one being made). When NRUNS is 1, control returns to the main program, where it is the programmer's responsibility to do one of the following: (1) call EXIT, (2) reinitialize non-GASP variables and then call GASP, or (3) call GASP. As long as NRUNS is greater than 1, DATAN is called. A control variable NEP specifies the GASP variables that need to be reinitialized by DATAN. This is explained in the discussion of DATAN. Through this procedure, a number of runs can be made with the same initial values of the non-GASP variables and then another set of runs made with new initial values for some or all of them. NRUNS is initialized in a READ statement in subroutine DATAN. When NRUNS equals zero, execution is stopped.

| NAME | DESCRIPTION | FLOW CHART FIGURE |
|---|---|---|
| HUSS | Heuristic Scheduling Simulation Program | Figure B-1 |
| NDTASK | End-of-Task | Figure B-2 |
| MCFLT | Memory Conflict Detection | Figure B-3 |
| PPSG | Longest Precedence Path (LPP) First Followed by Largest Successor Group (LSG) Rule | Figure B-4 |
| WAIT | Use PPSG Rule, Let Process Wait If Memory Conflict | Figure B-5 |
| MASQ | Memory Allocation via Sequential Assignment | Figure B-6 |
| MASS | Memory Allocation via Static Scheduling | Figure B-7 |
| OVRLAP | Compute Memory Overlap | Figure B-7 (Con't) |
| MAPP | Memory Allocation via Precedence Partition | Figure B-8 |
| COPLAT | Compute Partition Level and Allocation | Figure B-8 (Con't) |

GASP SUBPROGRAMS

A functional breakdown of the GASP II subprograms is shown in the following table. The GASP executive controlling the overall simulation, has been introduced in the previous section. Subroutine DATAN initializes all GASP variables and reads all initial events and entities into the filing array NSET.

| FUNCTION | SUBPROGRAM |
|---|---|
| GASP Executive | SUBROUTINE GASP(NSET) |
| Initialization | SUBROUTINE DATAN(NSET) |
| Information Storage and Retrieval | SUBROUTINE SET(JQ,NSET)<br>SUBROUTINE FILEM(JQ,NSET)<br>SUBROUTINE RMOVE(KCOL,JQ,NSET)<br>SUBROUTINE FIND(XVAL,MCODE,JQ,<br>JATT,KCOL,NSET) |
| Data Collection | SUBROUTINE COLCT(X,N,NSET)<br>SUBROUTINE TMST(X,T,N,NSET)<br>SUBROUTINE HISTO(X1,A,W,N) |
| Statistical Computations and Reporting | SUBROUTINE PRNTQ(JQ,NSET)<br>SUBROUTINE SUMRY(NSET) |
| Monitoring and Error Reporting | SUBROUTINE MONTR(NSET)<br>SUBROUTINE ERROR(J,NSET) |
| Random Deviate Generators | SUBROUTINE DRAND(ISEED,RNUM)<br>FUNCTION UNFRM(A,B)<br>FUNCTION RNORM(J)<br>FUNCTION RLOGN(J)<br>FUNCTION ERLNG(J)<br>SUBROUTINE NPOSN(J,NPSSN) |
| Other Support Routines | FUNCTION SUMQ(JATT,JQ,NSET)<br>FUNCTION PRODQ(JATT,JQ,NSET)<br>FUNCTION AMIN(ARG1,ARG2)<br>FUNCTION XMAX(IARG1,IARG2)<br>FUNCTION AMAX(ARG1,ARG2) |

## Data Collection

Three subroutines are provided for collecting data during a simulation. Subroutines COLCT and TMST are used to collect data from which estimates can be made of the parameters of a distribution describing the variables of a simulation. The minimum and the maximum values of the variables are also computed. Subroutine HISTO is used to classify values of a variable into given cells for preparation of a histogram showing the frequency with which the variable was within a given range. COLCT, TMST, and HISTO each collect data throughout a simulation run. For each variable that data is collected, subroutine SUMRY prints out the appropriate information at the end of simulation.

Subroutine COLCT and TMST are used to collect information on two different types of variables. When the value of a variable is a sample value of an attribute, subroutine COLCT is used. Examples would be the waiting time of a customer in a queuing situation or the test score of a student in a simulated classroom. A value of a variable that has persisted over a period of time can be collected in subroutine TMST. The number of customers in a system or the status of a server would be examples.

## Statistical Computations and Reporting

Two subroutines perform statistical computations and report their results. Subroutine PRNTQ computes information about file usage during a simulation. Statistics are automatically kept in each file and are printed when subroutine PRNTQ is called. For data collected in subroutines COLCT, TMST, and HISTO, subroutine SUMRY computes the desired statistical information and prints a summary report. PRNTQ and SUMRY do not alter the values of the statistical storage areas nor the filing area. They can be used by the programmer at any time during a simulation to obtain statistical information.

### Monitoring and Error Reporting

Subroutines MONTR and ERROR assist the programmer in debugging and tracing simulation programs. Both are convenient subroutines for the programmer to alter in order to obtain additional information during or at the conclusion of a simulation run.

### Random Deviates Generators

Subroutine DRAND generates an uniformly distributed random variable in the interval 0 to 1. Function UNFORM(A,B) also generates a deviate from a uniform distribution in the interval A to B. The rest of the generators are three functions and a subroutine, which generate deviates from a normal distribution, a lognormal distribution, an Erland distribution and a Poisson distribution correspondingly. These generators all use as arguments the parameters stored in a row of the array PARAM.

### Other Support Routines

GASP II includes the functions: AMIN(ARG1,ARG2), which finds the minimum of two floating point variables; XMAX (IARG1, IARG2), which finds the maximum of two fixed point variables; and AMAX (ARG1,ARG2), which finds the maximum of two floating point variables.

Two additional support functions, SUMQ and PRODQ, that obtain cumulative information concerning attributes in specific files are included in GASP II.

# APPENDIX B

# HEURISTIC SCHEDULING SIMULATION PROGRAM

The heuristic scheduling simulator is written in FORTRAN IV within the framework of GASP simulation facilities which are also written in FORTRAN IV language. It is intended for wide utilization and easy adaptation on FORTRAN-equipped general-purpose computers.

The simulator consists of a simple main program and a set of subroutines. The logic flow of the main program is shown in Figure B-1. Upon entering the main program, a set of data is read in and followed by selection of one memory allocation scheme via subroutine call. The memory allocation routine returns to the main program, a Memory Assignment Table (MAT) which is then passed on to the simulation management GASP subroutine called GASP(NSET). The dynamics of the simulation, however, is carried out by a subroutine called NDTASK (NSET) which depicts the end-of-task event and conducts all necessary activities such as status updating, next task selection, scheduling, execution time calculation, memory conflict detection, etc.

The various routines and subroutines used, except GASP routines which are described in Appendix A, are tabulated, and their logic flow charts and FORTRAN listings are attached.

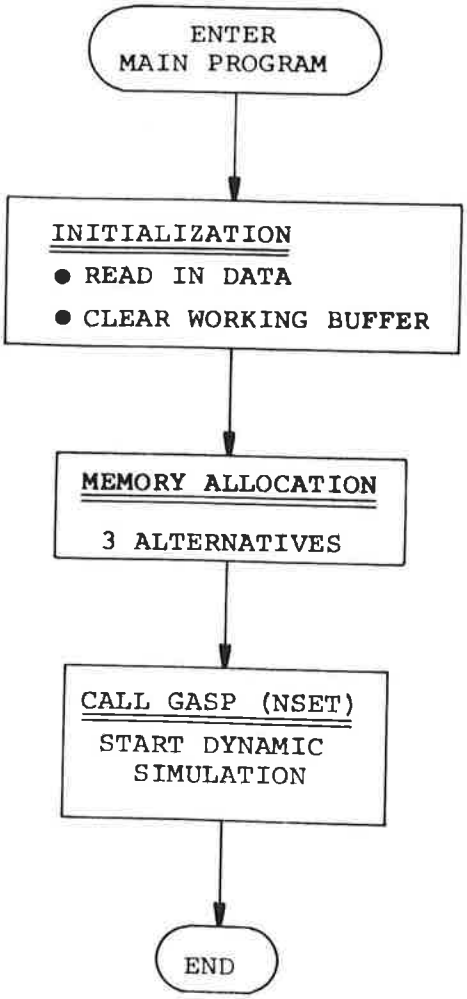



Figure B-1 Heuristic Scheduling Simulation Program

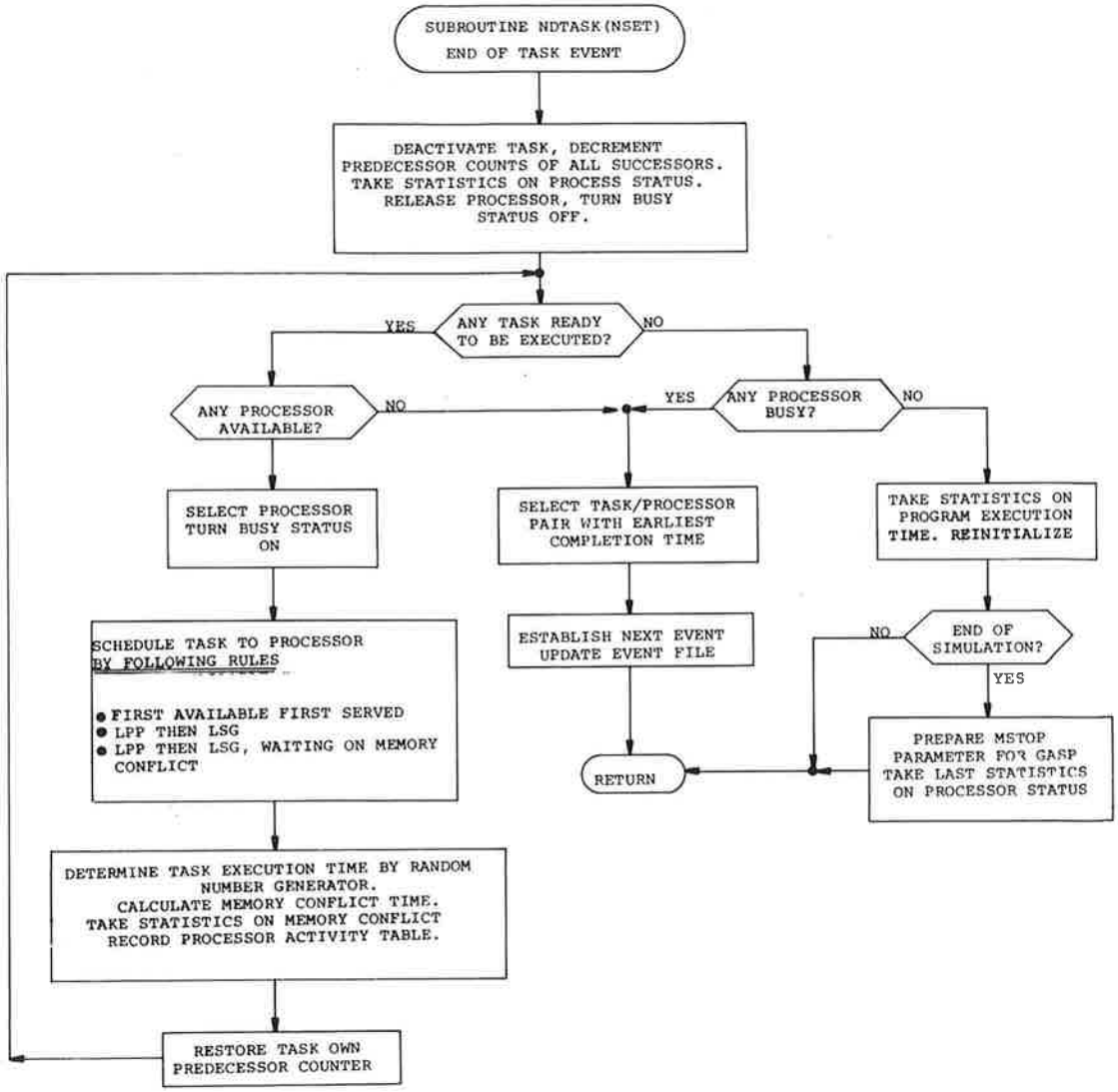



Figure B-2 End-of-Task Subroutine

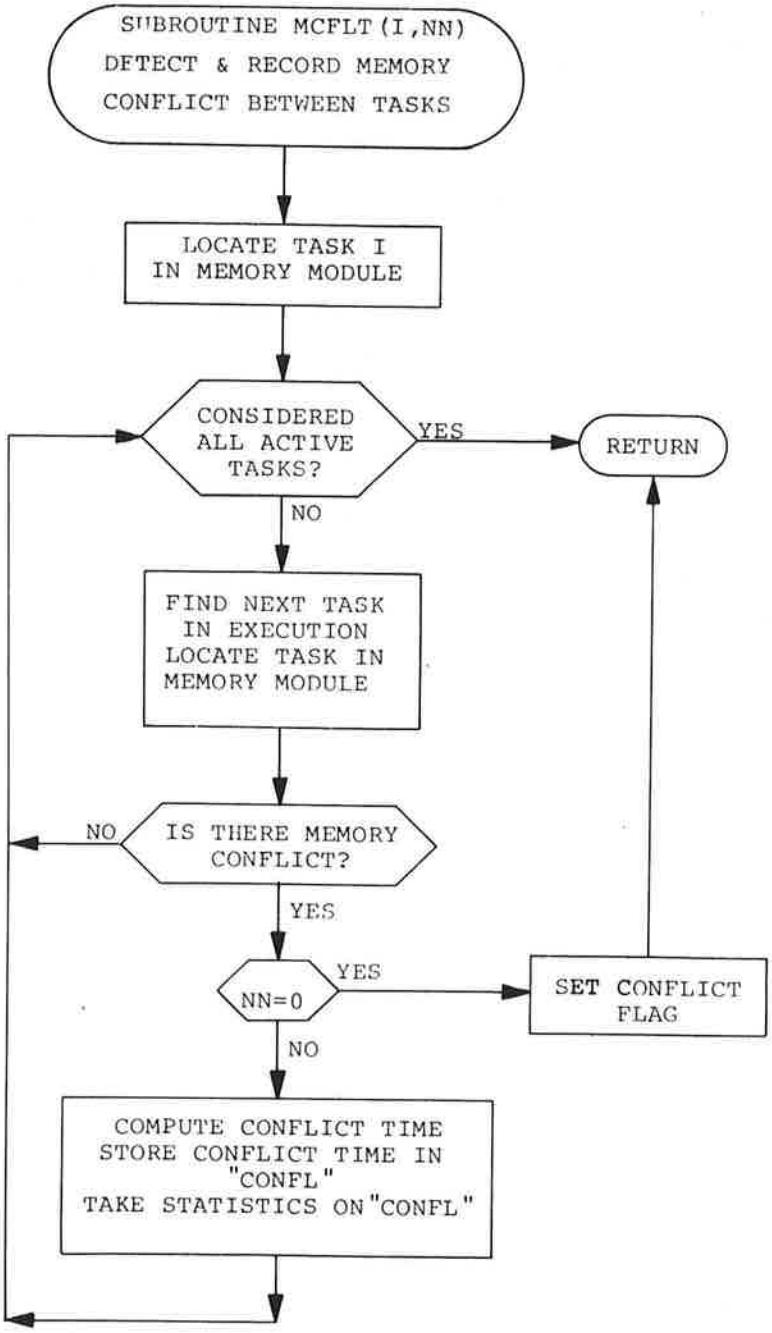



Figure B-3 Memory Conflict Subroutine

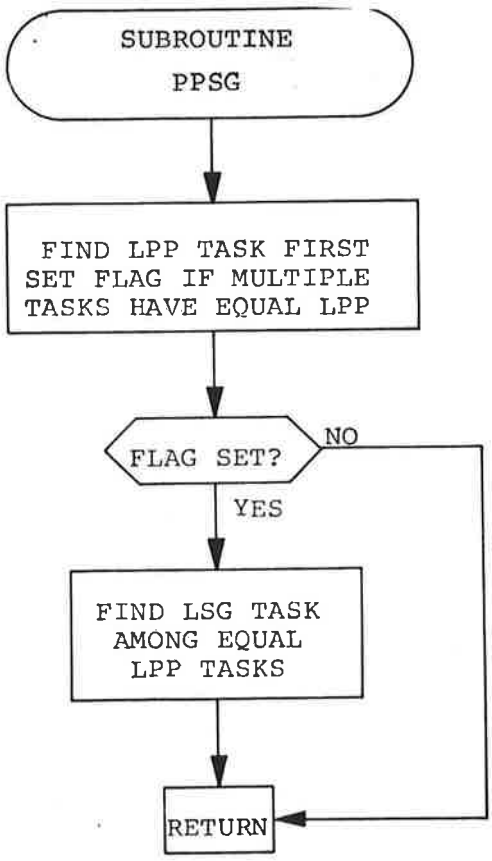



Figure B-4 Longest Precedence Path (LPP) - Largest Successor Group (LSG) Scheduling Rule

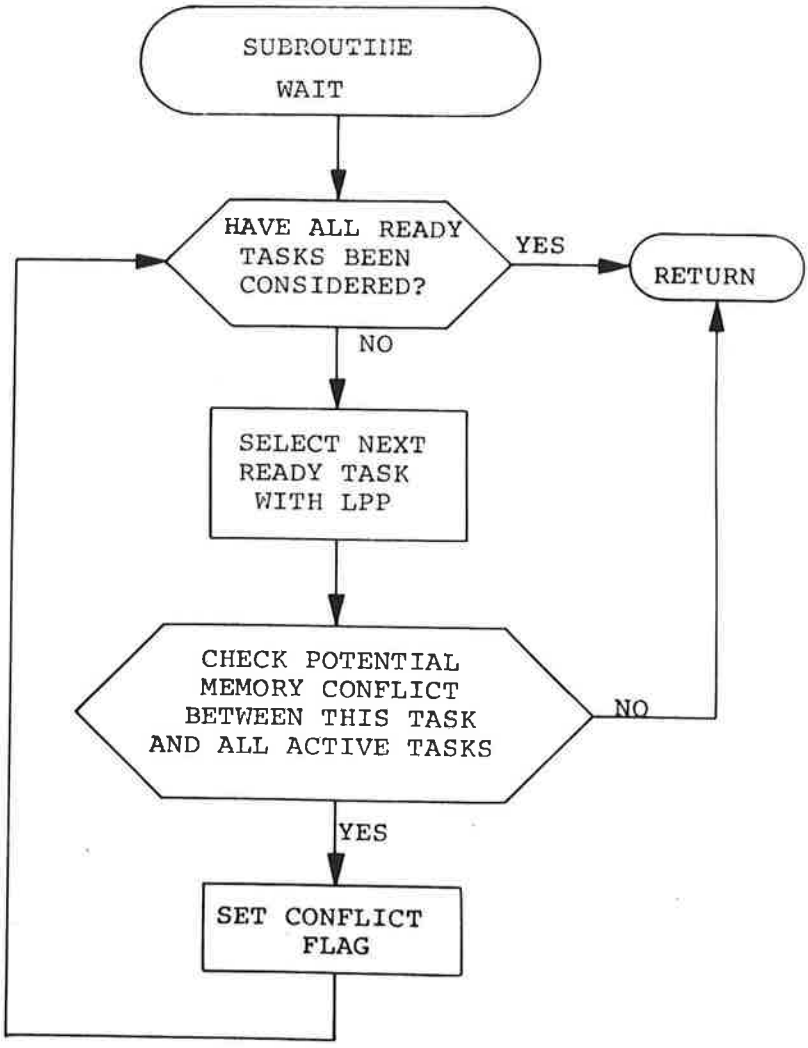



Figure B-5 Wait If Memory Conflict Scheduling Rule

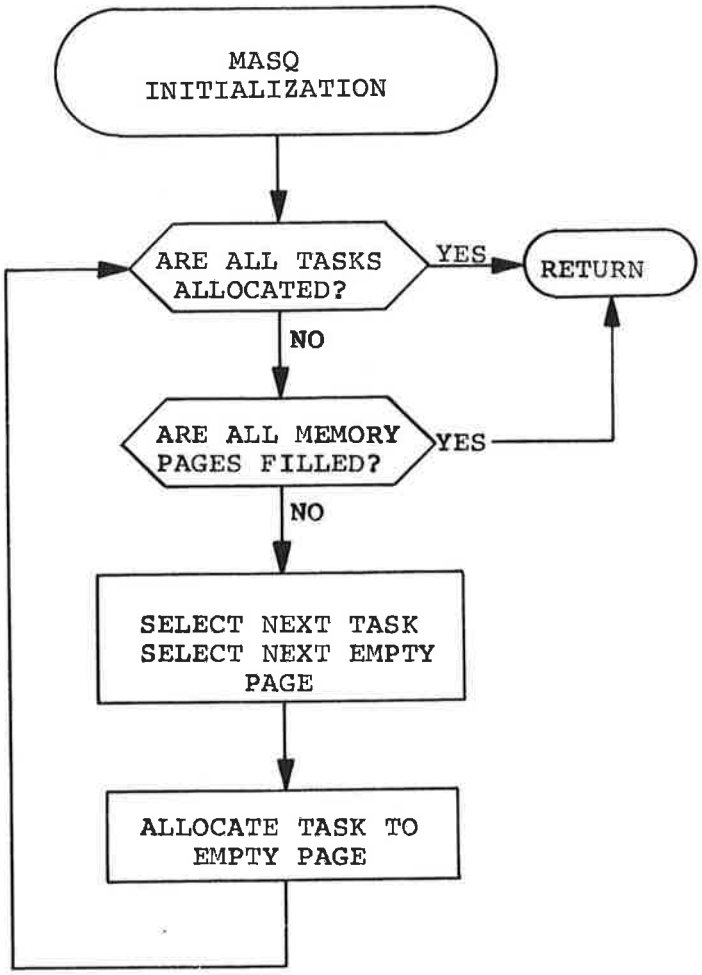



Figure B-6 MASQ - Sequential Memory Allocation

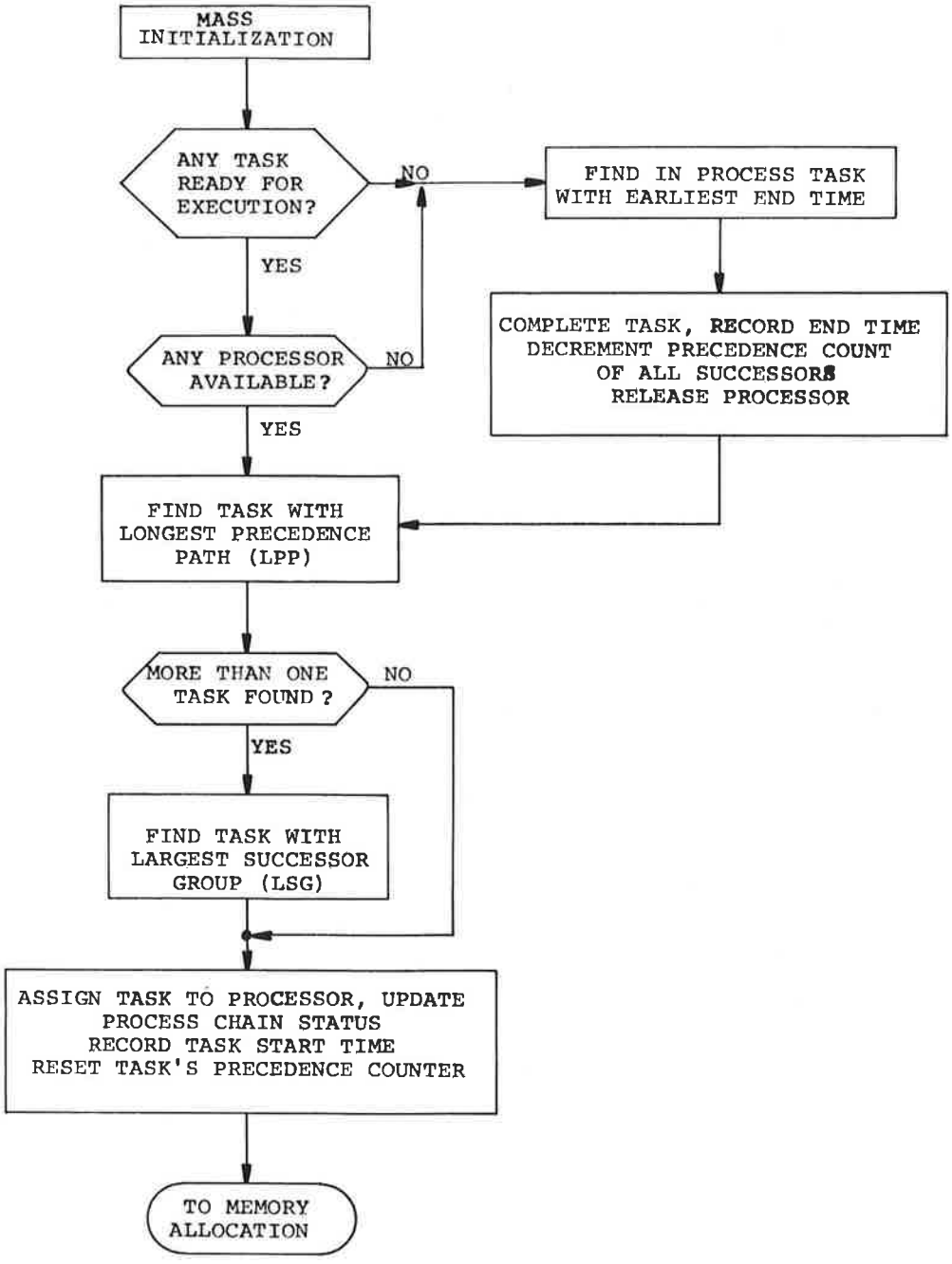



Figure B-7 MASS - Memory Allocation Via Static Scheduling

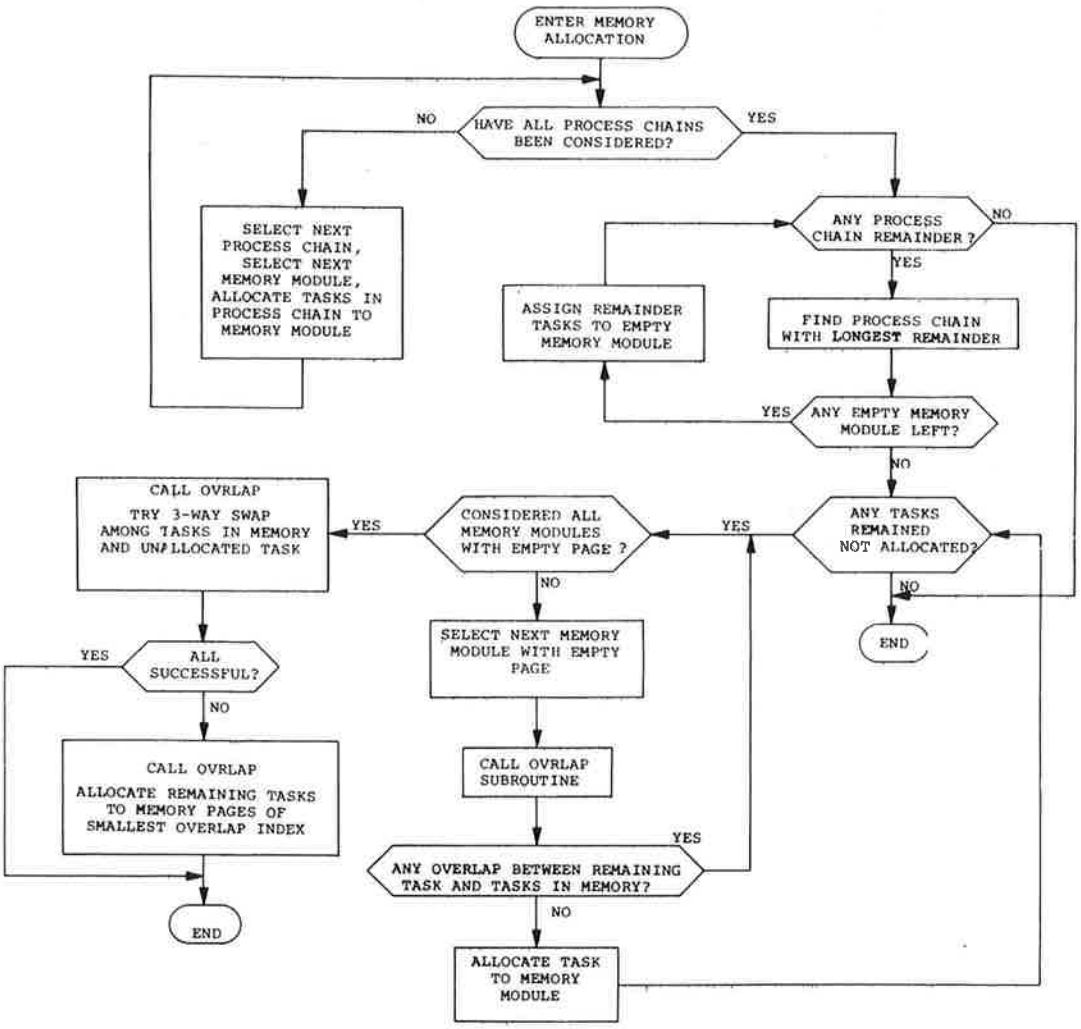



Figure B-7 (Continued) MASS-Memory Allocation Via Static Scheduling

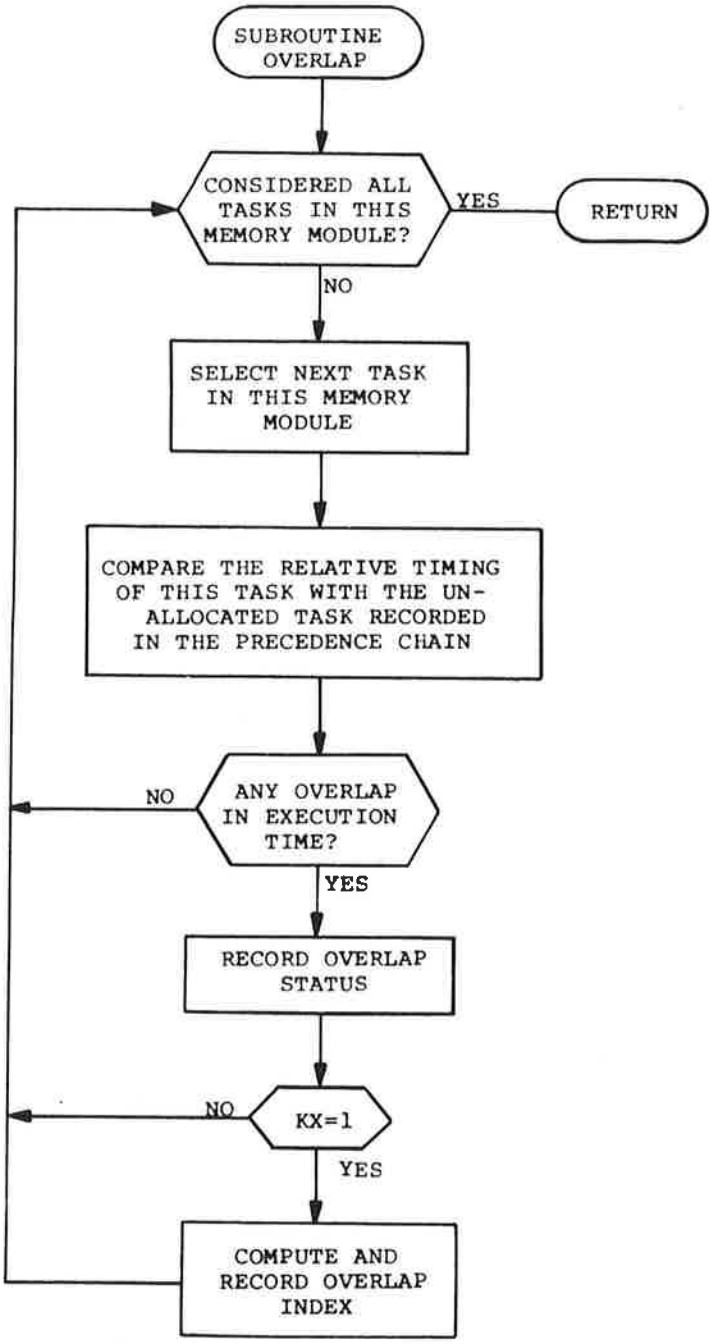



Figure B-7 (Continued) OVRLAP - Compute Memory Overlap Subroutine

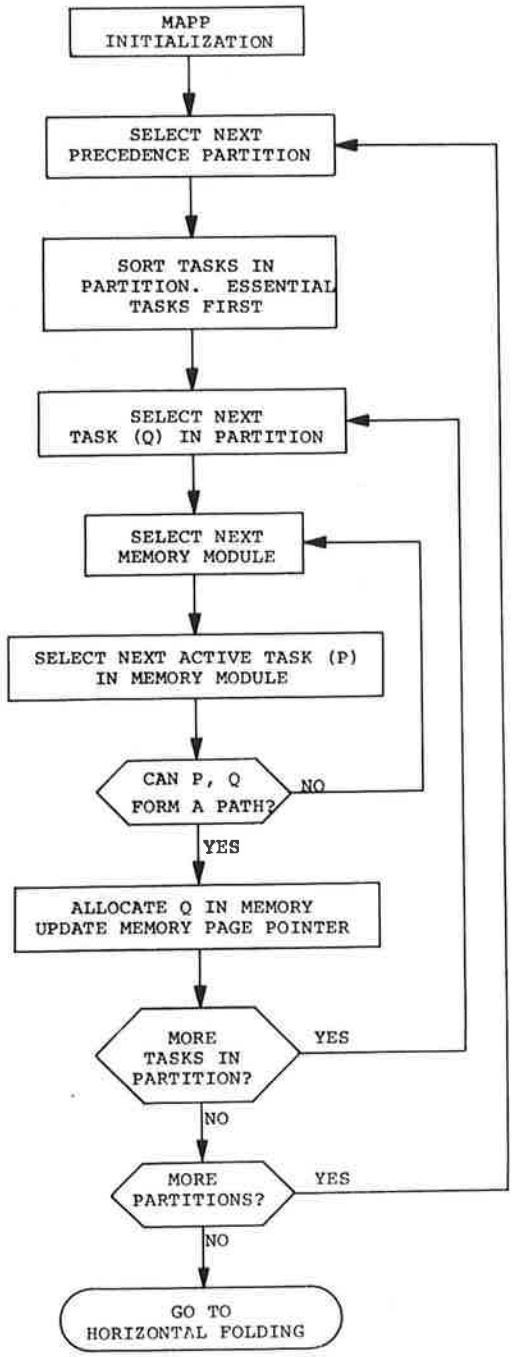



Figure B-8 MAPP - Memory Allocation Via Precedence Partition

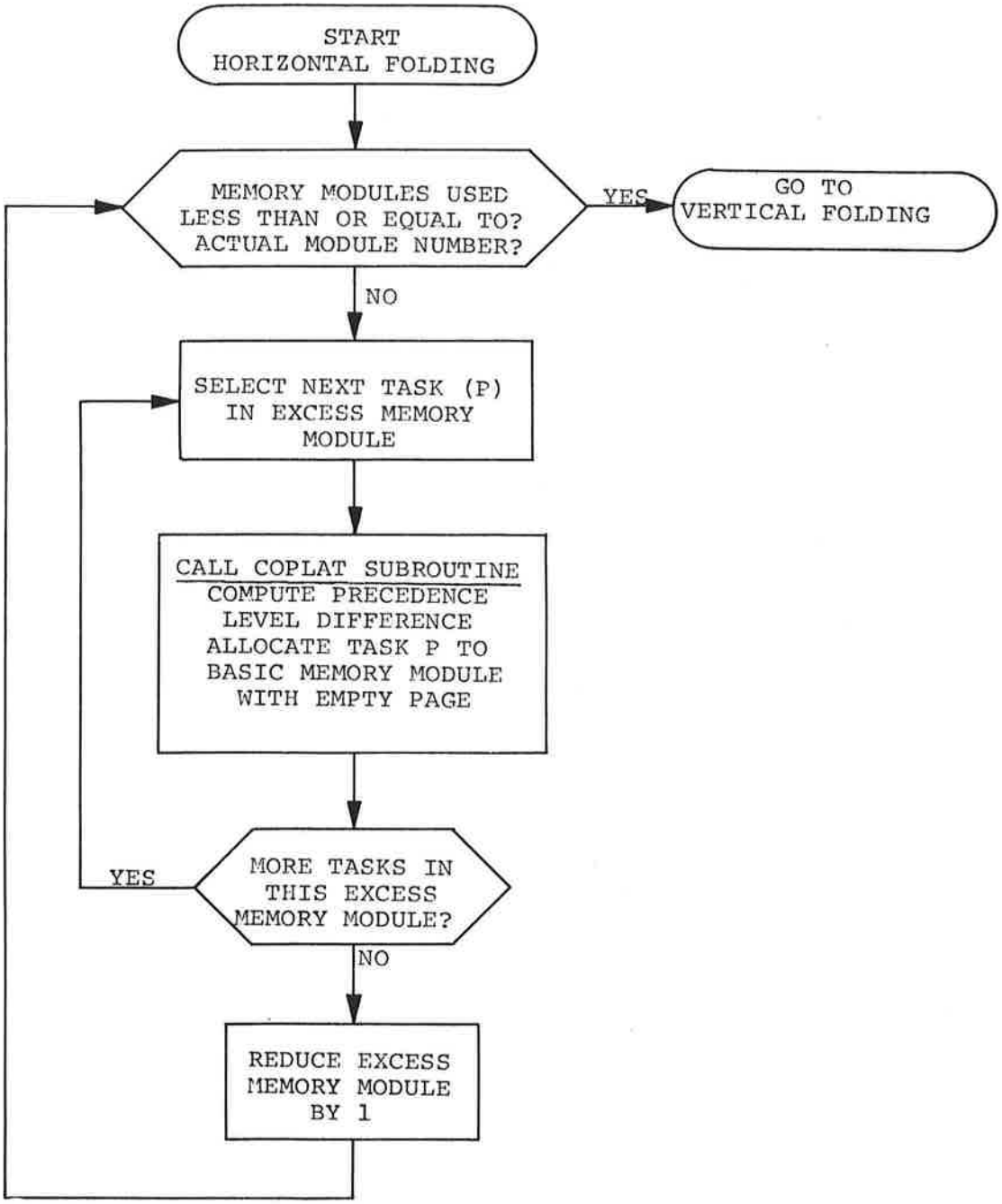



Figure B-8 (Continued) Horizontal Folding

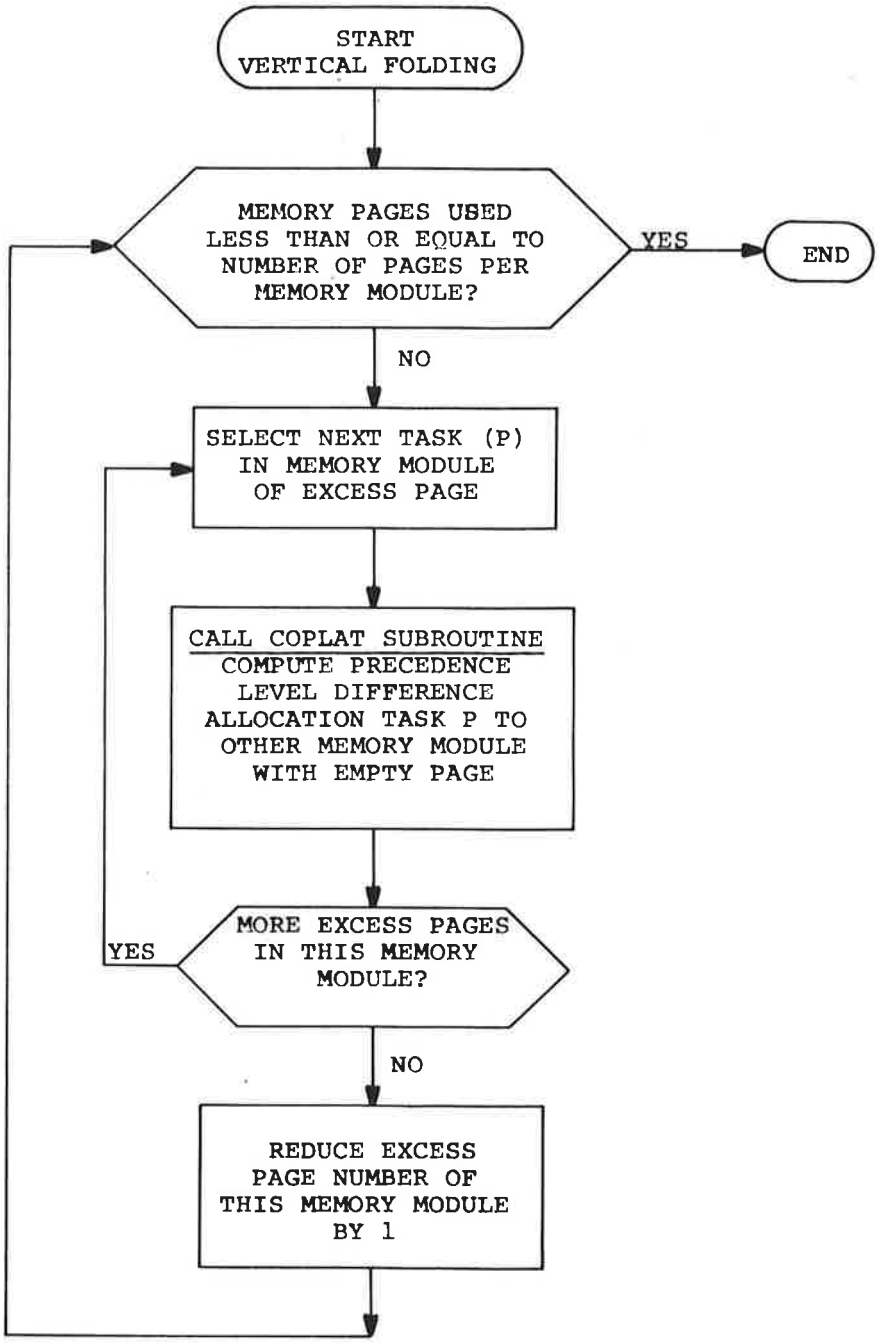



Figure B-8 (Continued) Vertical Folding

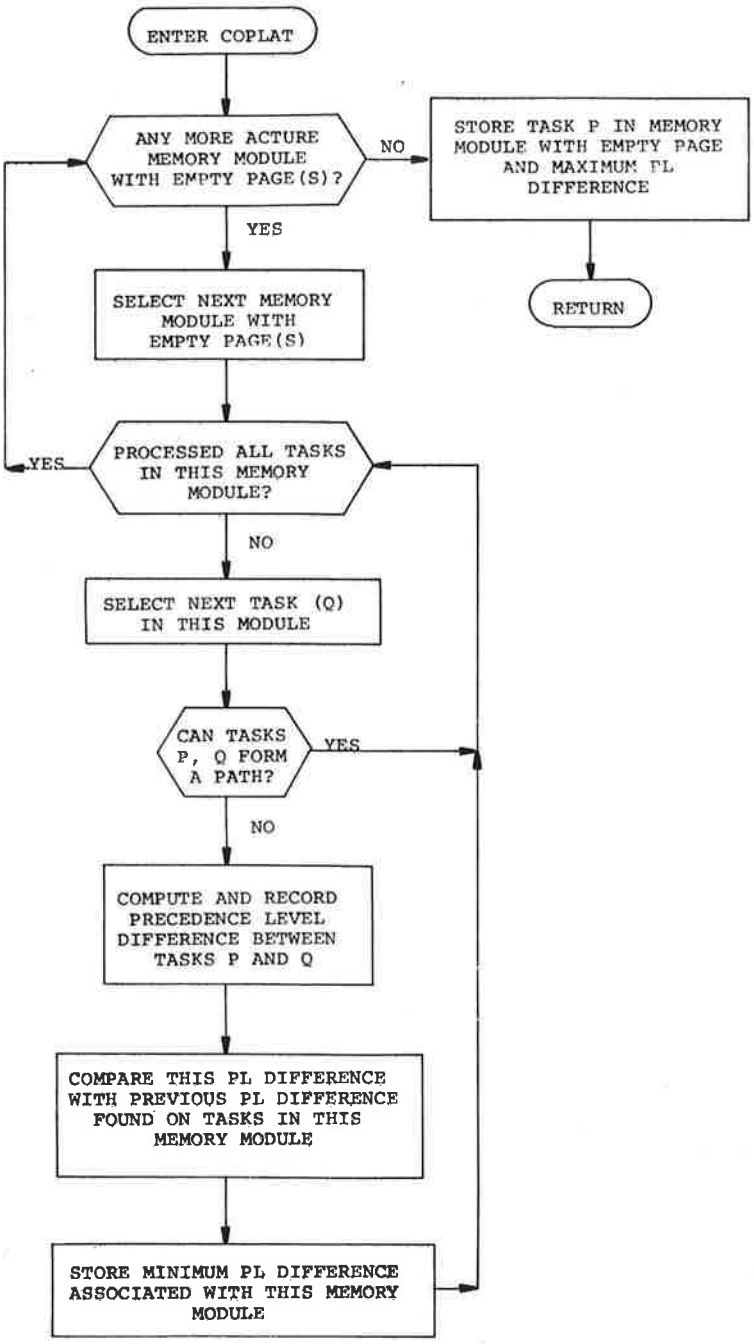



Figure B-8 COPLAT - Compute Partition Level and Allocate Subroutine (Continued)

MESP

```
C*****MULTIPRØCESSØR EXECUTIVE SIMULATIØN PRØGRAM - MESP
      DIMENSIØN NSET(6,3)
      CØMMØN ID,IM,INIT,JEVNT,JMNIT,MFA,MSTØP,MX,MXC,NCLCT,NHIST,NØQ,
     *        NØRPT,NØT,NPRMS,NRUN,NRUNS,NSTAT,ØUT,SCALE,ISEED,TNØW,
     *        TBEG,TFIN,MXX,NPRNT,NCRDR,NEP,VNQ(4)
      CØMMØN ATRIB(4),ENQ(4),INN(4),JCELS(5,22),KRANK(4),MAXNQ(4),
     *       MFE(4),MLC(4),MLE(4),NCELS(5),NQ(4),PARAM(20,4),QTIME(4),
     *        SSUMA(10,5),SUMA(10,5),NAME(6),NPRØJ,MØN,NDAY,NYR,JCLR
      CØMMØN /A/ KTASK(32,6),IST(32),NBSY(8),ITSK,
     *           NTSK,NMM,NPG,NCPU,NPL,MARGIN
      CØMMØN /B/ MAT(16,16),CØNFL
      CØMMØN /C/ PAT(32,4),PEXT,TEXT,TIME,ITEM,TEMP,JX,IZ,JY
      CALL SEARCH(1,6H(DMES),1,0)
      TIME=0
      TEXT=0
      PEXT=0
      NCRDR=31
      NPRNT=1
      READ(31,25) ((KTASK(I,J),J=1,6),I=1,16)
      READ(31,25) (IST(I),I=1,32)
      READ(31,25) ((MAT(I,J),J=1,4),I=1,4)
      READ(31,25) IZ,JX,NMM,NPG,NTSK,NCPU
   25 FØRMAT(12I4)
      DØ 10 I=1,NCPU
   10 NBSY(I)=0
      DØ 20 I=1,NTSK
      DØ 20 J=1,4
   20 PAT(I,J)=0
      CALL GASP(NSET)
      CALL EXIT
      END
$1
      SUBRØUTINE NDTASK(NSET)
      DIMENSIØN NSET(6,2)
      INSERT MEXCØM
      INSERT CØMA
      INSERT CØMB
      INSERT CØMC
      IF(IZ.EQ.1) GØ TØ 102
      IF(TNØW) 99,99,90
   99 WRITE(1,100)
  100 FØRMAT(/2X,5HEVENT,2X,4HTNØW,2X,4HTASK,9X,3HPAT,11X,1HI,
     *       1X,8HPAT(I,3),1X,2HKK,1X,9HPAT(KK,3),1X,5HCØNFL/)
   90 WRITE(1,101) TNØW
  101 FØRMAT(1X,6HNDTASK,1X,F6.2)
C*****TAKE STATISTICS ØN PRØCESSØR ACTIVITY,RELEASE ACTIVE PRØCESSØR
C*****AND DECREMENT PREDECESSØR CØUNTS
  102 IF(ATRIB(4)) 1,2,3
    1 CALL ERRØR(1,NSET)
      RETURN
    3 I=ATRIB(4)+0.5
      BSY=0.
      IF(NBSY(I).NE.0) BSY=1.
      CALL TMST(BSY,TNØW,I,NSET)
      NBSY(I)=0
      I=ATRIB(3)+0.5
```

```
      PAT(I,4)=1.
      IF(I.EQ.NTSK) GØ TØ 8
      J=KTASK(I,6)
      IF(IST(J).EQ.0) GØ TØ 13
      JJ=KTASK(I+1,6)
      LL=JJ-1
      DØ 5 K=J,LL
      KK=IST(K)
    5 KTASK(KK,5)=KTASK(KK,5)-1
C*****TEST TASK READINESS AND TEST PRØCESSØR AVAILABILITY
    8 DØ 10 I=1,NTSK
   10 IF(KTASK(I,5).EQ.0) GØ TØ 20
      DØ 12 I=1,NCPU
   12 IF(NBSY(I).NE.0) GØ TØ 30
C*****RECYCLE, REINITIATE DUMMY EVENT AND TAKE STATISTICS ØN
C*****PRØGRAM EXECUTIØN TIME
   13 PEXT=TNØW-TIME
      CALL CØLCT(PEXT,1,NSET)
      TIME=TNØW
      ATRIB(1)=TNØW
      ATRIB(2)=1
      ATRIB(3)=1
      ATRIB(4)=0
      KTASK(1,5)=0
      CALL FILEM(1,NSET)
      IF(IZ.EQ.1) GØ TØ 170
      WRITE(1,14)
   14 FØRMAT(/20X,24HPRØCESSØR ACTIVITY TABLE)
      WRITE(1,15)
   15 FØRMAT(/10X,7HTASK NØ,4X,6HCPU NØ,6X,10HSTART TIME,4X,8HEND TIME)
      DØ 17 I=1,NTSK
      WRITE(1,16) I,(PAT(I,J),J=1,3)
   16 FØRMAT(12X,I2,6X,3(F6.2,8X))
   17 CØNTINUE
  170 DØ 18 I=1,NTSK
      DØ 18 J=1,4
   18 PAT(I,J)=0
      IF(NRUNS.EQ.1) GØ TØ 95
      NRUNS=NRUNS-1
      RETURN
   20 ITSK=I
      DØ 25 II=1,NCPU
   25 IF(NBSY(II).EQ.0) GØ TØ 60
C*****ESTABLISH NEXT NØTASK EVENT AND UPDATE EVENT FILE
   30 TEMP=5000.
      DØ 40 I=1,NTSK
      IF(PAT(I,1).EQ.0.) GØ TØ 40
      IF(PAT(I,4).EQ.1.) GØ TØ 40
      IF(PAT(I,3).GE.TEMP) GØ TØ 40
      TEMP=PAT(I,3)
      II=I
   40 CØNTINUE
      ATRIB(1)=TEMP
      ATRIB(2)=1
      ATRIB(3)=II
      ATRIB(4)=PAT(II,1)
```

```
      CALL FILEM(I,NSET)
      RETURN
C*****SCHEDULE TASKS BY HEURISTIC RULES, DETERMINE TASK EXECUTION
C*****TIME, AND UPDATE PROCESSOR ACTIVITY TABLE
    2 ITSK=ATRIB(3)+0.5
      II=I
   60 NFLG=0
      GO TO (70,62,63,64,65), JX
   62 CALL RULE2
      GO TO 70
   63 CALL RULE3
      GO TO 70
   64 CALL RULE4
      GO TO 70
   65 CALL RULE5
      NFLG=1
      IF (ITSK.EQ.0) GO TO 30
   70 CALL DRAND(ISEED,RNUM)
      I=ITSK
      TEXT=FLOAT(KTASK(I,1))*(1.+RNUM)
      BSY=0.
      IF(NBSY(II).NE.0) BSY=1.
      CALL TMST(BSY,TNOW,II,NSET)
      NBSY(II)=ITSK
      PAT(I,1)=II
      PAT(I,2)=TNOW
      PAT(I,3)=TNOW+TEXT
C*****DETECT MEMORY CONFLICT, TAKE STATISTICS ON CONFLICT TIME
C*****ASSUMING 50% TIME DELAY DUE TO MEMORY CONFLICT
      IF(IZ.EQ.1) GO TO 72
      WRITE(1,71) I,(PAT(I,J),J=1,3)
   71 FORMAT(16X,I2,1X,3(1X,F6.2))
   72 IF(NFLG.EQ.1) GO TO 75
      NN=1
      CALL MCFLT(I,NN)
   75 KTASK(I,5)=KTASK(I,4)
      GO TO 8
C*****REVISE TASK END TIME DUE TO MEMORY CONFLICT, RESTORE PREDECESSOR
 C*****COUNT
C*****PREPARATION FOR THE END OF SIMULATION
   95 MSTOP=-1
      NORPT=0
      DO 96 I=1,NCPU
      BSY=0.
      IF(NBSY(I).NE.0) BSY=1.
   96 CALL TMST(BSY,TNOW,I,NSET)
      RETURN
      END
$I
      SUBROUTINE RULE2
      INSERT COMA
C*****SCHEDULE LONGEST PRECEDENCE CHAIN (LPC) FIRST RULE
      ITEM=0
      DO 10 K=1,NTSK
      IF(KTASK(K,5).NE.0) GO TO 10
      IF(KTASK(K,2).LE.ITEM) GO TO 10
```

```
      ITEM=KTASK(K,2)
      ITSK=K
   10 CØNTINUE
      RETURN
      END
$I
      SUBRØUTINE RULE3
      INSERT CØMA
C*****SCHEDULE LARGEST SECCESSØR GRØUP (LSG) FIRST RULE
      ITEM=0
      DØ 10 K=1,NTSK
      IF(KTASK(K,5).NE.0) GØ TØ 10
      IF(KTASK(K,3).LE.ITEM) GØ TØ 10
      ITEM=KTASK(K,3)
      ITSK=K
   10 CØNTINUE
      RETURN
      END
$I
      SUBRØUTINE RULE4
      INSERT CØMA
C*****SCHEDULE LPC FIRST IF MØRE THAN ØNE TASK THEN LSG RULE
      ITEM=0
      DØ 10 K=1,NTSK
      IF(KTASK(K,5).NE.0) GØ TØ 10
      IF(KTASK(K,2).LE.ITEM) GØ TØ 10
      ITEM=KTASK(K,2)
      ITSK=K
   10 CØNTINUE
      ITEM=KTASK(ITSK,3)
      DØ 20 J=1,NTSK
      IF(KTASK(J,5).NE.0) GØ TØ 20
      IF(KTASK(J,3).LE.ITEM) GØ TØ 20
      ITEM=KTASK(J,3)
      ITSK=J
   20 CØNTINUE
      RETURN
      END
$I
      SUBRØUTINE RULE5
      DIMENSIØN ICØNF(32)
      INSERT CØMA
      INSERT CØMB
C*****SCHEDULE LPC TASK FIRST AND THEN CHECK MEMØRY CØNFLICT
C*****IF CØNFLICT TRY NEXT LPC TASK, IF NØNE LET CPU WAIT
C*****AND ESTABLISH NEXT NDTASK EVENT.
      DØ 1 I=1,NTSK
    1 ICØNF(I)=0
      NN=0
    2 ITEM=0
      ITSK=0
      DØ 10 K=1,NTSK
      IF(KTASK(K,5).NE.0) GØ TØ 10
      IF(ICØNF(K).EQ.1) GØ TØ 10
      IF(KTASK(K,2).LE.ITEM) GØ TØ 10
      ITEM=KTASK(K,2)
```

```
      ITSK=K
   10 CØNTINUE
      IF(ITSK.EQ.0) GØ TØ 20
      CALL MCFLT(ITSK,NN)
      IF(CØNFL.EQ.0.) GØ TØ 20
      ICØNF(ITSK)=1
      GØ TØ 2
   20 RETURN
      END
$1
      SUBRØUTINE MCFLT(I,NN)
      DIMENSIØN NSET(6,2)
      INSERT MEXCØM
      INSERT CØMA
      INSERT CØMB
      INSERT CØMC
      IFLG=NN
      CØNFL=0
C*****LØCATE TASK I IN MEMØRY MØDULE MN
      DØ 5 M=1,NMM
      DØ 5 L=1,NPG
      IF(MAT(M,L).EQ.I) MN=M
    5 CØNTINUE
C*****FIND ACTIVE TASKS IN EXECUTIØN
      DØ 20 N=1,NCPU
      IF(NBSY(N).EQ.0) GØ TØ 20
      IF(NBSY(N).EQ.1) GØ TØ 20
      KK=NBSY(N)
      IF(PAT(KK,3).EQ.TNØW) GØ TØ 20
C*****LØCATE ACTIVE TASK IN MEMØRY MØDULE MM
      DØ 20 M=1,NMM
      DØ 20 L=1,NPG
   10 IF(MAT(M,L).NE.KK) GØ TØ 20
      MM=M
C*****DETERMINE AND CØMPUTE MEMØRY CØNFLICT TIME, STØRE THE MEMØRY
C*****CØNFLICT TIME IN CØNFL
      IF(MM.NE.MN) GØ TØ 20
      IF(IFLG.EQ.0) GØ TØ 30
      CØNFL=0
      IF(PAT(I,3)-PAT(KK,3)) 12,12,15
   12 CØNFL=CØNFL+PAT(I,3)-TNØW
      GØ TØ 17
   15 CØNFL=CØNFL+PAT(KK,3)-TNØW
   17 PAT(KK,3)=PAT(KK,3)+0.5*CØNFL
      PAT(I,3)=PAT(I,3)+0.5*CØNFL
      CALL CØLCT(CØNFL,2,NSET)
      CALL HISTØ(CØNFL,0.0,0.5,1)
      IF(IZ.EQ.1) GØ TØ 20
      WRITE(1,19) I,PAT(I,3),KK,PAT(KK,3),CØNFL
   19 FØRMAT(4(1X,I2,2X,F6.2,2X),F5.2,1X,F5.2)
   20 CØNTINUE
      IF(CØNFL) 22,22,25
   22 CALL CØLCT(CØNFL,2,NSET)
      CALL HISTØ(CØNFL,0.2,0.5,1)
   25 RETURN
   30 CØNFL=1.
```

```
      RETURN
      END
$1
      SUBRØUTINE EVNTS(IX,NSET)
      DIMENSIØN NSET(6,2)
      INSERT MEXCØM
      INSERT CØMA
      INSERT CØMB
      INSERT CØMC
      CALL NDTASK(NSET)
      RETURN
      END
$1
      SUBRØUTINE ØTPUT(NSET)
      RETURN
      END
$0
```

MASQ

```
C*****SIMPLE SEQUENTIAL MEMØRY ALLØCATIØN - MASQ
      INSERT C@MA
      INSERT CØMB
      I=1
      K=1
      J=1
   10 IF(K.GT.NTSK) GØ TØ 20
      IF(I.GT.NMM) GØ TØ 20
      MAT(I,J)=K
      K=K+1
      J=J+1
      IF(J.LE.NPG) GØ TØ 10
      I=I+1
      J=1
      GØ TØ 10
   20 IF(IZ.NE.0) GØ TØ 30
      WRITE(1,100) (I,I=1,4)
  100 FØRMAT(//16X,4(2X,1HM,I1)/)
      DØ 110 J=1,4
      WRITE(1,110) J,(MAT(I,J),I=1,4)
  110 FØRMAT(9X,4HPAGE,I2,1X,4I4)
   30 RETURN
      END
```

```
C*****MEMØRY ALLØCATIØN VIA PRECEDENCE PARTITIØN - MAPP
C*****INITIALIZATIØN
      LØGICAL LINK
      INTEGER PLD,PLT,PRT,STCK,P,Q
      INSERT CØMA
      INSERT CØMB
      CØMMØN /E/ MPS(16),PLD(16),PLT(12),PRT(16),STCK(32),LINK
      CALL SEARCH(1,6H(DATA),1,0)
C*****READ IN INPUT TABLES AND LISTS
      READ(31,10) ((KTASK(I,J),J=1,6),I=1,16)
      READ(31,10) (IST(I),I=1,32)
      READ(31,10) (PLT(I),I=1,12)
      READ(31,10) (PRT(I),I=1,16)
      READ(31,10) NMM,NPG,NTSK,NCPU,NPL,MARGIN
   10 FØRMAT(12I4)
      DØ 20 I=1,16
      MPS(I)=0
      PLD(I)=0
      DØ 20 J=1,16
   20 MAT(I,J)=0
C*****STEP 1 - PRELIMINARY ALLØCATIØN
      NMX=0
      I=1
      J=1
      K=1
   50 KK=PLT(K)
  100 Q=KK
  110 IF(MPS(I).EQ.0) GØ TØ 130
      J=MPS(I)-1
  115 P=MAT(I,J)
      CALL SRCH(P,0)
      IF(LINK) GØ TØ 125
  120 I=I+1
      J=1
      GØ TØ 100
  125 IF(J.GT.I) GØ TØ 150
      J=MPS(I)
  130 MAT(I,J)=Q
      J=J+1
      MPS(I)=J
      IF(I.LE.NMX) GØ TØ 140
      NMX=I
  140 KK=KK+1
      IF(KK.LT.PLT(K+1)) GØ TØ 160
      K=K+1
      IF(K.GT.NPL) GØ TØ 190
      I=1
      GØ TØ 50
  150 J=J-1
      GØ TØ 115
  160 I=1
      GØ TØ 100
C*****PRINT STEP 1 RESULTS
  190 WRITE(1,191)
  191 FØRMAT(///17X,17HSTEP 1 MEMØRY MAP//)
      WRITE(1,192) (K,K=1,NMX)
```

```
  192 FØRMAT(17X,10(1HM,I2,2X))
      DØ 193 J=1,NPL
  193 WRITE(1,194) (J,(MAT(I,J),I=1,NMX))
  194 FØRMAT(8X,4HPAGE,I2,16(3X,I2)/)
C*****STEP 2 - HØRIZØNTAL FØLDING
      I=NMX
      LIMIT=NPG
  210 IF(I.LE.NMM) GØ TØ 290
      J=MPS(I)-1
  220 P=MAT(I,J)
      CALL CØPLAT(P,LIMIT)
      IF(J.EQ.1) GØ TØ 230
      J=J-1
      GØ TØ 220
  230 I=I-1
      GØ TØ 210
C*****PRINT STEP 2 RESULTS
  290 WRITE(1,291)
  291 FØRMAT(///17X,17HSTEP 2 MEMØRY MAP//)
      WRITE(1,192) (K,K=1,NMM)
      DØ 293 J=1,NPL
  293 WRITE(1,294) (J,(MAT(I,J),I=1,NMM))
  294 FØRMAT(8X,4HPAGE,I2,10(3X,I2)/)
C*****STEP 3 - VERTICAL FØLDING
  300 IFLG=0
      I=1
      LIMIT=NPG
  310 J=MPS(I)-1
  320 IF(J.LE.NPG) GØ TØ 330
      P=MAT(I,J)
      CALL CØPLAT(P,LIMIT)
      J=J-1
      GØ TØ 320
  330 I=I+1
      IF(I.LE.NMM) GØ TØ 310
C*****PRINT STEP 3 RESULT
  390 IF(IFLG.EQ.1) GØ TØ 395
      WRITE(1,391)
  391 FØRMAT(///17X,17HSTEP 3 MEMØRY MAP//)
      WRITE(1,192) (K,K=1,NMM)
      DØ 393 J=1,NPG
  393 WRITE(1,194) (J,(MAT(I,J),I=1,NMM))
      IF(MARGIN.GE.NPG) GØ TØ 495
      GØ TØ 400
  395 WRITE(1,396)
  396 FØRMAT(///17X,17HSTEP 4 MEMØRY MAP//)
      WRITE(1,192) (K,K=1,NMM)
      DØ 493 J=1,NPG
  493 WRITE(1,194) (J,(MAT(I,J),I=1,NMM))
  495 STØP
C*****STEP 4 - SMØØTHING PRØCESS
  400 LIMIT=MARGIN
      IFLG=1
      GØ TØ 310
      END
$1
```

```
      SUBRØUTINE SRCH(P,Q)
      LØGICAL LINK
      INTEGER PLD,PLT,PRT,STCK,P,Q
      INSERT CØMA
      INSERT CØMB
      CØMMØN /E/ MPS(16),PLD(16),PLT(12),PRT(15),STCK(32),LINK
      I1=P
      I2=Q
      I=1
      IF(P.EQ.Q) GØ TØ 125
      IF(PRT(P).EQ.PRT(Q)) GØ TØ 120
      IF(PRT(P).GT.PRT(Q)) GØ TØ 100
      N=P
      P=Q
      Q=N
  100 IF(KTASK(Q,6).EQ.0) GØ TØ 115
      K=KTASK(Q,6)
  105 IF(P.EQ.IST(K)) GØ TØ 125
      L=IST(K)
      IF(PRT(P).LE.PRT(L)) GØ TØ 110
      STCK(I)=IST(K)
      I=I+1
  110 K=K+1
      IF(K.LT.KTASK(Q+1,6)) GØ TØ 105
  115 IF(I.EQ.1) GØ TØ 120
      I=I-1
      Q=STCK(I)
      GØ TØ 100
  120 LINK=.FALSE.
      GØ TØ 130
  125 LINK=.TRUE.
  130 P=I1
      Q=I2
  135 RETURN
      END
$I
      SUBRØUTINE CØPLAT(P,LIMIT)
      LØGICAL LINK
      INTEGER PLD,PLT,PRT,STCK,P,Q
      INSERT CØMA
      INSERT CØMB
      CØMMØN /E/ MPS(16),PLD(16),PLT(12),PRT(16),STCK(32),LINK
      N=1
  100 PLD(N)=32
      M=MPS(N)
      IF((MPS(N)-1).GE.LIMIT) GØ TØ 150
  110 M=M-1
      Q=MAT(N,M)
      CALL SRCH(P,Q)
      IF(LINK) GØ TØ 140
      IF(PRT(Q).GE.PRT(P)) GØ TØ 130
      ITEMP=PRT(P)-PRT(Q)
  120 IF(ITEMP.LT.PLD(N)) PLD(N)=ITEMP
      GØ TØ 140
  130 ITEMP=PRT(Q)-PRT(P)
      GØ TØ 120
```

```
  140 IF(M.NE.I) GØ TØ 110
      GØ TØ 152
  150 PLD(N)=0
  152 N=N+1
      IF(N.LE.NMM) GØ TØ 150
      N=1
  155 I=N
      IF(MPS(I).GT.LIMIT) GØ TØ 165
      ITEMP=PLD(N)
  160 N=N+1
      IF(N.GT.NMM) GØ TØ 170
      IF(PLD(N).LE.ITEMP) GØ TØ 160
      GØ TØ 155
  165 N=N+1
      GØ TØ 155
  170 J=MPS(I)
      MAT(I,J)=P
      MPS(I)=J+1
      RETURN
      END
$0
```

```
C*****MEMØRY ALLØCATIØN VIA STATIC SCHEDULING - MASS
      INTEGER TSKC,PCHN,PCHP
      INSERT CØMA
      INSERT CØMB
      CØMMØN /D/ PCHN(8,32,3),LPCH(8),NPCH,TSKC,NØVT,NAMT,MPGP(8),JJ,
     *           PCHP(8),ISTCK(8),JSTCK(8),ISTAK(8),JSTAK(8),NAT(16,2)
C*****STATIC TASK SCHEDULING
C*****INITIALIZATIØN - CLEAR REGISTERS
      CALL SEARCH(1,6H(DATA),1,0)
      NPCH=0
      TSKC=1
      ITME=0
      READ(31,10) ((KTASK(I,J),J=1,6),I=1,16)
      READ(31,10) (IST(I),I=1,32)
      READ(31,10) NMM,NPG,NTSK,NCPU
   10 FØRMAT(12I4)
      DØ 20 I=1,4
      NBSY(I)=0
   20 LPCH(I)=0
      DØ 30 I=1,4
      DØ 30 J=1,32
      DØ 30 K=1,3
   30 PCHN(I,J,K)=0
C*****FIND READY TASK AND AVIALABLE PRØCESSØR
  100 IF(TSKC.GT.NTSK) GØ TØ 400
  105 DØ 110 I=1,NTSK
  110 IF(KTASK(I,5).EQ.0) GØ TØ 160
C*****FIND TASK WITH SMALLEST EXECUTIØN TIME
  115 ISTØR=1000
      IZ=0
  120 IZ=IZ+1
      IF(IZ.GT.NCPU) GØ TØ 146
      IF(NBSY(IZ).EQ.0) GØ TØ 120
      JJ=LPCH(IZ)-1
      IF(ISTØR.LE.PCHN(IZ,JJ,3)) GØ TØ 120
      ISTØR=PCHN(IZ,JJ,3)
      KK=NBSY(IZ)
      II=IZ
      GØ TØ 120
C*****RELEASE PRØCESSØR,RECØRD END TIME,DECREMENT PRECEDENCE CØUNT
  130 WRITE(1,140)
  140 FØRMAT(/8X,2HII,4X,2HKK,4X,4HTIME)
      WRITE(1,145) II,KK,KTASK(KK,1)
  145 FØRMAT( 8X,I2,4X,I2,5X,I2 )
  146 CØNTINUE
      JJ=LPCH(II)-1
  150 L=KTASK(KK,6)
      LL=IST(L)
      M=KTASK(KK+1,6)-1
      MM=IST(M)
      DØ 155 I=LL,MM
  155 KTASK(I,5)=KTASK(I,5)-1
      ITME=PCHN(II,JJ,3)
      NBSY(II)=0
      GØ TØ 105
  160 DØ 165 J=1,NCPU
```

```
  165  IF(NBSY(J).EQ.0) GØ TØ 203
       GØ TØ 115
C*****FIND TASK WITH LØNGEST PRECEDENCE CHAIN
  200 WRITE(1,201)
  201 FØRMAT(/9X,1HI,5X,1HJ)
       WRITE(1,202) I,J
  202 FØRMAT( 8X,I2,4X,I2 )
  203 CØNTINUE
       IFLG=0
  205 N=I
       LPC=KTASK(I,2)
  210 IF(I.EQ.NTSK) GØ TØ 226
       I=I+1
       IF(KTASK(I,5).NE.0) GØ TØ 210
       IF(LPC.LT.KTASK(I,2)) GØ TØ 205
       IF(LPC.EQ.KTASK(I,2)) IFLG=1
       GØ TØ 210
C*****FIND TASK WITH LARGEST SUCCESSØR GRØUP AMØNG EQUAL LPC TASKS
  220 WRITE(1,224)
  224 FØRMAT(/8X,3HLPC,4X,1HN)
       WRITE(1,225) LPC,N
  225 FØRMAT( 8X,I2,4X,I2 )
  226 CØNTINUE
       IF(IFLG.EQ.0) GØ TØ 300
       LSG=KTASK(N,3)
       I=1
  230 IF(KTASK(I,5).NE.0) GØ TØ 240
       IF(LPC.NE.KTASK(I,2)) GØ TØ 240
       IF(I.EQ.N) GØ TØ 240
       IF(LSG.GE.KTASK(I,3)) GØ TØ 240
       LSG=KTASK(I,3)
       N=I
  240 I=I+1
       IF(I.LE.NTSK) GØ TØ 230
C*****ASSIGN TASK TØ PRØCESSØR,RECØRD TASK START TIME,RESET KTASK(I,5)
  300 NBSY(J)=N
       IF(NPCH.LT.J) NPCH=J
       I=J
       J=LPCH(I)
       IF(J.EQ.0) J=1
       PCHN(I,J,1)=N
       PCHN(I,J,2)=ITME
       PCHN(I,J,3)=ITME+KTASK(N,1)
       NAT(N,1)=ITME
       NAT(N,2)=ITME+KTASK(N,1)
       KTASK(N,5)=KTASK(N,4)
       J=J+1
       LPCH(I)=J
       TSKC=TSKC+1
       GØ TØ 100
  400 WRITE(1,403)
  403 FØRMAT(/9X,27HPRØCESS CHAIN - PCHN(I,J,K))
       WRITE(1,401) (J,J=1,8)
  401 FØRMAT(/16X,8I4/)
       DØ 405 I=1,NPCH
       WRITE(1,402)
```

```
  402 FØRMAT(/)
      DØ 405 K=1,3
  405 WRITE(1,406) I,K,(PCHN(I,J,K),J=1,8)
  406 FØRMAT(8X,2HI=,I1,2X,2HK=,I1,8I4)
  410 CØNTINUE
C*****BEGIN MEMØRY ALLØCATIØN - CLEAR DATA TABLES
      DØ 500 I=1,NMM
      MPGP(I)=0
      DØ 500 J=1,NPG
  500 MAT(I,J)=0
      DØ 510 I=1,NCPU
  510 PCHP(I)=0
C*****PRØCESS CHAIN TØ MEMØRY ALLØCATIØN TABLE TRANSFER
      I=1
      II=1
  520 IF(LPCH(I).EQ.0.ØR.I.GT.NCPU) GØ TØ 565
      IF(LPCH(I).GT.NPG) GØ TØ 540
      LL=LPCH(I)-1
      DØ 530 J=1,LL
  530 MAT(II,J)=PCHN(I,J,1)
      MPGP(II)=LPCH(I)
      LPCH(I)=0
      GØ TØ 560
  540 DØ 550 J=1,NPG
  550 MAT(II,J)=PCHN(I,J,1)
      MPGP(II)=NPG+1
      PCHP(I)=NPG+1
  560 II=II+1
      I=I+1
      GØ TØ 520
  565 WRITE(1,567)
  567 FØRMAT(//9X,32HSTEP 1 - PRØCESS CHAIN TØ MEMØRY/)
      CALL TYPE
C*****FIND PRØCESS CHAIN WITH LØNGEST REMAINDER
  570 IF(NMM.EQ.NPCH) GØ TØ 645
      M=0
      I=1
  600 IF(I.GT.NPCH) GØ TØ 620
      IF(LPCH(I).EQ.0) GØ TØ 610
      IF(LPCH(I)-PCHP(I).LE.M) GØ TØ 610
      M=LPCH(I)-PCHP(I)
      N=I
  610 I=I+1
      GØ TØ 600
  620 IF(M.EQ.0) GØ TØ 775
C*****FIND EMPTY MEMØRY MØDULE
      II=0
  630 II=II+1
      IF(II.GT.NMM) GØ TØ 645
      IF(MPGP(II).NE.0) GØ TØ 630
C*****ASSIGN REMAINDER TASKS IN PRØCESS CHAIN TØ EMPTY MEMØRY MØDULE
       IF(M.GT.NPG) M=NPG
      JJ=MPGP(II)+1
      J=PCHP(N)
      I=N
  640 MAT(II,JJ)=PCHN(I,J,1)
```

```
      JJ=JJ+1
      MPGP(II)=JJ
      J=J+1
      PCHP(N)=J
      IF(JJ.LE.M) GØ TØ 640
      IF(PCHP(N).GE.LPCH(N)) LPCH(N)=0
      GØ TØ 570
  645 WRITE(1,647)
  647 FØRMAT(//9X,40HSTEP 2 - REMAINDER CHAIN TØ EMPTY MØDULE/)
      CALL TYPE
C*****FIND EMPTY PAGE IN MEMØRY MØDULE - SCAN REMAINDER TASKS
  650 DØ 660 K=1,16
      ISTCK(K)=0
      JSTCK(K)=0
      ISTAK(K)=0
  660 JSTAK(K)=0
      K=1
      KX=0
      KK=1
  670 I=0
  675 I=I+1
      IF(I.GT.NPCH) GØ TØ 697
  676 IF(LPCH(I).EQ.0) GØ TØ 675
  677 LPCH(I)=LPCH(I)-1
      J=LPCH(I)
      II=0
  680 II=II+1
      IF(II.GT.NMM) GØ TØ 695
      IF(MPGP(II).GT.NPG) GØ TØ 680
C*****ALLØCATE TASK IF NØ ØVRLAP WITH RESIDING TASK IN MEMØRY MØDULE
       CALL ØVRLAP(I,J,II,KX)
      WRITE(1,685) PCHN(I,J,1),II,NØVT

  685 FØRMAT(/7X,3(2X,I2))
      IF(NØVT.NE.0) GØ TØ 680
      JJ=MPGP(II)
      MAT(II,JJ)=PCHN(I,J,1)
      MPGP(II)=JJ+1
      IF(LPCH(I).LE.PCHP(I)) LPCH(I)=0
      GØ TØ 676
C*****STØRE UNALLØCATABLE TASK INDEXES IN STACK
  695 ISTCK(K)=I
      JSTCK(K)=J
      K=K+1
      IF(LPCH(I).LE.PCHP(I)) LPCH(I)=0
      GØ TØ 676
  697 WRITE(1,698)
  698 FØRMAT(//9X,37HSTEP 3 - REMAINDER TASK TØ EMPTY PAGE/)
      CALL TYPE
C*****3-TASK SWAPPING ØPERATIØN
  700 IF(K.EQ.1) GØ TØ 775
      K=K-1
      I=ISTCK(K)
      J=JSTCK(K)
      II=0
  710 II=II+1
```

```
      IF(II.GT.NMM) GØ TØ 735
      IF(MPGP(II).GT.NPG) GØ TØ 710
      MM=0
  720 MM=MM+1
      IF(MM.GT.NMM) GØ TØ 740
      IF(MM.EQ.II) GØ TØ 720
      CALL ØVRLAP(I,J,MM,KX)
      IF(NØVT.NE.0) GØ TØ 720
      NN=0
  730 NN=NN+1
      IF(NN.GE.MPGP(MM)) GØ TØ 720
      CALL ØVRLAP(MM,NN,II,KX)
      IF(NØVT.NE.0) GØ TØ 730
C*****INITIATE ACTUAL SWAPPING ACTIVITY
      JJ=MPGP(II)
      MAT(II,JJ)=MAT(MM,NN)
      MAT(MM,NN)=PCHN(I,J,1)
      MPGP(II)=JJ+1
      GØ TØ 700
C*****STØRE UNSWAPPABLE TASKS IN STACK
  740 ISTAK(KK)=I
      JSTAK(KK)=J
      KK=KK+1
      GØ TØ 700
  735 WRITE(1,745)
  745 FØRMAT(//9X,30HSTEP 4 - MEMØRY MAP AFTER SWAP/)
      CALL TYPE
C*****ALLØCATE REMAINDER TASKS TØ MØDULES ØF SMALLEST ØVRLAP INDEX
  750 IF(KK.EQ.1) GØ TØ 775
      KK=KK-1
      I=ISTAK(KK)
      J=JSTAK(KK)
      ITEMP=1000
      II=0
  760 II=II+1
      IF(II.GT.NMM) GØ TØ 770
      KX=1
      CALL ØVRLAP(I,J,II,KX)
      IF(NAMT.GE.ITEMP) GØ TØ 760
      ITEMP=NAMT
      ISTØ=II
      GØ TØ 760
  770 II=ISTØ
      JJ=MPGP(II)
      MAT(II,JJ)=PCHN(I,J,1)
      MPGP(II)=MPGP(II)+1
      GØ TØ 750
  775 WRITE(1,780)
  780 FØRMAT(//9X,25HSTEP 5 - FINAL MEMØRY MAP/)
      CALL TYPE
  800 STØP
      END
$1
      SUBRØUTINE ØVRLAP(IX,JX,LX,KX)
      INTEGER TSKC,PCHN,PCHP
      INSERT CØMA
```

```
      INSERT CØMB
      INSERT CØMD
      NAMT=0
      NØVT=0
      MX=0
  100 MX=MX+1
      IF(MX.EQ.MPGP(LX)) GØ TØ 140
      IT=PCHN(IX,JX,1)
      LT=MAT(LX,MX)
      IF(NAT(LT,2).LE.NAT(IT,1).ØR.NAT(IT,2).LE.NAT(LT,1)) GØ TØ 100
       NØVT=NØVT+1
      IF(KX.EQ.1) GØ TØ 130
      NAMT=MAT(LX,MX)
      JJ=MX
      GØ TØ 100
  130 IF(PCHN(IX,JX,3).GE.PCHN(I,J,3)) NX=PCHN(I,J,3)
      IF(PCHN(IX,JX,3).LT.PCHN(I,J,3)) NX=PCHN(IX,JX,3)
      IF(PCHN(IX,JX,2).GE.PCHN(I,J,2)) NY=PCHN(IX,JX,2)
      IF(PCHN(IX,JX,2).LT.PCHN(I,J,2)) NY=PCHN(I,J,2)
      NAMT=NAMT+(NX-NY)
      GØ TØ 100
  140 RETURN
      END
$1
      SUBRØUTINE TYPE
      INSERT CØMA
      INSERT CØMB
      INSERT CØMD
      WRITE(1,100) (I,I=1,4)
  100 FØRMAT(/16X,4(2X,1HM,I1)/)
      DØ 110 J=1,4
      WRITE(1,110) J,(MAT(I,J),I=1,4)
  110 FØRMAT(9X,4HPAGE,I2,1X,4I4)
      RETURN
      END
$0
```

## REFERENCES


1. J.P. Anderson, et.al., "D832--A Multiple Computer System for Command and Control", Proceedings, FJCC, pp. 86-95.
2. B. Wald, "Utilization of A Multiprocessor In Command and Control", Proceedings, IEEE, Vol. 54, No. 12, December 1966, pp. 1885-1888.
3. R.L. Alonso, A.L. Hopkins, and H.A. Thaler, "A Multiprocessor Structure", Conference Digest, First Annual IEEE Computer Conference, 1967.
4. T.E. Burke and G.Y. Wang, "Spaceborne Multiprocessing Organizations", Technical Paper, Session 9, WESCON, August 1966, pp. 1-6.
5. G.Y. Wand, "System Design of A Multiprocessor Organization", Technical Report RC-T-079, NASA/ERC, Cambridge, Massachusetts, April 1970.
6. J.F. Keeley, "An Application-Oriented Multiprocessing System", IBM Systems Journal, Vol. 6, No. 2, 1967, pp. 78-79
7. J. Martin, "Design of Real-time Computer Systems", Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.
8. "Aviation Forecasts, 1969-1980", Economics Division, Office of Policy Development, Department of Transportation, FAA, Washington, D.C., January 1969.
9. "Forecast of Domestic Passenger Traffic for the Eleven Trunkline Carriers - Scheduled Service - 1968-1977", CAB Research Study, September 1967.
10. "A.T.A. Airline Airport Demand Forecasts - Industry Report", Air Transportation Association of America, Washington, D.C., July 1969.
11. "An Analysis of Project Beacon", Department of Transportation, FAA, Air Traffic Services, November 1962.
12. G.R. Blakeney, "Design Characteristics of the 9020 System", IBM Systems Journal, Vol. 6, No. 2, 1967, pp. 80-94

13. J.A. Devreaux, "9020 System Control Program Features", IBM Systems Journal, Vol. 6, No. 2, 1967, pp. 95-102.
14. D.C. Lancto and R.L. Rockefeller, "The Operational Error Analysis Program", IBM Systems Journal, Vol. 6, No. 2, 1967, pp. 103-115.
15. F.K. Seward, "Programs for the Intended Application", IBM Systems Journal, Vol. 6, No. 2, 1967, pp. 124-132.
16. J.C. Nelson and R.P. Sunderman, "Automation of Terminal Air Traffic Control (Past, Present and Future)", AIAA Integrated Information Systems Conference, AIAA Paper No. 71-242, February 1971.
17. "ARTS", Report PX6508/J13766, UNIVAC Division of Sperry Rand Corp., St. Paul, Minnesota, July 1971
18. "System Design Data for the ARTS III Modular Automated Terminal Air Traffic Control System, Beacon Tracking Level", Vol. 1 and 2, UNIVAC Division of Sperry Rand Corp., St. Paul, Minnesota, November 1, 1969.
19. "Report of the Department of Transportation Air Traffic Control Advisory Committee", Department of Transportation, FAA, Washington, D.C. September 1969.
20. W.L. Ashby, "Future Demand for Air Traffic Services", Proceedings of the IEEE, Vol. 58, No. 3, March 1970.
21. W. Graham and R.H. Orr, "Terminal Air Traffic Flow and Collision Exposure", Proceedings of the IEEE, Vol. 58, No. 3, March 1970.
22. K.E. Willis, J.B. Currier, and P. Flanagan, "Intermittent Positive Control", Proceedings of the IEEE, Vol. 58, No. 3, March 1970.
23. N.A. Blake, and J.C. Nelson, "A Projection of Future ATC Data Processing Requirements", Proceedings of the IEEE, Vol. 58, No. 3m March 1970.
24. "ARTS III Coding Specifications", Vol. 1,2, and 3, UNIVAC Division of Sperry Rand Corp., St. Paul, Minnesota, August 1970.

25. L.C. Hobbs and D.J. Theis, "Parallel Processor Systems, Technologies, and Techniques", Spartan Books, 1970.
26. J.H. Holland, "A Universal Computer Capable of Executing An Arbitrary Number of Subprograms Simultaneously", Proceedings AFIPS, EJCC, 1959, pp. 108-113.
27. W.T. Comfort, "A Modified Holland Machine", Proceedings AFIPS, FJCC, 1963, pp. 481-488.
28. ILLIAC IV System Study Final Report, Burroughs Corp., University of Illinois, Purchase Order No. 09852-B, December 1966.
29. D.L. Slotnick, W.C. Borck, and R.C. McRegnolds, "The Solomon Computer", Proceedings AFIPS, FJCC, 1962, pp. 97-107.
30. R.H. Fuller, "Content'Addressable Memory Systems", UCLA Department of Electrical Engineering, Report No. 63-20, June 1963.
31. J.E. Shore and F.A. Polkinghorn, Jr., "A Fast, Flexible, Highly Parallel Associative Processor", Naval Research Laboratory Report 6961, Washington, D.C., November 28, 1969.
32. W.C. Meilander, "The Associative Processor in Aircraft Conflict Detection", Goodyear Aerospace Corp., Report GER-13702, March 1968.
33. E.E. Eddey, "Ground Based Aircraft Collision Avoidance Using An Associative Processor", Goodyear Aerospace Corp., Report GER-14586, February 1970.
34. J.A. Rudolph, L.C. Fulmer, and W.C. Meilander, "The Coming Age of the Associative Processor", Electronics, February 15, 1971, pp. 91-96.
35. W. Shooman, "Orthogonal Processing", Parallel Processor Systems, Technologies, and Applications, ed, by L.C. Hobbs, et.al., Spartan Books, New York, 1970, pp. 297-308.
36. J.A. Githens, "An Associative, Highly-Parallel Computer for Radar Data Processing", Parallel Processor Systems, Technologies, and Applications, ed. by L.S. Hobbs, et.al., Spartan Books, New York, 1970, pp. 71-86.

37. W.S. Litzler, "A Design Trade-off Study for An Associative, Highly-Parallel Computer", M.S.E.E. Thesis, University of Texas at Austin, Austin, Texas, May 1971.
38. "STAR-100 Computer System", Hardware Reference Manual, Control Data Corp., Arden Hills, Minnesota, 1970.
39. D.C. Stanga, "UNIVAC 1108 Multiprocessor System", Proceedings AFIPS, SJCC, 1967, pp. 67-74.
40. M.E. Conway, "A Multiprocessor System Design", Proceedings AFIPS, FJCC, 1963, pp. 139-146.
41. HM4118 Reference Manual, Hughes Aircraft Company, FR 66-11-64, March 1966.
42. J.J. Pariser, "Multiprocessing with Floating Executive Control", IEEE International Convention Record 1965, pp. 266-275.
43. W.W. Plummer, "Asynchronous Arbiters", IEEE Transaction on Computers, Vol. C-21, No. 1, January 1972, pp. 37-41.
44. E.R. Borgers, "Characteristics of Priority Interrupts", Datamation, Vol. 11, June 1965.
45. J.G. Bennet, "Letters", Datamation, Vol. 11, October 1965.
46. R.J. Countanis and N.L. Viss, "A Method of Processor Selection for Interrupt Handling In A Multiprocessor System", IEEE Proceedings (Vol. 54 December 1966) pp. 1812-1819.
47. B.I. Witt, "M65MP: An Experiment In OS/360 Multiprocessing", Proceedings, 23rd National Conference, ACM, August 1968, pp. 691-703.
48. A.J. Critchlow, "Generalized Multiprocessing and Multiprogramming Systems", Proceedings, JFCC, 1963, pp. 107-126.
49. R.N. Thompson, et.al., "The D825 Automatic Operating and Scheduling Program", Proceedings, SJCC, 1963, pp. 41-49.
50. C.V. Ramamoorthy and M.J. Gonzalez, Jr., "A Survey of Techniques for Recognizing Parallel Processable Streams In Computer Programs", Proceedings, AFIPS, 1969, pp. 1-15.
51. H. Hellerman, "Parallel Processing of Algebraic Expressions", IEEE Transaction on Ec, Vol. 15, No. 1, February 1966.

52. H.S. Stone, "One-Pass Compilation of Arithmetic Expressions for A Parallel Processor", Comm. ACM, Vol. 10, No. 4, April 1967, pp. 220-223.
53. J.S. Squire, "A Translation Algorithm for A Multiprocessor Computer", Proceedings 19th ACM National Conference, 1963,
54. B.P. Ochsner, "Controlling A Multiprocessor System", Bell Laboratories Record 44, February 1966, pp. 59-62.
55. D. Martin and G. Estrin, "Experiments on Models of Computations and Systems", IEEE Transaction on Electronic Computer, Vol. EC-16, No. 1, February 1967, pp. 59-69.
56. D. Martin and G. Estrin, "Models of Computational Systems--Cyclic to Acyclic Graph Tranformation", IEEE Transaction on Electronic Computers, Vol. EC-16, No. 1, February 1967, pp. 70-79.
57. C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, "Optimal Scheduling Strategies In A Multiprocessor System", IEEE Transaction on EC, Vol. C-21, No. 2, February 1972, pp. 137-146.
58. R.R. Muntz and E.G. Coffman, Jr., "Pre-emptive Scheduling of Real-Time Tasks On Multiprocessor Systems", Journal of the ACM, Vol. 17, No. 2, April 1970, pp. 324-338.
59. P. Richards, "Parallel Programming" Technical Report No. TO-B60-27, Tech. Ops Inc., Burlington, Massachusetts, August 1960.
60. R.L. Graham, "Bounds On Multiprocessing and Timing Anomalies", SIAM Journal, Applied Mathematics, Vol. 17, No. 2, March 1969, pp. 416-429.
61. G.K. Manacher, "Production and Utilization of Real-time Task Schedules", Journal of the ACM, Vol. 14, No. 3, July 1967, pp. 439-465.
62. F.C. Holland, "System Simulation of Enroute Stage-A Model 16", MITRE Technical Report No. MTR 4043, November 1967.
63. D.C. Gunderson and W.L. Heinerdinger, "Associative Techniques for Control Functions In A Multiprocessor", Honeywell Document 12029-FR1, Honeywell Inc., St. Paul, Minnesota, August 1966.

64. "GASP - A General Activity Simulation Program", Monroeville, Pennsylvania, Applied Research Laboratory, U.S. Steel Corporation, July 8, 1963.

65. R.M. Karp and M. Held, "Finite State Process and Dynamic Programming" SIAM Journal on Applied Mathematics, Vol. 15, No. 3, May 1967, pp. 693-718.