CR-152285

UNCLASSIFIED

FINAL REPORT
NUMERICAL AERODYNAMIC SIMULATION FACILITY
FEASIBILITY STUDY

(NASA-CR-152285) NUMERICAL AERODYNAMIC SIMULATION FACILITY FEASIBILITY STUDY Final Report (Burroughs Corp.) 615 p
HC A99 MF A01 CSCL 14B
N79-26072
Unclas
G3/09 28494

Distribution of this report is provided in the interest of information exchange. Responsibility for the contents resides in the author or organization that prepared it.

Prepared under Contract No. NAS2-9897 by

Burroughs Corporation
Paoli, Pa.

for

AMES RESEARCH CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

UNCLASSIFIED

ERRATA

Chapter 4

pages 4-43, 4-44, 4-45; "LOCTRU" to read as "LOCATION"

page 4-56, after last line should be appended:

"2. Handling of the special constructs used to change terminal characteristics and the system's responses to the terminals."

Chapter 5

page 5-12, under Physical; should read as "Size: 1.2" x 11.5" x 27.5"

page 5-38, under Storage Capacities; should read as "65,536 words/module"

page 5-53, under #4, line 3 to read as "000000011"

page 5-54 bottom of page. The modified equation is
Port = 32 x (EMno MOD 512) + 2 x (EMno MOD4) + 1
for 512 < EMno < 527
which allows better distribution of the spare modules.

Chapter 7

page 7-7, paragraph 4, line 7; "CU" should read as "CR"

page 7-18, paragraph 1, line 5 should read as "would appreciably improve throughput."

Appendix A

page A-3, equation A.1. f_i and t_i are the number of floating point operations and the execution time of the ith piece, respectively.

page A-23, third bullet, line 1; "The correct algorithm" should read as "The given algorithm"

page A-35, paragraph 2, line 2; LAX should be deleted.

page A-40, paragraph 2, line 4 "NJ(J) should read as NM(J)"

page A-59, paragraph 2, line 8 should read as "the physical problem needs to be retained"

page A-63, second equation "T_{EM}=144" should read as "T_{EM} + 144"

ERRATA (Continued)

Appendix B

page B-32, "60%" column, "Double Omega, 512/512" row, "0.504" should read as "0.0504".

page B-40, paragraph 2, line 2; "of that" should read as "of requests that".

page B-45, paragraph 4, line 5; "0 i 10" should read as "$0 \leq i \leq 10$"

Appendix C

page C-8, paragraph 1, line 9; "SOTREM" should read as STOREM"

page C-24, under LOADEM, line 2; "TN" should read "CN".

page C-27, Under FILLRE; "FILIR" should read "FILLR".

page C-40, line 4; "CTIX±" to read as "CTIX1"

Appendix D

page D-10 paragraph 1, lines 6, 7, 8; "Hence, (1-F1) is the fraction of failures that cause a transition directly into the INTERRUPT state, " should be deleted.

page D-12, under TIME BETWEEN FAILUPES (PERMANENT), line 3; "intermittant type device failure" to read as "permanent type device failure".

Appendix F

page F-39, paragraph 5, line 9, "15,38K" to read as "15.38K".

Appendix H

page H-15, equation H.3 to read as P(A-UPPER=1) = P(INPUT) x P(0-BIT=1) x P(1-BIT=1)

UNCLASSIFIED

FINAL REPORT
NUMERICAL AERODYNAMIC SIMULATION FACILITY
FEASIBILITY STUDY

March 1979

Distribution of this report is provided in the interest of information exchange. Responsibility for the contents resides in the author or organization that prepared it.

Prepared under Contract No. NAS2-9897 by

Burroughs Corporation
Paoli, Pa.

for

AMES RESEARCH CENTER
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

UNCLASSIFIED

INTRODUCTION

This report presents the results of Burroughs Corporation's efforts on the Feasibility Study for the Numerical Aerodynamic Simulation Facility (NASF). The study has demonstrated that a particular form and architecture for the NASF (proposed originally during the Preliminary Study [1, 2] and improved during the present study) would meet the established objectives. The Numerical Aerodynamic Simulation Facility is conceived to be more than just a very high-speed computing machine. The facility must also include all that is required to support the users of such a high-speed capability. The feasibility study required consideration of all parts of the proposed NASF system. The depth of study of each part of the system varied depending on the complexity of that part of the system, on the impact of that part on the system capabilities and on whether or not there was sufficient prior knowledge about how to implement that part of the system.

The evaluations performed as part of the study focused on three major issues. First the ability of the proposed system architecture to support the anticipated workload was evaluated. Second, the throughput of the computational engine (the Flow Model Processor) was studied using real application programs. Third, the availability reliability, and maintainability of the system were modeled. The evaluations were based on the Baseline Systems of the Preliminary Studies [1, 2] as modified where appropriate during this study.

The results of these evaluations show that the implementation of the NASF, in the form considered, would indeed be a feasible project with an acceptable level of risk. The technology required (both hardware and software) either already exists or, in the case of a few parts, is expected to be announced this year.

This report describes many of the details of the system including the hardware configuration, user language, software, fault tolerance, and other aspects of the system on which this demonstration of feasibility is based. The first chapter summarizes the study objectives the evaluations made and the results. The NASF system architecture, which is the basis of discussion throughout the report, is described in Chapter 2. The system-level loading analysis performed as part of the study is summarized in Chapter 2 while Chapter 3 reports on the results of timing actual codes for the configurations assumed. The NASF Software and Hardware developments are detailed in Chapters 4 and 5. The various models used to evaluate reliability, availability, maintainability, trustworthiness and the results of that detailed evaluation are included in Chapter 6. Chapter 7 describes the models which have been used used during Flow Model Processor (FMP) instruction timing simulations. The report concludes with a chapter which identifies some of the management and control techniques which could be used to eventually manage a project of this scope. Even more detail concerning most of the areas discussed in the report is included in the Appendices. Each of the chapters includes an introductory section which can be scanned to gain a general perception of each part of the project after reading Chapter 1.

CONTENTS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

ILLUSTRATIONS

TABLES

TABLES

TABLES

CHAPTER 1
STUDY OBJECTIVES AND RESULTS

1.1 STUDY OBJECTIVES

The principal objective of the study has been to consider the feasibility that a facility (NASF), which could support a throughput well in excess of what would be commercially available, could be implemented. In particular, the goal is to have a system where time-averaged Navier-Stokes computation can be performed in 10 minutes or less (on steady fluid flow problems involving a million grid points). Not only is this throughput goal important, but since the intent of the facility is to support daily usage by a large user community, the NASF system availability needs to be better than 90% and the facility needs to be nominally available for 22 hours a day. In order that the NASF may support long runs, the mean time between interruptions should be longer than ten hours. In some cases, an alternate form of the throughput goal can be used. A sustained, average rate of execution of one billion floating point operations per second (one gigaflop/sec or 1 GFLOPS) corresponds roughly to the problem throughput desired on the aerodynamic flow codes.

The starting point of the effort in this study was the baseline configuration developed during the Preliminary Study under contract NAS2-9456 [1,2]. The overall goal was to gain an understanding of the characteristics, capabilities, and potential of the facility in order to make a judgment as to its feasibility. The study required the development of further specifications in order to consider the responsiveness to the desired application of the facility and to develop estimates of the schedule, cost, and risk of such a development.

Both functional and performance (timing) simulators were developed to be able to estimate (as accurately as possible) performance and reliability of the system. Although the primary application of the facility is likely to be aerodynamic flow modeling, the performance studies included both aerodynamic flow codes and weather modeling codes. The use of real programs in these application areas allowed an initial evaluation of the flexibility of the language constructs proposed. This evaluation was especially important since the facility needs to be sufficiently flexible that algorithm development could be supported for fluid dynamics algorithms as yet not investigated. In addition, the diverse user needs for input, output, and algorithm investigation must be supported.

Since the development of the baseline systems considered only aerodynamic flow modeling applications, the consideration of weather modeling codes was especially important. This consideration was used to evaluate the flexibility of the system as far as its support of other, related application areas and was used to determine whether further improvements might be needed to support these additional applications.

All of the goals could be met by the system described as a possible NASF configuration. No hardware modifications would be needed for weather code optimization. Some minor software extensions were proposed based on the weather code evaluations.

1.2 SYSTEM DESCRIPTION

Before describing the system evaluated during this study, the importance of considering all aspects of the facility must be emphasized. During the development of the system, the focus tends to be on the hardware and system software (such as operating systems and compilers). As shown in Figure 1.1, such a focus is limited. If only the system expense is considered, the other areas important to the successful utilization of the facility may be slighted. In particular users themselves face both the expense of their training in the use of the system and the day to day expense of developing and using their various application programs. This usage would include algorithm development, programming, model description data reduction, and so on. The users must be supported by a staff and whatever other support might be needed to keep the facility operational. Such support might include operators, power, cooling, training and supplies. Although the consideration of all these factors complicates the development of the facility, these factors must be carefully considered in order to have a facility that would not only be economical to acquire but also be economical to use. The system described below did consider these factors.

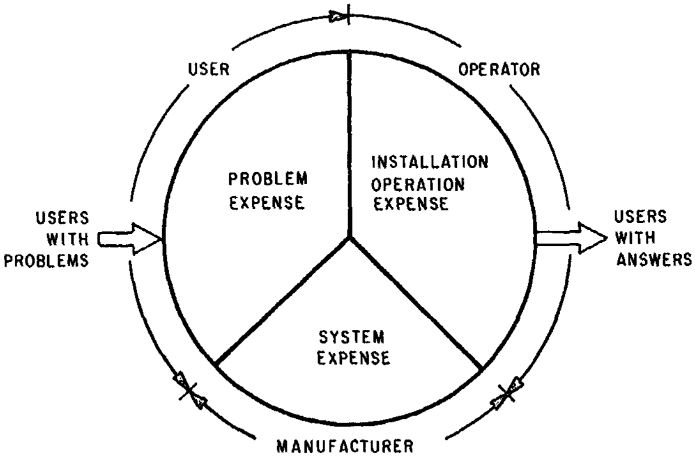


Figure 1.1 Total Cost of NASF Usage

1.2.1 Hardware

The system originally defined during the Preliminary Studies and modified during this study is shown conceptually in Figure 1.2. The Flow Model Processor (FMP), which provides the required computational power, is a dedicated computing engine with an architecture based on the special needs of modeling. The Support Processor the Peripheral Support System and the File System together constitute the Support Processing System. The Support Processing System interfaces with the users, maintains the data files, and controls the flow of jobs and data to and from the FMP. Not shown in the figure are the support elements including building, power, office space and cooling.

The architecture of the Flow Model Processor is based on the needs of discrete modeling and simulation. The FMP, which is described in more detail later, has 512 processors that normally would execute independent of, and concurrent with each other. A coordinator is used to allow the processors to execute in synchronism. The processors each have memory space for programs and data. In addition, a large memory (called the Extended Memory) can be accessed by all processors through a high-speed network called the Connection Network. The Extended Memory normally would contain the data common to the processes being independently evaluated

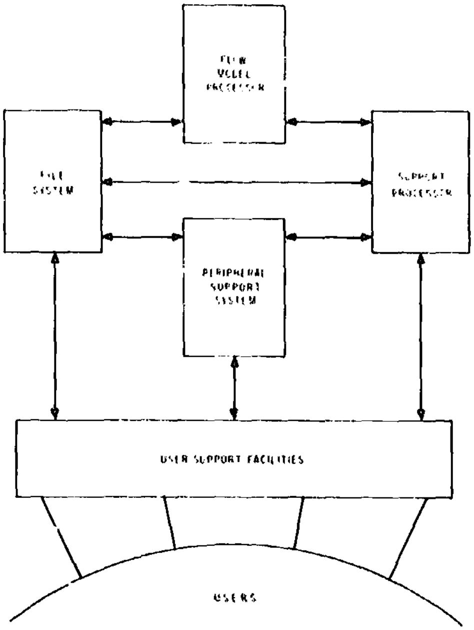


Figure 1.2 NASF Organization

each of the processors. Finally a slower staging memory (called Data Base Memory) would be provided to hold the next job, the last job and the current job. The Data Base Memory buffers programs and data in order to provide a smooth flow of tasks to and from the FMP. The memory sizes assumed during the study were based on the aerodynamic flow codes that are expected to be the primary application on the FMP.

The Support Processing System would consist of three portions; the Support Processor, the File System, and the Peripheral Support System. The Support Processor (the host processor) would run the main portion of the operating system (called the Master Control Program). A dual-processor B7800 was assumed for evaluation purposes. Most of the user interaction with the NASF would be through the Support Processor. The File System includes disk packs, an archival store, and the manager of the files. Data paths to and from the files would exist for the FMP, for the Support Processor, and for user support. The third element considered as part of the Support Processing System is the Peripheral Support System. The Peripheral Support System has been included because the evaluations performed in the study demonstrated that at least one of the supportive tasks involved such a level of work that a special processor for that task should be considered. In particular, the evaluations demonstrated an exceptionally heavy load can be expected to support Computer Output to Microfilm (COM). This load may be in excess of 10,000 frames of graphic information per day. The Peripheral Support System would include facilities specially designed to support such exceptional loads in order to improve the load balance across the entire facility.

1.2.2 Software

Not shown in Figure 1.2 is the software which would be used to support users and to control the efficient usage of the resources within the facility. A dialect of FORTRAN, called FMP FORTRAN, has been proposed which has a few simple extensions to standard FORTRAN. These extensions provide application-oriented approaches to use both the independent, concurrent mode of operation. In addition, statements are included which are capable of using a large number of processors at once on a single computation. Since the Support Processor would be a commercially available processor, standard languages such as ALGOL, FORTRAN, and COBOL would be used for process definition on that processor. The File System would not be programmed by the users, but would provide high-level file management and access capabilities.

The NASF operating system (called the Master Control Program, or MCP) would reside, in part, on all elements of the system. Since the Master Control Program (MCP) would be based on existing software, the major portion would reside on the Support Processor. The portion of the MCP on the FMP would manage the flow of jobs within the FMP and would be the primary focus of confidence and diagnostic procedures within the FMP.

1.ح.3 Fault Tolerance

Since the FMP will have between 200,000 and 250,000 integrated circuits, plus other components, both hard failures and transient failures can be expected. Means for preserving the integrity of the computation in the face of such failures must be provided. The level of Large Scale Integration to be used is expected to bring forth failure modes that have not been important in the past, such as background radiation which may cause transient errors in Data Base Memory. Defense against all these possibilities must be included, and has been included in the architecture described in this report. Where economically feasible, mechanisms for error correction have been included such as use of single error correction, double error detection (SECDED) codes in all memories. To reduce the probability of double errors in those memories where transient failures may be expected, mechanisms to "scrub" the memory by rewriting data back into memory with the errors corrected are provided. For the various types of faults which can be detected but are not easily corrected, on-line spare processors and memory modules can be automatically switched in under control of the MCP to replace failed elements.

Not only was the FMP considered when developing the necessary fault tolerant aspects of the system. The CPU in the B7800 Support Processor is duplexed, for example, as are the Data Communications and Input Output Processors. A distributed control scheme and a multiplicity of disk packs within the File System serve to keep the system available for useful work without having each and every one of them available at any given instant. The automatic recovery procedures in the software not only support the FMP as mentioned earlier, but exist as a standard part of the MCP in the Support Processor.

1.3 NASF EVALUATION

Evaluation of the NASF considered many aspects. Three specific issues received the major attention in terms of analysis performed. These issues were an evaluation of system-level capabilities to support the general work load of the facility, an evaluation of the throughput of the FMP using real programs, and an analysis of the availability, reliability, and maintainability of the system. The general approach used for the evaluation and the results observed is described below for each of these three areas. As a result of these evaluations and the other work to date, those areas which contribute to the risks of the program were identified. These areas, which relate to the assurance of success of the program are explained below.

1.3.1 System Utilization Studies

The evaluation of the NASF system organization showed the feasibility of the system to support the expected workloads. This evaluation was based on a hypothetical, but well thought out, workload supplied by NASA [4]. System-level models were developed and used as the basis of the implementation of system analyzer programs. The models were operationally based so that they may be easily verified by direct observation of an actual system as development might progress.

The system-level evaluation included consideration of the following:

FMP Loading
Support Processor CPU Loading
Average Data Transfer Rates between Files, Users FMP and Support Processor
Expected number of file management actions such as file creation, deletion, and accessing.

The results of the evaluation show that the dual-processor B7800 assumed could comfortably handle the expected load with the exception of the COM support activities discussed earlier. More significantly, if projection is made to equivalent processors which are likely to be available before the implementation of the facility, such processors could handle a significant amount of the COM support load. The average data transfer rates projected by the analysis are well below the channel capacities planned. Although more analysis of peak rate requirements has yet to be performed, the projections to date are consistent with the expected results.

1.3.2 Flow Model Processor Throughput Evaluation

Throughput of the FMP was evaluated by measuring, in simulation and by analysis, its performance on complete programs supplied by NASA. The use of entire programs for measuring performance avoids a common pitfall in predicting the performance of new and advanced computers, namely the reliance on throughput evaluations which look only at the "hard" parts of the problems, which also are by no coincidence the parts of the problem that the advanced computer is designed to work best on.

The results of the analysis of the two aerodynamic flow codes (referred to as aero flow codes) show that the goals for throughput for aero flow applications are met. One aero flow code, identified as the "3D implicit" code was projected to execute in less than five minutes at a throughput rate of 1.01 billion floating point operations per second. The second aero flow code, identified as the "3D explicit" code was projected to execute in less than seven minutes at a throughput rate of 0.89 billion floating point operations per second. Both codes were evaluated at the nominal size expected to run on the FMP, specifically one million grid points.

The results of the analysis of the weather codes shows that the FMP, as evaluated, is optimized for the weather codes as well. NASA supplied two weather (or climate) codes. The first was a version of the Mintz-Arakawa algorithm, as developed by the Goddard Institute for Space Studies ("GISS"); the second was a spectra. weather code. The same detailed analysis was applied to the GISS weather that had been applied to the aerodynamic codes. Fourteen days of simulated weather, with 20 minute time steps, in a 2.5° (latitude and longitude) model with a total of 115,334 grid points, would take 8 minutes to run on the FMP with an effective throughput rate of 0.53 billion floating point opertions per second. Scrutiny of the second weather code showed that it could be expected to run with slightly higher throughput than the GISS weather, but the detailed analysis was not made.

The analysis was very thorough. All programs evaluated were dissected into code segments, each of which was internally homogeneous. The throughput was estimated for each individual code segment. From an analysis of how often each code segment was executed, the individual throughput estimates were combined into an overall execution time and throughput rate.

As a verification of the hand analysis, sections of code were input to an instruction timing simulator. The code sections chosen for simulation verified throughput rates ranging from less than 0.1 GFLOPS to more than 1.5 GFLOPS. The instruction timing simulator was based on a reasonably detailed model of a processor in the FMP. The instruction times assumed in the model correspond to what could be expected using good engineering practices and a modern circuit family such as the Fairchild 100K family of ECL circuits. The times assumed in the model for access to the common memory via the Connection Network were based on detailed analysis of the Connection Network itself. A CN simulator was developed and used to analyze various access patterns including some taken from the aero flow codes. A stochastic analyzer was used to determine the probability of success in making connections. The stochastic analyzer used probability equations for analysis. Both methods validate a transfer rate through the connection network of over one billion words per second from all processors to all memory modules.

The analysis of the various programs required preparation of FMP FORTRAN versions to be used in the analysis and as the starting point for hand-compilation onto the instruction timing simulator. The conversion from the FORTRAN code supplied to FMP FORTRAN was generally straightforward. In some cases, significant reductions in the length of the code could be made because of the application-orientation of FMP FORTRAN.

1.3.3 Availability, Reliability and Maintainability Evaluations

Several methods were used to evaluate the availability, reliability, and maintainability of the NASF. The predictions for the FMP are based upon a computer model of reliability and availability with assumptions that are derived from the military standard methods for estimating reliability. In an attempt to be as realistic as possible, field data which included failures due to system software as well as hardware was used. In addition, intermittant failure modes were modeled, where the rate of intermittants was based on field experience.

With the fault tolerance mechanisms in place, the availability forecasts are 99% for the FMP by itself and over 99% for the Support Processing System. These individual predictions combine to an NASF availability of over 98%. An estimate of 14.1 hours between interruptions of processing was also made as a result of the reliability and availability modeling. These predictions for the SPS are based on field data for the B7700, which is similar to the B7800 for reliability and availability.

1.3.4 Program Success Assurance

To assure the success of the NASF project, one must assure success in all areas. Some areas, being dependent mainly on existing technology or existing methods, were only briefly addressed during the study. Other areas of concern, especially where the NASF and its FMP represent a break with past experience, were addressed at greater length. A discussion of some of the key points addressed is summarized below.

Although outside the scope of the study, the need for continuing committment to the successful implementation of the NASF on both NASA's and the vendor's parts must be carefully considered. The close technical interaction that was so important to the Preliminary and Feasibility Studies must be continued. The length of time from the eventual start of design to delivery of the system is long. Project attention must be kept firmly on the job at hand. Continual changes of direction, dilution of effort, and expansion of goals could make the project seem to have a constant time-to-completion. This study has shown that a project begun now, with currently available or imminently expected technology, could deliver an operational system which would fulfill NASA's objectives.

Software development could have several potential problem areas. Software has been notoriously hard to schedule, often because of incomplete or changing specifications. Software is especially subject to the temptation to add "just one more little feature" making the resulting product more and more complex and difficult to test. This problem must be handled by careful management. The two major areas of software concern in the NASF are the operating system, and the language and compiler. The operating system (called the Master Control Program, MCP) would be based on the existing MCP of the B7800 planned as the Support Processor. This MCP has a history of 19 years of development behind it and is already being modified by Burroughs to support job flow to the computational engine for the Burroughs Scientific Processor. With this work substantially complete, the integration of the FMP becomes a task with much less risk.

Compiler development is another area often assumed to be a problem area. Here risk has been significantly reduced by proposing a language which is essentially ANSI Standard FORTRAN with a structure surrounding the FORTRAN pieces. This structure allows the FORTRAN pieces to map directly onto the many individual processors of the FMP. The result is that most of the compilation is the same serial FORTRAN to processor-level code process that industry and Burroughs has considerable experience with. The coordination between the pieces of standard FORTRAN is simply described by the added structure and maps easily onto the section of the FMP specifically designed for such coordination (i. e., the coordinator).

As a result of the approaches proposed and evaluated during the study, the success of implementation of the necessary software seems assured.

Hardware presents no threat to the success of the project. The technology projections made during the Preliminary Study [2, 3] are proving to be conservative. Logic design would be straightforward and presents little in the way of new challenges. The organization considered is very modular which would allow implementation of the system with only a few types of modules. The one area in the hardware which represents a feature not found so far in any commercial computer is the Connection Network. This network provides the necessary data paths between the many processors and the large, common memory in the FMP. This network has been thoroughly simulated and otherwise analyzed during the course of this study.

1.4 CONCLUSION

The work summarized above has demonstrated the feasibility of the Numerical Aerodynamic Simulation Facility. Although some risks have been identified, the level of risk is low for the architecture and software considered during the evaluation. This system is believed to be the best approach to meeting the total system goals for the NASF. In particular, with these concepts no new advances, beyond the technology available today, are needed in order to successfully implement the facility.

CHAPTER 2

NASF SYSTEM ARCHITECTURE

As indicated in Chapter 1, the feasibility study of the NASF required broad consideration of the total needs of the proposed facility and of the expected user community. Because of time and budget constraints, detailed study was based on commercially available equipment wherever possible. The system architecture used for evaluation is substantially the same as that described during the Preliminary Study [1, 2,]. However, some changes were indicated, based on this feasibility study. The modeling which was done in support of this study was operationally based. That is, the system-level models are designed so that they may be easily verified by direct observation of an actual system. This approach was chosen to make future verification of the models straightforward.

2.1 OPERATIONAL ENVIRONMENT

Before considering the system architecture in detail, it is important to first consider how the facility is expected to support the user community. The planned operational environment of the NASF has been reviewed in two documents provided by NASA [3, 4]. The central computational facility (which includes the Flow Model Processor and a Support Processing System) will be accessed by a number of users at sites remote from the facility. Some of the "remote" sites would be physically nearby (such as the NASA Ames facility) while others would be at distant locations.

For the purposes of the study, some assumptions were made concerning the users. The operational environment described by NASA shows that many of the users will be directly concerned with production use of the facility for design work. These production users have been assumed, for purposes of the study, to be working in design teams at "design centers". These design centers have been assumed to have sophisticated graphics, processing, file storage, and communications capabilities. These design centers would reduce the processing load of the facility.

The NASA documents also pointed out that other users will be involved with code development, method development, and research in fluid physics and other areas. Some of these users have been assumed to be associated with the design centers, at least as far as use of facilities are concerned. Other users would have direct access to the computational facility from their terminals.

Figure 2.1 depicts the assumed operational environment with the central computational facility of the NASF at the top and with users having access to that facility either via terminals or via design centers. Figure 2.2 depicts the organization of a design center. All sophisticated graphics equipment was assumed to be associated with design centers. The processors which are part of each design center were assumed to provide support to the users both in terms of graphics I/O operations and in terms of text and file handling. If the "nearby" design centers are assumed to support fourteen active users and if the "remote" centers are assumed to support four active users, the configuration shown in Figure 2.1 would have at least 100 active users.

The design centers have not been studied further and are certainly not a required part of the overall system. The main reason for their consideration was to develop a realistic estimate of the amount of load on the Support Processing System for text input and editing tasks. Based on the environment just described, the fraction of users who require the Support Processor for data entry and editing was assumed to be 0.2. The other 80% of the users either use the facilities of a design center, or have terminals with built-in edit mode capabilities.

2.2 SYSTEM DESCRIPTION

The NASF consists of three elements; the Flow Model Processor (FMP), the Support Processing System (SPS) and the physical environment including the building, power, cooling, etc.

2.2.1 FMP

The Flow Model Processor (FMP) is a dedicated, single-user-at-at-time computing engine which has no I/O capabilities except through a staging memory. The FMP is based on a large number of independent processors, each executing FORTRAN code independently of the other. The extensions to FORTRAN described in Chapter 4 include constructs which allow description of significant amounts of independent, concurrent operations. In addition, provision was made (in both the hardware and software) to allow a single computation to utilize a large number of processors. The FMP also includes a very wide bandwidth memory that can be shared by all the processors. The memory sizes assumed for the study were based on the aero flow programs used for evaluation during the study. More details are included in Chapter 5.

2.2.2 Support Processing System

The Support Processing System serves as the central control, interfaces with users and peripherals, maintains the data files and provides that computational support necessary to keep the FMP effectively utilized. The Support Processing System consists of three portions; a host processor called the Support Processor, a File System, and a Peripheral Support System. Most of the discussion throughout the rest of this report refers only to the Support Processor (which is the host) and to the File System.

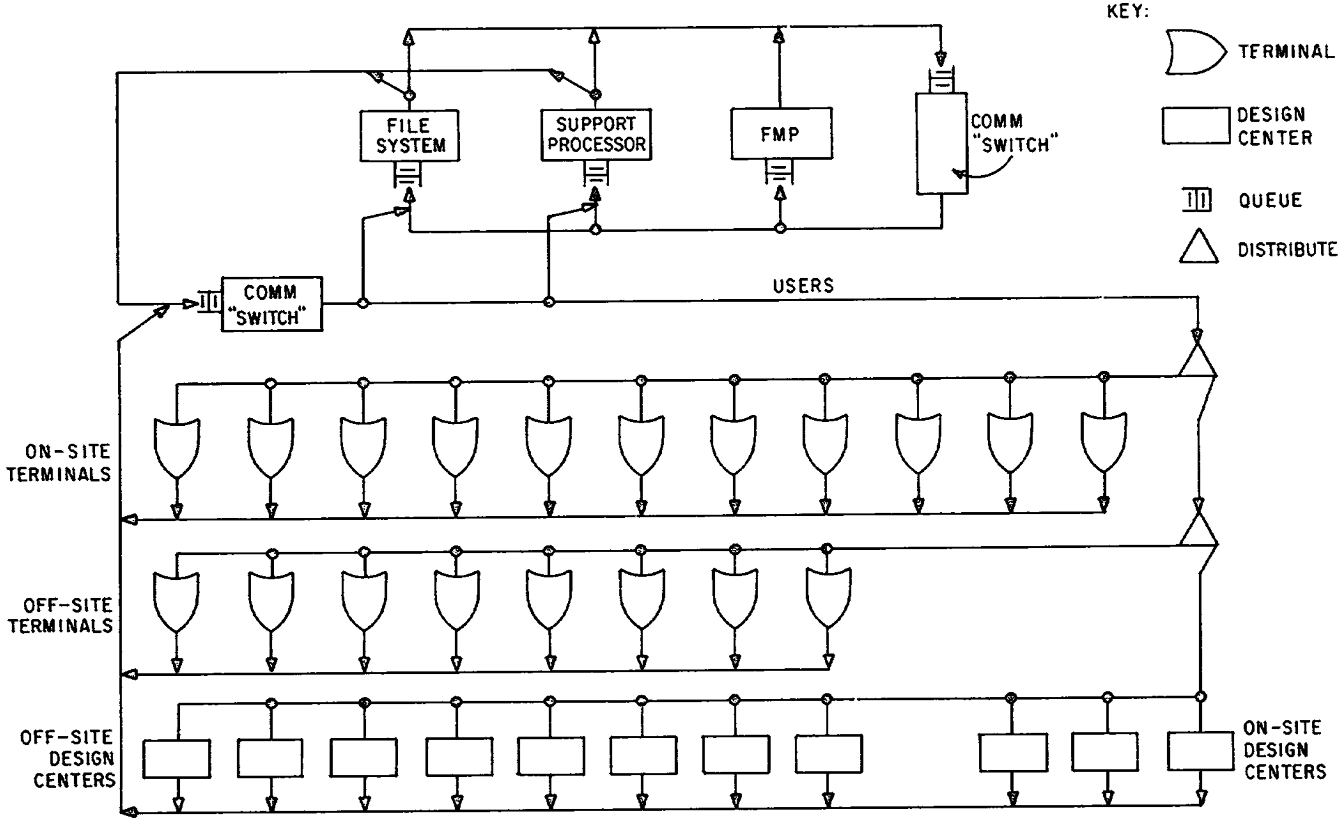


Figure 2.1 NASF Operational Environment

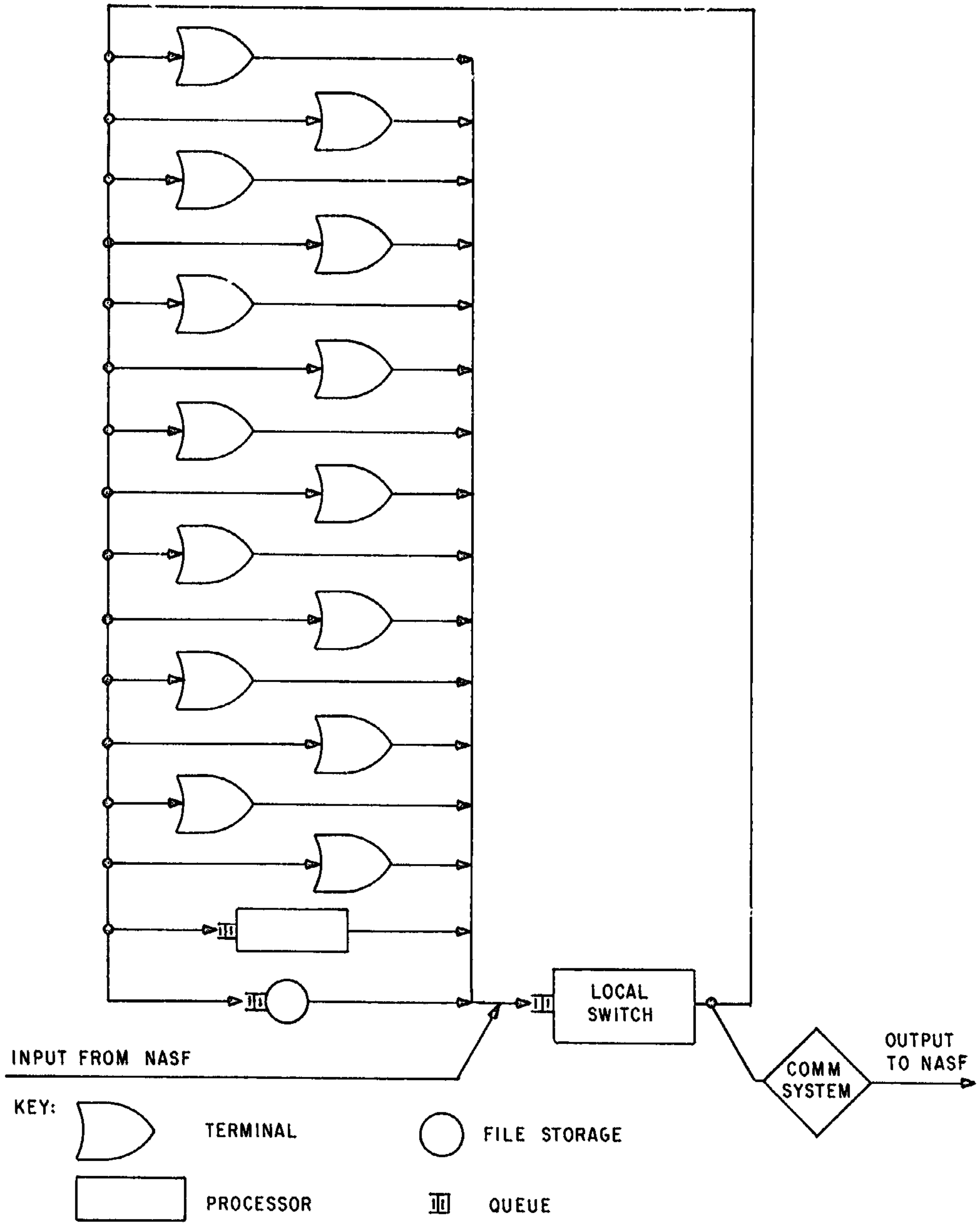


Figure 2.2 NASF Design Center Organization

2.2.2.1 Support Processor

The host processor, which is identified as the Support Processor during this report, was assumed to be a dual-processor B7800 for the purposes of the study. This processor was chosen for two major reasons. First, the B7800 system is a new, standard product which has evolved from the Burroughs 700 and 800 series machines over the past 16 years. A wide range of data communications and peripheral support is available on this system. Second, because the B7800 is an evolutionary system, it supports the Master Control Program (operating system), compilers, utilities, and application programs developed by Burroughs for the B6000 and B7000 series processors. The feasibility of this system for control of the FMP seems clear since the same functions are already being implemented for the Burroughs Scientific Processor (BSP) which also attaches to the B7800 system.

The B7800 employs independent functional processing to distribute both intelligence and control among various processing elements. The B7800 includes five independent functional processors. They include the central processor, the input/output processor, a memory control processor, a communications processor, and a maintenance and diagnostic processor. The configuration assumed for the study includes redundancy in essentially all elements of the system, resulting in very high availability.

Since the Support Processor would be the master control for the facility, most user communications would be supported with the Data Communications Processor portion of the B7800. The configuration assumed for the study included 96 input lines, of which four were synchronous broadband lines (19.2 Kbps - 1,344 Kbps). The remainder were assumed to be a combination of synchronous and asynchronous lines of various rates (1.2 Kbps to 9.6 Kbps).

In addition to the standard line control disciplines, the B7800 and its Data Communications Processor would provide the capability for network access and control. This capability would provide the needed flexibility for potential users to be connected to the facility "on their terms".

2.2.2.2 Peripheral Support System

In addition to the input/output processors on the B7800 Support Processor, the study demonstrated that some peripherals would require a significant amount of computational support. The most significant of these devices is the Computer Output to Microfilm (COM) device. The NASA supplied scenario of usage [4] postulated a very heavy COM load (in excess of 10,000 frames per day) where the output was assumed to be graphic images. The majority of this load was for "movies" of complex evaluation results.

The system utilization analysis (summarized in Section 2.3 below) clearly demonstrated the impact of the COM formatting, even when the formatting was only to produce listings of the points of interest rather than graphics control procedures. This load could be supported with additional central processors within the Support Processor. Alternatively, the load could be supported by doing the necessary formatting in the FMP prior to FMP task completion. A third alternative was to consider a separate system, specifically oriented to supporting this formatting and the COM device.

No study has been completed with regard to what impact the COM formatting would have on FMP loading. By studying the Support Processor loading with and without the COM formatting task, it was clear that one feasible way to support a load of this sort was with a specialized system specifically planned for that support. Although further study should be performed, a Peripheral Support System configured with two high-end minicomputers with special-purpose software should be capable of handling the required tasks.

2.2.2.3 File System

A separately managed and accessed file system is required as part of the Support Processing System. The volume of data and programs which will be moved in and out of the Flow Model Processor together with the amount of file management required for the total system indicate strongly that a separate system be provided for this purpose (rather than using the Support Processor itself for example). Secondly, when file management functions are in a processor different from any processors which may be executing user programs, the confidence in security capabilities can increase significantly. The File System includes the disk packs, the archival store, and the file manager. Conceptually, the File System also includes the Data Base Memory, the staging area for programs and data within the FMP.

The File System is another part of the facility where a detailed study has not been completed. Enough is known about the requirements of the File System to be confident that such a system can be configured from essentially standard components. These requirements will be summarized below.

The File System should be organized such that many simultaneous high-speed transfers are possible. The NASF architecture requires four major connections to the File System; the FMP, the Support Processor, the Peripheral Support System, and the Users. In addition, the File System would be capable of responding to requests for data movement within the File System and would provide automatic management of the space.

The FMP requires up to four simultaneous (12.5 Mbits/sec each) paths to and from the File System, although the use of these paths is not continuous. The peak requirement of each of the two Input-Output Processors of the B7800 Support Processor is also 50 Mbits/sec each, which like the FMP connection, is primarily disk I/O.

The Peripheral Support System requirements are insignificant by comparison. The interface from Users to the File System would be for the purpose of accessing graphics data files without having to funnel such requests through the B7800 Support Processor. Again, since the User loading would be on the order of the Peripheral Support System loading, the User loading would be supported well if the FMP and Support Processor are supported. The technology for connections of this sort has already been reported in the literature [15] and is in production. Equipment to support 50 Mbits/sec per channel is available now. The major thrust of development, at least at Burroughs, is to significantly reduce the cost of this technology or an equivalent.

The file manager would be expected to handle approximately 10,000 file creations and deletions per day. The File System would respond to approximately 25,000 requests for file accesses per day. All interfaces to the file system would be in terms of file "names" rather than physical media position. The File System would perform dictionary management and storage allocation functions. Also, the File System would be responsible for data ownership and access controls.

The analysis assumed a file configuration with both high-speed storage and lower-speed mass storage on-line. In particular more than 10^{11} bits of high-speed disk storage (25 msec average access, 3.6 MByte/sec transfer rate) was planned. More than 2 x 10^{12} bits of mass storage (3 seconds average access, 1 MByte/sec transfer rate) was also planned. Although these appear more than adequate, the utilization studies described below have not yet considered what file capacities would be required given tne scenarios supplied by NASA late in the period of the study.

2.3 SYSTEM UTILIZATION STUDIES

The feasibility of the ability of the NASF system organization to be able to support the expected workloads was evaluated. The evaluation is summarized below and discussed in more detail in Appendix F. In summary, the system organization described above would be capable of supporting the workload hypothesized by NASA [4].

The NASF Utilization document [4] provided by NASA described the use of the facility in terms of class of usage, called Cases, and in terms of the sequence of Tasks performed for each job. The Cases (such as method and code development or design simulations) and Tasks are summarized in Appendix F. Before confidence could be gained that the system could support the projected load, the committment of each system-level resource to each task was carefully charted. These event sequence charts identify the sequence of events needed to implement each task. The charts also identify those system resources (File System, Data Comm, Support Processor, FMP, ...) which must interact to implement each event. Samples of these charts are also included in Appendix F.

The charts were then used as a model to develop a program which was used to analyze the impact of the hypothetical workload on the various components of the system. Some of the data used in the analysis program was based on a benchmark of a mix of FORTRAN programs on a B7700. A known factor of improvement was used to project expected B7800 performance. In addition, a processor which might be available about the time that the NASF project would be implemented was hypothesized and used for evaluation purposes.

Table 2.1 summarizes the results of the analysis of the Support Processor loading with and without the COM formatting discussed earlier.

TABLE 2.1

Support Processor CPU Hours Needed/Hour
(Averaged over Day)

Processor	With COM	Without COM
Similar to B7700	14.2	1.3
Similar to B7800	9.5	.9
"Future Processor"	2.8	.2

In Table 2.1 note that a support processor implemented with the future processors expected to be available to the NASF project could support the COM workload with a reasonable-sized system (3-4 central processors).

Table 2.2 summarizes the Data Tra~sfer Requirements averaged over the day and by shift. Note that these data transfer rates only show the average rates, not the peak rates needed to prevent the data path from being a bottleneck. The daily average is over a full 24 hour day. The data rate (char/sec) assumes 8-bit characters plus error control.

TABLE 2.2

NASF Data Transfer Requirements

	RATE (Char/Sec)			
	Daily Average	Hourly Average		
		12M-3am	5am-5pm	5pm-12M
Support Processor - File System	29,240	83,388	16,678	35,937
Support Processor - FMP	.050	.02	.08	.02
Support Processor - Users	4,453	228	8,125	187
File System - Users	24,260	3,002	45,900	1,554
File System - FMP	163,400	294,770	210,032	73,770

Table 2.3 summarizes the File System control activity by day. The terms ACTIVE, LONGTERM, and ARCHIVE in the table indicate the different types of files expected to be found in the File System. Active files are those only recently created or actively used and would be on the devices with the fastest access times. Longterm files are those which have been in the active system for up to a week with little or no use before being copied onto a slower media. Some files are saved on on-line mass storage, called the Archive in the table. These files would have an access time on the order of seconds but would still be on-line.

TABLE 2.3

NASF File System Control Activity per Day

FILE ACTIVITY	FILE TYPE		
	ACTIVE	LONGTERM	ARCHIVE
Files Created	2483	1127	627.3
Files Deleted	2483	1127	627.3
Files Accessed	19810	827.7	118.3
Files Replaced	1302	---	---

The analysis performed to date is sufficient to give one confidence that the system studied would be capable of supporting the hypothesized workload. Before design can begin, more detailed studies should be performed to determine more accurate estimates of grid generation task requirements, the impact of interactive graphics support tasks and the sensitivity of system support to all parts of the hypothesized workload.

CHAPTER 3

APPLICATION ANALYSIS

3.1 INTRODUCTION

The requirement for performance for the FMP was initially stated as the execution of the "typical" 3D Navier-Stokes aero flow code on 200 x 50 x 100 grids in 10 minutes, with the provision that the FMP should be a flexibly programmable machine that can run a number of similar applications with similar throughput. These throughput goals can be restated, with respect to the sample aero flow codes supplied by NASA, in terms of a more hardware-related secondary standard of performance, that the FMP should be capable of achieving a sustained rate of 1.0 Gflops/sec on aero flow codes that take advantage of its architecture. These goals were met, as described in more detail below.

3.2 PRODUCTION APPLICATIONS

The statement of work specifically asked for a design that is adapted to the requirements of computational aerodynamic programming, with a secondary look at the requirements of weather computations. NASA supplied two examples of aerodynamic flow codes, identified as the "3D explicit" code and the "3D implicit" code. In addition, two programs exemplifying the weather applications were supplied, one being a Goddard Institution of Space Studies (GISS) version of the Mintz-Arakawa global circulation model, the other one being a spectral weather code from MIT (Spectral).

3.2.1 Functional Requirements

The application areas of interest, as exemplified by the codes supplied, represent a substantially different spectrum of applications that one would arrive at by questioning all of the users of very high speed numerical computation.

A general purpose very high speed numerical computing machine must support a wide variety of precision requirements. For example, users with sparse and ill-conditioned matrices, such as one finds in some structures applications, require very high precision, for some users well over 30 decimal digits. Aero flow and weather codes apparently will run happily with not more than 10 or 12 decimal places of precision, with much of the computation and most of the data needing only six or seven places of precision.

It has been appreciated for two decades that the speed of light puts an upper limit to the throughput of an uniprocessor, and that very high-speed machines must use some sort of parallelism or concurrency in order to achieve throughput. Traditionally, parallelism has taken two forms, first, vector machines in which only data with extreme regularity could be processed in parallel, and second, multiprocessors in which many totally independent programs run in parallel. Because aero flow and weather codes can be vectorized, a vector machine could be made to work. However, the vectorization imposes inefficiencies (for example, subroutine CHARAC in the 3D explicit, or COMP3 out of the GISS weather). As a result a machine that is efficient only for vectors is often not efficient when considering all of the programs that one expects that computational fluid dynamicists would want to write.

Hence, part of the problem is to demonstrate the feasibility of a flow model processor that is as efficient on vectors as the traditional vector machine, and is also efficient when the concurrent processing is on data that does not form vectors. Furthermore, the language should allow for the description of parallel (or vector) operations and for concurrent scalar processes which are independent of each other, or for any mixture of the two.

Demonstration of optimum feasibility of the FMP for its application set therefore includes:

- Provision of concurrency (or parallelism) for high throughput without the requirement for vectorization of the algorithm. Although the implicit algorithm is easily vectorized, and the 3D explicit is also easily vectorized, the earlier 2D explicit was not all easily vectorizable, and a large portion of weather (subroutine COMP3) can be vectorized only with difficulty and a large penalty in throughput.

- A language (FMP FORTRAN) in which one can write either nonvectorized concurrent operations, or vector operations.

- Word size. The computational fluid dynamics and weather community requires no more than 10 or 12 decimal digits of precision, corresponding to 33 or 36 bits in the fraction part of the floating point word, with some computation and most data requiring no more than six or seven digits (24 bit fraction part) of precision. This is not true of other "typical" users of very high-speed numeric processing. Requirements for precision run from 8 bits, for picture processing, to over 30 decimal digits, for users with large, sparse, ill-conditioned matrices, typically structures and applications. A large number of scientific processor users desire 14 to 16 decimal digits of precision.

- A language based on FORTRAN to accomodate the applications programmers, who, in the computational fluid dynamics and weather communities, have mostly been used to working in one dialect or another of FORTRAN.

3.2.2 Projected Performance, Summary

For the aero flow codes, the FMP here described would run the 3D implicit in 6 minutes and 16 seconds (100 times steps) at a throughput rate of 1.01 Gflops/sec* during that time. The 3D explicit runs in 8 minutes and 52 seconds (again, for a test case with 100 time steps) at a throughput rate of 0.89 Gflops/sec during that time. In both cases, the mesh had a million grid points (100 x 50 x 200 in the case of the implicit, 100 x 100 x 100 for the explicit). Feasibility is therefore demonstrated.

Other metrics can be used to describe the "raw" throughput, of which the above is the net:

2.22 Gflops/sec would be the maximum throughput rate given that operations are alternately add and multiply.

1.74 Gflops/sec would be the maximum throughput rate for register-to-register operations using the instruction mix derived from analysis of the aero flow codes.

1.33 Gflops/sec would be the throughput rate seen in about half of the sequences of code submitted to the simulator.

Of the above, the figure of 1.33 Gflops/sec represents a throughput rate achieved by a number of real sequences of code, taken from both aero flow codes and from weather. It represents an achievable throughput for "friendly" applications.

All of the above refers just to the FMP. The throughput of the NASF is just as much dependent on proper function of the Support Processor System (SPS) as it is on the FMP. The SPS, however, presents well-known problems, not unique problems for the particular set of applications.

3.3 PERFORMANCE PROJECTION BASED ON BENCHMARK PROGRAMS

3.3.1 Summary

The four programs used as benchmarks in evaluating the design were:

- NASA 3D implicit aero flow code supplied by Ames
- NASA 3D explicit aero flow code supplied by Ames
- GISS weather code, in several different versions
- Spectral weather code from MIT

Evaluations of the first three were comprehensive, resulting in the projections of 1.01 Gflops/sec and 6 minues, 16 seconds, for the implicit, 0.89 Gflops/sec and 8 minutes, 52 seconds for the

*Gflops/sec, Billion floating point operations per second.

explicit at the size of the benchmark, and 0.53 Gflops/sec and 4 minutes, 25 seconds for the GISS weather code. Appendix A discusses these evaluations in detail. These evaluations and the conditions leading to these conclusions are summarized in this chapter.

The implicit code achieves the 1.0 Gflops/sec being used as a guide for evaluating adequate throughput rate. The explicit code nearly does. Since the intent of the explicit is to be computationally more efficient than the implicit, the performance goals are deemed demonstrated.

On GISS weather, the non-vectorizable portions of the code executed at more than one Gflops/sec (subroutine COMP3), while the throughput rate observed in vectorizable portions (COMP1 and COMP2) was reduced by EM accessing and memory-to-memory moves that produced no floating point operations.

Examination of the spectral weather shows that the fluid dynamics portion should run with higher flop rate than the fluid dynamics portion of the GISS weather (COMP1 and COMP2), and that the chemistry and physics portions were essentially identical to COMP3. Hence, the spectral weather is expected to run at a higher flop rate than the GISS weather.

3.3.2 Method

The method used for performance evaluation was generally the same for all of the first three benchmark programs. Because of time and budget limitations, only a cursory look was taken at the Spectral weather code.

Throughput was analyzed on the basis of FMP computations. I/O operations were ignored. Transfers between DBM and file system are independent of, and go in parallel with, the FMP computation. It is assumed that the file manager stages the next job, and unloads the last job, in times which are completely overlapped with current computation. DBM-EM transfers are also ignored, since they go on concurrently with current processing as long as EM space is available and take negligible time. The 15 million words of a restart point of a typical aero flow code are loaded in 0.375 seconds, which can be compared with the 600 seconds duration of a typical run. Therefore, both system I/O and user I/O were ignored.

Each program was analyzed to find the calling tree of the subroutines, and subroutines were divided into sequences of code that were internally similar. Analysis was performed on each individual sequence and the results combined, taking into account processor utilization percentages and number of exceptions, into total figures for each of the benchmark programs. Thus the analysis included every line of code in the first three programs.

Analysis consisted of hand compilation and simulation for a selected number of code sequences, and estimates based on interpolation between known simulations for the rest of the sequences. In the implicit aero flow code, over 60% of the computations of the program are within the inner loop which was simulated. For those sequences which were not simulated, a formula was developed which interpolated between the simulated sequences. For a more detailed description of this method, see Appendix A. Various cases in which exception should be taken to the formula were also taken into account. It was found that almost allof the simulation results could be empirically fit by a formula of the form

$$T =(k_1 F + k_2M + k_3D)/P \qquad (3.1)$$

where T is the time required to execute a particular code segment, F is the number of floating point operations in that segment, M is the number of EM accesses, and D is the number of divisions over and above the 2% divides assumed by the "standard" instruction mix, and p is the number of processors processing. The constants k_1 and k_2 are determined empirically from the simulation results, and k_3 was set equal to the time of a divide instruction. Throughput for the individual code segment is given by F/T.

The hand compilation made certain assumptions about the compiler. Assignment of instances of the DOALL to processor was not optimized, but done by the simplest algorithm conceivable (see Chapter 4 for software discussions). Optimization steps such as the substitution of an add or subtract from exponent to replace a multiplication or division by a power of 2 were assumed.

3.3.3 Throughput of Aero Flow Codes

The implicit aero flow code, for which simulation covered over 60% of the computations, was estimated to have a throughput rate of 1.01 Gflops/sec at the 100x50x200 size and ran 100 time steps in 6 min, 16 sec. The implicit code showed an estimated throughput rate of 0.89 Gflops/sec at the 100 x 100 x 100 mesh size and ran 100 time steps in 8 min. 52 sec. Details are in Appendix A.

The language being considered, FMP FORTRAN (described in more detail in Chapter 4), was found to fit the aero flow codes very conveniently. A simple, one-to-one translation from FORTRAN codes provided into FMP FORTRAN goes as follows. All arrays subscripted with the grid variables are made GLOBAL. DO loops (single or nested) on the grid variables are automatically turned into DOALLs as long as the data dependence allows it. Temporary variables are allowed to be LOCAL by default. The implicit code, as supplied by NASA, is of such regularity that practically all of it can be transformed into FMP FORTRAN using such simple rules. Because of this, and in order to save time, most of the FMP FORTRAN versions of the aero codes were not even written down, since they are obvious from the FORTRAN versions provided by inspection.

During the hand compilation process it was found that translation from FMP FORTRAN to FMP machine code was simple and straightforward. This gives confidence that the compiler will be relatively simple to write.

Procesor utilization ranged from 93% (in the explicit) to an average of 97.4% (in the implicit). Some routines gave 99.9% processor utilization. All subroutines and other code sequences were included in the total time and total number of floating point operations. In neither aero flow code did any of those sequences with low processor utilization have any influence on the final throughput estimate.

3.3.4 Weather and Climate Codes

Two benchmark programs were supplied by NASA Ames for use in evaluating the performance of the FMP for weather and climate codes. The first, a Goddard Institute of Space Studies version of the Mintz-Arakawa global circulation model, came in several different versions written for several different machines. These various versions are seen to have variations in portions of the algorithm. The version analyzed was one written for the 360/195. This is the same version that had previously been used as a test case for analyzing BSP throughput.

The second is a "spectral" weather code from MIT, in which an FFT is used to regularize the hydrodynamical computations.

The GISS code was analyzed at an intermediate grid size (2° latitude steps, 2.5° longitude increments along the equator with 20 minute time steps). The program consists of an easily vectorizable fluid dynamics section (subroutines COMP1 and COMP2 and the subroutines they in turn call), and COMP3 and its callees, the physics and chemistry section. The average throughput rate for the entire program was determined to be 0.532 Gflops/sec with a 14-day simulation taking 4 minutes, 25 seconds of FMP time.

The GISS climate code demonstrated the advantages of the FMP architecture. The vectorizable portions tended to run slow because of many EM accesses, but COMP3 and its subroutines ran as independent scalar processes in parallel in all the processors, achieving over 1.2 Gflops/sec for the portion simulated. COMP3 and its subroutines have been shown to be hard to vectorize for existing vector machines, whereas it is not necessary to vectorize them for the FMP.

In this benchmark, substantial use is made of parts of the language that see little or no use in the two aero flow codes, including:

- Domain definitions using domain expressions that include previously defined domains

- INALL arrays

- Initialization of values in declaration by arithmetic expressions

- NEXTDO

- Branching within DOALLS

This is a worst-case analysis, in that any data dependent branches were assumed to demand the most computations. This approach was used in order to estimate the worst-case maximum running time of the GISS climate code. Other conditions which simplify the radiation calculations (such as the existence of cloud cover) will result in fewer floating point operations, and shorter times. Whether the Gflops/sec rate would go up or down under these conditions depends on whether flops or elapsed time is reduced proportionately more. This case was not analyzed.

The spectral weather code is expected to run with substantially higher throughput than the GISS climate code does. Its fluid dynamics portions are done by spectral analysis, with each processor processing an FFT independently of all other processors. Thus, the fluid dynamics computations are much more locally contained, since all the intermediate results in the FFT can be contained within processor memory (declared either INALL or LOCAL) and should run faster. The chemistry and physics portions of the spectral weather code are substantially identical to the chemistry and physics portions of the GISS climate code, and the analysis of one can represent the analysis of the other.

3.3.5 Applications Beyond the Benchmarks

The analysis summarized earlier in this chapter and in Appendix A demonstrates the applicability of the FMP described in Chapter 5 to aero flow and weather codes. This analysis is therefore a constructive demonstration of the feasibility of the NASF. The FMP as described has broader applicability than to applications similar to the four benchmarks, as the remarks in this section will indicate. The following are considered:

- Single FFT (In the spectral weather code the 512 processors do 512 FFT's concurrently)

- Sort

- Problems too large to fit in Extended Memory (in "core")

In Section A.6 of Appendix A, the FFT is discussed. That section shows that the FMP runs various sizes of FFT at throughputs varying from 0.5 through 0.7 Gflops/sec. The reduction from 1.3 Gflops/sec is due to data rearrangement.

Section A.6 also discusses a method for achieving concurrency in sorting keys or data elements that are contained in processor memory. The particular method shown runs at 100% processor utilization when sorting an array of elements that start out being sorted in inverse order, such a case being a kind of worst-case test for some sorting algorithms.

3.3.5.1 Large Problems

The "standard" scenario for the use of the FMP is that all files necessary for the use of an FMP task are in place within DBM at the time the task is started. During the course of a run, the task is essentially self-contained within the "main memory", namely EM, CM and PM. This does not preclude reading from DBM to EM, or writing to DBM from EM at appropriate times. A set of files are located in DBM at the start of the run. Files may be created within DBM during the course of the run, snapshot dumps for instance, and when such files are closed by the FMP program, the file system has the option of moving them out of DBM before the run is finished. The concept of having the high-speed computations contained within a bounded portion of the hardware, here the FMP, with no interaction with external devices such as the support processor, has been given the name "computational envelope". However, the computational envelope is not completely sealed even during the "standard" scenario.

Another scenario is the running of tasks that will not fit in main memory. The following questions are considered. First, what facilities should be available with the initially delivered compiler; second, what facilities are envisioned for possible later implementation; and third, what problem properties allow efficient operation for problems that do not fit in memory.

The system evaluated during this study does not have the system software required for automatic virtual memory management for taking care of overflow from EM. Hence, the user programmer will have to insert READ and WRITE statements for access to DBM files. As with any other direct I/O scheme, it behooves the programmer to initiate I/O ahead of time, and test for completion at the point of using it. Only one direct I/O operation can be going on at one time. If a second direct I/O is called for before the first has finished, the program would wait for the first I/O operation to finish before initiating the second. User processing would be suspended until that second direct I/O is started.

The FMP hardware, which is described in more detail in Chapter 5, is intended to be able to support a virtual memory mechanism whereby certain Extended Memory (EM) files can be held in Data Base Memory (DBM) when EM does not have enough space. EM space would be dynamically allocated, and addressed with base registers, hence one possible implementation of such virtual memory is to have the base addresses of non-present data point outside of actual memory space. The address-out-of-bounds interrupt would trigger transfers between DBM and EM, plus some processing to fix up base addresses.

The system considered during this study does not have the system software to implement such a virtual memory scheme. This lack of automatic virtual memory management did not impact the throughput studies of the aero codes and GISS codes since these benchmarks will be able to reside within the planned EM space.

Hardware mechanisms that allow virtual memory for Processor Memory (PM) space using EM memory space to back up the PM space should also be planned. Methods for supplying this feature are still under discussion. One suggestion is that EM module No. P could be assigned to processor No. P, giving each processor its own private EM module for back-up for virtual memory purposes.

Some of the characteristics of an aero flow code that would execute satisfactorily, even though it would not fit in the Extended Memory (EM) can be determined by analysis independent of the method used to extend the storage capacity for problems into the Data Base Memory (DBM).

The following discussion is based on the 3D implicit aero flow code, whose major computational effort is in subroutine STEP and its subroutine BTRI. A listing of BTRI in FMP FORTRAN is included in Appendix H.

The bandwidth between EM and DBM (detailed in Chapter 5) is 50 million words per second. Since data overlay requires moving idle data out to make room for the new data, half of this, or up to 25 million words per second, is the rate that files can be brought in to be worked on. If the throughput rate (1.0 Gflops/sec) is to be maintained, there must be 40 or more floating point operations for every word brought into the EM from DBM. This goal can be met and then some. As an example, consider the following demonstration of one way of programming a 220 x 220 x 220 3D implicit aero flow program.

The data is blocked into 64 blocks, each 55 x 55 x 55. The organization of these blocks is shown in Fig 3.1. At any given time, four blocks forming a "pencil" in one direction will be in EM. Computation sweeps from one end of the pencil to the other and back again, so that having anything less than a pencil in EM will increase the amount of overlays between EM and DBM dramatically. Analysis of subroutines STEP and BTRI shows that there are about 84 floating point operations on each datum, larger than the 40 required for the desired throughput.

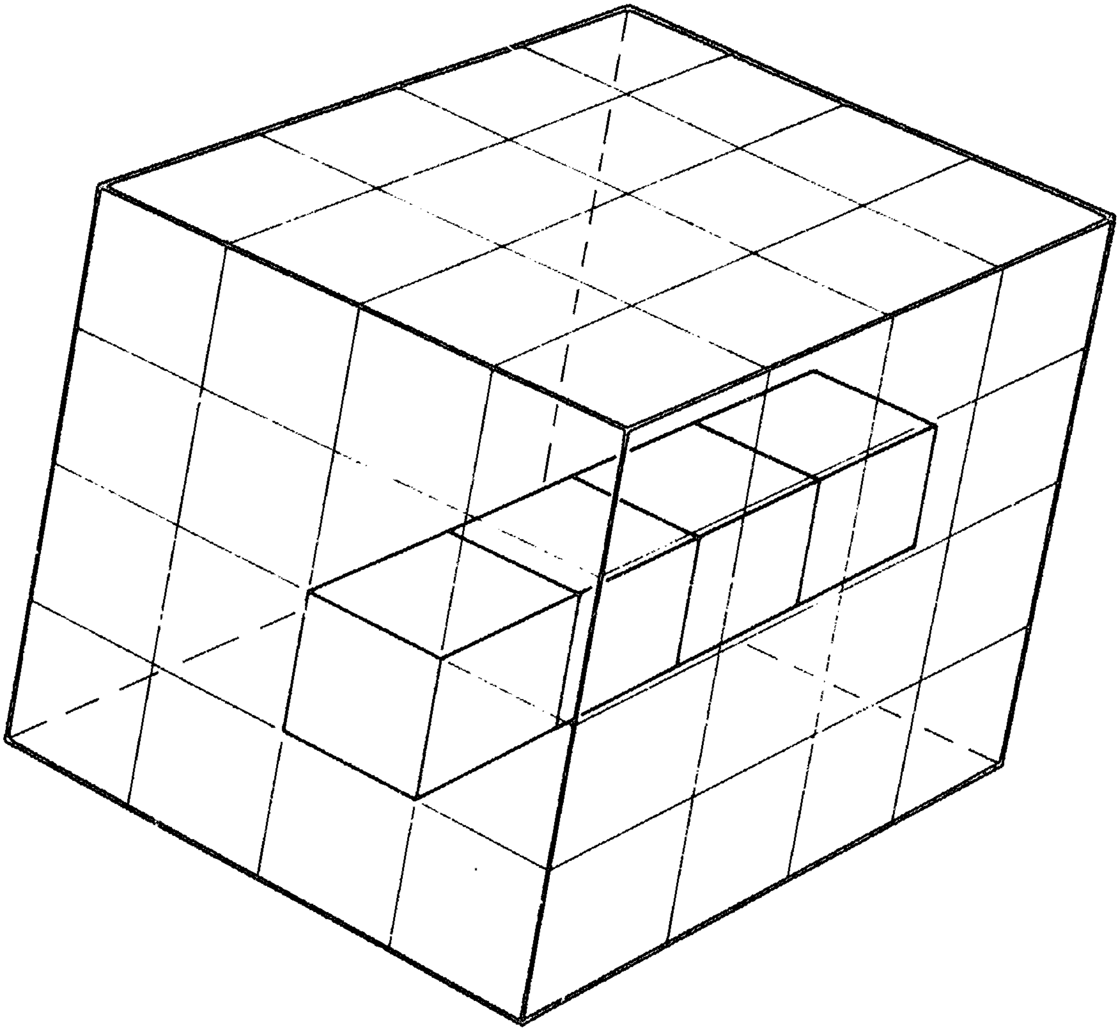


Figure 3.1 A Pencil of Four Blocks Taken From a Grid that Has Been ubdivided into Sixty-four Blocks

Memory requirements are dominated by the pencils. It is conventional in such overlaying situations to have one pencil in "core" being computed on, one pencil's worth of space allocated to the next pencil being brought in, and one pencil's worth of space allocated to the newly created data that is being written out. Since only one transfer is going on at a given time, in the present instance it may be possible to use the space vacated by newly created data to contain the next pencil, so that only two pencil's worth of space are needed. Assuming 15 variables per point, and 665,500 mesh points per pencil, three pencils (the worst case) would occupy 29,947,500 words in EM; two pencils (the more likely case) would occupy 19,965,000 words in EM.

Although the EM-DBM data transfers are completely hidden behind computation, and do not slow down the throughput, there will be a throughput reduction from the 1.01 Gflops/sec analyzed in Appendix A from another cause. Not all the arrays declared LOCAL in the 3D implicit of the analysis, will fit in processor memory. Some of these arrays will have to be held in EM, where the access time is longer. Alternatively, recomputation can be used to avoid the saving of precomputed results.

After sweeping 16 such pencils in one direction, direction is switched and 16 pencils are swept in the second direction, and then in the third.

Virtual memory machines have been on the market for 19 years at least; the Burroughs B5000 is an early example. All of the commercially available virtual memory mechanisms show varying degrees of throughput reduction when the data base for the problem is larger than the main memory of the machine. When the programmer controls his own direct I/O, there is the opportunity for favorable cases, such as the implicit aero flow above, to achieve full machine throughput on problems too large to fit in main memory.

3.3.6 Application Domain

The primary area of application of the FMP, according to the statement of work, will be the aero flow codes. A secondary area of application will be the weather and climate codes. Analysis of the benchmark programs shows that for reasonable grid sizes, the desired throughput is achieved. The range of problem sizes for which the throughput applies is analyzed here, as well as what is the largest problem that will fit in the DBM using the approach described in the previous Section 3.3.5.

In the aero flow code, the smallest "good" grid size is that which permits two dimensional DOALLs to run with reasonable efficiency. Hence, the smallest grid has a single dimension of not much smaller than $\sqrt{512}$. A grid of 22 x 22 x 22 is the smallest that runs with 94% processor utilization or better. The largest aero flow code is the largest one whose data base will fit in EM. Assuming fifteen variables per grid point, and 2^{25} words in the EM address space, that is a grid of about 2.2 x 10^6 grid points. Other EM space requirements reduce the figure. The largest program that will fit in EM and DBM using direct I/O has a complete data base allocated to DBM. At the currently specified size of DBM, namely 2^{27} words, this is an upper limit of about 9 x 10^6 grid points. If a larger upper limit were required, the size of DBM could easily be increased.

In the weather and climate codes, the grid has a much smaller dimension in height than it does in latitude or longitude. Not all weather code grids are in terms of longitude and latitude; other two-dimensional grids can be mapped onto the surface of the earth as well. In the GISS climate code, as translated for the FMP, almost all of the DOALLs are on a single layer. Thus, as long as that layer is nearly 512 elements, the bulk of the computations will be done with good processor utilization. The smallest "good" weather problem would be one with 15° longitude spacing along the equator, a grid of 20 x 24 in each layer.

Subroutine AVRX of the GISS code represents a non-negligible portion of the computation. Appendix A describes five different ways that AVRX may be mapped onto the FMP. Any one of the five ways will work, but all have some drawback. The throughput of AVRX will be poorer at the smaller grid sizes, and the preferred implementation may vary as a function of grid size. Hence, the throughput estimates of Appendix A (0.53 Gflops/sec), will have to be revised somewhat for different grid sizes to take into account the effects of AVRX*. At the grid size of 89 x 144 analyzed in Appendix A. AVRX was 2.2% of the running time, executing at 0.065 Gflops/sec.

The largest weather code that will fit in memory, assuming 16 variables per grid point, would be a grid size of about 432 x 268 x 15 levels, or roughly three times the resolution of the case explored. Alternatively, a grid size of 512 x 320 x 12 would also fit. As with the aero flow codes, a grid with four times as many points will fit into DBM, say 864 x 536 x 15 or 1024 x 640 x 12.

Running time on these latter codes would be quite long. Doubling the resolution roughly raises the running time by a factor of 8, assuming the same number of levels, since the spatial resolution and the time resolution are roughly proportionate. Hence, with a grid size of 432 x 268 x 9, which is triple the resolution of the analyzed case, 27 times the running time of the 89 x 144 x 7 grid size analyzed is expected. At this triple resolution, a fourteen day run, with a 7 minute time step, would take roughly two hours, based on multiplying 4 minutes, 25 seconds by twenty seven.

*A rough estimate of 0.36 Gflops/sec for the 20 x 24 size is arrived at by the following approximations. AVRX throughput (0.065 Gflops per second) and running times are assumed the same for the small grid as for the analyzed case. For the rest of the code, throughput is assumed the same, but the running time was reduced by a factor of 26, since DOALLs drop from 26 cycles to one cycle. The result is one twenty-sixth as much useful computation done in 6.08% of the time.

CHAPTER 4
SOFTWARE

4.1 INTRODUCTION

The primary uses of the NASF are expected to be design and modeling applications. These applications can be approached either by experimentation (such as with wind tunnels) or by simulation. Figure 4.1 shows the relationship of these two approaches. The NASF is expected to support the abstraction of the "Real World" with some mathematical system. Mathematical conclusions will be established as a result of the simulation and these conclusions will then be interpreted to determine the desired physical conditions.

The abstraction process represents the development of algorithms to model real-world situations. The NASF should provide tocis and support to assist in this abstraction process. The system considered in this Feasibility Study would provide support for the abstraction process both with simple extensions to the well-known FORTRAN language and with an interactive system which can be used to observe the results of the use of the model.

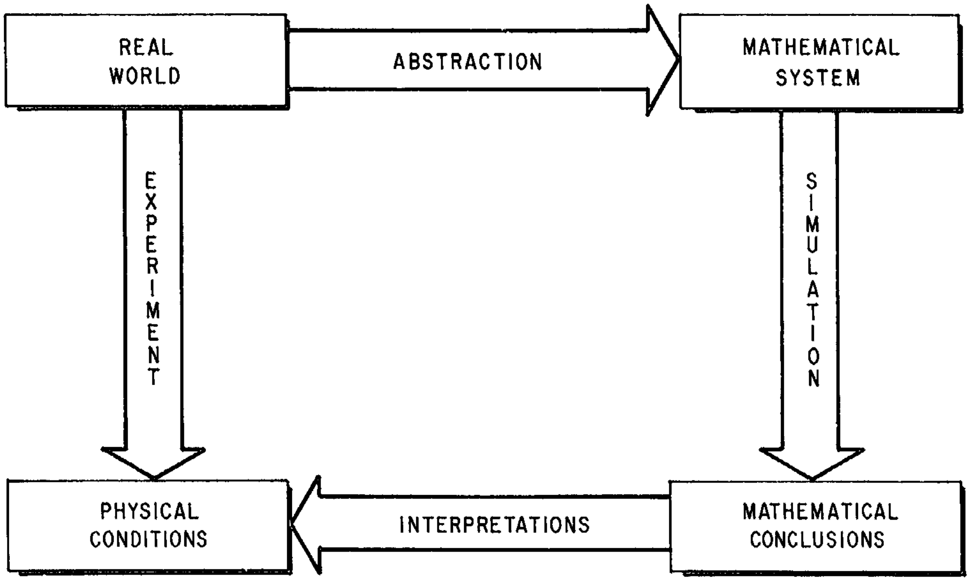


Figure 4.1 Relationship of Simulation and Experi"entation

The simulation process would also be supported with the language extensions. The Support Processor and the File System would be used with the FMP during the simulation process to provide the same careful controls and monitoring needed during an experimentation process. The results of simulations would be observed through use of the various NASF user facilities (printers, graphics terminals, COM, etc.) for interpretation by the users. Where the results of experiments might be available on the facility, comparisons between simulations and experiments would be made.

These processes (abstraction, simulation, and interpretation) require use of most of the components planned for the NASF. The system-level components were already described in Chapter 2. Chapter 5, which follows, discusses the Flow Model Processor (FMP) in detail. This chapter concentrates on the system-level software required to support these processes. The most direct software support of users comes from some means of describing the mathematical system which is the result of the abstraction process and of controlling the simulation process. In the NASF, the language used to define processes on the FMP provides the support required. Other forms of software support are the Master Control Program (the operating system which controls all parts of the NASF), the File System Control Software, Intrinsics, and Test and Diagnostic Support Software.

4.2 FMP FORTRAN

The language considered as a means for supporting the design and modelling applications on the NASF is a dialect of FORTRAN. This dialect is based on ANSI Standard X3.9-1978 [10] and includes a few extensions which are appropriate both to implementation of the models and their simulation.

The description of the FMP FORTRAN presented here is substantially the same as that actually used during the application analysis (see Chapter 3). The language constructs presented are particularly oriented to describing a set (or collection of sets) of discrete processes which may be used to define the desired models. This simple set of constructs seems sufficient to support the applications planned for the NASF.

4.2.1 Language Design Considerations

The design of a language must be concerned not only with the utility of its use for applications, but it must also consider problems of complexity, of implementation on the hardware of interest, and of debugging and verification capabilities.

4.2.1.1 Complexity

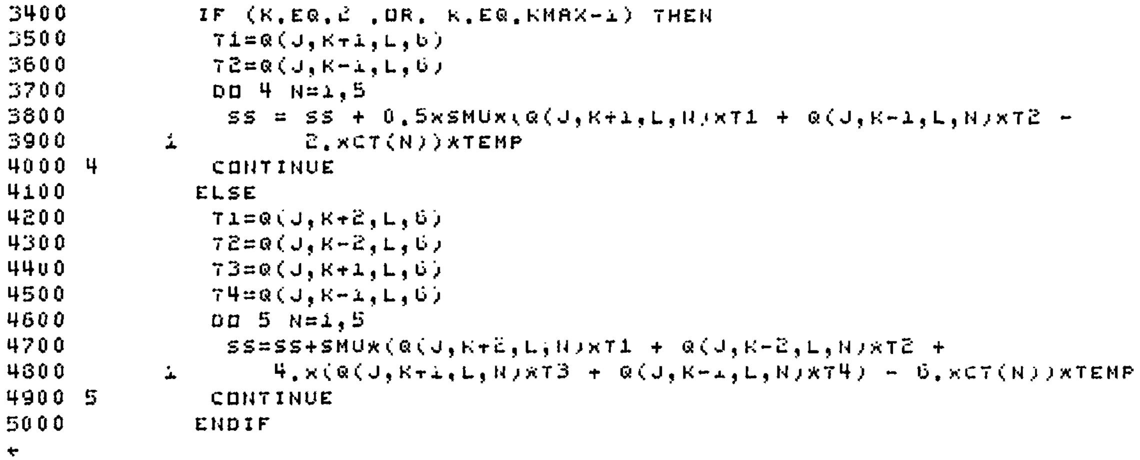
As the capabilites of available hardware expands, the uses of the hardware have expanded to the point where the software has become extremely complex. The development of a new language to support the NASF community therefore must consider the problems of complexity. Since programs are more often read than written, the language source becomes important in two forms of communication, one with other users and the other with the NASF. The most important concern then is to try to achieve a match between the apparent complexity in a program and our human ability to deal with that complexity. The language constructs described in the following sections have been chosen to highlight the essential major ideas of a model while using "standard" FORTRAN to define the details. Some of the complexity of the "standard" FORTRAN will be removed by optional automatic formatting of the source listings. For example, the section of SMOOTH shown in Figure 4.2 shows how indentation can be used to clarify the scope of the various control structures such as DO and IF.

Although the design of a language cannot provide the desired simplicity automatically, the constructs can be chosen to influence programming style in the desired way. Therefore, the constructs chosen in the language extension are few and general in nature. The programs written using these constructs should be easily understood, hopefully even more understandable than the same algorithm expressed in serial constructs. This result is expected because each part of a program can be kept conceptually simple and because the relations between the parts of the program are kept simple. The constructs chosen also make the representation of discrete models more natural. These constructs should allow simplification of the abstraction process without losing the ability to make efficient use of machine organization. In addition, subsequent modifications to the abstractions should be simpler.

4.2.1.2 Abstraction and Modeling

Before considering the proposed language constructs, the abstraction and simulation processes of Figure 4.1 should be discussed in more detail. The problems faced in the practical use of the NASF will be how to abstract the real-world systems of interest and how to control the simulation of such systems so that the results would be a meaningful adjunct to experimental results.

Since a digital system cannot directly model a continuous process, the abstractions must be to some discrete-system representation. The first step in such an abstraction is to identify the structure (and substructure) of the model. For example, the geometries or grids of interest would be defined. Then the information of interest throughout the model would be identified. Such information is usually called "state" information since it describes the current state (or value) at the point of the model with which it is associated. For example, when studying air flow around an object, wind velocity, wind direction, and pressure may be of



```
      IF (K.EQ.2 .OR. K.EQ.KMAX-1) THEN
       T1=Q(J,K+1,L,6)
       T2=Q(J,K-1,L,6)
       DO 4 N=1,5
        SS = SS + 0.5*SMU*(Q(J,K+1,L,N)*T1 + Q(J,K-1,L,N)*T2 -
     1         2.*CT(N))*TEMP
    4  CONTINUE
      ELSE
       T1=Q(J,K+2,L,6)
       T2=Q(J,K-2,L,6)
       T3=Q(J,K+1,L,6)
       T4=Q(J,K-1,L,6)
       DO 5 N=1,5
        SS=SS+SMU*(Q(J,K+2,L,N)*T1 + Q(J,K-2,L,N)*T2 +
     1      4.*(Q(J,K+1,L,N)*T3 + Q(J,K-1,L,N)*T4) - 6.*CT(N))*TEMP
    5  CONTINUE
      ENDIF
```

Figure 4.2 Example of Source Code Formatting

interest at each point of the model. At the same time, some state information may apply to the entire model (such state information is invarient over the model). For example, the Reynold's number of the fluid flowing around an object might be assumed to be the same everywhere. If such an assumption is made, one value can be used to represent the entire model.

Real systems of interest are usually not static. They show some "behavior" over time. Such behavior is observed through the state information. Thus some means must be provided to describe the process by which the state at a particular point in the model changes over time. Conceptually, such a process exists at each point in the model. Note that these processes are concurrent.

The language constructs chosen below provide means of describing both the spatial relationships (geometry and state) and the temporal relationships (processes) in a model. In general, standard FORTRAN constructs are used to define the process of state change at each point in the model while a new construct (called DOALL) is used to identify the natural points of concurrency. Although the normal FORTRAN variable and array mechanisms are available to describe the state of the model, two additional constructs are defined which are intended to make the abstraction of models more straight forward as far as geometry and state variables are concerned and which assist in efficient usage of the storage of the FMP.

4.2.2 Language Constructs

The language called FMP FORTRAN is based on ANSI FORTRAN 77 (X3.9-1978) [10] with extensions and modifications to improve its utility for use for the planned applications and to allow efficient use of the projected hardware. FMP FORTRAN is expected to implement all of the features of ANSI FORTRAN 77 except that CHARACTER type, all usage of CHARACTER type, and Input/Output Statements are as defined in the subset FORTRAN in the ANSI document [10].

The additional constructs described in the following sections have been motivated by the abstraction and simulation functions already described. The three major areas discussed are geometry, state of the model, and process modeling. As with other parts of the system approach discussed in this report, areas for continued improvement certainly exist. However, the language constructs reported here were sufficient as far as the specific application programs considered are concerned.

4.2.2.1 Introductory Example

To introduce the basic concepts of the language extensions, a simple example will be considered first. The sections which follow will explore each of the major areas in more depth.

Figure 4.3 shows the main computation section of TURBDA (from the explicit aerodynamic flow code). Note that there are three nested loops. Also note that the computation inside the loops is independent of the nesting order. The variable CV1 is used each time the inner loop is evaluated without change.

Now consider Figure 4.4 which is a corresponding FMP FORTRAN version of the code in Figure 4.3. Note that the nested DO loops are replaced with a statement called "DOALL" at the start of the loop and with a statement called "ENDDO" at the end of the loops. This version would execute exactly the same as the original version if only one processor is available. However, the DOALL construct gives the specific information that the computation of the inner loop for each combination of I, J, and K values is independent of all other I, J K combinations. In other words, if enough processors were available, all IL*JL*KL instances of the code in the inner loop could be computed concurrently. In this case, there would be IL*JL*KL copies of the inner loop code (one copy per processor). Execution of the DOALL statement would activate all IL*JL*KL instances simultaneously, (one per processor), each with its own set of I, J, K values. After all instances had completed, execution would continue after the ENDDO statement, just as control passes the CONTINUE statement in the original code when all loops are complete.

From an applications' standpoint, the DOALL statement identified a grid over I, J, and K. The arrays EI and RMUL have one element (of "state" information) corresponding to each point of the grid. The variable CV1 is a "global" state variable. The code between the DOALL and ENDDO statements describes the process of changing state variables from the old set of values to a new set of values. Note that the process is logically different along the J=1 and K=1 planes. The evaluation of the code for a specific combination of I,J,K values is called an instance.

The compiler is informed of the usage of variables in this case with the last part of the DOALL and ENDDO statements. The variables and arrays listed after the word USING in the DOALL statement and after the word GIVING in the ENDDO statement identify the state variables (usually in Extended Memory).

Before considering the details of each of the constructs, consider how each of the instances execute and use memory. In the case of IL*JL*KL processors, all processors begin execution, each on its instance. Each processor has a copy of the code and executes out of its own local storage. CV1 and the array EI(I,J,K) would be referenced in the common Extended Memory. The variable TEMP is completely local to an instance. Therefore, each processor would have a storage location for TEMP as used in the instance executing on that processor. The resulting array RMUL(I, J, K) would also be in Extended Memory. Since all IL*JL*KL instances need the value of CV1, space would be allocated in each processor for CV1 and the value would be broadcast to all processors. This approach costs a little storage and would save IL*JL*KL -1 references to the slower Extended Memory.

```
     CV1=1./CV
     DO 1 K=1,KL
     DO 1 J=1,JL
     DO 1 I=1,IL
     TEMP=ABS(EI(I,J,K))*CV1
     IF(K.EQ.1) TEMP=.5*ABS(EI(I,J,1)+EI(I,J,2))*CV1
     IF(J.EQ.1) TEMP=.5*ABS(EI(I,1,K)+EI(I,2,K))*CV1
     RMUL(I,J,K)=2.270E-08*SQRT(TEMP**3)/(TEMP+198.6)
   1 CONTINUE
←
```

Figure 4.3 Section of TURBDA

```
CV1 = 1.0/CV
DDALL I=1,IL;J=JS1,JE2;K=K21,KE2; USING /A12/,/A5/,CV1
    IF(J.NE.1 .AND. K.NE.1) TEMP = ABS(EI(I,J,K))*CV1
    IF (K.EQ.1) TEMP=0.5*ABS(EI(I,J,1)+EI(I,J,2))*CV1
    IF(J.EQ.1)TEMP=0.5*ABS(EI(1,1,K)+EI(I,2,K))*CV1
  RMUL(I,J,K) = 2.270E-08*SQRT(TEMP**3)/(TEMP+198.6)
ENDDO;   GIVING /A6/
←
```

Figure 4.4 FMP FORTRAN Version of Section of TURBDA

4.2.2.2 Geometry

When planning the discrete representation of a real-world system, three major steps are usually involved. First the geometry or general structure of the model is defined together with any substructure expected to be used during definition of the structural and temporal relationships of the model. In general, all of the discrete points or elements of a model will be incorporated into a set. Algorithms used to map this form of the model onto the hardware will be described in Section 4.2.2.2.7.

The examples of the proposed language constructs which follow are based on physical models and corresponding cartesian coordinate systems. These concepts apply as well to transform spaces.

The geometry of the model is defined by first describing a substructure of single dimension. This substructure is then used to describe structures of higher dimension. Since the points of the discrete model are usually identified with an ordered set of integers, the construct used, called DOMAIN, is capable of building ordered sets. For example,

```
DOMAIN /X/ : L=1, MAXX
```

Here the name of the domain is "X". The domain is an ordered set of values as defined with the implied-DO form. If MAXX=5, then /X/ = {1,2,3,4,5} . Two such linear domains can be used to define a two-dimensional domain. For example:

```
DOMAIN /LAT/ : I=1, IMAX
DOMAIN /LON/ : J=1, JMAX
DOMAIN /LAYER/ : /LAT/.X./LON/
```

Here the domain LAYER was defined to be the Cartesian product of the two linear domains LAT and LON. The result is that LAYER consists of all (I,J) pairs.

Two forms for describing geometries of interest will be described below. The first, called DOMAIN, allows the user to define the overall structure or framework of the model. As in standard FORTRAN, this form establishes the maximum structure of interest in the problem, and is used in the mapping to hardware to properly allocate storage and processors. The second form, called REGION, is used to dynamically specify an arbitrary set of elements from a domain.

4.2.2.2.1 DOMAIN Declarations. The domain declaration can take either of two forms; direct specification or construction.

For example:

```
DOMAIN /JK/: J=1, 10; K=1, 15
```

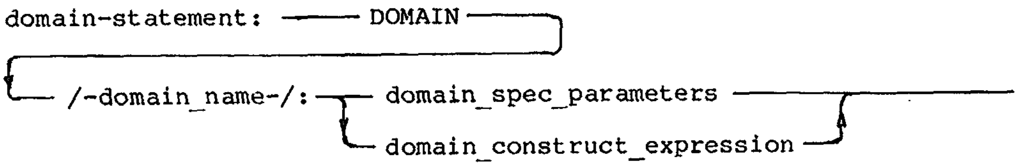
Note that if more than one domain-variable-set is included, the resulting domain is assumed to be the cartesian product of the individual linear sets defined by each domain-variable-set. In the example above, the domain JK consists of all (J, K) pairs where J is from the set 1,2,3, ..10 and K from the set 1,2,3,...15 .

```
DOMAIN /J/:  JJ = 1, 10
DOMAIN /K/:  KK = 1, 15

DOMAIN /JK/:  /J/.X./K/
```

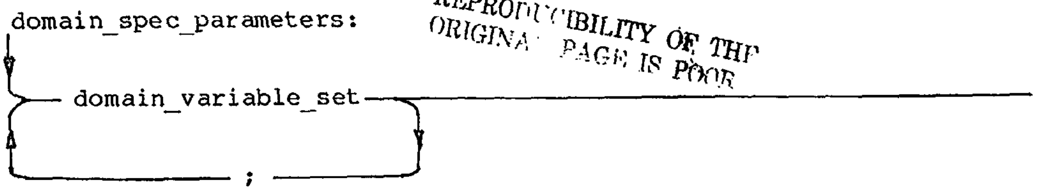
The last domain, JK, is defined as the explicit cartesian product (cross-product) of the sets defined in domains J and K.

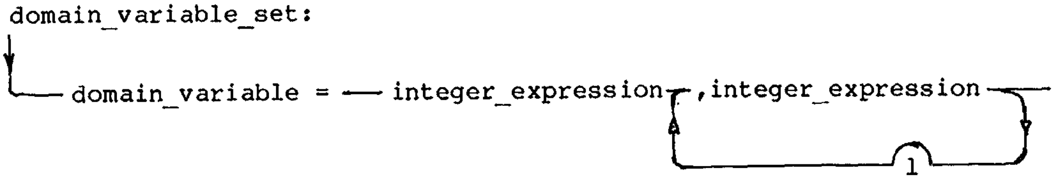
The syntax charts below use the same conventions as in the FORTRAN 77 ANSI standard document [10].

The syntax of the DOMAIN declaration statement is as follows:



A <u>domain-name</u> follows the normal FORTRAN rules for variable naming, and may not be the same as the name of any common block.

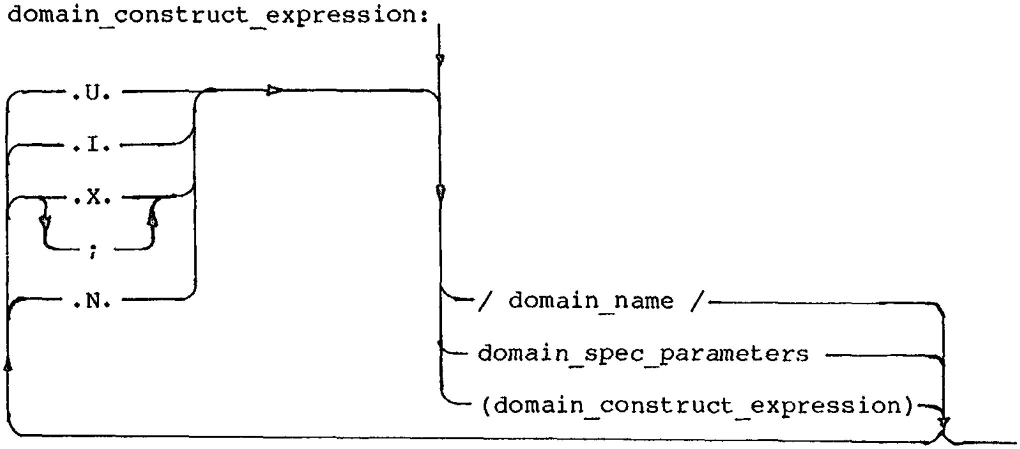




A domain-variable is an integer variable.

The domain-variable is used only for notational convenience during the definition of the domain. The domain-variable does not remain "attached" to the domain during execution. Other means for referencing elements of a domain are described below (instance-identifier-lists and instance-variables).

The integer expressions are evaluated at the point in the program where the domain is declared. The domain-variable-set establishes a sequence of values exactly as for a DO-loop. If the last integer expression is omitted it is taken equal to 1 by default.



A domain-construct-operator is a set operator used to construct a new set from two previously defined sets. The defined domain-construct-operators are as follows:

.U. (union). The resulting domain includes all the elements of both domains. All elements in the resulting domain are unique (duplicates are deleted). The dimensionality of the resulting domain will be that of the operands (which must match).

.I. (intersection). The resulting domain includes only elements that are present in both domains. Dimensionality of the operands must match.

.X. (product). Each element of the resulting domain corresponds to a pair of elements (one from each of the operands). The dimensionality of the resulting domain equals the sum of the dimensionality of the two operand domains.

.N. (relative complement). The resulting domain is the same as that of the first (left-hand) operand with any elements which occur in the second (right-hand) operand removed. The dimensionality of the operands must match.

The precedence order for the domain-construct-operators is .X., .N., .I., .U.. Evaluation is from left to right. Parenthesized expressions are allowed.

The size of a domain may be variable even if the domain is not a dummy argument to a procedure. The domain is defined at run time on entry to the procedure. Each procedure invocation may cause a different-sized domain to be defined.

The variables defining the extents of the domain in the domain-variable-set may be changed during execution of the procedure. Such change does not have any effect on the size or shape of the domain. Once the size and shape are determined on entry they are fixed for the duration of the procedure.

> *Dimensionality* is the number of domain-variables needed to define the domain.

4.2.2.2.2 *Examples.* The following are some examples of legal DOMAIN declarations together with the actual DOMAIN defined.

```
DOMAIN /LONG/ : I = 1,4
       {1, 2, 3, 4}

DOMAIN /LAT/ : J = 1, 5
       {1, 2, 3, 4, 5}

DOMAIN /ODDLAT/ : J = 1, 5, 2
       {1, 3, 5}

DOMAIN /NORTH/ : J = 5,5
       {5}

DOMAIN /SOUTH/ : J = 1, 1
       {1}

DOMAIN /MIDLAT/ : /LAT/ .N. /NORTH/ .N. /SOUTH/
       {2,3,4}

DOMAIN /LAYER/ : /LAT/ .X. /LONG/
       {(1,1) (2,1) (3,1) (4,1) (5,1) (1,2) ... (4,4) (5,4)}

DOMAIN /LEVEL/ : K = 1,2
       {1,2}

DOMAIN /ATMOS/: /LAT/ .X. /LONG/ .X. /LEVEL/
       {(1,1,1) (1,1,2) (1,2,1) (1,2,2) ... (5,4,2)}
```

An alternate form of the above is

```
DOMAIN /ATMOS/ : /LAT/ .X. I=1,4 .X. /LEVEL/
```

4.2.2.2.3 Restrictions The prototype compiler should have a domain dimensionality restriction to four domain-variables. This restriction would help limit the problem of mapping to hardware resources to an acceptable level of complexity. This restriction would likely be lifted with later releases of the compiler.

4.2.2.2.4 Scope. The scope of a domain-name and the corresponding set of points is a program unit. The scope of a domain-variable is the domain-declaration statement. When the same domain must be used in several program units, it must be delcared within each of them (like a named common block).

4.2.2.2.5 Required Order. The position of a domain-declaration statement within a program is the same as "other Specification Statements" (see Figure 4.5).

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY</td><td rowspan="2">PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Figure 4.5 Required Order of Statements and Comment Lines in a Program Unit

4.2.2.2.6 Application and Usage. The DOMAIN specification will be used to define all those discrete points of the model at which state information and/or processing will exist. Each discrete point of the structure or substructure of the model will be represented by an element in some domain.

4.2.2.2.7 Mapping. The geometry as specified must be mapped onto the available hardware as a mapping under which the actual modeling and simulation will run. Many possible mappings exist with many tradeoffs to consider.

A simple static mapping was proposed and used during the application analysis. Such a static mapping is easiest to implement, and results in the least compiler complexity. With such mapping the least run-time overhead is devoted to mapping and concomitant data rearrangements. With this mapping, the linear representation of a domain is mapped to its corresponding processor number modulo the maximum number active processors. The linear representation of a domain is the same as the storage order of an array with the same subscript values and subscript variable order as the domain-variables. For example for an 8 processor system, domain LAYER would be allocated as shown in Figure 4.6. The compiler code should be sufficiently modular that other mappings can be easily evaluated.

The static mapping was used during the application analysis summarized in Chapter 3. The application analysis results show that such a static mapping will support the applications studied. Thus, the FMP is feasible to implement, even without possibly more elegant mapping techniques.

DOMAIN /LAYER/ : J=1,5 .X. I=1,4

PROCESSOR

0	1	2	3	4	5	6	7
(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(1,2)	(2,2)	(3,2)
(4,2)	(5,2)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(1,4)
(2,4)	(3,4)	(4,4)	(5,4)				

Figure 4.6 Modulo Mapping of Elements of a DOMAIN to Processors

Many other possible mappings exist. Another simple static mapping is where the first "n" elements would be assigned to the first processor, the next "n" elements to the next, and so on. Here "n" is defined to be the next integer equal to or larger than the total number of elements divided by the number of processors. In the above example, domain elements (1,1), (2,1), (3,1) would be assigned to processor 0; (4,1), (5,1), (1,2) would be assigned to processor 1, etc.

Another consideration could be the locality of reference. In this case elements could be mapped so that the processes associated with all elements assigned to a processor tend to reference data already physically in that processor thus reducing traffic to the extended memory. Dynamic allocation strategies could also be considered. However, dynamic allocation must balance the benefits of a possible more uniform use of all processors with the likelihood of increased movement of data to and from the common memory. (in a static mapping, variables which are referenced only by the instances within a processor could be assigned storage space within that processor.) Further study is needed to determine the most cost-effective strategy.

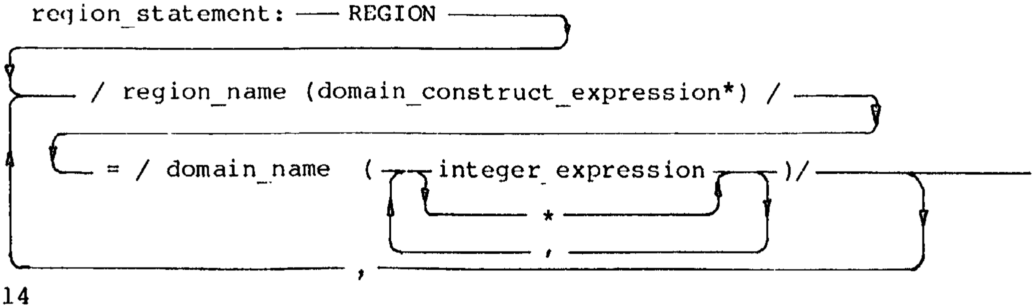
4.2.2.2.8 REGION Statement: Facilities to identify a REGION of active interest within specified DOMAINs are provided. These REGIONs do not constitute a separate structure. Essentially, a REGION is a virtual domain with dynamically selected elements of the original DOMAIN. The elements may be sparse or dense, rectangular or skewed sections of domains. The REGION declaration may be used to explicitly create a virtual domain with dimensionality greater than its original domain or to define which portion of a DOMAIN is to be "processed". The specification of a REGION may be dynamic. The general form of the REGION declaration is:

For example

```
      REGION/JKPART (J=1,5; K=1,9)/= /JK(J+5, K+2)/
```

The values of J and K specified for the region named JKPART are substituted in the expressions associated with domain JK to determined the correspondences. Specifically:

```
     JKPART (1,1) is "equivalent" to JK (6,3)
     JKPART (2,1) is "equivalent" to JK (7,3)
     JKPART (2,9) is "equivalent" to JK (7,11)
```


region_statement: — REGION
/ region_name (domain_construct_expression*) /
= / domain_name (
integer expression
*
,
)/
,

domain_construct_expression* is the same as the construct defined earlier except the / domain_name / part may not be used. The domain_variable names are not variables to which values are assigned. Rather they are dummy names used to define the mapping which identifies that part of the domain which is the region of interest.

Each of the integer_expressions may be a linear combination of the domain varibles used as part of the REGION declaration. References to intrinsic functions will be truncated to integer if necessary.

If the REGION is to choose a one-to-one mapping to DOMAIN elements in the same order as in the DOMAIN, the "*" identifier may be used instead of an integer expression for that domain dimension.

4.2.2.2.9 <u>REGION Declaration Examples.</u> Assume that the following DOMAIN declarations exist:

```
DOMAIN /LAT/ : I = 1, 20
DOMAIN /LON/ : J = 1, 30
DOMAIN /LAYER/ : /LAT/ .X. /LON/
```

The following regions correspond to the drawing in Figure 4.7.

```
REGION/LAYER1 (I=1,5;J=1,10)/=/LAYER (I,J+10)/
REGION/LAYER2 (I=1,10;J=1,10)/=/LAYER(I+5,J)/
REGION/LAYER3 (I=1,10;J=1,10)/=/LAYER(MOD(I+15,IMAX),J+20)/

Note in LAYER3 that LAYER3 (1,1) = LAYER (16,1)
                    LAYER3 (5,1) = LAYER (20,1)
                    LAYER3 (6,1) = LAYER (1,1)
                    LAYER3 (10,1) = LAYER (5,1)
```

4.2.2.3 Model State

After defining the geometry or structure of a model, the state of the model at each point of interest in the defined structures and substructures needs to be described. The state can be described through the use of the various variable types and declarations available in FORTRAN. Since with the applications of interest, the state variables are the same at all points throughout the structure, a new construct, called INALL, was defined to help simplify the description of the state throughout a structure. For example, the declaration:

```
REAL INALL /LAYER/ WNDVEL, WNDDIR, T, P
```

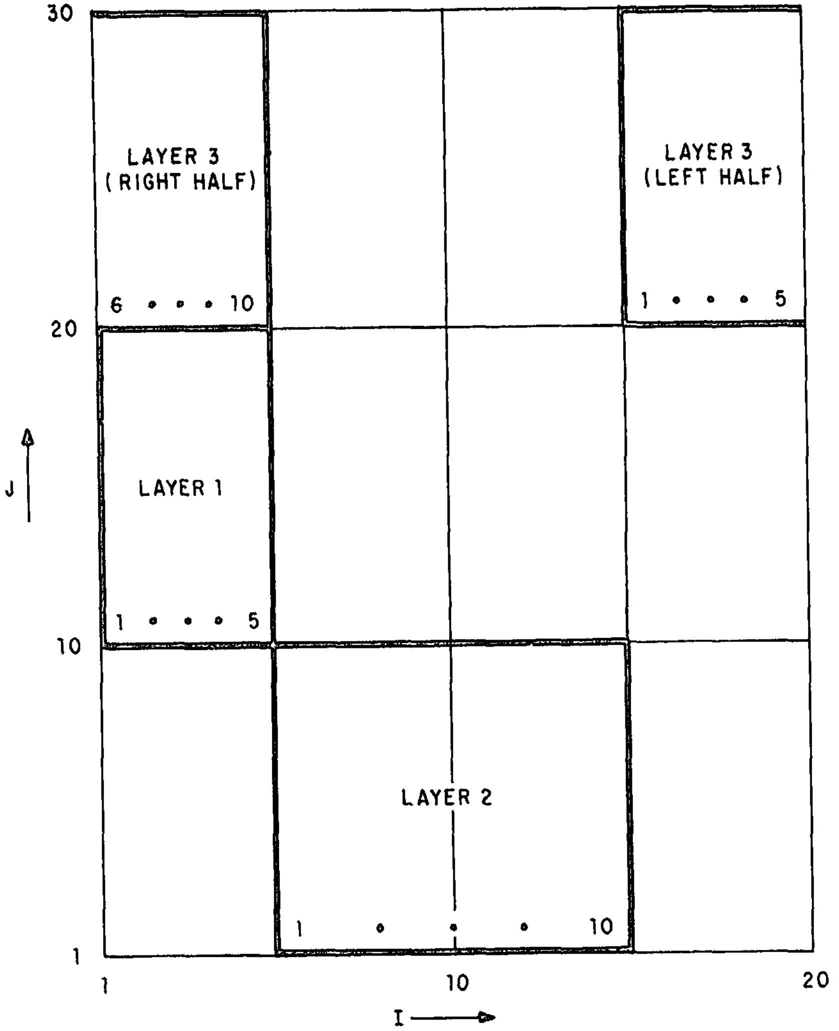


Figure 4.7 Example Regions Selected from Domain LAYER

defines the variables called WNDVEL, WNDDIR, T, and P. Unlike standard FORTRAN, this declaration specifies that each variable occurs "INALL" of the discrete points defined by the domain called ATMOS. In other words, a different variable WNDVEL exists at each point or element of the domain LAYER. (Recall that LAYER was defined in the example above as

```
DOMAIN /LAYER/: I=1,20;J=1,30
```

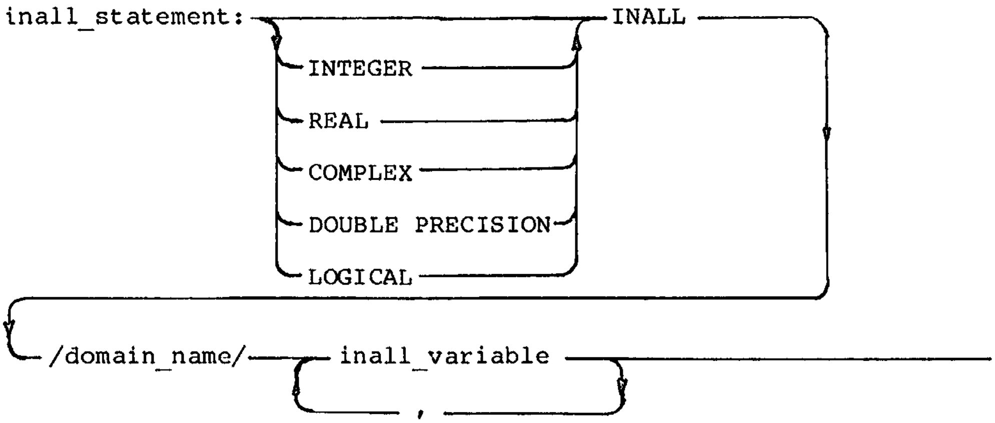
Variables defined with the INALL statement can be used in FORTRAN the same way as dimensioned variables as described later. In such a use, the subscripts identify the point (element) of interest in the structure.

The result of the INALL statement is a set of wind velocity, wind direction, temperature and pressure variables at each point of the domain. The storage reserved would be the same amount as a dimension statement of the form

```
DIMENSION WNDVEL(20,30), WNDDIR(20,30), T(20,30), P(20, 30)
```

However, unlike variables declared with standard dimension statements, the "inall-variables" can be considered to be simple, unsubscripted variables when defining the process to be simulated at each point in the model, as described later. When the names in the INALL declaration have dimensionality, the implied subscript positions of the domain variables precede the subscript positions whose dimensionality is explict.

4.2.2.3.1 <u>INALL</u> <u>Declarations.</u> The INALL declaration would take the form:



If a type is declared, it applies to each of the inall_variables listed. The inall_variable can be a variable name, an array name or array declarator. If no type is specified, each inall_variable on the list will be implicit type or as specified in a separate type statement.

The INALL declaration serves the dual function of declaring the type of the variables on the list and of declaring storage requirements. The INALL declaration semantically indicates that each element of the domain defined has associated with it variables identified in the list. Thus, if there are 10 variables on the list and if the domain declared has 3 dimensions with extents of 3, 4, and 5, then the storage space reserved would be 3*4*5*10 or 600 storage units.

4.2.2.3.2 Scope. The scope of the INALL-variable name is a program unit. If an INALL variable is in a named common block, all names in that named common block must match in the several program units where they occur.

4.2.2.3.3 Application and Usage. Each element of a domain will include the set of declared inall-variables. That is, a unique set of inall-variables will exist for each point in the domain. The language constructs used for referencing these variables are described in Section 4.2.2.6.

4.2.2.3.4 Mapping. The physical storage allocated for the inall-variable set corresponding to each point of the specified domain would be allocated to the storage of a physical processor in the same manner that the domains are mapped (see Section 4.2.2.2.7). As a result each processor will contain as many inall-variable sets as the number of elements assigned to that processor for each domain. Figure 4.8 is an example of this allocation.

The purpose of the DOMAIN and INALL statements is to define an application-oriented data structure. The structure is hierarchical. The major divisions (the grid) are defined by the DOMAIN statement. The subdivisions are defined by the INALL statement and consist of the state variables and arrays defined to exist at each grid point. The data structure definition is independent of how the structure is mapped onto the storage of the FMP.

4.2.2.4 Process Modeling

Once the structure and state of the model have been defined, some means of describing the process to be modelled must be provided. This description is done in two stages. First, the general flow of the sequence of events which occur during the process would be described. This general flow description allows the dependence of subprocesses over time to be defined. Second, the detailed relationships that exist within the defined structures and substructures are defined. These relationships exist for each of the events. The dynamic "behavior" of the discrete system is defined by the combination of the general flow and the detailed relationships.

```
DOMAIN /LAYER/ : J=1,5 .X. I=1,4
REAL INALL /LAYER/ V,D,P
```

PROCESSOR

0	1	2	3	4	5	6	7
V,D,P(1,1)	V,D,P(2,1)	V,D,P(3,1)	V,D,P(4,1)	V,D,P(5,1)	V,D,P(1,2)	V,D,P(2,2)	V,D,P(3,2)
V,D,P(4,2)	V,D,P(5,2)	V,D,P(1,3)	V,D,P(2,3)	V,D,P(3,3)	V.D,P(4,3)	V,D,P(5,3)	V,D,P(1,4)
V,D,P(2,4)	V,D,P(3,4)	V,D,P(4,4)	V,D,P(5,4)				

Figure 4.8 Allocation of Inall-Variable Sets to Processors

Standard FORTRAN is a language with inherent sequential dependencies. That is, execution is constrained to be one statement at a time in a well-defined order. The general flow does have such a sequential dependency. Standard FORTRAN control constructs can, therefore, be used to describe the sequential dependency. However, when modeling real processes, there are also concurrent actions which must be considered. If the language provides a means for describing the concurrency naturally inherent in the processes being modeled, the the mapping of the abstracted process to the hardware will be more straight forward and the user should find it easier to define the abstraction. This concurrency in the general flow can be described with the construct called DOALL.

4.2.2.5 DOALL Construct

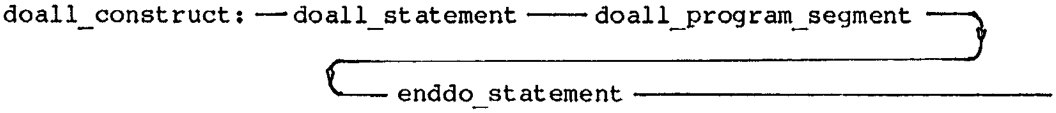
The basic form of the DOALL construct was shown in an earlier example (section 4.2.2.1). Recall that a segment of standard FORTRAN code (which describes the computation required to evaluate the process of getting new values from old values) is started with a DOALL statement and ended with an ENDDO statement. Figure 4.9 is another example. This is a section of the FMP FORTRAN version of SMOOTH (see Appendix A for a discussion of the application code). Note the region THREED. This region is three-dimensional. Variables SS, T1, T2, T3, T4 and a vector CT(5) are defined at each point in the domain MODEL. The statement marked ① is the control statement that indicates that all statements from that point to the statement marked ② are considered to be replicated (one set of statements to each point in the domain), that each set of statements (called an instance) is independent from all other sets and that all sets of statements could execute concurrently (given sufficient resources). The code marked ③ tests to determine whether the particular domain point is in either the J=2 or J=JMAX-1 plane. If so, then the next few statements are executed. If not, the statements following the ELSE are executed. Note that these sequencing decisions within an instance are based on data unique to the instance and do not depend on any other instances. Also note that the statement following the one marked ② is not permitted to begin until all instances have completed execution.

It is interesting to note that if there are fewer processors than instances to be evaluated, then the work would be spread out across the processors. Each processor would evaluate more than one instance. Since only one set of code would be required per processor, multiple instances are executed simply by cycling through the code for that segment. The term <u>cycles</u> is used to indicate how many instances a processor has evaluated of the currently executing doall construct. The allocation of specific instances to processors would be static and would use the scheme previously discussed with regard to assignment of elements of a domain to processors (which is the same problem).

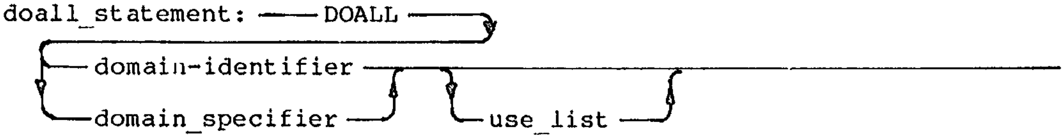
```
     100           SUBROUTINE SMOOTH
     200            COMMON/BASE/NMAX,JMAX,KMAX,LMAX,DT,GAMMA,GAMI,FSMACH,
     300          1    DX1,DY1,DZ1,FV(5),FD(5),HD,ALF,GD,OMEGA,HDX,HDY,HDZ,RM,
     400          2    CNBR,PI,ITR,NP,INT1,INT2,INT3
     410           DOMAIN /MODEL/; J=1,100; K=1,50; L=1,200
     420           REGION /THREED((J=2,JMAX-1),(K=2,KMAX-1),(L=2,LMAX-1))/
     430          *     = /MODEL(J,K,L)/
     440           INALL /MODEL/ Q(6),S(5),SS,CT(5),T1,T2,T3,T4
     700 C    4TH ORDER SMOOTHING, 2D ORDER AT THE BOUNDARIES
(1)→1000           DOALL /THREED(J,K,L)/ ;  USING Q, S, SMU
    1200           TEMP = 1./Q(J,K,L,6)
    1300           DO 1 N=1,5
    1400           CT(N)= Q(J,K,L,N)*Q(J,K,L,6)
    1500 1         CONTINUE
(3)→1600           IF (J.EQ.2 .OR. J.EQ.JMAX-1) THEN
    1700             T1 = Q(J+1,K,L,6)
    1800             T2 = Q(J-1,K,L,6)
    1900             DO 2 N=1,5
    2000             SS = S(J,K,L,N) + 0.5*SMU*(Q(J+1,K,L,N)*T1)- 2.*CT(N) +
    2100          1           Q(J-1,K,L,N)*T2)*TEMP
    2200 2           CONTINUE
    2300            ELSE
    2400             DO 3 N=1,5
    2500             T1=Q(J+2,K,L,6)
    2600             T2=Q(J-2,K,L,6)
    2700             T3=Q(J+1,K,L,6)
    2800             T4=Q(J-1,K,L,6)
    2900             SS = S(J,K,L,N) + SMU*(Q(J+2,K,L,N)*T1 + Q(J-2,K,L,N)*T2 +
    3000          1           4.*(Q(J+1,K,L,N)*T3 + Q(J-1,K,L,N)*T4) - 6.*CT(N))*TEMP
    3100 3           CONTINUE
    3200            ENDIF
(2)→3300           ENDDO/THREED/
       *
                     •
                     •
                     •
                     •
```

Figure 4.9 Section of FMP FORTRAN Version of SMOOTH

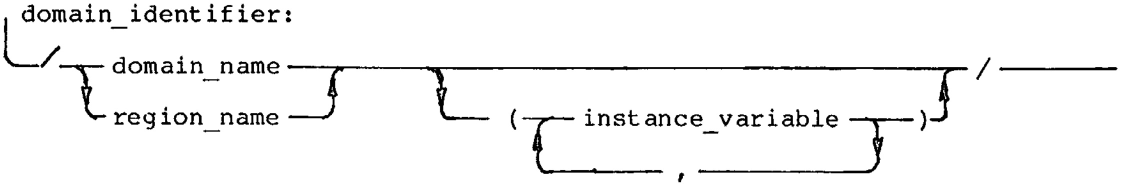
4.2.2.5.1 Construct Definition. The specific form of the DOALL construct is:



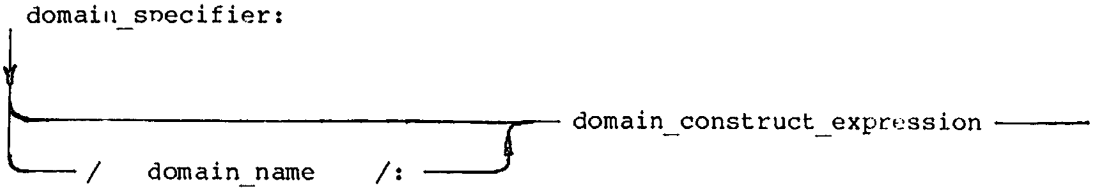
The doall_statement is defined as follows:



The main purpose of the DOALL construct is to identify those processes (in the doall program_segment which can be concurrently evaluated. The doall_statement identifies the points (grid values) at which the doall_program_segment would be evaluated. The points may be previously defined (as a domain or region) or may be defined as part of the doall_statement.



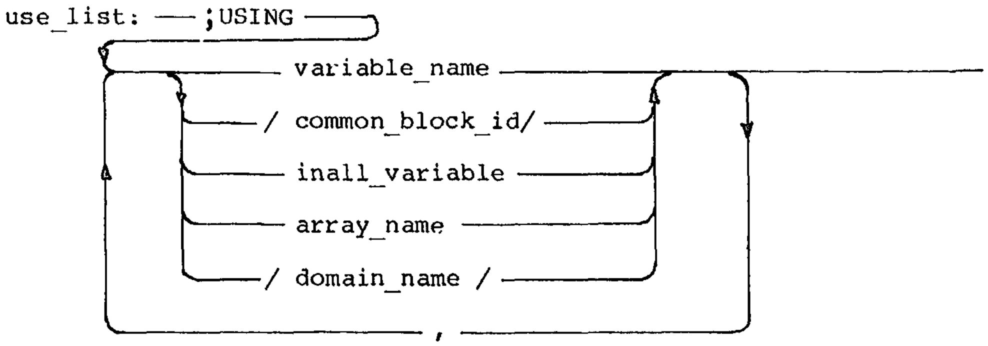
Each instance-variable is an integer variable and is unique from all others on the list. Each instance-variable represents one of the dimensions of the declared domain. Instance-variables are not required to be the same as the domain-variables used when the domain was originally declared. For each instance the set of values assigned to each of the instance-variables at the start of evaluation of each instance is the set of values used to uniquely identify that instance. The scope of the instance-variable is the doall-program-segment. The instance_variables are allocated storage in the local processor memory.



The domain-name could be included on the ENDDO statement which terminates the construct (to assist in program readability). When a domain is declared as part of the DOALL statement itself, its scope is local to the DOALL-program-block itself. The domain-variables used in the domain-specification-parameters become instance-variables as described above in addition to being used to determine the extent of each of the dimensions of the domain.

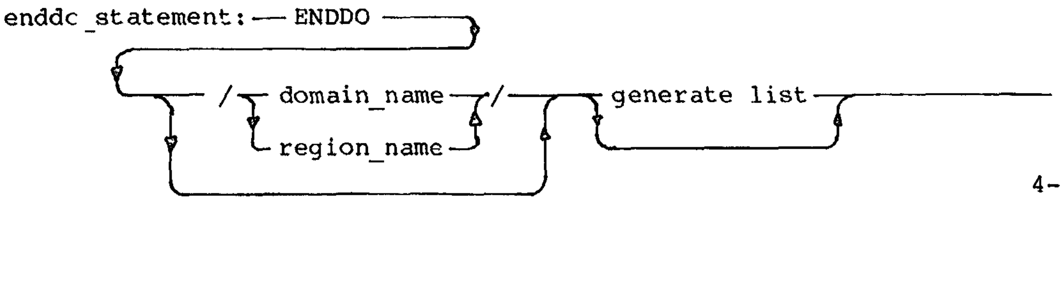
The doall-program-segment is a set of FMP FORTRAN statements which describe the process to be evaluated at each point specified. In terms of the model, the process defined in the doall-program-segment is conceptually evaluated at all points simultaneously. In addition, the process at any one point does not have access to the values computed by the same process at any other point in the model. Figure 4.10 shows this independent, concurrent structure in a "flow-chart" form. The evaluation of a doall-program-segment at a given point is called an instance. All instances of a doall-program-segment will complete execution before executing the next statement in sequence after the ENDDO. Although conceptually all instances execute concurrently, the actual order of execution is dependent on the processing resources available. The only implementation requirements are that all instances must complete execution prior to continuing with the next statement after the ENDDO.

The use list is to specifically identify which variables or arrays are used within instances of the doall-program-segment. The specific form is:



The purpose of this USING clause is explained in more detail later (section 4.2.2.6).

The last statement of a doall_construct is the enddo_statement.



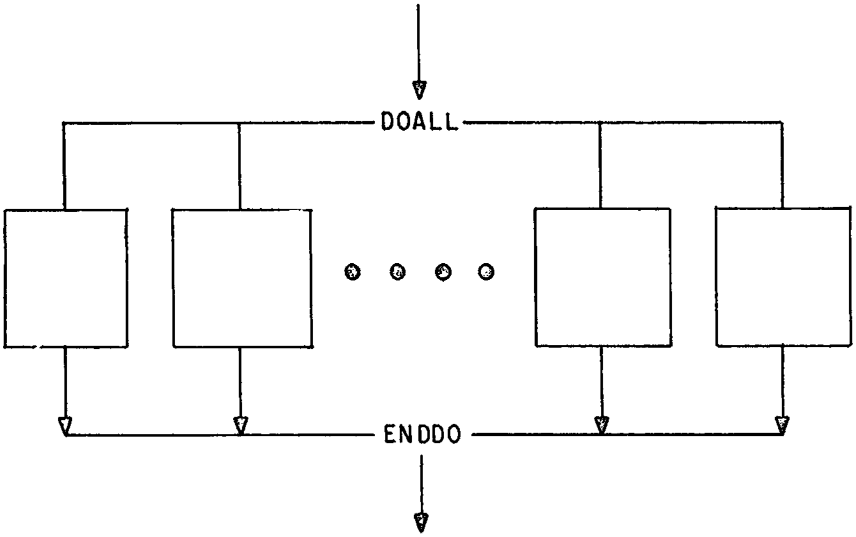
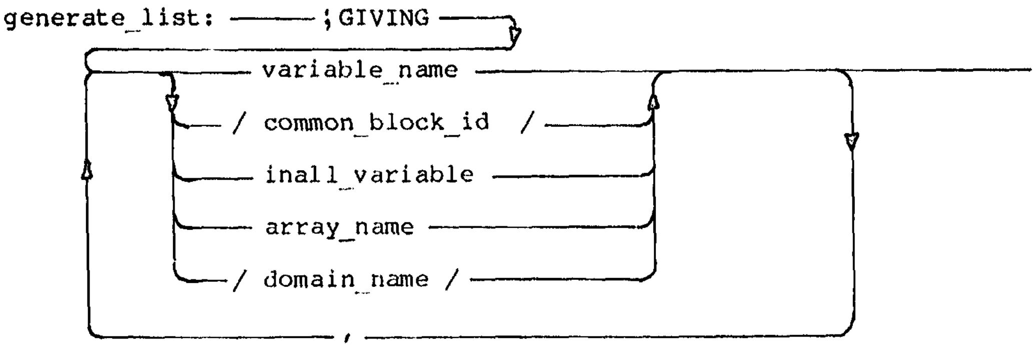


Figure 4.10 DOALL Construct Control Flow



The generate list specifically identifies those variables or arrays produced for reference upon completion of the doall_ construct.

4.2.2.5.2 Serial FORTRAN Equivalent Form. Any DOALL construct can be simply represented in standard FORTRAN with nested DO loops. The depth of nesting matches the dimensionality of the domain over which the DOALL is defined. In fact, the domain-variable-sets (See Section 4.2.2.2.1) used to define the domain become the control part of the DO statements.

4.2.2.5.3 Nested DOALLS. The doall-program-segment may itself include a DOALL construct. Since the application programs of interest did not require this construct, no evaluation of possible run-time efficiency or inefficiency has yet been made. Since dynamic resource allocation has not been proposed yet, nested DOALL's would be statically decomposed to serial form.

4.2.2.5.4 Mapping. The mapping of the doall-program segments is the same as that described for an element of a domain. However, since each instance executes the same doall-program-segment, only one copy of a program-segment need be kept by each processor.

4.2.2.5.5 Restrictions. No instance of a doall-program-segment may reference the results of the current computation of any other instance in the same doall-program-segment. Each doall-program-segment has access to all of the values of the model at the start of that program-segment. The entire DOALL construct must be treated as a whole in order to control the implementation and use of the construct. For example, consider a hypothetical system where such a restriction did not exist and suppose that the computations performed in one instance did use the value of variables in another instance of the current doall-program segment. Under these conditions, successive runs of the program are likely to get different results since the time order of execution of the two program segments is not necessarily the same from one run to the next. As a result, the variable values fetched from the second instance are either old values or are new values, but with no control or "knowledge" of the encompassing program that such a variation occurred. Programs would be very difficult to debug in such a hypothetical system.

Since no referencing between instances of the currently executing doall-program-segment is allowed, the results of evaluation are completely independent of the order of execution.

Because of the concurrency expressed in the DOALL construct, the arbitrary transfers of control which are allowed in standard FORTRAN must be restricted in FMP FORTRAN. No transfers into a DOALL construct may be made. Within a DOALL construct, normal FORTRAN control constructs (IF, GO TO, ...) are allowed, but control must remain within the DOALL construct. All instances exit via the ENDDO. No transfers out of an instance are allowed.

4.2.2.6 Variable Referencing

The standard FORTRAN referencing conventions apply. One extension has been defined to simplify the description of the models of interest. This extension supports the concept of "centered-subscripts".

4.2.2.6.1 Referencing Within a DOALL-program-segment. The DOALL construct described above is used to define the time sequencing of a modeling process. At each discrete time step, some sort of interaction between the various parts of a model takes place. In particular, the modeling task may involve the use of general state variables, of state variables unique to each element of a domain and intermediate variables used during the evaluation of a process. General state variables are used to describe an overall process or structure and may be referenced within each instance. The state variables defined at each point of a domain may be referenced by processes defined at other points. However, the intermediate values used during the evaluation of a process would be of concern only to each instance and not to any others. In order to have an orderly flow of data and allocation of storage in the system, some language constructs have been proposed which relate to the above dependencies.

The general state variables (those variables which apply to all points of a domain or region) will be called GLOBAL variables. Those state variables which have been defined for each of the points of a domain will be called STRUCTURE variables. Any variables with values generated and used only within an instance will be called LOCAL variables. Note that GLOBAL variables would not be defined using INALL statements.

The differentiation of these different "classes of use" is important in a multiprocessor such as the FMP because of the added complexities of storage allocation and storage management. The additional constructs already defined provide application-oriented ways to define variable usage to the compiler.

The USING and GIVING clauses of the doall and enddo statements are a natural way to explicitly define the data-dependency of the system modeled at the process-level. The compiler, in turn, will use these statements, together with analysis of the source code, to produce code to initiate early requests of data transfers from EM to the processors and back, thus further improving throughput by allowing more overlap of execution with fetching data from the Extended Memory.

Any variable used within a doall-program-segment but not declared in the USING clause must be self-defining within each instance prior to its use. If a variable is not included in a USING or GIVING clause the implication is that the variable is only needed temporarily during the evaluation of the process. Thus, in order to consider storage requirements, variables not declared either USING or GIVING need exist only for each "active" instance rather than for each instance. (An "active" instance is an instance which is being executed by a processor resource.)

If a variable is included on a USING or GIVING clause, but is not included in an INALL declaration, the implication is that all instances of the doall-program-segment will reference the same variable (GLOBAL variable). When this condition occurs, the compiler would allocate space for such a variable in each processor and generate code which would cause the value of such a variable to be broadcast to all processors rather than requiring each instance to separately request access to that variable. Variables of this sort were previously called "CONTROL" [1,2].

If a variable is included on both a USING or GIVING clause and on an INALL declaration, the implication is that each instance will require its corresponding INALL variable (recall tnat INALL creates a variable for each point in the domain). Special subscript forms defined in the next section can be used to reference INALL variables in other instances.

Figure 4.11 summarizes the variaole use interpretations based on the statements defined.

The importance of the independence of the instances of a doall_program_segment has already been pointed out. All GLOBAL and STRUCTURE variables as well as all instance identifiers (used to identify the set of indices which define the point in the domain) can be considered to be preinitialized to their value upon entry to the DOALL construct. At that point, at least conceptually, the evaluation rules within a particular instance are interpreted in classical FORTRAN fashion except that the values assigned to GLOBAL or STRUCTURE variables can be referenced only by the instance which did the assignment. All other instances would continue to reference the original values. Similarly, a set of instance identifier variables would exist for each instance. The initial values in the set for a particular instance would identify the instance. Any changes made to the values in one set could not be observed within any other instance. The FMP (hardware and software) will enforce these referencing procedures.

		in any USING or GIVING clause	
		YES	NO
declared INALL on DOMAIN	YES	STRUCTURE	LOCAL
	NO	GLOBAL	LOCAL

Figure 4.11 Variable Use Interpretation

4.2.2.6.2 Centered-Subscripts. The intent of the new constructs described (DOMAIN, INALL, DOALL,...) has been to allow the description of a model and the modeling process as it reflects the process and state at each discrete point of the structures of interest. References by the doall-program-segment to variables in the same element of the DOMAIN as the instance need only be by the simple name. For example,

```
     REAL INALL /ATMOS/ T,WNDVEL, AB(7)
```

is a statement declaring variables T and WNDVEL and a vector AB at each element of the domain ATMOS. In the following program segment, the process defined computes new values which are a function only of old values in the same instance:

```
       .
       .
       .
     DOALL /ATMOS / USING WNDVEL, AB
     T = (AB(1) + AB(2))/2
     WNDVEL = (WNDVEL + AB(3) + AB(5))/2*AB(4)
     ENDDO /ATMOS/GIVING T, WNDVEL
       .
       .
       .
```

Many models have dependencies between elements of the structure. When describing processes of this type, a natural approach is to describe the process centered at a particular element and consider the rest of structure with respect to the centered element. When referencing INALL variables in other instances, a suscript is used in a manner similar to normal array and vector referencing in standard FORTRAN.

Another example might be:

```
    .
    .
    .
    DOALL /ATMOS (I,J,K)/ USING T
    T = T(I,J 1)
    ENDDO /ATMOS/ GIVING T
    .
    .
    .
```

Here the new value of T at each element of the structure is made equal to the original value of T on the lower plane of the structure. (i.e. All elements of a column of /ATMOS/ have the same value of T as the value of T in the first element of the column.) Note that I and J are constants throughout the doall-program-segment since they are the instance-variables. The "*" may be used to indicate the value of the instance-variable corresponding to the element of the domain. For example, another way of writing the preceding example is:

```
    .
    .
    .
    DOALL /ATMOS / USING T
    T = T(*,*, 1)
    ENDDO/ATMOS/GIVING T
    .
    .
    .
```

When variables in adjacent elements of a domain are to be referenced, subscript expressions may be used. For example:

```
    .
    .
    .
    REGION/CENTRAL (L=1,IMAX-2;M=1,JMAX-2;N=1,KMAX-2)/
         =/ATMOS(I+1,J+1,K+1) /
    DOALL/CENTRAL(I,J,K)/USING T
    T = (T + T(I+1,*,*) + T(I-1,*,*) + T(*,J+1,*) + T(*,J-1,*)
   1     + T(*,*,K+1) + T(*,*,K-1))/7
    ENDDO/CENTRAL/GIVING T
    .
    .
    .
```

In this example a REGION was declared that excluded the outer boundary of ATMOS. The DOALL computed new values of T based on all immediately adjacent values. Note that variables declared INALL over a DOMAIN are also accessible to any REGION of the DOMAIN just as if they had been declared INALL on the REGION. Also note that values of T at adjacent elements of the DOMAIN are used to compute new values of T at each element of the REGION. All computation is based on the values of T throughout the DOMAIN upon entry to the DOALL construct.

As a last example, note that a doall-program-segment only treats values of INALL variables as having initial value upon entry if the GIVING or USING clause specified those variables. During execution of a program-segment, the variables may locally be assigned other values. Only the centered-variables are saved under the GIVING clause.

```
REAL INALL/ATMOS/ T, WNDVEL
DOALL/CENTRAL(I,J,K)/USING T
TOLD = T
T = (T + T (I+1,*,*) + T(I-1,*,*))/3
XY(1) = T
XY(2) = (TOLD + T(*,J+1,*) + T(*,J-1,*))/3
XY(3) = (TOLD + T (*,*,K+1) + T(*,*,K-1))/3
T = (XY(1) + XY(2) + XY(3))/3
ENDDO /CENTRAL/ GIVING T
```

In this example only T(*,*,*) is saved upon completion of all instances. The array XY and the variable TOLD are LOCAL variables. These variables are used only by the active instance. In order to conserve storage, the same processor memory locations used for LOCAL variables during execution of an instance in a processor can be used for the LOCAL variables of another instance when more than one instance of a doall_program_segment are evaluated in any given processor. Note that the original value of T(*,*,*) had to be saved since the second statement changed the value (as far as the particular instances was concerned). In this way execution of a doall-segment is the same as that of any FORTRAN segment with the INALL variables specified in GIVING clauses initialized as if with a DATA statement upon each entry.

4.2.2.6.3 Unreferenced Variables. In some cases, a variable identified within a separately compiled sgement, but never be used within that segment. This happens, for example, if the main program has a named common area that is used in a number of subroutines, and the area must exist in the main program for the purpose of holding data created by one of the subroutines and used by the other. In this case, the compiler will not have access to the USING and GIVING declarations, because of the separate compilation. Until a better way of handling this situation is defined, the declaration of these named common areas will be expanded by prefixing them with an indication of how they will be used, when they are used.

STRUCTURE declares that the variables and arrays within the given common block will be used as if they had been included in INALL statements and USING or GIVING clauses.

GLOBAL declares that the variables and arrays in the given common block will be used as if they had been included in USING or GIVING clauses.

All variables and arrays in a given named common must be used in the same way (i.e. as STRUCTURE or GLOBAL).

4.2.2.7 Storage Allocation

The Flow Model Processor has two major areas of storage to be concerned with during execution, the main memories of the processors and the extended memory. The primary use of extended memory is for the STRUCTURE data (the "old" state values). Processor memory is allocated to program, and to data storage space. The data storage space is further divided into temporary areas used only while an instance is active (the LOCAL variables) and into areas which are allocated to each instance resident in the processor. The data areas allocated to instances hold the NEW values as well as copies of OLD values. Note that although many instances of a process may be assigned to a particular processor, only the data areas reflect that assignment. Only one copy of the program-segment would exist.

The GLOBAL variables normally have storage space allocated both in the processor memories and in the EM. This allocation is a space-time tradeoff. If only the original copy existed in the EM, then each instance would have to access it separately with potential conflicts (when more than one processor try to access the same EM location simultaneously, only one is granted access; any others wait). If the value is broadcast to all processors simultaneously (say at the start of a doall), then any references would be to the local copy already resident in each processor.

4.2.2.8 Independent Compilation

Program units, as with any conventional FORTRAN, may be separately compiled. Note that there may need to be a distinction between two classes of subroutines. One class would be those called within a doall_program_segment. These subroutines would be completely independent of any coordinator code and would have any embedded DOALL constructs implemented as nested DO loops. The other class of subroutine would be those called outside a doall_program_segment. If a subroutine of this class did have an embedded DOALL construct, both coordinator and processor code would have to be generated in order to take advantage of the available processors.

One solution to the above situation is to somehow identify one class of subroutine from the other. This could easily be done with a simple construct added to the SUBROUTINE statement itself. For example

```
SUBROUTINE BTRI  DOMAIN /J=1,JM; I=1,IM/
```

This would indicate that BTRI would be called within a two-dimensional doall and that IM*JM copies of the subroutine should be available to the instances of the doalls.

Other solutions exist. They include independent compilation for checking purposes but full compilation to generate code files. Another solution would be to provide information concerning location of doall_constructs to the linker and have the linker include coordinator code where needed. All of the above solutions are still under consideration to determine the most effective solution.

4.2.2.9 Code Generation

The compiler will produce code for both the coordinator and all processors. A very straightforward division of control would exist. That code required to synchronize DOALLs and to support interaction with the external environment would be resident on the coordinator. All other code would be allocated to the processors. The DOALL constructs just described are simple examples of this. The processors would each contain a copy of the doall-program-segment together with some identification of that segment. When the flow of control of the program arrives at the DOALL, the coordinator would broadcast a "start segment n" command. When all instances have completed and all processors have notified the coordinator with "I got here", the coordinator would synchronize the updating of OLD values in the EM followed by initiating the next program-segments in the processors.

Program segments which are not part of DOALL constructs but which are standard serial FORTRAN could be analyzed by the compiler to determine any data dependencies. Separate, data independent code sequences would be defined with the appropriate conditional tests so that each processor would evaluate one section of the resulting program segment. The controls in the coordinator would be the same as for the DOALL case (in effect, a "DOALL" would have been constructed out of the original code). Since the processors can all operate autonomously, this approach should result in additional speed-up on serial codes. A speedup on the order of 2-5 over straight serial execution is likely from this approach. The application analysis summarized in Chapter 3 DID NOT assume this speed-up of sections of serial code. Note that a separate high-speed "scalar" processor is not required. Each processor is independently capable of scalar execution, so that concurrency is not dependent upon vectorization, as it is in today's vector machines.

4.2.2.10 Functions

Functions on the FMP will include not only the normal mathematical intrinsics, such as ATAN, LOG, EXP, and SQRT, but also a family of functions that are brought about because of the parallel nature of the FMP. The global intrinsics, which reflect the parallel structure of the system, are described in more detail below. Table 4.1 lists the functions which could be provided in FMP FORTRAN. In addition to listing the function, the table also lists the expected implementation (such as operator, in-line expansion, or calls on external function subprograms. Some of the functions will combine in-line code with external calls and are marked for both.

4.2.2.10.1 Global Functions. The global functions have no analog in a serial machine and are not normally used in the direct description of a model. These functions are useful in the simulation controls and in the summary and analysis of the results of a simulation.

The global functions operate across the declared parallelism defined in the model structures. For example, the following serial FORTRAN

```
      A = 0.0
      DO 1 J=1, 100
      A = A+B(J)
    1 CONTINUE
```

would be replaced by

```
      DOALL J=1, 100 USING B
      A = SUMALL (B(J))
      ENDDO GIVING A
```

Note that this is implemented in two levels. First the sum of all the instances assigned to a given processor generate a partial sum. At the end of the DOALL construct, the coordinator generates Log_2 P (P = # Processors) operation sequences to associate pairs of partial sums to get a set of higher-order partial sums. These sums are then paired and summed successively until one value remains.

TABLE 4.1

FMP INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Type Conversion	Conversion to	1	INT	-	I	I	None
	integer			INT	R	I	Single instruction
				IFIX	R	I	Single instruction
				IDINT	D	I	In-line
	See Note 1						
	Conversion to	1	REAL	FLOAT	I	R	In-line
	real						
				SNGL	D	R	
	Conversion to	1	DBLE	DFLOAT	I	D	In-line
	double			DBLE	R	D	Single operator
					D	D	None
Truncation	int(a)	1	AINT	AINT	R	R	In-line
	See Note 1			DINT	D	D	In-line

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Nearest Whole	if $\underline{a} \geq 0$ then	1	ANINT	ANINT	R	R	In-line
Number	int ($\underline{a}$ + .5)			DNINT	D	D	In-line
	else						
	int ($\underline{a}$ - .5)						
Nearest Integer	if $\underline{a} \geq 0$ then	1	NINT	NINT	R	I	In-line
	int ($\underline{a}$ + .5)			IDNINT	D	I	In-line
	else						
	int ($\underline{a}$ - .5)						
Absolute Value	if $\underline{a} \geq 0$ then	1	ABS	IABS	I	I	In-line
	$\underline{a}$ else - $\underline{a}$			ABS	R	R	Single Operator
				DABS	D	D	Single Operator
Remaindering	$\underline{a}$ 1 -	2	MOD	MOD	I	I	Single Operator
	int ($\underline{a}_1/\underline{a}_2$)*$\underline{a}_2$			AMOD	R	R	In-line
				DMOD	D	D	In-line
Transfer of Sign	$\lvert a_1 \rvert$ if $a_2 > 0$	2	SIGN	ISIGN	I	I	In-line
	0 if $\underline{a}_1$ or $\underline{a}_2$ = 0			SIGN	R	R	In-line
	$-\lvert \underline{a}_1 \rvert$ if $\underline{a}2 < 0$			DSIGN	D	D	In-line

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Positive Difference	$\underline{a}_1 - \min(a_1, a_2)$	2	DIM	IDIM	I	I	In-line
				DIM	R	R	In-line
				DDIM	D	D	In-line
Double Precision Product	$\underline{a}_1 * \underline{a}_2$	2	–	DPROD	R	D	Single Operator
Choosing Largest Value	$\max(\underline{a}_1, a_2, \ldots)$	$\geqslant 2$	MAX	MAX0	I	I	In-line
				AMAX1	R	R	In-line
				DMAX1	D	D	In-line
Choosing Smallest Value	$\min(\underline{a}_1, a_2, \ldots)$	$\geqslant 2$	MIN	MIN0	I	I	In-line
				AMIN1	R	R	In-line
				DMIN1	D	D	In-line

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Square Root	$(a)^{1/2}$	1	SQRT	SQRT	R	R	In-line
				DSQRT	D	D	Externai
				CSQRT	C	C	External
Exponential	e**a	1	EXP	EXP	R	R	External
				DEXP	D	D	External
				CEXP	C	C	External
Natural Logarithm	ln(a)	1	LOG	ALOG	R	R	External
				DLOG	D	D	External
Trigonometric Sine	sin (a)	1	SIN	SIN	R	R	External
				DSIN	D	D	External
Trigonometric Cosine	cos (a)	1	COS	COS	R	R	Externaı
				DCOS	D	D	External
	See Note 6			CCOS	C	C	External
Trigonometric Tangent	tan (a)	1	TAN	TAN	R	R	External
				DTAN	D	D	External

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Trigonometric	arcsin ($\underline{a}$)	1	ASIN	ASIN	R	R	External
Arcsine				DASIN	D	D	External
Trigonometric	arccos (a)	1	ACOS	ACOS	R	R	External
Arccosine				DACOS	D	D	External
Trignometric	arctan($\underline{a}$)	1	ATAN	ATAN	R	R	External
Arctangent				DATAN	D	D	External
	arctan($\underline{a}_1$/$\underline{a}_2$)	2	ATAN	ATAN2	R	R	External
				DATAN2	D	D	External
Exponentiation	$\underline{a}_1$ ** $\underline{a}_2$	2	-		I**I	I	In-line
					I**R	R	External
					I**D	D	External
					R**I	R	In-line
					R**R	R	External
					R**D	D	External
					D**I	D	In-line
					D**R	D	External
	Note 2				D**D	D	External

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Double Separation	Give first of double precision entity	1	–	FIRST Note 3	D		R None
	Give second word of double precision entity	1	–	SECOND	D		R None
Read Timers	Returns hardware timer value	0	–	TIMER			I Single instruction
Test Machine State	Test for overflow	0	–	OVERFL	–		L Single Operator
	Test for underflow	0	–	UNDERF	–		L Single Operator
	Test for undefined	0	–	UNDEF	–		L External
Least Whole	$\lfloor a \rfloor$	1	FLOOR	FLOOR	R		R In-line
Number				DFLOOR	D		D In-line

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Least Integer	$\lfloor \underline{a} \rfloor$	1	-	IFLOOR	R	I	Single Operator
				IDFLR	D	I	In-line
Greatest Whole	$\lceil \underline{a} \rceil$	1	CEIL	CEIL	R	R	In-line
Number				DCEIL	D	D	In-line
Greatest Integer	$\lceil \underline{a} \rceil$	1	-	ICEIL	R	I	Single Operator
				IDCEIL	D	I	In-line
Sum of all elements of A	$\sum_{i} a_i$ Note 4	imax (1)	SUM	-	I	I	In-line + External
					R	R	In-line + External
					D	D	In-line
Product of all elements of A	$\prod_{i} a_i$ Note 4	imax (1)	PROD		I	I	In-line + External
					R	R	In-line + External
					D	D	In-line + External
Largest of all values of A	largest value a_i max Note 4	imax (1)	MAXVAL		I	I	Inline + External
					R	R	Inline + External
					D	D	Inline + External
Smallest Value	smallest value a_i min Note 4	imax (1)	MINVAL		I	I	Inline + External
					R	R	Inline + External
					D	D	Inline + External

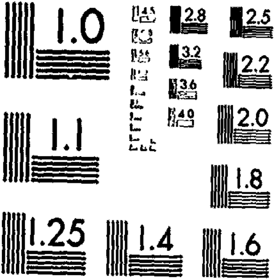

1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

TABLE 4.1

FMP INTRINSIC FUNCTIONS (Cont'd)

Intrinsic Function	Definition	Number or Arguments	Generic Name	Specific Name	Type of Argument	Type of Result	Implementation
Location	i at chosen result Note 5	none	LOCTRU	-	none	TBD	In-line + external
Logical location	i where true Note 5	imax (1)	LOCTRU	-	L	TBD	In-line + external
Or of a_i	OR of a_i over i	imax (1)	ANY	-	L	L	In-line
ALL	AND of a_i over i	imax (1)	ALL	-	L	L	In-line
Recurrence	$a(i+1)=c_i+b*a_i$ Note 6	3imax	RECURRENCE	-	R	R	In-line + External

Table 4.1 Intrinsic Functions (Cont.)

Notes for Table:

Note 1. The value returned by INT is that integer of the same sign as a with a magnitude not larger than $|a|$. If a is too large, integer overflow is reported.

Note 2. The representation of these functions in the FORTRAN source will use the standard double-asterisk notation for exponentiation. The function called will depend on the data types of A and B in A**B. The external function called is an alternate entry into the EXP function subprogram.

Note 3. In FMP format, FIRST and SNGL are different names for the same function.

Note 4. The values of i mark the elements of a domain. These, and the following functions occur across all those instances of a DOALL that execute the statement containing the function. Thus, with 2 arguments showing in the code, there will be 2x imax actual arguments, where imax is the size of the domain.

Note 5. LOCTRU finds the instance number of the instance in which the previous MAXALL, MINALL found a minimum. LOCTRU with a logical argument finds the instance number of one of the instances in which that variable is true.

Note 6. RECURRENCE is discussed below.

The result of a global function is not available for use within the DOALL in which it is called. Since the various instances of a doall-program-segment may be executed in arbitrary order, any given instance may complete before some other instance has provided its input to the global function. Thus, the output of the function is not defined until the execution of the last instance. The results of the global function are available after control passes the ENDDO.

4.2.2.10.2 LOCATION. The LOCATION function operates with the assumption that the value returned is the instance number of the successful instance of the most recent execution of MAXALL, MINALL, ... The subsequent use of this instance number as a subscript depends on the implicit equivalence between a one-dimensional subscript and a unique multiple-subscript.

For example, given a structure variable A declared INALL over a domain the largest element of the array could be determined as follows:

```
DOMAIN /LAYER/: I=1,10000
REAL INALL /LAYER/ A
DOALL /LAYER/ USING A
IPTR = LOCATION (GLOBALMAX(A))
ENDDO /LAYER/ GIVING IPTR
PRINT A(IPTR)
```

4.2.2.10.3 RECURRENCE. The RECURRENCE function is only defined over domains active on one-dimension. The RECURRENCE function would be invoked as shown in the example below:

```
A(J+1) = RECURRENCE (A(J)*B+C(J))
where A is declared INALL across the DOMAIN.
```

The prototype compiler should implement only the simple form of recurrence, with B constant. The additive term need not be subscripted and may be missing. The constant B may be omitted when it is equal to 1.

RECURRENCE, the global operation, is the formation of a parallel linear recurrence in nine (=$\log_2 512$) steps as demonstrated by Shyh-Ching Chen in his doctor's thesis at the U. of Illinois [13]. In FORTRAN, consider

```
   DO 1 J=1, 512
   A(J+1) = A(J)*B + C(J)
 1 CONTINUE
```

This program segment takes 512 steps, each with one multiply, and one add. The parallel algorithm in RECURRENCE produces the same result in nine steps.

Although global sums, global products, and the parallel linear recurrence are functions in the language, they are not always the optimum programming method for producing these particular results. For example, take the serial FORTRAN below.

```
     DO 1 J=1,1000
     DO 1 K=1,1000
     A(J,K+1) = A(J,K) * B(J  ) + C(J,K)
   1 CONTINUE
```

There are several ways to write this in FMP FORTRAN given that the order of nesting the loops is irrelevant otherwise. Two of them are:

Method I:

```
     DOALL J=1,1000
     DO 1 K=1,1000
     A(J,K+1) = A(J,K) *B(J  ) +C(J,K)
   1 CONTINUE
     ENDDO
```

Method II:

```
     DOALL K=1,1000
     DO 1 J=1,1000
     A(J,K+1) + RECURRENCE (A(J,K) * B(J  ) + C(J,K))
   1 CONTINUE
     ENDDO
```

Method I, which executes the recurrence serially in an inner loop, runs over nine times as fast as method II, which executes each one of the recurrences in parallel across each value of J in turn. That is, method I is 512 times as fast as a single processor, while method II is 57 times faster than a single processor. The global functions are included for those cases where method I is not an available option.

4.2.2.10.4 Efficiency of GLOBAL Functions. The global functions are logarithmic in efficiency for domains up to 512 in size. That is, it takes nine steps to produce the 512-way result across all 512 processors. For larger domains, the global function is executed serially with respect to all those instances executed on each processor (called CYCLES). As a result, the number of steps required for SUMALL, for example, is N/512 + 8 where the domain has N elements.

4.2.2.1C.5 Direct Calls on Global Functions. An alternative construct for global functions is:

```
global-function-name /domain-name/ (argument-list)
```

For example:

```
B = SUMALL/DD(J)/(A(J))
```

is equivalent to

```
DOALL /DD(J)/ USING A
B = SUMALL(A(J))
ENDDO /DD/ GIVING B
```

This form is the equivalent of single-statement DOALLs when the statement is a global function.

Boolean global functions may be used directly in IF statements once evaluated. For example:

```
IF (ANY /DD(J)/ (A(J))) ...
```

is equivalent to

```
DOALL /DD(J)/ USING A
DUMMY = ANY (A(J))
ENDDO /DD/ GIVING DUMMY
IF (DUMMY) ...
```

When LOCATION directly follows such an implied single-statement DOALL, the compiler combines it into the DOALL of the previous global function.

For example

```
MM = MAXALL/DD/(A(J))
IX = LOCATION
```

is equivalent to

```
DOALL /DD/ USING A
MM = MAXALL (A(J))
ENDDO /DD/ GIVING MM
IX = LOCATION
```

4.2.2.11 Assignment Statements

The following pertains to execution within each instance of a doall-program-segment. Recall from section 4.2.2.6.1 (and Figure 4.11) that three classes of variables exist in doall-program-segments. They are called GLOBAL, STRUCTURE, and LOCAL.

All STRUCTURE variables have their old values when any instance begins execution. Assignment to any structure variable from within an instance will result in the new value being available only within that instance. Other instances would still refer to the old value unless they too had executed a similar assignment statement. Once all instances are complete, the STRUCTURE variables are all updated with the new values computed within the instances.

Assignment to a GLOBAL variable or array element will redefine the value of that variable within the instance in which the assignment is made. However, the original value of the variable remains available for reference by any other instance. Since a GLOBAL variable must have only one value, a doall-program-segment may assign new values to GLOBAL variables only through a GLOBAL function which maps a set of STRUCTURED variable values onto a single value. Such a new value is available only after the ENDDO statement. All other apparent assignments to a GLOBAL variable within the DOALL define the GLOBAL variable to the end of the instance.

Assignment to LOCAL variables may take place at any point during execution of an instance. Operation is as with standard FORTRAN except that upon completion of the instance, the storage space allocated to such LOCAL variables would be reassigned upon exit from the doall-program-segment. A compiler option will exist such that LOCAL variables would be assigned unique storage locations for each instance. In this case, LOCAL variable values would carry over from one reference to another, even between different DOALL constructs.

External to a DOALL, all references and assignments to GLOBAL and STRUCTURE variables are valid. In such a case STRUCTURE variables must be subscripted.

4.2.2.12 Miscellaneous Features

4.2.2.12.1 Same-line Comments. A reserved character, not in the FORTRAN character set, will be defined that may be used to terminate a statement. Thus, anything following on the same physical card is comment. A likely character is "%".

When the syntax of a statement is such that the only allowable characters on the rest of the card are blanks, the compiler will not check. Thus, statements like ENDIF, IF (-boolean-) THEN allow comments to be placed on the rest of the card.

4.2.2.12.2 Recursion. Recursive calls are allowed. Note, that although the second, nested call on the subroutine gets a second set of subroutine-local variables and arrays (separate from the set belonging to the outer call) any named common that belongs to the subroutine will be the same named common area in both calls.

4.2.2.12.3 DO LOOPS. Since a domain consists of a finite ordered set, the control of a DO loop can be specified with a set of domain elements. For example:

```
     DOMAIN /LAT/: I=1,IM
     DO 1 /LAT/
```

is equivalent to

```
     DO 1 I=1,IM
```

If the domain is multidimensional, the order of nesting of the "implied" DO loops is FORTRAN subscript order. That is, the first variable is the index of the inner loop. The last variable is the index of the outer loop.

4.2.2.12.4 EXIT Statement. The EXIT statement can be used to terminate an individual instance of a DOALL construct. In addition, a DO loop may be terminated with an EXIT statement. EXIT statements are permitted wherever executable statements are allowed.

4.1.2.12.5 Dynamic Array Sizes. Space is not allocated for a named common until the first program unit using that named common is entered. Likewise, space is not allocated for variables and arrays of a program unit until that program unit is entered. Hence, sizes of common blocks and dimensions of arrays can be and may be set dynamically during program execution. The only requirement is that the expression determining the size be evaluated at the point in the program where the declaration occurs. In the case of arrays in named common areas, the size-determining expressions must evaluate to the same value in every program unit, or a run-time error is likely.

4.2.2.13 Input Output

All FMP input and output is staged via the Data Base Memory. Since I/O is inherently serial, a mapping of concurrent execution to the serial form supported by peripherals is required. That I/O specified within the serial parts of FMP FORTRAN programs occurs as specified. That I/O specified within a DOALL over a DOMAIN or a REGION is processed as requested over time. Since the instances of a DOALL are independent, no attempt to order I/O of one instance with respect to another is made although the time sequence of I/O within any one instance is maintained.

Formatted I/O is expected to be supported primarily by the Support Processor since the major formatting load is on output. In addition, the applications studied were such that input formatting could be accomplished prior to initiation of an FMP task. As a result all FMP I/O would be direct I/O via the DBM. These assumptions have affected the instruction set choices in the processors of the FMP. No powerful character handling instructions exist at this time. Due to the heavy loading of output formatting to support the COM load (excess of 10,000 frames of graphic info/day), continued consideration is being given to moving formatting support onto the FMP. The system as evaluated (with Support Processor formatting functions) could certainly support the expected workload. The remaining questions pertain to whether a more cost-effective solution might exist.

4.2.3 FMP FORTRAN Compiler

As with any large design problem, a compiler development project involves a number of stages including some means of testing design ideas. The compiler discussed below is actually envisioned to be a succession of compilers beginning with what would best be described as a "prototype FMP FORTRAN compiler". Where appropriate, these discussions will point out features or capabilities planned for the prototype compiler or planned to be deferred to later versions.

The FMP FORTRAN compiler would execute on the Support Processing System. Source input, generated code and other output would reside in the NASF File System.

4.2.3.1 Functional Objectives

The functional objectives of the compiler are:

4.2.3.1.1. Support to the User. Not only should compile-time messages be clear, but run-time aids should be provided for debugging, for gathering statistics and for monitoring the dynamic execution of a program. Other facilities should include generation of optional memory, array and extent bounds checks.

4.2.3.1.2 Support of the Language. The defined language (FMP FORTRAN) would be the language supported by the compiler. No changes to the language or compiler would be made without consideration of the other.

4.2.3.1.3 Make Efficient Use of FMP Resources. The compiler may never be capable of implementing the "most efficient" use of FMP Resources. This inability is due, in part, to the data-dependencies which are run-time sensitive and, in part, to the complexities of global optimization.

The prototype FMP FORTRAN compiler is expected to implement limited optimization. The level of optimization at the prototype stage would be "peephole" optimization giving improved overlap of the FMP functional elements during execution. For example, register allocation could be adjusted so that the store to memory ending one statement would follow the first fetch or two belonging to the succeeding statement. Requests for data from EM (LOADEM's) would be positioned near the start of a program-segment. This position should make it possible for CN delays to occur concurrent with processing. Where possible, integer and floating point instructions would be rearranged to improve overlap. Optimization of this sort requires local, straight-forward data flow analysis probably using the register addresses as data identifiers.

Since static allocation of the defined processes onto the memory and processor resources is planned, the resources might not be used as efficiently as in a dynamic, "load-leveling" run-time allocation scheme. Unfortunately no efficient, yet simple, dynamic scheme has been studied as yet. As experience is gained, static optimization will occur in two major areas; data or storage allocation and processor allocation. For example, as data-dependency analysis improves, code can be generated which maintains STRUCTURE variables always within a processor if all instances which refer to those variables are also within the same processor. Data-dependency analysis would also likely be used to assign instances of DOALLS to processors on the basis of least communication with Extended Memory.

Another means toward meeting the goal of efficient use of FMP resources is to generate efficient object code. Some of this efficiency will be derived from classical compilation techniques (feasible since most of the task involves generating code for individual processors). Some of the efficiency will come because of the simplicity of having only one program in execution at a time.

4.2.3.1.4 Support the Operational Environment. The FMP Compiler would be able to provide the necessary linkages to the logical input-output subsystem. In addition the compiler would produce the necessary information for the linkage editor.

Since the proposed FMP organization is very modular and is likely to be implemented first with a limited number of processors, an option which must be available with the prototype compiler is to compile for "N" processors and "M" memories. This capability should add considerably to the time available for debug and system integration of the software since not all 512 processors need be available to begin system testing.

4.2.3.2 Functional Organization

Figure 4.12 shows the expected functional organization of the FMP Compiler. The internal interface between all components shown would be a common representation of the compiled program. Such a common representation should allow the development of compiler design and debugging aids. For example, the source generator module could be used at any phase of compiler execution to generate a record of the current state of compilation.

4.2.3.3 Domains

The prototype compiler would handle only rectangular domains. In addition, the domains would be constrained to a maximum of four domain variables with constant spacing. These restrictions are suggested to reduce the prototype compile complexity. The hardware proposed would tolerate any kind of index set as a domain. Language features have yet to be proposed for describing such non-rectangular domains.

4.2.3.4 Data Flow Analysis

Data-flow analysis is not required to produce executable code so the prototype compiler is not expected to have such an analysis capability. However, the compiler can do a much better job of optimizing when a data-flow analysis is included. One of the chief uses of data flow analysis would be to improve memory allocation decisions. For example, if more structure variables can be held in processor memory, the number of EM fetches and stores would be reduced with a likely improvement in throughput.

4.2.3.5 Memory Allocation

Memory allocation is static in the sense that only one program occupies the FMP at any given time, and that the same variable in that program always occupies the same memory address if the same run is repeated. Allocation is dynamic in the sense that space is allocated to named common areas only when the first program unit using them is entered; space is allocated to variables local to a program unit only then that program unit is entered; and these spaces are deallocated when the last program unit using these local variables is exited. Hence, the same physical memory area may successively be allocated to local variables in a number of program areas. As mentioned earlier, an option would be available such that no deallocation of unused memory space would occur. This option would be useful if data values are carried from one call of a subroutine to the next.

Program and data areas have no relationship to each other. They would be separately managed. In fact, separate calls on the same subroutine from different places may have the local working space of the subroutine allocated to different places in memory, and the code file for that subroutine will not have moved.

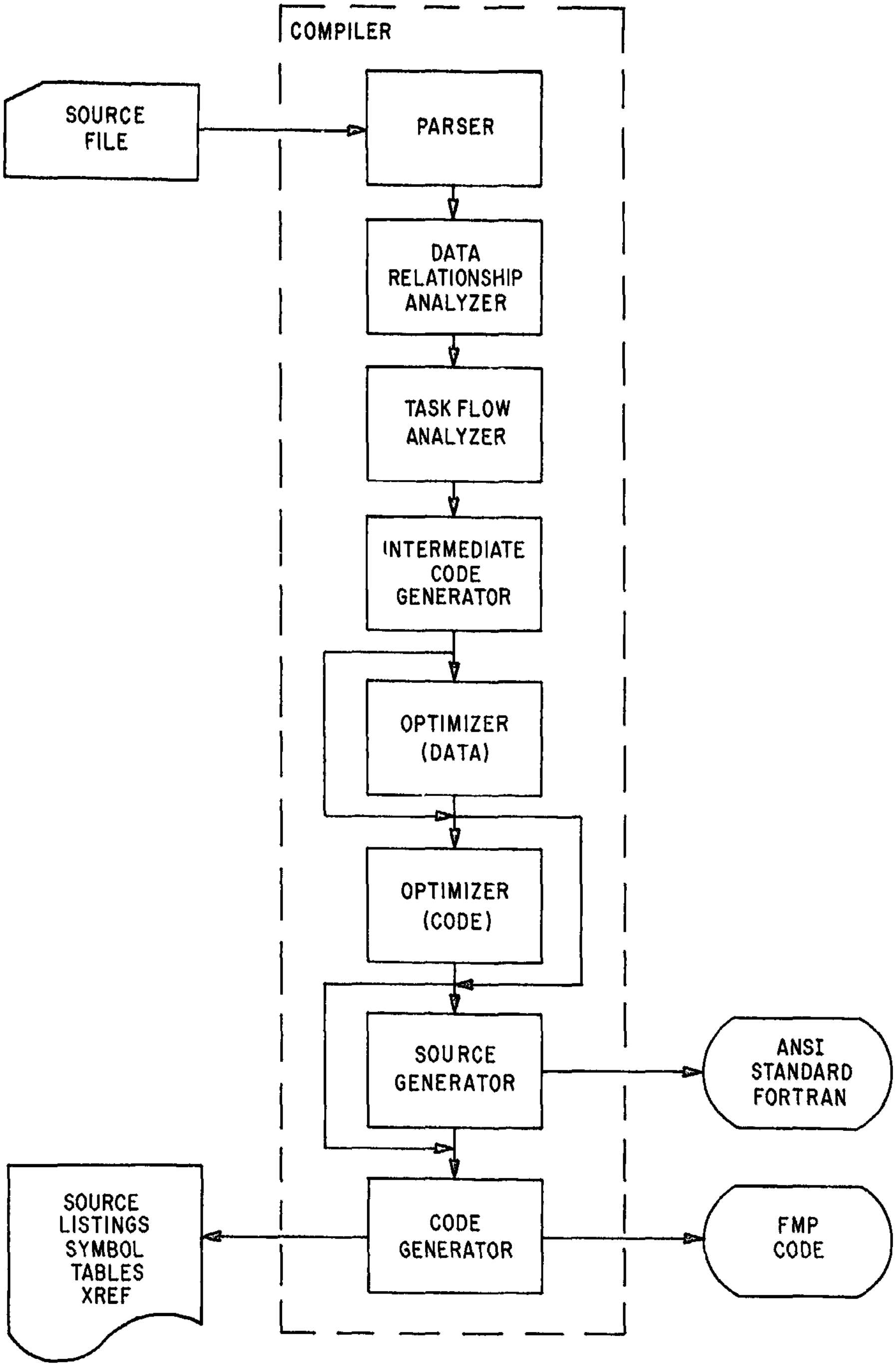


Figure 4.12 Functional Organization of FMP FORTRAN Compiler

4.2.3.6 Subroutine Entry and Return

The subroutine entry and return mechanism would be essentially that of standard Burroughs machines. This mechanism allows the deallocation of unused memory space rather than requiring the space of all subprograms to occupy physical memory addresses even during the time it is not being used. One of the integer registers would be used as a stack pointer. It points to a "return control word" (RCW) which contains: a) the memory address of the RCW of the procedure calling this one, b) the program counter setting to which return should be made, and c) the size of the memory area required by this program. Upon subroutine entry, the size field, plus the number of parameters to be passed, is added to the stack pointer, a new RCW is built, and written into memory at the new stack pointer. Upon subroutine return, the stack pointer and program counter are loaded from the RCW. The parameters that are passed include the base addresses of any shared named common areas, and pointers to any variables or arrays that are passed by name (in FORTRAN, all explicit parameters are passed by name. However, there is some implicit passing of parameters by value, as when calling a mathematical function.)

The result of managing subroutine working space as a stack is that recursive subroutine calls are allowed, even though there seems to be no use for them in aero flow and weather codes.

4.2.3.7 Concurrency

In the prototype compiler, the only concurrency allowed will be that of all the instances of a single DOALL. All instances would be executing copies of the same code file. Execution sequencing dependent on which domain element an instance is associated with could be controlled by testing the instance-variables to determine which element they represent. Nested DOALLs would have the inner DOALL implemented as an ordinary DO loop.

The hardware is not constrained to have all processors executing out of the same code file. Thus, in principle some instances of a DOALL could have one sequence of code, and other instances could have some other sequence, but this would not be allowed in the prototype compiler.

Capabilities for operations in which the processors operate asynchronously with no synchronization are not provided. Neither are capabilities provided in which a few processors are allowed to execute code separately from the other processors which are using the coordinator for synchronization.

4.2.3.8 Duplexed Computation Mode

A compiler option planned (but not for the prototype compiler) is to generate the code and controls to execute each sequence of code twice but with the spare processor switched between executions. In this mode, all execution occurs in a different set of processors on the second pass. The results of the two passes would then be compared as a confidence test for highly-reliable results.

4.3 OPERATING SYSTEM

The NASF should have only one operating system, pieces of which execute on the various portions of the system. In the discussions below, this operating system is called the Master Control Program (MCP). The purpose of the MCP is to provide software support for the following:

1. Scheduling and controlling the flow of programs and files to and from various processors in the system (including the Support Processing S,stem and the FMP),

2. Initiating staging of jobs onto the FMP,

3. Memory management including storage management and data management,

4. Support of the FMP FORTRAN programs for functions that cannot be performed in problem mode because of overall system implications,

5. Support of other functions of the Support Processor-FMP interface such as performance monitoring, error logging and operator control,

6. Support of the external environment including interrupt handling, I/O handling, peripheral control and data communications,

7. Providing certain system utilities such as dump, and system log analyzer,

8. Support of diagnostics and maintenance for all parts of the system.

The development of a system of this magnitude is a major task. During the study of the feasibility of the NASF, the MCP considered was based on the existing MCP on Burroughs 700 series and 800 series systems, in particular the B7800. The MCP of this system has evolved from systems as early as 1960 and is, therefore, a mature system which would need no modification to satisfy many of the above requirements. Recently, Burroughs has been developing the Burroughs Scientific Processor (BSP) as an attached processor to the B7800. The general philosophies of job flow and task management in the NASF and BSP are very similar. The MCP described below is therefore based on some of the design decisions and experience gained in the BSP project.

4.3.1 Assumptions

The evaluation of the proposed MCP implementation is based in part on the assumption that the FMP would be designed to operate most efficiently on tasks with the following characteristics.

1. Data areas up to the size of the extended memory (34 million words).

2. Long running programs: a minimum runtime of at least one second, a typical runtime of several minutes to several hours.

3. Batch job oriented: user interaction is not required.

Also, as discussed in Chapter 2, a self-managed file system supports the basic data management functions. This file system is assumed to not only provide the necessary data storage and retrieval functions, but would also maintain and enforce data ownership and access control.

4.3.1.1 Computational Envelope

An FMP task, once started, is assumed to run to completion within the high-performance computational and I/O environment of the FMP without requiring intervention of or access to the support processor or any of its I/O devices. The computational envelope is the high-performance environment. In particular:

1. All FMP program and data files are assumed to be fully contained within DBM while the program is in operation. All files holding the necessary input are assumed to be within the Data Base Memory (DBM) before the task is started.

2. Each FMP program is self-contained as far as resources are concerned. No dependencies on Support Processor actions shall occur during the runtime of the program. Therefore, no operator or user interaction would be permitted during execution of an FMP program. Operators and users would be able to query the MCP regarding the status of the job running on the FMP and would have normal controls such as cancel or suspend execution.

4.3.2 B7800 MCP

The existing B7800 MCP actually provides more functions than required for the Support Processor System of the NASF. Only those sections which are of major importance to the NASF MCP are summarized below.

4.3.2.1 Interrupt Handling

The B7800 style systems being considered for the Support Processor are all interrupt-driven. The interrupt handling section interfaces with all the resource-handling parts of the MCP. Interrupts are caused by the B7800 CPU by the I/O Processor and by software. Some of the interrupts processed by the interrupt handler are:

1. Caused by B7800 CPU
   a. Interval Timer
   b. Presence Bit not set (part of automatic memory management)
   c. Invalid Operand
   d. Invalid Index
   e. Processor-to-Processor Communications
2. Caused by B7800 I/O Processor
   a. Operator Request Pending
   b. I/O Complete
   c. Data Comm. Processor Ready-To-Send
3. Caused by Software
   a. Inter-Task or Intra-Job Communication

4.3.2.2 Memory Management

Memory management methods supported by the B7800 MCP are designed for implementation of the "virtual memory" concept within the B7800. Several methods of memory allocation are supported on the B7800. These methods include [11]:

1. On demand
2. Working set
3. SWAPPER

All methods use disk as the backup storage device.

4.3.2.3 MCP I/O Handling

Since the MCP is involved in all I/O to and from devices attached to the B7800, the MCP I/O handling functions are re-entrant code shared by all tasks running in the B7800 system. These I/O procedures perform the following functions:

1. Build the control words necessary to do a physical I/O operation
2. Send I/O instructions
3. Wait for an I/O operation to complete
4. Notify the associated program to continue
5. Handle physical I/O errors
   a. Retry where possible
   b. Enter user error routine if declared, or
   c. Discontinue the program

4.3.2.4 Process Control

The job selection process within the B7800 MCP considers the priority declared by the user, the time the process has been waiting, and the "class" (or system-level priority) of the task. The process control section supports the following functions in the B7800.

1. Inititation of tasks required by the user or by the MCP
2. Task scheduling
3. Perform "EOT/EOJ" duties such as deallocation and bookkeeping at End of Task or Job
4. Make administrative log entries

4.3.2.5 Peripheral Control

Peripheral Control procedures of the MCP are responsible for all peripheral devices on the B7800, except disk. These procedures perform the following functions:

1. Locate input data files
2. Assign output devices based on availability
3. Maintain and update table of all available units
4. Handle I/O parity recovery such as tape parity and card reader errors
5. Maintains system-level status such as ready, repair,... for all physical units including processors, memories and peripheral devices.

4.3.2.6 Work Flow Management

The processing of the tasks within a users job is specified through use of an easy-to-use, high-level work flow control language called WFL [12]. The work flow management software on the B7800 consists of a controller (which handles most keyboard input messages and places control records into a Job Description File), a WFLCOMPILER (which generates object code for presentation to the Process Control Section based on jobs in the Job Description File) and a job formatter (which selectively prints summary information about the job on the Job Summary sheets). Most operator keyboard messages are handled through the controller portion mentioned above.

4.3.2.7 Data Communications

The data communications section of the B7800 MCP is called the Data Communications Controller (DCC). The functions of the DCC include:

1. Allocation and deallocation of Data Communications Queues which are the interface mechanism between object programs, system routines such as the editors, and the DCC.

3. Dynamic reconfiguration of the Data Communications Subsystem

4. Generation and Maintenance of tables used by the Data Communications Processors

A system called NDL (Network Definition Language) [14] provides a user-oriented means of specifying network and terminal characteristics as well as what processing must be performed during I/O to match the terminal or network characteristics to the standard forms processed in the system.

4.3.3 Integration of FMP Task Management into MCP

FMP programs would exist as tasks within the standard WFL (Work Flow Language job structure of the B7800. The B7800 portion of the MCP schedules the FMP task to be staged into the FMP. Once such a task is initiated, it would run wholly within the computational envelope without any further B7800 dependence until the task terminates. The B7800 portion of the MCP may, optionally, query the status of FMP tasks, or override the FMP task-selection decisions.

4.3.3.1 Limitations

Some functions traditionally associated with operating systems are not provided on the FMP even though they are a normal part of the B7800 itself. Specifically:

1. FMP FORTRAN is the only language provided.
2. Interactive programs are not supported.
3. No provision, other than direct I/O, will be made for programs whose total file sizes exceed memory capacity.
4. Delays due to waiting for operator intervention on behalf of executing FMP programs would be eliminated.

The data base sizes expected are very large. If a job mix with a large number of very short jobs with large data bases is encountered, the file system and paths to and from the DBM would become a bottleneck. If this occurs, efficient utilization of the FMP would become difficult.

4.3.3.2 Interrupt Handling

The interrupt handling section of the existing MCP would be modified to include those interrupts caused by the FMP. The major interrupts from the FMP would be "Task Complete" and "Error State Pending". Task Complete would be a normal FMP task completion report. This response would be passed on to the Work Flow Management section to determine what task to process next. The Error State Pending would be the report of an abnormal termination. Status information would have to be scanned out of the FMP to determine whether the problem is user-related (such as overflow) or hardware related (such as a failure in that portion of the system which is not automatically corrected).

4.3.3.3 Memory Management

No change would be made to the B7800 Memory Management section of the MCP. However, input data and program staging would have to be initiated by the B7800 MCP for FMP destined jobs. Only the requests need to be made. The File System actually performs the function.

4.3.3.4 Process Control

The process control section of the B7800 MCP would be extended to support the scheduling and initiation of tasks on the FMP. The process control section would also maintain FMP log entries and statistics with respect to workload, job lengths, etc.

4.3.3.5 Work Flow Management

Extensions in the B7800 WFL (Work Flow Language) would provide the following functions:

1. Invoke the FMP FORTRAN compiler and linker.
2. Specify FMP resource requirements for scheduling and allocation purposes (such as the amount of DBM buffer area required during FMP task exectuion).
3. Specify job restart point following failure of any portion of the system.

In addition, the existing work flow management functions which support operator control of jobs and tasks in the system would be extended to include tasks running on the FMP. These extensions would include static controls to give visibility of the status of a task either queued or active on the FMP. In addition, the extensions would provide means for an operator to alter the priorities of tasks queued for service and even to force a roll-out of an active task (for later resumption). Such a roll-out would normally be only to the Data Base Memory.

4.3.3.6 Utilities

Various utilities specifically oriented to the support of FMP operations would be developed. These utilities would include various "analyzer" utilities to edit and format dumps.

4.3.4 <u>FMP Portion of MCP</u>

A portion of the NASF MCP would be resident in the FMP. In particular, the coordinator is the part of the FMP which would execute the FMP portion of the MCP. The functions provided would include:

1. Interface to the Suport Processor for FMP initialization, operator control, task forwarding, checkpoint/restart, dumps, etc.
2. Schedule and initiate tasks on the FMP from among those forwarded from the Support Processor. Provide wrap-up for normal and abnormal termination.

3. Establish connection between an active program executing on the FMP and the appropriate files in the Data Base Memory.
4. Service FMP interrupts such as invalid operand or errors.
5. Provide the appropriate run-time environment for FMP FORTRAN execution. This environment would include the appropriate intrinsics plus mechanizations of time, date, PAUSE and dump. The run-time environment would also support code overlay mechanisms, space allocation, and job roll-in and roll-out.

4.3.5 File Management

An independent file manager provides transparent management of all files on archive, disk, and in the Data Base Memory (DBM). This file manager is accessible from the FMP, the Support Processor, and the Users. Thus, the file management system will have capabilities exceeding those required only to support FMP execution.

One of the functions of the file management system will be to accept commands designating movement of or copying of files from one place to another. These commands would be utilized to initiate the movement of programs and input data to the Data Base Memory as needed for FMP execution.

The Data Base Memory, and its controller, are considered part of the File System portion of the NASF although the sole purpose of the DBM is as a staging memory of FMP jobs and data. Since the DBM is part of the File System and since the File System provides data and storage allocation capabilities, the portion of the MCP on the FMP does not require any filemanagement capabilities.

Another of the functions of the DBM will be to allow certain functions to be externally enabled. The best example of this capability would be a request to the File System by the Work Flow Management portion of the MCP (executing on the Support Processor) to cause the movement of result files of a particular FMP job back to the active files from the DBM. This request could be made contingent on a message from the FMP portion of the MCP to the DBM controller that the result files are closed and can be released.

Other functions to be provided by the file management system will include:

1. Dynamic allocation and deallocation of space as required.
2. Establishment and maintenance of directories or other techniques to map external requests (which will be in terms of the "name" of a file) to the appropriate physical storage area.
3. Backup and archiving of files based on specified conditions or time intervals.
4. File Security functions which would allow user control over which programs and/or users would be allowed to read and/or update their files.

4.3.5.1 FMP Interaction with File Subsystem

Since the file system is self-managed, all references to data within the file syrtem would be by name of the data rather than by direct reference to its physical position. FMP interaction with the file system occurs at two levels of the system. First, the coordinator provides the high-level interface to the file system, in particular to the Data Base Memory Controller. Second, the Data Base Memory is part of the File System, and as such has an operational interface to the File System Manager and the rest of the file system.

The operational interface between the DBM and the rest of the File System provides the required data paths as well as control paths to support:

1. movement of files within the file system
2. storage allocation
3. security functions

The interface between the coordinator and the DBM has basically the same functions as interfaces between the file system and other NASF subsystems such as the Support Processor and the Users. Allocation of space within the Data Base Memory is controlled by the File System, not by application programs. The DBM maintains a table to convert file names into DBM addresses. Thus, the files referenced by the coordinator are referenced by name rather than by physical location.

Control of the files within the DBM follows the philosophy of the rest of the file system. Once a particular file has been opened by an external request, that file is frozen as far as allocation is concerned and remains resident (for example in the DBM where coordinator requests are concerned) until closed. The coordinator would have the capability of initiating a transfer from DBM to EM very similar to a DMA (Direct Memory Access). Such a transfer identifies the name of the file in the DBM and the length and physical location of the EM area reserved for the transfer.

Operation over this interface can be summarized as follows. When an FMP task has been requested (in the Support Processor), the Support Processor passes the names of the files needed to start the task to the file system. In addition, the FMP portion of the MCP is notified of the expected arrivals and an entry would be made on a queue of "pending" job requests. In the meantime, the file system would be busy transferring the requested files to the DBM. When the job currently executing on the FMP completes and its files are closed, the file system begins transferring those files back to the bulk storage regions. At this time, the coordinator, under control of the pending task list, takes those steps needed to initiate execution of the next task for which all required files are resident in the DBM. This task scheduling requires that the status of the file loading into the DBM be available to the coordinator. To begin the startup of a job, the coordinator would then open the program code file and request that it be transferred to some specific area of the EM. Other files used for standard system monitoring would be opened at the same time. The FMP task would begin execution after the coordinator completed broadcast of the code files to the processors.

Not all files would wait to the end of an FMP run to be unloaded from the FMP. The Support Processor would be able to specify the destination of expected DBM output files prior to completion of FMP task execution. The file system would then provide automatic staging out of the DBM once the file of interest is closed. More discussion related to this area can be found in Section 5.9 (DBM Controller).

4.3.6 Job Structure

A job is the only unit of work in the NASF. The job is itself a very simple program which invokes and determines the relative sequence of a set of programs. These programs constitute a set of logically related tasks which perform some data transformation on files. A job is written in FMP Work Flow Language (WFL) and it runs on the Support Processor (B7800). FMP WFL contins B7800 standard WFL as a proper subset, so any existing B7800 (or B7700) job can run unmodified on the NASF. The WFL commands are either simple action commands (RUN, COMPILE) or tests of conditions (IF SUCCESSFUL-COMPILE THEN...).

4.3.6.1 Organization of a Job

The basic outline of a typical job is constrained by the computational envelope and LINKER concepts (see Section 4.3.7). The typical job will contain, in sequence:

1. None, one or more FMP FORTRAN compilations
2. If there is a compilation, a LINKER task
3. Specification of necessary input files for FMP program
4. One or more executions of FMP programs

In addition, any number of B7800 tasks may be interspersed with the above, such as to generate input files, or to process output files.

4.3.6.2 Flow of Job

Figure 4.13 shows a general view of the flow of a job in the NASF. A job enters at the upper left (BOJ=Beginning of Job). First the job itself, the Work Flow Language, must be analyzed so the job is scheduled and finally analyzed on the CPU. The result is a JOBFILE which controls the sequencing of the rest of the tasks in the job. If FORTRAN compilations and LINKER tasks are requested, control remains on the left of the figure. When an FMP task is specified, that request together with the identification of any files needed is passed to the File System (upper right of figure). Once all the files have been staged into the DBM, the task is placed READY for FMP execution (lower right of figure). Once the FMP task is complete, the Support Processor is notified so that the next task specified in the Work Flow can be specified. When all tasks are complete, the job terminates (EOJ-End of Job-at lower left of figure).

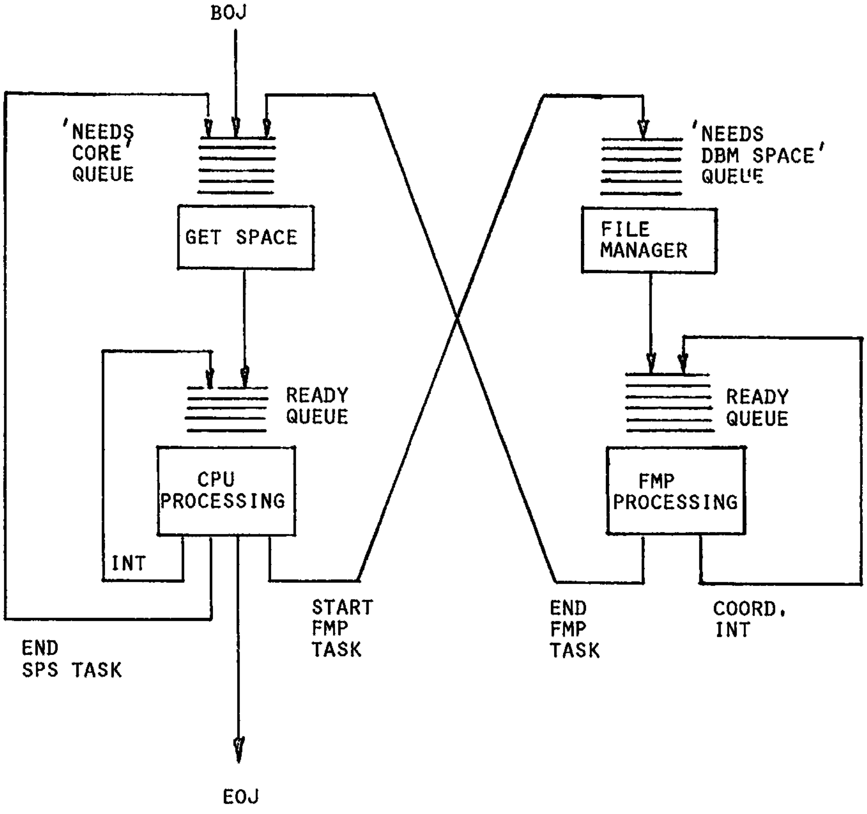


Figure 4.13 NASF Job Flow Diagram

4.3.7 Program Load and Overlay Support

The FMP evaluated would run only one program at a time. No additional program or data area may be preloaded into the EM or processor memories. Although preloading might minimize setup delays when starting the next task, additional hardware would be required to support the desired level of security. The Data Base Memory and its controller allow preloading of programs and data. Security can be better maintained at this level since all references to data in the DBM is by descriptor (or name).

The LINKER accepts object code files from one or more separate FORTRAN compilations and produces a single load code file, called the loadfile. In the process, the LINKER assigns memory locations to all program instructions and resolves or relocates address references accordingly.

For the case that the program memory part of the user program is too large (i.e. would not fit within the processor memory), the LINKER supports an overlay facility. With this mechanism, the user may divide a program into multiple phases and then may specify which phases share the same memory locations.

For the case that the data part of the user program is too large, the user may use the direct I/O facilities to and from files in the DBM. Automatic virtual memory mechanisms were not suggested for this system since the applications considered during the study did not require such mechanisms. If a significantly different workload and application for the system is expected (than the applications studied), the cost-benefit tradeoffs should be reevaluated.

Data is either initialized, uninitialized, or initialized to "invalid". Initialized segments have their initial contents present in DBM as generated by the Compiler/Linker. Uninitialized segments and segments to be initialized to "invalid" are not present on the DBM. In this case, storage is initialized by the execution of approprate FMP code.

4.3.8 Operations Support

4.3.8.1 Performance Monitoring

Certain information will be monitored during NASF operations, collected, and reported as part of the system log. Some of this information is accumulated by the B7800 as part of normal monitoring in the existing MCP. Other information would be collected by the FMP portion of the MCP. Some of the information that may be included in such monitoring is:

1. Interval timer reading at the time of the report
2. Real time clock at time of the report
3. Count of CN-using instructions
4. Some measure (to be determined) of the processor idle time
5. A measure of the time that the coordinator only is busy (i.e. all processors idle)
6. Count of succcessful error corrections
7. For each error correction, the address and the observed pattern
8. Time spent in specific subroutines
9. Others to be determined

The interval timer in the coordinator would be coordinated with the Support Processor at the beginning of a run.

Other monitoring in the FMP would be task related. Beginning-of-Task and End-of-Task of FMP tasks, OPEN and CLOSE of DBM files, and traffic to and from the DBM would be logged. Operator console system status display would be extended to include FMP tasks.

4.3.8.2 System Initialization

FMP initialization is that process whereby the FMP is transformed from an indefinite (i.e. any arbitrary) state into a state in which it normally processes user programs. This process reinitializes all parts of the system. Conceptually, the initialization process corresponds to a coldstart where not only is the MCP loaded, but all tables, directories, etc. are initialized.

Initially, no process corresponding to a "coolstart" (where the disk directory is saved) or to a "halt load" (where jobs are restarted from the last inactive point) will be implemented. Restart in the face of failures needs to be carefully studied since there seems to be a number of natural points at which execution could resume after a failure without having to reinitialize. In particular, while executing all the instances of a DOALL, if one processor failed, only those instances assigned to that processor would have to be recomputed in the spare processor. Since the ENDDO would have occured without successful completion of all instances, the old values from the start of the DOALL would still be available. Careful analysis of this sort of a circumstance may show other "natural" retry points in the system.

Initialization of the FMP itself consists of the following steps:

1. The driver program (executing on the Support Processor) determines that the B7800 - FMP connection is operational. This connection is a low bandwidth connection via the Diagnostic Controller (DC) part of the FMP and the Data Comm Controller on the B7800.

2. The driver transfers the FMP portion of the MCP to the coordinator via the DC. The coordinator then begins execution of its part of the MCP.

3. An initialization phase of the FMP MCP will perform various initialization functions, including confidence tests.

4. The MCP will then complete its initialization and inform the Support Processor.

The FMP is then ready to process programs.

4.4 OTHER SOFTWARE REQUIREMENTS

Although the FMP FORTRAN language and compiler, and the NASF Master Control Program (MCP) are the key elements of the NASF, a number of other software capabilities and requirements exist. These capabilities and requirements might be classified as those which are supportive to the language and MCP developments and as those which may provide more general utility of the system.

To support the language, software development cannot stop with the compiler (both a prototype version and a more final version). In addition, a system development language must be identified to support the development of the operational environment. Input-Output Formatting routines would need to be developed, especially if a final review of the impact of various system scenarios show that the Support Processor would then be the appropriate system resource to provide all I/O support. The program library and overlay facilities that may be desirable would be supported by a LINKER or BINDER.

Those jobs in execution on the NASF will need to be able to utilize various intrinsics, some of which will be resident on the FMP. These intrinsics would include FMP task initialization (including EM and PM loading), run-time execution monitoring, and mathematical intrinsics.

Some of the simulation support that would be needed in the development of the NASF could be based on work done as part of this study. Simulators at various levels would be utilized, including:

NASF block-level simulation
FMP simulation for timing estimates
Functional simulators for early code development support

Another important area of software would be the systems developed to support the diagnostics and maintenance of the NASF (which are discussed in more detail in Chapter 6). These software tools would include:

Off-line FMP diagnostics which would be initiated by the Support Processor and exercise the FMP when no jobs were active.

On-line processor diagnostics to be used both as part of the off-line FMP diagnostics above and as a means of testing the spare processors when not actively assigned to user problems.

Automatically managed FMP confidence tests

Diagnostic generation tools to be available both during development and initial test of the system, and also as a tool to allow the Field Engineer to produce new tests as required.

All standard diagnostics and maintenance tools provided as part of any standard equipment included in the NASF.

Tester Software

In addition to the above capabilities, most of which must be developed specifically for the NASF, software already exists for that portion of the system which may be implemented with standard products. For the B7800 Support Processor, a complete set of languages, utilities, and application packages exist including:

ALGOL
PL/I
FORTRAN
COBOL

BINDER (linker)

CANDE (a text editor)

WORK FLOW MANAGEMENT (operating system)
NETWORK DEFINITION LANGUAGE (for communications control)

4.5 CONCLUSIONS

The implementation of a system such as the NASF is a major undertaking. However, the software portion of the system studied is a realistic task to approach since it can be based in large part on existing software. The major part of the operating system exists, including the techniques to control an "attached processor" with a computational envelope supporting one user at a time.

The language extensions would be straight forward to implement. Since the extensions are strongly biased to description of the problems rather than explicit mapping to the hardware and since the architecture reflects the structure of the problems, the necessary flexibility exists to allow growth and improved efficiency over the future of the NASF.

CHAPTER 5

FLOW MODEL PROCESSOR (FMP) HARDWARE

5.1 INTRODUCTION

This chapter contains the results of the past year's study with respect to the design of the Flow Model Processor (FMP) hardware. In significant areas, the FMP design presented here is substantially more flexible and more general purpose than the FMP design of Ref. 1. Whereas that FMP was tailored to be efficient on programs that could be vectorized, with some extention to the case where the data did not form vectors, the current FMP performs essentially just as efficiently whether the data can be arranged in the form of vectors or not. In the present FMP, the 512 processors can work together efficiently as a vector machine; they can be just as efficient when working as 512 independent scalar processors.

The FMP is capable of execution in a manner similar to lock-step array machines such as ILLIAC IV or the Burroughs Scientific Processor (BSP). Simple programs (a copy resident in each processor), with no data-dependent branching, will produce this result. The FMP is not limited to this mode of execution however. It is also capable of performing in the manner of conventional multiprocessors. Interprocessor synchronization is implemented via special commands and use of the shared memory (Extended Memory).

It is expected that the multiprocessor capabilities of the FMP would be used on array-oriented problems. In particular, all processors are cooperating on the same job, with each processor independently executing some small portion of the job. In this mode of execution it becomes important to have as small a time penalty as possible when synchronization of the processors is required. The coordinator gives the FMP the ability to do array-wide synchronizations in one instruction.

The result is an architecture that is much more flexible than the current generation of high-performance processors, in that there is no requirement to vectorize the algorithm. It is also easy to put a great many processors to work on a single algorithm because of the degree of interprocessor cooperation available through the coordinator and the common Extended Memory. Although the aero flow codes are dominated by vectorizable algorithms, there are portions, such as subroutine CHARAC in the explicit aero code, where the data dependency is different from processor to processor, and the independent execution of each processor simplifies matters greatly. The radiation and physics computations of the weather codes use the independence of the processors to an even greater extent.

Some of the more important design considerations are discussed in the following subsection. The sections following in this chapter review the FMF architecture, briefly list the system parameters and describe esch of the major elements of the FMP in turn.

5.1.1 Design Constraints and Considerations

During the course of major hardware development project, such as the FMP, consideration of and compromise between many (sometimes conflicting) requirements must be made. Some of the important considerations on this project (throughput, economy, hardware/software compatibility, and schedule) are discussed briefly below.

5.1.1.1 Throughput

One major compromise in the design of any processor is between processor performance and its cost. In this project, the point of maximum performance per unit of cost is identified on the cost vs. performance curve for a single processor. Enough of those processors are built to deliver the required throughput. This approach contributes to maximizing performance vs. cost for the FMP as a whole. The above evaluations result in the choice of high-speed ECL and implementation on large boards.

5.1.1.2 Economy

Although those sections of programs which are vectorizable can be conceptually implemented on a processor that enforces lock-step cooperation among all the processors, the hardware required to enforce such lock-step operation is almost missing from the FMP. Each processor is self-contained, with as rudimentary connection to the rest of the machine as the problem requirements will allow. The MIMD* construction of the machine also simplifies the software, both in terms of system software as well as for application program development.

5.1.1.3 Hardware/Software Compatibility

The overall economy of a system is directly affected by the hardware support of software requirements. In some cases specific hardware features may be required to reduce software costs. On the other hand, when hardware features are not required, system costs could be reduced by not providing these features. Some specific considerations on this project include:

(1) The FMP has only one user program resident on it at any one time.

(2) Data addresses are independent of code locations. Some degree of dynamic run-time data allocation is done. For

*Multiple Instruction Stream, Multiple Data example, space local to a subprogram is allocated upon entry to that subprogram, and released upon exit, using a stack mechanism for allocating space. Space is allocated to a named common only upon entry to the first program unit naming that common, and is deallocated upon exit from the last. Integer registers are used as stack pointer, and as pointers to named common areas. Many machines of the older generation allocate space permanently, even during those periods that the FORTRAN 77 specification declares them to be undefined. In the present case, that will reduce the size of the problem that can be handled. For example, in the implicit aero flow code BTRID is a large named common in subroutine BTRI, and subroutine SMOOTH has arrays SS and CT. These do not exist concurrently, so processor memory can be devoted to BTRID during the execution of BTRI and to SS and CT during SMOOTH. If space had to be allocated for both of these all the time, the largest allowable BTRID would be substantially smaller.

(3) Automatic stack pushing and popping on subroutine entry and exit.

(4) A full set of interrupts both at the processor level and the coordinator level.

(5) Requests to the Data Base Memory controller, for data in Data Base Memory, carry the name of the file involved, not its address.

5.1.1.4 Schedule

Historically, every two years worth of technological development has resulted in the delivery of computers that are about three times more powerful for the same cost. Thus, adding an unnecessary year between the design freeze and the delivery of a computer amounts to using technology that is one additional year toward obscolescence, and has a penalty of a factor of $3^{1/2}$ in computational horsepower. This trend has slowed recently. Even so, it is important to use straightforward, low-risk designs to achieve timely delivery.

5.2 FMP ARCHITECTURE

Figure 5.1 shows general organization of the FMP. The major elements are:

(1) 512 Processors, each containing a scalar execution unit and storage for data and program,

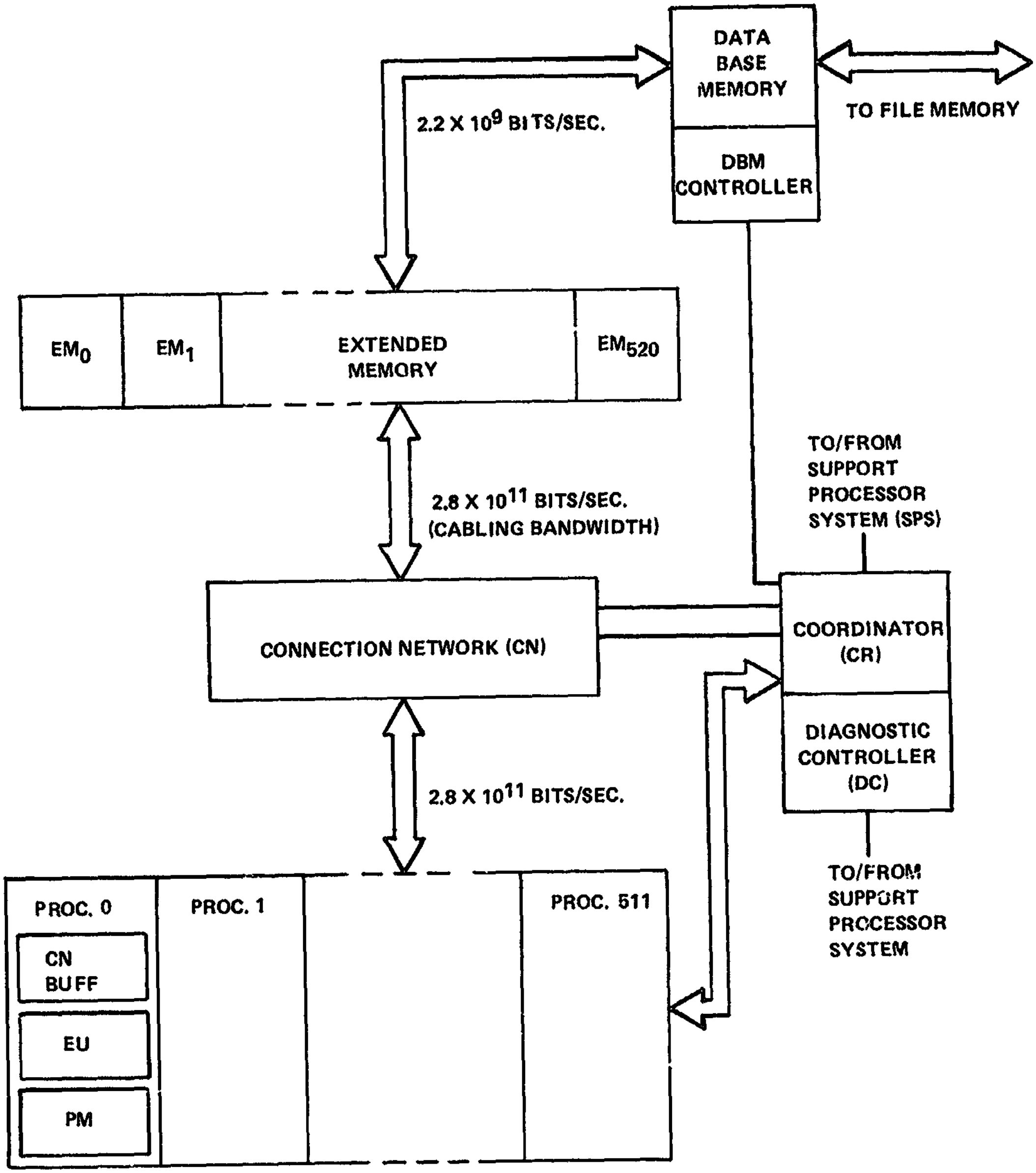


Figure 5.1 General Organization of FMP

(2) Connection Network used to interconnect processors and the Extended Memory,

(3) 521 Extended Memory modules, which hold the main data base of the program,

(4) Data Base Memory, used as a staging area for jobs to be scheduled and as a high-speed input/output buffer for jobs in execution,

(5) Coordinator, used to synchronize the processors, to interface to the Support Processor, and to run diagnostics, and

(6) Diagnostic Controller, which allows direct control of fault isolation in the FMP from the support Processor.

Each processor is self-contained, with integer and floating-point arithmetic units, its own instruction decoder, its own program and data memory. Four extra processors are included as on line spares to help achieve system availability requirements. In addition, four extra Extended Memory modules are included as on line spares, again to help achieve system availability requirements.

5.2.1 General Flow Through FMP

During normal operation, all data and program for the next run will be loaded into data base memory (DBM) prior to the beginning of the run. The DBM loading is initiated by the scheduler in the Support Processor via the File System Controller (these NASF system elements are described in Chapter 7). The scheduler initiates a run on the FMP through interaction with the coordinator (CR).

When the run starts, software in the coordinator initiates the transfer of code files from the DBM to the Extended Memory (EM). From there the coordinator causes its code files to be loaded in its memory and causes the Processor code files to be broadcast to each Processor. The initialization phase of the program (in the coordinator) then transfers necessary data to EM. These actions are automatically inserted by the compiler and the linker. With data in place in extended memory, and allocated space optionally initialized to "invalid", and with code files in place in coordinator and processors, user execution starts.

When user execution is in progress, the coordinator serves as a high-level "instruction sequencer". Processor tasks are explicitly initiated and when all processors complete their tasks (by indicating "I got here"), the coordinator causes the next task to be initiated in its sequence.

5.2.2 Changes from Baseline System

The Baseline System of the preliminary study (see Ref. 1 and Ref. 2) had the same basic organization as the system shown in Figure 5.1. The major difference is in the type of connection between the processors and the extended memory and in the system implications of that connection.

The Baseline System proposed use of a "Transposition Network" which allowed flexible access of vectors and array components from the Extended Memory. The "price" of this vector-fetching capability was that the processors had to be synchronized at each Extended Memory fetch time (accomplished by the Control Unit).

The modifications proposed during present feasibility study were to relax the need for coordination to only the start and end of concurrent, independent code sections. To accomplish this, an alternative scheme to interconnect the processors and memories was proposed which is called the Connection Network (CN). The reduction in synchronization requirements had the side-effect of greatly simplifying coordination tasks. These simplified tasks are handled by a unit now called the Coordinator (CR).

Evaluation of system loading has resulted in some proposed changes in bandwidth between FMP components. The current bandwidth plans are summarized on Figure 5.1

5.2.3 Basic System Parameters

No major changes have been made since the preliminary study. The choice of these parameters was covered in detail in previous reports (see Ref. 1 and Ref. 2). Following is a summary of the basic system parameters.

5.2.3.1 Logic Family

ECL is expected to be the preferred logic family. If the final design were being implemented at this time, Fairchild's 100K series would be chosen together with compatible memory circuits. F'.al selection of a logic family will be deferred to the appropriate point in the design cycle in order to gain the most effective, low-risk components.

Chip counts were made assuming chips projected to be available in 1980. Confidence in this count is supported by the count in a comparable processor which has been designed using circuit types available in 1978. See Appendix E of Reference 1 for preliminary data on this processor.

5.2.3.2 Clock Rate

The clock has been assigned a 40-nanosecond period. The instruction times, given in Appendix C are in terms of this clock period. These times are compatible with the instruction times derived from the processor design referenced to in Appendix E of Ref. 1. using ECL 100K.

5.2.3.3 Cabling Methods

The same flat belts used successfully in prior projects at Burroughs for transmitting high-speed signals with fast rise time and low crosstalk will be used for most of the interunit cables. Reference 1 discusses this choice.

5.2.3.4 Power

Power and grounding design details are discussed in detail later in this chapter. The primary design considerations are:

(1) A small number of centralized power conditioning modules that accept raw power from the mains,

(2) Switching regulators for efficiency

(3) Defense against faults in the incoming power,

(4) Defense against faults in the FMP,

(5) Noise reducing grounding methods, and

(6) Non-volatility of DBM contents.

5.2.3.5 Number of Processors

A key decision in the design of the FMP will be the choice of the number of processors to be implemented. Having designed the most cost-effective processor, then a sufficient number of them are linked together to produce the required throughput rate. Having done this, and found that 512 processors is the nearest round number to match the areo flow requirements, performance analysis then confirms that this approach produces a FMP that meets the aero flow (and weather) requirements. The processor design selected is one that matches the 80ns, 16K-bit by one, static RAM chips that are forecast to be available by the time the FMP is being designed. This is a fairly simple ECL processor, with 40 ns clock and 120 ns memory cycle.

A faster processor might allow the FMP to be built with 256 processors. This requires a faster memory, and therefore is projected to require a smaller (4K-bit), faster (30 ns) memory chip. The result is a doubling of the number of memory parts required. The faster processor is also estimated to require far more logic parts, with a net increase in parts count. More parts implies more failures, and hence a lowered reliability. Fewer processors, however, means reduced throughput penalty for those parts of some applications where concurrency cannot be found, and hence some extension of the spectrum of applications.

Final decisions will be postponed to take maximum advantage of components available at the time of design. For example, if the 16K-bit chips were faster than here forecast, a faster processor, but only 256 of them, might be perferred. If 64K-bit chips were available at the same speed of the 16K-bit chips here forecast, these would be preferred to the 16K-bit chips, since one would get twice as much memory with improved reliability due to the reduced parts count.

In such a case, it is possible that fewer processors would be needed to obtain the same throughput. When considering the 16-kilobit RAM versus the faster 4-kilobit RAM, the 4-kilobit RAM chip would require a 4-fold increase in the number of memory components. In this case, a trade-off between the reliability impact of a larger number of memory parts and possible reauced costs from a smaller number of processors seems to indicate that a more reliable system is the most cost-effective. It takes 512 processors, at 120 ns memory cycle (projected for 80 ns chips) and 40 ns logic clock, to yield the desired throughput of one billion floating point operations per second.

5.2.4 Modularity

Although the NASF requirements did not specifically address the problems of system modularity, the FMP design described below contains a very small numbor of standard modules. These modules are the Extended Memory module, the Connection Network switch module, and the Processor Module. The Processor Module, in turn, consists of an Execution Unit Module and a Processor Storage Module. There is also a Data Base Memory Storage Module.

This modularity allows the potential of configuring smaller (or larger) systems out of the same parts, with no impact on a user's perception of the system. In addition, such modularity greatly simplifies the magnitude of the design task for a system of the required capabilities and should reduce the fabrication costs since there will be many copies of a small number of parts built.

5.2.5 Preview of FMP Component Descriptions

Following is a brief description of each of the elements of the FMP together with a formatted tabulation of pertinent features and a block diagram of each.

For each element of the FMP, there is a table of characteristics given. A very short narrative description gives the intended function of the element in user programs. Source of control is identified, and the storage capabilities, both capacity and speed are also given. Connectivity to other elements is defined in detail.

The table also discusses the modes of error control built into the design. Most of these mechanisms are discussed in more detail in Reference 1 and Reference 2. The chip count is that projected for a 1980 design. "TBD" means "to be determined".

5.3 PROCESSOR

The array of 512 processors is charged with the task of executing the user computations in the program, namely the floating-point operations on the problem variables.

The processor executes code contained in its own program memory, and accepts commands from the coordinator. Certain instructions are executed in synchronism with the coordinator (and hence, by implication, in synchronism with the entire array, since the coordinator expects cooperation from all processors.)

The actions of the processor are delineated by the instruction set detailed in Appendix C. Figure 5.2 shows the division of the processor into an Execution Unit (EU), a Processor Memory (PM), and a CN Buffer (CNB). Table 5.1 provides data on the characteristics of the processor as a whole.

5.3.1 Execution Unit (EU)

Figure 5.3 is a block diagram of the Execution Unit (the logic part of the processor) and the CN Buffer, showing the independent integer and floating point units, with separate register files for each. Figure 5.4 is a diagram of the instruction fetching and overlap machinery. Table 5.2 provides data on the Execution Unit. Connections to the processor come from the control unit and the Connection Network. The synchronization signals and the 4-bit wide command path, and its strobe come from the coordinator. The data paths to and from the connection network are each accompanied by a strobe. In addition, each processor is connected to backplane wiring that expresses its own number.

Of the 129 processors in a cabinet, any one may be the spare processor. Suppose processor No. N is the spare processor.Then the backplane number for processors 0 through N-1 is correct but the backplane number for processors N+1 through 128 must be shifted down by one, to N through 127, in order that the processors being used by the program be consecutively numbered. Therefore, there is a 1-bit signal coming from the spares-designating machinery which tells the processor whether or not to subtract 1 from its hard-wired processor number to correct for the location of the spare. Two bits of processor number are the cabinet number, and do not enter into the subtraction.

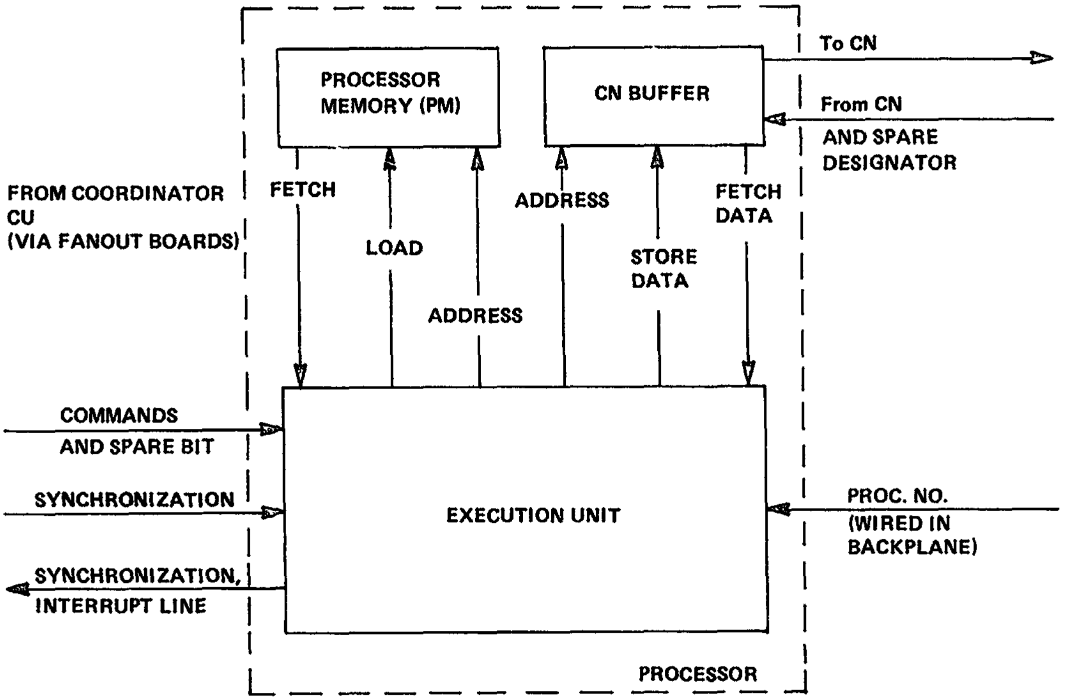


Figure 5.2 FMP Process Block Diagram

Table 5-1. Processor Characteristics

Number in System: 512 (No. of on-line spared: 4)

Function

To execute code written by FMP FORTRAN compiler, with an upper limit on speed of over three million floating point operations per second. The code is executed cooperatively with other processors and with the coordinator.

Mode of Operation

Execution of instructions fetched from processors own memory; execution of commands issued by the coordinator (diagnostics only); interaction with EM via the CN buffer.

Storage Capacities

32,768	words
120	ns cycle (odd-even interlace)
static RAM	technology

Connectivities

To/From	Function or Name	No. Signals	Timing
CN	Addresses and data to EM , data from coord. and EM,	24	20 ns per 11-bit frame 1st frame timed with 120 ns CN clock
CR	Commands plus strobe	5	Synch. with 40 ns clock
CR	Status bits to coord.	4	Change on any 40 ns clock
CR	"go" from coord.	1	40ns pulse
Backplane	Processor number	10	Wired-in levels
Fanout	Spare bit and spare designator	2	D. C. level
Fanout	Clocks	2	40ns clock pulse enable for selecting every 3rd one for CN clock

Table 5-1. Processor Characteristics (Cont'd)

Reliability/Repairability/Trustworthiness

SECDED checker on data bus

Numerous error checks leading to error interrupts

Parity on microprogram memory

For operation in the presence of failures spare processors can be switched in, or SECDED can be used to cover up failures in PM or EM.

Physical

Projected chip count:	240
Size:	1.2" x 11.5" 27.5 (narrow edge to backplane)
Power:	325 watts (including 100w losses in the switching regulator)
Additional Constraints:	Includes own self-contained switching regulator

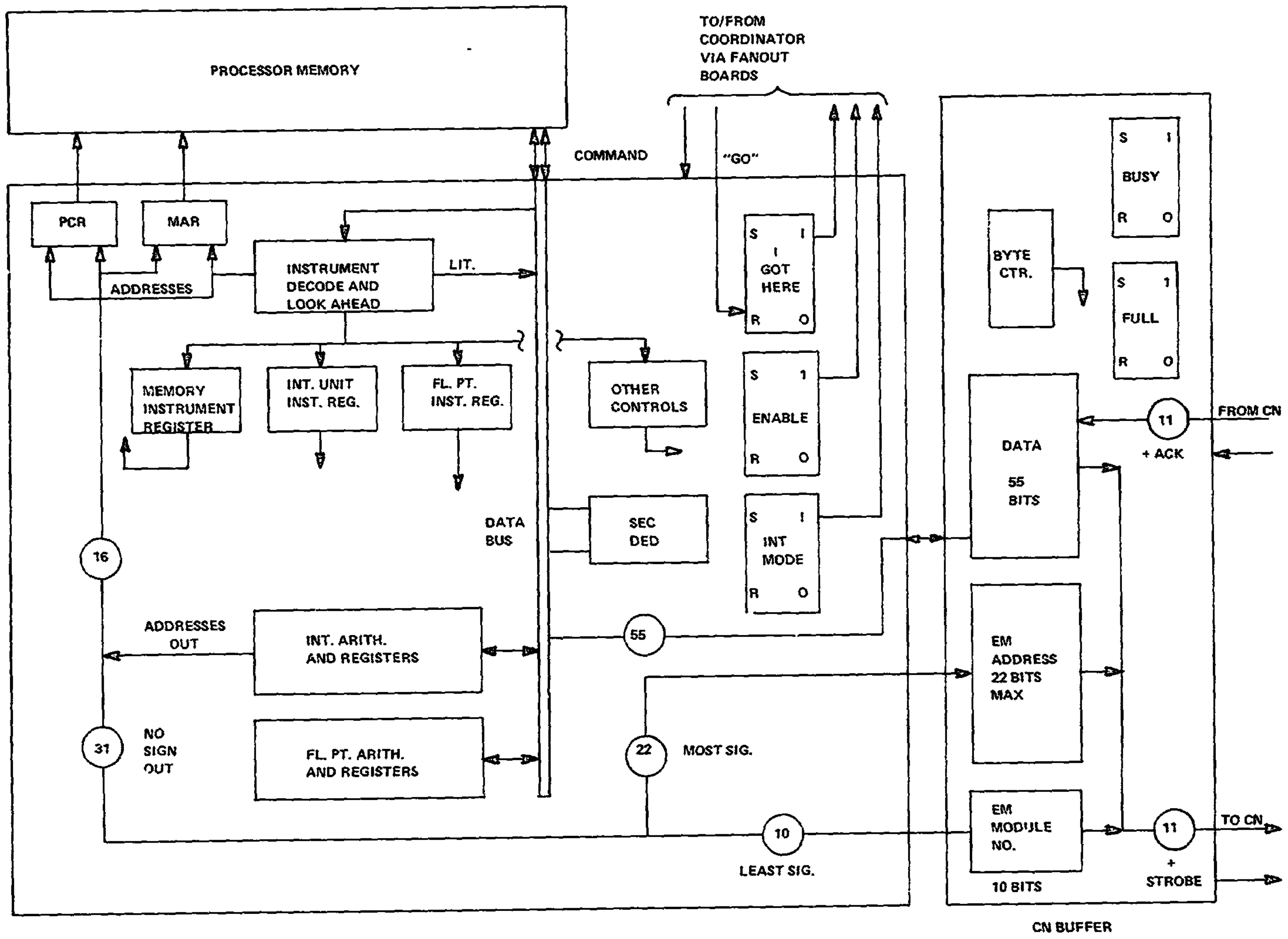


Figure 5.3 Processor Detail Block Diagram

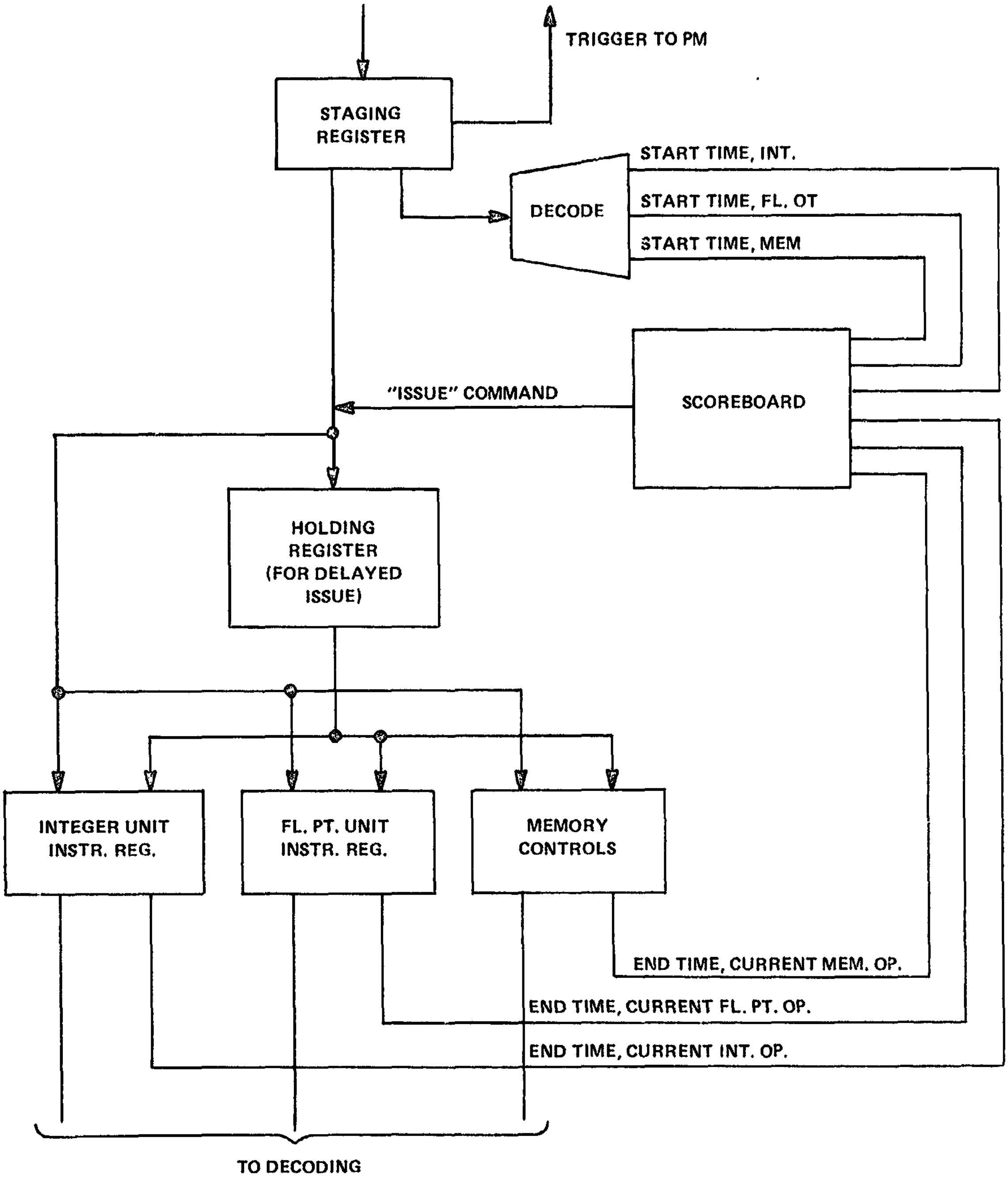


Figure 5.4 Instruction Fetching and Overlap Diagram

Table 5-2. Execution Unit (portion of processor) Characteristics

Number in System: 1 per processor

Function

Executes instructions and coordinator commands, accesses processor memory, and interfaces with CN buffer.

Mode of Operation

Clocked at 40ns clock, which is synchronous throughout entire system.

Storage Capacities

32	words in addressible registers, a few additoinal register also
40	ns cycle
ECL	technology

Connectivities

To/From	Function or Name	No. Signals	Timing	Comments
PM	Data (bidirectional)	110	Clocked	
PM	Address and command	20	Clocked	
CN buffer	Data (both directions)	110	Clocked	
CN buffer	Address path	34	Clocked	11 bits EM no. 23 bits address
CN buffer	Controls	5		
Fanout	Synchronizatoin & status	5		
Fanout	Commands from coord.	5		
Backplane	Processor number	10		
CN	Sparebit	1		
Fanout	Clocks	2		
Fanout	Spares designator and sparebit	2		

Table 5-2. Execution Unit (portion of processor) Characteristics (Cont'd)

Reliability/Repairability/Trustworthiness

Contains SECDED checker, microprogram parity, etc., as mentioned under processor

FAiled EU spared out by sparing out entire processor

Physical

Projected chip count:	100
Size:	About 11" x 10" within processor
Power:	125 watts

Error control within the processor includes SECDED on data bus transfers, parity on words in microprogram memory, and the assortment of error and bounds checks as listed in the description of the interrupt register.

5.3.2 Processor Memory (PM)

The Processor Memory (PM) contains data and program within each processor. Control is from the memory address register in the processor. There are 32,768 words of 55 bits each consisting of 48 bits of data and 7 bits of single-error correcting, double-error-detecting code. Data, address, and control connections are solely to the processor. 16k-bit static RAM chips are used. Table 5.3 describes major characteristics of the PM.

5.3.3 Connection Network Buffer (CN Buffer)

The CN Buffer accepts address, data, and commands from the EU, and in response to those commands, may transmit requests for either store or fetch to a named EM module, may accept data from the CN and may transmit data to the CN. The CN Buffer accepts commands from the EU only. The "strobe" or "acknowledge" received from the EM module via the CN is used as an indication of the success of EM requests.

Transmissions of data through the CN are synchronized with the CN clock, a submultiple of the processor clock. All CN buffers are synchronized to the same CN clock to eliminate time races in the CN.

Table 5.4 summarizes the characteristics of the CN Buffer. Figure 5.5 shows the states taken by the CN Buffer controls. The arcs in the graph of this figure are labelled with the events that cause change in state. For explanations of mnemonics, see the instruction set in Appendix C. All eight states in the top of the diagram are seen as "busy" by the EU. A four flip-flop internal state register is assumed. The six command lines from the EU carry different commands plus "go." Three of the requests (STOREM, LOADEM, and LOCKEM) result in codes being appended to addresses sent to the EM. In both cases where "go" is shown as triggering the change of state, an alternative would be for the "acknowledge" signal. on the 12th line on the data receiving side of the CN connection, to serve instead.

The 12 lines going from CN Buffer out to CN are 11 data lines plus a strobe that states the data is valid. The 12 lines coming from CN to CN Buffer are 11 data lines plus "acknowledge." Each 11-bit piece of data is called a "frame". Acknowledge is transmitted by an EM module upon successfully receiving a request through the CN, and stays up as long as the connection is to be maintained. The CN uses the acknowledge to latch up the chosen path, so the acknowledge is a logic level that stays up during the duration of the single operation.

Table 5-3. Processor Memory (PM) (part of processor, Characteristics

Number in System: 1 per processor

Function

To hold program for execution by the CU, and data to be fetched in response to that program.

Mode of Operation

Program counter (PCR) and memory address register (MAR) contains addresses for program and data respectively. The 16k-bit chips assumed by the implementation of choice, allow the interlace of odd and even modules.

Storage Capacities

32,768	words
120	ns cycle
NMOS static RAM	technology

Connectivities

To/From	Function or Name	Signals	Timing	Comments
EU	address	16	Clocked	
EU	data	110	Clocked	
EU	command	5	Clocked	

Reliability/Repairability/Trustworthiness

SECDED on all words fetched (SECDED generator/checker is in the EU)

Detection of illegal instructions, detection of the fetching of "unitialized" data, detection of fetching of unnormalized floating point words.

SECDED allows continued operation at reduced reliability in the face of single bit failures.

Sparing is done by sparing the entire processor.

Table 5-3. Processor Memory (PM) Characteristics (Cont'd)

Physical	
Projected chip count:	130
Size:	11" x 10" board in processor
Power:	100w

Table 5-4. CN Buffer (per processor) Characteristics

Number in System: 1 per processor plus 1 in coordinator

Function

To serve as an asynchronous interface with the CN, decoupling the program running in the PR (or the coordinator) from the access delays of EM and the CN.

Mode of Operation

Three registers hold EM number plus operation code, EM address within module, and one word of data. EM number serves as a request for an EM, when transmitted through the CN. The address register is loaded by the CR, and sent to the EM module at the appropriate time. The data word has bidirectional connections both to CR and CN.

Storage Capacities

1	words
40	ns cycle

Connectivities

To/From	Function or Name	Signals	Timing
CN	Data path (bidirectional)	24	20 ns per frame
EU	Data (bidirectional)	110	120 ns CN clock for initiations
EU	EM module no. and EM co command	14	40 ns clock
EU	Address within module	22	40 ns clock
EU	Misc. controls	9	40 ns clock
Fanout	"busy"	1	

Reliability/Repairability/Trustworthiness

All data passing through the CN buffer is checked at desintation for proper SECDED code

Sparing is with the processor of which the CN buffer is a part.

Table 5-4. CN Buffer Characteristics (Cont'd)

<u>Physical</u>

Projected chip count:	30 chips
Size:	NA
Power:	NA

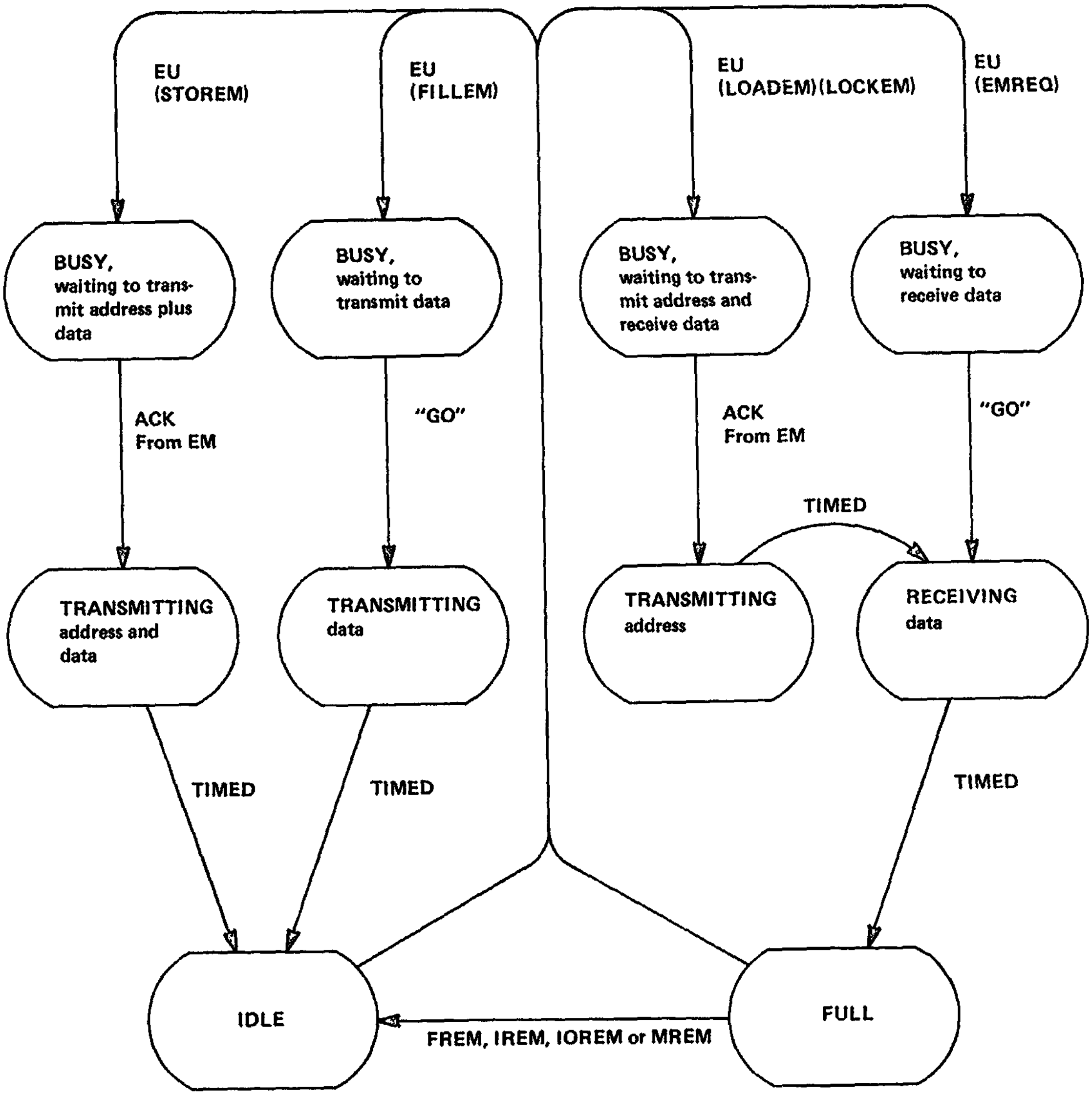


Figure 5.5 CN Buffer State Diagram

The CN Buffer also contains the capability of remapping from an EM module number of an EM module which has been spared out, to a different EM module number. There are 528 backplane slots for EM modules in the system, since all four EM cabinets are fabricated alike. This provides for up to seven spares. However, the reliability analysis is based on one spare per cabinet, and only four registers, in each CN Buffer, are planned for designating which modules are spare A 4 word associative memory, recognizing any one of four 10 bit EM module numbers, and substituting spare EM module numbers for them, is a suggested implementation.

5.3.4 Design Rationale and Changes from Preliminary Study

Size of the processor memory was selected on the basis of the known requirements of the implicit 3-D codes. In the preliminary study, the requirements were projected to be 16K words of data and 8K words of program. In this feasibility study, we have determined that it is less expensive to use a single uniform memory with no penalty in performance. Therefore, the Processor Memory (PM) now contains both program and data and is sized at 32K words.

As design progresses, it may become clear that 64 kilobit RAM chips will have adequate speed for this application. If that is the case and if the price is only twice the price per chip of the 16 kilobit RAM chips currently planned, then the design would be setup to use 64 kilobit chips. In this case a 64K word PM would result giving benefits both in larger storage capacity and higher reliability (fewer parts). See section 5.2.3.5 for other discussion.

Another area of change from the Baseline System (1) was the introduction of the Connection Network Buffer (CN Buffer) just described. The design objective of the CN Buffer is to provide an independent logic unit to which the CN-related operations can be passed while the EU proper continues processing. Waiting for EM access, or for CN connections, can be done in parallel with other processing instead of being in series with program execution. It is included in response to the asynchronous nature of the CN.

5.4 COORDINATOR (CR)

The coordinator serves two functions. The first is to serve as the focal point for array-wide synchronizations and array-wide cooperation. To this end, the coordinator is supplied with an array-wide synchronization mechanism, namely the "all processors ready", "go", "any processor enabled", "any processor in interrupt mode", and so on, as well as an access port to the CN which, in combination with processor cooperation, allows the passing of a single piece of data from coordinator or from one EM module to all processors, or from all processors, combined into a single word to the coordinator.

During diagnostics and initialization, the array-wide cooperation is imposed on the processors by the coordinator, which has a set of commands that are designed to read and write every accessible register within the processor, and generally to exercise any intraprocessor activity.

The second coordinator function is to run system software, interface with the support processor, and with the DBM controller for DBM-EM transfers, and also to be exercised by the diagnostic controller. Note that DBM access requests from the coordinator are in terms of file identifiers, not addresses.

The host initiates transfers between file-system and DBM using the DBM allocation map and issuing I/O commands directly to the DBM controller. No FMP-resident routine is involved in the initiation or completion of these transfers. The DBM controller resolves any potential conflict between these host transfers and a coordinator-CR-initiated DBM-EM transfer.

Figure 5.6 shows the Coordinator's two connections to the CN. One connection is a CN Buffer identical to the CN Buffer of the processor, and is used to access EM. The other connection is logically a memory port, and is used for injecting data to be broadcast to all processors, or for accepting data that has been harvested in parallel from all processors.

The Coordinator can be controlled by commands from the host (Support Processor) computer issued via the Diagnostic Controller. This interface is used to support the necessary interaction between the portions of the FMP Operating System resident in the Support Processor and in the Coordinator. In addition, the Support Processor can use this interface to initiate maintenance support procedures.

The speed of the Coordinator is set by the need to execute system software fast enough not to hold up user programming. That is, the Coordinator needs to be executing system software substantially less than the processors are processing user code. Handcompiled samples show that the Coordinator is almost completely idle during execution of user code. It will be recommended that system software be allowed to execute along with user code, letting "all processors ready" and "processor interrupt" pull the coordinator back to the user's code as required. It is also recommended that software conventions allocate certain coordinator registers for user program use only, and others for system program use only, thereby eliminating much of the swap time.

Figure 5.7 shows the block diagram of the Coordinator. Table 5.5 summarizes the characteristics of the Coordinator.

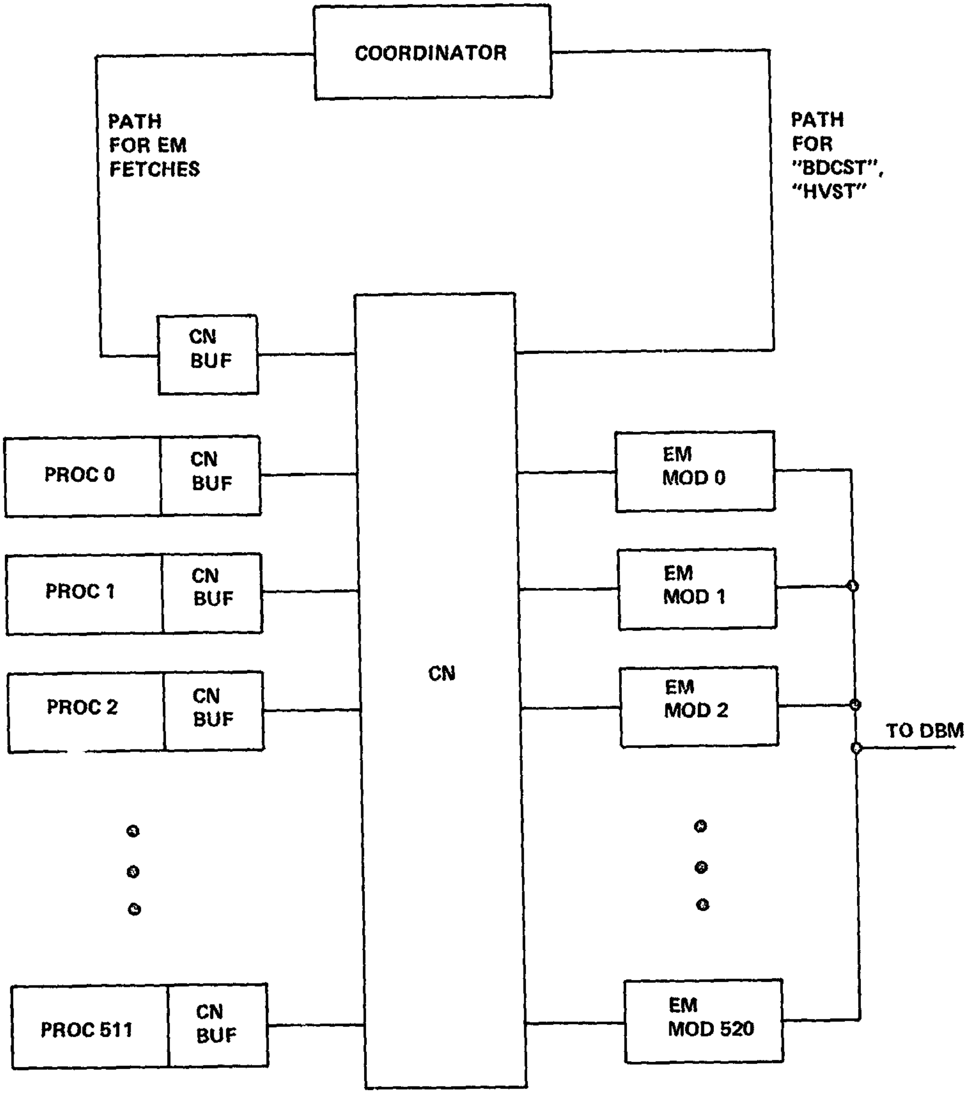


Figure 5.6 Connections to CN in FMP

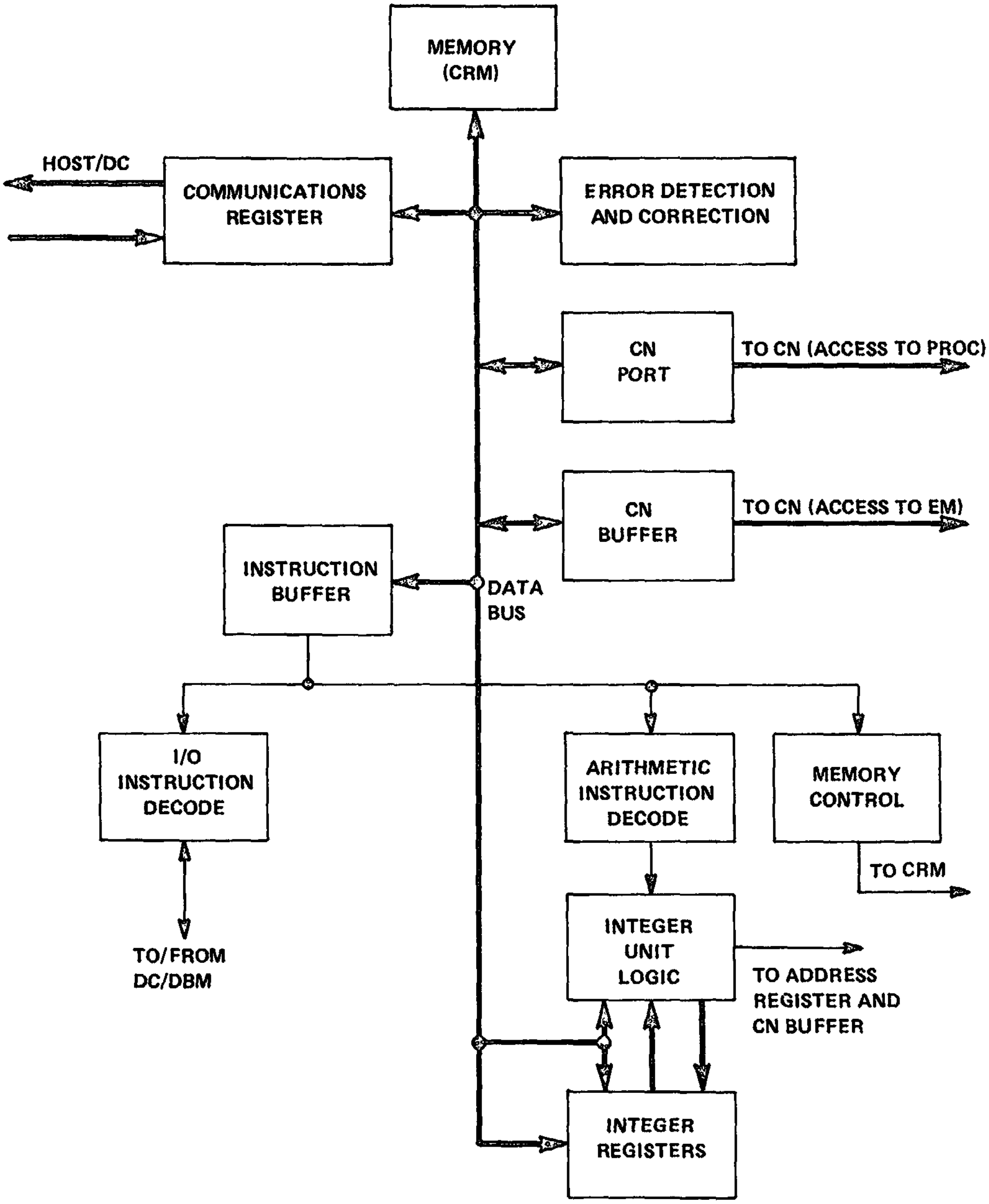


Figure 5.7 Coordinator Block Diagram

Table 5.5 Coordinator Characteristics

Number in System: 1

Function

Serves as a focal point for the achievement of array-wide cooperation of processors; serves as the issuing point of array-wide diagnostics.

Runs most FMP operating system segments, including interaction with host, logging of error events hardware reconfigurations.

Mode of Operation

Executes program. Interrupt mechanism allows switching back and forth between the two modes of operation.

Storage Capacities

32 registers (possibly more)
40 ns cycle
ECL register technology

Connectivities

To/From	Function or Name	No. Signals	Timing
Host	I/O channel	TBD	TBD
DBM	Descriptor issuance, controller status return	TBD	TBD
EM	Clock EM via EM fanout tree	2	40 ns Clock pulses 120 ns Select every 3rd as CN clock
EM	Error interrupts from EM	2	
CN	Control	24	with CN clock
CN	From CN buffer	24	20ns per frame; starts synch with CN clock
CN	to EM-like port	24	same but CN clock at this port is 60ns off from CN clock at CN buffer
Proc. via fanout	Command and strobe	5	
Proc. via fanout	Synch	5	

Table 5-5. Coordinator Characteristics (Cont'd)

Reliability/Repairability/Trustworthiness

Repertoire of error and reasonableness checks leading to error interrupt.

SECDED on data bus checks from coordinator memory, from CN buffer, and from CN to BDCST and HVST. Available for checking channels to/from host and DBM controller also.

Diagnostic controller has direct access to coordinator state.

Physical

Projected chip count:	2,000
Size:	20 to 30 large p/c boards
Power:	Not estimated

5.4.1 Execution Logic

The Coordinator has a number of semi-independent execution stations, so that more than one instruction may be in the process of execution at any given time, just as in the processor. The degree to which overlap, and its additional logic, are worthwhile, is a function of the amount of system software that the coordinator is required to execute. Using only the two aerodynamic flow models as benchmarks tells us that no overlap is required. Therefore the specification of a mechanism of overlap, as seen in the instruction listings, is only tentative pending further clarification of the computational load imposed by systems programming. The units are:

(1) Arithmetic unit,

(2) Memory,

(3) Interface to Support Processor and DBM controller, and

(4) CN buffer.

Instruction timing is given in Appendix D.

5.4.2 Coordinator Memory

The Coordinator Memory holds both program and data for the Coordinator. It is addressable only from the Coordinator and sends all data into the central data bus of the Coordinator.

The Coordinator Memory is identical in electrical design and uses the same 16k-bit RAM chips as the processor memories. The size resulting from considerations of the flow-model matching study is 32,768 words.

Table 5.6 summarizes the characteristics of this memory. Note that it is identical to the Processor Memories in all respects. As with PM, where the processor has a SECDED generator-checker for all memory words, so here the coordinator has SECDED also.

5.4.3 Design Rationale and Changes from Preliminary Study

The change from the old vector-oriented transposition network of the preliminary study to the random access connection network of the design currently described has released the processors from all requirements on regularity of relationship between the data processed by one processor and the data processed by any other. We now truly have 512 separate scalar processors in the FMP. Hence, all desire to have a separate, different, scalar processor associated with the Coordinator has disappeared, and the scalar processor in the control unit of Ref. 2 has not been carried over into the Coordinator.

Table 5-6. Coordinator Memory (CM) Characteristics

Number in System: 1

Function

To hold program for execution by the coordinator and data to be fetched in response to that program.

Mode of Operation

Program counter (PCR) and memory address register (MAR) contain addresses for program and data respectively. The 16k-bit chips assumed by the implementation of choice, allow the interlace of odd and even modules.

Storage Capacities

32,768	words
120	ns cycle
NMOS static RAM	technology

Connectivities

To/From	Function or Name	Signals	Timing	Comments
Coordinator	address	16	Clocked	
Coordinator	data	110	Clocked	
Coordinator	command	5	Clocked	

Reliability/Repairability/Trustworthiness

SECDED on all words fetched (SECDED generator/checker is in the coordinator)

Detection of illegal instructions, detection of the fetching of "uninitialized" data, detection of fetching of unnormalized floating point words.

SECDED allows continued operation at reduced reliability in the face of single bit failures.

Physical

Projected chip count:	130
Size:	11" x 10" board in CR
Power:	100w

5.5 PROCESSOR - COORDINATOR INTERACTION

5.5.1 Instruction Streams

The FMP is controlled by two instruction streams, which are created in parallel by the compiler from a single sequence of source statements. One instruction stream is being executed in the Coordinator; the other is being executed by all processors asynchronously of each other. Some statements in the source code result in instructions in both instruction streams. Some of these joint instructions require that the Coordinator and the processors synchronize themselves.

5.5.2 Synchronization

The simplest synchronization that may occur is the WAIT instruction, in which the processor sets "I got here". The coordinator is, or will be, executing a SYNC instruction. The SYNC instruction waits until "all processors ready" becomes true. "All processors ready" is the 512-way AND of each processors "I got here" OR NOT "enabled". That is, it is the N-way AND of the N enabled processors. After seeing "all processors ready", the coordinator issues a "go" command, received simultaneously by all processors, which then reset their "I got here" and execute the next instruction.

When the processor has raised its "I got here" line, but before it has received a "go" signal, it is said to be "waiting". The "I got here" line is dropped upon receipt of the "go" pulse.

A processor is not required to be idle while the "I got here" is set. Commands are provided to set the flag and to allow processing to continue. However, each "I got here" is considered a separate event so if the processor continued execution and wished to identify another "I got here" event, that command must wait as required for the flag to be cleared by a "go" command from the Coordinator.

5.5.3 Interface

Table 5.7 contains a list of Processor-Coordinator Interface signals and identifies their use.

In addition to the above synchronization, the CR also has the power to transmit commands. The commands are carried on a 4-bit-wide bus accompanied by a strobe line. Many of these commands are used in the diagnostic programs. Some of these commands are conditional on the "enable" bit of the processor, some are unconditional independent of the enable bit. No such command is used in user-generated FORTRAN programs, after initial program loading.

Table 5-7. Processor-Coordinator Interface

Processor	To or From Processor	Coordinator
"enabled"	from	"any processor enabled" = 512-way OR of "enabled"
"I got here"	from	"all processors ready" = 512-way AND of ("I got here" OR NOT "enabled")
"Go"	to	"Go" signal to CN buffer
"Interrupt coordinator"	from	"processor interrupt = 512-way OR of "interrupt coordinator" (a bit in the coordinator interrupt register
"Interrupt mode"	from	"any processor in interrupt mode" = 512-way OR of "interrupt mode" (tested by PINT instruction
"sparebit"	to	Designation of processor
"spare"	to	number of spare procesor
4-bit Command Bus	to	Synchronization and diagnostic mode command

In addition to the above synchronization, the CR also has the power to transmit commands. The commands are carried on a 4-bit-wide bus accompanied by a strobe line. Many of these commands are used in the diagnostic programs. Some of these commands are conditional on the "enable" bit of the processor, some are unconditional independent of the enable bit. No such command is used in user-generated FORTRAN programs, after initial program loading.

5.5.4 Fan-Out Tree (Coordinator-to-Processors)

A series of fan-out boards are supplied to implement the Coordinator-to- Processor Interface. Signals and clock fan out from the Coordinator to the final 516-processor destinations.

From the processors, the signals are combined, so that, within the Coordinator a single result appears in response to 516 signals emitted by the processors. For example, the "all processors ready" signal becomes true at the clock that the last enabled processor emits "I got here". Another such signal is the 516-input OR of "enabled".

At the processor, some signals are wired per-processor directly to the last level of fanout board; others are daisy-chained to eight processors from a single signal pin on the last board. The fanout boards are pin-limited. Simple buffers with one input pin and one output pin per signal dominate the circuit count, so hex buffers, easily available today, will not be improved upon by 1979-1980.

Figure 5.8 shows the Fan-out Tree. Table 5.8 summarizes the characteristics.

5.6 EXTENDED MEMORY MODULE

Extended memory (EM) is the "main" memory of the FMP, in that it holds the data base for the program during program execution. Temporary variables, or work space, can be held in either EM or Processor Memory (PM), as appropriate to the problem. All I/O to and from the FMP is to and from EM via DBM. Control of the EM is from two sources, the first is instructions transmitted over the CN, the second is the DBM controller which handles the DBM-EM transfers.

The Extended Memory consists of 521 on-line modules, and four spare modules, not used by the working program. Data is allocated to EM across the modules, with the allocation EM module number = Address modulo 521 (address is least significant portion) and address-within-module = address/512.

This addressing mode was chosen as a result of a software decision. Vectors are an important fetching pattern in the planned NASF applications (i.e., one vector element to each processor). It is therefore desirable to design the system so that vectors of 512 elements will be in 512 separate modules, reducing memory conflicts and allowing simultaneous access to EM for all processors. The number 521 is chosen because it is a prime number larger than the number of processors (512). This combination then contributes to the above desirable properties. For a more detailed discussion, see Ref. 1 & Ref. 2.

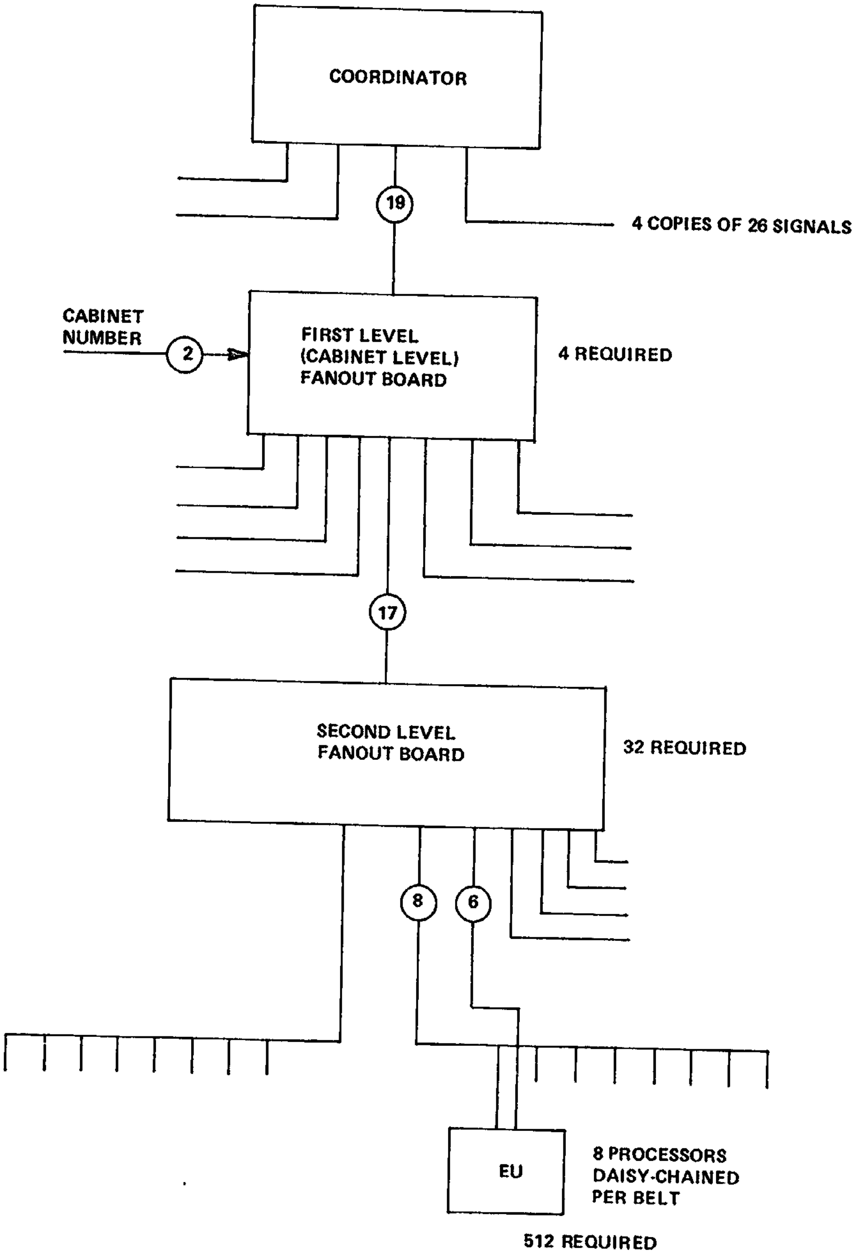

COORDINATOR
19
4 COPIES OF 26 SIGNALS
CABINET NUMBER
2
FIRST LEVEL (CABINET LEVEL) FANOUT BOARD
4 REQUIRED
17
SECOND LEVEL FANOUT BOARD
32 REQUIRED
8
6
EU
8 PROCESSORS DAISY-CHAINED PER BELT
512 REQUIRED

Figure 5.8 Processor Coordinator Fanout Tree Block Diagram

Table 5-8. Fanout (Coord-Processor) Characteristics

Number in System: 1

Function

Provide 512-to-1 connectivity from processors to coordinator. Provides 1-to-516 connectivity from coordinator to processors. Provides 1-to-129 connectivity from cabinet number to processors within cabinet.

Modes of Operation

Passive repetition of signals. No registers or program execution occurs within the fanout tree.

Storage/Capacities

none	words
	ns cycle
	technology

Connectivities

To/From	Function or Names	No. Signals	Timing	Comments
Coord.	Synch, status, and command and clock	19	Clocked	
Proc.	Synch, status command, clock, and cabinet no.	14		per processor

Reliability/Repairability/Trustworthiness

Very low parts count makes additional reliability precautions unnecessary

Physical

Projected chip count:	900 (of which 832 are hex buffers of one sort of another)
Size:	36 boards, 4 cabinet boards, 8 row fanout boards per cabinet

5.6.1 Basic Characteristics

Each EM module has a storage capacity of 64K words (48 bits data plus 7 SECDED bits/word).

From each EM module we need a transfer rate and access time consistent with the most economical implementation. An implementation in 64K-bit dynamic RAM is chosen for availability by 1980. The low chip count enhances reliability. A 240 ns cycle time of the memory is projected. Each word carries single-error-correction-double-error-detection code which is generated at the source (DBM, CR, or processor) and also checked there, so that transfer paths are covered by the same error control as the contents of EM. Figure 5.9 shows the general organization of each EM module. Table 5.9 summarizes the EM characteristics.

5.6.2 Connection Network (CN) Interface from Processors

The commands accepted by the EM module come either from the CN or from the DBM controller. From the CN, a "strobe" signals the arrival of a request. The EM module number accompaning the strobe is matched against the module's own number for error control purposes. Following the acceptance of the request by the EM, an "acknowledge" bit is raised by the EM module which locks up the CN path, and tells the requestor (processor or coordinator) that the request is being honored.

Following the strobe, and accompanying the address field, will be any one of four different commands, namely:

(1) STOREM. Data will follow the address; keep up the acknowledge until the last character of data has arrived. The timing is fixed; the data item will be just one word long.

(2) LOADEM. Access memory at the address given, sending the data back through the CN, meanwhile keeping the "acknowledge" bit up until the last 11 bits frame has been sent.

(3) LOCKEM. Same as LOADEM except that following the access of data, a ONE will be written into the least significant bit of the word. If bit was ZERO, the pertinent check bits must also be complemented to keep the SECDED code correct. The old copy is sent back over the CN.

(4) FETCHEM. Same as LOADEM except that the "acknowledge" is dropped as soon as possible. The coordinator has sent this code to imply that it will switch the CN to broadcast mode for the accessed data. The data is then sent into the CN which has been set to broadcast mode by the coordinator, and will go to all processors.

All of the above commands may arrive at any CN clock cycle. Except for the clocking, there is no synchronism imposed.

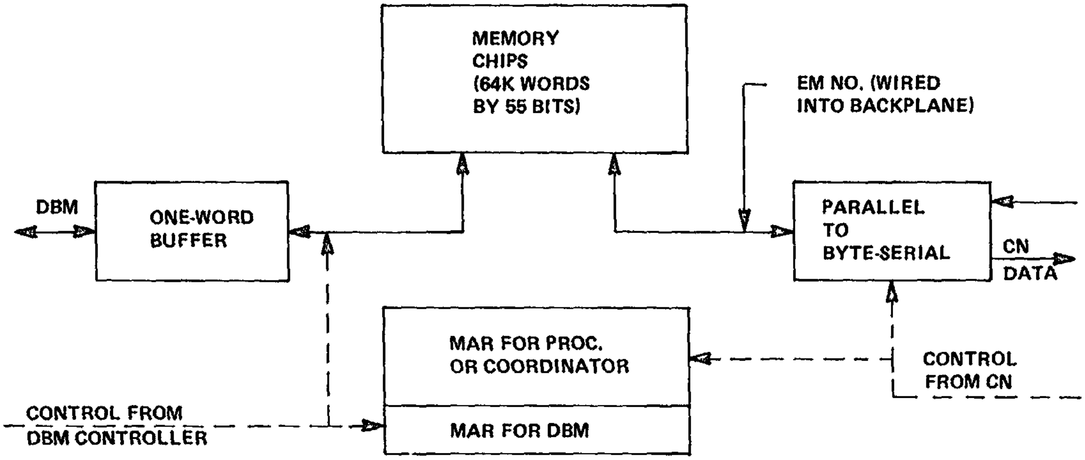


Figure 5.9 EM Module Block Diagram

Table 5-9. Extended Memory Module (EM module) Characteristics

Number in system: 521 (No. of on-line spares: 4)

Function

Serves as main memory for array processor; serves as shared memory among the processors.

Mode of Operation

Storage Capacities

65,636	words/module x 55 bits (48 data)
240	ns cycle
MOS dynamic RAM	technology

Connectivities

To/From	Function or Name	No. Signals	Timing
CN	Data, Addresses, Commands	24	20 ns per frame 1st frame synch. to 120 ns clock
DBM cont. via EM fanout	Read, Write, to DBM	36	Clocked by CN clock

Reliability/Repairability/Trustworthiness

All data is covered by SECDED. The generators and checkers are contained in the elements that are the source and destination of the data.

A parity checker checks parity on the module-number/address/op-code fields received through the CN.

Physical

Projected chip count:	85 (55 memory chips)
Size:	One 11" x 10" board
Additional constraints:	Each EM module may be self-contained for power regulation, just as is the processor, to simplify power distribution.

5.6.3 DBM Interface

In addition to the above, there are two commands that result in cycle-stealing for EM-DBM transfers. These commands and their addresses come from the DBM controller:

(1) Read from address to one-word buffer, and

(2) Write to address from one-word buffer.

The one-word buffers are loaded from, or unloaded to, the data bus to DBM under DBM controller control.

A transfer rate of 20 nanoseconds per word (50 million words per second) is achieved on this bus. Every 20 nanoseconds, the controls associated with this bus increment EM module number. Decoding logic for this module number is found in the EM fanout tree, where it is made conditional on the designation of spare EM module. The EM address space has 512 words at each EM address to simplify the address computations within the program. For writing, the EM modules are cycled after 512 words are loaded into the 1-word buffers, and those EM modules whose buffers are flagged "full" write, while the nine others do not. For reading, all 521 EM modules are caused to cycle, but only the 512 valid words at this address-within-module will be transferred to DBM. Incrementing of module number, for loading or unloading the 1-word buffers, is done in modulo 521. The address-within module is broadcast from the DBM controller, and is incremented every 512 words transferred.

5.6.4 EM Fanout

A second fanout tree, similar to that between the coordinator and the processors, comes from the DBM controller and carries requests for EM cycles from that controller.

It also carries EM addresses, and the two clock lines to the EM. Because of the requirement for addresses, this one has substantially more parts.

From the DBM controller comes address, command, clocks, and timing for loading or unloading the one-word buffers in the EM module. From the EM modules comes an "error" signal. Spares designation is done by controlling processor access, not by switching EM modules in and out, so no spares designation signals are in this tree. Figure 5.10 shows the EM Fanout Tree. Table 5.10 summaries the characteristics of this Fanout Tree.

5.6.5 Design Rationale

Size of the EM module is in direct response to Ames' statements about the size of the data base of the aero flow codes they expect to run on the NASF. Speed of the EM module is derived from observations about the number of EM accesses necessary to support a given quantity of floating point operations in the processor. The range of floating point operations per EM access was observed to typically lie between 5 and 20 for the aero flow codes. The resulting EM access times were seen not to impact the running time of the entire aero flow codes, although some minor sections of those codes were noticeably slowed by an accessing EM, at the currently designed speeds.

It should be noted here that advances in semiconductor memory technology may make it feasible to consider use of 256-kilobit chips instead of the current 64-kilobit chips. Also in the future, 64-kilobit chips can be expected to be reasonably faster than the current chips. Therefore, depending on when final design decisions are made, a tradeoff could be made between the following options:

(1) 256K words/module x 521 modules (large storage), or

(2) 64K words/module x 521 modules (current size but faster).

The considerations will be that option (1) would have much larger on-line storage with no impact on performance projections. Option (2) assumes existing plans for data storage requirements, but the faster parts would result in a faster system and increased throughput (note that here one could consider fewer processors and lower cost to get the requested throughput).

5.7 CONNECTION NETWORK (PROCESSORS TO EXTENDED MEMORY)

A flexible means of communication between the processors and the Extended Memory modules is required. In order to achieve a reasonable compromise between performance and hardware cost, the connection network is based on the "Omega" network (ref 6) rather than on the crossbar switch. The resulting network provides a path from each processor to the EM module selected by that processor. The network does <u>not</u> have a central, global control.

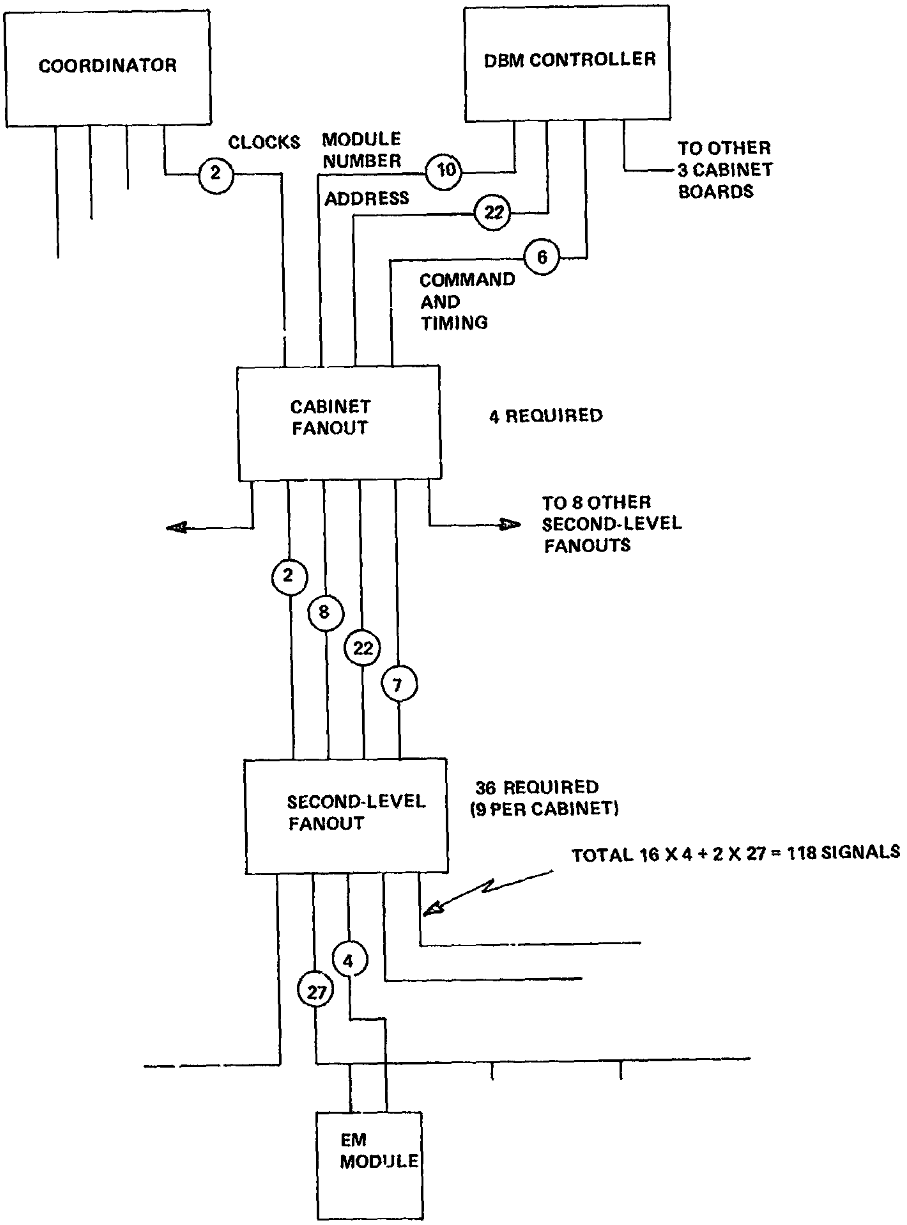


Figure 5.10 EM Fanout Tree Block Diagram

Table 5-10. EM Fanout Characteristics

Number in System: 1

Function

Distribute addresses and commands from DBM controller to EM modules. Distribute clock from DBM controller to EM modules.

Mode of Operation

Passive logic, no flip-flops, no execution of commands.

Storage Capacities

none words
ns cycle
technology

Connectivities

To/From	Function orName	No. Signals	Timing	Comments
DBM cont.	Addresses, control	34		22 bits of address
Coord.	Clocks	2		
EM mod.	16 above	36		per module

Reliability/Repairability/Trustworthiness

Low parts count makes additional reliability precautions unnecessary in comparison to the reliability of the rest of the FMP.

Physical

Projected chip count:	1250 (of which 116 are hex buffers of one sort or another)
Size:	36 boards

The requirements put on the Connection Network are that it have the immediate response to connectivity requests (tens of nanoseconds), that it have on the order of NlogN parts, as does the Omega or the Benes network instead of the N^2 parts of the crossbar switch, and that like a crossbar it be able to provide all N paths simultaneously when the requests for connection are a p-ordered vector, and that it be able to handle almost all N paths at once, with only modest delay imposed on a few of the requests, when the requests do not form a p-ordered vector. All of these can be accomodated in a design based on the connectivity of the Omega network as shown in Fig. 5.11.

The network has been designed with the added capability of processor to processor connection and provides transfer paths to and from the coordinator. Although the path connectivity of the network cannot be externally controlled, special communications modes (such as "broadcast") are available under control of the Coordinator.

The following discussion requires the use of certain definitions, as follows:

> A "p-ordered vector" is a set of requests in which the EM module number being accessed by processor N is equal to (d + pN) modulo 521, where d is called the "offset", and p is the "skip distance". When p is also the distance between successive addresses, p has also been called the "stride". "Stride" modulo 521 equals "skip distance."
>
> A "p-q-ordered vector" is defined in Appendix B, as a set of requests from processors 0 through 511 such that processor number i is requesting from memory module M_i given by M_i = (a + p*i + q*((i-b)DIV k)) modulo 521. In this equation, k is the length of each piece of vector, p is the skip distance within each piece, and q is the additional skip distance between pieces. The constant a is the offset. The constant b is the amount by which the first piece is short, since the first piece might be a leftover from some previous fetching of a p-q-ordered vector. A simple example is shown in Appendix B.

5.7.1 Functional Description

The Connection Network (CN) has two modes of control. First, in the normal mode, the CN establishes connections to the Extended Memory under control of the Processors. Second, the Coordinator may use the Network for a number of special purposes as described below.

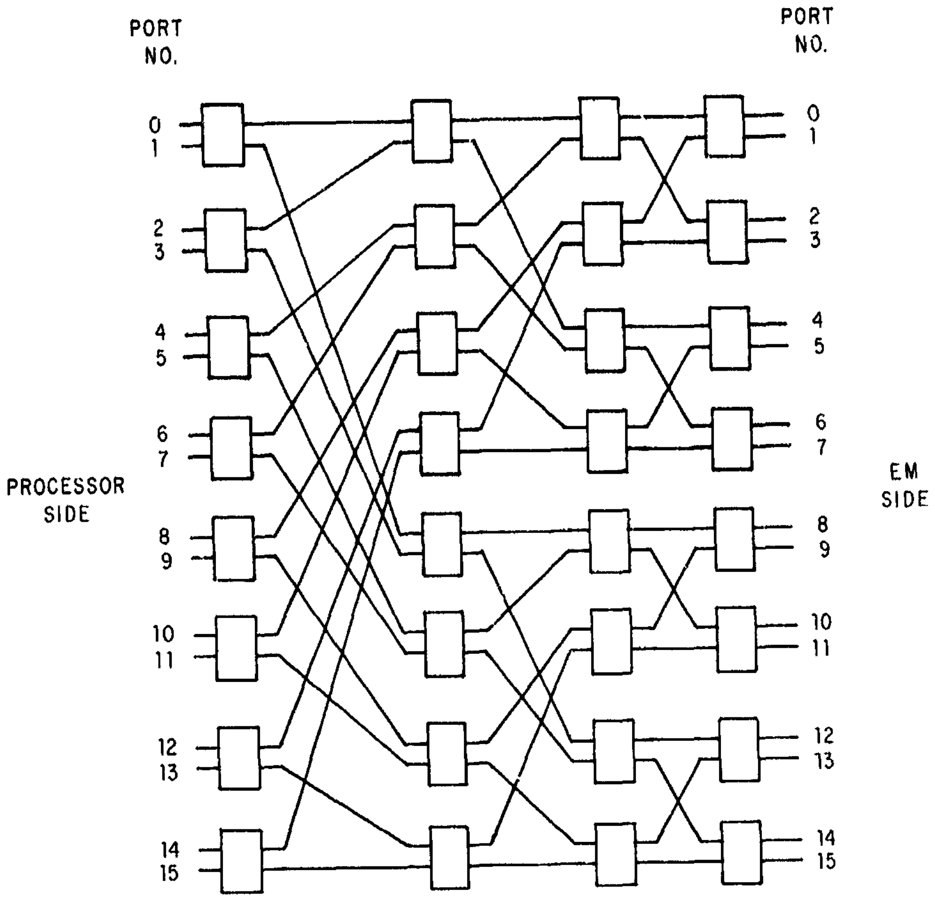


Figure 5.11 16 x 16 Omega Network

In the normal mode, a "request" establishes a two-way connection between requesting processor and the requested EM module. The establishment of the connection is acknowledged by the EM module. The "acknowledge" is transmitted to the requestor. The release of the connection is initiated by timing internal to the EM module. Only one request at a time arrives at a given EM module. The CN, not the EM module, resolves conflicting requests.

The following states of the connection network are established on command from the coordinator.

(1) "Broadcast from coordinator". One word of data is distributed from the coordinator to all processors.

(2) "Harvest to Coordinator". One word of data, representing the AND or OR or some mixture thereof, of the words presented by each of the enabled processors, is received at the Coordinator. Expected to be used by diagnostics with just one processor enabled.

(3) "Broadcast from EM". The EM module previously identified by a request from the Coordinator, will have the data being emitted by it broadcast to all processors.

(4) "Wraparound at stage n". Each pair of processors whose number differs by the bit at the nth bit position shall be connected, and data shall be swapped between them using the bidirectional path established. Processors whose port numbers are separated by 2^n swap data.

(5) Diagnostic control

(6) "Null". Respond to processor requests normally.

The connection network appears to be a dial-up network with up to 512 callers the processors, possibly dialing at once. There are 512 processor ports, 521 EM module ports, and two coordinator ports, one of which "looks like" a processor port, and the other like an EM port. Processor ports, and the coordinator port, are capable of accepting "requests".

The time required to set up each path is commensurate with the access time of EM, which in turn is designed to be suitable for the number of EM accesses observed in the applications studied. In the CN design described in this section, the minimum time to set up a connection is 120 ns. This time is achieved for most cases, including specific cases that are important in the aero flow and weather code applications studied.

5.7.2 CN Complexity Considerations

The basic Omega network provides only one possible path from a given processor-side port to an EM-side port. A network of this sort may experience blockage, especially during periods of heavy simultaneous usage by all processors. A number of methods were considered to reduce the probability of blockage and to increase the effective throughput through the network. Three of these methods will be summarized below.

The "natural" size (in terms of numbers of ports on each side) is a power of 2. Since there are 521 + spares + Coordinator connections on the EM-side, the network can be considered to be a 1024 x 1024 network. This additional size is the first method of reducing blockage. Half of the processor-side ports are unused and slightly less than half of the EM-side ports are unused. Thus, there is immediately a factor of two reduction in the maximum number of requests for service to the network. By spreading the active elements across all available ports, potential blockage is further reduced by reducing the total number of nodes in the network where blockage is physically possible, as explained in section 5.7.3 below.

The second method, a simple duplexed network, requires approximately twice the number of parts than the network just described. In this case, the network is duplexed (i.e., there are two copies) in order to provide alternate paths. Then requests that may be blocked on one Omega network may find a path on the second (which carries only those request blocked on the first "layer").

The duplexed network contains exactly twice as many 2 x 2 switch nodes and twice as many node-to-node connections (one set on each layer). In addition, a small amount of extra routing logic is needed on the processor- side and a small amount of arbiter logic is needed on the EM-side of the network.

A third method, a duplexed network with interlayer paths has even less blockage. In this method the total number of connections in the network is the same as the second alternative just discussed. The corresponding pair of 2 x 2 switch nodes (in the two Omega networks or layers) is replaced by one 4 x 4 switch node. Connectivity is provided between layers at each node, thus greatly increasing the total number of possible paths from a processor-side input to an EM-side output. The resulting network appears the same as the Omega network (Fig. 5.11) but each connection drawn actually is two independent connections and each node is a 4 x 4 switch rather than a 2 x 2 switch.

A threefold investigation has gone into the optimization of the CN. First, a functional simulator was written, in which a variety of test cases could be generated, and the resulting sets of requests submitted to the simulated CN to observe the behaviour. The processors in this simulation had a queue of up to five requests each. The number of processors making a request could be varied. There was provision to test 48 different CN design options.

Second, a statistical evaluator was written, in which the percentage of conflicts for random permutations on the inputs could be computed for a variety of different EN design options. For the CN option that they both handle, namely the single-layer Omega network, the evaluator and the simulator give identical results.

Third, an analytical evaluation of the CN behaviour, for particular CN design options, was carried out. Each of these is discussed in more detail in the Appendix B and Appendix H.

Either the simply duplexed network, or the duplexed network with interlayer ports would be acceptable. The latter has the least blockage, but a somewhat higher parts count. In the evaluations made, both the simple duplexed network and the duplexed network with interlayer paths had 100 percent success in fetching vectors in two of three directions. The simple duplexed network had a success rate of 77 percent in the third, or "hard" direction while the other, more complex network had a success rate of 87 percent in this case. (Success rate is defined to be the percentage of requests which connect immediately to EM-side outputs with no blockage. The experiments concerned had all processors active.) In either design, if vectors with preferred skip distances are presented to the network, 100 percent of the requests are satisfied immediately. A skip distance of one is always satisfied 100 percent. Table 5.11 is based on the simple duplexed network.

5.7.3 Processor and EM Connection Mapping

The Connection Network has 1024 ports on the processor side, numbered from 0 through 1023, and likewise on the EM module side. Because potential blockage in the network is a function of destination address and the origin of requests, the allocation of processors and EM modules to ports of the network becomes an important concern. The allocation function is called a mapping function. The mapping function serves to map processor number onto input port number and EM module number onto output port number.

Table 5-11. Connection Network (CN) Characteristics

Number in System: 1

Function

To serve as a dial-up network whereby each processor can access any EM module in a time comparable to the access time of the EM module. To serve also as a broadcase network wherein the coordinator or any EM module can broadcast to all processors. To serve as the converse of broadcasting in which teh coordinator can harvest a single word from all processors. To furnish some minimal processor-to-processor communication.

Mode of Operation

Individual 2 x 2 switches are combined into a locally controlled network. Control of the individual 2 x 2 node is generated within itself from the signals presented to it, without regard to the state of the rest of the network. There are no latches or flip-flops within the CN, it is entriely combinatorial logic.

Storage Capacities

words
ns cycle
technology

Connectivities

To/From	Function or Name	No. Signals	Timing	Comments
Proc/coord	Data path, processor side	24	20ns/frames 120ns major timing	513 such connection
EM mod./ coord	Data path, EM side	24	same	522 such connection
coord	Control	2		

Reliability/Repairability/Trustworthiness

All data passing through the CN is covered by SECDED, resulting in the correction of single transient errors, and the detection of all hard errors.

The internal redundancy of paths will provide that function continues for some, but not all, of the failure modes of the CN.

Table 5-11. Connection Network (CN) Characteristics (Cont'd)

Physical

Projected chip count: 39280

Size:

Power:

Port = 32 x (EMno MOD 512) + 1 for $512 \leq$ EMno $\leq$ 527

Within each cabinet, for the 256 ports in that cabinet, EM modules are attached to all even ports 0, 2, 4, etc., through 254, and to odd ports 1, 35, 39, and 103. In four cabinets, there are 512 + 16 ports thus addressible, allowing up to seven spares. Any spare can be used in place of any failed EM module, up to four total limited by the remapping in the CN buffer.

Furthermore, the remapping described above is done with simple wired-in shifting, and ORing. The substitution of spare for bad EM module is done by substituting one EM module number (521, 522, 523, or 524) for the EM module number of the failed module. The conversion from EM module number to port number is fixed, mostly just by wiring, in the CN buffer, as shown in Figure 5.12.

5.7.4 Hardware Aspects

5.7.4.1 Clocks and Synchronization

Requests are made in synchronism with the "CN clock". The CN clock is a submultiple of the processor clock. The CN clock will be synchronous and simultaneous across all requesting ports (512 processors plus coordinator). The acknowledge from EM module is received within a single CN clock period, since the CN clock period is greater than the roundtrip delay through the network. Since EM can be accessed only in synchronism with the CN clock, the EM cycle time will be a multiple of the CN clock.

Processor clock is distributed in synchronism to all processors. A signal which selects every mth processor clock pulse as the CN clock is also distributed from the clock source, but the timing reference is carried on the processor clock itself.

The values computed from projected characteristics are 40 ns for processor clock period, 120 ns for CN clock period, five CN clocks, or 600 ns, from the beginning of one request for read access (LOADEM) to the beginning of the next request for read access from the same processor, given that there are no blockages in the CN itself. For store access to one EM module, the CN buffer must wait 360 ns before accessing any other EM module, for either read or store.

5.7.4.2 Switch Element

Figure 5.13 shows the logic in one switch element. The control logic occurs once and the sets of AND-OR gates are each repeated twelve times as indicated on the diagram.

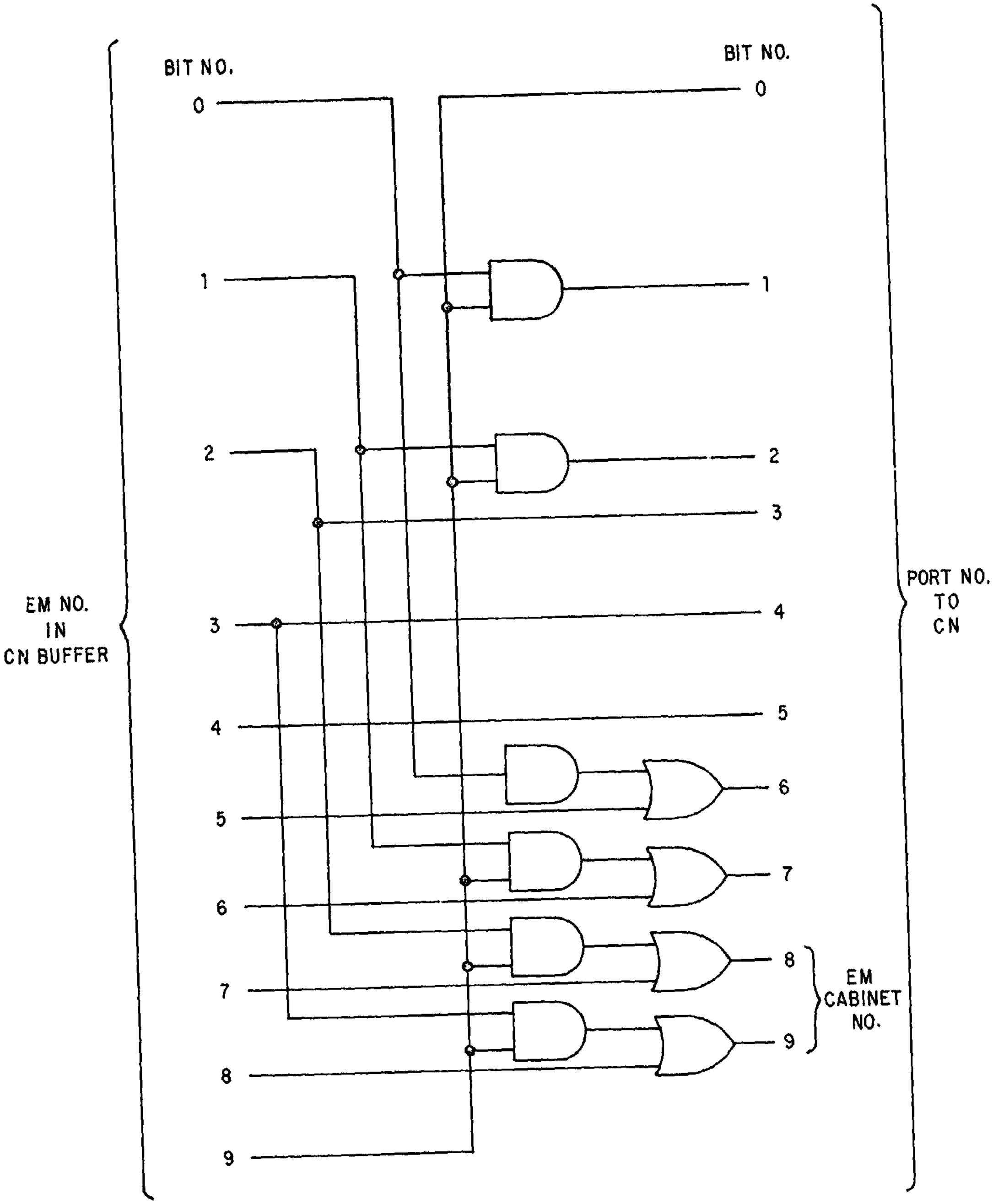


Figure 5.12 Mapping of EM Module Number to CN Output Port Number

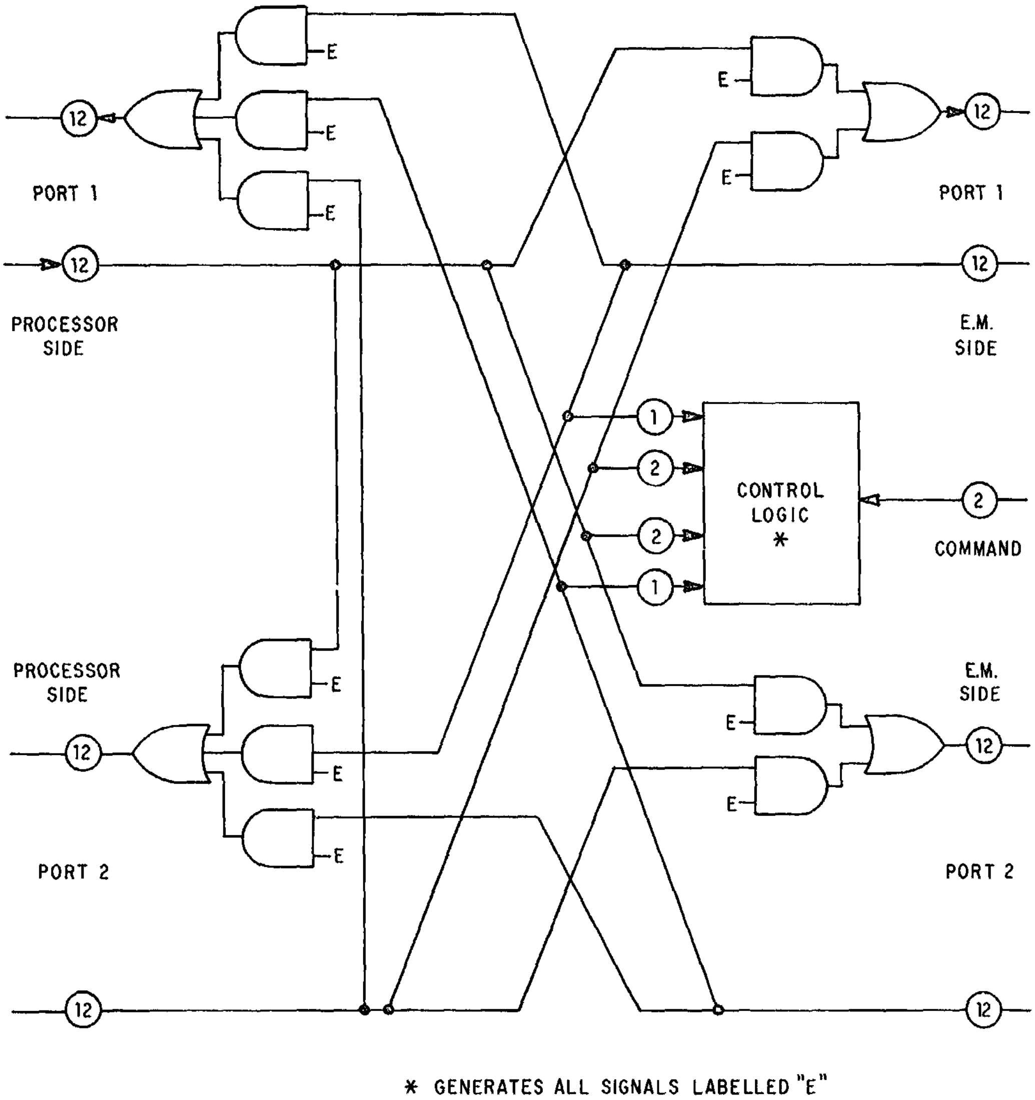


Figure 5.13 CN Switch Element

For the processors, several mappings have been tried or proposed:

1. processors 0 through 511 attached to ports 0 through 511.

2. Same, except processor number is bit-for-bit the reversal of port number. That is, processor number 110000000 is attached to port number 000000011; processor 1 is attached to port number 256; and so on.

3. Processors 0 through 511 connected to even numbered ports 0 through 1022.

4. Same as 3, except for the bit-for-bit reversal. That is, processor 110000000 is attached to port number 0000000110; processor 1 is attached to port number 512, and so on.

5. An assignment of processors to ports such that the connectivities of the omega network will make connection cyclically among the processors, processor N being able to transmit to processor N+1.

6. A random assignment of processors to ports.

Similar assignments can be made on the EM module side, except that the EM modules from number 512 to number 520 must be allocated also.

Mappings 1 and 2 can be eliminated by the observation that all the Processors, or EM mdoules, are crowded up into one part of the network, creating additional conflicts. This expectation is validated by the results of the CN simulator using these mappings. Mapping number 6 can be eliminated by the argument that other mappings give much better results for the frequently used p-ordered requests and p-q-ordered requests than they do for random requests. The best operation seen with the simulator suggests that mappings 3 and 4 should be used, one on either edge of the network. The best case actually simulated was processors using mapping 4 and EM modules using mapping 3 on the simple duplexed network. Call this the "baseline" mapping function.

With the above choice of mapping functions, the known frequently used requests are serviced with 100 percent or near-100 percent request success, and random cases are serviced. The simple duplexed network shows an average of 77 percent nonblocking in the network for random requests. The duplexed network with interlayer paths shows 87 percent nonblocking, and also represents the rate of request success seen on random p-ordered vectors. Success rate is 100 percent on requests with skip distance = 1. Success rate is near 100 percent on p-q-ordered vectors with skip distance = 1 within the pieces of vector.

For any mapping, there is a bad case, a permutation in which only 32 out of the 512 accesses requested are granted per EM cycle. It is desirable that this case be one that is not expected to occur (note that the bad case is not a catastrophe, it is merely a excess access time for one memory fetch). For mappings 3 and 4, the bad case is when the EM accesses desired are the bit-for-bit reversal of a sequential index. This case actually occurs once in one of the several ways to program a fast Fourier transform. Hence, investigation of mappings is expected to continue, including mapping No. 5, which moves the bad case to some more random permutation, and allows an interesting data exchange pattern for the SHIFCN instruction. However, the Fast Fourier transform, with one transform executed in parallel across the array, does not occur in any aero flow code or weather code evaluated. The FFT's in one weather code are executed serially, 512 FFT's in 512 processors, and do not contain the bit-for-bit reversed subscripted parallel fetch request.

It might be noted here that requests within the Connection Network refer to a CN port, not to a processor or EM module number. Therefore all mapping must be done external to the CN. Mapping of a processor number to a port has implications only for the wiring pattern that is used to let each processor know its own number. Off-line spare processors are inhibited from making requests to anything other than spare EM modules. This is done in the CN buffer logic of the processor. In addition, the CN buffer logic is responsible for mapping EM module number to CN output port. This implies that the provision for spare EM modules must be accommodated in the remapping from EM module number to CN port number, since the ports will not be physically relocated when a EM module is spared. In every CN buffer, four port numbers will be caught and replaced by substitute port numbers.

The suggested mapping from module number to port number is as follows:

First, put the most significant bit of EM module number at the least significant end of port number. This gives

$$\text{Port} = 2 \times \text{EMno} \qquad \text{for } 0 \leq \text{EMno} \leq 511$$

and would give

$$\text{Port} = 2 \times (\text{EMno MOD } 512) + 1 \text{ for} \qquad 512 \leq \text{EMno} \leq 520$$

This last formula is unacceptable as it puts all nine high-order EM modules into the first cabinet. Port numbers are rigidly assigned to cabinets, one quarter to the cabinet. The second formula may be modified as follows:

The simple duplexed network would be packaged as follows:

The 512-wide, 10-deep by 2 layer arrangement of nodes can be partitioned into 2-wide, 2-deep by 1 layer subsets in which every subset is like every other subset. A 1-bit-wide slice of this subset will fit on a 24-pin package as a single chip of moderate complexity, 24 x 256 x 5 x 2 such chips will implement the entire CN. This choice yields a total of 57,440 packages, all identical, all in 24-pin pa^kages. One observes that the use of the data lines is half duplex, not full duplex. If bidirectional data lines were used, a more complex chip, handling both directions of data on the same line, would still have the same pin count. Strobe and acknowledge, however, could not be combined. The result would be 13 packages per node, instead of 25, and the total chip count of 13 x 256 x 5 x 2 would be 39,280.

In a 40-pin package, the subset two nodes wide by two levels, and both layers deep could be accommodated, so that exactly half as many 40-pin packages would be used, or 28,720 packages without, and 19,640 packages with, bidirectional data lines. In any of the four cases, the control logic is replicated on each chip to reduce pin count. The next-to-largest of these various projections is used in Table 5.11 (which shows the CN Characteristics) to be conservative without complete pessimism.

A complete new chip design is not planned. Rather a gate array implementation is likely.

5.7.4.3 Packaging

Most of the CN is packaged within the EM cabinets, an identical subset of the CN being found in each of the four cabinets. Note that in Figure 5.11 the Omega network to the right of the second level of logic is exhibited as four identical Omega networks of one quarter the width. Thus, the 80 percent of the CN past the first two levels of logic is found in the EM cabinets.

(If the processor cabinets had enough room, and if processor numbers are assigned to cabinets in the correct pattern, the same partitioning of 80 percent of the CN to processor cabinets can also be achieved. An interesting puzzle is to devise those assignments of processor number to cabinet that allow all of the CN to be distributed among the processor and EM cabinets, with none of the CN assembled in any one central location, such as colocated with the coordinator and diagnostic controller.)

5.7.5 Design Rationale and Changes from Preliminary Study

The CN seen here represents a major change, and a major improvement, over the transposition network described in Ref. 1 and Ref. 2. The transposition network was at its most efficient only for 512-long vectors. For p-q-ordered vectors, the access time went up proportional to the number of pieces into which the vector had to be divided (five pieces for a 100 x 100 x 100 problem in the third, or hard direction). Conditional statements within DOALLs resulted in complex code in those processors that were trying not to execute anything; they had to pretend to be fetching and storing to EM like other processors in order to keep the synchronizations straight. Analysis in the compiler was therefore also complex.

With this connection network all these complexities disappear. Each processor is completely independent of any other processor. The language has been simplified, since restrictions on conditional statements and labels have been removed. The compiler has been simplified, since the conditional LOADEM and STOREM operations are no longer necessary, and a lot of address calculation that took place at compile time, or which had to be allowed for in the old control unit, is not needed with the present connection network.

The CN chip count represents another cost/performance optimization. For performance, a 516 x 528 crossbar switch, with no conflicts, and all accesses being granted on the first attempt at request, would be preferred. However, the crossbar switch has 275,088 crosspoints, whereas the CN has 40,960 crosspoints (four in each 2 x 2 node). This is just 15.2 percent as many crosspoints, reflecting a large ratio in hardware also. Despite this huge hardware saving, the CN has 100 percent success in fetching vectors in two of the three directions, and a success rate of 77 percent (or 87 percent if the alternate design is taken) in the third, or "hard" direction.

A second optimization of speed vs. hardware cost occurs in the path width of the CN. At 11 bits per frame, we need a path that is 12 signals wide, and takes five frame times to transfer a whole word. At 20 ns per frame this means that the delay due to serialization of the data word is an additional 80 ns, and dividing address into two characters adds 20 ns. The delay due to access time in EM is on the order of 200 ns (actually, it is yet to be determined, and the recent TI announcement of a 64k-bit RAM makes it appear that EM will be faster than projected in ref. 1). The delay due to round trip transfer time through the cables and logic of the CN is estimated at 120 ns. Thus the 100 ns added delay due to serialization of address and data is small compared to the 320 ns or so minimum possible access time. In Reference 1, a path width of 8 bits was chosen as adequate. This has been increased to 11 bits in order to present the request in fully parallel fashion; the request being the port number on the EM side of the CN.

A third optimization concerns the time it takes to compute the control of the CN. With unlimited amount of time for computing the setting, the Benes network can produce a set of paths such that all processors have their requests granted immediately, 100 percent of the time. The Benes has fewer components that the CN. Unfortunately, we are trying to make connections in nanoseconds. Opferman and Tsao-Wu, Ref. 7, show that the amount of computation required to find a non-blocking setting for a Benes network is on the order of N^2 computational steps, or Nlog N if an associative memory is available. This is certainly intolerable to compute at run time when the data is being fetched, and in our opinion is intolerable at compile time also. Furthermore, the computations impose synchronization onto the processors, since one new request, asynchronously added to existing set of latched up requests requires a whole new control computation. Hence, we have opted to search for suboptimum, but fast, control determinations, having each node making its own determination of its own setting on the basis of locally available information only, and ignoring the rest of the CN.

5.8 DATA BASE MEMORY (DBM)

Data Base Memory (DBM) is the window in the computational envelope of the FMP. All jobs to be run on the FMP are staged into DBM before running both program and data, all output from the FMP is staged through the DBM. DBM can be used by the programmer to back up EM for those problems whose data base is larger than EM. Control of the data base memory is from a DBM controller, (described in the next section), which accepts commands both from the coordinator for transfers between DBM and EM, and from the host for transfers between DBM and the file system.

The design chosen is a CCD memory based on 256k-bit chips which are projected to be available in the 1980 period. Alternatives are discussed in the section on design rationale. Figure 5.14 shows the general organization of the Data Base Memory. The details of this organization are discussed in more detail in the next subsection.

The primary use of the DBM is as a staging area for jobs going to and coming from the FMP. It can also be used as a source for overlaying data and program into the FMP for large jobs. It is possible to transfer less than a full block, but all transfers must begin at a block boundary.

5.8.1 General Storage Characteristics

The general organization of the DBM is a controller together with a general CCD chip array, used as the primary storage area, a number of block-sized buffers, used for speed matching on data transfer interfaces, and error controls.

The design described here is based on the assumption that 256k-bit CCD chips will be arranged in the form of 128 shift registers of 2,048 bits each. It is also assumed that the shift rate of the devices will be 2.5 MHz.

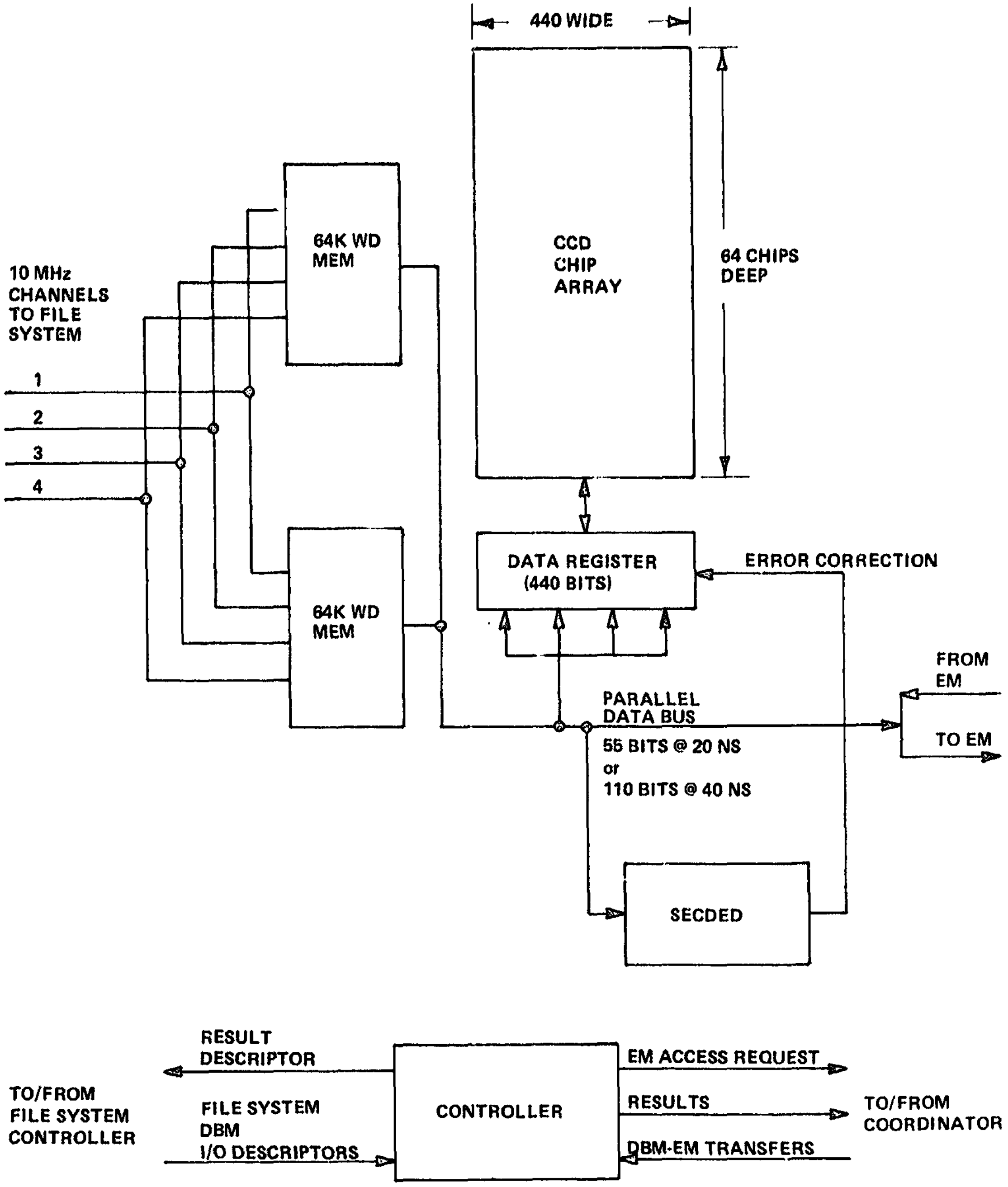


Figure 5.14 Data Base Memory Block Diagram

DBM files come from and are moved to the SPS file management system. Over 99 percent of this traffic is expected to be simple moves from DBM to disk pack. Twenty M-bits/sec on this path yields large safety factors over the traffic actually required, even after making allowance for the fact that short jobs will be bunched in prime time. The four channels provide 20 Mbits/sec with 5 MHz disk transfer rates and 40 mbits/sec with the 10 MHz rates available in recently announced products.

No buffering is needed on the EM side beyond the one-word buffers in each EM module. These one-word buffers, and the 240 ns cycle time of the EM modules, together ensure that the DBM controller never need wait for an EM response.

DBM-EM transfers have priority over CN servicing in the EM controls. However, there is little interference with processor accesses to EM. For example, when transferring from EM to DBM, one EM cycle loads 512 of the per-EM-module one-word buffers, and then waits for 12.8 microseconds before another EM cycle is required for the DBM transfer path.

Table 5.12 summarizes the characteristics of the DBM.

5.8.2 Soft Error Control

As a background job, the DBM controller periodically initiates an access for the purpose of reading the contents of a block and rewriting that same block with all detectable errors corrected, since errors are spontaneously created in CCD memories at a low rate. These errors are apparently caused by background radiation effects on the CCD chips, discharging the little capacitors by temporarily ionizing the oxide. The rate of periodically initiating access can rationally be determined only after getting the vendor's specification. Preliminary Fairchild data indicates that one should scrub through the entire DBM every seven minutes.

At that rate, this background access would be initiated for a new block every 55 ms. Error scrubbing accesses will not queue. If one is delayed beyond its 55 ms time slot, then the whole cycle will slip to 7 minutes plus 55 ms.

5.8.3 Design Rationale and Changes from Preliminary Study

The major change from Ref. 1 & Ref 2 was the reorganization of the internal structure of the DBM CCD storage array to allow higher bandwidths to and from the EM modules and to and from the file storage system.

Table 5-12. Data Base Memory (DBM) Characteristics

Number in System: 1

Function

To serve as staging area for FMP jobs; to serve as memory extension for FMP jobs that will not fit into EM and PMs.

Mode of Operation

Stores in blocks only. Has access to support processor file system on the one side, and to the EM on the other side. DBM areas may be used by the file system.

Storage Capacities

134,217,728	words		131072	words
400	ns cycle shift rate	plus	280ns	cycle
256k-bit CCD	technology		64k-bit dynamic MOS	RAM

Connectivities

To/From	Function or Name	No. Signals	Timing	Comments
Support Processor	Data channel	TBD	40 megabits/sec	
EM	Data channel	TBD	40 megabits/sec.	
DBM controller	Control	TBD	TBD	

Reliability/Repairability/Trustworthiness

All words covered by error correcting code.

Errors are periodically removed by reading, doing error correction, and rewriting.

Sections of DBM can be locked out by software, so that function can be provided by the remaining working portions.

Physical

Projected chip count:	29160 (28160 memory + 1000 control and misc.)
Size:	176 boards of 166 chips each
Power:	10kw operating, 1 kw standby

Two major data transfer paths exist, one to the EM and one to the disks of the File System. The desired transfer rate to and from the Extended Memory (EM) is 40 M words/sec. To accomplish this, the DBM storage area will be organized 440 chips wide for parallel emission of eight 55 bit words by 64 chips deep.

The natural block size with 2,048 bits in each shift register, the eight words in parallel delivering a block of 16,384 words, is adopted. There are 8k blocks for a total of 134,217,728 words. Error correction is a SECDED, probably the modified Hamming-plus-parity implemented by Motorola's 10,163 chip.

Since the array of CCD chips is 64 x 440, the DBM is constructed in a number of physical modules; each one 8 x 440 chips. Cards are 20 bits wide, 22 cards per module. The repair philosophy is to pull and replace individual cards, and the degraded mode of operation would be to run with one or more modules missing, and the operating system would have to be told to avoid assigning any data to that space.

There are eight block-sized buffers, which stand between the CCD storage and the host interface, in order to reduce the interference with DBM-EM transfers produced by simultaneous DMB-file system transfers. They also serve as timing buffers to the file system's disk packs, eliminating the need for block sized buffers elsewhere in the data channel. These buffers are contained in two memory modules constructed of the 64k-bit dynamic RAM chips used in the EM modules.

After the transfer of a block to or from the CCD store, the shift registers rest at the starting position until shifting is required by the refresh requirements, or until the CCD store is again addressed, whichever occurs first. The store will be periodically addressed for error control reasons, see 5.8.2 below. Therefore, whenever there are several requests for transfer pending at once, or when they occur with sufficient frequency, the access time is essentially zero to the first word of the block. For transfers arriving at random times, far enough apart in time so as not to interfere, the average access time is given by:

$$T_{av} = 1/2\ (T_b^2/T_r)$$

where T_b is the transfer time of a single block (0.82 ms) and T_r is the time between refreshes. T_r will be in the specification of the device, and is expected to lie between 1 ms and 10 ms. Therefore, the average access time for random data at low usage, to the first word of the block, has an upper bound which is expected to lie between 0.67 ms and 0.067 ms. As traffic increases, the access time is mostly due to interference between competing accesses, while the contribution due to delay in the memory goes to zero.

The DBM design seen here is the result of comparing a number of different devices. The other possibilities include:

Magnetic bubbles. Rejected because the bandwidths would require the reading and writing of thousands of bubble chips in parallel, and also because of the inherently greater complexity of bubble systems. Each bubble chip requires several support chips such as drivers, sense amplifier, etc.

Rotating magnetic storage. With enough heads in parallel, and a fast enough rotation rate, magnetic rotating storage can supply the DBM requirements. However, the programming becomes complicated by considerations of data organization and access time. Blocks want to be very large to amortize the large access times over the high transfer rate requirements. For example, to get full transfer rate from a 10 ms disk requires blocks that cover the entire track, or blocks 10 ms long. If full transfer rate is 40 million words per second, the blocks are almost half a million words each. For some purposes this is a severe restriction.

64k-bit CCD's. 64k-bit dyanmic RAMs will be preferred by almost all equipment designers over the shift register CCDs. With the recent appearance on the market of dynamic RAMs, it is to be expected that the 64k-bit CCDs will disappear.

64k-bit dynamic RAMs. These would make an acceptable back-up DBM design. With the increased cost would come a measure of increased performance and freedom from the hardware-defined block structure.

One last possibility should be mentioned for the future. The same device fabrication, tooling and lithography techniques which are expected to allow the development of 256k-bit CCD chips can be expected to result in 256k-bit dynamic MOS RAM chips within a year after the CCD chips are available. Enough advantages may accrue from the use of these chips in terms of increased performance and freedom from a fixed, hardware-defined block structure that these RAM chips would be used in the DBM design.

5.9 DATA BASE MEMORY (DBM) CONTROLLER

The DBM controller interfaces two environments, the FMP internal environment and the file system, since the DBM is the window in the computational envelope. DBM allocation is under the control of the file management function of the support processor. The DBM controller has a table of that allocation, which allows the DBM controller to convert names of files into DBM addresses. When the file has been opened by an FMP program, it is frozen as far as allocation is concerned, and must remain resident in DBM until either closed or abandoned. For open files, the DBM controller accepts descriptors from the coordinator which call for transfers between DBM and EM. These descriptors contain absolute EM addresses, but file names and record numbers for the DBM contents.

The DBM controller therefore has two main elements. First, a programmable controller and second, hardwired channel logic to accommodate the data transfers.

The software response time of the DBM controller shall be less than 100 microseconds to Coordinator requests. This demands that the conversion from file name to address be simple table lookup, and also that the response of the DBM controller to Coordinator commands be essentially instantaneous; i.e., either the normal state of the DBM controller is waiting for an Coordinator command, or Coordinator commands have a priority interrupt within the DBM controller.

The actual channel controls for transferring a block of data are independent of the controller that does the table lookup and handles the exception conditions. Address counters, limit registers, and limit comparators are separately implemented, not programmed, because of the high transfer rates involved. There are five such channel controls, one per host channel, and one for the EM interface. The entire bandwidth of the EM channel is devoted to whatever single transfer is being effected at a given time.

Operation is as follows. When an FMP task has been requested, the support processor passes to the file manager the names of the files needed to start that task. In some cases existing files are copied into newly named files for the task. When all files have been moved into DBM, the task starts in the FMP. When the task in the FMP opens any of these files, the allocation will be frozen within DBM. It is expected that "typical" task execution will start by opening all necessary files. During the running of a FMP task, other file operations may be requested by the user program on the FMP, such as creating new files and closing files.

EM space is allocated either at compile time or dynamically during the run. In either case, EM addresses are known to the user program. DBM space, on the other hand, is allocated by the file manager, which gives a map of DBM space to the DBM controller. In asking the DBM controller to pass a certain amount of data from DBM to EM, the Coordinator, as part of the user program, issues a descriptor to the DBM controller which contains the name of the DBM area, the absolute address of the EM area, and the size. The DBM controller changes the name to an address in DBM. If that name does not correspond to an address in DBM, an interrupt goes back to the Coordinator, together with a result descriptor describing the status of the failed attempt.

Not all files will wait to the end of an FMP run to be unloaded. For example, the number of snapshot dumps required may be data dependent, so we may wish to create a new file for each one, and certainly we shall want to close the file containing a snapshot dump so that the file manager can unload it from DBM. When the FMP task terminates normally, all files that should be saved will have been closed by the FMP program. The strategy that supports restart has not been detailed.

The file manager may choose to leave read-only files in place in DBM, on the chance that the same read-only file may be asked for by more than one task.

5.10 DIAGNOSTIC CONTROLLER (DC)

The diagnostic controller provides a channel whereby the Support Processor or logician at the maintenance panel, can impose diagnostics upon the FMP. The strategy behind the diagnostics is that any portion of the FMP can be set to some arbitrary state, and then caused to execute some fixed function or execute for some fixed amount of time, and that the resulting state can be observed. The Diagnostic Controller's access is direct to the coordinator. Access to the processors is indirect, in that the coordinator has direct access to the processors, and the diagnostic controller manipulates the coordinator. Chapter 6, in discussing the diagnostic programming, discusses these relationships in more detail.

The output of the diagnostic controller is a set of commands to the Coordinator and the DBM controller. These commands are yet to be determined in detail but they are of the general type of the following examples:

LOAD REGISTER R

READ REGISTER R

EXECUTE the instruction presently residing in the program register and then halt

HALT all operations, possibly by suspending the clock

INITIALIZE a predetermined subset of registers to a predetermined state (probably all zeroes)

The input of the diagnostic controller comes from either the support processor or from a maintenance terminal. The input can cause the diagnostic controller to emit single commands, or to emit a series of preprogrammed commands. In order to emit meaningful sequences, and to collect the results of those sequences, it is envisioned that the diagnostic controller contains a mini or microprocessor. A test control language will be provided.

The diagnostic controller is a debugging aid, a system integration aid, and is used only as a fall-back mode of operation during maintenance. System initialization, upon power up or other cold start of the FMP may also use some of the DC capabilities for initialization of selected registers and loading of bootstraps.

5.11 POWER CONSIDERATIONS

The power supply design for the FMP will consider the following:

- A small number of centralized power conditioning modules that accept raw AC power from the mains.
- Switching regulators for efficiency
- Defense against faults in the incoming power
- Defense against faults in the FMP
- Noise reducing grounding methods.
- Non-volatility of DBM contents

A power supply system that takes all these features into account is described in this section.

Total power for the FMP is estimated at 250 kw, based on an average of 0.8w for each of the 200,000 circuit packages and on 65 percent efficiency in the power supply system.

5.11.1 AC Modules

The block diagram of the power supply system is shown in Figure 5.15. Raw AC power is supplied to six places (labelled "1" in the figure), namely:

- The maintenance panel, which also contains the central power system control
- The DBM power system
- Four identical AC modules.

Each of the six places to which raw AC input is supplied contains an AC voltage monitor. The design intent is to shut the machine down for high line or low line that is potentially damaging to the machine, and to send a one-bit message to the maintenance panel and the support processor for low-line conditions that may cause garbling of data.

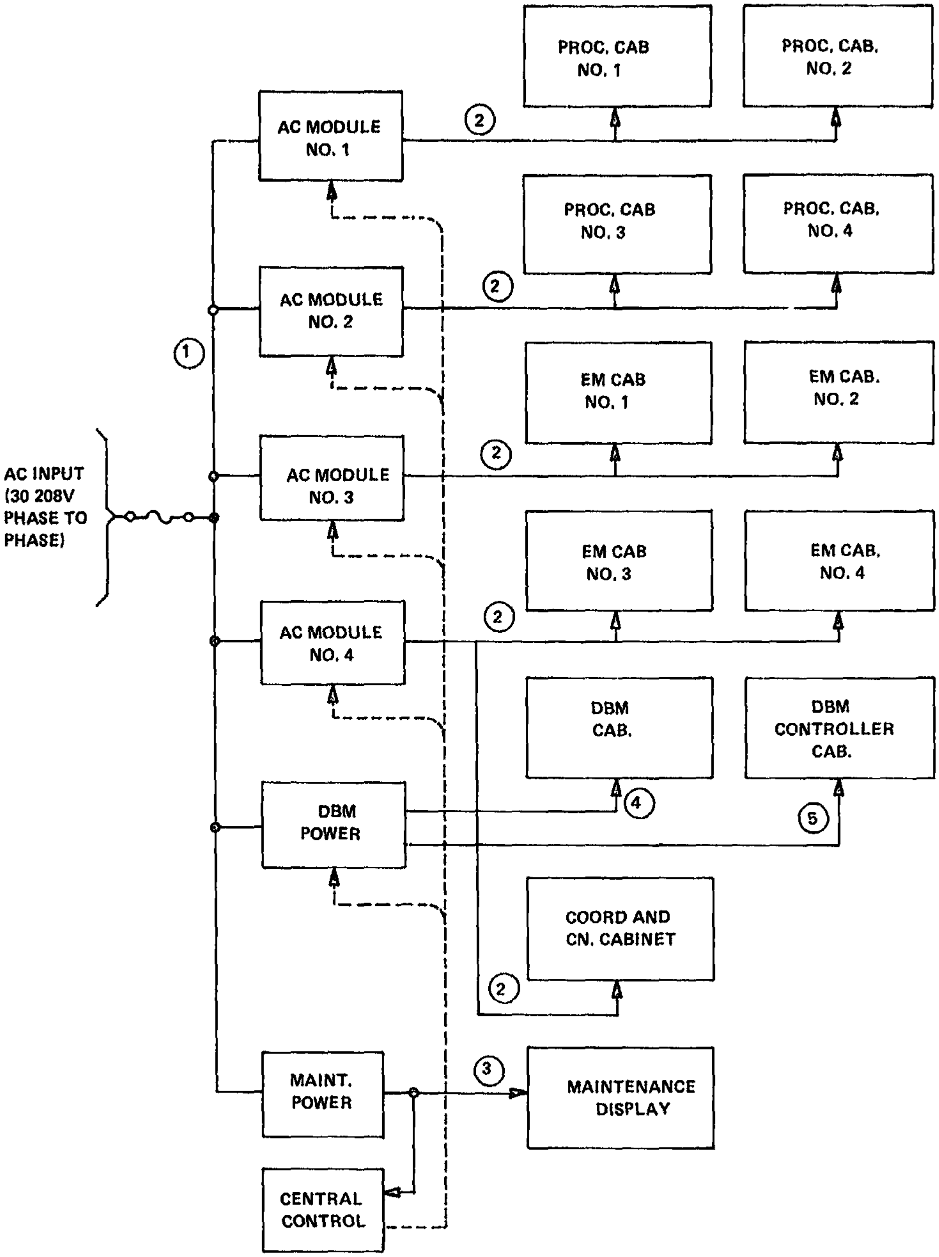


Figure 5.15 FMP Power System

Out of the maintenance panel's power system comes various DC voltages (labelled "3") for the maintenance display and the central power control. These include +5 at 20 amps for logic and LED drivers, +12 at 0.5a, and -12 at 1a, plus a switched 1Ø 115v AC for the CRT which has its own self- contained supply.

The AC modules receive "turn-on", "turn-off" signals from the central power system control, and send "fault" signals back to the central control. Each AC module supplies up to 250 a at 158 volts DC (labelled "2") for the switching regulators attached to it. The AC module also combines the fault signals from its attached power supplies into a "cabinet fault" line, and shuts down for any perceived faults. It contains line filters. In complexity, it is similar to the AC module of the B-6700. Power efficiency is between 96 percent and 99 percent.

The requirement that the FMP power system ride through undervoltage transients, and tolerate voltage spikes from the mains, influences the design of the power control modules. A transformerless rectifier in the central power control module, with switching regulators distributed around the FMP, is a system inherently tolerant to undervoltage sags and transients, and impervious to spikes. In addition, the switching transients of the regulators tend to be soaked up by the filter capacitors at the control module's rectifier. Whether either a motor-generator set or battery backup is needed, would depend on actual line characteristics at Ames. If the line characteristics are known before the design is carried out, the system can be designed so that the expense and inefficiency of the motor-generator set can be eliminated.

The DBM power unit provides 30 amps at 158v for the DBM controller logic supplies (labelled "5"), and a separate line ("4") for 35 amps at 158v for the memory chips of the DBM. There is also a stand-by mode in which 8 amps at 158v is supplied to the memory cabinet from batteries during power outages of up to 15 minutes. (15 minutes is selected on the basis that that is long enough to save all of DBM on a single disk pack through a single disk channel. The resulting 316 watt-hour requirement can possibly be supplied by an ordinary sealed lead-acid battery.) The DBM power unit also contains logic to handle fault situations, and the same line filters that are in the AC modules.

5.11.2 Other Power Supplies

Besides the seven units described briefly above, there are within the cabinets the following:

> 516 processor power supplies each contained physically within its own processor. Each one is a 70 percent or better efficient switching regulator supplying 40 amps at 5v, and 0.5 amps at -12v.

44 supplies at 5v, 160 amps. Except for the lower power level, these are similar to power supplies built by Burroughs for PEPE. There are eight in each EM cabinet, and four in the Coordinator-connection-network cabinet, and eight each in the DBM controller and the DBM memory cabinet. These are switching supplies at 75 percent efficiency.

6 supplies at 12v, 2 also working from the 158v out of the AC modules. These are used in Coordinator, DBM controller, and EM cabinets for +12v and -12v for various purposes.

Each of the supplies above contains remote voltage sensing, appropriate over-current sensing, current limiting or fold-back, over-voltage and under-voltage sensing.

5.11.3 Grounding Considerations

Grounding is an area of design in which even qualified electrical and electronic engineers sometimes propagate false myths. Some of the confusion is due to failing to distinguish between various functions of the conductors called "ground", which in any given case may or may not be at the same voltage, and may or may not be the same conductor. Some functions are:

- Neutral in an AC distribution system.
- Earth, or an external zero voltage reference.
- Safety ground, enclosing the equipment in order to prevent shock hazards.
- Shields, enclosing electrically active circuits in order to prevent transmission or reception of interfering electromagnetic signals.
- Reference voltage. The signal voltages in the equipment are measured with respect to the reference voltage. Reference voltage is often called "logic ground".
- Return paths for currents.

Some details resulting from these considerations are:

The ground return from backplane to power supply is never used as part of the path that connects one backplane ground to another backplane ground.

Every module has its logic ground tied to chassis, so that there will be no floating grounds when the modules are tested as stand-alone modules. These ties may be resistors if unwanted ground currents would be set up by direct connections.

Every single-ended signal which traverses from the area of one backplane to another is accompanied by a wire conductor for the return current of that signal, and the return conductor is connected to reference voltage at all points at which the signal is either generated or used.

5.12 CIRCUIT AND PACKAGING TECHNOLOGY

5.12.1 IMPLEMENTATION TECHNOLOGY UPDATE

5.12.1.1 SUMMARY

The semiconductor industry has continued to improve both device density and performance since the previous implementation technology submission. Smaller device geometries have been achieved in production with the application of Electron Beam processing techniques. The initial utilization of the E Beam tool has been in the mask generation area where smaller geometry and more rapidly generated masks have been produced. This advantage coupled with projection exposure of wafers as compared to the use of contact masks and plasma or dry etching has enabled higher precision devices to be generated in a production environment. Line widths are predicted to diminish to under one micron. The priority of devices to which the new processing technology is applied has been first in the memory area and second in the microprocessor area. Microprocessor availability in the 16 bit logic density area has increased from just a few, to a selection of a half dozen or so with performance estimated to be in the PDP 11/45 class or greater. Direct address capability has expanded from a 16 bit limitation of 65K to a 16 megabyte level.

During the initial manufacture of large (65K) CCD Memories a higher than expected random failure rate was observed. The failure mechanism was later identified as being caused by alpha particles which modified the charge being transported, thus destroying the information stored in the memory. Solutions were developed for greatly lowering this failure rate by reducing or eliminating the major source of alpha particles and providing a shield layer on the chip. The major source of alpha particles was reported to be in material used to package the CCD chips.

In the very high speed area, gate arrays were becoming an interesting alternative for achievement of dense logic implementation. The economy of using gate arrays is dependent on quantity of the devices required for the systems to be produced. Basic arrays exist at Motorola and Fairchild in the nigh speed ECL area. A gate array exists in the proprietary Burroughs CML circuit family (BCML).

Memories anticipated to be available in the 1979/1980 time frame have already been delivered on a sample basis to selected manufacturers. These include the 65K dynamic RAMS and 16K static RAMS. CCD 65K bit memory circuits have been delivered for incorporation into CCD memory modules. Work has begun in definition of 256K bit CCD and 256K bit RAM with expectations of availability in the 1980/81 time frame.

Gallium arsenide efforts in the high speed sub-nanosecond logic area have continued at a number of manufacturers' facilities. Specification circuit configurations for the gallium arsenide MESFETS are being reviewed along with development of production procedures to manufacture these devices. Speed power estimates vary from 100 picosecond propagation delay range to about 300 pico-seconds with power dissipations varying from about .08 - .3 milliwatts per gate.

Effort is being expended in utilization of the CMOS SOS type of circuit implementation. At the Solid State Circuits Conference in 1978 the general discussion seemed to indicate that CMOS SOS gate density problems would be somewhat overcome with the tighter line width. The attractiveness of the CMOS SOS circuit for NASF applications is the projected lower power dissipation of gates not memory in the CMOS LSI circuit.

The specific implementation approach to be selected for the NASF FMP must be postponed as long as possible due to the dynamic developments occurring in the semiconductor technology area. At present, the bipolar ECM or CML candidates look the most promising from a performance/risk point of view. Although developments in higher speed gallium arsenide devices are progressing, the risk involved in such an infant technology does not seem to warrant the advantages gained in higher speed.

During the current contract some additional information in both ECL arrays and BCML circuits has been reviewed. Some characterisitics of the ECL voltage compensated arrays as well as information on BCML are included in the following.

5.12.1.2 ECL Arrays

Motorola has announced the MECL 10K Macrocell Array that consists of 48 macro cells with 32 interface circuits and 28 output circuits. All cells can have series gating. Structured cells are predefined into logic elements. Interconnect channels are 12 x 12 for 9 macro cells. The macro cells consist of functional circuits which are interconnected to produce larger portions of logic. The total number of channels available for interconnection is approximately 108 x 94. Internal gate delays anticipated are approximately 900 picoseconds. A maximum of 1.3 nanoseconds is expected. The maximum power dissipation if all cells are used is anticipated to be approximately 4 watts. An equivalent gate complexity up to 750 gates can be realized on the array. The average gate power is projected to be 5.3 milliwatts. High drive outputs can be achieved at 8 of the interfaces. A capability of driving a 25 ohm line exists at these points. The die size is approximately 210 x 230 mils. It is anticipated that the semiconductor chips will be placed in a 68 pin leadless package. Some proposed connectors exist for the 68 pin package.

5.12.1.3 BCML

The BCML-2 (Burroughs' Current Mode Logic) family is a Burroughs developed circuit family intended for use in Burroughs' systems. The family consists of SSI, MSI and LSI circuit types, gate arrays, register files, ROM, PROM, EAROM, and RAM. All devices have on-chip voltage and temperature compensation. This assures constant logic levels and constant threshold, hence constant noise margins. It also assures constant propagation delay over the entire operating voltage and temperature ranges. Two types of power supply are specified. Logic circuits use -2.7V ± 30% or -4.8V ± 25%, while memories use only -4.8V ± 25%. All devices have on-chp output resistors which serve to source-terminate 50 ohm transmission lines. On-chip Test and Diagnostic (T&D) monitors are used to detect opens and/or shorts of any logic net and loss of power supply voltages to any circuit chip.

The salient features of the BCML family are given in Table 5.13.

5.12.2 Packaging

5.12.2.1 General

Final choices of packaging technology can be deferred until the system design is nearly complete. However, for performance and reliability analysis, scheduling and cost, preliminary selections must be made. Basic high speed (ECL) packaging technology has been developed over the past decade that provides high performance and reliability at quite reasonable cost. The manufacturing tools, and assembly and test procedures, are all fully developed. This technology includes a family of specified and use qualified components and hardware. Advances in this area are under continual study. The current status and performance characteristics of this technology are discussed in the following sections.

5.12.2.2 Printed Circuit Assemblies

Multi-layered printed circuit assemblies provide a straightforward approach for the packaging of standard commercial dual-in-line ECL circuits. The six layer 16 inches by 18.5 inches assembly used by Burroughs on the PEPE and latter programs provided a capability of mounting 300 sixteen pin dual-in-line packages or 280 sixteen pin and 10 twenty four pin packages. The board consists of six copper layers permitting two signal layers, two voltage layers and two ground layers. (Figure 5.16). Each signal layer references a ground plane providing two layers of 50-ohm microstrip. Proper tolerance is maintained over line width, dielectric spacing and dielectric constant.

OF 7

26072

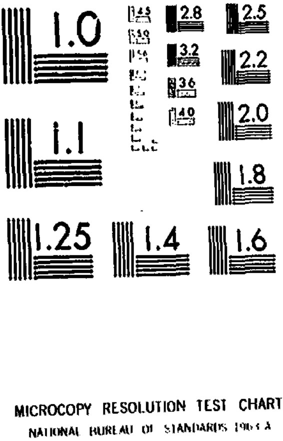


Table 5-13

FEATURES OF BURROUGHS CML CIRCUIT FAMILY

- High speed - 0.7ns per raw gate
- Low power delay product 4 p per internal gate for LSI, 6pj for MSI and 8pj for SSI
- Fully compensated logic levels and threshold - Noise margins and propagation delay remain constant over operating temperature and voltage ranges
- Source terminated interconnection - On-chip output resistors properly terminate 50 ohm transmission lines
- Complementary, simultaneous outputs - Simplifies design, minimizes cross-talk
- Small logic swing of 440mV - Provides higher speed at lower power, less noise generation
- Constant supply current - Reduces noise, fewer or smaller decoupling capacitors, no dependence on operating frequency
- Advanced Circuit Technique - On-chip use of series-gating, gate stacking, e^{2}l (Emitter-Function-Logic), Schottky-Diode gating, wired-OR and -AND, staggered thresholds, etc. provide best functional density at lowest power level
- Test & Diagnostic pin (T&D) - Facilitate testing of individual packages and isolate faulty packages in operating environment
- 50 pad package - Increases logic function capability per device and reduces package count
- Multi-chip package - Increases packaging density, improves performance and reduces system cost
- SSI to LSI densities - 50 pad package has capacity to accomodate gate densities in the order ot 1000 gates per chip

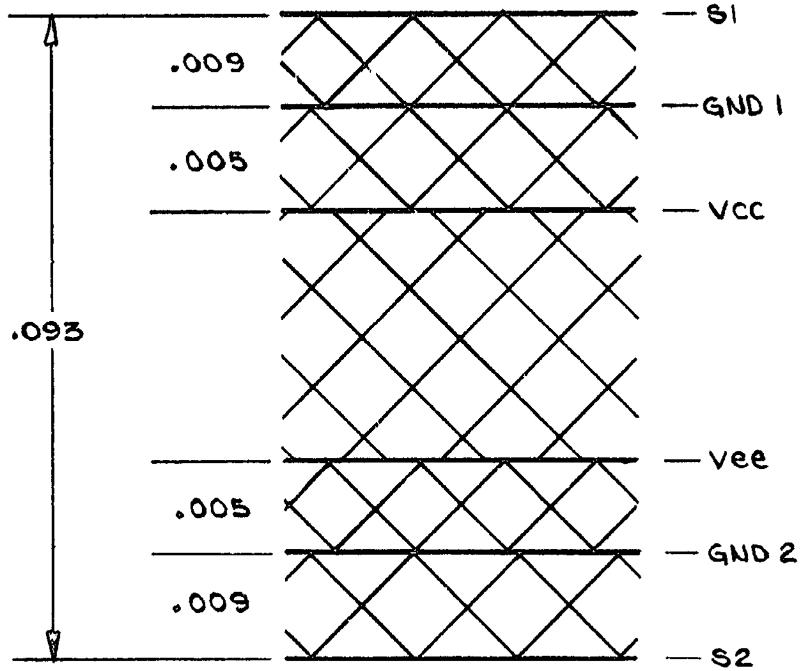


Figure 5.16 Multilayered Printed Circuit Boarđ for ECL

Figure 5.17 illustrates the component side of a fully populated board assembly of the PEPE type. The aluminum electrolytic capacitors along the top and bottom edges of the assembly are utilized to bypass voltage noise for frequencies below 1 MHz. Two other levels of bypassing control voltage noise, the interlayer capacity of the board for frequencies above 20 MHz and ceramic capacitors (contained in the terminator resistor packages) for frequencies between 1 and 20 MHz. The board assembly is mounted in a diecast aluminum alloy frame. Camming type handles are mounted on the front of the frame to provide the insertion force to mate the four 100-pin, I/O connectors. The I/O connectors incorporate a unique socket design that results in low insertion force and low contact resistance. A single 100-pin connector nominally requires around a 13-pound insertion force. Four connectors would result in an insertion force of approximately 52 pounds. The handles are also used to lock the board in place. Each circuit card module assembly is supported by shear/locating pins in front and rear.

This assembly can accomodate cam action zero insertion force connectors which in turn can accomodate the edge connector of belted cable paddleboard assemblies.

The assembly may be adapted to mount dual-in-line sockets. Each socket is soldered to the board to pick up the printed circuit signal trace. In addition, wire-wrap tail on the socket provides for two levels of wire-wrap. (Figure 5.18).

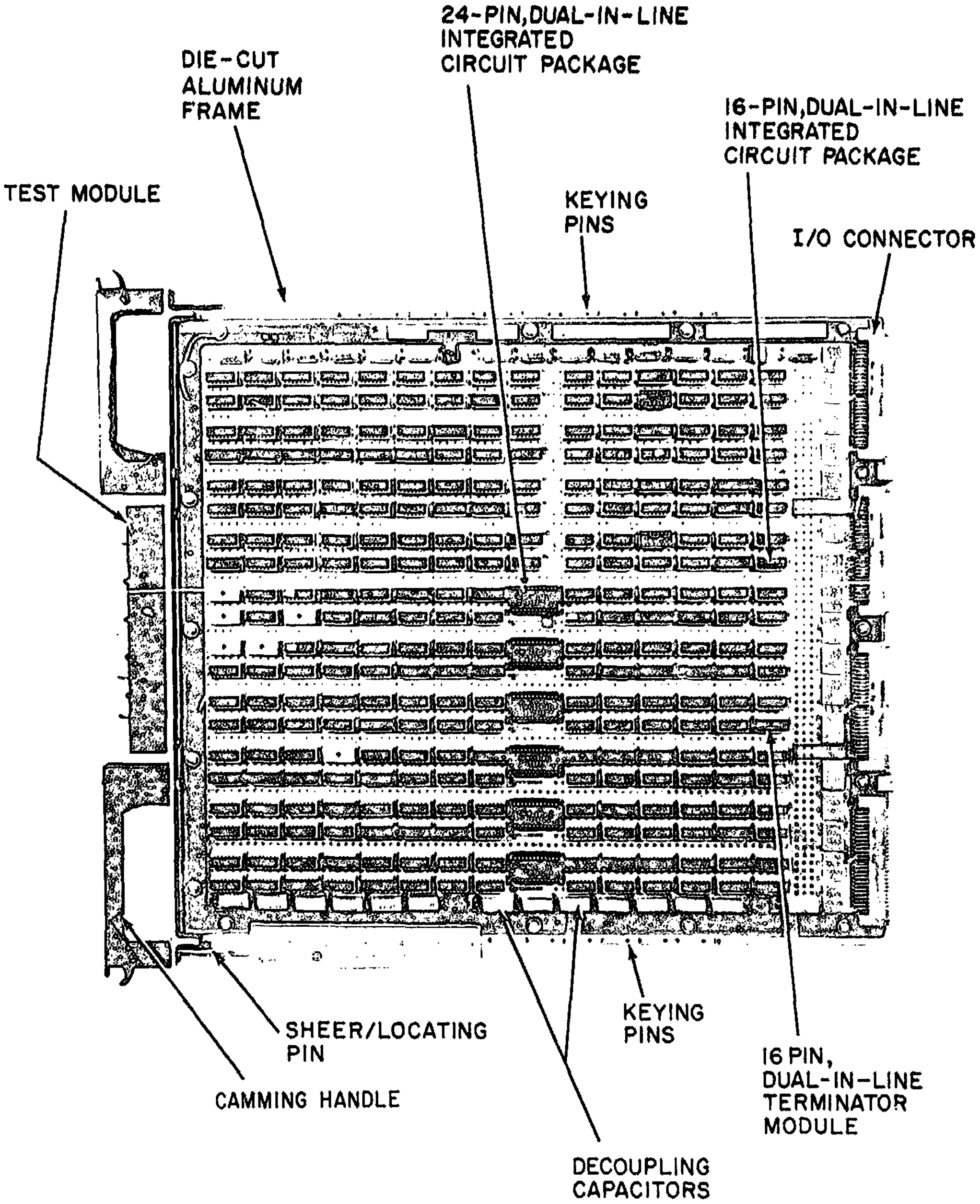


Figure 5.17 Component Side of Fully Populated Printed Circuit Board Assembly

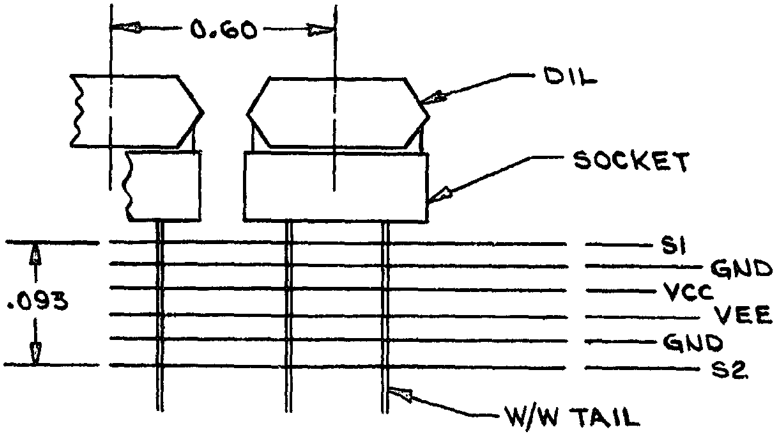


Figure 5.18 Multilayered Printed Circuit Assembly with Dual In-Line Devices and Sockets

5.12.2.3 Interconnections

Two primary techniques for the interconnection of the basic assemblies (processors, memory modules, etc.) help guarantee feasibility of the FMP. Wherever possible, interconnections will be made with paddle board and belted cable assemblies. Belted transmission cable with up to 70 conductors, (AWG 28 or 30, silverplated) on 0.025 inch centers suitable for the FMP signal levels and frequencies is readily available. Techniques for semi-automatic assembly of these cables to paddleboards with edge connectors are fully developed and provide the economical reliable interconnections.

Where the use of belted transmission line is impractical, interconnections are achieved with subminiature 50-ohm coaxial wire.

The coax consists of No. 32 AWG, silverplated-drain or ground conductor; a wrapped tape shield of aluminized mylar; and an outer jacket of laminated mylar. The maximum overall size of the cable is 0.033 inch x 0.043 inch. The drain conductor is compressed between the aluminum side of the shield and the primary insulation such that the drain wire is in contact with the shield along the full length of the cable. Both conductors (ground and signal) are wrapped simultaneously on adjacent pins (on 0.100 inch centers) using a dual-bit wire-wrap gun as shown in Figure 5.19.

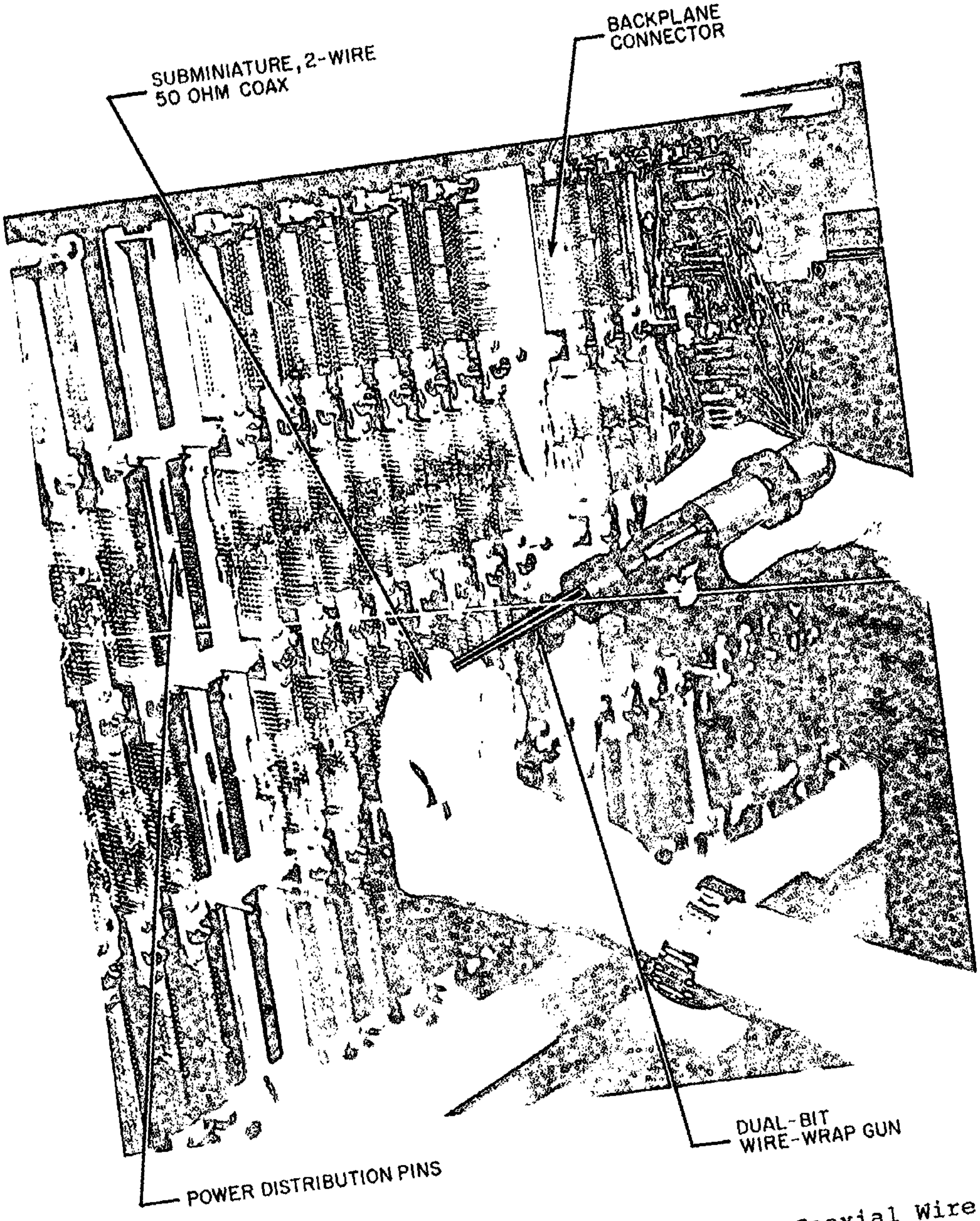


Figure 5.19 Backplane with Subminiature Coaxial Wire

5.12.2.4 Backplanes

Backplanes for power distribution are not required for the processors as they have individual power supplies. However, in the case of the coordinator and connection network it may be more desirable to have a centralized power source which for high speed ECL technology would normally require a laminated backplane assembly.

This assembly consists of three layers of epoxy-coated copper. It serves the dual functions of: 1) mounting the female half of the circuit card module assembly connectors, and 2) efficiently distributing power to each circuit card module assembly by providing a low impedance power distribution network.

Power is distributed to each circuit card module assembly via pins soldered to the individual backplane layers as shown in Figure 5.19. A wire wrap connection is then implemented between the backplane and associated connector pins. Multilayer, laminated backplanes are required to minimize backplane impedance (primarily inductive). A low inductance offers a low impedance to surge currents, guarantees power supply stability, and gives fast power supply response time.

5.12.2.5 Cabinet Frame Assembly and Doors

At this time, it is anticipated that the FMP equipment would be housed in cabinets similar to those used on other advanced processor systems currently being made by Burroughs. A description of these assemblies is provided in the following.

The cabinet frame is constructed of 0.120-inch-thick rectangular steel tubing welded into a unitized frame. In certain areas the rectangular steel tubing is increased in thickness to 0.180 inch for strength considerations. The overall dimensions of the basic weldment are typically 81 inches high by up to 72 inches wide by at least 30 inches deep. Maximum envelope dimensions of the cabinet assembly, including all doors and end panels are 81 inches high by 100 inches wide by 32 inches deep.

Bi-fold doors are utilized on the front and rear faces of the cabinet. Each bi-fold assembly (there are four) is composed of two 0.75-inch-thick aluminum honeycomb panels connected by a unique, extruded, continuous hinge. The stationary panel on the right-hand end of the cabinet is constructed of 0.062-inch-thick formed aluminum. A hinged split door configuration is utilized on the end of the cabinet to provide access to the rear. The overall thickness of the split door is 2.13 inches. Each door section is comprised of 1-inch-thick aluminum honeycomb and 0.0062 inch-thick formed aluminum.

5.12.2.6 BCML Packaging

5.12.2.6.1 General. A complete family of packing hardware has been specifically developed for the Burroughs CML circuit family. This advanced hardware family incorporates features to accomodate subnanosecond high density circuits of greater than 1000 gates each for use in commercial state of the art computer systems. The family includes low cost modular liquid cooling and power distribution systems. The design concepts placed high consideration on manufacturability and ease of assembly debugging and maintenance.

The BCML packaging system provides hardware that can be used across the Burroughs product lines of the computer systems and for other special applications. The basic philosophy of this packaging system was to partition the second level packaging to be compatible with functional logic partitioning. By packaging a system function within an integral unit, the number of I/O's between units is minimized and critical functions can usually be restricted within this unit.

5.12.2.6.2 Circuit Packaging. The basic partitioning size selected for the BCML system is a printed wiring board 14" x 21". This unit is referred to as an island and can accommodate 10,000 logic gates with the current normal mix of SSI, MSI, and LSI BCML parts. With the increased usage of LSI, and VLSI circuits island gate capacity will be enhanced.

Another basic goal of the BCML packaging system is to provide for ease of field maintainability. The following are some of the packaging as well as circuit features that facilitate serviceability:

1. Plug-in logic packages.

2 A probe system to allow simultaneous contact of all logic package pins.

3. Provision for in-place testing of circuits.

4. No external components in wiring nets.

5. Test and Diagnostic pin (T&D) incorporated on logic packages.

The first level of packaging was selected to accommodate a circuit family aimed at high gate densities. Two package sizes, 25 pins and 51 pins, are utilized. Multi-chip versions of the 51 pin package can accommodate up to 3 I.C. chips.

The packages themselves are a leadless hermetic ceramic construction. The package has gold plated contacts on 50 mil centers in two rows on its edges. The package also has an integral metal heat sink plate. This member conducts heat generated by the circuits to a liquid cooled frame and also serves as a low inductance ground connection.

Two 25 pad packages or one 51 pad package mates with a 50 pin connector. This connector will also accept two (24) signal I/O cables or one 50 signal I/O cable. The interfaces of all the pressure contact systems are gold plated for high reliability. Two types of connectors are available; this first type is soldered to the interconnecting printed circuit board and has a wire wrappable tail while the second makes a pressure contact to a gold pad on the printed circuit board. These two styles of connectors provide flexibility in design of the island interconnect media.

There are 108 connectors mounted on the logic island as well as a liquid cooled frame. The cold frame also serves as a low resistance ground return path. Interconnection of circuits on an island is accomplished by a combination of P.C. lines and open wire. A multi-layer board with internal voltage and ground planes and two external signal layers with 50 ohm lines are used for the bulk of the interconnections. The shorter lines can be implemented by automatic Gardner-Denver wiring with no performance penalty. An all wired utility board system utilizing controlled impedance twin lead and open wire is available for prototype and limited production systems. Higher density and lower cost P.C. interconnection systems are being developed for both the solder tail and double contact connectors.

Islands are interconnected with a high quality 50 ohm transmission belt (24 or 50 signals). Since a cable interfaces with the same socket as a logic package, the ratio of I/O pins to logic positions is not fixed by the hardware, but is established by the logic design. This flexibility provides for efficient island utilization. Figure 5.20 shows an island assembly mounted in an module with belted cables interconnections.

5.12.2.6.3 Frame, Cooling & Power: In addition to a standard logic family, island, and interconnecting belts, the BCML packaging system also provides a mounting structure, cooling system and power system for a 10 island module. This 10 island module can be used individually for smaller systems or can be stacked 2 and 3 high for larger systems. The 50 ohm belted calbes provide module to module interconnections.

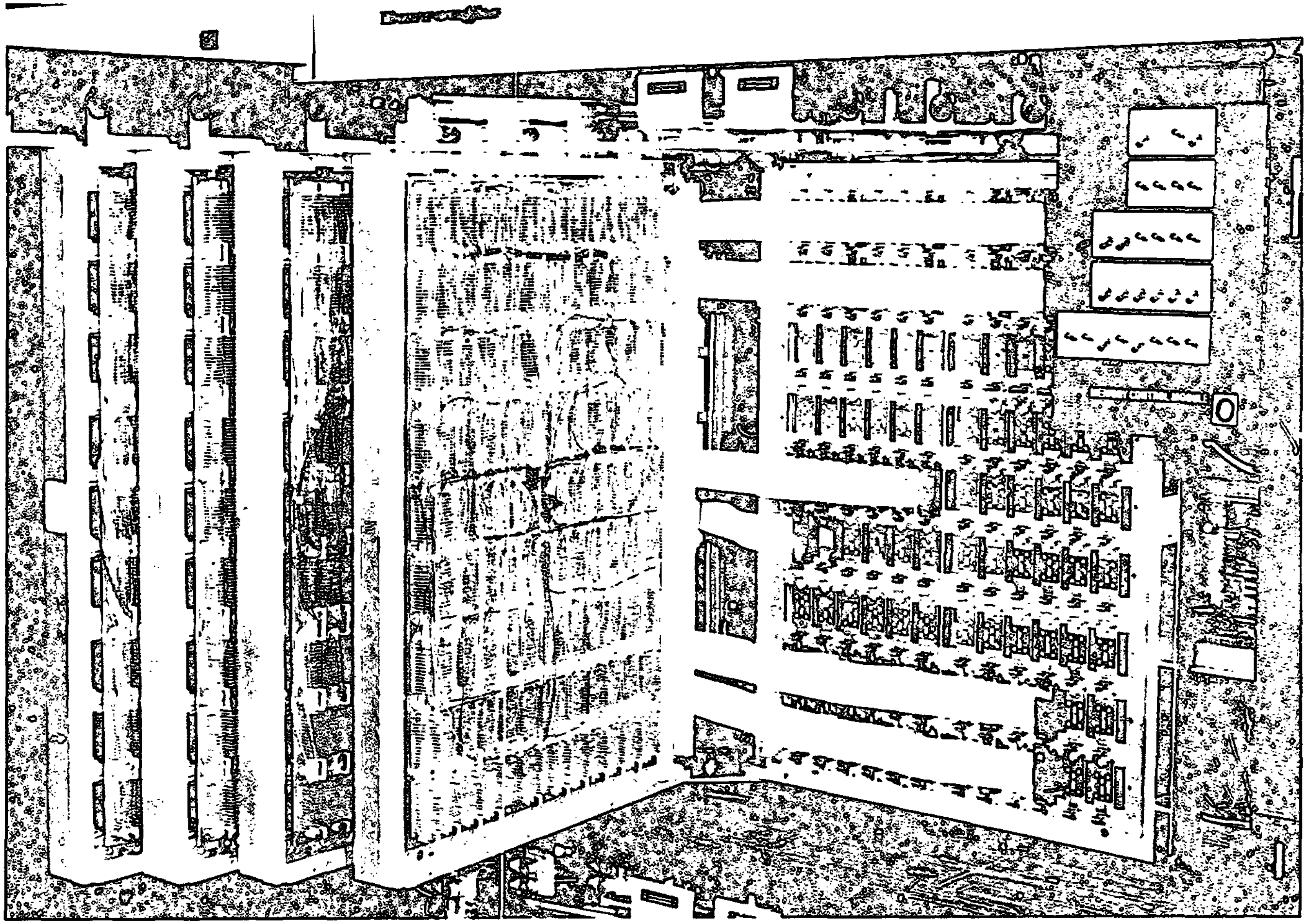


Figure 5.20 Island Assembly Mounted in Module with Belted Cable Interconnections

The module assembly enables the islands to fold out permitting front and rear access thus facilitating testing and maintenance. This feature is illustrated in figure 5.20.

The cooling system, which can dissipate a 3.6 KW heat load, consists of cold frames mounted on the islands, a circulating pump, fans, and a liquid to air heat exchanger. Air for the cooling system is drawn from the computer room. For highly reliable operation junction temperatures are restricted to 80°C with a 40°C ambient. Much higher power (or lower junction temperature) could be obtained by using a liquid to liquid heat exchanger with a chilled coolant circulated through the island. This system does not require air circulation in the computer room, with heat being dissipated directly to the building chilled water supply.

The BCML power system is designed to be driven by an M-G set or an equivalent line isolator. Large systems may be operated from a site M-G set but a 20 KVA M-G set has been packaged in a sound proof cabinet for installation in the computer room for use with small to medium systems.

The power supply itself is a very simple and reliable design, consisting of only a transformer and rectifiers. Output is -2.7V ± 30% and -4.8V ± 25%. Final regulation is provided by circuitry on the logic chip. This on-chip regulation produces a constant current load. Therefore voltage decoupling capacitors are not required on the P.C. board.

5.13 IMPLEMENTATION TOOLS

Burroughs Corporation has a central Design Assistance (CDA) Department which is charged with the responsibility of developing and maintaining a comprehensive set of tools to aid in the design, manufacture, and maintenance of computer systems. These tools are then adapted as needed and used by the various design and manufacturing groups.

The design of a complex system such as the FMP, requires the use of such tools. The Design Assistance System (DAS) and the Burroughs Interactive Logic Design (BUILD) program are examples of aids used during design. Specifically, the DAS programs provide assistance in the development of manufacturing tooling from a detailed logic design. The areas supported are:

logic partitioning
component placement
printed circuit routing
wire wrap routing
logic simulation
test generation
logic schematic generation
rules check
numeric control generation

In addition, design data is maintained in a centralized data base to insure design integrity.

The Burroughs Interactive Logic Design (BUILD) program allows a design engineer to hierarchically specify a logic design, and to verify its correctness using functional simulatin techniques. After logic verification, netlists are generated from the logic specification and entered into the DAS engineering data base for physical implementation.

Figure 5.21 depicts these two systems as they would be used by the NASF project.

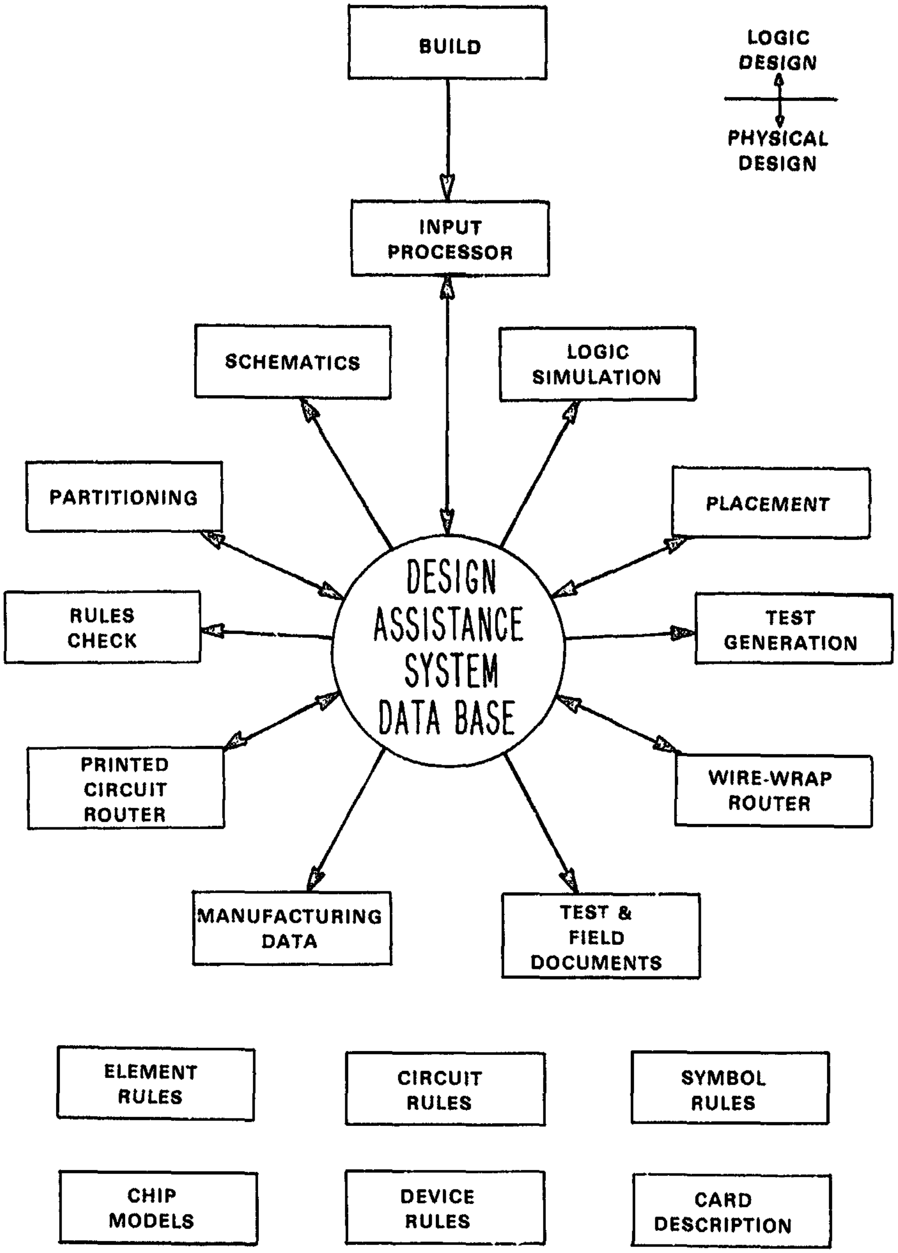


Figure 5.21 NASF Hardware Design and Implementation Support System

CHAPTER 6

TRUSTWORTHINESS AND AVAILABILITY

6.1 TRUSTWORTHINESS, AVAILABILITY, AND ERROR CONTROL

6.1.1 General Requirements

As the introduction to Chapter 5 has already emphasized, the FMP has certain requirements for trustworthiness, availability, and error control. Among these basic requirements are:

- System availability of 90% or better, implying an FMP availability of approximately 95% or better, for 20 hours a day.

- Mean time between aborts visible to the user of over 10 hours.

- Probability of apparently successful but wrong runs much lower that the probability of an abort.

In order to satisfy the above requirements, a number of features are built into the design, including:

- Spare processors and extended memory (EM) modules, with software-controlled reconfiguration

- Duplexed operation with comparison of results

- Error detection and error correction on all memories

- "Scrubbing" through CCD memory and dynamic RAM memory to find and correct any spontaneously occuring errors within them

- Fault detection within logic circuitry (processor, coordinator, etc.)

- Software-controlled restart following a program abort

- Logging of all errors, analysis of the logs

- Testing of invariants in the computation

- The ability to observe externally the state of the FMP

- A system of diagnostics and confidence checks

- Error detection in file system, both storage and transfer paths

These features are implemented by a combination of hardware and software.

The trustworthiness of computation on the NASF is the combined result of a series of influences, including

- System software
- Hardware reliability
- Hardware error detection
- Completeness of the confidence and diagnostic checks
- Applications programming characteristics
- Accuracy of failure identification
- Throughness of checks for software errors

6.1.2 Design Requirements

Additional characteristics can be derived from the basic requirements of the previous section. These characteristics were derived in Reference 5, and can be summarized as follows:

- Less than 1 bit in 10^{17} in undetected error from processor memory

- Less than 1 bit in 10^{15} with error detected but uncorrectible from processor memory

- Less than 1 bit in 10^{15} in undetected error from EM

- Less than 1 bit in 10^{13} with error detected but uncorrectible from EM

- Less than 1 bit in 10^{23} bits refreshed in DBM shall have an undetected error

The derivations were based on observations on how many bits were accessed from memory and from extended memory during the typical 15-minute run, and on an assumed time of residency in DBM that might be as long as a day.

6.1.3 Sparing and Duplex Processing

Every processor cabinet has 129 processor slots; every EM cabinet has 132 EM module slots. In the coordinator, there are four registers, one per processor cabinet, that designate the spare processor in that cabinet. Spare EM modules are designated by registers in the CN buffer of every processor. The coordinator broadcasts the designation of the spare to the CN buffer, using BDCST instruction, and follows that with a FILLR command to load these registers. Thus the designation of which modules are spare is changed by software resident on the coordinator.

Duplex processing has been proposed as a means of providing dynamic, run-time checking of processors by comparing the results of the same set of computations performed in two different processors. Two approaches were considered and are discussed below.

The spare processor designation is used to provide a duplex mode of operation. First, one must make sure that there are 516 good processors in the FMP. Second, processor #128 is designated "spare" in each cabinet. This makes programmatic processor numbers 0 through 127 fall on physical location number 0 through 127. Third, the program is run. Fourth, Processor #0 is designated "spare" in each cabinet. This makes programmatic processor numbers 0 through 127 fall on physical processors 1 through 128, so that every computation in the run will fall into a different processor than the first time. Fifth, the program is run again. Sixth, the results of the second run are compared for the expected match with the results of the first run.

Another form of duplexed processing was considered during the course of the study. Here the duplex mode would be implemented through some additional hardware. The 512 processors would be divided into 256 sets of 2 each. The application program would be compiled and run as if only 256 processors were available. Each set of 2 processors would execute the same code on the same data, and the resultts of each would be compared. The operation of the CN is such that continuous synchronization between the two members of the pair would require additional hardware means than described in Chapter 5, such as making both processors use the CN buffer of one of them. A hardware comparator would monitor the performance, and errors would be detected as soon as the outputs to the CN or to the coordinator, of the two processors, fail to match. Because of the synchronization problems, and because there seems to be no real advantage of this scheme over the purely software duplexed computation described first, the hardware comparator has not been included in the design.

6.1.4 Error Correction in Memories

All memory has error detection and error correction in order to achieve the very low error rates of the requirements. Error detection is a necessary part of hard failure detection. Error correction is proposed based on expected memory error rates between 1 bit in 10^9 and 1 bit in 10^{12}.

For processor memory and extended memory, a SECDED (single error correction, double error detection) code is proposed. The actual error rates in the chips would have to be very good indeed before simple parity plus retry would provide adequate correction. The actual error rates would have to be very bad (worse than 1 bit in 10^8) before simple SECDED was not good enough.

For Data Base Memory (DBM), a higher intrinsic error rate is expected from the chips, since the geometries on the chips are smaller, and since more refreshes occur per access. Also, a higher standard of performance is required, since any given datum will go through many read-write restorations during the lifetime of the data in DBM. As the computations in Ref. 5 show, we expect that the same simple SECDED will also be adequate error correction in DBM. However, the safety factor is substantially less, and a reevaulation of this choice should be made when the soft failure rate of the 256K CCD chips become known.

In the DBM it is also necessary to periodically read each word, make necessary corrections if possible, and write it back in, in order to keep the probability of multiple errors low enough. This process is called "scrubbing" and is expected to be designed into any memory system requiring it. Therefore, the DBM will not require any external controls for scrubbing. The errors removed by "scrubbing" are called "soft errors". This term, soft errors, implies failures where the contents of the storage cell have been modified in some unexpected or unplanned way (such as by the effect of background radiation), but which are not the permanent inability of a storage cell to operatoe correctly. The following paragraphs discuss the SECDED scheme proposed and also discuss the scrubbing of errors out of DBM.

6.1.4.1 SECDED

For soft failures, the previous studies (5) show that the improvement factor due to error correction is essentially infinite; that is, the system would be unable to produce useful results without error correction at the presumed soft error rates. For hard failures, the improvement factor due to the use of error correction depends on the failure modes. Some failures, such as an address decoding failure external to the memory chip that causes multiple bit errors, are not helped by error correction. A failure internal to a single memory chip is helped by error correction. In addition, the error correction circuits have failures that would not occur if there were no error correction. The analysis following in Section 6.3.3 recognizes the other effects contributing to undetected errors. That section uses a very conservative improvement factor of 5 in the number of observed errors when using SECDED for correcting hard failures vs the situation where no SECDED is used. The following discussion addresses these statements in more detail.

First consider the case of soft failures as represented by read failures. About 5 x 10^{12} operands are used or produced during the course of the "typical" 15 minute run (5). If half of these come from processor memory, that means almost 2 x 10^{14} bits are fetched from processor memory during the course of a typical run. Although accurate projections of bit error rates for large semiconductor memory chips await more experience , it is plausible that bit error rates may lie between one bit in 10^{10} to one in 10^{14} bits read. Under the above conditions, without error correction, it is unlikely that the typical run can even complete successfully due to soft errors.

For hard failures however, the picture is different. If one memory chip output is stuck in one processor, only 1/521 of the words accessed are affected by that failure. The processor memory delivers 4 x 10^{11} bits during the course of the run. If one bit in every word in that one processor is bad, and if the soft error rate is 1 in 10^{12} or better, the run will probably complete successfully. A failure at a specific bit in one chip is even less likely to cause trouble.

Since double errors or worse are not corrected automatically with the proposed SECDED code, it is important to use preventative measures. When the SECDED logic corrects a failure, a log will be updated indicating the word and bit position in the memory which was corrected. These logs will be examined regularly in order to detect and replace failed parts before they cause an abort.

The error correcting code of Table 6.1 appears to be the best choice for the FMP. First, it is directly implementable by the Motorola SECDED parity generator chip (each 8-bit wide slice of the code exhibits exactly the same pattern of parity checks as found in that chip). Second, it is much better than a randomly selected SECDED at detecting triple errors. Even the optimum SECDED is not very good at detecting triple errors when there are 55 bits used out of the underlying 64-bit long Hamming plus-parity code block. This proposed code is almost as good as that optimum.

Each "x" in Table 6.1 means that that bit is included in the parity check represented by its corresponding checkbit. The seven check bits are the 6 bits of the Hamming code, plus a bit that allows an overall parity check. For improved performance against multiple errors, the 7th bit contains an "x" only for those bit positions which enter into 0, 2, or 4 of the other check bits. Actual overall parity is the parity of all seven check bits. Odd parity is used.

The bit number in Table 6.1 is not the bit number of the data word. For one thing, the check bits are interspersed. The correspondence of bit number as seen by the programmer to the bit number of Table 6.1 is arbitrary. This mapping will be left as a logic designers option.

Triple errors appear to be single errors to the proposed code. Some triple errors will be detected when the SECDED circuits detect a failure and attempt to correct a bit outside the 55-bit field shown in the table (this is possible since the code chosen is a portion of an underlying 64-bit long Hamming plus-parity code block). The code shown in Table 6.1 detects 14.6% of all triple errors, whereas a randomly selected SECDED would be expected to detect 14.1% (nine bit locations out of 64 are outside the 55-bit word).

TABLE 6.1

Error Correcting Code

```
Check bits           XXX X   X       X               X

Bit Number *         0000000000111111111122222222223333333333444444444455555
                     0123456789012345678901234567890123456789012345678901234

                1st  .x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.x.
Check bit       2nd  ..xx..xx..xx..xx..xx..xx..xx..xx..xx..xx..xx..xx..xx..x
Parity          3rd  ....xxxx....xxxx....xxxx....xxxx....xxxx....xxxx....xxx
Patterns        4th  ........xxxxxxxx........xxxxxxxx........xxxxxxxx.......
                5th  ................xxxxxxxxxxxxxxxx................xxxxxxx
                6th  ................................xxxxxxxxxxxxxxxxxxxxxxx
             Parity  x..x.xx..xx.x..x.xx.x..xx..x.xx..xx.x..xx..x.xx.x..x.xx
```

* The assignment of bit number (corresponding to Hamming's) may be different than the assignment to be found in the register to which this parity check is attached. The bit number found here is the one generated as an indication of the bit to be corrected in the error correcting code.

Codes which are useful at detecting triple errors are also of interest. One additional check bit allows a code in which triple errors are almost always detected (better than 90% of the time). The price for this improved error detection capability is a connection network (CN) one bit wider or extended memory (EM) access time 20 ns longer, more complex parity checking, more complex decoding of the bit in error and 2% more memory. Current estimates of memory chip bit error rates imply that this additional complexity is not warranted.

SECDED checking and generating logic is found in the following locations:

- Processors, where the processor generates check bits for all memories it accesses (both PM and EM via the CN buffer), and checks words fetched from the PM or received via the CN buffer.

- Coordinator, where the function is parallel to that in the processors.

- DBM, in the channels to and from the file system.

SECDED logic is not needed in the EM modules, since all EM data will have check bits when stored, and will have their codes checked at some point after being fetched from EM, usually upon being read from a CN buffer in a processor.

In addition to the SECDED on all memory data, there are some simple parity checks. The address-plus-instruction-code sent through the CN for controlling EM buffers has parity checked at EM. The contents of microprogram memory in the processor have a parity bit.

The responses to SECDED and parity errors are as follows:

1. EM module detects parity error on module-number/address/op-code field sent from processor. The EM module does not return an Acknowledge on bad parity, so the processor will continue to send the same request. If the error was a transient, proper operation will resume. If the error was a hard error, the processor will hang on trying this request, eventually causing the coordinator to have a time-out interrupt. The EM module sends an "address parity bad" interrupt to the coordinator. This would normally be masked off to allow useful processing to continue in those cases where the retry works.

2. Processor corrects single error. An interrupt to processor-resident software results in the logging of the action in a table in processor memory.

3. Processor detects double error in word received from EM. The processor halts with interrupt, and the program is discontinued. Software can restart the program from some prior point, possibly after system reconfiguration.

4. In all of the above, the requestor may have been the CN buffer of the coordinator used by the coordinator for accessing EM. In these cases, read "coordinator" where the previous two sections say "processor".

6.1.4.2 Scrubbing Errors out of CCD Memory and Dynamic RAM

In the case of CCD memories, errors are not confined to the reading and writing process. Errors can also arise within the memory chips. If data is stored in a particular location with no reference for a long time, such as hours or days, the probability of errors may become intolerably high. It will be necessary, therefore, to continually scan through the data base memory (DBM) correcting all the single-bit errors in order to allow the survival of the data base for a long enough period of time.

Depending on the magnitude of the soft-error problem, it may be feasible to use a stronger error-correction code, and thus eliminate the scrubbing. With scrubbing, the probability of non-correctible errors grows linearly with time, the envelope of pieces that individually have the form t^e where e is the number of errors in the uncorrectible case (Figure 6.1). With stronger error correction, correcting f errors, the curve has the form t^f. e=2 for Hamming plus parity. f can equal any number for a BCH* code (7). Clearly, the "scrubbing" storage design has more latitude against variations in error rate.

The critical aspect of DBM is the storage of restart files, up to 10^9 bits, for times that presumably could be days. The method of error correction used will depend on the technology to be used for the file.

To determine the optimum rate for scrubLing errors out of CCD memory, we should know both the error rate for spontaneously occurring errors, and the error rate for the reading and writing process. For any given error correcting code, there will be an optimum scrubbing frequency where the two sources of error are in balance and are a minimum.

In Reference 5, the assumption was made that the CCD memory of DBM would lose on the average, 1 bit per 3 x 10^{16} bits shifted. This error rate was based on preliminary experience reported by Fair-

*Bose-Chaudhuri-Hocquenghem

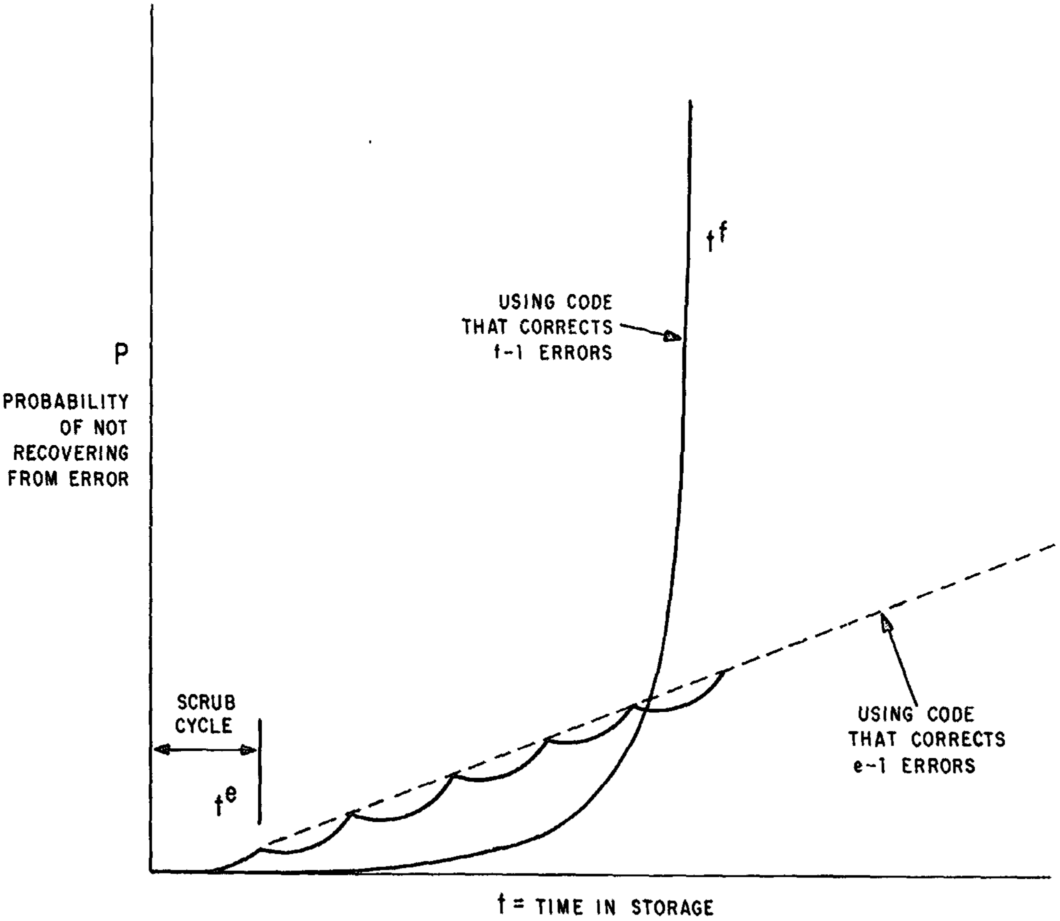


Figure 6.1 Scrubbing versus Read-Time Error Correction

child. Since then, the cause of loss of bits has been identified as background radiation, primarily due to alpha particles coming from contaminants in the package. More recent quantitative data is not available. New manufacutring techniques by the vendors appear to be solving the problems. On the basis of the original soft-failure rate data, a scrubbing rate of once every seven minutes will be enough to keep a 10^9 bit file error-free for one day with probability 0.999.

Scrubbing in the DBM will make use of hardware and data paths which would exist even if scrubbing were not necessary. In particular, the channels to and from the file system have buffers and SECDED checkers and generators associated with them. Part of the normal channel/interconnection path capabilities would be a loop-back mode for diagnostics. All of these capabilities can be utilized to implement scrubbing as needed. The DBM controller will schedule blocks (probably 16K words) to the channel buffers through the SECDED checker/generator and back to the CCD store.

The maximum transfer rate between the DBM and the file system is expected to be 40 Mbits/sec. At this rate, the entire DMB can be read in 3.5 minutes. Periods of high channel activity imply lowered requirements on scrubbing due to natural activity within the DBM. It is, therefore, reasonable to plan to use some of the channel capabilities (buffers, SECDED, loop-back) to implement the scrubbing functions. If DBM blocks are 16K words (a likely result of CCD organizations), and if the scrub cycle needs to be seven minutes then the scrub rate is one block every 51.4 msec.

As the geometries of the individual cells of integrated circuits shrink, other parts are expected to evidence soft-error problems similar to that being experienced by CCD parts now. 256 Kbit dynamic RAMs, which may be considered as a technological alternative to the 256 Kbit CCD's depending on the design and implementation schedule), are expected to experience a soft-error rate large enough to also require scrubbing. The parts currently planned for the extended memory (EM) have large enough geometries that the soft-error rate is very low. In addition, the EM does not contain any long-term data. Hence, no scrubbing is necessary or planned for the EM.

6.1.5 Error Detection and Correction in the Connection Network

The Connection Network (CN) is of central importance in the implementation of the proposed FMP. Since the design of interconnection systems are generally not as well understood as processors and since there appears to be less redundancy, the planned defenses against erroneous operation are described in some detail below.

6.1.5.1 Magnitude of the problem

As described later, single transient errors are self-correcting in the use of the CN. The only faults that might cause problems are hard failures (i.e. permanent failures). The discussion below shows that hard failures can always be detected during execution of user program (and therefore by implication detectable during confidence tests). Section 6.1.10.3, which follows, shows that these faults are diagnosable once the job in process has been aborted.

As to the magnitude of the problem, the CN is built from 39,280 identical LSI circuits. If these circuits have the "normal" failure rate of 0.1 failures per million hours, the expected MTBF will be 254 hours, or 33 failures per year. During the entire 10 year design life of the FMP 330 failures are expected. With the fault detection and isolation techniques outlined below, it is very unlikely that one of these expected 330 failures will be undetected.

6.1.5.2 Defense vs. Type of Fault

6.1.5.2.1. Single transient error in the request sent to EM. A single transient failure in module number, address, or opcode field causes the EM module to detect a parity error, which causes the processor to retry the operation in question. System software normally allows retries to proceed unmolested.

6.1.5.2.2. Single transient error in data. This is corrected by the error correcting code and logged. Computation proceeds.

6.1.5.2.3. Hard failure on the path from one processor to EM. This hard failure will either cause parity errors to be detected by the EM or SECDED errors in any words stored. The analysis in section 6.1.5.3 shows that over half of the addresses sent through the fault are detected as errors. The result is that such an error will be detected very quickly.

6.1.5.2.4. Hard failure on the data path from EM to processor. Only data, with SECDED, flows over this path. The analysis of the next section shows that over two-thirds of the faulty data words are detected. Thus, such a failure is quickly detected, usually on the first word transmitted after the failure occurs.

6.1.5.2.5. Hard failure in the path-selecting control logic. Here, there are several cases to consider.

First, if the wrong path is selected, and if the wrongly selected EM module has a different number of bits in its CN port number, a parity error is detected at the EM module. Half the EM modules will detect such a parity error, so that EM accessing will not go on for long without the error being detected.

Second, the correct path is selected, but with wrong priority, so that a particular processor is being discriminated against. The program will continue to execute correctly, but execution time will be lengthened for certain patterns of access conflicts in the CN. We believe that analysis will show that such priority failures are harmless for some programs, including aero flow codes, but no simulations to verify this expectation have yet been done. Such failures can be found by diagnostics. All processors are sent to fetch from EM, execution is allowed to proceed for a fixed time, and then it is observed that the processors with correct results are not the expected set.

Third, the strobe line is falsely high. This will cause the CN buffer to think that the EM module has granted access when in fact it has not. If there are no CN delays, the correct word will come back in spite of the fault. When there are delays, the CN buffer will pull in "garbage" since no real word is coming back at the time the false acknowledge says it is. Since the path from this CN buffer, if blocked, is blocked for at least one CN clock, that garbage is either all zeroes or all ones, for which the Hamming error correction identified bit 63 and bit 56 respectively as the bit in error. Since there is no such bit, the error is immediately caught.

6.1.5.3 Analysis

As described previously in Chapter 5, the Connection Network is designed to transmit a sequence of 11-bit frames. The main purpose of this approach is to reduce the number of wires and the complexity of the network itself. If the entire message is 33 bits long, then a stuck-at fault will change either 0, 1, 2, or 3 bits depending on whether those bits were the same value as the bit produced by the stuck-at fault or not. A stuck-at-ONE fault produces no errors when all the bits were ONE to start with. When the entire message is 55 bits long, the stuck-at fault jams five successive bits to the state at which the fault is stuck, producing 0, 1, 2, 3, 4, or 5 errors.

First consider the case that the module-number/address/opcode is being passed to the EM (33 bits) and the bit of the EM module number is the same as the value at which the fault is stuck. The remaining two bits can have either 0, 1, or 2 errors. When the remaining bits are address bits, it appears valid to assume that they behave as random bits. Hence we have 25% of the time no error, 50% of the time a single error that is detected by parity failure, and 25% of the time a double error that is not detected. Exactly two thirds of the errors are detected. Aften ten addresses have been passed through this fault, the probability of the error being detected is 99.9988%; after twenty, 99.99999997%.

2. Take the case as above, except the EM module number bit is wrong. When parity is checked, at the wrong EM module this time there will be either 1, 2, or 3 errors in the 33-bit package, again, with probability 25%, 50% and 25%. The single and triple errors result in parity errors and are detected. Thus, exactly one half of the errors are detected. After ten addresses have passed through this fault, the probability of the error being detected in 99.9% after twenty, 99.9999%, again on the random assumption for addresses.

3. For the third case, data, the analysis is of the same kind, but there are more cases Hence, it is easier to present the analysis in the form of a table for the cases that there are 0, 1, 2, 3, 4, or 5 errors. For each possible number of errors, Table 6.2 shows the percentage of time we expect to find such error, (the binomial distribution) when it occurs, the percentage of time that this hardware error causes no operational failure (for example, a single error is corrected using the SECDED code), the percentage of time that this number of errors is detected, and the percentage of time that a single data word can slip by in error. 14.6% of the triple errors will be detected, and 85.4% of them will appear to be correctible single errors and therefore not detected.

Table 6.2
Single 55-bit Data Word passing through CN with single hard fault

No. bit errors	Occurrence	No. Func. Failure	Error Detected	Error Undetected
0	3.12%	3.12%	-	-
1	15.62%	15.62%	-	-
2	31.25%	-	31.25%	-
3	31.25%	-	4.56%	26.69%
4	15.62%	-	15.62%	-
5	3.12%	-	0.46%	2.67%
TOTAL		18.75%	51.89%	29.36%

From the table, we see that the ratio of detected failures to undetected failues is 51.89/29.36. That is, 64% of all the functional failures are detected. After ten words have passed through this hardware fault, the probability that the fault has been detected is 99.9999%, assuming random data.

6.1.5.6 Logical Checks

Miscellaneous logical checks can be considered. The design intent of these checks is to localize the effects of some error in the FMP. The following list of checks includes those also listed in Appendix C in the list of interrupts, plus others.

- Parity checks on microprogram

- Memory bounds checks (optionally inserted by compiler)

- "Uninitialized" word fetched to instruction decoder or floating point unit

- Illegal opcode

- Detection of unnormalized floating point operand (except second half of double length floating point)

- Integer overflow or underflow

- Divide by integer zero

- Floating point overflow (either tested for or marked "unrepresentable", a compile time option)

- Timeout

In addition, there is a bit in the interrupt register reserved for any miscellaneous logic malfunction checks that will be built into the hardware. Lock-up of the end-around carry chain is an example of the sort of logic error whose occurrence would be reported in this bit.

6.1.7 Restart

Previous analysis (5) shows that the optimum time between restart dumps is given by

$$T_{opt} = (2T_gT_r)^{\frac{1}{2}}$$

where T_g is the mean time between failures (intermittent or hard) that cause an abort, and where T_r is the time spent taking one restart dump and also the time required to load the restart point and switch to user programming, assumed equal.

Typical runs are 10 minutes, and typical data bases are 15 x 10^6 words (5). Since 15 x 10^6 words are loaded into EM in 0.375 sec, we have T_r not more than about 0.5 sec. Since T_g will be on the order of 10 hours or longer, we have $T_{opt} = (2 \cdot 0.5 \cdot 36000)^{\frac{1}{2}}$ seconds, or just over 3 minutes. However, the amount of computation time saved by dividing a typical 10 minute run into three restartable segments is estimated to be about 0.3% of all FMP time[1]. Hence there is little point to providing restart points within the tyical 10-minute run.

Unless restart points are provided by the user, the restart strategy will be to restart the same task again automatically under software control. Automatic restart is limited to those aborts that are probably caused by hardware error, such as parity errors. Aborts that are likely to be software errors, such as addressing errors, will not trigger automatic restarts. The operating system handles automatic restarts, and reports their occurrence.

The two types of restart dumps mentioned above should be delineated Automatic restart dumps are likely to be a roll-out (with a later roll-in) of the entire job. In this case, all data space, variables, flags, et. al. would be dumped to the file system via DBM. Restart points provided by the user are expected to be more restricted. The user would be permitted to specify selected data areas to be affected and to specify when such snapshots are to be taken. These user selected restart dumps would be much more efficient and cause considerably less load on the system than the automatically generated dumps. In addition, the user will be permitted to insert an alternate entry point in his main program (a restart point), where appropriate arrays from the restart dumps would be reloaded. In the initially delivered system, automatic checkpoint restart transparent to the user will not be included. Such a facility would be included at a later time.

[1] T_{opt} is about 200 seconds; T_r is 0.5 seconds. At optimum, the fraction of time lost due to restart dumps is approximately equal to the time lost due to wasted computation, that is, computation that ends in an abort. In a 10 minute run there are two 0.5 sec restart dumps, plus the initial loading of the data base (total 1.5 sec) and an equal expected amount of time lost by aborting good computation. (1.5 sec. + 1.5 sec.)/600 sec reduces net throughput to 99.5% of what it would have been if defense against aborts were not necessary. If no restart dump is taken during the 10 minute run, the 1.5 sec of data moving is reduced to 0.5 sec. However, the time lost from wasted computation will triple, since 600 seconds is triple 200 seconds. Hence, the percentage of time lost is (0.5 sec + 4.5 sec.)/600, and net throughput is 99.2% instead of 99.5%. This is a small price to pay for the convenience of not having to worry about restarts.

Since the intent of on-line spare components is to provide the capability of maintaining the desired level of performance through aucomatic reconfiguration, the system software would be able, in most cases, of automatically restarting a job whose execution may have been interrupted by a hard failure. When the job is a program with user-specified restart dumps and restart points, the most recent restart dump would automatically be chosen and the execution would resume at the restart point. For example, in a one-hour run involving 500 time steps, it might be reasonable to specify a restart dump every 25th time step. The restart entry point would include the reinitialization of control variables to states saved as part of the restart dump. The above technique is particularly appropriate to the aero flow codes, where the computation converges.

An analysis of the effect of restart on the operation of the FMP, using the reliability model, is contained in Section 6.2. In that section, the assumed "restart" time of 6 minutes, corresponds, not to the T_r above, but to the total time spent at the time of restart, including system software response to the abort, logging of error, reconfiguration of the system, if any, and running of confidence (and possible diagnostics as well depending on the type of error detected). Six minutes seems extremely generous.

6.1.8 Error Logging

Where possible, all errors are logged. The mechanism for logging errors is via interrupt. Both the processor and the coordinator have three classes of interrupts which can be used for logging.

One class of interrupts reports non-fatal errors (such as single-error correction of a transient parity error detected in EM).

A second class of interrupts are the programmatic interrupts (CALLI instruction) which can be used for calling on system software to log errors. In many cases these may be errors detected by tests inserted by the compiler into the code stream.

The third class of interrupts is used to log all fatal errors. Since fatal errors involve some non-correctable situation, these interrupts are usually directed to the coordinator. In the case of the coordinator itself, they are directed to the diagnostic controller and the support processor.

The design intent is to record the memory address and bit number of bit in error (also called "syndrome") for all SECDED error corrections and detections. It is likely that programmatic interrupts will report not only the observed error condition, but also a code which would be used to obtain a link back to the original source code. A table in each memory holds the record of the last N errors corrected. The size of the error log tables and the frequency with which they are collected and reported has yet to be determined.

6.1.9 Invariants

Applications software is one of the links in the chain that maintains the trustworthiness of the NASF. Although the application software is outside of the scope of work when developing a facility, it is part of the system seen by the user and therefore must be considered when discussing the trustworthiness of a system. Inclusion of checks on quantities which should be invarient or in some way well behaved during the course of the computation seems appropriate. Examples might be:

- Total quantity of air within the mesh (as computed from the appropriate function of geometry and pressure) should change in accordance with air inflow and outflow at the boundary.

- Any global criterion for convergence should improve monotonically for steady airflows.

- Changes in total energy in the system should correspond to energy inflow and outflow at boundaries

Discussions are currently under way on constructs for the language which would make such invariant checking more convenient.

6.1.10 Diagnostics

All of the FMP shall be diagnosable. Creating diagnostics is difficult at best, because of its interdisciplinary nature. Hardware features for aiding diagnostics must be designed. The diagnostic programmer must be expert both in the logic design of the machine being diagnosed, and expert in programming at the machine dependent level. Completely automatic diagnostics for all conditions is an unreasonable goal. This project would plan on computer-assisted diagnostics.

The built-in fault detection mechanisms of the FMP have already been discussed. In order to meet the desired goals and availability and MTTR (Mean Time to Repair) for the system, direct and simple means for diagnosis of the system components is required. Because of the scope of the system, direct control of diagnostics from some central point (the diagnostic controller (DC) for instance) is not realistic. A hierarchy of controls will be provided. In general, every diagnostic interface to the next level of detail in the system is expected to have a mode of operation which allows the outer level direct control over setting and observing any state (bits) in the immediate next level of detail. For example, the logic in the diagnostic controller (DC) would be tested (set DC state including command, run the DC clock, observe the DC state) by the support processor. The diagnostic controller, in turn, tests the state control logic of the coordinator in the same way. The coordinator tests its own memory and the state control logic of each of the processors. The processors and coordinator jointly check the Connection Network and the EM modules. The processors will also be able to check each other. The coordinator also tests the state control logic of the Data Base Memory controller. The DBM controller then tests the rest of the DBM including the path to the File System.

Figure 6.2 shows the layered structure of the off-line diagnostics. Layer 1 is the initial phase of the "hard core", when the Support Processor is learning to trust the command-accepting portion of the DC. Layer 2 is the rest of the "hard core", also imposed by a Support Processor program, checking out the DC and enough of the coordinator so that the coordinator can be trusted to execute successfully. Layer 3 runs on the coordinator and exercises that portion of the FMP to which the coordinator has direct access. Layer 4 consists of those portions of the FMP to which the coordinator has only indirect access. The coordinator must cause the DBM controller and the processor to execute certain operations in order to get these portions exercised. Some diagnostics for layer 4 run on the DBM controller and the processors as an array.

This on-line form of diagnostics is used as needed to isolate or confirm an error to a replaceable unit (such as a processor). At that point the system is reconfigured, checked and execution resumes. If the automatic diagnostics are unable to confirm the location of a fault, the same controls are accessible to the maintenance personnel who can develop custom tests sequences as required. Note that when the system successfully detects a failure, isolates it to a system component, swaps in a spare component and resumes execution without requiring manual intervention, the system is defined to be continuously available. Only when manual intervention is required to isolate a problem and restart the system is the system considered to be unavailable.

Once a failure has been isolated to a system component, such as a particular processor, and that component has been switched "off-line", isolation of the bad component can proceed concurrently with the resumption of execution of the FMP. These off-line tests would consist of two types. Some tests will be possible with the system component still attached to the system. These tests would allow "in-situ" testing without disturbing the environment in which the error occurred. Thus, spare system components will be capable of access to other spare components without disruption of the on-line portion of the FMP.

In addition to the above test modes, test equipment is expected to be available to test the removable system components away from the system.

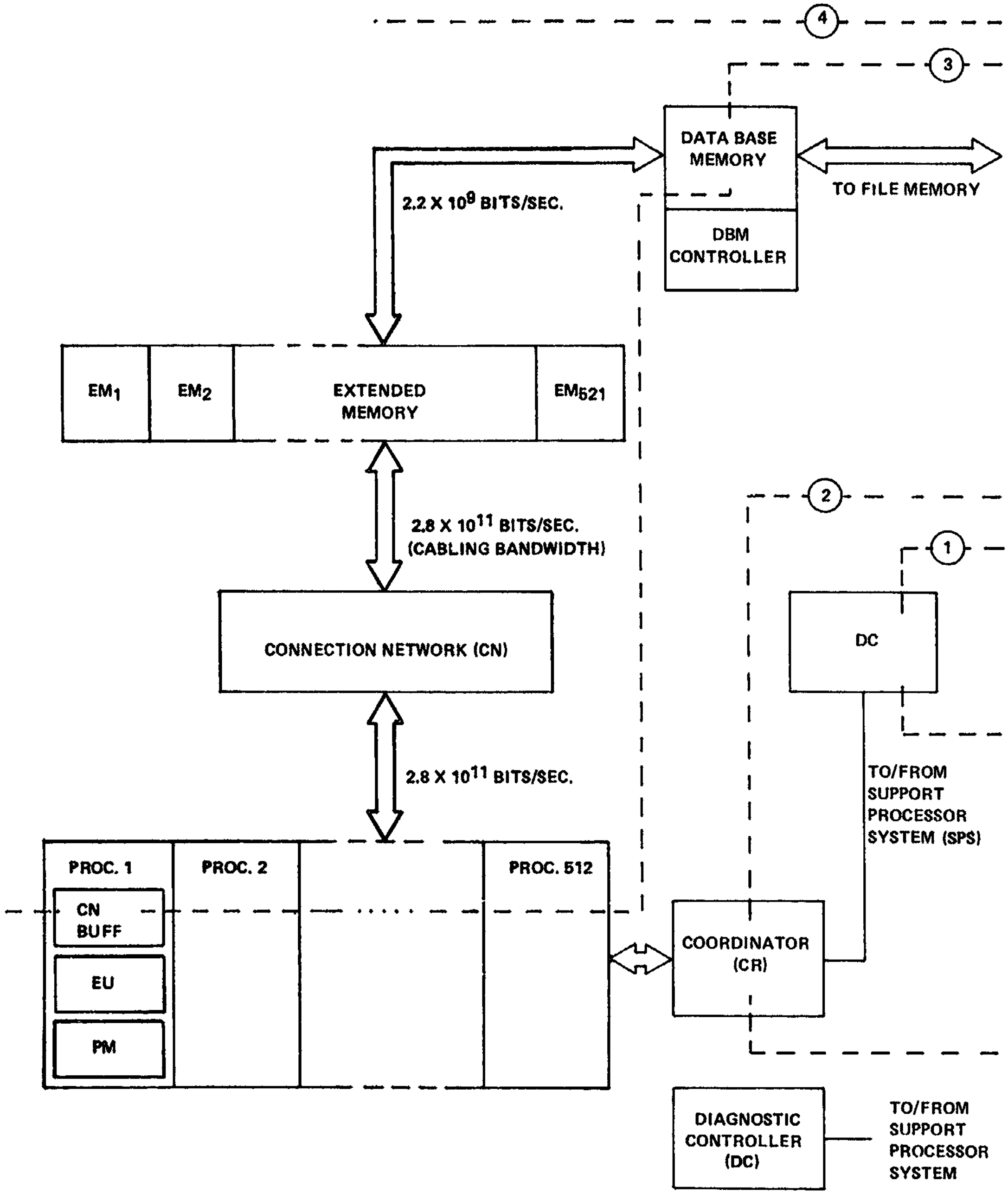


Figure 6.2 FMP Block Diagram with Diagnostic Layers Superimposed

6.1.10.1 Level of Performance

One should be aware that there is no single magic date on which the diagnostics will be "finished". The delivery date for the diagnostic software will merely mark a time at which the diagnostics achieved have a useful level of accuracy. On that date, there will be still room for improvement in the diagnostics. Diagnostic programs should continue to improve as operating experience shows up unanticipated failure modes and shows the areas in which the accuracy of the diagnostics can be improved. The goal is to achieve the highest possible uptime with the least amount of time lost to either downtime or lost to running diagnostics. It will be important to continue to fund diagnostic development at a modest level of effort for the life of the NASF in order to continually improve the efficiency of the support operations and to reflect the design updates and changes which are a normal part of the life of any system.

The initial capabilities of the automatic diagnostics system have yet to be defined. The automatically executed diagnostics would detect X% of all possible failures. The goal of this set of automatic diagnostic programs is to isolate faults to the least replaceable unit at the FMP level (coordinator or CN card, processor, EM module, ...). The diagnostics shall locate the failure to a single LRU Y% of the time, and shall locate the failure to within N LRU's Z% of the time. When a failure could be either on the backplane or on a LRU, the probability of detecting whether or not it is on the LRU itself or in the backplane behind the LRU will drop to W%.

The off-line LRU diagnostics (tester programs), shall localize failures to the chip, or to some number of chips with similar percentages. U% of the failures shall be found, V% shall be localized to within N chips, and T% shall be localized to within one chip or component. All of the above percentages need to be determined.

6.1.10.2 NASF Computer-Assisted Diagnostic Tools

A diverse set of diagnostics will be implemented for the NASF.

6.1.10.2.1. Support Processor System Diagnostics. The Maintenance Diagnostic Unit, a separate execution unit of the Support Processor system, can impose diagnostic operation on any off-line elements of the Support Processor. The MDU can write information into any flipflop of the Support Processor, cause the unit to execute any number of clocks, and then read the state of any flipflops. Results are then compared to precomputed results.

6.1.10.2.2. Support Processor Peripheral Exercisers. Programs resident on the Support Processor exercise the peripherals of the Support Processor.

6.1.10.2.3. FMP Off-line Diagnostics. These diagnostics are used when the FMP is not executing user programs and is considered "off-line" in terms of production commitments. These diagnostic procedures execute throughput the FMP depending on their purpose.

The "hard core" of these diagnostic procedures is a program resident on the Support Processor exercising the FMP via the DC. During early debugging, the DC will be available before the diagnostics have been written, and some diagnostic capability will exist by controlling the DC manually from the maintenance console.

After the coordinator has been diagnosed (or after confidence has been gained in the coordinator), most of the rest of FMP diagnostics will run in the coordinator. These run much faster than DC diagnostics do. The analysis portion of all those diagnostics runs in the Support Processor. The coordinator will check the viability of each of the EM module controls. The EM modules will be exercised in detail as part of the CN test.

Each processor will check its memory. CN diagnostics require the execution of EM accesses from a number of processors acting in concert. The CN diagnostics therefore occupy the entire array, just as does a user program.

6.1.10.2.4. Off-line LRU Diagnostics. These diagnostic programs execute on the test equipment. Every LRU can be diagnosed to the chip level, or exercised with sufficient flexibiltiy that the technician can diagnose to the chip level. The number of different types of testers which may be required is yet to be determined. All testers are expected to be program compatible with each other, so that one language creates tests for all of them. That test generation language would be linked to the design data base.

6.1.10.2.5 PAL (Programming Aid for Logicians). PAL is the language in which simple tests can be written on-the-spot for execution by the DC, or for execution on the B7800 for exerting control over the array via the DC. Eventually, the PAL programs would form a library that would continue to be useful after delivery, especially for the small residue of failure modes which the automatic diagnostics do not adequately support.

6.1.10.2.6 Analysis of Logged Errors. Tables which contain the error logs would be periodically collected and provided to a program which analyzes and summarizes the error activity in these logs. This program would execute on the Support Processor.

6.1.10.3 CN Diagnostics

The Connection Network represents a design which is novel when compared with previously existing circuitry for which diagnostics have been generated. The diagnostic approach described below would allow the FMP to isolate faults to the bit and node within the CN. Since the approach is a successive-refinement technique, some savings may be gained by stopping the FMP automatic diagnostic at the board level (the replaceable unit) and isolating the failed chip using off-line test equipment.

6.1.10.3.1 Assumptions and Design Requirements. The following features of the FMP design, and of the CN portion of it, are the basis for what follows.

- SECDED is checked on the data. The checking is performed in processor or coordinator

- Parity is checked on the addresses and operation codes sent from processor or coordinator to a single EM module.

- The Omega network, from which the CN design is derived, has one and only one path between a port on one side and a port on another, so that when an error is detected, the path through the network taken by that erroneous data is known.

- When processor number and EM module number are known, the operating system can translate these numbers into CN port number on the processor side and CN port number on the EM side. In general, errors will be reported by processor number and EM module number, whereas the diagnostics need to know physical CN port numbers. This translation needs to be done not only for CN diagnostics, but also for processor and EM module diagnostics as well.

6.1.10.3.2 Localizing a Hard Error in the CN. The analysis of the CN starts with the analysis of the Omega network. The argument will then be expanded to the more complex case of the actual CN. Between port n on one side and port m on the other side, there is a fixed path. All traffic between these two ports takes the same path. Between some other ports n' and m' there is also a fixed path. None or some of the nodes on the path n'-m' are the same as the nodes on the path n-m. Inspection of the four binary numbers n, m, n', and m', bit by bit, will disclose in which of the ten levels of logic in the CN do these paths have common nodes. By choosing two paths n-m and n'-m' which have some nodes in common, and finding that the same error occurs in data traversing both such paths, we localize the fault to those nodes.

Given a particular path from m to n, and knowing which contiguous levels of the ten logic levels we want to include in some other path, we make m' different from m for all those bits corresponding to levels on the left side of the Omega for which the path is not to be identical, and we make n' different from n for all those bits corresponding to levels on the right side of the Omega for which the path is not to be identical.

Presumably the diagnostics will be written using a binary search strategy. First we run tests in which the faulty path and the other path have four nodes in common, then tests in which they have two nodes in common, then one.

In the preferred CN version, there are two Omega networks, not one, with the result that the path is unique only to within a pair of nodes, (one in the upper Omega, one in the lower Omega) at each point. Two paths that must intersect in the simple Omega can pass each other without using the same gates, if one uses the node in the Upper Omega network and the other uses the node in the lower Omega network. The CN would be designed to inhibit this redundancy. If the two Omega networks communicate only at the ports (a version that was simulated on the CN simulator), we use a diagnostic control that disables either the upper or the .ower Omega while the other one is exercised.

If the two Omega networks allow paths to be connected between upper and lower network at each pair of nodes, then diagnostic disable/enable controls are needed on both Omegas at all ten node levels, twenty such signals in all, so that at each node level one can force a path to stay in the same (upper or lower) Omega network, or force it to jump (from upper to lower or vice versa). With these controls, all paths can be exercised under the same diagnostic scheme as described for a single Omega.

The error detection used by the diagnostics is the SECDED check on words that have been sent through the CN in one direction or the other, and the parity check on addresses and commands sent from processors to EM. Now a given SECDED error could be due to an error on the path to EM during a write, or due to an error in the EM module itself, or could be due to an error on the path from EM to processor. The diagnostics must distinguish between these several cases. A test on EM module M consists of writing into the possibility of faults in the CN). Whe৲ the memory is checked out, the diagnostics can tell the two directions in the CN by sending data between the EM module and several different processors. To make sure the failure is not a write failure if the read appears to fail, write commands to the EM would be generated from several processors. Likewise, redundant reading is used to check for write failures.

The diagnostics must detect the case that the EM port number is being erroneously interpreted in an EM access request so that the fault also causes one to traverse the wrong path. This case is detected by the parity check at the EM module which covers module number as well as address and opcode.

6.1.10.3.3 Diagnostic Generation Scheduling The schedule for creating diagnostics is contrained by the requirements of fabrication, debugging, and system integration. The first facility needed is the test equipment, with enough of the test equipment software written to facilitate manufacturing acceptance testing of the LRUs as they are built. The first LRUs built are the processor and the DC boards. Fabrication generally follows the same sequence as the diagnostics: the DC is completed first, the coordinator is completed before the last processor is plugged in, EM integration (including the CN) follows successful processor operation, and DBM is the last item to integrate. However, because of the number of processors involved, processors must be among the first components fabricated. On this basis, we see that the sequence of creation of the diagnostics is

1. Tester and tester programs start first. The first tester programs written are for DC boards and processor.

2. Processor on-line diagnostics to run on the processor while the processor is on the tester. This is an early version of the same processor diagnostic test used for FMP automatic diagnostics

3. The PAL assembler. This is used to generate tests on-the-spot by the debugging logicians as they debug the coordinator, the fanout boards, and the DBM controller

4. On-line diagnostic tests are used to verify proper design and operation of the FMP. The on-line tests are used as part of the acceptance tests.

6.2 RELIABILITY, AVAILABILITY AND MAINTAINABILITY

The efforts in reliability, availability and maintainability during this study addressed the following key areas:

- The effects of redundancy and parts quality on the FMP reliability.
- An updated and refined reliability and availability analysis of the FMP and the NASF system
- An estimate of the maintenance manpower required to support the FMP

The redundancy study showed that the use of redundant processors and extended memory modules and redundancy in the data base memory provided significant improvements in FMP availability and especially mean up time (MUT). The level of redundancy studied is now incorporated in the FMP architecture presented in Chapter 5. The use of B-2 quality level components versus C level quality was also shown to make a significant improvement (B-2 and C level component quality represent levels of quality resulting from different degrees of testing and screening; discussed in more detail in Section 6.2.3). The refined predictions of FMP (and NASF) reliability are based on the incorporation of these conclusions. The results of the refined reliability analysis of the NASF are summarized in Table 6.3 which presents mean up time, mean down time and availability of the three major subsystems of the NASF.

The refined reliability analysis of the FMP considered a range of failure rates for the LSI memories, as well as a range of improvement resulting from the application of SECDED, a range of intermittent or "soft" failure rates, and a range of efficiencies for recovery from interruptions. The results of this analysis provide three levels of reliability for the FMP. A lower bound (or worst probable case), a probable case and an upper bound (or best probable case). The results shown in Table 6.3 are for the probable case.

TABLE 6.3
NASF AVAILABILITY ANALYSIS

	FMP	FILE MANAGEMENT SUBSYSTEM	SUPPORT PROCESSOR SUBSYSTEM	COMPOSITE
MEAN UP TIME	14.9 HRS	19,310 HRS	263.0 HRS	14.1 HRS
MEAN DOWNTIME	0.14 HRS	1.9 HRS	0.68 HRS	0.17 HRS
AVAILABILITY	0.9904	.9999	.9974	.9880

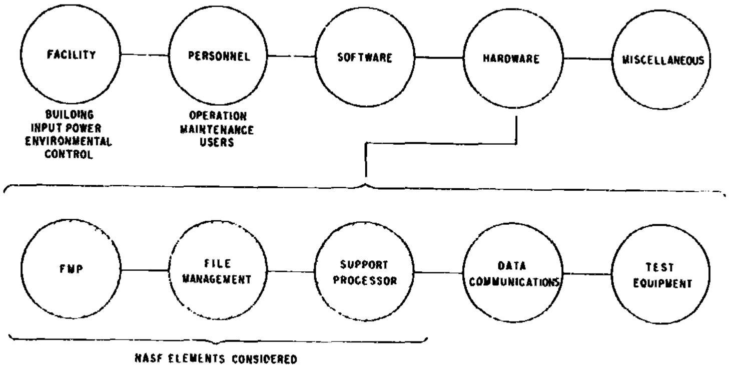
An examination of the FMP reliability analysis results reveals that the connection network (with no redundancy) has a signficant impact on the FMP failure rate. Similarly the reliability of the data base memory becomes a determining factor in the FMP reliability if only the lowest probable SECDED improvement factors are achieved and the failure rates of the LSI memory circuits (256K) are no better than that assumed for the worst probable case. Future efforts regarding the FMP should give careful consideration to these two areas to achieve optimum FMP reliability.

The maintenance analysis revealed that to have a 95% confidence of meeting the required repair and maintenance actions of the FMP for any given seven day week, a minimum of 13 maintenance personnel working five shifts each must be available (65 8-hour shifts). Assuming 21 shifts per week (3 shifts per day x 7 days per week), the maintenance of the FMP will require an average of 3 persons per shift, excluding operators, administrative and supervisory personnel.

6.2.1 Reliability/Availability Model

A generalized systems model for predicting the reliability and availability for a computer system includes many elements. Figure 6.3 describes this general model for NASF. There are five major elements: facility, personnel, software, hardware and miscellaneous. While all of these elements impact the ultimate system availability, the analysis and predictions conducted at this time consider only the hardware and some interruptions of a "soft" or intermittant nature contributed by the other elements.

This model, as well as the reliability block diagrams of NASF elements illustrated later on in this chapter, illustrate the inter-dependency of the subelements that contribute to the reliability of the system under consideration.



GENERAL RELIABILITY/I

Figure 6.3 General Reliability/Availability Systems Model for NASF

The NASF hardware includes five elements, (1) the FMP; (2) file management subsystem; (3) support processor subsystem; (4) the data communication subsystem and (5) the test and maintenance equipment. The data communications subsystem consists of a large number (over 100) of terminal interfaces, modems, networks and I/O devices. The data communication processors are included in the support processor subsystem. Failure of any one of the devices or interfaces in the data communication subsystem has no impact on the availability of the NASF for the other devices and does not impact the availability of the rest of the system, therefore the data communication subsystem portion of the NASF hardware was excluded from the study.

The availability of the test and maintenance equipment can be adjusted to a level, through the use of redundant equipment, that will have little impact on the overall system availability. The remaining hardware elements of the NASF (1) the FMP, (2) the file management subsystem and (3) the support processor subsystem, are addressed in this analysis. Detailed models (reliability block diagrams) of each of these elements are provided later in this chapter.

Programs developed by the Burroughs Corporation to aid in designing fault-tolerant computers were used with the above models to determine the system/subsystem reliability/availability/maintainability. Details of these programs and definitions of terms are included in Appendix D.

6.2.2 Redundancy Study

The FMP architecture consists of parallel elements in a number of areas. Parallelism readily permits the use of redundancy for improving availability. Redundancy however can also impact equipment and maintenance cost, increase failure rate and frequently increases the software and operations complexity. An analysis was conducted to compare the effects of the application of redundancy to the FMP in three areas: (1) the processors, (2) extended memory, and (3) data base memory. These areas represent three of the major areas of the FMP.

Calculations of the mean up time (MUT), mean down time (MDT), and availability (A), of the FMP were made with various combinations of redundancy. The level of redundancy used is that discussed in Chapter 5 This includes 4 on-line processors resulting in a redundancy of 128 required out of 129 processors available in each of 4 processor bays; 4 on-line extended memory modules resulting in 130 out of 131 extended memory modules for 3 of the 4 extended memory bays and 131 out of 132 extended memory modules in the forth bay and a partitioning of the data base memory into 4 sections of which any 2 are required for the FMP to be available.

Table 6.4 shows the results of this reliability and availability analysis for eight different combinations of redundancy (listed as Cases 1 through 8). The power of redundancy in improving mean up time and availability can be seen by reviewing these results which are summarized in figure 6.4. The use of redundancy in only one area makes only a modest improvement in the mean up time and availability. Use of redundancy in two areas increases the mean up time and availability, somewhat more. The use of redundancy in all three areas results in a significantly higher mean up time and availability.

TABLE 6.4
Effect of Redundant Elements on FMP Reliability

	REDUNDANT ELEMENTS			MEAN UP TIME (HOURS)	MEAN DOWN TIME (HOURS)	AVAILABILITY
CASE	PROCESSORS	EXTENDED MEMORY	DATA BASE MEMORY			
1	NO	NO	NO	10.2	0.65	.9403
2	YES	NO	NO	15.6	0.43	.9730
3	NO	YES	NO	12.7	0.25	.9449
4	NO	NO	YES	16.2	0.72	.9575
5	YES	YES	NO	22.3	0.51	.9878
6	YES	NO	YES	36.2	0.33	.9908
7	NO	YES	YES	23.5	0.92	.9622
8	YES	YES	YES	117.9	0.51	.9956

It should be pointed out that the data base used for these calculations does not include all the factors used in the analysis reported elsewhere in this chapter. The results presented in this section should only be used for ascertaining the sensitivity of the FMP reliability and availability to redundancy.

The conclusion of this study was that the application of redundancy to these three areas to the extent defined, represent a significant improvement in FMP reliability and availability. The predicted reliability and availability values for the FMP and NASF presented in this chapter are predicated on the use of this redundancy.

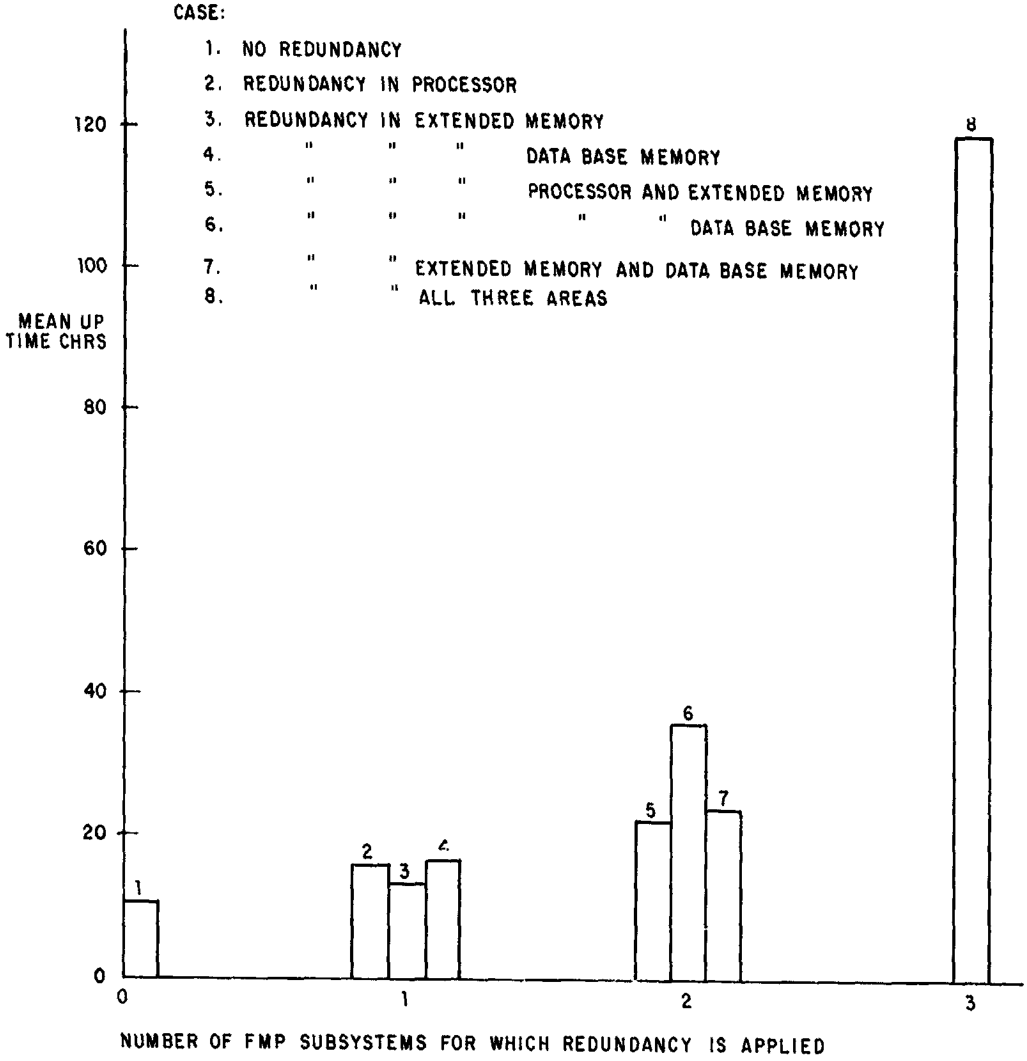


Figure 6.4 Effects of Redundancy on FMP Mean Up Time

6.2.3 Component Quality Study

Various levels of component quality are available for fabricating electronic systems. These levels are achieved through the application of certain screening and testing procedures as called out in various government specifications and statements. Levels most likely to be considered for the FMP are B-2 and C. The B-2 level represents the vendors equivalent of a number of these screening and testings procedures including a 168 hour burn-in. The C level has less stingent tests and no burn-in but is done specifically to the government specifications (See reference [9] for more information).

The effect on FMP reliability of using B-2 level versus C level quality components was investigated. Table 6.5 presents these results. Four cases were analyzed. The quality level was varied for the FMP with non redundancy and with the level of redundancy presented in section 6.2.2 above. It is noted that the higher the Mean Up time the greater the impact of component quality. The conclusion of this study is that if a high reliability in terms of mean up time is desired, higher quality (B-2 level) components should be used. The predictions of the FMP and NASF reliability and availability presented later in this cahpter are predicted on the use of B-2 level quality components.

Table 6.5
Effects of Component Quality on FMP Reliability

CASE	QUALITY LEVEL	REDUNDANCY	MEAN UP TIME (HOURS)	MEAN DOWN TIME (HOURS)	AVAILABILITY
1	B-2	NONE	10.2	0.65	.9403
2	C	NONE	8.9	0.68	.9296
3	B-2	YES	117.9	0.51	.9956
4	C	YES	75.0	0.51	.9932

6.2.4 FMP Reliability and Availability Prediction

Since the FMP is the most complex element of the NASF hardware and since the concept under consideration involves highly state-of-the-art technologies, a more detailed analysis has been conducted on this element. As described earlier, a number of factors have been considered in the FMP analysis. The value of these factors are varied over a range to provide an upper and lower bound as well as probable values for the reliability and availability. Figure 6.5 shows the reliability/availability block diagram used for the FMP. In addition to the redundancy shown, a B-2 quality level (in accordance with MIL-HDBK-217B) [9]was assumed for the integrated circuits and 6 minute recovery time assumed for manual operator restart. The mean times to repair (MTTR) are based on past experience and the estimated complexity for isolating and correcting a failure in the various elements.

Figure 6.5 points out the major redundant elements of the FMP. It should be noted that no redundancy is shown in the connection network. The reliability/availability analysis assumes a single layer network. The connection network presented in Chapter 5, is double layer network. A double provides some unknown level of redundancy since one of the purposes of the double layer is to provide alternate paths where blocking occurs between the processors and extended memory. At least some failures will appear as blocking to the network. Therefore some degree of redundancy (or fault tolerance) is available in the double layered network. Since the degree of redundancy from the double layer network cannot be identified and taken into consideration in these analyses, a single layer network and the component count of a single layered network was assumed.

The failure rates of the individual FMP elements were determined by using a tentative parts list for each element. The quantity and failure rates for each component are then applied to straight forward calculations which result in the element failure rate (or mean time between failures). Appendix E contains the figures listing the data and the resulting element failure rates. The failure rates of these elements and their estimated mean time to repair are then used with the DESIGN Program, described in Appendix D, along with other factors to be described, to predict the FMP reliability and availability.

Not all of the factors that impact the reliability and availability of the FMP can be readily delineated. Four factors were selected for which a range of values could be projected and used for the FMP reliability predictions. These four factors which are discussed in the following sections are:

(1) LSI Memory Failure Rate
(2) SECDED Improvement Factors
(3) Ratio of Permanent Failures to Intermittant Failures
(4) Recovery Efficiency

6.2.4.1 LSI Memory Failure Rates

Actual field data on LSI memory failure rates is relatively sparse. Some data is available on 16K devices [8]. Reliability models such as those in MIL-HDBK-217B [9] for predicting device failures generally do not hold for significant increases in complexity and density. A worst probable case (lower bound) failure

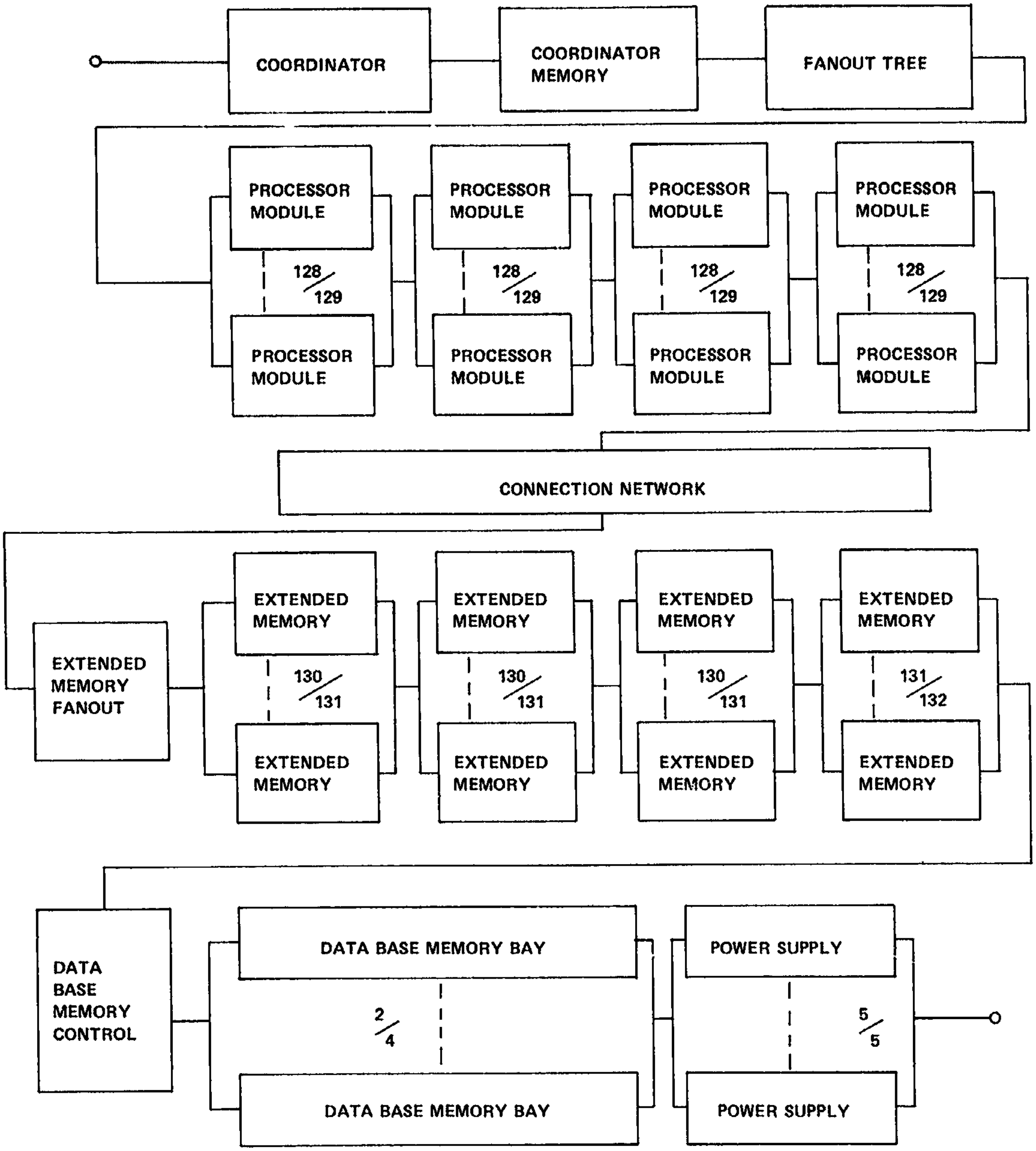


Figure 6.5 FMP Reliability/Availability Block Diagram

rate may be assumed by using the conservative estimate of .4 failure per million hours (FPMH) for a 16K device which is equivalent to the failure rate of four 4K devices (which have an accepted failure rate of .1 FPMH). Using this same philosophy, lower bound failure rates for the 64K and 256K were set at 1.6 FPMH and 6.4 FPMH respectively.

For an upper bound (best probable case), a value of .1 FPMH was set for all three memory devices. Curves showing the improvement of MOS memory devices failure rates with maturity tend to be asymptotic to a value in the range of .1 FPMH regardless of the density [8].

The most probable failure rate was determined using the model in MIL-HDBK-217B for the 16K device and then doubling that value for each quadrupling of memory sizes. This process results in failure rates of the 16K device being .32 FPMH, the 64K device being .64 FPMH and the 256K device being 1.28 FPMH.

6.2.4.2 SECDED Improvement Factor

Improvements in reliability of the FMP are made through the application of Single-Bit Error Detection and Correction and Double-Bit Error Detection (SECDED) in the FMP memories. The mathematical model discussed in Appendix B of reference [2] determined that gains could vary from a lower bound (worst probable case) of 2 to an upper bound (best probable case) of 164 for 16K, 327 for 64K and 653 for 256K memory packages. These two bounds represent the extremes of the probable SECDED improvement. It is anticipated that the real value will fall somewhere within this range. For the purpose of this analysis a value of 50 has been selected as being the most probable SECDED improvement factor.

The SECDED improvement factor is applied to the reliability analysis by direct division of the memory devices failure rates by the improvement factor. Note that application of the improvement factor to the memories circuit alone, however does not consider that SECDED also corrects transient error that may occur from other sources. For example, transient single bit errors occuring in the connection network, or due to software errors or due to noise problems in data being transmitted to a memory may be corrected through SECDED.

6.2.4.3 Ratio of Permanent Failures to Intermittant Failures

Burroughs field data has shown that the ratio of the mean time between permanent failures (MTBF(P)) to the mean time between intermittent failures (MTBF(I)) is estimated to vary over the range of about 10 to 1 to 1 to 1. These values have been selected for the lower and upper bound and the ratio of 5 to 1 selected for the most probable bound. The value of 5 to 1 for the MTBF(P)/MTBF(I) corresponds to the assumption that 5 out of 6 failures are due to intermittents.

6.2.4.4 Recovery Efficiency

The FMP like other large systems should be able to automatically recover from intermittant failures and in some case permanent failures. The system recovery should be designed with the goal of being 100% efficient, that is to say that 100% of the time after an interruption of the system automatically reconfigures and restarts with negligible down time. Unfortunately most systems do not enjoy this idealized goal. Experience shows that recovery efficiency varies and ranges in the levels of 70% to almost 100%. These levels (70% and 100%) were selected as the lower and upper bounds and a level of 80% selected for the predicted level.

6.2.4.5 FMP Reliability Analysis Results

The values of the various factors discussed above were used as inputs to the FMP model and reliability analysis program. Figure 6.6, 6.7 and 6.8 present the input data and the calculated results of this analysis for the lower bound (worst probable case), the probable case and the upper bound (best probable case). The input data include the following:

1) Name: Abbreviated name of an FMP element
2) R: Minimum number of elements required for FMP to be available.
3) N: Number of identical elements available
4) MTBF(P): Mean time between permanent failures
5) MTBF(I): Mean time between intermittent failures
6) SPFM: Single point failures (not used in this analysis)
7) DRT: Device repair time
8) SRT: Single point repair time (not used in this analysis
9) RE(P): Recovery efficiency from permanent type failures
10) RE(I): Recovery efficiency from intermittent type failures.
11) DMRT: Device manual recovery time (assumed to be .1 hours for the FMP)
12) MTBME: Meantime between maintenance errors (not used in this analysis)
13) MTBPM: Mean time between maintenance actions (not used in this analysis)
14) MTTPM: Mean time to perform preventive maintenance (not used in this analysis)

The output data consist of the following three items:

1) MUT: Mean up time
2) MRT: Mean repair time (which for the system being analyzed will be the same as the Mean down time (MDT)
3) Avail: Availability - Percent of time that system or required elements are available for use.

```
R DESIGNT
+RUNNING 0890
NAME            R  N     MTBF(P)  MTBF(I) SPFM     DRT     SRT  RE(P) RE(I)    DMRT    MTBME     MTBPM  MTTPM      MUT    MDT    AVAIL
COOR UNIT       1  1      4672      467  0.000     1.00    0.00 0.000 0.000    0.10        0         0   0.00    424.6  0.182 0.999572
COOR UN MEM     1  1     20963     2096  0.000     0.25    0.00 0.000 0.000    0.10        0         0   0.00   1905.5  0.114 0.999940
FANOUT TREE     1  1      2784      278  0.000     0.25    0.00 0.000 0.000    0.10        0         0   0.00    252.8  0.114 0.999551
PROC MODULE 128129       16497     1649  0.000     1.00    0.00 0.700 0.700    0.10        0         0   0.00     38.7  0.101 0.997397
PROC MODULE 128129       16497     1649  0.000     1.00    0.00 0.700 0.700    0.10        0         0   0.00     38.7  0.101 0.997397
PROC MODULE 128129       16497     1649  0.000     1.00    0.00 0.700 0.700    0.10        0         0   0.00     38.7  0.101 0.997397
PROC MODULE 128129       16497     1649  0.000     1.00    0.00 0.700 0.700    0.10        0         0   0.00     38.7  0.101 0.997397
CONN NET        1  1       188       18  0.000     0.50    0.00 0.000 0.000    0.10        0         0   0.00     16.4  0.135 0.991852
EXT MEM FAN     1  1      2466      246  0.000     1.00    0.00 0.000 0.000    0.10        0         0   0.00    223.7  0.182 0.999189
EXT MEMORY  130131       19025     1902  0.000     0.25    0.00 0.700 0.700    0.10        0         0   0.00     44.0  0.100 0.997731
EXT MEMORY  130131       19025     1902  0.000     0.25    0.00 0.700 0.700    0.10        0         0   0.00     44.0  0.100 0.997731
EXT MEMORY  130131       19025     1902  0.000     0.25    0.00 0.700 0.700    0.10        0         0   0.00     44.0  0.100 0.997731
EXT MEMORY  131132       19025     1902  0.000     0.25    0.00 0.700 0.700    0.10        0         0   0.00     43.6  0.100 0.997714
DBSE MEMCTR     1  1     11910     1191  0.000     1.00    0.00 0.000 0.000    0.10        0         0   0.00   1082.7  0.182 0.999832
DATABASE MEM  2  4          50        5  0.000     0.50    0.00 0.700 0.700    0.10        0         0   0.00      3.8  0.100 0.974524
POWER SUPPLY  5  5       50000     5000  0.000     1.00    0.00 0.000 0.000    0.10        0         0   0.00    909.1  0.182 0.999800

                                                                                                   TOTAL=        1.9    0.11 0.945887449
+ET=1:06.4 PT=3.4 IO=1.7
```

Figure 6.6   FMP Reliability/Availability Analysis - Lower Bound

```
R DESIGNT
+RUNNING 0410
NAME           R  N    MTBF(P)  MTBF(I) SPFM    DRT    SRT  RE(P) RE(I)   DMRT  MTBME  MTBPM MTTPM     MUT   MDT    AVAIL
COOR UNIT      1  1     4672      934  0.000   1.00   0.00 0.000 0.000   0.10      0      0  0.00   778.4 0.250 0.999679
COOR UN MEM    1  1   111324    22265  0.000   0.25   0.00 0.000 0.000   0.10      0      0  0.00 18554.1 0.125 0.999993
FANOUT TREE    1  1     2784      557  0.000   0.25   0.00 0.000 0.000   0.10      0      0  0.00   464.1 0.125 0.999731
PROC MODULE 128129     25432     5086  0.000   1.00   0.00 0.800 0.800   0.10      0      0  0.00   163.7 0.102 0.999379
PROC MODULE 128129     25432     5086  0.000   1.00   0.00 0.800 0.800   0.10      0      0  0.00   163.7 0.102 0.999379
PROC MODULE 128129     25432     5086  0.000   1.00   0.00 0.800 0.800   0.10      0      0  0.00   163.7 0.102 0.999379
PROC MODULE 128129     25432     5086  0.000   1.00   0.00 0.800 0.800   0.10      0      0  0.00   163.7 0.102 0.999379
CONN NET       1  1      188       38  0.000   0.50   0.00 0.000 0.000   0.10      0      0  0.00    31.6 0.167 0.994737
EXT MEM FAN    1  1     2466      493  0.000   1.00   0.00 0.000 0.000   0.10      0      0  0.00   410.9 0.250 0.999392
EXT MEMORY  130131    117950    58975  0.000   0.25   0.00 0.800 0.800   0.10      0      0  0.00  1500.1 0.100 0.999933
EXT MEMORY  130131    117950    59975  0.000   0.25   0.00 0.800 0.800   0.10      0      0  0.00  1500.1 0.100 0.999933
EXT MEMORY  130131    117950    58975  0.000   0.25   0.00 0.800 0.800   0.10      0      0  0.00  1500.1 0.100 0.999933
EXT MEMORY  131132    117950    58975  0.000   0.25   0.00 0.800 0.800   0.10      0      0  0.00  1488.7 0.100 0.999933
DBSE MEMCTR    1  1    11910     2382  0.000   1.00   0.00 0.000 0.000   0.10      0      0  0.00  1985.0 0.250 0.999874
DATABASE MEM   2  4     3053      611  0.000   0.50   0.00 0.800 0.800   0.10      0      0  0.00   636.5 0.100 0.999843
POWER SUPPLY   5  5    50000    10000  0.000   1.00   0.00 0.000 0.000   0.10      0      0  0.00  1666.7 0.250 0.999850

                                                                            TOTAL=    14.9   0.14 0.990381101

+ET=1:00.8 PT=3.5 IO=1.7
```

Figure 6.7 FMP Reliability/Availability Analysis - Probable Case

```
R DESIGNT
+RUNNING 0384
NAME            R      N     MTBF(P)  MTBF(I)  SPFM   DRT   SRT  RE(P) RE(I)  DMRT  MTBNF  MTBPM  MTTPM      MUT    MDT    AVAIL
COOR UNIT       1      1        4672     4672  0.000  1.00  0.00 0.000 0.000  0.10    0      0     0.00    2336.0  0.550 0.999765
COOR UN MEM     1      1      127798   127798  0.000  0.25  0.00 0.000 0.000  0.10    0      0     0.00   63899.0  0.175 0.999997
FANOUT TREE     1      1        2784     2784  0.000  0.25  0.00 0.000 0.000  0.10    0      0     0.00    1392.0  0.175 0.999874
PROC MODULE 128129             25851    25851  0.000  1.00  0.00 1.000 1.000  0.10    0      0     0.00   40674.0  0.500 0.999988
PROC MODULE 128129             25851    25851  0.000  1.00  0.00 1.000 1.000  0.10    0      0     0.00   40674.0  0.500 0.999988
PROC MODULE 128129             25851    25851  0.000  1.00  0.00 1.000 1.000  0.10    0      0     0.00   40674.0  0.500 0.999988
PROC MODULE 128129             25851    25851  0.000  1.00  0.00 1.000 1.000  0.10    0      0     0.00   40674.0  0.500 0.999988
CCNN NET        1      1         188      188  0.000  0.50  0.00 0.000 0.000  0.10    0      0     0.00      94.0  0.300 0.996819
EXT MEM FAN     1      1        2466     2466  0.000  1.00  0.00 0.000 0.000  0.10    0      0     0.00    1233.0  0.550 0.999554
EXT MEMORY  130131            128555   128555  0.000  0.25  0.00 1.000 1.000  0.10    0      0     0.00 NO EFFECT ON PERFORMANCE
EXT MEMORY  130131            128555   128555  0.000  0.25  0.00 1.000 1.000  0.10    0      0     0.00 NO EFFECT ON PERFORMANCE
EXT MEMORY  130131            128555   128555  0.000  0.25  0.00 1.000 1.000  0.10    0      0     0.00 NO EFFECT ON PERFORMANCE
EXT MEMORY  131132            128555   128555  0.000  0.25  0.00 1.000 1.000  0.10    0      0     0.00 NO EFFECT ON PERFORMANCE
DBSE MEMCTR     1      1       11910    11910  0.000  1.00  0.00 0.000 0.000  0.10    0      0     0.00    5955.0  0.550 0.999908
DATABASE MEM    2      4        5842     5842  0.000  0.50  0.00 1.000 1.000  0.10    0      0     0.00 NO EFFECT ON PERFORMANCE
POWER SUPPLY    5      5       50000    50000  0.000  1.00  0.00 0.000 0.000  0.10    0      0     0.00    5000.0  0.550 0.999890

                                                                                         TOTAL=      76.5   0.33 0.995761119

+ET=58.6 PT=3.2 IO=1.6
```

Figure 6.8   FMP Reliability/Availability Analysis - Upper Bound

Unless otherwise noted, all times are expressed in hours. More discussion on these terms can be found in Appendix D.

Examination of this data shows that the two major areas having greatest potential impact on the FMP reliability are the connection network (CN) and the database memory (DBM). The connection network, which for this analysis is assumed to have no redundancy, has the lowest MUT for both the most probable case and the best probable case. In the worst probable case the connection network has the second lowest MUT, the lowest MUT being that of the data base memory (DBM). Two factors contribute to the low MUT for the DBM in this case; the failure rate of 6.4 FPMH, and a SECDED improvement factor of only 2.

Conclusions from this analysis indicated that redundancy should be implemented for the connection network. Furthermore, special attention should be paid to the design and application of SECDED to the data base memory and in obtaining LSI memory circuits with a failure rate significantly less than 6.4 FPMH.

Table 6.6 summarizes the reliability analysis results for the FMP and shows the values of the factors considered in the different cases.

6.2.5 Support Processor and File Management Subsystems

Figures 6.9 and 6.10 show the reliability block diagrams of the support processors and file management subsystem. The high level of redundancy in these systems contributes significantly to its overall reliability. For the purpose of this analysis, the failure rates of the individual models include hard and intermittant failures. The failure rate data used for the support processor elements and the disk packs and file control ements of the file management subsystem are obtained from current field experience on similar systems. The equipment that might be used in the 1980's, though faster and of greater capacity than that in the field now, is expected to have reliabilities and availabilities that will equal or exceed that of these current systems.

Figure 6.11 lists the data used for support processor subsystem. Computed outputs for the mean up time (MUT), mean down time (MDT), availability and the meantime between interruptions for the individual items and the total support processor subsystem are shown in Figure 6.12. The CONFIGURE program described in Appendix D was used to generate these results. An example of actual field data of a similar system is provided for comparison in Figure 6.13.

	LSI MEMORY FAILURE RATES	SECDED IMPROVEMENT	MTBF (P)/ MTBF (I) RATIO	RECOVERY EFFICIENCY	MUT (HRS)	MDT (HRS)	AVAILABILITY
LOWER BOUND CASE	16K - 0.4 FPMH 64K - 1.6 FPMH 256K - 6.4 FPMH	2	10/1	.7	1.9	.11	.9459
PROBABLE CASE	16K - 0.32 FPMH 64K - 0.64 FPMH 256K - 1.28 FPMH	50	5/1	.8	14.9	.14	.9904
UPPER BOUND CASE	16K - 0.1 FPMH 64K - 0.1 FPMH 256K - 0.1 FPMH	16K - 164 64K - 327 256K - 653	1/1	1.0	76.5	.33	.9958

TABLE 6.6 FMP Reliability Analysis Summary

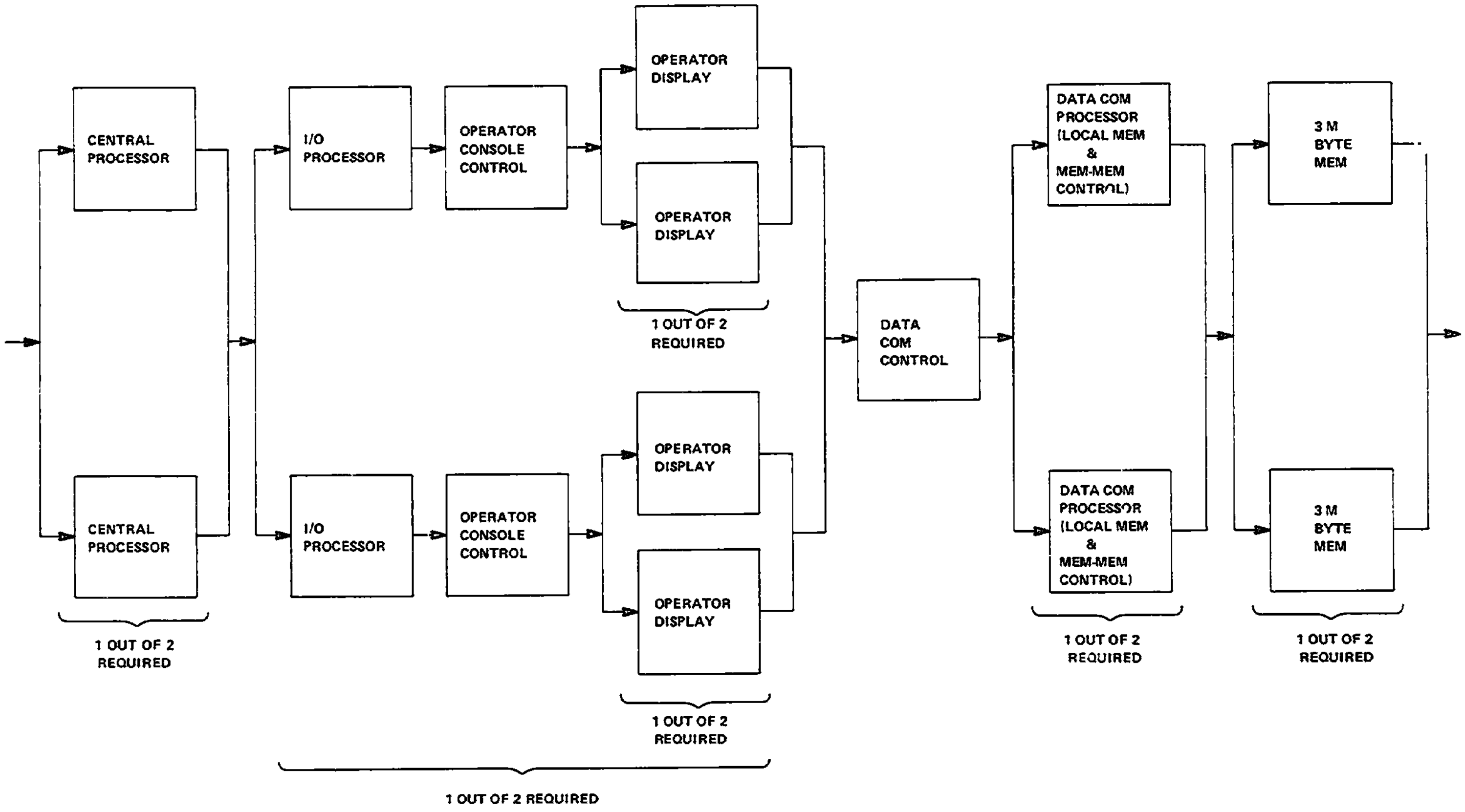


Figure 6.9 Support Processor Subsystem Reliability/Availability Block Diagram

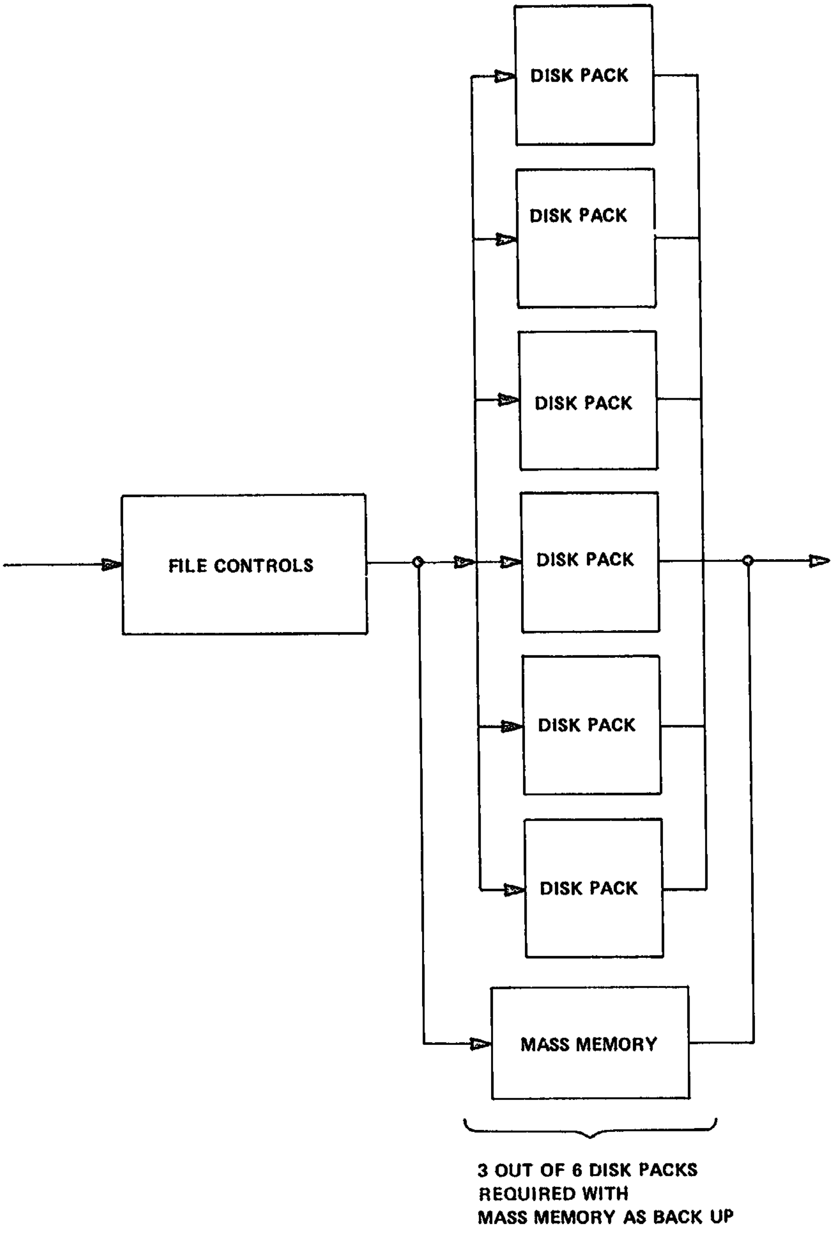


Figure 6.10 File Management Subsystem Reliability/Availability Block Diagram

	MTBF	*MINT*	*MTTR*	*F1*	*F2*
CENTRAL PROCESSOR UNIT CPU	283.00	0.17	3.40	0.20	0.00
MEMORY CNTR MODULE MCM	4153.00	0.26	3.20	0.34	0.00
MEM STORAGE CABINET MSC	1870.00	0.20	0.30	0.19	0.21
INPUT OUTPUT MODULE IOM	387.00	0.34	2.15	0.23	0.00
SINGLE LINE CONTROL SLC	17521.00	0.00	0.50	0.00	0.00
COMPUTER OPER DISPLAY COD	3737.00	0.00	1.31	0.00	0.00
DATA COMM PROCESSOR DCP	2101.00	0.00	1.80	0.00	0.00
DC ADAPTER CLUSTER DCAC	1282.00	0.00	2.25	0.00	0.00

MTBF = Mean Time Between Failures (in hours)
MINT = Mean Time of Interruptions (in hours)
MTTR = Mean Time to Repair (in hours)
F1 = Fraction of total failures which cause an interrupt
F2 = Fraction of total interrupts which require a repair action

The above data is based on actual field experience on similar equipment.

Figure 6.11
Support Processor Reliability Data

	INPUT DATA							COMPUTED OUTPUT			
	R	N	MUT	MTTR(P)	MINT	F1	F2	MUT	MDT	AVAIL.	MTBI
CENTRAL PROCESSOR UNIT CPU	1	2	283.0	3.40	0.17	0.20	0.00	688.0	0.23	0.999671	141.5
MEMORY CNTR MODULE MCM	1	2	4153.0	3.20	0.26	0.34	0.00	6104.4	0.26	0.999957	2076.5
MEM STORAGE CABINET MSC	1	2	1870.0	0.30	0.20	0.19	0.21	4916.9	0.20	0.999959	935.0
INPUT OUTPUT MODULE IOM	1	2	387.0	2.15	0.34	0.23	0.00	833.0	0.35	0.999580	193.5
SINGLE LINE CONTROL SLC	1	2	17521.0	0.50	0.00	0.00	0.00	307002962.0	0.25	0.999999	8760.5
COMPUTER OPER DISPLAY COD	2	4	3737.0	1.31	0.00	0.00	0.00	934234.2	0.00	0.999999	934.3
DATA COMM PROCESSOR DCP	1	2	2101.0	1.80	0.00	0.00	0.00	1228267.9	0.90	0.999999	1050.5
DC ADAPTER CLUSTER DCAC	1	1	1282.0	2.25	0.00	0.00	0.00	1282.0	2.25	0.998248	1282.0
SYSTEM TOTAL								263.0	0.68	0.997416	59.9

R	=	Number of items required for system operation
N	=	Number of items in the system
MUT	=	Mean Up Time
MTTR(P)	=	Mean Time To Repair permanent failures
MINT	=	Mean Time of Interruptions
F1	=	Fraction of total failures which cause an interrupt
F2	=	Fraction of total interrupts which require a repair action
MDT	=	Mean Down Time
MTB1	=	Mean Time Between Incedences (requiring repair actions)

Figure 6.12
Support Processor Subsystem
Reliability/Availability Analysis Results

```
SCHEDULED HOURS:  408   ACTUAL UPTIME:  391   DOWNTIME:   17  DELETE TIME:    0
                                        HOURS      ALL       HARD      SOFT       HS SYS     HSM SYS
HARDWARE EVENTS
  A   ELECT FAIL (REPLACE)               1.50        1          1         0          1           1
  B   ELECT FAIL (ADJ OR REP)            0.00        0          0         0          0           0
  C   ELECT FAIL (CAUSE UNKN)            0.00        0          0         0          0           0
  D   MECH FAIL (REPLACE)                0.00        0          0         0          0           0
  E   MECH FAIL (ADJ OR REP)             0.00        0          0         0          0           0
  F   MECH FAIL (CAUSE UNKN)             0.00        0          0         0          0           0
  H   MAINTENANCE DEFERRED               0.00        0          0         0          0           0
SOFTWARE EVENTS
  J   B SOFTWARE INTERRUPT               0.12        2          0         2          2           2
  M   CUST SOFTWARE INT                  0.00        0          0         0          0           0
MAN-MACHINE EVENTS
  Q   OPERATOR ERROR                     0.00        0          0         0          0           0
  R   MAINTENANCE ERROR                  0.00        0          0         0          0           0
  S   LACK INFO TO ISOL H OR S           0.00        0          0         0          0           0
ADMINISTRATIVE ACTION
  T   ADD/DELETE OF HARD/SOFT            0.15        2          0         0          0           0
  U   DEBUG OF HARD/SOFT                 0.00        0          0         0          0           0
  V   INSTALL OF MOD OR RIN              0.00        0          0         0          0           0
  W   CONVENIENCE OUTAGE                 0.00        0          0         0          0           0
  X   SCHEDULED MAINT(NO CH)            14.50        9          0         0          0           0
OTHER
  Y   ENVIRONMENTAL)                     0.83        1          0         0          0           0
  Z   DESCRIBE                           0.00        0          0         0          0           0

      TOTAL EVENTS                                  15          1         2          3           3

      SYSTEM MEAN UPTIME                         26.06     390.90    195.45     130.30      130.30

      SYSTEM MEAN DOWN TIME                       1.14       1.50      0.06       0.56        0.54

      SYSTEM AVAILABILITY                       0.9581     0.9962    0.9997     0.9959      0.9959
```

HOURS = Total time lost during scheduled period.
ALL = Number of events of all types contributing to downtime.
HARD = Number of events contributing to downtime due to hardware failure.
SOFT = Number of events contributing to downtime due to software failure.
HS SYS = Number of events contributing to downtime due to both hardware and software failures.
HSM SYS = Number of events contributing to downtime due to hardware, and software failures and human induced failures.

Figure 6.13 Example of Actual Field Data on a Large System Similar to the Support Processor

Table 6.7 lists the data used for the reliability/availability analysis of the file management and the subsystem results of this analysis. The MUT and MDT use for the mass memory are based on design specifications.

Table 6.7
File Management Subsystem Reliability
Data and Analysis Results

ELEMENT	R	N	MUT (HRS)	MDT(HRS)	A
File Control	1	1	19,310	1.9	
Disk Packs	3	6	250	1.0	
Mass Memory	1	1	5,246	1.8	
System Total			19,310	1.9	.9999

R = Required number of elements
N = Number Available
MUT = Mean Up Time
MDT = Mean Down Time
A = Availability
(Data from experience on similar equipment or design specifications)

6.2.6 Maintenance

6.2.6.1 Maintenance Philosophy

Maintenance of the FMP should be based on a remove and replace-with-spare philosophy at the lowest replaceable unit (LRU) level as determined by the maintenance analysis. Repair of the replaced failed items would be off-line using subassembly testers available at the site. The FMP should be equipped with fault detection circuits that, in conjunction with system confidence checks and diagnostics, would provide indications of an existing problem via a printout or status display. Errors can be logged automatically giving appropriate file information for isolation of failure(s). Upon detecting a fault, the maintenance personnel can initiate the isolation action required (hardware/software/manual diagnostics) to locate the fault to the malfunctioning subassembly within an element or LRU. A replacement subassembly will be withdrawn from spares and substituted in the FMP element. Before restoring the element to an active status, a confidence check would be performed to determine if the failure has been corrected. The malfunctioning subassembly can then forwarded to the appropriate repair facility (site, depot or factory). Upon repair at the site, the LRU can be returned to the spares stock.

The remove and replace philosophy requires that adequate spares be stocked on-site to preclude degradation of the FMP performance parameters (MUT, MDT, Availability). The actual quantity and types of spares required and the lead times should be determined from their actual usage in the equipment and their individual failure rates.

Preventive maintenance of the FMP consist of periodic testing of the power supplies, checking of rotating memories and general housecleaning. This effort would be minimal, and most of it can be accomplished on-line.

6.2.6.2 Maintenance Plan

Upon detection of a failure the system diagnostic can be automatically initiated to determine the malfunctioning element. The system automatically reconfigures under program control replacing the malfunctioning element if it is redundant. Maintenance diagnostics can be initiated to isolate the failure to the malfunctioning subassembly for removal and replacement by a spare and the process manually restarted if the failed element is not redundant. The design approach being investigated would allow removal/replacement of redundant modules with power-on. This approach would tend to reduce the equipment downtime by allowing more rapid access to the failed items.

Since SECDED is applied to the memories, most single-bit errors will not cause any equipment failure. When the log shows that a single bit is stuck, the system could be shut down when desired in an orderly fashion for maintenance action. This feature would provide a minimum loss of productive time. The information stored in the log could then be processed on an as-called basis for location of the failure or error. The system diagnostic would utilize the following means for error detection and error correction:

a) Processor Module
- Parity check on Microprogram Memory
- Reasonableness checks (See Appendix C for detailed list)

b) Data Base Memory
- Error correction with logging of errors for detecting repeated faults

c) Connection Network

- Error correcting codes as part of the data plus parity checks on address and instructions to memory

d) Coordinator

- SECDED in the memory
- Reasonableness checks (See Appendix C)

e) Memories

- SECDED

f) Power Supplies

- Detection of over voltage on input line will cause the FMP to automatically shut down to prevent damage to the equipment

- Detection of voltage out of range on output

6.2.6.3 Personnel Support Requirements

Detection, isolation, repair and checkout of a failure in the NASF System requires individuals with knowledge and experience of digital processing equipment. These individuals should have a thorough understanding of electronic principles, systems logic and solid state component operation as applicable to high speed digital data processing equipments. They should also have a thorough understanding of electronic test equipment operation, and reading schematics, logic, wiring diagrams and blueprints. Their background should include, at a minimum, a high school education and training in an advanced electronics digital data processing and computer maintenance course. Maintenance personnel should possess experience in the installation, repair, overhaul and modification of high speed digital data processing systems and be familiar with the test equipment applications associated with the accomplishment of these tasks.

An analysis has been conducted to ascertain the level of manpower required to provide for repair and maintenance of the FMP. The results of this analysis shows that to have a 95% confidence a meeting the required actions within the times allocated a minimum of 13 maintenance personnel, each working 5 shifts per week are required.

Estimates of the personnel support (labor hours) requirements for the NASF System provided in this section assumes the type of maintenance personnel described above. The estimates are based on 95% upper confidence bounds applied to element failure rates and the weighted average repair time of a given subsystem. (An upper confidence bound of 95% applied to the element failure rates means that for 95% of the time the failures of a given element will be within this bound.) The basic steps followed to determine these estimates are:

a. Determine the average number of failure expected in a 168 hour operational week ιor a given element in a given subsystem.

b. Determine the expected number of failures at the 95% upper confidence bound for each element and the corresponding subsystem total.

c. Determine the weighted average equipment repair time for the given subsystem.

d. Determine the labor hours expected at a 95% confidence level to be expended in performing corrective maintenance (CM) (on-line-localization, isolation, LRU removal and replacement).

e. Estimate the labor hours for performing preventive maintenance (PM).

f. Estimate the labor hours for LRU repair off-line (bench repair).

g. Estimate the total labor hours required per shift.

Steps a and b

Table 6.8 shows the average number of repair actions expected weekly as computed for each equipment in the FMP, File and Support Processor subsystems. The weekly (168 hours) period was chosen because it best satisfies operational conditions. The smallest value of n; that satisfies the Poisson formula condition given in equation 6.1 determines the maximum number of repair actions at 95% confidence for the jth element.

$$e^{-m_j} \sum_{i=0}^{n_j} \frac{(m_j)^i}{i!} \geqslant 0.95 \qquad (6\ 1)$$

where $m_j = \frac{\text{Qty(j) x t}}{\text{MTBF(j)}}$ = average number of repair actions for jth element in time t

Table 6.8 shows the input values of Qty(j) and MTBF(j) for each of the j elements and the calculated values m; and n; for t=168 hours. The subsystem total for the FMP subsysteπ ؛hows an average of about 11.5 repair actions per week, but at a 95% confidence level there will be no more than 33 repair actions per weeks. The corresponding values for the file and support processor subsystems are about 12 and 37 repair actions per week for the average and 95% confidence bound respectively.

Step c

Determine the weighted average equipment repair time for the given subsystem.

The mean time to repair for each of the j elements is tabulated in Table 6.8 as MTTR(j). The mean time to repair a subsystem, $\overline{\text{MTTR}}$, is obtained as a weighted average of the MTTR(j). The weighing is done using the quantity Qty(j), and inversely as the mean time between failures, MTBF(j), of the jth element since these determine the frequency with which repairs of the jth element comes up for repair. The appropriate formula for the subsystem mean timeto repair $\overline{\text{MTTR}}$, for corrective maintenance is:

$$\overline{\text{MTTR}} = \frac{\sum_j \frac{\text{MTTR(j)} \times \text{Qty(j)}}{\text{MTBF(j)}}}{\sum_j \frac{\text{Qty(j)}}{\text{MTBF(j)}}} \qquad (6.2)$$

When the values in Table 6.8 are used for j=1 to 10, the weighed average equipment repair time (for corrective maintenance) for the FMP is 0.618 hours, and for j=11 to 20, the mean time to repair for the file and support processor is 2.2 hours.

Steps d thru g

A good approximation to the distribution of mean repair time is a normal distribution. Thus, it then follows that the general equation for determining the manpower, personnel hours PCM, for corrective maintenance expected to be expended at 95% confidence can be expressed as:

$$P_{cm} = \left(\overline{\text{MTTR}} + \frac{1.69485\sigma}{\sqrt{n}} \right) n\,P \qquad (6.3)$$

P = Number of maintenance personnel

n = Number of repair actions

σ = Standard deviation of repair times

= 0.25 hours (as determined from observed data taken on similar equipment

Table 6.8

Number of Repair Actions Per Week for NASF System Elements

					Number of Repair Actions	
j	FMP Elements	Qty(j)	MTBF(j)	MTTR(j)	Average Per Week (m_j)	At 95% Confidence (n_j)
1	Coordinator	1	10295	1.00	0.01632	1
2	Coordinator Memory	1	94547	.25	0.00178	1
3	Fan Out Tree	1	8542	.35	0.09035	1
4	Processor Module	512	26620	1.00	3.23077	8
5	Connection Network	1	131	.50	1.28244	4
6	Extended Memory Fan Out	1	8613	1.00	0.01921	1
7	Extended Memory	521	31967	.25	1.06802	4
8	Data Base Memory Control	1	32931	1.00	0.00510	1
9	Data Base Memory Bay	4	117	.50	5.75346	11
10	Power Supply	5	50000	1.00	0.01680	1
			Total		11.48453	33
	File Management and Support Processor Elements					
11	File Control	1	19310	1.9	0.00870	1
12	Disk Pack	6	250	1.0	4.03200	9
13	Mass Memory	1	5246	1.8	0.03207	1
14	Central Processor	2	120	4.83	2.80000	7
15	I/O Processor	2	246	4.06	1.36585	5
16	Operator Console Contr.	2	4000	1.5	0.08400	2
17	Operator Display	4	1149	1.6	0.58486	3
18	Data Com Control	1	1250000	0.5	0.00013	1
19	Data Com Processor	2	115	0.6	2.92174	7
20	3M Byte Memory	2	8737	3.2	0.03846	1
			Total		11.86776	37

*MTBF and MTTR figures used here are earlier data than that used in Sections 6.2.4.5 and 6.2.5 for calculating system and FMP availabilities.

Table 6.9

CORRECTIVE MAINTENANCE LABOR HOUR ESTIMATES

Subsystem	No. of Maintenance Personnel, (P)	No. of Repair Actions wk/(n)	Labor Estimates at 95% (Maintenance Personnel Hours/Wk), (P_{CM})
PCM			
FMP	1	9	6.7949
	2	16	23.0630
	3	8	18.3193
	Subtotal:	33	48.1772
Support Processor and File Systems	1	10	23.3019
	2	19	88.6648
	3	8	56.2929
	Subtotal	37	168.2596

Experience shows that 27.9% of all equipment failure corrective action is performed with one (1) maintenance person, 49.6% with two (2) maintenance personnel, and 22.5% with three (3) maintenance personnel. Substitution of these values for P into equation 6.3 yields the results in Table 6.9 .

The results shown in Table 6.10 are based on the following assumptions: (1) previous field experience for repair-off-line utilizing a subassembly tester indicates a two hours repair time per equipment failure. (2) The amount of time selected for preventive maintenance (PM) are also based on previous field experience. However, it is to be noted that the final time values for PM can be better determined once the PM procedures are devloped.

The final results are adjusted to consider the efficiency of the personnel. An 80% personnel efficiency is assumed to cover contingencies such as set-up times, breaks, report writing and other documentation requirements, etc. These results indicate that, with a 95% confidence, thirteen (13) maintenance personnel can adequately support the NASF Computing System working 5 shifts each during a 21 shift, seven day week, or an average of about 3 persons per shift.

Additional personnel should be considered to account for time off and shift rotation within established personnel policies. The above manning level does not include those personnel required for supervision, administration, software support, system operation and maintenance of the data communication displays, terminals and other I/O equipment.

TABLE 6.10

ESTIMATED NASF MAINTENANCE LABOR REQUIREMENT

Subsystem	Maintenance Activity	Labor Required (Maint. Personnel Hours)
FMP	CM	48.18
	PM	14.00
	Repair Off-Line	66.00
	Subtotal:	128.18
File and Support Processor	CM	168.26
	PM	28.00
	Repair Off-Line	74.00
	Subtotal:	270.00
	TOTAL	398.44
	At 80% Efficiency	498.05 hours/week

6.2.6.4 Sparing Considerations

An important condition to the acquisition and maintenance of any system is the philosophy of sparing parts, assemblies, and subsystems to support the specified system operational requirements.

Sparing considerations are developed as a result of an overall logistics support study which takes into account requirements such as:

- System MUT, MTTR and Availability,
- Redundancy considerations,
- Recovery time of repairables,
- System maintenance philosophy,
- Hardware complexity,
- Corrective/Preventive maintenance skill requirements,
- Site, depot or factory repair,
- Special and standard test equipment or tools, required at the site, depot or factory locations
- Storage facilities (space, environment, etc.),
- Distance from spare part supply points,
- Turn-around time for repair on site, depot and factory ("Pipeline" time),
- Packaging for long term storage,
- Shelf life,
- Long term availability of discrete parts due to technology advances, etc.
- Cost tradeoff studies of repair at piece part versus assembly/subsystem level on throwaway,
- Identification of wear out items replaced at specific intervals.

The maintainability characteristics of any system backed up by the reliability, availability and performance requirements determine the system effectiveness, logistics supportability and the cost of system maintenance.

As new systems are developed, they become more complex with respect to the sophistication of new state-of-the-art circuitry and the application and density of circuitry within equipment elements.

Complex and large systems generally have inherently low mean up times, therefore, a viable logistics support plan becomes a prime factor in the operation of such systems.

As various elements of the NASF system become defined, final part types, part quantities and catagories ultimately selected and circuit packaging determined, a realistic and comprehensive logistic support/spares study can be performed on the FMP support equipment.

Spares are determined through a quantitative analysis which basically utilizes item failure rates, item population in the system and applies various confidence levels to meet the needs of the customer requirements and the established qualitative and quantitative maintenance requirements. Burroughs maintains computerized programs for establishing at the desired confidence level spare part support quantities. These programs can be used to establish the spare quantities for the FMP once the subsystem becomes better defined.

Spare parts selected for site maintenance consideration fall into three basic categories, namely, electronic and mechanical piece parts, subassemblies, and assemblies classified as site repairable in accordance with the established maintenance philosophy.

Typical part types in these three categories are:

Piece Parts:
Fuses
Integrated Circuits
Diodes
Resistors
Capacitors
Switches
CRT's
Connectors
Pin & Socket contacts
Indicator lamps & LEDs
Blowers
Drive Belts
Motors
Misc. parts (wire, etc.)

Subassemblies:
Individual Processors
Printed Circuit Cards (logic)
Power Supply regulator cards
Miscellaneous Printer subassemblies
Misc. Tape, Disc & Display subassemblies

Assemblies:
Power supplies (main)
Keyboards
Miscellaneous Printer assemblies
Misc. Tape, Disc & Display assemblies

Spare parts required for depot or factory repair support will be dependent on specific items identified through future support efforts. In addition to sparing parts required for consumption at the depot or factory level, additional items will be required to maintain site levels of high value and/or non-reparable items for maintaining the "pipeline" flow to the site.

CHAPTER 7
FMP TIMING SIMULATIONS

7.1 FMP MODEL

The FMP Model (Figure 7.1) includes Extended Memory (EM) Connection Network (CN), Coordinator (CR), and one or more processors each including one Execution Unit (EU), and two memory modules. Synchronizing signals between CR and EU's are modeled, as are the effects of CN and EM characteristics on EU instruction times.

The time resolution of the simulation is a single processor clock, nominally 40 ns for a 25 Mhertz clock.

The simulation is detailed to the single processor and coordinator (CR) instruction with sufficient accuracy in the models of the various functions to give good estimates of the execution times of actual code samples. The detail required is greatest in the processor, where it nearly equals that to which the design has been carried. The coordinator (CR) is modeled less completely, but well enough to model instruction-level execution of code with reasonable accuracy.

The EM and CN are modeled only to the extent that their performance parameters are accounted for in the timing of the instructions which use them.

7.1.1 Processor Model

Figure 7.2 shows the functions modeled in the processor. The way these functions perform is best shown by tracing the steps in executing instructions.

It is necessary in some cases to distinguish between functions performed by the simulator, which use no simulation time nor resources, and are indicted by (S), and the functions which take time and/or resources and which correspond to actual hardware functions, indicated by (M).

The simulated code file (S) contains one entry for each instruction of the actual code modeled. The PCR (S) points to the next entry of the simulated code, and this entry when fetched (S) points to a coded description of the instruction (S). This description is fetched and decoded as soon as the previous instruction starts executing. The coded information includes:

(1) Instruction length (code space taken)
(2) CU synchronizing action, if any
(3) Resources used
    (IP, FP, DM, CN buffer/CN)
(4) Time of use of each of resources
(5) Reporting information if a floating point arithmetic
    instruction.

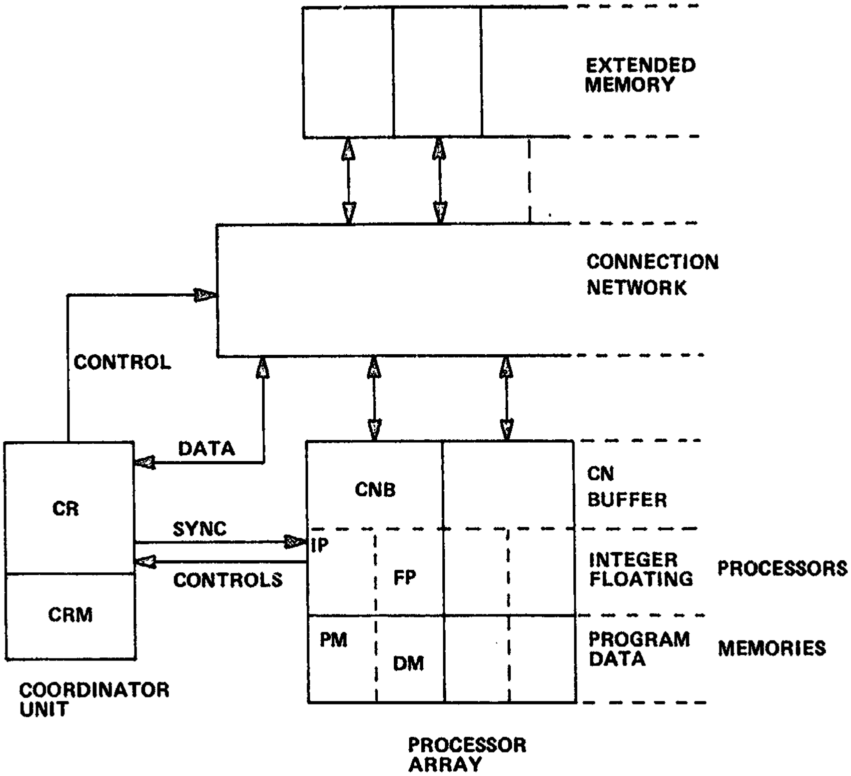


Figure 7.1 Flow Model Processor,
Showing Functions Included in Simulation Model

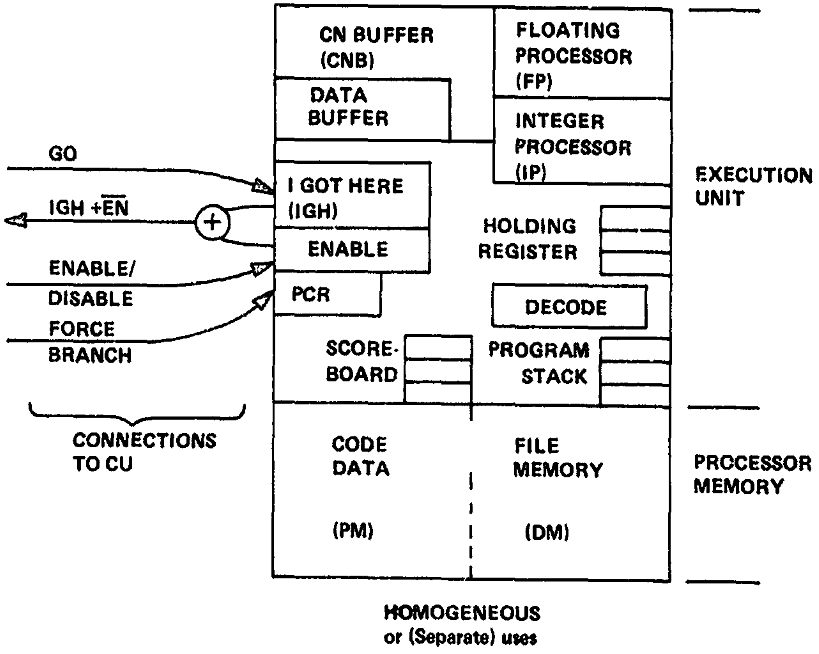


Figure 7.2
Functions Simulated in Processor Model

After fetching and decoding (S) the instruction, the actual behavior of the processor in fetching, decoding, and executing the instruction is modeled.

7.1.2 Program Fetch

The processor memory is modeled as static memory, with three clocks access time, that is, new data is available at the output three clocks after a new address is supplied, and remains available statically so long as the address is held. The actual program address register is not modeled, but it is assumed that the next program address is supplied as soon as the previous program word is fetched to the program stack, so the next program word is available three clocks later. The initiation of program fetches is therefore driven by the availability of space in program stack (M). The space that was occupied in program stack by an instruction (M) (as specified in its description) becomes available as soon as it starts executing, and when the total space available in program stack is enough, a code word (M) is transferred to it, and the next program fetch is initiated.

The above program fetch sequence is subject to two exceptions: When a jump or a conditional branch is executed, the program stack is marked empty, and the program fetch then in progress is restarted, so that the new program word is not available for execution for three clocks. Furthermore, this action itself does not begin until the branch instruction has been executed. The latter also applies for a test-and-branch instruction when the branch is not taken; the next instruction cannot start executing until the test is completed. Alternatively, the model may be altered so that program fetch delay occurs when the branch is <u>not</u> taken, with program execution from the new address continuing without delay when the branch is taken.

The second exception case for program fetch can occur only when the model of processor memory is made homogeneous, that is, both modules are shared between program and data storage. In this case a data access to one of the modules aborts the program access then in progress, and the next program word from that module is not available for two memory cycles (6 clocks). If the data access and the transfer to program stack are simultaneous, the transfer is completed without delay, so the maximum program fetch delay which can be caused by a single data access is five clocks. The module for data access is selected at random (S). When memory is used for data fetch or store, it is treated as a single resource, not accessible when busy, even in the case that both memory modules may be used for data storage and are independently accessible. This is because the memory addresses are always modified by an integer register, so the actual address and thus the module to be used cannot be known until after the instruction starts execution.

When memory is modeled as homogeneous, program fetches alternate between the two modules, but only a single program address register is assumed, so the program fetches from the two modules are initiated simultaneously, and the next fetch from the first module cannot be initiated until the fetch from the second module is complete.

7.1.3 Instruction Execution

After the instruction is decoded (S), and the resources needed and the times when their use starts have been determined, the scoreboard (M) is examined to determine whether the instruction must be delayed. The scoreboard, updated when each instruction starts executing, contains the time at which each resource will be released. If it is found that there will be a delay, further (S) processing waits until the resources are available. Then the content of Program Stack is examined, and if the operator syllable(s) are not present, the instruction queues (S) until the next program fetch makes the syllables available. When the instruction starts, if its use of any of the resources is specified as delayed, then that part of the instruction must wait in the proper Holding Register (M). If the required Holding Register is in use by the prior instruction, then the start of execution is delayed until the holding register is available.

Note the reversal of the actual order of operations: The instruction cannot actually be decoded until it has been fetched, and any waiting for resources must follow this. However, we wish to know how much the execution of code is delayed by program fetch, so we do not count any fetch time during which the instruction would have been waiting for resources anyway.

The actual processor probably would not use a scoreboard as above, because this mechanism for controlling instruction overlap is not fully effective when the execution time for instructions is data dependent, as will probably be the case for arithmetic operations. A mechanism similar to the holding registers would be used, where the various parts of the instruction can wait for their resources. An example of the difference in timing in the two cases is shown in the timing charts of Figure 7.3. Here the instructions using the Integer Processor (Numbers 2, 3, 5) start much sooner in (b) than in our model (a). This is because when an instruction enters the queue, the next instruction is available for decoding, whereas in (a), when an instruction is delayed by the scoreboard, the decoding hardware is tied up until the instruction starts. However, note that in both examples the Floating Processor is busy full time, and the FP instruction (4) starts at the same time in both. It is reasonably clear that the queueing mechanism will allow more instruction overlap in some sequences, giving a reduction in execution time, but such cases will be uncommon and only a small reduction in total execution time can be expected.

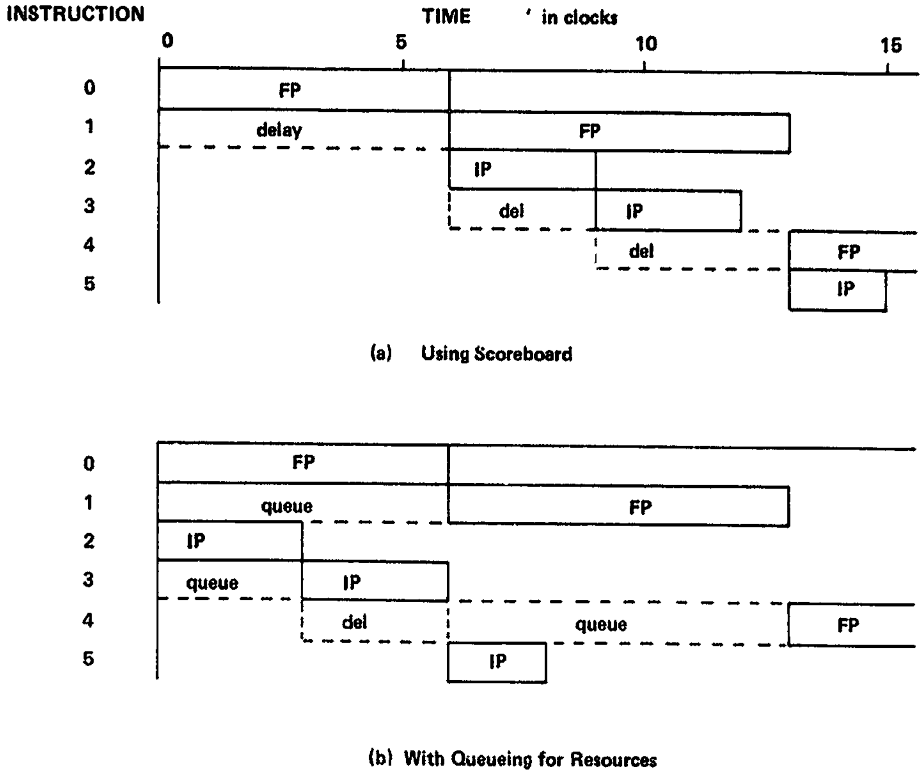


Figure 7.3
Simple Instruction Timing Diagram,
Contrasting Scoreboard and Queueing Implementations
of Instruction Overlap

The IP, FP, DM and CNB resources are modeled, and utilization of these resources is reported. Program memory (PM) is modeled as two separate resources when the memory is homogeneous, but is shown to be utilized only during the memory cycle time actually used for access. That is, neither cycles aborted by data access to that memory, nor the time spent holding output while waiting for space in the program stack are counted as program memory utilization time.

A running count (S) is maintained of the number of instructions executing, this count being updated whenever an instruction starts or ends. Special resources (S) are used causing reports of the percentage of the simulated execution time that 1, 2, or 3 instructions are executing concurrently.

7.1.4 Synchronizing Action

The state of synchronization of the processor is described by the state of two flipflops (M): I GOT HERE (IGH), which is set by certain instructions, and ENabled (EN), which is set whenever the processor is enabled, and when reset causes the processor to stop executing before the next instruction. A logic level formed by the logic combination (IGH or $\overline{\text{EN}}$), ANDed with the same logic level from all other processors is transmitted to the coordinator (CR). The IGH is reset when a GO pulse is received from the CR and EN is set by an ENABLE pulse from CR (M). Cable delays for these signals are zero from CR to EU, and three clocks from EU to CR, because the sys‑em clock is assumed to be in CR, so that signals from CR travel with, and arrive at the same time as the corresponding clock.

The uses of synchronizing action are as follows: Certain instructions (FETCHEM, BDCST, HVST, SHIFCN) involve exchange of data through the CN, under overall control of the CN by CR, with clocked, synchronized data transmission. Therefore, all processors must be at the proper point in the instruction (or disabled) before the data transmission can begin. Such instructions set IGH during execution and then wait for GO from CU before completing in synchronism across the array. Since all such instructions use the CN Buffer (CNB) unit, and the data exchange is via the data buffer internal to CNB the processor can continue executing succeeding instructions as soon as IGH is set, provided that none of them use CNB or set IGH. Certain other instructions (WAIT) set IGH and then wait for GO before starting the next instruction. Obviously, such an instruction cannot start if IGH is already set, but must wait for the GO to reset IGH before going on. The STOP instruction resets EN and then stops instruction execution until the ENABLE from CR again sets EN.

7.1.5 External Access Model

The CN buffer (CNB) unit contains registers for the Extended Memory address, which are loaded from Integer Registers, and a Data Buffer to hold data which is to be transmitted through the CN or which is received from it. The data buffer in CNB may be empty or full, and in either case may be busy or available. However, in our model, the buffer is always available when CNB is available, and is then either FULL or EMPTY depending on the last CNB instruction executed. The CNB functions are designed so that those which transmit data through CN (STOREM, HVST, SHIFTN) specify the processor source for the data (DM, FP register, one or more IP registers), and so appear in several versions. However, those instructions which receive data from CN (LOADEM, FETCHEM, BDCST, SHIFTN) do not specify a processor destination, but leave the data in the FULL data buffer in CNB, from which it is transferred by a second instruction (-REM) which specifies the processor destination. The reason for this is that there may be appreciable delay in using the CN, either by conflicts in CN or at EM for LOADEM, or by waiting for other processors in the synchronized CNB instructions. In either case, the separation of the CNB action and the transfer to processor destination allows the compiler to save execution time and mask the delay time by inserting the CNB instruction as early as possible, followed by as many other instructions as possible before calling for the data.

A CNB instruction (or -REM, which also requires CNB to be available) cannot start while CNB is still busy with a prior CNB instruction. In our model, succeeding instructions, therefore, must also be delayed, although in a queueing model, having a register for queueing CNB instructions, succeeding instructions not requiring any of the same resources might continue execution.

7.1.6 Branching

We model only the execution time of instructions, and the model does not "know" what they do, nor do the modeled instructions contain any data or addresses. Therefore, branching must be controlled by special code words in the simulated code file which do not model any actual instructions but contain coded branch control information to be used by the simulator. Such words can be inserted anywhere, and their processing does not take any time nor resources. However, every time a branch is executed, the simulated code addresses are reported, in order to allow tracing the execution of the first simulated code. The branch controls operate as follows: The first time a branch control is executed, a processor subroutine is initiated to process it. The code word contains algorithms to specify:

(a) The branch address, which may be dependent on processor number if desired,

(b) The repeat number, N, which may be dependent on processor number, and which may be calculated from a probabolistic algorithm if desired,

(c) The kind of branch action desired. Program control may either drop through N times and then branch or may branch N times and then drop through. In either case, when the branch control has been executed N+1 times, the simulator subroutine terminates, and the next time the control word is executed the process is reinitiated.

(d) Special CALL/RETURN constructs are available for subroutine calls.

7.1.7 Coordinator

The coordinator (CR) has its own instruction set and simulated code file. However, the CR is modeled in less detail than the EU, so the simulated instruction description requires only one coded word containing six parameters: Three kinds of synchronizing actions, size of instruction, CR memory action, and CR processor time. The processor is modeled as a single unit, so no instruction overlap is modeled, except for memory access portions. The CR memory is modeled as a single module used for both program and data. The interaction is simplified by assuming that program fetch is initiated only when there is space for the new word in program stack (two words capacity), and once initiated a program fetch is not interrupted by data access. There is usually little contention for memory, since data access in CR tends to be infrequent.

Branch control in CR is implemented in essentially the same way as in the EU, except, of course, there can be no dependence on processor number.

7.1.8 Extended Memory Access

The new Connection Network is used asynchronously, which has important advantages over the synchronous TN when the pattern of EM access is different in different EU's because of data dependent branching, or when the pattern is not a P-ordered vector. However, internal conflicts in the CN, or multiple requests to the same EM module can cause some accesses to be delayed. The delays are determined by the actual pattern and timing of accesses across the entire array. However, within the framework of this simulator it is impossible to model these delays exactly because:

(1) We model only a few processors (usually one), not 512.

(2) We model only the execution time, not the actual data processed, so the access patterns are not modeled.

(3) The simulation model for the CN is complex, so that it is impractical to incorporate it into the FMP model.

Therefore, the CN delay is modeled as a probability distribution. The nominal distribution is exponential, with five clocks expected value. Separate simulations (see Appendix B) of random accesses to Extended Memory through the Connection Network under maximum possible load conditions indicate an average access delay of about one CN clock (three processor clocks). Since the CN simulator has not been run long enough in any test case to reach steady state, we assume that the distribution of delays may have a longer tail than the exponential, so we approximate this worst case by an exponential distribution with four clocks expected value (and four clocks standard deviation). The standard deviation of the average delay for 100 accesses is therefore 0.4 clocks, so that the worst case out of 512 processors would be expected to have an average delay 2.9 standard deviations greater, or 5.2 clocks.

Extended-memory-access instructions are thus modeled with the CN delay added to normal execution time of the instruction. The decoding of the next instruction and its overlapping execution (if possible) is not delayed. The execution of Processor code is delayed by instructions which use the CN buffer only if the CNB is still busy with the last such instruction, since the EM accesses are managed by CNB without interfering with the use of any other processor resources. Delay is minimized by ordering the code so as to interpose other instructions between CNB uses. In particular, the -REM type instruction which uses the data fetched to the Data Buffer in CNB by an EM access, is placed as late as possible in the instruction stream. By these means, the code (FX subroutine) which suffered the largest delay from waiting for CNB was delayed only 11.4% (see Sec. 7.2.6.2, and Table 7.2). In this case, reducing the contention delay discussed above from five clocks to ½ clock increased the throughput only 4.0%.

7.1.9 Simulation Results

The primary information provided by the simulation run is the elapsed time required to run the simulated code, and the number of floating arithmetic operations performed, which together give the throughput in floating operations per second.

Much additional information is reported, such as the total exectuion time of the arithmetic part of floating point instructions, delays caused by branching and program fetch, utilization of the various processor and coordinator resources, and the extent of instruction overlap achieved in the processor(s). This information can be useful in understanding why the processing of the code behaved as it did, and is useful in guiding the details of hardware and software design.

7.2 SIMULATIONS PERFORMED

The codes segments simulated were selected from the Hung-MacCormack explicit and the 3-D implicit aero flow codes, and from a GISS weather code. The criteria for code selection were, first, to select a range of types of codes to cover a wide range of flop throughput and of factors influencing the throughput, and second, to include from each code samples typical of those portions which account for the major portion of the execution time of the program.

By comparing each block of code in an entire program with the code segments actually simulated, it was then possible to estimate throughput for that block, and by proper weighting, to estimate an average throughput for the entire run of each program. These estimates are probably on the low side because the parameters used in the simulation model are conservative:

(a) The assumed 40-ns clock period is ample for ECL logic. This allows safe, conservative logic design, and actual detailed design may show that a slightly faster clock is feasible.

(b) The execution time for arithmetic instructions is assumed constant. If the instruction logic is designed to give data- dependent execution time, the assumed constant value is near the worst case, and the average execution time will be considerably less.

(c) No great sophistication of the compiler is assumed, either in the generation of efficient code or in optimization of register allocation or instruction reordering.

(d) The scoreboard method for controlling the overlap of instruction execution is assumed. As discussed in Section 7.1, this is less effective than the queueing method which would probably be used.

(e) The use of the new connection Network for access to Extended memory will produce some delays caused by contention within the CN or at EM. These delays are difficult to estimate accurately, so a conservative estimate was used. The actual delays in real program runs will probably be considerably less than the simulated value.

7.2.1 Selected Codes

The selected code segments were TURBDA, AMATRX, and a portion of BTRI from the implicit code, LX and a portion of CHARAC from the Explicit code, and parts of AVRX, COMP2, and COMP3 from the GISS weather code. The throughput indicated by simulation of these code samples ranges from 70 MFLOPS for AVRX to 1330 MFLOPS for AMATRX. The simulation results are summarized in Table 7.1, which includes additional information on utilization of processor resources and delays in execution as reported by the simulator. A detailed discussion of this table, and the throughput-controlling features of the several codes follows. The FMP FORTRAN and assembly-language versions of the codes as simulated are given in Appendix G.

Some of the earlier simulations were performed with a model of the Transposition Network, in which accesses to EM are synchronous across the array, and controlled by the CU. Comparison of the CN and TN performance indicate that the average contention delay involved in using the CN is compensated by the fact that EM access instructions through CN can be more completely overlapped because of the buffering action of the associated CN Buffer unit in the processor, and by the use of a much faster implementation of the IMOD521 instruction in the later version of the model. The earlier simulation results therefore remain essentially valid.

7.2.2 TURBDA

This code has four EM accesses (three LOADEM's and one STOREM) in the inner loop, and 28 floating arithmetic operations, of which 18 are concentrated in an in-line Newton-Raphson square root. The somewhat low throughput of 835 MFLOPS is accounted for by three factors:

(a) The Average execution time of 8.1 clocks per floating arithmetic operation is about 10 percent longer than average because of a somewhat higher than average proportion of divides and multiplies.

(b) The integer operations form a higher than average proportion, as shown by the IP use percentage and are not overlapped as much as usual by floating operations.

(c) There are only seven floating operations per EM access.

7.2.3 AMATRX

This section of the implicit code is involved with generating the local five by five matrices to be inverted by BTRI. Each iteration of the inner loop performs 80 floating point operations for five EM accesses (LOADEMs), and 53 local memory accesses. This is a rather favorable case, as shown by the high (84 percent) utilization of the FP unit in the processor. The fact that the EM and local memory accesses are performed with no more than about 20 percent loss from the maximum theoretical throughput of 1680 MFLOPS for the per-FLOP time of 7.6 clocks indicates the effectiveness of the instruction overlap.

7.2.4 BTRI

A representative portion of the BTRI subroutine was hand compiled for this simulation. About 77 percent of the floating operations are concentrated in an inner loop, which is a 5 by 5 nested DO loop with only 10 floating operations and 7 indexed local memory fetches in the inner loop. BTRI runs slower than might be expected for a subroutine with no EM accessing because:

(a) The indexing of the local arrays causes a large number of integer operations which cannot be entirely overlapped by the few FLOPs.

(b) Several of the integer operations are large (48-bit) instructions, but with short execution time, so that they use up code faster than it can be fetched. The result is the indicated 14.2 percent of elapsed time spent waiting for program fetch.

(c) The nested DO loop causes a large number of branches, causing 3.3 percent of time to be spent waiting for program fetch after branch, and further aggravating (b) above because every branch causes any program look-ahead which has been done to be wasted.

Note that some of these inefficiencies could be reduced by unwinding the inner DO loop, which would involve repeating the same brief code section five times. Some of the cost of indexing the local arrays could be saved by reprogramming to store the N different five by five matrices which are generated by AMATRX in a 25 by N array instead of a five by five by N array. This, together with the unwinding of the inner drop, would considerably increase the throughput of BTRI. The loss of program look-ahead on branching can be reduced by implementing fetch on no-branch instead of fetch on branch.

7.2.5 GISS Climate Code Samples

Analysis of the weather/climate code throughput is discussed in Section 3.4.4. The samples selected for simulation were as follows.

7.2.5.1 AVRX

This routine smooths the data in the longitude direction for latitudes near the poles in order to compensate for the too-close spacing of these grid points. The number of iterations of the smoothing algorithm therefore depends on latitude and must be computed. The index computations are fairly complex, and the smoothing algorithm itself is very simple, so there are less than one floating arithmetic operation per EM access, and much integer computation.

Furthermore, the organization into a DOALL in which 26 instances are allocated to each processor in order to leave no processors idle considerably increases the integer computation to be executed in each instance, thus partially defeating the purpose.

The net result is a code sample which is in a sense a worst case for the FMP with floating point throughput of only 70 MFLOPS.

7.2.5.2 COMP2

Portions of the CORIOLUS FORCE and VERTICAL ADVECTION code were hand compiled and simulated. This code performs only about two floating point operations per Extended Memory access, and the addresses in two- or three- dimensional EM arrays are calculated from the indices with no shortcuts, so that one or two double-precision integer multiplies are required for each calculation.

The result is that integer arithmetic dominates the code, as shown by the COMP2 entry in Table 7.1, where the integer processor is busy 65 percent vs only 40 percent for the Floating Processor. The floating arithmetic also is above average in clocks per operation (10.3), and floating arithmetic is being executed only 30 percent of the elapsed time.

7.2.5.3 COMP3

The portion of COMP3 simulated was LINKHO having no EM accesses. The throughput of 980 MFLOPS indicated by simulation is only 25 or 30 percent less than the practical maximum of about 1300 to 1400 MFLOPS attained when the processors are doing floating arithmetic 80 percent of the time. The COMP3 result is lower because of three factors:

(a) The average floating arithmetic operation takes 9.1 clocks, compared with the nominal average of 7.3, because of a higher proportion of multiplies and divides.

CODE	TURBDA	AMATRX	BTRI	AVRX	COMP2	COMP3
MFLOPS	840	1330	1200	70	380	980
CLOCKS/FLOP	8.1	7.6	6.2	7.8	10.3	9.1
FLOPs/EM access	6.8	16.0	---	(note 1)	0.86	---
<u>Percent use</u>						
FL Arith	53	79	58	4	30	70
IP	40	16	39	76	65	23
FP	58	84	64	7	40	82
DM	9	21	31	7	9	17
PM	41	40	73	40	32	40
Instr. Overlap	1.0	1.2	1.2	.96	1.2	1.1
<u>Percent Delays</u>						
Branch	1.7	0.4	3.3	4.2	1.0	3.9
Prog. fetch	5.2	1.6	14.0	3.8	1.2	1.1
EM Access	NR	1.9	---	11.4	8.3	0.6

○ Significant Factors

Table 7.1 FMP Simulation Results

Note 1. See Appendix A for discussion of AVRX

(b) The FP arithmetic is being done only 70 percent of the time; although the FP unit is busy 82 percent. This is because of a number of non-arithmetic floating register operations such as change sign, move, and local memory access.

(c) Program branching causes delays amounting to nearly four percent of the time.

7.2.6 3-D Explicit Aero Flow Code

In the FMP model used for the simulations shown in Table 7.2 and reported below, the local memory is homogeneous: both modules are shared between data and program. One test run with the processor model having separate program and data memories shows about 5 percent lower throughput because of waiting for program fetch. Circles are used in Table 7.2 to call attention to those items which limited throughput for each simulation.

7.2.6.1 CHARAC

This third level subroutine from the explicit code has no EM accesses, but contains many data dependent branches, and DO loops whose iteration count varies because of data dependent exits. The CHARAC code would therefore be very difficult to vectorize, but presents no difficulty to the parallel machine, although of course the tests cost time.

The section of CHARAC code (shown in Appendix H) which was simulated consists of a DO loop on JC and a portion of the code following the JC loop. The JC loop is preceded by several local memory accesses to save local registers, and within the loop are 24 floating point arithmetic operations. It is exited by the AND of two comparisons of floating variables. The JC loop contains an inner DO loop on JJ, which performs only integer operations, and is exited by the AND of two comparisons of floating point variables.

Three simulations were performed, varying the JC and JJ counts, as shown in Table 7.2:

(a) JC loop performed eight times, with JJ performed six times in each. This gives the low throughput of 900 MFLOPS.

(b) JC loop performed 15 times, with JJ performed once in each, giving throughput of 1180 MFLOPS.

(c) Same as (a), but with JJ loop reprogrammed in FORTRAN to use only one rather than two comparisons of floating variables to decide the exit from the loop, and with the new JJ loop coded for maximum efficiency by hand, using tricks a compiler might be smart enough to use. The throughput of this version is 990 MFLOPS, or 10% more than version (a), because of the 40% reduction in running time of the recoded JJ loop. If the JJ loop is performed fewer times, the throughput will approach or slightly exceed case (b).

Table 7.2

Summary of Simulations of EXPLICIT CODE

CODE	CHARAC			LX/FX	LX	FX	SQRT
	(a)	(b)	(c)				
MFLOPS	900	1180	990	570	530	590	1500
CLOCKS/FLOP	8.0	7.9	8.0	8.3	8.4	8.4	7.0
FLOPS/EM Access	---	---	---	3.6	2.8	4.1	---
Percent use							
FL Arith.	56	73	62	37	34	39	81
IP	33	24	30	48	57	42	
FP	61	78	67	46	42	49	
DM	27	27	26	19	20	18	
PM	72	56	65	48	53	47	
CN	--	--	--	12	12	12	
Instr. Overlap	1.07	1.15	1.09	1.13	1.19	1.11	
Percent Delays							
Branch Fetch	9.3	3.6	6.7	4.5	5.9	3.7	
Prog. Fetch	5.8	4.5	6.7	1.2	1.0	1.4	
EM Access	---	---		9.2	4.6	11.4	

○ Significant Factors Affecting Throughput

Nearly half of the JJ loop time is accounted for by waiting for program fetch, both after a branch, and when code is being executed faster than it can be fetched. This is another case where more efficient packing of code or faster access to program memory would appreciably improve throughput

It is interesting to note that in some algorithms the programmer can use arithmetic comparisons and conditional branching to save some arithmetic. In such cases the throughput measure of programs would be more consistent if floating point comparisons were considered to be arithmetic operations; otherwise a program with superior performance might be measured as having lower throughput. If this measure were applied to the three cases of CHARAC discussed above, they would become nearly equal at about 1300 MFLOPS.

Examination of Table 7.2 shows the following important factors affecting throughput of the CHARAC sample.

(a) The average execution time of a floating arithmetic operation is 7 to 8 percent higher than the nominal 7.4 clocks. This is because of a higher than average proportion of divides

(b) Waiting for program fetch, both after branching and in other places accounts for 13 to 15 percent of the elapsed time in cases (a) and (c). The high utilization of Program Memory is not responsible: most of both delays occur in the JJ loop, which uses a good deal of program space while requiring little execution

(c) The utilization of the Floating Processor is low in cases (a) and (c), even allowing for program fetch delays, indicating a good deal of non-overlapped integer computation. Again, this is mostly in the JJ loop, as indicated by case (b) where the JJ range of code is executed only once, and the FP utilization is only about 12% below the values attained in AMATRX and COMP3.

7.2.6.2 LX/FX

The second level subroutine LX from the explicit aero-flow code executes within a DOALL with J and K as domain variables and each instance has inner DO loops on I, with IL or IL-2 iterations. The third level subroutine FX is called in an inner loop that is performed twice, so FX is called about twice IL times in each instance of JK. The simulations were run with the IL and IL-2 loops both performed 10 times, since the computer runs would have been too long with values of 100 and 98. At ten iterations the code in the loops dominates the running time, so there is little error in this approximation. Separate simulations of LX, with FX calls deleted, and of FX code (with no RETURN) were performed. For interest sake, the SQRT code which is present in-line in FX was also timed by using the trace in the simulation output.

The results are shown in Table 7.2. The FX calls contribute about 70 percent of the FLOPS in LX/FX, and the tabulated figures for LX/FX agree with the weighted average between LX and FX figures. The LX/FX throughput of 570 MFLOPS is limited by the factors circled in the table: (1) a mix of arithmetic instructions that gives an average execution time of 8.3 clocks or about 12% more than average, (2) a high usage (48%) of the integer processor, (3) appreciable delay (4.5%) for program fetch after branch and (4) 9.2% of the running time is spent waiting for extended memory access.

LX is structured in such a way that much of the EM data for the IL loops is pre-fetched to local arrays in a DO loop of IL iterations which performs no floating point arithmetic, and similarly, at the end the local array of results is written back to EM. These pre-fetch and post-store portions of LX take 12% of its time (not counting FX). Similarly FX was coded to precalculate and save indices and local variables used repeatedly in the code, and this, together with save and restore of registers used in FX takes 13% of FX time. The rest of LX and FX appear to be normal code, with no more than average amounts of pure integer operations and looping and branching, so that the results should be considered normal for the flops-per-EM-fetch ratio of these codes.

It is clear from the LX/FX simulations that at their rate of EM accesses about half the execution time is spent doing the EM accesses and the integer computations of EM addresses. As an experiment, a simulation was run with the average delay caused by contention in CN and EM reduced to 1/2 clock, as would be expected for the actual average loading of CN (12%). This reduced the running time of FX by 4.0%. In a second experiment, the execution of times of double precision integer arithmetic were reduced to the values estimated for single precision in a 32 bit integer arithmetic unit. This produced a further 6% reduction, for a total of 10% reduction with both changes, or an FX throughput of 650 MFLOPS.

7.2.6.3 SQRT

A new square root macro was programmed, using recently added integer/floating transformation instructions (FIX, FLOAT, ADDEX, MOVEX) to address a local memory table for a first approximation good enough so that only three iterations are necessary. The resulting SQRT has 14 flops, runs in 119 clocks, and has a throughput of 1500 MFLOPS for the array. The entry in Table 7.2 is incomplete because SQRT was not simulated by itself, the values shown being extracted from the trace of the FX simulation. This SQRT accounts for 10% of the flops of LX/FX. The SQRT found in TURBDA was an earlier version.

7.3 APPLICATION OF SIMULATOR RESULTS

The above simulation results from the basis for the application analysis summarized in Chapter 3 and described in more detail in Appendix A. The extension of the simulator measurements to those code sequences that were not simulated, is also described in those locations.

Chapter 8
SCHEDULE AND FACILITIES

8.1 SCHEDULE

8.1.1 Introduction

Realistic scheduling of a large program such as NASF requires the systematic definition of the tasks to be performed to levels for which reasonable estimates can be made. With each successive level of detail the time estimates become more accurate. For the purposes of this study only the first level has been estimated for the total effort. It therefore must be considered tentative. Second and third level schedules have been prepared in specific tasks areas to demonstrate the refinements that ultimately must be prepared for the total effort and to illustrate the management tools that can be used to monitor, analyze, and control the program schedule.

8.1.2 The Overall NASF Program Schedule

The NASF program schedule presented in Figure 8.1 is based on a number of factors and assumptions. It is assumed that the initial sixteen months is dedicated to the design and final specification effort. After this initial effort final design leading to procurement, tooling and manufacturing will begin. Most of these implementation tasks are of the order of fifteen to twenty one months. The final period of integrations, delivery, installation and testing is estimated at eighteen months. This results in a total program duration of 55 months. The estimates are based on past experience and best judgement. They do not represent either the best or worst case possibilities.

No attempt has been made to define a critical path for this summary schedule. However, critical paths have been determined on schedules of individual activities as will be demonstrated in the examples that follow. The final output of the overall program schedule is shown as the "deliverables".

The NASF schedule has been divided into nine task areas.

1. Program Management
2. Systems Management, Integration and Test
3. Flow Model Processor
4. File Memory Subsystem
5. Support Processor Subsystem
6. System Software
7. User Support Subsystem
8. Facility Engineering
9. System Support

The above breakdown is based on grouping of tasks of similar nature or relating to a major deliverable element. This same breakdown could be used for cost estimating as well.

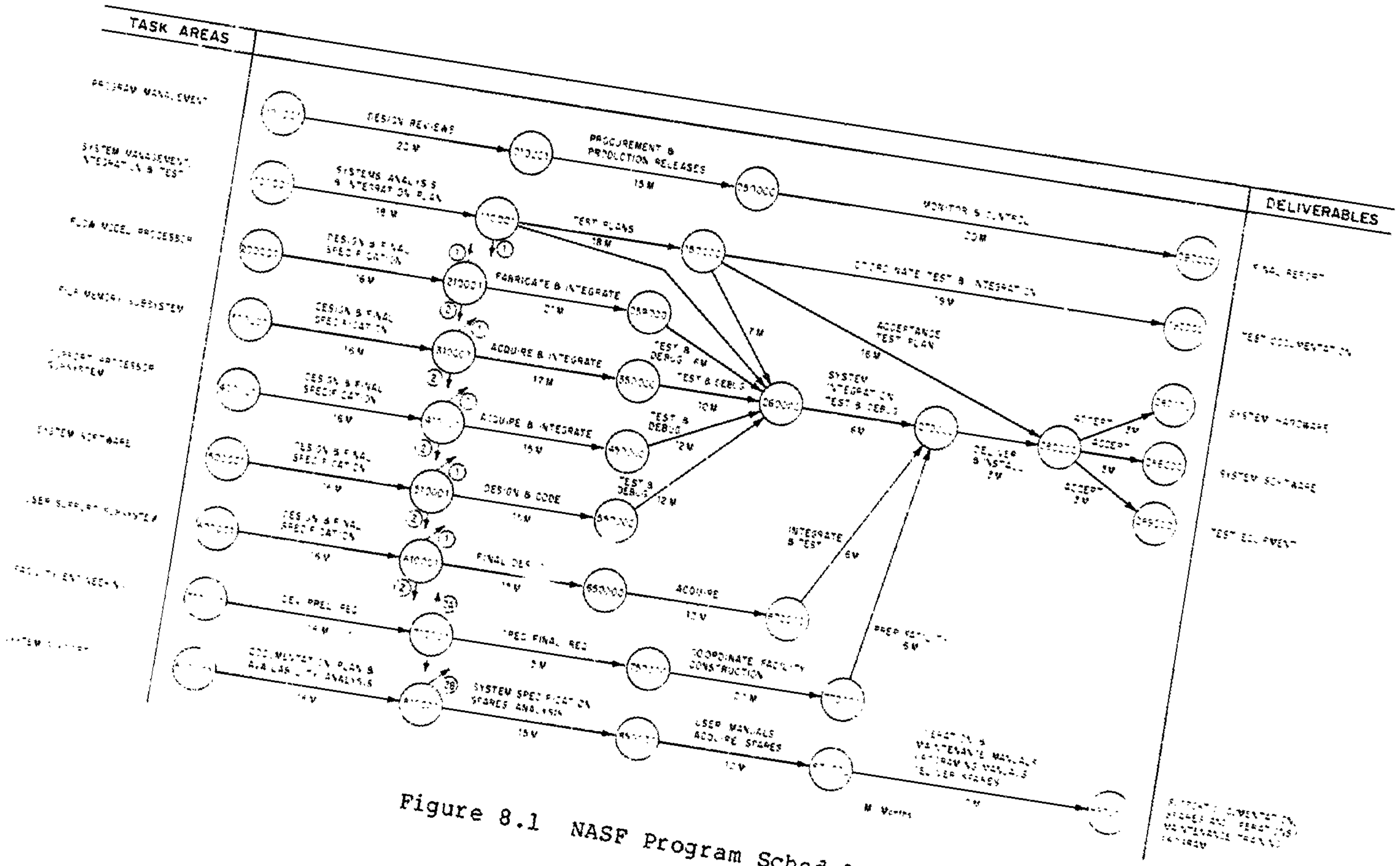


Figure 8.1 NASF Program Schedule

For the most part, the scope of these areas are self evident. Program Management includes the monitoring, review, reporting and control of the overall program activities. In addition, this task includes schedule, cost and configuration control, generation of procurement and production releases, subcontract performance monitoring and liaison with customer representatives. The last area, System Support, covers the tasks relating to reliability, maintainability, human factors, spares, documentation and manuals, and training. Intermediate milestones shown in these two task areas are not major events but represent the bounds of the time periods for certain emphasis.

The schedules presented assumes that most system integration and testing is done on the manufacturers premises. A trade off, depending on the availability of the structure for housing NASF, may show that final integration and testing may be more effectively done at NASA Ames, possibly shortening the schedule.

8.1.3 Schedule Management

The schedules for final design, fabrication, integration and installation of a large system such as the NASF Processing System should be developed on a multilevel basis. The first level should delineate the overall program showing major milestones and "deliverables".

Each activity for these tasks areas may be delineated in more detail in a second level of scheduling. These include such activities as the Integration Plan, or the Fabrication and Integration of the FMP. A third level of schedule detail further delineates the major activities within each of these task areas, such as the design, fabrication and testing of the FMP Processor.

For most planning, schedule control and resource management this level of detail is sufficient. However, fourth and fifth levels are usually desirable for specific hardware, software and documentation items to be produced or for individual personnel or group assignments.

The first three levels are best managed by PERT (Program Evaluation Review Technique) type schedules. In these schedules single events (start and/or completion dates) are depicted as nodes. The activities or tasks to be accomplished are depicted as the interconnecting lines. The linear flow represent sequential and approximate temporal relationships. Where the completion of one task is a prerequisite before the completion or beginning of another task that is not a natural sequence, a "dummy" activity is shown.

The result of this graphic representation of a group of activities, is a network, showing the starting event and activities, the major milestones (events) and activities required to accomplish a desired goal or goals which are in turn shown as the final event(s). The PERT network should clearly depict the interrelationships between various tasks.

Once the time elements are assigned tc the tasks of a network, the critical path can be ascertained. The critical path represents that sequence of activities required for completion of the end objective that requires the longest period of time; that is to say that a single day (or month) slip in any one of the activities in the path, will result in a day (or month) slip in the overall schedule.

One of the many advantages of PERT is that it lends itself to management, maintenance and analysis by data processing techniques. This is readily accomplished by the use of Burroughs PROMIS (Project Oriented Management Information System) which has many useful management outputs. Activities in the critical path are easily identified. The slack in noncritical activities is reported. The range of acceptable start and finish dates is provided. Holidays, overtime and shift work can be made part of the schedule. Flags for sorting of activities by discipline, organization, or other keys can be employed. A PROMIS data base is easily updated permitting rapid assessment of the impact of changes or other new inputs. The use of this tool in initial planning is shown in the discussion of the schedules that follows.

8.1.4 NASF SCHEDULES

Figure 8.1 illustrates the major activities of the nine NASF task areas leading to the achievement of the final program goals (also shown as deliverables). The interrelationships between some of the milestones are shown with arrows. For example, the completion of the final design and specification of the various hardware and software elements are all inputs to the final integration plans and system analysis efforts; the design and final specifications of the hardware items is needed as an input to the activity that will issue the final facility requirements documentation. For each activity an expected time for completion is indicated below the line representing the activity.

Each event is given an identifying number which is used in creating the data base for analysis and reporting. Table 8.1 delineates this numbering system and shows it to the lower levels for certain categories.

TABLE 8.1

NASF Event Identification Numbers

Event Numbers	Task Area
000000 - 099999	Program Management
100000 - 199999 and Test	Systems Management, Integration
200000 - 299999	Flow Model Processor
300000 - 399999	File Memory Subsystem
400000 - 399999	Support Processor
500000 - 599999	System Software
600000 - 699999	User Support Subsystem
700000 - 799999	Facility Engineering
800000 - 899999	System Support
210001 - 219999	FMP Processor
220001 - 229999	FMP Extended Memory
230001 - 239999	FMP Connection Network
240001 - 249999	FMP Coordinator
250001 - 259999	FMP Cabinets and Cables
260001 - 269999	FMP Power Distributor
270001 - 279999	FMP Test System
280001 - 289999	FMP Data Base Memory

To demonstrate the application of management tools for schedule monitoring, analysis and control, the next two levels of schedule detail for specific aspects of the NASF have been defined. The schedule for the fabrication and integration of the FMP which is a major hardware item of the NASF and the final design, fabrication and testing of the processors, which represent a major portion of the FMP hardware have been selected for further delineation. It is quite possible that the critical path for the NASF could be dependent on activities in these two areas.

Figure 8.2 takes the single activity "Fabricate and Integrate" of the Flow Model Processor task area and breaks it down in to the next level of detail. The first node of this schedule corresponds to the second node of the FMP path on the program schedule in Figure 8.1; the last node corresponds to the third node on the program schedule. The first node on Figure 8.2 divides (with no time allocation) into the eight major elements of the FMP.

For scheduling purposes a preferred sequence of integration is assumed. The first point of integration is that of the FMP power distribution system with the FMP cabinets and cables. The schedule then calls for the integration of the coordinator with the use of the FMP Test System (which will include the FMP diagnostic controller). Not all of the FMP cabinets, cables and power distribtuion system are required for the installation, checkout and debugging of the coordinator. Completion of some portions of these can be deferred until required. This level of detail can be included on the next lower level of scheduling.

The installation of the connection network is next followed by the installation integration and checkout of the processors and the extended memory modules. The end events of the processor and extended memory activities are shown as only two tasks for each element, "Install First Processor" and "Install Last Processor" and "Install First Extended Memory" and Install Last Extended Memory". These end events are used in lieu of having 585* individual inputs representing each processor and extended memory module. A rather large series of activities such as the schedules for each of the processors is best handled by a straight forward status list.

Figure 8.3 further delineates the detailed activities for the processor final design, fabricate and test activity shown on Figure 8.2. It will be noted that there are several different paths leading to the availabilty of the 585 processors. The upper most path shows the activities for the design and procurement of the printed circuit board. A second and third path are the activities relating to the design and development of the processor tester and test software. The lowest path which merges with the tester path involves the design, fabrication and evaluation of a prototype processor.

*The current estimates for the number of processors and extended memory modules manufacturing starts is 585 which takes into account shrinkage and spares.

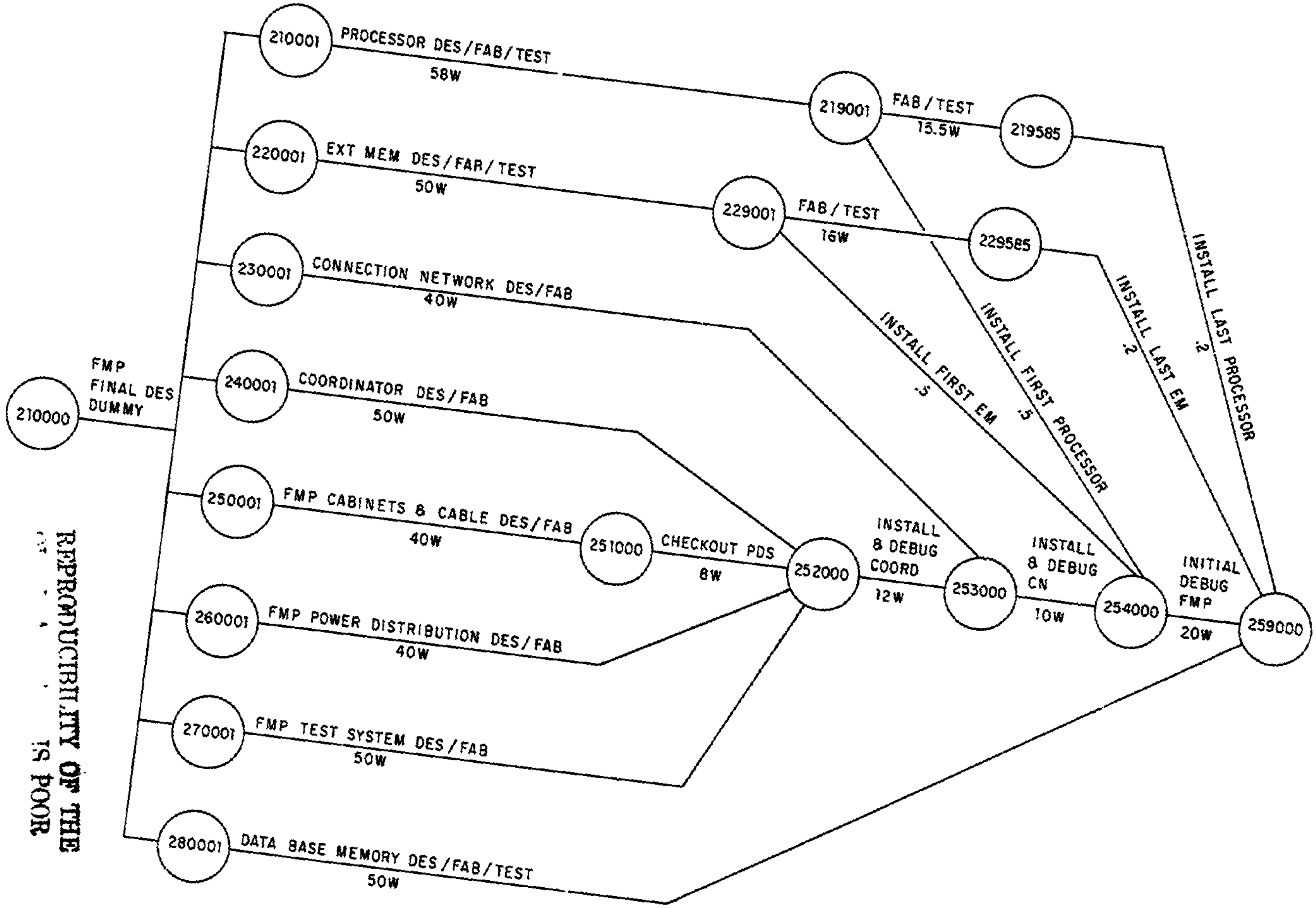


Figure 8.2 FMP Fabrication and Intergration Schedule

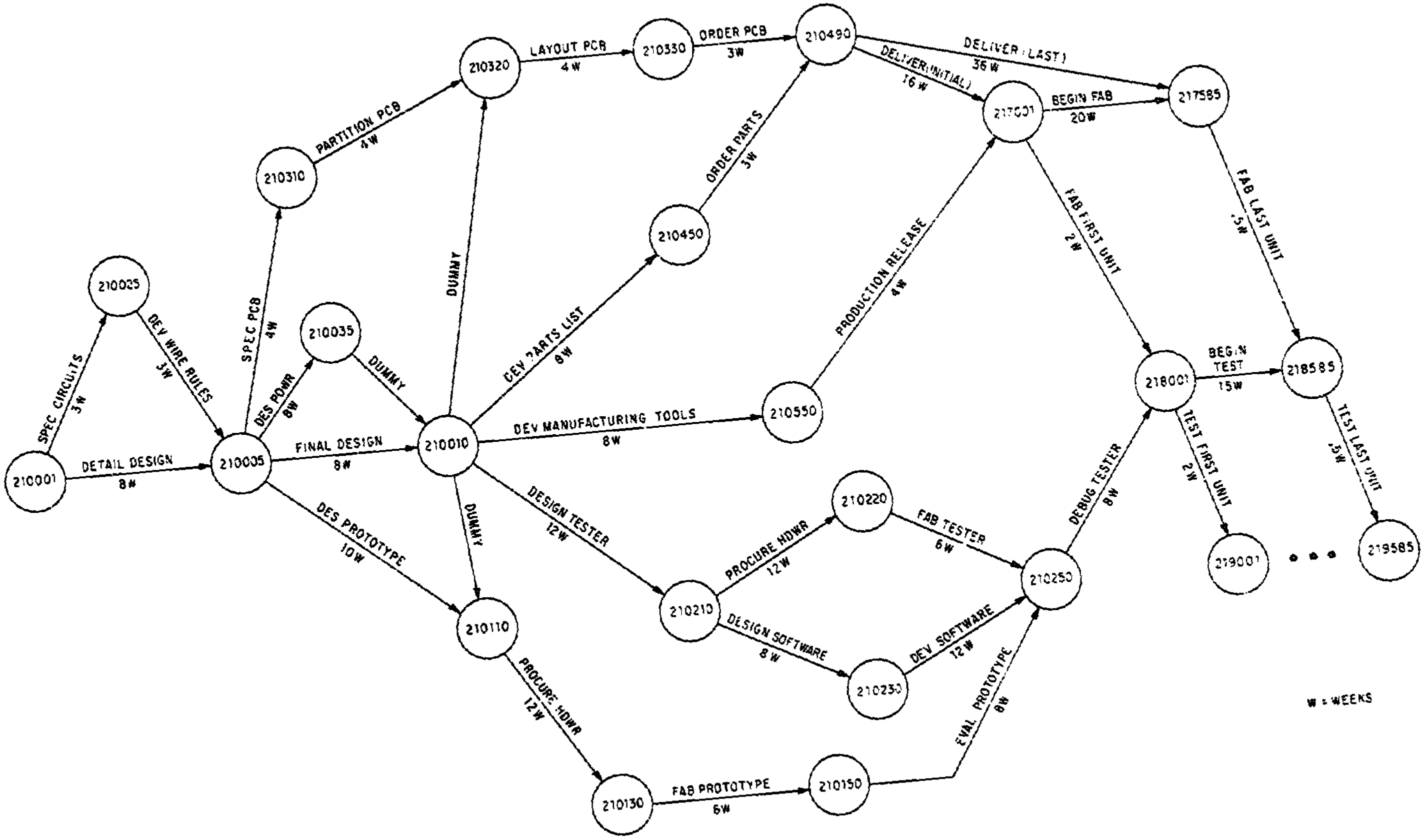


Figure 8.3 FMP Processor Final Design, Fabrication and Test

8.1.5 Critical Path

The critical paths for the schedules shown in Figures 8.2 and 8.3 were determined using Burroughs PROMIS. A data base was created listing each activity's starting and ending event, the mean time to complete and the activity description. A hypothetical start date was declared and a PROMIS output was generated providing the earliest and latest start date, earliest and latest end date and the amount of slack in each activity. A hypothetical start date in calendar terms is required, since PROMIS uses a real calendar for its time base. This is done to permit considerations of weekends, holidays and for convenience of reporting. For the purposes of this demonstration a start date of 1 July 1981 is hypothosized for the beginning of the final design of the FMP. Figures 8.4 and 8.5 show the PROMIS outputs for the schedules in Figures 8.2 and 8.3. Table 8.2 explains the abbreviations used on the PROMIS reports.

The critical path is that sequence of activities which show zero slack. In Figure 8.4, PROMIS output for the FMP schedule the critical path is seen as being:

Preceding Event Number	Succeeding Event Number	Activity Description	Mean Time
240001	252000	The coordinator design, fabrication and test,	50 weeks
252000	253000	Installation and debugging of the coordinator	12 weeks
253000	254000	Installation and debugging of the connection network	10 weeks
254000	299000	Initial debugging of the FMP.	20 weeks
		TOTAL	92 weeks

There is a parallel branch in the critical path in the test system design and fabrication. Examination indicates 42 weeks slack in the design, fabrication and testing of the data base memory. This shows that a starting date of any where between 1 July 1981 and 21 April 1982 would not impact the finish date of 5 April 1983 for that schedule element based on the estimate of 50 weeks for its completion. The ability to determine slack permits the manager to effectivetly allocate resources among the various parallel activities.

```
  PRED       SUCC    D E S C R I P T I O N        MEAN   EARLIEST     LATEST   EARLIEST      LATEST      TOTAL
 NUMBER     NUMBER                                TIME      START      START     FINISH      FINISH      SLACK
210001     219001    PR DES/FAB/TEST              58.0  01 JUL 81  02 SEP 81  10 AUG 82   12 OCT 82       9.0
219001     219585    PR FAB/TEST                  15.0  11 AUG 82  03 DEC 82  26 NOV 82   22 MAR 83      16.5
219001     254000    INSTALL FIRST PR              5.0  11 AUG 82  13 OCT 82  14 SEP 82   16 NOV 82       9.0
219585     259000    INSTALL LAST PR               2.0  26 NOV 82  23 MAR 83  10 DEC 82   05 APR 83      16.5
220001     229001    EM DES/FAB/TEST              50.0  01 JUL 81  28 OCT 81  15 JUN 82   12 OCT 82      17.0
229001     229585    EM FAB/TEST                  16.0  16 JUN 82  01 DEC 82  05 OCT 82   22 MAR 83      24.0
229001     254000    INSTALL FIRST EM              5.0  16 JUN 82  13 OCT 82  20 JUL 82   16 NOV 82      17.0
229585     259000    INSTALL LAST EM               2.0  06 OCT 82  23 MAR 83  19 OCT 82   05 APR 83      24.0
230001     253000    CN DES/FAB/TEST              40.0  01 JUL 81  02 DEC 81  06 APR 82   07 SEP 82      22.0
240001     252000    CR DES/FAB/TEST              50.0  01 JUL 81  01 JUL 81  15 JUN 82   15 JUN 82        -0
250001     251000    CABINET&CABLE DES/FAB        40.0  01 JUL 81  15 JUL 81  06 APR 82   20 APR 82       2.0
251000     252000    INST&C.O.PWR DIST SYS         8.0  07 APR 82  21 APR 82  01 JUN 82   15 JUN 82       2.0
252000     253000    INST&DEBUG CR                12.0  16 JUN 82  16 JUN 82  07 SEP 82   07 SEP 82        -0
253000     254000    INST&DEBUG CN                10.0  08 SEP 82  08 SEP 82  16 NOV 82   16 NOV 82        -0
254000     259000    INITIAL DEBUG FMP            20.0  17 NOV 82  17 NOV 82  05 APR 83   05 APR 83        -0
260001     251000    POWER DIST DES/FAB           40.0  01 JUL 81  15 JUL 81  06 APR 82   20 APR 82       2.0
270001     252000    TEST SYSTEM DES/FAB          50.0  01 JUL 81  01 JUL 81  15 JUN 82   15 JUN 82        -0
280001     259000    DBM DES/FAB/TEST             50.0  01 JUL 81  21 APR 82  15 JUN 82   05 APR 83      42.0
```

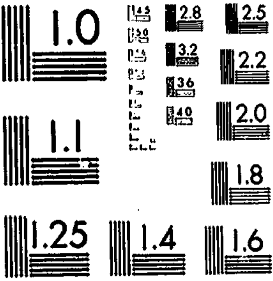
Figure 8.4 PROMIS Output for FMP Fabrication and Integration Schedule

```
  PRED       SUCC    D E S C R I P T I O N        MEAN   EARLIEST     LATEST   EARLIEST     LATEST      TOTAL
 NUMBER     NUMBER                                TIME      START      START     FINISH     FINISH      SLACK

 0001       0005     DETAIL DESIGN                 8.0  01 JUL 81  01 JUL 81  25 AUG 81  25 AUG 81        .0
 0001       0025     SPEC CIRCUITS                 3.0  01 JUL 81  15 JUL 81  21 JUL 81  04 AUG 81       2.0
 0005       0010     FINAL DESIGN RULES            8.0  26 AUG 81  26 AUG 81  20 OCT 81  20 OCT 81        .0
 0005       0035     POWER DESIGN                  8.0  26 AUG 81  26 AUG 81  20 OCT 81  20 OCT 81        .0
 0005       0110     DESIGN PROTOTYPE             10.0  26 AUG 81  23 SEP 81  03 NOV 81  01 DEC 81       4.0
 0005       0310     SPEC PCB                      4.0  26 AUG 81  14 OCT 81  22 SEP 81  10 NOV 81       7.0
 0010       0110     DUMMY ACTIVITY                 .0  20 OCT 81  02 DEC 81  20 OCT 81  02 DEC 81       6.0
 0010       0320     DUMMY ACTIVITY                 .0  20 OCT 81  09 DEC 81  20 OCT 81  09 DEC 81       7.0
 0010       0450     DEVELOPE PARTS LIST           3.0  21 OCT 81  16 DEC 81  10 NOV 81  05 JAN 82       8.0
 0010       0550     DEVELOPE MFG TOOLS            8.0  21 OCT 81  24 FEB 82  15 DEC 81  20 APR 82      18.0
 0010       0210     DESIGN TESTER                12.0  21 OCT  1  21 OCT 81  12 JAN 82  12 JAN 82        .0
 0025       0005     DEVELOPE WIRE RULES           3.0  22 JUL 81  05 AUG 81  11 AUG 81  25 AUG 81       2.0
 0035       0010     DUMMY ACTIVITY                 .0  20 OCT 81  21 OCT 81  20 OCT 81  21 OCT 81        .0
 0110       0130     PROCURE PROTOTYPE HDWR       12.0  04 NOV 81  02 DEC 81  26 JAN 82  23 FEB 82       4.0
 0130       0150     FABRICATE PROTOTYPE           6.0  27 JAN 82  24 FEB 82  09 MAR 82  06 APR 82       4.0
 0150       0250     DEBUG & EVAL PROTOTYPE        8.0  10 MAR 82  07 APR 82  04 MAY 82  01 JUN 82       4.0
 0210       0230     DESIGN TESTER SOFTWARE        8.0  13 JAN 82  13 JAN 82  09 MAR 82  09 MAR 82        .0
 0210       0220     PROCURE TESTER HDWR          12.0  13 JAN 82  27 JAN 82  06 APR 82  20 APR 82       2.0
 0220       0250     FABRICATE TESTER              6.0  07 APR 82  21 APR 82  19 MAY 82  01 JUN 82       2.0
 0230       0250     DEVELOPE TESTER SOFTWARE     12.0  10 MAR 82  10 MAR 82  01 JUN 82  01 JUN 82        .0
 0250       9001     DEBUG TESTER                  8.0  02 JUN 82  02 JUN 82  27 JUL 82  27 JUL 82        .0
 0310       0320     PARTITION PCB                 4.0  23 SEP 81  11 NOV 81  20 OCT 81  08 DEC 81       7.0
 0320       0330     LAYOUT PCB                    4.0  21 OCT 81  09 DEC 81  17 NOV 81  05 JAN 82       7.0
 0330       0490     CHECK PCB                     3.0  18 NOV 81  06 JAN 82  09 DEC 81  26 JAN 82       7.0
 0450       0491     ORDER TESTER PARTS            3.0  11 NOV 81  06 JAN 82  01 DEC 81  26 JAN 82       8.0
 0490       7001     INITIAL MAIL DELIVERY        16.0  09 DEC 81  27 JAN 82  30 MAR 82  18 MAY 82       7.0
 0490       7585     FINAL MAIL DELIVERY          36.0  09 DEC 81  27 JAN 82  17 AUG 82  05 OCT 82       7.0
```

Figure 8.5 PROMIS Output for FMP Processor Final Design, Fabrication and Test

4 OF 7

26072



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

```
   PRED      SUCC   D E S C R I P T I O N          MEAN  EARLIEST    LATEST   EARLIEST     LATEST    TOTAL
  NUMBER    NUMBER                                 TIME     START     START     FINISH     FINISH    SLACK
0550        7001     PRODUCTION RELEASE             4.0  15 DEC 81  21 APR 82  12 JAN 82  13 MAY 82   18.0
7001        7585     BEGIN PROCESSOR FAB           20.0  31 MAR 82  15 MAY 82  17 AUG 82  05 OCT 82    7.0
7001        9001     FAB FIRST PROCESSOR            2.0  31 MAR 82  14 JUL 82  13 APR 82  27 JUL 82   15.0
7585        9585     FAB LAST PROCESSOR             5.0  18 AUG 82  06 OCT 82  21 SEP 82  09 NOV 82    7.0
8001        8585     BEGIN PROCESSOR TEST          15.0  28 JUL 82  28 JUL 82  09 NOV 82  09 NOV 82     .0
8001        9001     TEST FIRST PROCESSOR           2.0  28 JUL 82  01 DEC 82  10 AUG 82  14 DEC 82   18.0
8585        9585     TEST LAST PROCESSOR            5.0  10 NOV 82  10 NOV 82  14 DEC 82  14 DEC 82     .0
```

Figure 8.5 PROMIS Output for FMP Processor Final Design, Fabrication and Test (Continued)

TABLE 8.2

PROMIS Report Terms

HEADINGS

PRED NUMBER — Preceeding event number

SUCC NUMBER — Succeeding event number

DESCRIPTION — A brief identification of the activity

MEANTIME — An Estimate in weeks (unless otherise noted) of the time expected for completion.

EARLIEST START — The earliest date that an activity can begin.

LATEST START — The lastest date that an activity can begin without impacting the schedule.

EARLIEST FINISH — The earliest date that an activity can be finished.

LATEST FINISH — The latest date that an activity can be finished without impacting the schedule.

TOTAL SLACK — The amount of time in weeks unless otherwise designated) in escess of the meantime during which a task may be completed and not impact the schedule.

ABBREVIATIONS

PR	Processor
EM	Extended Memory
CR	Coordinator
CN	Connection Network
DBM	Data Base Memory
FMP	Flow Model Processor
POW	Power
DIST	Distribution
SYS	System
PCB	Printed Circuit Board
HDWR	Hardware
MATL	Material
DES	Design
FAB	Fabricate
INST	Install
C.O.	Checkout
SPEC	Specify
MFG	Manufacturing
EVAL	Evaluate

Figure 8.5, PROMIS Report for Processor Design and Fabrication, reveals the following critical path:

Preceding Event Number	Succeeding Event Number	Activity Description	Mean Time
210001	210005	Detail Design	8 weeks
210005	210010	Final Design	8 weeks
(210005	210035)	Power (Supply) Design	(8 weeks parallel branch)
210010	210210	Design Testor	12 weeks
210210	210230	Design Testor Software	8 weeks
210230	210250	Develop Testor Software	12 weeks
210250	218001	Debug Testor	8 weeks
218001	218585	Begin Processor Tests	15 weeks
218585	219585	Test Last Processor	.5 weeks
			71.5 weeks

The 71.5 week period begins 1 July 1981 and ends 14 December 1982 with the completion of the off-line testing of the last (585th) processor. It should be noted that since there are no apparent constraints on the requirement date of the first processor there appears to be 18 weeks of slack. This apparent slack disappears as soon as the requirement for the availability of the first processor for installation into the FMP (as shown in Figure 8-2) is considered. Accordingly, in reality, there is a branch of the critical path after the sixth activity, Debug Testor, which is Test First Processor, 2 weeks. This results in a critical path of 58 weeks for the availability of the first processor. This same 58 weeks is shown as the time for the first activity of the upper path of Figure 8.2.

8.2 FACILITIES

Refinements made during this study on the concept of the NASF as presented in the initial study [1] have had no significant impact on the facility requirement (also presented in the initial study. Table 8.3 summarizes these facility requirements for power and floor space and places a maximum limit on each. Appendix J, General Design Guidelines delineates environmental factors for the design of the NASF Processing System hardware. These same limits should be consistent with the environmental capabilities of the physical building.

TABLE 8.3
Summary of NASF Power and Floor Space Requirements

	POWER	FLOOR SPACE
ESTIMATE	555 KVA	40,000 square feet
MAXIMUM	750 KVA	50,000 square feet

REFERENCES*

1. Final Report, Numerical Aerodynamic Simulation Facility, Preliminary Study, October 1977, Contract NAS2-9456, Burroughs Corporation, Paoli, PA. NASA CR 152060, 152061, 152062.

2. Final Report, Numerical Aerodynamic Simulation Facility, Preliminary Study Extension, February 1978, Contract NAS2-9456, Burroughs Corporation, Paoli, PA. NASA CR 152106, 152107.

3. NASF DESIGN GUIDANCE STUDY, Preliminary Draft M. V. Markoff, June 1978, Informatics, TN-78-2000-630-1.

4. NASF UTILIZATION, October 1978 (NASA Ames), Draft as amended through subsequent communications.

5. A Working Paper on Fault Tolerance with respect to Numerical Aerodynamic Simulation Facility, June 1, 1977, Burroughs Corporation, Paoli, PA.

6. Lawrie, Duncan H., "Access and Alignment of Data in an Array Processor", IEEE Trans., Comp., C24(1975) pp. 1145-1155.

7. Berlekamp, E. R., Algebraic Coding Theory, McGraw-Hill, N.Y. 1968.

8. Koppel, Robert, "RAM Reliability in Large Memory Systems - Significance of Predicting MTBT" Computer Design, February 1979.

9. MIL-HDBK-217B, Reliability Prediction of Electronic Equipment, September 1974.

10. ANSI X3.9-1978, American National Standard Programming Language FORTRAN (Fortran 77), 1978.

11. B7700 System Miscellanea, Doc. No. 5001886, Burroughs Corp.

12. B7/6000 Work Flow Language Reference Manual, Doc. No. 5001555, Burroughs Corporation.

13. Chen, Shy-ching, Speedup of Iterative Programs In Multiprocessing Systems, University of Illinois, Dept. of Computer Science, Report No. UIUCDCS-R-75-694, January 1975.

14. B7/6000 Network Definition Language Reference Manual, Doc. No. 5001522, Burroughs Corporation.

15. Thornton, J. E., "Overview of HYPERchannel", ** COMPCON SPRING 79 Digest of Papers, IEEE Computer Society, pp. 262-265.

* Appendix references are included at the end of each appendix.
** NOTE: HYPERchannel is a Trademark of Network Systems Corp.

APPENDIX A

PERFORMANCE PROJECTION BASED ON BENCHMARK PROGRAMS

A.1 INTRODUCTION

The four programs used as benchmarks in evaluating the design were:

(1) NASA 3D implicit aerodynamic flow (aero flow) code supplied by Ames

(2) NASA 3D explicit aerodynamic flow (aero flow) code supplied by Ames

(3) GISS weather code, in several different versions

(4) Spectral weather code from MIT

Evaluations of the first three were comprehensive, resulting in projections of 1.01 Gflops/sec for the implicit, 0.89 Gflops/sec for the explicit, both at one million grid points, and 0.53 Gflops/sec for the GISS weather code.

A range of throughput values from zero to 1.50 Gflops/second for individual code sections was derived from the simulation efforts. These variations are primarily caused by the relationships of individual subprograms to the data in local processor memory and extended memory, the choice of mesh size and the choice of the metric for performance measurement. An example of zero throughput is provided by the subroutines BCY, BCZ and OUTER in the 3D explicit aerodynamic flow code. These routines shuffle data in data arrays in the EM. As no floating point operations are required for this function a zero throughput value results. Data sorting algorithms would be similar examples.

A throughput value of 1.45 Gflops/second is illustrated by the intrinsic square root function. Square root operates entirely within the processor, mostly in high speed local registers. Substantial portions of the simulated codes run from 1.1 to 1.3 Gflops/second rates. Examples of this are

- subroutine BTRI in the implicit code
- subroutine CHARAC in the explicit code
- subroutine LINKHO in the GISS code

Examples with lower throughput values typically occurred in routines where a high frequency of access to the three dimensional global arrays was required. The ability to overlap array index calculations with floating point operations is limited under these conditions.

Performance is generally increased when the grid size is increased. The 3D explicit aerodynamic flow code showed 0.79 Gflops for 30,000 grid points and 0.89 Gflops for 1,000,000 grid points.

The frequency of execution of individual code segments must be known for the performance evaluations. Assumptions were made in those cases where data dependent loop counts and branches occur. Throughout the programs a mean value rule was generally employed with an occasional reduction to some more conservative value where appropriate. In one case, CHARAC from the 3D explicit, simulation was run at several different assumptions to test the sensitivity of throughput to the data dependency assumptions. In the case of CHARAC, throughput varied no more than 15%.

The implicit code achieves the 1.0 Gflops/sec throughput rate being used as a guide. The explicit code appears to be about 10% slower than the implicit code.

On GISS weather, the non-vectorizable portions of the code exceeded one Gflops/sec (COMP3), while the vectorizable portions (COMP1 and COMP2) were slowed down by EM accessing and memory-to-memory moves that produced no floating point operations.

The following sections discuss the methods used for projecting performance. Also to be reviewed are each program, and some other applications, namely sorting and fast Fourier transforms.

A.2 METHOD

The method used for performance evaluation was generally the same for all of the first three benchmark programs. Because of time and budget limitations, only a cursory look was taken at the Spectral weather code.

First, throughput was analyzed on the basis of FMP computations only. I/O operations were ignored. Transfers between DBM and file system are independent of, and go on in parallel with, the FMP computation. It is assumed that the file manager stages the next job, and unloads the last job, in times which are completely overlapped with current computation. DBM-EM transfers are also ignored, since they go on concurrently with current processing (as long as EM space is available). At a transfer rate of 40 Mw/s, the 15 million words of a restart point of a typical aero flow code are loaded in 0.375 seconds, which can be compared with the 600 seconds duration of a typical run. Hence, even when not overlapable because of EM allocation conflicts, they should have little effect on aero flow computations. Therefore, both system I/O and user I/O were ignored.

Each program was analyzed to find the calling tree of its subroutines. Major program parameters such as grid size, total number of time steps, etc. were then established. This data allowed the determination of the total number of executions of each subroutine.

An analysis of all data declarations was then performed to establish the GLOBAL or LOCAL memory palcement of all major variables. This analysis also determined those variables that were potential type INALL variables. The programs were then scanned to establish the placement of the DOALL statement construct throughout the program structure. This information determined the number of parallel machine cycles for each DOALL and the processor utilization level number. A handcount was then performed on all routines to determine the total number of all floating point operations (f), the number of floating point divide operations and the number of Extended Memory accesses (m_i). Processor utilization was also noted for each code sequence. Next, high usage sections of typical code were selected for hand compiling into FMP machine language. Results from detailed simulations of these code sections were then used to develop an empirical formula used to interpolate the performance of code sections not simulated. This formula is a linear function of the number of floating point operations, the number of floating point divide operations and the number of extended memory accesses. These three factors are sufficient to fit the simulation results, after constants are adjusted to provide agreement with detailed simulation results.

The following symbol definitions pertain to the equations below:

T_s = Total system throughput rate - Gflops/second

T_p = Single processor throughput rate - Gflops/second

E_f = Total floating point ope:ations - Flops

E_d = Total floating point divide operations over 2% of E_f

E_m = Total Extended Memory access operations

E_t = Total program elapsed time (1 processor)

R_i = Ratio of active to total processors

System throughput is then defined as:

$$T_s = \frac{\text{Total Flops}}{\text{Total Time}} = \frac{\sum f_i}{\sum t_i} \tag{A.1}$$

The linear approximation to this function was then determined as:

$$T_s = \frac{\sum f_i}{\sum t_i} = \frac{1.74}{K_0+5.0 * \frac{\sum m}{\sum f}} \qquad (A.2)$$

$$\text{as} \quad T_p = \frac{T_s}{512} \qquad (A.3)$$

E_t (Elapsed time) was then solved for as

$$E_t = \frac{\sum f * (K_0+5* \frac{\sum m}{\sum f}) * 512}{1.74} \qquad (A.4)$$

or E_t (A.5)

$$E_t = K_0*295* \sum f+1471* \sum m$$

The value of K_0 was then estimated as 1.0 or 1.2 based on individual estimates of the quantity of nonfloating point commands in a given code section. Basic system throughput could then be calculated knowing the individual counts of floating point operations and Extended Memory access via

$$T_s = \frac{\sum f_i}{\sum t_i} * R_i \qquad (A.6)$$

where R_i (ratio of active to total processors) was determined from the analysis of parallel DOALL statements. Where the formula gave results in excess of 1.33 Gflops/sec, for a particular code sequence, the value 1.33 Gflops/sec was adopted instead.

The above formula for calculating individual code segment times assumed that two percent of the floating point operations were divide operations. The divide instruction consumes 1460 nanoseconds which is nearly six times longer than the estimated nominal floating point instruction time. A special count of divide instructions was therefore included in the analysis. When this count exceeded the two percent rule a correction factor of 1460* excess count was added into the above time calculation formula.

Examples of exceptions are TRIB and EIGEN in the implicit (too many divisions), AVRX in the GISS weather (too much integer arithmetic and data-dependent processor utilization).

Figure A.1 plots the formula used against the results of simulations both for the implicit code, the explicit code, and the GISS weather code. It is seen that the formula is validated over a large assortment of "typical" codes. It is also obvious that the formula must be taken with a grain of salt, and that each and every section of code should be scrutinized to see if it represents some exception for which the formula will not work.

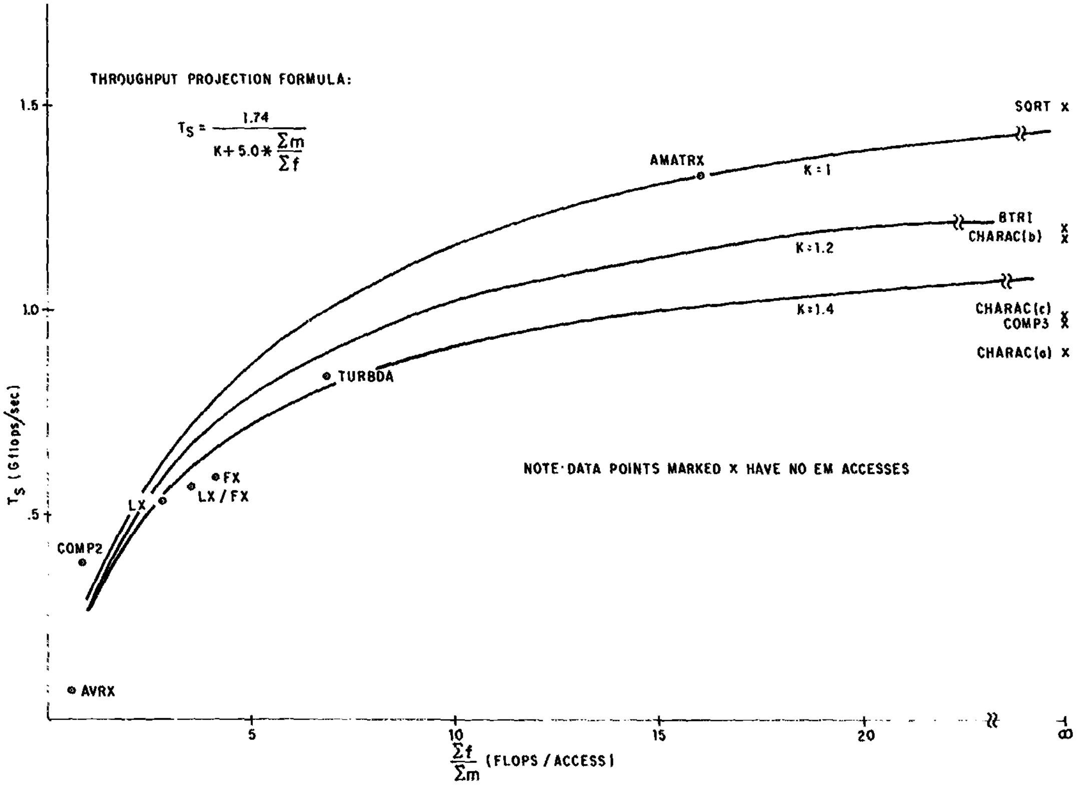


Figure A.1 Throughput Projection Formula vs. Simulation Results

A.3 THROUGHPUT OF IMPLICIT AERO FLOW CODE

A.3.1 Summary

The throughput of the implicit code is 1.01 Gflops/sec for the grid size of 100 x 50 x 200. This is the estimate resulting at the end of the analysis. During the course of the analysis, as various assumptions and corrections were being applied, the estimate varied from 0.973 Gflops/sec to 1.043 Gflops/sec.

A.3.2 Assumptions

The following were the assumptions and program modifications used to produce this result. Examples of the resulting code are included in sections which follow. In addition, Appendix G has a side-by-side comparison of some of the original codes and the FMP codes.

All variables indexed on the three grid variables J, K, L were assumed to be STRUCTURE arrays resident in Extended Memory. In one case, the accessing pattern was such that the variables could be assumed to be resident in Processor Memory. In this case, an instance being executed on a processor was able to access the STRUCTURE variables without having the time penalty of EM accessing. Not much improvement is expected when more of the STRUCTURE arrays are processor resident.

The grid size is IMAX, JMAX, KMAX = 100, 50, 200

The compiler is able to use a MAD or FADEXL instruction when one is appropriate, and to reorder arithmetic expressions. For example, the expression (A + B*C/2) would be implemented with a FADEXL and a FMAD.

I/O operations are ignored.

NMAX = 100, arbitrarily.

All arrays declared as A(720,6,30) where the 720 dimension is indexed on KL = (L-1)*ND+K, and the 30 dimension on J are assumed to be changed to A(100,50,200,6) where the subscripts used will be J, K, L, and whatever, respectively.

A total of 94 separate sequences of code were identified.

The computation of RESID at the beginning of STEP is assumed to be a SUMALL over the domain J=1,100; K=1,50; L=1,200. With 1920 cycles in this DOALL, the 9 extra steps at the end for the SUMALL are insignificant.

All calls on subroutines XXM, YYM, and ZZM were brought up into line. Further, the resulting code was put down into the DO loop that normally follows such calls so that XX, YY, and ZZ are recomputed each time. The result is that the four result values produced by each single iteration, within the former XXM, YYM, or ZZM, are used immediately, and can be LOCAL variables. If the program were left in its present structure, where all elements of the arrays XX, YY, and ZZ are computed at one time, the arrays would have to be either INALL, with 100-fold waste of memory space, or GLOBAL, with 100-way access conflicts in memory when these one-dimensional arrays are used in two-dimensional DOALLs. By computing these elements one at a time at the point where they are used, the memory to store them is saved. The amount of computation does not change but several copies of in-line code are needed to replace each such subroutine.

In VISRHS, and in BTRI, essentially identical code is seen replicated. One copy is executed at one end point (say I=1), the other copy is executed at the other end point (say I=IMAX), and the third copy is inside a DO I=2, IMAX-1 loop. In VISRHS these three cases were subsumed into a single DO I=1,IMAX loop. In BTRI, an observation on the incoming data shows that the first iteration is degenerate (a diagonal matrix is being decomposed , which is nearly a no-op), so the first copy is rewritten, and the latter two combined into a DO I=2,IMAX loop.

SMOOTH was rewritten into a single three-dimensional DOALL.

Only those named common areas that are actually used in a program unit are declared. This improves FMP operation, speeds up subroutine entry and sometimes releases memory space.

Where feasible, divisions were replaced by multiplication by the reciprocal, including every division by a literal.

In doubly nested DO loops with simple subscripting (DO N=1,5 and DO M=1,5), the code is assumed restructured either by the programmer or by a later optimizing version of the compiler such that there is no more than one integer multiply per set of subscripts. For example, one can increment auxiliary index variables per iteration. Two such loops contain 26 percent of all the floating point operations in the program.

A.3.3 Analysis of Implicit Aero Flow Code

Equation A.6 (Section A.2) is an extrapolation of the simulation results to the portions of the code that were not simulated. About 60 percent of the running time of the implicit code is represented in the two simulations that were done, namely subroutine BTRI and the portion of subroutine RHS that used to be subroutine AMATRX in a previous version of the program. This gratifyingly high percentage of execution actually simulated arises because BTRI itself represents over 55 percent of the computation of the implicit code. One statement in BTRI which is executed 25,000,000 times during the course of the program, represents 21 percent of all the floating point operations in the entire program, and is found in the test case.

Exceptions to Equation A.6 are code sequences in TRIB, EIGEN, and INITIA with an atypically high proportion of divides. These are executed so infrequently as to disappear from the total throughput figure. At the beginning of BC there is a section that could have been implemented as a series of SUMALL's. In this analysis, the summations were done serially with 38 percent processor utilization instead. On the other hand, a SUMALL was used at the beginning of STEP to compute the variable RESID. This runs with "typical" speed because of the size of the DOALL, which is across all three dimensions, or 1,000,000 instances, so that the final 512-way summation takes negligible time compared to the 1920 cycles in the DOALL. The processor utilization for this case is 99.97 percent.

"AMATRX" was simulated. It is the part of the subroutine STEP so identified in a line of comment. The test case consisted of 3750 floating point operations per processor, achieved by iterating several times around the code. Hence, the frequency of execution of loop control was somewhat higher than in the actual case in STEP, where additional operations are in the same loop. The observed time is counted in clocks per processor. At 40 ns per clock, and 512 processors, this computes to 1.330 Gflops/sec for the entire FMP. Overlap between the several execution units within the processor was such that on the average there were 1.20 instructions in the course of execution at any one time.

"BTRI" was also simulated. The test case was constructed by taking the doubly-nested DO loop identified by the comment "COMPUTE B PRIME", and following it with one pass through "INSERT LUDEC AGAIN", and wrapping an outer loop around both. There were a total of 650 floating point operations executed in simulation. For present purposes, it is instructive to separate the 500 operations executed during the doubly nested loop, and the 150 executed in LUDEC. The LUDEC protion of the simulation executed at 1.30 Gflops/sec, while the doubly nested loop executed at 1.170 Gflops/sec, at 512 processors busy.

Hence, the assumption of 1.33 Gflops/sec for "ordinary" code execution speed where all variables are local to the processor is justified. However, when single statements are found inside doubly nested DO loops with triple subscripting on most of the arithmetic primaries, performance is derated to 1.17 Gflops/sec. The two swatches of code deserving this derating are the loops in BTRI, and similar loop in VISMAT. The simulation printout associated with this loop in test case "BTRI" shows that 14.6 percent of the time the processor was waiting for instruction fetch. These were primarily integer instructions associated with subscript computations. A sequence of integer IADDs, for example, can be executed faster than the instructions can be fetched.

A.3.4 FMP FORTRAN Version

A.3.4.1 One-to-one Mapping from Serial FORTRAN

There is a simple one-for-one translation from FORTRAN furnished by NASA into FMP FORTRAN as follows. All arrays subscripted with the grid variables are made STRUCTURE. DO loops (single or nested) on the grid variables are automatically turned into DOALLs as long as the data dependence allows it. Temporary variables are allowed to be LOCAL by default. The implicit code, as supplied by NASA, is of such regularity that practically all of it can be transformed into FMP FOTRAN using such simple rules. Because of this, and in order to save time, most of the FMP FORTRAN version of the implicit aero code was not even written down, since it was obvious from the NASA-furnished version by inspection.

SMOOTH and BTRI were rewritten to better match the structure of the FMP. Discussion follows.

A.3.4.2 SMOOTH

A revised FORTRAN version of subroutine SMOOTH is exhibited in Figure A.2. All computation is put into a three-dimensional DOALL. Note that the arrays Q and S (which have total dimensionality Q(100,50,200,6) and S(100,50,200,5)) are defined as STRUCTURE variables since they are included in both an INALL statement and in a USING clause over the domain. These arrays would exist in Extended Memory. The other variables defined over the structure (SS, CT, and the temporaries T1, T2, T3, and T4) are allocated space within each processor. Note that only SS and CT must be unique to an instance over the sections of the DOALL. The temporaries could share storage with other instances. Computations on SS and CT, having 10^6 elements uniformly distributed over the processors, will take up 1862 (cycles) * 6 = 11178 words of processor memory during the execution of subroutine SMOOTH.

The other large user of processor memory space is BTRID, a LOCAL COMMON area which must be declared inside the DOALLs of STEP so it can be common to the calls on BTRI. Here is an example of the use of dynamic memory allocation. Upon leaving the last DOALL in STEP, this LOCAL COMMON is deallocated, leaving space for SS and CT to be allocated during SMOOTH. See the following section on the rewritten BTRI for further discussion.

Temporary varibles TEMP, T1, T2, T3, and T4 were used to hold copies of STRUCTURE array elements so that they could be used through several operations with only one fetch from EM.

The statement NEXTDO, used in this code but not explained in Chapter 4, is a convenience. The NEXTDO statement is shorthand for an ENDDO statement followed immediately by another DOALL on the same domain.

```
      SUBROUTINE SMOOTH
       COMMON/BASE/NMAX,JMAX,KMAX,LMAX,DT,GAMMA,GAMI,FSMACH,
     1   DX1,DY1,DZ1,FV(5),FD(5),HD,ALP,GD,OMEGA,HDX,HDY,HDZ,RM,
     2   CNBR,PI,ITR,NP,INT1,INT2,INT3
      DOMAIN /MODEL/; J=1,100; K=1,50; L=1,200
      REGION /THREED((J=2,JMAX-1),(K=2,KMAX-1),(L=2,LMAX-1))/
     *     = /MODEL(J,K,L)/
      INALL /MODEL/ Q(6),S(5),SS,CT(5),T1,T2,T3,T4
C   4TH ORDER SMOOTHING, 2D ORDER AT THE BOUNDARIES
      DOALL /THREED(J,K,L)/ ;  USING Q, S, SMU
      TEMP = 1./Q(J,K,L,6)
      DO 1 N=1,5
      CT(N)= Q(J,K,L,N)*Q(J,K,L,6)
1     CONTINUE
      IF (J.EQ.2 .OR. J.EQ.JMAX-1) THEN
        T1 = Q(J+1,K,L,6)
        T2 = Q(J-1,K,L,6)
        DO 2 N=1,5
        SS = S(J,K,L,N) + 0.5*SMU*(Q(J+1,K,L,N)*T1)- 2.*CT(N) +
     1        Q(J-1,K,L,N)*T2)*TEMP
2       CONTINUE
       ELSE
        DO 3 N=1,5
        T1=Q(J+2,K,L,6)
        T2=Q(J-2,K,L,6)
        T3=Q(J+1,K,L,6)
        T4=Q(J-1,K,L,6)
        SS = S(J,K,L,N) + SMU*(Q(J+2,K,L,N)*T1 + Q(J-2,K,L,N)*T2 +
     1        4.*(Q(J+1,K,L,N)*T3 + Q(J-1,K,L,N)*T4) - 6.*CT(N))*TEMP
3       CONTINUE
       ENDIF
      NEXTDO
       IF (K.EQ.2 .OR. K.EQ.KMAX-1) THEN
        T1=Q(J,K+1,L,6)
        T2=Q(J,K-1,L,6)
        DO 4 N=1,5
         SS = SS + 0.5*SMU*(Q(J,K+1,L,N)*T1 + Q(J,K-1,L,N)*T2 -
     1         2.*CT(N))*TEMP
4       CONTINUE
```

Figure A.2 FMP FORTRAN Version of SMOOTH

```
        ELSE
       T1=Q(J,K+2,L,6)
       T2=Q(J,K-2,L,6)
       T3=Q(J,K+1,L,6)
       T4=Q(J,K-1,L,6)
       DO 5 N=1,5
         SS=SS+SMU*(Q(J,K+2,L,N)*T1 + Q(J,K-2,L,N)*T2 +
     1           4.*(Q(J,K+1,L,N)*T3 + Q(J,K-1,L,N)*T4) - 6.*CT(N))*TEMP
 5     CONTINUE
       ENDIF
       NEXTDO
        IF (L.EQ.2 .OR. L.EQ.LMAX-1) THEN
         T1=Q(J,K,L+1,6)
         T2=Q(J,K,L-1,6)
         DO 6 N=1,5
          S(J,K,L,N) = SS + 0.5*SMU*(Q(J,K,L+1,N)*T1 +
                        Q(J,K,L-1,N)*T2 - 2.*CT(N))*TEMP
 6       CONTINUE
        ELSE
         T1 = Q(J,K,L+2,6)
         T2 = Q(J,K,L-2,6)
         T3 = Q(J,K,L+1,6)
         T4 = Q(J,K,L-1,6)
         DO 7 N=1,5
          S(J,K,L,N) = SS + SMU*(Q(J,K,L+2,N)*T1 + Q(J,K,L-2,N)*T2+
     1                 4.*Q(J,K,L+1,N)*T3 + 4.*Q(J,K,L-1,N)*T4
     2                 - 6.*CT(N))*TEMP
 7       CONTINUE
        ENDIF
       ENDDO /THREED/ ;  GIVING S
      RETURN
      END
+
```

Figure A.2 FMP FORTRAN Version of SMOOTH (Cont'd)

The resulting rewrite of SMOOTH reduces the number of flops from 225 x 10^8 to 201 x 10^8, and the number of EM accesses from 195 x 10^8 to 72 x 10^8, as compared to a mechanical translation of DO loops to DOALLs. Thus, the time is improved more than the throughput.

A.3.4.3 BTRI

The subroutine BTRI was also modified. Observe that when BTRI is entered, array B is a diagonal matrix with zeros off the diagonal. The first piece of code, which is a copy of LUDEC, therefore executes with most of its input variables equal to zero. LUDEC is a modified Cholesky decomposition. When faced with a diagonal matrix, it produces a copy of that matrix for the lower triangular matrix, and produces the identity matrix for the upper triangular matrix. In BTRI the variables L11, L22, L33, L44, and L55 are the reciprocal of the diagonal terms, in order to save repeated unnecessary divisons later on. The diagonal terms of the upper triangular matrix are unconditionally equal to 1.0 and hence are not computed.

The first copy of the former LUDEC, as shown in NASA's BTRI, can be simplified to the version shown in the attached listing, Figure A.3. The last iteration of the former LUDEC, at index equal to IUA, differs from the central iterations only by the omission of the computation of C PRIME. To simplify the source code, this copy was pulled into the main iteration in BTRI.

Common area BTRID would be declared in STEP:

LOCAL COMMON/BTRID/ A(LMAX,5,5), B(LMAX,5,5), (CLMAX,5,5), D(LMAC,5,5), F(LMAX,5) in that call on BTRI in which the limiting index is LMAX using the one-for-one translation of the original.

With LMAX=200, this means that common BTRID is 21,000 words long. When the extent is JMAX, BTRID will take 10,500 words and when the extent is KMAX, BTRID will be allocated 5,250 words. Note that in STEP, where this COMMON is initially specified, it is not declared in a USING or GIVING statement. For this reason, it is a LOCAL area allocated within each processor. The copy of the subroutine BTRI resident in each processor accesses the common area in that processor. By the time that BTRI is executing, the current instance of STEP would have initialized the appropriate part of that common block.

Within the separately compiled subroutine BTRI, the declaration of BTRI takes the form:

```
      COMMON /BTRID/ A(IUA,5,5), B(IUA,5,5), C(IUA,5,5),
     1    D(IUA,5,5), F(IUA,5)
```

```
      SUBROUTINE BTRI(IUA)
C
C     ASSUME STARTING INDEX = 1
C
      COMMON /BTRID/ A(IUA,5,5), B(IUA,5,5), C(IUA,5,5),
     1  D(IUA,5,5), F(IUA,5)
      DIMENSION H(5,5)
      IMPLICIT REAL(L)
C
C     INSERT LUDEC (SIMPLIFIED FOR DIAGONAL INPUT ARRAY B) FOR I=1
C
      L11 = 1./B(1,1,1)
      L22 = 1./B(1,2,2)
      L33 = 1./B(1,3,3)
      L44 = 1./B(1,4,4)
      L55 = 1./B(1,5,5)
C
C     COMPUTE LITTLE R'S OMITTED, THESE TEMPORARIES NOT NEEDED
C     THIS PASS, COMPUTE BIG R'S
C
       F(1,5) = L55
       F(1,4) = L44
       F(1,3) = L33
       F(1,2) = L22
       F(1,1) = L11
C
C      COMPUTE C PRIME FOR FIRST ROW
C
       DO 12 M = 1,5
C
C        C HAS BEEN ELIMINATED AS A SIMPLE
C        RESUBSCRIPTING OF THE D ARRAY
C
         B(1,5,M) = L55 x C(I,5,M)
         B(1,4,M) = L44 x C(I,4,M)
         B(1,3,M) = L33 x C(I,3,M)
         B(1,2,M) = L22 x C(I,2,M)
         B(1,1,M) = L11 x C(I,1,M)
   12  CONTINUE
```

Figure A.3 FMP FORTRAN Version of BTRI

```
C
C          HERE NOW STARTS THE MAIN LOOP OF BTRI
C
         DO 13 I = 2,IUA
C
C          COMPUTE B PRIME X BIGR
C
          DO 14 N=1,5
 14         F(I,N) = F(I,N) - A(I,N,1) X F(I-1,1) - A(I,N,2) X
     1               F(I-1,2) - A(I,N,3) X F(I-1,3) - A(I,N,4) X
     2               F(I-1,4) - A(I,N,5) X F(I-1,5)
C
C            COMPUTE B PRIME
C
             DO 11 N = 1,5
              DO 11 M = 1,5
 11            H(N,M) = B(I,N,M) - A(I,N,1) X B(I-1,1,M) -
     1                  A(I,N,2) X B(I-1,2,M) - A(I,N,C) X
     2                  B(I-1,3,M) - A(I,N,4) X B(I-1,4,M) -
     3                  A(I,N,5) X B(I-1,5,M)
C
C      INSERT LUDEC AGAIN
C
                 .
                 .
       HERE SHALL BE INSERTED A COPY OF THE FORMER LUDEC,
       EXACTLY AS SHOWN IN THE IMPLICIT CODE COMPILATION BY SCHAEFFER
                 .
                 .
C
C       COMPUTE LITTLE R'S
C
        D1 = L11 X F(I,1)
        D2 = L22 X (F(I,2) - L21 X D1)
        D3 = L33 X (F(I,3) - L31 X D1 - L32 X D2)
        D4 = L44 X (F(I,4) - L41 X D1 - L42 X D2 - L43 X D3)
        D5 = L55X(F(I,5) - L51XD1 - L52XD2 - L53XD3 - L54XD4)
```

Figure A.3 FMP FORTRAN Version of BTRI (Cont'd)

```
C
C     COMPUTE BIG R'S
C
       F(I,5) = D5
       F(I,4) = D4 -U45xD5
       F(I,3) = D3 - U34xF(I,4) - U35xD5
       F(I,2) = D2 - U23xF(I,3) - U24xF(I,4) - U25xD5
       F(I,1) = D1 - U12xF(I,2) - U13xF(I,3) - U14xF(I,4) - U15xD5
       IF (I .LT. IUA) THEN
        DO 15 M = 1,5
         D1 = L11xC(I,1,M)
         D2 = L22x(C(I,2,M) - L21xD1)
         D3 = L33x(C(I,3,M) - L31xD1 - L32xD2)
         D4 = L44x(C(I,4,M) - L41xD1 - L42xD2 - L43xD3)
          D5 = L55x(C(I,5,M) - L51xD1 - L52xD2 - L53xD3 - L54xD4)
          B(I,5,M) = D5
          B(I,4,M) = D4 - U45xD5
          B(I,3,M) = D3 - U34xC(I,4,M) - U35xD5
          B(I,2,M) = D2 - U23xB(I,3,M) - U24xB(I,4,M) - U25xD5
          B(I,1,M) = D1 - U12xB(I,2,M) - U13xB(I,3,M) - U14xB(I,4,M)
     1                - U15xD5
15        CONTINUE
        ENDIF
13     CONTINUE
C
C      THIS IS THE END OF THE MAIN I LOOP, INCLUDING I=IUA
C
C      NOTE THE NEGATIVE CODE INCREMENTS IN THE NEXT SECTION
C
       DO 20 I = IUA-1, 1, -1
        DO 19 N=1,5
19       F(I,N) = F(I,N) - F(I+1,1)xB(I,N,1) - F(I+1,2)xB(I,N,2)
20      CONTINUE
       RETURN
       END
```

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

Figure A.3 FMP FORTRAN Version of BTRI (Cont'd)

(Note: If the programmer is comfortable only with literal extents on arrys, all these declarations could be replaced by COMMON/ BTRID/ A(200,5,5), B(200,5,5), C(200,5,5), D(200,5,5), F(200,5) which, in the present instance, merely allocates some memory that was going to remain unused in any event.)

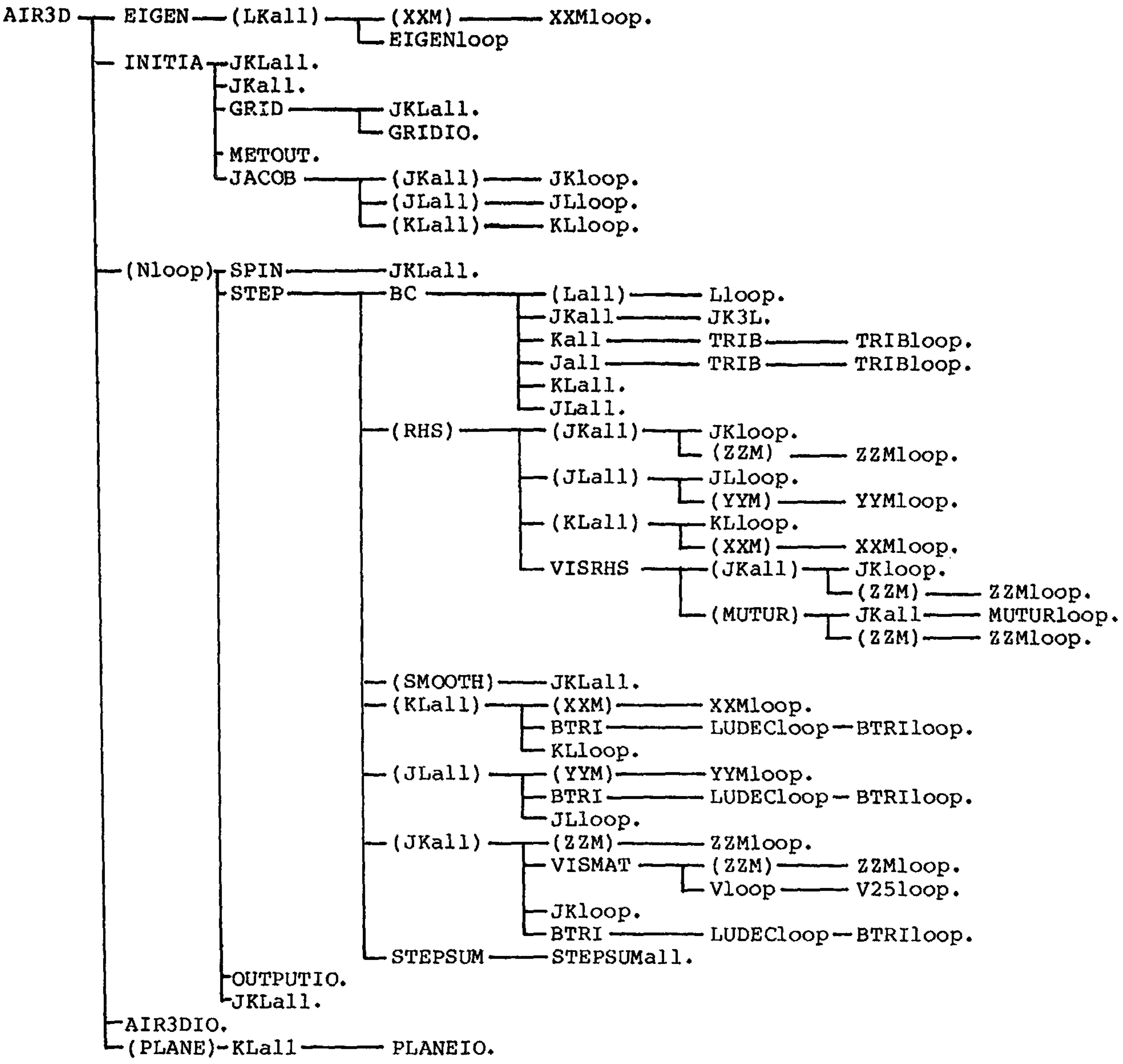
For handling a larger mesh, note that only the diagonal elements of A and C serve any real purpose. All off-diagonal element are simply copies of elements of array D with offset subscripts. Thus, with substantial complication, due to testing to see which array should be fetched at any given time, BTRID could be shoehorned into 13,000 words for the 200-long dimension, or into 19,500 words for an LMAX of 300. The present analysis ignores this possibility.

A.3.5 Analysis

Figure A.4 shows the sections of code into which the implicit program was dissected for the sake of analysis. Subsequent to this analysis, it was determined that all calls on subroutines XXM, YYM, and ZZM should be brought up into line primarily to avoid unnecessary saving of temporary variables, as described above under "assumptions".

Table A.1 shows some of the data abstracted from these sections. In Table A.1 the subroutines XXM, YYM, and ZZM have been combined into their callers.

Table A.2 shows this data recombined into an estimate of overall throughput. Rather than clutter this appendix with all intermediate computations, Table A.2 has the results accumulated by "group", where a group is a group of swatches all with the same multiplier, and the same, or approximately the same, processor utilization percentage. Three subtotals are exhibited. The first subtotal includes all the easy parts of the code, iterations or instances which are at least triply nested on the four main indices, J, K, L, and N the time step. The second subtotal includes all those swathces of code that are essentially negligible. Most of the time here represents serial computation where all the processors are computing CONTROL variables in parallel but only getting credit for the one operation that it would take in a serial machine to compute this value. The third subtotal gathers together some operations where one-dimensional DOALLs, with fewer instances than there are processors, result in low processor utilization. Even so, operations with low processor utilization are essentially negligible at the problem size considered here.



Key: CAPS = Program units.
-all = DOALL over indicated variables.
- loop = DO loop.
( ) = null node except for entry and return.

Figure A.4 Breakdown of Implicit Code into Segments of Code and Nodes for Analysis

A.3.5.1 Description of Table A.1

In the "multiplier" column, N, J, K and L are abbreviations for NMAX, JMAX, KMAX and LMAX respectively. Below that, is the multiplier ("E" stands for "times 10 to the") which results when these extents are replaced by 100, 100, 50, and 200 respectively.

"Ident" is the identifier from Figure A.4. Flops and EM accesses are the result of a hand count of operations.

"Special Case" is the column for notes. The only special cases noted are excess divisions, the occurrence of the SUMALL global function, and

> Note 1: Many of the variables accessed here involve triply or quadruply subscripted array elements. The progression of subscripts is extremely regular, say indexed on loop variables and by literals. It is assumed that the compiler or the programmer has reduced these subscript computations to not more than one integer multiply per accessed element. There are several ways to accomplish this simplification.
>
> Note 2: In reevaluating BTRI for this analysis, a substantially higher portion of the floating point operations were identified as FMAD than in the hand compiling that led to the simulator input. A small adjustment was made on account of this observation.

The notation "(x3)" or "(x2)" is used to signify that there are three or two nodes or sections in the branching tree (Figure A.4) with identical instruction counts, and identical number of times of execution. There seemed no need to repeat identical entries in the table.

TABLE A.1

Characterization of Implicit Code Sections

Multiplier	Ident.	Flops/ Section	EM access/ Section	Special Case	T	Proc. Util.
75NJKL 75E8	BTRIloop	10	0	Note 1	1.17	97.4%
25NJKL 25E8	VISMATloop	7	0	Note 1	1.17	97.4%
NJKL	STEPJKloop	131	29		0.81	98.4%
1E8	STEPKLloop	117	25		0.81	96.0%
	STEPJLloop	117	25		0.81	96.0%
	SPINJKL	6	4		0.39	97.4%
	BCJKL	10	6		0.43	
	RHSJKLoop	64	17		0.74	98.4%
	RHSJLloop	64	22		0.61	96.0%
	RHSKLloop	64	22		0.61	96.0%
	VISRHSJKloop	210	19		1.17	98.4%
	MUTURloop	533	99		0.89	98.4%
	LUDEC	376(x3)	0(x3)	Note 2	1.35	97.4%
	VISMATloop	224	18		1.23	98.4%

TABLE A.1 continued

Characterization of Implicit Code Sections

Multiplier	Ident.	Flops/ Section	EM access/ Section	Special Case	T	Proc. Util.
5NJKL	SMOOTHJKL	190	72		0.60	99.97%
5E8						
NJKL	STEPSUM	10	5	SUMALL on 10^6 inst's	0.50	99.97%
1E8						
NJK	BCJK	667	80		1.04	96.0%
5E5	MUTURJK	170	2		1.28	
	VISMAT	5	0		1.28	
	BTRI	10	0	2 DIV	1.05	
NJL	BCJL	12	24		0.15	98.4%
2E6	BTRI	10	0	2 DIV	1.05	
NKL	BCKL	33	16		0.49	96.0%
1E6	BTRI	10	0	2 DIV	1.05	
JKL						97.4%
1E6	EIGENloop	228	56		1.06	
	INITIAJKL	6	6		0.29	
	GRIDJKL	3	6		0.154	
	JACOBloop(s)	38	24		0.48	
	MAINloopJKL	11	5		0.52	
JK	INITIAJK	1	3	1DIV	0.088	96.0%
5E3	JACOBJK	0	2	No flops	0.000	
JL	JACOBJL	0	2	No flops	0.000	98.4%
2E4						
KL	JACOBKL	2	3		0.197	96.0%
1E4	PLANEKL	121	10		1.18	

TABLE A.1 continued

Characterization of Implicit Code Sections

Multiplier	Ident.	Flops/ Section	EM access/ Section	Special Case	T	Proc. Util.
NJ	BCJ	4	0		0.26	19.2%
1E4	TRIB	3	0	2 DIV	0.105	
NK	BCK	141	6		0.133	9.6%
5E3	TRIB	3	0	2 DIV	0.053	
N	SPIN	81	0		0.0026	0.19%
1E2	STEP	3	0		0.0026	
	BC	99	0		0.0026	
	VISRHS	5	0		0.0026	
	STEPSUMser	25	0		0.0026	
1	AIR3D	1	2		0.0003	0.19%
1E0	EIGEN	9	0	5 DIV	0.0012	
	INITIA	139	0	11 DIV	0.0021	
NJK 5E5	TRIBloop	10(x2)	0	1D DOALLs	0.21	16.0%
NKL 1E6	BCLloop	43	37	1D DOALL over L	0.126	38.4%

TABLE A.2
Throughput Computations for Implicit Code

Group	Proc. Util.	Flops per	Multi- plier	Total Flops	Time (sec.)	Throughput
NJKL	97.4%	3593	1E8	3583E8	343.2	
NJKL*	99.9%	190	1E8	190E8	31.6	
NJK	96.0%	852	5E5	4.3E8	.398	
NJL	98.4%	22	2E6	.44E8	.083	
NKL	96.0%	43	1E6	.43E8	.048	
JKL	97.4%	314	1E6	3.14E8	.50	
Subtotal				3792E8	375.8	1.010
JK	96.0%	1	5E3	.5E4	.000078	
JL	98.4%	0	2E4	0E4	.000117	
KL	96.0%	123	1E4	123E4	.000078	
NJ	19.2%	7	1E4	7E4	.000269	
NK	9.6%	3	5E3	1.5E4	.000024	
N	0.19%	210	1E2	2.1E4	.00794	
1	0.19%	149	1	149E0	.000057	
Subtotal				134E4	.00856	0.157
NJK*	16.0%	20	5E5	10E6	.046	
NKL*	38.4%	43	1E6	43E6	.341	
Subtotal				50E6	.387	0.129
TOTAL				3792E8	376.2	1.009

A.4 THROUGHPUT OF EXPLICIT AERO FLOW CODE

A.4.1 Summary

A.4.1.1 Results

A throughput rate of 0.89 gigaflops/second at an average system processor utilization of 97.7 percent is estimated for the Hung/MacCormack explicit aero flow code. This estimate is based on an assumed grid size of 100 x 100 x 100 elements and 100 time steps. A total of 4.73 x 10^{11} floating point arithmetic operations are executed in 100 time steps, in 532 seconds. An extended memory data base of approximately nine million words is also required.

A.4.1.2 Observations

The following general observations were made. Some of these show up as conclusions in Chapter 3.

° A direct conversion of this algorithm into extended FMP FORTRAN was accomplished, with considerable ease. All first and third level subroutines (19 of 30) require basically no change.

° The ease and efficiency of translation to FMP machine code was also excellent. A major compiler requirement is minimization of address indexing operations through recognition of common subexpressions which are abundant.

° The correct algorithm includes a considerable amount of simple moves from one Extended Memory address to another. This is visible in routines BCY, PRSETY, BCZ, PRSETZ and OUTER.

A.4.2 Assumptions

The basic formula used for calculating the total time per module was transformed to:

Time = K1*#Flops + K2*#EM + K3*#Divs
K1 = 295 nano seconds per floating point operation (flop)
K2 = 1500 nano seconds per EM access (#EM). This value includes time for address calculation.
K3 = 1460 per divide operation (DIV) in excess of 2 percent of the total flop count.

This approach is verified for the explicit code through detailed simulation of selected typical code segments. Subroutines LX and FX were selected for this purpose. This data is included in Figure A.1.

° The algorithm is a tree structured set of thiry subroutines on three levels. Figure A.4 depicts this structure.

All level two routines are modified to employ the FMP DOALL statement in place of the current dual nested DO statements. The level one main program initializes GLOBAL variables only. All level three routines are local subroutines (with copies resident in each processor) to be executed in parallel as they stand.

° All GLOBAL values and simple constants are stored locally in all processors.

° The grid size chosen for analysis was 100 x 100 x 100.

A.4.3 Method of Analysis

The initial phase of investigation was a review of available background material. The Navier-Stokes equations are the essential mathematical model of the dynamics of a compressible-fluid flow. Reference [A.1] provides the description of an explicit discrete mathematical algorithm for solving these equations. NASA supplied this methodology, and the FORTRAN listing of the resulting program. Figure A.5 shows this program's structure. This information was then synthesized into Figure A.6, a list of subroutine groups and the identification of the program's major outer loop. Further detailed analysis of individual code segments determined the number of static calls on each subroutine.

The next analysis was the identification of major data classes. The following standard FMP classes were identified:

(a) Nine three dimensional shared arrays from the data base. These arrays are in common and are STRUCTURE variables in Extended Memory. This data is accessed via three dimensional subscripts representing mesh points.

(b) System wide scalar variables (GLOBAL variables). These variables are replicated in all processor local memories.

(c) Local common in processor memory. No communication of this data between processors is required.

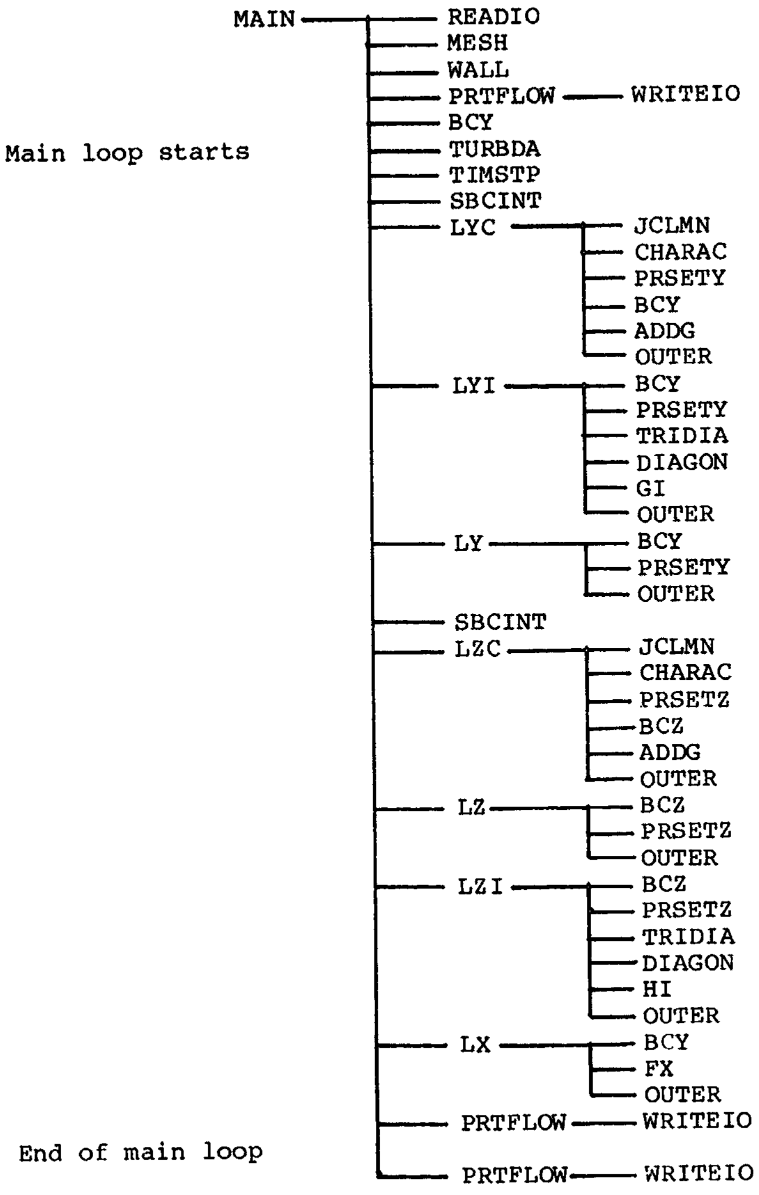


Figure A.5 Calling Tree of Explicit Aero Flow Code and Segments for Analysis

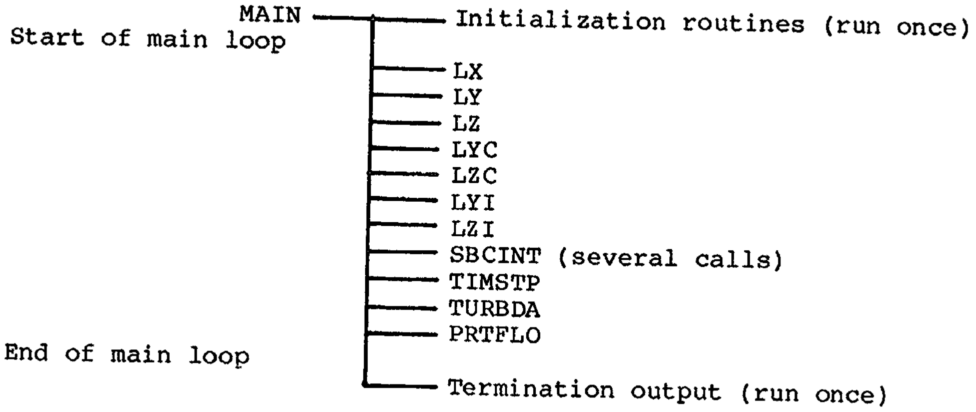


Figure A.6 Summary of Calling Tree for Explicit Code

(d) STRUCTURE data wherein an array of elements, one per mesh point, may be kept in individual processors. Subroutine SBCINT contains a three dimensional array "SBC" of this type.

(e) Strictly local temporary data for a single subroutine or DOALL block.

(f) Nameless temporary working store typically required in expression evaluations. These are typically assigned to processor registers by the compiler.

Except for type (d), examples are visible in the FMP FORTRAN version of subroutines LX and FX (Figures A.7 and A.8).

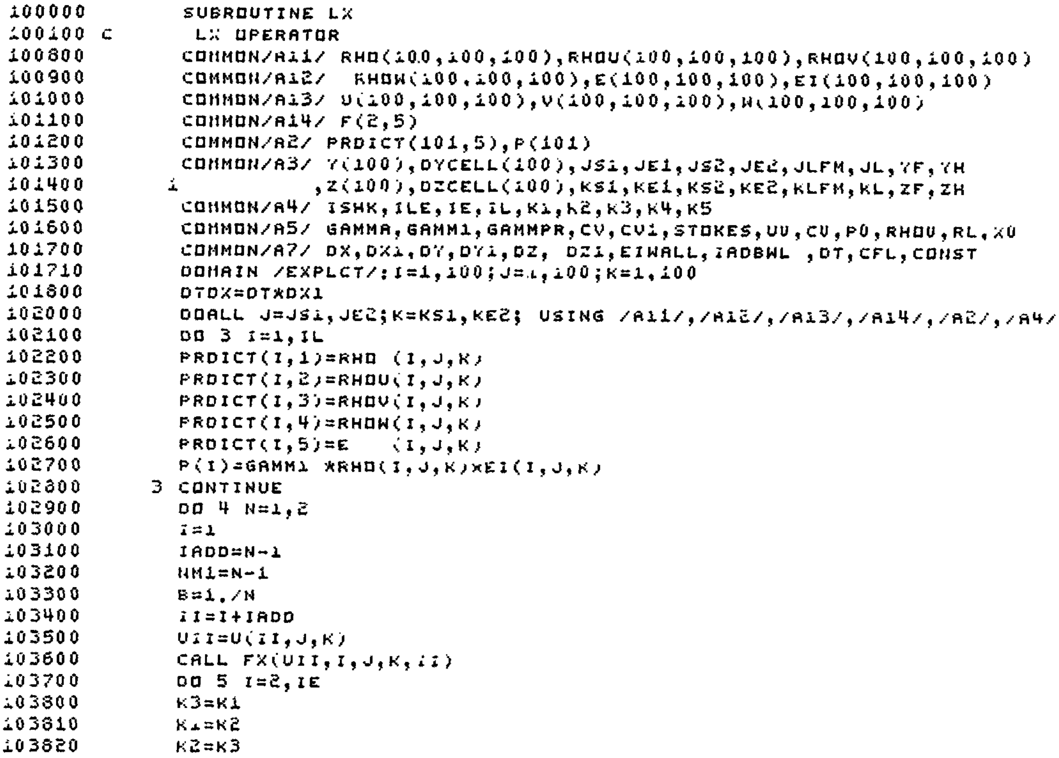
The next step in the analysis was a survey of all subroutines to identify the DOALL statements. No such statements are required in the main procedure or in any level three subroutine, which are all local to the instances of the DOALL's. All level two subroutines contain dual nested DO loops, which are directly converted to a DOALL with 10,000 instances. The LX subroutine provides a typical example. (See Figure A.7) Note in the listing of LX (in Figure A.7), that the DOALL begins at line 102000 and ends at line 107600. Thus, almost all of LX consists of 10,000 instances of this code (and the call to FX at line 104500 in each instance). Thus, twenty cycles are required of each level two and three subroutine to execute the 10,000 instances giving a processor utilization of 97.7 percent.

The initialization routines MESH and WALL, being executed only once, were ignored. The output routine PRTFLN was also ignored. The next phase consisted of counting floating point arithmetic operations, floating point divide operations and Extended Memory accesses in the subroutines.

This count includes the effects of DO loops, fine or coarse grid partial subscript range values and the program's branching structure. This information is given in the various columns of Tables A.3 and A.4. The product of these counts then produced a total count of operations per subroutine. The application of the formula: time = K1*#Flops+K2*#EM+K3*# Divs, then gave a total execution time per program module. The total number of flops was also given by the product of the number of flops per module times the number of active modules. The system throughput rate T_p was then compared to:

$$T_p = \frac{\sum \text{Flops}}{\sum \text{Time}}$$

which is an average flops/second rate value.

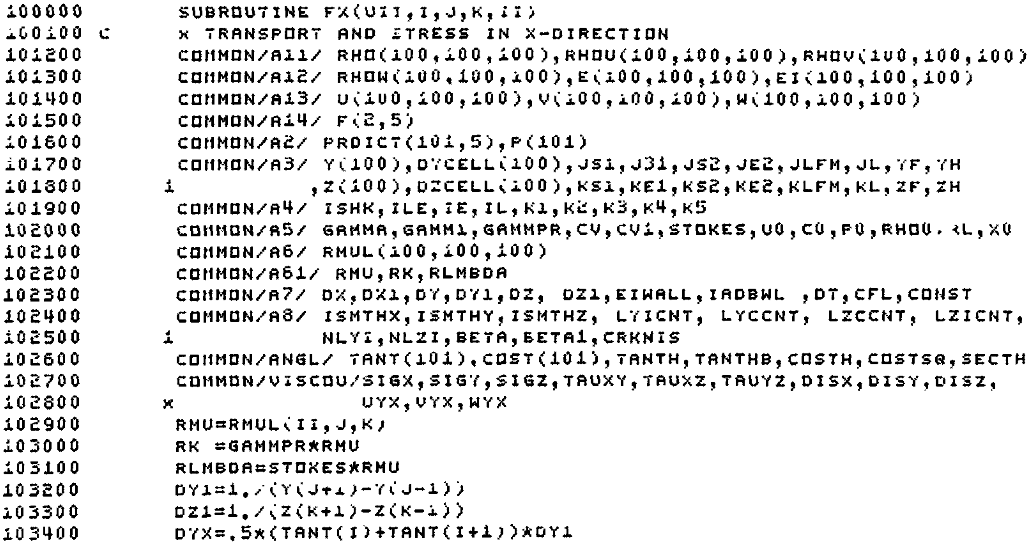


```
      SUBROUTINE LX
C      LX OPERATOR
      COMMON/A11/ RHO(100,100,100),RHOU(100,100,100),RHOV(100,100,100)
      COMMON/A12/  RHOW(100,100,100),E(100,100,100),EI(100,100,100)
      COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PROICT(101,5),P(101)
      COMMON/A3/ Y(100),DYCELL(100),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1           ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/ GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,UU,CU,P0,RHOU,RL,X0
      COMMON/A7/ DX,DX1,DY,DY1,DZ, DZ1,EIWALL,IADBWL ,DT,CFL,CONST
      DOMAIN /EXPLCT/:I=1,100;J=1,100;K=1,100
      DTDX=DT*DX1
      DOALL J=JS1,JE2;K=KS1,KE2; USING /A11/,/A12/,/A13/,/A14/,/A2/,/A4/
      DO 3 I=1,IL
      PROICT(I,1)=RHO (I,J,K)
      PROICT(I,2)=RHOU(I,J,K)
      PROICT(I,3)=RHOV(I,J,K)
      FROICT(I,4)=RHOW(I,J,K)
      PROICT(I,5)=E   (I,J,K)
      P(I)=GAMM1 *RHO(I,J,K)*EI(I,J,K)
    3 CONTINUE
      DO 4 N=1,2
      I=1
      IADD=N-1
      NM1=N-1
      B=1./N
      II=I+IADD
      UII=U(II,J,K)
      CALL FX(UII,I,J,K,II)
      DO 5 I=2,IE
      K3=K1
      K1=K2
      K2=K3
```

Figure A.7 FMP FORTRAN Version of LX

```
      II=I+IADD
      UII=U(II,J,K)
      UI1=U(I+1,J,K)
      UI2=U(I,J,K)
      IF(UI1.GT.UI2.AND.(3.*UI1-UI2)*(3.*UI2-UI1).LT.0.) UII=.5*(UI1+UI2
     * )
      CALL FX(UII,I,J,K,II)
      PRDICT(I,1)=(NM1*PRDICT(I,1)+RHO (I,J,K)-DTDX*(F(K2,1)-F(K1,1)))*B
      PRDICT(I,2)=(NM1*PRDICT(I,2)+RHOU(I,J,K)-DTDX*(F(K2,2)-F(K1,2)))*B
      PRDICT(I,3)=(NM1*PRDICT(I,3)+RHOV(I,J,K)-DTDX*(F(K2,3)-F(K1,3)))*B
      PRDICT(I,4)=(NM1*PRDICT(I,4)+RHOW(I,J,K)-DTDX*(F(K2,4)-F(K1,4)))*B
      PRDICT(I,5)=(NM1*PRDICT(I,5)+E    (I,J,K)-DTDX*(F(K2,5)-F(K1,5)))*B
    5 CONTINUE
C---
C     * DECODE  *
      DO 6 I=2,IE
      RHOI=1./PRDICT(I,1)
      U (I,J,K)=PRDICT(I,2)*RHOI
      V (I,J,K)=PRDICT(I,3)*RHOI
      W (I,J,K)=PRDICT(I,4)*RHOI
      EI(I,J,K)=PRDICT(I,5)*  RHOI      -.5*(U(I,J,K)**2+V(I,J,K)**2+W(I
     * ,J,K)**2)
      P(I)      =GAMM1*PRDICT(I,1)*EI(I,J,K)
    6 CONTINUE
C***   *DOWNSTREAM B. C. AT I=IL
      DO 9 K6=1,5
    9 PRDICT(IL,K6)=PRDICT(IE,K6)
      CALL BCY(K,2,IE,J,J)
    4 CONTINUE
      DO 7 I=2,IL
      RHO (I,J,K)=PRDICT(I,1)
      RHOU(I,J,K)=PRDICT(I,2)
      RHOV(I,J,K)=PRDICT(I,3)
      RHOW(I,J,K)=PRDICT(I,4)
      E   (I,J,K)=PRDICT(I,5)
    7 CONTINUE
      ENDDO; GIVING/A11/,/A12/,/A13/,/A2/
      CALL OUTER(JS1,JE2,KS1,KE2)
      RETURN
      END
```

Figure A.7 FMP FORTRAN Version of LX (Cont'd)



```
      SUBROUTINE FX(UII,I,J,K,II)
C     x TRANSPORT AND STRESS IN X-DIRECTION
      COMMON/A11/ RHO(100,100,100),RHOU(100,100,100),RHOV(100,100,100)
      COMMON/A12/ RHOW(100,100,100),E(100,100,100),EI(100,100,100)
      COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PROICT(101,5),P(101)
      COMMON/A3/ Y(100),DYCELL(100),JS1,J31,JS2,JE2,JLFM,JL,YF,YH
     1           ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/ GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,U0,C0,P0,RHO0.RL,X0
      COMMON/A6/ RMUL(100,100,100)
      COMMON/A61/ RMU,RK,RLMBDA
      COMMON/A7/ DX,DX1,DY,DY1,DZ, DZ1,EIWALL,IADBWL ,DT,CFL,CONST
      COMMON/A8/ ISMTHX,ISMTHY,ISMTHZ, LYICNT, LYCCNT, LZCCNT, LZICNT,
     1           NLYI,NLZI,BETA,BETA1,CRKNIS
      COMMON/ANGL/ TANT(101),COST(101),TANTH,TANTHB,COSTH,COSTSQ,SECTH
      COMMON/VISCOU/SIGX,SIGY,SIGZ,TAUXY,TAUXZ,TAUYZ,DISX,DISY,DISZ,
     x                UYX,VYX,WYX
      RMU=RMUL(II,J,K)
      RK =GAMMPR*RMU
      RLMBDA=STOKES*RMU
      DY1=1./(Y(J+1)-Y(J-1))
      DZ1=1./(Z(K+1)-Z(K-1))
      DYX=.5*(TANT(I)+TANT(I+1))*DY1
```

Figure A.8 FMP FORTRAN Version of FX

```
      UYX=U(II,J+1,K)-U(II,J-1,K)
      VYX=V(II,J+1,K)-V(II,J-1,K)
      SIGX=P(II)     -(RLMBDA+2.*RMU)*((U(I+1,J,K)-U(I,J,K))*DX1-UYX*DYX)
     * -RLMBDA*(VYX*DY1+(W(II,J,K+1)-W(II,J,K-1))*DZ1)
      TAUXY=-RMU*(UYX*DY1+(V(I+1,J,K)-V(I,J,K))*DX1-VYX*DYX)
      TAUXZ=-RMU*((U(II,J,K+1)-U(II,J,K-1))*DZ1+(W(I+1,J,K)-W(I,J,K))*
     * DX1-(W(II,J+1,K)-W(II,J-1,K))*DYX)
      DISX=SIGX*UII+TAUXY*V(II,J,K)+TAUXZ*W(II,J,K)-RK*((EI(I+1,J,K)-EI(
     *I,J,K))*DX1-(EI(II,J+1,K)-EI(II,J-1,K))*DYX)
      F(K2,1)=PRDICT(II,1)*UII
      F(K2,2)=PRDICT(II,2)*UII+SIGX
      F(K2,3)=PRDICT(II,3)*UII+TAUXY
      F(K2,4)=PRDICT(II,4)*UII+TAUXZ
      F(K2,5)=PRDICT(II,5)*UII+DISX
      IF(ISMTHX.EQ.0 .OR. I.LE.1 .OR. I.GE.IE) RETURN
C     * SMOOTHING TERMS  *
C
      COEF=CONST*ABS(P(II+1)-2.*P(II)+P(II-1))/(ABS(P(II+1))+
     *  2.*ABS(P(II))+ABS(P(II-1)))
      CII=SQRT(GAMMA*GAMM1*ABS(EI(II,J,K)))
      COEF=COEF*(ABS(U(II,J,K))+CII)
      DO 9 K6=1,5
    9 F(K2,K6)=F(K2,K6)-COEF*(PRDICT(I+1,K6)-PRDICT(I,K6))
      RETURN
      END
←
```

Figure A.8 FMP FORTRAN Version of FX (Cont'd)

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

Table A.3

Performm.nce Analysis of Explicit Aero Flow Code

Routine	# Calls/ Time Step	Cycles	# Time Stop³	Total # Calls	# Flops/Call	# Divs/Call	# EM's/Call	Total Time/ Call usec	Total Time usec	Total Flops/CPU	# Active CPU	Total Flops
INITIALIZATION												
. MAIN	1	1	100	100	0	0	0	0	0	0	1	0
. MESH	0	-	-	-	-	-	-	-	-	-	-	-
. WALL	0	-	-	-	-	-	-	-	-	-	-	-
. BCY	0	-	-	-	-	-	-	-	-	-	-	-
. PRT FLW	0	-	-	-	-	-	-	-	-	-	-	-
MAIN PROGRAM LOOP												
. LX	2	20	100	4000	9902	198	3241	8251	3.30E07	3.96E07	500	1.98E10
. FX	404	20	100	808000	89	3	25	71.2	5.75E07	7.19E07	500	3.60E10
.SQRT	404	20	100	808000	14	0	0	4.44	3.99E06	1.13E07	500	5.65E09
. BCY	4	20	100	8000	0	0	12	18	1.44E05	0	500	0
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. LY	2	20	100	4000	15665	387	3072	10042	4.02E07	6.27E07	500	3.13E10
. BCY	4	20	100	8000	0	0	12	18	1.44E05	0	500	0
. PRSETY	8	20	100	16000	266	0	333	582	9.31E06	4.25E06	500	2.13E09
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. SQRT	196	20	100	392000	14	0	0	494	1.94E06	5.48E06	500	2.74E09
. LZ	2	20	100	4000	11632	288	2688	8072	3.23E07	4.56E07	500	2.33E10
. BCZ	4	20	100	8000	0	0	12	18	1.44E05	0	500	0
. PRSETZ	8	20	100	16000	266	0	333	82	9.31E06	9.25E06	500	2.13E09
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. SQRT	196	20	100	292000	14	0	0	4.95	1.94E06	5.48E06	500	2.74E09
. LYC	2	20	100	4000	4913	4	799	2910	1.16E07	1.97E07	500	9.83E09
. ADDG	196	20	100	392000	24	1	0	8.5	3.33E06	9.40E06	500	4.70E09
. BCY	2	20	100	4000	0	0	12	18	7.20E04	0	500	0
. CHARAC	2	20	100	4000	36263	2012	0	13400	5.36E07	1.45E08	500	7.25E10
. PRSETY	6	20	100	12000	266	0	333	582	6.98E06	3.19E06	500	1.60E09
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. JCLMN	2	20	100	400	125	101	0	161	6.44E05	5.00E05	1	5.00E05
. SQRT	196	20	100	392000	14	0	0	4.94	1.94E06	6.27E06	500	3.14E09

Routine	# Calls/ Time Step	Cycles	# Time Steps	Total # Calls	# Flops/Call	# Divs/Call	# EM's/Call	Total Time/ Call usec	Total Time usec	Total Flops/CPU	# Active CPU	Total Flops
. LZC	2	20	100	4000	4313	2	898	2838	1.14E07	1.72E07	500	8.62E09
. ADDG	196	20	100	392000	24	1	0	8.5	3.33E06	9.40E06	500	4.70E09
. BCZ	2	20	100	4000	0	0	12	18	7.20E04	0	500	0
. CHARAC	2	20	100	4000	36263	2012	0	13400	5.36E07	1.45E08	500	7.25E10
. PRSETZ	6	20	100	12000	266	0	333	582	6.98E06	3.19E06	500	1.60E09
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. JCLMN	2	20	100	4000	125	101	0	161	6.44E05	5.00E05	1	5.00E05
. SQRT	196	20	100	392000	14	0	0	4.94	1.94E06	6.27E06	500	3.14E09
. LYI	2	20	100	4000	12333	105	1700	6852	2.74E07	4.93E07	500	2.46E10
. BCY	4	20	100	8000	0	0	12	18	1.44E05	0	500	0
. DIAGON	196	20	100	392000	128	1	0	45.2	1.77E07	5.02E07	500	2.50E10
. GI	200	20	100	400000	54	2	14	39.7	1.59E07	2.16E07	500	1.08E10
. PRSETY	8	20	100	16000	266	0	333	582	9.31E06	4.25E06	500	2.13E09
. TRIDATA	28	20	100	56000	650	50	0	258	1.44E07	3.64E07	500	1.82E10
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. LZI	2	20	100	4000	10096	201	2000	6514	2.61E07	4.04E07	500	2.02E10
. BCZ	4	20	100	8000	0	0	12	18	1.44E05	0	500	0
. DIAGON	196	20	100	392000	128	1	0	45.2	1.77E07	5.02E07	500	2.50E10
. HI	200	20	100	400000	33	2	17	36.7	1.47E07	1.32E07	500	6.60E09
. PRSETZ	8	20	100	16000	266	0	333	582	9.31E06	4.25E06	500	2.13E09
. OUTER	2	20	100	4000	0	0	20	30	1.20E05	0	500	0
. TRIDATA	28	20	100	56000	650	50	0	258	1.44E07	3.64E07	500	1.82E10
. SBCINT	4	20	100	4000	0	0	0	0	0	0	500	0
. TIMESTP	1	20	100	2000	7392	990	1182	5730	1.15E07	1.48E07	500	7.39E09
. SQRT	98	20	100	196000	14	0	0	4.94	9.68E05	2.74E06	500	1.37E09
. TURBDA	2	20	100	4000	800	101	300	937	3.75E06	3.20E06	500	1.60E09
. SQRT	200	20	100	400000	14	0	0	4.94	1.98E06	5.60E06	500	2.80E09
. PRTFLW	0.1	20	100	-	-	-	-	-	-	-	-	-

```
Total Execution Time   =  5.32E08 µs
Total Flops            =  4.73E11
Throughput             =  0.89 Gigaflops/Second
```

Table A.4

Throughput Computations for Explicit Code

PARAMETERS		
	100 x 100 x 100	GRID SIZE
	100	TIME STEPS

ROUTINE	TOTAL TIME - us	TOTAL FLOPS	THROUGHPUT
LX	3.30E07	1.98E10	0.60
FX	5.75E07	3.60E10	0.63
LY	4.02E07	3.13E10	0.78
LYC	1.16E07	9.83E09	0.85
LYI	2.74E07	2.46E10	0.95
LZ	3.23E07	2.33E10	0.72
LZC	1.14E07	8.62E09	0.76
LZI	2.61E07	2.02E10	0.77
SQRT	1.47E07	2.08E10	1.41
CHARAC	1.07E08	1.45E11	1.36
DIAGON	3.54E07	5.00E10	1.41
TRIDATA	2.88E07	3.64E10	1.26
PRSETY	2.55E07	5.86E09	0.23
PRSETZ	2.55E07	5.86E09	0.23
GI	1.59E07	1.08E10	0.68
HI	1.47E07	6.60E09	0.45
ADDG	6.66E06	9.40E09	1.41
JCLMN	1.28E06	1.00E06	0
BCY	5.04E05	0	0
BCZ	5.04E05	0	0
OUTER	8.40E05	0	0
SBCINT	0	0	0
TIMESTP	1.15E07	7.39E09	0.64
TURBDA	3.75E06	1.60E09	0.43
PRTFLW			
	5.32E08 us	4.73E11	.89 Gflops/ sec

A.4.4 Simulation and Hand Compiling

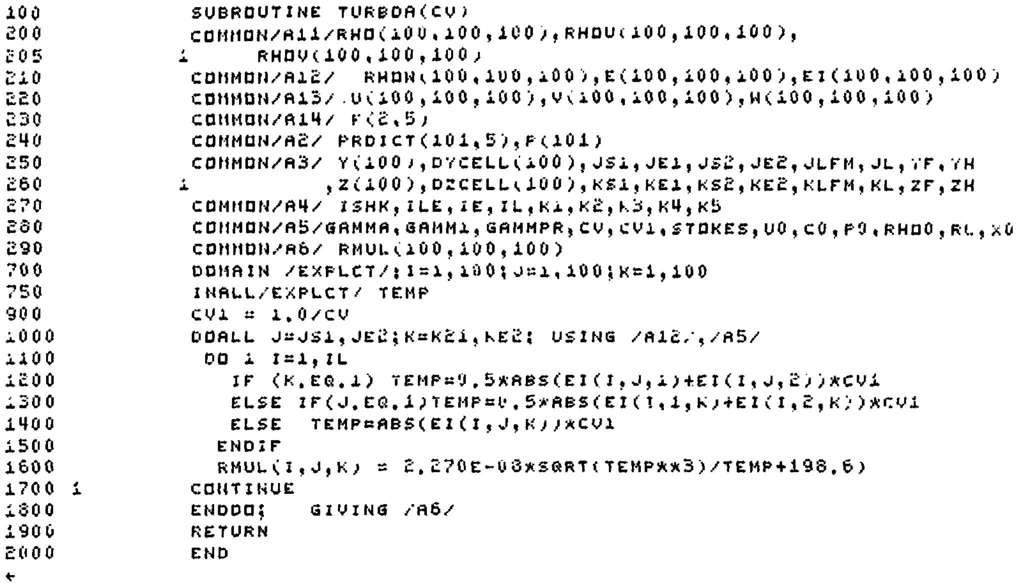
A validation of the above analysis was conducted by simulated execution of a typical code section. The FMP simulator is described in Chapter 7.

The main stream subroutines encompassing the bulk of execution time were LX, LY, LZ, LYC, LZC, LAX, LYI, LZI and their associated third level subroutines. The subroutine LX and its associated level three subroutine FX were selected as representative of this algorithm. LX and FX are shown in Figures A.7 and A.8 respectively. Figures A.9 and A.10 show the FMP FORTRAN versions of TURBDA and OUTER which were also simulated. No special handling was needed on these subroutines. Each is a demonstration of a simple conversion of nested DO loops to a DOALL construct.

The initial effort in preparing the simulator input was the revision of the original FORTRAN code sections into the extended FMP FORTRAN language. Modifications were primarily in the areas of data declarations, domain declarations, and DOALLs. Assignment to GLOBAL variables was assumed done in parallel across all processors. In addition to these changes, the code was reviewed for areas in which an optimizing compiler could be expected to achieve time savings. These changes typically take the form of a new local temporary variable holding the evaluation of a common subexpression in order to improve performance.

In particular, common subscript expressions were detected and evaluated separately during both the hand analysis and the simulations. These expressions all involve the integer mode sum or products used to compute an address from the subscript values. Although a mature, optimizing compiler will find such common subscript expressions and combine the results transparently to the user, the hand analysis performed this level of optimization by hand.

For example in the subroutine FX, 25 three dimensional subscript expressions may be reduced to seven common expressions. Other changes such as the use of an iterated DO Loop rather than straight line code were made to reduce the size of the generated machine code file.



```
      SUBROUTINE TURBDA(CV)
      COMMON/A11/RHO(100,100,100),RHOU(100,100,100),
     1     RHOV(100,100,100)
      COMMON/A12/  RHOW(100,100,100),E(100,100,100),EI(100,100,100)
      COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PRDICT(101,5),P(101)
      COMMON/A3/ Y(100),DYCELL(100),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1          ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,U0,C0,P9,RHO0,RL,X0
      COMMON/A6/ RMUL(100,100,100)
      DOMAIN /EXPLCT/;I=1,100;J=1,100;K=1,100
      INALL/EXPLCT/ TEMP
      CV1 = 1.0/CV
      DOALL J=JS1,JE2;K=K21,KE2; USING /A12/,/A5/
       DO 1 I=1,IL
         IF (K.EQ.1) TEMP=9.5*ABS(EI(I,J,1)+EI(I,J,2))*CV1
         ELSE IF(J.EQ.1)TEMP=0.5*ABS(EI(I,1,K)+EI(I,2,K))*CV1
         ELSE  TEMP=ABS(EI(I,J,K))*CV1
        ENDIF
        RMUL(I,J,K) = 2.270E-08*SQRT(TEMP**3)/TEMP+198.6)
1     CONTINUE
      ENDDO;   GIVING /A6/
      RETURN
      END
←
```

Figure A.9 FMP FORTRAN Version of TURBDA

```
         SUBROUTINE OUTER(JS,JE,KS,KE)
         COMMON/A11/ RHO(100,100,100),RHOU(100,100,100),RHOV(100,100,100)
         COMMON/A12/  RHOW(100,100,100),E(100,100,100),EI(100,100,100)
         COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
         COMMON/A3/ Y(100),DYCELL(100),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
        1             ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
         COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
C          DOWNSTREAM AT I=IL
         DOALL K=KS,KE;J=JS,JE ;  USING/A11/,/A12/,/A3/,/A4/
          RHO(IL,J,K) = RHO(IE,J,K)
          RHOU(IL,J,K) = RHOU(IE,J,K)
          RHOV(IL,J,K) = RHOV(IE,J,K)
          RHOW(IL,J,K) = RHOW(IE,J,K)
          E(IL,J,K) = E(IE,J,K)
         ENDDO ;  GIVING /A11/,/A12/
           IF (JE.LT.JE2) GO TO 3
C          UPPER B, C, AT J=JL
         DOALL K=KS,KE;I=2,IE ;  USING/A11/,/A12/,/A3/,/A4/
          RHO(I,JK,K) = RHO(I,JE2,K)
          RHOU(I,JK,K) = RHOU(I,JE2,K)
         RHOV(I,JK,K) = RHOV(I, JE2,K)
          RHOW(I,JK,K) = RHOW(I,JE2,K)
          E(I,JK,K) = E(I,JE2,K)
         ENDDO; GIVING /A11/,/A12/
3         IF (K.GE.KE2) THEN
C           EDGE B,C, AT K=KL
           DOALL J=JS,JE;I=2,IE ;  USING /A11/,/A12/,/A3/,/A4/
            RHO(I,J,KL) = RHO(I,J,KE2)
            RHOU(I,J,KL) = RHOU(I,J,KE2)
            RHOV(I,J,KL) = RHOV(I,J,KE2)
            RHOW(I,J,KL) = RHOW(I,J,KE2)
            E(I,J,KL) = E(I,J,KE2)
           ENDDO; GIVING /A11/,/A12/
          ENDIF
         RETURN
        END
```

Figure A.10 FMP FORTRAN Version of OUTER

The next phase was hand compiling. It was assumed that the first four registers in all groups were designated scratch registers. They were also used for passing parameters and results to and from subroutines. The remaining registers were employed for longer lifetime storage requirements.

Experience during the hand compilation demonstrated that the need for integer registers exceeded the supply. As a result, storing and restoring of these registers had to be employed.

In subroutine SBCINT and JCLMN a non-standard approach was assumed. Both routines have a very minor impact on total throughput. The SBCINT routine performs a clearing operation on the three dimensional array SBC and is called four times. SBC was declared to be an INALL array. A single statement, SBC=0, therefore clears it. The routine JCLMN is called from LYC and LZC subroutines outside of their DOALLs. Although the routine could be programmed using recurrence, there seems to be no advantage. This routine is, therefore, assumed to be executed serially.

A.5 GISS CLIMATE PERFORMANCE EVALUATION

A.5.1 Summary

The evaluation described below was done on an intermediate size (2° latitude steps, 2.5° longitude increments along the equator) weather program. The program consists of an easily vectorizable fluid dynamics section (subroutines COMP1 and COMP2 and the subroutines they in turn call), and a hard-to-vectorize physics and chemistry section (COMP3 and its subroutines). The average throughput for the entire program was determined to be 0.532 Gflops/sec. The time for a 14-day simulation with 20 minute time steps was projected to be 4 minutes, 25 seconds.

A GISS weather demonstrated the advantages of the FMP architecture over that of a vector machine. The vectorizable portions of the program tended to run slow because of many EM accesses, but the unvectorized portion of the program, namely COMP3 and its subroutines, ran at 1.2 Gflops/sec for the portion simulated.

A.5.2 Discussion of the Analysis

The following versions of the weather model codes were provided by NASA as input for selecting an FMP benchmark test.

GISS Models

A. 360/65 version
B. 360/195 version
C. STAR 100 version
D. ILLIAC IV version

The various versions are machine dependent versions of the Mintz-Arakawa differencing scheme which numerically solve the differential equations representing the physical dynamics of weather conditions. Reference [2] describes this methodology.

The basic database for the GISS model is a series of three dimensional arrays. The data values in individual arrays represent temperature, pressure, humidity, etc. at each point of the assumed latitude, longitude and altitude grid. Arrays of one and two dimensions are also utilized in various code sections in addition to various simple scalar values.

Minor variations in GISS versions exist due to selecting different granularities in grid size, time step, split grid, step size, I/O management and the nature of the Host machine architecture (Scalar/360, Vector/Star Array/ILLIAC) considerations. Grid sizes vary from a coarse (25, 40, 2) to a superfine (180, 288, 9) as indicated in Table 2.1 of reference [1]. The historical increase in computing power has provided the facilities for including the larger grid sizes and smaller time steps and thereby improving the accuracy of results.

A medium sized grid of (89, 144, 9) was selected for FMP Benchmark purposes. This size is a valid test of the system's dexterity, although a larger size would probably enable higher system efficiencies and simple program conversion to the 512 processor FMP system. The 360/195 non-split GISS version was used as the basic FMP benchmark model.

Simulation of code running on the FMP system is necessarily limited by time and cost. These requirements necessitate the separations of the GISS model into low and high use frequency classes in order to expedite the analysis. Routines of low frequency (once/run) and therefore considered of null impact were:

INPUT
GMP
SDET

Routines of high frequency and therefore maximum impact were:

```
     COMP1 - AVRX
     COMP2 - AVRS
     COMP3 - OZONE
           - SOLAR
           - LINKHO - SQRT
                    - EXP
```

AVRX is an extremely frequently used subroutine and presents an interesting opportunity for optimizing FMP performance. The function of AVRX is as follows. First, for every latitude J, compute a number NJ(J) (also called DRAT and FNM). Then, for each point J,I (I is the longitude index) perform a smoothing function S.

New PU(J,I) = S(Old PU(I,J-1), Old PU(I,J), Old PU(I,J+1))

over all values of I. Then update Old PU = New PU at all values of J, I. For any given latitude J, do the smoothing NM(J) times. NM(J) is a non-decreasing function of distance away from the equator, although this fact is not used in the original program. Several methods of converting this subroutine into FMP FORTRAN are discussed below.

1. A DOALL on J, with the programming over I and N serial inside. 89 out of 512 processors have instances, and the longest instances occur at the poles where NM has its maximum value.

2. An outer loop on N, iterating the number of times given by the maximum value of NM at the poles. Inside, a DOALL over both J and I allows all processors to execute on the first iteration of the N loop, but as the successive iterations of the N loop occur, those instances which test and find that N.GT.NM(J) exit without performing any work. At the end of the N loop, only those instances which lie at the poles are doing any work; the others are idle.

3. Like 2, except that when computing NM(J), the smallest J is computed (nearest the equator) for which the given value of NM occurred, giving JL(NM) as a GLOBAL array. The program structure would look like:

```
DOALL J=1, JMAX
NM(J) = arithmetic expression
JL1(NM) = smallest J in northern hemisphere for which NM has
          the value shown in the subscript
JL2(NM) = largest value of J in southern hemisphere for which
          NM has the value shown in the subscript
ENDDO
```

```
   DO 1 N= 1, NM(poles)          % N loop
   DOALL J=1, JL2(N); I=1,IMAX     % All points needing smoothing
                              % in the Southern hemisphere
   PU = S(Old PU values)
   ENDDO
 1 CONTINUE
```

Method 3 avoids the creation of instances that do no work, and hence enhances processor utilization. Even though the last few iterations on N have only 144 instances, since JL1 will equal JMAX for large N, and JL2 will equal 1 for large N, the average number of processor busy would be substantially better than that for either method 1 or method 2. The cost is increased overhead at the beginning of the DOALLs.

4. Method 2 can be modified as follows. First, the DOALL on I and J can be replaced by DOALL J=1, 89; II=1,109,36. Inside the DOALL, a loop, DO M=1,36 is added and the subscript I is set equal to II+M. The result is that 36 neighboring values of I are computed within a single processor, and the same old value of PU can be fetched once from EM for all three uses within the smoothing function. The result is a decrease in the number of required EM accesses by almost a factor of three, while processor utilization is reasonably good (356 processors out of 512, for the particular example).

5. With even more complexity in the management of the mapping between domain variables and I,J, one can have 36 values of I per instance, and keep 494 processors busy.

Time precluded simulating any more than one of the above options. Option 5 was selected for simulation, and produced the result shown in the table. One of the reasons for selecting option 5 is that the remapping of the values of I,J into particular processor might be done, not explicitly by the programmer as shown in example 4, but by having the compiler map particular instances into particular processors. In the prototype compiler, it is expected that the assignment of instance number to processor will be fixed at processor number equal to instance number modulo 512. Future compiler enhancements could include statements that allow the programmer to specify how instance numbers map onto processors. The simulation, to some extent, was an investigation of the value of such mappings.

After AVRX, the body of COMP1 and COMP2 are the next most frequently used. With minor exceptions, they have common coding characteristics. They were:

- Heavy use of Extended Memory
- Heavy use of three dimensional indexing
- Low number of floating operations/access

The initial section of COMP2 was judged to be typical and was therefore simulated on the instruction timing simulator.

COMP3 is executed once for every NCOMP3 executions of COMP1 and COMP2. The radiation routines, LINKHO, etc., are called every NHOGAN times that COMP3 is called once. Values of three, and five respectively were used for NCOMP3 and NHOGAN. In COMP3 and its subroutines, computations are carried on along the vertical direction, making each latitude-longitude point independent of any other. Thus COMP3 partitions into a set of independent instances, each having a specific location on the earth's surface. COMP3 and its subroutines are characterized by:

- Minimum use of Extended Memory
- Simple parallel partitioning
- High number of floating point operations
- Low number of indexing operations
- Data Dependent branching

The two maximum frequency inner loops of the LINKHO subroutine were judged typical of this code-section and simulated in detail.

The routines actually simulated during this analysis are summarized in Table A.5. They are:

* LINKHO (portions)
* COMP2 (portion)
* AVRX

A.5.3 FMP FORTRAN Version

Figures A.11, A.12 and A.13 repsectively show the FMP FORTRAN versions of AVRX and the portions of LINKHO and COMP2 simulated. Note that AVRX and COMP2 make substantial use of DOALL constructs. LINKHO does not demonstrate any DOALL constructs since it is called within each instance of sections of COMP3. LINKHO is an exceptionally good example of the data and instance-dependent computation in COMP3 which would execute efficiently on the system evaluated even through it would be difficult to vectorize. The aerodynamic flow codes analyzed did not exhibit the independence between instances to this degree. Substantial use is made of parts of the language that see little or no use in the two aero flow codes, including:

° Domain definitions constructed using domain expressions that
include previously defined domains. (See AVRX for example)
° INALL declarations (See AVRX for example)

Figure A.14 shows the branching structure of the subroutines. Note the presence of A**B, which is a form of call on the EXP function.

Table A.5

GISS WEATHER MODEL

BENCHMARK SIMULATION RESULTS

Measure	AVRX	Routine COMP2	LINKHO
Total no. of CU simulated instructions	48	32	30
Total no. of EU simulated instructions	3800	3094	2529
Total no. of EU machine clocks consumed	25318	23417	16705
Total no. of floating point register related instructions	338	900	1058
Total no. of floating point arithmetic operations	134	688	1266
Total no. of machine clocks for F. P. arithmetic operations	1039	7052	11624
Total no. of integer/logical instructions	2800	1449	425
Total no. of control type commands	662	745	1056
Average execution time for all instructions (NS)	266.5	302.7	264.2
Average execution time for floating point operations (NS)	310.2	410.0	367.3
Average total elapsed time for floating point operations	7557.6	1361.5	527.8

```
C      NOTE THAT THE CODE DEVELOPED BELOW IS MANUALLY MAPPED
C         TO THE HARDWARE BY STRUCTURING THE CODE
C         THE DOMAIN DEFINITIONS ARE USED TO ALLOCATE
C         WORK TO PROCESSORS AND TO CYCLES (INSTANCES)
C         WITHIN EACH PROCESSOR.
C
C
       SUBROUTINE AVRX
       STRUCTURE COMMON ....HERE ARE ALL ARRAYS IN BLANK COMMON...
       GLOBAL  COMMON .... HERE ARE SIMPLE VARIABLES IN BLANK COMMON..
      1      ,ALPH(16),DRAT(16)
       GLOBAL COMMON /WOTK/ PU(89,144)
       DOMAIN /PROC/; PNO=0,511
       DOMAIN /CYC/; INST=1,26
       DOMAIN/AVRXD/; /PROC/.X./CYC/
       INALL/PROC/ TPU(28),TTPU(28),II(28),JJ(28),EIM1,EI,EIP1
       STRUCTURE LOGICAL DONE(JMAX)=.FALSE.
C   CALCULATE DRAT(J),ALPH(J), AND NLIM(J). ONE VALUE PER LATITUDE
       DOALL J=2,JMAX-1
        TDRAT = DYP(2)/DXP(J)
        ALPH(J) = 0.125*(TDRAT-1)/FLOAT(FIX(TDRAT))
        DRAT(J) = TDRAT
        NLIM(J) = FIX(TDRAT)
       ENDDO
C   LOAD TPU WITH PU
       DOALL /PROC(PNO)/
        DO 1  M=1,26
C   NOTE THE INSTANCE NUMBER WHICH IS COMPUTED HERE
         INNO = 512*(M-1)+PNO
         II(M) = INNO/JMAX+1
         JJ(M) = MOD(INNO,JMAX) + 1
         IF (II(M) .GT. 144) EXIT
         TPU(M) = PU(JJ(M),II(M))
1       CONTINUE
        ENDDO/PROC/
C  START A DO WHILE
100    CONTINUE
```

Figure A.11 FMP FORTRAN Version of AVRX

```
      DOALL /PROC(PND)/
       EIM1 = TPU(1)
       EI = TPU(2)
       DO /CYC(INST)/
         I = II(INST)
         J = JJ(INST)
         IF (I .GT. IMAX) EXIT
         IF ((DRAT(J).LT.1) .OR. (N.GT.NLIM(J)) THEN
          DONE(J) = .TRUE.
          GO TO 2
         ENDIF
        EIP1 = TPU(INST+2)
        IF(I.EQ.0) EIM1=PU(J,IMAX)
        IF (I.EQ.IMAX)EIP1=PU(J,1)
        TTPU(INST) =  EI + ALPH(I)x(EIM1+EIP1-2.0xEI)
C   STORE CASES
        IF ((INST.EQ.1) .OR. (INST.EQ.26)) PU(J,I) = TTPU(INST)
        IF(I.EQ.1) PU(J,IMAXP1)=TTPU(INST)
        IF(I.EQ.IMAX) PU(J,0)=TTPU(INST)
        EIM1 = EI
        EI= EIP1
2      ENDDO /CYC/
C   SYNCH POINT
      NEXTDO
       DO INST = 2,26
        TPU(INST) = TTPU(INST)
       ENDDO
       DO INST=1,28,27
        TPU(INST) = PU(JJ(INST),II(INST))
        ENDDO
        DO INST = 1,28
          IF((I.EQ.0).OR.(I.EQ.IMAXP1))TPU(INST)=PU(JJ(INST),II(INST))
         ENDDO
         N=N+1
         DO M=1,2
          TPU(26+M) = PU(II(M),JJ(M))
         ENDDO
        ENDDO/PROC/
        DOALL J=1,JMAX
         IF (ALL(DONE(J))) RETURN
        ENDDO
         GO TO 100
```

Figure A.11 FMP FORTRAN Version of AVRX (Cont'd)

```
       SUBROUTINE LINKHO
       COMMON /RADCOM/PL(9),FLE(10),FLK(9),TG,TS,TL(9),TSTR(3)
     1      ,SHL(9),CLOUD(12),RE(10),RESTR(3),FLXDNG,SG,AS(9),ASSTR(3),
     2      SC,COSZ,RSURF,SCOSZ,RAP,RAM
       COMMON /CLDCOM/ SWALE(16),SWIL(15),AL(16),TAUL(16),OZALE(16),
     1      TOPABS
       LOGICAL CLDFLG,AERFLG,L1,L2
       REAL TAUCIR,CTAU55,X,PIO,TN,AER1,AER2,AERA,AERC,AERU,AERV,
     1      EX1,EX2,DENU,DNM0,DNMI,RERV,EXTAU,TAU,RDNCN,EDNCN,TDFCN,
     2      EUPCN,EDNCN
       INTEGER NCLOUD(12),NAERO(12)
       REAL CIREXT(12),TAUN(12,3),PICIR(12),PIZ(12,12),CB(12,12),
     1   BTOP(14),TDF(12),REF(12),EUP(12),EDN(12),TE3(301),EUPC(12),
     2   EDNC(12),TDFC(12),RDNC(12)
C
C     ADDITIONAL DECLARATIONS NOT USED IN THE SIMULATED PORTION
C     ARE OMITTED FOR BREVITY
C
C     STATEMENTS ABOUT PARALLELISM ARE OMITTED ALSO SINCE LINKHO
C     IS CALLED AS A SUBROUTINE WITHIN THE INSTANCES OF THE
C     DOALL /LAYERS/ OF COMP3.  IN THIS CASE, EACH INSTANCE
C     CALLS LINKHO INDEPENDENT FROM ALL OTHER INSTANCES AND
C     USES A LOCAL COPY OF CODE WITHIN THE PROCESSOR IN WHICH
C     THE INSTANCE RESIDES.  SEQUENCING OF THE EXECUTION WITHIN
C     THIS SUBROUTINE IS SOLELY DEPENDENT ON THE INSTANCE AND
C     LOCAL DATA, NOT ON ANY OTHER INSTANCES.
C
      DO 200 LAM = 1,12
       DO 100 K = 1,3
        DO 101 N = 1,NLAYRS
         NCC = NCLOUD(N)
         NAER = NAERO(N)
         TAUCIR = CIREXT(LAM) * CTAU55 * NCC
         X = TAUN(N,K) + TAUCIR
         TAUN(N,K) = X
         PIO =(TAUCIR*PICIRO(LAM) + PIZ(LAM,N))/(X+1.E-40)
         IF(N.GE.4) THEN
          TN = TL(N-3)/273.
          ELSE
          TN = TSTR(N)/273.
         ENDIF
         IF (TN.GE.0.65348 .AND. NCC.GT.0)PIO=0.
         IF(PIO.GT.1.E-4) THEN
          AER1 = 1. - PIO
          AER2 = 1. - (PIO*CB(LAM,N)
          AERA = SQRT(AER1/AER2)
```

Figure A.12 FMP FORTRAN Version of LINKHO

```
        AERU = (1. - AERA)/2.
        AERV = (1. + AERA)/2.
        AERC = SQRT(3.*AER1*AER2)
        X1 = -(AERC*X)
        EX1 = 0.0
        IF (X1 .GE. -180.218) EX1 = EXP(X1)
        IF (EX1.LT.1.0E-30)  EX1=0.0
        EX2 = EX1*EX1
        DEN0 = 1./((AERV*AERV) - (AERU*AERU*EX2))
        DNM0 = ((BTOP(N) - BTOP(N+1)/(X*AERC))*
     1               ((AERV - AERU*EX2) - (AERA*EX1))
        DNM1 = AERV + AERU*EX2
        EUP(N) = (BTOP(N)*DNM1 - DNM0 - BTOP(N+1)*EX1)*
     1               DEN0*AERA
        EDN(N) = (BTOP(N+1)*DNM1 + DNM0 - BTOP(N)*EX1)*
     1               DEN0*AERA
        REF(N) = AERU*AERV*(1.-EX2)*DEN0
        TDF(N) = (AERV-AERU)*DEN0*EX1
       ELSE IF (NCC.GT.0) THEN
        TDF(N) = 0.0
        REF(N) = 0.0
        EUP(N) = BTOP(N)
        EDN(N) = BTOP(N+1)
       ELSE IF ( X.LT.1.E-4) THEN
        TDF(N)=1.0
        REF(N) = 0.0
        EUP(N) = 0.0
        EDN(N) = 0.0
       ELSE
        IF (X .LE. 15.0) THEN
         EXTAU = EXP(-X)
         ITY =  X*20. + 1.
         TDF(N) = TE3(ITY) + (TY-ITY+1) * (TE3(ITY+1)-TE3(ITY))
        ELSE
         EXTAU = 0.0
         TDF(N) = 0.0
        ENDIF
        REF(N) = 0.0
        X1 = 1.0 - TDF(N)
        X2 = ((1.0 - EXTAU)/X-TDF(N)) * ((BTOP(N) - BTOP(N+1))*
     1             0.6666)
        EDN(N) = BTOP(N+1)*X1+X2
        EUP(N) = BTOP(N)*X1-X2
       ENDIF
       DEN0 = 1.0/(1.0 - RDNCN*REF(N))
       EDNCN = (EDNCN+EUP(N)*RDNCN) * TDF(N) * DEN0 + EDN(N)
```

Figure A.12 FMP FORTRAN Version of LINKHO (Cont'd)

```
      IF (NCC.GT.0) CLDFLG = .TRUE.
      IF(CLDFLG.AND.PID.GE.1.UE-4) AERFLG=.TRUE.
      IF (.NOT.(CLDFLG.OR.AERFLG)) THEN
        TAU = TAU + X
        IF (TAU .GT. 15) THEN
         TDFCN = 0.
        ELSE IF ((20.*TAU+1.).LT.1) THEN
          ITY=1
          TDFCN = TE3(ITY)+(TY-ITY+1)*(TE3(ITY+1)-TE3(ITY))
         ENDIF
        ENDIF
        IF (AERFLG) THEN
         RDNCN = REF(N) + TDF(N)*TDF(N)*RDNCN*DEN0
         TDFCN = TDFCN*TDF(N)*DEN0
       ENDIF
        IF(NCC.NE.0 .OR. FID.LT.1.0E-4) THEN
         TDFCN = 0.0
         RDNCN = 0.0
         TAU = 0.0
        ENDIF
        EUPC(N) = EUPCN
        EDNC(N) = EDNCN
        TDFC(N) = TDFCN
        RDNC(N) = RDNCN
101   CONTINUE
        DO 118 M = NG-1,1,-1
         DEND = 1.0/(1.0-RUPCN*REF(M))
         EUPCN = EUP(M) + ((EDN(M)*RUPCN+EUPCN) * TDF(M)*DENU
         IF (M.NE.1) THEN
          RUPCN = REF(M) + (TDF(M)*TDF(M)*RUPCN*DEN0)
          L = M-1
          DENU = 1./(1.-RDNC(L)*RUPCN)
          PEFUP = (EUPCN + EDNC(L)*RUPCN)*DENU
          PEFDN = (EUPCN + EDNC(L)*RDNC(L))*DEN0
         ELSE
          PEFUP = EUPCN
          PEFDN = 0.0
         ENDIF
         FE(M) = FE(M) + (PEFUP-PEFDN)*CLKAM
118     CONTINUE
100    CONTINUE
200   CONTINUE
```

Figure A.12 FMP FORTRAN Version of LINKHO (Cont'd)

```
C
C      THIS IS THE SECTION OF GISS , COMP2 THAT WAS SIMULATED
C
C
C      CORIOLIS FORCE
C
       DOALL J=2,JMM1;I=1,IM
        IF (I.EQ.1) THEN
         IM1=IM
        ELSE
         IM1 = I
        ENDIF
        FD(1,I) = 0.0
        FD(JM,I) = 0.0
        DO 100 L=1,NLAY
C
C       HERE THE COMMON SUBSCRIPT EXPRESSIONS ARE NOT GIVEN
C       BUT THE COMPILER IS ASSUMED TO HAVE EXTRACTED THEM APPROPRIATELY.
C
          FD(J,I) = F(J) + DXYP(J) + 0.25*(U(J,I,L)+U(J,I-1,L)+U(J+1,I,L)
     1          +  U(J+1,I-1,L))*(DXU(J)-DXU(J+1))
 100   CONTINUE
       ENDDO
       DOALL J=2,JM;I=I,IM
        IF (I.EQ.1) THEN
         IM1 = IM
        ELSE
         IM1 = I
        ENDIF
        DO 200 L = 1,NLAY
         FXCO = 0.125*DJ
         ALPH = FXCO * (P(J,I)+P(J-1,I))*(FD(J,I)*FD(J-1,I))
         UT(J,I,L) = UT(J,I,L) + ALPH*V(J,I,L)
         UT(J,IM1,L) = UT(J,IM1,L) + ALPH*V(J,IM1,L)
         VT(J,I,L) = VT(J,I,L) - ALPH*U(J,I,L)
         VT(J,IM1,L) = VT(J,IM1,L) - ALPH*U(J,IM1,L)
 200    CONTINUE
       ENDDO
C
```

Figure A.13 FMP FORTRAN Version of Part of COMP2

```
C      VERTICAL ADVECTION OF THERMODYNAMIC ENERGY
C
       DOALL J=1,JM;I=1,IM
        DO 300 L=1,NLAYM1
         LPA = P(J,I)
         LSIGA = SIG(L)
         LSIGB = SIG(L+1)
         PK1 = EXPBYK(PTROP + LSIGA*LPA)
         PK2 = EXPBYK(PTROP + LSIGB*LPA)
         LDSIGA = DSIG(L)
         LDSIGB = DSIG(L+1)
         LTA = T(J,I,L)
         LTB = T(J,I,L+1)
         LSDA = SD(J,I,L)
         LPITA = PIT(J,I)
         CU1 = LDSIGB/(LDSIGA+LDSIGB)
         CU2 = 1.0 - CU1
         TETAM = CU1*LTA/PK1 + CU2*LTB/PK2
         LTTA = TT(J,I,L) + DT*(LSIGA*KAPA*LPA*LTA*LPITA/PL1-
     1             LSDA*TETAM*PK1/LDSIGA)
         LTTB =TT(J,K,L+1) + DT*LSDA*TETAM*PK2/LDSIGA
         IF (LP1.EQ.NLAY) LTTB=LTTB + DT*LSIGB*KAPA*LPA*LTB*
     1           LPITA/PL2
         TT(J,I,L) = LTTA
         TT(J,I,L+1) = LTTB
  300   CONTINUE
       ENDDO
C
C    COMP2 CONTINUES BEYOND HERE, THIS IS THE END OF THE PIECE SIMULATED
C
```

Figure A.13 FMP FORTRAN Version of part of COMP2 (Cont'd)

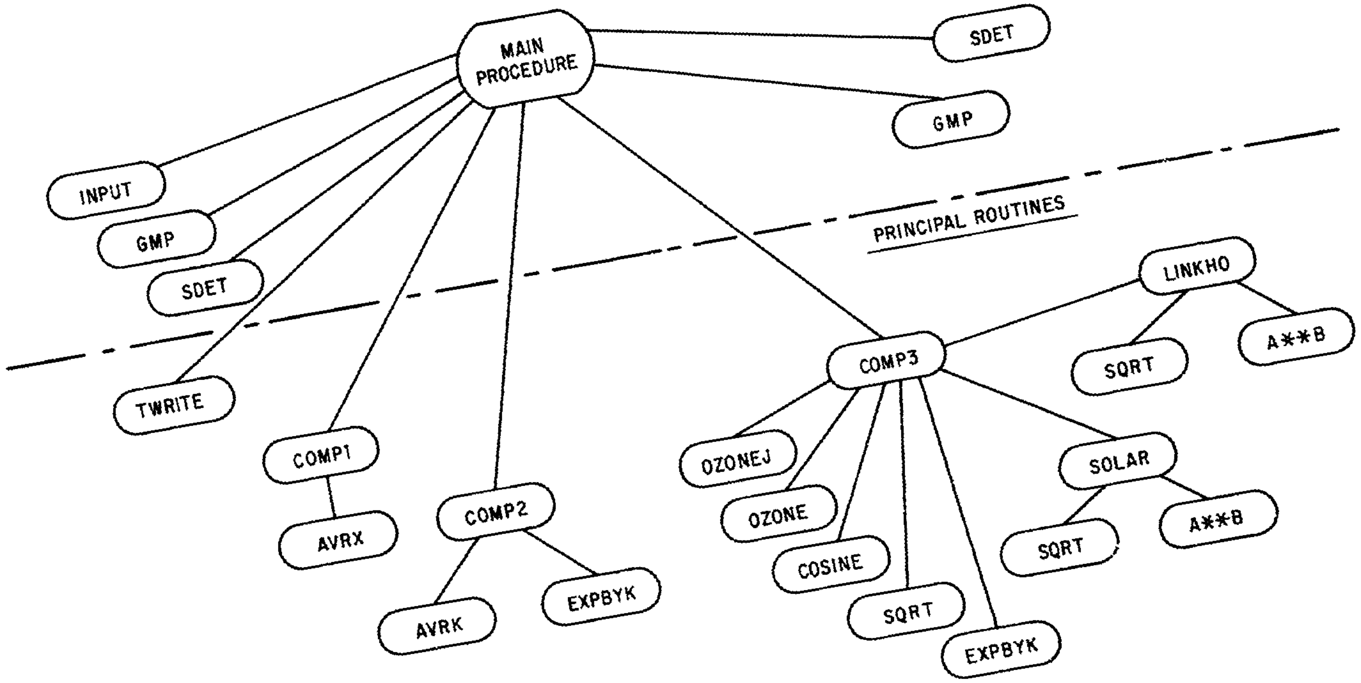


Figure A.14 Calling Tree of GISS Weather Code

A.5.4 Results

Table A.6 shows some of the assumptions and summarizes the results of the analysis. Table A.7 shows a more detailed breakdown of the analysis by subroutine.

This is a worst-case analysis, in that the data dependent branches were assumed to demand the most computations. This was done in order to estimate the worst-case maximum running time of the GISS weather code. Those weather conditions that result in faster running, such as clouds that reduce the amount of radiation computation needed, will result in a faster total run for the whole program. They also result in fewer computations actually performed.

An interesting detail of the analysis concerns the assignment of instances to processors. In the prototype compiler, instance number is computed as described in the section on FMP FORTRAN, and instance number i is processed in processor number (i mod 521). That is, at the beginning of the DOALL, processors 0 through 511 are given instance numbers 0 through 511 to do, and then each processor increments instance number by 512 to find its next instance to do, until all instances in the DOALL are exhausted.

In COMP3, a major contribution to whether a given instance will run for a long time or a short time is the condition of night vs. day. Radiation computations are much simpler on the dark side of the earth. At the equinox, computations would be for daytime along 72 meridans, and for nighttime along 72 meridians. As the DOALLs are arranged, with latitude subscript J first, all processors do daylight instances together, and all processors do nighttime instances together. This argument is somewhat oversimplified, because of dawn and twilight effects, and must be modified for other seasons where all points around one pole are in daylight, and the other are in darkness. However, more detailed analysis still confirms that, for the GISS weather, the straightforward assignment of instance number to processor number results in nearly equal distribution of not only daytime and nighttime within each individual processor but also latitude, thereby helping to distribute the computational effort evenly among all processors, and tending to make them finish nearly all together at the end of COMP3.

A.6 SPECTRAL WEATHER

A.6.1 Summary

The spectral weather is expected to run with substantially higher throughput than the GISS weather does. Its fluid dynamics portions are done by spectral analysis, with each processor processing an FFT independently of all other processors. (For a discussion of the case that only one FFT is to be executed in the FMP, see Section A.7.) Thus, the fluid dynamics computations are much more locally contained, since all the intermediate results in

TABLE A.6

Inputs Parameters				
	Grid Size	=	89 x 144 x 9	
	Time Step	=	20 minutes	
	Total Time	-	14 days	
	Total Time Steps	=	1008	
	NCYCLE	=	6	
	NHOGAN	=	5	Radiation call frequency
	NCOMP3	=	3	Physics call frequency
Output Result Totals				
	Flops/EU	=	2.88×10^8	
	Max Time/EU	=	4.42 minutes*	
	Flops/Systems	=	1.41×10^{11}	
	Gflops/System	=	0.532	

* Does not include system startup time

Table A.7

Performance Analysis of GISS Weather Model

Time step = 20 min.; total time = 14 day run
Total time steps = 1008

	Calls/ Time Step	Total Calls	Cycles/ EU	Executions/ EU	Flops/ Call	Flops/ EU	Time/ Flop(NS)	Time/ Module (MS)	% EU Utilization	Total Flops	Through- put
Main Program*	1	1008	1	1008	0	0	-	-	-	-	-
Input*	.001	1	-	-	-	-	-	-	-	-	-
Compl*	2	2016	26	52416	640	3.35E07	1362	4.57E04	95.0	1.63E10	0.356
AVRX	2	2016	1	2016	134	2.70E05	7556	2.04E03	96.2	1.33E08	0.065
Comp2	2	2016	26	52416	1944	1.02E08	1362	1.39E05	95.0	4.46E10	0.321
EXPBYK	64	64512	26	1.68E06	16	2.69E07	367	9.86E03	96.2	1.32E10	1.339
AVRX	4	4032	1	4032	134	5.40E05	7556	4.08E03	96.2	2.66E08	0.065
Comp3	1/3	336	26	8736	1474	1.29E07	520	6.70E03	90.0	5.92E09	0.884
OzoneJ	1/3	336	26	8736	95	8.30E05	520	4.32E02	96.2	4.0E08	0.926
Ozone	1/15	67	26	1747	60	1.05E05	520	5.45E01	96.2	5.16E07	0.947
Cos	2/3	672	26	17472	16	2.80E05	367	1.03E03	90.0	1.29E08	1.252
SQRT	4/3	1344	26	34944	16	5.59E05	367	2.05E02	96.2	2.75E08	1.341
EXPBYK	3.3	1008	26	26208	16	4.19E05	367	1.54E02	96.2	2.06E08	1.338
SOLAR	1/15	67	26	1747	26796	4.68E07	520	2.43E04	96.2	2.31E10	0.951
SQRT	1/15	67	26	1747	16	2.80E04	367	1.03E01	96.2	1.38E07	1.340
A**B	7/15	470	26	12230	16	1.96E05	367	7.18E01	96.2	9.66E07	1.345
Linkho	1/15	67	26	1747	25780	6.25E07	520	3.25E04	96.2	3.08E10	0.948
SQRT	4/15	268	26	6968	16	1.11E05	367	4.09E01	96.2	5.49E07	1.342
A**B	6/15	403	26	10478	16	1.68E05	367	6.15E01	96.2	8.27E07	1.345
GMP*	1/72	15	-	-	-	-	-	-	-	-	
SDET*	1/72	15	-	-	-	-	-	-	-	-	
TWRITE*	1/72	15	-	-	-	-	-	-	-	-	
TOTALS						2.88E08		2.65E05 MS (4.42 min)		1.41E11	0.532

Avg. time/flop = 2.65E08/2.88E08 = 0.92 μs/Flop
EU - Flops/sec = 1 x 106/0.92 = 1.087E06 flops/EU
System Flops/sec = 1.41E11/2.65E2 = 5.32E08

*Considered Negligible effects

the FFT can be contained within processor memory. The chemistry and physics portions of the spectral weather code are substantially identical to those of the GISS weather code, and the analysis of one can serve as the analysis of the other.

Therefore, the fluid dynamics portions of the spectral weather code are expected to run somewhat better than the fluid dynamics portions of the GISS; the chemistry and physics portions would have the same throughput exactly. (This ignores the effects on throughput of individual programming style, assuming that the spectral weather style is no better nor worse than the style seen in GISS.)

The FMP FORTRAN of Figure A.15 shows the essential portions of subroutines GDSPC1 and FFTFOR. It is clearly efficient. The inner loop in particular has only singly indexed local variables and a substantial proportion of multiply-add operations. A characteristic of all these loops is a short string of integer operations after the index test and before any floating point operations start. A characteristic of all these loops is a short string of integer operations after the index test and before any floating point operations start. The slowdown due to integer indexing that appeared in BTRI does not appear here except in DO 317 loop in GDSPC1 (see Fig. A.15) which is done N times, whereas the inner loop is done N*LOG N times.

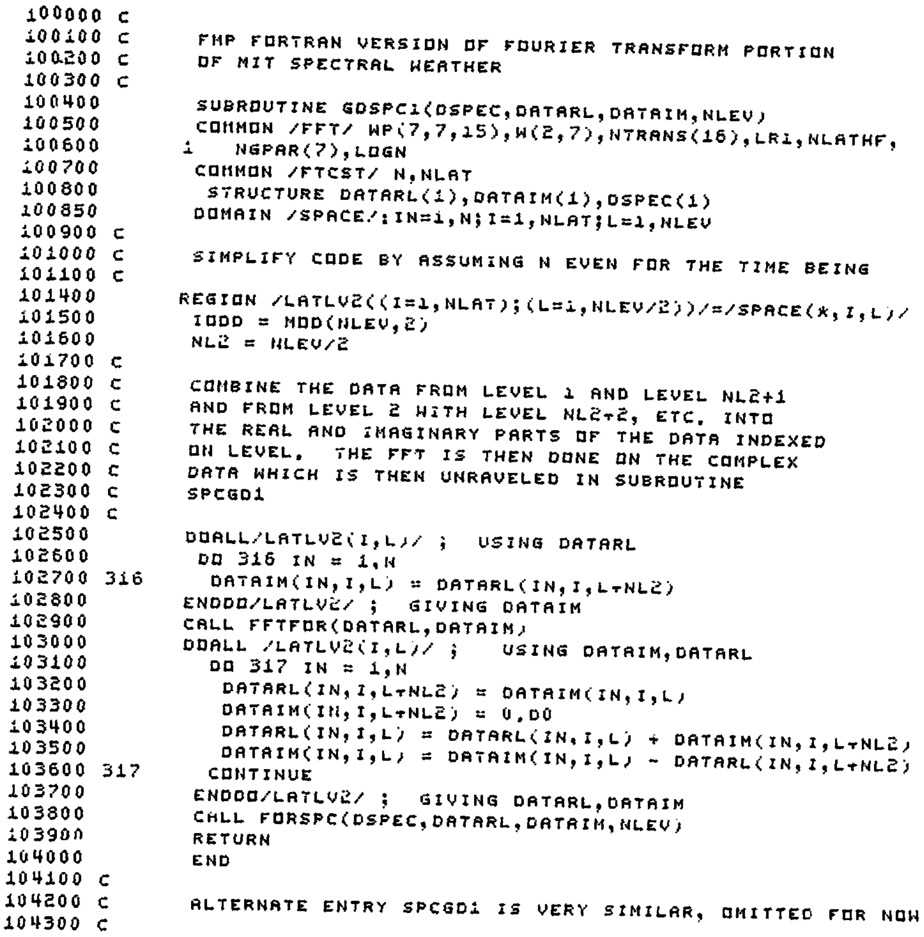
All local variables, including the local arrays, are substantially less than the 4096 words of address space that is accessible relative to the stack pointer, and which is reserved to subroutine local use. Such examples support the decision to have relatively short address fields within the instruction.

An estimate of 0.6 Gflops/sec was made for the spectral weather code. This estimate is not yet based on the detailed analysis performed on the other codes. The estimate is based on prior knowledge of the chemistry and physics portions of the GISS weather and an initial evaluation of the efficiency of execution of the FFT portion as described above.

A.6.2 FMP FORTRAN Version of FFT Portion of Program

In the MIT spectral weather code, the FFT appears as subroutine GDSPC1 which calls on subroutine FFTFOR. GDSPC1 takes NN arrays of data, splits each array's odd and even parts symmetrically about the center index, and rearranges the odd and even parts into real and imaginary parts of an array of complex values. FFTFOR then performs NN fast Fourier transforms on the NN ›mplex arrays, producing NN transforms.

NN is equal to the number of layers times the number of meridians. In a model with 144 meridians, 89 latitudes, and 12 layers, NN = 12 x 144 = 1728 transforms would be performed at once.



```
C
C     FMP FORTRAN VERSION OF FOURIER TRANSFORM PORTION
C     OF MIT SPECTRAL WEATHER
C
      SUBROUTINE GDSPC1(DSPEC,DATARL,DATAIM,NLEV)
      COMMON /FFT/ WP(7,7,15),W(2,7),NTRANS(16),LR1,NLATHF,
     1   NGPAR(7),LOGN
      COMMON /FTCST/ N,NLAT
       STRUCTURE DATARL(1),DATAIM(1),DSPEC(1)
      DOMAIN /SPACE/:IN=1,N;I=1,NLAT;L=1,NLEV
C
C     SIMPLIFY CODE BY ASSUMING N EVEN FOR THE TIME BEING
C
     REGION /LATLV2((I=1,NLAT);(L=1,NLEV/2))/=/SPACE(*,I,L)/
      IODD = MOD(NLEV,2)
      NL2 = NLEV/2
C
C     COMBINE THE DATA FROM LEVEL 1 AND LEVEL NL2+1
C     AND FROM LEVEL 2 WITH LEVEL NL2+2, ETC. INTO
C     THE REAL AND IMAGINARY PARTS OF THE DATA INDEXED
C     ON LEVEL.  THE FFT IS THEN DONE ON THE COMPLEX
C     DATA WHICH IS THEN UNRAVELED IN SUBROUTINE
C     SPCGD1
C
      DOALL/LATLV2(I,L)/ ;  USING DATARL
       DO 316 IN = 1,N
316     DATAIM(IN,I,L) = DATARL(IN,I,L+NL2)
      ENDDO/LATLV2/ ;  GIVING DATAIM
      CALL FFTFOR(DATARL,DATAIM)
      DOALL /LATLV2(I,L)/ ;   USING DATAIM,DATARL
        DO 317 IN = 1,N
         DATARL(IN,I,L+NL2) = DATAIM(IN,I,L)
         DATAIM(IN,I,L+NL2) = 0.D0
         DATARL(IN,I,L) = DATARL(IN,I,L) + DATAIM(IN,I,L+NL2)
         DATAIM(IN,I,L) = DATAIM(IN,I,L) - DATARL(IN,I,L+NL2)
317     CONTINUE
       ENDDO/LATLV2/ ;  GIVING DATARL,DATAIM
       CALL FORSPC(DSPEC,DATARL,DATAIM,NLEV)
       RETURN
       END
C
C     ALTERNATE ENTRY SPCGD1 IS VERY SIMILAR, OMITTED FOR NOW
C
```

Figure A.15 FMP FORTRAN Version of GDSPC1 and FFTFOR

```
      SUBROUTINE FFTFOR(DATARL,DATAIM)
      STRUCTURE DATARL(1), DATAIM(1)
      DOUBLE PRECISION WP,W,W2
      COMMON /FFT/ WP(7,7,15),W(2,7),NTRANS(16),LR1,NLATHF,
     1       NGPAR(7),LOGN
      COMMON /FTCST/ N,NLAT
C
C    FOR BREVITY, THIS EXAMPLE IS SIMPLIFIED TO THE FORWARD
C    FFT ONLY, LEAVING THE REVERSE TRANSFORM TO BE ADDED LATER
C
     DOMAIN /SPACE/:IN=1,N;I=1,NLAT;L=1,NLEV
     DOMAIN /LATLV2/: I=1,NLAT; L=1,NL2
     INALL /LATLEV/ DTRL(15),DTIM(15)
     ISIGN = 1
     DOALL /LATLEV(I,L)/;  USING DATARL,DATAIM,/FFT/,/FTCST/
      DO 12 J=1,N
       DTRL(J) = DATARL(NTRANS(J),I,L)
       DTIM(J) = DATAIM(NTRANS(J),I,L)
       DTIM(NTRANS(J)) = DATAIM(J,I,L)
12     DTRL(NTRANS(J)) = DATARL(J,I,L)
C
      DO 15 J=1,N/2
       TEMPR = DTRL(2*J-1) + DTRL(2*J)
       TEMPI = DTIM(2*J-1) + DTIM(2*J)
       DTRL(2*J) = DTRL(2*J-1) - DTRL(2*J)
       DTIM(2*J) = DTIM(2*J-1) - DTIM(2*J)
       DTIM(2*J-1) = TEMPI
15     DTRL(2*J-1) = TEMPR
C
      DO 90 II = 2,LOGN
       NUM = 2**II
       NUMHF = NUM/2
       NSS = 1*2**(LOGN-II)
C
C      THE ABOVE EQUALS N/NUM,AS SHOWN IN THE ORIGINAL PROGRAM,
C      BUT POWERS OF 2 ARE NUMERIC SHIFTS, MUCH FASTER THAN A
C      DIVIDE
C
       DO 90 J=1,NSS
        NUMJK = NUM*(J-1)
        LL = 1+NUMJK
        MM = LL+NUMHF
C
C       NOTICE THE DELETION FROM THESE VARIABLES OF OFFSETS
C       CORRESPONDING TO THE DOMAIN VARIABLES WHICH APPEARED
C       IN THE ORIGINAL
```

Figure A.15 FMP FORTRAN Version of GDSPC1 and FFTFOR (Cont'd)

```
C
         TEMPR = DTRL(LL) + DTRL(MM)
         TEMPI = DTIM(LL) + DTIM(MM)
         DTRL(MM) = DTRL(LL) - DTRL(MM)
         DTIM(MM) = DTIM(LL) - DTIM(MM)
         DTRL(LL) = TEMPR
         DTIM(LL) = TIMPI
C
         DO 90 K=2,NUMHF
          LL = K+NUMJK
          MM = LL + NUMHF
          MMM = NSS*(K-1)
          W2 = -W(2,MMM)
C
C         NOTE THAT THE ABOVE WOULD BE CONDITIONAL SIGN IF REVERSE FFT
C
          CROSSR = DTRL(MM) * W(1,MMM) + DATAIM(MM)*W2
          CROSSI = DTIM(MM) * W(1,MMM) - DATARL(MM)*W2
          DTRL(MM) = DTRL(LL) - CROSSR
          DTIM(MM) = DTIM(LL) - CROSSI
          DTRL(LL) = DTRL(LL) + CROSSR
          DTIM(LL) = DTIM(LL) + CROSSI
90        CONTINUE
      DO 100 II=1,N
C
C        NORMALIZE AND PUT BACK IN STRUCTURE VARIABLES
C        DIVIDE BY 2**LOGN IS A SUBTRACT FROM EXPONENT,
C         RUNS MUCH FASTER THAN DIVIDE BY N
C
       DATARL(II,I,L) = DTRL(II)/2**LOGN
        DATAIM(II,I,L) = DTIM(II) / 2**LOGN
100    CONTINUE
       RETURN
      ENDDO/LATLEV/;  GIVING DATARL,DATAIM
       END
+
```

Figure A.15   FMP FORTRAN Version of GDSPC1 and FFTFOR (Cont'd)

The obvious, and simple, strategy is to have a DOALL on layers and longitudes, with each instance performing a serial transform. That is:

```
DOALL I=1,144; L=1,NLEV
... here the code for a serial fast Fourier transform
ENDDO
```

One of the optimizations in the original program needs to be undone in order to separate the loop into a large DOALL and a short DO loop. The original version took the multidimensional arrays that naturally appear in the problem and unwound them into one-dimensional arrays. Thus, a substantial amount of index computation was saved by doing the index calculations separately. In order to make best use of the FMP, the structure inherent in the problem needs to be retained.

The FMP FORTRAN version shown in Figure A.15 includes the conversion from space variables to complex function, the forward Fourier transform (complex) on the complex function, but omits the conversion from complex function back to real frequency functions, and also omits the reverse Fourier transform, since both of these are trivially different from the code that is exhibited.

The arrays DATARL and DATAIM in the original FORTRAN version are used both to hold the entire input and output files of the transform, and also to use as working space during the course of the transformation. In this FMP FORTRAN version, two STRUCTURE arrays DATARL and DATAIM are used to hold the entire input and output files before and after the transformations, but two LOCAL arrays DTRL and DTIM are used as the working space during the course of the transformation.

Each processor is doing one FFT serially. There are as many FFT's being executed as there are points around the equator times the number of levels. The code as exhibited therefore would be efficient only for grids somewhat finer than the 16 latitudes x 24 longitudes of the MIT code as submitted.

The following list gives, in sequence, the variables that are candidates for being assigned to registers. This list covers subroutine FFTFOR only.

INTEGER REGISTERS	FLOATING-POINT REGISTERS
Stack pointer	TEMPR
Base address DATARL	TEMPI
Base address DATAIM	W2
Cycle index for IN ALL	CROSSR
J	CROSSI
Base address of common/FTCST/	
N	
Processor no., for DOALL control	
I	
L	
N/2 (loop limit)	
2*J (common subexpression)	
2*J-1 (common subexpression)	
Base address of common FFT/	
II	
LOGN	
NUM	
NUMHF	
NSS	
NUMJK	
LL	
MM	
K	
MMM	
2**LOGN (common subexpression)	

In addition to the above list, some scratch and accumulator registers need to be assigned (some double length). As was observed in the analysis of the explicit code, more integer registers would be needed to avoid saving and restoring them. The number of floating point registers is adequate.

A.7 OTHER ANALYSIS

As an example of additional applications, this section will discuss two application areas that fall outside of the benchmark programs. The first section discusses how well the FMP would do on FFT's when only one FFT is being done instead of 512 FFT's operating efficiently in parallel as in the spectral weather.

A.7.1 Fast Fourier Transforms on the FMP

A.7.1.1 Discussion

This section makes some preliminary estimates of the throughput of the FMP executing a single FFT across the entire array. If data length is assumed to be a power of 2 and at least 512 long, the resulting throughput is estimated to lie between 0.6 and 1.0 Gflops/sec. The exact throughput figure is dependent on the algorithm selected for the FFT. This section is a discussion of the algorithms, and a description of how they operate. An FMP FORTRAN version of one of the algorithms is presented.

Algorithms which have the final result stored on "scrambled" indices were developed to allow in-place computation to save memory. The data interactions in these algorithms correspond to swapping data between the upper half and the lower half of some subset of the data, the subset being a power of 2. At the end, the scrambled data is stored in memory, the indices are bit-for-bit reversed (so that 0000011 becomes 1100000), and the reversed indices are then used to reorder the resulting data.

Other algorithms, such as Glassman's [4], require that the data interchange in the body of the algorithm be a perfect shuffle. There is no rearrangement required at the end.

For a 512-point FFT the computations would be fully parallel across the processors, and the swaps, shuffles, or rearrangements would take place on all data. For the 512-point case:

> For the "scrambled" algorithms, there are 9 swaps and 1 rearrangement.
>
> For the Glassman algorithm, there are 9 perfect shuffles.

For FFTs with more points than 512, the amount of data being swapped doubles, and the number of swaps goes as $\log_2(N)$, while the number of multiplications and additions is proportional to N $\log_2(N)$. There are exactly 1/2N $\log_2(N)$ complex multiplications in the Glassman algorithm, for taking the Fourier transform of a real variable (since the odd and even parts of the real function can be combined into the real and imaginary parts of a complex function defined over half as many points).

Thus, the time required for each of the following needs to be considered.

- Swapping N/512 items of data
- PERFECTSHUFFLING N/512 items of data
- REARRANGING N/512 items of data

The times for the above would then be inserted into a formula where SWAPping and SHUFFLing are multiplied by $\log_2 N$. As a first approximation, these times would be added to the time taken for computation to get the net time for an FFT. The result is that the "scrambled" versions of the FFT run substantially better on the FMP, since the SWAP is the SHIFTN operator, while SHUFFLing and REARRANGING are stores to EM followed by fetches from EM.

A.7.1.2 Timing Estimates

Within the inner loop of an FFT, all processors do the same computations, and hence will stay in synchronism. Any synchronizations required do not imply any significant time wasted waiting for the slowest processor.

A SWAP consists of:

N/512 SHIFTN instructions, at 12 clocks each.

A PERFECTSHUFFLE consists of:

N/512 STOREMs. Each STOREM occurs in its proper place within its own instance; not as a string of successive STOREMs. Hence processing can be concurrent with the write to EM.

NEXTDO (the splitting of a DOALL into two successive DOALLS) requires the termination of the instances, a synchronization, and the hidden cycle loop of the subsequent DOALL. Hence, the following code is executed in the processor,

```
IJUMP     % end of instances

WAIT      % processor side of the synchronization

IMOVEL    % cycle loop variable initialization

IMOVEL    % cycle loop limit

ITIX      % cycle loop
```

which has a total of 13 clocks (before correcting for overlap and instruction fetching).

N/512 LOADEMs. If the STOREMs are to EM modules with a skip distance of 2, the perfect shuffle has the LOADEMs at a skip distance of 1, which is one of the "magic" skip distances at which the CN has no conflicts.

The final formula for timing, in terms of number of clocks, using T_{EM} to indicate the number of clocks per EM access (include address computation) is:

$$T_{ps} = 2(N/512)T_{EM} + 13$$

A REARRANGING of the data on scrambled indices consists of N/512 STOREMs, all occurring in succession, followed by the 13 clocks of the NEXTDO, followed by N/512 LOADEMs. The STOREMs are in succession, so EM module busy will keep them at least 9 clocks apart, but the "EM busy" of the last STOREM can be hidden behind the 13 clocks implied by the NEXTDO.

In addition, the subscripting on scrambled indices, bit reversed, is the worst possible permutation for CN conflicts. It will take 16 times CN access time plus EM cycle (144 clocks) to get all 512 requests through. These additional 144 clocks are approximately the same whether the bit reversed scrambled indexing occurs on the STOREM or the LOADEM. A formula is thus

$$T_r = 2(N/512)(T_{EM} + 144) + 13$$

The time added to the entire FTT by these operations can now be computed. Remembering that there are $\log_2 N$ passes through the inner loop, time for the "scrambled indices" algorithm (after reducing the formula) is:

$$T = \log_2 N(N/512)*12 + 2(N/512)(T_{EM} = 144)$$

For the perfect-shuffle (Glassman) type, time is:

$$T = \log_2 N(N/512)*T_{EM} + 13 \log_2 N$$

These are the times spent in data rearrangement. In addition, there are (N logN)/1024 complex multiplications and additions per processor, or 4 N $\log_2 N$ floating point operations per processor. Simulation shows that real programs that are fairly well adapted to the FMP run at about 1.3 Gflops (AMATRX, BTRI). This is about 9.8 clocks per floating point operation. If the rest of the FFT does as well, the time spent in computation is 39.2 $\log_2 N(N/512)$ clocks.

Table A.8 shows these numbers, together with an estimated throughput rate (in Gflops) for the FFT assuming that there is no overlap and that otherwise all the above assumptions hold. An estimate of 35 clocks for T_{EM} was used to cover address computations, CN delay, and EM access time.

A.7.1.3 FMP FORTRAN Version of Glassman's FFT Algorithm

Figure A.16 is an example of a FFT coded for the FMP. The attached FFT is Glassman's algorithm, and does not scramble the indices.

The FMP FORTRAN in Figure A.16 is a rather direct translation of an existing ALGOL program (Figure A.17). In translating from the ALGOL to the FMP FORTRAN, it is likely that the result is not optimized for the FMP. Specifically, the perfect shuffle in this particular code consists of fetching the Z items on shuffled indices into an INALL array, then the NEXTDO for finishing all instances for data precedence, and then four successive STOREMs. Successive STOREMs are not overlapped as they would be if mixed in with computation.

Table A.8

Summary of FFT Throughput Estimates

Type of FFT		data-shuffling time (from formulas in text)	assumed computation time	total time	Gflops (approximate)
"scrambled"	N=512	466	257	723	0.474
	N=1024	956	514	1470	0.466
	N=2048	1470	1028	2498	0.549
	N=4096	2008	2056	4064	0.675
"Glassman"	N=512	431	257	688	0.498
	N=1024	830	514	1344	0.510
	N=2048	1298	1028	2326	0.589
	N=4096	1836	2056	3892	0.704

This program runs for any binary value of N whatsoever, but is efficient only for N equal to 512 or greater. The language is completely independent of the number of processors.

The ALGOL program of Figure A.17 is a free-standing program, which reads in a data deck and prints out the transform. It was written for demonstration purposes, to show that the Glassman algorithm had indeed been understood and programmed. For the FMP, it is assumed that some main program supplies the data and uses the results. Thus, all of the I/O and some of the initialization has been sloughed off onto this assumed main program and does not appear in subroutine GLASMN.

A.7.2 A Parallel Sort

Sorting is a common computer application. This section demonstrates an in-core sort that makes use of all processors at a reasonable processor utilization. Seldom are the items to be sorted simply numbers to be sorted by magnitude; however, this is the easiest example to use to show how the algorithm works. The algorithm starts from a state in which the items to be sorted are distributed uniformly among the processors. "Processor" could mean either processor local memory, or a piece of EM address space allocated to a specific processor. The algorithm will work for the number of processors (2^n) equal to any power of 2. The example will be given for a number of processors equal to eight. The starting condition for the example is given in Figure A.18.A The succeeding steps in the algorithm go as follows:

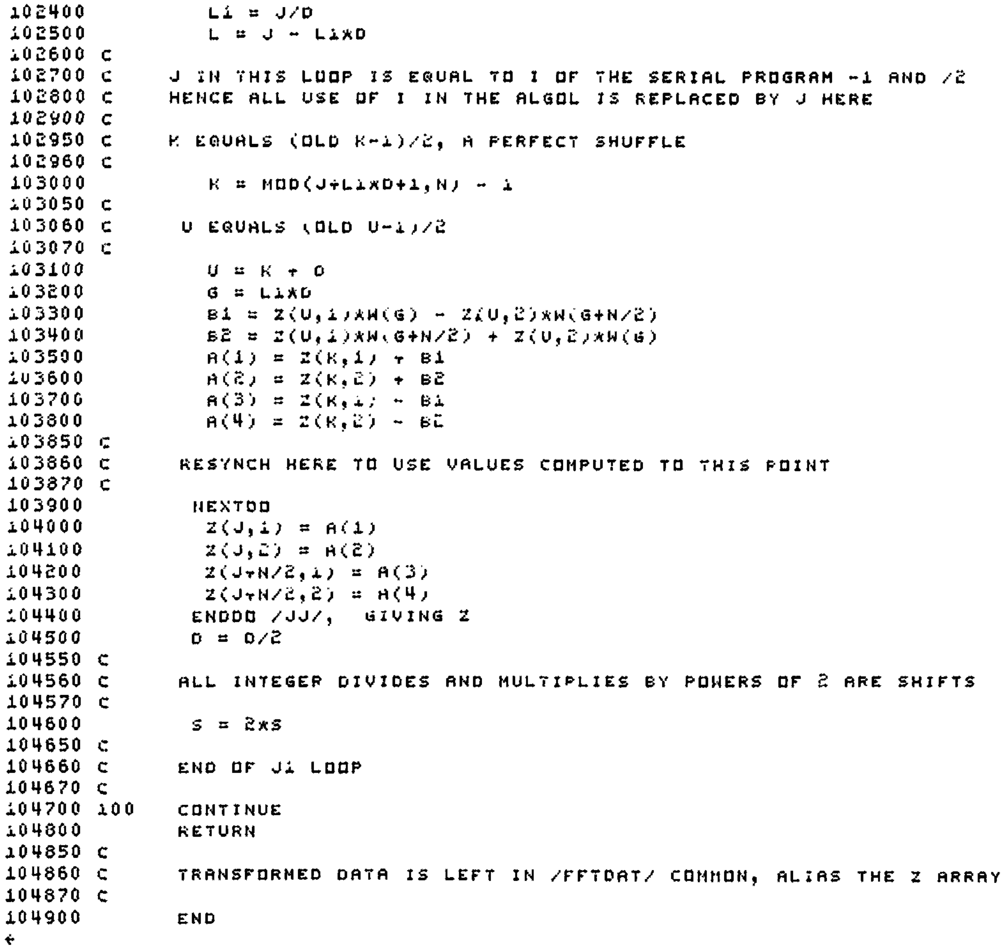
1. Sort the items local to each processor, yielding the state of Figure A.18.B.

2. Determine the median value globally. One method for doing this is to guess at a median, and then count how many items are greater or less than this guess. The total count is given by means of a SUMALL function on the individual processor counts. If this guess is not close enough to the median, one makes a new guess, and finds a new count. This procedure iterates until a value close enough to the median is found. Each processor divides its pile of sorted items intc two parts, one larger than the median, one smaller than the median. This division is marked in Figure A.18.B.

   Swap parts between processors that are 2^m $(m = n-1)$ apart. The lower numbered processor of each swapping pair sends the higher of its two parts to the higher numbered processor, and the higher numbered processor sends the lower of its two parts to the lower numbered processor. After the swap the contents of the various processors are like Figure A.18.C.

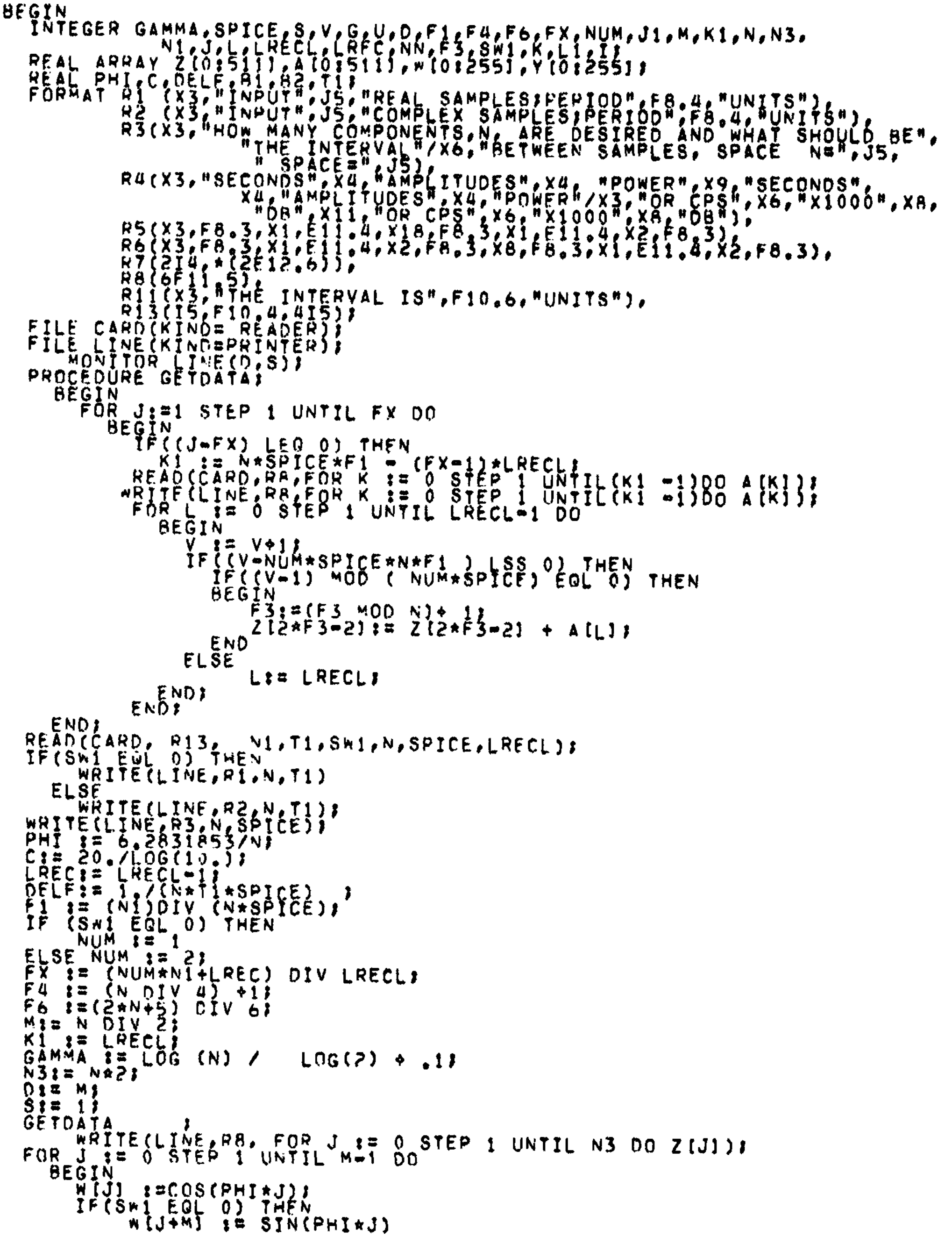
```
         SUBROUTINE GLASMN (N,SW,LOG2N)
C      ASSUME DYNAMIC ARRAY DECLARATIONS
            COMMON /FFTDAT/Z(N,2)
            LOGICAL SW
           INTEGER N, LOG2N
           REAL PHI,D,S
           DOMAIN /KK/;K=0,N-1
           INALL/KK/W
           DOMAIN /JJ/; J=0,N/2-1
           INALL /J/ A(4)
            INTEGER I, K, U, G
C
C        THE FOLLOWING FORM WAS CHOSEN TO TAKE ADVANTAGE OF THE FADEX
C        COMMAND WHICH IS MUCH FASTER THAN FDIV FOR DIVISION OF
C        INTEGER POWER OF 2
C
          PHI = 6.2831853072/2**LOG2N
C
C         THIS DO LOOP INITIALIZES THE TWIDDLE FACTORS
C
          DOALL J=0,N-1;  USING PHI,W,SW
           IF (J.LT.N/2) THEN
C
C         COSINES IN LOWER HALF OF W ARRAY
C
            W(J) = COS(PHI*J)
            ELSE IF (SW) THEN
             W(J) = SIN(PHI*(J-N/2))
             ELSE
               W(J) = -SIN(PHI*(J-N/2))
           ENDIF
          ENDDO;  GIVING W
C
C         INITIALIZATION, IN WHAT FOLLOWS D*S ALWAYS EQUALS N/2
C
          D = N/2
          S = 1
          DO 100 J1 = 1,LOG2N
           DOALL/JJ(J)/;  USING Z,W,D
C
C         THE DOMAIN IS DIVIDED INTO S BLOCKS OF D ELEMENTS EACH
C         S AND D ARE BOTH POWERS OF 2
C
```

Figure A.16 FMP FORTRAN Version of Glassman's FFT Algorithm



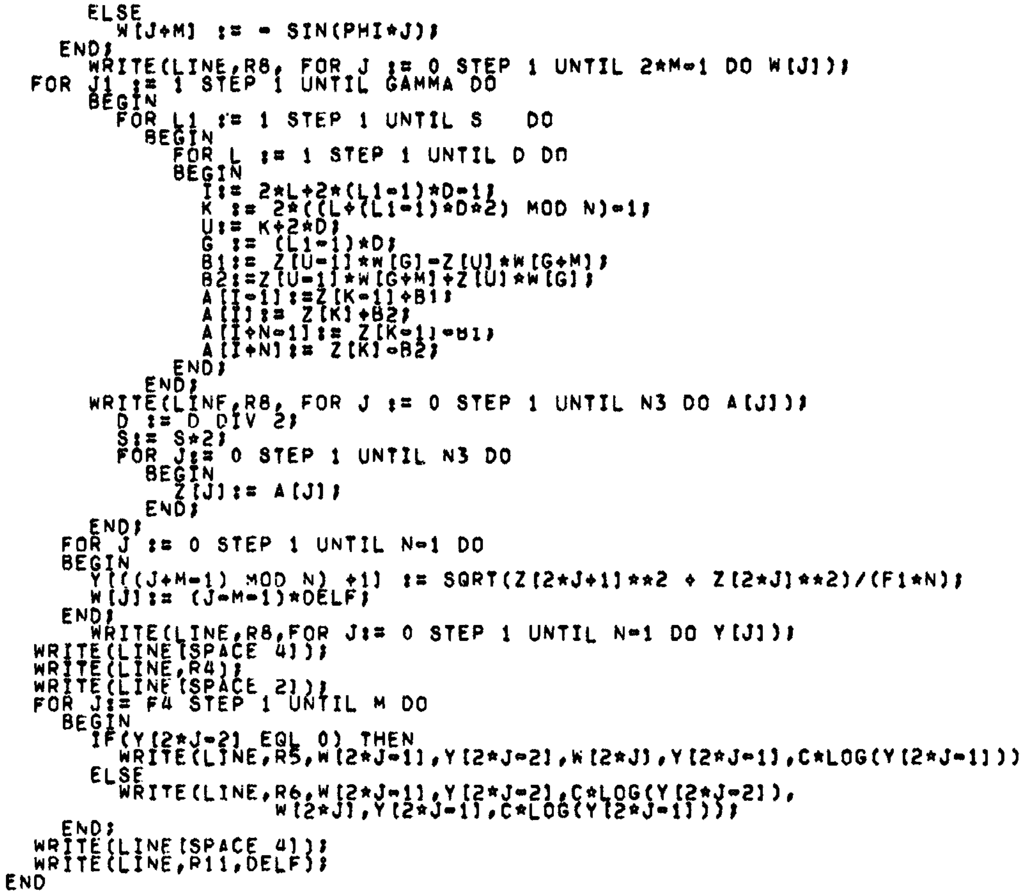
```
          L1 = J/D
          L = J - L1*D
C
C     J IN THIS LOOP IS EQUAL TO I OF THE SERIAL PROGRAM -1 AND /2
C     HENCE ALL USE OF I IN THE ALGOL IS REPLACED BY J HERE
C
C     K EQUALS (OLD K-1)/2, A PERFECT SHUFFLE
C
          K = MOD(J+L1*D+1,N) - 1
C
C      U EQUALS (OLD U-1)/2
C
          U = K + O
          G = L1*D
          B1 = Z(U,1)*W(G) - Z(U,2)*W(G+N/2)
          B2 = Z(U,1)*W(G+N/2) + Z(U,2)*W(G)
          A(1) = Z(K,1) + B1
          A(2) = Z(K,2) + B2
          A(3) = Z(K,1) - B1
          A(4) = Z(K,2) - B2
C
C      RESYNCH HERE TO USE VALUES COMPUTED TO THIS POINT
C
        NEXTDO
         Z(J,1) = A(1)
         Z(J,2) = A(2)
         Z(J+N/2,1) = A(3)
         Z(J+N/2,2) = A(4)
        ENDDO /JJ/,  GIVING Z
        D = D/2
C
C      ALL INTEGER DIVIDES AND MULTIPLIES BY POWERS OF 2 ARE SHIFTS
C
        S = 2*S
C
C      END OF J1 LOOP
C
  100  CONTINUE
       RETURN
C
C      TRANSFORMED DATA IS LEFT IN /FFTDAT/ COMMON, ALIAS THE Z ARRAY
C
       END
```

Figure A.16 FMP FORTRAN Version of Glassman's FFT Algorithm (Cont'd)



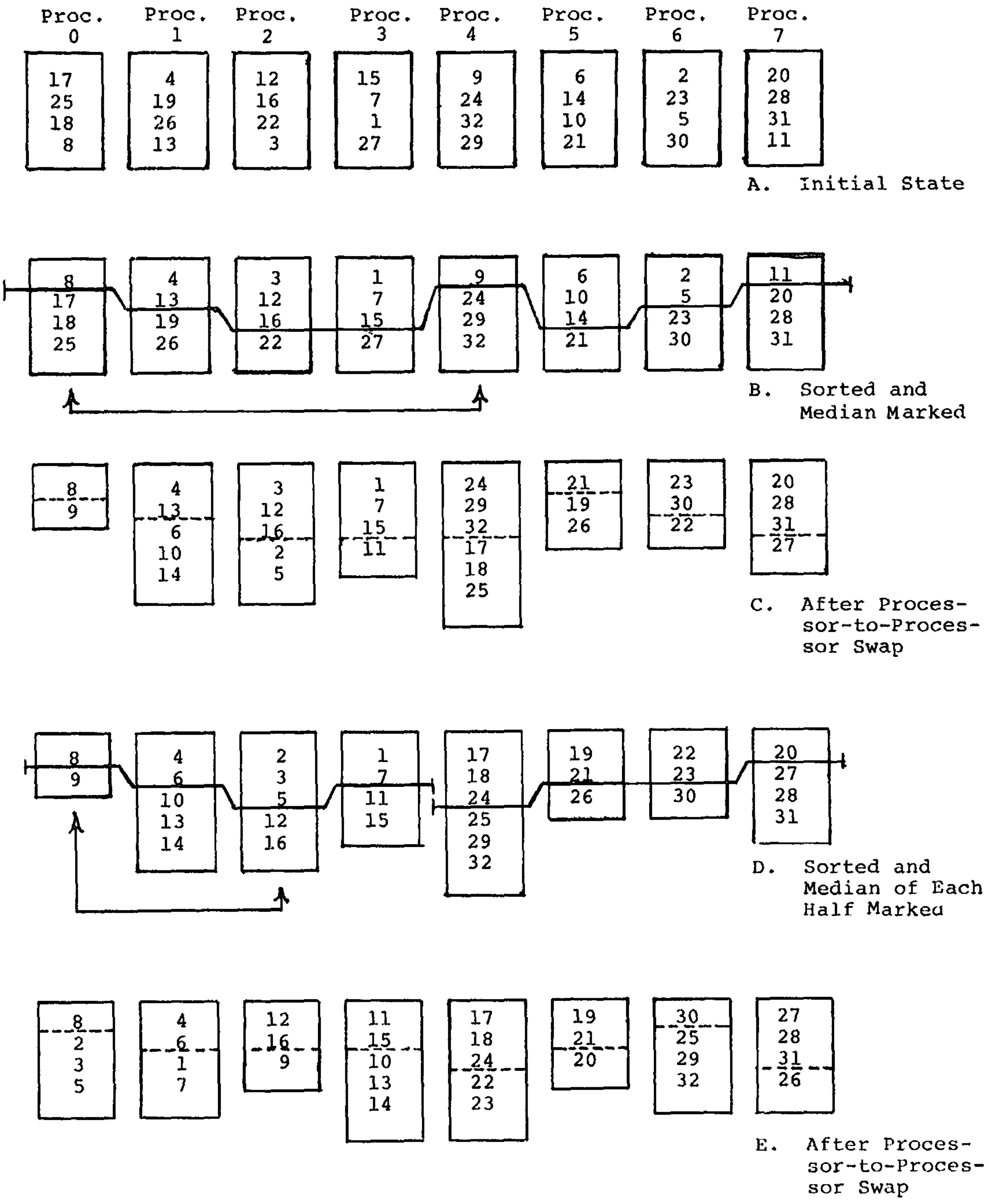
```
BEGIN
  INTEGER GAMMA,SPICE,S,V,G,U,D,F1,F4,F6,FX,NUM,J1,M,K1,N,N3,
            N1,J,L,LRECL,LRFC,NN,F3,SW1,K,L1,I;
  REAL ARRAY Z[0:511],A[0:511],W[0:255],Y[0:255];
  REAL PHI,C,DELF,R1,R2,T1;
  FORMAT R1 (X3,"INPUT",J5,"REAL SAMPLES;PERIOD",F8.4,"UNITS"),
         R2 (X3,"INPUT",J5,"COMPLEX SAMPLES;PERIOD",F8.4,"UNITS"),
         R3(X3,"HOW MANY COMPONENTS,N, ARE DESIRED AND WHAT SHOULD BE",
                  "THE INTERVAL"/X6,"BETWEEN SAMPLES, SPACE  N=",J5,
                   " SPACE=",J5),
         R4(X3,"SECONDS",X4,"AMPLITUDES",X4, "POWER",X9,"SECONDS",
                  X4,"AMPLITUDES",X4,"POWER"/X3,"OR CPS",X6,"X1000",X8,
                   "DB",X11,"OR CPS",X6,"X1000",X8,"DB"),
         R5(X3,F8.3,X1,E11.4,X18,F8.3,X1,E11.4,X2,F8.3),
         R6(X3,F8.3,X1,E11.4,X2,F8.3,X8,F8.3,X1,E11.4,X2,F8.3),
         R7(2I4,*(2E12.6)),
         R8(6F11.5),
         R11(X3,"THE INTERVAL IS",F10.6,"UNITS"),
         R13(I5,F10.4,4I5);
  FILE CARD(KIND= READER);
  FILE LINE(KIND=PRINTER);
     MONITOR LINE(D,S);
  PROCEDURE GETDATA;
    BEGIN
      FOR J:=1 STEP 1 UNTIL FX DO
        BEGIN
          IF((J-FX) LEQ 0) THEN
            K1 := N*SPICE*F1 - (FX-1)*LRECL;
          READ(CARD,R8,FOR K := 0 STEP 1 UNTIL(K1 -1)DO A[K]);
         WRITE(LINE,R8,FOR K := 0 STEP 1 UNTIL(K1 -1)DO A[K]);
          FOR L := 0 STEP 1 UNTIL LRECL-1 DO
            BEGIN
              V := V+1;
              IF((V-NUM*SPICE*N*F1 ) LSS 0) THEN
                IF((V-1) MOD ( NUM*SPICE) EQL 0) THEN
                BEGIN
                   F3:=(F3 MOD N)+ 1;
                   Z[2*F3-2]:= Z[2*F3-2] + A[L];
                END
              ELSE
                   L:= LRECL;
            END;
          END;
    END;
  READ(CARD, R13,  N1,T1,SW1,N,SPICE,LRECL);
  IF(SW1 EWL 0) THEN
      WRITE(LINE,R1,N,T1)
    ELSE
      WRITE(LINE,R2,N,T1);
  WRITE(LINE,R3,N,SPICE);
  PHI := 6.2831853/N;
  C:= 20./LOG(10.);
  LREC:= LRECL-1;
  DELF:= 1./(N*T1*SPICE)  ;
  F1 := (N1)DIV (N*SPICE);
  IF (SW1 EQL 0) THEN
      NUM := 1
  ELSE NUM := 2;
  FX := (NUM*N1+LREC) DIV LRECL;
  F4 := (N DIV 4) +1;
  F6 :=(2*N+5) DIV 6;
  M:= N DIV 2;
  K1 := LRECL;
  GAMMA := LOG (N) /   LOG(2) + .1;
  N3:= N*2;
  D:= M;
  S:= 1;
  GETDATA       ;
      WRITE(LINE,R8, FOR J := 0 STEP 1 UNTIL N3 DO Z[J]);
  FOR J := 0 STEP 1 UNTIL M-1 DO
    BEGIN
      W[J] :=COS(PHI*J);
      IF(SW1 EQL 0) THEN
          W[J+M] := SIN(PHI*J)
```

Figure A.17 ALGOL Version of Glassman's FFT Algorithm



```
   ELSE
    W[J+M] := - SIN(PHI*J);
  END;
   WRITE(LINE,R8, FOR J := 0 STEP 1 UNTIL 2*M-1 DO W[J]);
 FOR J1 := 1 STEP 1 UNTIL GAMMA DO
   BEGIN
    FOR L1 := 1 STEP 1 UNTIL S   DO
     BEGIN
      FOR L := 1 STEP 1 UNTIL D DO
      BEGIN
       I:= 2*L+2*(L1-1)*D-1;
       K := 2*((L+(L1-1)*D*2) MOD N)-1;
       U:= K+2*D;
       G := (L1-1)*D;
       B1:= Z[U-1]*W[G]-Z[U]*W[G+M];
       B2:=Z[U-1]*W[G+M]+Z[U]*W[G];
       A[I-1]:=Z[K-1]+B1;
       A[I]:= Z[K]+B2;
       A[I+N-1]:= Z[K-1]-B1;
       A[I+N]:= Z[K]-B2;
      END;
     END;
   WRITE(LINF,R8, FOR J := 0 STEP 1 UNTIL N3 DO A[J]);
    D := D DIV 2;
    S:= S*2;
    FOR J:= 0 STEP 1 UNTIL N3 DO
     BEGIN
      Z[J]:= A[J];
     END;
   END;
  FOR J := 0 STEP 1 UNTIL N-1 DO
  BEGIN
   Y[((J+M-1) MOD N) +1] := SQRT(Z[2*J+1]**2 + Z[2*J]**2)/(F1*N);
   W[J]:= (J-M-1)*DELF;
  END;
   WRITE(LINE,R8,FOR J:= 0 STEP 1 UNTIL N-1 DO Y[J]);
 WRITE(LINE[SPACE 4]);
 WRITE(LINE,R4);
 WRITE(LINE[SPACE 2]);
 FOR J:= F4 STEP 1 UNTIL M DO
  BEGIN
   IF(Y[2*J-2] EQL 0) THEN
    WRITE(LINE,R5,W[2*J-1],Y[2*J-2],W[2*J],Y[2*J-1],C*LOG(Y[2*J-1]))
   ELSE
    WRITE(LINE,R6,W[2*J-1],Y[2*J-2],C*LOG(Y[2*J-2]),
          W[2*J],Y[2*J-1],C*LOG(Y[2*J-1]));
  END;
 WRITE(LINE[SPACE 4]);
 WRITE(LINE,R11,DELF);
END
```

Figure A.17 ALGOL Version of Glassman's FFT Algoritm (Cont'd)



A. Initial State

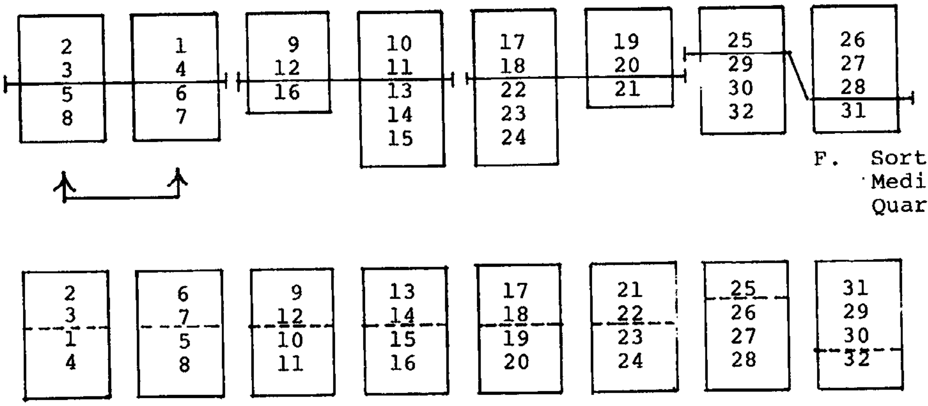
B. Sorted and Median Marked

C. After Processor-to-Processor Swap

D. Sorted and Median of Each Half Markeu

E. After Processor-to-Processor Swap

Figure A.18 Example of a Sort Algorithm Using 2^N Processors



F. Sorted and Median of Each Quarter Marked

G. After Processor-to-Processor Swap

Figure A.18 Example of a Sort Algorithm Using 2^N Processors (Cont'd)

3. Sort again within each processor.

4. Divide the range over which the median is to be found in half, and find separate medians for each half. The state is now Figure A.18.D.

5. Decrease m by one, that is, divide the swapping distance in half, and swap again. The result is Figure A.18.E.

6. Repeat steps 3, 4, and 5, finding medians over ranges which are divided in half each time, and swapping over distances which are divided in half each time, until the swappping distance is reduced to one. For the example with eight processors, step six goes only once, producing the result shown in Figure A.18.G.

7. Sort again in each processor.

Processor utilization depends on the uniformity with which the data is distributed among the processors in the intermediate steps, since the data is equally divided among all 2^n processors both at the beginning and the end of the algorithm. As an example, consider the sorting between the states of Figures A.18.C and A.18.D. Assume that the amount of time taken in a single processor is proportional to NlogN. Processor No. 4 has 6 items, and takes a time proportional to 6log6. Processor 0 has 2 items and takes a time porportional to 2log2. The total time spent working is proportional to 56.66 while the longest processor time times 8 is 86.02, giving a processor utilization during this step of 65.9 percent.

The actual FMP has 512 processors. Again, in the first and last step, data is uniformly distributed among the processors. In the intermediate steps, there will be some spread. If data were randomly distributed among the processors it would take on approximately the Poisson distribution, and the amount of data in the fullest processor could be estimated from that. Given that there are an average of N items per processor, in the Poisson case the processor with the most elements would have about $N+3N^{\frac{1}{2}}$ elements for N large enough. For N=10, a table of the Poisson distribution shows that one processor in 512 is expected to have 19.8 elements, whereas the approximation for N large gives 19.5.

Finally, an interesting observation: if the items to be sorted happen to be in inverse order, it turns out that the distribution among processors remains uniform through the entire procedure, and processor utilization is 100 percent.

Appendix A
References

1. C. M. Hung and R. W. MacCormack "Numerical Solution of Supersonic Laminar Flow over a Three Dimensional Compression Corner" AIAA preprint 77-694

2. "A Documentation of the GISS Nine-level Atmospheric General Circulation Model", Computer Sciences Corporation

3. William D. Stanley, Digital Signal Processing, Reston Publ Co.

4. J. A. Glassman, "A Generalization of the Fast Fourier Transform", IEEE Transactions on Computers, C-19, #2, Feb. 70.

5. Anonymous, FFT/ALGOL, An Algol program compiled on the BSP projects B-7700 (Oct. 28, 1975.)

APPENDIX B
FMP CONNECTION NETWORK - ANALYSIS AND EVALUATION

B.1 SUMMARY

A connection network (CN) is to stand between the 512 processors and the 521 EM modules and is to satisfy these requirements. The connection network would accept requests from the processors, possibly all 512 simultaneously and establish connections between the requesting processor and the requested EM module at EM memory speed. A crossbar switch between processors and memory modules can provide this function, but at a terrible cost in hardware. It has N^2 crosspoints where N is the number of ports along one side.

This appendix describes a connection network based on the Omega network (described below and in [4]. The Omega network has $O(N\log_2 N)$ components, not $O(N^2)$. The particular network which appears to best satisfy the Connection Network requirements is a duplex Omega network, providing redundancy for additional reliability, as well as providing the required function.

The appendix is arranged in the following sequence. First, some background information and definitions are presented. Second, the advantages and disadvantages of providing a CN that satisfies the requirements of being functionally "almost" a crossbar switch are presented, especially as compared to the original TN (1,2). Third, various candidate versions of the connection network are described in detail, including estimates of relative hardware complexity of each. Fourth, Simulation results on these candidates are presented, obtained by a functional simulator of the CN and by a second program called the stochastic analyzer. Fifth, further discussion of the simulation results in used to narrow down the selection of CN to one or two of the cases simulated and analyzed. Following that, there is a discussion of other CN-related topics, including some of the design details that were disclosed by the simulation results, and finally, a paragraph of conclusions. The conclusion reached is that sufficient study has been completed to give confidence in the feasibility of the Connection Network in the FMP architecture, but that cost/performance tradeoffs deserve to be further considered.

Discussion of the simulators and analyzers has been relegated to Appendix H.

B.2 BACKGROUND

The connection network can be visualized as a circuit-switched dial-up network in which up to 512 callers (the processors) are placing short calls to the 521 callees (the EM modules). Connections are to be made in tens of nanoseconds, and held for a few hundred nanoseconds. Except for the time scale, the action is like that of the telephone network, and hence, the design of such a network starts with work done at Bell labs (3). The work of Duncan Lawrie (4) is especially applicable.

Many similar networks have been developed, but which have been shown to be topologically equivalent to each other (5,9)* One name, the "Omega" network has been chosen as the term to use for any of this class of networks. The Omega network is shown in a form called the "baseline" network in (9).

In the FMP architecture being evaluated, each processor computes its own address in EM. There is no central location where the switching pattern for the entire network is defined. All patterns of connection are possible. Since connections must be made in tens of nanoseconds, there is no time to take a global look at the entire pattern, and generate a set of control bits for the network. Hence, control of the various portions of the network must be local to those portions.

Several different networks have been investigated, and feasibility of the FMP can be achieved with several of them. The underlying Omega network design, on which the preferred versions are based, has 1024 ports on the processor side, 1024 ports on the EM module side, and ten levels of nodes in between. There are 1024 data paths connecting one level to the next. Each level consists of 512 two-by-two switches, which are described in more detail below. The connections between nodes exhibit a pattern of connections designed to permit as many processors as possible to access EM modules simultaneously in parallel for the patterns of accessing which occur in the aero flow and weather codes.

The previously described transposition network (1, 2), was centrally controlled, and required two ten-bit control settings, one of which was the skip-distance of a p-ordered vector (defined below). The transposition network consisted of two barrel switches, one 521 wide, one 520 wide, and some appropriate wiring. Since a barrel switch that is wider than 512 but not more than 1024 wide, can be built of five levels of one-by-four switches, the TN also had ten levels of logic. All transfers through the TN must be synchronized to the control settings, and only those processors whose requests fit the constant skip-distance, constant offset description could execute during the duration of a particular control setting.

* Names include: "baseline network" (5), "binary n-cube", "butterfly", "flip network", "Omega network", "reverse baseline", "simplified data manipulator", "hypertorus", and "SW banyan network".

B.2.1 Definitions

Certain definitions are necessary in order to understand the rest of this appendix. They are:

B.2.1.1 P-Ordered Vector

A p-ordered vector is a set of EM addresses such that the address being accessed by the ith processor is in EM module number (d + p*i) modulo 521 where d is called the "offset" and "p" is the "skip distance".

B.2.1.2 P-Q-Ordered Vector

A p-q-ordered vector is a set of EM addresses such that the EM module number being accessed by the ith processor is (s + p*i + q* $\lfloor i/k \rfloor$) modulo 521 where k is the "length of each piece", s and p are "offset" and "skip distance" as above, and q is the "distance between pieces". The bottom brackets represent the "largest integer not greater than".

For a system where there are 16 processors and 17 EM modules, an example of a p-ordered vector would be for the 16 processors to request access to the following memory modules respectively:

10 13 16, 2, 5, 8, 11, 14, 0, 3, 6, 9, 12 15, 1, 4
where the offset is 10, and the skip distance is 3. For this same system, a p-q-ordered vector with p equal to 1, and five elements per piece, the processors might be requesting from the following EM modules respectively:

11, 12, 13, 14, 15, 3, 4, 5, 6, 7, 12, 13, 14, 15, 16, 1
In this case, p is 1, q is 4, and the length of the piece is 5. Numbers are interpreted modulo 17.

B.2.1.3 Random Request

A set of EM addresses such that the EM module being accessed by the ith processor is a random variable, from 0 through 520, which is independent of the module number being accessed by any other processor.

B.2.1.4 Blockage

Blockage is the result when two requests try to share the same path in the connection network, which can then only supply a path for one of them.

B.2.1.5 Conflict

Conflict is more than one processor accessing the same EM module simulaneously.

B.2.1.6 Pileup

Pileup is the number of processor having a conflict at a given EM module.

B.2.1.7 Frame

A frame is a parcel of data of fixed size sent over a transmission path. In the CN, each frame is 11 bits; five successive frames make a data word.

B.3 ADVANTAGES

The advantages of the CN over the previously studied Transposition Network (TN) (1, 2) are simplification of user programming, simplification of the compiler, improved performance, and broader spectrum of applications.

Compiler simplification arises because each processor computes EM addresses independently of the other processors. The compiler need not be aware of the relationships between those addresses. No code is emitted to compute offsets, or skip distnaces, or to control how many LOADEM instructions are issued. No restrictions need be imposed on subscript expressions. All of these represent simplications of the situation for an FMP using the previously studied TN, where the compiler would have had to create an alternate branch with dummy LOADEM instructions to keep synchronization, even when a given processor will skip all actual computation in a section of code containing EM accesses. The connection network (CN) does not require any synchronization, and thus eliminates all dummy LOADEM instructions.

When the various instances of the DOALL fetch a set of array elements that do not form a set of linearly spaced elements, no user precautions and no analysis by the compiler are required. Examples of nonlinearly spaced elements are the wraparound on longitude in the GISS weather, and the offsetting of the index J in subroutine CHRVAL of the 2D MacCormack aero flow code. In the baseline system these would not have been allowed. The user would have had to vectorize them. The independent programming of the processor can make the FMP more than merely a vector machine, so this restriction, imposed by the TN, represented an incompatability with the system objectives.

When the Connection Network is used, system performance would be affected much less by problem size than when the TN is used. For example, consider the fetching of data subscribed with the domain variables in two-dimensional DOALLs. Say the subscripts are I, J and K. Within the DOALL over I and J, fetches of an array

A(I,J,K) from p-ordered vectors with p equal 1. Within the DOALL over J, and K, fetches of A(I,J,K) form p-ordered vectors with p equal to IMAX. Within the DOALL over I and K, fetches of A(I,J,K) form p-q-ordered vectors with p equal 1 and q equal to IMAX*(JMAX - 1). With the TN, all p-ordered vectors are fetched in one LOADEM, but a fetch of a p-q-ordered vector required that each piece of a p-q-ordered vector be fetched with a separate LOADEM operation, or 512/IMAX fetch operations just to load one datum per processor. With the new CN, the number of EM cycles required for a p-q-ordered vector is controlled by the largest pileup, usually a much smaller number than the number of LOADEM instructions needed with the TN. The pileup for p-q-ordered vectors is discussed further in Section B.7.4, together with some simulation results.

For the specific example that comes from the 3D explicit code given us by NASA, in the smaller than normal mesh size of 31 x 31 x 31, the improvement is dramatic, from 17 LOADEM instructions required to fetch A(I,J,K) over I and K, to a maximum pileup of depth 2. Simulation of this case showed that all processors received their data within two EM cylces.

The following development shows this advantage of CN over TN, in analytic form. The slowest processor is the one holding up the synchronization at the end of a DOALL. If access times were normally distributed with mean T_{av} and standard deviation S, then the worst total of N access times, out of 512 such totals (i.e., the most-delayed processor) would have a value given in Equation B.1.

$$\text{Max Delay} = N \cdot T_{av} + 3 \cdot N^{\frac{1}{2}} \cdot S \qquad \text{(B.1)}$$

Because of the central limit theorem, this formula is valid for large enough N without any need for assuming an underlying normal distribution. Equation B.2 gives the corresponding formula for the old TN.

$$\text{Max Delay} = N\ T_{max} \qquad \text{(B.2)}$$

T_{max} is the time for however many LOADEM instructions are required per fetch and may be many times T_{av}. The reason for the improvement of equation B.1 over B.2 is that synchronization among processors for EM accessing is not required with the CN, so that each processor continues executing without any wait for the slowest processor.

A possible wait would exist only at the end of a DOALL where data precedence may force a synchronization. N in equations B.1 and B.2 is the number of accesses between such waits.

A substantial gain in user convenience would be achieved with the CN. All tricks such as adding dummy instances to the DOALL to make the domain size equal to the array extents are unnecessary. There are no "magic" array extents or DOALL domain sizes for making the third direction have the same speed as the other two with the CN. Likewise, all need to distort the algorithm to regularize the subscript expressions would disappear. The code shown in Figure B.1 is an FMP FORTRAN version of some statements abstracted from subroutine CHRVAL in a 2D explicit code given to Burroughs during the previous study. The subscript "J + OFFSET", being the result of data dependent computations, would have been disallowed in the originally proposed FMP FORTRAN (1, 2) because of the restrictions imposed by the TN. Such a subscript is perfectly proper in the currently described FMP FORTRAN. The many awkward and arbitrary restrictions on the language, imposed by the access pattern limitations of the old TN, are not required in a system using the proposed Connection Network. Any integer expression can be used as a subscript.

B.4 CN DESCRIPTION

The Connection Network (CN) has two modes of operation. First, when the processors are independently operating, it would provide a path from any given processor to the EM module of that processor's choice, without regard to any other (up to 511) connections from other processors to other EM modules. Second, certain functions would be performed in synchronism, because these functions are much more economical to implement when the processors are synchronized. This second class of functions would be done under coordinator command, at a time when all processors are in synchronism. This second class includes

- Broadcast from coordinator to processors
- "Harvest" data from processors to coordinator
- Broadcast from one EM module to all processors (FETCHEM)
- Swap data between pairs of processors

Various CN design options are based on either a Benes or Omega network. The Benes can make any permutation of connections between processors on one side and EM modules on the other, but only at the cost of having each connection a function of the connectivity of all others. Opferman and Tsao-Wu (6) show that the computation of this "perfect" connection takes on the order of N^2 computational steps (or N $\log_2(N)$ if a content addressible memory is used). Thus, for making connections in nanoseconds, it is not possible to compute the control settings of a Benes network for each set of new EM addresses. Instead, each node of the network determines its own setting, based on some very simple computation, with sufficient redundancy that a path for sufficiently many of the processors is set up in the desired access time, with random distribtuion of the excess time among all processors.

```
      DOALL, J=1,JM; K=1,KM
1     ... statements  ..
      IF (condition) GO TO 10
      OFFSET = OFFSET + 1
      ... statements ...
      IF (DYX(J,K) LT.0) OFFSET = OFFSET - 1
      IF (J + OFFSET .GT. 1) GO TO 1
      ... statements ...
10    ... statements ...
      DYX(J + OFFSET,K) = expression
      ENDDO
```

Figure B.1 FMP FORTRAN of Portions of Subroutine CHRVAL of 2D Explicit Code

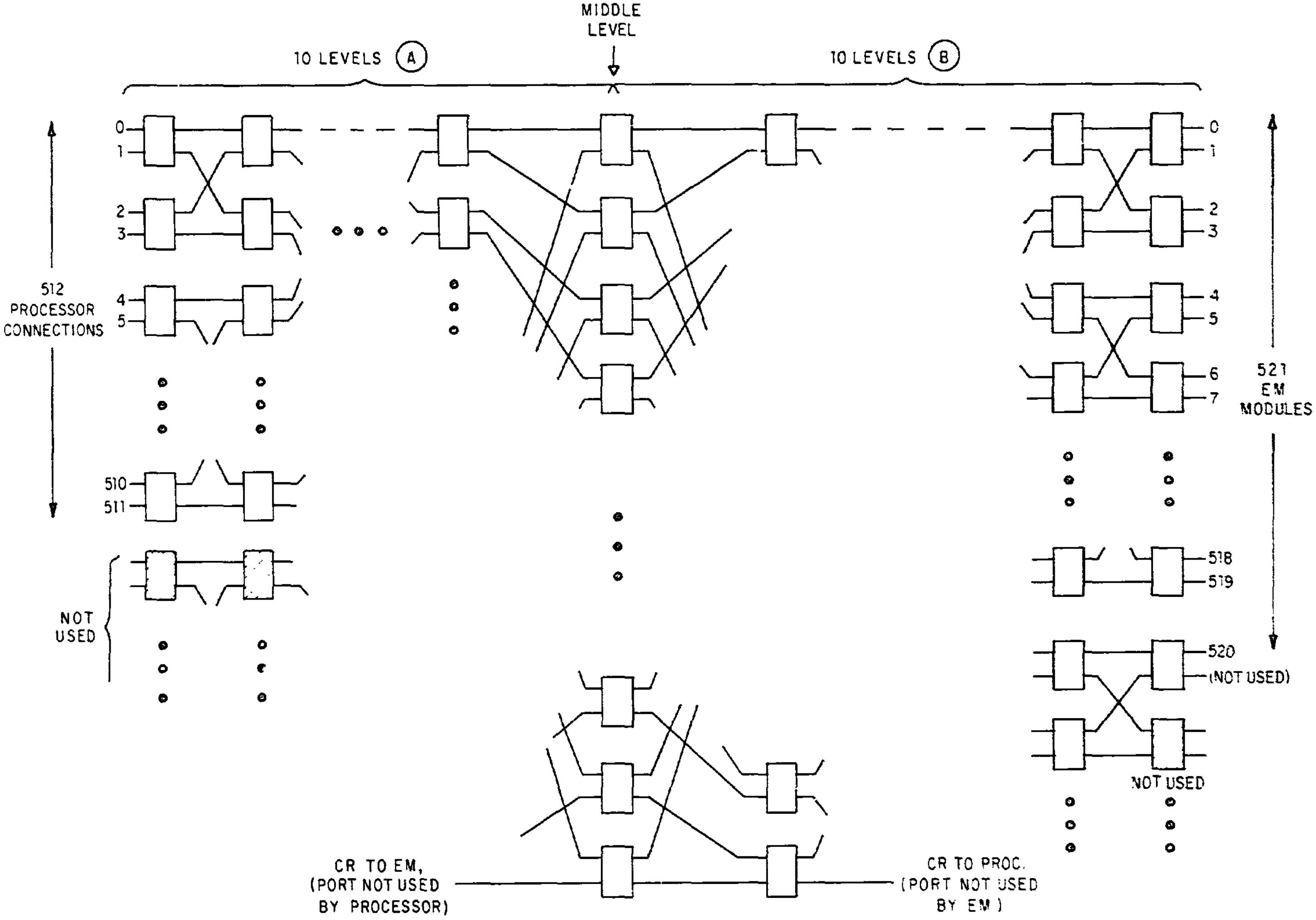


Figure B.2   Partial View, Benes Network

Out of several candidate designs, simulations have been used to indicate the most efficient, i.e., fastest access time, lowest occurrence of blocking, and smallest parts count.

Figure B.2 shows the first variation considered. In this case, a 1024-wide Benes network (only some of the edge nodes are shown) has the first 512 ports on the left attached to the 512 processors, and the first 521 ports on the right attached to the 521 EM modules. Detailed examination shows many of the nodes can never be used. The middle level must have a full 512 nodes to switch iuts 1024 data paths (at two paths per node). In the remaining nine levels not all the data paths can reach any EM module, so that only some of the data paths need be implemented. Parts counts can be derived from Table B.1 which gives the number of paths required at each level of nodes:

TABLE B.1
Width vs. Layer Number

Layer No.	Number of paths (=2 x nodes)
10	1024
11	1024
12	768
13	640
14	576
15	544
16	528
17	528
18	524
19	522

On the side with processor ports, the 512 processors can all fit into a 256 node, 512-wide path, with the result that of the 1024 paths to the middle, half are unused. Figure B.3 shows a smaller example with 8 processors and 11 EM modules.

Each node is a 2 x 2 crossbar switch and is described in detail in Section B.4.2. When paths from individual processors to individual EM modules are set up (the "normal" mode of operation), each node connects in either one of two ways:

1. Processor-side port A to EM-side port X, also B to Y (straight-through).
2. Processor-side port A to EM-side port Y, also B to X (crossed).

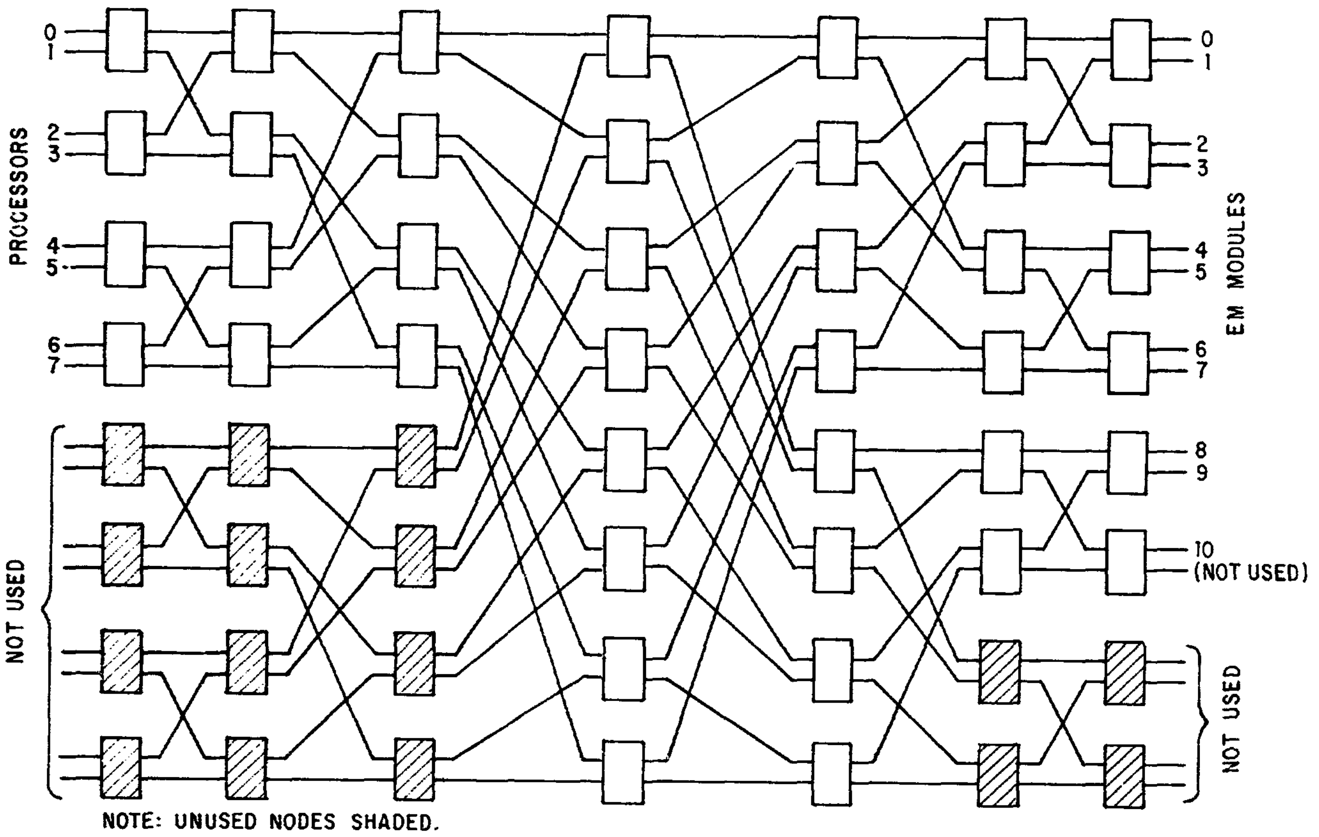


Figure B.3 Benes Network Connecting 8 Processor to 11 EM Modules

As long as mode of operation is "normal", only one bit of information is required to determine the setting of a node. When only one processor-side port has a pending request, that port provides the bit of control information. When both ports have pending requests, one port must be chosen to provide the bit of control information. That port is said to have "priority" over the other one.

Each node determines its setting from this bit as follows. If the bit is ONE, the port with the request is connected to the lower EM-side port; if the bit is ZERO, the input port with the request is connected through to the upper EM-side port. The control bit is one of the bits of the port number on the EM side. The middle level of nodes uses the most significant bit of output port number, the two levels on either side of the middle use the next to most significant bit, and so on to the first and last node levels which use the least significant bit.

B.4.1 <u>Versions of Networks Considered</u>

Several variations on this idea have been devised and simulated. Figure B.4 is a revised version of Figure B.2, showing the entire network, but eliminating the detailed depiction of each individual 4node. It is, as has been previously noted, isomorphic to a base-2 Benes network with some of the nodes omitted. Processor ports, and EM ports, are each packed into the first 512 network ports on both sides.

Figure B.5 shows the processors spread across every other port at the left side of a 1024-wide network. The additional nodes hopefully provide some redundancy in the connectivity.

Figure B.6 shows both processors and EM modules spread across a 1024-wide Benes. To simulate the spreading of EM modules, transform module number M into a new module number M' as in equation B.3.

$$\begin{array}{ll} M' = 2M & 0 \leqslant M \leqslant 511 \\ M' = 2(M-512)+1 & 512 \leqslant M \leqslant 520 \end{array}$$

These expressions result when M is shifted left end-around one bit position.

Figure B.7 shows the same number of nodes as in Figure B.6 arranged as two second-halves of the Benes network. Duncan Lawrie calls such a second half an "Omega network" [4]. The idea is that if an access is not granted through the upper Omega, the processor could try a few nanoseconds later through the bottom Omega.

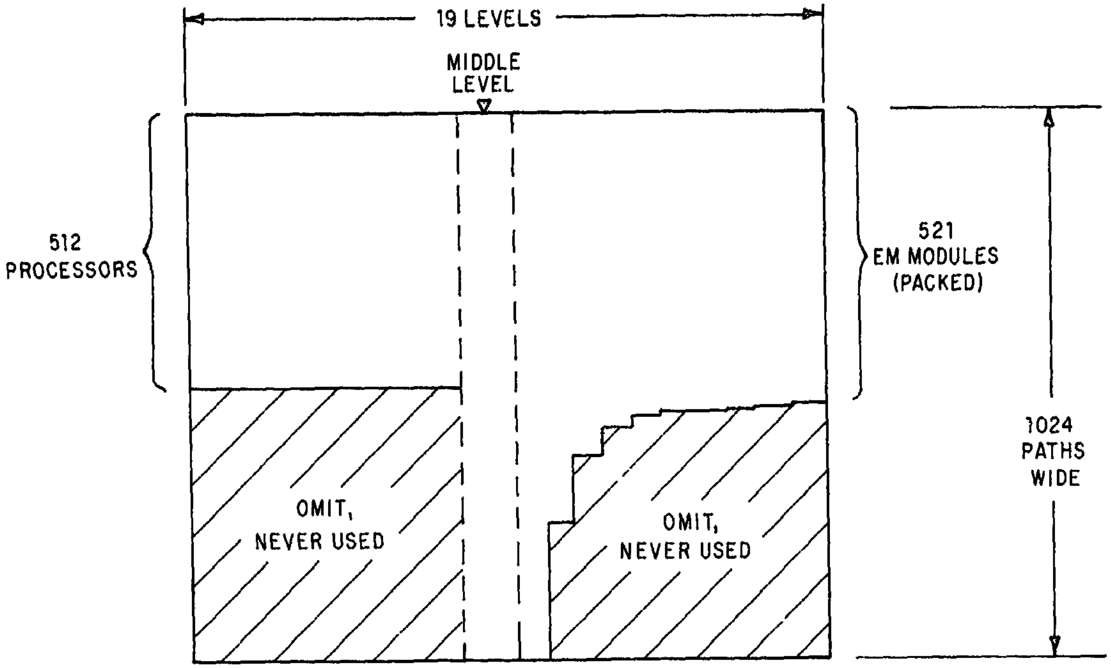


Figure B.4 Full View of Figure B.1, Details Suppressed. Benes Network

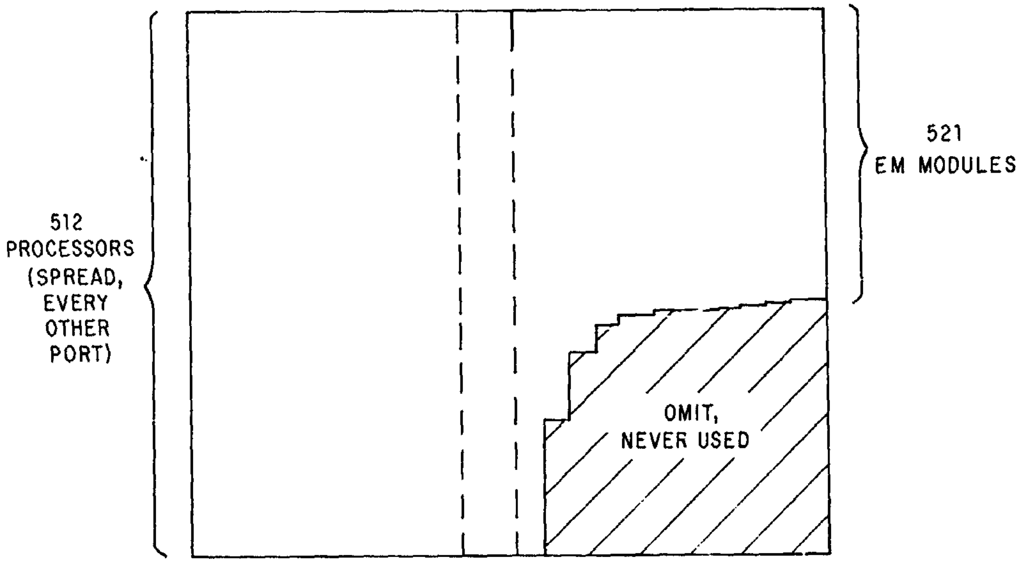


Figure B.5 Benes Network with Processor Ports Spread

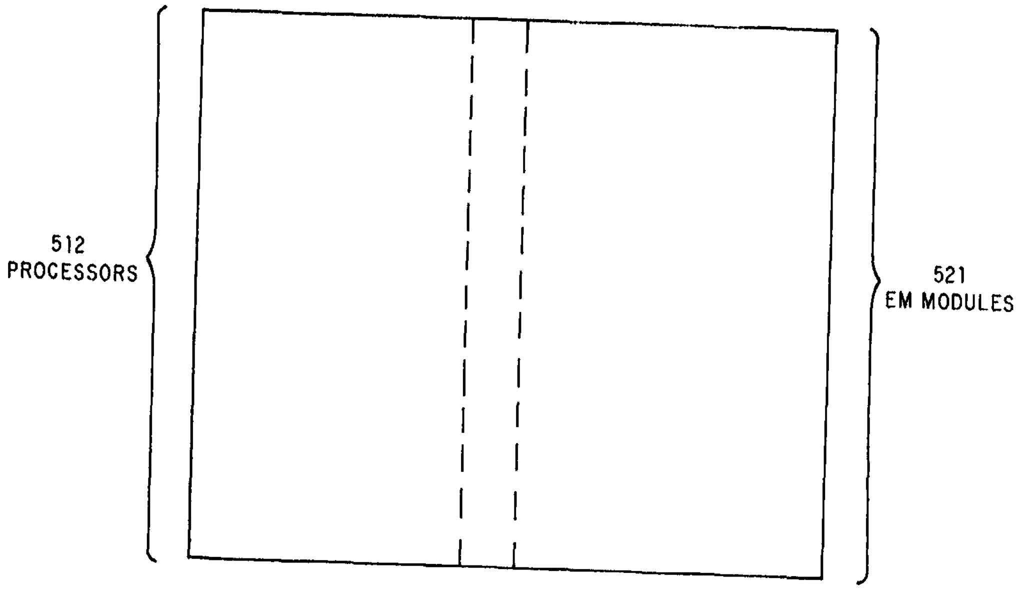


Figure B.6 Benes Network, Both Edges with Ports Spread

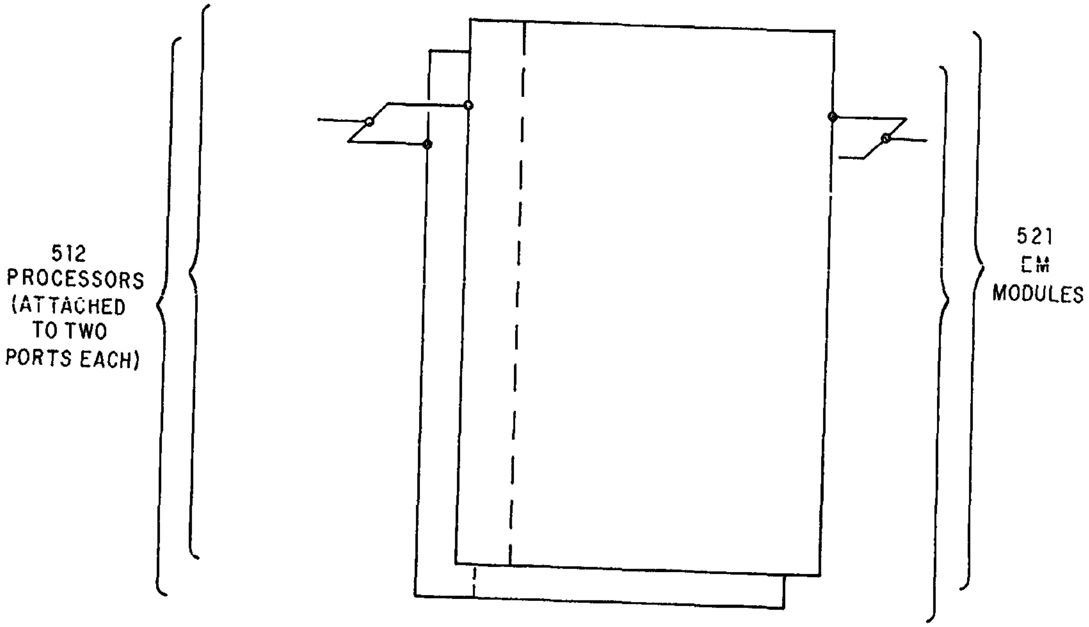


Figure B.7 Double Omega Network, Two Layers, Each the Second Half of a Benes

B.4.2 Logic Design

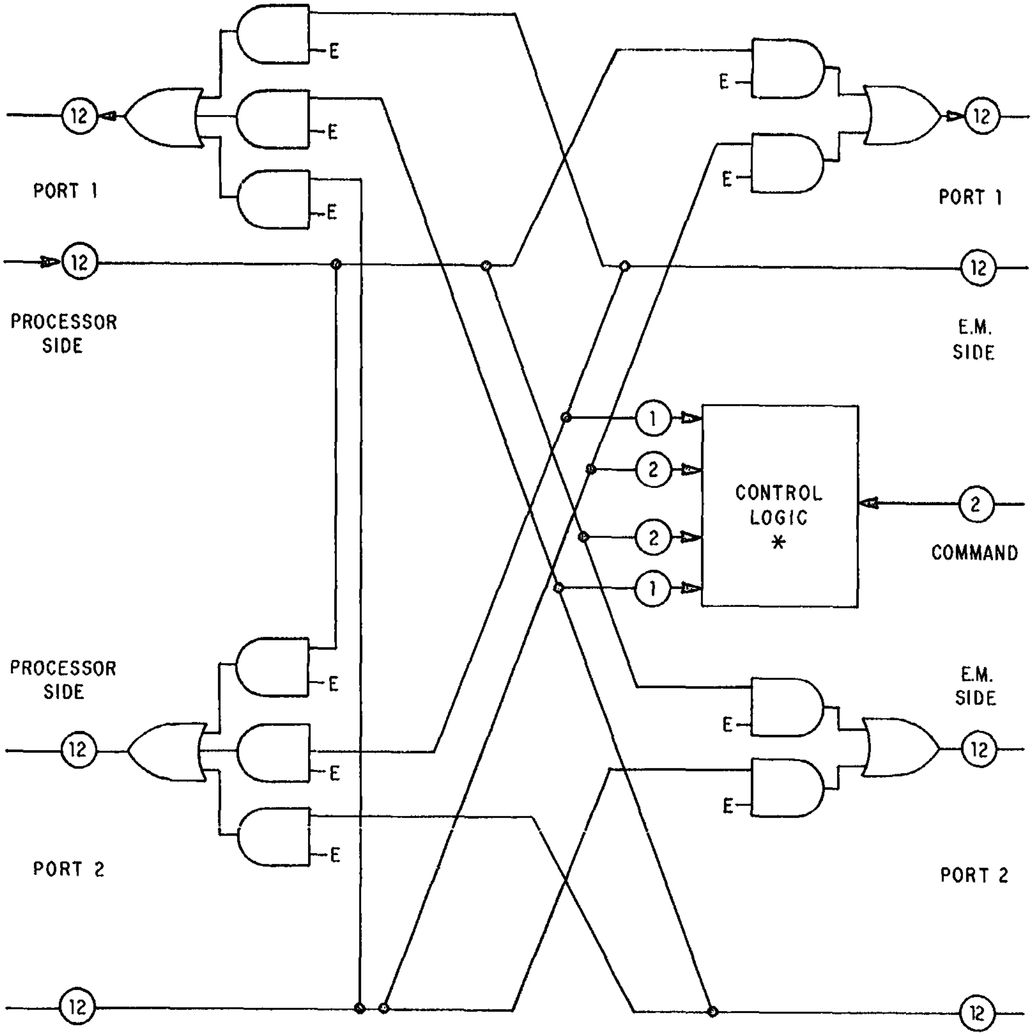
Figure B.8 shows the basic node of any version. Two bidirectional 12-bit-wide paths connect to each side. On the processor side they are labelled A and B, and the EM side X and Y. Internal connections may be made from A to either X or Y, and from B to either X or Y. The 12 bits from processor to EM are used for EM module number +1 bit plus "strobe" when transfers are going on. The twelve bits returning to the processor are 11 bits of data plus a "latchup" bit. The "latchup" bit is a command to the node to keep this path connected. The "latchup" bit would be transmitted from the EM module upon recognizing a valid request coming from the CN, and serves to keep the path connected as long as "latchup" is true.

Logic in the CN buffer of the processor uses latchup as the "acknowledge" that signals that a request has been granted. Latchup could be dropped by the EM module after the operation being performed ceases to need the data path. Alternatively, timing could occur in the CN buffer, and the dropping of strobe could be the signal to the EM to drop latchup.

The resting state is shown in Figure B.9. "Requests", consisting of EM module numbers, may or may not be coming out of the processors, and the connectivity of the node is set up according to the specified function of module number bit and port bit (A vs. B). The "latchup" bit coming back from the EM modules is false. Connectivity is switched as fast as the requests change, since the initial path connection is pure combinational logic. The command lines from the CU have a "null" command.

At some time one of the requests finds its way to the correct EM module, which then emits a "latchup" pulse. Other processors must not disrupt the chosen path before it is latched. Therefore, there is a "CN clock" in all processors, with a period longer than the round-trip time of the CN, so that new requests are emitted only after old requests have had a chance to latch up. The round trip delay is about 40 feet of wire, plus 19 gates worth of logic delay going out and 19 gates worth of logic delay coming back. If gate delays are 1.5 ns (including some allowance for wiring on (the boards), and wire is 1.6 ns/ft (Teflon or polyester belts), this delay is 120 ns and sets a lower limit to the CN clock period. This clock is a second timing signal to each p ocessor (the first is the main clock), not a countdown of the clock within the processor. This timing signal selects every Nth pulse of the main clock.

Figure B.10, B.11 and B.12 show various latched-up states. Figure B.11 shows the "straight-through" connection, Figure B.12 shows the "crossed" connection.



* GENERATES ALL SIGNALS LABELLED "E"

Figure B.8 Basic Node

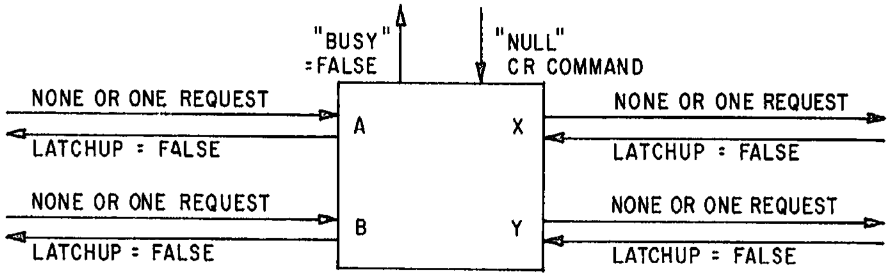


Figure B.9 Resting State of Node

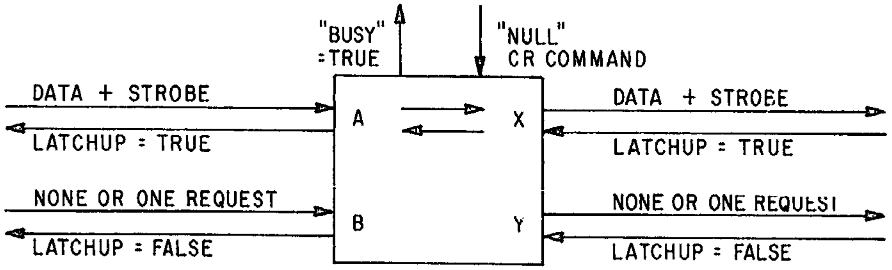


Figure B.10 Latchup State, One Path, With Data Transferring to EM

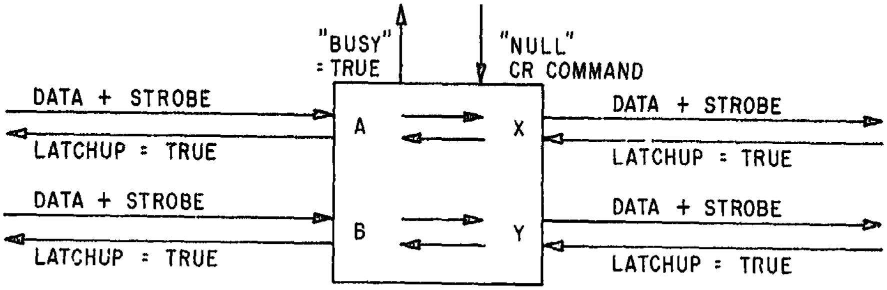


Figure B.11 Latchup State, Both Paths, Both Transferring to EM

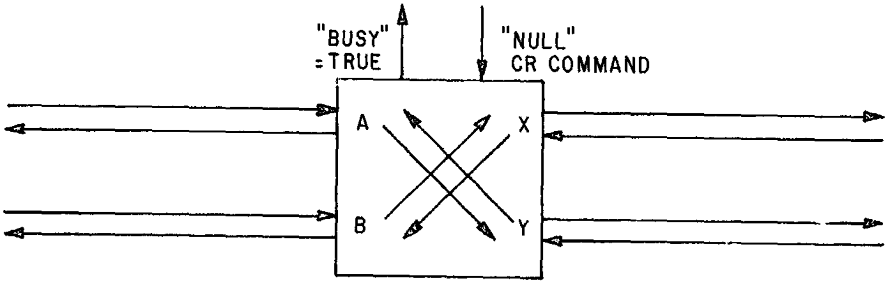


Figure B.12 Latchup State, Both Paths, Crossed Connectivity

B.4.3 CN Function Controls

The Connection Network serves other interconnection functions in the proposed system besides the processor-EM paths. Other functions are controlled from the coordinator. A list of CN states defined by the coordinator's control is the following paragraphs.

B.4.3.1 "BDCST/HVST"

The "BDCST/HVST" command makes a connection from both A and B to both X and Y. Data from the CR enters all nodes at Y, and by fanning out to both A and B, will reach all processors. Data from the processors enters at both A and B, and is either ORed or ANDed (it does not matter which) to be combined at the Y-port that the coordinator listens to. This command is used for FETCHEM as described below, and for HVST.

B.4.3.2 "Null"

With the coordinator (CR) node command turned off, the node carries out its wired-in function of passing on requests, and latching up for the "latchup" signal from the EM module as previously described.

B.4.3.3 "Wraparound"

Connect port A to port B. This implements the SHIFCN function in this CN. When the Nth level has a wraparound command, then every processor is connected to the processor whose CN port number is different in the Nth bit. N is counted from the left in both Figure B.4, and in Figure B.7. The "wraparound" command is used for processor-to-processor data swapping. Depending on in which of the ten levels of the CN is the node getting the "wraparound" command, data will be swapped between two CN ports which differ only in one bit of their number. Normally, all nodes of the same level get the wraparound command, with the result that all CN ports swap data with those ports that differ by just one bit in the specified bit position. SUMALL for example, can be implemented by swapping data that is just one apart on port number ("wraparound" on the least significant bit) and adding, then by swapping that sum two apart and adding four apart, and so on up to 256 apart.

B.4.3.4 Diagnostic Commands

As described in Chapter 6, Section 6.1, the individual Omega networks (layers) of the two-Omega network must be tested separately for diagnostic purposes. Thus, we need a command to disable one Omega network while testing the other one. See Section 6.1 for additional details.

B.4.3.5 FETCHEM

The FETCHEM instruction is implemented in two steps. First, the EM module number is sent from the coordinator as a normal request after the processors are synchronized (in order to ensure that processors are not making requests of their own). This request is accompanied by a command code to the EM module that causes reading without sending any latchup.

During the access time of the EM module, the coordinator turns the etnire CN on with the "BDCST/HVST" command. Data from the selected EM module is therefore broadcast to all processors. Inactive EM modules emit zeroes to be ORed with the data (or ONEs to be ANDed with the data, depending on how the nodes are implemented).

B.4.3.6 HVST

The HVST instruction is implemented by the coordinator setting the CN to the "BDCST/HVST" state, at a time when the CN buffers are "full" and the processors otherwise idle, and then issuing "go", which is the command being expected by the CN buffer for dumping the data into the CN. The data arrives at all EM-side CN ports, including the port that delivers the data to the coordinator. HVST is intended to be used primarily for the case that only one processor is enabled, therefore, it does not matter whether that data is ORed with zeros, or ANDed with ones, in the CN. The result is that it shall be left as logic designer's option whether the words combined during "BDCST/HVST" are ORred or ANDed.

B.4.3.7 Coordinator Access to EM

CR fetches and stores from EM are no different from processor LOADEMs and STOREMs. The CR has its own CN buffer.

B.4.3.8 CN to Coordinator Status

Each node emits "busy" = 1 whenever one of its two paths is latched up. The condition that no node be "busy" is necessary before the CN can switch to some other command. Now it may be possible for the CR to tell, from the state of synch of the array, when the CN is idle, so there may be no need for the "busy" bit. Until the rest of the design is finalized, however, a "busy" bit is assumed.

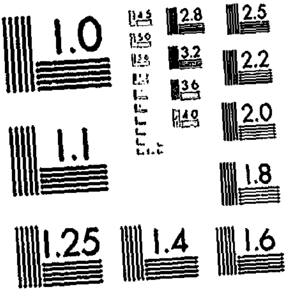
B.4.4 Implementation Details

B.4.4.1 Flip Flops

Two alternative design approaches for the node are:

1. No flipflops, just logic that is latched up by the "latchup" signal, as described above.

26072



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

2. Path-holding flipflops that are clocked by the same 120 ns CN clock that is seen as needed for timing the processor requests. These flipflops hold the path for just one CN clock period.

Approach 2 has more gates, but permits faster access to extended memory. Figure B.13 shows the timing diagram for the two cases. In case 1, where the node is combinational logic only, the EM module number contained in the "request" must be held statically by the processor until the "latchup" signal returns from the EM module. Then, and only then, would the processor be free to emit an address toward the EM module over the now latched-up path.

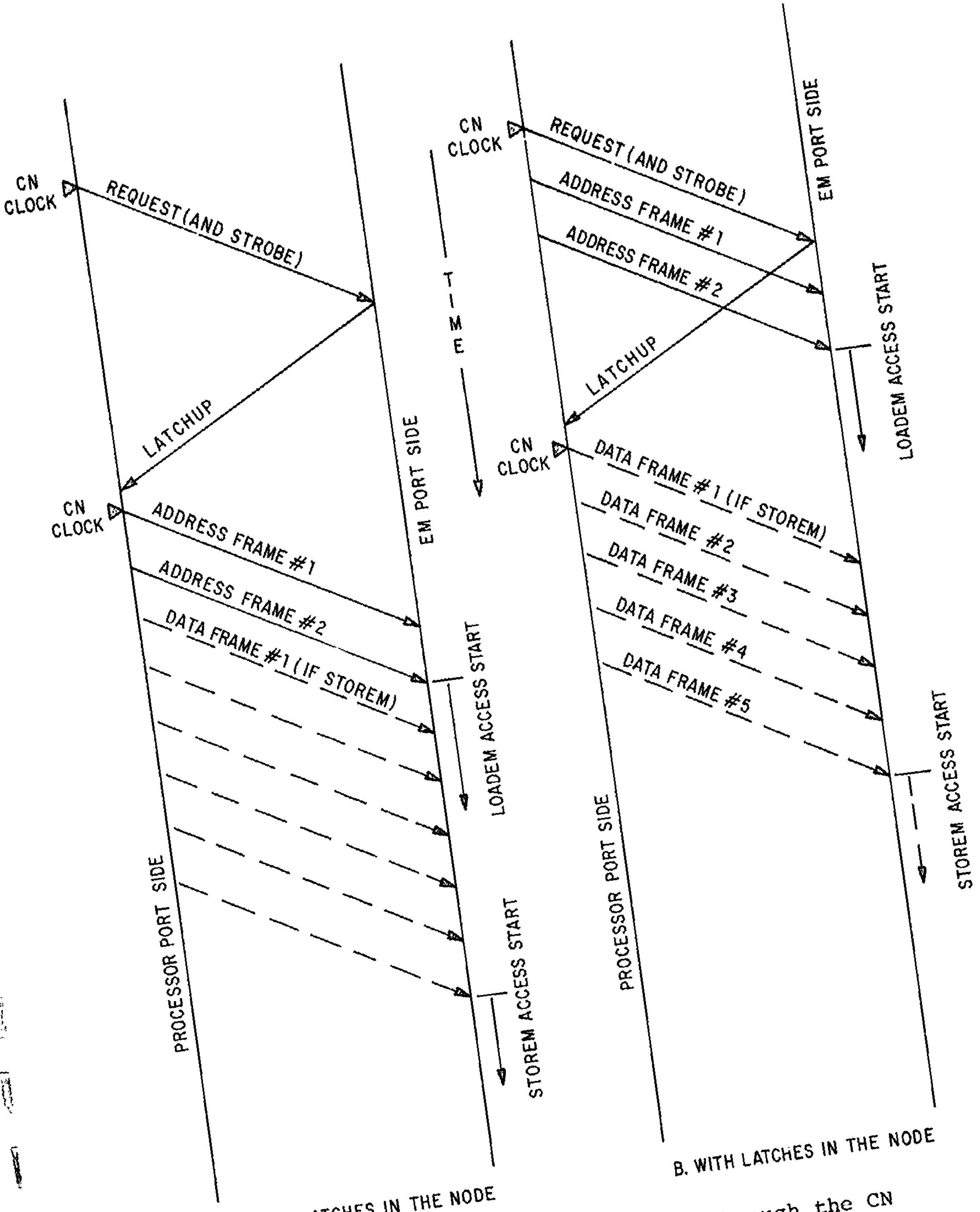
In approach 2, the processor emits the request followed by two frames of address plus operation code. Each frame is 11 bits. The node, seeing one, or two, requests on ports A and B, sets the flipflops with the CN clock, so that the address can continue down the path, if it is possible to reach the EM module for this node. The EM module gets its address about 80 ns sooner than it does in approach 1, cutting 80 ns off the access time. These flipflops will not stay up without the "latchup" signal coming back before the next clock, thus, if the EM module is not reached, a new path is free to be set up on the next CN clock.

B.4.4.2 Wiring

Each node is controlled by one and only one bit of the EM module number in the request of the port with priority. Since all nodes are to be physically identical, the control bit must show up at the same physical location in each node. Thus, previous nodes must have a wiring pattern for the bits it passes along, such that they show up in the control bit position after passing through the correct number of nodes. Figure B.14 shows such a wiring pattern for a 32-wide CN, such as might be appropriate for 16 processors and 17 memories. Figure B.15 shows the first few levels of the 512/521 network, showing the connections from X or Y output from one level to the A or B inputs of the next. Since the interlevel cables are belts, where wires must lie parallel, and since all nodes are to be identical, these crossovers occur on the paddleboards, not in the belts or within the nodes. (Similar offset-by-one wiring patterns are seen on some of the Illiac IV paddleboards.)

B.4.4.3 Logic

Each node contains two 12-wide two-way selector gates, one for each of X and Y outputs, and two 12-wide three-way selector gates, one for each of A and B. The third input would take care of wraparound. Each node also contains some decision making logic. The inputs to the decision making logic are:



A. WITHOUT LATCHES IN THE NODE

B. WITH LATCHES IN THE NODE

Figure B.13 Timing of Transfers Through the CN

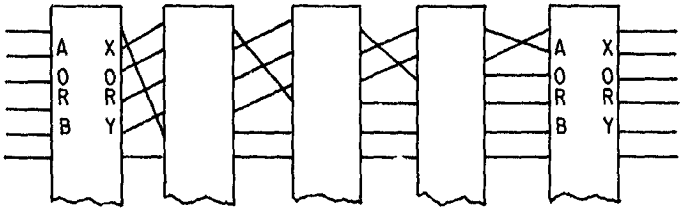


Figure B.14 Wiring Crossover Map, 16 Processors To/From 17 EM Modules

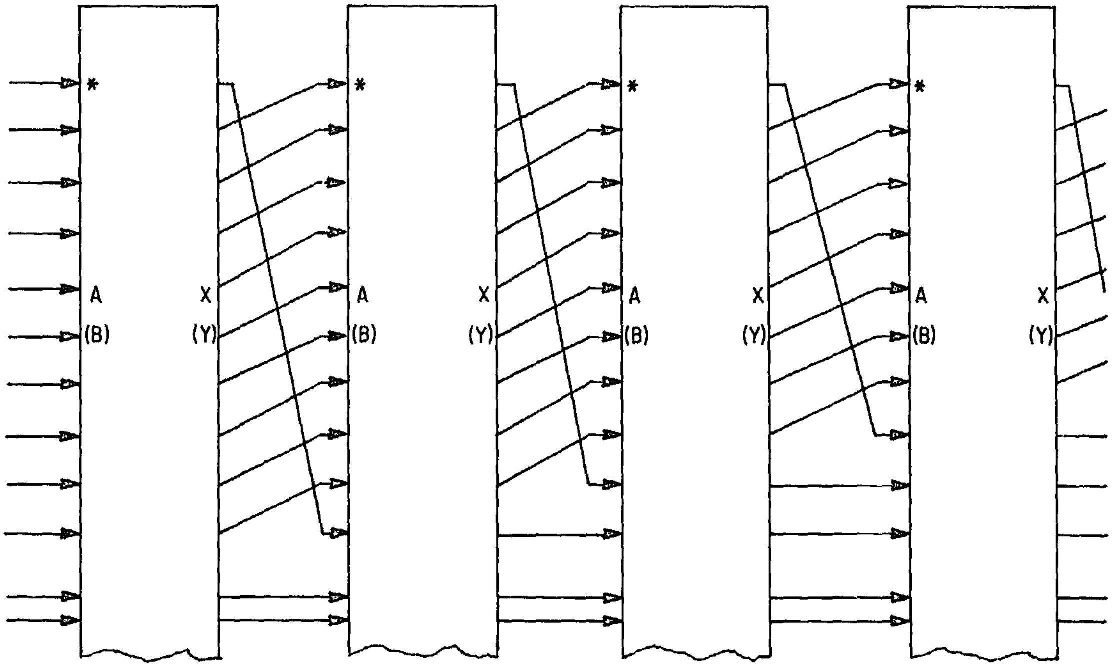


Figure B.15 Wiring Crossover Maps, Full Size System

* 1 bit of the EM module number (for every node, this is the least significant bit of the frame seen at that node because of the wiring patterns of Figure B.12)

1 bit to control priority

* 1 bit, the 11th bit of the "request" frame could be used for calling for processor-to-processor wraparound

3 bits, commend from coordinator

1 "Latchup" bit

* 1 "strobe" bit, bit 12 of the processor-to-EM path

(1 CN clock, if design 2, with flipflops, is used)

Asterisked items occur on both ports (either A and B, or X and Y), leading to a total count of 12 signals that have to be combined in the combinatorial logic. The logic has the following output signals: Select A or B or both or no-output for X, select A or B or both or no-output for Y, select X or Y or B for A, select X or Y or A for B, "busy". Only 16 logically different output signals are needed. "No output", or all lines FALSE, is substituted for an input request that cannot reach its destination.

Feierbach and Stevenson [5] recommend the following algorithm for determining priority: If the request at A and the request at B can both be satisfied, then the node is set to either straight-through or crossed connection, whichever is requested; if the requests conflict, the node is set to straight-through, which will be correct for one of the requests. Thus, A has priority if its request bit is zero; B has priority if its request bit is one. This algorithm introduces no bias against either A or B, but means that certain memory modules will be more easily accessible from A, and other memory modules will be more easily accessible from B. If memory addressing averages out in some sense, then this algorithm is unbiassed.

The priority rule used successfully in validating the CN goes thusly, for the double Omega. In the upper Omega network, the upper port of each node has priority; in the lower Omega network, the lower port of each node has priority. Early simulation results showed priority could not be left to chance, and all double Omega simulation results reported in the next Section, B.5, were done with the priority according to this rule. For the single Omega network, the priortiy was alternated each CN clock period, and there were an odd number of clock periods per EM cycle. This is slightly more complicated than Feierbach and Stevenson's rule, but was judged to be less biassed.

B.4.4.4 Parts Count

The node's parts count, or at least the gate count, is dominated by the selection gates, three-input selection gates for processor directed signals going back out of parts A and B, and two-input selection gates for EM-bound signals out of ports X and Y Using the Benes network with processor ports packed into the first 512 prossor-side ports. and with EM ports packed in the first 521 as an example, the parts count goes as follows. The first nine levels have 512 nodes each. For the last ten levels, we can count the number of nodes per level from the data in Table B.1. The number of nodes is just half of the number of paths passing through a given level. Adding together the 19 numbers representing the number of nodes at each level, gives a total of 5643 nodes. At each node, there are two ports, and a data path that is 12-wide, with 3-input gates in one direction and 2-input gates in the other. Computation gives

```
5643 x 12 x 2 = 135,432 3-way 1-input selection gates, plus
               135,432 2-way 1-input selection gates, for a
 total of      270,864 selection gates
```

By comparison, the Transposition Network in the preliminary study [1, 2] consists of two barrel switches, each bidirectional and 9 bits wide in both directions. If these barrels were to be implemented with 2-way 1-input selection gates, they would have

(520 + 521) x 9 wide x 10 levels x 2 directions = 187 380
2-way 1-input selection gates

Parts count of the other variations differ accordingly.

If the same level of integration can be achieved in both designs, then the Benes network of Figure B.2 should take more chips than the TN by about the ratio of the number of inputs in these selection gates, or

$$\frac{135,\ 336 \times 2 + 135,\ 336 \times 3}{187,\ 380 \times 2} = 180.6\%$$

To such a node count must be added some additional increase because of the combinatorial logic in each node, some additional increase because of the additional processor interface required, and if the two-layer Omega network of Figure B.7 is used some additional increase in the EM module to resolve conflicting accesses arriving from the two redundant networks. On the other hand, the network controls, simple enough in the baseline design, are even simpler here since most control is local to the node.

Some of the increase in size is due to the increase in width, from 9 lines in the baseline design to 12 lines here. This increase is necessary since the EM module number plus a strobe must be transmitted in parallel. However, this also would give some speed advantage; a data word being transmitted in 5 frames instead of 7 bytes.

If only one Omega network, one sheet of Figure B.7 is implemented, then some means of eliminating the bias against certain processors must be adopted. Alternating the priority on a regular cycle has been simulated, and appears to be satisfactory. The suggestions of Feierbach and Stevenson (5) when adapted to this network appear to eliminate bias more economically, but perhaps with side effects; they have not been simulated.

Another variation on the two sheet layer Omega network would be to provide, at each node, a path for data to go up or down to the corresponding node of the other sheet. To the node logic of Figure B.8, a path would be added from port B of the other node, entering into output gates X and Y, as well as a path from both X and Y of the other node, entering the outputs at port B. Port B is selected on the basis that it is the low priority port; the high priority port will always find a path on its own sheet. With a two-sheet Omega, one probably uses fixed priorities, favoring the A port on one sheet, and the B port on the other sheet. However, the hardware on both sheets can be identical because of symmetry. This variation increases the number of inputs of selection gates from ten to 14. Since data paths dominate the hardware, this is approximately a 40% increase in gate count for the entire CN to provide these additional paths.

B.5 SIMULATION RESULTS

B.5.1 Summary

Various CN configurations were simulated with the functional simulator and the simulator results were studied for various indicators of goodness of function. Test cases consisted of filling a queue of requests in each processor. In some tests all processors had requests in a given queue position, so that all 512 processors made requests. The requests in the 512 processor could form p-ordered vectors, or could be p-q-ordered, or could be random. The easiest criterion for performance evaluation is the percentage of the 512 requests that are granted on the first EM cycle. Section B.5.3 and B.7.4 go into more detail on the performance after that first cycle or when only a portion of the processors are requesting access.

Table B.2 shows, for a number of possible CN designs, this performance on the first cycle, and also lists the gate count by number of nodes and as a percentage of the gate count of the TN [1, 2].

Table B.2 Summary of Simulator and Analyzer Results for CN

Type of Transposition network	Parts Count (in nodes)	Ratio to original TN	Performance (% success)			
			Random Requests	Requests are p-ordered permutation	Stochastic Analyzer	p-q-ordered permutations (with p = 1)
TN of Ref. 1 and Ref. 2	equivalent to 3120	100%	0.2%	100%		
Benes, ports packed (Fig. B.4)	5643	180%	33%	51% p≠1 100% p=1		
Case 2. Benes, proc. spread, EM ports packed (Fig. B.5)	7947	254%	43%	50% p≠1 100% p=1		
Benes, all ports spread (Fig. B.6)	9728	312%	34%	42% p≠1 100% p=1		
Case 1. Double Omega (Fig. B.7)	10240 + 512 conflict res. + 512 port selector	361%	60%	77% p≠1 100% p=1		90%
Case 3. Single Omega, one of the two sheets of Fig. B.7, all ports spread	5120	164%	44%	44% p≠1 100% p=1	43%	76%
Single Omega, all ports packed	6670	214%	17%			
Case 4. Double Omega with inter-layer connections	10240(but more complex)	361% to 485%			87%	

The four variations of networks with the best combination of non-blocking and parts count were

Case 1. A two layer Omega Network (Figure B.7) with 361% times as much hardware as the transposition network.

Case 2. A Benes Network with processors spread (Figure B.5) with 254% as much hardware as the Transposition Network.

Case 3. One sheet of Omega network with 164% as much hardware as the TN.

Case 4. A two layer Omega network but with sheet-to-sheet paths at each node. This is estimated to take about 485% of the hardware of the baseline TN.

Table B.2 also includes results obtained with the stochastic analyzer, which computes the probability of blockage within the CN under the assumption that the input is a random request. Since the functional simulator could not handle case 4, the stochastic analyzer results are all that is available for this case. In the case where the functional simulator and the stochastic analyzer were both used, it is seen that the stochastic analyzer agrees with the simulator results for the case of random requests. For case 4, there are two outputs from the CN to the same EM module. The stochastic analyzer does not count the two conflicting requests arriving at these two ports for the same EM module as a blockage.

The data of Table B.2 is plotted in Figure B.16, where it shows the tradeoff between speed vs. amount of hardware, for the CN. Speed is represented indirectly, as percent success for the case of all processors requesting simultaneously; hardware is also represented indirectly, as a gate count, which in actuality can be only a rough guide for hardware cost. Three of the four cases previously listed show on this figure as local optima. All networks investigated have the property that a p-ordered vector with a skip distance of 1 found all 512 paths simultaneously on the first request.

B.5.2 Data

The individual simulation averages in Table B.2 are reported in more detail in Table B.3. The simulator generated either a p-ordered vector for the 512 requests, or generated 512 random numbers. Requests could be queued in the processors. In early runs, Case 1, the two-sheet Omega network was simulated by combining two successive cycles of requests of a simulation of case 3, the one-sheet Omega. When the original set of requests is a permutation containing no duplicate EM module numbers, this is correct for all cycles. However, when the original set of requests is random, corrections for multiple accesses to the same EM module must be made and only the first cycle is correctly simulated for these early results.

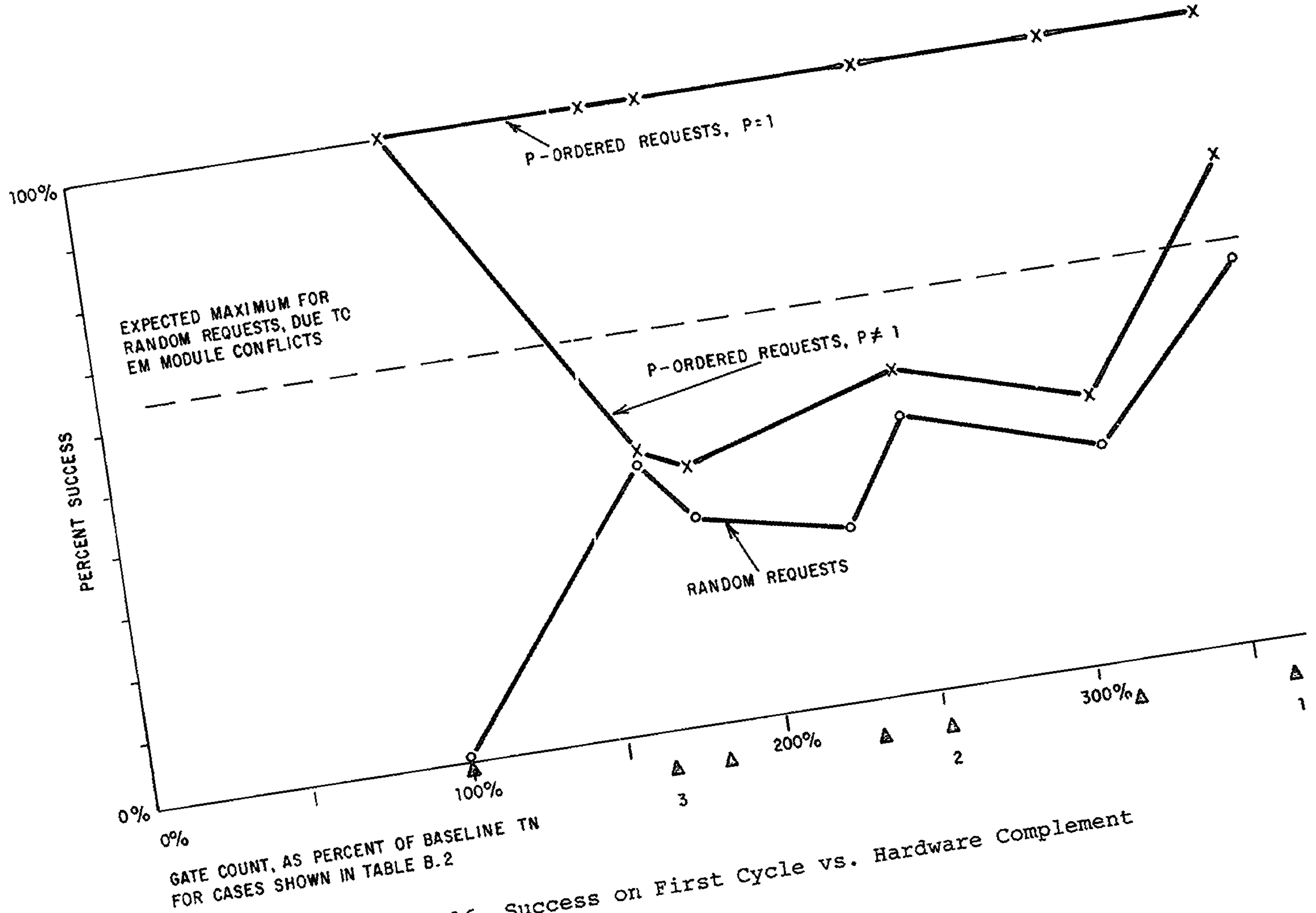


Figure B.16 Success on First Cycle vs. Hardware Complement

Table B.3   Summary of Individual Simulation Runs

Network	Parts Count	Type of Access	Offset,skip		Number of successes 1st cycle	Percent of 512 requests	Number of cycles for all 512 requests
Baseline TN	3120 equiv. (100%)	p-ordered	any	any	512	100%	1
		random			1	0.2%	512
Benes, Proc. ports packed, EM ports unpacked (Fig. B.4)	5643 (180%)	p-ordered	1	1	5.2	100%	1
			0	48	268	52.4%	4
			397	293	199	38.9%	4
			17	17	136	27.6%	5
			131	426	181	35.4%	4
			123	344	248	48.5%	4
					average..	40.6%	
		random			175	34.2%	-
					174	34.0%	-
					172	33.6%	-
					170	33.2%	-
					163	31.8%	-
					168	32.8%	-
					173	33.8%	-
					177	34.6%	5
					average..	.33.5%	
Benes, Proc ports unpacked EM ports packed (Fig. B.5)	7947 (254%)	p-ordered	123	344	325	63.5%	3
			356	197	241	47.0%	7
					average..	.50.2%	
		random			213	41.6%	6
					228	44.5%	5
					average..	.43.0%	
Benes, all unpacked (Fig. B.6)	9728 (312%)	p-ordered	246	179	215	42%	4
		random			176	34.4%	-
Double Omega, all ports unpacked (Fig. B.7)	11264 (361%)	p-ordered	123	344	431	84.2%	3
			246	179	444	86.6%	3
			17	17	333	64.9%	3
			0	228	372	72.7%	2
					average..	.77.1%	
		random			308	60.2%	-
					287	59.0%	-
					average..	.59.6%	
		p-q-ordered		= 31	438	85.6%	-
		with p=1		= 31	465	90.9%	2
		length of		= 31	443	86.6%	-
		piece		= 100	508	99.3%	2
				= 100	460	89.9%	2
					average..	.90.5%	

Table B.3 Summary of Individual Simulation Runs (Cont'd.)

Network	Parts Count	Type of access	Offset,skip		Number of successes 1st cycle	Percent of 512 requests	Number of cycles for all 512 requests
Single Omega All unpacked	5120 (164%)	p-ordered	123	344	257	50.2%	5
			246	179	273	53.4%	5
			17	17	176	34.4%	5
			0	228	200	39.1%	4
					average..44.3%		
		random			227	44.4%	6
					218	42.6%	6
					average..43.5%		
		p-q-ordered		= 31	382	74.6%	-
		with p=1,		= 31	412	80.4%	-
		length of		= 100	436	85.4%	-
		piece		= 100	435	63.5%	-
					average..76.0%		
Double Omega All packed	7191 (230%)	random			143	27.9%	-
Single Omega All packed	3083 (99%)	random			87	17.0%	12

The stochastic analyzer data is summarized in Table B.4. The output of the stochastic analyzer produced detailed description of the blockage of the network at each level, giving a probability of a requests blocked at each level. Only the totals are shown here. In addition, the number of processors making a request was varied from 256 (50% of the processors) to 512 (100% of the processors). The body of the table gives the probability of a request being blocked within the CN, and hence the fraction of requests that are expected to be blocked.

In the case of the single Omega, any EM module conflict will result in blockage within the CN. For the double Omega up to two requests for the same EM module may show up at the output port. The stochastic analyzer does not count this as blockage.

In addition to the actual 512-processor/521-EM module case, the stochastic analyzer was run for curiosity on a number of other sizes of arrays, to investigate sensitivity to the exact number of processors and EM modules. These are also listed.

All the data of Table B.4 is plotted in Figure B.17.

B.5.3 Discussion of Simulation Experiments

P-ordered requests had considerable variation in the percentage of success, in the functional simulator, as a function of the skip distance p. This constrasts with the behavior of random requests, whose behavior was nearly uniform and independent of the seed for the random number generator. Almost all values of p produce p-ordered vectors whose percent of requests granted is substantially better than for random requests.

Certain skip distances (including p = 1) are "magic", in all that EM accesses are attained on the first cycle, with no interference. Figures B.18 and B.19 show the distribution of percentage success over the various skip distances tried for two of the networks. The experiment has a defect; skip distances were not selected at random, but were partly picked on hunches that said they would be "magic", with high success rate, or "perverse", with low success rate. p = 17 was expected to be perverse and it was. p = 228 was expected to be "magic" (228 is the reciprocal of 16 in modulo 521 arithmetic) but it was not.

In normal operation, processors are not spending full time accessing EM, but are spending most of their time doing other things. Furthermore, since they are processing independently of each other, processor requests will often get out of synch with similar requests in other processors. Therefore, a question of interest is to what degree does the blocking in the CN become less as the percentage of requests is less. Two methods were used to investigate this question. First, the stochastic analyzer could be run with the probability of a request being issued from a given processor set to various values. These results are shown in Figure B.20. These results are for single Omega (case 3), and

TABLE B.4

Stochastic Analyzer Data, Fraction of Requests Blocked

Network Version	Case, Proc/EM mods	Percent Processors Requesting Access					
		50%	60%	70%	80%	90%	100%
Single Omega	512/512	0.3966	0.4420	0.4809	0.5152	0.5442	0.5695
	512/521	0.3950	0.4403	0.4792	0.5134	0.5424	0.5677
	512/1024	0.3699	0.4142	0.4525	0.4865	0.5156	0.5411
	1024/512	0.5896	0.6296	0.6599	0.6821	0.6981	0.7085
	1024/1024	0.5660	0.6070	0.6385	0.6618	0.6789	0.6904
Double Omega	512/512	0.0339	0.504	0.0692	0.0902	0.1121	0.1348
	512/521	0.0332	0.0494	0.0679	0.0886	0.1102	0.1326
	512/1024	0.0271	0.0406	0.0560	0.0734	0.0916	0.1108
	1024/512	0.1471	0.1995	0.2435	0.2888	0.3317	0.3711
	1024/1024	0.1270	0.1669	0.2131	0.2545	0.2943	0.3315

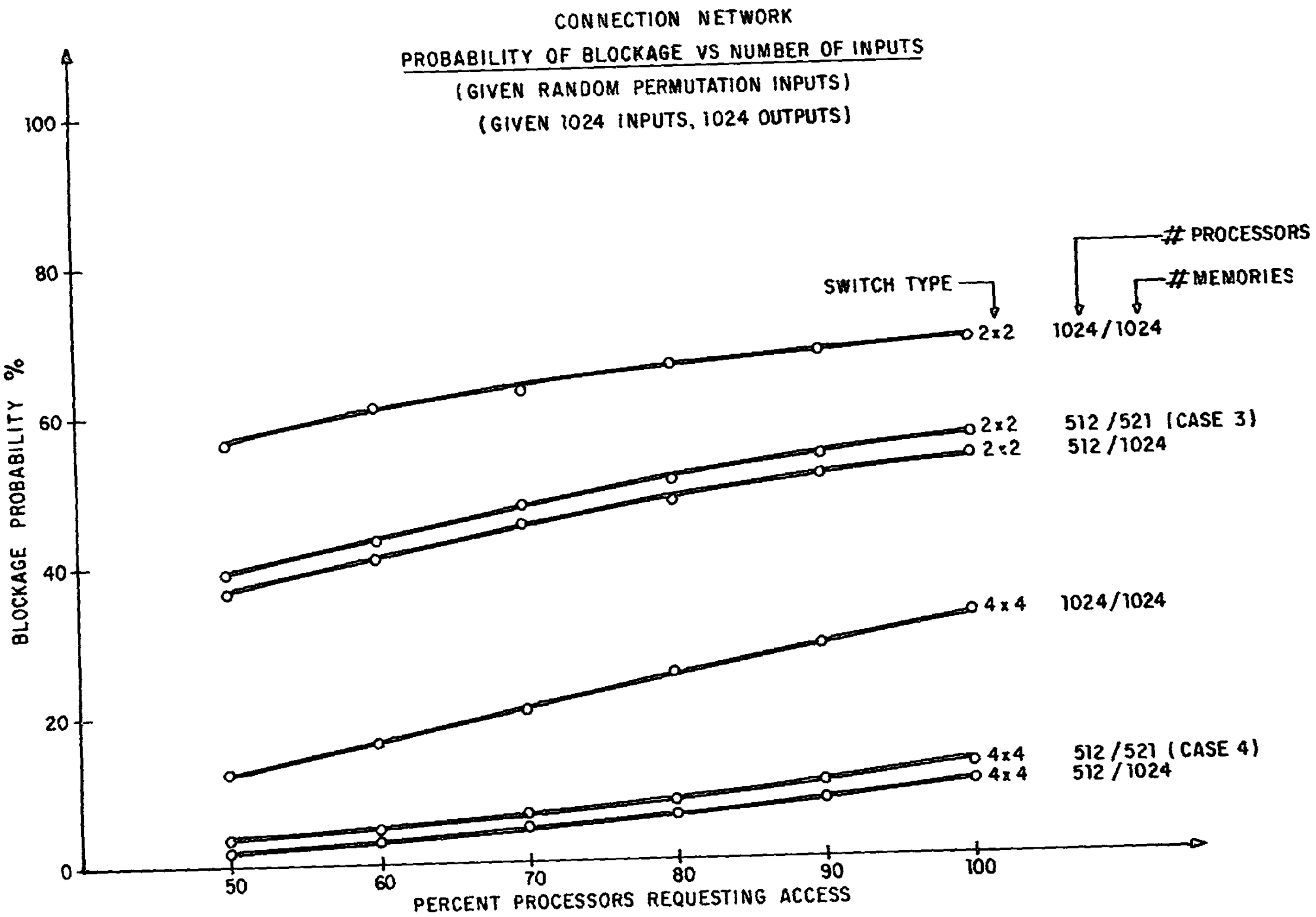


Figure B.17 Response to Full and Partial Loading, Stochastic Analyzer Data

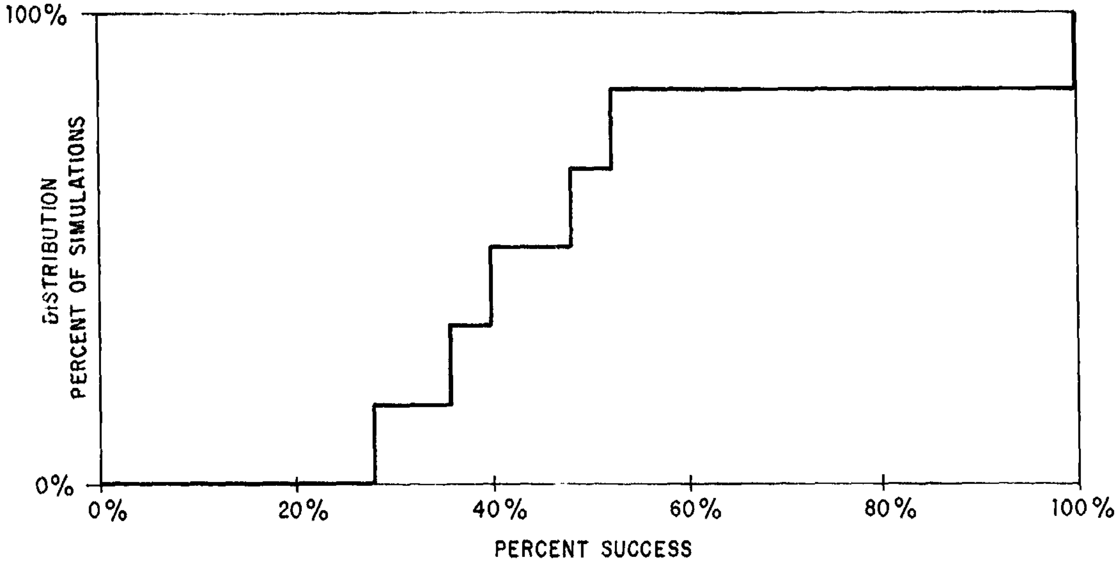


Figure B.18 Distribution of Percent Success for p-ordered Vectors, Benes Network

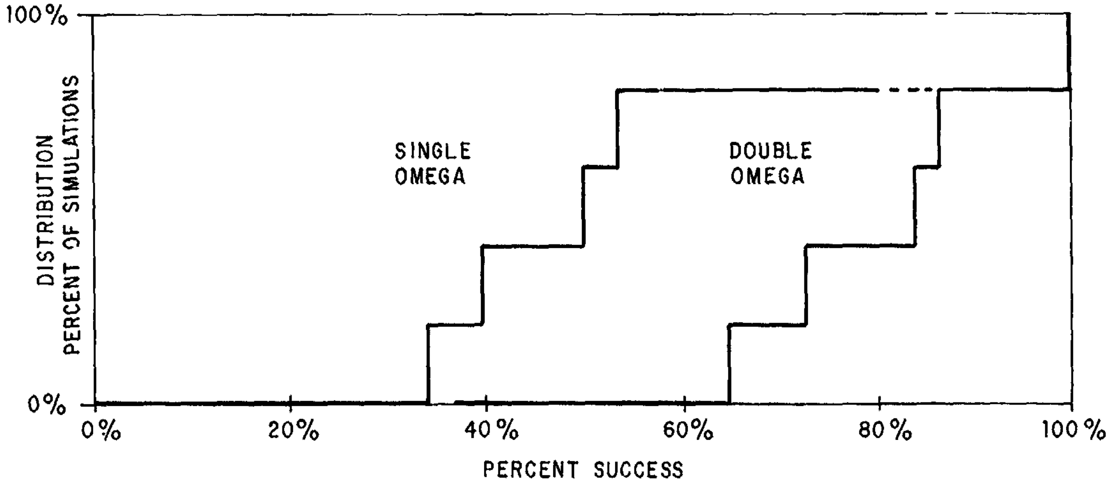


Figure B.19 Distribution of Percent Success for p-ordered Vectors, Omega Network

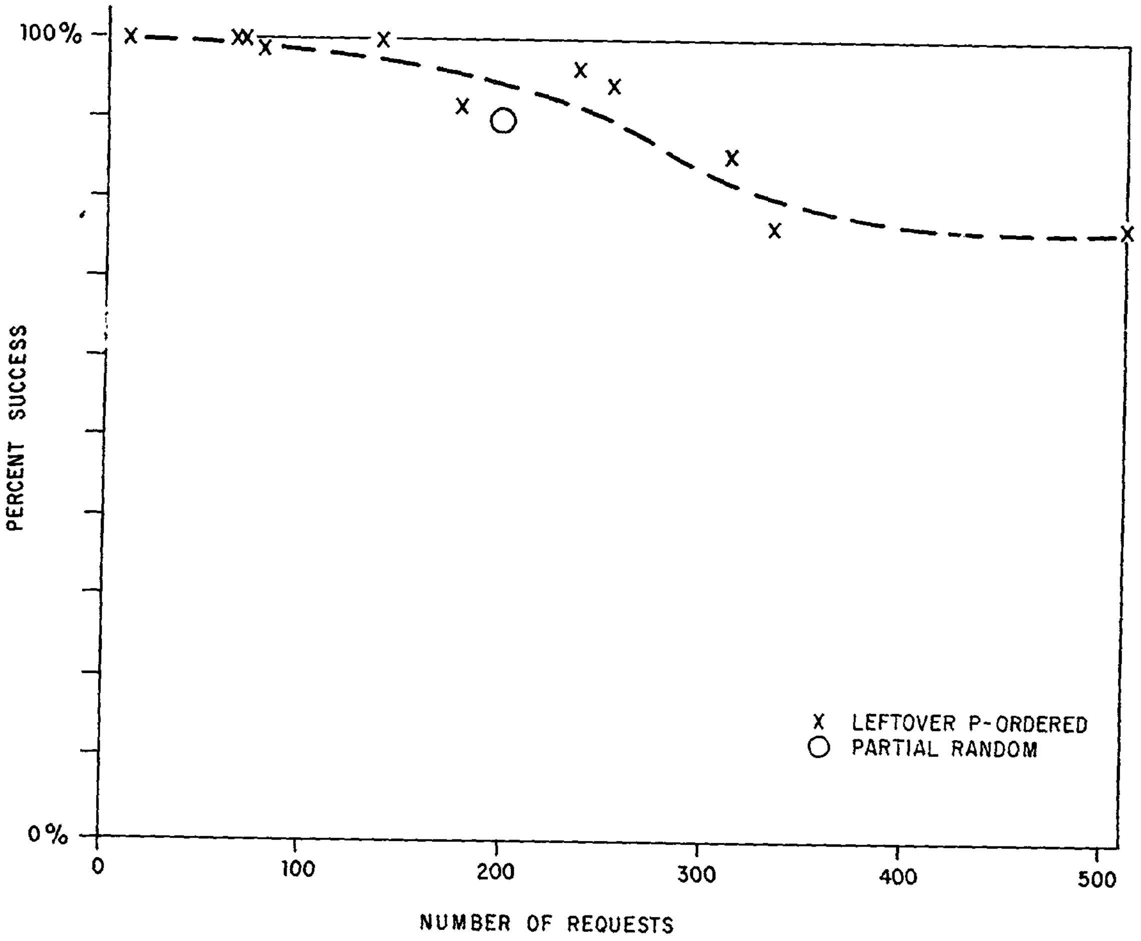


Figure B.20 Response to Partial Loading, Case 1

double Omega with interlayer data paths (case 4), plus some non-realistic cases. The second result was by simulation with only some portion of the processors having CN requests. Most of the results of the second method were obtained with an early version of the simulator which could not be initialized to fewer than 512 requests. However, the simulator did keep retrying all requests that failed to be satisfied on the first EM cycle. Hence, these leftover requests can be used to estimate the response of the CN to the situation that only a portion of the processors are making a request. Figures B.20, B.21 and B.22 are these data for the double Omega (case 1), the Benes (case 2), and the single Omega networks (case 3) respectively. Data points derived from requests leftover from p-ordered vectors are marked with X; points representing leftover requests from originally random requests are marked with dots; and points marked with circles are cases that were run with partial random requests after the functional simulator was improved so as to initialize the processor request queues for an arbitrary number of processors less than 512.

B.5.4 Test Cases Abstracted From the Aero Flow Codes

In two directions of accessing, the aero flow codes produce p-ordered vectors as access requests. In the third ("hard") direction, a p-q-ordered vector is produced. A full-scale implicit might have dimensionality (100, 50, 200) leading to p=1 and q=4900=211 modulo 521. The explicit code as supplied (a small-mesh test case) has dimensionality (31, 31, 31) leading to p=1 and q = 931=410 modulo 521. Several test cases were run using the double Omega, Case 1, which by that time was targeted as the CN most likely to be recommended, and the results are shown in Table B.3, where they are called "p-q-ordered with p=1". The first sheet of simulator printout, on the first CN clock, gives a printout for the first layer which is identical with the first clock of a single Omega, hence, this data is also listed in Table B.3.

B.6 SELECTION AMONG THE CN ALTERNATIVE APPROACHES

Four preferred approaches to Connection Network (CN) implementation were listed in section B.5.1. The arguments presented below show that the double Omega network (case 1 or case 4) is preferred. Trade-off studies between these two cases are incomplete. Table B.5 compares the characteristics of the four preferred cases.

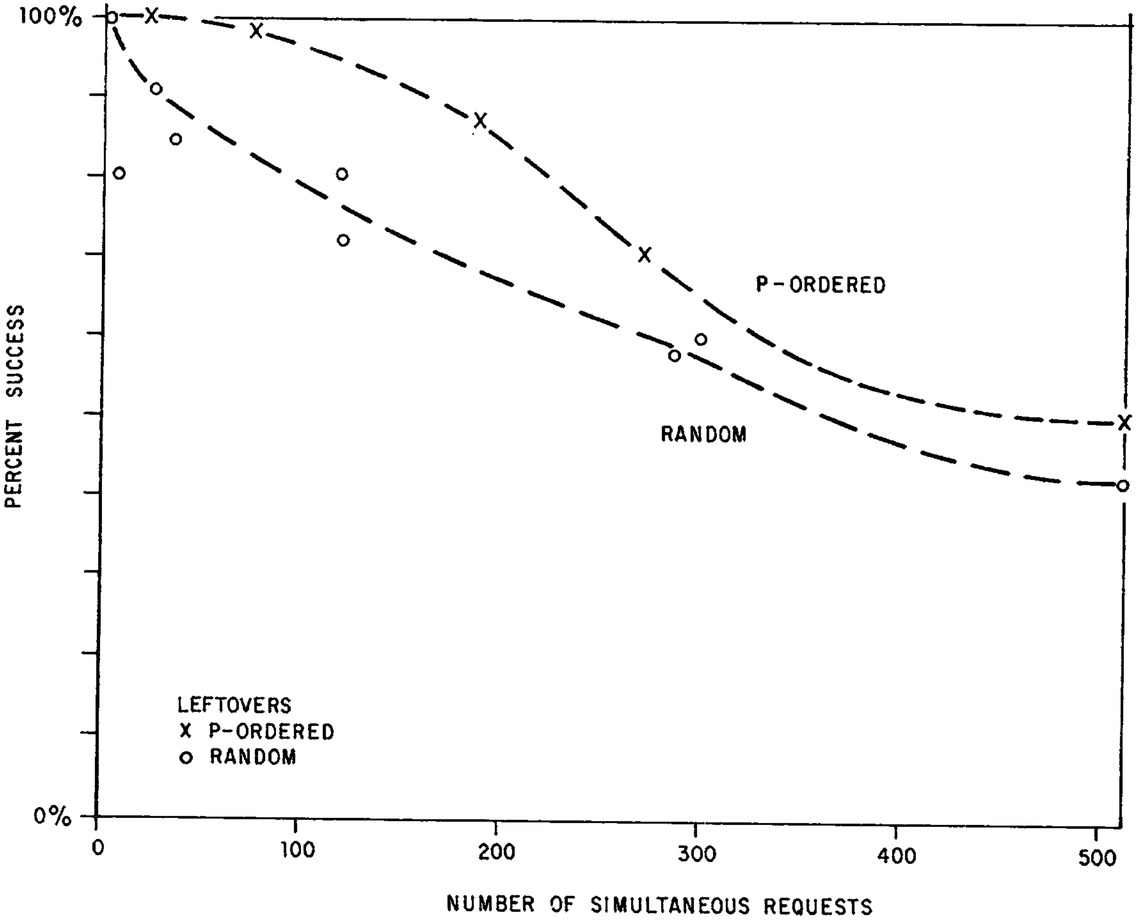


Figure B.21 Response to Partial Loading, Case 2

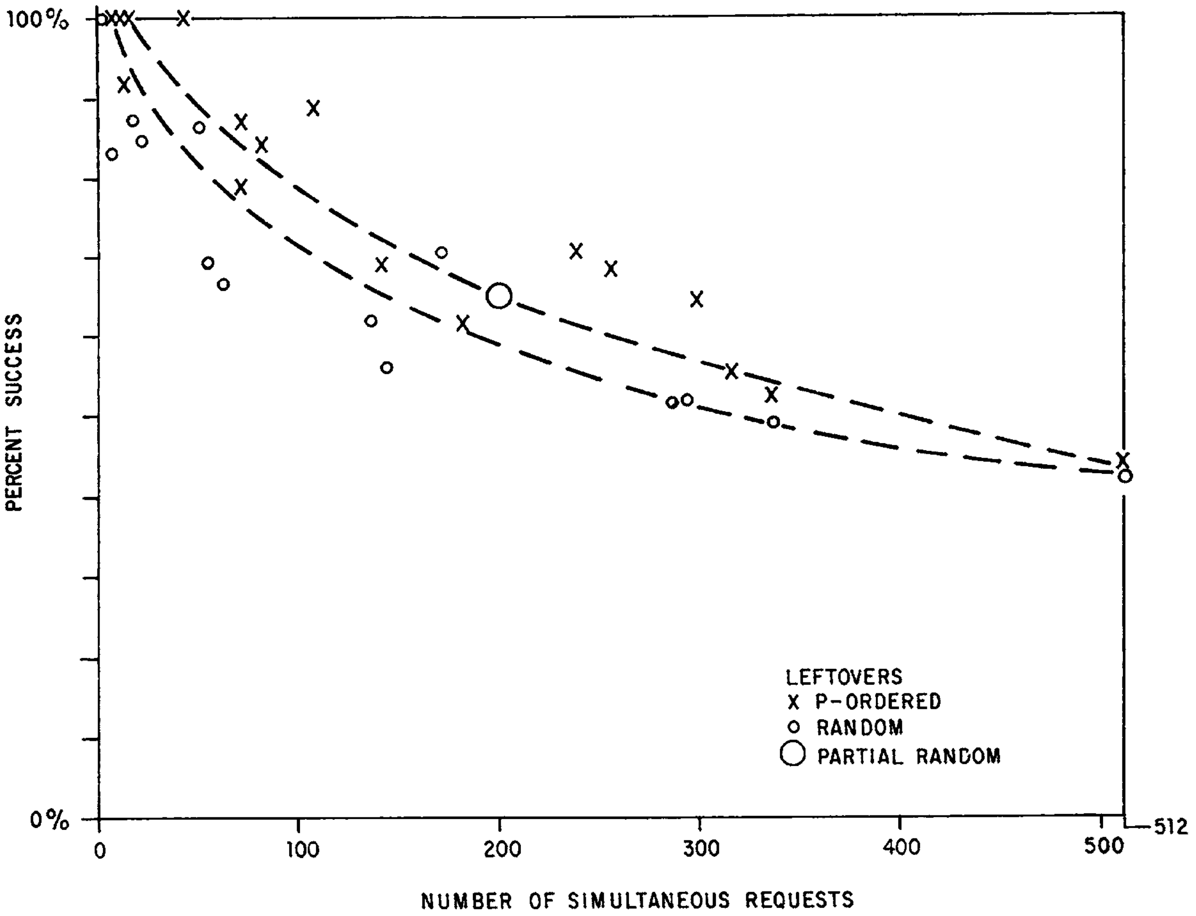


Figure B.22 Response to Partial Loading, Case 3

TABLE B.5

Comparison of Connection Network Options

Item		Network Version			
		Case 1 Double Omega	Case 2 Benes	Case 3 Single Omega	Case 4 Double Omega Interlayer Conn.
Hardware, compared to baseline system		361%	254%	164%	505%
Random Success		59.6%	43.0%	43.5%	
P-ordered Success		77.1%	50.2%	44.3%	
p = 1 Success		100%	100%	100%	100%
p-q-ordered success with p=1		87.6%			
non-blockage from stochastic analyzer				43%	87%
Average No. EM Cycles		1.236	1.546	1.86	
Calc. Percent Success		80.9%	64.8%	53.8%	
N_{max} Calculated		3.77	5.98	8.16	
N_{max} observed	p-ordered	3	3 to 7	4 to 5	
	random		5 to 6	6	

In Table B.5, the "Hardware" column compares the gate count of data carrying gates of the various versions with the corresponding gate count of the Transposition Network considered during the Preliminary Study [1, 2]. This comparison is used since package count is subject to uncertainties of packaging, suitable part availability, etc.

"Random Success" is, for sets of 512 simultaneous random EM access requests the average percentage of that were serviced on the first EM cycle.

"P-ordered Success" is the corresponding average percentage for sets of 512 p-ordered requests. Substantial variation in this percentage from one set to another was observed, although performance was consistantly better than it was for random requests. The percentage given does not include p=1, the simple vectors, for which the success percentage is always 100%, as the next line reminds us.

"P-q-ordered success" gives the percent success observed with so-called "P-q-ordered" vectors, in which the module numbers come from the set M_i = (i*p+(iDIVk)*q)mod 521. The value of p was always 1 in the test cases, which come from actual aero-flow codes.

B.6.1 Discussion of Results

The data in Table B.3 comes from a simulator which makes 512 simultaneous requests of the EM modules. In actual programs, this is expected to happen only on the first cycle of the DOALL on the first EM access. Once some processor has been delayed on accessing EM memory, it will no longer be in synch with the access requests of other processors, and so the system should be self-regulating for all but the shortest DOALLs, with an effective delay controlled by the average access time observed when some fraction of all the processors are requesting access to memory.

Consider a program that averages five floating point operations per EM access (for example, the 2D version of the explicit code, according to the Preliminary Study) [1, 2]. Each EM access ties up the CN an estimated four CN clocks of 120 ns each, if the success rate were 100%. Five floating point operations in 512 processors will take at least 1471 ns so that the CN would have 162 requests pending on the average at any given time. Figure B.20 shows that the percent success with case 1 the double Omega, is nearly 100% at this level of loading. Figures B.21 and B.22 show about 80% success at 210 requests loading for case 2 the Benes Network, and 60% success at 270 requests for case 3, the single Omega Network. The 162 requests are 42.4% of maximum loading for case 1, 210 requests are 65% of the maximum loading for case 2 Network and 270 requests are 75% of the maximum loading for case 3.

None of the cases carries a hardware cost greater than about one third that of the set of 516 processors. The paragraph above shows that the simple double Omega (case 1) can handle programs with as few as five floating point operations per EM access and still have margin to accommodate bursts of EM access. Such bursts are planned; we expect a flurry of fetching from EM at the beginning of many DOALLs and another shorter burst of stores to EM is expected at the end of many DOALLs.

The double Omega Network with interlayer paths (case 4) is even better than the case 1 double Omega at not blocking. Unfortunately, modification of the CN simulator to include case 4 would have been a major effort, and was not done in time for this report. Hence the evaluation of this network is incomplete.

The choice between case 1 and case 4 both double Omega Networks, must take into account a number of other factors if the choice is to be optimized. Among these are:

* Characteristics of the applications programs. The four benchmark programs represent only four points in applications space. The main characteristic of interest here is the number of EM accesses, and their distribution in time.

* Relative cost of the two versions. The gate count is in the ration of 1.4:1, case 4 with interlayer connections having the more gates. If the CN chips turn out to be strictly pin- limited, the extra gates may not cost much at all.

* Ease of diagnosing hard failures. In the simple two-layer network diagnostics are straight forward, since each single Omega network, tested separately, is easy to diagnose, as shown in the Chapter 6 of this report. More complex hardware controls are needed to make the more complex version as easy to diagnose.

B.7 ADDITIONAL CONSIDERATIONS

The remainder of this appendix considers an assortment of various behaviors of the CN and aspects of EM accessing. These include references to or discussions of

* Modular partitioning
* Mapping of module number to CN port number and sparing
* Processor-to-Processor transfers
* EM module conflicts for p-q-ordered vectors
* Approximate validity of the assumption of random EM module numbers when EM accesses are queued within the processors.

Appendix H contains brief discussions of the CN simulator and of the stochastic analyzer respectively. Listings of the CN simulator (prior to the insertion of the capability of testing p-q-ordered requests) and of the stochastic analyzer have been provided to NASA Ames.

Appendix I contains an analysis of the connectivity of various networks which was performed soon after CN considerations began.

B.7.1 Modular Partitioning

Note the division of an Omega network (Figure B.23 is a 16 x 16 Omega network) into distinct upper and lower halves after the first level of nodes, and into quarters after the second. It is expected that after the second level of nodes. identical quarters can be put into each of the four EM cabinets. Thus, the CN would not physically exist as a single central item except possibly for the first two levels of nodes.

B.7.2 Mapping

Mapping is described in adequate detail in Chapter 5, and need not take much space here. The probable mappings are as follows:

The CN ports on the processor side are numbered 0 to 1023. The first seven bits plus the least significant bit will be called CN-port-within-cabinet. The two intervening bits are the cabinet number. Within the cabinet, the processors are numbered 0 through 128, including the spare. Processors 0 through 127 are assigned port numbers as follows; reverse the processor number end for end, least significant bit to most significant bit position, and vice versa, multiply this result by 2. The result at this point is CN-port-number-within-cabinet. Processor 128 is assigned to CN-port-number-within-cabinet No. 1. All others have even numbered ports.

The CN ports on the EM module side are numbered from 0 to 1023. There are 525 EM module slots, and hence, 525 CN port numbers to be assigned. EM module numbers 0 through 511 are assigned to the even port numbers from 0 through 1022 respectively. The additional 16 slot numbers are assigned four per cabinet as shown in Equation B.6.

$$\text{CN Port No.} = 32 \times (\text{EM No. modulo } 512) + 1 \text{ for EMno} > 511. \quad \text{(B.6)}$$

On the EM side, the most significant two bits of port number are the cabinet number.

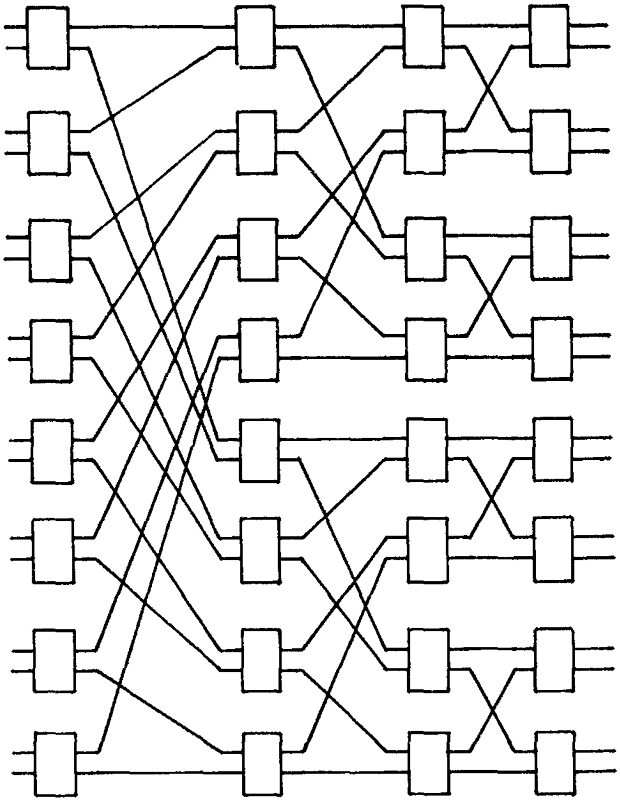


Figure B.23   16-Wide Omega Network

Sparing of EM modules would be accompanied by replacing a reference to a failed EM module with a reference to one of the spares (numbered 521 through 524). The remapping of such a reference would occur in the CN buffer. The remapping carried out in the CN buffer would change up to four EM module numbers from their normal CN destination port number assignments to the CN destination port number for the spares. Sparing of processors is done by designating one as spare, whereupon all processors whose physical location numbers are higher than the physical location of the spare within the same cabinet, interpret physical location minus one as their processor number.

B.7.3 Processor to Processor Transfers

SHIFCN using "wraparound" in the simplest way is effective only when processor in physical location 128 in each cabinet is designated as the spare. The "wraparound" command, as described, makes connection between two CN ports whose numbers differ only in one bit position. Although the positions of the bits in a CN port number are different than the positions of the bits in a processor physical location, they are the same bits, rearranged (swapped end for end and shifted by one). Thus, to get bits of processor number to correspond to bits of CN port number, we must have the processors in the first 128 out of the 129 physical slots.

Thus, some modification to the simple "wraparound" described in the previous sections is called for in order to accomodate both sparing and the SHIFCN instruction. The SHIFCN instruction is not used anywhere in the aero flow or weather codes except as part of the SUMALL function. In SUMALL, since the use of SHIFCN is hidden inside system software, deficiencies of SHIFCN could be avoided by programming. However, the SWAP function will require either a solution to the SHIFCN problem exposed above, or else a store to EM followed by a fetch from recalculated addresses.

B.7.4 EM Module Conflicts on p-q-ordered Vectors

Failure to access all 512 memory words in parallel can be due just as much to request conflicts, where several processors are trying to access one memory module, as to CN blockage. Case 3, the single Omega has the property that all EM module conflicts are eliminated by a CN blockage that occurs somewhere within the CN. These Blockages that resolve conflicts should not be blamed on a CN inadequacy, since even a perfect CN will not eliminate the original conflicts.

Depth of conflict, or "pileup", is defined as the number of processors requesting the same EM module on one CN cycle. Pileup is not to be confused with the queues of requests within the processor, which could conceivably contain even more requests for the same EM module, but these would not come to light until some later CN clock.

P-q-ordered vectors occur frequently in the aero flow codes (in the "hard" direction). When arrays are placed in Extended Memory with successive elements in adjacent EM modules and when the processors are each accessing an element of a p-q-ordered vector concurrent with all the other processors, then EM conflicts of the sort just described can occur. This situation is discussed below.

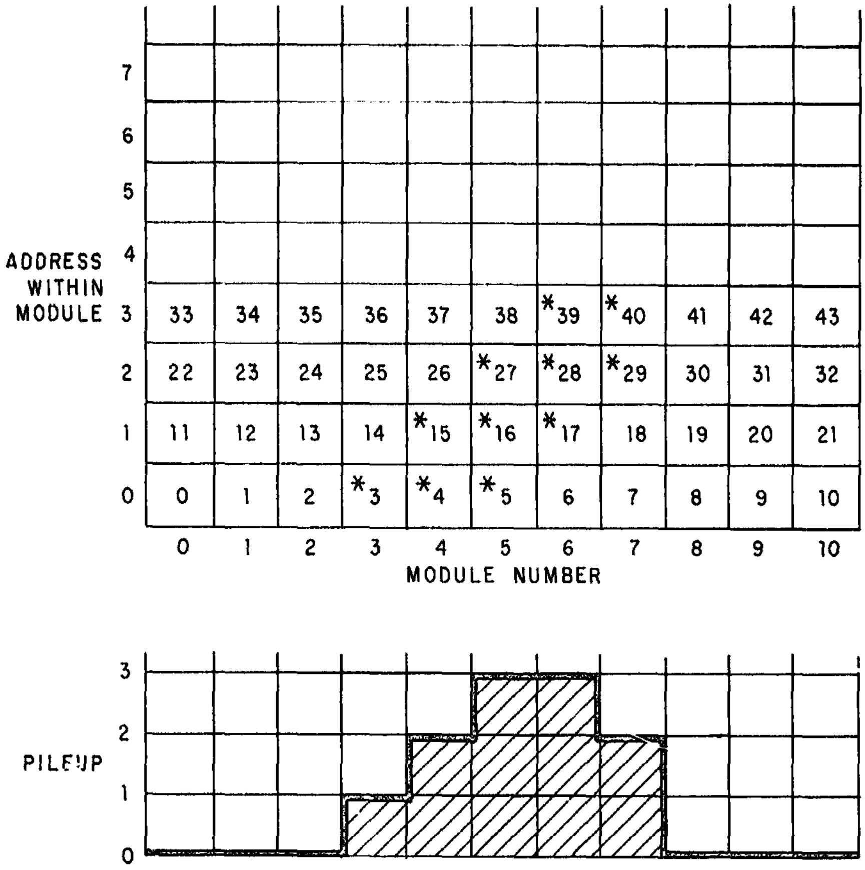
Table B.6
Worst "p-q-ordered" cases

Array Dimensons	Pileup	Array Dimensions	Conflict Depth
20 x 26	20	29 x 18	18
26 x 40	13	29 x 36	14
39 x 40	13	34 x 46	16
42 x 62	13	41 x 89	13
50 x 73	11	45 x 81	12
43 x 97	12	49 x 85	11
34 x 23			

Since any array size declaration picked at random is not likely to be one of the bad cases, and since the bad cases are all smaller than the problem sizes for which the FMP is targeted, the problem would appear to be a minor one, of the sort most conveniently handled by having the compiler issue a warning to the programmer when one of the bad cases is seen. The depth of conflict can never be more than the number of p-ordered pieces in a p-q-ordered vector, since the p-ordered pieces never have conflicting access internally.

In Table B.6, the number of conflicts may be different depending on the order of M, N. Usually, array dimensions (M, N, X) where M is less than N, have more conflicts than (N, M, X). In the table, the worst of the two cases is listed.

Figure B.24 shows an example of the pileups that occur when an adverse p-q-ordered vector is accessed from a smaller number of EM modules. For example, the number of modules and the number of processors are both 11. The vector being fetched is M_i = (3 + 1*i + 9*(i DIV 3)) modulo 11 for 0 i 10. The top portion of the figure shows the address space in these 11 modules, plotted within the two-dimensional representation based on module number vs. address within module. The addresses being accessed are marked with an asterisk. The lower part of Figure B.24 shows the resulting pileups. In this case, the worst pileups are of depth three at module numbers five and six.



ADDRESS WITHIN MODULE	0	1	2	3	4	5	6	7	8	9	10
7											
6											
5											
4											
3	33	34	35	36	37	38	*39	*40	41	42	43
2	22	23	24	25	26	*27	*28	*29	30	31	32
1	11	12	13	14	*15	*16	*17	18	19	20	21
0	0	1	2	*3	*4	*5	6	7	8	9	10

Figure B.24 Example of Pileup

If q plus length of piece is nearly equal to 521, then the successive pieces of vector will tend to coincide in the same EM modules, generating substantial conflicts. When an array has a dimensionality (M N, X), and the DOALL is on the first and third subscript, the result is a p-q-ordered accessing of that array with p=1 and q+K=M(N-1). All numbers that are close to multiples of 521 which can be factored into an M and an N that are within a factor of two of each other were surveyed. The depth of conflict in the most-accessed EM module for each of these cases was computed. Pileups can also occur when MN is close to 260 modulo 521. Out of all possible pairs of numbers M, N that lie within the above range, exactly fifteen pairs generate EM module conflicts that are 10 deep or more (listed in Table B.6). The worst case is M, N = 20, 26 which yields a depth of conflict of 20 in six memory modules, and which takes 26 cycles of accessing to resolve, as shown by simulation.

B.7.5 Non-Randomness

Given a random set of EM module numbers as a request, there will be conflicts at some of the memory modules. After the first cycle of satisfying the requests has occurred, the memory module with an N-way conflict will still have an N-way or (N-1)-way conflict. Hence, if there is a succession of random requests for memory in the processors, the leftover requests will tend to bunch up to some configuration that is worse than a random request. In order to test this effect, a test case was run with all 512 processors each having a queue of three random requests. The case 1 double Omega network, was used for simulation purposes.

Tables B.7 and B.8 trace the history of this test through the 12 EM cycles that it took to satisfy all processors. For each cycle Table B.8 gives the number of processors requesting memory, the number of memory modules over which such a request is expected to fall (by ref. 1), the smaller number of memory modules that the bunched-up requests actually asked for, the percentage the number of memory modules actually reached (any difference between the second and third is due to conflicts in the CN), the percentage of non-blocking in the CN), and finally, the length of the longest pileup observed. Table B.7 gives the history of memory module conflicts per cycle for this test. Cycle 11 included one processor that was requesting the second item in the processor's queue of three items. This lone processor's third item constitutes cycle twelve.

TABLE B.7 Pilueup History

Number of EM Modules with Specific Conflict Depths

Cycle	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7	Depth 8
1	197	90	31	4	4	1	0	0
2	182	89	26	13	3	0	1	0
3	199	76	30	8	3	4	0	0
4	165	59	19	6	4	0	1	0
5	137	36	13	7	0	1	0	0
6	99	30	10	1	1	0	0	0
7	74	17	4	0	1	0	0	0
8	52	6	0	1	0	0	0	0
9	20	2	1	0	0	0	0	0
10	9	1	0	0	0	0	0	0
11	3	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0

From Table B.8 one can see that on Cycle 1 there were 512 requests, and that, if purely random, one should expect these requests to involve 327.2 EM modules on the average. There were 327 memory modules in this first cycle, whose requests come directly from the random number generator. In subsequent cycles, there are always slightly fewer EM modules being requested than one should expect if the number of processors requesting were issuing random requests. At cycle 5, there are only 194 different memory modules in the 282 requests being issued by 282 processors, whereas if those 282 requests were random, one expects 217.9 different memory modules to be named. This is the worst bunching of requests seen in the whole run.

Whether these results are statistically significant was not analyzed; they might be within the normal range of random variations. Whether significant or not, the indication is that the expected bunching effect is fairly small.

B.7.6 Redundancy

The double Omega, case 1, network has the propery that either half can be disconnected from the system under coordinator control. This feature is provided to increase system availability, since the double Omega with one of its networks turned off is precisely the single Omega, case 3, and will support FMP program execution, but at some increase in effective EM access time.

TABLE B.8

Investigation of Non-Random Effects

Cycle	No. Proc. Requesting EM access	Expected No. EM modules	Actual No. EM modules	% Reduction due to Punching	Accesses Granted	Success Rate (% of possible)	Longest Pileup	Average Pileup
1	512	327.2	327	99.9%	295	90.2%	6	1.57
2	512	372.2	314	96.0%	273	86.9%	7	1.63
3	512	327.2	320	97.8%	272	85.0%	6	1.60
4	391	275.2	254	92.3%	214	84.3%	7	1.54
5	282	217.9	194	89.0%	164	84.5%	6	1.45
6	198	164.9	141	85.5%	133	94.3%	5	1.40
7	125	111.2	96	86.3%	91	94.8%	5	1.30
8	68	64.7	59	91.2%	57	96.6%	4	1.15
9	27	26.3	23	87.5%	23	100.0%	3	1.17
10	11	10.9	10	91.7%	10	100.0%	2	1.10
11	3	3.0	3	100.0%	3	100.0%	1	1.00
12	1	1.0	1	100.0%	1	100.0%	1	1.00

B.8 CONCLUSION

The study has shown that the double Omega Network (case 1 of this discussion) can be expected to give the required performance at reasonable cost. Its performance has been validated by simulation and analysis. Various options giving either higher performance or lower cost have also been presented. Additional options were considered during the course of this study, but were omitted from this discussion in order to avoid digressions.

Although sufficient study has been completed to give confidence in the feasibility of the Connection Network in the FMP architecture, cost/performance trade-offs deserve to be further considered.

REFERENCES

1. Final Report, NASF Preliminary Study, contract NAS2-9456, Burroughs, Oct. 77.

2. Final Report, NASF Preliminary Study extension, contract NAS2-9456, Burroughs, Feb., '78.

3. Benes, V.E., "Optimal Rearrangable Multi-stage Connecting Networks, Part 2," B.S.T.J. 43(1964) 1641-1656.

4. D. H. Lawrie, "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers, C-24(1975) 1145-1155.

5. G. Feierbach and D. Stevenson, "A Feasibility Study of Programmable Switching Networks for Data Routing", IAC Phoenix project memorandum 003, May '77.

6. D. C. Opferman and N. T. Tsao-Wu, "On a Class of Rearrangeable Switching Networks", BSTJ, May-June 1971, Vol 50, #1.

7. P. J. Willis, Derivation and Comparison of Multiprocessor Contention Schemes, Computer and Digital Techniques, Vol. I, No. 3, Aug 1, 1978.

8. J. Lenfant, "Fast Random and Sequential Access to Dynamic Memories of any Size", IEEE Transactions on Computers, Sept 77, Vol C-26, No.9.

9. C. Wu and T. Feng, "Routing Techniques for a Class of Multistages Interconnection Networks", Proc. of the 1978 International Conf. on Parallel Processing, Wayne State U, pub. by IEEE Computer Society, 1978.

APPENDIX C

INSTRUCTION SET AND TIMING INFORMATION

C.1 INTRODUCTION

The instruction set has undergone substantial refinement since the instruction set of the Preliminary Study [1,2]. Additional functions have been identified, including the necessity for hardware double precision, a "read with lock" operator in Extended Memory, additional operators for the system software, and so on. The unsynchronized CN has required substantial changes in the operators that access Extended Memory, including the addition of a MOD 521 operator in every processor, and the elimination of the CN controls from the coordinator for EM accessing.

One set of processor instructions is known to be necessary, namely a set of operators to allow formatting of output, and unformatting of input. These have yet to be specified. Insofar as the instruction set presented here still does not have them, it is incomplete. In evaluating the processor against the banchmark aero flow codes and weather codes, these character-manipulating operators are not needed, even though they will be needed in a final design.

Table C.1 is a listing of the instructions. It is divided into three sections. Processor instructions are in the first. Commands issued by the coordinator and effected in the processor are the second. Coordinator instructions are the third. Since every processor is a serial scalar processor, and can execute scalar code on data residing in EM, no separate 513th "scalar" processor is any longer required, nor are the "scalar unit" instructions of the baseline system any longer included. Hence, no floating point instructions are listed for the CR. If floating point requirements become identified in the system software that executes on the CR, there will have to be floating point capability included in the CR.

Table C.2 at the end of this appendix is a list of the timing of the instructions. The format used is similar to that used in the Preliminary Study [2], except that the instruction descriptions have been moved to Table C.1.

C.2 DESCRIPTION OF TABLE C.1

The Table C.1 is a description of the complete instruction set. A buffer register interfaces the CN, and that the buffer can hold an address and a word of data for a STOREM, or accept a word of data from EM on a LOADEM, without interfering with, or requiring assistance from, whatever instruction is in the processor. Hence, instructions which access this buffer must be able to test whether it is "busy", dedicated to an uncompleted LOADEM or STOREM, and whether or not it is "full". To a large extent, these tests on "full" and "busy" replace the waiting for "go" in the baseline system of the instruction set. For example, a STOREM, having told this buffer to empty itself at the designated memory module, need not wait for anything more to happen, but the next instruction may start immediately.

The list has been simplified by using condensed notation. A "(L,M)" following a mnemonic means that either L or M can be appended to the mnemonic to create other instructions in which the designated operand can come from memory, or is literal, instead of register two. Likewise, instructions with almost identical descriptions will be combined into a single description.

An "F" prefix designates a floating point operation using floating point registers, "I" designates an integer operation using integer registers, and "C" is the coordinator, using the integer registers in the coordinator.

The symbol "&" designates concatenation. "Next" designates the register next after the designated one. Names in quotes are specific control bits. "→" designates that the data just described is to be inserted into the location designated just after.

Major changes from the Preliminary Study [2] are listed in the following paragraphs.

Most synchronizations are put onto the CN buffer so that individual instructions are not held up waiting. "I got here" is set by one instruction, and then usually tested at some later time to see if "go" reset it, although WAIT and LOOP still wait for "go". LOADEM and STOREM with the new CN are completely free of any synchronization requirements, thanks to the CN buffer.

The instructions by which the coordinator causes diagnostics to be imposed on the processor are more complete in this list than previously.

The data path directly from coordinator to processor through fanout boards, of ref. 1 and 2 has been eliminated. Instead, the coordinator has been given access to a CN port, which can be then set to a "broadcast" condition where it connects to all processor parts in parallel. The control path from coordinator to processors remains.

Double-precision floating point has been included. Double-length format is two words in single precision format, with an exponent difference of 36, and with the second word not necessarily normalized.

Several corrections, such as incrementing before testing in ITIX and CTIX, also make these instructions differ from the previous description.

C.3 MICROPROGRAMMABILITY

Burroughs, on its own funds, has been building an evaluation model of a processor similar to the single FMP processor (see Appendix E of ref. 1). This exercise shows that the preferred implementation, even for a fixed instruction set, will be instruction decoding by ROM or PROM. Hence, there will be room to modify the instruction set until fairly late in the design cycle, as long as the new instructions use the same basic hardware resources as the defined instructions. Thus, for example, a Newton-Raphson square root could be included as a microprogrammed instruction, but the square root algorithm that uses a slight modification of the divide algorithm would involve a one or two gate change in the arithmetic chip and could not. Double precision instructions are microprogrammed from single-precision hardware.

C.4 COORDINATOR OPERATIONS

In all test cases extracted from aero flow or weather codes, the coordinator has nothing to do for long stretches of time, only an occasional SYNC instruction to enforce the data precedence conditions at the end of the DOALL.

On the other hand, the coordinator will have system functions to perform, such as responding to I/O-complete interrupts at the end of DBM-EM transfers. These two functions are interlaced at the same instruction execution station; "all processors ready" is an interrupt that is allowed in system-function code execution, and masked off in user code, so that system functions can be executed during the long waits in coordinator user code.

C.5 FORMATS

This instruction set is presented to demonstrate feasibility of the FMP. Some of the assumptions underlying this instruction set could conceivably be changed during the actual design of the FMP. These assumptions include addressible registers, a desire to sometimes use absolute addresses and a data word size of 48 bits. A data word size of 48 bits points to 48 bits and its submultiples as preferred instruction sizes also. This instruction set assumes 24 and 48-bit instructions. Within 24 bits we get an opcode and 3 register addresses; or an opcode, tw) register address and a 7-or-8-bit countfield; or a 4-bit opcode, a register address, and a 16-bit literal. Within 48 bits we can get two address-sized fields with or without index designations plus one register address and an opcode; or one address-sized field and two or three register addresses.

If, instead, one assumes 16, 32, and 48-bit formats, the register instructions would largely be two-address, either Reg$_1$ op Reg$_2$ Reg$_1$ or Reg$_1$ op Reg$_2$ Accumulator, in order to fit the common instructions into 16 bits.

C.6 ADDRESSING

The address field (18 bits) consists of either "00" + 16-bit absolute address, "01" + 16-bit literal, "10" + 4-bit register identifier + 12 bits offset, or "11" undefined so far.

Absolute addressing is intended to be used only for system software and for FORTRAN common. Simple variables and "descriptors" have relative addresses with respect to the stack pointer, just like in B 6700, and 12 bits should be enough. "Descriptions" is loosely used to refer to base addresses of named common areas and base addresses of local arrays (or "IN ALL" arrays whose scope is within the subprogram only).

In test cases, 12 bits was enough to access any element of any local array. A base address of a local array, once fetched to a local register, can be used for several accesses to that array. When a single computed address is not enough then the restriction to only one register that can be added to the offset creates some additional integer arithmetic that has to be programmed. The test cases show enough cases where the programming consists of a single integer add, as to suggest that a fourth address format ought to be "11" followed by two four-bit integer register addresses and an 8-bit address. The saving is one of code file size only, and not directly in execution time, since the act of adding two integers together takes one clock whether those integers are specified in a separate IADD instruction or specified as indices associated with an address field. Such double indexing would add one clock to the beginning of any instruction in which it occurred. It is not included in this description.

C.7 NUMBER OF INSTRUCTIONS

How reasonable is the expectation that the opcode field will be 8 bits? In this list are 174 processor instructions, 64 floating-point-only, 79 integer-only, and 31 other. Character or string operators still are to be added. There are 100 coordinator instructions, of which 29 are for system and diagnostic actions. Some instructions occur very frequently, and it is worth shortening the opcode to pack them into a smaller word. For example, IMOVEL and IJUMP are candidates for being 24-bit instructions. If they are, then their opcode is only 4 bits long, and they each occupy 16 of the 256 slots in an 8-bit opcode space. It is not possible to have a floating-vs.-integer bit in the opcode, and a half-word vs. full-word bit too, leaving 64 instructions in each category.

C.8 INSTRUCTION EXECUTION TIMING

Timings are given in Table C.2. For the processor instructions there are four separate functional units involved. Each instruction has a starting time in each of the three units and an ending time or does not use that unit. The time of execution of each instruction is dependent on its time of occupancy (if any) in each of the first three independent execution units, namely: integer unit, floating point unit, and memory controls. The timing is described most easily with respect to the instruction fetching process, which determines the starting time of each successive instruction. The fourth function unit, the CN buffer, allows EM fetches and stores to transpire in parallel with other processing. It executes independently, once started, and does not affect the starting of the next instruction, but may affect the starting of the next instruction to use the CN buffer.

Entries in the table have the following significance:

"No. of clock periods" is the number of clocks from when the instruction normally issues to a functional unit, to the termination of the instruction. The instruction will always have been decoded from out of the staging register for at least one clock prior to this.

"Unit busy" is of the form n-m, where n is the number of the latest clock that previous instruction is allowed to occupy this unit, and m is the last clock that this current instruction occupies this unit.

Some instructions stop the instruction fetching process for a while, until the coordinator or CN buffer restarts it. The clock times given for these instructions represent the time from first decoding such an instruction in the staging register, until the start of decoding of the next instruction, under the most favorable circumstances. These are WAIT, STOP, HELP, and any instruction using the CN buffer.

C.8.1 Instruction Fetch Timing

Timing of the instruction fetching mechanisms can be seen with respect to Figure C.1. The next instruction is being held in a staging register. Out of the staging register is decoded the start times required for the functional units if this instruction were to start at this clock, and the time it will occupy the holding register. Also decoded are CN buffer requirements. Out of the integer, the floating point, and the memory control functional unit is decoded the ending time associated with the currently executing instruction. Out of the CN buffer are the "I got here", "busy" and "full" conditions. The "scoreboard" compares all inputs. When all comparisons say the next instruction will not interfere with current instructions, the instruction is transferred from the staging register to the one or more functional unit instruction registers. If delayed starts in other functional units are part of this instruction, the instruction is passed to the holding register to free the staging register for the next instruction.

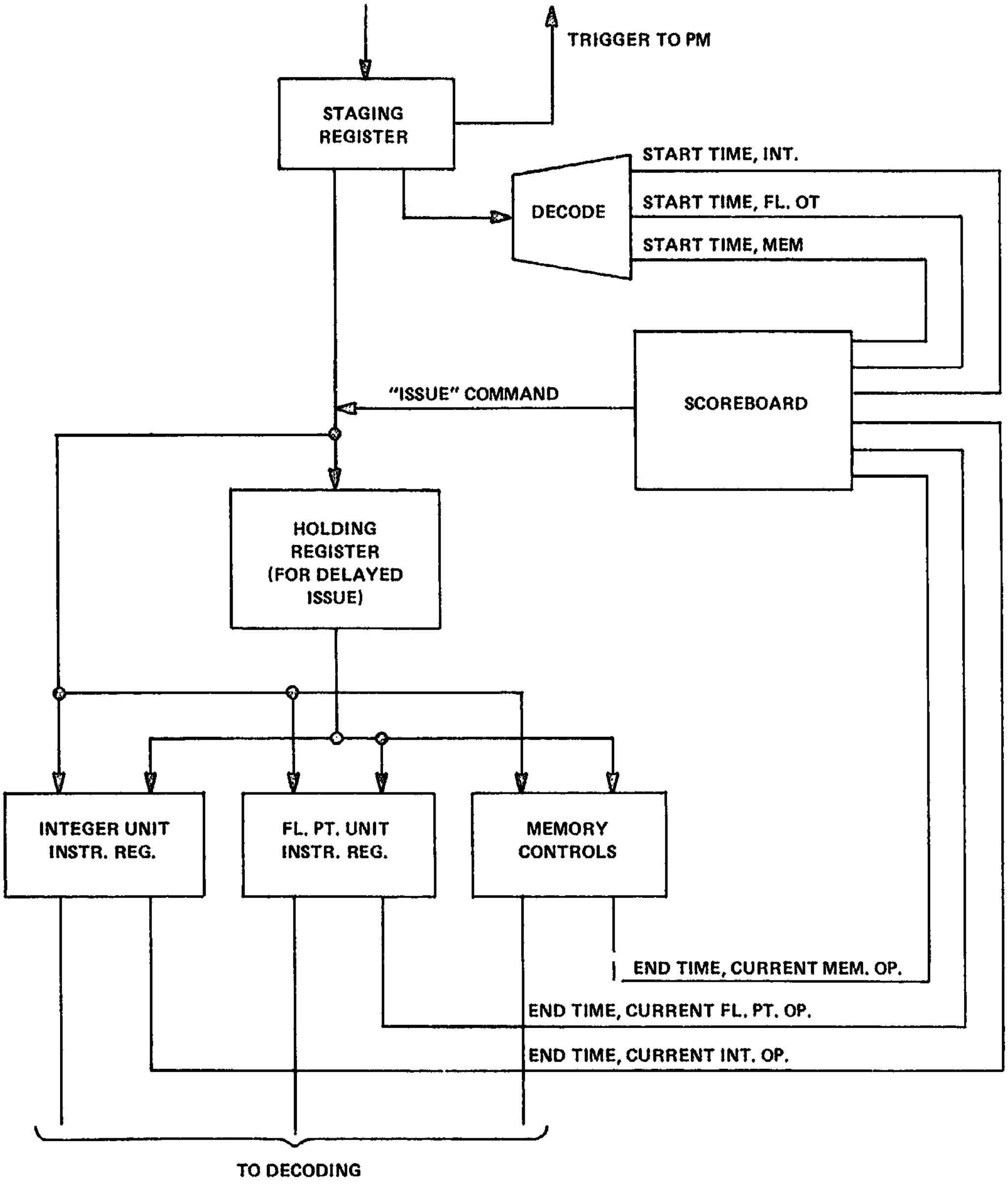


Figure C.1 Instruction Fetching Mechanism

The program counter always points to the next word in memory after the staging register contents. Thus, normally the PM will be holding teh next instruction word statically at its output lines. Only when the staging register is unloaded in less than three clocks (the PM cycle) or PM is accessing data will the next word not appear.

A complexity is the existence of half-word and full-word instructions. Second halves of instructions words carry the next half word instruction, so full-word instructions may only have their first half present in the staging register. The first half is sufficient to determine the timing. However, the second half will contain any memory addresses, so when a fetch from memory is involved, the second half must also be fetched before the memory part of the operation can start.

Those instructions which contain a memory address (either for data or as a branch address), or a literal, are full-word 48-bit instructions. Others are 24 bits. FL, floating literal, is one and a half words.

The arithmetic timings assume perfect rounding on single length floating point operations, but that the excess precision makes rounding unnecessary on double length operations.

Instructions labelled "branch" will cause all lookahead to hold up until the direction that the branch takes is determined. Branches defeat overlap. If the branch is taken, there will be additional five clocks, three for fetching the instruction and two for filling the instruction lookahead mechanism, bofore the instruction after the branch can start executing.

An alternative method of providing branching capability is to separate the testing operation, which sets one or more result bits, and the branch instructions, which test those bits. This method has the advantage that one can define a scheme for having lookahead fetch instructions along the branched-to path, rather than in the fall-through direction. Since branches are usually taken, some slight improvement in performance would accrue, in addition to which the instruction set becomes somewhat simplified.

The instructions CLOADEMN, and CSTOREMN are assumed to be implemented as a microprogrammed sequence of successive single EM accesses. These could be substantially speeded up if hardware were added so that the EM module could recognize these commands as different from single LOADEM and STOREM, and keep the CN path locked up.

The CN clock frequency is the third (submultiple) of the main clock frequency.. With the main clock 40 ns, the CN clock is 120 ns. All passing of addresses and data through the CN will be synchronized with this CN clock. Thus, the 6, 9, 12, or 15 clocks taken by instructions that pass data through the CN are actually 2, 3, 4, or 5 CN clocks. Operations involving the CN buffer only, such as loading its registers or testing its flipflops, can be done on any processor clock, and are not locked to any one of the three phases of the CN clock. For example, the instruction FSTOREM takes three clocks to load address and data into the CN buffer if it finds it free. These clocks do not depend on CN clock phase. However, the minimum of 6 clocks that the CN buffer is busy involves sending data to EM, and can be the minimum of 6 only if the SOTREM loads the data into the CN buffer at the proper CN clock phase.

For an example of these timing rules applied, see Reference 2.

C.8.2 Coordinator Timing

The coordinator has a similar set of independent units. There is an arithmetic unit similar to the processor's integer unit. There is a memory control unit. For accessing EM, there is a CN buffer unit identical to those found in each processor. The coordinator also has access to a port on the EM side of the CN, from which it can broadcast data to all processors, and "harvest" data from all processors. This second port is part of the arithmetic unit, for timing, and the compiler will ensure that the CN is idle whenever the instructions that use this port, mostly the instructions that are included for diagnostics, are used. These are the instructions from BDCST through READPM in Table C.2. Although they use the CN, they do not use the CN buffer.

The diagnostic controller is not used during normal program running. It is used only for diagnostics and system initialization. Hence, diagnostic controller information is not required to generate timing information about user programs.

C.8.3 Synchronization

Synchronization enters into the timing analysis in two ways. First, the instructions that use the CN buffer may test to see whether "I got here" is up, and may test whether the CN buffer is "full", "busy" or neither. The actual tests required are listed in the descriptions of the individual instructions. These instructions then wait until the CN buffer takes on the appropriate state before continuing. Some of these instructions leave the CN buffer with an unexecuted command, such as STOREM that will be "busy" until the address and data has been successfully emptied into an EM module, or LOADEM which will be "busy" until data comes back from EM to make it "full". The processor will be free to go on executing any instruction except those which depend on the CN buffer having gotten to the new state. Some CN buffer states require action on the part of the coordinator. For example, only after all processors execute EMFILL can the coordinator execute the HVST instructions. Only after all processors execute EMREQ can the coordinator execute the corresponding FETCHEM or BDCST. Only after the coordinator has executed FETCHEM or BDCST can the processor execute the REM instruction that accepts the broadcast data.

The second synchronization method involves single processor instructions such as WAIT. The processor checks to see if "I got here" is down from any previous case. If not, it waits for "go" to come from the coordinator to reset "I got here". Then the processor raises "I got here", and waits for "go" before fetching the next instruction.

C.8.4 Exceptional Cases

Within the processor, all fault cases result in an interrupt to system software that is resident in the processor estimated at less than 1K words. It is possible to handle some interrupts without interrupting the CR Floating-point out-of-range detection does not cause interrupts, but results in setting the floating-point variables into "infinity" or "infinitesimal". Any integer overflow causes an interrupt, on the theory that most integer operations are address calculations and overflow represents a faulty address. Attempting to insert a number outside the range $\pm2^{15}-1$ into a 16-bit integer register causes an integer interrupt; likewise executing a FIXD (double-length integer) on a number outside the range $\pm2^{31}-1$ results in interrupt. Any detection of error in the error-detection-correction logic results in processor interrupt. When the error is correctible, the interrupt merely logs its occurrence and returns to user processing within a few microseconds.

The processor enters interrupt mode whenever any bit of the interrupt register, not disabled by the corresponding bit of the mask register, is set. The "interrupt" mode flipflop is visible to the coordinator, which can interrogate whether any processor is in interrupt mode. One of the bits of the coordinator interrupt register is the "all processors ready" signal, thereby allowing the coordinator to perform system software functions during its long waits in user program.

Note that there are two lines from the processor to the coordinator that can be called "interrupt" lines. The processor HELP instruction raises an "interrupt" line that sets the "processor interrupting" bit in the coordinator's interrupt register. The "processing interrupt" mode of each processor can be interrogated by the PINT instruction of the coordinator. In one case the intent is for the processor to interrupt the coordinator; in the other, the processor has been interrupted.

C.9 INTERRUPTS

Both coordinator and processor have an interrupt register. Processor interrupts are to processor-resident software, for logging recoverable errors, processor software will return to user processing within a few microseconds. For non-recoverable errors, processor software issues an interrupt to the coordinator in order to shut down the entire FMP. In the processor, the list of interrupts is (with reccverable interrupts identified):

Single error corrected in processor memory (recoverable)

Double error detected in processor memory

Single error corrected in word received from CN buffer (recoverable)

Double error detected in word received from CN buffer

Parity error in microprogram word

Memory bounds error

Uninitialized word fetched from EM

Unnormalized floating point operand detected

Integer overflow

Divide by zero integer

Divide by zero floating point

Error detected in logic operation of EU

Software generated interrupt (set by ICALLI) (recoverable)

Illegal Op Code

Floating point overflow and underflow are caught by changing the word to "unpresentable" (or loosely, "infinity") and "infinitesimal". Divide by floating point zero also results in "unrepresentable", so for some purposes this interrupt would be masked of: as redundant. There is a control bit which determines whether integer underflow results in infinitesimal or zero. The single error corrections are serviced by a routine resident in the processor which logs their occurrence. Return is to the user program. Most other interrupts will result in program termination. It is the design intent to save the memory address and the corrected bit number for error corrections and the memory address of double error detections.

In the coordinator, the interrupt register has the following bits:

EM module error EM module parity error data in (address)

Single error detected in coordinator memory

Double error detected in coordinator memory

Single error detected in word receiv:d from CN buffer

Double error detected in word received from CN buffer

Parity error in microprogram word

Processor interrupt (sent by processor)

All processors ready (interrupts system software to get user program's instruction executed)

Memory bounds in coordinator memory

Illegal opcode

Memory bounds in EM

Support processor interrupt

DBM result descriptor ready

Diagnostic controller interrupt

Timeout, no instruction executed for the last X ms.

Interval timer count down to zero

Integer overflow

Divide by zero

Logic error detected in coordinator operation

DBM controller error detected

Software generated interrupt (set by CCALLI)

Unrecoverable interrupts enter interrupt processing at address 0. Recoverable interrupts (single error corrections and ICALLI in the processor, in the coordinator single error, CCALLI, interval timer, support processor interrupt) enter interrupt processing at a second, hard-wired address. In the coordinator, the "all processors ready" interrupt has its own hard-wired address. Processor interrupts interrupt to processor-resident software; coordinator interrupts interrupt to coordinator-resident software.

C.10 SUBROUTINE ENTRY AND RETURN

A description of how this is done.

<u>Environment</u>

Two integer registers are permanently designated as

SB The pointer to the "base" of the address space for the current subroutine

SL the pointer to the limit of this address space (actually points to the first word beyond the allocated space).

"S" stands for "stack", since space is allocated as a stack. Fig. C.2 shows this stack.

C.10.1 Subroutine Entry

Prior to the call, the variables temporarily held in registers must be stored back in PM if there is any chance the called subroutine will reference them. Registers that the called subroutine will use must also be saved. The compiler simply stores everything back to its "home" address in PM.

At the place pointed to by SL, the caller next writes any parameters passed by value (where this is allowed in our FORTRAN), and the base addresses of any arrays being passed, and the descriptors of any named common areas. There are P words in this area, where P is known to the compiler.

Next, the CALL instruction is executed. It does the following:

1. The content of SB, SL, and program counter are concatenated and written into address P+SL.

2. Register SB is loaded by SL + P

3. Register SL is loaded by the new value of SB plus a literal, the space allocation known to the compiler.

CALL therefore has two parameters, the number of parameters passed, and amount of space allocated. In ANSI FORTRAN 77, both of these would be literal fields in the instruction. For some of the dynamic array sizes that are allowed in FMP FORTRAN, it will be necessary to insert code to compute the size, and leave it in an integer register. The absolute program address is computed from the content of the branch address field and inserted into PCR for fetching the next instruction.

C.10.2 Subroutine Return

RETURN executes as follows:

1. Fetch the word addressed by SB

2. Unpack that word into SB, SL, and the program counter.

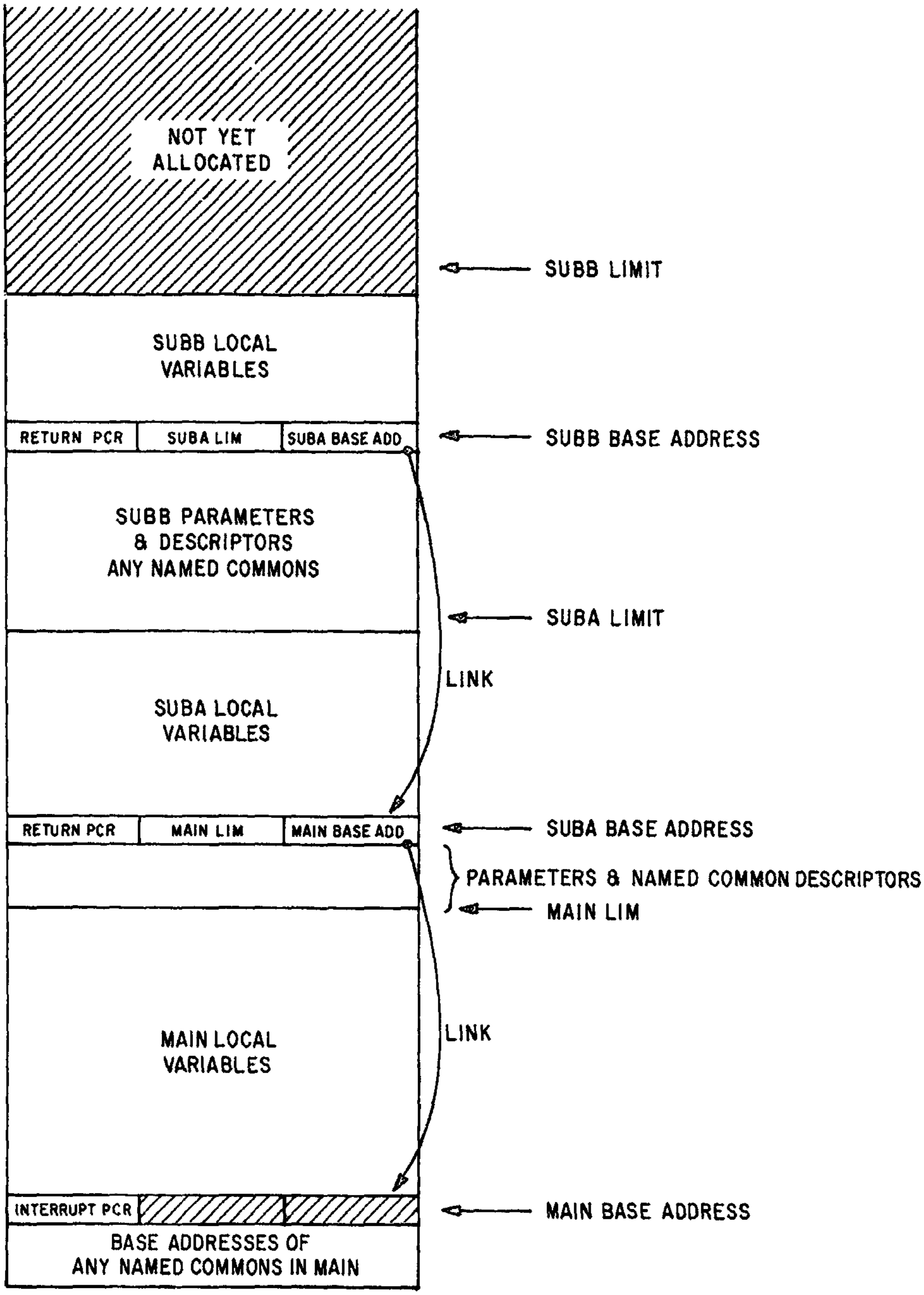


Figure C.2 Subroutine Stack

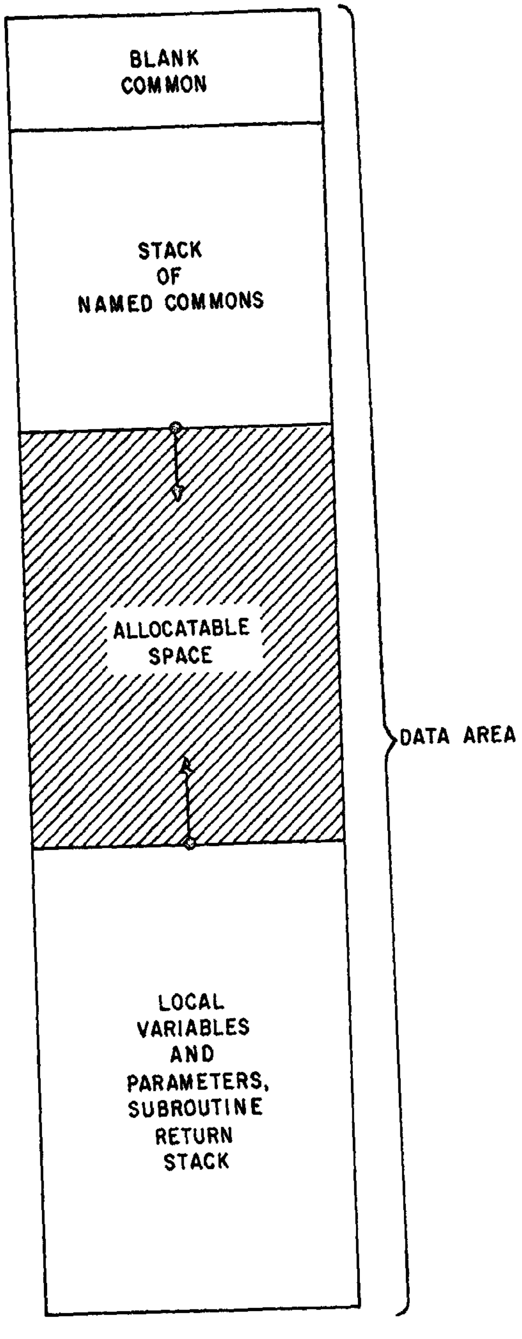


Figure C.3 Stack Allocation in the Data Area

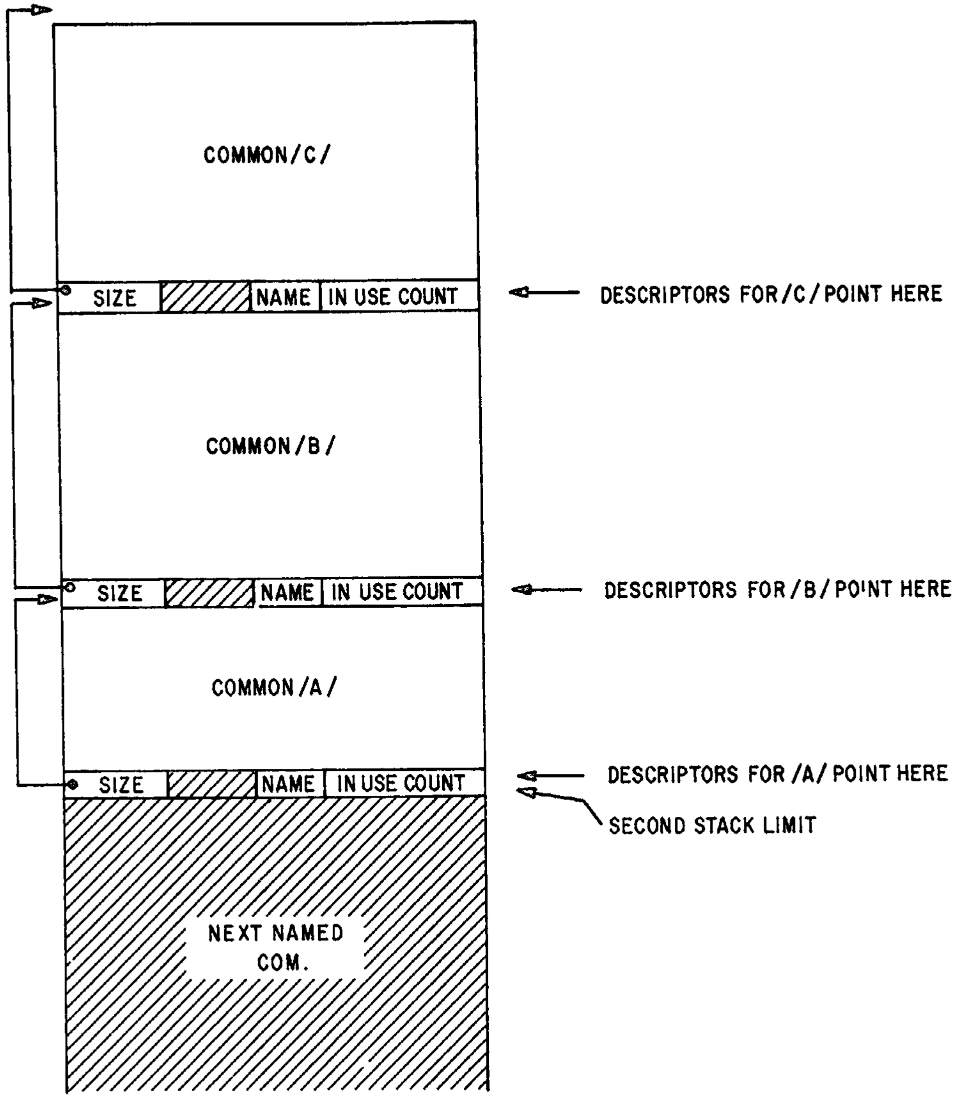


Figure C.4 Organization of Named Common

If the subroutine is a function, the results of that function will be left in a single-length or double-length integer or floating-point register as appropriate. The register is determined by convention, and is the same for all functions of the same type.

C.10.3 Within the Subroutine

Working space is addressed by positive offsets relative to SB. We have 12 bits of address that may be added to an integer register as part of the normal addressing machinery. When 12 bits is not enough, the compiler will have to use integer instructions to build the address.

Parameters and base addresses of named common areas are accessed by negative offsets from SB, as implied in the description of entering a subroutine in C.10.1.

C.10.4 Addressing

With the above structure, absolute addresses may be used for simple variables in the main program and for blank COMMON. Varying degrees of indirection are implied, the most complicated case being an element of an array in a named common in a subroutine, where an offset from SB is used to find the base address of the named common, an offset from that base is the base of the array, and the element is offset from the array beginning. (A smart compiler may combine the last two into a single offset and will fetch the base address of a named common to an integer register upon its first use.)

C.10.5 Named Common Mechanism

A second stack of space is allocated to all the named commons. If the first stack grows by increasing addresses, the second stack may grow by decreasing addresses. For example, see Figure C.3. At address zero of each named common is a count of how many subroutines are currently active which name that common. Each CALL goes through the descriptors in the parameter area and increases each count. Each RETURN goes through the descriptors in the parameter area and decreases each count. A named Common used in all subroutines lasts the entire run, therefore as does blank common. The words at address zero also contain the size of the named common, so that they form a relatively-addressed linked list to each other. See. Fig. C.4. Whenever the count goes to zero at the last named common, the stack limit of the second stack is decreased to the first non- zero count.

In ALGOL, where addressing environments are nested in lexic levels, the above mechanism always releases space upon the exit from the last lexic level that needs that space. In FORTRAN, if we adopt the above mechanism, it is possible to undefine a block of space inside this second stack, but it won't be released until the spaces "above" it in the second stack are themselves released.

A named common disappears whenever no subroutine owning it is active. A named common descriptor will either be found in the calling subroutine, upon subroutine entry, or must be created.

Thus, the presence of the appropriately named descriptor in the calling subroutine causes the descriptor to be copied; while the absence of an appropriately named descriptor causes new space to be allocated, and a new descriptor to be created.

A provision for statically allocated common areas, that survive for the life of the job, can easily be made if desired. They have been omitted from this description because such statically allocated variables occupy needed space during times that they are inactive, and because such static allocation, outside of blank common, is not needed for compliance with FORTRAN 77. In the 3D implicit code, as explained in Appendix A, the maximum mesh size would be smaller if all variables were statically allocated.

C.10.6 Arithmetic Details

The design intent is to provide perfect rounding. A floating point number is a discretized representation of an assumed underlying real number. When two floating point numbers are combined, the result is to be the closest representation possible of the real number result from combining the two underlying real numbers. Thus, whenever the guard bits are less than one half a least significant bit, the surviving part of the mantissa shall be left alone. When they are more than one half a least significant bit, one is to add 1 to the surviving part of the mantissa. In the FMP processor, a full double length accumulator cannot be justified. Therefore, when the eight guard bits are exactly one ONE followed by ZEROs, they may represent one ONE followed by seven ZEROs, followed by additional unknown bits, hence we round by adding 1 in the least significant place whenever the most significant guard bit is ONE. Alignment of addends is done in one clock, with a barrel, hence the implementation of a "sticky" bit represents substantial hardware investment.

Guard bits and rounding are used to preserve precisions in single-length arithmetic (36 bit precision, but not in double length (72 bits precision), giving roughly 11 decimal digits and 21 decimal digits of precision respectively.

Rounding occurs after normalization. Since we have six-or-eight guard bits, rounding is a no-op when normalization requires a left shift of more than six-or-eight places. The guard bits, shifted into the result by the normalization, protect precision effectively.

Rounding after addition can be simplified by observing that whenever mantissa overflow occurs after the rounding, the resulting mantissa must be .100000 . . . However, we have to add one to the exponent. Since the exponent adder is not otherwise busy during rounding, we have exponent in the result register and exponent +1 being presented at the output of the exponent adder, so that, if rounding overflows, exponent +1 is loaded into the result exponent field, while .1000000000 is loaded into the mantissa, all without requiring any additional clock.

A zero result that may get rounded away from zero is a special case. The sign and exponent of an apparently zero result must be saved until after rounding, to accommodate the case that the result will be rounded away from zero. All zeroes have positive sign and the smallest allowable exponent.

In multiply, normalization and rounding are done together in one clock. A product never overflows, and normalization is by either no or one place. Add 1/4 of a LSB to the product on the last cycle of multiply using the carry input to the second guard bit. Thus, if normalization by one place is required, the product is already rounded. If normalization is not required, add another 1/4 of a LSB. Only 2 guard bits are needed at the end of multiply (although more are needed to keep the partial products honest during the formation of the final product). Thus normalization and rounding take one clock altogether. At that last clock: normalize the already-rounded product if the leading bit is ZERO; select the output of the adder (Result + 1/4 (LSB)) if the leading bit of mantissa is ONE.

C.10.7 Other Instructions

Some operations are implemented as simple by-products of the instructions in Table C.1. By-product instructions include:

Convert from single-precision to double-precision. Given by FADDXL, literal=zero.

Convert from double-precision to single-precision. Address the first half only of a double-precision word.

Divide (multiply) integer by power of two. ISHN(L).

Extract fraction-part from floating-point word. FMOVEXL, literal = zero. Useful in mathematical functions.

Half-word and full-word No-ops.

TABLE C.1
Processor Instructions

Floating

FADD(M,L), FSUB(M,L)	Reg$_1$ plus (or minus) operand → Reg$_3$
FMUL(M,L)	Reg$_1$ times operand → Reg$_3$
FDIV(M,L)	Reg$_1$ divided by operand → Reg$_3$
FDVR(M,L)	Operand divided by Reg$_1$ → Reg$_3$
FMAD(M,L), FSUB(M,L)	Reg$_1$ times operand is added to (subtracted from) Reg$_3$ → Reg$_3$
FSSQ(M)	Reg$_1$ squared plus operand squared → Reg$_3$
FADEX(L)	Add operand (if literal, may be limited to 8-bit literal in countfield) to exponent field in fl. pt. reg. If operand is in register, it will be an integer register.
FMOVEX(L)	Transfer operand (from int. reg., or literal) to exponent field in fl. pt. reg.
	(in both the above instructions, the operand is in integer format and will be converted to floating point exponent format during the course of the instruction.)
FABS(M)	$\lvert \text{operand} \rvert$ → Reg
FNEG(M)	minus the operand → Reg
FADDX(M,L), FSUBX(M,L)	Reg$_1$ plus (minus) operand → Reg$_3$ & next (double-length)
FMULS	Double length product of Reg$_1$ and Reg$_2$ → Reg$_3$ & next
FADDD, FSUBD	Double length sum (difference), Reg$_1$ & next (op) Reg$_2$ & next → Reg$_3$ & next
FMULD	Double length product of two double-length operands. Reg$_1$ & next * Reg$_2$ & next → Reg$_3$ & next
FL	The 48 bits following this opcode → Reg$_3$

FMOVE(M,L) Operand → Reg$_3$

FPAKM Most sig. 24 bits of Reg$_1$ & most sig. 24 bits of Reg$_2$ → memory

FUPKM From memory, the most sig. 24 bits of memory word & 24 zeroes → Reg$_1$, the least sig. 24 bits of memory word & 24 zeroes → Reg$_2$.

FSTORE Reg → memory.

FIX Convert operand in fl. pt. Reg. to nearest rounded integer value → int. Reg.

FIXD Convert operand in fl. pt. reg. to nearest rounded integer value → int. Reg & int. Reg.$_{+1}$

FIXF Convert operand in fl. pt. reg$_1$to integer whose absolute value is the largest possible but not larger than the absolute value of the original operand. Result → int. Reg$_2$. (floor)

FIXC Convert operand in fl. pt. Reg$_1$ to integer whose absolute value is the smallest possible not smaller than the absolute value of the original operand. Result → Int. Reg$_2$. (Ceiling)

FINFLZ If Reg$_1$ contains "infinitesimal", zero → Reg$_1$

FIXEX Convert exponent in fl. pt. reg. to integer format → int. Reg.

FMT Convert content of fl. pt. reg. to floating point format used by the B-7800. (Will be microprogrammed, and will use logic in the integer unit.) If "unrepresentable" or exponent out of range, interrupt.

FMTI Content of fl. pt. register is assumed to be in B-7800 floating point format, and is converted to internal FMP floating point format.

FLOAT Convert integer in int. Reg. to fl. pt. format → fl. pt. Reg.

FLT(L,M),
FLE(M,L),
FGT(L,M),
FGE(L,M) If Reg$_1$ tests LT (or LE, GT, GE) operand, then GOTO branchaddress.

FEQL — If 1st 16 bits of Reg equal 16-bit literal, GOTO branchaddress. This yields tests for zero, "uninitialized", "infinitesimal" and "unrepresentable/infinity", since these are all encoded in ghe exponent field. No floating-point word with zero exponent is allowed except zero itself. (Tests for equal in floating point have otherwise been eliminated as useless and misleading.)

FLTD — Double-precision compare. If Reg$_1$ & next is less than Reg$_2$ & next, GOTO branchaddress. (Reverse registers for .GE.)

FGTD — If Reg$_1$ & next is greater than or equal to reg$_2$ & next, GOTO branchaddress. (Reverse register addresses for .LE..)

SETFL — Set infinitesimal control bit. Exponent underflow thereafter results in "infinitesimal".

SETZ — Reset infinitesimal control bit. Exponent underflow thereafter results in zero.

Integer

IADD(M,L), ISUB(M,L) — Reg$_1$ plus (minus) operand → Reg$_3$

IADD1, ISUB1 — Reg$_1$ plus (minus) 1 → Reg$_1$

IMUL(M,L) — Reg$_1$ times operand → Reg$_3$

IDIV(M,L) — Reg$_1$ divided by operand → Reg$_3$

IMOD(M,L) — Reg$_1$ modulo operand → Reg$_3$

IMOD521 — Reg$_1$ & next modulo 521 → Reg$_3$. (This is a special, fast, instruction, as it is needed to determine EM module number from EM address. Estimated, 4 clocks.) (Note the absence of IDIV521. A DIV512 will be built into the address path to CN buffer, taking no time. The 1.8% holes left this way in memory can be addressed by a different set of address computations; they will be in a logically disjoint address space.)

IADDX(M,L), ISUBX(M,L) — Reg$_1$ & next plus (minus) operand → Reg$_3$ & next (Double-length and single-length operands combined into a doublelength result)

IMULX(M,L) — Reg$_1$ & next times operand → Reg$_3$ & next

IDIVX(M,L) — Reg$_1$ & next divided by operand → Reg$_3$ & next

IMODX(M,L) — Reg$_1$ & next modulo operand → Reg$_3$

IADDD, ISUBD — Reg$_1$ & next plus (minus) Reg$_2$ & next → Reg$_3$ & next

ISH(C,S,N)(L) — Shift Reg$_1$ end-around (or end-off, or numeric with sign-bit fill if right or zero fill if left) by the distance shown by the operand (positive is shift left, to coincide with the requirements of numeric shifts)

ISH(C,S,N)D(L) — Shift, as above, except double-length. Reg$_1$ & next.

IOR(M,L), IAND(M,L), IIMP(M,L), IXOR(M,L) — Reg$_1$ OR (or AND, implies, exclusive OR) operand → Reg$_3$

INOT(M,L) — NOT operand → Reg$_3$

IDL — Literal (32 bits) → Reg & next

IADDL — Reg & next plus literal (32 bits) → Reg & next

IMOVE(M,L) — Operand → Reg$_2$

IDMOVE(M,L) — Operand → Reg$_2$ & next (if operand is register, it is a double length register)

IPNO — Processor number (wired into backplane) minus "sparebit" → Reg. Sparebit = 1 if processor above the spare location, =0 if processor below. Leading two bits are cabinet number, and are not involved in the subtraction, since each cabinet has one spare.

IPAK3M — Reg$_1$ & Reg$_2$ & Reg$_3$ → memory

IUPK3M — Memory → Reg$_1$ & Reg$_2$ & Reg$_3$

IPAK3F — Reg$_1$ & Reg$_2$ & Reg$_3$ → fl. pt. reg. (because of instruction format limitations, not all three int. Reg. will be explicitly addressed, one or two of them will be "next" int. Reg.

IUPK3F Fl. pt. Reg. → Reg$_1$ & Reg$_2$ & Reg$_3$

ISTORE 32 zeroes & Reg → memory

IDSTORE 16 zeroes & Reg & next → memory

ILT(M,L), ILE(M,L), IGT(M,L), IGE(M,L), IEQ(M,L), INE(M,L) If Reg$_1$ test LT (or LE, GT, GE, EQ, NE) operand, then GOTO branchaddress

IDLT, IDGT, IDEQ, IDNE If Reg$_1$ & next tests LT (or GT, EQ, or NE) to Reg$_2$ & next, then GOTO branchaddress. (Reversal of registers provides the relations.GE. and .LE..)

IBIT(L) If any bit of Reg ANDed with operand is ONE, GOTO branchaddress

<u>CN Buffer</u>

FSTOREM Wait for CN buffer to become NOT "busy". Send int. Reg.$_1$ (EM module number) and int. Reg$_2$ & next (EM address) to CN buffer address portion, send fl. pt. Reg$_3$ to CN buffer data portion. Mark CN buffer "busy". (Following this instruction, CN buffer will story "busy" until an acknowledge is received from the EM module, and the buffer contents transmitted. Buffer will then be NOT "busy" and NOT "full". The processor instruction execution does not wait for any of this to happen.)

ISTOREM Same as FSTOREM except substitute int. Reg$_3$ for fl. pt. Reg$_3$.

IDSTOREM Same as FSTOREM except substitute int. Reg$_3$ & next for fl. pt. Reg$_3$.

I3STOREM Same as FSTOREM except substitute int. Reg$_3$ &Int Reg$_4$ & int. Reg$_5$ for fl. pt. Reg$_3$. Format limitations will probably force the use of implicit addresses for Reg$_4$ and Reg$_5$. They are likely to be the next two after Reg$_3$.

MSTOREM Same as FSTOREM except substitute memory for fl. pt. Reg$_3$. (Note the asymmetry between STOREM and LOADEM. In LOADEM, the selection of destination is separated from the EM address operation, in order to allow the compiler to optimize the sequencing of instructions. In STOREM, the instructions are combined in order to save code space.)

EMREQ (Formerly "FETCHEM" and "BDCST", but with the new CN these initiating actions are the same for both, i.e., just one instruction) Wait for "I got here" to be reset, if up. If CN buffer is "busy", wait for CN buffer to become NOT "busy". Raise "I got here". (Later, data will arrive in the CN buffer, which will then be marked "FULL", and the data can be read by any of the -REM instructions. Depending on whether the coordinator executed a FETCHEM or a BDCST instruction, that data will have arrived from EM or from the coordinator itself.)

EMFILL (Formerly "HVST" and "SHIFTN", but with the new CN these initiating actions are the same for both, just one instruction.) If CN buffer "busy", wait for NOT "busy". If "I got here" is up, wait for "go" to reset it. Raise "I got here", load CN buffer (datapart) from fl. pt. Reg, and set CN buffer to "busy". (Following this instruction, the coordinator will set the CN, to a "broadcast" condition if HVST, or to a "wraparound" condition if SHIFCN, and move the data from the CN buffer. If SHIFCN is in the coordinator instruction stream, then the compiler will have inserted some form of -REM instruction later on in the processor instruction stream to read the now "full" CN buffer. Other sources of data are expected to be used so seldom that instructions to HVST or swap data to and from integer registers and memory are judged to be a waste of decoding complexity.)

LOADEM Send Reg$_1$ (EM module no.) and Reg$_2$ & next (EM address) to CN buffer, after waiting for TN buffer to become NOT "busy". CN buffer will now become "busy" until data arrives from EM, whereupon CN buffer becomes "full". Fetch next instruction without waiting.

LOCKEM Send Reg$_1$ (EM module no.) and Reg$_2$ & next (EM address) to CN buffer, after waiting for CN buffer to become NOT "busy". CN buffer now becomes "busy" until data arrives from EM, whereupon CN buffer becomes "full". Processor does not wait in this instruction beyond the loading of the CN buffer. EM module will set the least significant bit of the word in memory to ONE after transmitting the previous contents to the CN buffer. (Used for inter processor cooperation via EM independently of the coordinator.)

FREM Wait for CN buffer NOT "busy". If now NOT "full", error interrupt. CN buffer → Fl. pt. Reg. Mark buffer NOT "full" and NOT "busy".

IREM Same as FREM except CN buffer → Int. Reg.

IDREM Same as FREM except CN buffer → Int. Reg & next

I3REM Same as FREM except CN buffer → Int Reg & next & next

MREM Same as FREM except CN buffer → memory.

ITIX First; Reg$_1$ + Reg$_3$ → Reg$_1$. Then, test Reg$_1$ against Reg$_2$, test for greater if Reg$_3$ is positive, for less if Reg$_3$ is negative. If test succeeds, GOTO branchaddress.

ITIX1 Same, except an implied literal value of +1 substitutes for Reg$_3$

ITIXL Same, except an actual literal substitutes for Reg$_3$

IJUMP GOTO branchaddress

ICALL Subroutine entry. Push subroutine stack. Parameters and new working area are relative to the new stack address pointer.

IRETURN Subroutine return. Pop subroutine stack.

PUSH Push subroutine stack, do not change PCR (diagnostics).

POP Pop subroutine stack, do not change PCR (diagnostic use)

TOS(L) Set stack address pointer and the word pointed to new values found in Reg$_1$, Reg$_2$ and in operand. (Stack mechanism involves not only a stack address pointer, but also return information and address bound(s) in word 0 relative to that pointer.)

WAIT Wait for reset of "I got here" (if it is up). Raise "I got here". Wait for "go" before fetching next instruction.

STOP Wait for reset of "I got here". Reset "enable". Raise "I got here". Resetting of "enable" disables all further instruction fetching.

HELP Same as STOP plus raise "interrupt" line to coordinator.

IINT(L) Interrupt register AND operand -- Reg$_3$. Interrupt register AND NOT operand → interrupt register. Operand is Reg$_2$ or literal.

ISMASK(L)	Interrupt mask register OR operand -> interrupt mask register
IRMASK(L)	Interrupt mask register AND operand -> interrupt mask register
ICALLI	Enter interrupt mode
IRETI	Return from interrupt

Table C.1, part 2

Processor operations induced by commands issued by the coordinator.

RESETE Reset "enable" immediately. Do not wait for current instruction to finish. Reset "busy" in CN buffer, Reset "I got here".

HALT Reset "enable" only. Allow current instruction to complete.

FILLM Load word in CN buffer into processor memory. Increment memory address by 1. (MAR has previously been loaded)

FILLME Same but conditional on "enable".

FILLR Load register from CN buffer. Register address will follow this code on the command lines.

FILLRE Same as FILIR except conditional on "enable".

ADDR Address processor. Reset "enable". Check contents of CN buffer against processor number. If matches, Set "enable".

READR Transmit contents of register to CN buffer. Register address will follow this code on the command lines.

READM Read word from memory and transmit to CN buffer. Increment memory address by 1. (Register addresses will include registers not addressible by the address fields in the processor instruction set, such as PCR and memory address register.)

TABLE C.1, part 3

Coordinator Instruction Set

<u>Arithmetic</u>

CADD(N,L), CSB(N,L)	Reg$_1$ plus (or minus) operand → Reg$_3$. (Note the use of "N" to designate coordinator memory)
CADD1	Reg$_1$ plus 1 → Reg $_1$
CSUB1	Reg$_1$ minus 1 → Reg$_1$
CMUL(N,L)	Reg$_1$ times operand → Reg$_3$
CDIV(N,L)	Reg$_1$ DIV operand → Reg$_3$
CMOD(N,L)	Reg$_1$ module operand → Reg $_3$
CMOD521	Reg$_1$ modulo 521 → Reg$_3$. (Substantially faster than CMODL with literal = 521)
CSH(C,S,N)(L)	Shift end-around (or end-off, or numeric) the operand in Reg$_1$ by the distance shown in operand (reg$_2$ or literal)
CAND(N,L), COR(N,L), CIMP(N,L), CXOR(N,L)	Reg$_1$ AND (OR, implies, exclusive OR) operand — Reg$_3$
CNOT(N,L)	NOT operand → Reg
CMOVE(N,L)	Operand → Reg
CDL	32-bit literal → Reg
CADL	Reg$_1$ plus 32-bit literal → Reg$_1$
CSTORE	Reg → memory
CGT(N,L), CGE(N,L), CLT(N,L), CLE(N,L), CEQ(L,N), CNE(N,L)	Test Reg$_1$ for GT (or GE, LT, LE, EQ, NE) against operand, if test is true, GOTO branchaddress.
CBIT(N,L)	If any bit of Reg AND operand is ONE, GOTO branchaddress.

Other Branch Controls

CTIX First: $Reg_1 + Reg_3 \rightarrow Reg_3$. Then, test Reg_1 against Reg_2, test for greater if Reg_3 is positive, for less if Reg_3 is negative. If test succeeds, GOTO branchaddress.

CTIX1 Same, except an implied literal value of +1 substitutes for Reg_3.

CTIXL Same, except an actual literal substitutes for Reg_3

CJUMP GOTO branchaddress

CCALL Call subroutine, push subroutine stack.

CRETURN Return from subroutine, pop subroutine stack.

CPUSH Same push-stack action as CCALL, but do not change program counter.

CPOP Same pop-stack action as CRETURN, but do not change program counter.

CTOS(L) Change stack pointer by loading it with operand

CRETI Return from interrupt

CCALLI Enter interrupt mode.

Other

CLOADEM Fetch to Reg_1 from EM. EM address is in Reg_2, EM module no. is in Reg_3. (Separation of EM address and EM module no. permits accessing of both address spaces within the EM. Note that the "EM address" will be stripped of its last 9 bits before being transmitted to the EM as an "address within module".)

CLOCKEM Same as CLOADEM except that the EM module will set the least significant bit of the word in memory to ONE after fetching the word sent to the coordinator.

CLOADEMN(L) Fetch N words from EM: store to coordinator memory. Memory address is in instruction. EM address is in Reg_2, EM module no. is in Reg_3. N is Reg_1 or literal.

CSTOREM Store Reg_1 to EM. EM address is in Reg_2, module no. in Reg_3.

CSTOREMN(L) Store N words from memory to EM. EM address is in Reg$_2$, module no. in Reg$_3$. N is Reg$_1$ or literal (actually, countfield)

CINT(L) Interrupt register AND operand → Reg$_3$. Interrupt register AND NOT operand → interrupt register. Operand is Reg$_2$ or literal

CSMASK(L) Interrupt mask register OR operand → interrupt mask register.

CRMASK(L) Interrupt mask register AND operand → interrupt mask register. Any interrupt bit so unmasked causes interrupt when ONE.

(Note: The instructions in the coordinator up to this point represent functionally a subset of the processor capabilities. One possible implementation of them would be to use a copy of the processor as most of the coordinator. We believe that the coordinator needs 32-bit integer, and needs more integer registers, too often for this to be a good idea.)

. . . . . .

(The following instructions represent coordinator capabilities which are not needed in the processor. Indeed, one of the reasons for having a separate coordinator is so that these functions need not be replicated 512 times, once per processor, nor do the processors require the connectivity to the points (DBM controller, host, etc.) that these functions imply.)

Processor Cooperation

FETCHEM From EM address in Reg$_2$, and EM module no. in Reg$_3$, cause the given EM module to cycle, and the result broadcast to the CN buffer of all processors. Start of instruction will wait on "All processors ready" and "go" will be issued at an appropriately delayed time.

SHIFCN(L) Wait for "All processors ready". Send "wraparound" command to CN level N, where N is found in Reg or literal. Send "go".

LOOP Wait for "all processors ready". If NOT "any processor enabled", set the "enable" bit of all processors, and exit the instruction. If "any processor enabled", issue "go" and GOTO branchaddress.

SYNC Wait for "all processors ready". Issue "go"

BDCST Wait for "all processors ready". Set CN to "broadcast" mode, last 48 bits of Reg & next to CN buffers of all processors. Issue "go".

BDCSTN Wait for "all processors ready". Set CN to "broadcast" mode, send word fetched from coordinator memory thru CN to all CN buffers. Memory address has normal address format.

HVST Wait for "all processors ready". Set CN to "harvest" mode, contents of all CN buffers that are "full" are combined (ORred is acceptable; the actual formula for combining is logic designer's option) and transmitted to the last 48 bits of Reg$_1$ & next.

PINT If "any processor in interrupt mode". GOTO branchaddress

Actions Imposed on Processors

UBDCST Send N words to processor. N in Reg$_1$. Words taken from successive addresses in coordinator memory starting at address given in instruction. (Processor will have previously been put into a waiting or NOT "enable"d state, and its MAR loaded with the starting address in PM.)

UBDCSTE Same, except acceptance of data is conditional on "enable" bit of processor.

USETP Send contents of Reg$_1$ to processor register whose address is in Reg$_2$. (Used for initializing PCR, setting MAR, as well as for transmitting ordinary data.)

USETPE Same except conditional on "enable" bit of processor.

USETPO Same as USETP except that "enable" bit of all processors is turned on at end of instruction.

USETPEO Same as USETPO except that acceptance of data is conditional on previous state of enable bit.

HALTP Reset "enable" of every processor.

STOPP Reset "enable", "I got here", and "busy" of every processor. Processors will cease executing in mid-instruction.

TESTP If "all processors ready", GOTO branchaddress.

READP Word from processor register addressed in Reg$_1$ is brought back to Reg$_2$ and the register following Reg$_2$.

READPM Word from processor memory (at address set by MAR of processor) is brought back to Reg$_2$ and the register next after Reg$_2$.

READPMN N words from processor memory, starting at the address found in this instruction. This and previous two instructions are conditional on the "enable" bit.

PROC Wait for "All processors ready". Send ADDR command with contents of Reg$_1$ as the processor number.

TESTE If "any processor enabled", GOTO branchaddress.

SPARE Change the designation of spare processor, or of spare EM module. There are four registers designating spare processor, and four registers designating spare EM module. These registers are readable with the CMOVE instruction.

SETCN(L) Set CN controls to bit pattern found in register (literal). This command modifies CN function for diagnostic purposes, such as restricting access to one or the other sheet of a two-sheet CN.

I/O

TIOM Reg$_1$ & next transmitted to DBM controller as control word.

STATUS Status word of DBM controller fetched into Reg$_1$ & next (Status will be same as control word, except word count will be decremented to current state, and a field of status bits may have been changed by the DBM controller. Format TBD.)

TIOH Reg$_1$ & next transmitted to host-readable register. Interrupt host.

HOST Read host read and writable register into Reg$_1$ & next.

SCLOCK Transfer Reg to real-time clock. Clock decrements at a fixed rate, TBD, causing interrupt when it decrements past zero. Setting the clock resets the interrupt bit, if up.

RCLOCK Transfer contents of real-time clock counter to Reg.

TABLE C.2

PROCESSOR INSTRUCTIONS

Mnemonics	Half or Full Word	Proc. Clock Count	Int. Unit Busy	F.P. Unit Busy	Mem. Busy	Min. CF Buf. Busy
FADD, FSUB	½	6		0-6		
FADDM, FUSUBM	1	9	0-1	3-9	0-3	
FADDL, ASUBL	1	6		0-6		
FMUL	½	9		0-9		
FMULM	1	10	0-1	3-12	0-3	
FMULL	1	9		0-9		
FDIV, FDVR	½	44		0-44		
FDIVM, FDVRM	1	47	0-1	3-47	0-3	
FDIVL, FDVRL	1	44		0-44		
FMAD, FSUB	½	11		0-11		
FMADM, FSUBM	1	14	0-1	3-14	0-3	
FMADL, FSUBL	1	11		0-11		
FSSQ	½	21		0-21		
FSSQM	1	24	0-1	3-24	0-3	
FADEX	½	2	0-2	0-2		
FADEXL	½	2		0-2		
FMOVEX	½	2	0-2	0-2		
FMOVEXL	½	2		0-2		
FABS, FNEG	½	1		0-1		

TABLE C.2

PROCESSOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Proc. Clock Count	Int. Unit Busy	F.P. Unit Busy	Mem. Busy	Min. CF Buf. Busy
FABSM, FNEGM	1	4		3-4	0-3	
FADDX FSUBX	½	7		0-7		
FADDXM, FSUBXM	1	10	0-1	0-10	0-3	
FADDXL, FSUBXL	1	7		0-7		
FMULX	½	15		0-15		
FADDD, FSUBD	½	7		0-7		
FMULD	½	22		0-22		
FL (48-bit, only 1½ word format)	1½	4		1-4		
FMOVE	½	1		0-1		
FMOVEM	1	4	0-1	3-4	0-3	
FMOVEL	1	1		0-1		
FPAKM	1	9	6-7	0-6	6-9	
FPUPKM	1	5	0-1	2-5	0-3	
FSTORE	1	3	0-1	0-3	0-3	
FIX, FIXF, FIXC	½	4	3-4	0-3		
FIXD	½	5	3-5	0-3		
FINFLZ	½	1		0-1		
FIXEX	½	3	0-3	0-1		
FMT	½	7		0-7		

TABLE C.2

PROCESSOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Proc. Clock Count	Int. Unit Busy	F.P. Unit Busy	Mem. Busy	Min. CF Buf. Busy
FMTI	½	5		0-5		
FLOAT	½	3	0-1	0-3		
FLT, FLE, FGT, FGE (branch)	½	2	1-2	0-2		
FLTM, FLEM, FGTM, FGEM (branch)	1	5	4-5	3-5	0-3	
FLTL, FLEL, FGTL, FGEL (branch)	1	2	1-2	0-2		
FEQL (branch)	1	3	2-2	0-3		
FLTD, FGTD (branch)	½	3	2-3	0-3		
SETFL, SETZ	½	1		0-1		
IADD, ISUB, IADD1, ISUB1	½	1	0-1			
IADDM, ISUBM	1	4	0-4		0-3	
IADDL, ISUBL	1	1	0-1			
IMUL	½	9max	0-9			
IMULM	1	12max	0-12		0-3	
IMULL	1	9-max	0-9			
IDIV, IMOD	½	16max	0-16			
IDIVM, IMODM	1	19max	0-19		0-3	
IDIVL, IMODL	1	16max	0-16			
IMOD521	½	4	0-4			
IADDX, ISUBX	½	2	0-2			

TABLE C.2

PROCESSOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Proc. Clock Count	Int. Unit Busy	F.P. Unit Busy	Mem. Busy	Min. CF Buf. Busy
IADDXM, ISUBXM	1	5	0-5		0-3	
IADDXL, ISUBXL	1	2	0-2			
IMULX	½	17max	0-17			
IMULXM	1	20max	0-20		0-3	
IMULXL	1	17max	0-17			
IDIVX, IMODX	½	32max	0-32			
IDIVXM, IMODXM	1	35max	0-35			
IDIVXL, IMODXL	1	32max	0-32			
IADDD, ISUBD	½	2	0-2			
ISH(C,S,N)(L)	½	2	0-2			
ISH(C,S,N)D(L)	½	5	0-5			
IOR, IAND, IXOR, IIMP	½	1	0-1			
IORM, IANDM, IXORM, IIMPM	1	4	0-4		0-3	
IORL, IANDL, IXORL, IIMPL	1	1	0-1			
INOT, IMOVE, ITOS	½	1	0-1			
INOTM, IMOVEM	1	4	0-4		0-3	
INOTL, IMOVEL, ITOSL	1	1	0-1			
IDL, IADL	1	2	0-2			
IDMOVE	½	2	0-2			

TABLE C.2

PROCESSOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Proc. Clock Count	Int. Unit Busy	F.P. Unit Busy	Mem. Busy	Min. CF Buf. Busy
IDMOVEM	1	5	0-5		0-3	
IDMOVEL	1	2	0-2			
IPNO	½	2	0-2			
IPAK3M	½	6	0-4		3-6	
IPUK3M	½	6	0-6		0-3	
IPAK3F	½	4	0-4	3-4		
IUPK3F	½	4	0-4	0-1		
ISTORE	1	4	0-2		1-4	
IDSTORE	1	5	0-3		2-5	
ILT, ILE, IFT, IGE, IBIT (branch)	1	3	0-3			
ILTM, ILEM, IGTM, IGEM (branch)	1	6	0-6		0-3	
ILTL, ILEL, IGTL, IGEL, IBITL (branch)	1	3	0-3			
IEQ, INE (branch)	1	4	0-4			
EIQM, INEM (branch)	1	7	0-7		0-3	
IEQL, INEL (branch)	1	4	0-4			
IDLT, IDGT (branch)	1	4	0-4			
IDEQ, IDNE (branch)	1	6	0-6			
FSTOREM	½	3	0-3	2-3		0-9

TABLE C.2

PROCESSOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Proc. Clock Count	Int. Unit Busy	F.P. Unit Busy	Mem. Busy	Min. CF Buf. Busy
ISTOREM, IDSTOREM	½	3	0-3			0-9
I3STOREM	½	3	0-3			0-6
MSTOREM	1	4	0-4		0-3	1-8
EMREQ, EMFILL	½	1				0-1
LOADEM, LOCKEM	½	3	0-3			0-12
FREM	½	2		1-2		0-2
IREM	½	2	1-2			0-2
IDREM	½	3	1-3			0-3
I3REM	½	4	1-4			0-4
MREM	1	4	1-2		1-4	0-4
ITIX, ITIXL, ITIXL (branch)	1	3	0-3			
IJUMP (branch)	½	2	0-2			
ICALL, IRETURN, PUSH, OPO, IRETI	½	30 (TBD)				
WAIT, STOP, HELP	½	4				0-4
ICALLI	½	1				
IINT	½	2	0-2			
IINTL	1	2	0-2			
ISMASK, IRMASK	½	1	0-1			
ISMASKL, IRMASKL	1	1	0-1			

COORDINATOR INSTRUCTIONS

Mnemonics	Half or Full Word	Coord Clock Count	Arith Unit Busy	Mem. Busy	Min. CN Buf. Busy
CADD, CSUB, CADD1, CSUB1, CSH(C,S,N)(L), CAND, COR, CIMP, CXOR, CNOT, CMOVE, CTOS	½	1	0-1		
CADDN, CSUBN, CANDN, CORN, CIMPN, CXORN, CNOTN, CMOVEN	1	4	1-4	0-3	
CADDL, CSUBL, CADL, CDL, CANDL, CORL, CIMPL, CXORL, CNOTL, CMOVEL, CTOSL	1	1	0-1		
CMUL	½	16max	0-16		
CMULN	1	19max	3-19	0-3	
CMULL	1	16max	0-16		
CDIV, CMOD	½	32max	0-32		
CDIVM, CMODM	1	35max	0-35	0-3	
CDIVL, CMODL	1	32max	0-32		
CMOD521	½	4	0-4		
CSTORE	1	3	0-3	0-3	
CGT, CGE, CLT, CLE, CBIT (branch)	1	3	0-3		
CGTN, CGEN, CLTN, CLEN, CBITN (branch)	1	6	0-6	0-3	
CGTL, CGEL, CLTL, CLEL, CBITL (branch)	1	3	0-3		

COORDINATOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Coord Clock Count	Arith Unit Busy	Mem. Busy	Min. CN Buf. Busy
CEQ, CNE (branch)	½	4	0-4		
CEQN, CNEN (branch)	1	7	0-7	0-3	
CEQL, CNEL (branch)	1	4	0-4		
CTIX, CTIXL, CTIX± (branch)	1	3	0-3		
CJUMP (branch)	½	2	1-2		
CCALL, CRETURN, CPUSH, CPOP, CRETI	½	30 (TBD)			
CLOADEM, CLOCKEM	½	13	0-13		0-12
CLOADEMN(L)	1	9+ 12N	0-(7 +12N)	13-(9 +12N)	0-(9 +12N)
CSTOREM	½	3	0-3		0-7
CSTOREMN(L)	1	9N-6	0- (9N-6)		0-9N
CINT	½	2	0-2		
CINTL	1	2	0-2		
CSMASK, CRMASK	½	1	0-1		
CSMASKL, CRMASKL	1	1	0-1		
FETCHEM	½	13	0-3		0-12
SHIFCN, SHIFCNL	½	9	0-3		0-9
LOOP, PINT, TESTP, TESTE (branch)	½	2	0-2		

COORDINATOR INSTRUCTIONS (Cont)

Mnemonics	Half or Full Word	Coord Clock Count	Arith Unit Busy	Mem. Busy	Min. CN Buf. Busy
SYNC	½	2	0-2		
BDCST	½	9	0-5		
HVST	½	9	0-9		
BDCSTN	1	12	3-8	0-3	
UBDCST, UBDCSTE	1	7+6N	0-(7 +6N)		
USETP, USETPE, PROC	½	9	0-4		
USETPO, USETPEO	½	11	0-4		
HALTP, STOPP	½	1			
READP, READPM	½	15	0-15		
READPMN	1	15+ 6N	0-(15 +6N)		
TIOM, TIOH, STATUS, HOST	½	2	0-2		
SCLOCK, RCLOCK	½	3 (TBD)			

APPENDIX D

RELIABILITY, AVAILABILITY, AND MAINTAINABILITY PROGRAMS

From a system engineering viewpoint, the design of a reliable and maintainable digital computer system encompasses many interdisciplinary technical trade-off decisions. This appendix describes the computer programs called DESIGN and CONFIGURE which have been developed by the Burroughs Corporation to focus attention on critical Reliability, Availability, and Maintainability (RAM) design factors that have been repeatedly observed to dominate the frequency of abnormal system interruption and the duration of downtime in fault-tolerant computer systems. In analyzing the RAM characteristics of the Flow Model Processor (FMP), the DESIGN Program was used to pinpoint critical factors pertinent to the failure, repair, and recovery processes of the FMP that require concentrated design attention as the design progresses. The CONFIGURE program was used to predict the performance of the Support Processor and File Management Subsystems.

The following paragraphs describe the DESIGN and CONFIGURE programs in terms of the computer system models applied to the FMP and the NASF and the computations performed. Salient theoretical and practical assumptions associated with the mathematical model utilized and definitions of all input parameters and computed results are discussed to aid in understanding the analysis performed and interpreting computed results. Definitions of terms used in this appendix are presented in Section D.4.

D.1 COMPUTER SYSTEM MODEL

Traditionally, in mathematical analyses of repairable redundant systems, it has been common to assume that system failure occurs due to the depletion of hardware resources when an active hardware element fails before the previously active redundant hardware element(s) is repaired. Although this conventional failure and repair cycle type model has been applied successfully to investigate the hardware availability aspects of certain types of redundant systems, it has been of little practical value in predicting the operational RAM characteristics of fault-tolerant computer systems in which hardware elements operate under software control.

In an operational environment, the failure of a computer system to operate continuously frequently occurs for reasons other than the depletion of hardware resources due to permanent type failures which require repair actions. Common causes of computer system interruption and downtime include intermittent failures and the inability to automatically recover from certain single critical hardware failures. Since an accurate reliability estimate must take into account all applicable sources of system interruption and downtime, to the extent possible, Programs DESIGN and

CONFIGURE have been developed to treat overall computer system behavior in terms of hardware subsystems operation under software control as depicted in the availability block diagram of Figure D.1. From a reliability point of view, each of the critical subsystems in Figure D.1 must operate successfully in order sustain proper system operation.

As shown in Figure D.1, any number of independent hardware subsystems operating under software control can be defined to take into account as many functions as required. The subsystem model is based on the premise that if a redundant hardware element fails, the particular subsystem involved may be interrupted for a short time to effect reconfiguration. After a short delay, the subsystem is restored to operation, and continues to operate while the failed hardware element is being repaired. However, if more than the specified allowable number of hardware elements are down for repair or if a critical hardware element has failed, then the subsystem is down until the appropriate repair has been effected. As shown in Figure D.1, the failure of any subsystem breaks the critical success path, causing the system to fail.

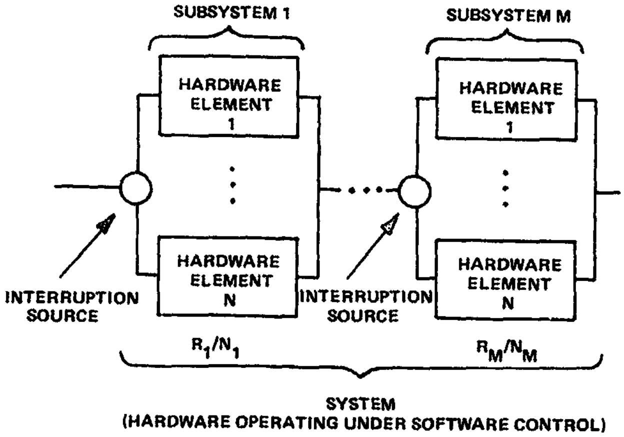


Figure D.1. Computer System Availability Block Diagram

Computation of Mean Up Time (MUT), Mean Down Time (MDT), and Availability for each of the specified critical elements and subsystems is performed using the mathematical model discussed in the following paragraph. System MUT, MDT, and Availability are then computed based on the successful operation of all subsystems using conventional methods. The assumption associated with this system decomposition technique requires only that the system be composed of independent subsystems, each of which can be regarded as having two possible outputs (working and failed) and that it is possible to identify a certain set of subsystem states as "working states" and the remaining states as "failed states".

D.2 MATHEMATICAL MODEL FOR THE DESIGN PROGRAM

The mathematical model employed in the DESIGN Program is a discrete-state continuous-time model called a Markov process. As with any type of Markov model, the underlying assumption of this process is that the transition probability P_{ij} from any state i to any state j depends only on the states of i and j and is completely independent of all past states except the last one. The transition probabilities must obey the following two rules:

- The probability of a transition in time Δt from one state to another is given by $Z(t)\Delta t$ where Z(t) is the hazard associated with the two states in question. If all $Z_i(t)$'s are constant, as assumed herein, the model is called homogeneous.

- The probabilities of more than one transition in time Δt are infinitesimals of higher order and can be neglected.

These properties and assumptions are quite widely accepted as being appropriate to modeling the failure and repair cycles of computer systems.

D.2.1 Markov Graphs

Figure D.2 is a Markov graph dipicting the transitions between states for each of the subsystems defined in the Computer System availability Block Diagram of Figure D.1. In Figure D.2, shaded states represent subsystem failure, and consequently system failure since all subsystems are required to be functioning properly to achieve system success.

For hardware subsystems operating under software control, the Markov Graph is quite complex. Therefore, the simplified Markov Graph shown in Figure D.3 will be described first as an introduction to considering the chain pertaining to the depletion of redundancy shown in Figure D.3.

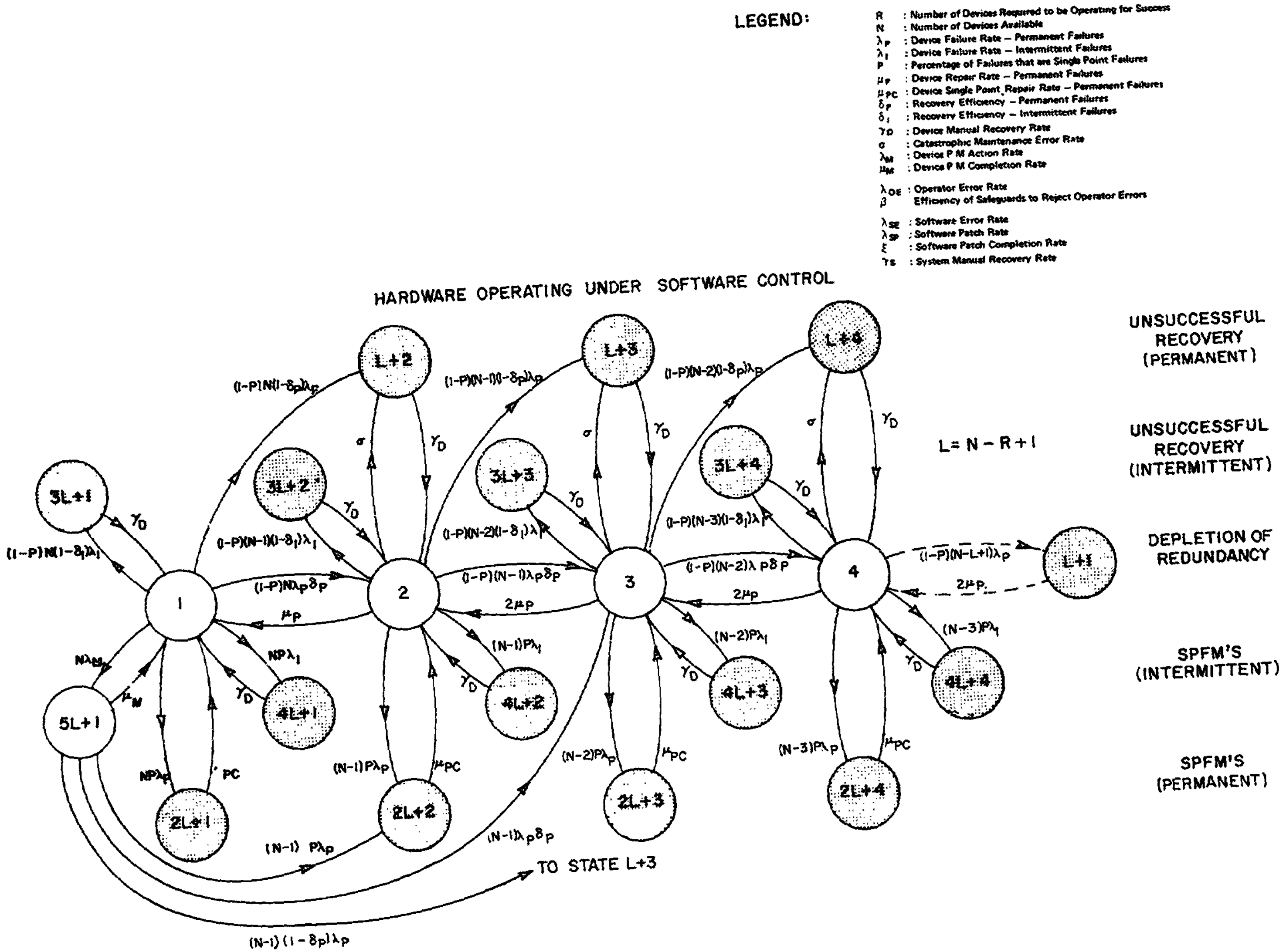
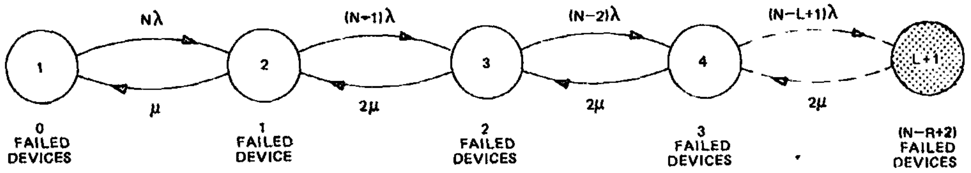

LEGEND:
R : Number of Devices Required to be Operating for Success
N : Number of Devices Available
λ_P : Device Failure Rate – Permanent Failures
λ_I : Device Failure Rate – Intermittent Failures
P : Percentage of Failures that are Single Point Failures
μ_P : Device Repair Rate – Permanent Failures
μ_{PC} : Device Single Point Repair Rate – Permanent Failures
δ_P : Recovery Efficiency – Permanent Failures
δ_I : Recovery Efficiency – Intermittent Failures
γ_D : Device Manual Recovery Rate
σ : Catastrophic Maintenance Error Rate
λ_M : Device P M Action Rate
μ_M : Device P M Completion Rate
λ_{OE} : Operator Error Rate
β Efficiency of Safeguards to Reject Operator Errors
λ_{SE} : Software Error Rate
λ_{SP} : Software Patch Rate
ξ : Software Patch Completion Rate
γ_S : System Manual Recovery Rate
HARDWARE OPERATING UNDER SOFTWARE CONTROL
UNSUCCESSFUL RECOVERY (PERMANENT)
UNSUCCESSFUL RECOVERY (INTERMITTENT)
DEPLETION OF REDUNDANCY
SPFM'S (INTERMITTENT)
SPFM'S (PERMANENT)
L = N − R + 1
1
2
3
4
L+1
L+2
L+3
L+4
2L+1
2L+2
2L+3
2L+4
3L+1
3L+2
3L+3
3L+4
4L+1
4L+2
4L+3
4L+4
5L+1
$(1-P)N(1-\delta_I)\lambda_I$
γ_D
$(1-P)N(1-\delta_P)\lambda_P$
$(1-P)(N-1)(1-\delta_P)\lambda_P$
$(1-P)(N-2)(1-\delta_P)\lambda_P$
σ
$(1-P)(N-1)(1-\delta_I)\lambda_I$
$(1-P)(N-2)(1-\delta_I)\lambda_I$
$(1-P)(N-3)(1-\delta_I)\lambda_I$
$(1-P)N\lambda_P\delta_P$
μ_P
$(1-P)(N-1)\lambda_P\delta_P$
$2\mu_P$
$(1-P)(N-2)\lambda P\delta P$
$(1-P)(N-L+1)\lambda_P$
$2\mu_P$
$NP\lambda_I$
$(N-1)P\lambda_I$
$(N-2)P\lambda_I$
$(N-3)P\lambda_I$
$N\lambda_M$
μ_M
$NP\lambda_P$
μ_{PC}
$(N-1)P\lambda_P$
$(N-2)P\lambda_P$
$(N-3)P\lambda_P$
$(N-1)\ P\lambda_P$
$(N-1)\lambda_P\delta_P$
TO STATE L+3
$(N-1)(1-\delta_P)\lambda_P$

Figure D.2. Markov Graph of the DESIGN Program Model

D.2.2 Conventional Failure and Repair Cycle Model

The Markov Graph shown in Figure D.3 is typical of conventional failure and repair cycle models for repairable redundant systems where the mechanism for removing failed hardware elements from the system and replacing repaired hardware elements are tacitly assumed to be perfect. As shown in Figure D.3 the number of states in the Markov Graph is a variable since each subsystem may contain 0, 1, 2, 3 or more active redundant hardware devices. If, for instance, a subsystem contains two active identical devices (n), only one of which is required to be operating for subsystem success (R), State L+1 becomes State 3 (a DOWN state) which terminates the chain since L+1=N-R+2, or L=N-R+1.

For the subsystem with one redundant device, it is common to hypothesize that at least two device failures must occur before a subsystem failure can occur. Normally, the Mean Time to Repair (MTTR) of a device is very short compared with the Mean Time Between Failures (MTBF); therefore, many allowable device failures are expected to occur before a subsystem failure occurs due to a second device failure during the time when a failed device is being repaired. Thus, on the surface it appears that tremendous gains are in store if sufficient redundancy is provided for critical devices in each subsystem since the probability of a second failure during a repair cycle is a rare event.



R: NUMBER OF DEVICES REQUIRED TO BE OPERATING FOR SUCCESS
N: NUMBER OF DEVICES AVAILABLE
λ: DEVICE FAILURE RATE
μ: DEVICE REPAIR RATE

Figure D.3. Simplified Markov Graph for Depletion of Redundancy

As shown in Figure D.3, "N" subsystem devices are operating successfully in State 1. Therefore, the rate at which device failures occur is "N" times the failure rate (λ) of a single device, since all devices are required to be identical. In State 2 one device is failed and either of the following two transitions may occur:

- The failed device may be repaired, at rate μ, before a second failure occurs and placed back into service, returning the subsystem to State 1.
- A second failure may occur before the repair is complete, further degrading the subsystem to State 3.

In State 2, failures occur at rate (N-1) times λ since one device is already being repaired. The rate at which failures occur in subsequent states diminishes as shown in Figure D.3 until the subsystem contains an inadequate number of hardware devices to sustain acceptable functional operation. Once the subsystem is in a failed state, it is assumed that operations cease and no additional failures occur.

Since program DESIGN is intended to investigate system design potential, an ideal support environment is assumed in which replacement spares, trained repairmen, documentation, test equipment, etc., are all immediately available when required. As shown in Figure D.3 the rate at which repairs are enacted when one device is failed is μ, and the rate at which repairs are enacted when more than one device is failed is 2μ. The underlying assumptions for the coefficients of μ are that only one repairman will be assigned to a failed device and that the maximum number of repairmen available for assignment to a failed subsystem is two. Thus, if one hardware device fails, one repairman goes to work; only when two or more devices require repair are both available repairmen busy. Normally, the probabilities associated with degradation to states where more than one or two devices require service simultaneously are very small.

D.2.3 Design Model for Hardware Elements Operating Under Software Control

There are several critical factors involved in adding redundant devices in computer subsystems which tend to severely reduce the potential benefits of hardware redundancy. First, the mechanism for automatically detecting, isolating, and switching failed devices out of the system and adding repaired devices back into the system is a complex interdisciplinary design problem. Also, clocks, controllers, busses and interface circuitry between hardware devices tend to contain Single Point Failure Modes (SPFM's) which cause subsystem failures even though the subsystem is not depleted of sufficient hardware resources. Unlike some types of systems, both permanent type failures which require a repair action and intermittent type failures which disappear before being isolated must carefully be considered in designing computer systems since either can cause abnormal subsystem interruption. For continuous operation, the problem of performing scheduled maintenance actions becomes an important consideration, and safeguards are necessary to prevent accidental system interruption when unscheduled maintenance actions are being performed on failed devices which cannot be physically disconnected from the system.

Referring to the Markov Graph for hardware operating under software control in Figure D.2, it can be seen that the center portion labeled "Depletion of Redundancy" corresponds closely to the previously discussed simplified Markov Graph. As before, the number of states is a variable depending upon how many redundant devices are provided in the subsystem. Failure states for permanent and intermittent type failures related to the recovery process and SPFM's are organized in line with the labels on the right-hand side of Figure D.2. State 5L+1 in Figure D.2 provides for considering scheduled maintenance actions in systems where continuous operation is desired, and consideration of maintenance errors during unscheduled maintenance actions is factored into states where repair actions are being performed while the system is still operating.

Considering first only the depletion of redundancy, the Markov Graph for hardware subsystems operating under software control is based on the premise that if a redundant hardware device fails, the particular subsystem involved may be interrupted for a negligible time to effect automatic reconfiguration. After automatically decommitting the failed device, the subsystem is immediately restored to operation, and the failed hardware device is then repaired. When repair of the failed hardware device is completed, it is recommitted to the subsystem without any discernable interruption in subsystem service. However, if more than the specified allowable number of hardware devices are down for repair, or if a critical hardware device has failed, then the subsystem is down until the appropriate repair has been effected.

As previously discussed, the five primary sources of system interruption and downtime diagrammed in Figure D.1 for hardware operating under software control are:

- Depletion of Adequate Resources
- Unsuccessful Recovery (Intermittent)
- Unsuccessful Recovery (Permanent)
- Single Point Failure Modes (Intermittent)
- Single Point Failure Modes (Permanent)

Starting in State 1, "N" identical, independent subsystem devices are operating successfully. Therefore, the rate at which failures occur (either permanent, λ_P, or intermittent, λ_I) is N times the failure rate of a single device. If no redundancy is provided, any failure causes a subsystem failure. With redundancy, when a failure occurs in State 1, any of the following transitions may occur.

- If the failure is a permanent type failure related to an SPFM, no recovery is possible and there is a transition to State 2L+1, which requires a repair action to reestablish subsystem operation. The rate at which SPFM repairs are enacted is μ_{PC}. When the repair is completed in State 2L+1, there is a transition back to State 1, where the subsystem is again operating successfully with all hardware devices present.

- If the failure is an intermittent type failure to an SPFM, no recovery is possible and there is a transition to State 4L+1. Since intermittent failures do not require a repair action, the device is returned to the subsystem and there is a transition from State 4L+1 back to State 1 at a rate γ_D, which is the device manual recovery rate including the time required for the intermittent failure to disappear.

- If the failure is a permanent type failure and the automatic recovery system is successful, there is a transition to State 2, in which case the subsystem continues to operate with one device decommitted from the subsystem. If no additional events occur before the failed device is repaired and recommitted to the subsystem, there is a transition back to State 1. These transitions occur at rate μ_P, which is the device repair rate for permanent type failures. Additional events in State 2 will be discussed subsequently.

- If the failure is a permanent type failure and the automatic recovery system is unsuccessful, there is a transition to State L+2. In State L+2, manual recovery procedures are enacted at rate γ_D and there is a transition to State 2 where the subsystem is operating with one device decommitted from the subsystem. As indicated above, State 2 will be discussed subsequently.

- If the failure is an intermittent type failure and the automatic recovery system is unsuccessful, there is a transition to State 3L+1. Again, since intermittent type failures do not require a repair action, the device is returned to the subsystem and there is a transition back to State 1 at rate γ_D.

- For systems which are required to operate continuously, State 1 provides the best opportunity for performing any required scheduled maintenance on subsystem hardware devices. Therefore, scheduled maintenance is restricted to being performed only when all subsystem hardware devices are operating successfully. When a hardware device is decommitted for scheduled maintenance, there is a transition to State 5L+1 where the subsystem is operating successfully, but depleted of one of its hardware resources. If the scheduled maintenance is completed (rate μ_M) and returned to the subsystem before an event occurs, there is a transition back to State 1. However, if an event occurs before the scheduled maintenance action is completed any of the following may occur:

  - To State of 2L+2 if the event is related to permanent type SPFM (Subsystem DOWN)
  - To State L+3 if the event is related to a permanent type failure and automatic recovery is unsuccessful (Subsystem DOWN)
  - To State 3 if the event is related to a permanent type failure and automatic recovery is successful (Subsystem UP) provided the subsystem contains two or more redundant devices, otherwise, State 3 is a DOWN state.

In State 2, subsystems with one or more redundant devices are operating successfully with one failed device decommitted from the subsystem. Therefore, the rate at which events related to the number of hardware devices occur diminishes to a multiplier of N-1. The subsystem operates essentially as described for State 1 except that the failed device being repaired may be a hazard to subsystem operation. If the failed device is not disconnected from the subsystem and safeguards are inadequate, a maintenance error could occur which brings the subsystem down. In State 2, the transition from state 2 to to State L+2 accounts for this potential mode. As shown, the rate at which catastrophic maintenance errors occur is designated as σ.

D.3 MATHEMATICAL MODEL FOR THE CONFIGURE PROGRAM

In contrast to the traditional two-state failure and repair cycle reliability model, the CONFIGURE program employs the three-state model shown in Figure D.4 which enables the effects of manual recovery from non-permanent failures and errors to be taken into consideration. This separation of repair and nonrepair events is the key to modeling the effects of intermittent failures, software errors, maintenance errors, unisolated events, and unsuccessful automatic recoveries in close approximation to the physical system being analyzed.

As shown in Figure D.4, when an element fails, a transition occurs from the UP state into either the REPAIR state or the INTERRUPT state. The rate at which these transitions occur is the reciprocal of the element Mean Up Time (1/MUT). Variable F1 defines the fraction of failures or errors which cause a transition directly into the INTERRUPT state. Hence, (1-F1) is the fraction of failures that cause a transition directly into the INTERRUPT state. Hence, (1-F1) is the fraction of failures that cause a transition directly into the REPAIR state.

Once an element is in the REPAIR state, the only possible transition is to the UP state. The rate at which this transition occurs is, of course, the reciprocal of the element Mean Repair Time (1/MRT). When an element is in the INTERRUPT state, transitions to either the REPAIR state or the UP state can occur. Variable F2 is the fraction of total interrupt events which go into the REPAIR state rather than going directly into the UP state, and the reciprocal of the Mean Interrupt Time (1/MINT) is the rate at which these transitions occur.

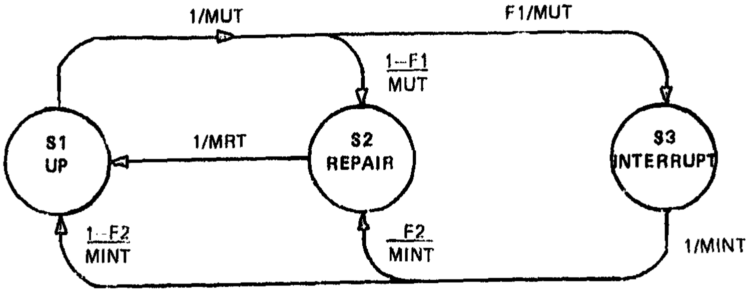
For hardware subsystems, the subsystem is considered to be operating successfully if every element is in the UP state, and the subsystem is considered to be down if any element causes a transition to the INTERRUPT state. The subsystem can be operating succesfully with some of the hardware elements in the REPAIR state. This depends on the definition of how many hardware elements of the subsystem can be in the REPAIR state with the subsystem still capable of performing its intended function. For critical subsystems, the subsystem is considered to be operating successfully only when no repair action or interruption are in process. Thus, for unisolated events, operator errors, and maintenance errors which require no repair action, transitions from the UP state go directly into the INTERRUPT state. In this case, restoration of system operation is accomplished by a manual recovery action. Software errors which disappear follow this same pattern. However, if a software patch is required, the repair state becomes involved in a manner analogous to the situation discussed with respect to permanent type hardware failures.

The summary table provided below the transition diagram outlines conditions in states S_1, S_2, and S_3, and defines the type of recovery required for the specified conditions.

The critical assumptions associated with the derivation of the state probability equations shown in Figure D.4 are:

- Failure and repair hazards are assumed to be constant, which is equivalent to stating that individual elements are assumed to fail in accordance with the negative exponential distribution, and the times to repair are also exponentially distributed.

TRANSITION DIAGRAM



STATE	SUBSYSTEMS	ELEMENTS	RECOVERY
S1	UP	N UP	NONE
S2	UP DOWN	UP ≥ R UP < R	AUTOMATIC MANUAL
S3	DOWN	: DOWN	MANUAL

$$PU = \frac{1}{1 + F1\,\frac{MINT}{MUT} + ((1 - F1) + F1\ F2)\,\frac{MRT}{MUT}}$$ = PROBABILITY OF BEING IN THE UP STATE

$$PI = F1\,\frac{MINT}{MUT}\,PU$$ = PROBABILITY OF BEING IN THE INTERRUPT STATE

$$PR = ((1 - F1) + F1\ F2)\,\frac{MRT}{MUT}\,PU$$ = PROBABILITY OF BEING IN THE REPAIR STATE

$$MUT = \frac{TOTAL\ TIME}{NE}$$

$$MRT = \frac{TOTAL\ REPAIR\ TIME}{NRE}$$

$$MINT = \frac{TOTAL\ SYSTEM\ INTERRUPT\ TIME}{NIE}$$

MUT = MEAN UP TIME
MRT = MEAN REPAIR TIME
MINT = MEAN INTERRUPT TIME

NE = NUMBER OF OBSERVED EVENTS
NRE = NUMBER OF REPAIR EVENTS
NIE = NUMBER OF INTERRUPT EVENTS

F1 = FRACTION OF TOTAL FAILURES WHICH CAUSE AN INTERRUPT

F2 = FRACTION OF TOTAL INTERRUPTS WHICH REQUIRE A REPAIR ACTION

Figure D.4. Three-State Model -- CONFIGURE Program

- Each subsystem element is completely independent of other elements in the subsystem.

- Each subsystem element is identical in any given subsystem.

D.4 DEFINITIONS

The definitions summarized below are provided for reference to aid in interpreting computed results.

D.4.1 Program Inputs

The following definitions pertain to Program inputs. Each definition describes a program variable. The symbol used is given in brackets following the definitions. In cases where these symbols are reciprocals of the various transition rates discussed, an equivalence relationship is given which correlates the symbols in Figure D.2 with the input data symbols.

- DEVICES REQUIRED. Minimum number of identical subsystem devices required to be working for acceptable subsystem operation (R).

- DEVICES AVAILABLE. Number of identical subsystem devices provisioned for active subsystem operation (N).

- TIME BETWEEN FAILURES (PERMANENT). Time interval from an instant when a repairable device is working to the next intermittent type device failure which requires a manual recovery action (mean: $\mathrm{MTBF(P)} = I/\lambda_P$).

- TIME BETWEEN FAILURES (INTERMITTENT). Time interval from an instant when a repairable device is working to the next intermittent type device failure which requires a manual recovery action (mean: $\mathrm{MTBF(I)} = I/\lambda_I$).

- SINGLE POINT FAILURES*. Percentage of total failures in a redundant subsystem configuration which result in subsystem failures (permanent or intermittent) even though an adequate number of devices are working (SPFM = P).

- DEVICE REPAIR TIME. Time interval from an instant when repair of a device is initiated to readiness as an active subsystem device, excluding waiting times for repairmen, spares, etc. (mean: $\mathrm{DRT} = 1/\mu_P$).

- SINGLE POINT REPAIR TIME*. Time interval from an instant when repair of a single point failure is initiated to readiness of associated subsystem, excluding waiting times for repairmen, spares, etc. (mean: $\mathrm{SRT} = 1/\mu_{PC}$).

- RECOVERY EFFICIENCY (PERMANENT). Percentage of automatic recovery actions from permanent type failures which are completed successfully (without manual intervention) within a negligible period of time (RE(P) =δ_P).

- RECOVERY EFFICIENCY (INTERMITTENT). Percentage of automatic recovery actions from intermittent type failures which are completed successfully (without manual intervention) within a negligible period of time (RE(I) -δ_I).

- DEVICE MANUAL RECOVERY TIME. Time interval from an instant when a system failure related to an unsuccessful automatic recovery from a device failure (permanent or intermittent) occurs until the system is restored to normal operation via manual recovery procedures (mean: DMRT = $1/\gamma_P$).

- TIME BETWEEN MAINTENANCE ERRORS. Time interval from an instant when a system recovery related to a maintenance error is completed until the next occurrance of a maintenance error which causes system interruption (mean: MTBME = $1/\sigma$).

- TIME BETWEEN PREVENTIVE MAINTENANCE ACTIONS. Time interval from an instant when a device has been recommitted to active operation following a scheduled preventive maintenance action unti the next scheduled preventive maintenance action is due (mean: MTBPM = $1/\lambda_M$).

- TIME TO PERFORM PREVENTIVE MAINTENANCE. Time interval from an instant when a device is decommitted from active operation (mean: MTTPM = $1/\mu_M$).

- SYSTEM MANUAL RECOVERY TIME. Time interval from an instant when a system failure related to a transient software or operator error occurs until the system is restored to normal operation via manual recovery procedures (mean: SMRT = $1/\gamma_S$).

* This variable is provded for convenience in preparing inital estimtes; if desired, SPFM's can be modeled as a single non-redundant device in series with the associated subsystem.

D.4.2 Program Outputs

The following six useful measures of system performance are frequently encountered in modeling and analyzing repairable, redundant system configurations:

- Availability
- Mean Up Time (MUT)
- Mean Down Time (MDT)
- Mean Cycle Time (MCT)
- Mean Time to First Failure (MTFF)
- Mean Time to Failure (MTTF)

Although the terminology given above is common in the field of reliability, the precise meaning of the terms can easily become confused. The following definitions of these terms appear in the paper by Buzacott [1]. These definitions have been extracted almost directly since the unified presentation in the Buzacott paper tends to relieve much of the misunderstanding encountered regarding the various mean times of interest and the concept of point and interval availability in treating repairable, redundant systems. Only the MUT, MDT, and interval availability definitions are directly applicable to the outputs of the DESIGN program. The remaining definitions are provided for reference to further clarify the precise meanings of MUT, MDT, and interval availability.

D.4.2.1 System Availability

Let SYSTEM INITIATION be the instant when system operation begins for the first time. The system and all of its components are assumed to be working correctly and not to be subject to wearout during the time interval of interest.

Let SYSTEM FAILURE be the instant when the system changes from working to failed. Let SYSTEM REPAIR be the instant when the system changes from failed to working.

The first group of definitions applies to the concept of availability. POINT AVAILABILITY (at time t): The probability that the system is working at time t from system initiation. It is assumed that at time t no information is available about system failures and system repairs during the time interval (0, t) (symbol $P_W(t)$). INTERVAL AVAILABILITY (at time t): The expected proportion of the time interval from system initiation (time 0) to time t during which the system is working. (symbol I(t)). Hence

$$I(t) = \frac{1}{t}\int_0^t P_W(u)du \qquad \text{D.1}$$

It is assumed in calculating all quantities except the mean of the time to first failure that the system has been operating long enough for initial effects to have died away; i.e., statistical equilibrium (or the asymptotic behavior) has been reached. The mean cycle time is then obtained from the ratio of the length of some long time interval to the number of failures N in that time interval, i.e.,

$$\mathrm{MCT} = \lim_{Y \to \infty} \left\{ Y/N \right\}$$

and

$$\mathrm{AV} = \text{Availability} = \frac{\mathrm{MUT}}{\mathrm{MCT}}$$

Asymptotically, for large t, it can be shown that point availability is numerically the same as interval availability. Thus

$$A = \lim_{t \to \infty} \left\{ P_w(t) \right\} = \lim_{t \to \infty} \left\{ I(t) \right\}$$

This steady-state availability is the availability calculated herein.

D.4.2.2 System Time Interval Between Failures

The next group of definitions refer to the concepts related to the time interval between failures. Each definition defines a random variable. The symbol that will be used herein for the mean is given in parentheses following the definition.

- UP TIME. Time interval from system repair to next system failure (mean: MUT).

- DOWN TIME. Time interval from system failure to next system repair (mean: MDT).

- CYCLE TIME. Time interval from one system failure to the next system failure. The cycle time is the sum of an Up Time and a Down Time (mean: MCT).

- TIME TO FIRST FAILURE. Time interval from system initiation (mean: MTTF).

The first three time intervals, up time, down time, and cycle time apply to a system that is alternately working, failed, working, failed, and so on. The system is said to be repaired when sufficient, but not necessarily all components are repaired so that the system is working.

APPENDIX D REFERENCES

1. Buzacott, J.A.; "Markov Approach to Finding Failure Times of Repairable System," IEEE Transactions on Reliability, Vol. R-19 No. 4 November, 1970.

APPENDIX E
FMP RELIABILITY DATA BASE

This Appendix presents the predicted results of the MTBF's and failure rates for the elements of the FMP system (see figures E.1, E.2, and E.3.). Three sets of results are shown, the difference being the failure rate and SECDED improvement factors used for the LSI memory circuits.

The form of these predictions is a hierarchical structure. Each FMP element is listed as a level 01; the parts constituting the elements are listed as level 02. For a better defined system, the number of levels may increase, showing the assemblies that make up an element as level 02, the subassemblies as level 03 and the components in the subassemblies as level 04, etc.

For each item listed in an element, a part number and description are provided. For the FMP, hypothetical part numbers are used. In the case of the I.C.'s, typical parts in the generic family assumed have been selected to represent all the I.C's used in the various logic functions.

The quantity of each part in the structured listing is shown. The quantity is multiplied by the quantity of the encompassing level item. For example, where a quantity of 2 of an assembly or element is listed and a quantity of 3 of a particular part in an assembly or element is required, the quantity listed for this part will be 6.

Failure rates for each individual part and the total quantity of that part are shown and expressed in failures per million hours (FPMH). The aggregate of these failures are used to predict the failure rate of the element and the mean time between failures (MTBF) in hours. The failure rates used except for the LSI memories have been developed from the guidelines in MIL HDBK 217B.

While not used in this study, columns for the spares confidence level are shown. When used, these data indicate the maximum number of repair actions (and therefore spare elements) required at a specific confidence level for specified number of years and duty cycles.

```
ASMET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:PROC MODE  PROC MODE
MTBF=  16497.23 HOURS, FAILURE RATE=    60.6162 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD CNS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0


   LEVEL        PART NUMBER DESCRIPTION                        QUAN    FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                         UNIT       TOTAL       70      80      90      95      99    99.5
01 -------------PROC MODE  PROC MODE                              1     0.0000    60.6162       1       1       1       2       3       3
  02 -----------FMP 1060       * ECL LOGIC IC 6024         DIG  100     0.2816    28.1600
  02 -----------FMP RC         * RC MODULE                MODR   40     0.0540     2.1600
  02 -----------FMP 2010         ELECTRO CAP               CSR   10     0.0786     0.7856
  02 -----------FMP 2020         CER CAP                   CK1   25     0.0430     1.0740
  02 -----------FMP 4010       * 16K RAM                   RAM  110     0.2000    22.0000
  02 -----------FMP 1040       * ECL LOGIC IC 1016         DIG   34     0.0377     1.2819
  02 -----------FMP 1030       * ECL LOGIC IC 0618         DIG   30     0.0292     0.8760
  02 -----------IO CONN          IO CONNECTOER            CONN    1     2.5954     2.5954
  02 -----------COAX CONN        COAX CONNECTOR           CONN   14     0.1033     1.4458
  02 -----------PWR CONN         POWER CONNECTOR          CONA    1     0.2377     0.2377
ASMET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:FANOUT TREE FANOUT TREE
MTBF=   2783.72 HOURS, FAILURE RATE=   359.2309 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD CNS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0


   LEVEL        PART NUMBER DESCRIPTION                        QUAN    FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                         UNIT       TOTAL       70      80      90      95      99    99.5
01 -------------FANOUT TREE FANOUT TREE                          32     0.0000   359.2309       4       5       5       6       8       9
  02 -----------FMP 1020       * ECL LOGIC IC 0615         DIG 2560     0.0288    73.7290
  02 -----------FMP RC         * RC MODULE                MODR 1280     0.0540    69.1200
  02 -----------FMP 2010         ELECTRO CAP               CSR  640     0.0786    50.2773
  02 -----------IO CONN          IO CONNECTOER            CONN   64     2.5954   166.1056
ASMET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:COOR UNIT   COOR UNIT
MTBF=   4672.38 HOURS, FAILURE RATE=   214.0235 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD CNS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0


   LEVEL        PART NUMBER DESCRIPTION                        QUAN    FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                         UNIT       TOTAL       70      80      90      95      99    99.5
01 -------------COOR UNIT   COOR UNIT                            20     0.0000   214.0235       2       3       4       4       5       6
  02 -----------FMP 1010       * ECL LOGIC IC 0415         DIG 2000     0.0235    47.0000
  02 -----------FMP RC         * RC MODULE                MODR  800     0.0540    43.2000
  02 -----------FMP 2010         ELECTRO CAP               CSR  200     0.0786    15.7117
  02 -----------FMP 2020         CER CAP                   CK1  100     0.0430     4.2958
  02 -----------IO CONN          IO CONNECTOER            CONN   40     2.5954   103.8160
ASMET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:COR U MEM   CORE U MEM
MTBF=  20963.13 HOURS, FAILURE RATE=    47.7029 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD CNS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0


   LEVEL        PART NUMBER DESCRIPTION                        QUAN    FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                         UNIT       TOTAL       70      80      90      95      99    99.5
01 -------------COR U MEM   CORE U MEM                            2     0.0000    47.7029       1       1       1       2       2       3
  02 -----------FMP 4010       * 16K RAM                   RAM  200     0.2000    40.0000
  02 -----------FMP RC         * RC MODULE                MODR   60     0.0540     3.2400
  02 -----------FMP 2010         ELECTRO CAP               CSR   20     0.0786     1.5712
  02 -----------COAX CONN        COAX CONNECTOR           CONN   28     0.1033     2.8916
ASMET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:FMP NETWK   FMP NETWK
MTBF=    188.18 HOURS, FAILURE RATE=  5314.1929 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD CNS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0
```

Figure E.1 FMP Reliability Data - Lower Bound

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

```
   LEVEL         PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                           UNIT       TOTAL      70     80     90     95     99   99.5
01 -------------CON NETWK   CON NETWK                            200     0.0000   5314.1929      50     52     55     58     63     65
  02 -----------FMP 1050        * ECL LOGIC IC 3224        DIG 20000     0.1883   3766.0000
  02 -----------FMP RC          * RC MODULE               MODR  6000     0.0540    324.0000
  02 -----------FMP 2010          ELECTRO CAP              CSR  2000     0.0786    157.1166
  02 -----------IO CONN           IO CONNECTOER           CONN   400     2.5954   1038.1598
  02 -----------COAX CONN         COAX CONNECTOR          CONN   280     0.1033     28.9164
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:EXT MEM FO  EXT MEM FANOUT
MTBF=   2466.11 HOURS, FAILURE RATE=   405.4972 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL         PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                           UNIT       TOTAL      70     80     90     95     99   99.5
01 -------------EXT MEM FO  EXT MEM FANOUT                        32     0.0000    405.4972       4      5      6      7      9      9
  02 -----------FMP 1020        * ECL LOGIC IC 0616        DIG  2560     0.0288     73.7280
  02 -----------FMP RC          * RC MODULE               MODR  1280     0.0540     69.1200
  02 -----------FMP 2010          ELECTRO CAP              CSR   640     0.0786     50.2773
  02 -----------IO CONN           IO CONNECTOER           CONN    64     2.5954    166.1056
  02 -----------COAX CONN         COAX CONNECTOR          CONN   448     0.1033     46.2663
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:EXM MEM     EXT MEM MODUL
MTBF=  19025.37 HOURS, FAILURE RATE=    52.5614 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL         PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                           UNIT       TOTAL      70     80     90     95     99   99.5
01 -------------EXM MEM     EXT MEM MODUL                          1     0.0000     52.5614       1      1      1      2      3      3
  02 -----------FMP 4020        * 64K RAM                  RAM    56     0.8000     44.8000
  02 -----------FMP 1010        * ECL LOGIC IC 0416        DIG    30     0.0235      0.7050
  02 -----------FMP RC          * RC MODULE               MODR    20     0.0540      1.0800
  02 -----------FMP 2010          ELECTRO CAP              CSR    10     0.0786      0.7856
  02 -----------IO CONN           IO CONNECTOER           CONN     2     2.5954      5.1908
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:DBM CONTR   DBM CONTR
MTBF=  11909.89 HOURS, FAILURE RATE=    83.9638 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL         PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                           UNIT       TOTAL      70     80     90     95     99   99.5
01 -------------DBM CONTR   DBM CONTR                             10     0.0000     83.9638       1      1      2      2      3      4
  02 -----------FMP 1010        * ECL LOGIC IC 0416        DIG   800     0.0235     18.8000
  02 -----------FMP RC          * RC MODULE               MODR   100     0.0540      5.4000
  02 -----------FMP 2010          ELECTRO CAP              CSR   100     0.0786      7.8558
  02 -----------IO CONN           IO CONNECTOER           CONN    20     2.5954     51.9080
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:DBM MEM     DBM MEM
MTBF=     50.43 HOURS, FAILURE RATE= 19831.0348 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL         PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)      SPARES CONFIDENCE LEVEL(PERCENT)
                                                                           UNIT       TOTAL      70     80     90     95     99   99.5
01 -------------DBM MEM     DBM MEM                               48     0.0000  19831.0348     177    180    184    189    195  [illegible]
  02 -----------FMP 4030        * 256K RAM                 RAM  6144     3.2000  19660.8000
  02 -----------FMP 1010        * ECL LOGIC IC 0416        DIG   672     0.0235     15.7920
  02 -----------FMP 1020        * ECL LOGIC IC 0616        DIG   816     0.0288     23.5008
  02 -----------FMP RC          * RC MODULE               MODR    48     0.0540      2.5920
  02 -----------FMP 2010          ELECTRO CAP              CSR    48     0.0786      3.7728
  02 -----------IO CONN           IO CONNECTOER           CONN    48     2.5954    124.5792
```

Figure E.1 FMP Reliability Data - Lower Bound (continued)

```
ASDPT  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:PROC MODUL  PROC MODUL
MTBF=  25432.20 HOURS, FAILURE RATE=    39.3202 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION                    QUAN    FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                     UNIT       TOTAL     70     80     90     95     99   99.5
01 -------------PROC MODUL  PROC MODUL                         1   0.0000    39.3202     0      1      1      1      2      3
  02 -----------FMP 1060        * ECL LOGIC IC 6024    DIG   100   0.2816    28.1600
  02 -----------FMP RC          * RC MODULE            MODR   40   0.0540     2.1600
  02 -----------FMP 2010          ELECTRO CAP          CSR    10   0.0786     0.7856
  02 -----------FMP 2020          CER CAP              CK1    25   0.0430     1.0749
  02 -----------FMP 4010        * 16K RAM              RAM   110   0.0064     0.7040
  02 -----------FMP 1040        * ECL LOGIC IC 1016    DIG    34   0.0377     1.2815
  02 -----------FMP 1030        * ECL LOGIC IC 0618    DIG    30   0.0292     0.8760
  02 -----------IO CONN           IO CONNECTOR         CONN    1   2.5954     2.5954
  02 -----------COAX CONN         COAX CONNECTOR       CONN   14   0.1033     1.4459
  02 -----------PWR CONN          POWER CONNECTOR      CONN    1   0.2377     0.2377
ASDPT  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:FANOUT TREE FANOUT TREE
MTBF=   2783.72 HOURS, FAILURE RATE=   359.2309 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION                    QUAN    FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                     UNIT       TOTAL     70     80     90     95     99   99.5
01 -------------FANOUT TREE FANOUT TREE                       32   0.0000   359.2309     4      5      5      6      8      9
  02 -----------FMP 1020        * ECL LOGIC IC 0616    DIG  2560   0.0288    73.7280
  02 -----------FMP RC          * RC MODULE            MODR 1280   0.0540    69.1200
  02 -----------FMP 2010          ELECTRO CAP          CSR   540   0.0786    50.2773
  02 -----------IO CONN           IO CONNECTOR         CONN   64   2.5954   166.1055
ASDPT  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:COOR UNIT   COOR UNIT
MTBF=   4672.38 HOURS, FAILURE RATE=   214.0235 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION                    QUAN    FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                     UNIT       TOTAL     70     80     90     95     99   99.5
01 -------------COOR UNIT   COOR UNIT                         20   0.0000   214.0235     2      3      4      4      5      6
  02 -----------FMP 1010        * ECL LOGIC IC 0416    DIG  2000   0.0235    47.0000
  02 -----------FMP RC          * RC MODULE            MODR  800   0.0540    43.2000
  02 -----------FMP 2010          ELECTRO CAP          CSR   200   0.0786    15.7117
  02 -----------FMP 2020          CER CAP              CK1   100   0.0430     4.2958
  02 -----------IO CONN           IO CONNECTOR         CONN   40   2.5954   103.8160
ASDPT  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:COR M MEM   CORE M MEM
MTBF= 111323.73 HOURS, FAILURE RATE=     8.9828 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION                    QUAN    FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                     UNIT       TOTAL     70     80     90     95     99   99.5
01 -------------COR M MEM   CORE M MEM                         2   0.0000     8.9829     0      0      0      1      1      1
  02 -----------FMP 4010        * 16K RAM              RAM   200   0.0064     1.2800
  02 -----------FMP RC          * RC MODULE            MODR   60   0.0540     3.2400
  02 -----------FMP 2010          ELECTRO CAP          CSR    20   0.0786     1.5712
  02 -----------COAX CONN         COAX CONNECTOR       CONN   28   0.1033     2.8916
ASDPT  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:COM NETWK   COM NETWK
MTBF=    188.18 HOURS, FAILURE RATE=  5314.1929 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0
```

Figure E.2 FMP Reliability Data - Probable Case

```
   LEVEL        PART NUMBER DESCRIPTION
                                                               QUAN    FAILURE RATE(FMPH)        SPARES CONFIDENCE LEVEL(PERCENT)
 01 -------------CON NETWK   CON NETWK                                  UNIT       TOTAL        70     80     90     95     99    99.5
   02 -----------FMP 1050         * ECL LOGIC IC 3224             200   0.0000   5314.1929       50     52     55     59     62     65
   02 -----------FMP RC           * RC MODULE              DIG  20000   0.1883   3766.0000
   02 -----------FMP 2010           ELECTRO CAP            MODR  6000   0.0540    324.0000
   02 -----------IO CONN            IO CONNECTOER          CSR   2000   0.0786    157.1166
   02 -----------COAX CONN          COAX CONNECTOR         CONN   400   2.5954   1029.1598
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION        CONY   290   0.1033     29.9164
FOR:EXT MEM FO   EXT MEM FANOUT
MTBF=    2466.11 HOURS, FAILURE RATE=   405.4972 PER MILLION HOURS(FMPH)
SPARING FOR   1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION
                                                               QUAN    FAILURE RATE(FMPH)        SPARES CONFIDENCE LEVEL(PERCENT)
 01 -------------EXT MEM FO  EXT MEM FANOUT                             UNIT       TOTAL        70     80     90     95     99    99.5
   02 -----------FMP 1020         * ECL LOGIC IC 0616              32   0.0000    405.4972        4      5      6      7      9      9
   02 -----------FMP RC           * RC MODULE              DIG   2560   0.0288     73.7280
   02 -----------FMP 2010           ELECTRO CAP            MODR  1280   0.0540     69.1200
   02 -----------IO CONN            IO CONNECTOER          CSR    640   0.0786     50.2773
   02 -----------COAX CONN          COAX CONNECTOR         CONN    64   2.5954    166.1056
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION        CONV   448   0.1033     46.2633
FOR:ETM MEM      EXT MEM MODUL
MTBF= 117949.81 HOURS, FAILURE RATE=      8.4782 PER MILLION HOURS(FMPH)
SPARING FOR   1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION
                                                               QUAN    FAILURE RATE(FMPH)        SPARES CONFIDENCE LEVEL(PERCENT)
 01 -------------ETM MEM      EXT MEM MODUL                              UNIT       TOTAL       70     80     90     95     99    99.5
   02 -----------FMP 4020         * 64K RAM                         1   0.0000      8.4782        0      0      0      1      1      1
   02 -----------FMP 1010         * ECL LOGIC IC 0416      RAM     56   0.0128      0.7168
   02 -----------FMP RC           * RC MODULE              DIG     30   0.0235      0.7050
   02 -----------FMP 2010           ELECTRO CAP            MODR    20   0.0540      1.0800
   02 -----------IO CONN            IO CONNECTOER          CSR     10   0.0786      0.7856
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION        CONN     2   2.5954      5.1908
FOR:DRM CONTR    DRM CONTR
MTBF=  11909.99 HOURS, FAILURE RATE=    83.9638 PER MILLION HOURS(FMPH)
SPARING FOR   1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION
                                                               QUAN    FAILURE RATE(FMPH)        SPARES CONFIDENCE LEVEL(PERCENT)
 01 -------------DRM CONTR   DRM CONTR                                  UNIT       TOTAL        70     80     90     95     99    99.5
   02 -----------FMP 1010         * ECL LOGIC IC 0416              10   0.0000     83.9638        1      1      2      2      3      4
   02 -----------FMP RC           * RC MODULE              DIG    800   0.0235     18.8000
   02 -----------FMP 2010           ELECTRO CAP            MODR   100   0.0540      5.4000
   02 -----------IO CONN            IO CONNECTOER          CSR    100   0.0786      7.8558
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION        CONN    20   2.5954     51.9081
FOR:DRM MEM      DRM MEM
MTBF=    3053.24 HOURS, FAILURE RATE=   327.5212 PER MILLION HOURS(FMPH)
SPARING FOR   1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION
                                                               QUAN    FAILURE RATE(FMPH)        SPARES CONFIDENCE LEVEL(PERCENT)
 01 -------------DRM MEM      DRM MEM                                   UNIT       TOTAL        70     80     90     95     99    99.5
   02 -----------FMP 4020         * 256K RAM                       48   0.0000    327.5212        4      4      5      6      7      8
   02 -----------FMP 1010         * ECL LOGIC IC 0416      RAM   6141   0.0256    157.2864
   02 -----------FMP 1020         * ECL LOGIC IC 0616      DIG    672   0.0235     15.7920
   02 -----------FMP RC           * RC MODULE              DIG    816   0.0288     23.5008
   02 -----------FMP 2010           ELECTRO CAP            MODR    48   0.0540      2.5920
   02 -----------IO CONN            IO CONNECTOER          CSR     48   0.0786      3.7728
                                                           CONN    48   2.5954    124.5792
```

Figure E.2 PMP Reliability Data - Probable Case (continued)

```
[illegible]
[illegible]
MTBF= [illegible] HOURS, FAILURE RATE= [illegible] PER MILLION HOURS(FPMH)
SPARING FOR 1.000 YEARS, DUTY CYCLE= 1.[illegible] , FRAC. AIR CYC= [illegible], QUANTITY=1 , MINIMUM SPARES= [illegible]

  LEVEL        PART NUMBER DESCRIPTION
                                                                 QUAN    FAILURE RATE(FPMH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL       70     80     90     95     99    99.5
 01 -------------[illegible]                                        1    [illegible] [illegible]   1      1      1      1      1      3
  02 ------------FMP [illegible]        * [illegible] IC 6000   [illegible]  11   0.2815     18.1600
  02 ------------[illegible]            * [illegible]          M[illegible]  40   0.1540      6.1600
  02 ------------FMP [illegible]          [illegible] CAP      [illegible]  10   0.0786      1.7866
  02 ------------FMP [illegible]          [illegible] CAP      [illegible]  28   0.0420      1.1760
  02 ------------FMP 4010               * 16K RAM              [illegible] 112   0.0026      0.2871
  02 ------------FMP 1040               * [illegible] IC 1014  [illegible]  34   0.0477      1.6212
  02 ------------FMP 1030               * [illegible] IC 0416  [illegible]  32   0.0002      1.3760
  02 ------------IC CONN                  IC CONNECTOR         [illegible]   1   2.5954      2.5954
  02 ------------COAX CONN                COAX CONNECTOR       [illegible]  14   0.1035      1.4459
  02 ------------PWR [illegible]          PWR CONNECTOR        [illegible]   1   0.1377      0.1377
ASSET [illegible] RELIABILITY AND SPARES PREDICTION
FOR:FAULT TREE FAN OUT TREE
MTBF=    2783.72 HOURS, FAILURE RATE=   359.2300 PER MILLION HOURS(FPMH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. AIR CYC=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

  LEVEL        PART NUMBER DESCRIPTION
                                                                 QUAN    FAILURE RATE(FPMH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL       70     80     90     95     99    99.5
 01 -------------FAULT TREE FAN OUT TREE                            32   0.0000    359.2300      4      5      5      6      8      9
  02 ------------FMP 1030        * ECL LOGIC IC 0610         DIG  2560   0.0288     73.7280
  02 ------------FMP 90          * RC MODULE                MODS  1280   0.0540     69.1200
  02 ------------FMP 2010          ELECTRO CAP               CAP   640   0.0786     50.2773
  02 ------------IC CONN           IC CONNECTOR             CONN    64   2.5954    166.1056
ASSET STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:FAN OUT UNIT   FAN OUT UNIT
MTBF=    4672.38 HOURS, FAILURE RATE=   214.0235 PER MILLION HOURS(FPMH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. AIR CYC=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

  LEVEL        PART NUMBER DESCRIPTION
                                                                 QUAN    FAILURE RATE(FPMH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL       70     80     90     95     99    99.5
 01 -------------COOR UNIT   COOR UNIT                              20   0.0000    214.0235      3      3      4      4      5      6
  02 ------------FMP 1010        * ECL LOGIC IC 0416         DIG  2000   0.0235     47.0000
  02 ------------FMP 90          * RC MODULE                MODS   800   0.0540     43.2000
  02 ------------FMP 2010          ELECTRO CAP               CAP   200   0.0786     15.7117
  02 ------------FMP 2020          CER CAP                   CAP   100   0.0420      4.2958
  02 ------------IC CONN           IC CONNECTOR             CONN    40   2.5954    103.8160
ASSET STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:COOR U MEM   CORE U MEM
MTBF= 127798.62 HOURS, FAILURE RATE=     7.8248 PER MILLION HOURS(FPMH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. AIR CYC=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

  LEVEL        PART NUMBER DESCRIPTION
                                                                 QUAN    FAILURE RATE(FPMH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL       70     80     90     95     99    99.5
 01 -------------CORE U MEM   CORE U MEM                             2   0.0000      7.8248      0      1      1      1      1      1
  02 ------------FMP 4010        * 16K RAM                   RAM   200   0.0006      0.1200
  02 ------------FMP 90          * RC MODULE                MODS    60   0.0540      3.2400
  02 ------------FMP 2010          ELECTRO CAP               CAP    20   0.0786      1.5712
  02 ------------COAX CONN         COAX CONNECTOR           CONN    28   0.1093      2.8936
ASSET STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:[illegible] MEM   [illegible] MEM
MTBF=    153.18 HOURS, FAILURE RATE=  6214.1029 PER MILLION HOURS(FPMH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. AIR CYC=  [illegible] , QUANTITY=1 , MINIMUM SPARES=
```

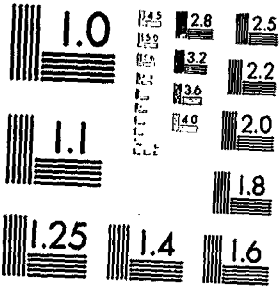
Figure E.3 FMP Reliability Data - Upper Bound

```
   LEVEL        PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL     70     80     90     95     99   99.5
01 --------------CON NETWK   CON NETWK                           210     0.0000  5314.1929     50     5[illegible]  55   58   63     65
  02 ------------FMP 1050        * ECL LOGIC IC 3224     DIG   73000     0.1883  374[illegible].[illegible]
  02 ------------FMP RC          * RC MODULE             MODR  [illegible]  0.0540   324.[illegible]
  02 ------------FMP 2010          ELECTRO CAP           CAP    2000     0.0786   157.1[illegible]
  02 ------------IC CONN           IC CONNECTOR          CONN    400     2.5954  1038.1[illegible]
  02 ------------COAX CONN         COAX CONNECTOR        CONN    280     0.1033    29.0[illegible]
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:EXT MEM FO  EXT MEM FANOUT
MTBF=  2466.11 HOURS, FAILURE RATE=   405.4972 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL     70     80     90     95     99   99.5
01 --------------EXT MEM FO  EXT MEM FANOUT                       12     0.0000   405.4972      4      5      6  [illegible]  8      9
  02 ------------FMP 1020        * ECL LOGIC IC 0616     DIG    2560     0.0288    73.7280
  02 ------------FMP RC          * RC MOD LF             MODR   1280     0.0540    69.1200
  02 ------------FMP 2010          ELECTRO CAP           CAP     640     0.0786    50.2773
  02 ------------IC CONN           IC CONNECTOR          CONN     64     2.5954   166.1056
  02 ------------COAX CONN         COAX CONNECTOR        CONN    448     0.1033    46.2[illegible]
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:EXT MEM     EXT MEM MODUL
MTBF= 128555.49 HOURS, FAILURE RATE=      7.7787 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES= 0

   LEVEL        PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL     70     80     90     95     99   99.5
01 --------------EXT MEM     EXT MEM MODUL                         1     0.0000     7.7787      0      0      0      1      1      1
  02 ------------FMP 4020        * 64K RAM               RAM      56     0.0003     0.0174
  02 ------------FMP 1010        * ECL LOGIC IC 0416     DIG      30     0.0235     0.7051
  02 ------------FMP RC          * RC MODULE             MODR     20     0.0540     1.0800
  02 ------------FMP 2010          ELECTRO CAP           CAP      10     0.0786     0.7856
  02 ------------IC CONN           IC CONNECTOR          CONN      2     2.5954     5.1908
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:DRM DRVR    DRM DRVR
MTBF=  11909.89 HOURS, FAILURE RATE=    83.9638 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES=

   LEVEL        PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL     70     80     90     95     99   99.5
01 --------------DRM DRVR    DRM DRVR                             10     0.0000    83.9638      1      1      2      2      3      4
  02 ------------FMP 1010        * ECL LOGIC IC 0416     DIG     800     0.0235    18.8000
  02 ------------FMP RC          * RC MODULE             MODR    100     0.0540     5.4000
  02 ------------FMP 2010          ELECTRO CAP           CAP     100     0.0786     7.8558
  02 ------------IC CONN           IC CONNECTOR          CONN     20     2.5954    51.9080
ASDET  STRUCTURED RELIABILITY AND SPARES PREDICTION
FOR:DRM MEM     DRM MEM
MTBF=   5842.61 HOURS, FAILURE RATE=   171.1564 PER MILLION HOURS(FMPH)
SPARING FOR  1.000 YEARS, DUTY CYCLE=  1.000 , FRAC. ADD ONS=  0.000, QUANTITY=1 , MINIMUM SPARES=

   LEVEL        PART NUMBER DESCRIPTION                        QUAN     FAILURE RATE(FMPH)     SPARES CONFIDENCE LEVEL(PERCENT)
                                                                          UNIT       TOTAL     70     80     90     95     99   99.5
01 --------------DRM MEM     DRM MEM                              48     0.0000   171.1564      2      2      3      4      5
  02 ------------FMP 4030        * 256K RAM              RAM    6144     0.0002     0.9216
  02 ------------FMP 1010        * ECL LOGIC IC 0416     DIG     672     0.0235    15.7920
  02 ------------FMP 1020        * ECL LOGIC IC 0616     DIG     816     0.0288    23.5008
  02 ------------FMP RC          * RC MODULE             MODR     48     0.0540     2.5920
  02 ------------FMP 2010          ELECTRO CAP           CAP      48     0.0786     3.7708
  02 ------------IC CONN           IC CONNECTOR          CONN     48     2.5954   124.5792
```

Figure E.3 FMP Reliability Data - Upper Bound (continued)

OF

26072



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

APPENDIX F
SYSTF'' THROUGHPUT AND UTILIZATION ANALYSIS

F.1 SUMMARY

The study of the feasibility of the NASF would be incomplete if only the high-performance computational engine, called the Flow Model Processor (FMP), were considered. The facility must have sufficient support equipment so that the FMP will not be idle due to bottlenecks elsewhere in the facility.

Chapter 2 of this report introduced the expected operational environment. Figure F.1 shows the organization of the facility at the level considered in this initial study of the facility. Reference can be made back to Figures 2.1 and 2.2 to understand the level of this model. In particular, analysis to this point does not include the structure of the data communications, processing and terminals local to the users. All of those capabilities are lumped under the term "Users".

Since draft copies of the system-level operational scenarios were not available until late in the study, some of the system-level analysis originally planned has not been completed. The analysis, described in more detail below, specifically considers the loading of the Flow Model Processor, the File System and the Support Processor. The data transfer requirements between each of these major system components and to the Users are also considered.

The analysis shows that the system proposed during the Preliminary Study [1, 2] would be inadequate to support the operational scenarios provided during this feasibility study. In particular, the support processor would have been a bottleneck as far as computational capability is concerned and the data transfer requirements to and from the support processor system were underestimated originally. Part of the excess loading of the support processor system was alleviated near the start of this study when the decision was made to consider the feasibility of a system where file management was a function supported by the file system itself rather than by the Support Processor. This analysis has shown that support of the major formatting requirements for both hardcopy printers and, most especially, for Computer-Output to Microfilm (COM) should be removed from the Support Processor to either a peripheral-support processor or perhaps to the FMP itself.

F.2 MODEL AND ASSUMPTIONS USED FOR ANALYSIS

Figure F.1 shows the general model used for this analysis. The analysis performed was an operational-type analysis based on NASF operational scenarios included in the original NASF Utilization document [3] as updated during subsequent discussions.

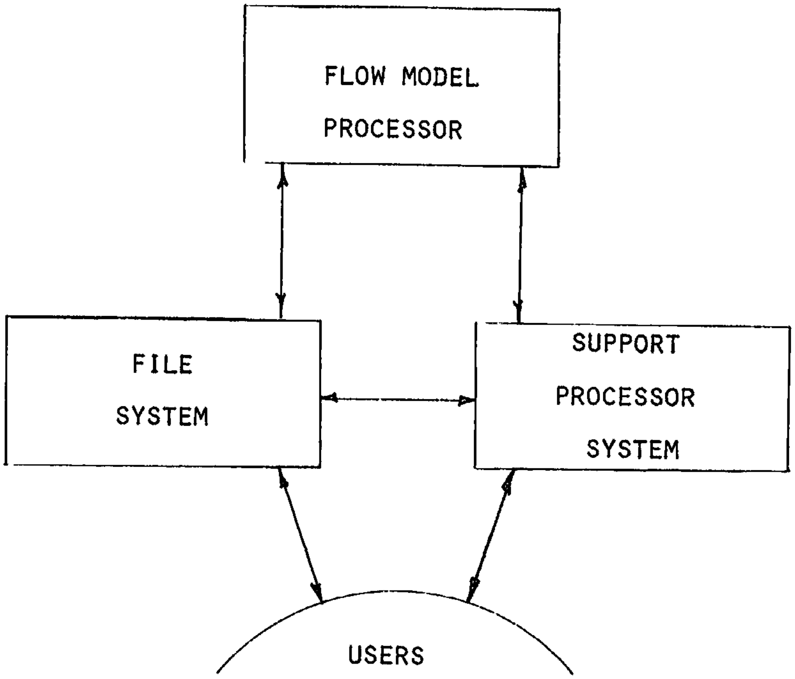


Figure F.1 NASF System Throughput and Analysis Model

The data presented in the scenario was in terms of job classifications. The following cases were used to represent the various types of use encountered during a "typical" NASF day.

1.A Method and Code Development using scaled down problems.

1.B Grid Modification.

2.A Larger code development as well as grid and result array generation.

2.B Grid Generation.

3. Simpler simulations on a large grid (such as inviscid flows with boundard layer correction).

4. Typical viscous, steady flow simulations used for design, resulting in a single solution.

5. Viscous, steady flow simulations requiring several solutions, such as design optimizations.

6. Unsteady viscous flow simulations for design applications.

7. Large fluid physics research simulations.

These cases correspond to the column headings in Table F.1.

Regardless of case, each user has a sequence of tasks to be performed in order to complete his job. These tasks were generally identified in four major areas:

A. Simulation Program Input
B. Simulation Input Data Preparation
C. Simulation (execution)
D. Output of Simulation Results

The actual detailed tasks defined in the utilization document [3] were:

(A. Simulation Program Input)

1. Source Module Generation is the task of inputting "new" simulation source programs into the system.
2. Source Module Editing is the task of editing source modules as required by input or compile errors.
3. Source Module Compilation is the task of compiling source modules of simulation programs into object programs.
4. Linking is the task of collecting all the object modules which are required for a simulation and cleaning up incomplete address binding prior to loading into the FMP for execution.

(B. Simulation Input Data Preparation)

5. Configuration Generation is the task where surface coordinate tables are input and surface patches are computed. The model assumed that, in some cases, the FMP could compute the surface patch coefficients but that the processor local to the graphics stations could also perform this computation.
6. Surface Grid Generation (Not separately modelled but considered part of Task 7).
7. Flow Field Grid Generation is the task which computes the coordinates of the grid to be used during flow-field computations on the FMP. The model assumed that this task would be executed on the FMP when operator verification of the resulting grid was not necessary. Otherwise, it would be executed on the Support Processor in order to have prompt display of results to the operator.
8. Input Gathering is the task which is used to specify the parameters of a particular simulation run and to begin staging data to the FMP.
9. FMP Execution is the task which runs a job on the FMP.
10. Preselected Data Display is the task which outputs data which had been organized during FMP execution to line printers, to graphics terminals and to microfilm printers.
11. Interactive Post-Execute Display is the task which supports the selective extraction of data from the FMP output files. The data extracted would be requested by and displayed to the user at a graphics console.
12. Debugging Display is the task of formatting and displaying (in some appropriate manner) that information saved by the FMP when a run aborts.
13. Restart Dumps is a subtask of Task 9 and involves taking a snapshot of the status of a simulation run to be used as a restart or initialization point on a later run.

Table F.1 summarizes the important data for evaluated the model studied.

Table F.1 NASF Operational Scenerio Data

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
1	SOURCE MODULE GENERATION									
	FREQUENCY (MODULES/DAY)	10		10		1.5	1.0	2/3	.3	.45
	EXECUTABLE STMTS / MODULE AVG.	100	200	200		200	200	200	200	200
	MODULES/CODE AVG.	10	20	20		40	40	40	40	15
2	SOURCE MODULE EDITING									
	FREQUENCY (MODULES/DAY)	450		100		3.75	5	.5	.375	.75
	SESSION DURATION (MINUTES) AVG.	30		30		15	15	15	15	15
	SESSIONS/DAY AVG.	300		120		3.75	5	.5	.375	.75
3	SOURCE MODULE COMPILATION									
	FREQUENCY (MODULES/DAY)	[COUNT EQUALS SUM OF MODULES/DAY FROM TASKS 1 AND 2]								
	STMTS/MODULE AVG.	100		200		200	200	200	200	200
	FRACTION MODULES WITH COMPILE ERRORS/DAY	.25		.25		.20	.20	.20	.20	.20

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
4	LINK									
	FREQUENCY MODULES/DAY			SAME AS NUMBER OF COMPILES LESS COMPILE ERRORS						
5	CONFIGURATION GENERATION									
5A	BASELINE									
	FREQUENCY MODULES/DAY			.5		.375	.5		.075	
	TASK DURATION HOURS			24		48	48		48	
	FILE SIZE WORDS			50K		100K	100K		100K	
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM			1.0		1.0	1.0		1.0	
	TOTAL NO. FILES			25		5	12.5		2	
	FILE LIFETIME WEEKS			10		2.5	5		5	
	FRACTION THAT ARE ARCHIVED			1.0		1.0	1.0		1.0	

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
5B	BASELINE MODIFICATION									
	FREQUENCY MODULES/DAY					4	5		.8	
	TASK DURATION HOURS					3	3		3	
	FILE SIZE WORDS					100K	100K		100K	
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM					1.0	1.0		1.0	
	TOTAL NO. FILES					50	125		20	
	FILE LIFETIME WEEKS					1.25	2.5		2.5	
	FRACTION THAT ARE ARCHIVED					.1	.1		.1	

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
5C	FMP GENERATION INPUT PREPARATION									
	NEW FILES/DAY	10	.5	2.5		.75	1.0	.33	.15	.033
	BODY COORDINATE FILE SIZE WORDS	20K	20K	50K		100K	100K	100K	100K	100K
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM	1.0	1.0	1.0		1.0	1.0	1.0	1.0	1.0
	TOTAL NO. FILES	← 484 →		100		5	12.5	5	2	4
	FILE LIFETIME WEEKS	← 10 →		10		2.5	5	3	5	26
	FRACTION THAT ARE ARCHIVED	← 0 →		0		1.0	1.0	1.0	1.0	1.0
5D	FMP GENERATION INPUT MODIFICATION									
	FILES MODIFIED/DAY					4	5		.3	
	FILE SIZE (WORDS)					100K	100K		100K	
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM					1.0	1.0		1.0	
	TOTAL NO. FILES					50	125		20	
	FILE LIFETIME WEEKS					1.25	2.5		2.5	
	FRACTION THAT ARE ARCHIVED					.1	.1		.1	

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
6	SURFACE GRID GENERATION			INCLUDED AS PART OF TASK 7						
7	FLOW FIELD GRID GENERATION	[NOTE: HALF OF RUNS ARE ASSUMED TO HAVE GRID GENERATIONS PART OF TASK 9] [THE OTHER HALF OF THE RUNS EXECUTE ON THE SPS]								
7A	BASELINE									
	FREQUENCY GENERATIONS/DAY			.5		.375	.5		.075	
	TASK DURATION HOURS			8		8	8		8	
	FILE SIZE WORDS			600K		3000K	3000K		3000K	
	FMP USAGE	[NOTE: SEE TASK 9, CASE INDICATED AS BELOW FOR FMP CPU REQUIREMENTS]								
	TASKS/DAY			.25		.1875	.25		.0375	
	RUNS/TASK			10		10	10		10	
	RUNS/DAY			2.5		1.875	2.5		.375	
	CASE OF RUN			1		2	2		2	
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM			1.0		1.0	1.0		1.0	
	TOTAL NO. FILES			12.5		2.5	6.25		1	
	FILE LIFETIME WEEKS			10		2.5	5		5	
	FRACTION THAT ARE ARCHIVED			1.0		1.0	1.0		1.0	

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
7B	BASELINE MODIFICATION	[NOTE: HALF OF THE RUNS ARE ON THE SPS, HALF ON THE FMP]								
	FREQUENCY MODS/DAY					3.75	5		.75	
	TASK DURATION HOURS					3	3		3	
	FILE SIZE WORDS					3M	3M		3M	
	FMP USAGE	[NOTE: SEE TASK 9, CASE INDICATED AS BELOW FOR FMP CFU REQUIREMENTS]								
	TASKS/DAY					1.875	2.5		.375	
	RUNS/TASK					3	3		3	
	RUNS/DAY					5.625	7.5		1.025	
	CASE OF RUN					1	1		1	
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM					1.0	1.0		1.0	
	TOTAL NO. FILES					50	125		20	
	FILE LIFETIME WEEKS					1.25	2.5		2.5	
	FRACTION THAT ARE ARCHIVED					.1	.1		.1	

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
3	INPUT GATHERING									
3A	SOLUTION PARAMETERS									
	ACCESSED AND CHANGED /DAY	283	17	115	5	30	40	4	3	3
	FILE SIZE WORDS	1K	1K	1K	1K	1K	1K	1K	1K	1K
	LONGTERM FILES;									
	FRACTION THAT BECOME LONGTERM	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	TOTAL NO. FILES	177	11	96	.4	38	100	20	15	15
	FILE LIFETIME HOURS	7.5	7.5	10	10	15	30	60	60	60
	FRACTION THAT ARE ARCHIVED	0	0	0	0	0	0	0	0	0
3B	CONFIGURATION GEOMETRY									
	ACCESSED/DAY	283	17	92	5	30	40	4	3	3
	FILE SIZE WORDS	20K	100K	50K	100K	100K	100K	100K	100K	50K
3C	FLOW FIELD GRID									
	ACCESSED/DAY		17	23	5	15	20		1.5	
	FILE SIZE WORDS		3M	600K	3M	3M	3M		3M	
3D	RESTART DATA									
	ACCESSED/DAY		17	5		10	20	4	3	3
	FILE SIZE WORDS		3M	3M		7M	10M	10M	10M	50M

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
10B	TERMINAL GRAPHICS									
	JOBS/DAY	200	17	80	5	30	40	4	3	3
	FRAMES/JOB	5	5	10	10	10	10	25	200	200
	POINTS/FRAME (3 COORDS/POINT)	2K	2K	3K	3K	3K	3K	3K	3K	3K
	FILE SIZE WORDS	30K	30K	90K	90K	90K	90K	225K	1800K	1800K
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM			.1		.5	.5	.5	.5	.25
	TOTAL NO. FILES			400		200	200	30	40	100
	FILE LIFETIME WEEKS			10		50	50	50	50	25
	FRACTION THAT ARE ARCHIVED			.05		.25	.25	.1	.25	.25
10C	COM									
	JOBS/DAY	30	1.7	12	.5	8	10	2	2	2
	FRAMES/JOB	10	10	10	10	10	10	25	2500	2500
	POINTS/FRAME (3 COORDS/POINT)	5K	5K	5K	5K	5K	5K	5K	10K	10K
	FILE SIZE WORDS	150K	150K	150K	150K	150K	150K	150K	75,000K	75,000K

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
9	FMP EXECUTION									
	RUNS/DAY	283	17	115	5	30	40	4	3	3
	CPU TIME MINUTES	.167		1.0		1.0	10.	60.	60.	60.
	MEMORY REQUIRED WORDS	4M		8M		20M	30M	30M	30M	150M
	FRACTION RUNS ABORTED/DAY	.33		.33		.1	.1	.1	.1	.1
10	PRESELECTED DATA DISPLAY									
10A	PRINT									
	JOBS/DAY	283	17	115	5	30	40	4	3	3
	FILE SIZE WORDS	5K	10K	10K	10K	10K	10K	10K	10K	10K

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
11	INTERACTIVE POST EXECUTE DISPLAY									
	JOBS/DAY	10		9		3	10		.3	1.5
	FILE SIZE SCANNED WORDS	1M		2M		7M	10M		10M	50M
	FRAMES DISPLAYED	10		50		50	100		100	100
	POINTS/FRAME (3 COORDS/POINT)	2K		2K		2K	3K		3K	3K
	SPS CPU TIME/FRAME SECONDS	1		2		5	5		5	5
	TASK DURATION HOURS	.5		2		2	4		4	4
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM	.1		.1		.5	.5		.5	.5
	TOTAL NO. FILES	44		25		30	60		4	50
	FILE LIFETIME WEEKS	10		10		1.25	2.5		2.5	13
	FRACTION THAT ARE ARCHIVED	0		0		.5	.5		.5	1.0

Table F.1 NASF Operational Scenerio Data (continued)

TASK NO.	DESCRIPTION	CASE 1		CASE 2		CASE 3	CASE 4	CASE 5	CASE 6	CASE 7
		A	B	A	B					
.2	DEBUGGING DISPLAY	[NOTE: THIS IS A SELECTIVE DUMP OF STORAGE REGISTERS, ETC]								
	JOBS/DAY	1	SAME AS TOTAL JOBS PER DAY WHICH ABORT. SEE ABORT RATE BELOW							1
	ABORT RATE	.33	.1	.33	.1	.1	.1	.1	.1	.1
	WORDS PRINTED	10K	10K	10K	10K	10K	10K	10K	10K	10K
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	TOTAL NO. FILES	200		100		100	100	100	100	100
	FILE LIFETIME DAYS	2		2		2	2	2	2	2
	FRACTION THAT ARE ARCHIVED	0		0		0	0	0	0	0
.3	RESTART DUMPS									
	JOBS/DAY		17		5	10	20	4	3	3
	FILE SIZE WORDS		3M		3M	7M	10M	10M	10M	50M
	LONGTERM FILES:									
	FRACTION THAT BECOME LONGTERM		1.0		1.0	1.0	1.0	1.0	1.0	1.0
	TOTAL NO. FILES		34		10	6	25	10	6	6
	FILE LIFETIME WEEKS		1		1	.5	.5	.5	.5	.5
	FRACTION THAT ARE ARCHIVED		0			0	0	0	0	0

In addition to the information provided by NASA, it was necessary to make some assumptions during the course of the analysis. These assumptions were based in part on experience and in part on judgement. Table F.2 summarizes these assumptions.

TABLE F.2
Significant Assumptions

0.2	Fraction of Users who use the Support Processor to do data entry & editing
40	Average length (chars) of source statements
50	Average length (chars) of control messages
0.2	Fraction of a module fixed or modified on each edit
4000	Average additional compiler output (characters) over and above the source statements per module
8	Number of words of object program per line of source
0.25	Fraction of modules with bugs which are waiting change and which will be batched as far as BINDING (linking)
0.1	Fraction of edited program codes which must be completely bound or linked (others will be replacement bound)
0.5	Fraction of solution parameters (of an earlier run) modified to setup the next run
8	Number of characters out of FORMATTER for each word in (used to format printouts & COM)
6	Number of 8 bit characters per word - (6x8 = 48 bits)
1.25×10^{11}	Max size of archive (characters)
.20	Fraction of file access from active data
.70	Fraction of file access from long-term data
.10	Fraction of file access from archive data

F.3 ANALYSIS

The analysis first defined the sequence of events required to implement each task. The relationship of system resources during each task was then charted based on an approach suggested by Prof. Anatol Holt (Boston University). These charts or diagrams are intended to separate spatial relationships and temporal relationships. Figure F.2 shows the resource-relationship charts for Tasks 1 through 5 and 7 respectively. The interpretation of these charts is straight-forward. The various NASF resources, sometimes including equipment local to the users, are shown left to right across the chart. The sequence of events required to complete a task is represented from the top to the bottom of the chart. For example, the first chart of Figure F.2 is for Task 1 - Source Module Generation. The sequence of events shown is:

Create File
Enter Records
Save File
Create File
Save Working File

The first Create File event involves the user, his terminal, data comm controls, the operating system on the support processor and working files (in the support processor). Thus, the system resources which interact to implement each event of the task are delineated. The second event, Enter Records, involves a bulk move of data. This type of interaction is shown with the curved corners. This notation allows the natural flow of data, here from the user to the working file, to be shown clearly. Each resource involved in an event commits something of that resource to the event, whether it be space (storage) or time. These committments were the point of the analysis performed. Note that the charts shown contain more information than was utilized in the analysis to date. In particular, the processing capabilities and communications local to a user were not studied yet.

After the resource-relationship charts were prepared, they were used as templates to prepare a straight-forward program which would collect the operational scenario data in the manner described by the charts in order to generate the results. Thus the charts could be used to identify how many control messages moved and what data transferred between elements of the model. The NASA provided data was used to identify the average frequency of a task,the amount of data involved, and the processing time required.

TASK 1 - SOURCE MODULE GENERATION

Input of source program via terminal and editor (on SPS)

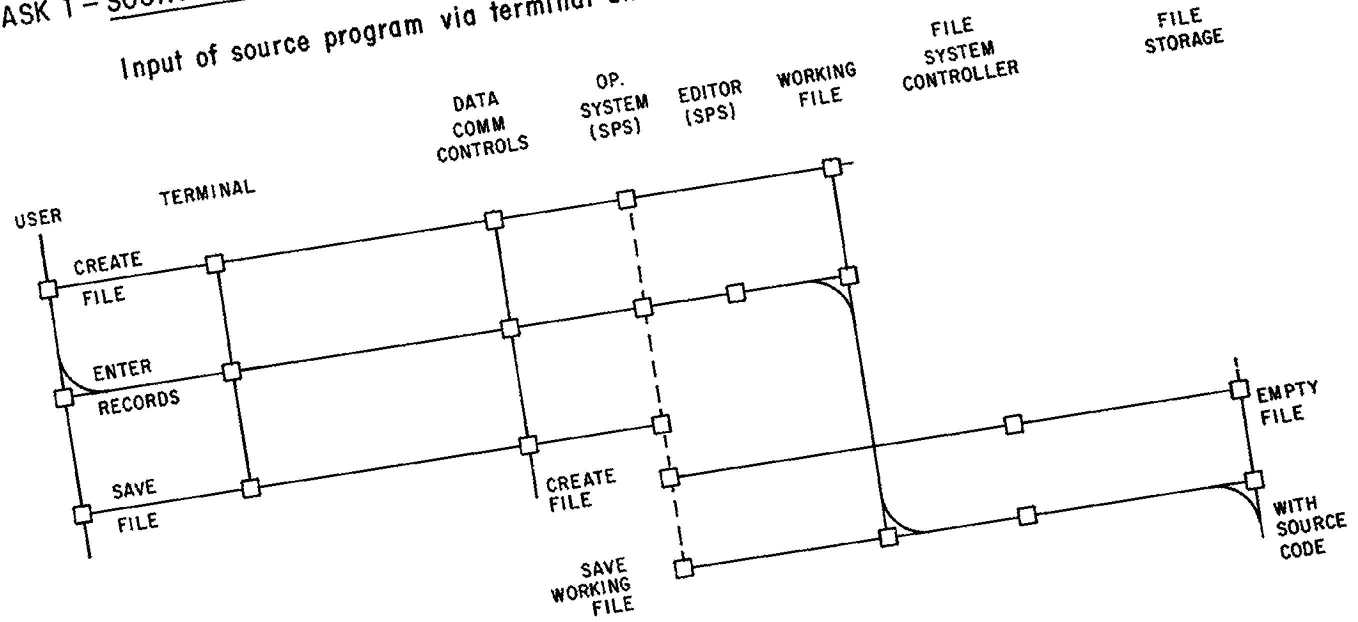


Figure F.2 NASF Resource Relationships

TASK 2 - SOURCE MODULE EDITING

Edit of source program via terminal and editor (on SPS)

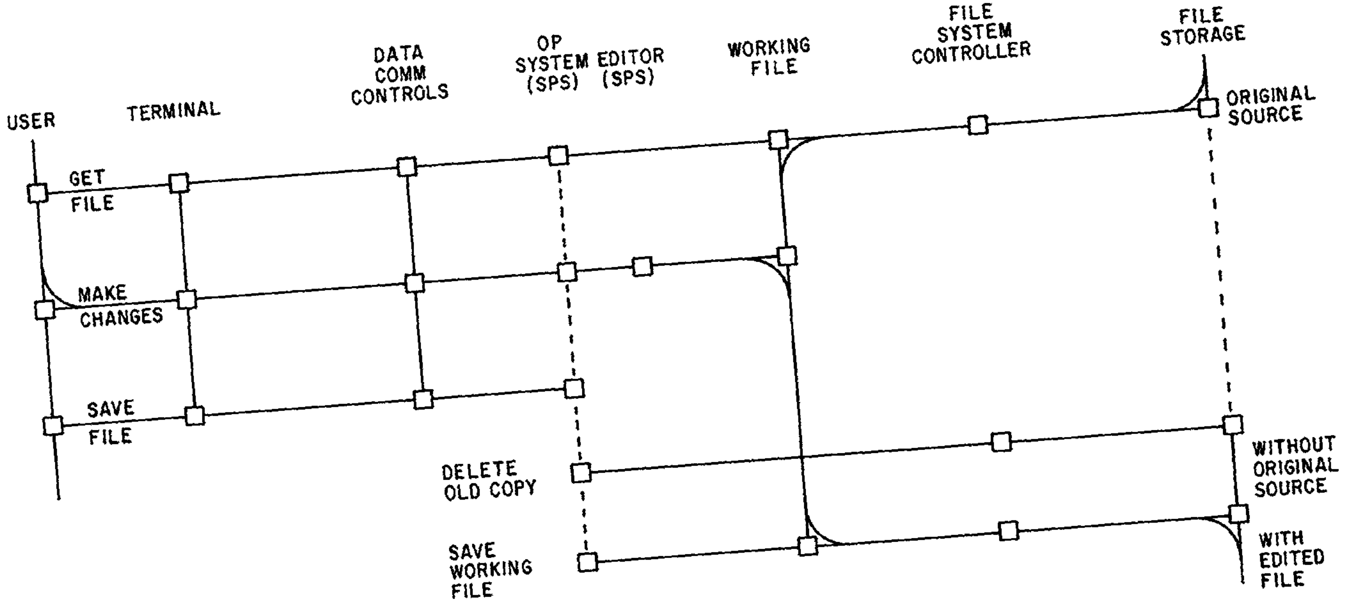


Figure F.2 NASF Resource Relationships (continued)

TASK 3 - SOURCE MODULE COMPILATION

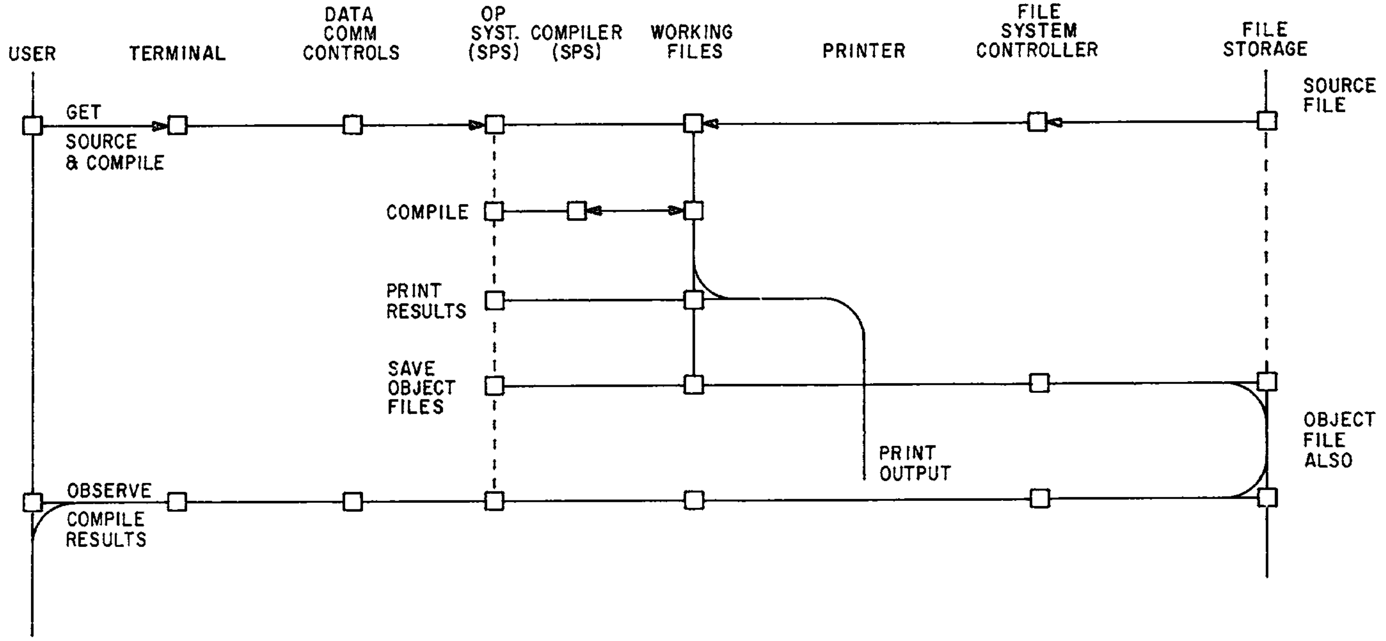


Figure F.2 NASF Resource Relationships (continued)

TASK 4 - LINK CODE

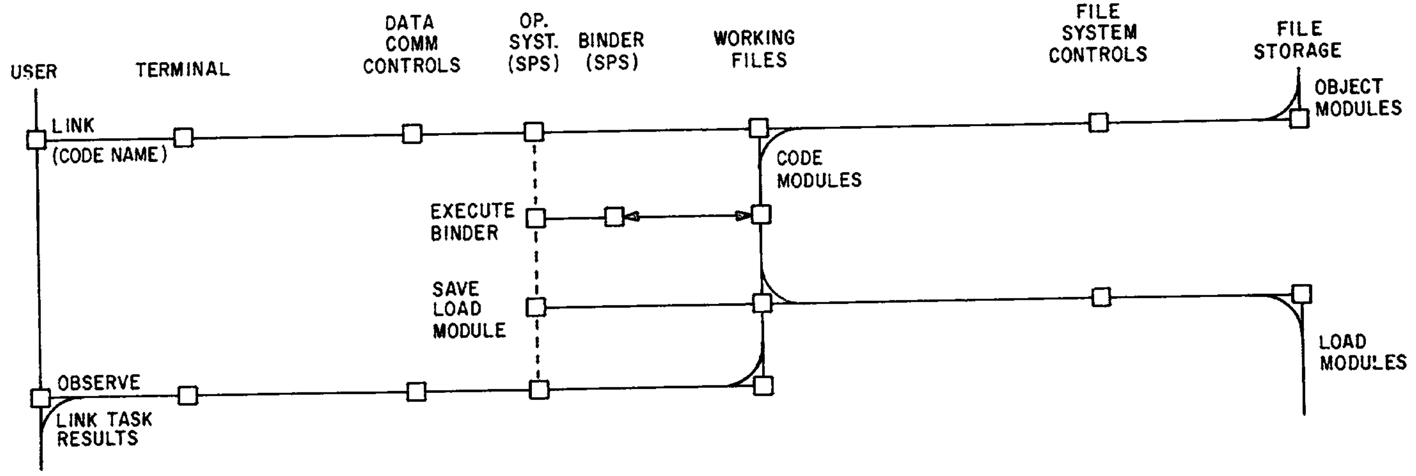


Figure F.2 NASF Resource Relationships (continued)

TASK 5 - CONFIGURATION GENERATION A - Baseline

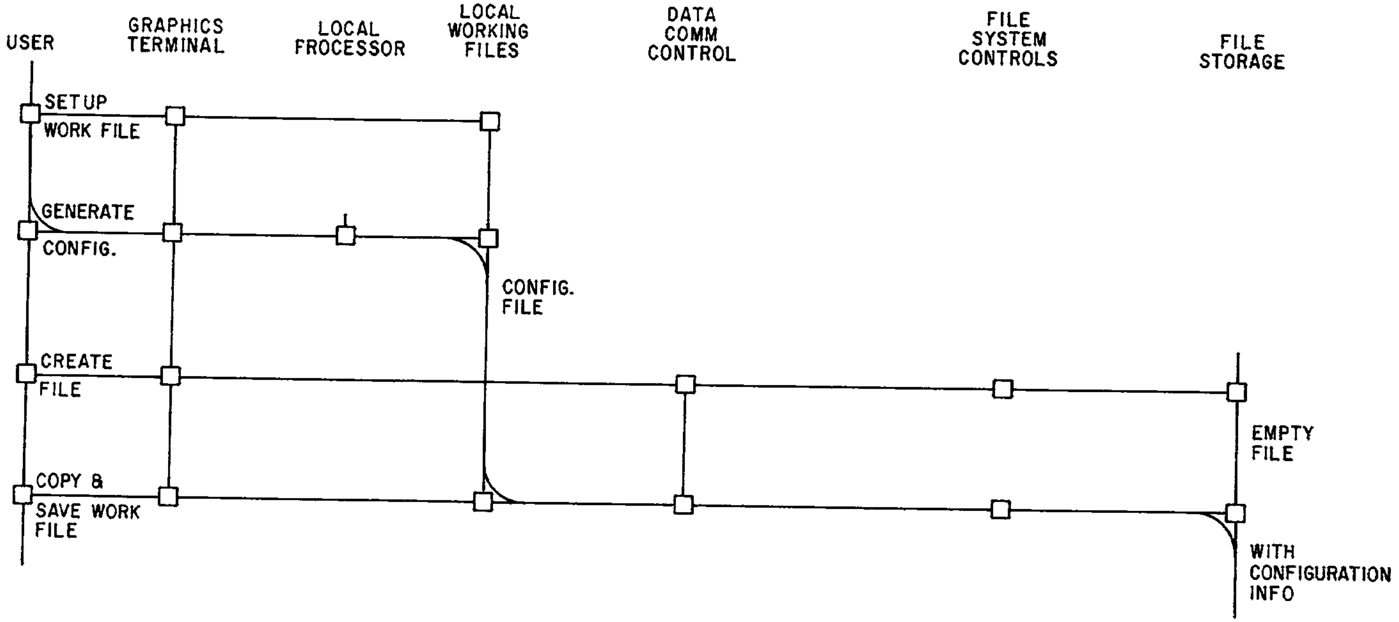


Figure F.2 NASF Resource Relationships (continued)

TASK 5 - CONFIGURATION GENERATION   B - Baseline Modification

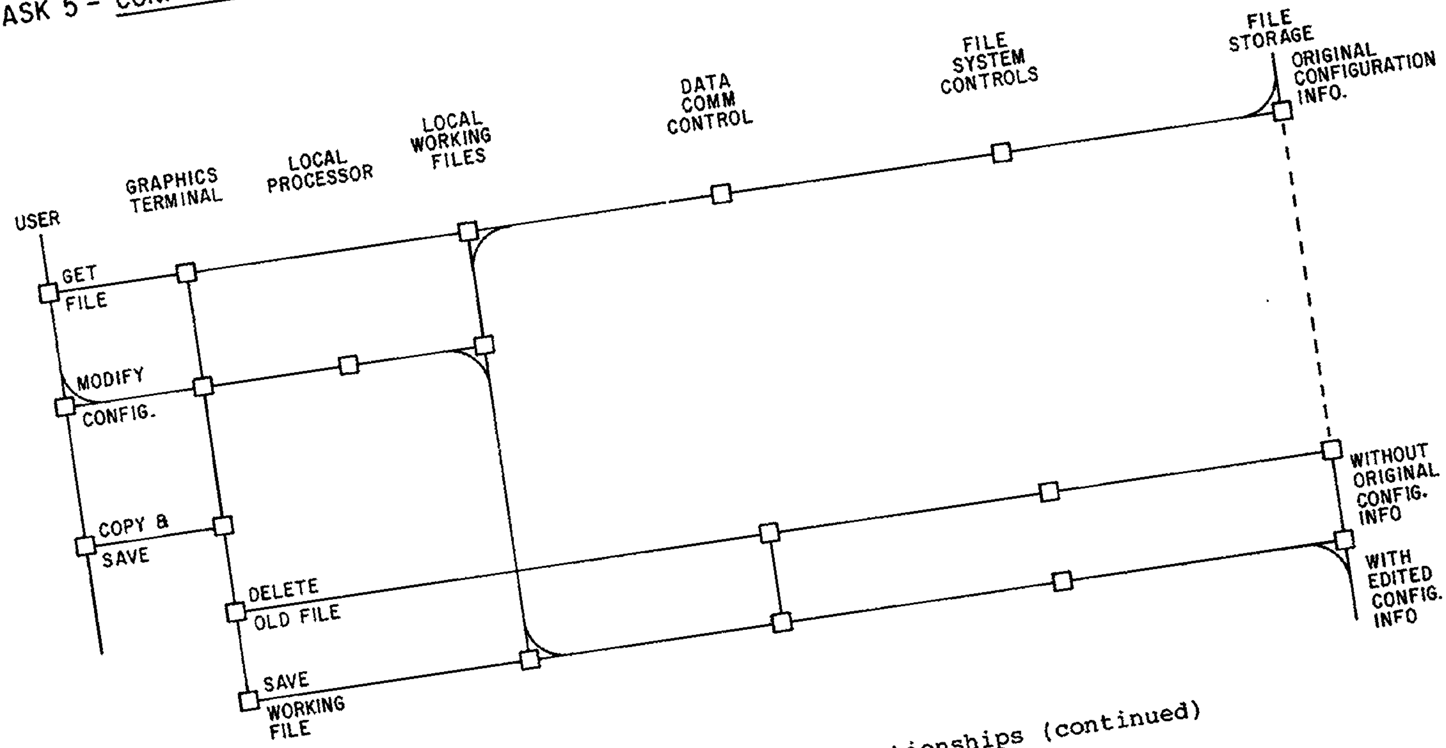


Figure F.2 NASF Resource Relationships (continued)

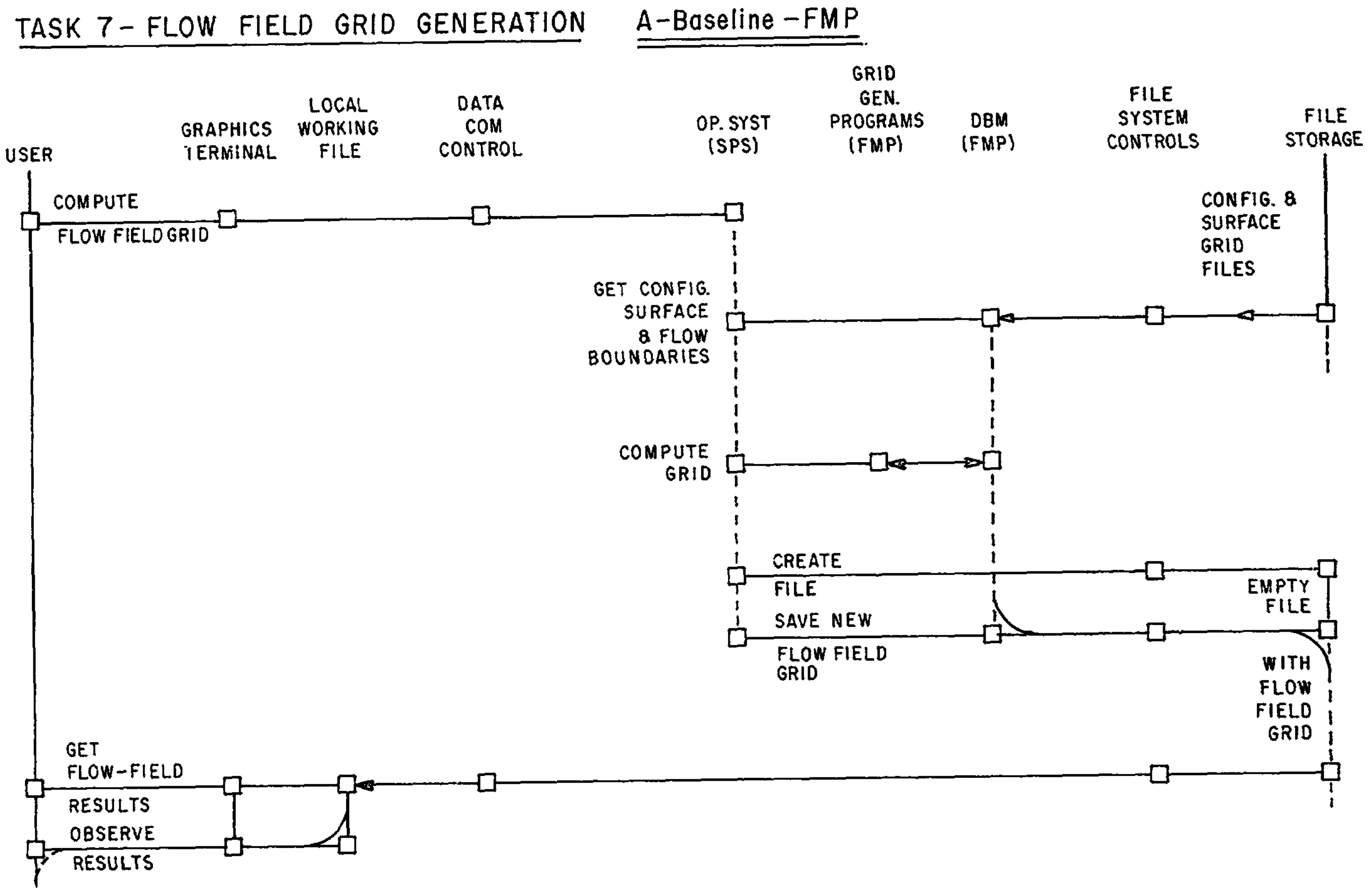


Figure F.2 NASF Resource Relationships (continued)

TASK 7 — FLOW FIELD GRID GENERATION B-Baseline Modification - FMP

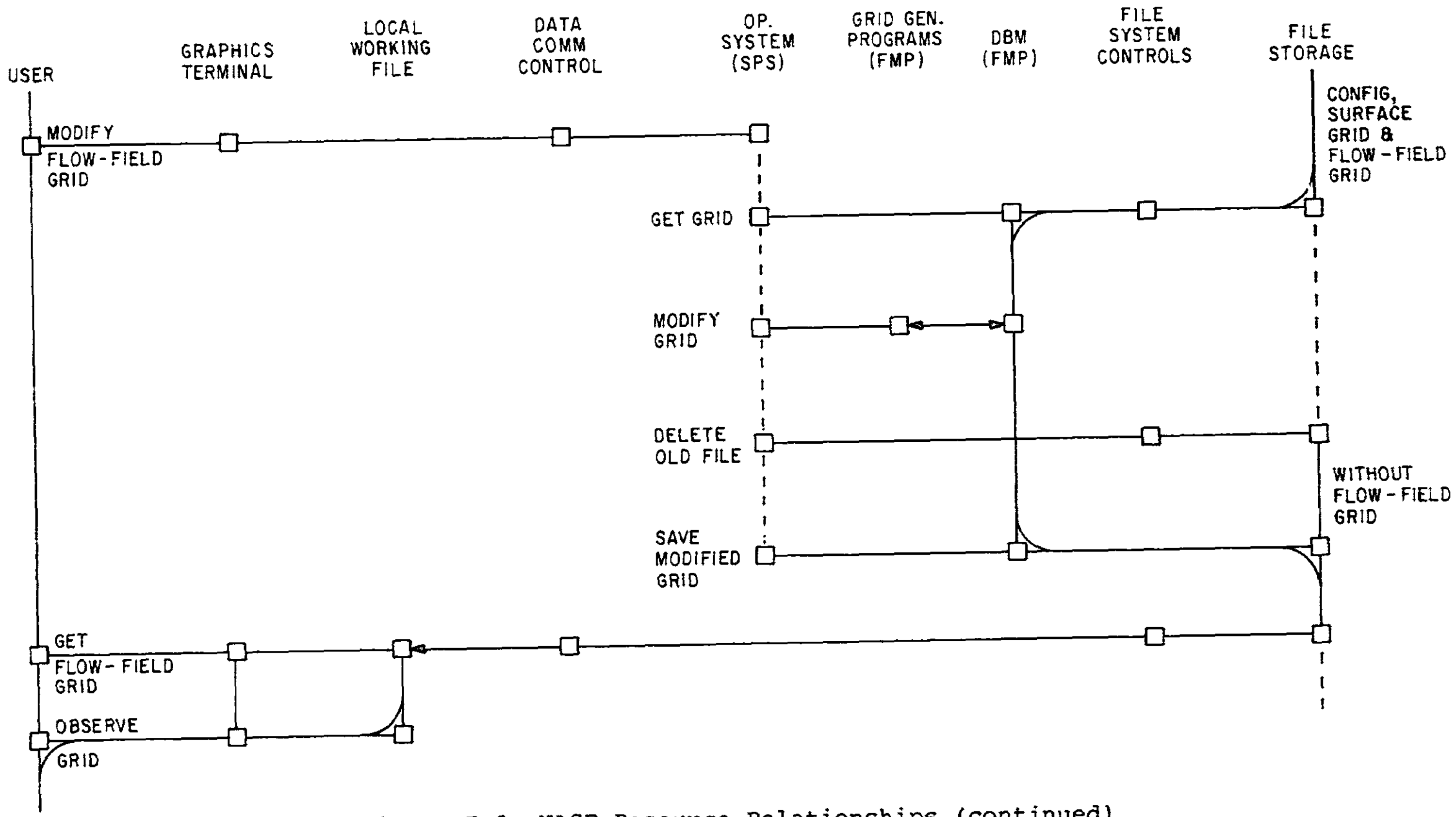


Figure F.2 NASF Resource Relationships (continued)

TASK 7 - FLOW FIELD GRID GENERATION   A- Baseline — SPS

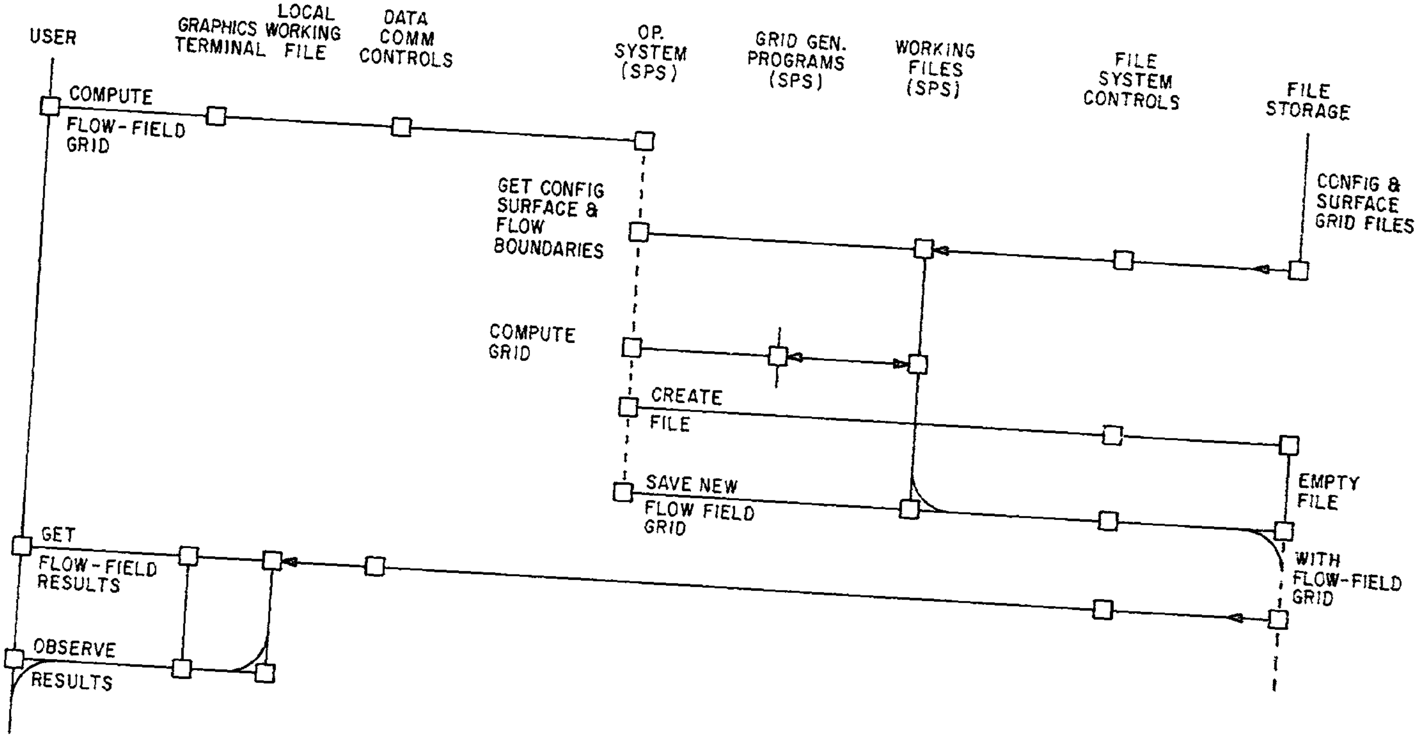


Figure F.2 NASF Resource Relationships (continued)

TASK 7 – FLOW FIELD GRID GENERATION B - Baseline Modification – SPS

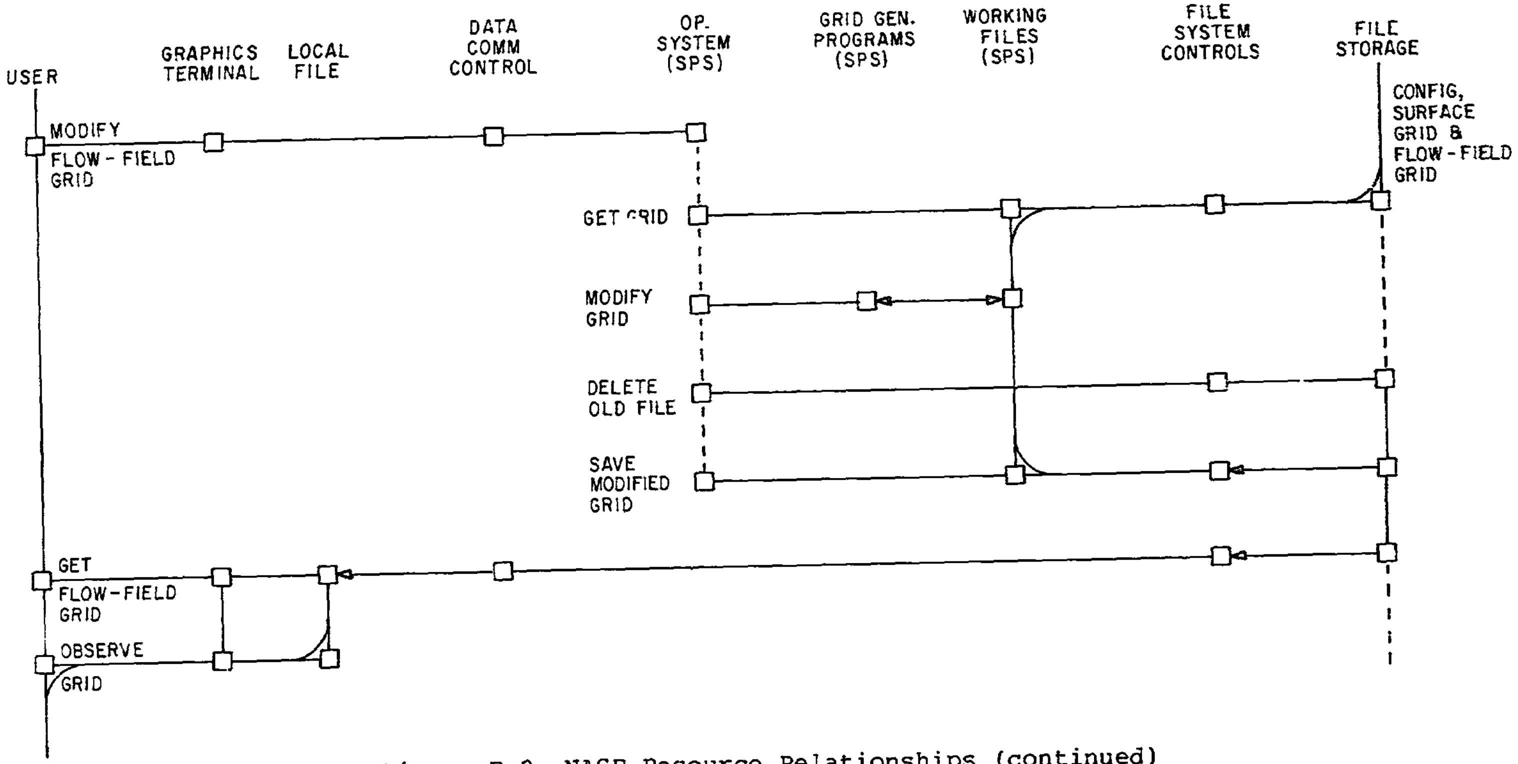


Figure F.2 NASF Resource Relationships (continued)

TASK 8 — INPUT GATHERING

Note – It was assumed here that modification can be done via "editing" only using the SPS

A – Solution Parameters

B – Configuration Geometry

C – Flow Field Grid

D – Restart Data

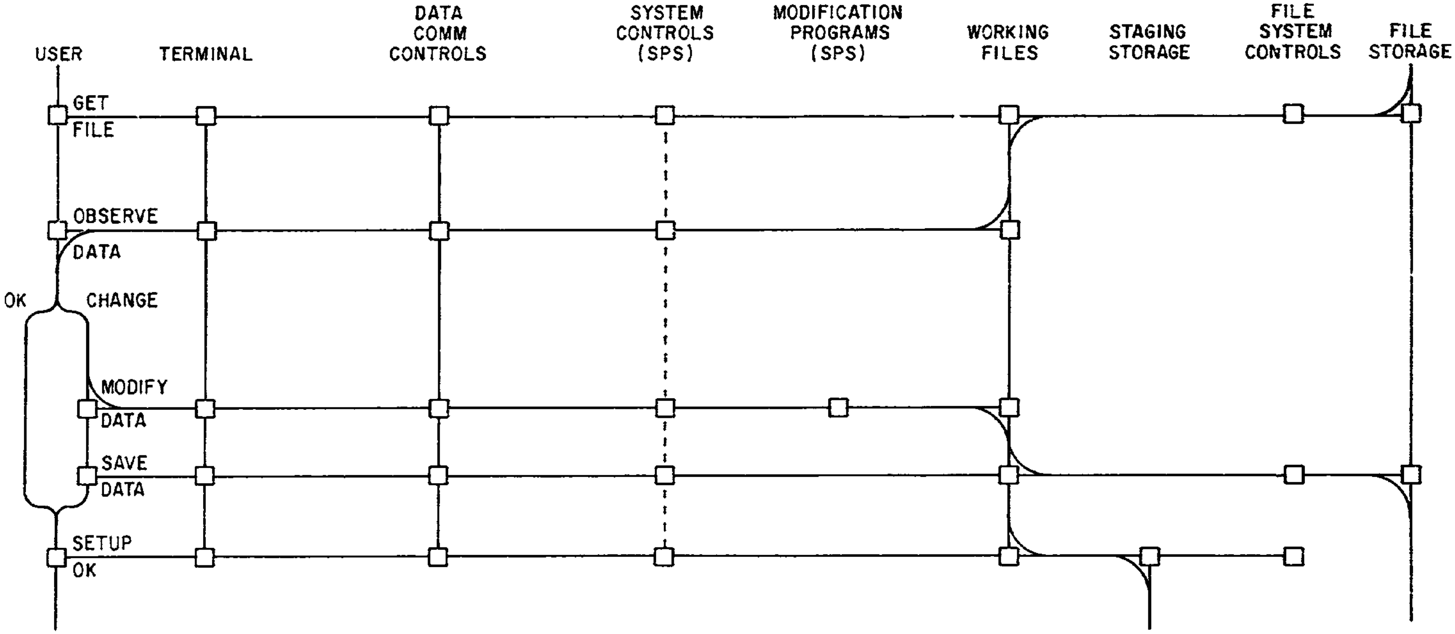


Figure F.2 NASF Resource Relationships (continued)

TASK 9 — FMP EXECUTION

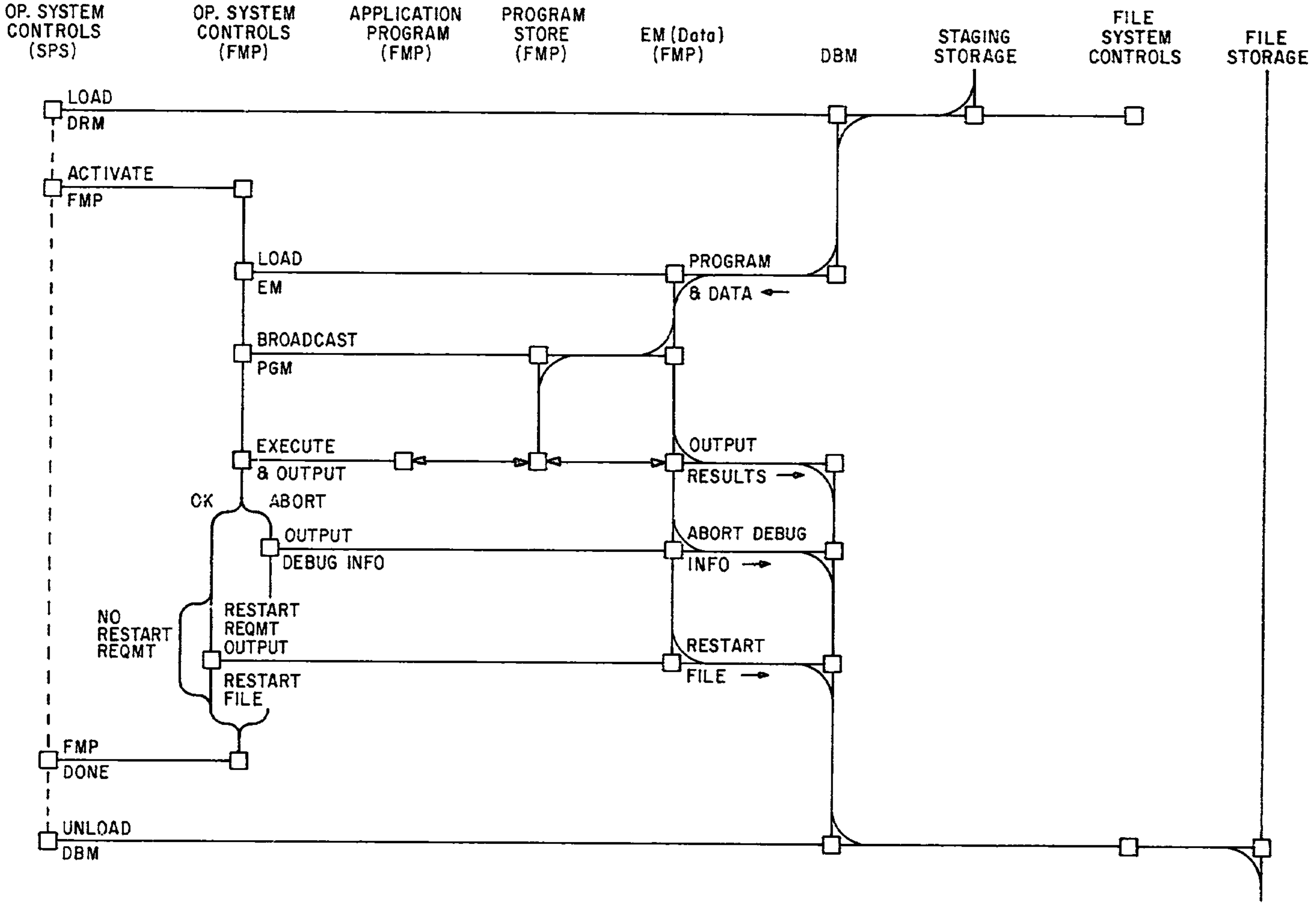


Figure F.2 NASF Resource Relationships (continued)

TASK 10 — PRESELECTED DATA DISPLAY A - Print

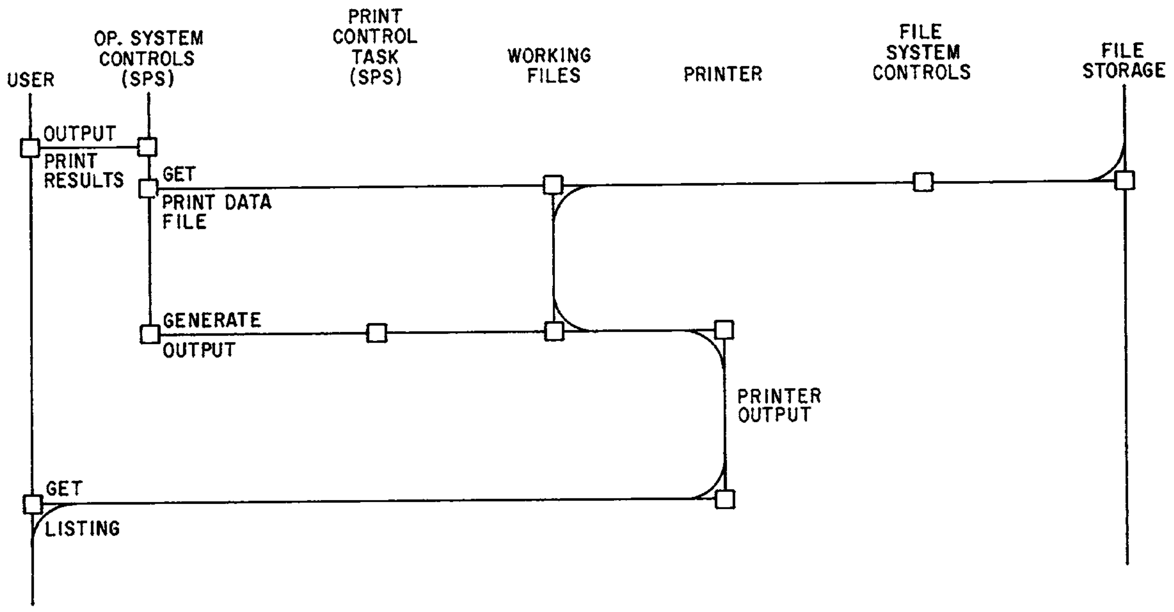
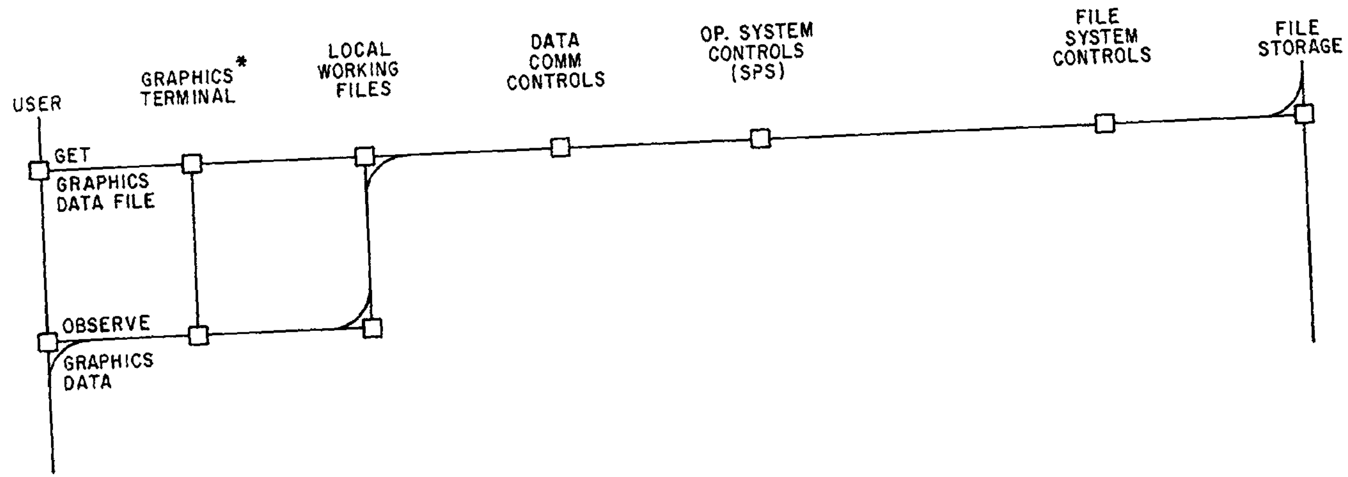


Figure F.2 NASF Resource Relationships (continued)

TASK 10 - PRESELECTED DATA DISPLAY   B - Terminal Graphics



* Graphics Terminal may be E&S System, Limited - Capability Terminal, Graphic Hard-Copy Device ....

Figure F.2 NASF Resource Relationships (continued)

TASK 10 — PRESELECTED DATA DISPLAY C-COM

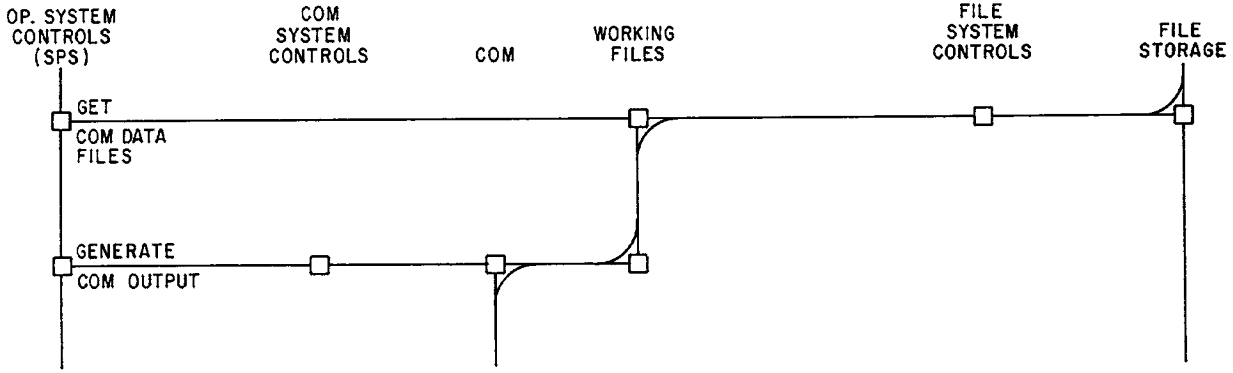


Figure F.2 NASF Resource Relationships (continued)

TASK 11 - INTERACTIVE POST-EXECUTE DISPLAY

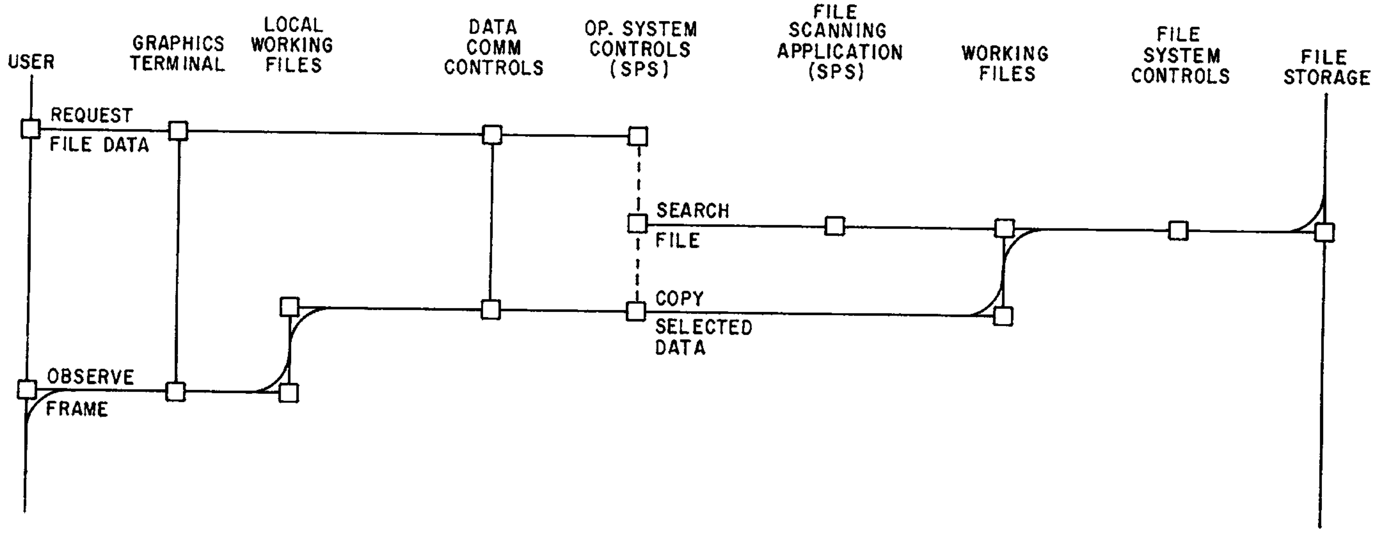


Figure F.2 NASF Resource Relationships (continued)

TASK 12 - DEBUGGING DISPLAY

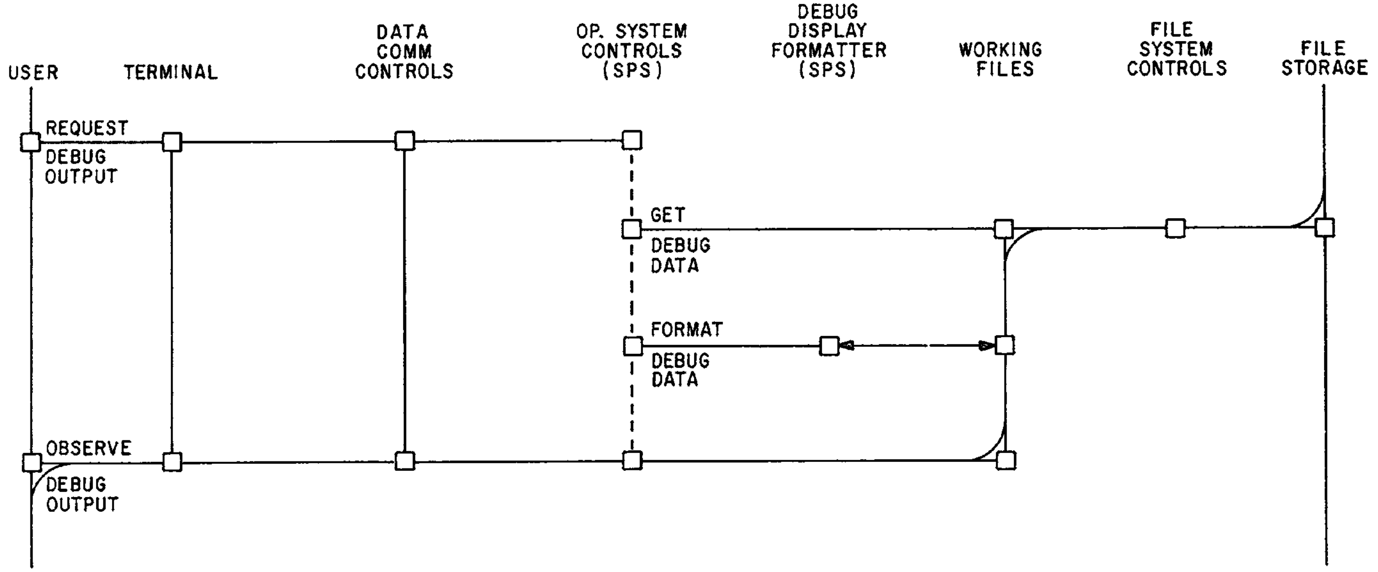


Figure F.2 NASF Resource Relationships (continued)

The program developed to support the analysis allowed specification of the parameters of the support processor. Since the number of CPU's needed as part of the support processor system was of interest, the capabilities of a particular single processor were defined. The analysis then showed the support processor loading in terms of that one processor. From that analysis, the number of processors required could be determined. For example, if the average load was determined to be 1.6 processor-hours for each hour, then at least two processors are required.

Table F.3 summarizes the characteristics of three support processors considered during the analysis. The processors are identified "A", "B", and "C". This form of identification was chosen since none of the data has been completely verified on any processor. In general, the processors are characterized as

A	B7700
B	B7800
C	Future Processor

The data concerning editing, compiling, and formatting on processor A is based on benchmarks run on a mix of FORTRAN programs. The other values are estimates based on best knowledge and judgment.

In addition, where it seemed appropriate during the evaluation, some modifications were made to the NASA supplied operational scenarios. In particular, the anlaysis was performed for some cases without the COM output (Task 10C) in an attempt to identify the impact of that large amount of output.

The analyzer currently generates results in terms of daily average loads. Hand reduction of the results from the analyzer is used to generate hourly average loads.

F.4 RESULTS

Before considering the results, a WARNING must be stated. The analysis to date only considers average data rate and processor time requirements. This would only be true under conditions of optimum system balance and concurrency. Factors which are needed to predict the peak rates which an eventual design must consider have not been included.

The analysis considered three major factors of the NASF system model. First, the data transfer requirements for transfers between each of the components of the model in Figure F-1 were determined. Then the amount of file level activity was determined in order to estimate the processor required to support the file system. Finally, an amount of Support Processor and FMP processing time was determined using the assumptions previously stated.

TABLE F.3

SUPPORT PROCESSOR CHARACTERIZATION

	A	B	C
Edit Time (Sec/Stmt)	.01	.0067	.001
Compile Time (Sec/Stmt)	.007	.0047	.001
Compile Overhead (Sec/Module)	0.1	.067	.013
Linking Time (Sec/Object Word)	.0007	.00047	.0001
Linking Overhead (Sec/Code)	.01	.0067	.001
Grid Generation Rate (Sec/Grid Element)	.001	.00067	.0001
Operating System overhead (Sec/Task)	0.0	.067	.0133
Formatting Rate (Sec/Word Formatted)	0.0006	.0004	.0001
Output Selection Rate (Sec/Point Selected)	0.01	.0067	.0013

F.4.1 Processor Loading

Table F.3 summarized the basic characteristics of the processors considered in this analysis. The data in this table concerning output formatting is based on the normal interpretive execution of a formatter package which is driven by the FORMAT statements. One of the major reasons such a system is interpretive is the possibility of variable format statements. However, most of the applications observed (both the aero and weather codes among others) have rather straight-forward, fixed formatting. The improvement in formatting time that could occur if the formatter was compiled and executed per the statements given rather than interpreted was assumed to be a factor of 5. Each of the processor of Table F.3 were considered under both the standard scenario and under a scenario with no COM activity (no Task 10C). In addition, all cases were studied with both the existing means of formating and with the hypothesized improvements. Table F.4 summarizes the results in terms of support processor loading.

Note that processor "B" seems to be committed to 9.5 CPU hours per hour when the standard scenario (interpretive formatting and COM output) is considered. A 10 processor system would satisfy such a requirement assuming a better than optimum multiprocessing system. If the COM formatting task is off-loaded, then only .88 CPU hour/ hour are committed.

Now consider how this load is distributed through the operational day. Figure F.3 shows a distribution of jobs over the day. This distribution is slightly simplified from the scenario given. In this case, 22 hours of operations were assumed. The loading shown is such that no workload case which represents long job overlaps with a work-load case which represents short jobs. When this distribution of jobs is assumed, the average SPS processor loading per shift can be determined. Figure F.4 shows this evaluation for processor B with no COM formatting.

Figure F.4 shows the same schedule of job execution as Figure F.3. The columns on the right side of the figure show the total CPU load determined for each case (output of the analyzer). The load for each case was then averaged over the shift in which it is scheduled to determine the average CPU loading over the shift. The loading is shown a CPU-hours per hour. Note that in Table F.4, .88 CPU-hours per hour is the average load over a day. However, Figure F.4 shows that when the load is distributed by shift, the peak load is 1.64 CPU-hours per hour.

Even this rate is optimistic since the actual loading of an interactive system is not uniformly distributed across the day. Loading will tend to have peaks and valleys. If there is a variation of 30% from the average, then the peak rate would be 2.13 processor - hours/hour. A trade-off between response-time and system complexity and cost must now be considered. By limiting the system to two processors of this sort, these processors would be busy most of the time, depending on load peaks which are not under the control of the operations staff.

Table F.4

Daily Average: Support Processing CPU Hours/Hour

PROCESSOR	FORMATTING	Scenario	
		With COM	Without COM
A	Interpretive	14.2	1.31
	Direct Execute	3.7	1.12
B	Interpretive	9.5	.88
	Direct Execute	3.3	.76
C	Interpretive	2.8	.19
	Direct Execute	.7	.15

The analysis described above is shown in Figure F.5 for processor "C" (a future processor). The case shown includes the COM output load and assumes the improved formatting rate. The analysis of loading due to COM formatting is an approximation unfortunately. The actual use planned is to produce graphics images on the COM device. A sequence of many frames would become a movie showing the dynamics of a model. Since no information concerning graphics formatting was available at the time, the approximation was made that COM formatting would be the same magnitude of task as formatting printed alphanumeric printout. This approximation may be somewhat optimisic.

Figure F.5 shows that peak CPU loads on the support processor occur during second and third shift. Note that the case shown in Figure F.4 has the amin load during prime-shift. The difference is the COM formatting which peaks in Cases 6 and 7 (see Task 10C in Table F.1). The loading shown in Figure F.5 indicates additional processor time available during the prime shift (8 am EST to 5 pm PST).

One variation of the scenario was tested to see the impact on support processor loading. The variation was to change the fraction of editing done on the support processor from 0.2 to 0.8. The increase in editing load brings the CPU loading from .315 to .317 CPU hours per hour (average over the shift).

F.4.2 File System Activity

In order to begin to evaluate the file system in detail, the major functional demands were determined, based upon the previously described scenarios. Two types of demands were considered; data transfer and control.

Data transfer demands were considered based on each of the interconnection paths in Figure F.1. The data transfer rates, averaged by day and by shift (as defined by Figure F.3) are shown in Table F.5. Here again, the rates are averaged either over the day or a shift and do not consider peak loading. It is interesting to note that if the Support Processor is relieved of the COM formatting task, the rates for Support Processor -- File System (corresponding to the first line of Table F.5) become 7.711K char/sec average over the day and the three shift rates become 56.90, 15,38K, and 44.5 char/sec respectively averaged uniformly over the shift. Again, the major reduction is during non-prime time.

Control functions were considered with respect to file activity. Based on the NASA-supplied scenarios, the number of file creations, file deletions, and file accesses per day were determined for the active high-speed access files, for the long-term, somewhat slower access files and for the slow access archive files. In addition, the number of times that an active file's contents were replaced by new contents was determined. These results are shown in Table F.6.

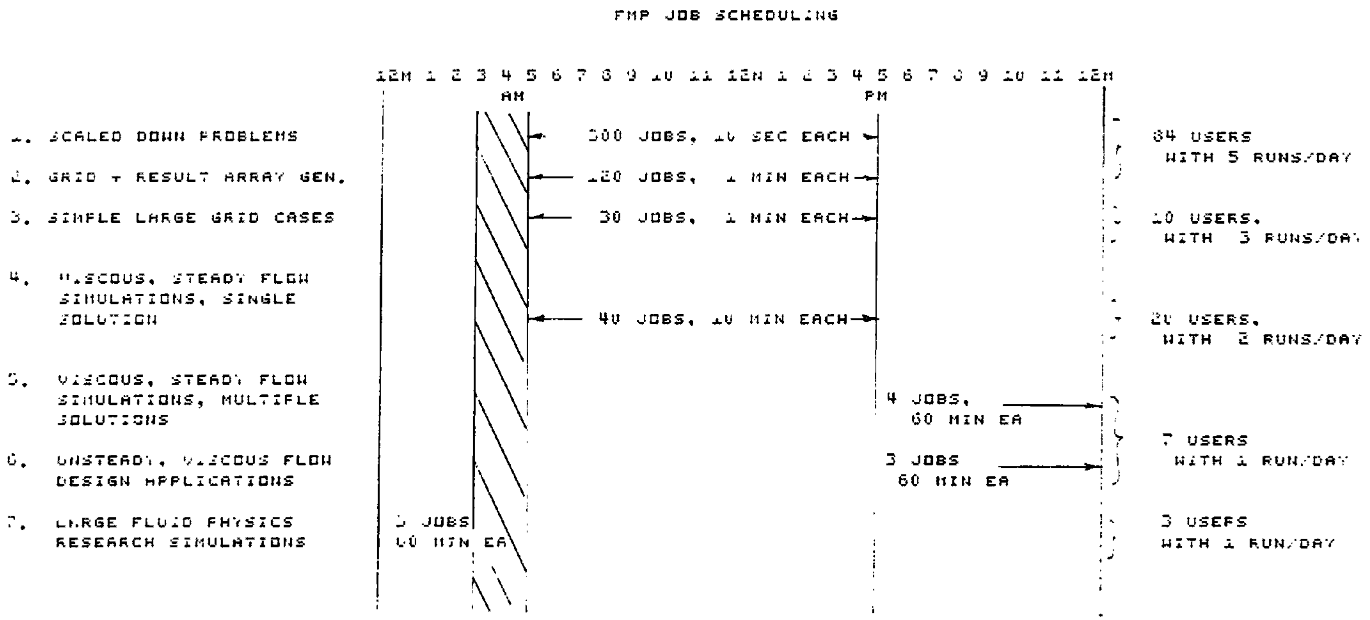


Figure F.3 NASF Job Load Scenario

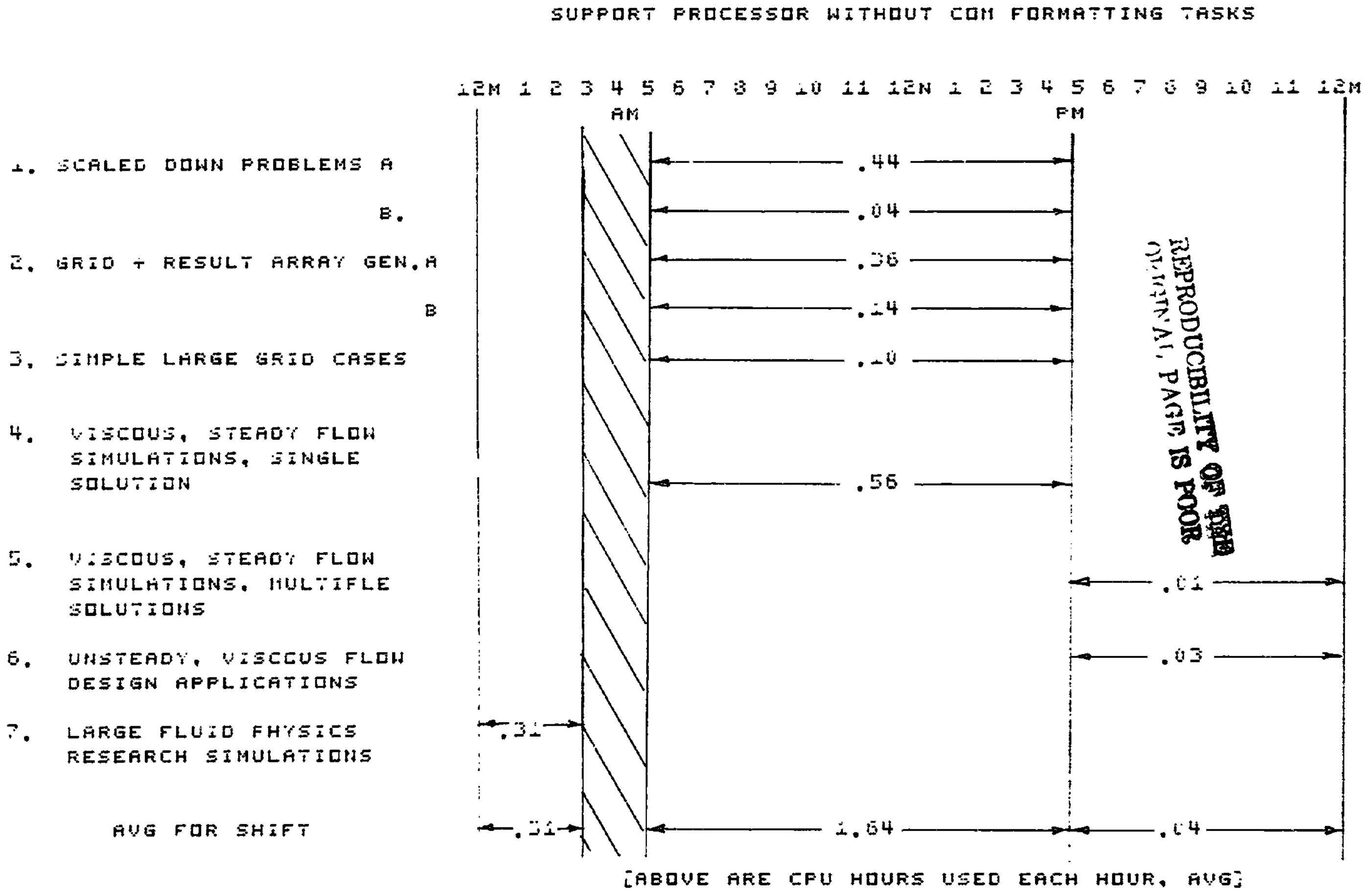


Figure F.4 Support Processor Average Load Per Shift

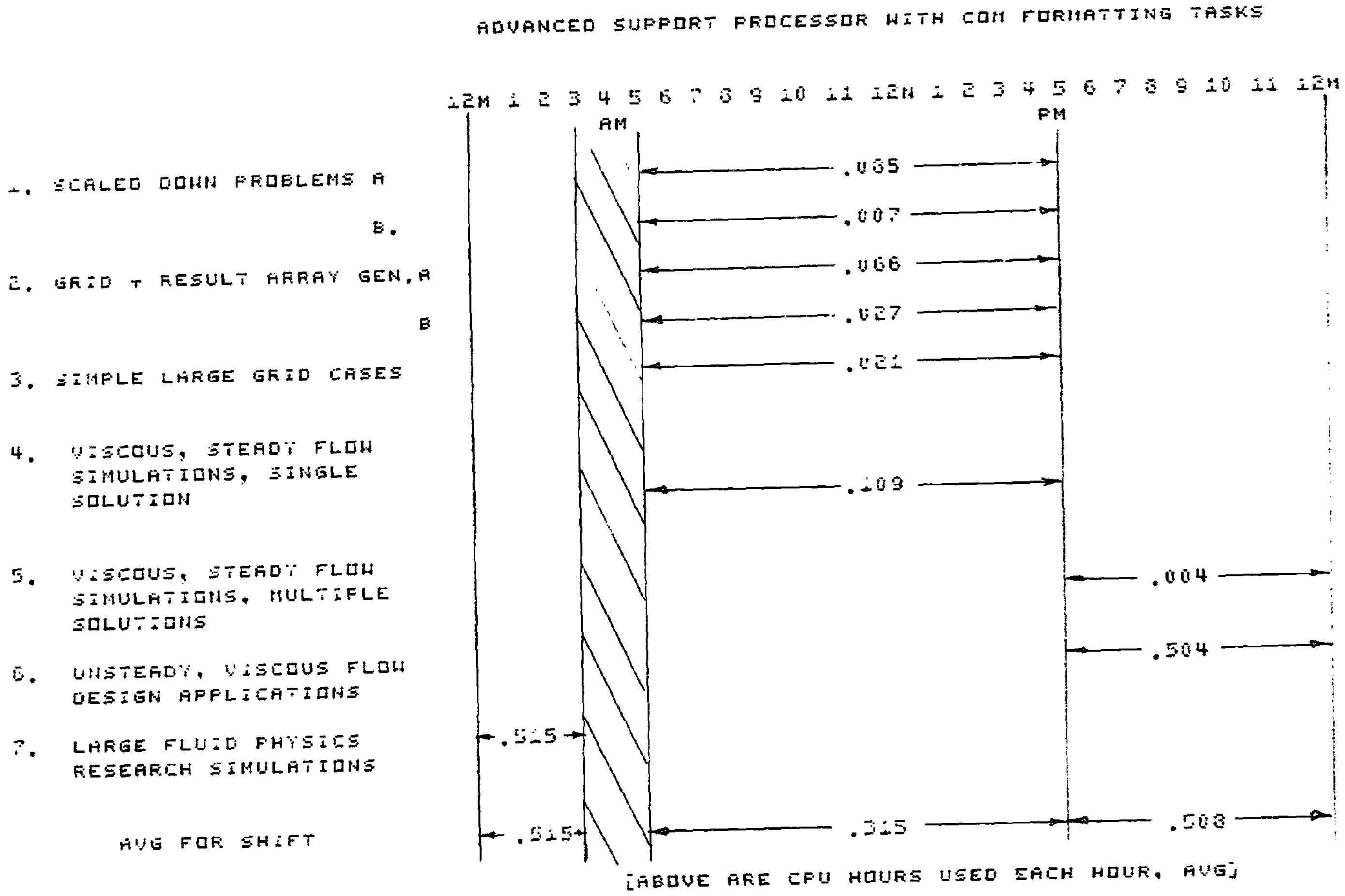


Figure F.5 Advanced Support Processor Average Load Per Shift

TABLE F.5

NASF DATA TRANSFER REQUIREMENTS
(with COM)

	RATE (Char/Sec)			
	Daily Average	Hourly Average		
		12M-3am	5am-5pm	5pm-12M
Support Processor - File System	29,240	83.388K	16.678K	35.937K
Support Processor - FMP	.050	.02	.08	.02
Support Processor - Users	4,453	.228K	8.125K	.187K
File System - Users	24,260	3.002K	45.9K	1.554K
File System - FMP	163,400	294.770K	210.032K	73.770K

TABLE F.6

NASF FILE SYSTEM CONTROL ACTIVITY PER DAY

FILE ACTIVITY	FILE TYPE		
	ACTIVE	LONGTERM	ARCHIVE
Files Created	2483	1127	627.3
Files Deleted	2483	1127	627.3
Files Accessed	19810	827.7	118.3
Files Replaced	1302	---	---

F.5 FUTURE WORK

In order to make the system analysis more accurate, benchmarks must be developed which represent the work to be supported by the Support Processor. The magnitude of editing, compiling, and linking can be determined given the existing codes and given the assumption that the compilers and linkers developed to support the FMP would be of the same complexity as that of the existing codes on existing machines. Benchmarks can be developed to study the actual formatting rate and the SPS committment to task management and I/O. More accurate estimates of the grid generation task and the interactive graphics support tasks need to made.

All system-level modelling must be operationally based. That is, the results of any system-level modelling should be easily verified by direct observation of an actual system. If this guideline is followed, verification of the models will become straight-forward.

REFERENCES

1. Final Report. NASF Preliminary Study, Contract NAS2-9456, Burroughs, October 1977.

2. Final Report, NASF Preliminary Study extension, constract NAS2-9456 Burroughs February 1978.

3. NASF Utilization, October 1978 Draft, NASA Ames

APPENDIX G

FMP FORTRAN EXAMPLES with ORIGINAL FORTRAN SOURCE

The listings which follow in this appendix are examples of FMP FORTRAN codes with their original FORTRAN source code. The FMP FORTRAN versions are all mentioned in Appendix A with regard to the analysis and simulation activities of the study. In some cases (e.g. LINKHO, COMP2 of the GISS weather codes, and the BTRI of the Implicit Aero Code), the resulting FMP FORTRAN is incomplete. These cases were taken only far enough to be able to generate an accurate timing since a functional simulation was not required at this time.

The listings provided are identified in the table below:

Figure	Application	Identification
G.1	Implicit Aero	Smooth
G.2	Implicit Aero	BTRI
G.3	Explicit Aero	OUTER
G.4	Explicit Aero	TURBDA
G.5	Explicit Aero	LX
G.6	GISS Weather	COMP2 Section
G.7	GISS Weather	LINKHO (part of COMP3)

Original FORTRAN

```
      SUBROUTINE SMOOTH                                                 SMOOTH 2
      COMMON/BASE/NMAX,JMAX,KMAX,LMAX,JM,KM,LM,DT,GAMMA,GAMI,SMU,FSMACH BASE   2
     1   ,DX1,DY1,DZ1,ND,ND2,FV(5),FD(5),HD,ALP,GD,OMEGA,HDX,HDY,HDZ    BASE   3
     2,RM,CNBR,PI,ITR,INVISC,LAMIN,NP,INT1,INT2,INT3                    BASE   4
      COMMON/GEO/NB1,NB2,RFRONT,RMAX,XR,XMAX,DRAD,DXC                   BASE   5
      COMMON/READ/IREAD,IWRIT,NGRI                                      BASE   6
      COMMON/VIS/RE,PR,RMUE,RK                                          BASE   7
      COMMON/VARS/Q(720,6,30)                                           VARS   2
      COMMON/VARU/S(720,5,30)                                           VARS   3
      COMMON/VAR1/X(720,30),Y(720,30),Z(720,30)                         VARS   4
      COMMON /VAR3/P(120,30),XX(60,4),YY(60,4),ZZ(60,4)                 VARS   5
      LEVEL 2,Q,S,X,Y,Z                                                 VARS   6
      COMMON /FLSH/DX2,DY2,DZ2                                          FLSH   2
      COMMON/IDX/LMM,KMM,JMM                                            IDX    2
C                                                                       SMOOTH 7
C   FOURTH-ORDER SMOOTHING                                              SMOOTH 8
C   SECOND ORDER AT THE BOUNDARIES                                      SMOOTH 9
C                                                                       SMOOTH10
      SM1 = SMU*.5                                                      SMOOTH11
      DO 10 L = 2,LM                                                    SMOOTH12
      DO 10 K = 2,KM                                                    SMOOTH13
      KL = (L-1)*ND+K                                                   SMOOTH14
      DO 12 J = 3,JMM                                                   SMOOTH15
      DO 12 N = 1,5                                                     SMOOTH16
   12 S(KL,N,J) = S(KL,N,J)-SMU*(Q(KL,N,J-2)*Q(KL,6,J-2)+Q(KL,N,J+2)*   SMOOTH17
     1 Q(KL,6,J+2)+6.*Q(KL,N,J)*Q(KL,6,J)-4.*Q(KL,N,J+1)*Q(KL,6,J+1)    SMOOTH18
     2  -4.*Q(KL,N,J-1)*Q(KL,6,J-1))/Q(KL,6,J)                          SMOOTH19
      DO 10 N = 1,5                                                     SMOOTH20
      S(KL,N,2) = S(KL,N,2)+SM1*(Q(KL,N,3)*Q(KL,6,3)-2.*Q(KL,N,2)*      SMOOTH21
     1  Q(KL,6,2)+Q(KL,N,1)*Q(KL,6,1))/Q(KL,6,2)                        SMOOTH22
      S(KL,N,JM) = S(KL,N,JM)+SM1*(Q(KL,N,JMAX)*Q(KL,6,JMAX)-2.*Q(KL,N, SMOOTH23
     1  JM)*Q(KL,6,JM)+Q(KL,N,JMM)*Q(KL,6,JMM))/Q(KL,6,JM)              SMOOTH24
   10 CONTINUE                                                          SMOOTH25
C                                                                       SMOOTH26
C                                                                       SMOOTH27
          DO 20 J = 2,JM                                                SMOOTH28
          DO 20 L = 2,LM                                                SMOOTH29
          DO 22 K = 3,KMM                                               SMOOTH30
          KL = (L-1)*ND+K                                               SMOOTH31
          DO 22 N = 1,5                                                 SMOOTH32
   22     S(KL,N,J) = S(KL,N,J)-SMU*(Q(KL+2,N,J)*Q(KL+2,6,J)+Q(KL-2,N,J)*SMOOTH33
```

Figure G.1 Implicit Aero - SMOOTH

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

```
      SUBROUTINE SMOOTH
       COMMON/BASE/NMAX,JMAX,KMAX,LMAX,DT,GAMMA,GAMI,FSMACH,
     1   DX1,DY1,DZ1,FV(5),FD(5),HD,ALP,GD,OMEGA,HDX,HDY,HDZ,RM,
     2   CNBR,PI,ITR,NP,INT1,INT2,INT3
      DOMAIN /MODEL/; J=1,100; K=1,50; L=1,200
      REGION /THREED((J=2,JMAX-1),(K=2,KMAX-1),(L=2,LMAX-1))/
     *    = /MODEL(J,K,L)/
      INALL /MODEL/ Q(6),S(5),SS,CT(5),T1,T2,T3,T4
C   4TH ORDER SMOOTHING, 2D ORDER AT THE BOUNDARIES
      DOALL /THREED(J,K,L)/ ;  USING Q, S, SMU
      TEMP = 1./Q(J,K,L,6)
      DO 1 N=1,5
      CT(N)= Q(J,K,L,N)*Q(J,K,L,6)
 1    CONTINUE
      IF (J.EQ.2 .OR. J.EQ.JMAX-1) THEN
        T1 = Q(J+1,K,L,6)
        T2 = Q(J-1,K,L,6)
        DO 2 N=1,5
        SS = S(J,K,L,N) + 0.5*SMU*(Q(J+1,K,L,N)*T1)- 2.*CT(N) +
     1         Q(J-1,K,L,N)*T2)*TEMP
 2      CONTINUE
       ELSE
        DO 3 N=1,5
        T1=Q(J+2,K,L,6)
        T2=Q(J-2,K,L,6)
        T3=Q(J+1,K,L,6)
        T4=Q(J-1,K,L,6)
        SS = S(J,K,L,N) + SMU*(Q(J+2,K,L,N)*T1 + Q(J-2,K,L,N)*T2 +
     1         4.*(Q(J+1,K,L,N)*T3 + Q(J-1,K,L,N)*T4) - 6.*CT(N))*TEMP
 3      CONTINUE
       ENDIF
      NEXTDO
       IF (K.EQ.2 .OR. K.EQ.KMAX-1) THEN
        T1=Q(J,K+1,L,6)
        T2=Q(J,K-1,L,6)
        DO 4 N=1,5
         SS = SS + 0.5*SMU*(Q(J,K+1,L,N)*T1 + Q(J,K-1,L,N)*T2 -
     1           2.*CT(N))*TEMP
 4      CONTINUE
```

Figure G.1   Implicit Aero - SMOOTH (Cont'd)

Original FORTRAN

```
     1 Q(KL-2,6,J)+6.*Q(KL,N,J)*Q(KL,6,J)-4.*Q(KL+1,N,J)*Q(KL+1,6,J)  SMOOTH34
     2 -4.*Q(KL-1,N,J)*Q(KL-1,6,J))/Q(KL,6,J)                          SMOOTH35
       DO 20 N = 1,5                                                    SMOOTH36
       KL = (L-1)*ND+2                                                  SMOOTH37
       S(KL,N,J) = S(KL,N,J)+SM1*(Q(KL+1,N,J)*Q(KL+1,6,J)-2.*Q(KL,N,J)SMOOTH38
     1 *Q(KL,6,J)+Q(KL-1,N,J)*Q(KL-1,6,J))/Q(KL,6,J)                   SMOOTH39
       KL = (L-1)*ND+KM                                                 SMOOTH40
       S(KL,N,J) = S(KL,N,J)+SM1*(Q(KL+1,N,J)*Q(KL+1,6,J)-2.*Q(KL,N,J)SMOOTH41
     1 *Q(KL,6,J)+Q(KL-1,N,J)*Q(KL-1,6,J))/Q(KL,6,J)                   SMOOTH42
   20  CONTINUE                                                         SMOOTH43
C                                                                       SMOOTH44
C                                                                       SMOOTH45
          DO 30 J = 2,JM                                                SMOOTH46
          DO 30 K = 2,KM                                                SMOOTH47
          DO 32 L = 3,LMM                                               SMOOTH48
          KL = (L-1)*ND+K                                               SMOOTH49
          I1 = KL+ND                                                    SMOOTH50
          I2 = KL+2*ND                                                  SMOOTH51
          I3 = KL-ND                                                    SMOOTH52
          I4 = KL-2*ND                                                  SMOOTH53
          DO 32 N = 1,5                                                 SMOOTH54
   32     S(KL,N,J) = S(KL,N,J)-SMU*(Q(I2,N,J)*Q(I2,6,J)+Q(I4,N,J)*   SMOOTH55
     1    Q(I4,6,J)+6.*Q(KL,N,J)*Q(KL,6,J)-4.*Q(I1,N,J)*Q(I1,6,J)-4.* SMOOTH56
     2    Q(I3,N,J)*Q(I3,6,J))/Q(KL,6,J)                                SMOOTH57
          DO 30 N = 1,5                                                 SMOOTH58
          KL = ND+K                                                     SMOOTH59
          I1 = KL+ND                                                    SMOOTH60
          I2 = KL-ND                                                    SMOOTH61
          S(KL,N,J) = S(KL,N,J)+SM1*(Q(I1,N,J)*Q(I1,6,J)-2.*Q(KL,N,J)*SMOOTH62
     1    Q(KL,6,J)+Q(I2,N,J)*Q(I2,6,J))/Q(KL,6,J)                      SMOOTH63
          KL = LMM*ND+K                                                 SMOOTH64
          I1 = KL+ND                                                    SMOOTH65
          I2 = KL-ND                                                    SMOOTH66
          S(KL,N,J) = S(KL,N,J)+SM1*(Q(I1,N,J)*Q(I1,6,J)-2.*Q(KL,N,J)*SMOOTH67
     1    Q(KL,6,J)+Q(I2,N,J)*Q(I2,6,J))/Q(KL,6,J)                      SMOOTH68
   30     CONTINUE                                                      SMOOTH69
      RETURN                                                            SMOOTH70
      END                                                               SMOOTH71
```

Figure G.1 Implicit Aero - SMOOTH (Cont'd)

```
       ELSE
        T1=Q(J,K+2,L,6)
        T2=Q(J,K-2,L,6)
        T3=Q(J,K+1,L,6)
        T4=Q(J,K-1,L,6)
        DO 5 N=1,5
         SS=SS+SMU*(Q(J,K+2,L,N)*T1 + Q(J,K-2,L,N)*T2 +
     1      4.*(Q(J,K+1,L,N)*T3 + Q(J,K-1,L,N)*T4) - 6,*CT(N))*TEMP
5       CONTINUE
        ENDIF
        NEXTDO
         IF (L.EQ.2 .OR. L.EQ.LMAX-1) THEN
          T1=Q(J,K,L+1,6)
          T2=Q(J,K,L-1,6)
          DO 6 N=1,5
           S(J,K,L,N) = SS + 0,5*SMU*(Q(J,K,L+1,N)*T1 +
                         Q(J,K,L-1,N)*T2 - 2.*CT(N))*TEMP
6         CONTINUE
         ELSE
          T1 = Q(J,K,L+2,6)
          T2 = Q(J,K,L-2,6)
          T3 = Q(J,K,L+1,6)
          T4 = Q(J,K,L-1,6)
          DO 7 N=1,5
           S(J,K,L,N) = SS + SMU*(Q(J,K,L+2,N)*T1 + Q(J,K,L-2,N)*T2+
     1                  4.*Q(J,K,L+1,N)*T3 + 4.*Q(J,K,L-1,N)*T4
     2                  - 6.*CT(N))*TEMP
7         CONTINUE
         ENDIF
       ENDDO /THREED/ ;  GIVING S
      RETURN
      END
+
```

Figure G.1   Implicit Aero - SMOOTH (Cont'd)

Original FORTRAN

```
+FILE (F1000015450SAM)IMPLICIT/BTRI ON NSS
      SUBROUTINE BTRI(ILA,IUA)
      COMMON/BTRID/A(60,5,5),B(60,5,5),C(60,5,5),D(60,5,5),F(60,5)
      DIMENSION H(5,5)
      REAL L11,L21,L22,L31,L32,L33,L41,L42,L43,L44,L51,L52,L53,L54,L55
      IL=ILA
      IU=IUA
      IS=IL+1
      IE=IU-1
C     INSERT LUDEC
      L11=1./B(IL,1,1)
      L21=B(IL,2,1)
      U12=B(IL,1,2)*L11
      L22=1./(B(IL,2,2)-L21*U12)
      U13=B(IL,1,3)*L11
      U14 = B(IL,1,4)*L11
      U15=B(IL,1,5)*L11
      L31=B(IL,3,1)
      L32=B(IL,3,2)-L31*U12
      U23=(B(IL,2,3)-L21*U13)*L22
      L33=1./(B(IL,3,3)-U13*L31-U23*L32)
      U24=(B(IL,2,4)-L21*U14)*L22
      U25=(B(IL,2,5)-L21*U15)*L22
      L41=B(IL,4,1)
      L42=B(IL,4,2)-L41*U12
      L43=B(IL,4,3)-L41*U13-L42*U23
      U34=(B(IL,3,4)-L31*U14-L32*U24)*L33
      L44=1./(B(IL,4,4)-U14*L41-U24*L42-U34*L43)
      U35=(B(IL,3,5)-L31*U15-L32*U25)*L33
      L51=B(IL,5,1)
      L52=B(IL,5,2)-L51*U12
      L53=B(IL,5,3)-L51*U13-L52*U23
      L54=B(IL,5,4)-L51*U14-L52*U24-L53*U34
      U45=(B(IL,4,5)-L41*U15-L42*U25-L43*U35)*L44
      L55=1./(B(IL,5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
C     COMPUTE LITTLE R S
      D1=L11*F(IL,1)
      D2=L22*(F(IL,2)-L21*D1)
      D3=L33*(F(IL,3)-L31*D1-L32*D2)
      D4=L44*(F(IL,4)-L41*D1-L42*D2-L43*D3)
      D5=L55*(F(IL,5)-L51*D1-L52*D2-L53*D3-L54*D4)
C     COMPUTE BIG R S
      F(IL,5)=D5
      F(IL,4)=D4-U45*D5
      F(IL,3)=D3-U34*F(IL,4)-U35*D5
      F(IL,2)=D2-U23*F(IL,3)-U24*F(IL,4)-U25*D5
      F(IL,1)=D1-U12*F(IL,2)-U13*F(IL,3)-U14*F(IL,4)-U15*D5
```

Figure G.2 Implicit Aero - BTRI

FMP FORTRAN

```
       SUBROUTINE BTRI(IUA)
C
C     ASSUME STARTING INDEX = 1
C
      COMMON /BTRID/ A(IUA,5,5), B(IUA,5,5), C(IUA,5,5),
     1  D(IUA,5,5), F(IUA,5)
      DIMENSION H(5,5)
      IMPLICIT REAL(L)
C
C     INSERT LUDEC (SIMPLIFIED FOR DIAGONAL INPUT ARRAY B) FOR I=1
C
      L11 = 1./B(1,1,1)
      L22 = 1./B(1,2,2)
      L33 = 1./B(1,3,3)
      L44 = 1./B(1,4,4)
      L55 = 1./B(1,5,5)
C
C     COMPUTE LITTLE R'S OMITTED, THESE TEMPORARIES NOT NEEDED
C     THIS PASS, COMPUTE BIG R'S
C
       F(1,5) = L55
       F(1,4) = L44
       F(1,3) = L33
       F(1,2) = L22
       F(1,1) = L11
```

Figure G.2   Implicit Aero - BTRI (Cont'd)

Original FORTRAN

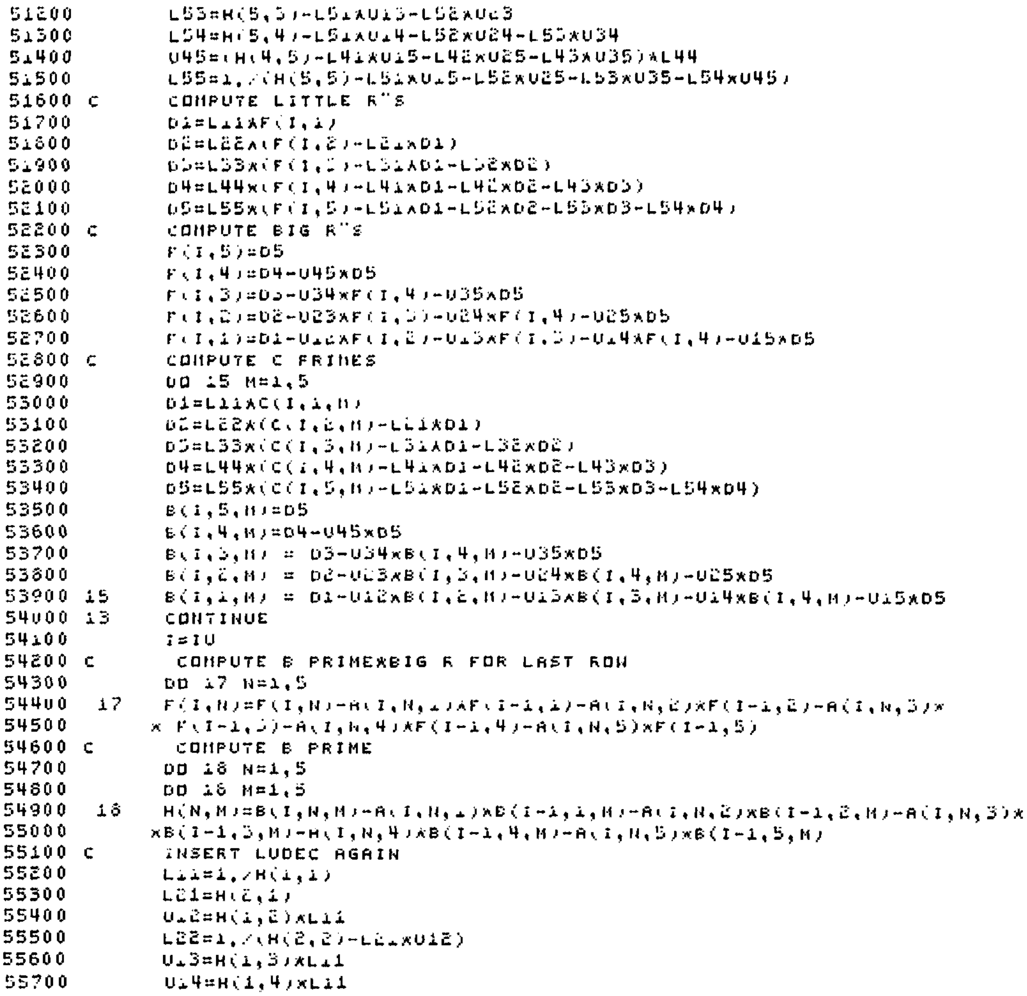
```
C     COMPUTE C PRIME FOR FIRST ROW
      DO 12 M=1,5
      D1=L11*C(IL,1,M)
      D2=L22*(C(IL,2,M)-L21*D1)
      D3=L33*(C(IL,3,M)-L31*D1-L32*D2)
      D4=L44*(C(IL,4,M)-L41*D1-L42*D2-L43*D3)
      D5=L55*(C(IL,5,M)-L51*D1-L52*D2-L53*D3-L54*D4)
      B(IL,5,M)=D5
      B(IL,4,M)=D4-U45*D5
      B(IL,3,M) = D3-U34*B(IL,4,M)-U35*D5
      B(IL,2,M) = D2-U23*B(IL,3,M)-U24*B(IL,4,M)-U25*D5
12    B(IL,1,M) = D1-U12*B(IL,2,M)-U13*B(IL,3,M)-U14*B(IL,4,M)-U15*D5
      DO 13 I=IS,IE
C      COMPUTE B PRIME*BIGR
      DO 14 N=1,5
 14   F(I,N)=F(I,N)-A(I,N,1)*F(I-1,1)-A(I,N,2)*F(I-1,2)-A(I,N,3)*F(I-1,3
     *)-A(I,N,4)*F(I-1,4)-A(I,N,5)*F(I-1,5)
C      COMPUTE B PRIME
      DO 11 N=1,5
      DO 11 M=1,5
 11   H(N,M)=B(I,N,M)-A(I,N,1)*B(I-1,1,M)-A(I,N,2)*B(I-1,2,M)-A(I,N,3)*
     *B(I-1,3,M)-A(I,N,4)*B(I-1,4,M)-A(I,N,5)*B(I-1,5,M)
C     INSERT LUDEC AGAIN
      L11=1./H(1,1)
      L21=H(2,1)
      U12=H(1,2)*L11
      L22=1./(H(2,2)-L21*U12)
      U13=H(1,3)*L11
      U14=H(1,4)*L11
      U15=H(1,5)*L11
      L31=H(3,1)
      L32=H(3,2)-L31*U12
      U23=(H(2,3)-L21*U13)*L22
      L33=1./(H(3,3)-U13*L31-U23*L32)
      U24=(H(2,4)-L21*U14)*L22
      U25=(H(2,5)-L21*U15)*L22
      L41=H(4,1)
      L42=H(4,2)-L41*U12
      L43=H(4,3)-L41*U13-L42*U23
      U34=(H(3,4)-L31*U14-L32*U24)*L33
      L44=1./(H(4,4)-U14*L41-U24*L42-U34*L43)
      U35=(H(3,5)-L31*U15-L32*U25)*L33
      L51=H(5,1)
      L52=H(5,2)-L51*U12
```

Figure G.2   Implicit Aero - BTRI (Cont'd)

```
C
C       COMPUTE C PRIME FOR FIRST ROW
C
        DO 12 M = 1,5
C
C         C HAS BEEN ELIMINATED AS A SIMPLE
C         RESUBSCRIPTING OF THE D ARRAY
C
          B(1,5,M) = L55 x C(I,5,M)
          B(1,4,M) = L44 x C(I,4,M)
          B(1,3,M) = L33 x C(I,3,M)
          B(1,2,M) = L22 x C(I,2,M)
          B(1,1,M) = L11 x C(I,1,M)
12      CONTINUE
C
C         HERE NOW STARTS THE MAIN LOOP OF BTRI
C
         DO 13 I = 2,IUA
C
C         COMPUTE B PRIME * BIGR
C
          DO 14 N=1,5
14          F(I,N) = F(I,N) - A(I,N,1) x F(I-1,1) - A(I,N,2) x
     1               F(I-1,2) - A(I,N,3) x F(I-1,3) - A(I,N,4) x
     2               F(I-1,4) - A(I,N,5) x F(I-1,5)
C
C             COMPUTE B PRIME
C
               DO 11 N = 1,5
                DO 11 M = 1,5
11               H(N,M) = B(I,N,M) - A(I,N,1) x B(I-1,1,M) -
     1                    A(I,N,2) x B(I-1,2,M) - A(I,N,C) x
     2                    B(I-1,3,M) - A(I,N,4) x B(I-1,4,M) -
     3                    A(I,N,5) * B(I-1,5,M)
C
C     INSERT LUDEC AGAIN
C
             .
             .
      HERE SHALL BE INSERTED A COPY OF THE FORMER LUDEC,
      EXACTLY AS SHOWN IN THE IMPLICIT CODE COMPILATION BY SCHAEFFER
              .
              .
```

Figure G.2 Implicit Aero - BTRI (Cont'd)

Original FORTRAN



```
      L53=H(5,3)-L51*U13-L52*U23
      L54=H(5,4)-L51*U14-L52*U24-L53*U34
      U45=(H(4,5)-L41*U15-L42*U25-L43*U35)*L44
      L55=1./(H(5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
C     COMPUTE LITTLE R"S
      D1=L11*F(I,1)
      D2=L22*(F(I,2)-L21*D1)
      D3=L33*(F(I,3)-L31*D1-L32*D2)
      D4=L44*(F(I,4)-L41*D1-L42*D2-L43*D3)
      D5=L55*(F(I,5)-L51*D1-L52*D2-L53*D3-L54*D4)
C     COMPUTE BIG R"S
      F(I,5)=D5
      F(I,4)=D4-U45*D5
      F(I,3)=D3-U34*F(I,4)-U35*D5
      F(I,2)=D2-U23*F(I,3)-U24*F(I,4)-U25*D5
      F(I,1)=D1-U12*F(I,2)-U13*F(I,3)-U14*F(I,4)-U15*D5
C     COMPUTE C FRIMES
      DO 15 M=1,5
      D1=L11*C(I,1,M)
      D2=L22*(C(I,2,M)-L21*D1)
      D3=L33*(C(I,3,M)-L31*D1-L32*D2)
      D4=L44*(C(I,4,M)-L41*D1-L42*D2-L43*D3)
      D5=L55*(C(I,5,M)-L51*D1-L52*D2-L53*D3-L54*D4)
      B(I,5,M)=D5
      B(I,4,M)=D4-U45*D5
      B(I,3,M) = D3-U34*B(I,4,M)-U35*D5
      B(I,2,M) = D2-U23*B(I,3,M)-U24*B(I,4,M)-U25*D5
 15   B(I,1,M) = D1-U12*B(I,2,M)-U13*B(I,3,M)-U14*B(I,4,M)-U15*D5
 13   CONTINUE
      I=IU
C      COMPUTE B PRIME*BIG R FOR LAST ROW
      DO 17 N=1,5
  17  F(I,N)=F(I,N)-A(I,N,1)*F(I-1,1)-A(I,N,2)*F(I-1,2)-A(I,N,3)*
     * F(I-1,3)-A(I,N,4)*F(I-1,4)-A(I,N,5)*F(I-1,5)
C      COMPUTE B PRIME
      DO 18 N=1,5
      DO 18 M=1,5
  18  H(N,M)=B(I,N,M)-A(I,N,1)*B(I-1,1,M)-A(I,N,2)*B(I-1,2,M)-A(I,N,3)*
     *B(I-1,3,M)-A(I,N,4)*B(I-1,4,M)-A(I,N,5)*B(I-1,5,M)
C     INSERT LUDEC AGAIN
      L11=1./H(1,1)
      L21=H(2,1)
      U12=H(1,2)*L11
      L22=1./(H(2,2)-L21*U12)
      U13=H(1,3)*L11
      U14=H(1,4)*L11
```

Figure G.2 Implicit Aero - BTRI (Cont'd)

```
C
C       COMPUTE LITTLE R'S
C
        D1 = L11 x F(I,1)
        D2 = L22 x (F(I,2) - L21 x D1)
        D3 = L33 x (F(I,3) - L31 x D1 - L32 x D2)
        D4 = L44 x (F(I,4) - L41 x D1 - L42 x D2 - L43 x D3)
        D5 = L55x(F(I,5) - L51xD1 - L52xD2 - L53xD3 - L54xD4)
C
C      COMPUTE BIG R'S
C
        F(I,5) = D5
        F(I,4) = D4 -U45xD5
        F(I,3) = D3 - U34xF(I,4) - U35xD5
        F(I,2) = D2 - U23xF(I,3) - U24xF(I,4) - U25xD5
        F(I,1) = D1 - U12xF(I,2) - U13xF(I,3) - U14xF(I,4) - U15xD5
        IF (I .LT. IUA) THEN
         DO 15 M = 1,5
          D1 = L11xC(I,1,M)
          D2 = L22x(C(I,2,M) - L21xD1)
          D3 = L33x(C(I,3,M) - L31xD1 - L32xD2)
          D4 = L44x(C(I,4,M) - L41xD1 - L42xD2 - L43xD3)
           D5 = L55x(C(I,5,M) - L51xD1 - L52xD2 - L53xD3 - L54xD4)
           B(I,5,M) = D5
           B(I,4,M) = D4 - U45xD5
           B(I,3,M) = D3 - U34xC(I,4,M) - U35xD5
           B(I,2,M) = D2 - U23xB(I,3,M) - U24xB(I,4,M) - U25xD5
           B(I,1,M) = D1 - U12xB(I,2,M) - U13xB(I,3,M) - U14xB(I,4,M)
     1                  - U15xD5
   15     CONTINUE
         ENDIF
   13   CONTINUE
C
C        THIS IS THE END OF THE MAIN I LOOP, INCLUDING I=IUA
```

Figure G.2   Implicit Aero - BTRI (Cont'd)

Original FOTRAN

```
      U15=H(1,5)*L11
      L31=H(3,1)
      L32=H(3,2)-L31*U12
      U23=(H(2,3)-L21*U13)*L22
      L33=1./(H(3,3)-U13*L31-U23*L32)
      U24=(H(2,4)-L21*U14)*L22
      U25=(H(2,5)-L21*U15)*L22
      L41=H(4,1)
      L42=H(4,2)-L41*U12
      L43=H(4,3)-L41*U13-L42*U23
      U34=(H(3,4)-L31*U14-L32*U24)*L33
      L44=1./(H(4,4)-U14*L41-U24*L42-U34*L43)
      U35=(H(3,5)-L31*U15-L32*U25)*L33
      L51=H(5,1)
      L52=H(5,2)-L51*U12
      L53=H(5,3)-L51*U13-L52*U23
      L54=H(5,4)-L51*U14-L52*U24-L53*U34
      U45=(H(4,5)-L41*U15-L42*U25-L43*U35)*L44
      L55=1./(H(5,5)-L51*U15-L52*U25-L53*U35-L54*U45)
C     COMPUTE LITTLE R"S
      D1=L11*F(I,1)
      D2=L22*(F(I,2)-L21*D1)
      D3=L33*(F(I,3)-L31*D1-L32*D2)
      D4=L44*(F(I,4)-L41*D1-L42*D2-L43*D3)
      D5=L55*(F(I,5)-L51*D1-L52*D2-L53*D3-L54*D4)
C     COMPUTE BIG R"S
      F(I,5)=D5
      F(I,4)=D4-U45*D5
      F(I,3)=D3-U34*F(I,4)-U35*D5
      F(I,2)=D2-U23*F(I,3)-U24*F(I,4)-U25*D5
      F(I,1)=D1-U12*F(I,2)-U13*F(I,3)-U14*F(I,4)-U15*D5
      I=IU
20    I=I-1
      DO 19 N=1,5
 19   F(I,N)=F(I,N)-F(I+1,1)*B(I,N,1)-F(I+1,2)*B(I,N,2)-F(I+1,3)*B(I,N,3
     * )-F(I+1,4)*B(I,N,4)-F(I+1,5)*B(I,N,5)
      IF (I.GT.IL)GOTO20
      RETURN
      END
```

Figure G.2   Implicit Aero - BTRI (Cont'd)

FMP FORTRAN

```
C
C      NOTE THE NEGATIVE CODE INCREMENTS IN THE NEXT SECTION
C
      DO 20 I = IUA-1, 1, -1
       DO 19 N=1,5
19      F(I,N) = F(I,N) - F(I+1,1)*B(I,N,1) - F(I+1,2)*B(I,N,2)
20     CONTINUE
      RETURN
      END
←
```

Figure G.2   Implicit Aero - BTRI (Cont'd)

```
      SUBROUTINE OUTER(JS,JE,KS,KE)
C        XX OUTER BOUNDARY CONDITIONS  XX
CC
CCALL A1
CCALL A3
CCALL A4
      COMMON/A11/ RHO(31,31,31),RHOU(31,31,31),RHOV(31,31,31)
      COMMON/A12/  RHOW(31,31,31),E(31,31,31),EI(31,31,31)
      COMMON/A13/ U(31,31,31),V(31,31,31),W(31,31,31)
      COMMON/A3/ Y(31),DYCELL(31),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1            ,Z(31),DZCELL(31),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
CXXX   DOWNSTREAM AT I=IL
      DO 1 K=KS,KE
      DO 1 J=JS,JE
      RHO (IL,J,K)=RHO (IE,J,K)
      RHOU(IL,J,K)=RHOU(IE,J,K)
      RHOV(IL,J,K)=RHOV(IE,J,K)
      RHOW(IL,J,K)=RHOW(IE,J,K)
      E   (IL,J,K)=E   (IE,J,K)
    1 CONTINUE
      IF(JE.LT.JE2) GO TO 3
CXXX   UPPER B. C. AT J=JL
      DO 2 K=KS,KE
      DO 2 I=2, IE
      RHO (I,JL,K)=RHO (I,JE2,K)
      RHOU(I,JL,K)=RHOU(I,JE2,K)
      RHOV(I,JL,K)=RHOV(I,JE2,K)
      RHOW(I,JL,K)=RHOW(I,JE2,K)
      E   (I,JL,K)=E   (I,JE2,K)
    2 CONTINUE
    3 CONTINUE
      IF(KE.LT.KE2) RETURN
CXXX   EDGE B. C. AT K=KL
      DO 4 J=JS,JE
      DO 4 I=2, IE
      RHO (I,J,KL)=RHO (I,J,KE2)
      RHOU(I,J,KL)=RHOU(I,J,KE2)
      RHOV(I,J,KL)=RHOV(I,J,KE2)
      RHOW(I,J,KL)=RHOW(I,J,KE2)
      E   (I,J,KL)=E   (I,J,KE2)
    4 CONTINUE
      RETURN
      END
```

Figure G.3 Explicit Aero - OUTER

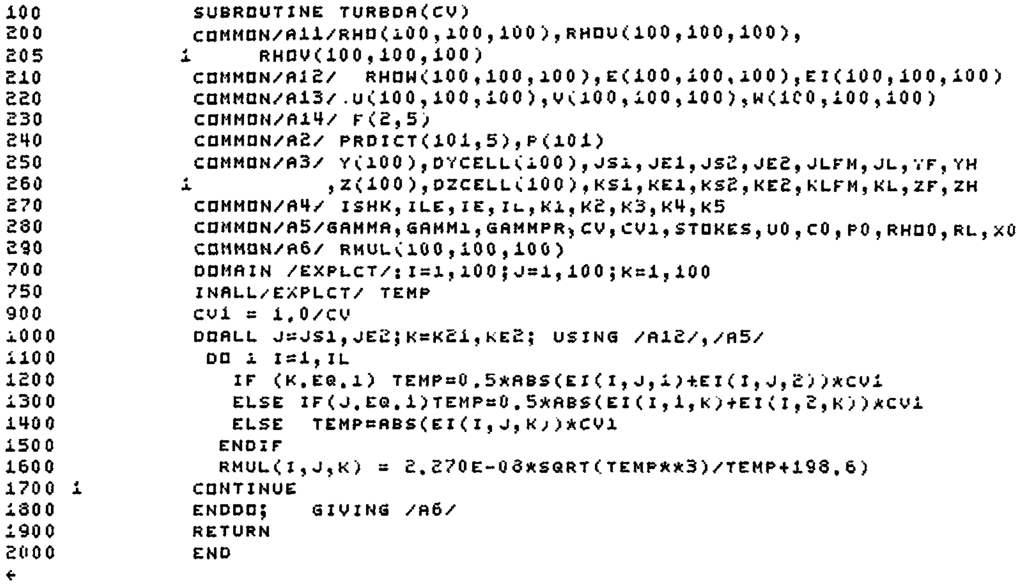
```
      SUBROUTINE OUTER(JS,JE,KS,KE)
      COMMON/A11/ RHO(100,100,100),RHOU(100,100,100),RHOV(100,100,100)
      COMMON/A12/  RHOW(100,100,100),E(100,100,100),EI(100,100,100)
      COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
      COMMON/A3/ Y(100),DYCELL(100),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1           ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
C       DOWNSTREAM AT I=IL
      DOALL K=KS,KE;J=JS,JE ;  USING/A11/,/A12/,/A3/,/A4/
       RHO(IL,J,K) = RHO(IE,J,K)
       RHOU(IL,J,K) = RHOU(IE,J,K)
       RHOV(IL,J,K) = RHOV(IE,J,K)
       RHOW(IL,J,K) = RHOW(IE,J,K)
       E(IL,J,K) = E(IE,J,K)
      ENDDO ;  GIVING /A11/,/A12/
        IF (JE,LT,JE2) GO TO 3
C       UPPER B. C. AT J=JL
      DOALL K=KS,KE;I=2,IE ;  USING/A11/,/A12/./A3/,/A4/
       RHO(I,JK,K) = RHO(I,JE2,K)
       RHOU(I,JK,K) = RHOU(I,JE2,K)
      RHOV(I,JK,K) = RHOV(I, JE2,K)
       RHOW(I,JK,K) = RHOW(I,JE2,K)
       E(I,JK,K) = E(I,JE2,K)
      ENDDO; GIVING /A11/,/A12/
3      IF (K.GE.KE2) THEN
C         EDGE B.C. AT K=KL
        DOALL J=JS,JE;I=2,IE ;  USING /A11/,/A12/,/A3/,/A4/
         RHO(I,J,KL) = RHO(I,J,KE2)
         RHOU(I,J,KL) = RHOU(I,J,KE2)
         RHOV(I,J,KL) = RHOV(I,J,KE2)
         RHOW(I,J,KL) = RHOW(I,J,KE2)
         E(I,J,KL) = E(I,J,KE2)
        ENDDO; GIVING /A11/,/A12/
       ENDIF
     RETURN
    END
```

Figure G.3   Explicit Aero - OUTER (Cont'd)

Original FORTRAN

```
      SUBROUTINE TURBDA
CCALL A1
CCALL A3
CCALL A4
CCALL A5
CCALL A6
      COMMON/A11/ RHO(31,31,31),RHOU(31,31,31),RHOV(31,31,31)
      COMMON/A12/  RHOW(31,31,31),E(31,31,31),EI(31,31,31)
      COMMON/A13/ U(31,31,31),V(31,31,31),W(31,31,31)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PRDICT(32,5),P(32)
      COMMON/A3/ Y(31),DYCELL(31),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1          ,Z(31),DZCELL(31),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/ GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,UU,C0,PU,RHOU,RL,XU
      COMMON/A6/ RMUL(31,31,31)
      CV1=1./CV
      DO 1 K=1,KL
      DO 1 J=1,JL
      DO 1 I=1,IL
      TEMP=ABS(EI(I,J,K))*CV1
      IF(K.EQ.1) TEMP=.5*ABS(EI(I,J,1)+EI(I,J,2))*CV1
      IF(J.EQ.1) TEMP=.5*ABS(EI(I,1,K)+EI(I,2,K))*CV1
      RMUL(I,J,K)=2.270E-08*SQRT(TEMP**3)/(TEMP+198.6)
    1 CONTINUE
      RETURN
      END
←
```

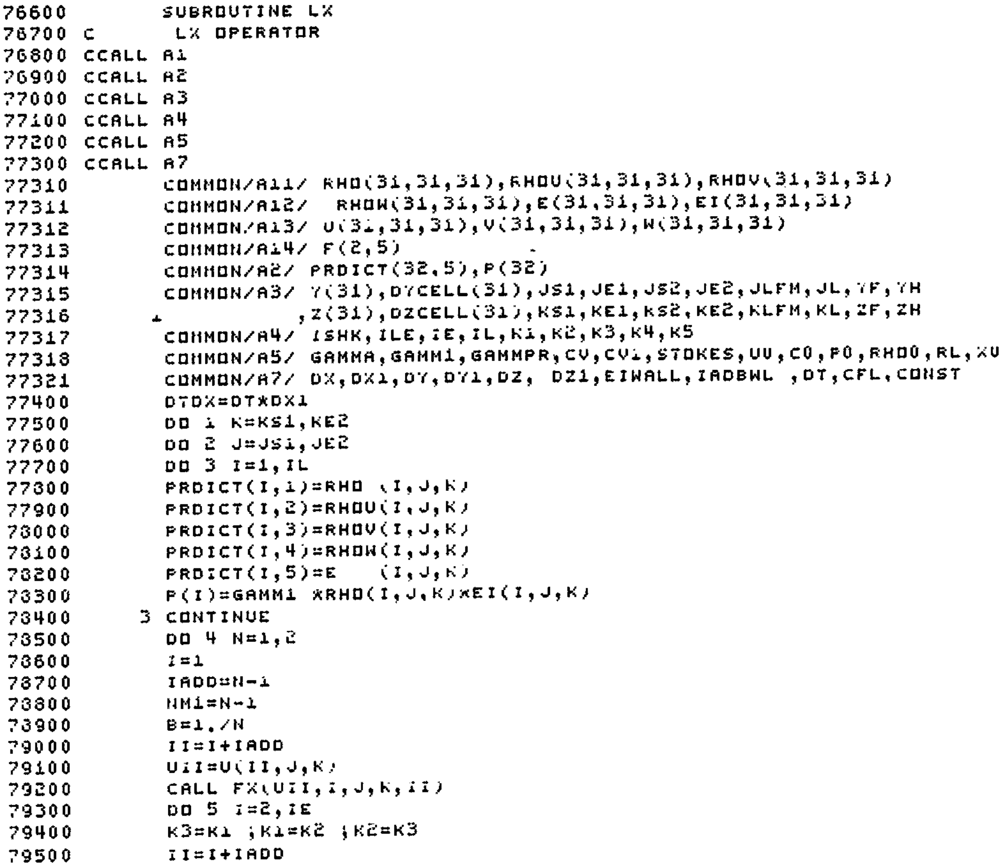
Figure G.4   Explicit Aero - TURBDA



```
      SUBROUTINE TURBDA(CV)
      COMMON/A11/RHO(100,100,100),RHOU(100,100,100),
     1     RHOV(100,100,100)
      COMMON/A12/  RHOW(100,100,100),E(100,100,100),EI(100,100,100)
      COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PRDICT(101,5),P(101)
      COMMON/A3/ Y(100),DYCELL(100),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1          ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,U0,C0,P0,RHO0,RL,X0
      COMMON/A6/ RMUL(100,100,100)
      DOMAIN /EXPLCT/:I=1,100;J=1,100;K=1,100
      INALL/EXPLCT/ TEMP
      CV1 = 1,0/CV
      DOALL J=JS1,JE2;K=K21,KE2; USING /A12/,/A5/
       DO 1 I=1,IL
         IF (K.EQ.1) TEMP=0.5*ABS(EI(I,J,1)+EI(I,J,2))*CV1
         ELSE IF(J.EQ.1)TEMP=0.5*ABS(EI(I,1,K)+EI(I,2,K))*CV1
         ELSE  TEMP=ABS(EI(I,J,K))*CV1
        ENDIF
        RMUL(I,J,K) = 2.270E-08*SQRT(TEMP**3)/TEMP+198.6)
 1    CONTINUE
      ENDDO;   GIVING /A6/
      RETURN
      END
←
```

Figure G.4   Explicit Aero - TURBDA (Cont'd)

Original FORTRAN



```
      SUBROUTINE LX
C      LX OPERATOR
CCALL A1
CCALL A2
CCALL A3
CCALL A4
CCALL A5
CCALL A7
      COMMON/A11/ RHO(31,31,31),RHOU(31,31,31),RHOV(31,31,31)
      COMMON/A12/  RHOW(31,31,31),E(31,31,31),EI(31,31,31)
      COMMON/A13/ U(31,31,31),V(31,31,31),W(31,31,31)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PRDICT(32,5),P(32)
      COMMON/A3/ Y(31),DYCELL(31),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1          ,Z(31),DZCELL(31),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/ GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,UU,C0,P0,RHO0,RL,XU
      COMMON/A7/ DX,DX1,DY,DY1,DZ, DZ1,EIWALL,IADBWL ,DT,CFL,CONST
      DTDX=DT*DX1
      DO 1 K=KS1,KE2
      DO 2 J=JS1,JE2
      DO 3 I=1,IL
      PRDICT(I,1)=RHO (I,J,K)
      PRDICT(I,2)=RHOU(I,J,K)
      PRDICT(I,3)=RHOV(I,J,K)
      PRDICT(I,4)=RHOW(I,J,K)
      PRDICT(I,5)=E    (I,J,K)
      P(I)=GAMM1 *RHO(I,J,K)*EI(I,J,K)
    3 CONTINUE
      DO 4 N=1,2
      I=1
      IADD=N-1
      NM1=N-1
      B=1./N
      II=I+IADD
      UII=U(II,J,K)
      CALL FX(UII,I,J,K,II)
      DO 5 I=2,IE
      K3=K1 ;K1=K2 ;K2=K3
      II=I+IADD
```

Figure G.5   Explicit Aero - LX

```
      SUBROUTINE LX
C      LX OPERATOR
      COMMON/A11/ RHO(100,100,100),RHOU(100,100,100),RHOV(100,100,100)
      COMMON/A12/  RHOW(100,100,100),E(100,100,100),EI(100,100,100)
      COMMON/A13/ U(100,100,100),V(100,100,100),W(100,100,100)
      COMMON/A14/ F(2,5)
      COMMON/A2/ PRDICT(101,5),P(101)
      COMMON/A3/ Y(100),DYCELL(100),JS1,JE1,JS2,JE2,JLFM,JL,YF,YH
     1           ,Z(100),DZCELL(100),KS1,KE1,KS2,KE2,KLFM,KL,ZF,ZH
      COMMON/A4/ ISHK,ILE,IE,IL,K1,K2,K3,K4,K5
      COMMON/A5/ GAMMA,GAMM1,GAMMPR,CV,CV1,STOKES,U0,C0,P0,RHO0,RL,X0
      COMMON/A7/ DX,DX1,DY,DY1,DZ, DZ1,EIWALL,IADBWL ,DT,CFL,CONST
      DOMAIN /EXPLCT/:I=1,100;J=1,100;K=1,100
      DTDX=DT*DX1
      DOALL J=JS1,JE2;K=KS1,KE2; USING /A11/,/A12/,/A13/,/A14/,/A2/,/A4/
      DO 3 I=1,IL
      PRDICT(I,1)=RHO (I,J,K)
      PRDICT(I,2)=RHOU(I,J,K)
      PRDICT(I,3)=RHOV(I,J,K)
      PRDICT(I,4)=RHOW(I,J,K)
      PRDICT(I,5)=E   (I,J,K)
      P(I)=GAMM1 *RHO(I,J,K)*EI(I,J,K)
    3 CONTINUE
      DO 4 N=1,2
      I=1
      IADD=N-1
      NM1=N-1
      B=1./N
      II=I+IADD
      UII=U(II,J,K)
      CALL FX(UII,I,J,K,II)
      DO 5 I=2,IE
      K3=K1
      K1=K2
      K2=K3
      II=I+IADD
```

Figure G.5   Explicit Aero - LX (Cont'd)

Original FORTRAN

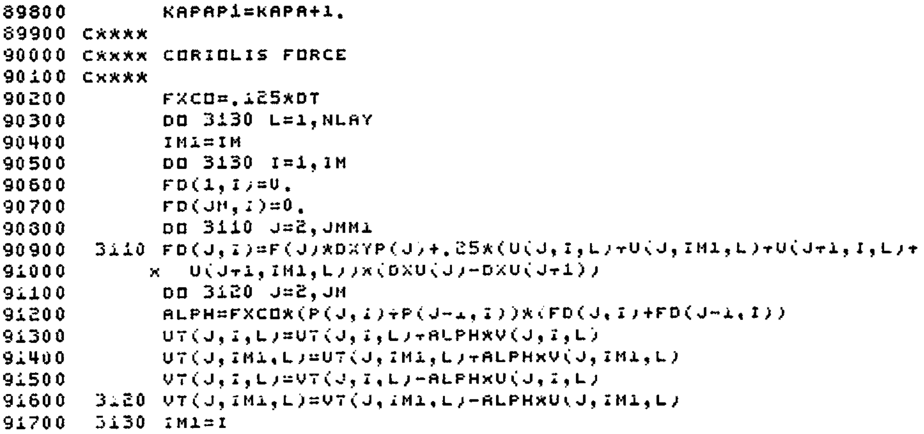
```
      UII=U(II,J,K)
      UI1=U(I+1,J,K)
      UI2=U(I,J,K)
      IF(UI1.GT.UI2.AND.(3.*UI1-UI2)*(3.*UI2-UI1).LT.0.) UII=.5*(UI1+UI2
     * )
      CALL FX(UII,I,J,K,II)
      PRDICT(I,1)=(NM1*PRDICT(I,1)+RHO (I,J,K)-DTDX*(F(K2,1)-F(K1,1)))*B
      PRDICT(I,2)=(NM1*PRDICT(I,2)+RHOU(I,J,K)-DTDX*(F(K2,2)-F(K1,2)))*B
      PRDICT(I,3)=(NM1*PRDICT(I,3)+RHOV(I,J,K)-DTDX*(F(K2,3)-F(K1,3)))*B
      PRDICT(I,4)=(NM1*PRDICT(I,4)+RHOW(I,J,K)-DTDX*(F(K2,4)-F(K1,4)))*B
      PRDICT(I,5)=(NM1*PRDICT(I,5)+E    (I,J,K)-DTDX*(F(K2,5)-F(K1,5)))*B
    5 CONTINUE
C---
C    * DECODE  *
      DO 6 I=2,IE
      RHOI=1./PRDICT(I,1)
      U (I,J,K)=PRDICT(I,2)*RHOI
      V (I,J,K)=PRDICT(I,3)*RHOI
      W (I,J,K)=PRDICT(I,4)*RHOI
      EI(I,J,K)=PRDICT(I,5)*  RHOI      -.5*(U(I,J,K)**2+V(I,J,K)**2+W(I
     * ,J,K)**2)
      P(I)      =GAMM1*PRDICT(I,1)*EI(I,J,K)
    6 CONTINUE
C***   *DOWNSTREAM B. C. AT I=IL
      DO 9 K6=1,5
    9 PRDICT(IL,K6)=PRDICT(IE,K6)
      CALL BCY(K,2,IE,J,J)
    4 CONTINUE
      DO 7 I=2,IL
      RHO (I,J,K)=PRDICT(I,1)
      RHOU(I,J,K)=PRDICT(I,2)
      RHOV(I,J,K)=PRDICT(I,3)
      RHOW(I,J,K)=PRDICT(I,4)
      E    (I,J,K)=PRDICT(I,5)
    7 CONTINUE
    2 CONTINUE
    1 CONTINUE
      CALL OUTER(JS1,JE2,KS1,KE2)
      RETURN
      END
```

Figure G.5 Explicit Aero - LX (Cont'd)

```
      UII=U(II,J,K)
      UI1=U(I+1,J,K)
      UI2=U(I,J,K)
      IF(UI1.GT.UI2.AND.(3.*UI1-UI2)*(3.*UI2-UI1).LT.0.) UII=.5*(UI1+UI2
     * )
      CALL FX(UII,I,J,K,II)
      PRDICT(I,1)=(NM1*PRDICT(I,1)+RHO (I,J,K)-DTDX*(F(K2,1)-F(K1,1)))*B
      PRDICT(I,2)=(NM1*PRDICT(I,2)+RHOU(I,J,K)-DTDX*(F(K2,2)-F(K1,2)))*B
      PRDICT(I,3)=(NM1*PRDICT(I,3)+RHOV(I,J,K)-DTDX*(F(K2,3)-F(K1,3)))*B
      PRDICT(I,4)=(NM1*PRDICT(I,4)+RHOW(I,J,K)-DTDX*(F(K2,4)-F(K1,4)))*B
      PRDICT(I,5)=(NM1*PRDICT(I,5)+E    (I,J,K)-DTDX*(F(K2,5)-F(K1,5)))*B
    5 CONTINUE
C---
C    * DECODE  *
      DO 6 I=2,IE
      RHOI=1./PRDICT(I,1)
      U (I,J,K)=PRDICT(I,2)*RHOI
      V (I,J,K)=PRDICT(I,3)*RHOI
      W (I,J,K)=PRDICT(I,4)*RHOI
      EI(I,J,K)=PRDICT(I,5)*  RHOI      -.5*(U(I,J,K)**2+V(I,J,K)**2+W(I
     * ,J,K)**2)
      P(I)      =GAMM1*PRDICT(I,1)*EI(I,J,K)
    6 CONTINUE
C***   *DOWNSTREAM B. C. AT I=IL
      DO 9 K6=1,5
    9 PRDICT(IL,K6)=PRDICT(IE,K6)
      CALL BCY(K,2,IE,J,J)
    4 CONTINUE
      DO 7 I=2,IL
      RHO (I,J,K)=PRDICT(I,1)
      RHOU(I,J,K)=PRDICT(I,2)
      RHOV(I,J,K)=PRDICT(I,3)
      RHOW(I,J,K)=PRDICT(I,4)
      E   (I,J,K)=PRDICT(I,5)
    7 CONTINUE
      ENDDO; GIVING/A11/,/A12/,/A13/,/A2/
      CALL OUTER(JS1,JE2,KS1,KE2)
      RETURN
      END
```

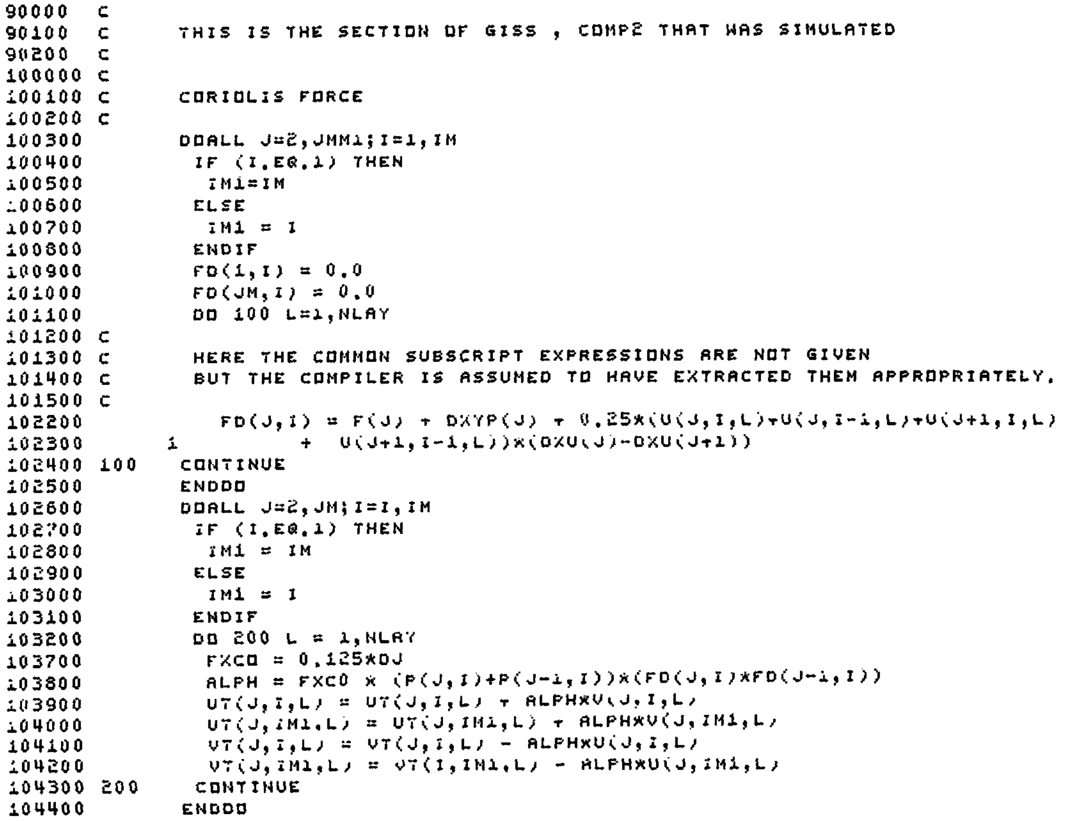
Figure G.5   Explicit Aero - LX (Cont'd)

Original FORTRAN



```
      KAPAP1=KAPA+1.
C****
C**** CORIOLIS FORCE
C****
      FXCO=.125*DT
      DO 3130 L=1,NLAY
      IM1=IM
      DO 3130 I=1,IM
      FD(1,I)=0.
      FD(JM,I)=0.
      DO 3110 J=2,JMM1
 3110 FD(J,I)=F(J)*DXYP(J)+.25*(U(J,I,L)+U(J,IM1,L)+U(J+1,I,L)+
     *  U(J+1,IM1,L))*(DXU(J)-DXU(J+1))
      DO 3120 J=2,JM
      ALPH=FXCO*(P(J,I)+P(J-1,I))*(FD(J,I)+FD(J-1,I))
      UT(J,I,L)=UT(J,I,L)+ALPH*V(J,I,L)
      UT(J,IM1,L)=UT(J,IM1,L)+ALPH*V(J,IM1,L)
      VT(J,I,L)=VT(J,I,L)-ALPH*U(J,I,L)
 3120 VT(J,IM1,L)=VT(J,IM1,L)-ALPH*U(J,IM1,L)
 3130 IM1=I
```

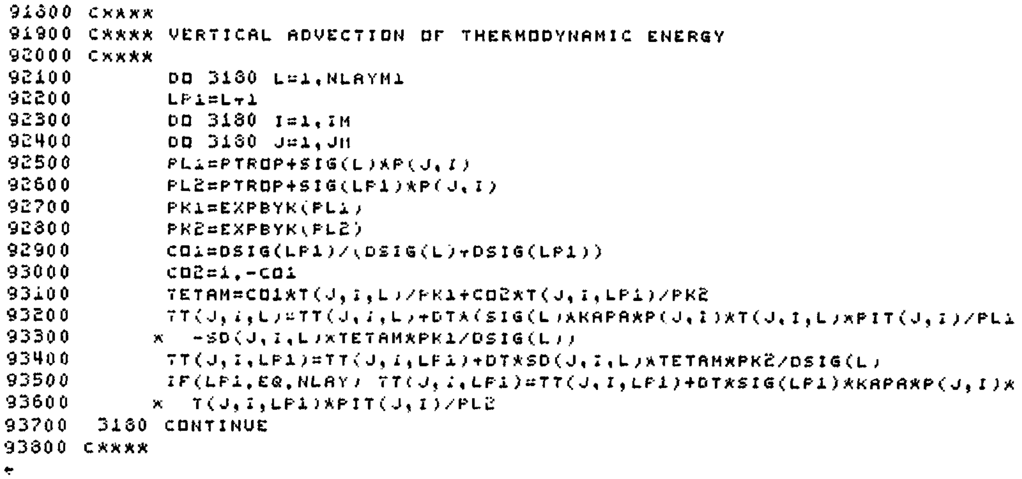
Figure G.6   GISS Weather - Section of COMP2



```
C
C      THIS IS THE SECTION OF GISS , COMP2 THAT WAS SIMULATED
C
C
C      CORIOLIS FORCE
C
       DOALL J=2,JMM1;I=1,IM
        IF (I.EQ.1) THEN
         IM1=IM
        ELSE
         IM1 = I
        ENDIF
        FD(1,I) = 0.0
        FD(JM,I) = 0.0
        DO 100 L=1,NLAY
C
        HERE THE COMMON SUBSCRIPT EXPRESSIONS ARE NOT GIVEN
        BUT THE COMPILER IS ASSUMED TO HAVE EXTRACTED THEM APPROPRIATELY.
C
          FD(J,I) = F(J) + DXYP(J) + 0.25*(U(J,I,L)+U(J,I-1,L)+U(J+1,I,L)
     1          +  U(J+1,I-1,L))*(DXU(J)-DXU(J+1))
100    CONTINUE
       ENDDO
       DOALL J=2,JM;I=I,IM
        IF (I.EQ.1) THEN
         IM1 = IM
        ELSE
         IM1 = I
        ENDIF
        DO 200 L = 1,NLAY
         FXCO = 0,125*DJ
         ALPH = FXC0 * (P(J,I)+P(J-1,I))*(FD(J,I)*FD(J-1,I))
         UT(J,I,L) = UT(J,I,L) + ALPH*V(J,I,L)
         UT(J,IM1,L) = UT(J,IM1,L) + ALPH*V(J,IM1,L)
         VT(J,I,L) = VT(J,I,L) - ALPH*U(J,I,L)
         VT(J,IM1,L) = VT(I,IM1,L) - ALPH*U(J,IM1,L)
200     CONTINUE
       ENDDO
```

Figure G.6   GISS Weather - Section of COMP2 (Cont'd)

Original FORTRAN



```
CXXXX
CXXXX VERTICAL ADVECTION OF THERMODYNAMIC ENERGY
CXXXX
      DO 3180 L=1,NLAYM1
      LP1=L+1
      DO 3180 I=1,IM
      DO 3180 J=1,JM
      PL1=PTROP+SIG(L)*P(J,I)
      PL2=PTROP+SIG(LP1)*P(J,I)
      PK1=EXPBYK(PL1)
      PK2=EXPBYK(PL2)
      CO1=DSIG(LP1)/(DSIG(L)+DSIG(LP1))
      CO2=1.-CO1
      TETAM=CO1*T(J,I,L)/PK1+CO2*T(J,I,LP1)/PK2
      TT(J,I,L)=TT(J,I,L)+DT*(SIG(L)*KAPA*P(J,I)*T(J,I,L)*PIT(J,I)/PL1
     *  -SD(J,I,L)*TETAM*PK1/DSIG(L))
      TT(J,I,LP1)=TT(J,I,LP1)+DT*SD(J,I,L)*TETAM*PK2/DSIG(L)
      IF(LP1.EQ.NLAY) TT(J,I,LP1)=TT(J,I,LP1)+DT*SIG(LP1)*KAPA*P(J,I)*
     *  T(J,I,LP1)*PIT(J,I)/PL2
 3180 CONTINUE
CXXXX
```

Figure G.6 GISS Weather - Section of COMP2 (Cont'd)

```
C
C     VERTICAL ADVECTION OF THERMODYNAMIC ENERGY
C
      DOALL J=1,JM;I=1,IM
       DO 300 L=1,NLAYM1
        LPA = P(J,I)
        LSIGA = SIG(L)
        LSIGB = SIG(L+1)
        PK1 = EXPBYK(PTROP + LSIGA*LPA)
        PK2 = EXPBYK(PTROP + LSIGB*LPA)
        LDSIGA = DSIG(L)
        LDSIGB = DSIG(L+1)
        LTA = T(J,I,L)
        LTB = T(J,I,L+1)
        LSDA = SD(J,I,L)
        LPITA = PIT(J,I)
        C01 = LDSIGB/(LDSIGA+LDSIGB)
        C02 = 1.0 - C01
        TETAM = C01*LTA/PK1 + C02*LTB/PK2
        LTTA = TT(J,I,L) + DT*(LSIGA*KAPA*LPA*LTA*LPITA/PL1-
     1         LSDA*TETAM*PK1/LDSIGA)
        LTTB =TT(J,K,L+1) + DT*LSDA*TETAM*PK2/LDSIGA
        IF (LP1.EQ.NLAY) LTTB=LTTB + DT*LSIGB*KAPA*LPA*LTB*
     1       LPITA/PL2
        TT(J,I,L) = LTTA
        TT(J,I,L+1) = LTTB
300    CONTINUE
      ENDDO
C
C   COMP2 CONTINUES BEYOND HERE, THIS IS THE END OF THE PIECE SIMULATED
C
+
```

Figure G.7 GISS Weather - Section of COMP2 (Cont'd)

```
      SUBROUTINE LINKHO
C*****
C*****INTERFACE
      COMMON/RADCOM/PL(9),PLE(10),PLK(9),TG,TS,TL(9),TSTR(3),SHL(9),
     *CLOUD(12),RE(10),RESTR(3),FLXDNG,SG,AS(9),ASSTR(3),S0,COSZ,RSURF
     *,SCOSZ,RAP,RAM
      COMMON/CLDCOM/SHALE(16),SWIL(16),AL(16),TAUL(16),DZALE(16),TOPABS
C*****
C*****GRID ARRAYS (STORAGE PROBLEM ON STAR)
      LOGICAL CLDFLG,AERFLG,L1,L2
      INTEGER ITY
      REAL A,AAA,SE,BE,UN1,TAU1,TAUT,AA,BB,CC,TNSQ,TN,
     * TAU,EDNCN,TDFCN,RDNCN,TAUCIR,PIU,EXTAU,TY,AER1,AER2,AERA,
     * AERU,AERV,AERC,EX1,EX2,DENO,DNN0,DNN1,PEFUP,PEFDN
      DIMENSION UNH2O(12),UNCO2(12),UNO3(12),F(12),E(12),NCLOUD(12),
     * TE(14),BTOP(14),FE(13),TAUN(12,3),FKGAS(3),FKGAS2(3),
     * EUP(12),EDN(12),EUPC(12),EDNC(12),TDF(12),TDFC(12),
     * REF(12),RDNC(12)
      EQUIVALENCE (FKGAS(1),EUP(1)),(FKGAS2(1),EUP(4))
      EQUIVALENCE (A,EDNCN),(AAA,TDFCN),(SE,RDNCN),
     * (BE,TAUCIR),(UN1,PIU),(TAU1,EXTAU),(TAUT,TY),
     * (AA,ITY),(AER1,BB),(AER2,CC),(TNSQ,AERA)
C*****
C*****SCALAR ARRAYS (TABLES OR USED FOR INITIALIZATION)
      DIMENSION CB(12,12),FI2(12,12),TA(12,12),FF1(12),FF2(12),
     * TEMP(23),TE3(301),DU(11),FIAERO(12,2),NAERO(12),
     * ACOSBR(12,2),AEREXT(12,2),ATAU55(2),PICIRO(12),
     * CIREXT(12),CCOSBR(12),COCLAM(12),COEK(3)
      DIMENSION SH2O(3,3),BH2O(3,3),WK(5,3),A1(12,3),A2(12,3),A3(12,3),
     *A4(12,3),B1(3,3,2),B2(3,3,2),B3(3,3,2),C1(2,3),C2(2,3),WK20(2,2)
      COMMON /EINT/ TE3
      DO 101 N=1,NLAYRS
C****
C**** SINGLE LAYER COMPUTATION
C**** EUP=UPWARDS EMISSION
C**** EDN=DOWNWARDS EMISSION
C**** TDF=TRANSMISSION
C**** REF=RE _ECTION
C****
      NCC=NCLOUD(N)
      NAER=NAERO(N)
      TAUCIR=NCLOUD(N)*(CIREXT(LAM)*CTAU55)
      TAUN(N,K)=TAUN(N,K)+TAUCIR
      X=TAUN(N,K)
```

Figure G.7   GISS Weather - LINKHO

```
      SUBROUTINE LINKHO
      COMMON /RADCOM/PL(9),PLE(10),PLK(9),TG,TS,TL(9),TSTR(3)
     1      ,SHL(9),CLOUD(12),RE(10),RESTR(3),FLXDNG,SG,AS(9),ASSTR(3),
     2       SC,COSZ,RSURF,SCOSZ,RAP,RAM
      COMMON /CLDCOM/ SHALE(16),SHIL(15),AL(16),TAUL(16),OZALE(16),
     1       TOPABS
      LOGICAL CLDFLG,AERFLG,L1,L2
      REAL TAUCIR,CTAU55,X,PIO,TN,AER1,AER2,AERA,AERC,AERW,AERV,
     1       EX1,EX2,DEN0,DNMU,DNMI,RERV,EXTAU,TAU,RDNCN,EDNCN,TDFCN,
     2       EUPCN,EDNCN
      INTEGER NCLOUD(12),NAERO(12)
      REAL CIREXT(12),TAUN(12,3),PICIR(12),FI2(12,12),C6(12,12),
     1    BTOP(14),TDF(12),REF(12),EUP(12),EDN(12),TE3(301),EUPC(12),
     2    EDNC(12),TDFC(12),RDNC(12)
C
C     ADDITIONAL DECLARATIONS NOT USED IN THE SIMULATED PORTION
C     ARE OMITTED FOR BREVITY
C
C     STATEMENTS ABOUT PARALLELISM ARE OMITTED ALSO SINCE LINKHO
C     IS CALLED AS A SUBROUTINE WITHIN THE INSTANCES OF THE
C     DOALL /LAYERS/ OF COMP3.  IN THIS CASE, EACH INSTANCE
C     CALLS LINKHO INDEPENDENT FROM ALL OTHER INSTANCES AND
C     USES A LOCAL COPY OF CODE WITHIN THE PROCESSOR IN WHICH
C     THE INSTANCE RESIDES.  SEQUENCING OF THE EXECUTION WITHIN
C     THIS SUBROUTINE IS SOLELY DEPENDENT ON THE INSTANCE AND
C     LOCAL DATA, NOT ON ANY OTHER INSTANCES.
C
      DO 200 LAM = 1,12
       DO 100 K = 1,3
        DO 101 N = 1,NLAYRS
         NCC = NCLOUD(N)
         NAER = NAERO(N)
         TAUCIR = CIREXT(LAM) * CTAU55 * NCC
         X = TAUN(N,K) + TAUCIR
         TAUN(N,K) = X
```

Figure G.7   GISS Weather - LINKHO (Cont'd)

```
      FIO=PIZERO(N,K)
C*********************************************************************
C IN CASE CODE DIDN'T GO THROUGH PROPERTIES CALCULATION
C SET FIO TO ZERO
C*********************************************************************
      IF(N.LE.3) TN=TSTR(N)/273.
      IF(N.GE.4) TN=TL(N-3)/273.
      IF(TN.GE..85348.AND.NCLOUD(N).GT.0)FI0=0.
      IF(FI0.GT.1.E-04) GO TO 160
      IF (NCC.GT.0) GO TO 102
C****UPWARD AND DOWNWARD FLUXEDN(N) OF SINGLE(EUP(N),EDN(N)) AND COMPOS
122   IF(X.LT.1.E-04) GO TO 103
      IF(X.GT.15.E0) GO TO 104
      X1=-TAUN(N,K)
      EXTAU=EXP(X1)
C*****CLEAR LAYER--FI0.LT.1.E-4
      TY=20.E0*X
      ITY=TY+1.E0
      TDF(N)=TE3(ITY)+(TY-ITY+1)*(TE3(ITY+1)-TE3(ITY))
      GO TO 105
104   CONTINUE
      EXTAU=0.
      TDF(N)=0.
105   REF(N)=0.
      DFB=(BTOP(N)-BTOP(N+1))*6.6667E-01
      FGRAD=DFB*((1.0-EXTAU)/X-TDF(N))
      ANS=1.0-TDF(N
      EDN(N)=BTOP(N+1)*ANS+FGRAD
      EUP(N)=BTOP(N)*ANS-FGRAD
      GO TO 109
103   TDF(N)=1.0
      REF(N)=0.
      EUP(N)=0.E0
      EDN(N)=0.E0
      GO TO 109
102   TDF(N)=0.0
      REF(N)=0.
      EUP(N)=BTOP(N)
      EDN(N)=BTOP(N+1)
      GO TO 109
```

Figure G.7 GISS Weather - LINKHO (Cont'd)

```
      PIO =(TAUCIR*PICIRO(LAM) + PIZ(LAM,N))/(X+1.E-40)
      IF(N.GE.4) THEN
       TN = TL(N-3)/273.
       ELSE
       TN = TSTR(N)/273.
      ENDIF
      IF (TN.GE.0.85348 .AND. NCC.GT.0)PIO=0.
      IF(PIO.GT.1.E-4) THEN
       AER1 = 1. - PIO
       AER2 = 1. - (PIO*CB(LAM,N)
       AERA = SQRT(AER1/AER2)
       AERU = (1. - AERA)/2.
       AERV = (1. + AERA)/2.
       AERC = SQRT(3.*AER1*AER2)
       X1 = -(AERC*X)
       EX1 = 0.0
       IF (X1 .GE. -180.218) EX1 = EXP(X1)
       IF (EX1.LT.1.0E-30)  EX1=0.0
       EX2 = EX1*EX1
       DEN0 = 1./((AERV*AERV) - (AERU*AERU*EX2))
       DNM0 = ((BTOP(N) - BTOP(N+1)/(X*AERC))*
   1              ((AERV - AERU*EX2) - (AERA*EX1))
       DNM1 = AERV + AERU*EX2
       EUP(N) = (BTOP(N)*DNM1 - DNM0 - BTOP(N+1)*EX1)*
   1              DEN0*AERA
       EDN(N) = (BTOP(N+1)*DNM1 + DNM0 - BTOP(N)*EX1)*
   1              DEN0*AERA
       REF(N) = AERU*AERV*(1.-EX2)*DEN0
       TDF(N) = (AERV-AERU)*DEN0*EX1
```

Figure G.7   GISS Weather - LINKHO (Cont'd)

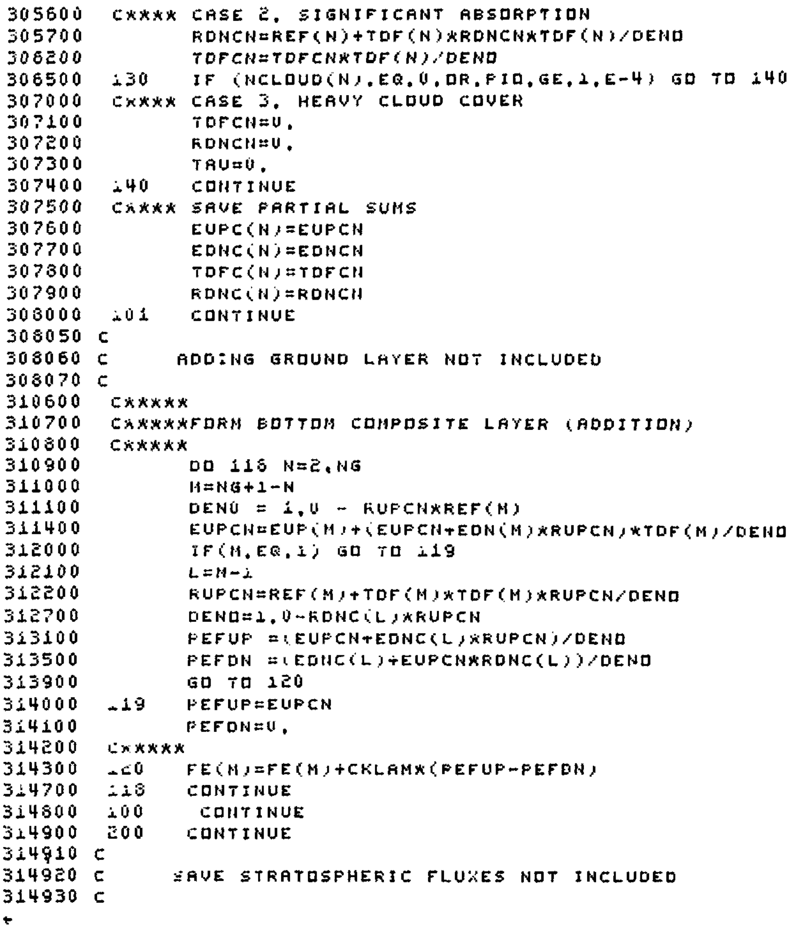
Original FORTRAN

```
C****EMISSION CALCULATIONS FOR HAZE LAYER,EXACT IN THE SEUP(N)SE OF ISO
C****IC SCATTERING
C****EXACT SOLUTION=TWO-STREAM SOLUTION*FORGE FACTOR(PI0,TAU0)
160    AERA = SQRT((1.-PI0)/(1.-PI0*CB(LAM,N)))
       AERU = (1-AERA)/2.0
       AERV = (1+AERA)/2.0
       AERC = SQRT(AER2*3.0*AER1)
       EX1 = EXP(-AERC*TAUN(N,K))
       IF (EX1 .LT. 1.E-30) EX1=0.0
       EX2=EX1*EX1
C*****FORGE FACTOR FOR ISOTROPIC SCATTERING
       FTWO=1.E0
C**** PI02=PI0*PI0
C*** FTWO=1.0+0.14*EXTAU+0.1*PI02*(1.0-EXTAU)+(-1.03+0.4019*PI0+0.6631*
C***1PI02)**EXTAU+(2.0172-0.6804*PI0-1.3597*PI02)*****EXTAU*EXTAU
       DEN0 = (AERV**2 - AERU**2)*EX2
       DNM0 = ((BTOP(N)-BTOP(N+1))/TAUN(N,K))/AERC*
      1     (AERV-AERU*EX2-AERA*EX1)
       DNM1 = AERV + AERU*EX2
       EUP(N)=(BTOP(N)*DNM1-DNM0-BTOP(N+1)*EX1)/DEN0*FTWO*AERA
       EDN(N)=(BTOP(N+1)*DNM1+DNM0-BTOP(N)*EX1)/DEN0*FTWO*AERA
C*****REF(N),TDF(N) BASED ON TWO STREAM SOLUTION
       REF(N)=AERU*AERV*(1.0-EX2)/DEN0
       TDF(N)=(AERV-AERU)/DEN0*EX1
C****
C**** FORM TOP COMPOSITE LAYER (ADDITION)
C****
109    DEN0=1.0-RDNCN*REF(N)
       EUPCN=EUPCN+(EUP(N)+EDNCN*REF(N))*TDFC(N)/DEN0
       EDNCN=EDN(N)+(EDNCN+EUP(N)*RDNCN)*TDF(N)/DEN0
       IF(NCLOUD(N).GT.0) CLDFLG=.TRUE.
C**** SET AEROSOL FLAG IF CIRRUS CLOUDS (HIGH ALBEDO)
       IF(CLDFLG.AND.PI0.GE.1.E-4) AERFLG=.TRUE.
C**** TRANSMISSION COMPUTED DIFFERENTLY FOR 3 CASES
       IF (CLDFLG.OR.AERFLG) GO TO 125
C**** CASE 1. ATMOSPHERE HAS NO AEROSOLS OR CLOUDS THRU HERE
C**** USE EXPONENTIAL INTEGRAL APPROXIMAION
       TAU=TAU+TAUN(N,K)
C**** PROTECT AGAINST TABLE OVERFLOW
       IF(TAU.GT.15.) GO TO 124
       TY=20.*TAU
       ITY=TY+1.
         IF(ITY.LT.1) ITY=1
C      TDFCN=TE3(ITY)+(TY-ITY+1)*(TE3(ITY+1)-TE3(ITY))
       GO TO 125
124    TDFCN=0.
125    IF(.NOT.AERFLG) GO TO 130
```

Figure G.7 GISS Weather - LINKHO (Cont'd)

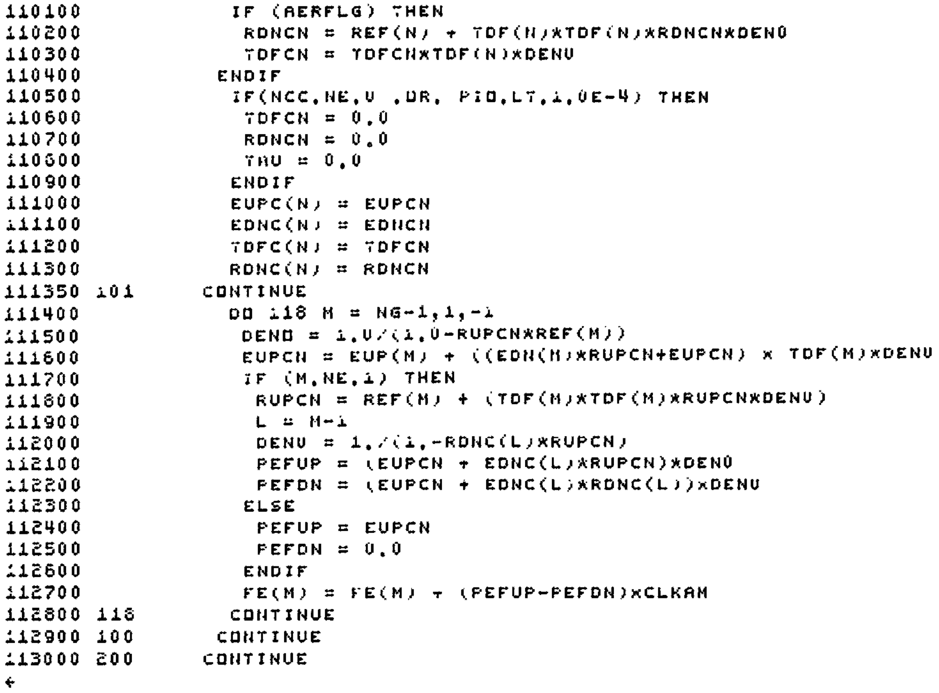
```
      ELSE IF (NCC.GT.0) THEN
       TDF(N) = 0.0
       REF(N) = 0.0
       EUP(N) = BTOP(N)
       EDN(N) = BTOP(N+1)
      ELSE IF ( X.LT.1.E-4) THEN
       TDF(N)=1.0
       REF(N) = 0.0
       EUP(N) = 0.0
       EDN(N) = 0.0
      ELSE
       IF (X .LE. 15.0) THEN
        EXTAU = EXP(-X)
        ITY =  X*20. + 1.
        TDF(N) = TE3(ITY) + (TY-ITY+1) * (TE3(ITY+1)-TE3(ITY))
       ELSE
        EXTAU = 0.0
        TDF(N) = 0.0
       ENDIF
       REF(N) = 0.0
       X1 = 1.0 - TDF(N)
       X2 = ((1.0 - EXTAU)/X-TDF(N)) * ((BTOP(N) - BTOP(N+1))*
     1          0.6666)
       EDN(N) = BTOP(N+1)*X1+X2
       EUP(N) = BTOP(N)*X1-X2
      ENDIF
      DEN0 = 1.0/(1.0 - RDNCN*REF(N))
      EDNCN = (EDNCN+EUP(N)*RDNCN) * TDF(N) * DEN0 + EDN(N)
      IF (NCC.GT.0) CLDFLG = .TRUE.
      IF(CLDFLG.AND.PID.GE.1.0E-4) AERFLG=.TRUE.
      IF (.NOT.(CLDFLG.OR.AERFLG)) THEN
        TAU = TAU + X
        IF (TAU .GT. 15) THEN
         TDFCN = 0.
        ELSE IF ((20.*TAU+1.).LT.1) THEN
          ITY=1
          TDFCN = TE3(ITY)+(TY-ITY+1)*(TE3(ITY+1)-TE3(ITY))
         ENDIF
        ENDIF
```

Figure G.7   GISS Weather - LINKHO (Cont'd)



```
 C**** CASE 2. SIGNIFICANT ABSORPTION
       RDNCN=REF(N)+TDF(N)*RDNCN*TDF(N)/DENO
       TDFCN=TDFCN*TDF(N)/DENO
 130   IF (NCLOUD(N).EQ.0.OR.PIO.GE.1.E-4) GO TO 140
 C**** CASE 3. HEAVY CLOUD COVER
       TDFCN=0.
       RDNCN=0.
       TAU=0.
 140   CONTINUE
 C**** SAVE PARTIAL SUMS
       EUPC(N)=EUPCN
       EDNC(N)=EDNCN
       TDFC(N)=TDFCN
       RDNC(N)=RDNCN
 101   CONTINUE
C
C     ADDING GROUND LAYER NOT INCLUDED
C
 C*****
 C*****FORM BOTTOM COMPOSITE LAYER (ADDITION)
 C*****
       DO 116 N=2,NG
       M=NG+1-N
       DENO = 1.0 - RUPCN*REF(M)
       EUPCN=EUP(M)+(EUPCN+EDN(M)*RUPCN)*TDF(M)/DENO
       IF(M.EQ.1) GO TO 119
       L=M-1
       RUPCN=REF(M)+TDF(M)*TDF(M)*RUPCN/DENO
       DENO=1.0-RDNC(L)*RUPCN
       PEFUP =(EUPCN+EDNC(L)*RUPCN)/DENO
       PEFDN =(EDNC(L)+EUPCN*RDNC(L))/DENO
       GO TO 120
 119   PEFUP=EUPCN
       PEFDN=0.
 C*****
 120   FE(M)=FE(M)+CKLAM*(PEFUP-PEFDN)
 118   CONTINUE
 100    CONTINUE
 200   CONTINUE
C
C     SAVE STRATOSPHERIC FLUXES NOT INCLUDED
C
```

Figure G.7 GISS Weather - LINKHO (Cont'd)



```
             IF (AERFLG) THEN
              RDNCN = REF(N) + TDF(N)*TDF(N)*RDNCN*DEN0
              TDFCN = TDFCN*TDF(N)*DENU
            ENDIF
             IF(NCC.NE.0 .OR. PI0.LT.1.0E-4) THEN
              TDFCN = 0.0
              RDNCN = 0.0
              TAU = 0.0
             ENDIF
             EUPC(N) = EUPCN
             EDNC(N) = EDNCN
             TDFC(N) = TDFCN
             RDNC(N) = RDNCN
    101    CONTINUE
             DO 118 M = NG-1,1,-1
              DEN0 = 1.0/(1.0-RUPCN*REF(M))
              EUPCN = EUP(M) + ((EDN(M)*RUPCN+EUPCN) * TDF(M)*DENU
              IF (M.NE.1) THEN
               RUPCN = REF(M) + (TDF(M)*TDF(M)*RUPCN*DENU)
               L = M-1
               DENU = 1./(1.-RDNC(L)*RUPCN)
               PEFUP = (EUPCN + EDNC(L)*RUPCN)*DEN0
               PEFDN = (EUPCN + EDNC(L)*RDNC(L))*DENU
              ELSE
               PEFUP = EUPCN
               PEFDN = 0.0
              ENDIF
              FE(M) = FE(M) + (PEFUP-PEFDN)*CLKAM
    118      CONTINUE
    100     CONTINUE
    200    CONTINUE
←
```

Figure G.7   GISS Weather - LINKHO (Cont'd)

APPENDIX H
CONNECTION NETWORK SIMULATION TOOLS

H.1 SUMMARY

Two computer-based tools were developed as an aid to the study of the various Connection Networks. The first was a functional simulator. This functional simulator supported evaluation of the Benes, the single-layer Omega, and the simple double-layer Omega networks described in Appendix B. The networks could be exercised in a number of modes including random inputs and p-ordered inputs. Section H.2 below discusses these capabilities in more detail.

The second tool was a stochastic analyzer (see Section H.3). This tool used the probability of addresses occurring and the probability of requests occurring within the network to predict blockage within the network. Although this approach precluded actually observing where specific blockages would occur for a particular input situation, the tool was felt to be necessary because it would be unreasonable to run all possible input combinations. The stochastic analyzer was used to evaluate both the single-layer Omega network and the double-layer network which included inter-layer connectivities.

It was noted in Appendix B that both tools gave comparable results when run on the same cases. This correspondence gave confidence in the results obtained using these tools.

H.2 CONNECTION NETWORK FUNCTIONAL SIMULATOR

H.2.1 Model

The CN simulator is designed to simulate a CN in which requests propagate through at the speed of transmission delay in cable and combinatorial logic, after which the path is locked up for the duration of the EM cycle. After an EM cycle, the nodes involved in this path may be unlocked if they are not involved in another EM request.

In addition to nets with the connectivity of Benes networks and Omega networks (see Appendix B), there are options on the amount of redundant paths supplied. There can be twice as many ports on the processor side as there are processors, or there can be just 512. The EM module ports can be spread across the entire 1024 ports on that side, or they can occupy the first 521. The simulator basically has a 1024-wide network of 2 x 2 switches.

The number of CN-clock cycles per EM cycle time can be adjusted from 1 to 9 by an input parameter.

Each simulated processor has a queue of up to six memory requests. The Nth entry in this queue may be either a set of "S" random EM module numbers, with 512-S of the processors having null requests, or the entry may be a p-ordered or a p-q-ordered vector of EM module numbers, with 512-S of the processors having their requests nulled before the program starts. S is an input parameter ranging from 0 to 300, or equal to 512.

The four-digit seed of the random number generator is included in the set of input parameters.

H.2.2 Simulator Controls

The input commands accepted by the functional simulator are listed in Table H.1 below. Some of these inputs are optional and have default values as indicated.

Table H.1
CN Functional Simulator Input Commands

Command	Description
Fn	Type of Network where n is the sum of 0: if a 19-level Benes Network 1: if a 10-level single-layer Omega network 2: if a 10-level double-layer Omega network with alternating priorities 4: if processor M is attached to input port 2M 8: if EM module N is attached to output port 2N up to 511 with the other 9 attached output ports 1, 3 5, ... 17. (If no F command, F0 used as default)
An	Algorithm to be used within each 2 x 2 node where n is: 0: the node gives priority, in case of conflict, to the lower-numbered ("upper") input on all one-sheet (single layer) cases, and to give priority to the higher-numbered ("lower") input for the second sheet in a double-layer Omega network. 1: the node sets a straight-through connection in the case of conflict. 2: the node alternates the priorities between upper and lower input ports on alternate CN clocks. If this mode is chosen, it is recommended that the number of CN clocks/EM clock be odd (see Tn command below).

Table H.1 (Continued)

	(If no A command, A0 is used as default.)
Snnn	Command which causes all but "nnn" of the 512 entries in each queue position in the processor to be erased. The choice of which entries to erase is random. (If no S command, S512 is used as default. This corresponds to no erasures.)
Tn	Command which sets "n" cycles of the CN clock for each EM access time. (If no T command T0 is used as default.)
BR	An optional command that signals the "bit-reversal" of processor number to TN porc number. That is, if BR, then proc. 1 goes to port 256, proc. 2 goes to port 128, proc. 3 goes to port 384. That is, proc. 00000011 goes to port 11000000. Processor no. 00010111 goes to port no. 11101000, etc.
nnnn	A four-digit number sets the seed for the random number generator.
Pnnnmmm	Sets a p-ordered vector into the next entry across all processor queues. The entry has an offset of "nnn" and a skip distance of "mmm".
Qaaasssskkkxxxqqq	Sets a p-q-ordered vector into the next entry across all processor queues. "aaa" is the offset to the start of the first vector piece, sss is the skip distance within pieces, kkk is the length of each piece, xxx is the number of elements omitted if the first piece is shorter than kkk, qqq is the skip between the end of one piece and the beginning of the next.
R	Sets a vector of 512 random requests into the next entry in all the processor queues. (The seed for the random number generator should precede this command.)
Lnnn	This command imposes a limit on the number of CN cycles through which the simulation will run. Termination will be after "nnn+1" CN cycles. (If no L command, L047 is used as default.)

Warning: Although the input is free/form, in that the sequence of commands does not matter and any number of intervening blanks are allowed, each number must follow its command without any intervening blank, and must consist of exactly the correct number of digits.

Following all commands, any character (such as "X") that is not ?, E, N, D, or the first character of any valid command, will terminate the input. The rest of the card can then be used for comment which will be printed out on the first line of output.

H.2.3 Simulator Output

Figures H.1, H.2, and H.3 are examples of three, typical CN Functional Simulator outputs. These examples happen to use p-q-ordered vectors as inputs with piece lengths of 31, 100, and 30. The cases were taken from the explicit and implicit aero flow code. Two of these cases are in mesh sizes as exhibited in the listings supplied by NASA, and one of the cases exhibits the full size.

The first line, of the printout which begins with "?END" prints the input commands as previously described. For example, Figure H.1 shows (on the first line):

T2 (2 CN Clocks per EM access time)
BR (Bit reversal of processor number to CN port number)
F14 (Double-Layered Omega Network w. alternating priorities (2) + processor M attached to port 2M (4) + EM module N attached to port 2N.(8))

Q047 1 31 1409 (p-q-ordered vector with
047 offset
__1 skip distance within pieces
_31 length of each piece
__1 number of elements omitted if the first piece is shorter than 31
409 skip between the end of one piece and the beginning of the next.

The next several lines of the output summarize the simulation conditions specified by the input commands.

The remaining output, summarized below, is printed at each network layer at each CN clock.

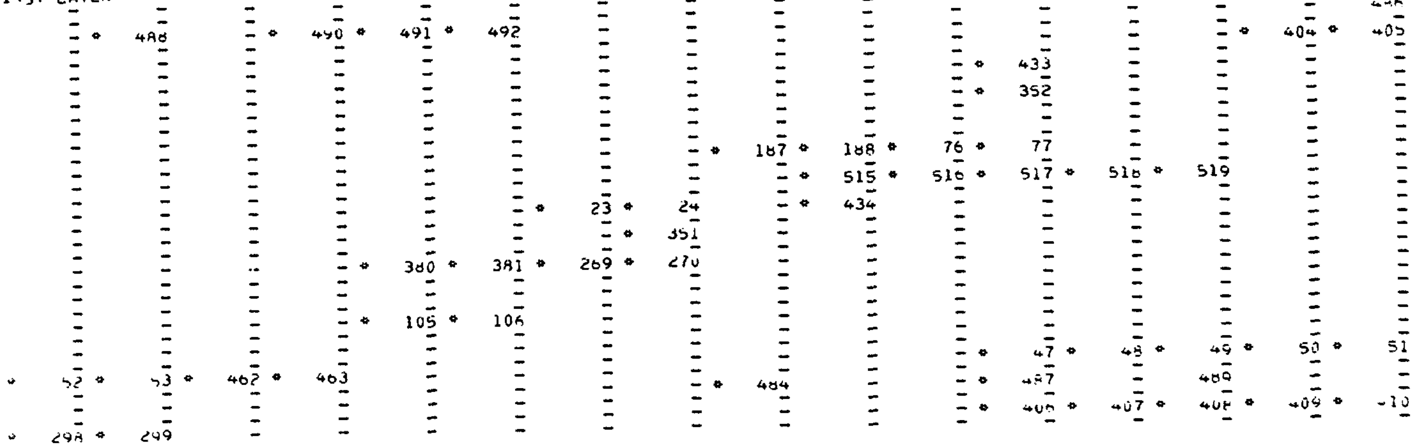
1st line: Number of items left in processor queue before beginning. Does not include items picked up by bumping the processor queue pointers.

(Text continued on page H-13)

```
? END T2 BR F14 Q047  1 31  1409 XX ACTUAL CASE FROM EXPLICIT          000.000000000000000000000000000000000000000000000000
OFFSET= 47 SKIP=  1   31 ELEMENTS PER PIECE; FIRST PIECE=  30, 409SPACE BETWEEN PIECES
TN TEST.  EM/TN CLOCK RATI.  2 NREQ512 ZMAX +7 PRIORITY  0    FRIDAY  02/16/79   16:05:53
PORTS: PROC UNPACKED; EM UNPACKED. NET NO.  2
NO OF ACCESS CYCLES REQUESTED IS  1

     512 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
  52X 0  426X 1   43X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  1TH CYCLE
THERE WERE 412 SUCCESSES I. 512 REQUESTS.

FIRST LAYER
 *    47 *    48 *    49 *    50 *    51 *    52 *    53 *    54 *    55 *    56 *    57 *    58 *    59 *    60 *    61 *    62
 *    63 *    64 *    65 *    66 *    67 *    68 *    69 *    70 *    71 *    72 *    73 *    74 *    75 *    76 *    77      486
 *   487     488 *   489     490     491     492     493     494     495 *   496 *   497 *   498 *   499 *   500 *   501 *   502
 *   503 *   504 *   505 *   506 *   507 *   508 *   509 *   510 *   511     512     513     514     515     516     404     405
 *   406 *   407 *   408 *   409 *   410 *   411 *   412 *   413 *   414 *   415 *   416 *   417 *   418 *   419 *   420 *   421
 *   422 *   423 *   424 *   425 *   426 *   427 *   428 *   429 *   430 *   431 *   432     433 *   434 *   322 *   323 *   324
 *   325 *   326 *   327 *   328 *   329 *   330 *   331 *   332 *   333 *   334 *   335 *   336 *   337 *   338 *   339 *   340
 *   341 *   342 *   343 *   344 *   345 *   346 *   347     348 *   349 *   350 *   351     352 *   240 *   241 *   242 *   243
 *   244 *   245 *   246 *   247 *   248 *   249 *   250 *   251 *   252 *   253 *   254 *   255 *   256 *   257 *   258 *   259
 *   260 *   261 *   262 *   263 *   264 *   265 *   266 *   267 *   268 *   269 *   270 *   158 *   159 *   160 *   161 *   162
 *   163 *   164 *   165 *   166 *   167 *   168 *   169 *   170 *   171 *   172 *   173 *   174 *   175 *   176 *   177 *   178
 *   179 *   180 *   181 *   182 *   183     184     185     186     187     188      76      77      78      79 *    80 *    81
 *    82 *    83 *    84 *    85 *    86 *    87 *    88 *    89 *    90 *    91 *    92 *    93 *    94 *    95 *    96 *    97
 *    98 *    99 *   100 *   101 *   102 *   103 *   104 *   105 *   106     515     516     517     518     519     520 *     0
 *     1 *     2 *     3 *     4 *     5       6 *     7 *     8 *     9 *    10 *    11 *    12 *    13 *    14 *    15 *    16
 *    17 *    18 *    19      20      21      22      23      24 *   433     434     435     436     437 *   438 *   439 *   440
 *   441 *   442 *   443 *   444 *   445 *   446 *   447 *   448 *   449 *   450 *   451 *   452 *   453 *   454 *   455 *   456
 *   457 *   458 *   459 *   460 *   461 *   462 *   463     351 *   352 *   353 *   354 *   355 *   356 *   357 *   358 *   359
 *   360 *   361 *   362 *   363 *   364 *   365 *   366 *   367 *   368 *   369 *   370 *   371 *   372 *   373 *   374 *   375
 *   376     377     378     379     380     381     269     270     271 *   272 *   273 *   274 *   275 *   276 *   277 *   278
 *   279 *   280 *   281 *   282 *   283 *   284 *   285 *   286 *   287 *   288 *   289 *   290 *   291 *   292 *   293 *   294
 *   295 *   296 *   297 *   298 *   299 *   187 *   188 *   189 *   190 *   191 *   192 *   193 *   194 *   195 *   196 *   197
 *   198 *   199 *   200 *   201 *   202 *   203 *   204 *   205 *   206 *   207 *   208 *   209 *   210 *   211 *   212 *   213
 *   214 *   215 *   216 *   217     105     106     107     108     109 *   110 *   111 *   112 *   113 *   114 *   115 *   116
 *   117 *   118 *   119 *   120 *   121 *   122 *   123 *   124 *   125 *   126 *   127     128 *   129 *   130 *   131 *   132
 *   133 *   134 *   135 *    23 *    24 *    25 *    26 *    27 *    28 *    29 *    30 *    31 *    32 *    33 *    34 *    35
      36 *    37 *    38 *    39 *    40 *    41 *    42 *    43      44 *    45 *    46      47      48      49      50      51
      52      53     462     463     464     465     466     467     468 *   469 *   470 *   471 *   472 *   473 *   474 *   475
 *   476 *   477 *   478 *   479 *   480 *   481 *   482 *   483     484 *   485 *   486     487 *   488     489 *   490     491
 *   492 *   380 *   381 *   382 *   383     384     385     386     387     388     389     390     391 *   392 *   393 *   394
 *   395 *   396 *   397 *   398 *   399     400     401     402     403     404 *   405     406     407     408     409     410
     298     299     300     301     302 *   303 *   304 *   305 *   306 *   307 *   308 *   309 *   310 *   311 *   312 *   313

     100 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 425X 0   92X 1    4X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  1TH CYCLE
THERE WERE 465 SUCCESSES I. 512 REQUESTS.

BOTH LAYERS
 *    47 *    48 *    49 *    50 *    51 *    52 *    53 *    54 *    55 *    56 *    57 *    58 *    59 *    60 *    61 *    62
 *    63 *    64 *    65 *    66 *    67 *    68 *    69 *    70 *    71 *    72 *    73 *    74 *    75 *    76 *    77      486
 *   487     488 *   489     490     491     492 *   493 *   494 *   495 *   496 *   497 *   498 *   499 *   500 *   501 *   502
 *   503 *   504 *   505 *   506 *   507 *   508 *   509 *   510 *   511 *   512 *   513 *   514 *   515 *   516     404     405
 *   406 *   407 *   408 *   409 *   410 *   411 *   412 *   413 *   414 *   415 *   416 *   417 *   418 *   419 *   420 *   421
 *   422 *   423 *   424 *   425 *   426 *   427 *   428 *   429 *   430 *   431 *   432     433 *   434 *   322 *   323 *   324
 *   325 *   326 *   327 *   328 *   329 *   330 *   331 *   332 *   333 *   334 *   335 *   336 *   337 *   338 *   339 *   340
 *   341 *   342 *   343 *   344 *   345 *   346 *   347 *   348 *   349 *   350 *   351     352 *   240 *   241 *   242 *   243
 *   244 *   245 *   246 *   247 *   248 *   249 *   250 *   251 *   252 *   253 *   254 *   255 *   256 *   257 *   258 *   259
 *   260 *   261 *   262 *   263 *   264 *   265 *   266 *   267 *   268 *   269 *   270 *   158 *   159 *   160 *   161 *   162
 *   163 *   164 *   165 *   166 *   167 *   168 *   169 *   170 *   171 *   172 *   173 *   174 *   175 *   176 *   177 *   178
 *   179 *   180 *   181 *   182 *   183 *   184 *   185 *   186     187     188      76      77 *    78 *    79 *    80 *    81
 *    82 *    83 *    84 *    85 *    86 *    87 *    88 *    89 *    90 *    91 *    92 *    93 *    94 *    95 *    96 *    97
 *    98 *    99 *   100 *   101 *   102 *   103 *   104 *   105 *   106     515     516     517     518     519 *   520 *     0
 *     1 *     2 *     3 *     4 *     5 *     6 *     7 *     8 *     9 *    10 *    11 *    12 *    13 *    14 *    15 *    16
 *    17 *    18 *    19 *    20 *    21 *    22      23      24 *   433     434 *   435 *   436 *   437 *   438 *   439 *   440
 *   441 *   442 *   443 *   444 *   445 *   446 *   447 *   448 *   449 *   450 *   451 *   452 *   453 *   454 *   455 *   456
 *   457 *   458 *   459 *   460 *   461 *   462 *   463     351 *   352 *   353 *   354 *   355 *   356 *   357 *   358 *   359
 *   360 *   361 *   362 *   363 *   364 *   365 *   366 *   367 *   368 *   369 *   370 *   371 *   372 *   373 *   374 *   375
 *   376 *   377 *   378 *   379     380     381     269 *   270 *   271 *   272 *   273 *   274 *   275 *   276 *   277 *   278
 *   279 *   280 *   281 *   282 *   283 *   284 *   285 *   286 *   287 *   288 *   289 *   290 *   291 *   292 *   293 *   294
 *   295 *   296 *   297 *   298 *   299 *   187 *   188 *   189 *   190 *   191 *   192 *   193 *   194 *   195 *   196 *   197
 *   198 *   199 *   200 *   201 *   202 *   203 *   204 *   205 *   206 *   207 *   208 *   209 *   210 *   211 *   212 *   213
 *   214 *   215 *   216 *   217     105     106 *   107 *   108 *   109 *   110 *   111 *   112 *   113 *   114 *   115 *   116
 *   117 *   118 *   119 *   120 *   121 *   122 *   123     124 *   125 *   126 *   127 *   128 *   129 *   130 *   131 *   132
```

Figure H.1 CN Simulator Output, First Example

```
 *   133 *   134 *   135 *    23 *    24 *    25 *    26 *    27 *    28 *    29 *    30 *    31 *    32 *    33 *    34 *    35
 *    36 *    37 *    38 *    39 *    40 *    41 *    42 *    43 *    44 *    45 *    46      47      48      49      50      51
      52      53     462     463 *   464 *   465 *   466 *   467 *   468 *   469 *   470 *   471 *   472 *   473 *   474 *   475
 *   476 *   477 *   478 *   479 *   480 *   481 *   482 *   483     484 *   485 *   486     487 *   488     489 *   490 *   491
 *   492 *   380 *   381 *   382 *   383 *   384 *   385 *   386 *   387 *   388 *   389 *   390 *   391 *   392 *   393 *   394
 *   395 *   396 *   397 *   398 *   399 *   400 *   401 *   402 *   403 *   404 *   405     406     407     408     409     410
     298     299 *   300 *   301 *   302 *   303 *   304 *   305 *   306 *   307 *   308 *   309 *   310 *   311 *   312 *   313

        47 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 474X 0   47X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  2TH CYCLE
THERE WERE   0 SUCCESSES IN  47 REQUESTS.

FIRST LAYER

        47 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 474X 0   47X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  2TH CYCLE
THERE WERE   0 SUCCESSES IN  47 REQUESTS.

BOTH LAYERS

        47 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 474X 0   47X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  3TH CYCLE
THERE WERE  45 SUCCESSES IN  47 REQUESTS.

FIRST LAYER
```

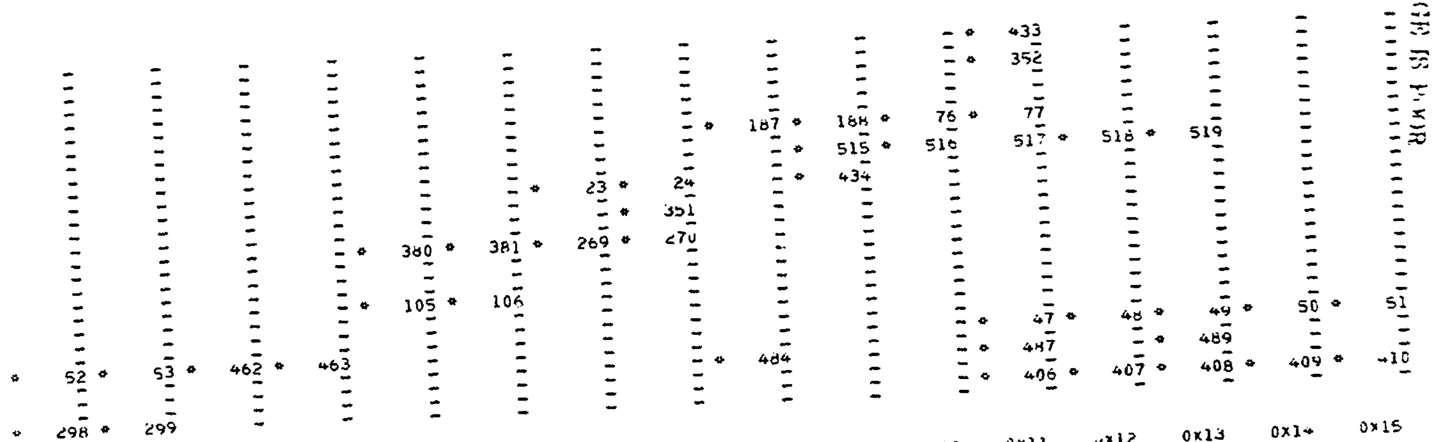


```
        2 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 519X 0    2X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  3TH CYCLE
THERE WERE  47 SUCCESSES IN  47 REQUESTS.

BOTH LAYERS
```

488
490
491
492
404
405
446

Figure H.1 CN Simulator Output, First Example (Cont'd)

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR



```
                0 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
521X 0     0X 1     0X 2     0X 3     0X 4     0X 5     0X 6     0X 7     0X 8     0X 9     0X10     0X11     0X12     0X13     0X14     0X15     0
                0 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
521X 0     0X 1     0X 2     0X 3     0X 4     0X 5     0X 6     0X 7     0X 8     0X 9     0X10     0X11     0X12     0X13     0X14     0X15     0
```

Figure H.1 CN Simulator Output, First Example (Cont'd)

```
? END BR F14 0 47  1100 37211 XX ACTUAL CASE FROM 100,50,200 3D HARD DIREC.     00000000000000000000000000000000000000000000000000000
OFFSET= 47 SKIP=  1  100 ELEMENTS PER PIECE; FIRST PIECE=  63, 211SPACE BETWEEN PIECES
TN TEST.  EM/TN CLOCK RATIO  1 NREQ512 ZMAX 47 PRIORITY  0    FRIDAY  02/16/79   16:11:50
PORTS: PROC UNPACKED; EM UNPACKED. NET NO.  2
NO OF ACCESS CYCLES REQUESTED IS  1

     512 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
  13X 0  504X 1    4X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  1TH CYCLE
THERE WERE 427 SUCCESSES I. 512 REQUESTS.

FIRST LAYER
*    47 *    48 *    49 *    50 *    51 *    52 *    53 *    54 *    55 *    56 *    57 *    58 *    59 *    60 *    61 *    62
*    63 *    64 *    65 *    66 *    67 *    68 *    69 *    70 *    71 *    72 *    73 *    74 *    75 *    76 *    77 *    78
*    79 *    80 *    81 *    82 *    83 *    84 *    85 *    86 *    87 *    88 *    89 *    90 *    91 *    92 *    93 *    94
*    95 *    96 *    97 *    98 *    99 *   100 *   101 *   102 *   103 *   104 *   105 *   106 *   107 *   108 *   109 *   110
*   321 *   322 *   323 *   324 *   325 *   326 *   327 *   328 *   329 *   330 *   331 *   332 *   333 *   334 *   335 *   336
*   337 *   338 *   339 *   340 *   341 *   342 *   343 *   344 *   345 *   346 *   347 *   348 *   349 *   350 *   351 *   352
*   353 *   354 *   355 *   356 *   357 *   358 *   359 *   360 *   361 *   362 *   363 *   364 *   365 *   366 *   367 *   368
*   369 *   370 *   371 *   372 *   373 *   374 *   375 *   376 *   377 *   378 *   379 *   380 *   381 *   382 *   383 *   384
*   385 *   386 *   387 *   388 *   389 *   390 *   391 *   392     393     394     395     396     397     398     399     400
    401     402     403     404     405     406     407     408     409     410     411     412     413     414     415     416
    417     418     419     420     110     111     112     113     114     115     116     117     118     119     120     121
    122     123     124     125     126     127 *   128 *   129 *   130 *   131 *   132 *   133 *   134 *   135 *   136 *   137
*   138 *   139 *   140 *   141 *   142 *   143 *   144 *   145 *   146 *   147 *   148 *   149 *   150 *   151 *   152 *   153
*   154 *   155 *   156 *   157 *   158 *   159 *   160 *   161 *   162 *   163 *   164 *   165 *   166 *   167 *   168 *   169
*   170 *   171 *   172 *   173 *   174 *   175 *   176 *   177 *   178 *   179 *   180 *   181 *   182 *   183 *   184 *   185
*   186 *   187 *   188 *   189 *   190 *   191     192     193     194     195     196     197     198     199     200     201
*   202 *   203 *   204 *   205 *   206 *   207 *   208     209 *   420 *   421 *   422 *   423 *   424 *   425 *   426 *   427
*   428 *   429 *   430 *   431 *   432 *   433 *   434 *   435 *   436 *   437 *   438 *   439 *   440 *   441 *   442 *   443
*   444 *   445 *   446 *   447 *   448 *   449 *   450 *   451 *   452 *   453 *   454 *   455 *   456 *   457 *   458 *   459
*   460 *   461 *   462 *   463 *   464 *   465 *   466 *   467 *   468 *   469 *   470 *   471 *   472 *   473 *   474 *   475
*   476 *   477 *   478 *   479 *   480 *   481 *   482 *   483 *   484 *   485 *   486 *   487 *   488 *   489 *   490 *   491
*   492 *   493 *   494 *   495 *   496 *   497 *   498 *   499 *   500 *   501 *   502 *   503 *   504 *   505 *   506 *   507
*   508 *   509 *   510 *   511 *   512 *   513 *   514 *   515     516 *   517 *   518     519 *   209 *   210     211 *   212
*   213 *   214 *   215 *   216 *   217 *   218 *   219 *   220 *   221 *   222 *   223 *   224 *   225 *   226 *   227 *   228
    229     230     231     232     233     234     235     236 *   237 *   238 *   239 *   240 *   241 *   242 *   243 *   244
*   245 *   246 *   247 *   248 *   249 *   250 *   251 *   252 *   253 *   254 *   255 *   256 *   257 *   258 *   259 *   260
*   261 *   262 *   263 *   264 *   265 *   266 *   267 *   268 *   269 *   270 *   271 *   272 *   273 *   274 *   275 *   276
*   277 *   278 *   279 *   280 *   281 *   282 *   283 *   284 *   285 *   286 *   287 *   288 *   289 *   290 *   291 *   292
*   293 *   294 *   295 *   296 *   297 *   298 *   299 *   300 *   301 *   302 *   303 *   304 *   305 *   306 *   307 *   308
*   519     520       0       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 *    26 *    27 *    28 *    29
*    30 *    31 *    32 *    33 *    34 *    35 *    36 *    37 *    38 *    39 *    40 *    41 *    42 *    43 *    44 *    45

      85 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 435X 0   85X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  1TH CYCLE
THERE WERE 507 SUCCESSES I. 512 REQUESTS.

BOTH LAYERS
*    47 *    48 *    49 *    50 *    51 *    52 *    53 *    54 *    55 *    56 *    57 *    58 *    59 *    60 *    61 *    62
*    63 *    64 *    65 *    66 *    67 *    68 *    69 *    70 *    71 *    72 *    73 *    74 *    75 *    76 *    77 *    78
*    79 *    80 *    81 *    82 *    83 *    84 *    85 *    86 *    87 *    88 *    89 *    90 *    91 *    92 *    93 *    94
*    95 *    96 *    97 *    98 *    99 *   100 *   101 *   102 *   103 *   104 *   105 *   106 *   107 *   108 *   109 *   110
*   321 *   322 *   323 *   324 *   325 *   326 *   327 *   328 *   329 *   330 *   331 *   332 *   333 *   334 *   335 *   336
*   337 *   338 *   339 *   340 *   341 *   342 *   343 *   344 *   345 *   346 *   347 *   348 *   349 *   350 *   351 *   352
*   353 *   354 *   355 *   356 *   357 *   358 *   359 *   360 *   361 *   362 *   363 *   364 *   365 *   366 *   367 *   368
*   369 *   370 *   371 *   372 *   373 *   374 *   375 *   376 *   377 *   378 *   379 *   380 *   381 *   382 *   383 *   384
*   385 *   386 *   387 *   388 *   389 *   390 *   391 *   392 *   393 *   394 *   395 *   396 *   397 *   398 *   399 *   400
*   401 *   402 *   403 *   404 *   405 *   406 *   407 *   408 *   409 *   410 *   411 *   412 *   413 *   414 *   415 *   416
*   417 *   418 *   419     420     110 *   111 *   112 *   113 *   114 *   115 *   116 *   117 *   118 *   119 *   120 *   121
*   122 *   123 *   124 *   125 *   126 *   127 *   128 *   129 *   130 *   131 *   132 *   133 *   134 *   135 *   136 *   137
*   138 *   139 *   140 *   141 *   142 *   143 *   144 *   145 *   146 *   147 *   148 *   149 *   150 *   151 *   152 *   153
*   154 *   155 *   156 *   157 *   158 *   159 *   160 *   161 *   162 *   163 *   164 *   165 *   166 *   167 *   168 *   169
*   170 *   171 *   172 *   173 *   174 *   175 *   176 *   177 *   178 *   179 *   180 *   181 *   182 *   183 *   184 *   185
*   186 *   187 *   188 *   189 *   190 *   191 *   192 *   193 *   194 *   195 *   196 *   197 *   198 *   199 *   200 *   201
*   202 *   203 *   204 *   205 *   206 *   207 *   208     209 *   420 *   421 *   422 *   423 *   424 *   425 *   426 *   427
*   428 *   429 *   430 *   431 *   432 *   433 *   434 *   435 *   436 *   437 *   438 *   439 *   440 *   441 *   442 *   443
*   444 *   445 *   446 *   447 *   448 *   449 *   450 *   451 *   452 *   453 *   454 *   455 *   456 *   457 *   458 *   459
*   460 *   461 *   462 *   463 *   464 *   465 *   466 *   467 *   468 *   469 *   470 *   471 *   472 *   473 *   474 *   475
*   476 *   477 *   478 *   479 *   480 *   481 *   482 *   483 *   484 *   485 *   486 *   487 *   488 *   489 *   490 *   491
*   492 *   493 *   494 *   495 *   496 *   497 *   498 *   499 *   500 *   501 *   502 *   503 *   504 *   505 *   506 *   507
*   508 *   509 *   510 *   511 *   512 *   513 *   514 *   515     516 *   517 *   518 *   519 *   209 *   210 *   211 *   212
*   213 *   214 *   215 *   216 *   217 *   218 *   219 *   220 *   221 *   222 *   223 *   224 *   225 *   226 *   227 *   228
*   229 *   230 *   231 *   232 *   233 *   234 *   235 *   236 *   237 *   238 *   239 *   240 *   241 *   242 *   243 *   244
```

Figure H.2 CN Simulator Output, Second Example

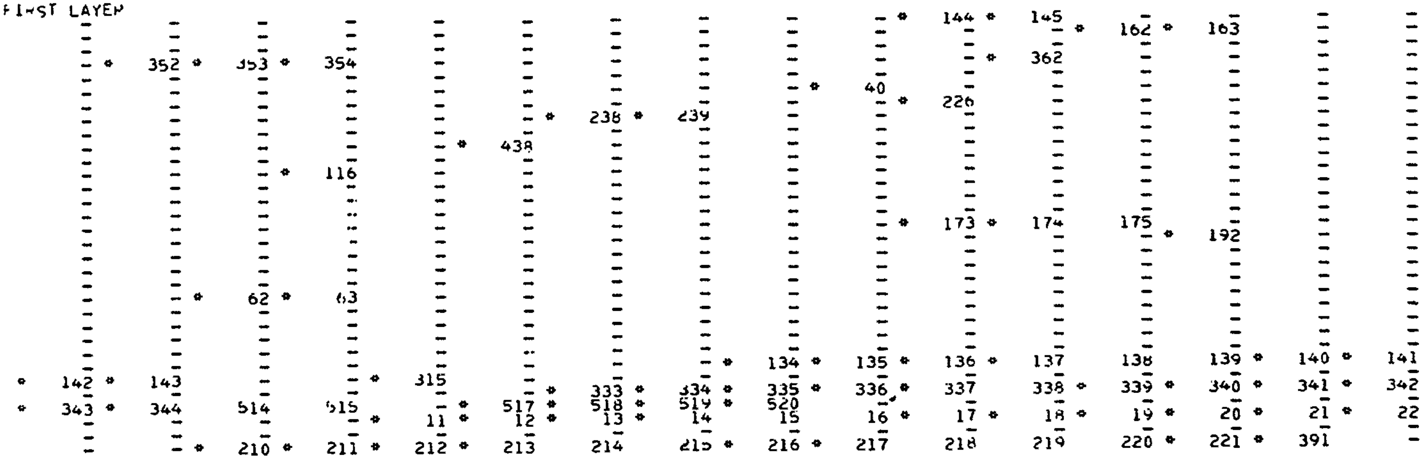
```
  *     245 *     246 *     247 *     248 *     249 *     250 *     251 *     252 *     253 *     254 *     255 *     256 *     257 *     258 *     259 *     260
  *     261 *     262 *     263 *     264 *     265 *     266 *     267 *     268 *     269 *     270 *     271 *     272 *     273 *     274 *     275 *     276
  *     277 *     278 *     279 *     280 *     281 *     282 *     283 *     284 *     285 *     286 *     287 *     288 *     289 *     290 *     291 *     292
  *     293 *     294 *     295 *     296 *     297 *     298 *     299 *     300 *     301 *     302 *     303 *     304 *     305 *     306 *     307 *     308
  *     519 *     520 *       0 *       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 *      26 *      27 *      28 *      29
  *      30 *      31 *      32 *      33 *      34 *      35 *      36 *      37 *      38 *      39 *      40 *      41 *      42 *      43 *      44 *      45
         5 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 516X 0      5X 1      0X 2      0X 3      0X 4      0X 5      0X 6      0X 7      0X 8      0X 9      0X10      0X11      0X12      0X13      0X14      0X15      0

ON 2TH CYCLE
THERE WERE   5 SUCCESSES IN   5 REQUESTS.

FIRST LAYER
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         - *     420 *     110         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         - *     209         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         - *     516         -         - *     519         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -
         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -         -

         0 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 521X 0      0X 1      0X 2      0X 3      0X 4      0X 5      0X 6      0X 7      0X 8      0X 9      0X10      0X11      0X12      0X13      0X14      0X15      0
```

Figure H.2 CN Simulator Output, Second Example (Cont'd)

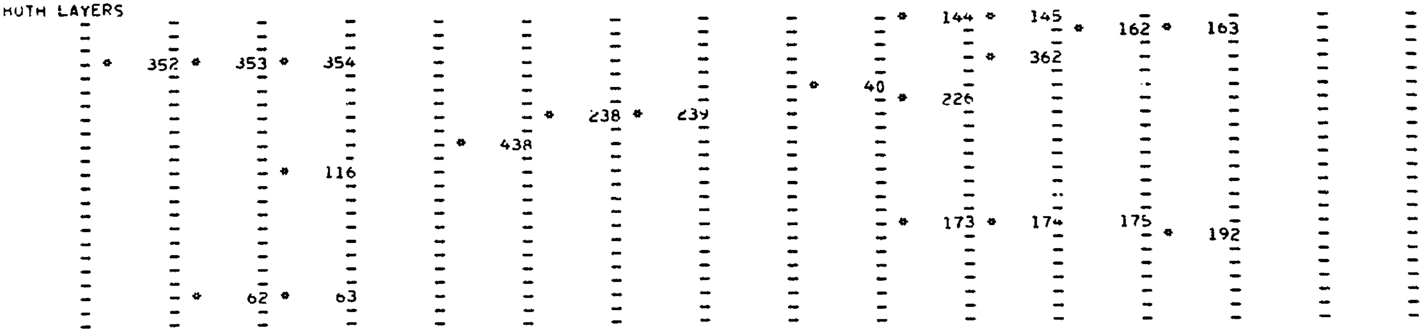
```
? END BR TI F14 Q134  1 30  1170 XX SMALL-GRID STEGER OF NASAS LISTING        000000000000000000000000000000000000000000000000
OFFSET=134 SKIP=  1   30 ELEMENTS PER PIECE; FIRST PIECE=  29, 170SPACE BETWEEN PIECES
IN TEST.  EM/TN CLOCK RATIO  1 NREQ512 ZMAX 47 PRIORITY  0    FRIDAY  02/16/79   16:24:44
PORTS: PROC UNPACKED; EM UNPACKED. NET NO.  2
NO OF ACCESS CYCLES REQUESTED IS  1

      512 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
  57X 0  396X 1   58X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  1TH CYCLE
THERE WERE 327 SUCCESSES IN 512 REQUESTS.

FIRST LAYER
 *   134 *   135 *   136 *   137 *   138 *   139 *   140 *   141 *   142 *   143     144     145 *   146 *   147 *   148 *   149
 *   150 *   151 *   152 *   153 *   154 *   155 *   156 *   157 *   158 *   159 *   160 *   161     162     163 *   333 *   334
 *   335 *   336 *   337 *   338 *   339 *   340 *   341 *   342 *   343 *   344 *   345 *   346 *   347 *   348 *   349 *   350
 *   351     352     353     354 *   355 *   356 *   357 *   358 *   359 *   360 *   361     362 *    11 *    12 *    13 *    14
 *    15 *    16 *    17 *    18 *    19 *    20 *    21 *    22 *    23 *    24 *    25 *    26 *    27 *    28 *    29 *    30
 *    31 *    32 *    33 *    34 *    35 *    36 *    37 *    38 *    39      40 *   210 *   211 *   212 *   213 *   214 *   215
 *   216 *   217 *   218 *   219 *   220 *   221 *   222 *   223 *   224 *   225     226 *   227 *   228 *   229 *   230 *   231
 *   232 *   233 *   234 *   235 *   236 *   237     238     239 *   409 *   410 *   411 *   412 *   413 *   414 *   415 *   416
 *   417 *   418 *   419 *   420 *   421 *   422 *   423 *   424 *   425 *   426 *   427 *   428 *   429 *   430 *   431 *   432
 *   433 *   434 *   435 *   436 *   437     438 *    87 *    88 *    89 *    90 *    91 *    92 *    93 *    94 *    95 *    96
 *    97 *    98 *    99 *   100 *   101 *   102 *   103 *   104 *   105 *   106 *   107 *   108 *   109 *   110 *   111     112
     113     114 *   115     116 *   286 *   287 *   288 *   289 *   290 *   291 *   292 *   293 *   294 *   295 *   296 *   297
 *   298 *   299 *   300 *   301 *   302 *   303 *   304 *   305 *   306 *   307 *   308 *   309 *   310 *   311 *   312 *   313
 *   314 *   315 *   485 *   486 *   487 *   488 *   489 *   490 *   491 *   492 *   493 *   494 *   495 *   496 *   497 *   498
 *   499 *   500 *   501 *   502 *   503 *   504 *   505 *   506 *   507 *   508 *   509 *   510 *   511 *   512 *   513 *   514
 *   163 *   164 *   165 *   166 *   167 *   168 *   169 *   170 *   171 *   172     173     174     175 *   176 *   177 *   178
     179     180     181     182     183     184     185     186     187     188     189     190     191     192 *   362     363
 *   364 *   365 *   366 *   367 *   368 *   369 *   370 *   371 *   372 *   373 *   374 *   375 *   376 *   377     378     379
     380     381     382     383 *   384 *   385 *   386 *   387 *   388 *   389 *   390 *   391 *    40      41 *    42 *    43
 *    44      45      46      47 *    48 *    49 *    50 *    51 *    52 *    53 *    54 *    55      56      57      58      59
      60      61      62      63      64      65      66      67      68      69 *   239 *   240 *   241 *   242 *   243 *   244
 *   245 *   246 *   247 *   248 *   249 *   250 *   251 *   252 *   253 *   254     255 *   256 *   257 *   258 *   259 *   260
 *   261 *   262 *   263 *   264 *   265 *   266 *   267 *   268 *   438 *   439 *   440 *   441 *   442 *   443 *   444     445
     446     447 *   448 *   449 *   450 *   451 *   452 *   453     454     455     456     457     458     459     460     461
     462     463     464     465     466     467 *   116     117 *   118 *   119 *   120 *   121 *   122 *   123 *   124 *   125
 *   126 *   127     128     129     130     131     132     133     134     135     136     137     138     139     140     141
     142     143 *   144 *   145     315 *   316     317     318     319     320     321     322     323     324     325     326
     327     328     329     330     331     332     333     334     335     336     337     338     339     340     341     342
     343     344     514     515     516     517     518     519     520       0       1       2       3       4       5       6
       7       8       9      10      11      12      13      14      15      16      17      18      19      20      21      22
     192     193     194     195     196     197     198     199     200     201     202     203     204     205     206     207
     208     209     210     211     212     213     214     215     216     217     218     219     220     221     391     392

      185 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 337X 0  183X 1    1X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  1TH CYCLE
THERE WERE 438 SUCCESSES IN 512 REQUESTS.

BOTH LAYERS
 *   134 *   135 *   136 *   137 *   138 *   139 *   140 *   141 *   142 *   143     144     145 *   146 *   147 *   148 *   149
 *   150 *   151 *   152 *   153 *   154 *   155 *   156 *   157 *   158 *   159 *   160 *   161     162     163 *   333 *   334
 *   335 *   336 *   337 *   338 *   339 *   340 *   341 *   342 *   343 *   344 *   345 *   346 *   347 *   348 *   349 *   350
 *   351     352     353     354 *   355 *   356 *   357 *   358 *   359 *   360 *   361     362 *    11 *    12 *    13 *    14
 *    15 *    16 *    17 *    18 *    19 *    20 *    21 *    22 *    23 *    24 *    25 *    26 *    27 *    28 *    29 *    30
 *    31 *    32 *    33 *    34 *    35 *    36 *    37 *    38 *    39      40 *   210 *   211 *   212 *   213 *   214 *   215
 *   216 *   217 *   218 *   219 *   220 *   221 *   222 *   223 *   224 *   225     226 *   227 *   228 *   229 *   230 *   231
 *   232 *   233 *   234 *   235 *   236 *   237     238     239 *   409 *   410 *   411 *   412 *   413 *   414 *   415 *   416
 *   417 *   418 *   419 *   420 *   421 *   422 *   423 *   424 *   425 *   426 *   427 *   428 *   429 *   430 *   431 *   432
 *   433 *   434 *   435 *   436 *   437     438 *    87 *    88 *    89 *    90 *    91 *    92 *    93 *    94 *    95 *    96
 *    97 *    98 *    99 *   100 *   101 *   102 *   103 *   104 *   105 *   106 *   107 *   108 *   109 *   110 *   111 *   112
 *   113 *   114 *   115     116 *   286 *   287 *   288 *   289 *   290 *   291 *   292 *   293 *   294 *   295 *   296 *   297
 *   298 *   299 *   300 *   301 *   302 *   303 *   304 *   305 *   306 *   307 *   308 *   309 *   310 *   311 *   312 *   313
 *   314 *   315 *   485 *   486 *   487 *   488 *   489 *   490 *   491 *   492 *   493 *   494 *   495 *   496 *   497 *   498
 *   499 *   500 *   501 *   502 *   503 *   504 *   505 *   506 *   507 *   508 *   509 *   510 *   511 *   512 *   513 *   514
 *   163 *   164 *   165 *   166 *   167 *   168 *   169 *   170 *   171 *   172     173     174     175 *   176 *   177 *   178
 *   179 *   180 *   181 *   182 *   183 *   184 *   185 *   186 *   187 *   188 *   189 *   190 *   191     192 *   362 *   363
 *   364 *   365 *   366 *   367 *   368 *   369 *   370 *   371 *   372 *   373 *   374 *   375 *   376 *   377 *   378 *   379
 *   380 *   381 *   382 *   383 *   384 *   385 *   386 *   387 *   388 *   389 *   390 *   391 *    40 *    41 *    42 *    43
 *    44 *    45 *    46 *    47 *    48 *    49 *    50 *    51 *    52 *    53 *    54 *    55 *    56 *    57 *    58 *    59
 *    60 *    61      62      63 *    64 *    65 *    66 *    67 *    68 *    69 *   239 *   240 *   241 *   242 *   243 *   244
 *   245 *   246 *   247 *   248 *   249 *   250 *   251 *   252 *   253 *   254 *   255 *   256 *   257 *   258 *   259 *   260
 *   261 *   262 *   263 *   264 *   265 *   266 *   267 *   268 *   438 *   439 *   440 *   441 *   442 *   443 *   444 *   445
 *   446 *   447 *   448 *   449 *   450 *   451 *   452 *   453 *   454 *   455 *   456 *   457 *   458 *   459 *   460 *   461
 *   462 *   463 *   464 *   465 *   466 *   467 *   116 *   117 *   118 *   119 *   120 *   121 *   122 *   123 *   124 *   125
```

Figure H.3 CN Simulator Output, Third Example

```
*   126 *   127 *   128 *   129 *   130 *   131 *   132 *   133     134     135     136     137     138     139     140     141
    142     143 *   144 *   145     315 *   316 *   317 *   318 *   319 *   320 *   321 *   322 *   323 *   324 *   325 *   326
*   327 *   328 *   329 *   330 *   331 *   332 .   333     334     335     336     337     338     339     340     341     342
    343     344     514     515 *   516     517     518     519     520 *     0 *     1 *     2 *     3 *     4 *     5 *     6
*     7 *     8 *     9 *    10      11      12      13      14      15      16      17      18      19      20      21      22
*   192 *   193 *   194 *   195 *   196 *   197 *   198 *   199 *   200 *   201 *   202 *   203 *   204 *   205 *   206 *   207
*   208 *   209     210     211     212     213     214     215     216     217     218     219     220     221     391 *   392

      74 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 447X 0   74X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  2TH CYCLE
THERE WERE  61 SUCCESSES IN  74 REQUESTS.

FIRST LAYER
```



```
      13 ITEMS THIS CYCLE
DISTRIBUTION OF EM CONFLICTS
 508X 0   13X 1    0X 2    0X 3    0X 4    0X 5    0X 6    0X 7    0X 8    0X 9    0X10    0X11    0X12    0X13    0X14    0X15    0

ON  2TH CYCLE
THERE WERE  73 SUCCESSES IN  74 REQUESTS.

BOTH LAYERS
```



REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

Figure H.3 CN Simulator Output, Third Example (Cont'd)

```
*   142 *   143          *   315               *   134 *   135 *   136 *   137 *   138 *   139 *   140 *   141
*   343 *   344 *   514 *   515        *   333 *   334 *   335 *   336 *   337 *   338 *   339 *   340 *   341 *   342
                                  *  517 *  518 *  519 *  520
                         *    11 *   12 *   13 *   14 *   15 *   16 *   17 *   18 *   19 *   20 *   21 *   22
              *   210 *   211 *   212 *   213 *   214 *   215 *   216 *   217 *   218 *   219 *   220 *   221 *   391
       1 ITEMS THIS CYCLE
UISTRIBUTION OF EM CONFLICTS
 520X 0    1X 1    UX 2    0X 3    0X 4    UX 5    0X 6    0X 7    UX 8    0X 9    UX10    0X11    UX12    0X13    0X14    0X15    0

ON  3TH CYCLE
THERE WERE   1 SUCCESSES IN   1 REQUESTS.

FIRST LAYER

                                                                                     *  175

       0 ITEMS THIS CYCLE
UISTRIBUTION OF EM CONFLICTS
 521X 0    0X 1    UX 2    UX 3    0X 4    UX 5    0X 6    0X 7    UX 8    0X 9    UX10    0X11    UX12    0X13    0X14    0X15    0
```

Figure H.3 CN Simulator Output, Third Example (Cont'd)

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

2d and 3d line: Report of distribution of EM conflicts (pileups). (number of pileups) x (length of pileup), from lengths of 0 through 15. Any EM module with a pileup of 10 or more will have a line stating its module number and the size of the pileup.

4th line: "On the nth cycle"

5th line: "There were sss successes in rrr requests". For version A, the number of successes listed in the first report is for the first layer; the number of successes listed in the second report for the nth cycle is the total for both layers.

Next 32 lines: 512 entries, one for each processor. At each entry we find "-" if no request was made, otherwise the EM module number of the request, prefixed by "*" if the request was granted, by "EM" if the EM cycle is still running, so the path is locked up.

H.3 CONNECTION NETWORK STOCHASTIC ANALYZER

The Connection Network Stochastic Analyzer is used to compute the probabilities of input, output and blockage for each switch across the connection network (CN). These computations are then used to determine blockage at each level, and finally to determine total blockage. This tool was not developed to test the performance of the CN under specific conditions. Rather, the question raised was what would the effect of the CN be <u>on the average</u>. An initial assumption was made that the inputs to be evaluated would be random permutations of the destination addresses. Under this assumption, no blockage would occur due to simultaneous reference to the same destination. Although such a situation will actually occur, it is a misleading situation when studying the effect of the network itself. The functional simulator did allow consideration of such simultaneous reference situations.

H.3.1 <u>Model</u>

The Stochastic Analyzer was implemented to study the single-layer Omega network and the double-layer Omega network with interlayer paths at each node. An example of such a network is shown in Figure H.4. In this figure 8 processors and 11 memories are connected. For the purposes of this model, the 11 extended memory modules are "spread" as evenly as possible across the output ports of the net (i.e. with 16/11 steps between each connection.) This mapping should be equivalent (although it is not the same) to some of the mappings discussed in Appendix B.

H.3.1.1 Input Probabilities

Since only random permutations are considered, each destination port address (for those ports with memory modules attached) occurs with equal probability. As pointed out earlier, there may not be as many memory modules connected as there are output ports on the network. As a result, the probability that a specific bit in the destination address = 0 or 1 is likely to vary from bit to bit.

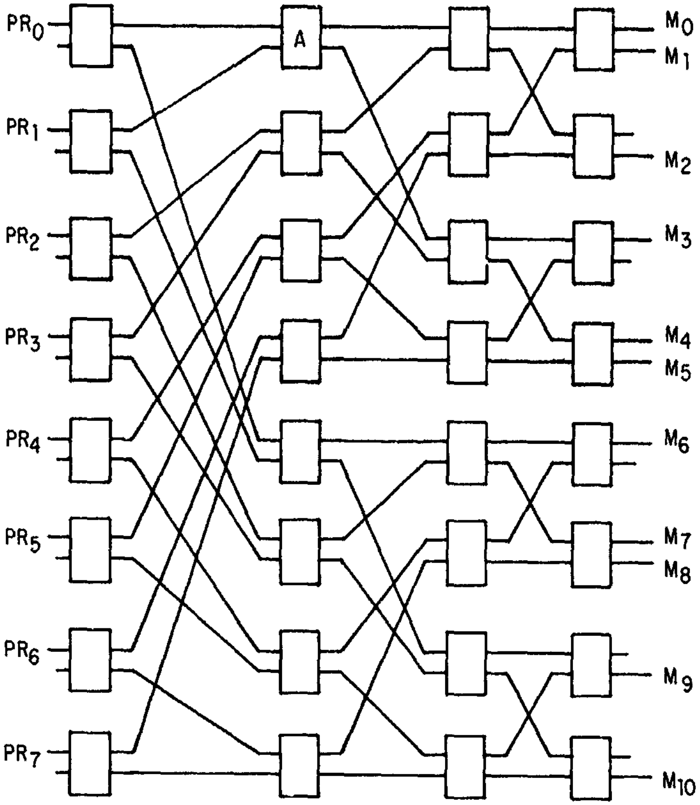


Figure H.4 Omega Network with 8 processors and 11 Extended Memory Modules

For example, in Figure H.4, the probability that the high order bit of the destination address = 1 is 5/11 and the probability that that bit = 0 is 6/11. This probability affects the probability of an address occurring on one or the other outputs of a switch.

H.3.1.2 Probability Computations

The computations performed by the analyzer are based on the probability of occurrence of each possible input combination to each switch or node in the network. For example, consider the switch marked A in Figure H.4. For this example, assume that the network is a single-layer Omega network. The probability of blockage in that switch is the probability that the inputs are either both 0 or both 1 simultaneously. If P(INPUT) is the probability of an input request occurring and if P(0-BIT), P(1-BIT), P(2-EIT), P(3-BIT) are the probabilities of the high order through low-order bits of the destination address, then the upper input to A exists with the probability:

$$P(\text{A-UPPER}) = P(\text{INPUT}) \times P(\text{0-BIT}=1) \qquad (\text{H.1})$$

Similarly, for the lower input

$$P(\text{A-LOWER}) = P(\text{INPUT}) \times P(\text{0-BIT}=1) \qquad (\text{H.2})$$

Then, the probability of blockage in switch A can now be determined.

$$P(\text{A-UPPER}=1) = P(\text{INPUT} \times P(\text{0-BIT}=1) \times P(\text{L-BIT}=1) \qquad (\text{H.3})$$
$$P(\text{A-UPPER}=0) = P(\text{INPUT}) \times P(\text{0\_BIT}=1) \times P(\text{1-BIT}=0) \qquad (\text{H.4})$$
$$P(\text{A-LOWER}=1) = P(\text{INPUT}) \times P(\text{0-BIT}=1) \times P(\text{1-BIT}=1) \qquad (\text{H.5})$$
$$P(\text{A-LOWER}=0) = P(\text{INPUT}) \times P(\text{0-BIT}=1) \times P(\text{1-BIT}=0) \qquad (\text{H.6})$$
$$P(\text{BLOCK-IN-A}) = P(\text{A-UPPER}=1) \times P(\text{A-LOWER}=1) + P(\text{A-UPPER}=0) \times P(\text{A-LOWER}=0) \qquad (\text{H.7})$$

Substituting known values:

$P(\text{INPUT}) = 1$ (assume all inputs active)
$P(\text{0-BIT}=1) = 5/11$
$P(\text{1-BIT}=0) = 6/11$
$P(\text{1-BIT}=1) = 5/11$

Then:

$$P(\text{BLOCK-IN-A}) = (5/11 \times 5/11) \times (5/11 \times 5/11) + 5/11 \times 6/11) \times (5/11 \times 6/11) = .104$$

Using similar techniques, the probability of outputs occuring on the outputs of switch A can be determined. This sort of computation can then be carried on through the network, taking into account the probability of blockages and the probability of the corresponding address control bit.

H.3.2 Analyzer Controls

The user inputs the number of processors and memory modules in the system as well as the number of switch levels (up to 10), the number of input connection points, the number of active processors (total processors), and the number of layers (1 or 2) in the network. Using this information the analyzer builds a table representing the connection network. This table provides for processors to be mapped onto input ports, outputs from one level mapped onto inputs of the next level, and switch outputs mapped onto memory modules. Each switch's input probability is used to compute its own output and blockage probabilities.

H.3.3 Analyzer Output

When the calculations for each switch are completed, a listing is prepared which fully describes the network analyzed. All of the user-input information is printed as well as processor and memory mod mappings. Total blockage and blockage for each level is printed, as well as each switch's output probabilities.

Figure H.5 shows an example of an additional output which summarizes the results of a number of runs. The output for each run specifies the number of processors, of memory modules and of ports in the network being evaluated. The number of active processors identifies the average number of processors actively presenting requests to the network. Cumulative blockage probability is the probability that any request made is blocked somewhere within the network. The number of inputs per switch identifies which type of network was run. A 2-input switch is used on the single-layer Omega network. A 4-input switch is used on the double-layer Omega network with interlayer communication. The line identified as Probability of Blockage summarizes the cumulative blockage at each level through the network from the processors (on the left) to the memory modules (on the right).

REPRODUCIBILITY OF THE
ORIGINAL PAGE IS POOR

```
                                              OUTPUT PROBABILITIES


                              PROCESSORS:    512
                              MEMORY MODULES:    521
                              INPUT CONNECTION POINTS:    1024
                              ACTIVE PROCESSORS:    256          INPUT PROBABILITY:  0.5000
                              CUMULATIVE BLOCKAGE PROBABILITY:  0.0332
                               NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:   4



PROBABILITY OF BLOCKAGE:   0.0000 0.0000 0.0010 0.0016 0.0026 0.0033 0.0038 0.0044 0.0048 0.0115
                                              OUTPUT PROBABILITIES


                              PROCESSORS:    512
                              MEMORY MODULES:    521
                              INPUT CONNECTION POINTS:    1024
                              ACTIVE PROCESSORS:    256          INPUT PROBABILITY:  0.5000
                              CUMULATIVE BLOCKAGE PROBABILITY:  0.3950
                              NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:   2



PROBABILITY OF BLOCKAGE:   0.0000 0.0620 0.0546 0.0484 0.0432 0.0389 0.0352 0.0319 0.0292 0.0516
                                              OUTPUT PROBABILITIES


                              PROCESSORS:    512
                              MEMORY MODULES:    521
                              INPUT CONNECTION POINTS:    1024
                              ACTIVE PROCESSORS:    307          INPUT PROBABILITY:  0.5996
                              CUMULATIVE BLOCKAGE PROBABILITY:  0.0494
                              NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:   4



PROBABILITY OF BLOCKAGE:   0.0000 0.0000 0.0017 0 0030 0.0041 0.0050 0.0058 0.0064 0.0069 0.0165
```

Figure H 5 Stochastic Analyzer Sample Output

```
                                       OUTPUT PROBABILITIES


                               PROCESSORS:    512
                               MEMORY MODULES:     512
                               INPUT CONNECTION POINTS:    1024
                               ACTIVE PROCESSORS:    307        INPUT PROBABILITY:  0.5996
                               CUMULATIVE BLOCKAGE PROBABILITY:  0.4403
                               NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:  2


PROBABILITY OF BLOCKAGE:   0.0000 0.0741 0.0635 0.0551 0.0483 0.0427 0.0380 0.0341 0.0307 0.0553
                                       OUTPUT PROBABILITIES


                               PROCESSORS:    512
                               MEMORY MODULES:     512
                               INPUT CONNECTION POINTS:    1024
                               ACTIVE PROCESSORS:    358        INPUT PROBABILITY:  0.6992
                               CUMULATIVE BLOCKAGE PROBABILITY:  0.0679
                               NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:  4


PROBABILITY OF BLOCKAGE:   0.0000 0.0000 0.0026 0.0045 0.0059 0.0071 0.0080 0.0087 0.0092 0.0220
                                       OUTPUT PROBABILITIES



                               PROCESSORS:    512
                               MEMORY MODULES:     512
                               INPUT CONNECTION POINTS:    1024
                               ACTIVE PROCESSORS:    358        INPUT PROBABILITY:  0.6992
                               CUMULATIVE BLOCKAGE PROBABILITY:  0.4792
                               NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:  2


PROBABILITY OF BLOCKAGE:   0.0000 0.0861 0.0719 0.0610 0.0525 0.0455 0.0401 0.0355 0.0317 0.0550
```

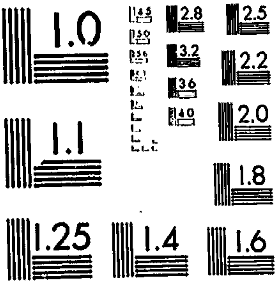
Figure H.5 Stochastic Analyzer Sample Output (Cont'd)

```
                                              OUTPUT PROBABILITIES


                                    PROCESSORS:    512
                                    MEMORY MODULES:    521
                                    INPUT CONNECTION POINTS:    1024
                                    ACTIVE PROCESSORS:    410         INPUT PROBABILITY:   0.0006
                                    CUMULATIVE BLOCKAGE PROBABILITY:  0.0686
                                     NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:   4



PROBABILITY OF BLOCKAGE:    0.0000 0.1000 0.0037 0.0063 0.0061 0.0095 0.0104 0.0112 0.0116 0.0277
                                               OUTPUT PROBABILITIES


                                    PROCESSORS:    512
                                    MEMORY MODULES:    521
                                    INPUT CONNECTION POINTS:    1024
                                    ACTIVE PROCESSORS:    410         INPUT PROBABILITY:   0.0006
                                    CUMULATIVE BLOCKAGE PROBABILITY:  0.5134
                                    NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:   2



PROBABILITY OF BLOCKAGE:    0.0000 0.0981 0.0797 0.0662 0.0559 0.0479 0.0416 0.0364 0.0321 0.0554
                                               OUTPUT PROBABILITIES


                                    PROCESSORS:    512
                                    MEMORY MODULES:    521
                                    INPUT CONNECTION POINTS:    1024
                                    ACTIVE PROCESSORS:    461         INPUT PROBABILITY:   0.0004
                                    CUMULATIVE BLOCKAGE PROBABILITY:  0.1102
                                    NUMBER OF SWITCHES:  512      NUMBER OF INPUTS PER SWITCH:   4



PROBABILITY OF BLOCKAGE:    0.0000 0.0000 0.0052 0.0065 0.0106 0.0120 0.0130 0.0136 0.0140 0.0333
```

Figure H 5 Scochastic Analyzer Sample Output (Cont'd)

OF 7

26072



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

```
                                        OUTPUT PROBABILITIES


                         PROCESSORS:    512
                         MEMORY MODULES:    512
                         INPUT CONNECTION POINTS:    1024
                         ACTIVE PROCESSORS:    461         INPUT PROBABILITY:  0.9004
                         CUMULATIVE BLOCKAGE PROBABILITY:  0.3424
                         NUMBER OF SWITCHES:  512       NUMBER OF INPUTS PER SWITCH:   2


PROBABILITY OF BLOCKAGE:   0.0000 0.1097 0.0869 0.0707 0.0587 0.0496 0.0425 0.0369 0.0323 0.0552
                                        OUTPUT PROBABILITIES


                         PROCESSORS:    512
                         MEMORY MODULES:    512
                         INPUT CONNECTION POINTS:    1024
                         ACTIVE PROCESSORS:    512         INPUT PROBABILITY:  1.0000
                         CUMULATIVE BLOCKAGE PROBABILITY:  0.1326
                         NUMBER OF SWITCHES:  512       NUMBER OF INPUTS PER SWITCH:   4


PROBABILITY OF BLOCKAGE:   0.0000 0.0000 0.0070 0.0109 0.0133 0.0147 0.0156 0.0161 0.0163 0.0387
                                        OUTPUT PROBABILITIES


                         PROCESSORS:    512
                         MEMORY MODULES:    512
                         INPUT CONNECTION POINTS:    1024
                         ACTIVE PROCESSORS:    512         INPUT PROBABILITY:  1.0000
                         CUMULATIVE BLOCKAGE PROBABILITY:  0.1677
                         NUMBER OF SWITCHES:  512       NUMBER OF INPUTS PER SWITCH:   2


PROBABILITY OF BLOCKAGE:   0.0000 0.1211 0.0934 0.0745 0.0609 0.0503 0.0431 0.0370 0.0322 0.0547
```

Figure H.5 Stochastic Analyzer Sample Output (Cont'd)

APPENDIX I
BENES AND OMEGA NETWORKS FOR FLOW MODEL PROCESSING*

I.1 INTRODUCTION

Parallel processing machines gain time at the expense of additional processing elements. However, parallelism entails processor access problems. The major assumptions of the NASF Flow Model Processor are:

1) There are 512 processing elements and 521 extended memory modules.

2) Some hybrid of a Benes or Omega network is used to connect processor elements to EM modules and processing elements to processing elements (See Figure I.1A and I.1B).

Roughly, the more processing elements, the faster the machine can run, given a program which exhibits a large degree of parallelism. If there is a prime number of memory modules -- 521 is prime--then corresponding column elements of a p-ordered vector are stored in different extended memory modules, making it particularly easy to access a column at a time (see Figure I.2). However, in assuming 521 EM modules, we presume that matrices are to be stored across the EM's. It may be beneficial to be a slight bit heretical and ask whether matrices stacked into a single EM might not be more effective in executing block transfers to local memory. It is to be remembered that a single processor will most often want, say VECT(I), VECT(I+1), and VECT(I-1), which may be stored concurrently in local memory. This, however, seems to be mainly a software problem.

The choice of a Benes or an Omega network is a pragmatic one based on required hardware and expected transmission time. (See the chart on pg. 109 of Ref. 1). Ultimately, we settle for Benes and Omega networks because they appear to be the most efficacious solution presently available. While Benes (2, 3, 4) has shown that for the network which bears his name, there exists a nonblocking control pattern for every arrangement of inputs to outputs, practically speaking, computation time is prohibitive. Thus, the concept of distributed control arises; this concept works especially well with an Omega--since at the ith level in the network, there is a relatively simple mapping between the ith most significant bits of two or more addresses and the state of the switch.

*Originally submitted in September, 1978.

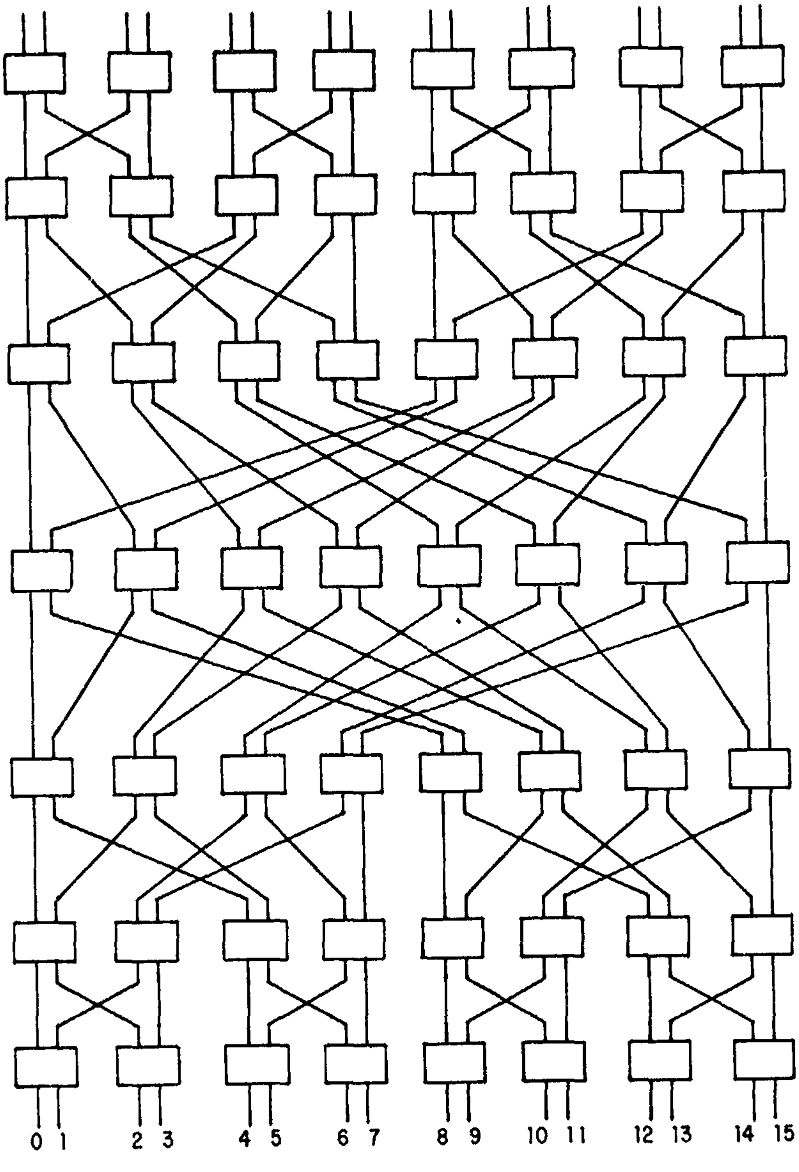


Figure I.1A
Bene; Network (n=4)

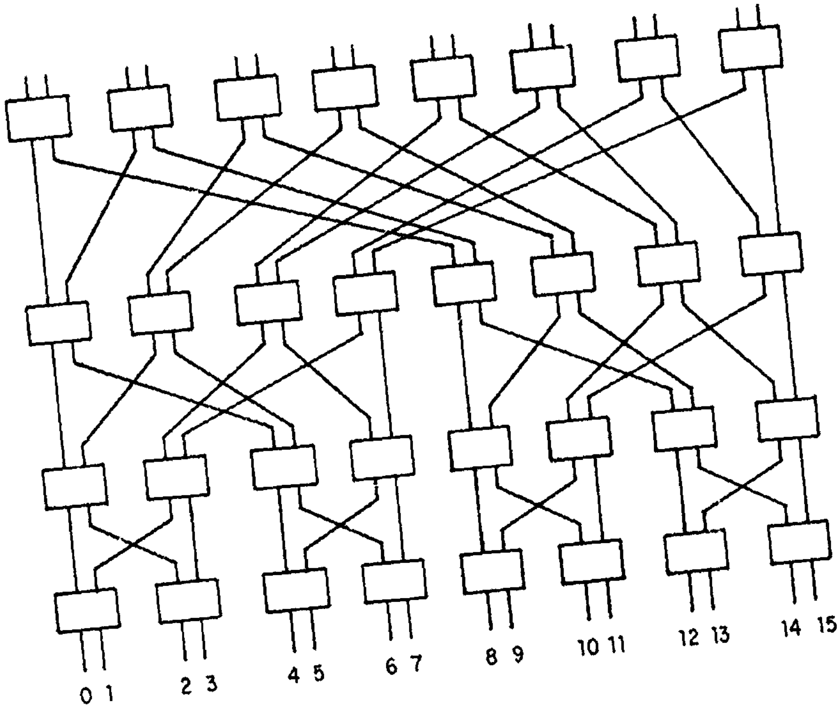
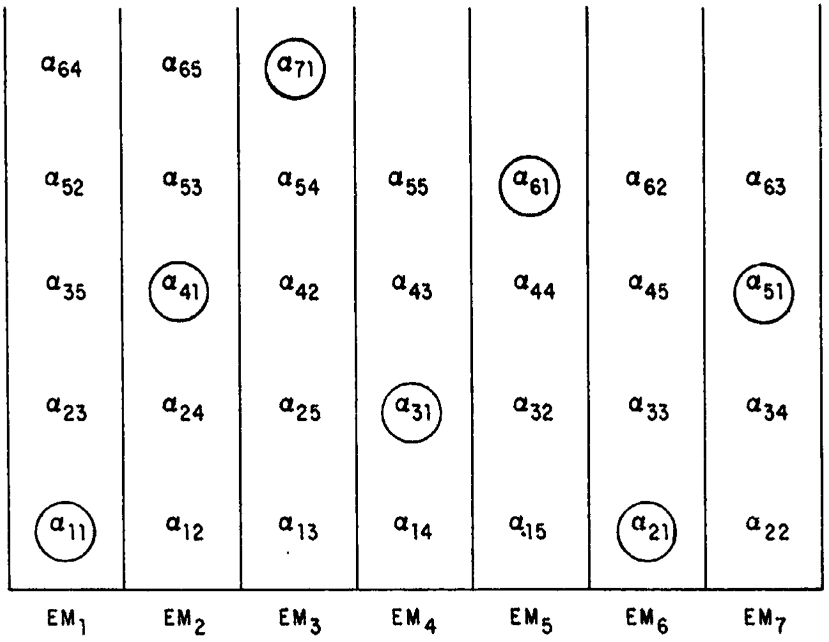


Figure I.1B
Omega Network (n=4)



a_{64}	a_{65}	a_{71}				
a_{52}	a_{53}	a_{54}	a_{55}	a_{61}	a_{62}	a_{63}
a_{35}	a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{51}
a_{23}	a_{24}	a_{25}	a_{31}	a_{32}	a_{33}	a_{34}
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{21}	a_{22}
EM_1	EM_2	EM_3	EM_4	EM_5	EM_6	EM_7

Storing a p=5 Matrix in a Prime Number of EM Modules

Figure I.2

Storing a p=5 Matrix
in a Prime Number of EM Modules

I.2 ANALYSIS OF TWO-LEVEL OMEGA NETWORK WITH INTER-LAYER CONNECTIONS

The object of the two-layer Omega node-level analysis is to obtain the blocking probability at each level in the network from its input probabilities. The output probabilities can then be calculated from the input probabilities and blocking that goes on at each switch. These outputs are then re-ordered by the connectivity of the network, and they become the inputs for the next level.

An important fact in this analysis is that each switch in a given level has the same set of input probabilities. Thus, the probability of a block at one switch becomes the probability of blocks at N switches. We assume an unpacked Omega (with N processors attached to 2N input ports), so that the inputs to level one are all at the A-port of the first layer node. (Figure I.3). There are then two possible inputs since it is equally likely that the address bit will be a one or a zero. These bits determine the switching operations performed by the node on the address under the switching rules. It is clear that on the first level of an unpacked network, there will be no blocks, and that, furthermore, the second layer is not used. The topology of the network implies that there are nine possible input combinations to the second level, each of which has an associated probability. On the second level, the fact that I_A= address, I_B = address (where I_A and I_B are inputs A and B) is now a possible combination implies that the second layer is now used, although there are still no blocks on this layer. There are 49 possible input combinations to the third level. Blockage is now possible since there can be three inputs to one two-layer switch pair. On the fourth, and all subsequent levels, there are 81 input types. This is basically base three in four places where the three characters 0,1 and blank are permuted over I_A, I_B, I_A and I_B. Table I.4 gives the input, output and blocking probabilities for these first four levels done in the hand simulation.

There are two concepts which should be understood concerning the evaluation of the network through each succeeding stage. They are a) increasing randomness, and b) decreasing density. While initially most of the addresses are on the lower layer, conflicts on the lower layer tend to send more addresses to the upper layer. In equilibrium, both layers will be equally occupied. Now a necessary condition for a block in the network is that there be two inputs on one layer, and one on the other. One might think, therefore, that maximum blocking will occur when the first layer has twice as many addresses as the second layer, but since blocking is symmetric between layers, maximum blocking is expected to occur when the two layers are equally dense, i.e. when the system is completely randomized. On the other hand, the fact that blocking implies a decreased density of addresses as the addresses are blocked means that the number of blocks should decrease as re-

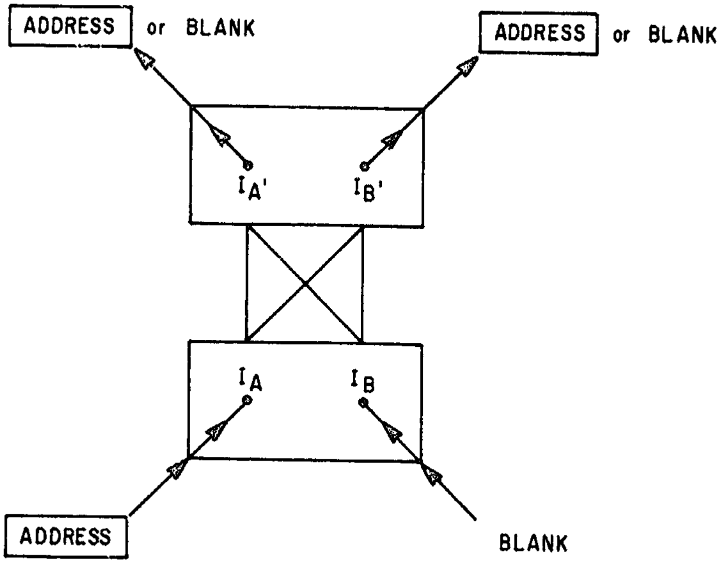


Figure I.3

A node of level 1

quests progress through the network. However, since the actual number of blocks is small, this effect will be small at first, but will become more important as the system tends towards equilibrium. Thus, we would expect the blocking probability to increase initially due to the effect of randomization, and then to decrease due to the effect of decreasing density. However, it is not clear on which level the turning point will occur.

When it was realized that more blocks would occur on some levels than on others, a search began for the best way of adding a small amount of hardware to improve performance. First, it was felt that one probably not want to take any levels off of the second layer (with the exception perhaps of the first level), since the loss in efficieny at that level would be greater than any "marginal gains" at any point in a third layer. Secondly, it is clear that once a new layer is initiated, it must be continued to the destination ports if the addresses are not to be injected back into the lower layers.

This led to the concept of the three-layer network by asking how such a system might "grow". Indeed, there are some similarities between the transposition network and the corpus callosum (which unites the two hemispheres) of the human brain. However it seems somewhat deceptive to think in terms of layers, for each switch pair may be reduced to a planar circuit with, say, four inputs being mapped to four outputs. As described in Chapter 5, each of these input and output sets are composed of twelve or more wires, at least nine of which control the switch settings for the various levels of the network; the other three or more wires may play special parts in the local control of the switch. The frames of data may follow the 'net-code' through the network to be stored in buffers at the terminal end.

I.3 SKIP DISTANCE ANALYSIS

When a p-ordered vector is stored across extended memory, corresponding column elements are stored in modules (o+pi)mod521 as shown in Figure I.2, where o is the offset and i is the row number. When each processor gets a succeeding row element, i becomes the processor number. This is particularly important in lock-step operation, but is also relevant in the early stages of any loop.

Results of hand-simulations which were performed for 8 x 11 unpacked Benes and Omega networks are summarized in Tables I.1, I.2 and I.3. It becomes clear from these charts that these networks are symmetric with respect to skip distance, i.e. there is a correspondence:

skip 1 to skip 10
skip 2 to skip 9
skip 3 to skip 8
skip 4 to skip 7
skip 5 to skip 6

Level 1	$O_aO_{a'}$	$O_bO_{b'}$	Blocking
**	½	½	0%
A*	½	½	
*A	0	0	
AA	0	0	
Level 2			
**	9/16	9/16	0%
A*	6/16	6/16	
*A	0	0	
AA	1/16	1/16	
Level 3			
**	9604/16384	9605/16384	1.46% (7.47 blocks)
A*	5096/16384	5096/16384	
*A	392/16384	392/16384	
AA	1292/16384	1292/16384	
Level 4			
	Not completed		1.79% (8.99 blocks)

Table I.1

Summary of Node-Level
Hand Analysis

Table I.2

Skip Distance Analysis for OEMGA Network

	Offset											
Skip ↓	0	1	2	3	4	5	6	7	8	9	10	Ave.
1	4	1	4	1	4	1	3	2	2	3	1	2.6
2	0	1	0	0	0	1	0	0	1	1	0	0.4
3	0	1	2	2	2	2	2	2	2	2	1	1.8
4	2	1	3	1	1	1	2	2	3	1	1	1.8
5	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0
* 7	1	2	2	3	1	1	2	1	3	1	1	1.8
* 8	0	1	2	2	2	2	2	2	2	2	1	1.8
* 9	0	1	0	0	0	1	0	0	1	1	0	0.4
10	3	2	3	2	1	4	1	4	0	4	0	2.4

*Assured by Symmetry

Table I.3

Skip Distance Analysis for BENES Network

	Offset											
Skip ↓	0	1	2	3	4	5	6	7	8	9	10	Ave.
1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	2	1	3	2	2	0	1	0	0	1	1.1
3	0	0	0	0	0	0	0	0	0	0	0	0
4	1	1	1	1	1	1	2	1	2	1	2	1.3
5	0	0	0	0	1	1	0	1	1	0	0	0.4
6	0	0	1	1	0	0	0	0	0	0	2	0.4
7	1	2	1	2	1	2	1	1	1	1	1	1.3
8	0	0	0	0	0	0	0	0	0	0	0	0
9	2	0	2	2	3	1	2	0	1	0	1	1.3
10	0	0	0	0	0	0	0	0	0	0	0	0

I.4 TOWARD A GENERAL ANALYSIS OF TRANSPOSITION NETWORKS

In its most abstract formulation, a system such as a transposition network can be described in terms of its states in a stochastic process. In an unpacked one layer Omega network, there are $n2^n$ switches, where n is the number of levels. Each switch can occupy one of nine possible states, two of which are blocking, and seven non-blocking. Much like a system of N pennies, which can take on 2^N different states, an n-leveled transposition network can take on 9^{n2n} states of which 7^{n2n} are non-blocking. These very large numbers give us every possible combination of switch configurations which the network can occupy.

One problem with such an analysis[2], is that not every state corresponds to a physically realizable configuration. In particular, there will be states for which no continuous path can be drawn.

One might then come to take the path rather than the state of a set of switches, as our unit of analysis. In an Omega network there is one and only one path by which any given input can reach any given output. (In a Benes, there are 2^n such paths, one for each switch from the middle level.) Now assume that there are 2^n inputs-- one for each switch -- and 2^n+ r outputs -- where r includes additional outputs plus one output corresponding to a null request. Then there are order 2^{2n} states in the sample space, each described by 2^n input-output pairs.

The problem then becomes one of obtaining the blocking probability for each of these states. This must involve the structure of the network itself. One can note, however, that blocking in an Omega is a function of input pairs, for on any level only two inputs may share the same switch. A mathematical algorithm for determining whether any given input pair results in a block is given in the following section, Part B. It is noted here that such an algorithm requires, at most, a comparison of each of 2^n input-output pairs for each of n levels. Thus, for order 2^{2n} states, there are order $n2^{4n}$ or $N^4\log_2 N$ comparisons that must be made to completely determine the blocking probabilities for all possible states.

This number may well be dishearteningly large for practical results, even if it need be done only once for simulation purposes. Says Benes: [1]

> In most congestion problems, it is easy enough to construct (say) a Markov process that is a probabilistic model of the system of interest. But it is dififcult, because of the large number of states and complexity of the structure, to obtain either analytic results or fast reliable procedures. This circumstance has been a major obstacle to rpgress in the congestion theory of large systems. One of its consequences has been that in some cases, models known to be poor representations of systems have been used merely because they were mathematically amenable, and no other tractable models were available. (pp. 1216-1217)

In another place he talks of possible "equivalence relations" between simple models and more complex ones.

The following is actually a model for determining the probability that x random assignement from N inputs to N outputs will be unique. The first input may choose any of the N outputs. The second input has an (N-1)/N probability of choosing one of the empty ones. The ith input has an (N-i+1)/N chance of choosing one of the empty ones. For x random assignements, the probability is

$$E(x \text{ in } N) = \frac{N!}{N^x(N-x)!}$$

that such a mapping will be unique.

The above formula is expected to be related to the probability of obtaining z successes across N ports in a packed Omega network. (This suspicion is based on the fact that there is one and only one path for each input-output pair.) For small x, this function is presumed to increase linearly, but for larger x and z z seems to increase more slowly than x. Qualitatively, unpacking the network corresponds to increasing N, which increases E(z in N). To find the expected number of successes in an equilibrium condition, set E(z in N) equal to 1/2 and solve for z. However, for a more exact and more complicated procedure for obtaining this result, see Section I.7.

I.5 PERMUTATION GROUPS AND PARTITION SETS

Bene's proof of the fact that a network of $2N\log_2N$ switches is sufficient to ensure the rearrangeability of N inputs to N outputs was published in 1964 [3]. This article draws heavily on group theory and the concept of the partition of the set (1, 2, 3, ..., N). The partition of a set is a finite collection of disjoint sets whose union is the given set.

A. Consider storing the Benes transposition network of 2n-1 levels as a matrix. (Storing the n-level Omega network is a special case of this.) On the first row, store the vector (0, 1, 2, ..., N-1). On the second row, s.ore the vector (0,2,1,3,4,6, ...,N-1), taking the first two even numbers, then the first two odd numbers, then the next two even numbers, until all N elements of the vector are stored. On the ith row, for i less than n, store the first 2^{i-1} even element, then the first 2^{i-1} odd elements and alternate until all the elements of the set (0, 1, ..., N-1) are used up. For the nth, and middle row, store the 2^{n-1} even elements, then the 2^{n-1} odd elements. (The first half of the N=16 Benes network is shown in Table I.5.) For row i between n and 2n-1, store just as the 2n-ith row. Now to compute the path that a given address would follow in the absence of other addresses, adopt the following procedure.

BENES (F12)

Skip	Offset	Successes	Options	Comments
2	3	251		
2	4	246		
3	4	512		Magic
3	5	512		Magic
5	6	413		
5	357	382		
8	357	139		
128	357	230		
128	357	266	BR	
256	357	233		
518	357	512		Magic
520	357	512		Magic

OMEGA (F13)

Skip	Offset	Successes	Options	Comments
--	--	199	R	Seed=0013
--	--	211	BR & 2	Seed=0013
1	0	32		Worst
1	1	36		
2	0	63		
3	0	85		
4	0	78		
13	357	111		
128	357	279		
128	357	259	BR	
210	357	307		
260	0	79		

Note: Standard deviation of N Count is $\sqrt{N}$

Table I.4

Simulation of Skip Distances

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	1	3	4	6	5	7	8	10	9	11	12	14	13	15
0	2	4	6	1	3	5	7	8	10	12	14	9	11	13	15
0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15

Table I.5

First Half of Stored Benes

Experiments on the CN Simulator confirm this hypothesis. However, it is not intuitively clear why this is the case, nor is a strict correspondence between offsets obvious. In part, the answer lies in the fact that to each skip distance there corresponds a cyclic ordered set of permutations on the outputs (0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5). This set is itself the correspondence set for skip 1.

For skip 5, the ordered set is (0, 10, 5, 8, 3, 6, 1, 4, 14, 2, 12). For skip 6, the ordered set is (0, 12, 2, 14, 4, 1, 6, 3, 8, 5, 10). These sets are the same, save that they are oppositely ordered).

A natural question which arises in this analysis is whether there are any skip distances which are particularly bad. Of course, the very worst case will be a skip of 521--which corresponds to a skip of zero--in which case all the processors will attempt to access the same memory module. Other than this, and this seems to be a rather important fact, the greatest number of blocks occurs in an 8 x 1^1 Omega for skip distances of one, especially those with small even offsets. Table I.2 verifies this. Furthermore, for all the trials which have run on the simulator, skip = 1, offset = 0 was the worst, with only 32 successes in 512 trials. The reason for this is as follows: the second level of an unpacked Omega will account for blocking of half the inputs if inputs from adjacent nodes wish to access the same quadrant of the network. Similarly, if adjacent nodes on the next level wish to access the same octant of the network, half this number will be blocked. This halving process, as the addresses are "funneled together", continues until they are half-way through the network, at which point they are "funneled back out" to their separate outputs. For odd offsets, the funneling process does not begin until the third level. For larger offsets, the mod521 configuration of the unpacked Omega tends to randomize the pattern.

It is not yet clear just what the overall relation between blockages and skip distances actually is; largely this problem is irrelevant. It could be solved empirically by running, say two hundred simulations picked from the skip distance range (0,260) for offsets (0,1) and plotting a curve. (Results from a few selected simulations are offered in Table I.4.) One would expect some kind of periodicity. But in fact, every such experiment which has been run for an Omega network has resulted in a success rate less than that for random requests (although for Benes networks skips 1 and 3 are "magic"), and significantly, it seems that the bit reversal procedure used for mapping (see Appendix B for details) is tantamount to a 'pseudo-randomization' of sorts. If this randomization is hardwired, no skip distance should be particulaly bad.

1. Initialize the input node, column j.
2. If the control bit for the ith level calls for "go straight", then the node for level i + 1 will be found on row i+1, column j.

If the control bit calls for a "go across" and
   a. j is even, then the node for level i+1 will be found on row i+1, column j+1, or
   b. j is odd, then the node for level i+1 will be found on row i+1, column j-1.

3. Let j=node number and increase i.
4. If i is less than 2n, go to 2.

The value of the node number for each i describes that path taken by the given address.

B. Suppose one wishes to know whether two given input-ouput pairs u-w and v-x result in a block for an Omega network. Let U be the set (0, 1, 2, ..., N-1). Let i be the level number for i=0, ..., n. Partition U sequentially into $2^n/2^{n-i}$, 2-by-2^{n-1} matrices. Call these $I_1^i, \dots, I_{2n/2i}$. These are the input matrices. Also partition U into $2^n/2^{n-i}$, 2^{n-i}-by-1 matrices. Call these $O_1^i, \dots, O^i_{2^n/2^{n-i}}$ These are the output matrices. Then for all (u, v, w, x), if there exists a j such that u is an element of I_j^i and v is an element of O_k^i and there exists a k such that w is an element of O_k^i and x is an element of O_k^i, then u-w blocks v-x by stage i. Symbolically, this condition can be written:

$$\forall x \forall w \forall v \forall u \exists i \exists j \exists k [u \in I_j^i \wedge v \in I_j^i \wedge w \in O_k^i \wedge x \in O_k^i]$$

The partition sets for n=4 are given in Figure I.4. For example, note that 4-8 blocks 7-11 since 4 and 7 are elements of I^2 and 8 and 11 are elements of O^2 . So 4-8 blocks 7-11 by level 2.

Note also that i is not unique but is satisfied for any i greater than some minimum i. To make i equal to this minimum i, require that u and v be from different columns of I_j^i.

This can be proven by considering, for the ith level, the ith most significant bit. If the ith most significant bit of the two inputs to any switch are the same, i.e.

XXXX...0_i...______ XXXX...0_i...______	or	XXXX...1_i...______ XXXX...1_i...______

where X's and ___'s represent bits that may assume any combination of 1's and 0's, then there will be a block. Bits which are more significant than i can occur in all possible combinations, but these bits determine which inputs the addresses could have come

i = 0 $\quad I_1^0=(0) \quad I_2^0=(1) \quad I_3^0=(2) \quad I_4^0=(3) \quad I_5^0=(4)$
$I_6^0=(5) \quad I_7^0=(6) \quad I_8^0=(7) \quad I_9^0=(8) \quad I_{10}^0=(9)$
$I_{11}^0=(10) \quad I_{12}^0=(11) \quad I_{13}^0=(12) \quad I_{14}^0=(13) \quad I_{15}^0=(14) \quad I_{16}^0=(15)$

$O_1^0 = (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14\ 15)$

i = 1 $\quad I_1^1=(0\ 1) \quad I_2^1=(2\ 3) \quad I_3^1=(4\ 5) \quad I_4^1=(6\ 7)$
$I_5^1=(8\ 9) \quad I_6^1=(10\ 11) \quad I_7^1=(12\ 13) \quad I_8^1=(14\ 15)$

$O_1^1 = (0,1,2,3,4,5,6,7) \quad O_2^1 = (8,9,10,11,12,13,14,15)$

i = 2
$$I_1^2=\begin{pmatrix}0 & 2\\ 1 & 3\end{pmatrix} \quad I_2^2=\begin{pmatrix}4 & 6\\ 5 & 7\end{pmatrix} \quad I_3^2=\begin{pmatrix}8 & 10\\ 9 & 11\end{pmatrix} \quad I_4^2=\begin{pmatrix}12 & 14\\ 13 & 15\end{pmatrix}$$

$O_1^2 = (0,1,2,3) \quad O_2^2 = (4,5,6,7) \quad O_3^2 = (8,9,10,11)$

$O_4^2 = (12,13,14,15)$

i = 3
$$I_1^3=\begin{bmatrix}0 & 4\\ 1 & 5\\ 2 & 6\\ 3 & 7\end{bmatrix} \qquad I_2^3=\begin{bmatrix}8 & 12\\ 9 & 13\\ 10 & 14\\ 11 & 15\end{bmatrix}$$

$O_1^3=(0,1) \quad O_2^3=(2,3) \quad O_3^3=(4,5) \quad O_4^3=(6,7)$

$O_5^3=(8,9) \quad O_6^3=(10,11) \quad O_7^3= (12,13) \quad O_8^3=(14\ 15)$

Figure I.4
Partition Sets for n=4

from. Thus a pairing between a set of output addresses and a set of input addresses can be formed. While this is not a formal proof, this result can be shown combinatorically by enumeration.

C. Suppose one wishes to know the mean probability that two random addresses will result in a block. This is a function of a relation between two inputs which is called their 'distance'. Consider input 0. There is a 1/2 probability that it will be blocked by input 1 since this blocking occurs on the first level. There is a 1/4 probability that it will be blocked by inputs 2 or 3; this would occur on the second level. For inputs 4, 5, 6 and 7 the probabilitity is 1/8 since the level is the third. In general, the distance for input 0 is the level on which the two inputs could block, so call it i. Then the probability that the inputs will block on level i is $(1/2)^i$.

Now assumedly there is a function g(x,y) of any two input numbers (x,y) such that i=g(x,y). Then taking the average value of the function $f(g(x,y))=(\frac{1}{2})g(x,y)$ the probability of a block is obtained. But for this to be a truly random distribution, one must average over both x and y as shown in Equation I.3.

$$\overline{f(g(x,y))} = \frac{\sum_x \sum_y f(g(x,y))\Delta x \Delta y}{\sum_x \sum_y \Delta x \Delta y} \qquad \text{(I.3)}$$

Now, for x=0,

x	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
g(x,y)	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4

For x=0, there will be in general 2^{z-1} values of g(x,y)=z.

Now pick some other random value of x, say x=4.

x	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
y	0	1	2	3	5	6	7	8	9	10	11	12	13	14	15
g(x,y)	3	3	3	3	1	2	2	4	4	4	4	4	4	4	4

so, again, there are 2^{z-1} values of g(x,y). We thus have a basis for a change of variables,

$$\overline{f(z)} = \frac{\sum_{z=1}^{n} f(z)2^{z-1}\Delta z}{\sum_{z=1}^{n} \Delta z} \qquad \text{(I.4)}$$

where z=g(x,y) and $\Delta x \Delta y = 2^{z-1}\Delta z$. Now f(z) just equals $(1/2)^z$; and the limits of the summation are from 1 to n, where n is the number of levels.

$$\overline{f(z)} = \frac{\sum_{z=1}^{n} (\frac{1}{2})^z 2^{z-1}\Delta z}{\sum_{z=1}^{n} 2^{z-1}\Delta z} = \frac{\sum_{z=1}^{n} \frac{1}{2}\Delta z}{\sum_{z=1}^{n} 2^{z-1}\Delta z} \qquad \text{(I.5)}$$

and since $\sum_{z=1}^{n} 2^{z-1}\Delta z = 2^{n-1}$, and $\sum_{z=1}^{n} \frac{1}{2}\Delta z = \frac{n}{2}$ (I.6)

then $\overline{f(z)} = \frac{n}{2(2^{n}-1)}$ (I.7)

which depends only on n. For n=9, $\overline{f(z)}$ = 9/1022.

I.6 TERM ANALYSIS FOR RANDOM BLOCKING

Consider a packed Omega. The blocking probability for two inputs (i,j) with random addresses is given by f(i,j) = $(1/2)^{g(x,y)}$. Thus a matrix can be made out of the f(i,j)'s. In general, the f(i,j)'s will either be 1/2, 1/4, 1/8, 1/16, or 0. In particular, if any input k has no request, then both f(k,j)=0 and f(i,k)=0. Also, f(i,i)=0.

Here notation will be changed so that it is more in line with symbolic logic and set theory. Let a_{iu}= f(i,j) and not-a_{ij}=1 -f(i,j). Now consider the prospects for adding one more input to the net. Inputs can be added from left to right to see when it becomes probable that the new input is blocked. The probability that input 0 is blocked by 0, a_{00}, is zero, of course. The first term will be the probability that 1 is blocked by 0, (i.e. a_{10}, which equals 1/2 for a packed and 0 for an unpacked Omega). The second term will be the probability that 2 is blocked by 0, but 1 is not, (i.e. $P(\alpha_{20} | \neg\alpha_{10}) = \alpha_{20} \neg \alpha_{10}$) which is 1/4 x 1/2 for a packed Omega. The third term will be probability that 1 blocks 2 given not-a_{20} and not-a_{10}, (i.e. $P(\alpha_{21} | \neg\alpha_{10} \wedge \neg\alpha_{20}) = \alpha_{21} \neg\alpha_{10} \neg\alpha_{20} = \frac{1}{4}\cdot\frac{1}{2}$
The next term will be

$$P(\alpha_{30} | \neg\alpha_{10} \wedge \neg\alpha_{20} \wedge \neg\alpha_{21})$$

and the term after that is

$$P(\alpha_{31} | \neg\alpha_{10} \wedge \neg\alpha_{20} \wedge \neg\alpha_{21} \wedge \neg\alpha_{30}).$$

In general, the kth term will be the product of k such atomic units, of which k-1 are negated. In the iterative procedure, one would have a 'tail' to the end of which the negated form of the last atomic unit is multiplied before multiplying by the new atomic unit obtained from the matrix. The kth term is then summed to the present value of the first k-1 terms.

When in the course of this procedure the current value of this summation becomes greater than 1/2, the procedure may be abandoned. The fact that the probability rises above 1/2 means that it is expected that this new element will be blocked. A new procedure is now adopted, in an attempt to find the probability that any two elements are blocked. The first two terms are zero. The third group of terms is $\alpha_{21}\alpha_{10}\neg\alpha_{20}+\alpha_{21}\alpha_{20}\neg\alpha_{10}$. The fourth group of terms will be $\alpha_{30}\alpha_{10}\neg\alpha_{21}\neg\alpha_{20}+\alpha_{30}\alpha_{20}\neg\alpha_{10}\neg\alpha_{21}+\alpha_{30}\alpha_{21}\neg\alpha_{10}\neg\alpha_{20}$. The fifth group will be $\alpha_{31}\alpha_{10}\neg\alpha_{20}\neg\alpha_{21}\neg\alpha_{30}+\alpha_{31}\alpha_{20}\neg\alpha_{10}\neg\alpha_{21}\neg\alpha_{30}+\alpha_{31}\alpha_{21}\neg\alpha_{10}\neg\alpha_{20}\neg\alpha_{30}+\alpha_{31}\alpha_{30}\neg\alpha_{10}\neg\alpha_{20}\neg\alpha_{21}$. In general, for the kth term group, there will be k-1 subgroups, each composed of k products. The k products will be given by the permuting of one addtional affirmative over the smallest k-1 matrix element, on one side of the diagonal. The smallest elements are defined by the fact that the left subscript must be less than or equal to that of the new, and the right subscript must be less than it. If the sum of these terms at any time is greater than 1/2, this procedure, too, is terminated and a procedure which tests for three blocks is implemented.

In general, in a procedure looking for i blocks, the kth group of terms will have $(k-1)!/(i-1)!(k-i)!$ subterms, each of which is a permutation of i-1 affirmative α's over k-1 α's. When i is large enough so that the whole network is done while the sum is less than 1/2, then this is just the expected blocking rate. Since there are on the order of $.5n^2$ groups of terms (for half the atomic coefficients in the square array, with $(k-1)!/(i-1)!(k-i)!$. subterms in each group, then there are at least $\frac{1}{2}N^3(N-1)!/(i-1)!(N-i)!(N-i)!$ operations in this procedure. For large problems there are many blocks, and i may be on the order of 100, making the computation even more prohibitive than that suggested in Section I.4.

However, there may be a "coarser" way to estimate the network blocking We have noted that there are k atomic elements in each of the $(k-1)!/(i-1)!(k-i)!$ subgroups. The minimum number of elements in any subgroup is i, for i blockages. For the kth group, each of these subgroups will be composed of i affirmative α's and k-i negative 's. Now the average value of one of these α's is as shown in Section I.5, $n/2(2^n-1)$ or $\log_2 N/2(N-1)$. Similarly, one could show that the average value of the function $1-(1/2)^{g(x,y)}$ is $1-(\log_2 N/2(N-1))$. (Assume that this average value of each of the terms found in this way will be good estimators for the product. Basically this average says that the typical block will occur at the $\log_2(2(N-1)/n)$th level. For n=9, this is about 6. In this way, the computation can be drastically reduced.) Each of these groups can be written as a product of one of these estimator terms times the number of such terms. And since there are such groups for all k from i to N, we are left with the sum

$$E(i \text{ in } N)=\sum_{k=i}^{N}\frac{(k-1)!}{(i-1)!(k-i)!}\left(\frac{\log_2 N}{2(N-1)}\right)^{k-i}\left(1-\frac{\log_2 N}{2(N-1)}\right)^{i}$$

which depends only on i and N. The largest such term occurs for k=i, and is just $(1-(\log_2 N/2(N-1))^i$. (Since E(i in N) is greater than its first term, a good way to make a lower limit approximation for i is to use the least i such that this term is less than 1/2. For N=512, this is just $(1013/1022)^i$.) The last and smallest term in the series for k=N, which we call E_{min}(i in N) may be written

$$E_{min}\,(i\ in\ N) = \left(1-\frac{\log_2 N}{2(N-1)}\right)\left(\frac{\log_2 N}{2}\right)^{N-i}\left[\frac{(N-1)!}{(N-i)!\,(i-1)!\,(N-1)^{N-i}}\right]$$

Note the resemblance between the part in brackets and the formula in Section I.4, with N-i corresponding to x. One major difference between the two is that the 'i' in the former is the number of blocks, while the 'x' in the latter is the number of successes.

I.7 STATE OF THE CONNECTION NETWORK

One of the networks proposed is the two-layer unpacked Omega with bit reversal and alternating priorities between layers and cycles. In fact, it is suspected that a hardwired processor-to-input randomization would work as well as a bit reversal. Any priority rule that favors the left port will favor addresses going to the left side of the network, and vice versa. However, a random priority rule, where the priority is determined by a random number from 1 to 4 (favoring left, right, straight-through, and crossed) would probably be optimal. One way to improve the priority rules is to add a bit to the address which says: "I am a success so far." Then if there is a conflict, and if one or the other of the addresses has say a 1 in this place, then the switch will give that addresss the priority.

The Benes network now appears suboptimal. In the absence of overall control, an algorithm must be developed which produces an address from the first half of the network. The algorithm studied obtained the address through an "exclusive or" on the processor and memory module numbers. This algorithm has a serious flaw in it for any unpacked Benes. As long as the ports are unpacked, both processor and EM numbers are even--except for the nine odd-valued memory module numbers. The fact that the least significant bit is zero implies that the addresses go straight through on the first level; this in turn implies that at the middle level of the network all the addresses are in the left half of the switches. Thus, at the middle layer, the addresses are 're-packed'. Also, at every level prior to the middle one, only half the switches are used. Needless to say, this seriously degrades the simulation of any unpacked Benes. However, this problem should be rectifiable with a bit reversal.

Part of this analysis considered various alternative network organizations in case additional throughput is required. One way to improve performance is to 'double-unpack' the inputs, so that there is only one address for every four ports. This implies, as well, adding another level to the network. To compare the effect of doubling the number of ports with that of adding another layer, a packed two-layer Omega and a packed one-layer Omega with 256 requests were simulated. If the simulation is to scale properly, one would expect to find half as many successes with 256 inputs to 512 ports as with 512 inputs to 1024 ports. In simulation, the packed two-layered Omega with 256 inputs produced 70 successes. Thus in performance a double-unpacking appears equivalent to doubling the number of layers. This further suggests that for the first cycle the success rate depends only on the number of input ports for an Omega network.

Still probably the best way to reduce blockages experienced in the networks studied is to add more layers. As was previously noted, it is deceptive to think in terms of "Layers". Any n-layered network of 2 x 2 switches can actually be represented as one layer of 2n x 2n switches. When Benes proved his theorem, he did so for any Benes network of square switches, i.e., n x n. Now for an Omega Network, the address generates the path at the switch by a simple procedure of left-right responses. And while it may be difficult to construct a local control procedure f̄or binary addresses or odd-valued n x n switches, equivalences can be set up between n inputs and n outputs for a 2n x 2n switch. The resulting logic at each switch would be more complicated in this case.

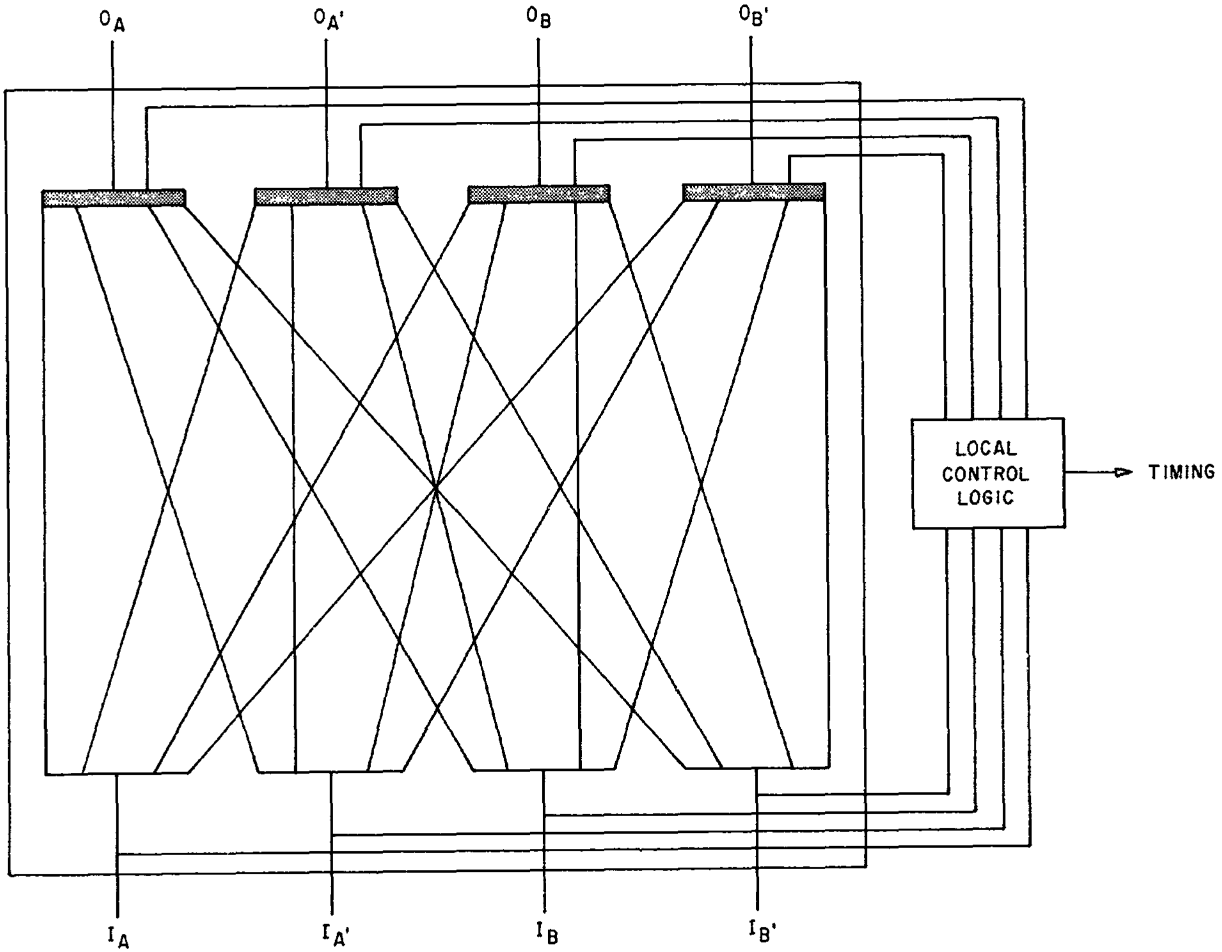


Figure I.5

A 4x4 Switch
(Note: Each wire is 12-16 wires)

References

(1) Lawrie, Duncan Hamish, "Memory-Processor Connection Networks", February 1973.

(2) Benes, V. E., "Heuristic Remarks and Mathematical Problems Regarding the Theory of Connecting Systems", Bell System Technical Journal, 41, (1962), pp. 1202-1247.

(3) Benes, V. E., "Permutation Groups, Complexes, and Rearrangeable Connecting Network", B.S.T.J., 43, (July 1964), pp. 1619-1640.

(4) Benes, V. E., "A 'Thermodynamic' Theory of Traffic in Connecting Networks", B.S.T.J., 42 (1963), pp. 567-607.

APPENDIX J

DESIGN GUIDELINES FOR NASF PROCESSING SYSTEM

J.1 SCOPE

This document delineates general guidelines that may be used in the design, fabrication and assembly of the hardware required for the Numerical Aerodynamic Simulation Facility (NASF) Processing System.

J.2 DESIGN CONTRAINTS

J.2.1 Environmental

The environmental limits specified represent the conditions normally found in most laboratory or office buildings deemed suitable for professional employees and assumes that air conditioning and other controls have been provided to attain these levels. It is incumbant on the design of the FMP hardware not to adversely affect this environment.

J.2.1.1 Atmospheric Conditions

Table J.1 defines the limits of temperature, humidity, and altitude for operating, non-operating, storage and shipping conditions.

Dust levels may exist to the extent resulting from a filtered air conditioning system meeting NBS blackness test with a minimum rating of 50% efficiency using atmospheric dust.

J.2.1.2 Mechanical Stress

Table J.2 delineates the mechanical stress levels for the equipment installed (operating and non-operating) and in shipping containers.

Shock is defined as a non-periodic mechanical pulse of large amplitude about a fixed point.

Vibration is a steady state periodic or random oscillation which may have a sinusoidal or a complex waveform and may have a single frequency or broad spectrum.

J.2.1.3 Acoustic Noise

The equipment should not be affected by exposure to sound pressures of 130 dB* (c) for a period of 30 minutes.

* Ref. 2 x 10^4 dynes/cm^2

TABLE J.1 ATMOSPHERIC CONDITIONS

	OPERATING	NON-OPERATING	SHIPPING AND STORAGE
	(Installed)	(Installed)	
DRY BULB TEMP.	18°C to 30°C	-40°c to 50°c	-40°c to 70°c
WET BULB TEMP.	NA	30°c Max	40°c Max.
RELATIVE HUMIDITY	40% to 60%	90% Max.	95% Max.
ALTITUDE	0-3 km	0-3 km	0-15 km

TABLE J.2 MECHANICAL STRESS

	Operating and Non-Operating	Shipping and Storage
	(Installed)	(In Shipping container)
SHOCK		
Peak Acceleration	.5 g	5g
Duration	.1 to 1 sec.	5 to 50 millisec.
Waveshape	½ sine	½ sine
Force application	Horizontal	3 Orthogonal axes
VIBRATION		
Frequency Range	5 to 500 Hz	5 to 500 Hz
Peak Acceleration	.1 g	1.5 g
Force Application	3 Orthogonal axes	3 Orthogonal axes

J.2.1.4 Radiation

The equipment should not be affected by radiation of the following intensities

(1) Stray magnetic fields	.0005 tesla
(2) External RFI	1.0 Volt/Meter 500 kHz to 10GHz

J2.1.5 Static Electricity

Externally exposed hardware should be immune to static electric discharges of up to 10 kilovolts from 500 pF through 50 ohms.

J.2.1.6 Fungus

Fungus inert parts and materials should be used to the greatest extent possible. Parts or materials not inert to fungus growth should be treated with fungicidal material. No damage to parts or material should result from treatment with fungicidal material or fungicidal coating.

J.2.2 Electromagnetic Interference Control

The NASF equipment should be compatible with: a) other electronic devices operating in the immediate area, and b) communications services. Control of the electromagnetic emanations from the NASF equipment must be an integral part of overall system design.
Based on the nature of the NASF design and mission, conducted and radiated emanations shall comply with the limits illustrated on Figures J.1 and J.2 respectively.

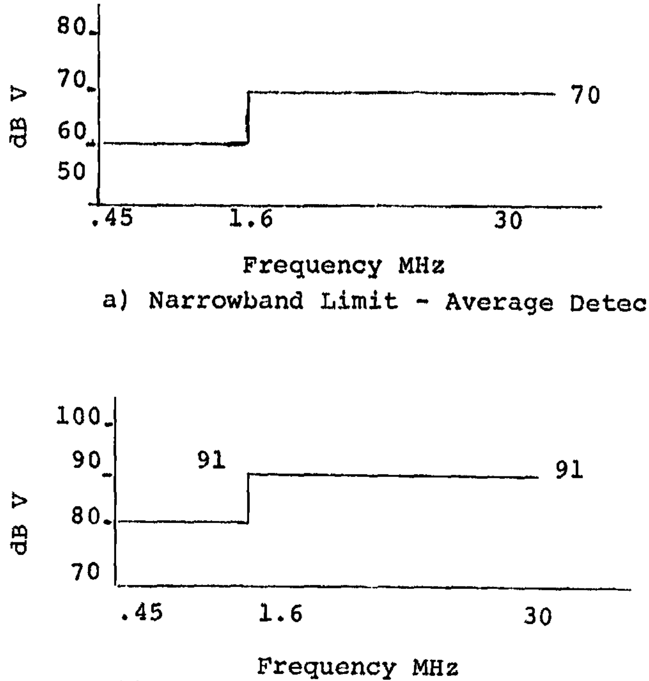
J.2.3 Acoustic Noise Control

Personnel should be provided an acoustical environment which will not interfere with, or in any way degrade overall NASF effectiveness. To ensure compliance with this requirement, acoustic noise levels of the NASF Processing System should not exceed the following criteria:

EQUIPMENT AREAS	75 dB (A) * 68 dB SIL **
OPERATOR AREAS	65 dB (A) 58 dB SIL
I/O AREAS (ELECTRO MECHANICAL DEVICES)	80 dB (A) 71 dB SIL

* dB(A): Measurement using (A) weighting network on Sound Level Meter.

** dBSIL: Speech Interference Level,- The arithmetic average of the sound-pressure levels in the octave bands centered on 500, 1000, and 2000 Hz.



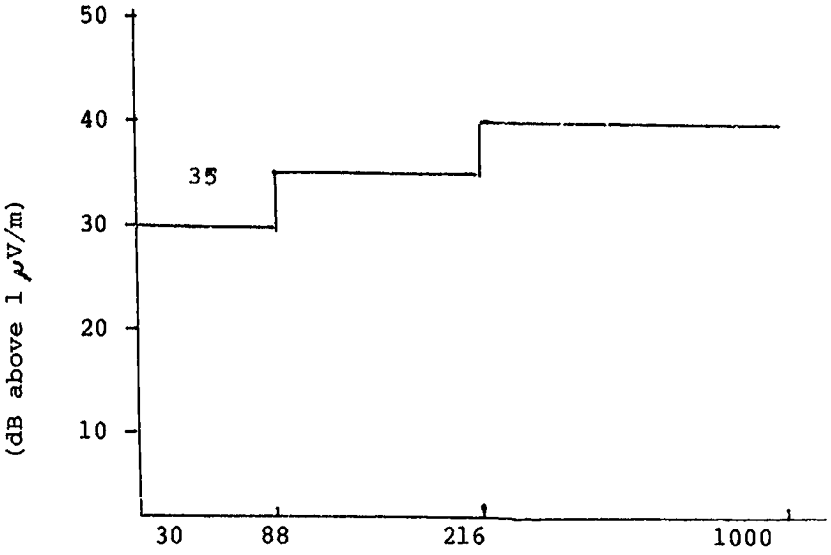
a) Narrowband Limit - Average Detector

b) Broadband Limit - Quasipeak Detector

* AVERAGE DETECTOR: A detector, the output voltage of which approximates the time average value of the envelope of an applied signal. Refer to ANSI C63.2-197X

** QUASIPEAK DETECTOR: A detector having specified electrical time constants, which, when regularly repeated pulses of constant amplitude are applied to it, delivers an output voltage which is a fraction of the peak value of the pulses, the fraction increasing towards unity as the pulse repetition rate is increased. Refer to CISPR Publications, 1, 2, and 4.

Figure J.1 Conducted Limits



Frequency - MHz

Broadband Limit - Quasipeak Detector
Narrowband Limit - Quasipeak Detector

Figure J.2 Radiated Limits (30 Meters)

J.2.4 Input Power

Table J.3 and J.4 delineate the minimum quality level of the power that should be available for the NASF Processing System. The Processing System should have its own power control and distribution subsystem that will operate from this input power and supply the appropriate power to the various hardware elements of the processing system.

J.2.5 Design and Construction

Unless otherwise specified, the NASF Hardware should be designed in accordance with good commercial practices.

J.2.5.1 Physical Characteristics

J.2.5.1.1 Cabinets - Removable panels and doors should be utilized to enclose the structure.

J.2.5.1.2 Size and Weight - No single unit, cabinet or component should exceed 3,600 pounds or exceed the following dimensions:

Height 72"
Width 84"
Depth 35"

The floor loading should be no more than 250 lbs/ft^2 for fully operable equipment.

J.2.5.1.3 Marking

J.2.5.1.3.1 Marking of Equipment - Each major assembly should be permanently and legibly marked with the manufacturer's identification (name, initials, trademark, code number, or symbol) serial number, and model number. Permission shall be granted to the manufacturer to place its name/symbol on the front of the equipment.

J.2.5.1.3.2 Marking of Controls - Controls related to the operation or conditioning of the equipment, either remotely or locally, should be clearly identified.

J.2.5.1.3.3 Marking of Subassemblies - All removable and repairable plug-in subassemblies should be identified and marked with a serial number. Labels should be positioned so they can be readily seen.

Table J.3
Power Source Description

Total Power: 750 KVA

Frequency: 60 Hz
Tolerance: ± 3%
Rate of Change: 1.5 Hz/Sec Max

Voltage: 480 3 Phase, 3 Wire plus ground
Range of slow-averaged + 10%, -15%
rms voltage (including
brown outs)

Imbulance 5% Max
Modulation 1% Max
Harmonics (total) 20% Max
Max Any Harmonic 10% Max
Deviation Factor 25% Max
D. C. Component 1% Max

Table J.4
Power Source Transients, Recovery and Capability

	LEVEL	REMARKS
*Power Source Transients (on any or all phases) 1/2 Cycle or Longer Maximum Transient Surge	To 130% of nominal rms voltage recovering to 120% in 50ms or less, then within 110% in 3 sec. or less	*Voltage deviations shall be within the limits shown when the utilization voltage is within its tolerance limits (+10%, -15%)
Maximum Transient Sag	To 50% of nominal rms voltage, recovering to 70% in 100ms or less, then to 85% or more in 0.5 sec. or less	
Less Than 1/2 Cycle But More than 100 microseconds Maximum Transient surge peak volts	150% of nominal peak voltage (212% of monial rms voltage) provided than volt-second limit i:. not exceeded	Surge component only, would be 250 . total if occuring at wave peak
Maximum Overvoltage Transient volt-seconds	150% of nominal volt-seconds provided surve voltage limit above is not exceeded	Composite wave form
Maximum Transient sag peak	To zero volts	
Maximum Undervoltage Transient volt-seconds	To zero for 1/2 cycle	
Impulses (either polarity 100 usecs. or less, RFI Bursts		
Maximum Voltage Deviation of RFI. 10kHz or greater	400% of nominal peak voltage (566% of nominal rms voltage)	Impulse component only, would be 500% if occuring at wave peak
Event Rate	Maximum of 10 in 10 minutes and at least 6 seconds between maximum limit events and full recovery to specified range of rms voltage between event.	Starting condition for all events shall be within specified range of rms voltage and may be at worst case to maximize the event.
Power Source Capability		
Peak Inrush Limit	4kVA or 3 to 8 x rated kVA of load	
Load Imbalance	25% max or 10kVA whichever is greater	
Source Impedance	0.5% to 5% of Rated "Base" ohms at the power frequency	
Ground Return Impedance	Impedance low enough to create ground fault current of 10 x breaker trip rating for ground fault	

J.2.5.1.4 Accessibility - All adjustments and any other work required after the system is assembled should be readily accessible for servicing. Special tools or other mechanical devices required to readily and accurately adjust the equipment should be supplied as part of the unit. The equipment should be designed to protect operating and maintenance personnel from contact with hazardous devices. The equipment may, however, be designed to operate with the frame covers removed (but not necessarily meet acoustic and EMI requirements under these conditions).

J.2.5.1.5 Grounding - Circuit grounds should be isloated from chassis grounds but may be connected to the chassis if required. Ground connections to chassis should be mechanically secured by soldered terminals and locked by means of a lockwasher and nut. A chassis ground tiepoint should be provided at the power interface of individual cabinets and major elements of the system.

J.2.5.1.6 Mechanical Operation - All controls should operate freely and smoothly without binding, scraping, or cutting. Play and backlash should be minimized and should not cause poor contact or inaccurate setting.

J.2.5.1.7 Transportability - The elements of the NASF Computing System should be transportable by qualified domestic common carrier without damage or deterioration when packaged, preserved, and prepared for shipment.

J.2.5.2 Materials, Processes, and Parts

J.2.5.2.1 Parts Selection - The following principles should be utilized in parts selection:

(a) The variety and types of parts required should be kept to a minimum.

(b) Common and regularly stocked parts should be used whenever feasible to simplify maintenance, storage and supply.

(c) The use of proprietary components should be avoided where practicable.

J.2.5.3 Workmanship - The equipment should be processed in such a manner as to be uniform in quality and free from defects that will adversely affect life, serviceability, and appearance. All metal surfaces should be clean and free from burrs, roughness, oxide, scales, and sharp edges. Printed circuit boards should be free from cold soldering, corrosion, salts, smut, grease, finger prints, flux residue, and foreign materials.

J.2.6 Product Safety

The NASF equipment should be designed and constructed so that in normal operation and maintenance the equipment will function reliably without causing injuries to persons or damage to property, considering possible careless use that may occur in normal service. Specifically, the equipment should be designed to comply with the requirements of Underwriters Laboratories (UL) Standard for Safety, Data Processing Units and Systems, UL 478.

J.2.7 Service Life

The intended life of the system as a system is ten years. Service life is the anticipated life of the system, as a system, without reference to the anticipated useful life of the parts of the system and, therefore, assumes that necessary maintenance and repairs will be performed as required.