AMT File Copy
Nas12-33
CR86141
N69-23811

Interim Scientific Report 4

TECHNIQUES FOR THE REALIZATION OF ULTRARELIABLE SPACEBORNE COMPUTERS

STANFORD RESEARCH INSTITUTE

MENLO PARK, CALIFORNIA

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

December 1968

Interim Scientific Report 4

TECHNIQUES FOR THE REALIZATION OF ULTRARELIABLE SPACEBORNE COMPUTERS

By: J. GOLDBERG H. S. STONE A. WAKSMAN

Prepared for:

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
ELECTRONICS RESEARCH CENTER
575 TECHNOLOGY SQUARE
CAMBRIDGE, MASSACHUSETTS 02139

CONTRACT NAS12-33

S R I Project 5580

Approved: D. R. BROWN, DIRECTOR
Information Science Laboratory

TORBEN MEISLING, EXECUTIVE DIRECTOR
Information Science and Engineering

Copy No. 15

ABSTRACT

This is the fourth scientific report of a study dedicated to the development of techniques for the realization of ultrareliable, high-performance, spaceborne computers. The techniques developed are in support of computer structures in which reliability is achieved through autonomously controlled logical reconfiguration and fault masking. The report presents techniques for the accommodation of faults in data commutation networks based upon crossbar-type switching arrays. Schemes are developed for accommodating switching failures and for embedding logic for the control of alternative switching setups within the network. Several schemes are developed that are appropriate for random and for correlated fault types. The second topic is a consideration of criteria for the selection of an instruction set for a spaceborne computer. Several merits are found for the use of stack-organized instructions, together with special registers for specifying the context of a process. Possible advantages of indirect addressing are also discussed.

FOREWORD

This is an interim report, summarizing the major results of work accomplished during the first six months of the third phase of a three-year program, the goal of which is the development of techniques for the realization of ultrareliable space computers. This study has been conducted in the Computer Techniques Laboratory of Stanford Research Institute, under the sponsorship of the Electronics Research Center of the National Aeronautics and Space Administration.

The goals of the first phase were to survey the state of the art of design for achieving ultrareliable spaceborne computers, and to form a basis for research that would advance that art. The final report, which resulted from the first phase of the program, was concerned with the following:

- The basic characteristics of an advanced spaceborne computer
- A description of fault-masking techniques for general logic functions
- A survey of codes for storage and arithmetic operations
- Problems of system organization for dynamic error control
- Tests for diagnosis of fault conditions
- Some initial descriptions of network designs for a reconfigurable computer, including commutation or interconnection networks, programmable processing modules, and programmable control units
- Error-control techniques for memory systems
- Distributed power supply systems

- The application of magnetic logic
- A survey of the published literature on the attainment of reliable systems through the use of redundancy.

The goal of the second phase was to develop detailed techniques for the logical design of an advanced, ultrareliable spaceborne computer. The techniques to be developed were to demonstrate the feasibility of achieving reliability through autonomously controlled logical reconfiguration and fault masking. An investigation was to be conducted of techniques for effecting the reconfiguration at various logical levels in the system.

These techniques have been developed in five steps. The first step was the development of a system organization that facilitates dynamic maintenance processes. In the second step, on the basis of the selected system organization, a detailed logical design was performed of networks that are uniquely appropriate for a reconfigurable computer. Third, diagnostic procedures, reliability enhancement techniques, and reliability analysis measures were developed for these networks, where the requirements exist. A fourth step in the approach involved the investigation of software techniques to aid in the diagnosis, detection, and recovery from failures. Also concerning software, some techniques were developed for designing reliable programs. A final step, yet to be completed, is the development of reliability analysis techniques for the overall system.

The interim report that resulted from the first six months of effort on the second phase was concerned with the following:

- The sketch of a multiprocessor system organization, and a description of operating policies that embodied self-checking and self-repair
- The development of logical design techniques for networks identified with the memory, control, and microprogram control functions

- The development of design principles for commutation networks, which perform the important function of data switching in a multiprocessor
- A formal description of program-design techniques that facilitate the composition of mistake-free programs.

An important conclusion of the research completed at this stage was that a byte-sliced structure was particularly attractive for the implementation at a low system level, because of the high degree of system reconfigurability provided for a given number of reconfiguration switches.

The interim report on the work of the second six months of the second phase was concerned with the following:

- Review of the goals, methods, and assumptions of the study
- Techniques for the design of a reconfigurable microprogrammed processing unit
- Techniques for controlling failures within random-access memory systems, with primary emphasis on the word-access function
- Further results on commutation networks
- Discussion of software problems of ultrareliability
- Techniques for the diagnosis of iterative digital networks.

The present report is concerned with two subjects: (1) Fault accommodation schemes for crossbar data commutation networks, and (2) Criteria for the selection of an instruction set for a spaceborne computer. During the current period we have also investigated the problem of the design of executive systems for a self-diagnosing, self-reconfigurable computer. The approach taken has been to employ the formalism of finite-state sequential machines for the description of executive control programs. This work is currently in progress, and the results are too premature for inclusion in the present report.

CONTENTS

ILLUSTRATIONS

I FAULT ACCOMMODATION SCHEMES FOR CROSSBAR DATA COMMUTATION NETWORKS*

A. Introduction

Previous reports of this study[1][†] have given extensive treatments of data commutation networks based upon the 2-permuter module. In general, the networks have been quite economical in numbers of switch and control circuits. Two disadvantages, compared to the well-known crossbar network scheme, are the high number of levels of logic traversed by a switched signal, and the complexity of setup logic. In this chapter, several schemes for accommodating faults in crossbar switches are developed. A major consideration in this study is the number of levels of logic. Economical logical design techniques for effecting alternative setups in redundant networks are also illustrated.

B. Two-Level Fault-Correcting Networks

1. The Problem

The subject of this section is the design of crossbar switching networks in which single faults may be corrected, and which are limited to having no more than two levels of switching. The basic approach taken is to provide redundant switches and data paths, together with new modes of control, so that several alternate paths through the network may be set, depending upon the location and nature of the fault. The faults are assumed to be a single switch permanently closed (stuck) or permanently open.

We will first consider the case of n-input, n-output networks, and then extend the solution found to n-input, r-output networks.

---

*By A. Waksman and J. Goldberg.

†References are listed at the end of the report.

2. Networks with Equal Numbers of Inputs and Outputs

A single-level network can be excluded immediately from consideration since it is impossible to incorporate in it the capability of correcting both types of fault. This is true since any individual switch in such a network when becoming stuck at the closed state permanently connects a pair of input-output terminals.

The next step is to consider a two-level network. Here an upper bound to the size of a two-level network with single-fault-correcting capabilities can be immediately established by considering two crossbar switching networks connected in series, as depicted in Fig. 1. For n the number of inputs, the size of such a network is $2n^2$.

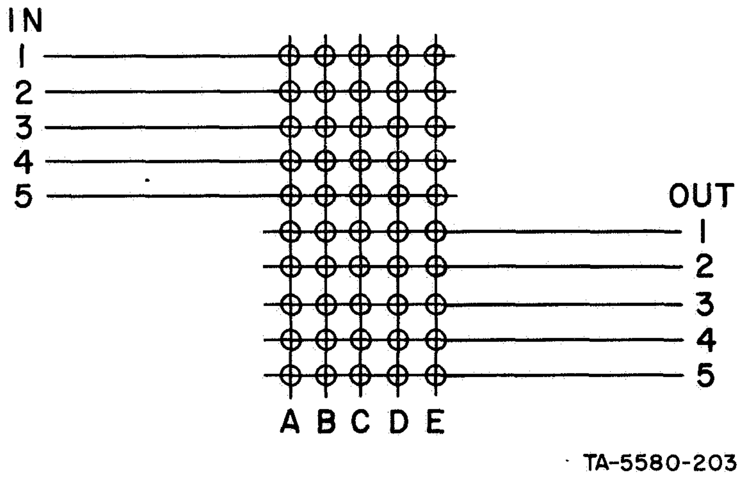


FIG. 1 SIMPLE TWO-LEVEL REDUNDANCY SCHEME FOR n x n NETWORKS

We correct an open-switch fault between a terminal and a bus by always assigning a different bus to this terminal for any given input-output assignment. We correct a closed-switch fault between a terminal and a bus by always assigning this bus to the terminal for any given input-output assignment.

To design a more efficient two-level network, we realize that none of the crossbar switches can be eliminated as long as the number of crossbars is equal to the number of terminals. This is true since by omitting a single switch, the associated terminal will be inaccessible under some fault conditions. As an example consider Fig. 1 where on the input side, switch (1,A) is omitted; then whenever on the output side, switch (1,A), (2,A), (3,A), or (4,A) becomes faulty at the closed position, i.e., permanently closed, input-output assignments including $1 \rightarrow 1$, $1 \rightarrow 2$, $1 \rightarrow 3$, or $1 \rightarrow 4$ cannot be completed.

The following approach is proposed:

> Construct a cascade network of two rectangular crossbar switches such that each of them consists of n buses crossed by (n + k) buses and the (n + k) buses are common for both switches. For design variables m and k:

$$\text{one switch performs } n! \binom{n+k}{n} \cdot \frac{1}{m} \text{ permutations} \quad ,$$

$$\text{the other switch performs } \binom{n+k}{n} \cdot m \text{ permutations}$$

It is not obvious what should be the optimum values of k and m for such a network to contain a minimum number of switches, and it is not obvious how these switches should be distributed.

One possible design for an n-input network, which gives a total of $(n^2 + 3n)$ switches, is given by $k = m = 1$. Here we assign $n \cdot (n + 1)$ switches to one level and 2n switches to the other. Figure 2 implies the general scheme by displaying a network of 5 inputs, 5 outputs, and 6 middle-buses.

Any middle-bus associated with a faulty switch in this network is made redundant; i.e., all other switches associated with it are left

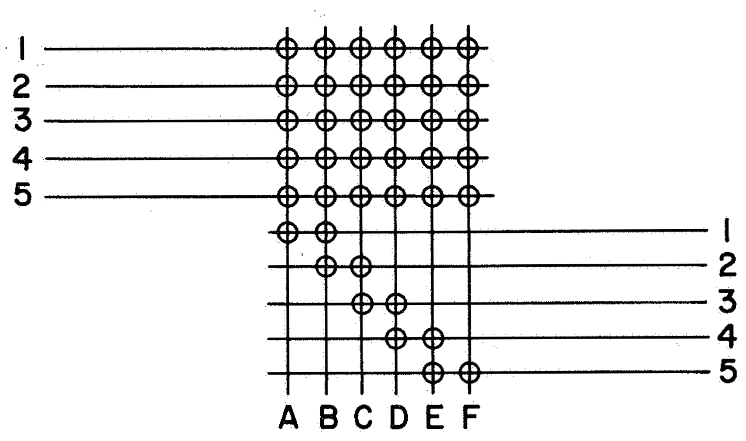


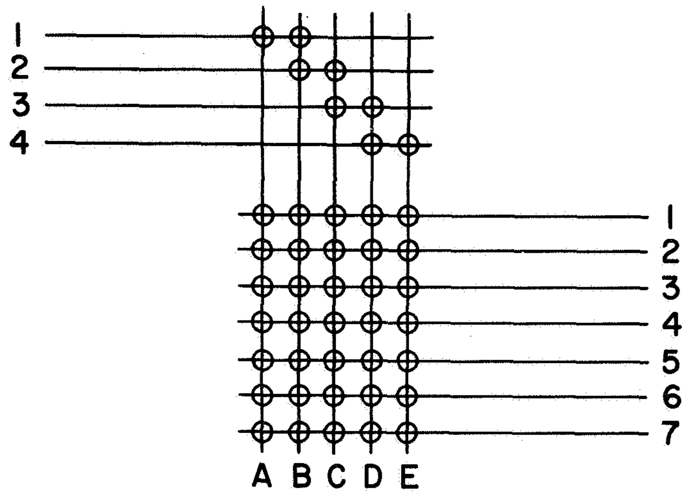
FIG. 2 ECONOMICAL TWO-LEVEL SCHEME FOR n x n NETWORKS

in the open position. We are left then with an n × n crossbar switch capable of any of the n! assignments and another crossbar switch capable of connecting any n of the (n + 1) middle-buses to the n outputs.

3. Networks with Unequal Numbers of Inputs and Outputs

The results for n × n networks may be extended easily to n × r networks. Thus, the use of a cascade of two identical crossbars requires 2nr contacts. The corresponding design for the cascade of nonidentical crossbars consists of an (r + k) × n complete crossbar [giving $\binom{r + k}{n} n!$ permutations] in cascade with a (r + k) × n crossbar [giving $\binom{n + k}{n}$ permutations], having two contacts per input, for the n inputs. This case is illustrated in Fig. 3, for n = 4, r = 7, k = 1.

The cost of the network is (n + k)r + 2n switches, that is, a cost of kr + 2n over the cost of the unprotected crossbar. For the case considered, the cost is 43 switches, while the cost of the simple, duplicate crossbar design is 56 switches.



TA-5580-205

FIG. 3 ECONOMICAL TWO-LEVEL SCHEME FOR $n \times r$ NETWORKS

4. Discussion

The proposed solution offers substantial savings in contacts compared to the "obvious" duplicate crossbar solution. It would be of interest to determine how close this solution is to the minimum. Another important problem is the extension of the design to the correction of more than one fault.

C. General Level-Limited Fault-Tolerant Networks

1. Introduction

The following is a design procedure for a single-fault-tolerant switching network with equal numbers of inputs and outputs, where the number of levels of the network enters as a design parameter. By a fault-tolerant network we mean here a network comprised of on-off contacts capable of assigning its n input terminals to its n output terminals in $n!$ ways, and also an algorithm exists for making any such assignment when a single contact is faulty, i.e., stuck-at-closed or stuck-at-open position.

Definition 1:

The number of levels of a network is the largest number of contacts a signal has to traverse in any input-output assignment.

Definition 2:

$C(N,k)$ is the number of contacts for a fault-tolerant switching network of N inputs and k levels.

Definition 3:

$c(N,k)$ is a single-fault-tolerant network of N inputs and k levels.

Observe that $k \geq 2$, for when $k = 1$ any network fails to be fault tolerant. This is true since a stuck-at-closed position contact causes a permanent link between a pair of input-output terminals. In the preceding section it was found that:

$$C(N,2) \leq N(N + 3) \qquad (1)$$

It is also known that:

$$C(N,2 \log N) \leq 4N \log_2 N \quad \text{(see Appendix A)}. \qquad (2)$$

Therefore for $2 < k < 2 \log N$ we seek an iterative construction for $c(N,k)$ such that

$$C(N,k) < C(N,k - 1)$$

Our iterative construction scheme will be as described in the appendix, namely: $c(N,k)$ will consist of two peripheral levels constructed from 2×2 crossbar switches and a center stage comprised of two $c(N/2,k - 2)$ networks. We will use the $c(N,2)$ as the network for the iteration of even k and we will use c(N,3), which is a modified Clos network, as the network for the iteration of odd k.

2. Iterative Construction for c(N,2k)

Theorem: Let c(N/2,2(k - 1)) be given; then the network of Fig. 4 is a c(N,2k).

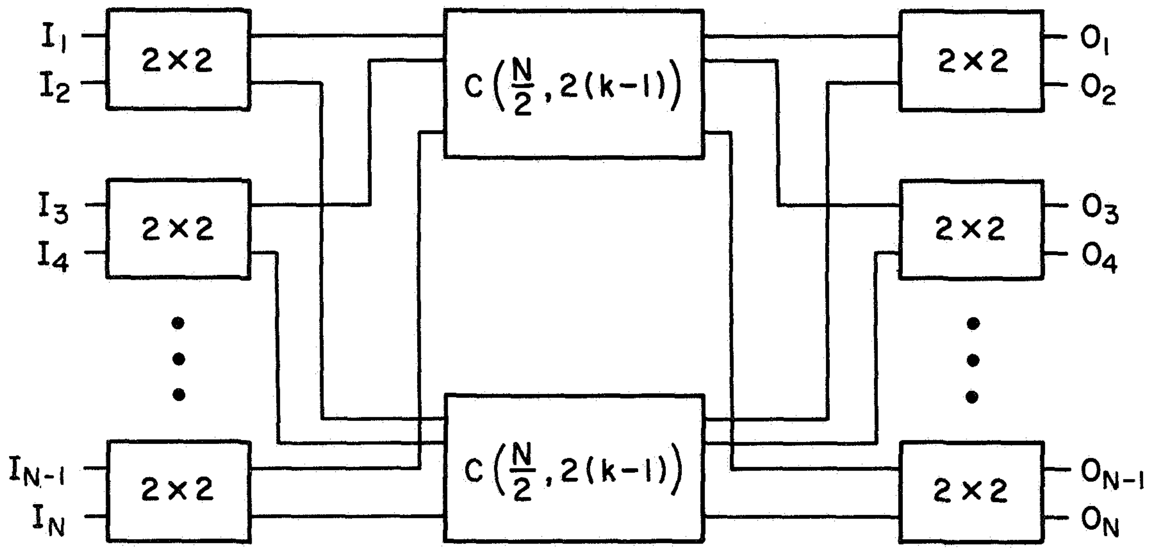


FIG. 4 ITERATIVE CONSTRUCTION SCHEME FOR n x n NETWORKS

Proof:

(a) A single fault in any of the c(N/2,2(k - 1)) is correctable by assumption.

(b) A single fault in any of the peripheral 2 × 2 switches will cause the two associated terminals to be permanently connected to one of the c(N/2,2(k - 1)) switches each. This is still a valid nonfault-tolerant network, as described in Appendix A.

By iterative construction, we have

$$C(N,2) = N(N + 3) \quad , \tag{3}$$

and

$$C(N,2k) = 4N(k - 1) + 2^{k-1}C\left(\frac{N}{2^{k-1}},2\right) \quad , \tag{4}$$

so that

$$C(N,2k) = N\left(2^{-(k-1)}N + 4k - 1\right) \tag{5}$$

Now for $N = 2^{k+m}$, so that $k = \log_2 N - m$, and $C(N,2k) = C(N,2 \log N - 2m)$, we get

$$C(N,2k) = N\left(2^{m+1} + 4 \log N - 4m - 1\right) \tag{6}$$

We observe that $2^{m+1} = 4m$ only for $m = 1$, $m = 2$; so for these values of m

$$C(N,2 \log N - 2m) = N(4 \log N - 1) \tag{7}$$

Remarks: The network $c(N,2)$ could always be constructed by cascading two crossbar switches of $N \times N$ each, so that $C(N,2) = 2N^2$ contacts. However, for $N = 2$, $N(N + 3) > 2N^2$ in Eq. (3), so that for $N = 2^k$ and $m = 0$, we have $C(N,2k) = C(N,2 \log N) = N(4 \log N + 1)$. But since there are in the network $N/2$ $c(2,2)$ networks which could be replaced by two 2×2 cascaded switches each, and since for $N = 2$, $2 \times 2^2 = 8$ and $2(2 + 3) = 10$, there is a saving of 2 contacts in each such replacement; thus we have a total savings of $2 \times N/2 = N$ contacts so that the new construction gives

$$C(N,2 \log N) = 4N \log N \tag{8}$$

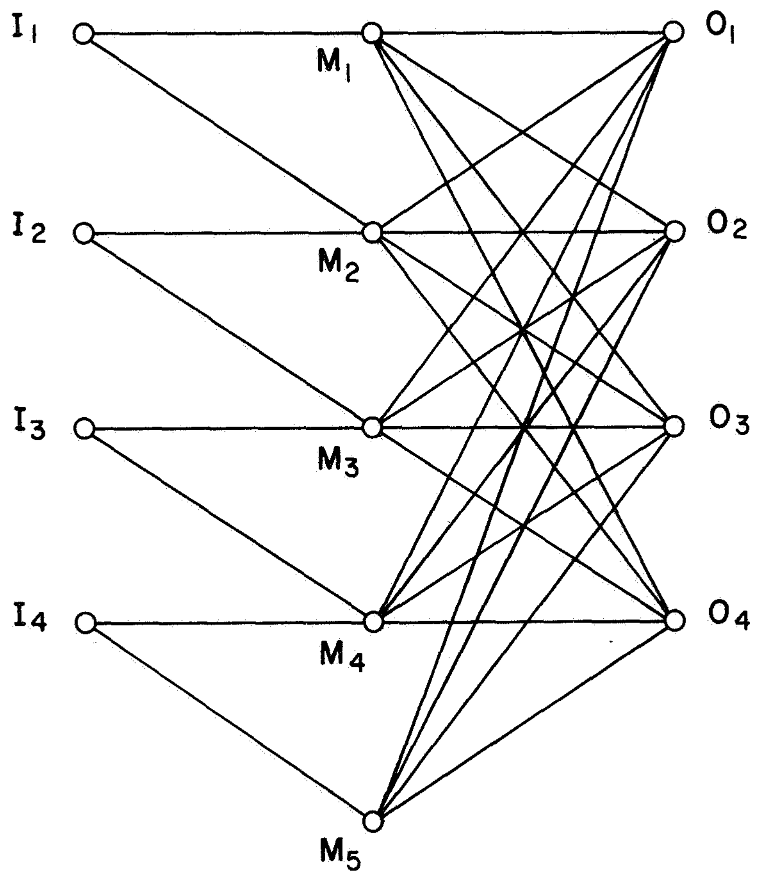
We can conclude that a minimal network is arrived at for:

$$C(N,2 \log N - 2) = C(N,2 \log N - 4) = N(4 \log N - 1)$$

3. An Assignment Algorithm

In general, for a switching network the price we pay for an increase in contact economy is an increase in the complexity of the assigning algorithm.

Figure 5 represents c(4,2) where the nodes are contact bus-wires and a line represents the existence of a contact between the respective two bus-lines.



TA-5580-206

FIG. 5 GRAPH MODEL OF AN ASSIGNMENT SCHEME

Consider an assignment algorithm for the network of Fig. 5. Let the required assignment be:

$$\begin{pmatrix} I_1, & I_2, & I_3, & I_4 \\ \\ O_x, & O_y, & O_z, & O_w \end{pmatrix} ,$$

where x, y, z, w range over 1, 2, 3, 4, mutually exclusive. Under the no-fault condition we make the following assignments:

$$\begin{pmatrix} I_1, & I_2, & I_3, & I_4 \\ \\ M_1, & M_2, & M_3, & M_4 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} M_1, & M_2, & M_3, & M_4 \\ \\ O_x, & O_y, & O_z, & O_w \end{pmatrix}$$

Notice that M_5 was excluded from the assignment since it was redundant. Now, under a fault condition we exclude M_i (i = 1, 2, 3, 4, 5), associated with the faulty contact. Let the contact between M_3 and O_2 be faulty; then our assignment will be:

$$\begin{pmatrix} I_1, & I_2, & I_3, & I_4 \\ \\ M_1, & M_2, & M_4, & M_5 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} M_1, & M_2, & M_4, & M_5 \\ \\ O_x, & O_y, & O_z, & O_w \end{pmatrix}$$

From the above example we can easily generalize the algorithm to any c(N,2). Namely, for any assignment, under a fault condition, make the middle node associated with the fault redundant. The assignment algorithm for a four-level network will consist of going through a double assignment as explained in Appendix A for the peripheral 2 $\times$ 2 switches, which in turn specifies the assignments on the two c(N/2,2) networks. We can conclude that for c(N,2k) there will be 2k different subassignments of N elements each in order to complete the N $\rightarrow$ N assignment.

4. Iterative Construction for c(N,2k + 1)

Theorem: Figure 6 is a modified three-level Clos network which is single-fault tolerant, i.e., c(N,3).

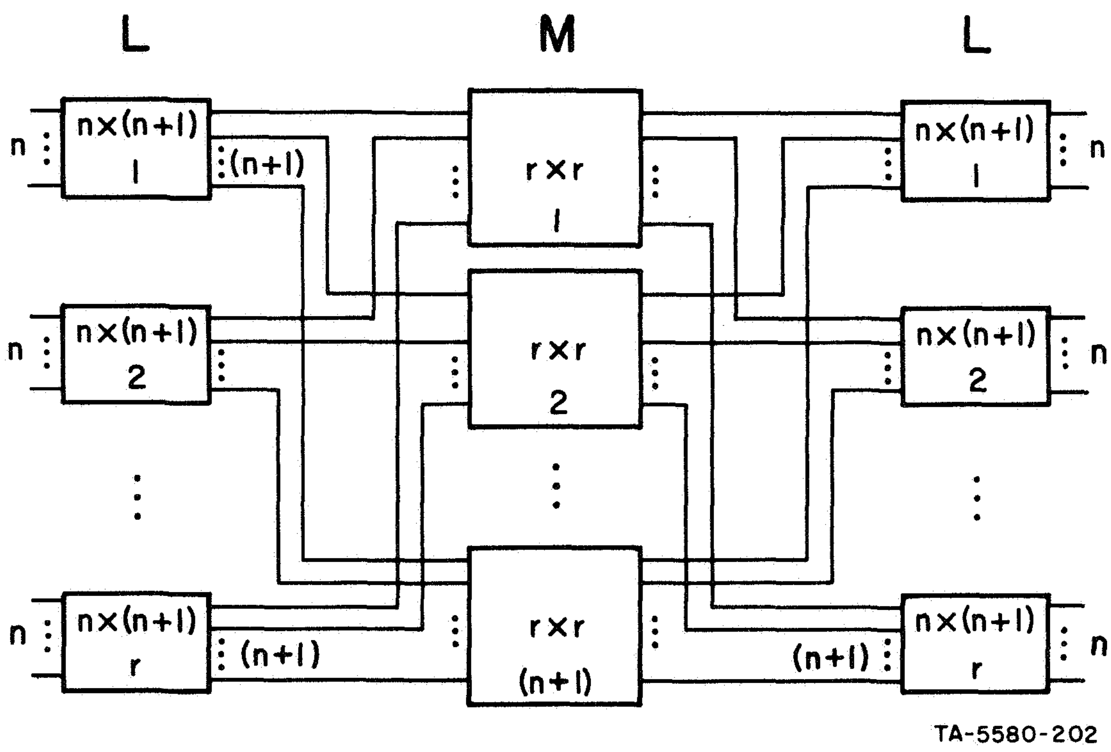


FIG. 6 A SINGLE-FAULT-TOLERANT THREE-LEVEL CLOS NETWORK

Proof: Let a fault accrue in one of the L boxes; then we omit from the assignment consideration the following:

(a) One of the (n + 1) crossbars associated with the faulty contact

(b) The link associated with the crossbar of (a)

(c) The M box associated with the link in (b)

(d) All links associated with the M box of (c).

We are then left with a Clos-type network which is nonfault tolerant but which is still a complete switching network.

From Fig. 6, we have

$$C(N,3) = 2rn(n + 1) + r^2(n + 1)$$

$$C(N,3) = 2rn^2 + 2rn + r^2m + r^2$$

and clearly $N = nr$, so that

$$C(N,3) = 2Nn + 2N + \frac{N^2}{2} + \frac{N^2}{n^2}$$

To find the value of n for minimal $c(N,3)$, we get

$$\frac{d}{dn} C(N,3) = 2N - \frac{N^2}{n^2} - \frac{2N^2}{n^3} = 0$$

and for $n \gg 2$ we get

$$N \cong 2n^2 \quad ,$$

and

$$r \cong 2n \quad ,$$

so that:

$$C(N,3) = 8n^2(1 + n)$$

or

$$C(N,3) = 4n\left(\left(\frac{N}{2}\right)^{1/2} + 1\right)$$

We proceed to construct $c(N,5)$ in the same manner that we have constructed $c(N,4)$ so that

$$C(N,5) = 4N + 2C\left(\frac{N}{2}, 3\right)$$

and

$$C(N,2k + 1) = 4N + 2C\left(\frac{N}{2},2k + 1\right) \quad ,$$

so that

$$C(N,2k + 1) = 4N\left(\left(\frac{N}{2^k}\right)^{1/2} + k\right)$$

It is possible to arrive at a further saving of contacts for c(N,3) by observing the following:

(a) In a regular three-level Clos network a single L switch can always be eliminated if we can start with the assignment procedure using the inputs to this switch <u>first</u>.

(b) In a c(N,3) we can reduce one L switch to a simple switch with 2n contacts which assigns its n inputs to any n of the (n + 1) outputs, thus accommodating a single fault in the switch and allowing for the rest of the assignment procedure to be followed.

Thus, there will be a saving of $(n^2 - 2n)$ contacts in every c(N,3) making

$$C(N,3) = 2N\left((2N)^{1/2} + \frac{1}{2}\left(\frac{1}{2N}\right)^{1/2} + \frac{3}{4}\right)$$

5. Discussion

We have considered only the case of equal numbers of inputs and outputs. As an example of the savings obtained by going to a third level, the cost of an (8,3) switch is 79 contacts, while the cost of an (8,2) switch is 88 contacts. The case of n × r three-level switches has not been treated as yet, but the relative savings, compared with n × r two-level switches, may be expected to be at least as great as for the n × n case.

The design of multilevel switches for two or more arbitrary faults remains an interesting, unsolved problem.

D. Reconfiguration Control

1. The Problem

In some multiprocessor designs, the data switching networks may be required to change settings at each instruction. For the redundant-path schemes discussed in Secs. I-B and I-C, it is therefore necessary to provide for rapid "calculation" of the alternative paths for particular input-output terminal pairings.

It appears that the amount of logic required for choosing between alternate paths is small and its structure is uncomplicated. In this part, a hardware scheme is described for the 5 X 6 X 5 network described in Sec. I-B.

2. An Example of a Hardware Solution

The basic selection logic for a nonredundant crossbar is illustrated in Fig. 7. The address of the output bus to which the input terminal is to be connected is applied to a decoder, one of whose outputs sets a flip flop, which controls the appropriate data switch.

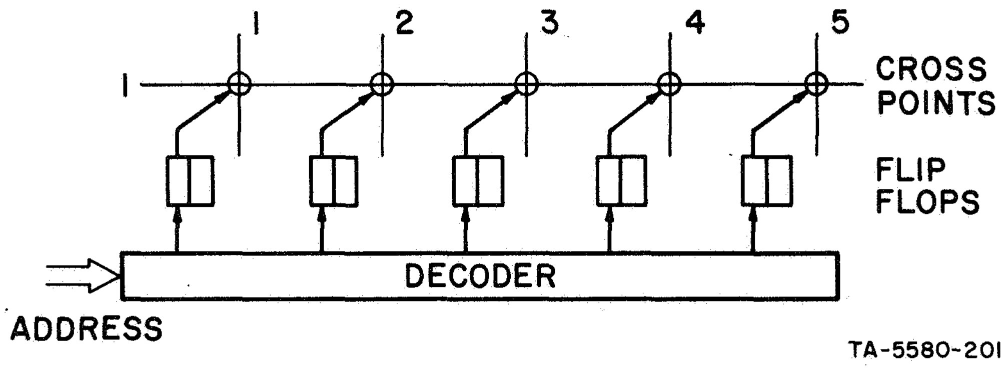


FIG. 7 SELECTION LOGIC FOR A NONREDUNDANT CROSSBAR

A suggested scheme for the 5 $\times$ 5 redundant-cascade crossbar, with six intermediate buses, is shown in Fig. 8. Here, the decoder outputs feed the flip flops via a ladder network, composed of 2 $\times$ 2 permuter cells. This ladder, which has been employed in earlier reports, provides for the one-place, right-shifting of all signals to the right of a specified column. By this means, if a vertical bus is to be disconnected (B_2 in the example), the decoder outputs from 2 to 5 are shifted one place to the right. Thus, if B_2 is to be disconnected, and input 1 is to be connected to terminal 2, the address "2" is applied, but the crosspoint for bus B_3 is energized.

A similar ladder network is provided at the output network. The flip flops in the upper network may also be used for controlling the lower network. Normally, when the flip flop for row 1, column 2 (top network) is energized, the crosspoint for row 2, column 2 (bottom network) is also energized. If B_2 is to be bypassed, the B_3 flip-flop output will be directed to the row 2, column 3 (bottom network), by the setting of the bottom network's ladder.

An alternative scheme to using the ladder network in the upper network is to operate on the input addresses, using special logic, such that, for the bypassing of bus j, the address $\underline{a}$ is replaced by a + 1 for $a \geq j$.

Since this function has arisen in several earlier reports, a logical design has been developed. It is illustrated in Fig. 9, for $0 \leq \underline{a} \leq 15$. A modified address $\underline{a}'$ is produced as a combinational function of the initial address $\underline{a}$ and a stored index $\underline{j}$. The scheme operates by scanning the bits of $\underline{a}$ and $\underline{j}$ from the most significant bits downward, and testing $\underline{a}$ and $\underline{j}$ in the first place in which they differ. Three intermediate functions are generated:

$(\underline{a} = \underline{j})$: true when $a_k = j_k$, for all k

$(\underline{a} > \underline{j})$: true when, for any k: $a_k \bar{j}_k (a_m = j_m$, for all $m > k) = 1$

$(\underline{a} \geq \underline{j})$: true when $(\underline{a} = \underline{j}) + (\underline{a} \geq \underline{j}) = 1$.

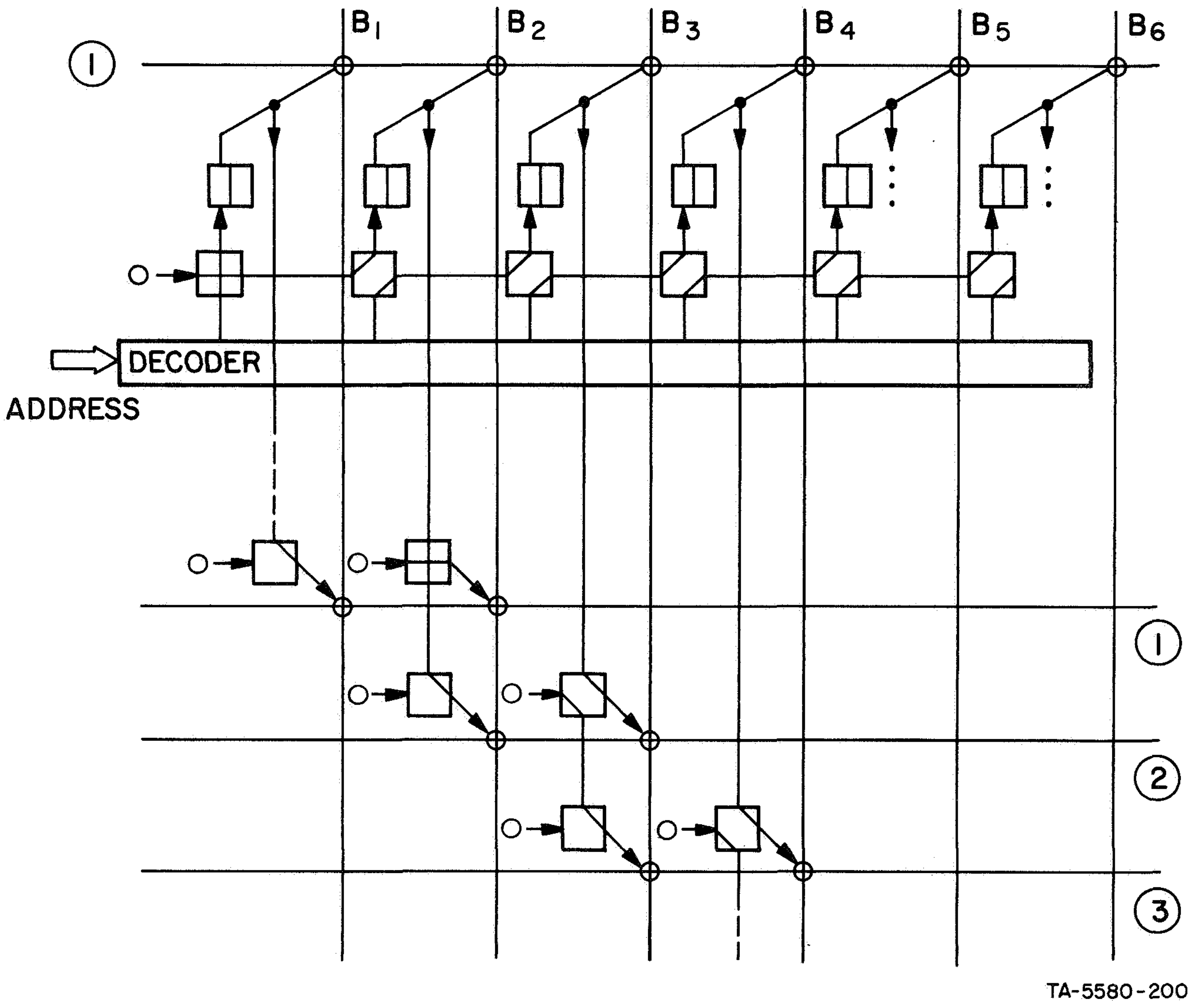
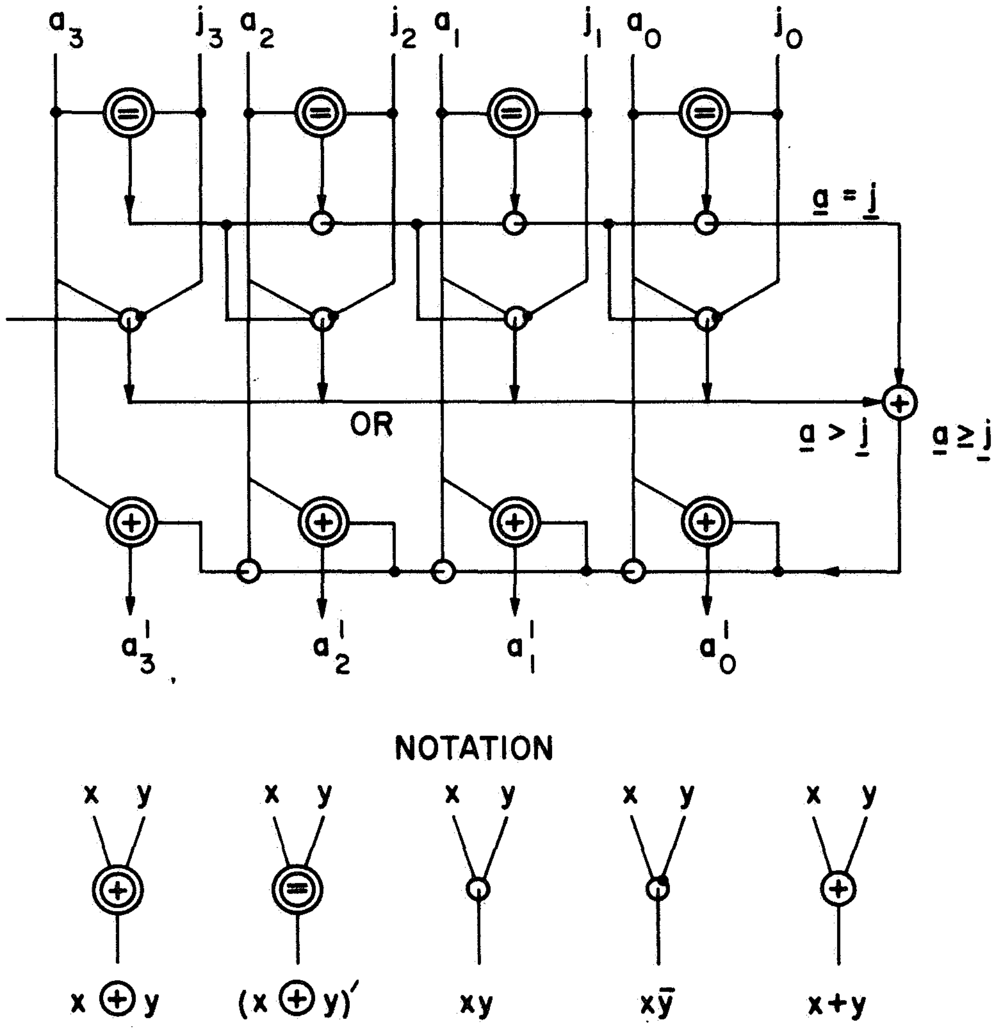


FIG. 8 SELECTION LOGIC FOR A REDUNDANT CROSSBAR



TA-5580-207

FIG. 9 NETWORK FOR ADDRESS MODIFICATION

The final result $\underline{a}'$ is produced by a set of half-adders, which adds 2^0 to $\underline{a}$ when $(a \geq j)$ is true.

The delay through the network is 2N levels for N bits. This could be reduced substantially (e.g., to about 5) by acceleration of the "carry" signals in the two AND cascades.

An alternative scheme may be constructed in which $(\underline{a} \geq \underline{j})$ is generated as the overflow of the carry chain logic of a parallel subtractor, but the delay for such a scheme is at least twice that of the design given here.

E. Switch Partitioning and Distribution of Contact

1. The Problem

The data-connection switch for a spaceborne multiprocessor will probably need to transfer data with a parallelism of from ten to 30 bits. Thus a practical crossbar switch will be composed of a set of crossbars of the type described in the preceding sections, one for each bit of the parallel data message. From the viewpoint of controlling errors in the data-switching contacts of these crossbars, the greatest number of such errors could be accommodated when each crossbar may be controlled independently of the others. Such replication of control, however, introduces a degradation of reliability due to the added equipment.

In this section, we examine the problem of partitioning a parallel-data crossbar system into independently controlled data switches, for optimum reliability. Our treatment should be considered as an introduction to the problem, since we ignore the important question of packaging (pin limitations, connector unreliability, etc.), and the possible use of redundancy within the control circuits.

We do, however, distinguish two important cases of error distribution among the data switches: (1) multiple errors independently located, and (2) multiple errors tending to occur in the same data switch. These cases correspond, respectively, to discrete-component and integrated logic component realizations. In the first case, protection is most efficiently achieved by applying redundancy within each data switch; in the second, redundancy is best applied in the form of spare, whole data switch networks. In practice, some combination of the two redundancy schemes would be likely, depending on the expected probabilities of independent and dependent faults.

The two extreme cases will be considered in the next two subsections.

2. Analysis of a System Suitable for Independent Contact Faults

We assume a data-switching system with the following parameters:

$\underline{w}$ parallel bits per word, each bit switched by a separate, identical data-switching network

$\underline{m}$ identical control modules, each controlling the setting of a group of $\underline{k}$ data-switching networks (all switches in a group have the same contact settings), thus $w = mk$.

The following failure probabilities are assumed:

q_c = probability of failure of a control unit

q_s = probability of a faulty contact in a switching net, assumed $\ll 1$, with multiple faults assumed to be independent.

The system is illustrated in Fig. 10. The dimensions of the switches are not explicit, although, of course, they affect q_s indirectly. It is clear that the system can accommodate the condition of at least m switching networks with a single fault each. As it stands, the system cannot handle more than one fault in any one network. This point will be discussed further in the conclusion.

Although some double-fault conditions will not cause the failure of a group, we assume that the overwhelming number of possible combined

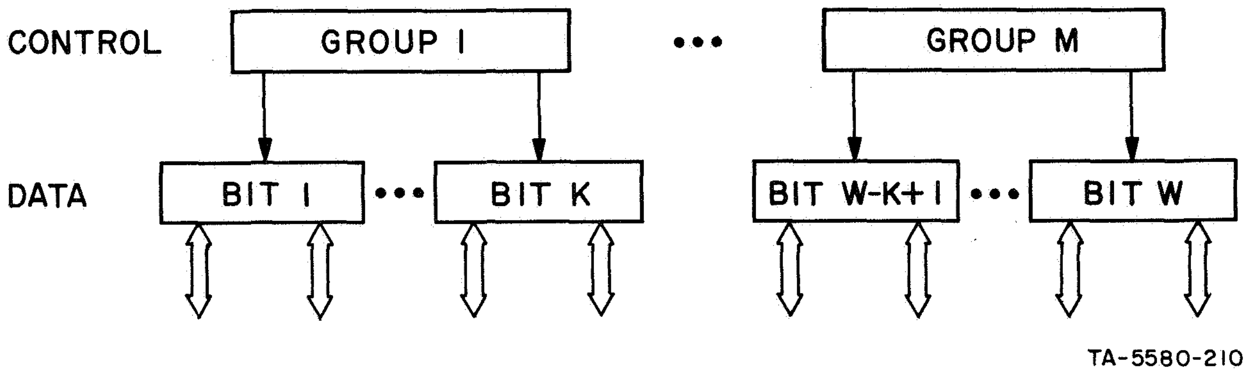


FIG. 10 PARTITIONED SWITCHING SYSTEM FOR CORRECTION OF INDEPENDENT FAULTS

faults will do so. Further, since we assume $q_s \ll 1$, the probability of more than three faults may be taken as negligible compared to the probability of two. This argument enables us to take the probability of group failure as the probability of exactly two faults. By virtue of our assumption on the independence of faults, there are k^2 possible distributions of two faults in the group.

It follows that

$$P_1 = \text{probability of group failure due to switch-net faults} = k^2 q_s .$$

Considering the failures due to the control unit for each group,

$$P_2 = \text{probability of successful group} = (1 - k^2 q_s)(1 - q_c) .$$

For the system,

$$\begin{aligned} P_3 &= \text{probability of a successful system} = P_2^m \\ &= (1 - k^2 q_s)^m (1 - q_c)^m \approx (1 - mk^2 q_s)(1 - mq_c) \\ &\approx 1 - mk^2 q_s - mq_c \approx 1 - m \frac{w^2}{m^2} q_s - mq_c \\ &\approx 1 - \frac{1}{m} (w^2 q_s) - mq_c . \end{aligned}$$

Treating m as a continuous variable, and differentiating P_3 with respect to m, the optimum number of groups is found to be

$$m_{opt} = w\left(\frac{q_s}{q_c}\right)^{1/2} \quad ,^*$$

with corresponding system reliability

$$P_3 = 1 - 2w\sqrt{q_s q_c}$$

3. An Approach to the Accommodation of Nonindependent Array Faults

If the dominant failure mode is the failure of an entire array, (e.g., due to the breaking of a monolithic logic chip containing several or all the elements of a switch network), the use of redundancy within an array will not be effective. A more effective approach is the use of spare networks, as in bit-slicing organization.

Such an approach, for a W-bit system, is illustrated in Fig. 11. One spare network is provided. Each channel is converted to a switching network via a 2×2 crossbar, i.e., a 2-permuter, as used (ubiquitously) in our designs of commutation switches. The 2-permuters are connected so as to permit the displacement of a signal to the right-adjacent network. In Fig. 11, the settings are shown for the bypassing of network 2.

One bit of control information is sufficient for the set of 2-permuters serving one network. Control for the setting of the data switches may be common for the entire array, unless it is desired to accommodate some independent faults, in which case the scheme of the preceding subsection may be superimposed.

It should be noted that the present scheme requires at least three levels of switching.

---

*In practice, m must be an integer in the range 1 to w.

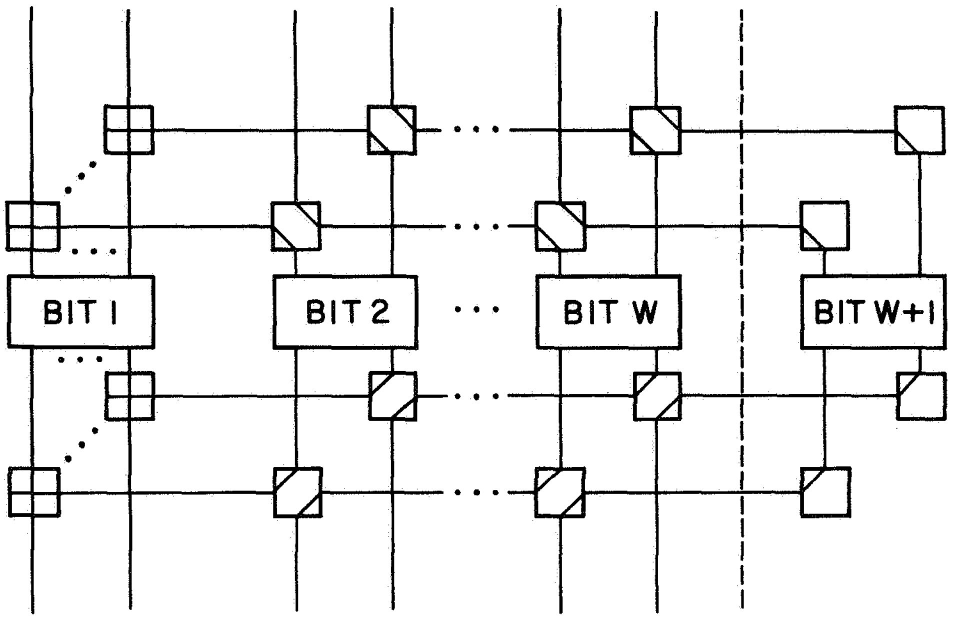


FIG. 11 SWITCHING SYSTEM FOR WHOLE-NETWORK BYPASSING

Multiple array faults may be accommodated by a straightforward extension of the scheme. The additional hardware is not costly, but each additional level of fault protection adds two levels of system delay.

4. Discussion

Schemes have been presented for accommodating faults in data-switching networks, for the cases in which faults are independently distributed among the contacts, or where faults are coherently distributed.

The schemes have not depended on any internal property of the network, so the redundancy schemes of Secs. I-B and I-C may be applied within these schemes.

An important omission in our partitioning schemes has been the consideration of packaging reliability and packaging constraints such as number of pins and number of circuits.

F. Summary and Conclusions

We have developed schemes for the control of errors in crossbar-type data commutation networks. These schemes are reasonably economical in both the data-switching logic and the control logic. Schemes were developed both for the case of independent crosspoint faults, and for the case in which such faults may be correlated, e.g., as when several crosspoints are combined in a single semiconductor device.

The general motivations for using crossbars, as opposed to networks based on permutation operations, are that the former insert fewer levels of delay and have simpler control logic, at the price of a higher component cost. The designs developed here do increase delay and complicate the control. A final evaluation would require comparison of several complete designs, for various assumed fault types. Such comparison would have to include the important factors of packaging and interconnections.

II CRITERIA FOR THE SELECTION OF AN INSTRUCTION SET FOR A SPACEBORNE COMPUTER*

A. Introduction

It is the thesis of this chapter that the instruction repertoire of a computer is intimately related to the reliability and effectiveness of a computer system. In this chapter we explore several fundamental computer processes and their implementation in different repertoires. In general, the push-down stack instruction repertoire and machine organization appears to be the best suited to spaceborne computers because of its highly efficient use of program memory, with relatively little hardware complexity. Increased memory utilization is possible with more complex instruction repertoires, but the repertoires add to machine complexity and therefore might negate gains in reliability, memory utilization, weight, and power consumption that can be attributed to the increased effectiveness of the instruction repertoires.

B. Basic Assumptions and Guidelines

The spaceborne computer environment is significantly different from the environment of land-based computer systems. In spacecraft, the precious resources are weight, volume, and power, whereas the usual precious resource for land-based computers, time, is not nearly so dear on spacecraft. The difficulty or impossibility of maintenance together with the high cost of failure places special importance on high reliability, which is seldom the case for ground computers. In considering the constraints of the spaceborne environment, it is not surprising that spaceborne computers should be designed and programmed differently from their ground-based counterparts.

As we explore the alternatives that exist for the design of a spaceborne computer, we shall assume the following:

---

*By H. S. Stone.

(1) Special hardware that contributes mainly to the reliability of the computer system might be practical for spaceborne computers while similar hardware cannot be justified for ground-based computers.

(2) Trade-offs can be made between processing speed and system reliability. It is appropriate to identify these trade-offs.

(3) After launch, the collection of programs in a spaceborne computer remains relatively constant. The volume of data within the system may fluctuate considerably.

(4) The computer system is a modular system, possibly containing multiple processors and multiple memory modules. Some or all items held in computer memory are stored redundantly so that they can still be retrieved in the event of memory failure.

(5) Instruction streams might be multiprogrammed in a single processor, and independent instruction streams might be executed simultaneously in the multiple processors.

(6) A bulk storage medium is available for spacecraft use that is suitable for storage of large files of data but is not suitable as a storage medium for programs that are undergoing execution.

The reason for the capabilities allowed in the last three assumptions is that we specifically do not want to make decisions that rule out these capabilities. It may be the case, on the other hand, that the assumed capabilities are not used. For example, a multiprocessor computer might actually be operated as a simplex processor with the unused processors idling in a low-power-drain state or performing redundant computation for error detection.

In essence, assumptions 4, 5, and 6 force us to consider certain aspects of the computer system that otherwise may be overlooked.

The next section describes the attributes of instruction repertoires that are pertinent to the design of spaceborne computers.

C. Instruction Repertoires

Two aspects of instruction repertoires concern us here, reliability and efficient use of memory. The importance of reliability in spaceborne computers is clear from the assumptions in the previous section. The importance of the efficient use of memory is due to the fact that memory size is highly correlated to the cost, volume, weight, power consumption, and reliability of a spaceborne computer system. It is by far the most precious resource in the computer system. Thus the efficient use of memory contributes substantially to the overall effectiveness of the computer system.

In the discussions that follow we will illustrate how the instruction repertoire has a very strong influence on memory utilization and system reliability, and we will attempt to derive a skeleton instruction repertoire that reflects "good" characteristics.

In most first- and second-generation computers, the instruction repertoires follow a rigid format. Each instruction occupies a half or the whole of one computer word and contains an operation code, tag bits to control effective addressing, and one or more operand addresses. In such rigid formats, as the number of addressable memory locations increases, then so must the size of the operand address fields. The larger address field in turn dictates that the word size be larger so that the number of bits in memory tend to increase as N $\log_2$ N, where N is the number of memory locations. While the factor of $\log_2$ N is not serious for small memories, it contributes to cost and degradation of reliability of larger computers. In fact, the increase in address field size is unnecessary. Programs characteristically do not address locations uniformly through memory so that there is a potential savings in memory if the most frequently used addresses are encoded with fewer bits than the less frequently referenced locations. For those programs whose behavior is independent of the total size of memory, the size of the operand address field should also be independent of the total size of memory. Consequently, it is possible in practice to eliminate the $\log_2 N$ factor in the growth rate of the number of bits in memory.

Apart from the addressing problem there are two distinct problems concerning the instruction repertoire and its relation to efficient memory utilization. The first problem is to find an encoding of a given set of instructions so that storage is used with high efficiency, yet the hardware required to implement the instructions is reasonably simple. The second problem is to select the instructions for a repertoire under the condition that there is a constraint on the total number of different instructions allowed. A measure of the utility of the instruction set is the compactness of programs for common computer processes.

The problem of encoding a specific instruction set can be solved in theory by measuring the frequency of use of various instructions and employing a Huffman code[2] scheme for encoding them with respect to the frequency distribution. Since the Huffman code yields code words of varying lengths, the problem of decoding Huffman-coded instruction places an undue burden on the computer control unit. Compromise schemes have been implemented in several computers. These schemes entail the use of several different instruction formats of varying length, but the lengths are carefully chosen to be multiples of an accommodated field length. In the instruction sets described in succeeding sections, it is assumed that multiple instruction formats of different lengths will be utilized when conditions permit.

In the following sections we investigate the problems of addressing and the selection of instructions for a repertoire.

D. Utilization of Address Fields

We assume that the address field in an instruction has a fixed length. There are several different ways that the address field can be interpreted by a processor to form an effective address. The various alternatives can be used individually or in combination in a computer system. It is our purpose to examine these alternatives and identify those that lead to the greatest storage economy.

Among the alternatives available for the interpretation of address fields are the following:

Method a--Use the entire field as an absolute memory address.

Method b--Use selected bits in the field to select one or more index registers whose contents are added to the field. The registers can hold longer addresses than the address field in the instruction.

Method c--Use a bit to specify indirection. The indirect address field can be longer than the direct address field.

Method d--Treat the address field as if it specified a contiguous region of memory, but use subfields of the address as page and segment numbers to permit memory remapping.

Method e--Use a subfield to specify addressing relative to one register of a group of registers whose state determines the context of the program. The address fields in the registers can be longer than the address field in the instruction.

Methods a, b, and c are commonly used in present computers. Method d is the well-known "paging" technique and may be used independently of a, b, and c. An example of Method e is the Dijkstra display register technique for addressing variables in ALGOL.[3] It has been implemented in hardware in the Burroughs' B-6500.[4] Methods b and e are related but are, nevertheless, distinct methods. The distinction lies primarily in the mechanism for loading the registers. We shall adopt the point of view that index registers are loaded and unloaded by issuing instructions specifically for that purpose. Context registers, on the other hand, are assumed to be modified as the side effect of instructions that cause context to be changed.

Note that when total memory size is small, absolute addresses work satisfactorily for addressing purposes. The interesting case is that in which memory is sufficiently large that it is desirable not to waste space by using absolute addresses in the instructions themselves. In the next section we examine Methods c and e more closely.

E. Addressing with Context Registers--Stack Instruction Sets

Three of the methods for utilizing address fields are concerned with addressing with the aid of machine registers. In this section we see that one of the three methods, Method e, can be implemented in a fashion that leads to very high memory efficiency. The form of implementation is known as the "stack" instruction set because the instructions perform operations on a push-down stack.

When registers are used to extend addressability (Methods b, d, and e), storage economy of addresses is achieved only if the contents of the registers can be modified to permit programs to operate in different address spaces. Two modes of operation are possible. In one mode, each program in a system of programs may operate in its own address space so that address registers need only be changed when control is passed from program to program. Within the context of any one program, the contents of the registers are fixed. The second mode of operation permits programs to operate in an address space that is larger than that immediately available in a fixed address field. In this mode, programs must modify the contents of the address registers in order to change the area of memory that is accessible to the program.

Regardless of the mode of operation, the use of registers to extend addressability has an interesting consequence on the computer system. It is essential that there be some mechanism for saving the contents of registers and for loading the registers with new data in order to permit programs to change to a new context or restore a prior context. In general, this requires memory for use as register save areas* and to hold the instructions necessary to store and reload the registers. This is a nontrivial amount of storage. (Programs written for the IBM 360 computer system following normal conventions require about 100 bytes per subroutine to perform the register saving and reloading operations.) It has been found that the memory used for the register save area storage and register manipulation can be greatly reduced by using a stack instruction set; hence this approach is very significant to our concern for efficient use of memory.

The stack instruction approach utilizes registers as described in Method e. In practice, all addresses in instructions would be paired with a tag field that specifies which register is to be added to the address to form an absolute memory address. We shall call the registers "context registers," for they determine the context of a program. A minimum of four registers are needed and these are:

(1) Program counter

(2) Global data area--the address of a data area that all subprograms share

(3) Current top of stack

(4) Local data area--the address of a data area that is "private" to the subroutine currently in execution.

We shall first discuss the operation of the stack instruction set with respect to the four registers designated above; then we shall consider some variations of the scheme.

---

*For convenience, we will refer to such areas of memory as "save areas."

A change of context within a program is normally viewed as an entry to a subroutine in the program. Each entry to a subroutine requires that some of the context registers be saved, initialized for the subroutine, and eventually restored prior to exit from the subroutine. For the scheme employing four context registers, all but the global data register must be saved. The global data register would be saved when an entirely different program is entered, e.g., when an interrupt is recognized and an interrupt program is entered.

Instead of allocating sufficient memory for saving the registers in each subroutine, the memory for save areas can be allocated dynamically by saving the registers in a push-down stack. The "call subroutine" instruction should operate by placing the three context registers at the top of a push-down stack, and the "return from subroutine" instruction should perform the inverse operation. In this way, no special instructions need be stored and issued to cause the registers to be saved and restored. Moreover, the memory requirements are determined dynamically so that the amount of memory used for register saving depends on the maximum that is actually needed.

The advantages of the stack instruction repertoire can easily be extended to the problem of passing parameters from routine to routine. Parameters have the characteristics of the context registers. In order to pass parameters from one subroutine to another, the parameters must be moved into a parameter area, and this entails the use of memory for parameter areas and for instruction sequences that move parameters to the parameter areas. In a stack machine, the stack serves well as a parameter area. The maximum space allocated for parameters is determined by the true maximum program requirements since the memory for parameters is allocated dynamically. When a subroutine is exited, the space used for parameters to that routine becomes available automatically when the stack is "popped" during the exit. A minimum amount of memory is needed for instructions to move the parameters to a communication region because it is necessary to issue only "push-down" instructions. The "push-down" instruction is effectively equivalent to a pair of instructions, "load" and "store," that would be required in a conventional organization in this case.

One last point concerning memory allocation is worth mentioning. Local memory can be allocated in part in the push-down stack, thereby effecting an economy in the use of memory through automatic dynamic allocation. Variables that are local to subroutines that must retain their identity after the routine is exited cannot be stored in the stack. They must be allocated in the global data area. The stack is appropriate only for the storage of local variables that are reinitialized at each entry of a subroutine.

Increased program reliability can be achieved through the use of reentrant programs, and the stack mechanism is completely compatible with this concept. Return addresses, for example, are stored in the stack as we have described the situation, and thus are not stored in program areas as is the case for some computers. When return addresses are stored in program areas, the self-modification of the program prevents reentrant use of the program.

In some situations a single global data register is undesirable. For example, the implementation of ALGOL 60 using a Dijkstra display requires the use of many registers to define the context of data. Additional registers can be added to a computer and still retain the character of the stack instruction set. However, additional registers require more bits in the address field to select one register from the set that defines a context. The B-5500, a stack machine, uses only one global data register and illustrates that the use of one register is not a serious limitation. Its successor, the B-6500, uses 32 registers to define a context within its stack organization.

We have mentioned that the program counter can be used to define a context within a program area. It can be used to reference constants within programs and to reference the target instructions of branches. When the program counter is used, references must generally be indicated as increments or decrements relative to the contents of the program counter. Both types of references occur because branches may be forward or backward. Since the first instructions of a program branch mainly in the forward direction, and the last instructions in a program branch mainly in the backward direction, only half of the potential addressability relative to the program counter is used at the beginning and end of a program. This points out that one can save one bit in program-relative addresses by using a program base register instead of the program counter to determine the context of items in the program area. Addressing relative to the program base register uses positive increments only.

To summarize the points in this section, we have shown that the stack instruction set yields very efficient memory utilization for the following reasons:

(1) Contexts are changed with a minimum of machine instructions.

(2) Register save areas are allocated dynamically.

(3) Parameter areas are allocated dynamically.

(4) Parameters can be moved to parameter areas with a minimum of instructions.

(5) Return addresses are stored in data areas rather than in program areas, thereby facilitating the execution of reentrant programs.

The points above strongly support the implementation of stack instructions in a spaceborne computer. We note that while it is possible to simulate a stack with both Methods b and c, the loss of efficiency in performing the stack operations in software negates much of the potential gains in utilizing a stack.

F. Extension of Addressability with Indirection

Indirect addressing (Method c) has been used in some computers to obtain increased addressability. The technique used is simply this. An address field of an instruction is used to access a storage location that contains the true memory address of an operand. Addressability is enhanced when the instruction address field is smaller than the address field used in the indirect address. Indirect addressing is compatible with the stack instruction repertoire described previously, so it can be used in combination with context registers.

There are important aspects of indirection which bear discussion, beyond the simple explanation given above. In particular, there are two different ways that indirection can be controlled. The most usual way is for the indirection to be specified by the program. A less frequently used mechanism is to specify indirection in the data itself. We shall examine these techniques in the remainder of this section.

In most implementations the instruction determines if the operation is to be executed in the direct mode or in the indirect mode. If the mode is indirect, then the contents of the reference address are inspected, to determine if it specifies another level of indirection. It is not necessary to support multiple-level indirection if one can achieve economy or reliability by supporting only a single level in hardware. One level of indirection is sufficient to achieve most of the potential economy that indirection offers.

Data-specified indirection works somewhat differently. In this mode of operation, the instruction itself does not specify whether the operation is direct or indirect. Instead it specifies a reference address that contains either the operand or the address of the operand to be used for the instruction. The reference address must be inspected in order to determine if the operation is direct or indirect. This mode of operation requires that data and indirect addresses must be distinguishable. For example, a single bit per word might be allocated to identify whether the other bits in the word are to be interpreted as an address or as a datum.

There are two points of comparison of the two techniques for implementing indirection. First we shall compare the two with respect to their utility. Then we shall consider the relative storage efficiency.

From the point of view of utility, data-controlled indirection can be used to perform all operations that can be performed by program-controlled indirection, using precisely the same number of instructions, but the converse is not true. In fact, data-controlled indirection can lead to efficiency that is not available if indirection is only under program control.

To support that claim, note that it is quite straightforward to convert a program that utilizes program-controlled indirection into an equivalent program of the same length that utilizes data-controlled indirection. On the other hand, if indirection is data-controlled, then the activity of subroutines can be controlled to greater extent at the time the subroutine is invoked. In fact, the crucial aspect of call-by-name or call-by-value can be controlled exclusively through data-controlled indirection. Thus, it is possible to achieve more general use of subroutines if data-controlled indirection is available since the question of direct vs. indirect operation for the instructions is not built into the subroutine but is resolved when the subroutine is invoked.

Now let us examine the question of relative efficiency of the two methods for specifying indirection.

If indirection is program specified, then it is necessary to extend the instruction field sufficiently to include the capability of specifying indirection. In most implementations this amounts to one bit per instruction. We mentioned earlier that data-controlled indirection can be implemented by using one bit per datum. In essence, the specification bit can be placed either in the instruction or in the data. The most efficient allocation of the indirect bit depends on the relative size of data and program, and on the potential savings in memory space that can be attributed to data-controlled indirection.

No recommendation is included here because the choice depends on characteristics of program and data that are not known at present. The important point is to recognize that a choice exists and that certain advantages accrue to data-specified indirection that have been ignored in most computer implementations.

G. Address Arithmetic

Instruction repertoires have often included instructions for performing higher-level processes. For example, it is usual to include an instruction that both increments an index and does a conditional branch. Address arithmetic operations is another class of higher-level process that is represented extensively in some instruction repertoires where this representation is primarily through the use of index register fields that control the calculation of effective addresses. In this section we examine address arithmetic associated with one- and two-dimensional arrays. The analysis shows that there is considerable improvement possible in the way address arithmetic operations are specified in a computer.

Address operations for one-dimensional arrays generally assume the following rigid format. Given the base address of an array, an index to the array, and a lower bound on the indices represented in storage, the address of the ith element of the array is computed as follows:

$$\text{address of ith element} = \text{base address} + \text{index} - \text{lower bound}$$

For two-dimensional arrays, two different generalizations have been adopted in practice. The first, polynomial addressing, requires knowledge of the upper bound of one dimension of the array as well as both lower bounds. Given an $N \times M$ array with lower bounds equal to one in both dimensions, the array elements can be arranged in storage in the order (1,1), (1,2), ..., (1,M), (2,1), (2,2), ..., (2,M), ..., (N,M). Then the address of element (i,j) can be calculated as follows:

$$\text{address of } (i,j) = \text{base} + (i - 1) \times M + (j - 1)$$

This is called <u>row major</u> order. The elements can also be stored in <u>column major</u> order; i.e., the order is computed by varying the column index more slowly than the row index. This leads to a polynomial of the form

$$\text{address of } (i,j) = \text{base} + (j - 1) \times N + (i - 1)$$

If the lower bounds can be arbitrary, then the formulas take the following general form:

$$\text{address of } (i,j) = \text{base} + \left(E_1\right) \times L_2 + E_2$$

where E_1 and E_2 are effective indices and L_2 is a length. In general, for any dimension,

$$E = \text{index} - \text{lower bound}$$

and

$$L = \text{upper bound} - \text{lower bound} + 1$$

The second method of addressing two-dimensional arrays utilizes indirection. For this form we obtain the address of element (i,j) by computing

$$\text{address of } (i,j) = [\text{base address} + i - 1] + j - 1$$

where the brackets mean "the contents of" and we assume that lower bounds are equal to 1 in both dimensions. For arbitrary lower bounds the formula generalizes to

$$\text{address of } (i,j) = \left[\text{base} + E_1\right] + E_2$$

where E_1 and E_2 are effective indices as before. Notice that the upper bound does not appear in this formula.

Calculation of addresses using the method of indirection requires that the contents of the memory locations starting at the base address of the array contain addresses of the base addresses of the rows of the array. In general, if there are L rows in an array, then L memory locations must be used to hold addresses. Hence, there is storage penalty if indirection is used. This is balanced by the time penalty associated with the polynomial calculation method due to the necessity to fetch the value of the length and to perform a multiplication.

For multidimensional arrays the two methods generalize as follows:

Polynomial:

$$\text{address of } \left(i_1, i_2, \ldots, i_k\right) = \text{base} + \left(\ldots\left(E_1\right) \times L_2 + E_2\right) \times L_3 \ \ldots\right) \times L_k + E_k\right)$$

Indirection:

$$\text{address of } \left(i_1, i_2, \ldots, i_k\right) = \left[\left[\ldots\left[\text{base} + E_1\right] + E_2\right] \ldots + E_{k-1}\right] + E_k$$

For spaceborne applications the polynomial method of addressing is attractive becuase it does not require the additional memory for holding addresses that is characteristic of the indirect method of addressing. However, it is also desirable that the instruction repertoire assist the calculation of address polynomials so that we achieve additional storage savings by reducing the length of instruction sequences.

To see how this can be accomplished, note that we must have access to both the base address of an array and to the length of the first dimension of that array when we compute indices to a two-dimensional array. By convention, let us organize the array so that the length of the first dimension is stored immediately below the (1,1) element of the array. Then the base address of the array can be computed from the address of the length of the first dimension. This observation leads to a material generalization in which the data and the instruction set take the form described as follows.

Assume that we are dealing with a k-dimensional array. The data are organized so that the first 2k locations of the storage area contain the lower bounds and lengths of the k dimensions in order, and the following locations contain the data in the array arranged in the k-dimensional generalization of row major order. (The first index varies slowest, and the last index varies fastest.) The base address of the array points to the first lower bound.

We assume that there exists an INDEX instruction in the repertoire that has the following behavior. One register is assumed to hold a base address, a second register holds a partially computed index, a third register holds the length of the current dimension, and a fourth register holds the value of the index of the "current" dimension where the lower bound has not yet been subtracted from the index. When the INDEX instruction is issued the following events occur:

(1) The base address is interpreted as the address of the lower bound of the current dimension. That lower bound is fetched into a temporary register.

(2) The lower bound is subtracted from the current index to create an effective index.

(3) The effective index is examined. A negative index indicates an error has been made.

(4) The base address is incremented by 1. (The base address now points to the length of the current dimension.)

(5) The length of the current dimension is fetched and placed in the length register.

(6) The effective index is compared to the length. If it is greater, then an error has been made.

(7) The partial index is multiplied by the length, and the result replaces the partial index.

(8) The effective index is added to the partial index to create a new partial index.

(9) The base address is incremented by 1.

Although the instruction sequence appears complex, it merely performs the operation

$$(\ldots) \times L_i + E_i$$

in the address polynomial calculation, including bounds checking. The partially computed index must be initialized to 0. This can be done in one instruction if we assume that there exists a special instruction LOAD BASE REGISTER which also performs initialization. To obtain the address or the value of an array element after indexing, we need the instructions LOAD INDEXED ADDRESS or LOAD INDEXED VALUE that add the base address to the partial index at the end of an address computation.

Multiple precision and packed arrays fall into this general scheme of operation. To index into arrays such that one array element occupies p words (p may be an integer or a rational fraction), we must multiply the partial index by p just prior to the addition of the partial index to the base address. This can be done either by issuing an instruction to perform the multiplication or by incorporating the operation into the LOAD INDEXED ADDRESS or LOAD INDEX VALUE instructions. In the latter case, the parameter p would be stored immediately following the upper and lower bounds.

Using the instructions above we have the following examples. To compute the value of A[i], we issue

```
LOAD BASE ADDRESS       A

INDEX                   i

LOAD INDEXED VALUE
```

To compute the address of B[i,j], we issue

```
LOAD BASE ADDRESS       B

INDEX                   i

INDEX                   j

LOAD INDEXED ADDRESS
```

Although we have not stated how the registers associated with the INDEX instruction are organized, there is some advantage in using a push-down stack to avoid the problems of register allocation and saving-reloading. The push-down stack organization allows the efficient computation of expressions such as the following:

$$A[B[i,j], C[k]]$$

where the stack is used for temporary storage of the registers used by the INDEX instruction.

The instructions introduced here are designed to facilitate commonly executed program process. Although they are complex, they are adaptable to a general form of array storage. Moreover, the data are organized to facilitate automatic bounds checking so that increased reliability is obtained without a corresponding increase in the number of program instructions. The instructions can be made simpler in this instance if we assume that all lower bounds are 0. This saves the subtraction of the lower bound and it also saves the storage location allocated to the lower bound.

More generally speaking, the analysis carried out in this section might fruitfully be applied to other common processes to arrive at instructions that assist processing in similar ways. Other processes of particular interest are operations on data structures such as stacks, queues, trees, and packed-data structures.

H. Summary and Conclusions

This chapter has concentrated on three particular aspects of instruction repertoires--address specification, indirection, and indexing.

With respect to address specification, the context registers of a stack-organized instruction set offer a potentially significant savings in memory utilization. For this reason a strong case can be built for adopting this mechanism in an instruction repertoire instead of more popular alternatives.

The questions of indirection and address calculations are still unsettled. The points raised here indicate that there exist alternatives that have often been neglected in the usual implementation but which are potentially useful in spaceborne applications. The discussion has also indicated the direction that further exploration might take in order to resolve the outstanding questions.

Appendix A

FAULT PROTECTION FOR PERMUTATION NETWORKS

The paper "A Permutation Network"[5] describes a switching network comprised of N log N - N + 1 cells where each cell can be looked at as a 2 × 2 crossbar switch, thus making a total contact count of 4(N log N - N + 1).

Now in Fig. 2 of the above paper if P_A and P_B are fault-tolerant networks and an additional 2 × 2 switch is added at inputs V_1 and V_2, we can consider the whole network fault tolerant since the assignment algorithm in page 6 can be followed starting with the inputs to a faulty peripheral 2 × 2 switch. The total modification of a switching network on 2 log N - 1 levels will consist of

$$4\left(2^0 + 2^1 + 2^2 + \ldots 2^{(\log N-1)}\right)$$

additional contacts and one additional level. The added level is due to the duplication of the center level for fault-tolerance consideration, so that for $N = 2^k$

$$4\left(2^0 + 2^1 + 2^2 + \ldots 2^{k-1}\right) = 4\left(2^k - 1\right) = 4(N - 1) \quad ,$$

we have

$$c(N, 2 \log N) = 4N \log N$$

REFERENCES

1. J. Goldberg et al., "Techniques for the Realization of Ultrareliable Spaceborne Computers," Interim Scientific Report 3, Contract NAS 12-33, SRI Project 5580, Stanford Research Institute, Menlo Park, California (June 1968).

2. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE, Vol. 40, No. 10, pp. 1098-1101 (October 1952).

3. B. Randell and L. J. Russell, Algol 60 Implementation (Academic Press, London and New York, 1964).

4. E. A. Hauck, B. A. Dent, "Burroughs' B6500/B7500 Stack Mechanism," AFIPS Conf. Proc., SJCC, pp. 245-251 (Thompson Book Co., Washington, D.C., 1968).

5. A. Waksman, "A Permutation Network," J. ACM, Vol. 15, No. 1, pp. 159-163 (January 1968).

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Stanford Research Institute 333 Ravenswood Avenue Menlo Park, California 94025	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP N/A

3. REPORT TITLE

TECHNIQUES FOR THE REALIZATION OF ULTRARELIABLE SPACEBORNE COMPUTERS

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Interim Scientific Report 4

5. AUTHOR(S) *(First name, middle initial, last name)*

Jacob Goldberg Harold S. Stone Abraham Waksman

6. REPORT DATE December 1968	7a. TOTAL NO. OF PAGES 62	7b. NO. OF REFS 5
8a. CONTRACT OR GRANT NO. NAS12-33 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Interim Scientific Report 4 SRI Project 5580 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. DISTRIBUTION STATEMENT

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY NASA Electronics Research Center Cambridge, Massachusetts 02139

13. ABSTRACT

This is the fourth scientific report of a study dedicated to the development of techniques for the realization of ultrareliable, high-performance, spaceborne computers. The techniques developed are in support of computer structures in which reliability is achieved through autonomously controlled logical reconfiguration and fault masking. The report presents techniques for the accommodation of faults in data commutation networks based upon crossbar-type switching arrays. Schemes are developed for accommodating switching failures and for embedding logic for the control of alternative switching setups within the network. Several schemes are developed that are appropriate for random and for correlated fault types. The second topic is a consideration of criteria for the selection of an instruction set for a spaceborne computer. Several merits are found for the use of stack-organized instructions, together with special registers for specifying the context of a process. Possible advantages of indirect addressing are also discussed.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6801

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Data Switching Networks						
Reliability						
Logical Redundancy						
Spaceborne Computers						
Computer System Organization						
Memory Addressing						