(1)

N74-17909

DESIGN OF A FAULT TOLERANT AIRBORNE DIGITAL COMPUTER, VOLUME I - ARCHITECTURE

J. H. Wensley, et al

Stanford Research Institute
Menlo Park, California

October 1973

COST ITEM $ 7.00

NATL AERONAUTICS AND SPACE ADM ; NASA-CR-132252

DISTRIBUTED BY:

NTIS
National Technical Information Service
U. S. DEPARTMENT OF COMMERCE
5285 Port Royal Road, Springfield Va. 22151

LM123822E

M75-12723

Final Report

DESIGN OF A FAULT TOLERANT AIRBORNE DIGITAL COMPUTER

Volume I — Architecture

By: J. H. WENSLEY, K. N. LEVITT, M. W. GREEN, J. GOLDBERG, and P. G. NEUMANN

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
LANGLEY RESEARCH CENTER
HAMPTON, VIRGINIA 23365

NICHOLAS MURRAY — CONTRACT MONITOR

CONTRACT NAS1-10920

```
(NASA-CR-132252)  DESIGN OF A FAULT                     N74-17909
TOLERANT AIRBORNE DIGITAL COMPUTER.
VOLUME 1:  ARCHITECTURE  Final Report
(Stanford Research Inst.)                               Unclas
                                 CSCL 09B   G3/08      16207
```

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
MENLO PARK, CALIFORNIA 94025
(415) 326-6200

ERRATA

December 10, 1973

To: Distribution

Reference: Final Report
"Design of A Fault Tolerant Airborne Digital Computer Volume I Architecture"
"Design of a Fault Tolerant Airborne Digital Computer Volume II Computational Requirements and Technology"

SRI Project Number 1406

Prepared for: National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23365

Eclosed is errata labal to add to your cover of volumes one and two of Final Report.

Volume I: 132252
Volume II: 132253

Thank you,

[illegible]

EDWARD E. COTTON
Sr. Clerical Assistant

eec
Encls.
1406

1a

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

Final Report

October 1973

DESIGN OF A FAULT TOLERANT AIRBORNE DIGITAL COMPUTER

Volume I — Architecture

By: J. H. WENSLEY, K. N. LEVITT, M. W. GREEN, J. GOLDBERG, and P. G. NEUMANN

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
LANGLEY RESEARCH CENTER
HAMPTON, VIRGINIA 23365

NICHOLAS MURRAY – CONTRACT MONITOR

CONTRACT NAS1-10920

SRI Project 1406

Approved by:

D. R. BROWN, *Director*
Information Science Laboratory

BONNAR COX, *Executive Director*
Information Science and Engineering Division

ib

ABSTRACT

Volume I of this report is concerned with the architecture of a fault tolerant digital computer for an advanced commercial aircraft. All of the computations of the aircraft, including those presently carried out by analogue techniques, are to be carried out in this digital computer. Among the important qualities of the computer are the following: (1) the capacity is to be matched to the aircraft environment, (2) the reliability is to be selectively matched to the criticality and deadline requirements of each of the computations, (3) the system is to be readily expandable and contractible and (4) the design is to appropriate to post 1975 technology. Three candidate architectures are discussed and assessed in terms of the above qualities. Of the three candidates, a newly conceived architecture, Software Implemented Fault Tolerance (SIFT), provides the best match to the above qualities. In addition SIFT is particularly simple and believable. The other candidates, Bus Checker System (BUCS), also newly conceived in this project, and the Hopkins multiprocessor are potentially more efficient than SIFT in the use of redundancy, but otherwise are not as attractive. Volume II of the report is concerned with a detailed description and categorization of the computations and with a discussion of the technology available for realizing the computer system.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

PREFACE

This report, issued in two volumes, summarizes the work of Stanford Research Institute on Contract NAS1-10920. The goal of the contract was to specify the design of a computer, destined for use as the central computer in an advanced, high-performance commerical aircraft. Because of the critical nature of many of the computations, fault tolerance was the primary design goal of the computer. Other important design goals of the computer relate to

- The matching of the architecture to the aircraft computations
- The capability for expansion or contraction to meet the requirements of various missions
- The suitability to post 1975 technology.

Volume I is concerned with the architecture of fault tolerant computers, that are matched to the aircraft environment. We selected and studied three candidate architectures as part of Task I of the contract. Two of these architectures, Software Implemented Fault Tolerance (SIFT) and Bus Checker System (BUCS) are new and as such are described in detail. The third candidate architecture is a multiprocessor concept that is due to Al Hopkins of MIT-Draper Laboratories. We are aware of the extensive work that has been devoted to fault tolerant techniques and architectures over the past decade. However, a survey of this work has pointed out significant deficiencies in each architecture, for our particular constraints. For the most well-known of these previously studied architectures we document the deficiencies.

Volume II of the report is organized as two parts. Part 1 is concerned with the computational requirements of an aircraft, wherein it is assumed that all of the computations scattered among special purpose analogue and mechanical computers would be carried out by a centralized digital computer. In addition to the usual computations associated with a commercial aircraft, e.g. navigation, stability augmentation, decrab, we also assume advanced cockpit displays and fly-by-wire. These various computations are categorized according to criticality, and for each computation we derive such parameters as memory requirements, processor requirements, iteration rates, the tolerable down time and the amount of data that must be saved in the event of failure. These results are concisely tabulated.

Part 2 of Volume II is concerned with the technology for realizing the central computer. It is assumed that production would commence in the late 1970's. The two aspects of the realization that we consider are concerned with logic and memory and with module interconnections. With regard to logic and memory we assess the various technologies, MOS, CMOS, BIPOLAR, etc., as a function of requirements of speed, reliability, cost, number of units. In addition we discuss such realizations as customized large scale integrated (LSI), medium scale integrated (MSI), programmable logic in the light of the above requirements. With regard to interconnection technology the primary goal is to prevent the propagation of faults beyond some predetermined module boundaries. Various approaches toward achieving this fault confinement are assessed in terms of speed, cost, reliability.

We would like to acknowledge the support of Nick Murray and his colleagues at NASA-Langley--Sal Bavuso, Larry Spencer, Bill Dove, and Brian Lupton. They interacted with us on all phases of the project and provided valuable guidance. On the computation aspects many aircraft and avionic specialists provided us with detailed descriptions of algorithms as well as experience on the conversion of analogue algorithms to a digital representation. With regard to architecture, we have had stimulating discussions with Al Hopkins, Al Avizienis, Bill Carter, Bill Martin, Barry Borgerson and Jim Miller. Many of their ideas are reflected in our candidate architectures.

I INTRODUCTION

A. Purpose of the Study

The purpose of this study is to aid NASA in specifying the design of a computer, destined for use as the central computer in an advanced, high-performance commercial aircraft. This computer, or more realistically, computer complex, is to

(1) be responsible for all of the aircraft computations currently being carried out by analogue, mechanical or dedicated, special purpose digital computers,

(2) be capable of carrying out computations associated with an advanced aircraft of the type similar to the proposed Advanced Technology Transport (ATT),

(3) exhibit sufficiently high reliability such that the probability of a computer failure adversely effecting the flight is negligible as compared with other system failures. In particular, the design goal is 10^{-8} failures/hour for the flight critical function. For computations that are not flight critical higher failure rates can be tolerated.

(4) be readily expandable and contractible so as to meet the needs of various missions,

(5) be matched to the post 1975 component technology.

The present study is composed of three tasks, as follows:

Computational Requirements: The purpose of this task is to survey all of the computations being carried out in contemporary commercial aircraft (e.g. navigation, autopilot, autoland, control of cabin pressure) and in the projected ATT (e.g. advanced stability augmentation, collision avoidance). The survey is to reveal aspects of these computations that influence the architecture of the computer including, word size, memory requirements, reliability, recovery time from a failure. Volume II of this report describes, in detail, the algorithms for each computation and extracts from these algorithms some of the crucial parameters for the individual computations. Chapter III of this volume summarizes the information and presents a global view of the implications of the computations on the computer aircraft.

Technology: The purpose of this task is to survey the technology associated with the various system blocks of the computer. If this computer is to be produced in large quantities, production will commence in the late 1970's. Volume II of this report discusses in detail the prognoses for the following technologies: logic, main read-write memory, read-only memory, buffer-type memory, back-up memory, and interconnections. For each of these technologies the report discusses speed, reliability and cost.

Architecture: The purpose of this task is to specify the architecture of three candidate computers, each of which can match the computational and reliability requirements, and the technology constraints revealed in the other two tasks. The hardware and software systems of the candidates are to be specified to a level such that the overall reliability can be assessed. In essence then, each candidate is to be specified in terms of a concept, wherein detailed design and analysis is to be avoided unless required for the reliability assessment. For example, a detailed description is needed of the process of recovering from a suspected transient fault, but a detailed description of the arithmetic unit is not needed unless it embodies some particular fault tolerance scheme. This volume summarizes our work on the architecture task.

B. Organization of the Report

Chapter II briefly reviews the reliability enhancement techniques that have been proposed and analyzed during the past 15 years. Our review is purposely terse since most of these techniques are, by now, well-known. The intent is to delineate a set of concepts that are used in the three architectures.

Chapter III is concerned with the implications of the computational requirements task. Here we summarize the results of the computation survey and convert the results into computer parameters: e.g., word size, I/O bandwidth, fault recovery time, multiprogramming characteristics.

Chapter IV discusses constraints on the architecture as revealed by the technology task. For example, we discuss the impact of LSI technology on approaches toward applying redundancy and on the desirability of using slow memories to aid in system recovery. A primary feature of the technology task is a review of interconnection technology, at all system levels (chip-chip, card-card, subsystem-subsystem), and the impact of various approaches to interconnection on the problem of fault isolation.

Chapter V is a "checklist" for the design of a fault-tolerant computer, for the particular aircraft environment that we are confronted with. We summarize here, for example, the fault types that must be accounted for, the various alternative approaches to system design, the components of the reliability analysis, and in general the details to look for in the design and analysis of the computer.

Perhaps the simplest architectural concept that can meet the reliability requirements involves the use of three or more complete computers operating in a locked-step manner. We call such an approach the _multi-channel concept_ (MCC). The basic reliability technique is trivial here--a simple voting or adaptive voting suffices. However, there are still important design decisions required for the multi-channel concept, in particular relating to the operating system procedures for transient fault recovery, and to communication among the computer units. However, we decided not to pursue this concept, for the following reasons

- NASA Langley is in the process of establishing a detailed design and implementation effort based on the MCC.
- Comparable reliability performance can be achieved with a system that consumes less hardware redundancy. Such a cheaper system takes advantage of the possibilities for allocating sub-tasks to independent processor units, and the possibilities for using less costly coding techniques in memory.

- All of the interesting software problems attendant to the MCC are confronted in practically all types of fault tolerant architectures.

In the second half of Chapter V we survey existing architectures that have been suggested for various fault tolerant applications. It is clear that NASA-Langley should pursue an existing concept provided it is matched to the aircraft requirements. The systems that we surveyed are the following:

- Self-Testing and Repairing Computer (STAR) of JPL.
- All Application Digital Computer (AADC) of the Naval Air Systems Command.
- Experimental Aerospace Multiprocessor (EXAM) of NASA-Electronics Research Center.
- Automatically Reconfigurable Modular Multiprocessor System (ARMMS) of NASA-Marshall Space Flight Center and of Hughes.
- Modular Architecture for Reliable Computer Systems (MARCS) of IBM-Yorktown.
- A Fault-Tolerant Information Processing System for Advanced Control Guidance and Navigation (Hopkins' Multiprocessor) of MIT Draper Laboratory.

This chapter will present detailed information on each of these systems, but for purposes of summary we feel that only the Hopkins' Multiprocessor should be considered as a candidate architecture, although extensive design and analysis must be carried out to ensure that the concept is suitable to our environment. To be fair to the architects of some of the other systems we should state that their concepts have not yet confronted all of the reliability questions. Hence most of the concepts can be adapted to achieve fault tolerance although this design process would probably entail as much work as developing a new system from start, and may not produce a better machine than one designed explicitly for the environment we have in mind.

Chapter VI presents a discussion of the Software Implemented Fault Tolerance Concept (SIFT), one of two candidate architectures that we designed anew. We recommend that SIFT be selected for pursual in further design phases since it meets all of our reliability, computational, and cost objectives, and, moreover, it is likely to be the least costly in the construction of a prototype. This attractiveness in cost is due to the following factors:

(1) The fault detection and recovery processes are carried out by software (although firmware or special hardware would also suffice).

(2) All units could be simple "off-the-shelf" computers, or be realized from existing designs.

(3) Very little ancillary hardware, (the only exception being a bus structure) is required to implement the concept.

(4) The architecture is well-suited to varying reliability requirements among the computations.

Hence, the SIFT concept could be experimentally evaluated by programming special executive routines on 3 or 4 minicomputers.

Chapter VII presents a discussion of another, newly conceived candidate architecture, denoted as the Bus Checker System (BUCS). BUCS was inspired by the architectures covered in our survey.

As in SIFT, processing units that are nearly conventional suffice as the primary computing elements. However, the memories utilize coding techniques to enhance reliability at a relatively low cost, and a "smart" bus acts in concert with a software executive to detect faults and reconfigure the system.

Chapter VIII summarizes the characteristics of the three candidate architectures and presents our conclusions and recommendations.

Appendix A describes the detailed design of an algorithm for SIFT that allocates computational tasks to processors. Appendix B presents a detailed comparison of the SIFT architecture with the multi-channel concept, in terms of reliability and redundancy.

II REVIEW OF RELIABILITY TECHNIQUES

A. Introduction

In this chapter we review the various redundancy techniques for improving digital system reliability that have been proposed during the past 15 years. Our review is intentionally brief since

(1) There have been no fundamental breakthroughs since our last survey[1] of the field six years ago.

(2) Most of the redundancy techniques that are theoretically interesting are only applicable at the component level. The constraints imposed by the emerging LSI technology (see Chapter IV) are such that redundancy should be applied over chips, not within the components of chips. Hence relatively few redundancy techniques remain relevant, and moreover system architecture considerations are now of primary interest.

(3) The underlying concepts are relatively simple.

We have concluded that architectural considerations are of primary importance in the design of a reliable digital computer, as compared with so-called low-level redundancy techniques. For example, workable strategies for recovering from transient faults, or the protection of programs and data in the event of a fault are less understood than abstract redundancy techniques. However, some of these redundancy techniques, originally intended for low-level application, form the basis for enhancing system reliability even when applied at a high level.

The attainment of high reliability in a system requires that

(1) Faults* be detected subsequent to their occurrence.

(2) Errors produced by these faults be masked or the faulty unit should be disconnected and be replaced by an operational one.

*The following terminology has become more or less standard in the reliability field. A fault is the actual malfunctioning of an element. An error is the appearance of incorrect data, on some data line, as a result of a fault.

Two techniques have been seriously proposed to accomplish fault detection, namely:

(1) The data lines of the system are encoded so that under fault-free conditions the signals on the data lines form a code word in an error detecting/correcting code. The occurrence of any fault, within a detectable class, is to introduce an error such that a non-code word appears on the data lines. (Duplication is an instance of a trivial error-detecting code.)

(2) All system blocks are subject to periodic diagnosis in order to determine if a fault has occurred since the last attempt at diagnosis. There are important system questions concerned with such high-level diagnosis, in particular how to account for faulty diagnosing units.[2,3]

A third possible technique involves the use of a checking program that can carry out consistency checks on another program. This technique is not studied here because it is clearly dependent on the particular application programs.

For our application the use of periodic diagnosis is not recommended as the <u>primary</u> fault detection process since

(a) transient faults are not detected nor prevented from causing data loss by periodic diagnosis,

(b) the effectiveness of diagnostic routines remains suspect, given the present state of the art,

(c) the real-time nature of most of the computations precludes the possibility of buffering computed results pending the results of a diagnostic test,

(d) diagnostic test schedules derived by contemporary systematic approaches usually guarantee detection of only single or double gate type failures. However, in LSI chips more complicated fault behavior seems to be possible including multiple gate faults, and shorts in interconnections.

The diagnostic approach to fault detection does have a role in certain system functions. As we demonstrate later a memory system can be effectively diagnosed provided other auxiliary error-detection techniques are used. Moreover a form of diagnosis is essential in initializing the computer system, and in effecting a recovery from a "massive transient" fault.

One notes that technique (1) incorporates only spatial redundancy while technique (2) incorporates primarily temporal redundancy. That is in the diagnosis approach certain time periods that otherwise would be devoted to useful computation are set aside for purposes of detecting faults. It might be fruitful to explore approaches that combine the attractive features of the spatial and temporal approaches.

Once a fault has been detected, and pinpointed, it remains to apply some form of corrective action. When error correcting codes are used, the decoder can produce at its output the intended code word, thus correcting the effects of the fault. Another approach is to utilize the error detecting possibilities of the code to point to a faulty unit, in which case the faulty unit is replaced by an operational unit. For the diagnosis approach the implication is that the corrective action is unit replacement. One recognizes that there are important questions concerned with unit replacement, e.g., data recovery, and establishing the proper state in the newly connected unit. The solution of these problems is a dominant theme of the later sections of this report. In this chapter we concentrate on fault detection and fault correction using coding approaches.

In Section B below we outline the aspects of coding theory pertinent to this discussion. Section C briefly discusses the use of the trivial duplication and triplication codes in fault detection and masking. Section D discusses the disadvantage (apparent) of utilizing more complicated codes in the processor portion of the computer. Section E demonstrates the use of coding techniques in the memory sections, in particular involving techniques wherein a symbol in a higher order alphabet (greater than two) is associated with an LSI memory chip. Section F presents some comparisons among the techniques for memory systems, and summarizes our views on redundancy techniques for this application.

B. Coding Techniques

The following is a very brief background of the pertinent aspects of coding theory. An $(n, k;q)$ code contains q^k code words of length n, wherein each of the symbols is taken from the field GF(q). The number of redundant digits is $r = n - k$. The Hamming distance of the code, d, is the number of places in which a pair of code words contain differing symbols, minimized over all pairs of code words. For a code of distance d, any combination of t or fewer errors can be corrected, $t < d/2$, and any combination of δ or fewer errors beyond t can be detected when $\delta = d - (2t-1)$. In a linear code the q^k code words form a linear subspace.

For a systematic code, the encoding processes may be thought of as being applied to an n-digit codeword (X, Y) that consists of a k-digit information portion X and an r-digit check portion Y. The task of the encoder is, of course, the calculation of the Y vector from the X vector, assumed given. For an $r \times n$ parity check matrix H in echelon canonical form, namely

$$H = [Q_{r \times k} \vdots I_{r \times r}] ,$$

the computation of Y may be expressed as $Y = QX$.

The decoding process, when error correction is desired, involves converting a "received" vector (X^*, Y^*), which may differ from the "transmitted" vector because of errors, into the code word which is closest in Hamming distance to the received word. The decoding process might only involve error detection in which case the decoding process merely indicates that a received word is not equal to a code word. The decoding process might involve a combination of error correction and detection, in which case, for example, if the received word is within distance two of a code word that code word emerges from the decoder, otherwise an error indication emerges.

The distance properties of the code bear a simple relationship to the columns of the H-matrix. The code has distance d iff (if and only if) all combinations of $d - 1$ or fewer columns of the H-matrix are linearly independent, and there exists a set of d columns that are linearly dependent. For low distance codes, say $d \leq 4$, the decoding process is easily carried out by reference to the H-matrix. The product QX^* is formed from X^*, and then the reference to the H-matrix. The product QX^* is formed from X^*, and then the received digits are added to form the error syndrome

$$Z = QX^* - Y^*,$$

where it is understood that all of the operations are carried out according to the rules of the field GF(q). An error has occurred if and only if Z is non-zero. For single-error correction, the digit position in error is identified by the corresponding column in H being a nonzero multiple (in GF(q)) of Z. For double error correction the correspondence is between a linear combination of a pair of columns in H and Z.

C. The Trivial Duplication and Triplication Codes

Perhaps the simplest error detection scheme that can be visualized is to use two independent systems, each computing ostensibly the same result and to compare the results for disagreement. If each system has a single binary output line then the code being utilized is a (2, 1; 2) code, with H-matrix $H = [1\ 1]$. If the systems contain more than one output line then the entire system is duplicated, and the corresponding lines in each replicate form the positions of distinct duplication codes. The underlying assumption is that a single fault only introduces an error on an output line(s) in one system. Hence, with knowledge of the probability of a fault the effectiveness of the duplication approach can be assessed when incorporated in a system configuration. There are several possible system configurations that use duplication as the basic error detection mechanism. These include

(1) an ensemble of processor pairs wherein one pair is active in computation at a given instant. The detection of an error in this active pair, by virtue of a disagreement, precipitates the replacement of this pair by another pair. It is possible that no effort is expended to diagnose the faulty pair to pinpoint the possible single faulty system--a concept proposed in the Hopkins candidate architecture discussed in Chapter V.

(2) The same concept as above except that a diagnosis is carried out to pinpoint the possibly fault-free unit in a failed pair, and return it to service.

Section F discusses the reliability performance of a coded memory system.

The simplest error correction scheme involving coding is to use three independent systems to carry out a calculation, and to take a majority vote of the results. As in the duplication case above, if each system contains more than one output line, the vote is taken independently for each output line trio. The code being utilized is a (3, 1;2) code with H-matrix

$$H = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} .$$

There are several extensions and system implications of this simple triplication code as follows.

(1) An (n, 1;2) code can be utilized wherein any combination of $\lfloor (n-1)/2 \rfloor$ or fewer* failures are masked and one extra failure is detected if n is even. The decoder is a threshold network with n inputs and a threshold of $\lceil n/2 \rceil$.

(2) A (3, 1; 2) code is utilized to both correct a single error and to distinguish the system block in error. Thus upon the occurrence of an error some control external to the blocks in question switches out the dissenting block, if it is other than a combinational network, the inserted block must be initialized to the state of the other blocks. Note that in the candidate architectures to be described the switchover is accomplished by an executive operating in part with software and in part with hardware.

* The notation $\lfloor x \rfloor$ and $\lceil x \rceil$ signifies the largest integer less than or equal to x, and the largest integer greater than or equal to x, respectively.

(3) An (n, 1;2) code is utilized wherein n is varied as faults occur and are corrected. This adaptive scheme, proposed by Pierce,[4] Goldberg,[5] and undoubtedly others, operates as follows. As in technique (1) above the decoder is initially set at a threshold of $\lceil n/2 \rceil$, and thus the system can correct any combination of $\lfloor (n-1)/2 \rfloor$ or fewer faults (permanent or transient). When the first failure occurs, the malfunctioning block is logically disconnected, whence the code is transferred to a $(n', 1; 2)$ code, where $n' = n - 1$. The next failure results in an additional decrementing to $n'' = n' - 1$, and so on until the code length is reduced to three or possibly two. Obviously, this approach (3) exhibits more fault tolerance than technique (1), and is comparable in fault tolerance to technique (2) is generally to be preferred because it is compatible with graceful degradation. Technique (3) is to be preferred only for those applications that cannot endure the switchover time associated with (2), or for those applications that cannot attain multiple transient fault tolerance with technique (2).

D. Use of Less Trivial Codes for Enhancing Processor Reliability

Ever since the coding theorists have demonstrated the effectiveness of error correcting codes in increasing the reliability of information transmission, a search has been underway to uncover a similar concept for computation. There have been numerous noble attempts, but, in our opinion there seems to be no coding technique for the processor portion of the computer that is more effective than the trivial techniques of the previous section. This is especially the case for the processors that are suitable for our environment since they are relatively small, being only about 2,000 gates or several LSI chips.

We divide this discussion into sections concerned with arbitrary logic and arithmetic.

1. Arbitrary Logic

By arbitrary logic we refer to the irregularly structured combinational and sequential circuits in a processor responsible for timing, control, condition sensing, register transferring, logic operations, etc. For such circuits no preferred or canonical methods of realization have emerged, and hence in attempting to apply redundancy techniques to such circuits it must be assumed that arbitrary logic is utilized in the realization.*

The goal of using coding techniques in association with arbitrary logic is to detect the occurrence of failures in a replaceable module, without incurring the excessive redundancy attendant to the use of replication codes. In communication systems, wherein the main design consideration is transmission rate, the redundancy associated with the use of coding is simply r/n, the ratio of redundant bits to code length. The situation is significantly more insidious in the case of coding techniques as applied to arbitrary logic.

To illustrate this point consider the circuit depicted in Figure II-1, where it is assumed that g_1 is an arbitrary function of f_1, f_2, f_3, and is generated in an indivisible module. A similar assumption applies to g_2, g_3, g_4, h_1, h_2, h_3. The indivisibility property of the various modules means that a module failure will cause an error in the output. Winograd and Cowan[6] (WC) have suggested a coding scheme to provide fault tolerance in a circuit of the type depicted in Figure II-1. The WC method for this circuit is depicted in Figure II-2. The intention here is to provide a mechanism so that an error due to failure at any level is corrected before it propagates to the next level. To achieve this three extra

* Microprogramming techniques for the realization of control functions can eliminate much of the arbitrary logic associated with conventional combinational and sequential realizations. Moreover, the microprogram memory can be made fault tolerant by the relatively inexpensive coding techniques described in Section E below. For the aircraft environment we have in mind the processor(s) constitute a relatively small proportion of the total system and hence, unless a significantly larger computation load is envisaged, the trivial replication coding techniques will suffice.

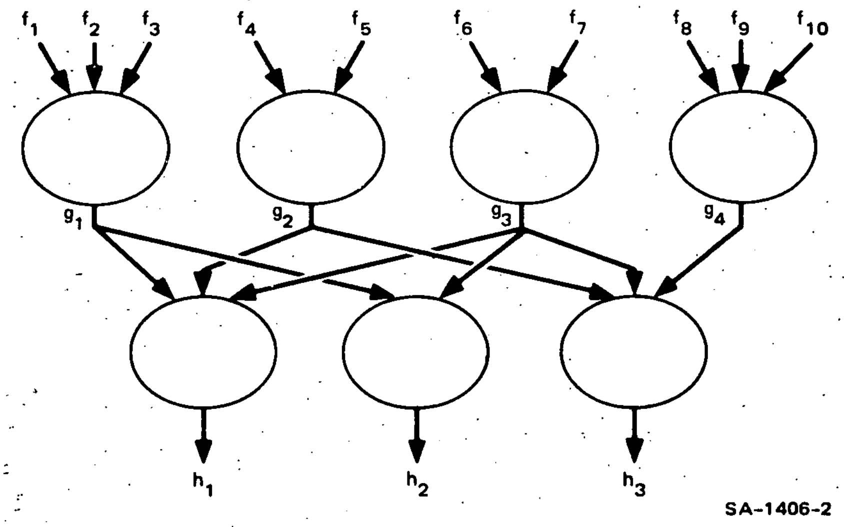


FIGURE II-1 A NONREDUNDANT CIRCUIT

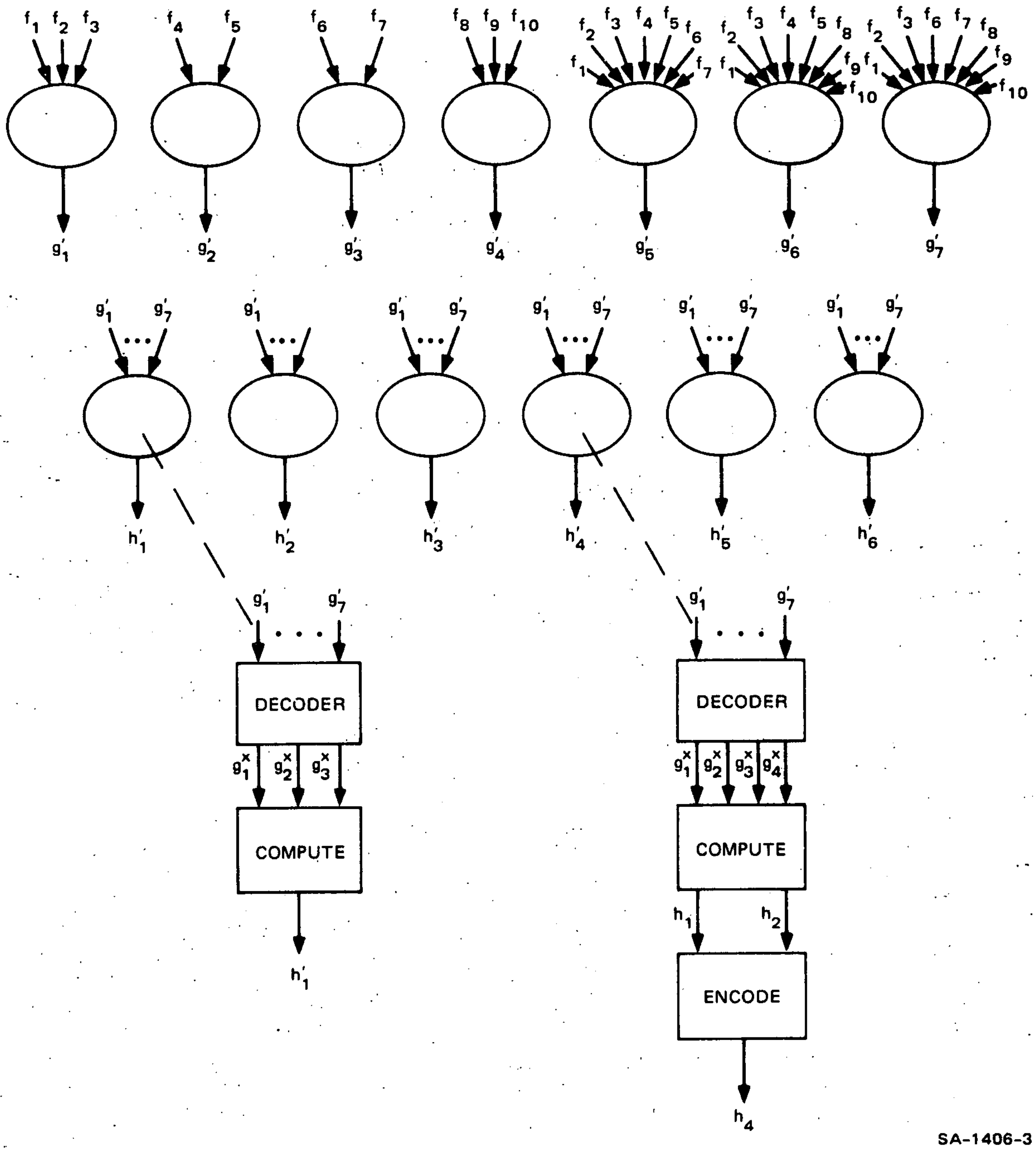


FIGURE II-2 CODED VERSION OF CIRCUIT USING WINOGRAD-COWAN CONCEPT

g-modules are included such that in the absence of failures the signals on the lines g_1, ..., g_7 correspond to a Hamming single error-correcting (7, 4;2) code. (The number of information lines at a level, in this case k = 4, is determined by the number of modules in the nonredundant circuit at that level.) At the next level, namely the h-level, the modules produce signals corresponding to a single error-correcting (6, 3; 2) code. (The H-matrices for these two codes are depicted in Table II-1.) The particular way in which this is to be carried out is to provide a set of modules each of which exhibits three roles: (1) decodes the signals appearing on the previous level to produce estimated error-free versions of the signals needed by the corresponding irredundant modules, (2) computes the irredundant signal, (3) encodes the signal so as to produce the corresponding signal in the code at the next level.

This three stage process is illustrated in Figure II-2 for the computation of signals h_1' and h_4''. With regard to h_1 the signals g_1', g_2',...,g_7', are passed through a decoder to generate estimates, g_1^*, g_2^*, g_3^* of the signals g_1, g_2, g_3. (If no failures or only a single failure occurred in the ensemble of g modules, then the estimates are indeed the desired signals.) The signals g_1, g_2, g_3 are then combined to form the signal h_1'. The encoding process for h_1' is simply the identity operation since h_1 corresponds to an information digit. On the other hand h_4' corresponds to a check digit, which by reference to Table II-1 is specified by $h_4' = h_1' + h_2'$. This exclusive OR operation is carried out in the encode portion of the h_4' module.

As we previously mentioned, the coded version, as shown, implementing single error-correcting codes, can mask failures, provided no more than one module in a given level is afflicted. If one measures the redundancy as the number of modules in the coded version, then the cost seems quite favorable as compared with the use of the trivial replication codes. In the case illustrated single fault masking is achieved with less than twice the number of modules. However, one immediately notes that the modules utilized are significantly more complex than those appearing in the original version. This added complexity is both in the number of inputs to the modules and in the functions realized in the modules.

$$\begin{array}{ccccccc} g_1 & g_2 & g_3 & g_4 & g_5 & g_6 & g_7 \\ \left[\begin{matrix}1 \\ 1 \\ 1\end{matrix}\right. & \begin{matrix}1 \\ 1 \\ 0\end{matrix} & \begin{matrix}1 \\ 0 \\ 1\end{matrix} & \begin{matrix}0 \\ 1 \\ 1\end{matrix} & \begin{matrix}1 \\ 0 \\ 0\end{matrix} & \begin{matrix}0 \\ 1 \\ 0\end{matrix} & \left.\begin{matrix}0 \\ 0 \\ 1\end{matrix}\right] \end{array}$$

(7, 4; 2)

$$\begin{array}{cccccc} h_1 & h_2 & h_3 & h_4 & h_5 & h_6 \\ \left[\begin{matrix}1 \\ 1 \\ 0\end{matrix}\right. & \begin{matrix}1 \\ 0 \\ 1\end{matrix} & \begin{matrix}0 \\ 1 \\ 1\end{matrix} & \begin{matrix}1 \\ 0 \\ 0\end{matrix} & \begin{matrix}0 \\ 1 \\ 0\end{matrix} & \left.\begin{matrix}0 \\ 0 \\ 1\end{matrix}\right] \end{array}$$

(6,3;2)

Table II-1

H-Matrices for (7, 4; 2) and (6, 3; 2) Codes

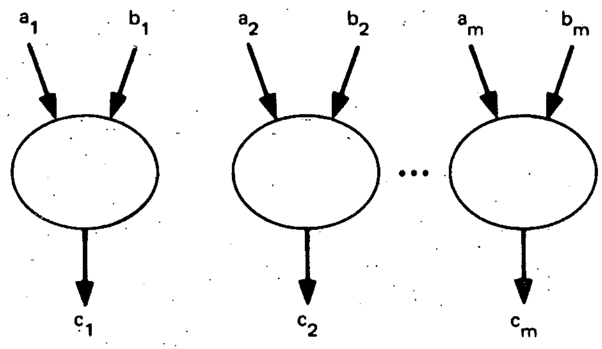
In general it is not easy to measure the complexity of the redundant, as compared with the nonredundant case but a few general observations can be made.

(1) Typically, the realization of modules as LSI chips results in a pin limited design, rather than one which is logic complexity limited. Hence the WC cost measure that relates to the number of modules is not tenable.

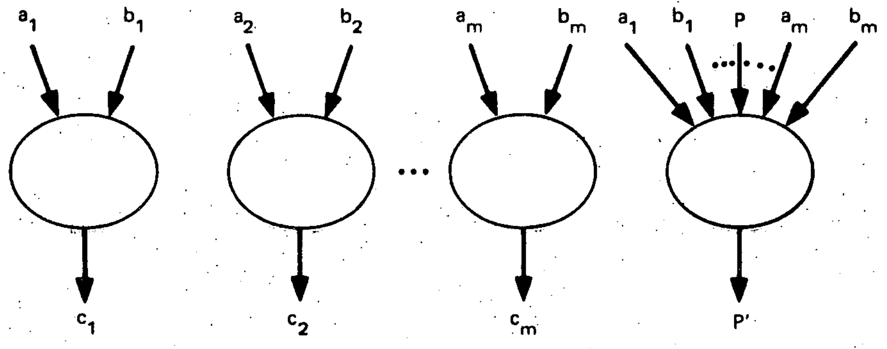
(2) In view of (1) the increase in complexity due to the use of close packed codes, e.g., Hamming codes, is greater than the threefold increase in complexity due to the use of the trivial (3, 1; 2) code.

(3) The use of low-density codes[7] is likely to decrease the module complexity, but at the cost of more modules. Briefly, low density codes are codes wherein the H matrix has a small proportion of 1's relative to 0's and hence the decoder portion of the module is likely to require fewer inputs.

Several modifications of the WC approach are possible to reduce the complexity, and possibly lead to more attractive coding implementations. Instead of supplying an encoder and decoder at each level in a multi-level circuit, it is possible to achieve fault tolerance with a single encoder at the overall circuit input and a decoder at the output. In this case the errors are clearly not corrected at each level in the circuit, but at each level sufficient redundant modules are provided to preserve the code. As an example consider the situation depicted in Figure II-3 wherein, in the nonredundant case, Figure II-3(a), the circuit is a row of AND gates. The code in question is the (n, n-1; 2) code that corresponds to a single parity check. As illustrated in Figure II-3 the input lines to the circuit level include the single parity check P. The parity check P' at the output is to be such that if parity is not satisfied at the input, due to a failure at some previous level, then parity is to be satisfied at the P' levels. Also if a failure occurs in the set C_1, and C_n, the parity is also set to be



(a) NONREDUNDANT CASE



(b) CODED CASE

SA-1406-4

FIGURE II-3 ILLUSTRATION OF CODING APPLIED TO LOGIC WITHOUT ENCODER AND DECODER AT EACH LEVEL

satisfied at P′. Thus the equation for P′ becomes

$$P' = \underline{\text{if}}\ a_1 + b_1 + a_2 + b_2 + \ldots + a_n + b_n + P = 1\ \underline{\text{then}}$$

$$c_1 + c_2 + 1\ \underline{\text{else}}\ c_1 + c_2 \quad .$$

It takes little observation of this function to realize that the module computing P′ is at least as complicated as the initial row of AND gates. Hence we conclude that, for this type of coding scheme simple duplication is no less attractive for single error detection than more complex coding schemes. A similar conclusion has been reached by Pierce.[8]

For the processors we have in mind it appears that for the detection and correction of errors the simple duplication and triplication codes are the most attractive. We stress that this conclusion is based upon the present art of processor design and of applying codes to arbitrary logic. As we show later, coding techniques as applied to memory are quite attractive and feasible. It is also possible that codes will be useful for portions of the bus systems. Since the application of codes to the system is dependent on the particular architecture, we defer discussion of this question to those sections concerned with architecture.

2. <u>Arithmetic</u>

The case with regard to codes to detect and correct errors due to failures in the arithmetic unit is somewhat more promising, although for our case the duplication and triplication schemes seem optimal. Historically,[9,10] arithmetic codes were suggested to detect and/or correct errors of a particular type in an adder. A parallel adder is envisioned comprising n-stages, but not including any logic for fast carry propagation. A single failure is assumed to affect only a single output in a single stage. Thus in a given stage either the sum <u>or</u> carry outputs could be in error. The effect of such a failure in the i-th stage of an adder is to produce an error in the sum as follows:

$\pm 2^i$ if the error is in the sum output

$\pm 2^{i+1}$ if the error is in the carry output .

For the purposes of detecting the occurrence of such errors, all numbers to be added can be multiplied by three; the code in question is then an AN code with A = 3. Any single failure of the type described above introduces an error of the form $\pm 2^j$ and hence, the occurrence of an error is ascertained by the sum not being a multiple of three. In this code two redundant bits are required.

For the purposes of detecting a larger class of errors the use of a larger value of A, notably 15, has been suggested by Avizienis[11] as an approach toward detecting a larger class of errors. This expanded set of errors includes (a) simultaneous errors in the sum and carry outputs of a single stage (b) errors occurring in the multiple usage of the adder as in multiplication, (c) errors occurring due to failures in a fast carry propagation circuit.

Avizienis has also noted some advantages in encoding and decoding attendant to the use of A's of the form $2^m - 1$. For such A's if b is the number to be encoded then a check number c is computed as $C = A - (|b| \bmod A)$. If the binary digits are appended to the most significant portion of b, then the resultant word is indeed Ab. The decoding process, which involves determining if the sum is a multiple of 15, is accomplished by casting out A's. Avizienis accomplishes this, with low cost, using a 4 bit adder (for A = 15), and requires one cycle in this decoder for each 4-bit byte in the sum. The total equipment required for an arithmetic unit, including encoder, decoder and extra bits in the adder itself, for Avizienis scheme is about 1½ times an irredundant unit, irrespective of any spares. This redundancy is less than the greater than 100 percent associated with duplication.

Despite the apparent attractiveness of the Avizienis scheme we do not recommend the use of arithmetic codes for the computer for the following reasons:

(a) For an arithmetic unit realized with LSI technology it seems unlikely that a failure will be confined to a single stage of the adder; hence, a significant proportion of the failures will lead to undetected errors. (Note that the detection of a failure is likely to be dependent on the input data to the adder; hence, the inability of the code to detect the failure during a particular cycle of the adder might be rectified during the next cycle when different inputs appear.)

(b) In the STAR computer the information to/from the arithmetic unit is transferred byte serial. Thus there is no appreciable delay introduced by the byte-oriented encoder and decoder. For a system wherein the transfer is to be accomplished in parallel, the encoding/decoding delay would be significant. Moreover, it is not clear that a duplicated byte serial adder would be less preferable then a coded parallel adder for a byte-oriented machine.

(c) The arithmetic unit is a relatively small part of a processor, comprising about 20 percent of the total number of gates. Hence, without comparable coding techniques for the remainder of the processor it appears that little is to be gained by the use of coding in the arithmetic unit.*

---

* Recent work by Rao[12] and Neumann[13] has shown that logic operations can be performed within an arithmetic unit provided the carry lines are available as outputs. More significantly, the same error detecting codes used for arithmetic can also serve to detect failures when a logic operation is being carried out <u>provided</u> n cycles through the adder are allowed for, say, the logical ORing of two n-bit vectors.

E. Use of Codes for Memory

1. Memory System Organization

The situation with regard to codes for main memory is far more promising than for logic or arithmetic. Intuitively the primary reasons for this optimism are based upon the organization to be described.

A memory can be very easily designed such that any failure within an independent unit results in only an error within a byte. What we have in mind is a semiconductor memory organized as in Figure II-4. The memory consists of f·p chips organized in p-blocks and f-frames. Each chip contains its own decoder, read amplifiers, and read/write control circuitry, besides the storage flip-flops. There is considerable latitude in organizing the separate memory chips, but several typical choices for a 4096 bit chip are the following:

1 (bit wide) X 4096 (words), 2 X 2048, 4 X 1024, 8 X 512.

By redundantly including the decoders in each chip, a single fault within a chip affects only the byte associated with that chip. (We will assume that a chip failure can be catastrophic to the chip in question, to the point wherein all bits in the chip-byte are suspect.) Hence an error-correcting code can be effectively utilized here, provided the code can correct all errors within a frame (byte) width, corresponding to the width of the chip. Thus with this organization there is no need to consider distinct protection strategies for the memory decoders, sense amplifiers, read/write control circuitry, and so forth.

Coding schemes for arithmetic and arbitrary logic suffer in that the encoder/decoder are just about as complex as the logic performing the real computation. Fortunately, such is not the case for memory coding schemes. As we will shortly demonstrate, a relatively small memory of 4K 29 bit words (including redundant bits for coding) consumes about 30 chips while an encoder/decoder can be implemented with one or two chips.

There are several generalizations and embellishments of the simple scheme of Figure II-4, mostly related to the use of such a memory within a complete system. Among these aspects are the following:

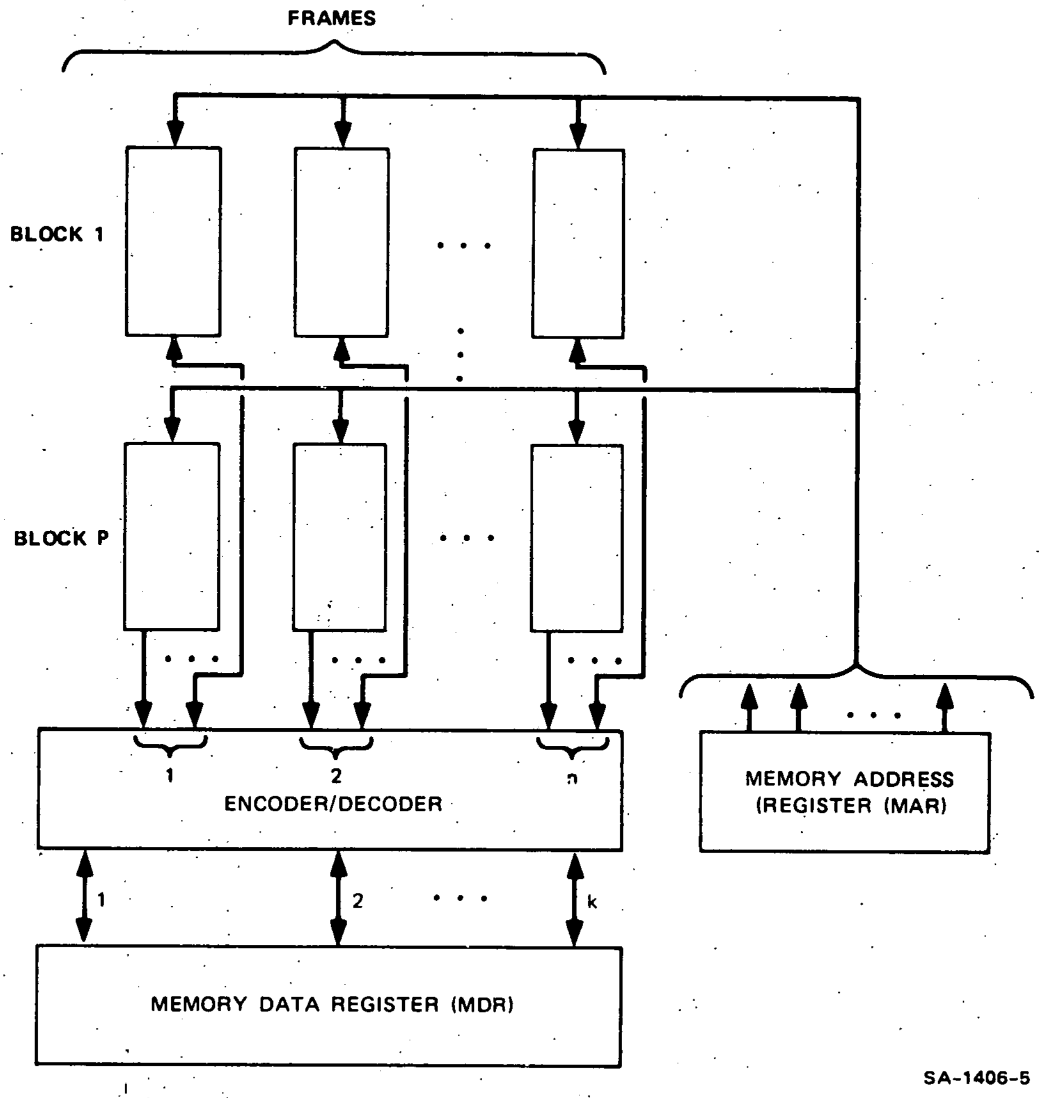


FIGURE II-4 CODING SCHEME FOR MEMORY

(a) Providing spare frames in each block for purposes of additional fault tolerance. Such spares, presumably "addresable" by the encoder/decoder, would be used to replace a failed chip as revealed by the correction process.

(b) Providing for fault tolerance in the encoder/decoder and MAR (Memory Address Register). This coding scheme as utilized in the Bus Checker architecture embodies the use of a pair of encoder/decoders and MAR's as part of a dual or triplex processor system. Thus the processor is protected by the trivial codes while the inherently costlier memory system is protected by more efficient codes. It is also possible to distribute portions of the encoder/decoder among the various memory chips--an issue that should be explored in later studies.

Hence processors which are primarily constituted of memory blocks, can be effectively protected almost in toto with coding techniques. In such processors, control, e.g., instruction processing, could be achieved with microprogramming, and logic operations, e.g., vector AND's and OR's by combinations of logic in memory, table-lookup, and associative memories. The memories associated with these functions could be protected by coding. There will be residual arbitrary logic, but for such a memory-oriented processor this logic will be minimal and protected by the replication codes. Since the arbitrary logic is minimal only a small incremental increase in system cost is incurred.

Accordingly, in the sections below we discuss bounds on the redundancy of the memory codes, properties of the Hamming and other codes that achieve or almost achieve the bounds, implementation of the encoder/decoder as a programmable cellular array on a single chip, and performance of these codes within a system.

2. Bounds on Code Redundancy

As inferred in the above section we are interested in codes that can correct all single byte (or equivalently single-frame) errors.

That is, for a 32-bit word, a_1, a_2,..., a_{32}, assuming a 4-bit frame size, the following errors (among many others, of course,) should be correctable by the code:

$$e_1;\ e_1e_2;\ e_1e_3e_4;\ e_1e_2e_3e_4;\ e_{19},e_{20}\ .$$

The following errors need not be correctable:

$e_1e_3e_5$ (error pattern spans two bytes)

e_1e_{24} (double byte errors).

The situation of interest to us is depicted in Figure II-5

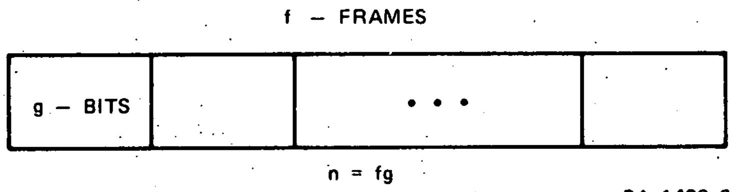


FIGURE II-5 FRAMES OF A BLOCK-CODE WORD

From observation of this illustration, one notes that a code of length $n = fg$ over GF(2) is desired or possibly a code of length f over GF(2^g). To describe a completely general situation we will seek bounds wherein the burst to be corrected by the code is b, where $b \geq g$. It is emphasized that the case of most interest is $b = g$.

A burst of length (at most) b is a pattern of errors (not necessarily solid) confined to b consecutive digit positions; for example, the patterns 1, 11, 101, 111 are bursts of length three, 1, 11, 101, 111, 1001, 1011, 1101, 1111 of length four. There are clearly 2^{b-1} bursts of length b, beginning in a given position. A cyclic burst is one which assumes the last digit and the first digit of a code word are contiguous. The length is similarly defined. A framed burst is a burst error that occurs solely within one frame. Its length b is therefore at most g.

Analysis of codes for correcting a single burst of various types is made based on the number of syndromes required. The base-two logarithm of this number is a lower bound on the number of redundant digits required. A cyclic-burst-correcting binary code which corrects any single burst of length b must have (at least)

$$S_C(n,b) = n2^{b-1} + 1 \qquad (1)$$

distinct syndromes, one for each of the 2^{b-1} bursts beginning in each of the n digit positions (plus one for the case of no errors).

Now consider the situation wherein the code is to correct only noncyclic bursts, i.e., the last digit of the word is not considered to contiguous with the first.

Let N(b) be the number of bursts of length b which are omitted from a cyclic-burst-correcting code in going to a noncyclic-burst-correcting code, i.e.,

$$S_C(n,b) = S_N(n,b) + N(b) \quad .$$

The number N(b) is easily obtainable as

$$N(b) = \sum_{i=1}^{b-1} i\ 2^{i-1} = (b-2)2^{b-1} + 1 \quad ,$$

which leads immediately to the bound, for $b \geq 2$,

$$S_N(n,b) = (n-b+2)2^{b-1} \quad . \tag{2}$$

Finally, consider the situation of a code which only corrects single framed bursts of length b within a frame of length g. The number of such bursts <u>within</u> a frame of length g is

$$g2^{b-1} - N(b) = (g-b+2)2^{b-1} - 1, \quad b \leq g \quad . \tag{3}$$

Consequently, using (3) for each of the f frames, plus one for the error-free syndrome yields the desired number of syndromes

$$S_F(n,b) = f(g-b+2)2^{b-1} - f + 1, \quad b \geq g. \tag{4}$$

Note that for g = b the above bound reduces to

$$S_F'(n,b) = f2^g - f + 1 \quad . \tag{5}$$

Equation (5) is identical to the Hamming bound for a code of length f over 2^g that corrects any single 2^g errors, which is equivalent to any of the 2^g-1 possible error patterns within a frame.

Hong and Patel[18] have shown how to construct minimum redundancy framed burst codes. These codes come very close to achieving the redundancy implied by (5), namely $\log_2 S_F'$.

It is desirable to examine the saving that can be obtained by using framed bursts. This is done first by examining the ratio of the numbers of syndromes, then by examining the number of redundant digits which can be saved.

Consider the saving of framed bursts over arbitrary cyclic bursts of length b. The ratio of (1) to (4) gives the factor by which the number of syndromes may be potentially reduced:

$$S_{CF} = \frac{n2^{b-1} + 1}{f(g-b+2)2^{b-1} - f + 1} \sim \frac{g}{g - b + 2} \quad \text{for } g \geq b \geq 2,$$

$$= \frac{4}{3} \text{ for } b = 2, \qquad (6)$$

$$= 1 \text{ for } b = 1.$$

This ratio is independent of n. For a given b (or for a given g), (6) is maximized when b = g, in which case the number of syndromes required is reduced by a factor of about b/2 for $b \geq 2$. (Note that the ratio (6) drops off sharply as g-b increases.) This maximum for b = g corresponds to a potential saving on the order of $\log_2 b - 1$ check digits for codes correcting only framed bursts instead of cyclic bursts. Note however that for small b there is comparatively little saving in going from (1) to (4) (or even to (2)).

Incidentally, any code correcting cyclic burst errors of some length b is capable of correcting noncyclic bursts of the same length, while a code correcting noncyclic bursts of that length is capable of correcting framed errors of the same length. Thus one could always resort to a code correcting cyclic bursts, especially if encoding and decoding for framed bursts are inordinately complicated.

Table II-2 presents the lower bounds for the number of redundant digits r, required for the cases b = g = 1, b = g = 2, b = g = 4. The Hamming codes for b = 1 and the Hong-Patel codes for b = 2 and 4 achieve the redundancy implied by S_F in all cases shown.

# check digits r

k	b = g = 1	b = g = 2		b = g = 4	
		S_C	S_F	S_C	S_F
4	3	5	4	7	6
8	4	5	5	7	6
12	5	6	5	8	7
16	5	6	6	8	7
24	5	6	6	9	7
28	6	7	6	9	8
56	6	7	7	10	8
60	6	8	7	10	8
64	7	8	7	10	9
128	8	9	8	11	10
256	9	10	9	12	10
512	10	11	10	13	11
1024	11	12	11	14	12

TABLE II-2 REDUNDANCY FOR CORRECTING CYCLIC AND FRAMED BURSTS

3. Specific Codes for Correcting Framed Burst

From the standpoint of minimum redundancy the optimal codes, for the special case of $b = g$, are the generalized Hamming codes[14] over $GF(2^g)$. For r_F redundant frames, i.e., gr_F redundant bits, the $r_F \times f$ H-matrix is given as follows. The columns of the H-matrix consist of distinct r_F-tuples, with components in $GF(2^g)$ with the additional provision that no pairs of columns are multiples of each other. Thus for a given value of r_F the maximum number of columns in the H-matrix is given by $(2^{gr_F}-1)/(2^g-1)$. Table II-3 depicts the H-matrix for the case $g = 2$, $f = 15$, $r_F = 3$, corresponding to a code to handle 24 information bits, 2 bits per frame; 6 check bits. We have also included, in Table II-3(b) the multiplication table for GF(4). It is recognized that bits of a frame are suitably interpreted as elements of the field $GF(2^g)$. In order to augment this code to provide for single frame-error correction, double frame-error detection, an additional row and column are added to the H-matrix as shown in Table II-4.

Besides the generalized Hamming code, the Abramson code[14] is also a possibility for g = b = 2. For the Abramson code the number of redundant digits is typically one more than for the Hamming single error-correcting code over GF(2). This code although, of course, offering no redundancy advantages as compared with Hamming codes does offer slight advantages in decoding, particularly if portions of the decoding can be done in parallel.

For the case b = g = 4 Table II-5 gives the smallest known value of r for the case of single frame-error correction. The best codes for correcting cyclic bursts are based on Reference 15; most are truncations of cyclic codes (and therefore not cyclic). For our application cyclicity is not crucial since the decoding will most likely be accomplished in parallel; it may however imply greater structure to enhance parallel implementation. A cyclic code (21, 12) exists, as does (trivially) a cyclic code (12, 4) the latter being effectively triple-modular redundancy. The latter code can also be achieved by interlacing two Abramson codes (6, 2), in addition to interlacing four Hamming codes (3, 1). Best codes with r = 8 which correct framed bursts (b = 4) with k from 12 to 60 are obtainable as base 16 Hamming codes. The code (68,60) is perfect. Best codes for k = 4, 8 and 12 for these framed bursts of length 4 can be derived from the cyclic Gilbert code (20,12) (which has the advantage that it is systematic)--see Reference 16. Although this code corrects cyclic bursts with b = 2, it corrects framed bursts with length 4. Decoding for the Gilbert codes is very simple in the given framework, although these codes become noncompetitive in terms of r as k increases.

4. Implementation of Framed Burst Correction Codes

In order not to incur any time delay in the encoding and decoding of the codes, it will probably be necessary to carry out a parallel implementation. Moreover, the decoding can in part be accomplished in parallel with computation. That is the results of the computation are held in abeyance until the syndrome calculation determines if an error is present in the word. Although such an implementation requires

$$H = \left[Q_{r_F \times k_F} \quad I_{r_f \times r_f} \right] .$$

$$H = \begin{bmatrix} 0 & 2 & 3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 2 & 3 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 2 & 3 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(a)

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

(b)

Table II-3 H-Matrix for (15, 12; 4) Code

$$H = \begin{bmatrix} 0 & 0 & 2 & 3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 2 & 3 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 2 & 3 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Table II-4 H-Matrix for (16, 12; 4) Code

K	S_c	S_F
4	8	8
8	9	8
12	9	8
16	10	8
24	10	8
28	10	8*
56	10*	8*
60	10*	8*
64	11	9
128	12	12
256	12*	12
512	13*	12
1024	14*	12*

TABLE II-5 REDUNDANCY FOR BEST KNOWN CODES FOR SINGLE-FRAME CORRECTION, b = g = 4

significantly more gates than a serial implementation for, say, a cyclic Hamming code or Abramson code, the cost will still not be excessive. In fact, for the implementations to be described a single LSI chip realization is possible for a combined encoder/decoder.

a. Cellular array for single error correcting binary code

In Reference 17 we presented several approaches to the parallel realization of encoders and decoders, with the stress placed on cellular array realizations that are well-suited to the LSI technology. The following discussion, abstracted from Reference 17, indicates the method by which a single cellular array can serve both as an encoder and decoder for a single error correcting code over GF(2).

As a simple illustration of how encoding and decoding arrays might be efficiently realized in cellular form, first consider the encoding and decoding for a single-error-correcting plus multiple-error-detecting linear binary code.

This array is shown in Figure II-6, along with the logical circuitry of a typical cell of the array. The cells (stages) of the input register X and the output register Z are also depicted in Figure II-6. Each array cell is seen to consist of a single flip-flop with binary contents q, plus a small amount of cascade logic (a total of about 13 elementary NOR gates, as illustrated in Figure II-7.

For a systematic code, the encoding and decoding processes may be thought of as being applied to an n-digit codeword (X, Y) that consists of a k-digit information portion X and an (r = n - k)-digit check portion Y. The task of the encoder is, of course, the calculation of the Y vector from the X vector, assumed given. For a parity check matrix H in echelon canonical form, namely

$$H = [Q_{r \times k} \vdots I_{r \times r}] ,$$

the computation of Y may be expressed as

$$Y = QX ,$$

where all vectors are treated as column vectors.

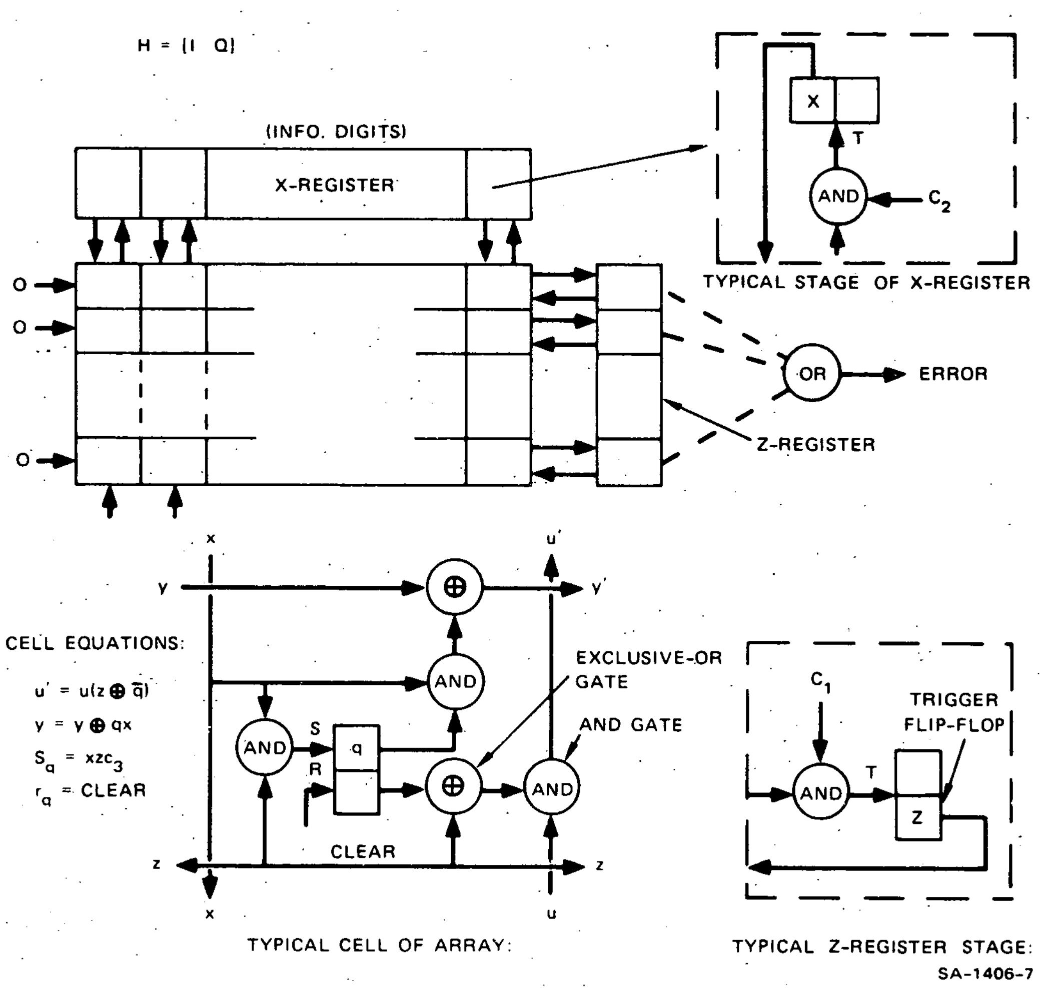


FIGURE II-6 ARRAY FOR ENCODING AND FOR DECODING SINGLE-ERROR-CORRECTING CODES

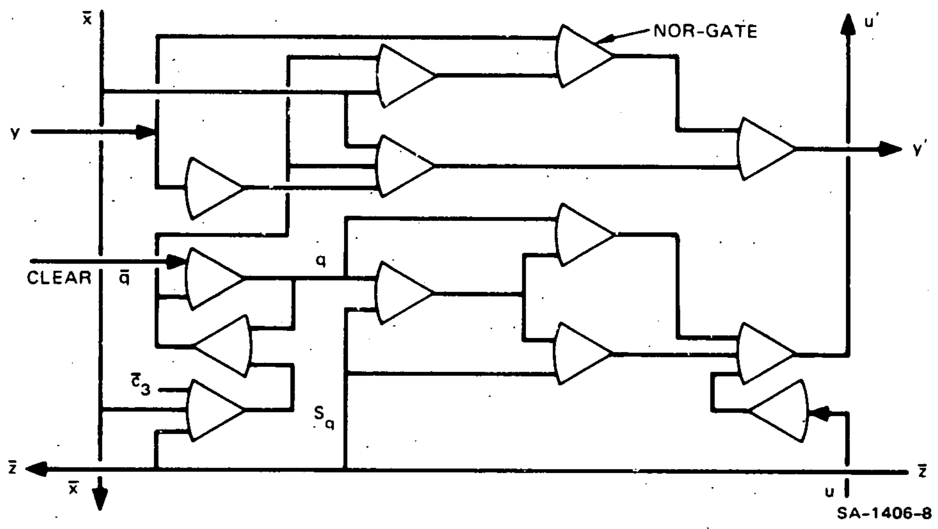


FIGURE II-7 NOR-GATE REALIZATION OF MAIN-ARRAY CELL

For encoding, then, the array and its two associated registers operate as follows. For a given code, each digit q_{ij} of the Q portion of the matrix H is placed in the flip-flop in row i, column j of the cellular array (by a setup process to be described subsequently). The block of k information digits of a particular codeword is placed in the k digit register X, and the r check digits are computed combinationally by the array and inserted in the r-digit check register Z in a single clock time. This computation proceeds through the chain of exclusive OR gates along each row, independently of the other rows, on the basis of the x digits that are bussed vertically down each column. Each x digit x_j contributes to the sum in a particular row i if and only if the corresponding digit q_{ij} of the Q matrix equals 1. Thus, after the clock has been applied to the Z register, this register contains the block of check digits that are to be associated with the given block of information digits.

In general, decoding of a received, possibly erroneous codeword (X^*, Y^*) may be carried out by first recomputing the check digits QX^* from X^* and adding them to the received check digits Y^*, to obtain the error syndrome

$$Z = Y^* \oplus QX^* .$$

An error has occurred if and only if Z is nonzero. For single-error correction, the digit position in error is identified by the corresponding column in H that is identical to Z; that is, by the corresponding column of Q (stage of the X resister) if the error is in the information portion of the codeword, and the corresponding column of I (stage of the Z register) if the error is in the check portion. As soon as the digit in error is identified, it may be corrected by complementing it.

For decoding, then, let the data portion X^* and the check portion Y^* of the received codeword be entered into the X and Z registers, respectively. When clock c_1 is applied, exactly the same operation is carried out as in encoding, except that the calculated check digits QX^* are added (digitwise) to the received check digits Y^*, thereby leaving the syndrome Z in the Z register. The digits of this syndrome are now passed back along the rows of the array on the z busses, for digitwise comparison with the flip-flop contents (columns of the Q matrix) in each column of the array. When clock c_2 is applied, any column in which exact coincidence is found causes a 1-signal to be applied to the corresponding column of the X register, thereby complementing it. If no such column generates such a coincidence signal, then either there was no error, or the error occurred in the check portion Y^* of the received codeword and not in the data portion X^*.

It is trivially noted that error detection may be provided by attaching a simple OR gate gating r inputs onto the Z register, as indicated in Figure II-6. A 1-output from this gate following the application of clock c_1 during decoding then indicates the presence of one or more errors.

For loading the flip-flops of the main array, assume first that the flip-flops are all reset initially. To load a 1 into the flip-flop in row i and column j, inject 1's into registers Z and X in this row and column, and apply clock c_3, $(i = 1,2, \ldots, n - k, j = 1,2,\ldots,k)$. The contents of these flip-flops are not changed during the encoding and decoding operations.

b. <u>Cellular array for codes with frame width, $g > 1$.</u>

The cellular array as discussed above can be easily modified to handle frame widths of any arbitrary size. For the $g = 1$ case above the syndrome is formed on the z-lines and passed left-wise through the array until a match is found with a q-column vector stored in the main array. In the case of a frame size of $g = 2$ the decoding situation is easily illustrated with an example.

Assume that the code in question is the (5,3;4) code with

H-matrix

$$H = \begin{array}{c} \begin{array}{ccc|cc} & Q_{2\times 3} & & I_{2\times 2} & \end{array} \\ \left[\begin{array}{ccc|cc} 1 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 0 & 1 \end{array}\right] \end{array},$$

wherein the components are in GF(4). As in the GF(2) case above the Q portion is stored in flip-flops in the main array. Here 2 flip-flops are required per cell to store a GF(4) component. The encoding process is to form the product QX. This process requires the inclusion in each cell of a GF(4) multiplier and adder, each of which are simple four variable functions. Thus, for example, if the information on vector X is (1 1 1) then the check vector is computed as (1 1) + (1 2) + (1 3) = (1 0). The decoding process is to first form the syndrome Z = QX* - Y*. This syndrome is then reflected back through the array, but instead of searching for a stored q-column that identically matches the syndrome, the search is for a q-column that is a GF(4) multiple of the syndrome. For example, as indicated above (1 1 1 1 0) is a code word in the code in question. (Note that 10 binary digits are involved in representing this code word in main memory. Thus the word as stored in main menory could be (0 1 0 1 0 1 1 0 0 0) Assume that the GF(4) version of the word presented to the decoder is (2 1 1 1 0) corresponding to the first frame being 10 instead of 01. The syndrome is thus (3 3), which is noted to be 3 times column 1. The error is thus identified as in the first frame and the magnitude at the error is the ratio of the syndrome to the "matched" column, which in this case is 3. The data bit in error is corrected by computing X^* - error pattern .

We have not carried out a complete locigal design of this modified decoder, but for the case of g = 2 the cell complexity is about twice that of the single encoder/decoder for GF(2) case. In a similar manner the encoder/decoder for g = 4 incurs a cell complexity about 16 times that for the g = 1 case.

Table II.6 summarizes the memory configurations and decoder complexities for a memory word size of 24 bits, and 4,096 words of memory. We assume a memory chip size of 4096 bits, which can be configured in any of the following modes: a (1 bit wide x 4096), (2 x 2048), (4 x 1024), (8 x 512). In each case we assume a Hamming single error correcting code. The table indicates a strong preference to g = 2 with regard to decoding complexity. Note that the total number of memory chips for the g = 4 case can be reduced by 1 as follows. Seven check bits are sufficient to implement the code, so one of the check frames can be 3 bits wide, instead of 4, providing for $1\frac{3}{4}$ check frames. Assuming 4 blocks the total number of check chips is $4 \times 1\frac{3}{4} = 7$. A similar reduction is possible for the case g = 8.

Memory chip Configuration	# of "check" frames	# of frames (total)	# of chips total	# of decoder gates
1 x 4096	5	29	29	1560
2 x 2048	3	15	30	936
4 x 1024	2	8	32	2496
8 x 512	2	5	40	19968

Table II-6

Summary of Decoder Complexities for Cellular Decoding of Generalized Hamming Codes

F. Summary of System Aspects of Redundancy

For the attainment of fault tolerance in the cpu we recommend the use of the duplication or triplication codes. In general the duplication approach with spares will be used when

- an ultra-reliable mechanism is available for the analysis of disagreement indicators and for the insertion of an operative processor pair
- sufficient time is available for the various processes associated with duplication.

Otherwise the triplication approach will be required. In one of the candidate architectures, SIFT, the basis for selecting duplication or triplication is the criticality of the application program. The executive that accomplishes error control will always be run in a triplicated mode. In another candidate, BUCS, the mechanism for processor switchover is sufficiently fast to permit the use of duplication for processors. However, the approach requires a modest size hardware executive that must be triplicated.

In the case of memory the use of coding techniques is recommended. The main reason for this is the economy in redundancy as compared with the duplication and triplication approaches. In particular, we recommend the use of the framed burst codes described in Section E. As a minimum, the codes should be capable at correcting all single-frame errors, and in applications where back-up is possible, the codes should be augmented to provide double-frame error detection. On the basis of encoding/decoding cost the optimum frame width, g, is 2.

Another organizational factor that relates to the choice of frame width is the following. Assume a memory organization as depicted in Figure II.4 with spare blocks included to handle the function of failed blocks. We assume that a single-frame error correcting code is utilized throughout, and that subsequent to the detection of an error the offensive block is discarded in favor of a spare block. On one hand the number of redundant chips per block is directly related to frame width, g. However, the smaller the value of g, the larger the block of memory that is discarded subsequent to an error. A problem of concern then is, for a given number of memory words required, W, and a given probability of not having W words available at time t, what is the frame width that leads to a minimum number of spare chips. A partial analysis of this problem has been carried out elsewhere,[19] wherein the results indicate that g = 4 is optimal. However, the optimal point is not sharp enough to override the decoding advantage of the g = 2 case.

REFERENCES - CHAPTER II

1. J. Goldberg, K. N. Levitt, and R. A. Short, "Techniques for the Realization of Ultra - Reliable Spaceborne Computers," Final Report - Phase 1, Contract NAS12-33, SRI Project 5580, Stanford Research Institute, Menlo Park, California (September 1966).

2. F. P. Preparata, et al "On the Connection Assignment Problem of Diagnosable Systems," IEEE Trans. Vol. EC-16, pp. 848-854 (December 1967).

3. R. P. Vancura and C. R. Kime, "On Numerical Bounds in Diagnosable Systems," Digest of 1972 IEEE Internation Symposium on Fault - Tolerant Computing, IEEE Publication 72CH 0623-9C, pp. 148-153 (June 1972).

4. W. H. Pierce, "Adaptive Vote-Takers Improve the Use of Redundancy," in Redundancy Techniques for Computing Systems, pp. 229-250, (Spartan Books, Washington, D. C. 1962).

5. J. Goldberg, "Network Schemes for Combined Fault Masking and Replacement," Working paper presented at the Workshop on the Organization of Reliable Automata, Pacific Palisades, California (2-4 February 1966).

6. S. Winograd and J. D. Cowan, "Reliable Computation in the Presence of Noise," (MIT Press, Cambridge, Mass., 1963).

7. R. G. Gallager, "Low Density Parity - Check Codes," IRE Trans., Vol. IT-8 No. 1, pp. 21-28 (January 1962).

8. W. H. Pierce, Fault - Tolerant Computer Design (Academic Press, New York, 1965).

9. W. W. Peterson, "On Checking an Adder," IBM J. Res. Dev., Vol. 2, No. 2, pp. 166-168 (April 1958).

10. D. T. Brown, "Error Detecting and Correcting Binary Codes for Arithmetic Operations," IRE Trans., Vol. EC-9, No. 3, pp. 333-337 (September 1960).

11. A. Avizienis, et al., "The STAR (self-testing and repairing) Computer: An Investigation of the Theory and Practice of Fault - Tolerant Computer Design," IEEE Trans., Vol. C-20, No. 11, pp. 1312-1321 (November 1971).

12. P. Monteiro and T. R. N. Rao, "A Residue Checker for Arithmetic and Logical Operations," Digest of the 1972 IEEE International Symposium on Fault - Tolerant Computing, IEEE Publication 72CH 0623-9C, pp 8-13 (June 1972).

13. P. G. Neumann, private communication.

Preceding page blank

14. W. W. Peterson, Error Correcting Codes (John Wiley and Sons and MIT Press, New York, N. Y. 1961).

15. B. Elspas, "Design and Instrumentation of Error - Correcting Codes," Final Report under Contract AF 30(602)-2327, SRI Project 3318, Stanford Research Institute, Menlo Park, California (October 1962).

16. P. G. Neumann, "On Gilbert Burst Correcting Codes," IEEE Trans., Vol. IT-11, pp. 377-384 (July 1965).

17. K. N. Levitt and W. H. Kautz, "Cellular Arrays for the Parallel Implementation of Binary Error Correcting Codes," IEEE Trans., Vol. IT-15, No. 5, pp. 597-607 (September 1969).

18. S. J. Hong and A. Patel, "A General Class of Macimal Codes for Computer Applications," IEEE Trans., Vol. C-21, pp. 1322-1331, (December 1973).

19. P. G. Neumann, K. N. Levitt, J. Goldberg, J. H. Wensley, "A Study of Fault-Tolerant Computing," Final Report, Stanford Research Institute, Project 1693, Contract N0014-72-C-0254, Menlo Park, California (July 1973).

III IMPLICATIONS OF COMPUTATIONAL REQUIREMENTS

Our purpose in this chapter is to abstract from Volume II the aspects of the aircraft computations that strongly influence the design of the computer. It should be emphasized that the loading requirements are quite tentative, and are possibly inexact by as much as 50 percent. However, they do provide a well considered estimate of the computer complexity required, and together with the results of the tasks on architecture and technology indicate that a computer can be specified to meet the computational and reliability requirements.

Table III.1 summarizes the computing requirements. The most critical phase of the flight from a computational standpoint is during an instrument landing. Those applications that are involved in that phase are indicated with an "*". Small tasks that are not required during that phase do not influence the design of the computer system and therefore have not been estimated to the same accuracy as the more important tasks.

The column headings of Table III.1 are defined as follows:

Task----------------	The name given to the application program.
Criticality Class---	1. Immediate safety of flight impact.
	2. Eventual safety of flight impact.
	3. Significant change-of-mission impact.
	4. Operation impact.
	5. Economic impact.
Iteration Rate/Sec--	The number of times per second that the calculation must be carried out. When two figures are quoted, they represent two calculations within the same functional task.
Equivalent MIPS-----	The millions of instructions per second to carry out the calculations.
Memory Required-----	The number of words of memory required for instructions and data.
Missed Iteration----	The maximum number of consecutive iterations that can be missed before the application is jeopardized.

Table III.1 Computing Requirements, Applications

	Task	Criticality Class		Iteration Rate/Sec	Equivalent MIPS	Memory Required Inst.	Memory Required Data	Missed Iteration
APPLICATIONS								
A1	Attitude Control	1		5,20	.023	1845	230	2-3
A2	Flutter Control	1		250	.069	70	22	2-3
A3	Load Control	3,5	*	240	.014	45	15	2-3
A4	Autoland, Horiz.		*	20				
A5	Autoland, Vert.	1	*	160	.055	750	275	2-3
A6	Autoland, Throttle		*	33				
A7	Autopilot	4		5	?	150	100	4-5
A8	Elec. Att. Control	1	*	30	.077	790	520	?
B1	Supervisor	4		?	?	75	15	?
B2	Inertial	2	*	1-25	.034	2100	150	0-4
B3	VOR/DME	4		5	.004	250	50	4-5
B4	DME, OMEGA	4		5	?	400	105	4-5
B5	Air Data	4		?	?	110	25	4-5
B6	Kalman Filter	4		1/5	.001	250	65	2-3
B7	Flight Data	4		5	.028	450	100	2-3
B8	Airspeed, Altitude	4	*	8,16	.009	360	70	2-3
B9	Graphic Display	4	*	1,8	.032	890	5360	2-3
B10	Text Display	4		10	.019	640	8700	4-5
C1	Collision Avoidance	4	*	1/3,670	.021	550	650	1-2
C2	Data Comm, A/C		*	various	.006	210	400	?
C3	Data Comm, Ground	4	*	≤ 4	.001	450	112	?
D1	AIDS	5	*	1/4 to 4	.002	650	650	4-5
D2	Inst. Monit.	4	*	5	.014	800	100	2-3
D3	Syst. Monit.	1-4	*	1/2	.001	900	50	2-3
D4	Life Support	1-4	*	< 1/2	.001	900	50	3-4
D5	Engine Control	1-2	*	33	.119	1300	200	1-2

*Tasks to be run during most critical phase.
?Indicates that task exerts a negligible load for the parameter in question.

For purposes of interpreting the table and discussing its implications on the computer architecture we briefly discuss the following aspects: reliability, roll-back delay, main memory requirements, processor speed, processing variations within a mission, and data rates.

Reliability--We have assumed that the probability of not successfully carrying out the most critical computation should be less than 10^{-8} per mission. These computations, corresponding to criticality classes 1 and 2, could cause an aircraft crash if not carried out or if carried out with gross errors. With this assumed computation reliability, for a fleet of 1000 aircraft flying 4 daily missions, each of 5 hours without repair between flights within a day, about one crash due to a computer failure would occur in 100 years. For the other criticality classes the assumed reliability is not as stringent--typical failure probabilities are 10^{-4}--since the failure to carry out these computations only results in a mission change or an economic loss. In an actual system design it would be beneficial to allocate redundancy such that each task is carried out with the indicated reliability. For our purposes in specifying an architecture we will assume that all of the computations are to be carried out with the more stringent reliability--an assumption that is reasonable since those computations in criticality classes 1 and 2 constitute a majority of the memory and processor requirements.

Roll-back--An important parameter of a fault-tolerant computer is the maximum time interval that the computer can be in a rollback/reconfiguration mode in responding to a failure. During this interval certainly some processing of certain computations ceases, and newly appearing data might be lost. The missed iterations column of Table III.1 indicates the number of iterations that can be ignored in a given computation without adversely affecting the aircraft. In the worst case (collision avoidance) the system must be "down" for no more than 1.5 msec. Several other critical computations--flutter control, load control, autoland--require reconfiguration times nearly as low. For these computations, depending upon the architecture it might be necessary to reload programs, which indicates that the computer might be required to be totally engaged in reconfiguration following a failure. Fortunately, the computations with large amounts of data, e.g., display, can tolerate a down time of approximately 0.5 sec., thus allowing ample time for the possible reloading of data interleaved with the more critical computations.

Memory Requirements--The application programs for the critical phase require 19.247K words. This figure is undoubtedly low for the following reasons:

- the difficulty of estimating accurately
- the need for memory space for the executive routines.

Hence we assume a memory requirement of 24K words. Note that this is a nonredundant requirement; the demand for fault tolerance will increase this figure. For architectures relying totally on the triplication codes this storage requirement must be tripled to 72K. For architectures utilizing only single-frame correction in memory (plus possibly a few extra frames for double frame detection and sparing) the figure is about one-third in excess of 24K or about 32K.

Processor Speed--For the critical phase the application tasks require .386 MIPS (millions of instructions per second). Once again we must regard this figure as being low in part due to inaccuracies, but mostly due to "wasted" cpu power in multiprogramming and the processing of executive routines. For these reasons we assume a processor load of 0.5 MIPS. An important attribute of the computations is their relative independence. That is, the sharing of functions and data among the computations does not substantially reduce the overall memory or processor requirements. Each computation requires access to the state of the aircraft, but all other data can be considered to be local. Hence it is quite simple to impose a multiprocessor discipline on the computations, with almost an arbitrary number of processors.

Under certain allocation of tasks to processors it is not necessary to do any task interruption within a processor. That is, a task can be allowed to run through completion before initiating another task. Five processors each of 0.1 MIPS would enable such an allocation. However, near the end of the useful life of the computer, say if just one or two unfailed processors remain, it is possible that a high rate task (flutter control) might be allocated to the same processor as a low rate but long task (graphic display). If such a joint allocation is unavoidable, then interruption of the longer task will be essential.

Processing Variations Within a Mission--All applications denoted with an "*" are required during an instrument landing. This represents about 60 percent of the total cpu requirement and about 50 percent of the memory requirement. Hence some graceful degradation is possible as during the mission, tasks will be naturally deallocated as they are no longer needed as part of the flight. Hence, when a task is no longer needed, its memory area can be allocated to another task, or conversely, a failure in a memory module is automatically handled by a reduced memory requirement. However, we note that the degradation with respect to memory is not uniform, assuming that all programs and constants are retained in main memory.† For example, in mid-flight, although not all tasks are being processed, all programs must be stored reliably in main memory. Hence the graceful degradation with regard to main memory is not exploitable until the last minutes of the flight, and hence is of questionable utility to the architecture.

Data Rates--An important measure of computer power required is the load on the bus structure for transfer of instructions and data. With a computing load of 0.5 MIPS we assume that an instruction will, on average, require 24 bits.‡ Different instructions require varying amounts of data including the following cases:

0	bits for register to register operations
8	bits for byte operations, e.g. text display
16	bits for integer operations
32	bits for floating point operations.

Making an estimate that the average data required is 16 bits, the total flow between memory and CPU is 20 M bits/sec. In some architectures (e.g., STAR) the bus would have to be capable of maintaining this rate. (Hence the expansion of the STAR system to accommodate a larger computation burden will require a significant modification in bus design.) In the case of the Hopkins scheme or BUCS, a significant reduction would be

†The issue of back-up memory in an aircraft environment is yet to be completely resolved. Rugged discs can be obtained but their cost/bit is not significantly less than that for LSI main memories.

‡In a 16 bit computer this implies equal number of single and double length instructions.

achieved by the use of the local CACHE on the processors. An additional reduction is achieved by providing a multi-bus structure or allowing multi-parts into main memory. In the SIFT system most of the bus load would be in individual modules, with only an estimated one percent between modules. All of these issues are discussed in Chapters V, VI, and VII.

IV IMPLICATIONS OF TECHNOLOGY

Task 3 of this study (reported in Volume II of this report reviewed the likely technologies that would be appropriate for a fault-tolerant computer in the period 1975-1980. This section of the report examines the implications of that task on the architecture of the computer.

The most important advance in technology will be the continued advance of LSI. The cost of LSI circuits will continue to drop throughout the 1970's, and will result in processor and memory costs that are low enough so that extensive redundancy of units is practical from a cost viewpoint. This redundancy can be either by replication or by coding, the latter being more applicable to memories. It is expected that the cost of a computer system to carry out all computation within an aircraft will be comparable with the present cost of existing single function avionic units (e.g., inertial navigation).

A second advantage in the use of LSI is the small size of such units, making it possible to achieve far more efficient shielding from both electric and magnetic fields, thereby reducing the probability of noise and crosstalk. It is expected that fault modes of this type, which are manifested as data-dependent transient faults will be insignificant within the central units. However such faults may still exist in connections to external sensors and actuators.

With the use of LSI most of the connections at the device and gate level take place within a chip, rather than on a board or through a connector as in the use of discrete circuits. The reduction in soldered and wrapped joints is estimated to be at least an order of magnitude lower than that associated with, say, integrated circuits, with a consequent reduction of faults in the connection system.

LSI circuits, though relatively cheap in high volume production, have a high development cost. This implies that an efficient design would contain as small a number of different chip types as possible. This affects architectural decisions at two levels. At the unit level (memory, bus, arithmetic unit, control, etc.), there will be strong advantage in using replication of identical units rather than units designed specifically for particular functions. At the logic level, the high development cost of customized units makes it more attractive to transfer arbitrary logic to a form of memory as in the use of micro-programming.

Replacement and maintenance strategies in a reconfigurable computer are also influenced by LSI. The large number of gates per chip, together with the tendency for a chip fault to affect many gates, imply that small units such as registers should not be replaced, but rather that groups of registers, all on the same chip should be.

The choice of LSI technologies is between the lower speed, lower cost MOS and the higher speed and cost bipolar technologies. The total computing power required among the elements of the several candidate architectures is such that MOS will be sufficiently fast for memories, busses and arithmetic units. In addition, the use of a multiprocessor organization permits the attainment of high computation capacity with slower processors. The higher speed of bipolar circuits may still be necessary in the control sections where the microprogram cycle time will typically be an order of magnitude faster than the instruction cycle time. Recent advances in technology have tended to bring the two types closer in both speed and cost.

We note, that the choice between different LSI technologies, discussed above, was on the basis of speed and cost. The lower cost alternative of MOS is possible because of the higher density within the chip, thereby enabling the use of fewer chips. This will have the desirable effect of increasing the inherent reliability due to the reduction in number of chips. LSI memory systems appear to be potentially more reliable than core or plated wire because of the reduced numbers of discrete semi-conductor devices and interconnections. The use of batteries is deemed to be a fully adequate assurance of non-volatility.

The MTBF for LSI circuits is estimated to be between 10^6 and 10^7 hours. The lower number is factored into the reliability estimates for the architectures considered. The requirement to achieve a MTBF of 10^8 hours for the whole system can be shown to be achievzble by several architectures.

The use of optical coupling between units can provide great protection against damage propagation through several units. The architecture must therefore be more concerned with fault propagation through erroneous data, than by damaging electrical phenomena. The added cost for such protection is substantial, though not prohibitive, so careful evaluation of reliability value will be needed.

The availability of mass memory (disk, drum, etc.) of a sufficiently high reliability is not crucial to the application, as the total data required to be stored is such that only a small cost penalty will be involved by using LSI techniques for all memory. Advanced bulk-memory schemes will be useful in later systems, but are deemed not ready (as well as not essential) for the time period of interest.

To achieve an estimate of total computing system cost we assume that the cost of memory dominates that for processors, busses, etc. The unreplicated memory requirement of 24K words of 16 bits ($\approx$ 400K bits) yields a total OEM chip cost of $2K. If we assume threefold replication with some sparing, this figure becomes approximately $8K. Allowing an additional 25 percent for the other (non-memory) units a chip cost of $10K is reached. We use, as a rough rule, a factor of 3 to account for wiring, boards, frames, power supplies, installation, etc. The gross estimate of the computer cost is therefore $30K. This figure should not be regarded as firm, owing to the significant design choices still to be made in further study.

V A CHECKLIST AND A SURVEY OF FAULT TOLERANT ARCHITECTURES

A. Introduction

The purpose of this chapter is to survey some of the existing computer systems that have fault tolerance as one of the design goals. The systems that are surveyed here are the STAR computer of JPL, the EXAM computer of NASA-ERC, the ARMMS computer jointly of NASA-Marshall and Hughes, the MARCS concept of IBM-Yorktown, the AADC of Naval Air Systems Command, and the Hopkins Scheme (HS) of MIT-Draper Labs. The STAR exists as an advanced breadboard and the HS as a preliminary breadboard. Under present plans the LSI chips for the AADC should exist within a few years. The future of the other three systems is undetermined.

For reasons which should later be apparent, of the above systems only the HS is a candidate architecture for our application. However, even in the case of HS some key details of error detection, executive operation, and hardware design are currently unspecified, e.g., multiprogramming for the diverse avionic computations. Some additional design work is thus required before we can be assured of the suitability of HS. We should also emphasize that the other five mentioned systems could be modified to better match our environment. However, in modifying these systems we are likely to produce a variant of the three candidates--HS, SIFT, BUCS--with the addition of some techniques found useful for the environment of particular systems. Among such techniques are the special associative memory used for scheduling in EXAM or the arithmetic codes of STAR. Thus it seemed appropriate to us to design the candidates from scratch, borrowing particular concepts of existing systems.

It should also be mentioned that numerous other architectures that embody fault tolerance have been proposed. Among these are the MECRA[1], the PRIME[2] system of University of California, Berkeley, the Three-Fault Tolerant System[3] of North American, ESS[4] of Bell Telephone Laboratories, the Burroughs Multiprocessor[5] and systems currently being pursued by Raytheon and Ultra-Systems. We do not survey these systems here for one or more of the following reasons: (1) insufficient documentation is currently available, (2) fault tolerance is an incidental design goal, or achieved in only limited degree, (3) the probability of further design work being done by the architect is low, (4) the design goals are quite different from ours. In another SRI report[6] these and other systems are surveyed and classified.

Before embarking on a detailed description of the six architectures to be surveyed, we present a checklist for the design of a fault-tolerant architecture. In essence, this checklist is a summary of the constraints imposed by our environment, as measured by the computational requirements and technology.

B. Checklist

Most of the items below pertain to the qualities that are expected from a fault tolerant computer system for commercial aircraft. We have also included some items that enable the reader to evaluate the surveyed architectures and the candidate architectures.

Computational Environment -- As abstracted from Chapter III the aircraft computations as presently envisioned have the following characteristics:

- Independence -- the 26 computations are essentially independent with respect to shared data, shared programs and shared effectors. Almost all of the computations need access to the state of the aircraft which is a relatively small amount of data

- Total processor load -- 0.5 MIPS

- Total memory load -- 24K, exclusive of executive

- Maximum processor load per computation -- .12 MIPS

- Maximum memory load per computation -- 9K words, although this can probably be reduced to 4K words needed for any iteration. Most computations require less than 2K words of memory.

- System down-time -- no more than 10 msec for certain critical computations. Fortunately, the memory requirements for these computations are relatively small. Other computations have less stringent down-time requirements.

<u>Technology</u> -- It is expected that LSI chips will be employed throughout the system. One would expect that at least several hundred airplanes would be equipped with the computer and if each computer incorporates at least five chips of each type -- a figure easily satisfied for each candidate -- then sufficient chips would be utilized to warrant the per-chip set-up costs. As indicated in Chapter V, universal acceptance of LSI memory should appear around 1975 despite the volatility aspect. In addition, main memory will be used exclusively for all programs and data except possibly for the storing of alternative runway information. The failure rate of an LSI chip is taken as 10^{-6}/hr. with the major failure mechanism being interconnections. However, to a first approximation it is assumed that any chip failure causes all chip outputs to be suspect. In designing the system for an LSI implementation it is desirable to utilize as few chip types as possible, and it is necessary to minimize the number of pins/chips -- 40 is an absolute maximum.

<u>Faults</u> -- the following faults are expected

- Permanent -- An LSI chip can fail according to the rate mentioned above. Failures are assumed to be independent among the chips.

- Failure propagation -- Failures are assumed not to propagate from chip to chip, i.e., a chip A in failing will not cause a contiguous chip B to fail. (Of course if an output of A is an input to B then an output of B can certainly be in error.) As discussed in Chapter IV the prevention of failure propagation for certain failure modes (e.g. supply voltage applied to a signal line) may require that the chips be isolated by using light coupling or special driver-receiver combinations as the interface mechanism.*

- Transient -- We are assuming here that at some moment a chip produces an erroneous output, but when the chip is presented with the same data a short time later it will produce the correct result. A transient fault can be caused by a local electrical disturbance, or a design error coupled with a unique data dependency. It is expected that a transient would dissipate itself in 10 msec, or when the timing of the signals is changed slightly. The system, however, should embody some policy to handle larger duration transients.**

- Massive transient -- We are assuming here a transient failure that causes every modifiable element in the system to be suspect. Such an effect could be caused by lightning, by a static discharge when the aircraft passes through a cloud, or even by an accidental power outage. At present there is scant information on the probability of massive-transients or their duration or scope. As we note later a key difficulty in responding to a massive transient is that the tables of operative and failed resources are subject to corruption along with all other modifiable information. A tape cassette or

---

*There may be contiguous chips wherein the mutual propagation of faults can be tolerated. For such chip pairs, of which several schemes, -- and SIFT to a high degree -- have many, the careful isolation is not required.

**As mentioned previously certain critical computations cannot be unserviced for more than 10 msec. Hence in responding to a transient fault the system cannot perform retries for a longer period than 10 msec, but must (at least temporarily) assume that the fault is permanent and connect a spare resource to service the critical function. SIFT, BUCS, HS, and STAR have good transient-response characteristics. SIFT is, perhaps, the best of these, since it always performs a voting of 3 results and thus can continue in operation despite a long-duration, local transient.

other slow, reliable back-up memory can be used to restart the system. It seems clear that a system might not recover in less than 10 msec from a massive transient of much longer duration. One policy is to have the critical computations be capable of being executed in practically all processors with little intervention on the part of the failure-prone executive. SIFT permits such a policy in that some of the critical programs, e.g. flutter control, tend to be small and hence can be permanently resident in all processors so that unless all processors are disabled by the massive transient there is a possibility that the function can be serviced. Some careful design of the effectors might permit a longer MISS time than we are assuming.

Reliability -- For the critical computations the probability of delivering a wrong answer should not exceed 10^{-8}. For less critical computations the reliability requirements are not as stringent -- typically being around 10^{-4}. Actually, these reliability figures should be weighted according to the inaccuracy of the result, but throughout this study we are, pessimistically, assuming that a chip failure disables all of the chip outputs.*

*Of course this does not preclude a single chip failure causing only (for example) the least significant digit of a computation to be in error. However, we expect that the entire arithmetic unit will be realized on one or two chips and hence a chip failure here could indeed corrupt the entire computation.

Availability -- At any time during a mission sufficient resources must be available to handle the critical computations. Once again the probability of the system being unavailable for the critical computations must not exceed 10^{-8}. There are two aspects to availability. Firstly, after the detection of a failure the system should be unavailable for no more than a MISS time that is dependent on the computation. For the critical computations the MISS time is about 10 msec. Secondly, at the end of the mission only about half of the rescurces are required to effect a landing, so that a gracefully degraded system can achieve the availability requirements. In Chapter III, however, we showed that graceful degradation is not as exploitable as one would hope for.

Primary Fault-Tolerance Mechanism -- It seems clear that multiprocessing should be utilized as the primary fault-tolerance mechanism. The computations, being relatively independent and individually requiring low processor and memory loads, are easily implemented with a multiprocessor. The multiprocessor concept can be exploited to provide relatively cheap fault tolerance -- the fault tolerance grows linearly with the number of processors -- and to accommodate to the limited graceful degradation possibilities.

Secondary Fault-Tolerance Mechanisms -- Here the issue is not quite as clear. The system comes close to being memory dominated; 24K words of main memory for a 0.5 MIP processor load is somewhat high by contemporary stanards. Hence the use of error correcting codes for memory tends to be cost effective as compared with triplication. Similarly, for a memory dominated system the use of expensive duplication or triplication techniques for arithmetic-logic processors does not imply an excessive system cost. We emphasize, however, that the issue is not completely clear, and we do not rule out an architecture because it utilizes arithmetic codes!

Redundancy Estimation -- The particular measure that we are looking for here is the ratio of equipment in the redundant system to equipment in a non-redundant version. The comparison is compounded since there is no canonical realization of the non-redundant version, but we might assume that a minimum system contains 24K words of memory and 0,5 MIPS processing power. Hence in computing the cost of the redundant system the following cost factors must all be included: spare processors and memory blocks, check bits for codes, main memory words required to store error control procedures, spare busses, extra equipment required to effect a multiprocessing discipline, extra power supplies, and extra I/O controllers.

Reliability/Availability Modeling -- The problem here is to verify that the probability of a wrong answer and the probability of equipment being unavailable do not exceed the desired figure of merit -- as a function of the computation. For a purely static system, e.g., one that employs only voting, the reliability calculation is trivial. For dynamic systems like the candidate architectures the situation is more insidious. Among the factors that must be included are: the occurrence of a second fault while the system is responding to a first fault, failures in unflexed equipment, i.e., components that are not utilized until an error occurs, the frequency of diagnosis, and the scope of a transient failure that can disable the system. Previous work on dynamic system modeling has always relied on a static model of the situation. Based on our study of the several architectures, it is our feeling that the results produced by this modeling tend to be pessimistic. For some architectures, this means that a given reliability figure of merit may be attainable with less redundancy than indicated by the model.

Transparency to User -- The application programmer should not have to be concerned with the fault tolerance procedures in the composition of his programs. This is similar to general ignorance of an application programmer of the executive functions of any large computer installation. Again we would expect that sound executive system design would attain this goal.

Expandability -- It is desirable that the candidate architectures be expandable to handle a larger computational load. What this means is that for a given system configuration, extra main-frame equipment programs and I/O devices are easily added to produce a larger system. It is not entirely clear what range of system power will be ultimately desired, but for commercial aircraft over the next 15 years a computational range of 0.2 MIPS - 2 MIPS and a memory range of 10K - 100K seem reasonable. Systems of less capability than the smallest configuration are too small to worry about, and it does not seem possible that there would be enough computations in any conceivable commercial aircraft to occupy a system larger than the largest configuration. It is clear that a system is readily expandable if processors and memory blocks are easily connected without requiring a major modification of the bus system, or without requiring modifications to the executive other than the setting up of tables for the new equipment and programs. The reader is warned that there are insidious factors involved in designing an expandable system. For example, the executive overhead might grow excessively as procesors are added, or in a minimal system the executive, in an attempt to make it general, might overwhelm the system -- the problem with running OS/360 on the low line 360 machines.

Prototype Development -- Although the cost and performance of the ultimate production system are of primary importance,it is of some interest to estimate the cost of developing a prototype unit since most of the systems presently exist only on paper. The major factors of interest here are:

- the feasibility of using off-the-shelf processors and memories
- the amount of special-purpose logic that must be designed and implemented
- the possibility of using microprogramming techniques instead of costly hardware for fault tolerance functions
- the scope of a system that must be prototyped in order to demonstrate the concept.

In the next sections we attempt to discuss the various architectures relative to the items in this checklist. Chapters VI and VII discuss the SIFT and BUCS architectures also relative to the checklist.

C. STAR

System Organization

Essentially, the STAR is a single processor decomposed into separate functional units each of which is replicated two or more times. With the exception of the memory modules needed for critical functions, the TARP and the logic processor, only one functional unit of each type is normally operating. Faulty functional units are replaced by power switching. Communication between units is via two 8-bit byte-serial busses to which units are passively connected (no switches on bus lines). Two forms of error detecting code are employed in information transfer between units. Every transmitted word is examined for error syndrome by hardware contained within a critical "hard-core" test and repair processor called the TARP.

Design Goals

Extremely long meantime to failure, low weight, low power consumption, low computation rate, relatively rapid recovery from transient faults.

Technology

Discrete transistors and small IC packages. Magnetic-core R/W memories, a "rope-core" fixed memory, magnetic amplifier type power switches.

Faults

Intended to recover from all single permanent or transient failures confined to an IC Package. Massive transients and program failures can, at present, not be tolerated.

Reliability Assessment

Survival for 100,000 hours with .95 probability. Recovery from transient faults in less than 50 ms. See reference 7.

Expandability

STAR is not particularly well suited to expandability because of the limited band-width of the communication busses, which must transmit each word as eight serial bytes of four bits each. Current memory address space is also limited to 2^{16} words. A two-processor version of STAR might be based on a duplication of the TARP (test and repair processor) and bus system with the sharing of normally unpowered spare functional units between the two systems. This plan would lead to less than a doubling of the original hardware but would require additional "hard-core" control structure to handle multi-programming and memory sharing. The STAR is not easily contractible unless the error detecting code features are abandoned, in which case the principal means of fault detection would be lost.

Reliability Mechanism

The principal mechanisms employed in STAR to produce over-all system reliability are:

1. The use of redundant coding to detect errors in transmission of data or instructions from one functional unit to another.

2. Hardware for internal consistency checking within functional units that detects some types of malfunction. For example, a word placed on a bus by a unit not agreeing with what actually appears on the bus, causes the unit to report such an occurrence via a status-bus to the TARP.

3. A TARP with at least 3 identical versions powered up at all times (with voting) plus spare units to be switched on line to replace disagreeing members. This is the "hard core" of the system.

4. A highly reliable magnetic switch mechanism for disconnecting malfunctioning units from active status and powering up replacement units.

Recovery from Failures

The TARP contains a mechanism for re-executing instructions when some indication of failure has occurred. If a fault occurs because of a short transient, normal operation may be resumed by attempting the last previous instruction. On a repeated failure the TARP may then attempt to resume computation at a designated "roll-back" point, which it is the responsibility of the programmer to provide. Otherwise (on repeated failure) the unit indicating a defective status or transmitting an improper code is switched off and replaced.

The claim is made that fault detection and unit replacement can be handled in 50 ms. However, this does not imply that at most a 50 ms delay in program execution would result, since some error circumstances might require roll-back to a previously established program check point.

With regard to programming errors, STAR has no explicit means for coping with this variety of fault, although machine instructions are represented in an error detecting code, since an assembly-language program could easily be faulty (i.e., in a tight loop) without causing any error report. Recovery from such blunders would have to be a strictly software implemented function, and at present STAR does not have a program-interrupt feature in the TARP.

Measure of Redundancy

In the STAR origanization redundancy depends upon the number of spare units provided. Where memory space is not a vital consideration (so that memory units can be lost without replacement) a reliable configuration containing at least one extra replacement for each functional unit would appear to require somewhere between 1.5 and 2.5 times the hardware for a minimum organization with no spare units. As a rough estimate, a minimum configuration STAR would probably contain at least 3 times the hardware of a computer of the same computational capacity, but without special reliability features.

Transparency to User

In the current implementation the user must insert roll-back points in his program and save the status himself. We would expect that in a later version a compiler and executive routine could handle this chore.

Current Status

The STAR has been implemented and tested in "bread-board" form at JPL and continues to be an experimental prototype of on-board computers for unmanned deep-space exploration.

General Conclusions

While the STAR computer organization has many novel features uniquely suited to its mission it does not match well with our given mission objectives. In particular, the weight, size and power requirements are irrelevant, the STAR's rate of computation is far too low to handle the several more critical tasks of stability augmentation, inertial navigation, etc., STAR is not well suited to expansion, and lastly, the STAR is inefficient in the sense that a large amount of hardware remains completely idle unless a real malfunction actually occurs.

D. EXAM

System Organization

The EXAM system[8] is a very homogeneous multiprocessor, multi-memory configuration in which any processor can have access to any memory through what amounts to a huge cross-bar switch. Each processor can find itself in one of three states:

1. A problem state in which it is working on some applications program.

2. The executive state in which it is doing the usual system-monitor functions.

3. An idle state in which it attempts to seize executive control.

Hardware interlock features are provided to prevent more than one processor from securing executive status. An associative memory, accessible to any processor, is provided to handle some of the executive control processes. Memory access by more than one processor to the same memory unit is handled by a "round-robin" priority scheme.

Design Goals

Principal motivations for the EXAM architecture were: high performance (as measured by executed instructions per second), the possibility of a highly repetitive LSI construction, and very flexible expandability to many-processor organizations

Technology

The ultimate use of LSI is assumed as the only reasonable technology for the cross-point array. This array connects processors to memories in a fully parallel set of 36-bit communication paths. Otherwise, processor and memory technologies are optional although the associateive memory required for executive functions would imply semi-conductor implementation.

Faults

The documentation is not clear on this point, although processor failures and memory bank failures should be handled by a simple switchover processor. The system seems very vulnerable to massive transients and permanent failures.

Reliability Assessment

No estimates are avialable. The possibilities for producing a reliable machine would seem to depend on software augmented identification of faulty processors and/or memory modules, followed by modification of the executive program to deny communication to or from these units. A particularly dangerous feature of the EXAM organization from the reliability standpoint is that each processor can write in every memory (except perhaps the executive associative table). Thus there is no mechanism to prevent a faulty processor from destroying data in any memory unit.

Also very critical to the reliability of the EXAM organization is the integrity of the LSI crosspoint switches. A switch failure might, for example, prevent access by a processor to several memory units, or render some memory unit unavailable to any processor.

Reliability Mechanisms

The EXAM organization contains no explicit plan for ensuring reliable operation. Assuming that the cross-bar switch connecting processors with memories could be made "hard-core" reliable by TMR techniques, then survival of some of the initially available processors and memories may be assumed. In this case a continuing but degraded performance of the system would be feasible, but only if sophisticated software diagnostic methos were employed to reschedule tasks. The most vulnerable area is the possibility of a runaway processor destroying information in an unprotected memory.

Recovery from Failures

Here again no particular recovery means have been supplied by the system architects. One may visualize schemes in which several processors take part in one task, comparing results. On discovery of a disagreement the offending processor might be denied memory accesses by dropping it from the round-robin memory queue. Here, recovery times could be as short as one instruction cycle or as long as the execution time between programmer-assigned check points. The whole strategy is vague and it is not at all clear what might be expected to occur on a transient that affected some processors but not others.

Measure of Redundancy

The EXAM architecture has no redundancy in one sense because some programs could employ all processors and memories in useful computation. To construct a "reliable" version of EXAM would require at least a TMR realization of the cross-bar switch plus considerable internal redundancy in the processors, memories and in memory access circuitry, and coding in memories. This might amount to a factor of 3 over a non-redundant version.

Expandability

The EXAM concept is certainly very flexible with regard to computational power since almost any number of processors and memories can be accommodated by the same general organization. However, computational power is not proportional to the number of processors employed since conflicts on memory access can lock out several processors desiring to look at the same memory module. Efficient use of the EXAM configuration requires careful partitioning of tasks into independent program modules -- an elaborate scheduling algorithm that must be confronted for any multiprocessor organization.

Prototype Development

Prototype development should not be overly involved because the processors, memory banks and associative memory are quite standard. The cross-bar switch will require special design but this should not be difficult to realize since its structure is uniform, and since some design work on it is complete.

General Conclusions

The EXAM computer has the merits of structural simplicity through the use of many similar or identical modular units. It also offers a high degree of utilization of available computational power. From the reliability standpoint, however, there seem to be no intrinsic features to recommend it over other multiprocessor organizations that have a less elaborate inter-connection structure. In particular, the vulnerability of memory to any faulty processor and the necessity of faultless operation of the elaborate cross-bar switch seem to be two good reasons to reject this design concept.

E. Hopkins Scheme (HS)

General Organization

The HS, Ref. 9, is a multiprocessor reasonably well suited for the aircraft environment. Each processor is in reality a processor-pair, PP, with a small triplicated scratchpad memory, TSM. A large main memory, MM, is also present. The TSM holds the status of the PP -- presumably an alias exists in TSM for all registers, flip-flops in PP. An error in a PP, as revealed by a comparator, or in the TSM causes the TSM to gain control of a bus to MM, in which case the contents of TSM are dumped to a preset area in MM. An executive then retries the computation on the same PP-TSM combination or on another PP-TSM combination. As presently conceived, the processors in the PP must operate in locked step. The bus is in reality a triplicated bus wherein the outputs of a TSM are voted upon prior to their insertion on the bus. The main memory organization is at present not clear but presumably it is amenable to the same coding techniques utilized in BUCS. All of the executive processing is accomplished in a particular PP-TSM until it fails whence another acts as the executive. A major drawback of the present version to access the bus at each instruction. The use of the TSM as a cache would improve the situation, but at the expense of increasing the cost of the TSM -- a discardable unit.

Computational Environment

HS seems well suited to our environment as discussed above. The system is intended for use on a manned spacecraft that exerts a computational load similar to that of the aircraft.

Technology

Probably LSI throughout.

Faults

HS should handle single permanent or transient faults confined to primitive processor unit within a PP, a single scratchpad memory within TSM, or a single bus. Since the memory organization is not specified the issue is not clear here. Multiple PP-TSM failures can be handled provided they do not occur within a recovery interval. There is no policy specified for massive transients, and at present it seems that a processor can write anywhere in main memory, thus leading to a vulnerability to programming failures.

Reliability Assessment

No goal has been specified but with a few spare PP-TSM's and, say coding in MM, our reliability goal is attainable. HS tentatively specifies a procedure for handling the diagnosis of the PP comparators, otherwise these would be unflexed hardware.

Availability

In responding to a fault the system effects a recovery by dumping the TSM contents and reloading these contents in another TSM. As presently envisioned the TSM is probably no more than a few hundred words, thus enabling a restart in 1 msec.

Redundancy

If MM is replicated, say by a factor of three, the redundancy ratio is at least 3. For the use of a coding scheme in MM Similar to that used for BUCS the ratio is probably about 2. As we will note in Chapter VII. this ratio is very sensitive to the size of the TSM.

Reliability Model

No reliability model has been conceived, but a rough estimate of reliability and availability is attainable as the product of the following factors: the probability of PP-TSM failures in a mission, the probability of a bus failure in a mission, and the probability of a second failure within a recovery interval. The unflexed situation is not included.

Transparency to User

The application programmer is not involved in the error control procedures; the executive handles all aspects of recovery. The application programmer might wish to partition his program so that it runs efficiently in the cache environment.

Expandability

The degree of expandability is limited by the traffic that the bus can carry. Without a cache and with a single TMR protected bus, and with a single port memory, the maximum processor load is probably no more than 0.5 MIPS. With the use of a cache with each processor, or the use of a more complex bus and a multi-port memory, the processor load can be pushed to 2 MIPS. The bus would have to be designed initially to handle the insertion or deletion of processors.

F. ARMMS

General Organization

The Automatically Reconfigurable Modular Multiprocessing System (ARMMS) was conceived at NASA-MARSHALL and is currently being pursued at Hughes Ground Systems Group. We emphasize that much of the design work remains to be done, as of July 1972, so that there are gaps in the concept. The ARMMS system is intended for a spaceborne environment, wherein the reliability goal is long life and wherein a varying computational load is anticipated throughout a mission. The system consists of eight processors, capable of being configured in a variety of modes. (At present, it is not clear at what rate the configuration can be modified.) Up to three tasks can be processed simultaneously on three processors. These three processors presumably operate in a near locked step TMR mode. Thus up to five processors can be simultaneously active; the remainder of operative processors are spares. Each processor contains a local memory of, say, 128 words with the main memory serving as a back-up to this local memory. We would guess that special logic within the processor controls the transfer of blocks of words between the local memory and main memory. The main memory is organized as 32 banks although no complete error detection or error correction procedures have been specified for it. There are hardware voters on the processor to memory links so that when the processors are operating in a TMR mode any processor errors are corrected before the errors propagate to memory. Most of the system control is contained within a specialized executive unit called BOSS. BOSS performs all of the executive functions associated with scheduling, allocation of resources to tasks, error reconfiguration, I/O control. Internal redundancy is used to make BOSS reliable, i.e., only one distinguishable BOSS unit exists.

Mission

A general spaceborne computer is sought that is responsive to a variety of missions. In this regard the system embodies variable redundancy -- a multiprocessed simplex operation for low criticality tasks, a duplex mode for moderately critical tasks, and a TMR mode for high criticality tasks. In the multiprocessed simplex mode the computation capacity is to be 2 MIPS, spread over three processors. The reliability goal is the survival of at least one processor, and a suitable portion of memory, with probability 0.99 after 5 years.

Technology

It is not immediately clear if special purpose LSI chips are planned. A processor called Space Ultrareliable Modular Computer (SUMC) has been designed as the basic processor. Some effort is being devoted to partitioning SUMC into modules. There appear to be no special constraints on the processor operation, so that an cff-the-shelf high performance LSI processor should suffice.

Faults

Certainly in the TMR mode any fault that disables a single processor can be tolerated. A duplex mode is also possible in which single processor faults are detected. There seems to be no fault tolerance in the simplex mode. Masking of memory faults with error correcting codes is planned for the future. BOSS will probably be protected with TMR applied at the module level. The design has not proceeded to the point where the response to massive transients or permanent faults can be assessed.

Reliability

For our aircraft environment only the TMR mode of ARMMS can satisfy the 10^{-8} reliability requirement.

Availability

ARMMS can be equipped with sufficient resources to meet our availability requirement.

Redundancy

It is difficult to measure the redundancy in ARMMS since the design is still in its conceptual stage, in particular with regard to memory and BOSS. It is our guess that the redundancy ratio for a TMR mode will exceed three.

Transparency to User

The user is not involved in fault tolerance procedures except possibly to specify the criticality of his task, which in turn determines the mode in which ARMMS will process the task.

Expandability

Memory is to be expandable to 512 K words. The BOSS design does not seem to allow for the addition of more than 8 processors, although it should be noted that each processor is quite powerful -- 0.5-1 MIPS.

Prototype Development

The prototype development is likely to be quite costly, mostly because of BOSS. It seems that BOSS is to be realized as a special purpose processor. A portion of BOSS can be a simple but high-speed, general-purpose computer since many of BOSS' functions require little more than table look-up or simple processing. However, much of the high-speed communication links between BOSS and the other system blocks will require special attention.

Conclusions

Perhaps the most novel feature of ARMMS is the concept of variable fault tolerance. This is potentially attractive for our environment because of the varying reliability among criticality classes. However, as presently conceived the simplex mode of ARMMS does not provide any fault tolerance -- a situation that cannot be endured even for the least critical functions. If this mode could be modified to provide a probability of error detection of 0.99 per hour then the ARMMS concept would deserve a more critical study. Another disadvantage of ARMMS is the specialized BOSS unit. It is our feeling that the redundancy of the system would be significantly reduced by incorporating most, if not all, of the executive function within a processor, as in HS, SIFT, and BUCS.

G. AADC

The Advanced Avionics Digital Computer[10] (AADC), now called the All Applications Digital Computer, is currently being developed by the Naval Air Systems Command with technical support from the Naval Research Laboratory and several industrial organizations. The AADC is intended to satisfy the majority of the Navy's computing requirements in the 1975-1985 time period. Our discussion here is brief since the error control features of the AADC remain to be specified. The tack currently being pursued is to design and build the hardware components, and to later incorporate fault tolerance procedures in specialized hardware and software executives.

The system consists of a set of processor elements (PE's), an associative processor (AP), a large random access main memory (RAMM), a bulk storage, and a bus structure interconnecting the above units. For our purposes here the PE is of most interest. A PE consists of a processor and a task memory, wherein a program requiring service is loaded into a task memory for execution. Significant design effort is being devoted to the PE design in an attempt to realize the entire PE as a small number of wafers. If this is achieved, then the basic PE is certainly a candidate as the processor in SIFT, HS or BUCS. (HS and BUCS might require special interface logic to effect comparisons between a pair of processors.) Presumably a PE will be a discardable element in any fault-tolerance scheme.

It is impossible to assess the reliability aspects of AADC since they are yet to be conceived. It seems clear that the system will be highly redundant -- but perhaps this is not of much concern if the PE's cost is nominal. However, it is our feeling that fault-tolerance must be incorporated into the system at initial design phases in order to produce a truly reliable, efficient system. Invariably special interface logic or special processor instructions are required, for example, to prevent error propagation, to permit the use of error correcting codes in main memory, to permit the recovery from massive transients, or to protect main memory access from a faulty processor.

H. MARCS

General Organization

The Modular Architecture for Reliable Computer Systems[12] (MARCS) is currently being pursued at IBM - Yorktown. At the time of writing this report MARCS is not a clearly defined system but a concept embodying some very sound fault-tolerant principles. In essence the concept has been a testbed for the study of coding techniques and circuit diagnosis algorithms.

The concept involves an interconnection of primitive subunits to form a uniprocessor. The subunits are: arithmetic logic unit, scratchpad memory and program control unit, bus control unit, I/O processors, recovery control unit, and main storage. The main storage incorporates frame coding, as discussed in Chapter II, with the inclusion of spare frames. When the system is operating in a uniprocessor configuration spare subunits are provided (except for main storage), where the computer is operational if at least one subunit of each type is operative. An interesting logic coding scheme is utilized such that a single gate fault in any subunit eventually produces an error signal at the subunit's output. In principle, this circumvents the unflexed hardware problem, at least for units that perform comparison or decoding of an error correcting code. A multiprocessor mode is also possible.

It is impossible to evaluate MARCS as a potential candidate since much of the computer is unspecified. We have been strongly influenced by the coding techniques both for logic and memory, and by the diagnosis techniques, particularly those of Roth's, that in essence have set the stage for most of the current work on diagnosis.

1. F. P. Maison, "The MECRA: A Self Reconfigurable Computer for Highly Reliable Process," IEEE Trans., Vol. C-20, No. 11, pp. 1382-1388 (November 1971).

2. B. R. Borgerson, "A Fail Softly System for Time Sharing Use," Digest of the 1972 IEEE International Symposium on Fault-Tolerant Computing, pp. 89-93 (June 1972).

3. L. J. Koczela, "A Three-Failure Tolerant Computer System," IEEE Trans., Vol. C-20, No. 11, pp. 1389-1390 (November 1971).

4. H. J. Beuscher et al., "Administration and Maintenance Plan of No. 2 ESS, " Bell System Technical Journal, Vol. 48, pp. 2765-2815 (October 1969).

5. J. D. McGonagle and R. L. Davis, "Advanced Multiprocessor," Technical Report AFAL-TR-69-308, Burroughs Corp., Paoli, Pa. (June 1970).

6. P. G. Neumann, J. Goldberg, K. N. Levitt, and J. H. Wensley, " A Study of Fault-Tolerant Computing," Final report Report under Contract N00014-70-C-0254, Stanford Research Institute, Menlo Park, California, AD 766974 (July 1973).

7. A Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (self testing and repairing) computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Trans., Vol. C-20, No. 11, pp. 1312-1321 (November 1971).

8. G. Y. Wang, "System Design of a Multiprocessor Organization," Memorandum RC-T-079, NASA Electronics Research Center, Cambridge, Mass. (1969).

9. A. L. Hopkins, Jr., "A Fault-Tolerant Information Processing Concept for Space Vehicles," IEEE Trans., Vol. C-20, No. 11, pp. 1394-1403 (November 1971).

10. "Design of a Modular Digital Computer System," Phase I Report under Contract NAS8-27926, Hughes Aircraft Company, Fullerton, California (April 1972).

11. R. S. Entner, "Presentation of Advanced Avionic Digital Computer Baseline Definition," Naval Air Systems Command, Washington, D. C. (September 1969).

12. W. C. Carter et al., "Logic Design for Dynamic and Interactive Recovery," Proceedings of Symposium on Fault Tolerant Computing, Pasadena, California (March 1971).

PRECEDING PAGE BLANK NOT FILMED

VI SOFTWARE IMPLEMENTED FAULT TOLERANCE (SIFT)

This section describes a proposed design of a computer system in which fault tolerance is achieved using software techniques, to remove the need for special fault-tolerance hardware units. The computer resembles a multiprocessor with a restriction that each processor may not write into the memory associated with other processors. The executive system and the application programs are protected by identical fault tolerance procedures. The computer design gives the system programmer the ability to vary the degree of fault tolerance by changing the extent to which a program is replicated among the processors.

The fault tolerance procedures to be described may be implemented by combinations of program, micro-program and hardware elements. The present discussion emphasizes programming. The fault tolerance features can be made completely transparent to the application programmer, but it is also possible to allow the application programmer to prescribe fault tolerance procedures that are specially appropriate to a given computation.

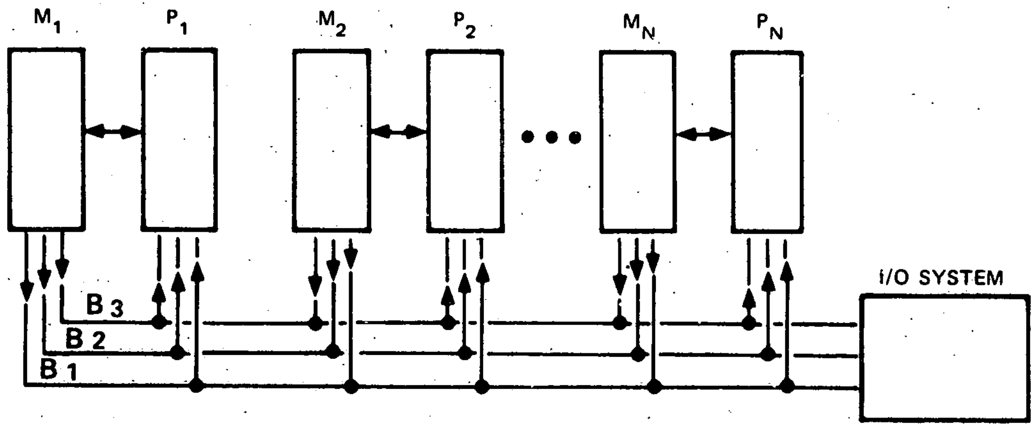
Also included in this section are analyses of the reliability and fault tolerance to be expected from the design. The size and speed of the individual units (processors, memories, buses, etc.) are based upon the computational power requirements as outlined in Chapter III, and as described in detail in the Volume II report of the project. Included in this section are estimates of the storage and computing requirements for the executive of the system. Some alternative design options are discussed.

A. System Design Overview

The system (Figure VI-1) consists of a number of modules, each composed of a memory and processing unit.

The individual processing units within the modules are connected to the corresponding memory units with wide-bandwidth busses. The inter-module bus organization (B_1, B_2, B_3)* is designed to allow a processor to read from any memory but not to write into other memory units. This is a novel and central feature of the system, and serves to prevent fault propagation.

*The bus logic envisioned does not use voting. The number three is chosen for convenience of discussion.



M_i	Memory
P_i	Processor
B_i	Bus

TA-710522-220R

FIGURE VI-1 SIFT SYSTEM CONFIGURATION

The inter-module bus is expected to have a much lower bandwidth than an intra-module bus.

The input/output (I/O) system, discussed in a later section, is assumed to be connected to the busses B_1, B_2, B_3. The input/output (I/O) system shown in Figure VI-1 is all the non-computing units, e.g., transducers, actuators, sensors. That part of the total input/output which is carried out by program, e.g., formatting or code conversion, is handled in the same manner as any other task, i.e., is replicated in several processors.

The system is viewed as being regular in that no module is given special facilities or is, a priori, assigned a special role. All computations that require high reliability are carried out in several modules. We assume for the purpose of this description that critical tasks are processed in three units.

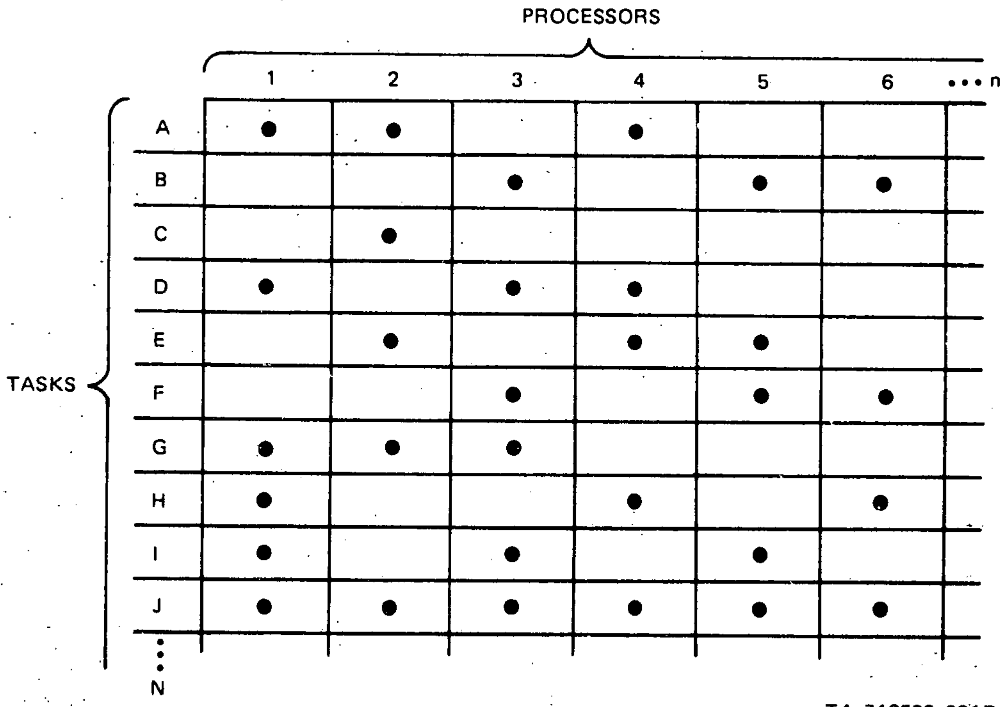
The computations which must be carried out are broken into a number of tasks in such a way that no task requires more computing power than can be supplied by one processor. The tasks are given the designations, A, B, C...; the processors are numbered 1, 2, 3... . Each processor is capable of being multiprogrammed over a number of tasks, as illustrated in Figure VI-2.

The control of the computing system is carried out by an executive system that can be segmented by function into two parts:

(1) LOCAL EXECUTIVE: Functions that apply to each processor (e.g., dispatching, reporting errors, loading new task programs).

(2) SYSTEM EXECUTIVE: Functions that are global to the system (e.g., allocation and scheduling of work load, reconfiguring).

A complete set of the software functions of Class (1) is present in each processor (possibly in microprogram); those in Class (2) are carried out in a sufficient number of processors to provide the degree of fault tolerance required. The functions are realized by programs that have the same task structure as all other programs.



	1	2	3	4	5	6	••• n
A	●	●		●			
B			●		●	●	
C		●					
D	●		●	●			
E		●		●	●		
F			●		●	●	
G	●	●	●				
H	●			●		●	
I	●		●		●		
J	●	●	●	●	●	●	
⋮ N							

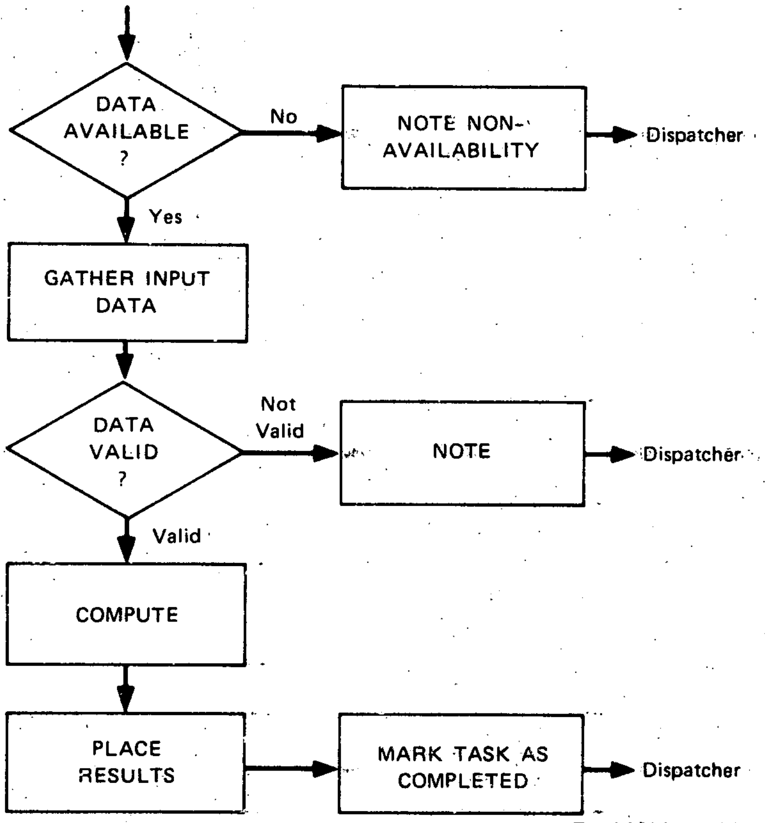
FIGURE VI-2 AN EXAMPLE OF TASK/PROCESSOR ALLOCATION

The normal operating mode for a processor carrying out a task is to follow the flow of control shown in Figure VI-3. Data required for the task are assumed to have been computed by several processors (including possibly the same one carrying out the task). A check is made to see if the data are available in all processors. If not, the fact is noted in the memory of the module and the dispatcher program within the module is entered to determine which task is next to be processed. The next processing is the reading of input data from the several processors where copies of that exist. A validation is now carried out, typically (but not necessarily) by a two-out-of-three vote. If any of the copies of the input data are found not to agree, then this fact is noted for later processing by the executive. If all the copies are different, the fact is noted and control moves to the dispatcher program. The computation of the task is now carried out, the results left in the memory of the module, and note is made (in the module) of the fact that the task is computed.

Certain important principles are obeyed in the above scheme:

- No processor writes into the memory of another module.
- Input data in a module are not destroyed during the computation. If the computation is repetitive, the results of one cycle that may be used as input for the next cyle are placed in a different location in memory. Similarly, because the input data within one module may be needed later by another processor carrying out the same task, the input data must not be destroyed until all cooperating processors have read, validated, and used the data. This may require that the data have to be preserved over several iterations if they are used by another task which is delayed behind the first.
- All conditions (e.g., errors, task complete) are left as notes to be read later by the system executive.
- The dispatcher program, which exists in each module, maintains a queue of tasks to be computed. The data for this queue are read from memories of the modules that are running

*Dispatching is the executive function that initiates a new task at the completion of the previous one.



TA-710522-222R

FIGURE VI-3 TYPICAL TASK FLOW

the executive. The flow of control of the dispatcher is itself similar to that shown in Figure VI-3, except at the end, when the control is transferred to the task that is at the head of the queue.

- The dispatcher in each processor checks from time to time to see if the system executive has changed the queue of task for that processor. A single bit (per processor) is set in the system executive tables to indicate a change of the queue. If this bit is not set, the dispatcher waits some time (e.g., 1 msec.) before querying it again, thereby preventing continuous interrogation and consequent heavy inter-module bus traffic.

The above scheme achieves a high degree of fault tolerance without special hardware requirements. In particular, an erroneous calculation carried out by a module does not destroy the validity of the total system, because results are rejected by the next calculation.

B. Major Characteristics of SIFT

The system described above has many properties which, in total, distinguish it from other fault-tolerant systems.

- Replicated units do not operate in lock-step mode, but are only loosely synchronized. The communication between CPUs is asynchronous, thereby removing the need for an ultra-reliable system clock.
- Agreement between replicated units is verified only at the completion of program segments (tasks).
- Faulty units are not necessarily removed but can either be ignored or assigned to tasks having no overall effect.
- Transient faults do not necessarily cause permanent removal of the faulty units. Furthermore, the looseness of synchronization among sets of tasks makes it possible to enhance immunity from transients by providing that redundant versions of a computation may be done at different moments in time.
- The degree of fault tolerance can be different for different tasks being performed, and/or can be different at different times for the same task.

- No special hardware is used to carry out fault detection or correction.
- Communication between CPUs is minimized so that low-bandwidth busses can be used, thereby facilitating physical separation of modules in environments where physical damage is a hazard.
- The design concept is independent of the way in which the units are built, i.e., no specialization of CPU or memory design is required for fault tolerançe, thereby allowing the choice to be based on other properties, e.g., speed, or availability.
- The total computing power of the system can be varied by using units of different speed or by changing the number of units.

C. Input/Output

The input/output subsystem must be designed and operated with the same fault tolerance as the central processing complex. Different modes of operation are possible, depending on the various devices that are connected to and controlled by the system. The favored principle is to use replication wherever possible. Varying capabilities of fault tolerance in the central computing system can be achieved by using varying replication and by voting at all times when valid data are required (e.g., at the start of a task). The results of a calculation will exist in several (usually three) copies and eventually a vote must be taken. The vote that is required to allow another task calculation is carried out in multiple modules; however, if the vote is for output, then the output system or output unit must conduct the final vote.

There are circumstances where the nature of the input/output unit assists fault tolerance through replication, as in the following cases:

- Certain input systems (sensors) can be replicated; each sensor is then individually read (and voted on) by all modules requiring the input.
- Certain output devices can be built in a way that employs a "natural" kind of voting process in the final output medium. For example, a CRT display could be refreshed with each frame derived from a different module. Data on which all modules agreed would be displayed brightly; other data would be more

faint. Assuming that faults persist only for short periods, this would result in a temporary flicker for a few frames before the executive removed the malfunctioning module from the calculation. In the application to which the design is aimed, there are other output devices, e.g., flap controls possessing similar "natural" voting capabilities.

In the event that the device is not in one of the above classes, another "final voter" must be designed that inherently possesses the required reliability. This consideration is independent of the architecture chosen for the central computing system.

We note that the architecture described here can operate in a mode whereby the replicated versions of output data (or the replicated data from input sensors) can be processed by any of the processing modules; hence, no modules need be specially designed for this function.

D. Bus Design

The bus system (B_1, B_2, B_3, Figure VI-1) used for communication between modules must be designed to be fault tolerant. We remind readers that the bus system is used only to allow the processor of one module to read from the memory of a different module. The design need not be such that all bus traffic is checked (as in most other fault-tolerant architectures); however, it should allow a processor the choice of different busses in the event that a bus has failed.

A structure based on a four-port memory module is shown in Figure VI-1. In this structure, each module would have connection between its units (processor and memory). The bus structure, B_1, B_2, B_3, would enable a processor to choose different paths in reading data from the memory units of different modules. It would be appropriate to connect the I/O system to this bus structure. In the event that a four-port memory unit such as shown in Figure VI-1 is not available (or not suitable from other standpoints), then the structure can be achieved by attaching a single-port memory to all busses using conventional techniques.

A processor that needs to read from the memory of a different module must seize control of a bus. Logic associated with a bus must ensure that only one processor has control of a bus at any time. In addition, the bus must be allocated to a processor for only a finite time, thereby preventing a faulty processor from seizing a bus permanently. An internal clock associated with each bus can control the period for which the processor in question dominates a bus. A failure in this control logic causes the loss of that bus. It remains to be shown that no situations can occur where the failure of one unit can cause incorrect action of several other units, i.e. we require a design so that faults remain localized.

In summary, the following sequence of action is carried out in reading data word (w) from memory (m) to processor (p) via bus (b).

1. Processor p places b, m, and w in registers and signals all busses with a DATA REQUEST.
2. All non-busy busses scan all processor DATA REQUEST lines (continuously).
3. If a data request line is on, and b equals the bus number, the bus goes into BUSY state and stops scanning the processors. The requested bus has now been selected by the processor.
4. The selected bus transmits m, w, and DATA REQUEST from the processor registers to all memory modules.
5. Each non-busy memory module continuously scans all busses for a DATA REQUEST line that is on, and then compares the m on that bus with its own number.
6. If a match is found, the memory goes into BUSY state and ceases scanning the busses. The w on the bus is placed in the memory address register and a read request issued to the memory. The memory is now selected.
7. When the word is read by the memory, it is placed on the data lines of the bus and a DATA READY line is turned on.
8. The DATA READY and data are transmitted to the requesting processor. When the data has been received by the processor, the DATA REQUEST line from that processor is turned off.

9. Action 8 will cause the BUSY states (actions 3 and 6) to be dropped and the bus and memory resume scanning for other requests.

In the above sequence each unit that requests action of another unit makes a request (e.g. DATA REQUEST). The granting of the request is made by the requestee. This arrangement will, for example, prevent a processor from requesting all the busses simultaneously, since the busses will respond only if the bus request (b) agrees with their bus number. It would, therefore, require failure of all of the busses to completely disable the bus structure.

In addition to the above, it is assumed that each unit has logic associated with it that prevents it being seized indefinitely. This logic, in effect, says "if I have been BUSY for greater than a time interval DELTA, then the particular connection will be broken and scanning resumed for other units requiring action." It is possible to incorporate in this logic the capability to ignore requests from the offending requestor, in the future thereby removing that unit from affecting further system operation. The time interval DELTA mentioned above will be chosen to be just greater than the greatest time of any correct action request.

The scanner in either the bus or the memory that is examining either the processors or the busses can be implemented so as to scan in any sequence that is convenient (to the detail logic designer), i.e. it is not necessary to scan in numeric order. In the event that the scanner has more states than the number of units scanned (e.g. a 4 stage counter scanning 9 units), the spare positions can be stepped over by suitable design, or alternatively can be left OPEN.

The word address (w) that is transmitted to the memory module can be subject to any transformation that is convenient in the design of the processor or memory, i.e., we can use indirect addressing, indexing, base registers, paging or any convenient combination of these. In addition, it is possible to incorporate a cache (in the 360/85 sense) in the processor design.

The scheme outlined above obviates the need to provide a BURST MODE type of transmission as each word that is transferred can follow the sequence given. In the event that several words are required, the processor successively requests each word and the bus is seized and the word is delivered. If other processors require the use of the bus during the period of the multiple word transfer, a form of cycle stealing will take place as the bus scans the other units and honors the request before resuming scanning.

A suitable structure for the processor/bus/memory connection is shown in Figure VI-4.

E. Program Structure

Within the computing system as described above there exists a program structure, that contains program segments and their interconnections. We distinguish those program segments that must be present in all processor memories from those that exist only in the memory of the processors that are carrying out the function of system executive. We can divide the program functions as follows.

All Processors	Executive Processors
Application Tasks	Allocation
Loader	Scheduling
Dispatcher	Fault location
Fault detection	Fault correction
Fault avoidance	Reconfiguration
Fault location	
Fault reporting	

The dependence of one program segment upon another is illustrated in Figure VI-5a, in which four examples of applications processing, and three of executive processing are shown. (It must be remembered that separate processors are not necessary for each task or for the executive. Each processor is multiprogrammed and may at different times be computing an application program or the executive.) Figure VI-5a shows the interconnection among program segments for one executive and one application processing example. Multiple connectivity is assumed and is illustrated in Figure VI-5b. The several functions shown in Figure VI-5a are described below. We assume for the purposes of the description that threefold and fourfold replication is used respectively both for the executive and each application task. Greater or lesser replication is possible if the requirements for fault tolerance are more or less stringent.

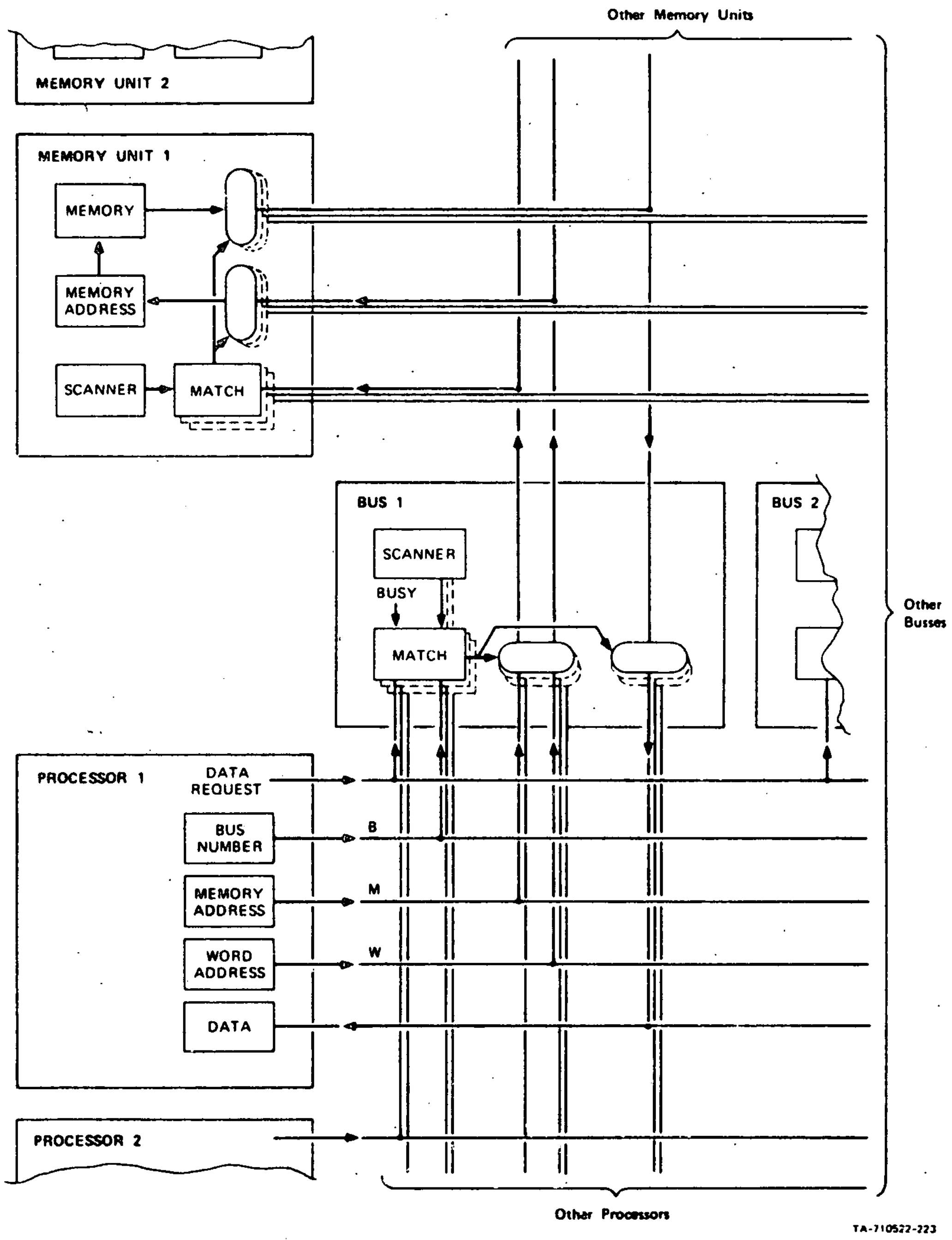


FIGURE VI-4 PROCESSOR/BUS/MEMORY CONNECTION

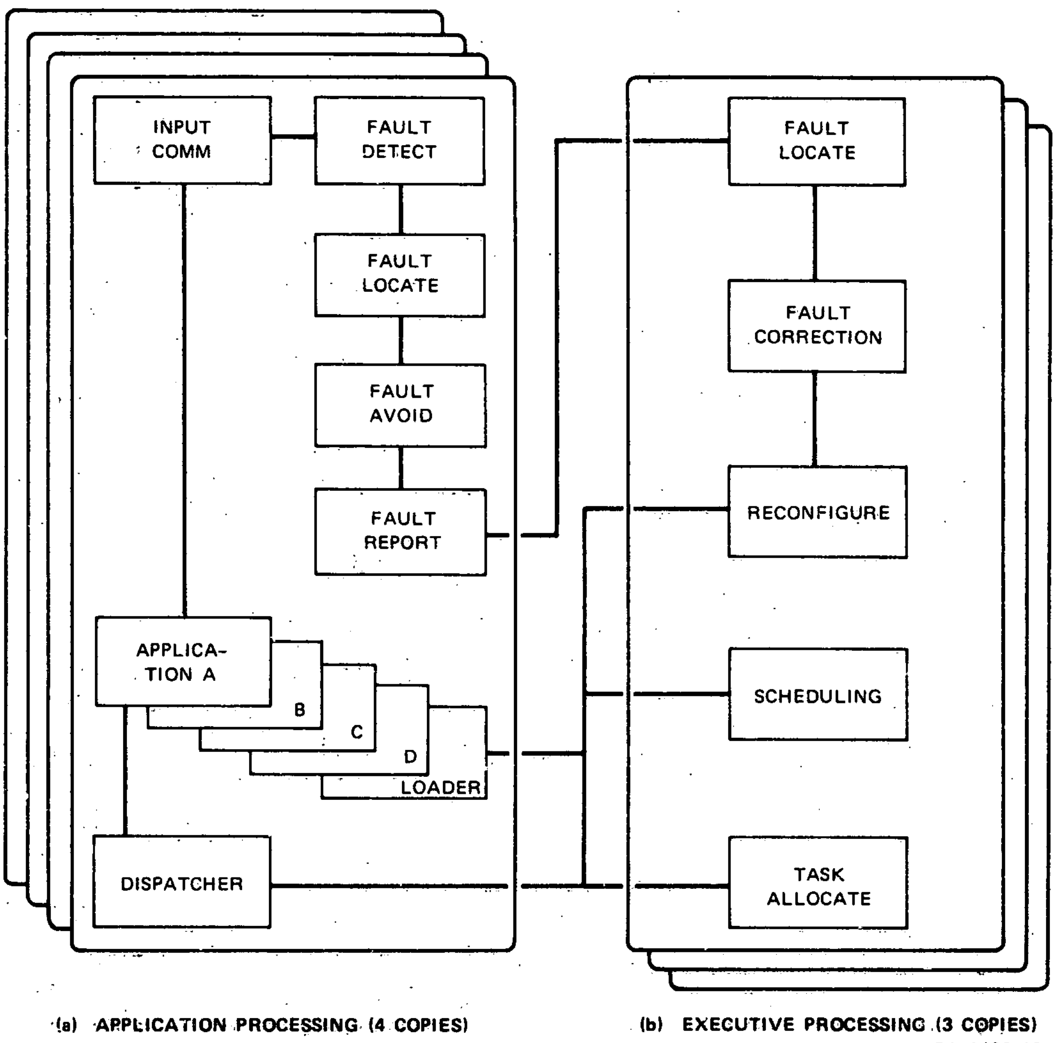


FIGURE VI-5a APPLICATION AND EXECUTIVE STRUCTURE

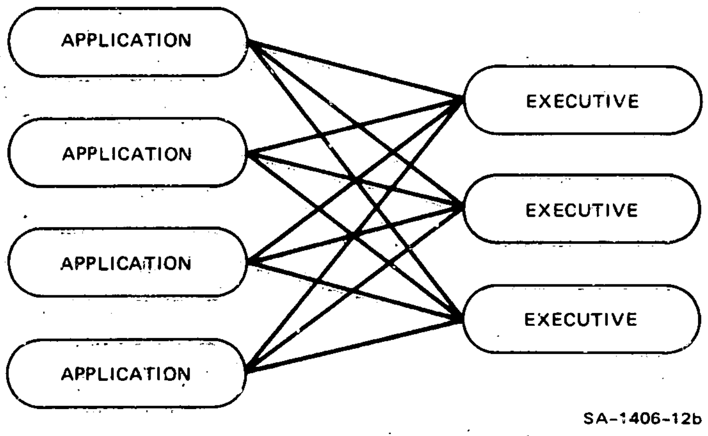


FIGURE VI-5b APPLICATION AND EXECUTIVE CONNECTIVITY

1. System Executive

Allocation and Scheduling

The executive carries out allocation of processors to tasks and scheduling (deciding when task programs should be run). These two functions will be described together, in Appendix A, in terms of the data structures and the way in which they are manipulated.

The task matrix is a Boolean matrix indicating the results of the allocation process. A mark in any element signifies that a processor has been assigned to carry out a particular task. The task matrix will remain unchanged for long periods and will be changed only under two circumstances:

- Change of flight phase (takeoff, cruise, etc.).
- Reconfiguration after a fault condition.

There are three options available in carrying out the allocation function:

(a) Compute new allocations, when needed, in real time.

(b) Pre-compute (i.e., before mission) a set of allocations for each flight phase and for all possible configuration changes.

(c) Pre-compute a set of allocations for different flight phases, and carry out perturbations of these, in real time, upon reconfiguration in the event of a fault condition.

It is expected that alternative (c) above will be used in an aircraft environment.

Reconfiguration

Reconfiguration in the SIFT concept consists of changing the allocation of tasks to processors. This will occur under two circumstances

- Fault conditions that occur, and require the removal of some units from active processing.
- Change of flight phase, e.g., from take-off to climb, to cruise.

No special hardware is required to carry out the reconfiguration, and the software routines are identical to those described above for allocation.

Task Timing

Certain application tasks require action at regular intervals, for example the flutter alleviation calculations must be processed every 4 msecs. For this task, it is necessary to ensure that all the cooperating processors are synchronized with respect to a 4 msec period. The basic rule of synchronization is that no processor should compute iteration (n + 1), nor destroy the data from iteration (n - 1) until all processors have carried out iteration n.

The idea of using a single system-wide clock is rejected as this unit would need to be built with exceptional reliability and would represent a "hardcore" in which no faults could be tolerated.

The preferred hardware would be to use a number of replicated clocks which could be treated as input units, whose data could be read in the same way as any other units.

Using replicated clocks as input units, it then becomes necessary to ensure that all processors read the clocks at sufficiently frequent intervals (e.g., 1 msec.). This can be achieved by an interrupt system in each module, that is driven by another clock (of higher frequency) local to the module. These interrupt systems would be independent from each other, and failure of one of them would be no different from any other failure in a processor.

Fault Location

Fault location in the SIFT concept consists of determining

(i) Which unit is at fault.

(ii) Whether the fault condition is permanent or transient.

Each processing module, when reading data from other modules, carries out a vote as described in Section VI-E-2. In the event that the vote does not yield unanimity, a single "error flag" is set and details of the erroneous transaction are placed in the module's memory. The executive reads the error flags from time to time, and if all are off, correct operation is assumed. If all error flags are not off we must consider two cases.

Case 1 Single Error Flag On

This indicates that a fault exists that is related to that processor, or to the particular connections that were used in the particular data read, i.e. from processor to a particular bus, or from that bus to the particular memory. The analysis of this condition cannot be carried out on the basis of the single instance of the fault. The data must be remembered and further instances of errors correlated with them to determine the faulty unit that must be removed from use by reconfiguration. If no further instances of errors occur in a short period, the assumption may be made that a transient fault condition occurred.

Note that an executive could read an error flag and determine that it is on due to a fault in the processor that is running that executive, or due to a fault in the data path used to read that error flag. The executive would attempt, erroneously, to diagnose an error that did not exist. This will not produce faulty operation of the total system because the other versions of the executive will effectively override the faulty executive.

Case 2 Multiple Error Flags On

Multiple error flags on indicate that several processors have detected an error. The details of the error will be read from the several processors and by correlating the data, the executive may determine which unit is faulty.

2. Local Executive

Input Communication and Fault Detection, Location, Avoidance and Recording

The first step in the computation of any task is to read the data required to carry out the task. This data will exist in the memory of three computing modules. We will use the phrase "Input Data Set" (IDS) to denote the set of words required to carry out the calculation of a task. We envisage that all tasks that require data will obtain it by calling a single program or microprogram subroutine. This subroutine is the only code (outside the executive) that is concerned with detecting errors, correcting them in some cases, and in all cases reporting errors to the executive. By the use of a single subroutine for error detection, avoidance and reporting, the application programmer is relieved of the concern for this aspect of the system. This routine GETDATA is shown in flow chart form in Figure VI-6. Its functional specification is:

<u>Input Parameters</u>

IDS Number	(The identification of which input data set is to be input.)
IDS Size	(The number of words to be input.)
Buffer	(The address of the buffer in which the words are to be placed.)
Proc List	(The address of a list of processor numbers from which to input.)

<u>Output Parameters</u>

Failure Flag	(A Boolean output variable, set = 1 if the input could not be accomplished.)
Error Flag	(A Boolean output variable, set = 1 if input was successful but an error was detected.)
Error Vector	(The specification of the IDS, word position, bus and memory involved in an erroneous input.)

<u>Action</u>

Read an input data set (IDS Number) consisting of IDS size words from the processor memories specified by "Proc List." If all versions of each word obtained from the different processor memories agree, the data is placed in the memory at address "Buffer," the error and failure flag will be set to 0 and a return is made to the calling program.

If all versions of a word do not agree but a majority agreement exists, the data is placed in the buffer, the error flag is set to 1 and the details of the (presumed) erroneous input are placed in memory to be read later by the executive.

If no agreement can be found, the error and failure flags are set to 1, the data is not placed in the buffer and a return is made.

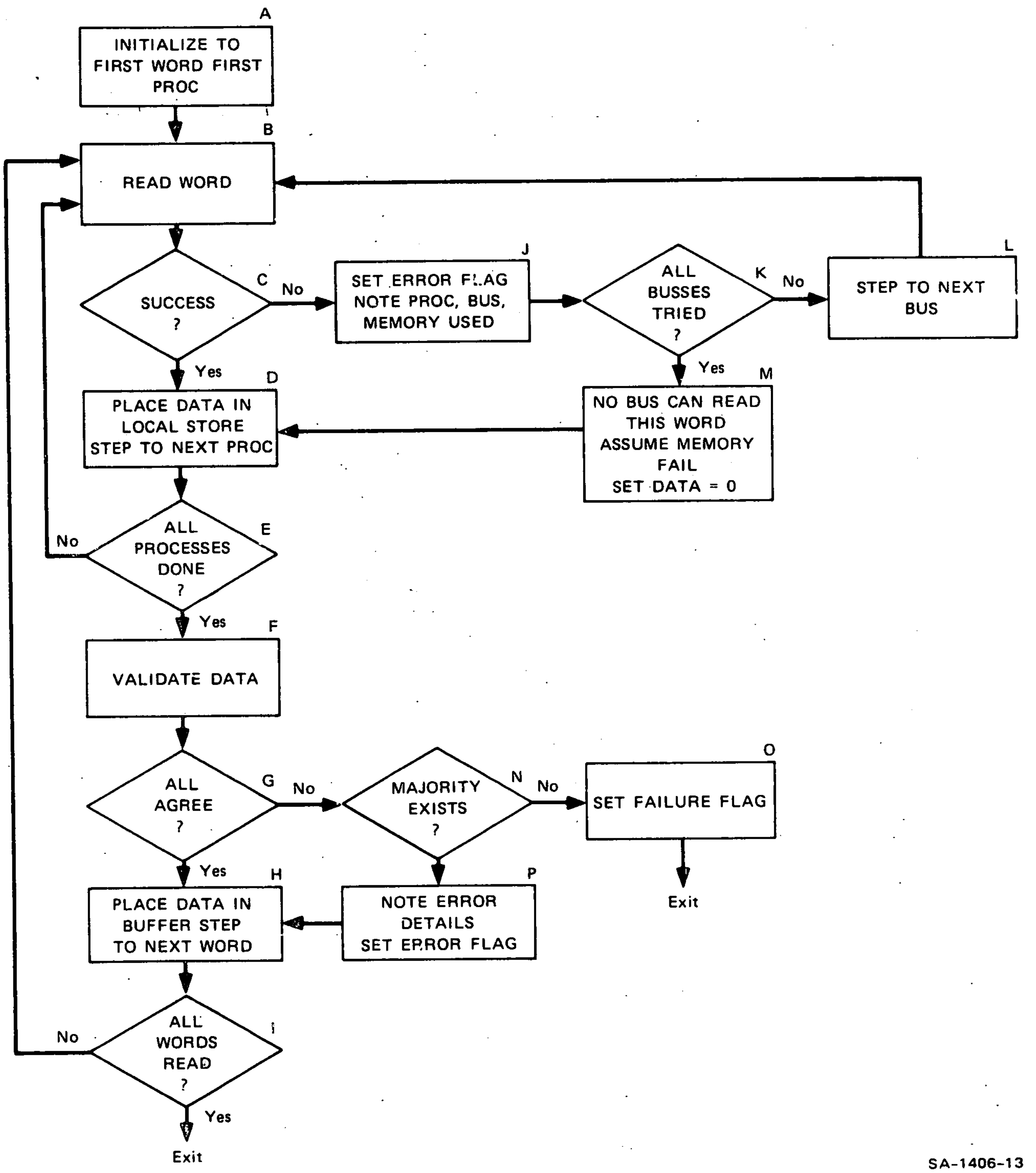

A
INITIALIZE TO FIRST WORD FIRST PROC
B
READ WORD
C
SUCCESS ?
No
Yes
J
SET ERROR FLAG NOTE PROC, BUS, MEMORY USED
K
ALL BUSSES TRIED ?
No
Yes
L
STEP TO NEXT BUS
D
PLACE DATA IN LOCAL STORE STEP TO NEXT PROC
M
NO BUS CAN READ THIS WORD ASSUME MEMORY FAIL SET DATA = 0
E
ALL PROCESSES DONE ?
No
Yes
F
VALIDATE DATA
G
ALL AGREE ?
No
Yes
N
MAJORITY EXISTS ?
No
O
SET FAILURE FLAG
Exit
H
PLACE DATA IN BUFFER STEP TO NEXT WORD
P
NOTE ERROR DETAILS SET ERROR FLAG
I
ALL WORDS READ ?
No
Yes
Exit
SA-1406-13

FIGURE VI-6 THE GETDATA SUBROUTINE

If no action can be accomplished (e.g. because of a faulty bus system), the units that are faulty are noted, and a return is made.

The subroutine will attempt to use different busses for each word transferred. If no response is obtained from an input request, the subroutine steps to the next higher bus.

Analysis of the time required for carrying out the GETDATA routine shows that when no errors occur, 75 percent of the time is used in functions B, C, D, and E, and 15 percent in functions F, G, and H. This assumes that the CPU has no special operations to facilitate these operations. The use of microprogrammable CPUs with special operation codes will significantly reduce the total time to execute this routine. Such special operation codes are discussed in Section VI-H-2.

Dispatcher

The dispatcher carries out the function of initiating the computing of tasks. The dispatcher will always be entered at the end of each task plus being entered regularly on an interrupt basis, probably every 1 msec.

Upon entry, the dispatcher will first query the executive tables to see if the queue of tasks for it has been changed. If so, it reads the queue into its local store. If the item at the head of the queue is the same as a task which has just been interrupted, a return is made to that task program to continue the task calculation. If the task at the head of the queue is different than the task just interrupted, the dispatcher knows that a change of task is required. This may be of two types:

(a) A change of allocation wherein the previous task is no longer required.

(b) A requirement to interrupt the previous task temporarily (for a more urgent task) and resume it when required.

These two cases are distinguished by the dispatcher by the contents of the queue. There exists a task present in all CPUs (the terminator) whose function is to terminate tasks. If case (a) above exists, the queue will contain an entry calling for the TERMINATOR to be run, thereby removing the interrupted task.

The above discussion centered around the actions of the dispatcher in the event that it interrupted a task. When the dispatcher is entered at task completion it does not need to resume an interrupted task, but simply transfers control to that task at the head of the queue.

When the dispatcher is entered at the completion of a task it is possible for it to be interrupted. The interrupt should be ignored except to update the internal software clock within the CPU.

All communication between the dispatcher and the system executive is through the GETDATA routine.

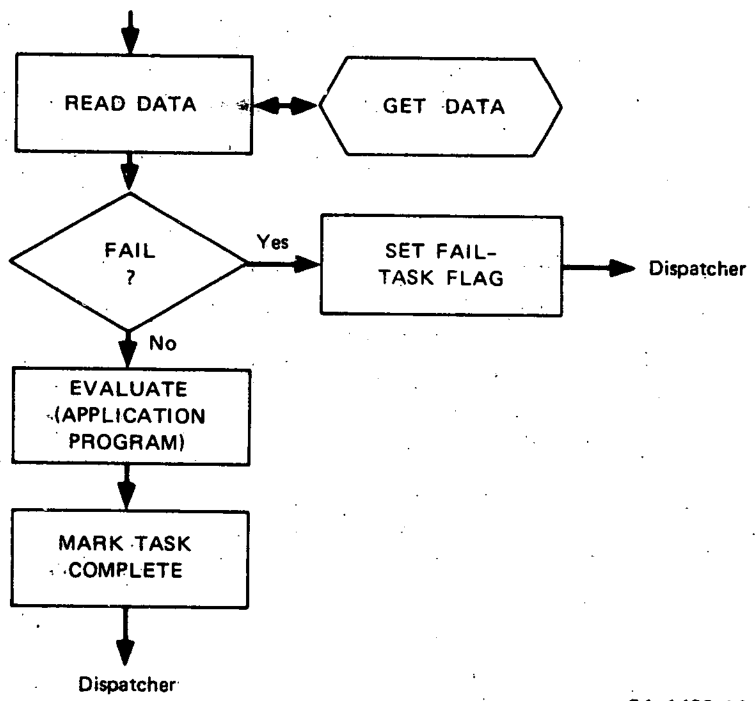
Loader

All processors need to be able to load new programs whenever a reallocation of tasks is made. Two versions of the loader can be considered, a simple version that loads programs from the memories of other modules, and a version that can also load programs from a backup store (e.g. drum) in the event that such a unit is available.

We assume that programs are in absolute binary form, i.e., no assembling, linking or editing is required. We also assume that the CPU/memory hardware enables programs to be relocatable. With these assumptions the loader only has to read a program from the replicated copies and place it in memory. In effect this will be a single call on the GETDATA routine for the version that reads from other memories. For a version that reads from a backup store, a modified version of the GETDATA routine would be required.

3. Application Programs

The application programs carry out such tasks as integrating differential control and navigation equations, formatting graphic displays, and so forth. Each of the programs handle certain functions in a uniform way. These functions include: data communication, error or failure reporting, and connection to the dispatcher and executive. Each application program will be embedded into the SIFT software structure in a uniform way. The typical structure for this embedding is shown in Figure VI-7.



SA-1406-14

FIGURE VI-7 TYPICAL APPLICATION TASK

The function carried out by each application program, and the computing and memory resources required are detailed in the report on Task II of this project.

F. Computing Load Within SIFT

Within SIFT, the computing and memory load are summarized in Tables VI.3a and VI.3b for application and executive tasks, respectively. (Table VI.3a is a repetition of Table III.1, included here for the convenience of the reader.)

The total computer load is as shown below, for the most critical flight phase (instrument landing).

	CPU (MIPs)	MEMORY (KW)
Application Tasks	0.5	24
Local Executive	0.04	1
System Executive	0.1	2

The above table does not include either CPU or memory requirements required for fault tolerance procedures.

G. Reliability and Fault Tolerance

This section discusses procedures for achieving fault tolerance with a sufficient reliability. In addition, a specific case is analyzed to determine the expected performance of the SIFT concept under a set of assumptions concerning such matters as: number of LSI chips in system, reliability per chip, length of flight, and so forth.

The reliability analyses are carried out for different numbers of processing modules.

The system architecture can, by suitable design of the executive, support different fault tolerance procedures appropriate to different requirements. The assumed fault detection method is by comparison of multiple copies of data. This comparison is carried out by software embedded in a system routine, a copy of which is present in all processors.

Fault detection by software voting is compatible with hardware techniques such as parity schemes. Such hardware, if it exists in memories, busses, or processors can be used to assist detection and diagnosis of

Table VI.1a Computing Requirements, Applications

Task			Iteration Rate/Sec	Equivalent MIPS	Store Required	
APPLICATIONS(1)					Inst.	Data
A1	Attitude Control		5,20	.023	1845	230
A2	Flutter Control		250	.069	70	22
A3	Load Control	*	240	.014	45	15
A4	Autoland, Horizontal	*	20			
A5	Autoland, Vertical	*	160	.055	750	275
A6	Autoland, Throttle	*	33			
A7	Autopilot		5	?	150	100
A8	Attitude Indicator	*	30	.077	790	520
B1	Supervisor		?	?	75	15
B2	Inertial	*	1-25	.034	2100	150
B3	VOR/DME		5	.004	250	50
B4	DME, OMEGA		5	?	400	105
B5	Air Data		?	?	110	25
B6	Kalman Filter		1/5	.001	250	65
B7	Flight Data		5	.028	450	100
B8	Airspeed, Altitude	*	8,16	.009	360	70
B9	Graphic Display	*	1,8	.032	890	5360
B10	Text Display		10	.019	640	8700
C1	Collision Avoidance	*	1/3,670	.021	550	600
C2	Data Comm, A/C	*	Various(8)	.006	210	400
C3	Data Comm, Ground	*	$\leq$ 4	.001	450	112
D1	AIDS	*	1/4 to 4	.002	650	650
D2	Instrument Monitor	*	5	.014	800	100
D3	System Monitor	*	1/2	.001	900	50
D4	Life Support	*	< 1/2	.001	900	50
D5	Engine Control	*	33	.119	1300	200

Table VI.1b Computing Requirements, SIFT Executive

Task	When Activated	Number of Operations per Activation	Store Required	
			Inst.	Data
STEM EXECUTIVE				
Allocation	Reconfiguration	50,000	300	(P + 20)x(T + 10) (2)
Scheduling	* 500	100	40	(3)
Timing				
Fault Location	On Error	5,000	30	70
CAL EXECUTIVE				
Input Comm.	* Task Start (9)	20 + 6W (4)	10	20
Error Detection	* On Error	25	50	50
Dispatcher	* Task End (9)	45	40	40
Interrupt	* 500 (5)	23	40	30
Newtask	Change of Task	56	10	Uses F3 space
Loader	Reconfiguration	20 + 6P (6)	10	20+ space of loaded program
BROUTINES				
GETDATA	* By Call	12 + 6W (4)	50	20
Math Routines (Sin, Cos etc.)	* By Call	(7)	200	200

Notes for tables VI.1a and VI.1b

* Tasks required to be run during most critical flight phase (auto landin

(1) Task names are abbreviated. Tables 2 and 3 of the report on task 2 gives full names.

(2) P = number of processors, T = number of tasks.

(3) Data space for scheduling and timing is included in allocation task.

(4) W = number of words transferred.

(5) Figures for interrupt handling, are for the increase in the dispatcher to handle interrupts.

(6) P = number of instructions in program to be loaded.

(7) Operation counts for math routines are included in figures for those tasks which call them.

(8) Data communication A/C represents the input/output load of the computer system. Assuming that simultaneous I/O is possible. No increase in MIPS is required.

(9) To compute load due to task start and stop, the total number of tasks run per second is computed for the most critical phase yielding a figure of 1259 tasks per second. The average data input (W) is assumed to be 10 per task. The MIPS for task start and stop is therefore 1259 (65+60) $\doteqdot$ .15 MIPS.

? Task exerts insignificant load.

faults. The primary advantage of incorporation of hardware checking is to allow faster checking in the event that an application requires faster correction of fault conditions than can be achieved by software.

An important benefit in using software techniques for fault detection and tolerance is that freedom is retained to change the degree of fault tolerance, either because experience gives data on which better methods can be based, or because the different applications require different degrees of fault tolerance, i.e. some are more critical than others.

If threefold replication is used throughout the system a single faulty unit will result in one of the replicated processes computing a wrong result. The use of the wrong result in subsequent calculations will be avoided by the fact that other (correct) copies of the data will exist in other modules and when used will, by voting, enable a processor to distinguish the correct data from that which is erroneous.

A further interesting possibility would be to use non-identical but computationally equivalent algorithms for the several replicated processes. The computation of a variable can also be carried out using different scaling factors, for example, compute 10x, x, and x/10. This will reduce the possibility of error due to bit pattern-sensitive faults.

Consider now the case of double faults existing simultaneously. We must distinguish two cases, uncorrelated and correlated faults. By correlated faults we mean two faults that cause the computation of two equal but incorrect results. Clearly two correlated faults cannot be tolerated if the fault tolerance procedure consists merely of voting among three versions of all results. The probability of such correlated faults will be very low and for most applications is acceptable. We can, however, in the system as described, achieve greater reliability in the event that the application is so critical that this low probability is still unacceptable. Two such strategies are:

- Use threefold replication for all critical applications, and in the event of <u>any</u> disagreement, do not use the results until yet further processors have carried out a repetition of the calculation, for example use two more processors (making a total of five) and only act if three or more agree.

- Use fivefold (or greater) replication[*] of tasks for all critical applications.

Both of the above strategies will prevent double correlated errors from causing the use of a wrong result in subsequent programs or output. The cost penalty involved in the above strategies implies that they will only be used for extremely critical applications, where the cost of extra computing equipment is small compared with the penalty for failure, e.g. in aircraft and space missions.

In the case of double uncorrelated faults we need only consider the case of simultaneous faults. Double faults that occur separated by a time sufficient for the executive to have carried out corrective action after the first fault do not need to be regarded as different than two instances of single faults, which can be tolerated.

Two simultaneous but uncorrelated faults will have the possible effect of producing two different incorrect results from a calculation. These two results will be compared with the one correct result produced by the nonfaulty unit in a threefold replication scheme. Before the result is used in any subsequent calculation (or output), the presence of three differing results will be detected and the executive will initiate greater replication in other processors until sufficient agreement can be found to distinguish the correct from the incorrect result.

The executive of the system must itself be fault tolerant. This is achieved by the same techniques as for application programs. Each of the replicated copies of the executive will use data from itself and the other copies. In the event of errors in one of the executives, the other copies will not use the data computed by it, thereby keeping their results valid. The correctly functioning copies will initiate a new copy of the executive in another processor (which may involve copying the program to that processor) and will signal the malfunctioning processor to discontinue processing the executive. In addition, all processors will, upon inspection

[*]This requires availability of a sufficient number of the various units (processors, memories, busses).

of the data in the correct copies of the executive, cease referencing the data in the incorrect copy, thereby preventing a system breakdown in the event that the malfunctioning processor continues processing the executive even though requested to discontinue.

The fault tolerant procedures outlined above can be summarized as follows:

- Given at least triple replication, all single faults can be tolerated, and all uncorrelated double faults detected.
- Given greater resources (memories, busses and processors), multiple uncorrelated or correlated faults can be tolerated.

It is expected that, in the event of a permanent fault detected, a unit will be relieved of any active part in subsequent calculation. The capacity of the system will therefore be reduced, but until a large fraction of the system is faulty, the fault tolerance procedures can be continued without jeopardy. The removal of faulty units will be accomplished by allocating them to null tasks in the case of processors, and not referencing them in the case of memories. The overall effect of these strategies is to achieve a graceful degradation either of computer capacity or fault tolerance whichever is desired in the particular application.

The foregoing has been concerned with the possible fault tolerant strategies that can be employed within the general SIFT concept. We now examine the specific case where threefold replication is used, and where the computing system is to handle the application tasks and the executive tasks summarized in Section VI-F.

1. Reliability Estimates

A full reliability analysis of a SIFT system will be possible only when the design has been carried to sufficient detail to enable basic reliability parameters to be estimated. These parameters would include a count of number of chips in each module type. In addition, knowledge of the possible failure mode for each unit type is required. The following analysis makes certain simplifying assumptions (yielding what we believe to be a conservative reliability estimate), in order to show that the SIFT architecture can achieve the required reliability. We do not, for example, incorporate the beneficial effects of using error correcting codes in memories, or the possible substantial reduction in memory capacity required through the use of secondary stored or the use of ingenious encoding schemes for memory data.

In deriving an estimate of reliability we make certain assumptions concerning such factors as the probability of chip failure, number of chips required, etc. These assumptions are listed below, together with definitions of the terms and symbols used.

R	= degree of replication employed (usually 3)
N	= number of processor/memory modules
H	= length of computer service required (assume 10 hours)
PR	= probability of system function failure after H time
PF	= probability of no fault tolerance after H time.

The last two terms above distinguish between two ways of expressing the probability of the computer continuing to perform in a satisfactory manner. PR, the probability of system function failure, expresses the chance that an incorrectly performed computer function is carried out during a flight. PF expresses the probability that the system will degrade during the length of a flight to the point where it can no longer tolerate any fault in the remaining operational computing equipment.

We assume a load on the computer system as below

		CPU (MIPS)	MEMORY (KW)
	Application Tasks	0.5 R	24 R
A....	Local executive	0.04 N	1 N
	System executive	0.1 R	2 R

In order to estimate the probability of fault in any module (CPU, memory, bus) we start with the following assumptions.

B...Probability of chip failure = 10^{-6} per hour

C...In the CPU it requires 30 chips to achieve 1 MIP (compare the DEC, PDP-11 which would take 10 chips and yields 0.3 MIPS and the INTEL microcomputer with 2 chips for 0.05 MIPS)

C...Each LSI storage chip provides 0.1 KW of memory.

From the following assumptions we can state

Total memory (KW)	$= 26\ R + N$
Total CPU	$= 0.6\ R + 0.04\ N$ MIPS
Total chips	$= 10(26\ R + N) + 30(0.6\ R + 0.04\ N)$
	$= 278\ R + 11.2\ N$
Memory per module	$= 1 + 26\ R/N$
MIPS per module	$= 0.04 + 0.6\ R/N$
Chips per module	$= 11.2 + 278\ R/N.$

The operating philosophy of SIFT assumes that each processor and each memory be capable of handling any task. The heaviest CPU load of any task is .08 MIPS and the largest memory requirement is 6.3 KW. With the necessity to be able to compute the local executive we have capacities of 0.12 MIPS and 7.3 KW as the minimum capacity of a CPU memory module in a uniform arrangement. This yields a lower bound on the size of each module, and for an economic design (i.e., one that does not have excessive initial spare capacity) it implies a bound on the number of modules.

We now consider the specific case of R = 3 which is expected to be the most common mode of operation. From the assumptions above

$$\text{memory/module} = 1 + 78/N \text{ KW}$$
$$\text{MIPS/module} = 0.04 + 1.8/N$$
$$\text{chips/module} = 11.2 + 834/N.$$

We neglect the chips required for the busses as it is estimated that they represent a negligible ($\doteq$1%) of the total circuits.

Using R = 3, all single faults are tolerated, plus all double faults that are separated by a time greater than the reconfiguration time. Assume the latter to be 1 sec. The probability of two faults occurring in any 1 second interval during a 10 hour flight

$$= \{(11.2N + 834)\ 10^{-5}\}\{(11.2N + 834)\ 10^{-6}/3600\}$$
$$= (11.2N + 834)^2\ 10^{-11}/3600$$
$$= (0.19N + 13.9)^2\ 10^{-11}.$$

The above double fault only causes errors in the event that the two faults occur in a pair of modules that are calculating the same task. If calculating different task sets the fault can be tolerated. For this reason, the figures below represent a conservative estimate of non-tolerated fault condition. Table VI.2 shows the probability of non-tolerated double faults for different values of N.

N	Probability
3	2.1×10^{-9}
6	2.3×10^{-9}
10	2.5×10^{-9}
20	3.1×10^{-9}

Table VI-2 Probability of Non-Tolerated Double Fault

Consider now the question of having sufficient capacity during a 10 hour flight.

Probability of a single error = $(11.2N + 834) \cdot 10^{-5} = P$. Multiple errors may disable several modules or may effect only one module. The probability of losing different numbers of modules is given by* Table VI.3.

Number disabled	Probability
0	$1 - P$
1	P
2	$P^2(N - 1)/N$
3	$P^3(N - 1)(N - 2)/N^2$
4	$P^4(N - 1)(N - 2)(N - 3)/N^3$

Table VI-3 Probability of Failure of Different Number of Modules

* Second order terms are ignored. The error involved is < 1 percent.

Table VI.4 gives the probability of different numbers of computing modules having failed as a function of the number initially in the system.

Number of Modules Initially	Number of Failed Modules				
	0	1	2	3	4
3	9.91E-01	8.68E-03	5.02E-05	1.45E-07	0.00E-00
4	9.91E-01	8.79E-03	5.79E-05	2.55E-07	5.59E-10
5	9.91E-01	8.90E-03	6.34E-05	3.38E-07	1.20E-09
6	9.91E-01	9.01E-03	6.77E-05	4.07E-07	1.83E-09
10	9.91E-01	9.46E-03	8.05E-05	6.10E-07	4.04E-09
15	9.90E-01	1.00E-02	9.37E-05	8.14E-07	6.52E-09
20	9.89E-01	1.06E-02	1.06E-04	1.01E-06	9.11E-09

Table VI.4 Probability of Losing N Modules

Figure VI-8 shows graphically the probability of different percentages of computing power still remaining after 10 hours as a function of the initial number of modules.

Making the assumption that 50 percent computing power must exist at the end of the flight, the shaded region indicated the acceptable operating range, i.e., it indicates that $N \geq 8$ for probabilities $\leq 10^{-8}$ of system failure.

The foregoing analysis is conservative in several ways, which are discussed below.

An implied assumption is that a chip failure will result in erroneous calculation within that module and a consequent requirement to remove that module from service. There are several chip failures that will not cause the above effects. A failure of a chip in the memory will, in general, only invalidate the data that are stored in that part of the memory. The number of chips in the memory exceeds that of the processor by at least 10 to 1 and therefore represents the most probable place of chip failure. Even in the processor some failures will not cause removal of the module from service, for example the loss of floating point capabilities will not prevent the module from being used for the executive and other functions which do not require floating point capability. The design of the executive

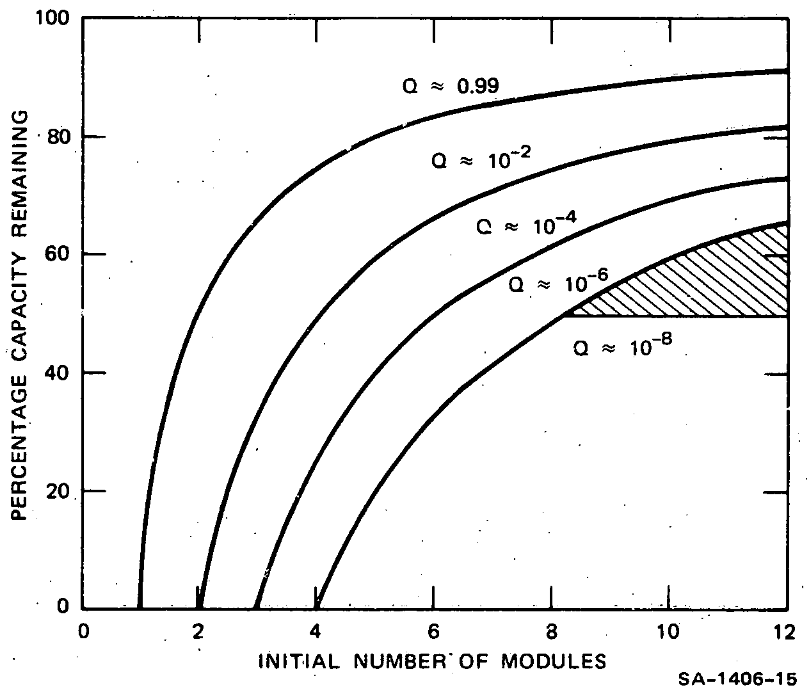


FIGURE VI-8 PROBABILITY (Q) OF DIFFERING PERCENTAGE CAPACITY REMAINING, AS A FUNCTION OF THE INITIAL NUMBER OF MODULES

to take advantage of partial failure of a processor may unnecessarily complicate it, but it remains an option that is available after experience shows the most common failure modes. This type of flexibility distinguishes the SIFT architecture from others where the reconfiguration capability is designed into the hardware.

Greater reliability can be achieved by different memory structures, for example by the use of back-up memory and by the use of coding techniques. The availability of a back-up memory such as disc, drum, or cassette with its low cost per bit would enable extensive replication of programs in back-up memory and allow the main processing memories to be reduced in size thereby requiring fewer chips which will result in a lower probability of faults occurring. Coding techniques can also be used in the memory both for error detection and for correction, thereby improving the reliability of the memories which account for greater than 90 percent of the equipment.

If one assumes that faults in the computer are random, then there will be a significant proportion where the loss of computing power will have occurred before the aircraft flight phase that requires highest computer power. This is during a blind landing and particularly in the last minute before touchdown. Failures that occur before this time, and which are sufficiently massive as to significantly reduce computing power, will only require change of flight plan* rather than endangering the aircraft.

The reasons given above lead to the conclusion that with reasonable chip failure rates, more than adequate reliability can be achieved for the intended application.

H. Processor and Memory

This section discusses some characteristics of the processor and memory units. Full specification (i.e., detailed design) of these units is outside the scope of the present study. Certain desirable features can, however, be distinguished at this time. Questions of speed and capacity have

* For example, to change airports to one in which a blind landing is not required.

been considered in Section VI-F. This section is concerned with structural aspects.

1. Word Length and Addressing

Some application (e.g., Inertial Navigation) tasks require high accuracy of numeric data over a large range. The use of floating point arithmetic is therefore assumed with a minimum of 32 bits (8 bit exponent, 24 bit mantissa) giving a range:

$$\text{maximum positive number} = 2^{(2^7-1)} \approx 1.6 \times 10^{38}$$

$$\text{minimum positive number} = 2^{-2^7} \approx 3.1 \times 10^{-39}.$$

Some reduction of word length could be allowed. It is, however, more economic to keep the word length as a multiple of 8 bits, and a reduction to 24 bits for a floating point number yields insufficient range and accuracy.

In addition to floating point numbers, many data words can be of 16 bits, and character strings can be implemented with 8-bit bytes. We therefore envisage a structure in which data can be either 8, 16 or 32 bits.

With the different data modes, as discussed above, we must consider the possibility of distinguishing between those modes either by labeling the words by extra descriptor bits or alternatively by information carried within the instruction. The latter method implies, for example, that there will be more than one add instruction in the machine to handle the cases of floating point, fixed point, and possibly even single byte addition. Both approaches are possible and acceptable for the application. However, the form in which the distinction is made within the instruction is more common in the computer field and will, therefore, be more available if a choice is made to use already existing designs of processor and memory.

Owing to the requirement to be able to address character strings for such application tasks as display formatting, it is necessary for the instructions to be capable of addressing at the byte level as in the IBM 360 series. This in turn implies a larger number of address bits than if the addressing were only possible at the word level. If one presupposes that a practical word size for the instructions is 16 bits, which would organize well with the various sizes of data words, then we must find a mechanism for incorporating the address bits within an instruction word. Several alternatives are possible and are discussed below.

Alternative A--Conventional machine

In the conventional approach, a number of bits of an instruction word would be allocated to the operation code. A likely number would be 6 bits with the remaining 10 bits for addressing. In most machines that use this structure, the 10 bits can be interpreted to address special parts of memory. For example, the lowest 1024 words, or words in the range $\pm$ 512 words from the current instruction. In addition, the use of indirect addressing enables one to address data through a base sector of memory that can contain words which are interpreted as 16 bit addresses, thereby enabling addressing up to 64K bytes or words. An additional alternative is to use a structure as in the IBM 360 with base registers whose contents are added to the address field of the instruction before execution. This adds a small penalty in time, but gives freedom to address a larger space.

In other machines using the conventional approach, the option exists of using double length instructions whereby the second word contains a 16-bit address, thereby enabling the addressing of a larger memory.

In SIFT, the preferred arrangement among the foregoing choices would be to use base registers for addressing data. The primary reason for this is that a protection mechanism can easily be implemented so that the base register itself can only be set by privileged instructions within the local executive, thereby preventing any application task from inadvertently addressing data to which it was not authorized.

Alternative B--Stack-oriented machine

In a stack-oriented machine, addresses for data can be placed on the stack before an operation such as fetch, or store, or jump is executed. Thus only one instruction is required that can contain an address. (This unique instruction can be labeled by a single bit.) Successive words can fetch, store, or whatever, and since these words do not need an address, most of the operation codes can be implemented within 8 bits. The option therefore exists of allowing two operations per word whenever it is not necessary to place another address on the stack. For example, see Table VI.5 where the evaluation of an assignment statement is shown coded in 16-bit words.

Mathematical form	$r = \sqrt{x^2 + y^2}$	
High level language	R = SQRT (X * X + Y * Y)	
Assembly language	Load Address	R
	Load Address	X
	Load Address	X
	Multiply	
	Load Address	Y
	Load Address	Y
	Multiply	
	Add	
	Load Address	SQRT Routine
	Enter Subroutine	
	Store	
Machine Code Form		bit
	1	2 16
	1	R
	1	X
	1	X
	0	*
	1	Y
	1	Y
	0	*
	0	+
	1	SQRT
	0	JUMP
	0	STORE

Table VI.5 Stack Organized Computer Organization

In a stack machine, indirect addressing and indexing can be carried out and in the latter case, the value of the index is itself placed on the stack. In effect, the indexing operation becomes identical to the normal add operation for data. Namely, the index is placed on the stack, followed by the address. An addition then takes place followed by, for example, a fetch from store. The effect of the addition is to add the value of the index to the address which accomplishes the desired result. Indirect addressing is achieved by simply using the instruction sequence that will cause an address to be placed on the stack followed by an arbitrary number of fetch instructions, each of which puts a word at the top of the stack. If that word is to be interpreted not as data, but as an address, an additional fetch instruction will yield the contents that are pointed at by that word.

The stack organization has great advantages in that addressing of 32K words in a 16-bit machine can be achieved simply. It has certain disadvantages, however, primarily due to the fact that it is a less accepted form of machine organization and is not easily available in existing computers.

The choice between the two alternatives above does not change the basic concepts of the system organization which can be implemented using either structure of machine. Other considerations, such as the availability of microprogramming facilities and with it the ability to create special operational codes, far outweigh the differences between the two alternatives in terms of the choice that must be made.

The above discussion shows that acceptable designs can be produced in which small computers (i.e., 16-bit word computers) can be designed that have a capability to address a sufficiently large memory.

2. Special Operation Codes

Given the availability of a microprogramming capability in the computer, we can implement special operation codes that facilitate either the application tasks or the executive. We are particularly concerned with the efficient implementation of those operation codes that transfer data between modules and carry out checking of that data. For example, in a general-register machine we would require an operation that could take three versions of some data that should be the same and carry out in one operation a check to see if this is indeed so, with certain control flip-flops set to indicate whether complete agreement exists or partial (i.e., two out of three) agreement exists. In addition, operation codes could be implemented to enable flexible control of the input/output function when several versions of data have to be gathered from different memories using different busses. In effect, we are looking for a more sophisticated channel for this control of I/O. However, for small machines, such channels frequently are controlled directly from the instruction stream within the processor.

Within the application tasks, the only operation code requirements that are different from normal could be for the programs that manipulate the graphic and textual displays. These two application tasks both require large programs and also large amounts of data. Questions of special operation codes to make this more efficient should be considered.

The use of special arithmetic operation codes (e.g., for square-root, sin, cos) has been studied, but for the application being considered none of these functions require sufficient CPU or memory capacity to justify their inclusion unless this can be achieved with trivial cost.

I Alternative Design Options

In Section VI.8 we discussed some of the requirements on the structure of the processor and its connection to memory. In this section we consider some of the ways in which the structure of SIFT could be changed while not invalidating the basic concepts. We first reiterate one of the fundamental features of the SIFT organization, namely that in order to prevent fault propagation, a processor can only write into its own memory. We have suggested implementing this by a bus structure that contains only one-way paths from memories to other processors. There are other possible implementations. For example, a single monolithic memory could be used and the protection needed could be carried out by constraining the address space of each processor so that only a small segment of memory can be addressed for writing.

Another way of looking at this type of memory structure is to view it as one in which each processor has a larger address space for reading than writing and the writing spaces of processors are non-overlapping. This can be implemented by an instruction set that has a smaller number of bits in the address field for instructions that write than for those that read from memory. The advantage of such a memory structure is that in the event that a processor becomes faulty and has to be removed by the executive from the system, the memory associated with that processor could be reused at least in the reading form by other processors, thereby preventing the need for transfer of programs from one module to another in the event of a processor failure.

As discussed in Section VI.6, the minimum size of store in any processing module is largely determined by the size of the largest task to be carried out, plus the size of the local executive. The largest jobs in the application task set that we consider are for graphics and textual display. Most of the storage space used for these applications is concerned with the retention of information that is infrequently used, i.e., we must remember the configuration, not only of the airfield to which the plane is flying, but also to back-up airfields. We must similarly remember the configuration of the various runways of that field, even though eventually a choice is made for a landing on a particular runway and these are the only data that are eventually used. Choices of different airfields or different runways are human choices and the time response to be able to change the display for a different airfield or runway is very long in computer terms. Certainly, there appears no strong necessity to keep these data in storage with access time of the order of microseconds. The availability of back-up stores that could possibly be of a rotating magnetic medium type, such as drum or disk, would enable much smaller processors to be used in the system.

This would mean that a larger number of smaller processors would be used that would provide a more reliable and fault-tolerant system. Further investigation should be carried out to determine the practicality of using some back-up store for information of this type.

An additional problem is the size of the program required to carry out the formatting of dynamic displays and the computational time required for these programs, A design option would be to significantly enhance the display hardware itself so that it was capable of carrying out more automatic generation of display. An example of this more automatic operation might be an ability to interpolate between two displays so that the computer need only transmit updated views from time to time, (e.g., every second) and the hardware would carry out an interpolation so that the transition from one to the other was carried out smoothly, simulating more clearly the actual operation of the plane.

We note that considerations of this type regarding the display hardware are independent of the architecture of the computer and whichever central computing system is chosen will benefit from the ability to have either a back-up store for data or a more sophisticated display system.

In the context of architectural alternatives, we note again that there are significant alternatives among fault-tolerance procedures. While the most common mode will probably be three-fold replication, a higher degree of replication may be appropriate for some tasks. In addition, we can carry different error correction procedures. For example, in the event of an error being detected, instead of taking the majority vote remaining, we can repeat the calculation by the same or a different group of processes. It is felt that this is a useful procedure during the critical phase of a flight.

A large component of computing capacity is needed for automatic landing. Only a very small proportion of flights will occur where both automatic landing is needed and faults have reduced the computing equipment to the minimum necessary to carry out that operation. For those cases where the faults have not occurred, it would seem natural to use the spare capacity to provide extra fault tolerance. In the event that the spare capacity is not available, we can opt to change the mission, i.e., to land at an alternative destination. Such options as these within the SIFT architecture make it different from architectures whereby the fault-tolerant procedures are designed into the hardware and are not changeable dynamically at run time.

J. Conclusions

The SIFT system architecture as presented achieves great flexibility in fault-tolerance procedures. The salient points of the design objectives that are achieved are:

- Fault tolerance can be varied so that for some tasks it can be arbitrarily high, using suitable replication and reconfiguration strategies, and for other tasks the fault tolerance can be less.
- No special design requirements are placed upon the processing units or memories, thereby enabling different designs to achieve different computer power.
- Fault-tolerance procedures can be implemented by software, microprograms or hardware.
- Fault detection, avoidance, and correction functions are achieved by procedures that can be transparent to the application programmer.
- The reliability required can be achieved by assuming reasonably reliable LSI circuits and three-fold replication.

VII BUS CHECKER SYSTEM

A. Introduction

This chapter describes a third candidate architecture, the Bus Checker System (BUCS). We start by summarizing the implications of the aircraft environment on the required fault tolerance and performance:

1. The application computations are largely independent of each other with respect to
   . the order in which they are to be executed
   . the sharing of data and common subroutines
   . peripheral devices (generally each task is associated with private sensors and effectors.)

2. For the critical computations it is essential that the computer not deliver wrong results; it is preferable that no result be delivered rather than a wrong result.

3. For certain high iteration rate computations (flutter control, automatic landing, collision avoidance) it is essential that the system not be down for more than 10 msec (a few iterations). Generally, for these computations the pertinent program and data base ($\approx$ 4K words), a down-time of 100 msec or more can be tolerated.

4. Some graceful degradation is possible. During landing about 50 percent of the computations (measured in terms of processor load and memory requirements) need not be considered. Hence the plane can land according to mission plans with only half of the computer resources available. However, the only positive ramification of this is that if a back-up memory is available to hold the landing programs during the non-landing portions of the flight, then these landing programs can overlay other programs at the time of landing.

5. The relative independence of the computations leads naturally to a multiprocessor system -- a good approach to follow, in any event, to achieve fault tolerance. As such, each processor can be of low power (say, 0.1 MIPS), which in turn implies that each processor can be realized with few (say 3) LSI chips. Implementation in a few LSI chips implies that redundancy and error checking should be applied over a processor rather than within it. Assuming that failures are independent among chips, the frame coding approach for memory seems natural. As will become clear, the independence criterion is not as important for processors wherein many failures that disable several chips can still be tolerated.

The BUCS system embodies the following qualities:

1. The main memory is centralized to allow the full benefits of frame coding. In addition, program and data sharing and relocation are easily attained within the concept.

2. The processor load is divided among a set (five to ten) of small local processors (LP), which are duplexed or triplexed for error detection/correction purposes.

3. Each LP contains a small (2 - 4K word) local memory. This memory is large enough to hold all of the program and data associated with any task (except display) so that the common bus traffic is low. The memory is loaded from main memory with the appropriate program just prior to each task execution.

4. A small (about 3 chips) unit, called the bus checker (BC) coordinates the flow of control between the major blocks. It also does initial processing of error signals. The BC is triplicated, and its outputs are fault masked by voters distributed at appropriate locations.

5. A disaster restart mechanism is provided within the BC subsystem. This mechanism permits a relatively simple recovery from a massive transient that may corrupt the executive tables.

6. Diagnoses of main memory and of certain unflexed processor functions (e.g., comparators) are carried out periodically. Spare blocks, or possibly memory frames, can be utilized if the diagnostic routines reveal faults.

7. A two-level executive is postulated. The first (higher rate) level controls task sequencing and error checking. The second (slower rate) level controls modifications in task sequencing and periodic diagnosis.

8. It is envisioned that the executive will permanently reside in one of the LP's until the LP fails, after which another LP will take on the executive role.

The reader will observe that the BUCS system as summarized above incorporates some features of the surveyed systems. The concept of local duplexed processors and centralized memory are parts of the Hopkins' multiprocessor scheme. Coding as the primary protection for memory is part of the JPL-STAR and IBM-MARCS. A special bus checking block is part of the NASA-Marshall-Hughes ARMMS system. The loading of task programs into local processors from a central store is a property of the Navy AADC system.

In the sections below we present the over-all view of the architecture, a scenario for each major system function, tentative design features of the major system blocks, a preliminary reliability analysis, a brief review of possible extensions to the BUCS system. Chapter VIII presents a detailed comparison of BUCS with the other two candidates.

B. High-Level Description of Architecture

1. Hardware Components

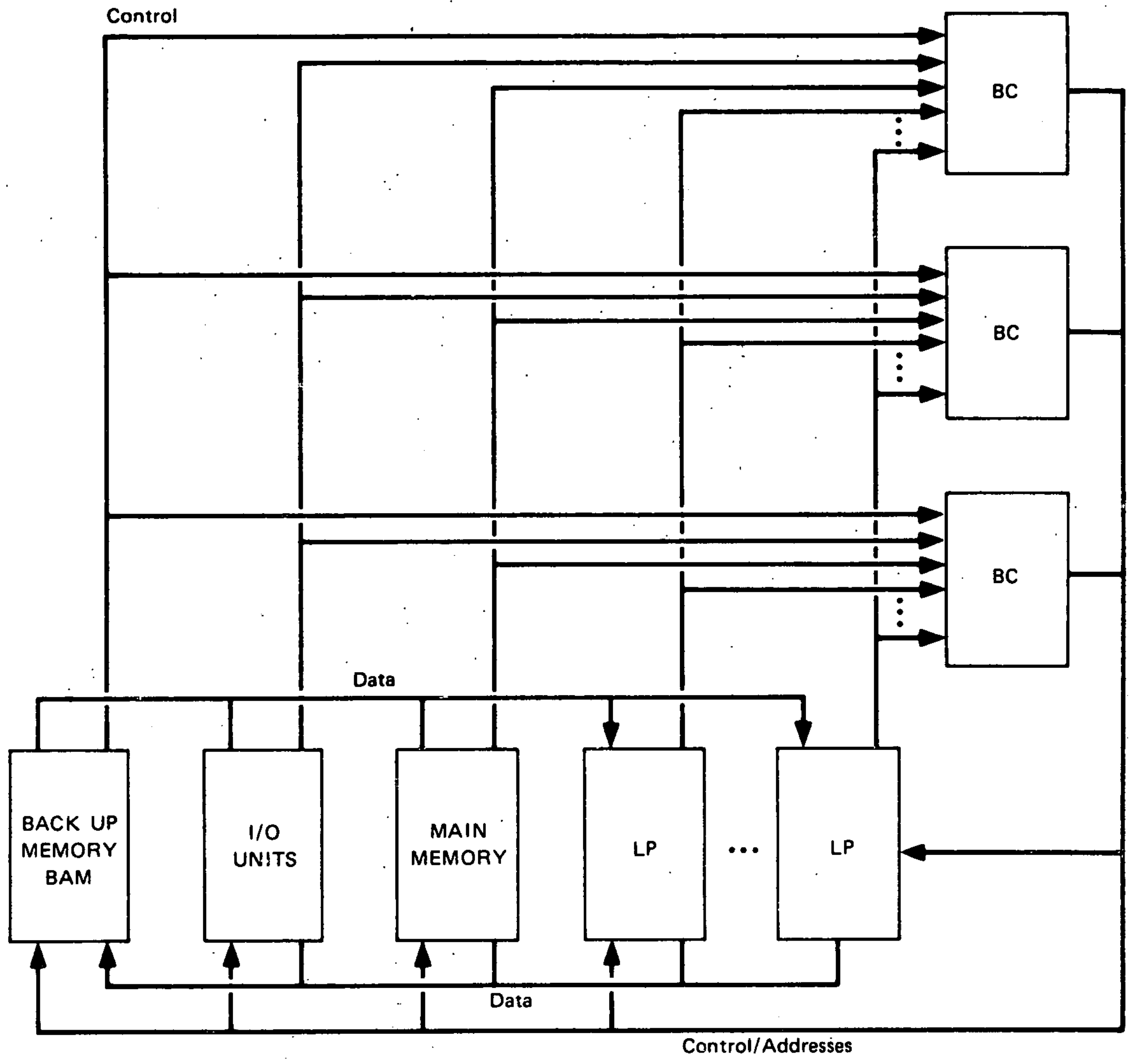
As shown in Figure VII-1, the BUCS system consists of the following major system blocks: a set of local processors (LP's) (five to ten, 0.1 MIPS each, for this application), a centralized main memory (MM), a set of I/O units, a bus checker (BC), and possibly a back-up memory (BAM).

The BC is assumed to consist of three identical independent units operating in near locked-step.* As shown in Figure VII.1, each independent unit delivering information to the BC has a separate set of input ports to the BC, and each of the three independent BC's can deliver signals simultaneously to all units, on a single triplicated bus. The separate units will generally contain their own address decoders to recognize BC signals and their own voters to correct any single BC errors.

The MM is realized as a two-dimensional array of frames and blocks (see Chapter II) wherein frame coding is used. A spare block under the addressing control of the executive is also included to provide additional required reliability. A set of LP's are included to enable the simultaneous processing of several programs, and also to provide spares.

Within an LP are two independent locked-step processors (P) with a deplexed comparator. The comparators, which, physically can be part of the processors, broadcast any processor disagreement to the BC. The local memory (LM) within each LP is relatively small, since 4K words are completely adequate to handle the executive and any of the individual computations except display (which can be easily decomposed into subtasks). Most of the tasks can be handled with 2K words, so it is likely that a 2K LM will suffice with the larger tasks using the MM as a backup memory to the LM. Single

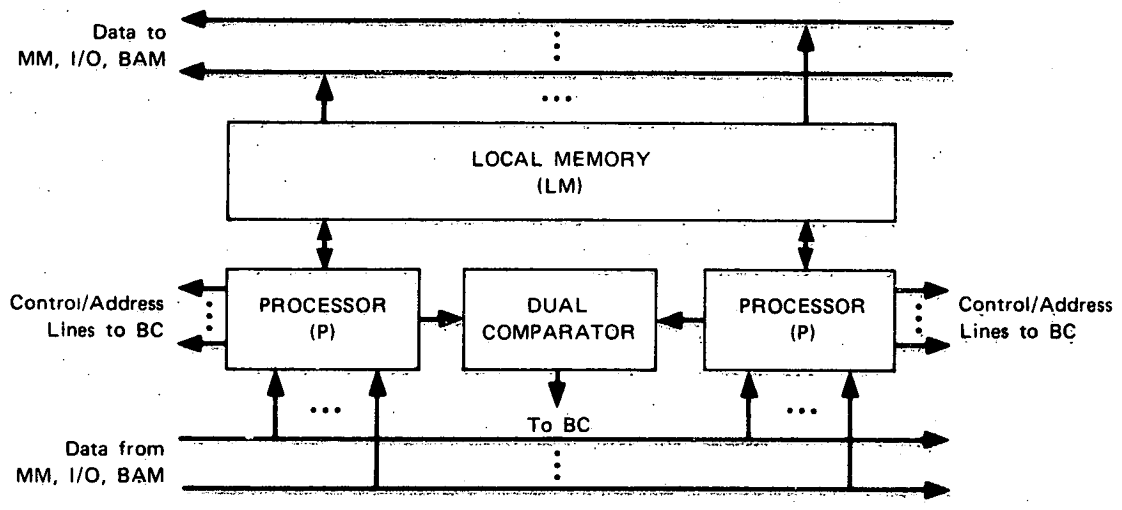
*As we show later the BC can be realized with 3 LSI chips. Since this block is so small a reliability analysis will show that triplication with voting yields sufficient reliability.



(a) BASIC ORGANIZATION

SA-1406-16a

FIGURE VII-1 HIGH LEVEL VIEW OF BUCS



(b) **LOCAL PROCESSOR ORGANIZATION**

SA-1406-16b

FIGURE VII-1 HIGH LEVEL VIEW OF BUCS (Concluded)

frame-error correcting, double frame-error detecting codes are utilized to increase the reliability of the LM. One notes that the LM comprises about 80 percent of the LSI chips of the LP.

An LP will fail when (1) a second frame within the LM fails, or (2) one of the two P's fails, as determined by a comparator disagreement, or (3) a comparator fails, as determined by a disagreement between the comparators or by a diagnosis. The failure of an LP will induce its replacement with a spare LP. At first glance it might appear extravagant to discard an entire LP even though most of its memory and at least one P are still operative. However, as the reliability analysis will show, only one spare LP is necessary, and moreover a single LP can be as little as 10 percent of the entire system.

The issue with respect to I/O is not as clear-cut. It is certain that multiple I/O controllers will be present so that a given sensor or effector has access to the bus checker through more than one controller. With three controllers, which seems reasonable, there can be a distinct controller communicating with each BC. (The I/O controllers can be viewed as merely extensions of the BC's.) If there is triplication of the sensors then independent sampling of the sensors can easily be accomplished.* Similarly, if there can be three independent effectors for a given aircraft function, each such effector can receive an independent signal -- the "voting" in this case is accomplished by the aircraft frame. If only one effector is possible, then the voting of the three I/O controller outputs is accomplished at the effector -- clearly the last possible point for the vote.

We have not given much attention to the characteristics of the back-up memory (BAM). Our present view is that there is not a clear role for a BAM

---

* There is a problem with carrying out a vote of independent sensor readings. It is unlikely that the three sensors will supply exactly the same value, and hence the vote, if done in a bit by bit manner, will fail. The solution is to supply the three readings to the pertinent LP's and have the vote taken in software, thus allowing for a disagreement precision warranted by the sensor in question.

in our system. The storage requirements do not imply a need for mass storage, except possibly (1) for the logging of flight information, (2) for the storing of runway parameters for all possible landing sites, or (3) for the storing of programs and status information in anticipation of a massive transient. The technology survey has not shown a clear reliability advantage or substantial cost advantage for, say, discs, as compared with MOS memory. If it is needed, a reasonable approach for the storage of critical information is to use two independent BAM's, each with error detection, and each communicating with the set of BC's.

We have not yet decided on the width of the communication paths between units. As might be expected, the bus checker (BC) is pin limited. A narrow bus is thus preferable as it leads to a smaller BC unit (fewer chips). The total data rate to and from (mostly from) the MM to the BC, for the aircraft computations, is estimated as 2M bits/sec. This is a factor of 10 less than required by a version of the Hopkins scheme that does not incorporate a cache. The reduction is due to the fact that the bus in BUCS is mainly used for loading the LM's with programs and constants, while in Hopkins, the bus is used for program execution. Assuming 16 bit words and, say, a transfer width of 8 bits, the bit rate on the bus is 250K bits/sec. (Note that the I/O bit rate through the BC is negligible.) With any reasonable interconnection technology an order of magnitude scaling upward in computation load can be tolerated without taxing the bus.

2. Global Description of System Operation

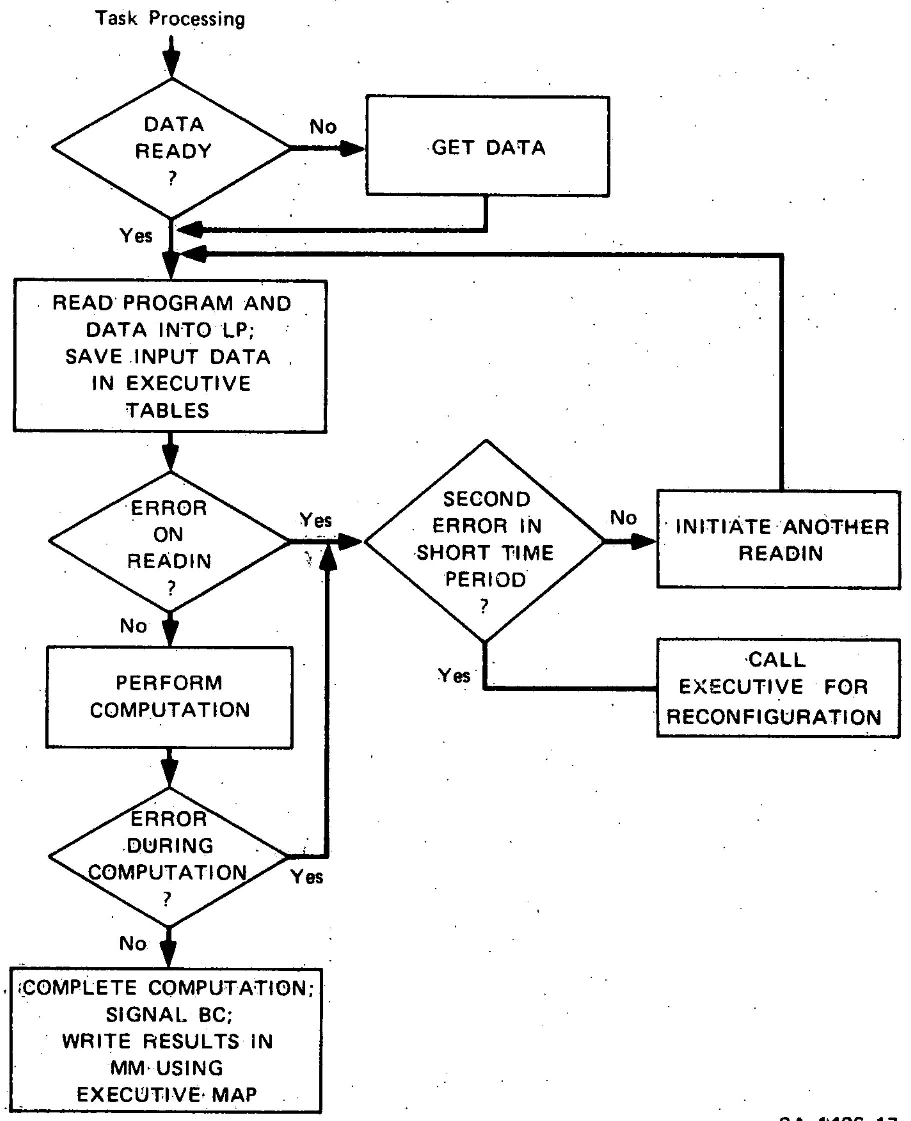
The following is a brief overview of the system operation. A detailed description of each of the executive functions appears in Section D below.

As in the SIFT system, computation tasks are assigned to LP's. However, in BUCS the program, constants and data are retained in MM until the task is initialized. The pertinent information is then transferred to the LM, by means of a mapping provided by the executive. That is, the executive in initiating a task supplies an absolute MM address and a word count for the program and data of the task in question, which are then loaded into the LM under control of the BC. Tasks should be allocated to LP's so as to preclude the need for multiprogramming within an LP. This may be accomplished by assigning to a given LP either all short, high iteration rate tasks or all long, low iteration rate tasks. As in the Hopkins scheme (but different from SIFT) a task is assigned only to a single LP.

If an LP fails permanently, all of its application tasks are assigned to a spare LP. As the flight progresses and the task sequencing changes, the executive will reassign tasks to LP. It is likely that all of the assignments needed for a flight can be programmed in advance so that the executive function for a reallocation of tasks is merely a table look-up.

The executive is initially resident in one of the LP's, and it will probably not share the LP with application tasks. A replica of the executive program and tables is retained in MM so as to permit the reassignment of the executive to another LP upon failure. The BC retains knowledge of the identity of the executive LP in addition to the absolute MM location of the executive program. A failure of this LP is counteracted by the BC assigning another LP to take on the executive role. The occurrence of a massive transient could prevent the "normal" error response as indicated above. For example, the register that holds the identity of the executive LP could be corrupted in two BC's. If this undesired state change as well as others were to occur, then it is likely that several LP comparators would issue disagreements, but the executive would not be called to respond to these possible errors. The BC, if a succession of errors are reported to it, responds with a hard-wired routine that selects a new executive and attempts a dead-start recovery.

A summary of this global view of the BUCS operation is depicted in the flow chart of Figure VII-2. The details of each of these operations and of the executive appears below in Section D.


Task Processing
DATA READY ?
No
GET DATA
Yes
READ PROGRAM AND DATA INTO LP; SAVE INPUT DATA IN EXECUTIVE TABLES
ERROR ON READIN ?
Yes
SECOND ERROR IN SHORT TIME PERIOD ?
No
INITIATE ANOTHER READIN
Yes
CALL EXECUTIVE FOR RECONFIGURATION
No
PERFORM COMPUTATION
ERROR DURING COMPUTATION ?
Yes
No
COMPLETE COMPUTATION; SIGNAL BC; WRITE RESULTS IN MM USING EXECUTIVE MAP

SA-1406-17

FIGURE VII-2 HIGH LEVEL VIEW OF BUCS OPERATION

C. Description of Major Blocks

In this section we present some design details of the major system blocks. The designs have been carried out only to the point where we are convinced of the viability of the entire approach.

1. Local Processor

The tentative details of the LP are displayed in Figure VII.3. As indicated, it consists of two replicas each of a processor (P), encoder/decoder, and comparators and a single replica of a LM. It is anticipated that each encoder/decoder would be realized as a single chip. Each processor, together with its comparators would probably consist of two to three chips. The LM, which will probably require no more than 4K words--of 16 bits plus six check bits--should consist of no more than 22 chips. The optimum coding for the LM is frame coding, with frame-width f = 2, and with a Hamming single frame error correcting, double frame error detecting code. We recall that the words in Main Memory (MM) are also encoded as a Hamming frame-error correcting code. Hence the encoder/decoder within an LP can suffice for the code conversion between the LP and MM. That is, words being transferred between the LP and LM always pass through the LP encoder/decoder. Thus an error in either the LP or LM is immediately identified, corrected, and pinpointed. Another possibility, albeit not as attractive, is to use the LM as the interface between the LP and MM. In this case, the words passing between the two units will be encoded properly by definition.

We recall that single frame errors within the LM are masked, requiring no additional reconfiguration action. The occurrence of a double frame error causes each encoder/decoder to send an error indication to the BC, signifying the failure of the LP. Similarly, a disagreement between two processors also is an indication of LP failure.

In Figure VII-3 we have shown exclusive-OR gates to indicate the need for a test for disagreement between the P1, and P2 units. However, this somewhat begs the issue. We believe the best solution is to effect a comparison whenever a signal leaves either P1 or P2. We clearly do not need to compare

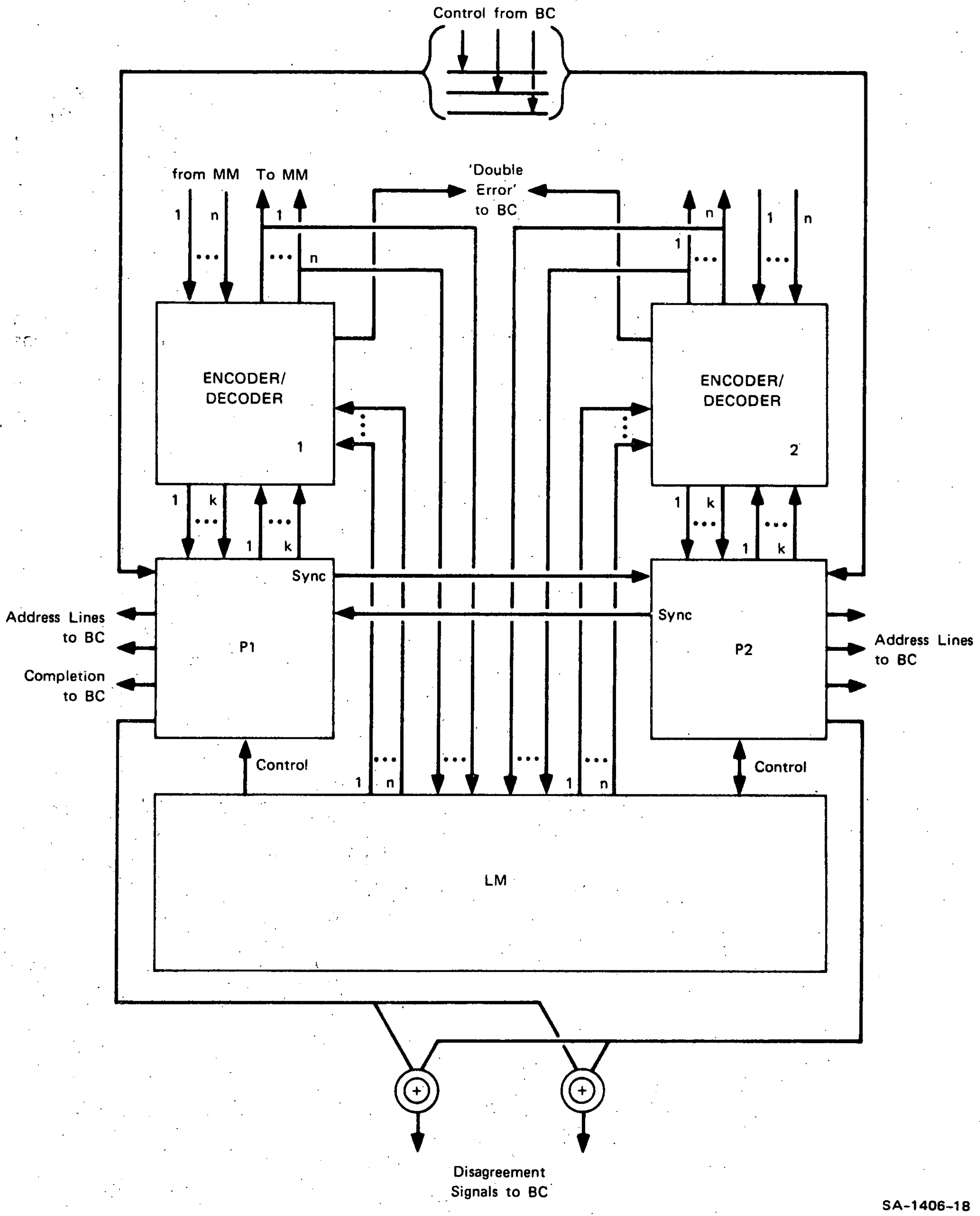


FIGURE VII-3 DETAILS OF LOAD PROCESSOR

at each cycle of P since roll-back is not achieved as a single instruction restart but as a task initiation restart. Hence, the comparisons could be done on the completion lines or on the address lines going from P1 and P2 to the BC's. Similarly, a comparison could be effected on the words from P1 and P2 to be written into LM.

As mentioned previously the control signals from the BC are triplicated--one for each BC--and hence within the P's there is an error-correcting voting mechanism. Except for coded data words, signals emerging from the LP destined for the BC are duplicated--one version for each P--and hence there is a comparison mechanism within the BC that will signal a possible failure if there is a disagreement.

The two P units are synchronized but only loosely. We envision a clock control line between the units that is activated only when a comparison is called for, say when a completion signal is to be sent to BC. Thus, a reliable clock is not needed. If, say P1 (or P2) fails to receive a control signal from P2 (or P1) within a certain tolerance then it signals an error condition to BC. The inter-unit control signal could also resynchronize the clocks within each P so that the clocks do not drift appreciably apart. Similarly the communication between an LP and BC is done asynchronously. Thus a reliable system clock is not needed.

2. Main Memory Organization

An appropriate main memory (MM) organization is depicted in Figure VII.4. We have also shown the form of the interconnection between the MM and other system units. Briefly, the address lines and clock control lines are derived from the bus checker (BC). We assume, as shown, that a separate voter is provided to supply address data for each frame of memory within MM, and the voter is driven by the three BC's. The voted address lines are passed to all chips in the MM since each such chip contains an internal address decoder. Note that a single failure such as one chip "shorting-out" (at its input) a voted address line will disable only one frame of the MM. Consequently the coding of the MM words will handle this fault. However, there are double faults that can disable the MM. For example, the failure of two frame address voters will potentially corrupt two frames of

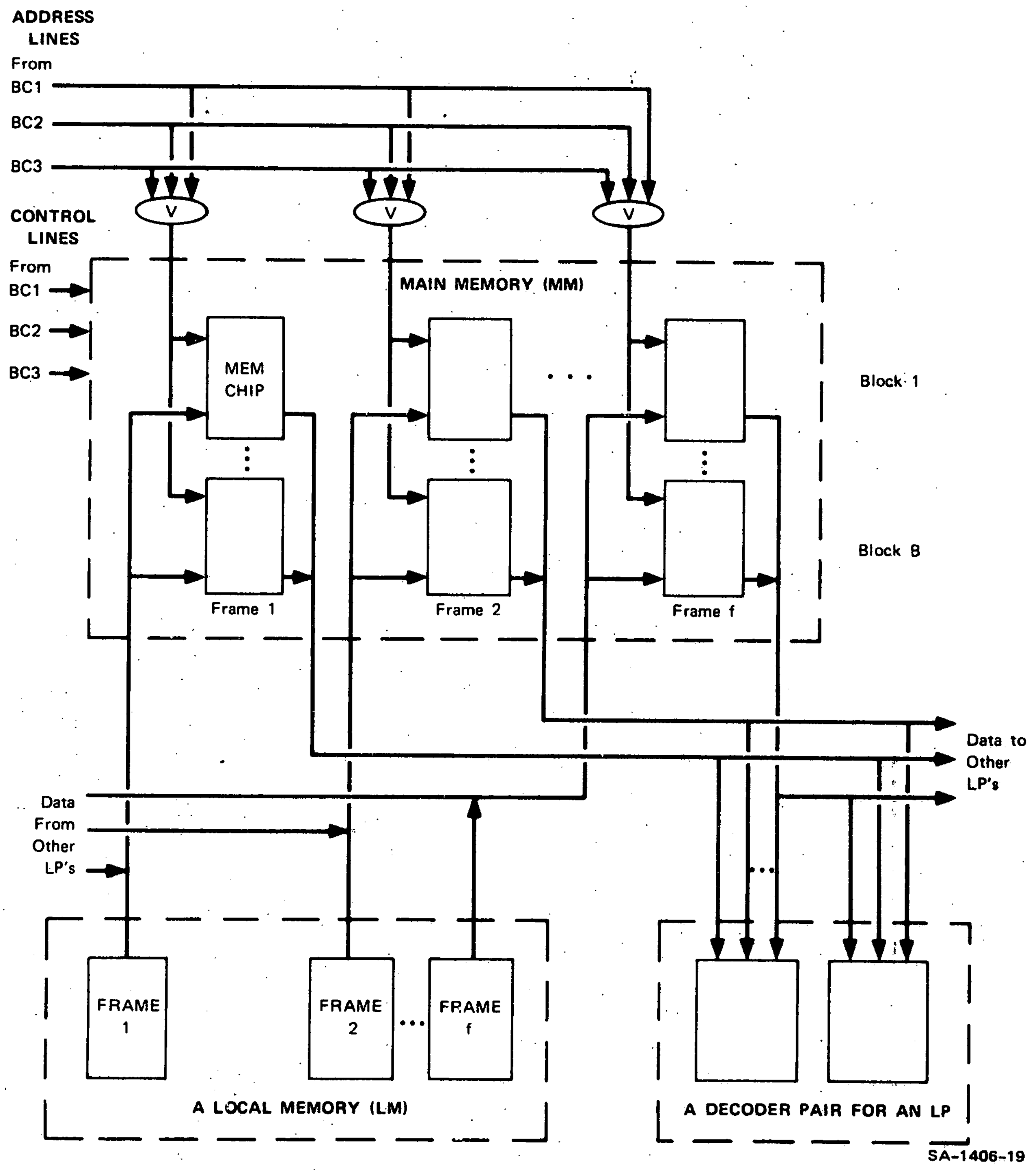


FIGURE VII-4 MAIN MEMORY ORGANIZATION

MMs and as such is not correctable. We believe that such double failures represent a small percentage of the possible double failure patterns, but they will be included in the reliability analysis.

In one approach for the transfer of data from an LP to a MM the MM data input lines are derived from the LM's. Thus the same error correcting code is conveniently used in both the LM's and MM. It seems possible to wire together the corresponding bit outputs of the LM's destined for the MM. Because of the error-correcting coding a chip failure anywhere that shorts out an entire frame for all LM's is not fatal. Simultaneous with transferring words from an LM to an MM the decoders associated with LP's could check the validity of the transferred words to pinpoint a possible LM failure.

The MM output words are destined for decoding in one of the decoders associated with an LP. Hence it seems reasonable again to wire together all of the corresponding decoder inputs and MM frame outputs. Once again, a single chip failure can only disable a frame in all decoders or in all blocks of MM and, hence, the failure is correctable by the code.

It is assumed that the MM is organized as a two-dimensional array of B blocks and f frames. At each block-frame intersection, there is an LSI chip containing 4096 bits of memory. As in the case of LM's, the apparently optimum organization of the chip is 2 bits wide by 2048 words. Thus for a MM of 32K words--reasonable for this application--16 blocks will be required.

As mentioned above it seems prudent to incorporate the same code within the MM as in the LM's, namely a single frame-error correcting, double frame error detecting code. In the case of the MM, our reliability analysis will show that single error correction is not quite adequate to achieve the stringent reliability requirements, for the reason that the MM has too many failure-prone chips within it. However, the portection against all double failures appears adequate. There are several approaches toward achieving double fault tolerance, among which are:

(a) the use of a double error correcting code for all words

(b) the use of a spare frame within each block

(c) the use of a single spare block.

We have ruled-out (a) as being too costly, both in terms of the extra redundant bits and the decoding costs. Approach (b) adds very little to the cost of the MM--approximately 5 percent--but it introduces the need for extra switching to provide a route around the faulty frame.

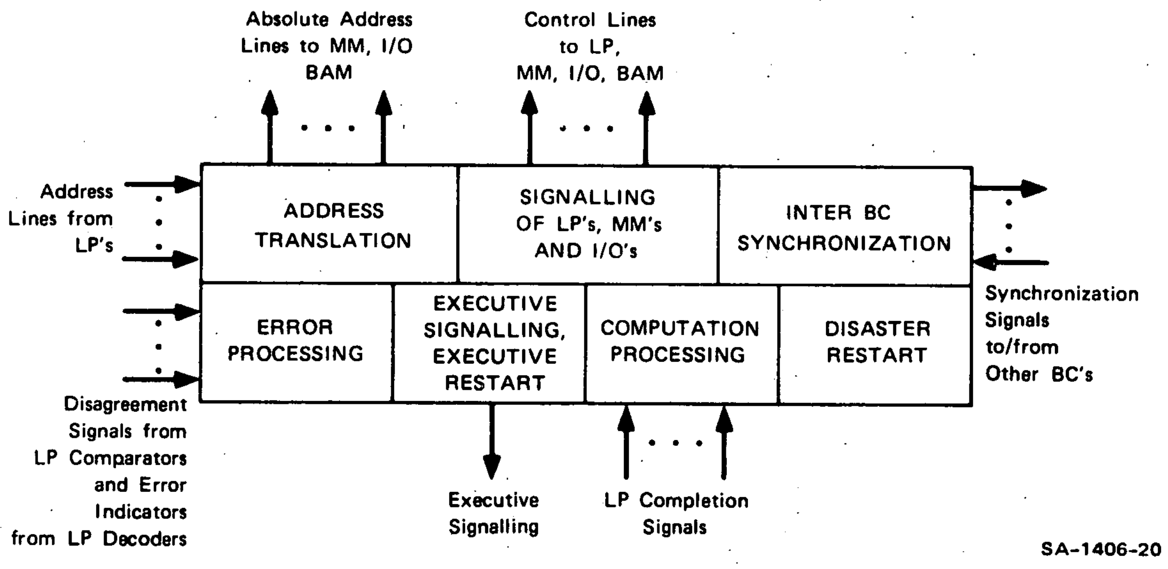
It appears that the spare block approach (c) is the most attractive. The extra cost is certainly moderate--6 percent for an original 16 block memory. The routing around a bad block is relatively simple, involving merely an address translation. In addition, a separate power supply could be provided for each block, thus providing a tolerance to single power supply failures.

There are two approaches toward identifying a faulty block of MM. As the MM words are decoded by the decoders in the LP's, single errors in MM will be revealed, but also corrected. Secondly, our intention is to periodically diagnose MM, mostly to ensure that the rarely used programs retain their integrity throughout the flight. As we show later a low rate of diagnosis--once every 10 seconds--is quite adequate.

We emphasize that the MM provides for single-frame fault masking with block switchover to yield a system that is essentially double fault tolerant. There are, however, certain double failures that can disable the entire MM. Since the MM data inputs and outputs corresponding to a frame are wired together, two failures that each short out frames would not be correctable. Although we do not have sufficient information to assess this possibility, we suspect it is unlikely.

3. Bus Checker Organization

The organization of each of the three bus checkers (BC) is displayed in Figure VII.5. For convenience of description we have shown the BC as composed of seven distinct sections. This particular decomposition may not hold in practice. A brief discussion of these sections follows.



Absolute Address Lines to MM, I/O BAM
Control Lines to LP, MM, I/O, BAM
Address Lines from LP's
ADDRESS TRANSLATION
SIGNALLING OF LP's, MM's AND I/O's
INTER BC SYNCHRONIZATION
ERROR PROCESSING
EXECUTIVE SIGNALLING, EXECUTIVE RESTART
COMPUTATION PROCESSING
DISASTER RESTART
Synchronization Signals to/from Other BC's
Disagreement Signals from LP Comparators and Error Indicators from LP Decoders
Executive Signalling
LP Completion Signals
SA-1406-20

FIGURE VII-5 BUS CHECKER ORGANIZATION

(a) Address Translation

As indicated previously, addresses emanating from the LP's are used relative to a base address that is computed by the executive. The address translation process, which is merely forming the sum of the base address and relative address to compute the true MM address, is carried out by the BC.

(b) Signaling of Units

The BC is assigned the role of signaling the LP, MM, I/O, and Back-up Memory (BAM) units. Such signaling generally signifies that a word of data is being sent to the unit in question.

(c) Inter BC Synchronization

It is recalled that three BC's are provided so as to afford a tolerance to single BC failures. The voting of the BC outputs is not accomplished within the BC's but by voters at the inputs to the units--LP's, MM, I/O's, and BAM's. However, there is a need to synchronize the operation of each of the BC's. As with the duplexed LP's the three BC's need not operate in locked-step, but must be coordinated upon delivering a signal to another unit. This coordination is attained by having each BC deliver a synchronization signal to the other BC's when it is ready to deliver a signal to a LP, MM, etc. The appearance of two synchronization signals, including a self-generated one, is sufficient to enable the transmission. Synchronization signals appearing at times other than transmission times are ignored, thus preventing a faulty BC from corrupting the other two BC's by emitting random synchronization signals.

(d) Error Processing

All disagreement signals from a duplexed processor pair or error signals from an LP decoder are directed towards the BC. For "normal" error processing, i.e. the response to single unit permanent failures or non-massive transient failures, the BC merely records the identity of the unit reporting the failure and signals the executive. It is likely that the various

LP's will be sampled on a round robin basis to prevent a permanently disagreeing LP from seizing permanent control of the BC. It would also be desirable for the BC to ignore a permanently disagreeing LP so as not to continuously bother the executive.

(e) Completion Processing

When an LP completes a task it alerts the BC. Once again, the BC merely records the fact and signals the executive. A round robin sampling of the LP outputs is again in order here.

(f) Executive Signaling and Restart

This section of the BC is responsible for signaling the executive upon the completion of an application task or an error condition. Accordingly, the BC knows the address of the executive LP. The signal is sent to all LP's but only the executive should respond. There is a possibility that a nonexecutive LP could fail (in both P's) such that it thought it was the executive, but this requires two correlated failures.

This section of the BC is also responsible for processing failures in the executive LP, and as such is the final arbiter. If the error signal emanates from the executive LP, the BC initiates the recovery by first trying to restart the executive LP, and then, if that attempt is unsucessful, finally switching in a spare LP to function as the executive. As such the BC must retain the following information: (1) identity of the executive, (2) absolute location in MM of a program that will restart the executive and load in last input data, and (3) identity of all spare LP's.

(g) Disaster Restart

We mentioned previously that it is desirable for the BUCS system to be able to respond to a massive transient that causes severe state changes. Under such a transient, for example, the registers in the BC that store the address of the executive LP could be corrupted. The Disaster section is intended to combat such a transient by restarting the system. In carrying out this role the BC performs global consistency checks on the rest of the system by executing a small program stored in an internal ROM. For example, if continuous errors are reported by many of the LP's or if the executive fails to respond to signals, then the BC restarts the system. We presume that it has access to a BAM that holds the program and possibly a last status dump. It is presumed that the identity of failed LP's and failed MM blocks is also lost during the transient, so that the BC must proceed slowly in a bootstrap approach toward activating the system. It selects an executive LP and, after determining that it is operative, lets the executive continue with the reconfiguration.

We believe that the design of the BC according to the above specifications is relatively straightforward. One possible difficulty concerns the number of BC terminals. The control signal lines are not a problem here because only a few are associated with each unit. However, the address input lines from the LP's could pose some difficulty since about 12 such lines will emanate from each P unit, corresponding to 24 from each LP. Clearly, they cannot be wired together since there is no error correcting coding of these lines. One approach is to have these address inputs transferred byte serially. Address requests to the BC should occur infrequently enough to warrant this serialization.

The BC as described above is a special purpose controller. The BC could be realized as a small microprogrammed processor with about 100 words of memory. It would also require a multiplexer to coordinate the communication with the LP's, MM's, etc.

D. Executive Operation

This section briefly summarizes the functioning of the executive in the BUCS system. First we will distinguish the main roles of the executive.

1. Application Task Scheduling

We recall that tasks are loaded into a LP when they are to be serviced. The executive supplies a MM address, and a word count to the BC in order to initiate the loading process. Since the computation load can be predicted for all flight contingencies during preflight planning, the task load and task sequencing for each LP can be prestored in the executive tables. In this case the executive merely cycles through a task schedule* associated with each LP. The scheduler could be driven by a clock whose frequency is about the same as the highest iteration rate associated with a task--250 Hertz in our environment. At each clock occurrence, the executive determines if any task that should have completed its execution has failed to do so. It could also schedule any tasks that require service at that time, including performing of an I/O operation for a task.

2. Background Executive Processing

There are several executive functions that require periodic but infrequent service. Among these functions are the diagnosing of "unflexed" hardware and the changing of the application task schedules due to a possible mission change. A relatively slow clock, say 0.1 Hertz could drive this portion of the executive. It is envisioned that these functions would be carried out in the executive LP.

With regard to diagnosis there is a need to periodically check portions of the system that are "unflexed," i.e. very rarely stimulated. Such portions include programs that are needed only for landing, executive programs that control reconfiguration, unused portions of MM, and the comparators associated with LP's. The particular problem of concern here is that two failures could possibly occur in a hardware section that is not exercised during normal computer usage. Hence when the particular hardware section is needed, say a memory block holding a reconfiguration program, it would not be operative. The solution is to diagnose such sections. With regard to rarely used programs the solution is simple--merely read through the storage holding the programs and

* This simple cyclic scheduling process also requires that no preemption of tasks is required prior to their completion. If it turns out that interruption is required to effect the multiprogramming, then some swapping of LM memory space will be called for, without introducing any fundamental difficulties.

check for the appearance of any errors that are handled by the decoder. The effect of an error is handled by the executive transferring the information in the offending MM block to another MM block.

For multiple failures in the comparators and other error indicators, the solution is also quite simple. Within the LP's let there be a few routines that introduce errors. Corresponding to such routines provide a few words in LM that purposely contain one or two errors. The reading out of these words should induce a properly functioning decoder to emit an error signal. The comparators associated with P units are checked by comparing the values stored in two registers that are purposely set to be different. These types of diagnoses by the executive should improve the situation, but clearly do not solve the problem completely.

For example, with regard to a program intended to carry out a reconfiguration, is it sufficient to merely check the integrity of the stored version of the program? The possibility, albeit unlikely, exists that a particular portion of a register, used only for this program, could fail in each of the P units. Under this event the reconfiguration program would not function properly, although no comparator would emit an error signal. With the basic architecture of BUCS (or a three-voting SIFT or Hopkins' scheme), it is unlikely that the system can be made totally tolerant to all double faults. The design problem we have addressed is the reduction of the fraction of all double faults--say to one percent--that can result in system failure.

3. Single Error Processing

If an error is detected as a disagreement among P units, as a single error indication in an MM block, or as a double error indication in an LM, the executive is signaled. Since the failure might only be transient the executive should reinitiate the computation with the initial data and allow the computation to be retried. If a failure occurs several times in succession, another LP or another MM block should be initiated with the same initial data.

A question arises concerning the response to a second error that occurs in the executive LP during the retry reconfiguration process. The probability of the second error arising from a second permanent failure is indeed remote, since we do not expect more than two faults in a mission. However, a second transient fault is possible and should be handled by the system. The process is as follows: the pertinent error data--identy of possibly failed LP, and the number of retries effected so far--is stored in the executive tables as part of the initial executive program data. (The second failure occurring during this brief time it takes to store the initial data would possibly be untolerated.) Hence, if the executive fails, the bus checker can activate another executive to effect a retry and possible reconfiguration of the first executive. A failure in the second executive would result in a similar procedure, involving the activation of a third executive, etc. The process operates as a pushdown stack of executive status, although the system's resources are likely to be depleted before the stack gets very deep.

4. Executive Load

The executive routines as described above should require no more than 4K words of storage, including data, and hence the entire executive should fit into a LM. The task scheduling routines, including the process of loading programs into LM's, should consume no more than 0.1 MIPS, and as such represents about a 20 percent overhead, compared with the application tasks. The background executive processing should consume about 100 msec every sec, or equivalently 0.01 MIPS--a negligible overhead. Note that the MM diagnosis portion of the executive will by necessity be multiprogrammed with other executive functions. The decomposition, involving the diagnosis of a block of MM in a given computational interval, is easily achieved.

Our analysis has shown that an executive that is centralized and is normally operative in a particular LP (except when that LP fails) is adequate for controlling the presently conceived system. If it becomes desirable in the future to incorporate the BUCS concept within a much larger system--say containing 25 LP's--some modifications are called for. In this case, the task scheduling and background task part of the executive could be distributed among the processors, as in SIFT. If the scheduling routine can be made small enough, a copy could be permanently resident in each of the LM's. In addition, if it becomes feasible (on the basis of application program size) to use a 2K LM instead of 4K then the scheduling portion of the executive will be permanently resident in an LP. The remainder of the executive will be treated as an application task to be loaded into an LP, when needed, under control of the scheduler.

E. Reliability and Performance

This section presents a simple analysis of the reliability of the BUCS system as described in the previous sections. Although the analysis is simple, we believe that it is conservative, i.e. the actual reliability is expected to be better than described here. Our intention is to show that with rather low redundancy--less than 40 percent--the reliability goal of 10^{-8} for a 5 hour mission is attained.

Our assumptions on the failure mechanisms are as follows:

(1) Failures from chip-to-chip are assumed to be independent, and failures do not propagate from chip to chip.

(2) The failure rate is 10^{-6} failures/hour/chip.

(3) The failure rate is independent of time--a good assumption for short missions.

(4) The probability of failure of the total system is derived as the sum of the failure probabilities of the individual subsystems--a good assumption for the low probabilities that we are dealing with.

(5) Only permanent failures are included in the model.

The BUCS architecture exhibits the following failure states for each of the major subsystems.

(1) Bus Checker (BC)--The failure of two of three BC's leads to a system failure. Each BC is assumed to contain 3 chips.

(2) Processor (P) portion of LP's--we assume one spare duplex P unit, and 6 duplex P units required for a non-redundant system (5 for the application tasks and 1 for the executive). Hence the failure of two P-pairs leads to system failure. Each P pair consists of 6 chips.

(3) Local memory (LM) portion of LP's--we assume one spare LM and 6 required for a non-redundant system. Hence the failure of two LM's leads to system failure. Each LM is protected by a single frame-error correcting code, i.e. two frame failures within an LM lead to an LM failure. Each LM contains no more than 22 chips (arranged as 2 blocks of 11 frames each).

(4) Main memory (MM)--Each block of MM is protected by a single frame-error correcting code, with a switchover to a spare block whenever a failure occurs. Hence, it takes three block failures to cause a system failure. The MM contains 176 chips arranged as 16 blocks of 11 frames each. (This allows for sufficient memory to hold all application programs, executive programs, plus a single spare block.) We pessimistically assume that three chip failures anywhere lead to system failure.

(5) Second failure within reconfiguration interval--a second failure occurring while the system is responding to the first failure will lead to system failure. We assume that the reconfiguration interval is 10 seconds--clearly a pessimistic guess for the response to a failure, but reasonable as an interval between diagnoses of the MM. Given the first failure in MM, the second failure would have to occur in the same block of MM as the first failure to bring the system down. On the other hand, given the first failure in a LP, the second failure would have to occur in the spare LP taking over the tasks of the first LP to bring the system down. The net effect is to reduce the fraction of equipment that is vulnerable to about 1/10 of the total equipment. The total system contains about 400 chips.

(6) Fatal double failures--because of the "unflexed" components (Section D-2) certain double failure patterns lead to system failure regardless of when they occur during the flight. A rough analysis of the double fault patterns indicates that less than 1 percent of the double fault patterns will lead to system failure.

We are now in a position to evaluate the contribution of each of the above failure mechanisms to the total system failure probability.

For failure mechanism i, i = 1,..., 6, assuming the parameters given in the opening of this section, the failure probability P_{f_i} is:

$$P_{f_1} = \binom{3}{2} \cdot (3\cdot 10^{-6})^2 \cdot 5 = 1.35\cdot 10^{-10}$$

$$P_{f_2} = \binom{7}{2} \cdot (6\cdot 10^{-6})^2 \cdot 5 = 3.78\cdot 10^{-9}$$

$$P_{f_3} = \binom{7}{2} \left[\binom{22}{2} \cdot 10^{-12}\right]^2 \cdot 5 = \text{negligible}$$

$$P_{f_4} = \binom{176}{3} \cdot 10^{-18} \cdot 5 = \text{negligible}$$

$$P_{f_5} = 400\cdot 10^{-6} \cdot \left(\frac{10}{3600} \cdot 399\cdot 10^{-6}\right) \cdot 5 \cdot (1/10) = .22 \cdot 10^{-9}$$

$$P_{f_6} = \binom{400}{2} \cdot \left(\frac{1}{100}\right) \cdot 10^{-12} \cdot 5 = 4\cdot 10^{-9}$$

Forming the sum $\sum_{i=1}^{6} P_{f_i} = \approx 8 \times 10^{-9}$, which is within the goal of 10^{-8} for the five hour mission.

Clearly, the two most significant factors are P_{f_2} and P_{f_6}. The former can be reduced to a negligible quantity by merely providing one additional spare LP (for two spares) total, at an additional redundancy cost of 15 percent.

On the other hand, P_{f_6} can be reduced only by carrying out more diagnoses to reduce the quantity of unflexed hardware or by good engineering design (e.g., locate sensitive gates on opposite ends of a chip).

We emphasize that these figures are tentative, although probably pessimistic. The redundancy is quite low, being about 30 to 50 percent in total extra chips required for the fault tolerance (dependent upon LM size) plus about 10 percent extra chips required to implement the executive.

A few comments are in order with regard to BUCS performance measurements other than failure probability. One important parameter of the system is the "down-time" following a transient or permanent failure. Several of the crucial computations (flutter control, load control, collision avoidance) cannot be unserviced for more than a few milliseconds. We will now demonstrate that no such discontinuity need occur in BUCS.

Typically, when a failure indication is broadcast to the BC, the BC will initiate a retry. The main delay in effecting this retry is the time in reloading the program into the LM from the MM. However, the critical programs never require more than 1K words of storage, so it seems reasonable to accomplish this reloading in a few milliseconds. If the executive LP fails, the entire 4K word executive must be reloaded, an operation that could take 10 msec. There might be some tasks that cannot wait for this transfer to be completed. The solution is to reload first that portion of the executive concerned with scheduling--a portion that should not require more than a few hundred words. Then the reloading of the rest of the executive can be multiprocessed with the critical tasks, encumbering no further delay. However, if a second failure occurs in an application task LP, the system might come to a temporary halt. However, as we demonstrated above, the probability of a second failure is remote.

F. Embellishments and Discussions

The BUCS system is an attractive architecture for fault tolerance because of its relatively low redundancy. In our application the redundancy is low because the system is memory dominated permitting the use of coding protection for much of the hardware, and because the application program mix permits a fine decomposition of processing power into relatively small processors.

The three main deficiencies of the BUCS approach as described above are the following:

- High speed operation will preclude the byte transfer of information between LP's and the BC. Thus an extremely large number of pins will be required for the BC.
- The expandability of BUCS is limited by the capacity of the BC. This speed limitation is common to all single bus multiprocessors.
- The redundancy, although low as compared with most fault tolerant architectures is still needlessly dominated by the LM's associated with each LP. These LM's were inserted to reduce the bus traffic, and represent 80 percent of the LP hardware.

One way to circumvent these difficulties is to incorporate a multibus structure--a concept suggested by Jim Miller of Intermetrics. The main memory system as described would be replaced by a system composed of numerous modules. With such an embellishment the LM's can be discarded in favor of simultaneous communication among several LP-MM modules. A single bus checker unit (triplicated), almost identical to the one described in this chapter, would establish the communication links, respond to error signals and coordinate all calls on the executive. It is envisioned that a communication link between an LP and a MM module would remain established for the duration of a computational task. If the processing required temporary access to another MM module, say, for a particular data set, the BC would be signaled. The signaling process would correspond closely to a page fault in a contemporary paged machine.

The type of multiprocessing as conceived here is grealy enhanced by predictable computational demands. Thus the programs do not require any dynamic linking except subsequent to a permanent fault. However, even in this case the relinking involves only a simple address translation.

VIII SUMMARY AND CONCLUSIONS

A. Summary

In this study we have attempted to identify the architectural features of a digital computer that are well-matched to the reliability and computational requirements of an advanced commercial aircraft. The intention is to have a central computer complex carry out the computations presently distributed among small digital systems, analog computers and mechanical computers. As part of the study we investigated three candidate architectures that could meet the specific requirements. One of the candidates is a multiprocessor due to Hopkins; the other two candidates were conceived during this present study--SIFT and BUCS. Before embarking on a detailed design effort on these systems we surveyed the numerous fault-tolerant architectural concepts that have been developed over the past decade. One of these concepts (JPL-STAR) has been pursued to the breadboard stage, but the rest have remained "paper" designs.

We do not feel that any of the surveyed systems is suitable for NASA-Langley, mostly because the designs have not been carried to the necessary detail such that the fault-tolerant procedures could be evaluated. The one notable exception is the STAR, in which all fault-tolerant procedures have been specified, and in some cases implemented. It is clear that the STAR can tolerate any single fault that disables one of the independent units that comprise the system. This fault tolerance is achieved with a relatively low cost and less than 100 percent redundancy, assuming sufficient spares for single fault tolerance. Transient faults that have the same effect can also be tolerated. However, the following deficiencies of STAR have induced us to pursue the three aforementioned candidates.

- STAR is not readily expandable/contractible to accommodate large variations (say, an order of magnitude) in computational load.
- STAR was not designed with an LSI environment in mind. It contains more module types than need be, and the arithmetic codes may not be useful for the massive failures expected of an LSI implemented arithmetic unit.

PRECEDING PAGE BLANK NOT FILMED

- The design does not take advantage of the independence of the computations. That is, the aircraft environment almost implies that the fault tolerance and graceful degradation should be carried out by a multiprocessor.

The three candidate designs, in addition to STAR, can be contrasted in their mechanisms for carrying out the various fault tolerant procedures. Table VIII.1 indicates the comparisons for these four systems. In the table we have included the following functions:

<u>Fault detection</u>	The detection of error conditions in memories, buses and processing units.
<u>Fault masking</u>	The removal of errors, within a particular unit, for example by error-correcting codes within a memory so that they are not aparent to other units.
<u>Fault correction</u>	The procedures that are used to initiate a recovery procedure after an error has occurred.
<u>Roll-back</u>	The ability to re-run part-of a program either for error detection or correction purposes (sometimes called "check point/restart").
<u>Fault location</u>	The ability to determine the unit that is faulty, when an error occurs.
<u>Reconfiguration</u>	The ability to change the units that are used for a calculation.
<u>Periodic diagnosis</u>	The flexing at various system blocks or function, in background, to determine if any permanent failures are present.

The following abbreviations are used

H ...	hardware
M ...	microcode
S ...	system routine
E ...	executive
A ...	application program.

Multiple entries in Table VIII. 1 indicate that design options still exist.

	JPL STAR	HOPKINS	BUS-CHECKER	SIFT
Fault detection	H	H	H, S(3)	H,M,S
Fault masking	H(1)	H(4)	H	(2)
Fault correction	H(1)	H	S	M,S
Roll-back	H	H	S	S,E
Fault location	H	H	S	E
Reconfiguration	H	E	E	E
Periodic Diagnosis	E	E(5)	E(6)	S,E(7)

Table VIII.1 Comparison of Fault-Tolerant Procedures

(1) Within the TARP (Test and Repair Processor).

(2) Within SIFT the memory could use hardware fault masking; otherwise no fault masking is required.

(3) Assuming that the bus checker units are implementing system functions.

(4) By voting on the buses and possibly by coding within the memory.

(5) Of the hardware voters.

(6) Of the memory system.

(7) Of the software voters and reconfiguration programs.

Table VIII. 2 compares the three candidates according to the check-list categories of Chapter V. Table VIII.3 compares the candidates in terms of performance.

	SIFT	HS	BUCS
COMPUTATIONAL ENVIRONMENT			
• MULTIPROGRAMMING	Yes	No	"Yes"
TECHNOLOGY			
• NUMBER OF CHIP TYPES	Low	Medium	Medium
• SPECIAL FAULT ISOLATION HARDWARE NEEDED	Low	Moderate	High
REDUNDANCY	High	High	Low
RELIABILITY MODELING			
• CREDIBILITY OF MODEL	Very High	High	Moderate
EXPANDABILITY	Very High	Low	Moderate
PROTOTYPE DEVELOPMENT EFFORT	Low	High	Moderate

Table VIII. 2 Candidate Comparisons by Checklist Categories

	SIFT	HS	BUCS
TIME TO DETECT MOST FAILURES	100 μsec - 10 msec	2 μsec	2 μsec - 10 msec
ERROR CORRECTION TIME	0	< 1 msec	5 msec
RECOVERY TIME	> 10 msec	2 msec	10 msec
BUS DATA RATE	200 kb	20 mb	2 mb
REDUNDANCY	≥ 3	≥ 3	≥ 1.5

Table VIII.3 Performance Comparison of Candidates

B. Conclusions

We conclude that a fault tolerant computer can be built for commercial aircraft at this time. A number of factors have led to this conclusion:

- Component costs continue to drop to the point where the hardware costs of a triplicated (quadruplicated, etc.) computer system are an insignificant fraction of the total aircraft cost (less than 1 percent).
- The aircraft computations do not exert a large computational load, relative to the capacity of contemporary computers.
- An understanding has been acquired of the fault tolerant procedures to the point where almost any desired reliability can be convincingly demonstrated. Included herein is a tolerance to permanent faults that disable a chip, and to transient faults that do not completely destroy critical memory functions.

Of the three candidate architectures investigated we recommend that SIFT be pursued for future development. This recommendation is made recognizing that all of the architectures exhibit sufficient reliability and fault tolerance, and SIFT incurs the largest cost in terms of component ccunt. The primary reasons for selecting SIFT over the others are:

- It should incur the lowest development costs and perhaps development costs are more crucial than ultimate component costs.
- It accommodates varying reliability requirements among computations.
- The redundancy can be reduced by utilizing a central memory that is protected by coding techniques.
- It is easy to modify since the fault tolerance is implemented in software or microcode.
- The concept can be adequately demonstrated with off-the-shelf processors and memories; only the bus must be specially designed.

- Although the concept requires software for error detection, etc., it does not require special software for enforcing protection disciplines. Such protection is inherent in the design.

ALLOCATION AND SCHEDULING SYSTEM ROUTINE

These two functions are described in terms of the data structures used, and the way in which they are manipulated.

The major data structures are shown in Figure A-1. The six vectors T, DT, MT, CT, CI, MR contain, for each task, the data required to calculate the task matrix. The information contained is:

T:	The iteration period for the task
DT:	The permissible variation of T, i.e., a task will normally be carried out between T - DT and T + DT after the previous iteration
MT:	The 'Miss Time' which is the maximum time that can be tolerated without the task being carried out.
CT:	The computer time required to carry out one iteration of the task.
CI:	The current iteration being computed for the task
MR:	The memory requirement for the task.

The flight phase matrix contains the priority number for each task for different flight phases. In our application, approximately ten flight phases are assumed. These will include normal phases such as takeoff, climb, cruise, etc., and also potential abnormal phases (e.g., when an engine is inoperative). A priority of 0 indicates that a particular task is not carried out during this phase. It is assumed that transitions from one phase to another is initiated by aircrew action.

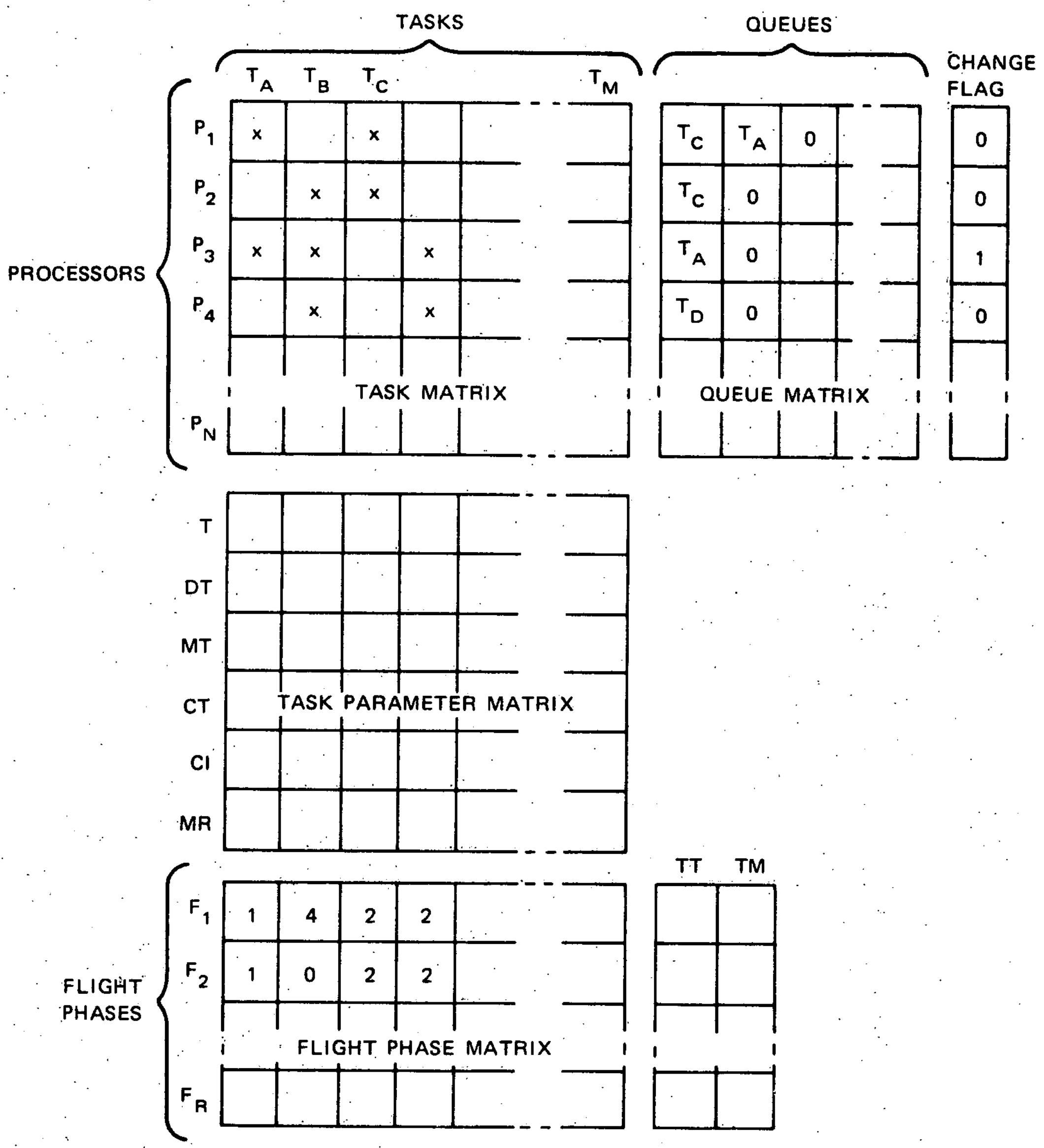


FIGURE A-1 ALLOCATION AND SCHEDULING DATA STRUCTURES

The vectors TT and TM hold, respectively, the total computer time required and the total memory required for each phase of the flight.

The queue matrix represents the output of the scheduling task within the executive. For each processor a queue is maintained as to which tasks are to be computed next. It is assumed that each processor will examine the queue and store a local copy from time to time, thereby reducing the number of accesses required to be made into the queue matrix. If the scheduler has to change a queue, a bit is set in the change flag vector indicating that the processors' local copy of the queue must be discarded and a change made. This will normally occur when the flight phase changes, and also in the event of reconfiguration following a fault condition. It is expected that the vectors within the queue matrix will be maintained as circular buffers, thereby preventing any requirement to change the numbers except to place new items on the queue (change a 0 to the task number) or to remove tasks that have been completed (change a task number to a 0).

In addition to the data structures described above, the allocation task will store a note as to which flight phase is current, and also it will keep a note of the resources currently available, normally those processors, busses and memory modules that are operative.

The allocation function consists of deciding which tasks are computed in each processor. Two possible algorithms are described -- the first algorithm attempts to find a solution, the second algorithm adjusts the result of the first to remove cases where necessary constraints have been disobeyed.

The allocation problem can be stated as follows:

Let P be the number of processing modules having the same computing power and associated memory capacity. Let N be the number of tasks. for the j^{th} task $(j = 1, \dots N)$,

c_j = fraction of computing power required

m_j = fraction of memory capacity required

r_j = replication required for the task

Determine an allocation matrix A_{ij}: $i = 1,\ldots,P$; $j = 1,\ldots,N$

where

$A_{ij} = 0$ or 1 (No fractional allocation) (1)

$\sum_i A_{ij} = r_j$ (Correct replication) (2)

$\sum_j A_{ij} m_j \leq 1$ (Memory constraint) (3)

$\sum_j A_{ij} c_j \leq 1$ (Computing constraint) (4)

Note that we are only concerned with finding an allocation matrix and not with any defined optimum. It may be desirable to define an optimum solution as one which leaves all spare capacity (both computing and memory) uniformly distributed among processors; however, a contrary case can be made to leave spare capacity concentrated in one processor in order to aid the reconfiguration and re-allocation in the event of a processor failure. The algorithms below were not intended to find such an optimum, however defined.

Replication of tasks can be both active and passive. By active, we mean that the tasks will run in those processors. By passive, we mean that sufficient capacity is available to allow them to run in the event of failure of one of the active allocated processors. Passive allocation would only be used for those tasks where the task must be carried out in a time interval that is too small to allow loading it to a new processor upon reconfiguration.

Algorithm 1

The general scheme of the algorithm is to allocate tasks one by one starting with that task which has the greatest demand upon a resource (i.e. maximum c_j or m_j) and progressing down to the last task which has the easiest constraint. The allocation is made to those processors that have the maximum available of the most demanded resource. Ties are decided arbitrarily, in our example, by allocating to the lowest numbered processor.

In the event that $m_j = c_j$, the tie is broken by deciding on the basis of the resource which is most critical (memory in our example). The algorithm is shown in flow-chart form in Figure A-2.

Algorithm 2

Algorithm 2 is applied after algorithm 1 and is applied repetitively (including possibly zero times) until constraints 3 and 4 are satisfied. The algorithm consists of finding that constraint 3 or 4 which is disobeyed to the greatest extent, and moving a task to that processor which has the maximum available of that resource. The movement is subject to the constraint that a single task may not exist twice in the same processor (i.e., $A_{IJ} = 0$ or 1). The task that is moved is that which most tends to equalize the constraints. A more sophisticated version of the algorithm is to re-allocate (using algorithm 1) all the tasks previously allocated to the two processors.

Example

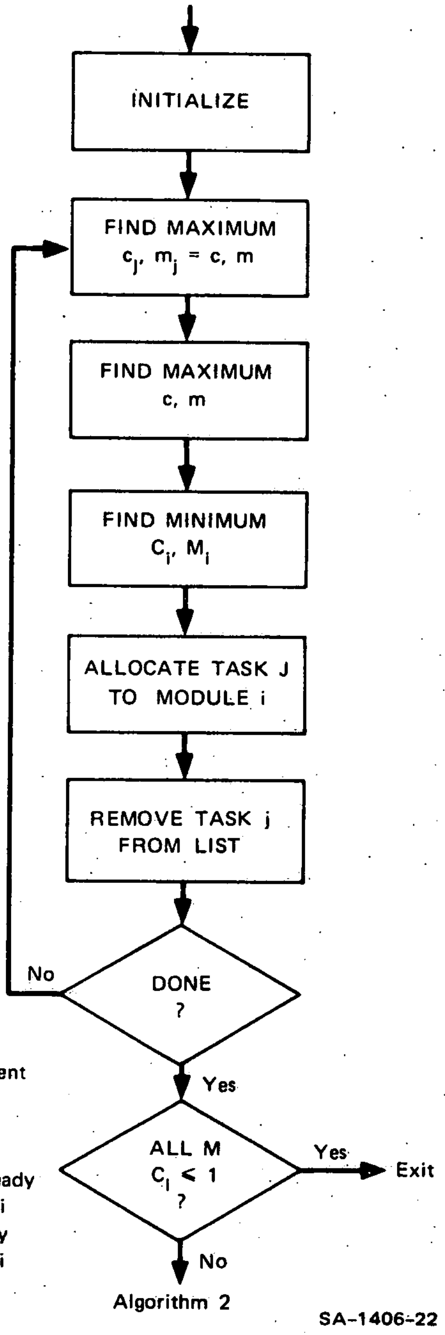
Table A-1 shows the values of r_j, m_j, and c_j for a real example.

Table A-2 shows the application of algorithms 1 for this example. As can be seen, algorithm 2 does need to be applied as all constraints are satisfied. To illustrate the use of algorithm 2 consider the artificial example set of 3 tasks, with P = 2.

Task	m_j	c_j	r_j
1	.8	.1	1
2	.1	.8	1
3	.7	.7	1

The result of algorithm 1 will yield an allocation matrix

$$\begin{array}{cc} & \text{Tasks} \\ \text{Processors} & \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$



c_j = Computing Requirement for Task j

m_j = Memory Requirement for Task j

C_i = Computing Load Already Allocated to Module i

M_i = Memory Load Already Allocated to Module i

FIGURE A-2 ALLOCATION ALGORITHM 1

Table A-1

Example Data For Allocation Function

Task No.	m_j	c_j	r_j
1	.02	.04	3
2	.02	.33	4
3	.10	.12	4
4	.03	.02	3
5	.23	.14	4
6	.03	.01	3
7	.43	.25	3
8	.12	.09	3
9	.20	.06	3
10	.10	.01	3
11	.10	.01	3
12	.12	.12	4
13	.15	.01	3
14	.10	.20	4

Table A-2

Application of Algorithm 1 for Data in Table 1 and P = 6

PROCESSORS

	1		2		3		4		5		6	
Task just allocated	Σm*	Σc	Σm	Σc	Σm	Σc	Σm	Σc	Σm	Σc	Σm	Σc
7	43	25	43	25	43	25						
2	45	58					2	33	2	33	2	33
5			66	39			25	47	25	47	25	47
9							45	53	45	53	45	53
14			76	59	53	45	55	73	55	73		
13	60	59			68	46					60	54
3	70	71	86	71	78	58					70	66
8	82	80					67	82	67	82		
12					90	70	79	94	79	94	82	78
10	92	81					89	95	89	95		
11			96	72			99	96		92	79	
1			98	76	92	74					94	83
4	95	83			95	76			92	97		
6	98	84							95	98	97	84
Totals	98	84	98	76	95	76	99	96	95	98	97	84

*Capacity is expressed as a percentage, unchanged values not entered.

with

$$\Sigma A_{1j}m_j = 1.5 \quad , \quad \Sigma A_{2j}m_j = .1$$

$$\Sigma A_{1j}c_j = .8 \qquad \Sigma A_{2j}c_j = .8$$

Algorithm 2 calls for movement of a task from processor 1 to processor 2, based upon balancing the systems as far as possible.

The changed matrix would become

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This allocation matrix satisfies the constraints.

Appendix B

COMPARISON OF THE SIFT SYSTEM AND A MULTI-CHANNEL SYSTEM

INTRODUCTION

In this appendix we compare the SIFT architecture with the multi-channel concept. The multi-channel concept uses n independent computers, each capable of performing all tasks, and a highly reliable voter at the output to compare all data before it is transmitted to an effector.

Variation on both the SIFT and the multi-channel concept are discussed.

In all cases we assume a chip failure probability of 10^{-6} per hour. We use the notation that P[event] = probability of the event occurring per hour.

REQUIREMENTS

We distinguish between tasks of most criticality (MC) where error* probabilities should be below 10^{-8} per hour and those less critical (LC) tasks where errors should be beloe 10^{-4} per hour (approximately). We also distinguish those tasks required for automatic 'blind' landing and other tasks. The landing phase is the most demanding in terms of computing load. Based on the entries of Tables 2 and 3 of Volume II, we summarize in Table B-1 a representative set of requirements, where M = memory requirements in K words and P = processor requirements in MIPs.

Table B-1

Computation and Memory Requirements

	Landing	Other

M.C.	P = .29
	M = 8.7
		P = .09
		M = 2.2

L.C.	P = .9
	M = 6.8
		P = .05
		M = 5.5

*In this analysis, we do not distinguish between erroneous outputs to actuators and null outputs. A more comprehensive analysis would need to make this distinction.

Note that no allowance has yet been made for an executive to carry out such functions as multiprogramming, voting, etc.

We assume that words contain, on the average, 24 information bits. We further assume that a memory chip contains 4K bits, and that it requires 30 chips/MIP to realize the CPU.

Case 1 Multi-channel

We assume 10 percent extra memory and processor requirement to handle the multiprogramming and other executive requirements (interrupt handling, etc.). The multi-channel concept requires enough memory in each channel to hold all tasks (= 23.2K + 10 percent $\doteq$ 26K), and the CPU must handle the heaviest task load (= .38 + 10 percent $\doteq$ .42 MIPs). Therefore for each channel we have

$$\left.\begin{array}{l}\text{26K words} = \text{156 chips} \\ \text{0.42 MIPs} \doteq \text{13 chips}\end{array}\right\} \doteq \text{170 chips} \quad .$$

Assume that the chips in the voter (sufficiently replicated for reliability) are negligible and consider the probability of error for 3, 4 and 5 channel configurations. The results are displayed in Table B.2.

Table B-2

Reliability Estimates for Multi-channel System

<u>3 Channel</u>		
Total chips	= 540	
P[1 fault]	= $.51 \times 10^{-3}$, ...	voting masks error, discard faulty channel
P[2 faults]	= $.17 \times 10^{16}$, ...	system failure
<u>4 Channel</u>		
Total chips	= 680	
P[1 fault]	= $.68 \times 10^{-3}$, ...	voter removes faulty channel
P[2 faults]	= $.34 \times 10^{-6}$, ...	voter masks second fault, discard faulty channel
P[3 faults]	= 1.2×10^{-10},...	system failure
5 Channel		
Total chips	= 850	
P[1 fault]	= $.85 \times 10^{-3}$, ...	voter removes faulty channel
P[2 faults]	= $.58 \times 10^{-6}$, ...	voter removes faulty channel
P[3 faults]	= $.3 \times 10^{-9}$, ...	voter masks fault, discard faulty channel
P[4 faults]	= 1×10^{-13},...	system failure

<u>Case 2 SIFT With Fault Tolerance Achieved by Uniform Replication</u>

For this case, the strategy is to triplicate all tasks, and when faults occur to reduce the LC tasks to duplicate, then single processors, finally removing them entirely in the event that resources are drastically reduced. We assume 20 percent overhead for executive plus voting routines.

The memory and processor requirements are as in Table B-3. The reliability results are displayed in Tables B-4, B-5 and B-6, for a SIFT system decomposed into 4, 6 and 10 modules, respectively.

Table B-3

SIFT Processor and Memory Requirements

	Landing	Other

M.C.	P = .35
	M = 10.4
		P = .11
		M = 2.6

L.C.	P = .11
	M = 8.2
		P = .06
		M = 6.6

Total memory requirement = 27.8 ≑ 28K
Maximum CPU requirment = .46 MIPs

Table B-4

Reliability Estimates for a 4-Module SIFT

Each memory = (28 X 3)/4 = 21K = 126 chips } Each CPU = (0.46 X 3)/4 = 0.35 ≑ 10 chips } 136 chips Total chips = 544 We denote MC . during landing as MC/L etc.
P[1 fault] = .54 X 10^{-3}, M = 63K, P = 1.05 During Landing: Remove LC, MC survive During Other: MC survive, all LC to SIMPLEX, future removal LC/L
P[2 faults] = .22 X 10^{-6}, M = 42K, P = 0.70 During Landing: MC/L only survive in DUPLEX During Other: All MC to DUPLEX in memory, all LC to SIMPLEX, future removal of LC/L
P[3 faults] = .6 X 10^{-10}, System Failure

Table B-5

Reliability Estimates for a 6-Module SIFT

Each memory = (28 X 3)/6 = 14K = 84 chips Each CPU = (0.46 X 3)/6 = 0.23 $\doteq$ 7 chips } 91 chips Total chips = 546
P[1 fault] = .55 X 10^{-3}, M = 70K, P = 1.15 During Landing(1): Fault masked, LC to SIMPLEX During Other: Fault masked, future LC/L to SIMPLEX
P[2 faults] = .25 X 10^{-6}, M = 56K, P = 0.92 During Landing: Fault masked, MC, LC to DUPLEX During Other: LC failure, MC fault masked
P[3 faults] = .91 X 10^{-10}, M = 42K, P = 0.69 During Landing: System failure During Other: LC failure, MC fault masked
P[4 faults] = .16 X 10^{-14} All: System failure

(1) Assumes slight increase of CPU to .24 MIPs

Table B-6
Reliability Estimates for a 10-Module SIFT

Each memory = (28 X 3)/10 = 8.4K = 51 chips Each CPU = (0.46 X 3)/10 = 0.14 ÷ chips Total chips = 550 chips	} 55 chips
$P[1 \text{ fault}] = .55 \times 10^{-3}$, M = 75.6, P = 1.26 During Landing: Fault masked, LC to DUPLEX During Other: Fault masked, LC/O to DUPLEX, Future LC/L to DUPLEX	
$P[2 \text{ faults}] = .27 \times 10^{-6}$, M = 67.2, P = 1.12 During Landing: MC fault masked, LC failed During Other: Fault masked, Future LC/L fail	
$P[3 \text{ faults}] = .19 \times 10^{-9}$, M = 48.8, P = 0.98 During Landing: MC fault masked, MC/L to DUPLEX During Other: Fault masked, Future LC/L fail	
$P[4 \text{ faults}] = .73 \times 10^{-3}$, M = 40.4, P = .84 During Landing: Possibility of system failure During Other: Possibility of LC failure, future MC/L in DUPLEX	

Case 3 SIFT with Coding in Memory

The majority of chips for SIFT in Case 2 are used in the memory. We can add protection by using an error detecting/correcting code. The analysis displayed in Tables B-7 and B-8 is for a single error correcting, double error detecting code with an assumption of 25 percent increase in memory cost. A module failure requires failure of one chip in the CPU or two chips in the memory. The reconfiguration strategy is as shown in Figure B-1, for most critical tasks. Low criticality tasks are run in SIMPLEX mode.

Reliability Estimates for a 3-Module SIFT With Coding in Memory

Memory per module = (13 X 2 + 15)/3 + 25 per cent ÷ 17K = 102 chips } 110

CPU per module = (.35 X 2 + .11)/3 = .27 = 8 chips

Total chips = 330 chips

P[CPU fault] = $.8 \times 10^{-5}$/per module

P[single memory fault] = 10^{-4}

P[double memory fault] = 10^{-8}

P[LC task failure] = $.8 \times 10^{-5}$

P[reconfiguration for MC] = $.3 \times 10^{-3}$

P[second module failure (undiagnosable)] = $3 \times 10^{-4} \times .8 \times 10^{-5} = 2.4 \times 10^{-9}$, system failure

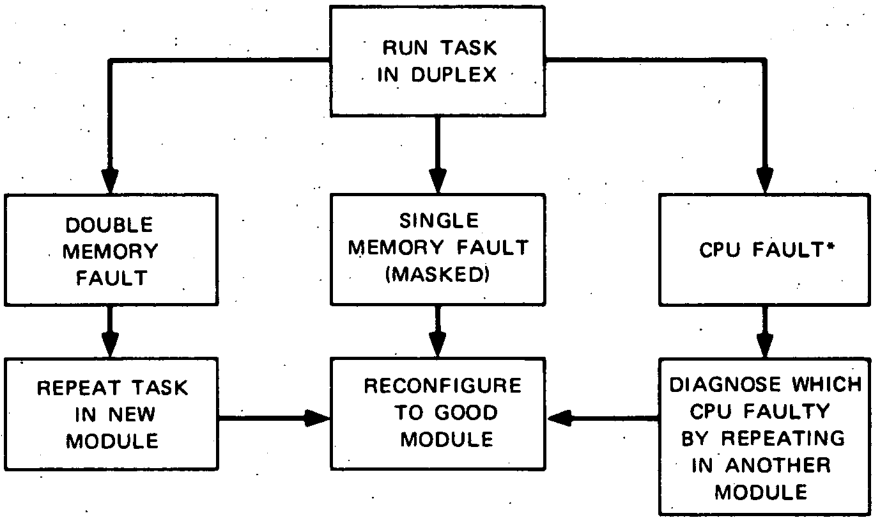

RUN TASK IN DUPLEX
DOUBLE MEMORY FAULT
SINGLE MEMORY FAULT (MASKED)
CPU FAULT*
REPEAT TASK IN NEW MODULE
RECONFIGURE TO GOOD MODULE
DIAGNOSE WHICH CPU FAULTY BY REPEATING IN ANOTHER MODULE
SA-1406-25

FIGURE B-1 SIFT WITH CODING, RECONFIGURATION STRATEGY FOR MOST CRITICAL TASKS

*CPU fault is any disagreement between the cooperating pair of modules, without indication of memory fault. It therefore includes the case of multiple non-correctable or detectable faults in the memory of the module, which is far less probable than a CPU fault.

Table B-8

Reliability Estimates for 4- and 6-Module SIFT
With Coding in Memory

4 Module
Memory per module = (13 × 2 + 15)/4 + 25% ÷ 13K = 78 chips } 84 chips
CPU per module = (.35 × 2 + .11)/4 ÷ .2 = 6 chips } 84 chips
Total chips = 332
P[CPU fault] = $.6 \times 10^{-5}$/per module
P[single memory fault] = $.8 \times 10^{-4}$
P[double memory fault] = $.6 \times 10^{-8}$
P[LC task failure] = $.6 \times 10^{-5}$
P[reconfiguration] = $.3 \times 10^{-3}$
P[second module fail] = $.8 \times 10^{-7}$
P[MC task fail] = 1.3×10^{-11}
6 Module
Total chips = 348
P[LC fail] = $.4 \times 10^{-5}$
P[MC fail] = 2×10^{-11}

Note that in the SIFT architecture, either with or without coding the occurrence of faults reduces the available CPU power preventing the MC/L tasks from being carried out. The CPU is only approximately 8 percent of the total. If the CPU were to be designed with double the capacity (in MIPs) the cost would rise by 8 percent but the probability of system failure would, for some systems, be improved immensely. The results for the different configurations are displayed in Table B-9 and in Figure B-2.

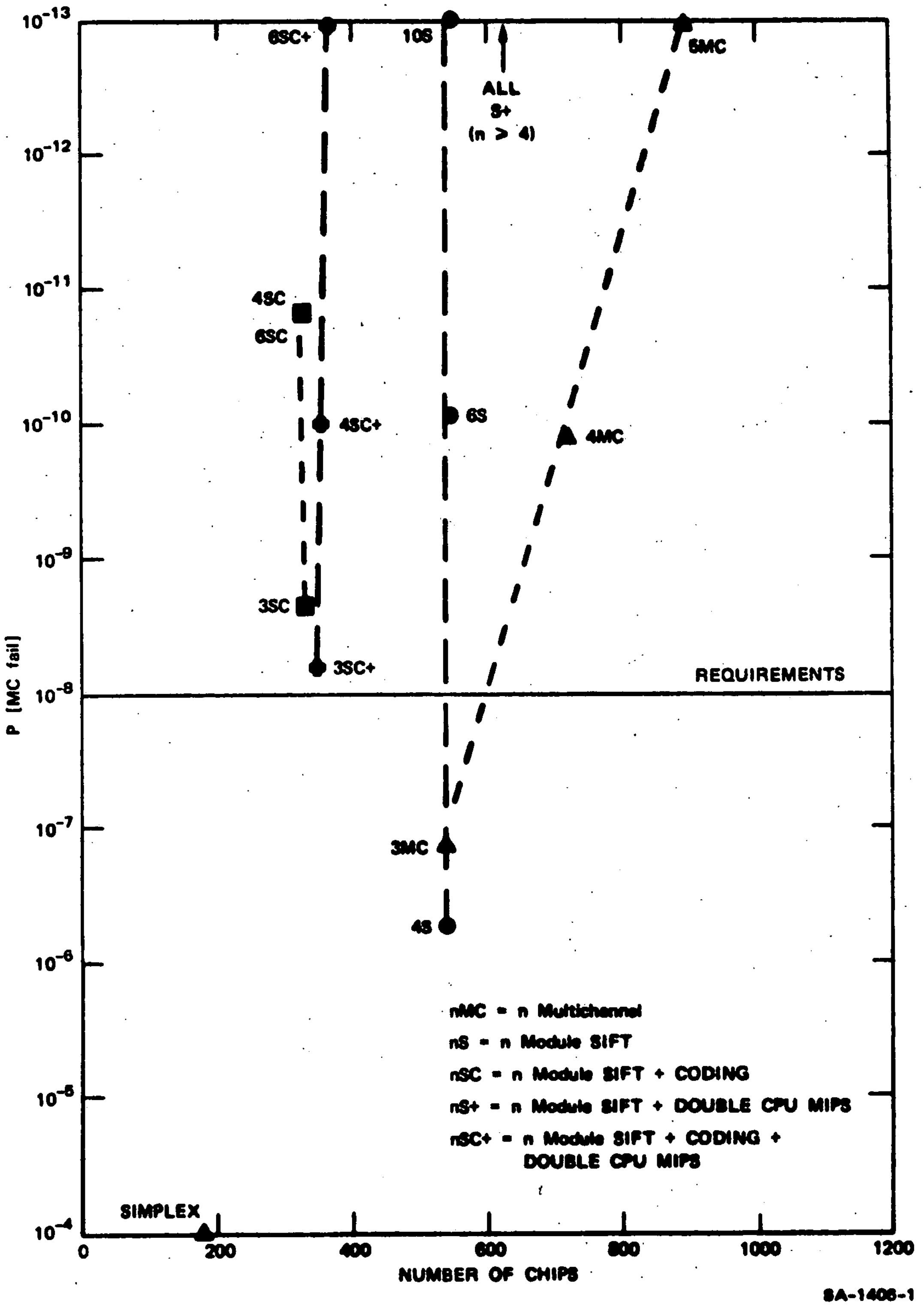


FIGURE B-2 PROBABILITY OF FAILURE OF MOST CRITICAL FUNCTIONS P [MC fail] AGAINST NUMBER OF CHIPS

Table B-9

Summary of Size and Reliability of Alternative Architectures

System	No. of Chips	P[LC fail]	P[MC fail]
REQUIREMENTS		$0.(10^{-4})$	10^{-8}
SIMPLEX	170	2×10^{-4}	
<u>MULTI-CHANNEL</u>			
3 MODULE	510	1.7×10^{-7}	
4 "	680	1.2×10^{-10}	
5 "	850	1.0×10^{-13}	
<u>SIFT</u>			
4 MODULE	544	5.4×10^{-4}	6×10^{-7}
6 "	546	2.5×10^{-6}	9×10^{-11}
10 "	550	2.7×10^{-7}	7.3×10^{-4}
<u>SIFT + CODING</u>			
3 MODULE	330	8×10^{-6}	2.4×10^{-9}
4 "	332	6×10^{-6}	1.3×10^{-11}
6 "	348	4×10^{-6}	2×10^{-11}
<u>SIFT (DOUBLE CPU MIPs)</u>			
4 MODULE	584	2×10^{-7}	$\approx (10^{-14})$
6 "	588	$\approx (10^{-10})$	$\approx (10^{-18})$
10 "	590	-	-
<u>SIFT + CODING + DOUBLE CPU MIPs</u>			
3 MODULE	354	1.6×10^{-5}	9×10^{-9}
4 "	356	1.2×10^{-5}	$\approx (10^{-10})$
6 "	372	8×10^{-6}	$\approx (10^{-4})$