SEL-76-032

PRINCIPLES OF SELF-CHECKING PROCESSOR DESIGN AND AN EXAMPLE

John F. Wakerly

Technical Report No. 115

December 1975

This work was supported by the Joint Services Electronics Program
under contract N-000 14-75-C-060 1 and by the
National Science Foundation under grant GK-43322

DIGITAL SYSTEMS LABORATORY
STANFORD ELECTRONICS LABORATORIES
STANFORD UNIVERSITY · STANFORD, CALIFORNIA

SEL-76-032

PRINCIPLES OF SELF-CHECKING PROCESSOR DESIGN AND AN EXAMPLE

John F. Wakerly

Technical Report No. 115

December 1975

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

This work was supported by the Joint Services Electronics Program under contract N-00014-75-C-0601 and by the National Science Foundation under grant GK-43322.

PRINCIPLES OF SELF-CHECKING PROCESSOR DESIGN AND AN EXAMPLE

John F. Wakerly

Technical Report No. 115

December 1975

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

ABSTRACT

A self-checking processor has redundant hardware to insure that no likely failure can cause undetected errors and all likely failures are detected in normal operation. We show how error-detecting codes and self-checking circuits can be used to achieve these properties in a microprogrammed processor. The choice of error-detecting codes and the placement of checkers to monitor coded data paths are discussed. The use of codes to detect errors in arithmetic and logic operations and microprogram control units is described. An example processor design is given and some observations on the diagnosis and repair of such a processor are made. From the example design it appears that somewhat less than 50% overall redundancy is required to guarantee the detection of all failures that affect a single medium- or large--scale integration circuit package.

INDEX TERMS: Arithmetic codes
Checkers
Control units
Diagnosis
Error-detecting codes
Logical operations
Microprogramming
Self-checking circuits

1. INTRODUCTION

A self-checking processor has redundant hardware to detect internal failures. Processors with some degree of self-checking have been designed for spacecraft applications [Avizienis, et. al., 1971] , telephone switching systems [Chang, et. al., 1973], and even general business use [IBM, 1973]. In this paper we will give principles for the design of self-checking processors using commercially available integrated circuits (IC's). Many processors also have built-in diagnostic aids in the form of additional hardware and microprograms. While our ultimate goal is to design self-diagnosing processors, in the paper we will focus on the prelude to diagnosis: detection.

Fault detection will be achieved through the use of error-detecting codes and self-checking circuits. Self-checking circuits [Carter and Schneider, 1968; Anderson and Metze, 1973; Wakerly, 1974] have properties of self-testing and fault-secureness. Fault-secureness for a set of faults insures that no fault in the set can produce an undetected error, while self-testing insures that all faults from a set are eventually detected in normal operation. The design of a system using self-checking circuits is guided by the principle that no likely fault should produce an undetected error and all likely faults should be detected in normal operation. The definition of "likely fault" depends on the system implementation. In our case, we consider likely an arbitrary fault that affects a single integrated circuit package. Our goal is to design a processor in which no such fault can cause an undetected error.

The principles given in this paper apply to microprogrammed processors with the general structure given in Fig. 1. The registers and arithmetic/logic unit (ALU) of the processor are contained in the block labeled "data paths." The data paths may transmit data to or receive data from an I/O controller, which is not shown. The independent control points [Flynn, 1975] of the data paths are connected to the outputs of the microprogram memory. The address for the microprogram memory is provided by the microprogram sequencer. At each clock period, the sequencer provides a new microprogram address as a function of the current address, sequencing information from the microprogram memory, and conditions from the data paths. The current technology allows the microprogram sequencer to be implemented as a single integrated circuit [Wakerly, et. al., 1975], while the microprogram memory and the data paths are "sliced" into a number of circuits operating in parallel. For example, a 1024-word by 40-bit microprogram memory could use ten 1024-word by 4-bit read-only memories (ROM's), while a data path with 32-bit registers and ALU operations could be implemented with eight 4-bit data path slices [AMD, 1975] or sixteen 2-bit data path slices [Hoff, et. al., 1975].

The processor architecture of Fig. 1 is made self-checking as shown in Fig. 2. Data in the data path block is encoded in an error-detecting code, so that one extra slice is required to process check symbols. Likewise the microprogram memory output is encoded, and an extra ROM package is required

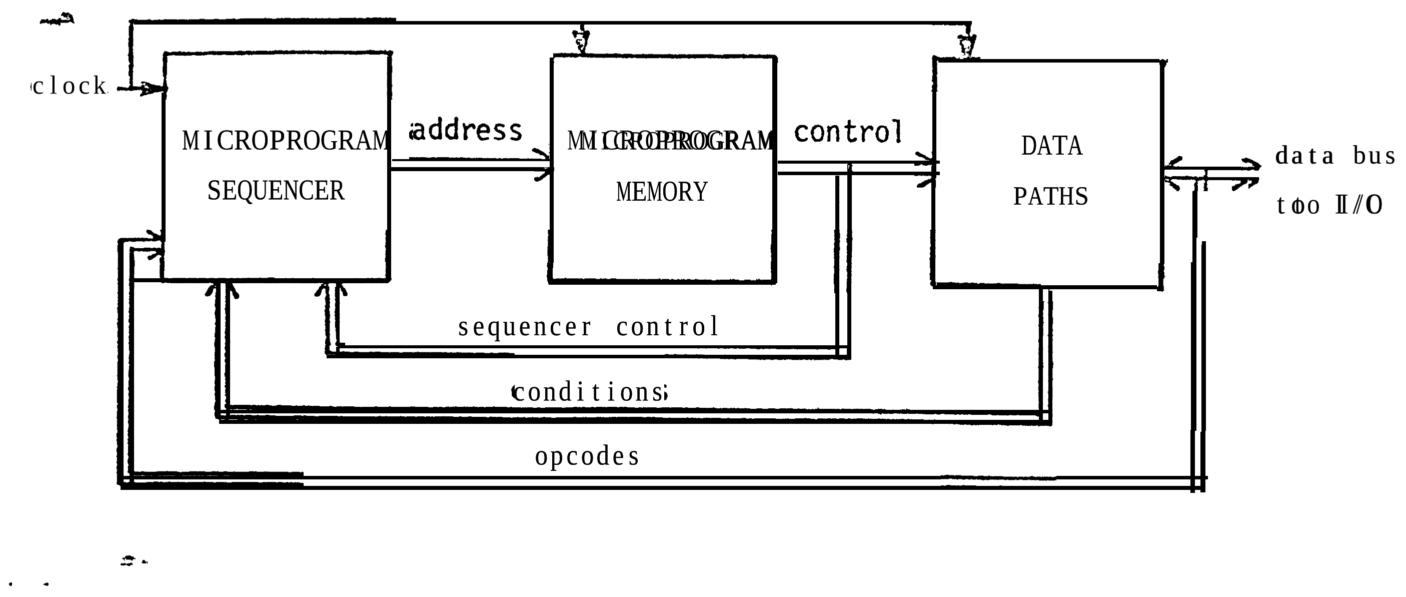


Fig. 1 Microprogrammed processor.

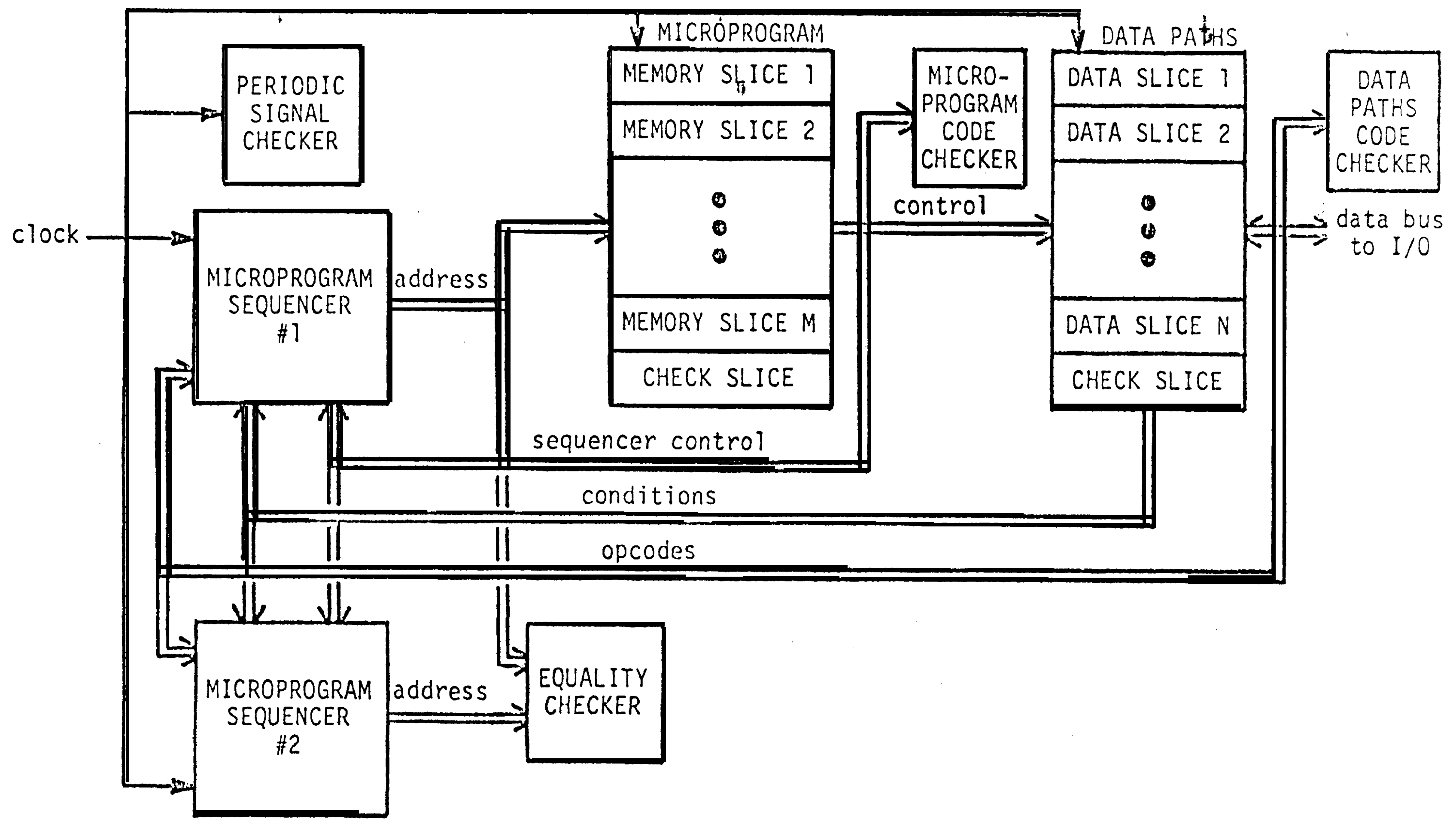


Fig. 2 Microprogrammed processor checking

to store check symbols. The microprogram sequencer is duplicated. Checkers monitor the microprogram and data path outputs for noncode words and the duplicated sequencer outputs for a mismatch. Totally self-checking checkers for these outputs can be designed in a straightforward manner [Anderson, 1971; Wakerly, 1973]. A periodic signal checker [Usas, 1975] monitors the clock. Data transactions with the I/O controller use the data path code. I/O controller checking is discussed in detail in [Usas, 1976]. In the remainder of this paper we discuss the design of the blocks of Fig. 2 in more detail and give a design example. Section 2 discusses the choice of error-detecting codes and Section 3 discusses the placement of checkers in the data paths. In Sections 4 and 5 we describe means of checking arithmetic and logic operations, respectively. Section 6 discusses micro-program memory and decoder checking, and Section 7 discusses the sequencer and clock. Next, Section 8 gives an overview of a self-checking processor based on these principles that is currently being designed and built. Finally, Sections 9 and 10 make a few observations about fault diagnosis and repair of such a processor.

2. CHOICE OF ERROR-DETECTING CODES

Errors in data transmission and storage can be detected if data is encoded in an error-detecting code. Codes used for this application are almost always systematic. Codes are chosen on the basis of the cost of their implementation and their effectiveness in detecting errors. A code's implementation cost can be measured by the number of redundant bits in the code; however, the cost of self-checking checkers for the code should also be considered. A code's effectiveness can be measured by its minimum distance; but additional insight can be obtained by determining the size of the tested and secure fault sets [Wakerly, 1973) of self-checking circuits using the code. A code should be chosen so that the likely physical failures are contained in the tested and secure fault sets of the circuits using the codes. If a circuit is bit(byte)-sliced, then a single bit (byte) error-detecting code should be used.

The circuits used in data paths and memory are generally bit(byte)-sliced circuits with no interaction between bits (bytes), except for data selection and addressing, which are checked separately (see Section 6), and arithmetic operations, which are considered in Section 4. If the data is encoded in an error-detecting code and if the circuit performs only data storage, transmission, or routing, then the circuit is self-checking, since the output *is* a code word if no failures have occurred. These circuits are called non-transforming circuits [Wakerly, 1976a]. Examples of such circuits are registers, multiplexers and demultiplexers. The secure and tested fault sets of such circuits depend on the error-detecting code used, and we can use the size of these sets to compare the effectiveness of several important systematic codes. Any bit(byte)-sliced non-transforming circuit can be used to make the comparison if we assume that the normal input set of the circuit tests all fault that affect a single bit(byte)-slice.

We compare five classes of codes for possible application in self-checking processors. The first two codes are well known - the single bit even parity check and duplication. The next code, the distance-2 b-adjacent code, consists of b interleaved single bit parity checks [Bossen, 1970; Peterson and Weldon, 1972]. The $(k,2^b)$ checksum code has k b-bit data bytes and a check byte which is the modulo 2^b sum of the data bytes [Garner, 1958; Wakerly, 1975]. The last code is an arithmetic code, a low-cost residue code with check base $A = 2^b-1$.

The characteristics, advantages, and disadvantages of each code are discussed in detail in [Wakerly, 1976a]. The discussion is summaried in Table 1. The example processor of Section 8 uses a low-cost residue code with check base 2^4-1 for the data paths, a distance-2 4-adjacent code for the microprogram, and duplication for the microprogram sequencer.

Table 1 Properties of important codes.

Code	Single-bit even-parity
Data part	n bits
Check symbol	1 bit
Fault-secure	Single bit slice faults
Self-testing	Faults affecting fewer than all bits
Checker	n-input Exclusive OR tree and 1 inverter
Comments	Least redundancy, cheapest checker

Code	Duplication
Data part	n bits
Check symbol	n bits
Fault-secure	Faults affecting different bits in the two duplicates
Self-testing	Faults affecting different bits in the two duplicates
Checker	n-bit totally self-checking equality checker
Comments	Most redundancy, expensive checker, not self-testing for many double faults

Code	Distance-2 b-adjacent
Data part	k b-bit bytes
Check symbol	1 b-bit byte
Fault-secure	Single byte slice faults
*Self-testing	Faults affecting fewer than all bytes
Checker	b k-input Exclusive OR trees and b-bit totally self-checking equality checker
Comments	Cheapest, fastest checker of all codes with b check bits, not self-testing for many k+l-bit faults

Code	$(k,2^b)$ checksum
Data part	k b-bit bytes
Check symbol	1 b-bit byte
Fault-secure	Single byte slice faults
Self-testing	Faults affecting fewer than all bits
Checker	k-byte tree of b-bit modulo 2^b adders and totally self-checking equality checker
Comments	Cheaper than b-adjacent for parity-predicted arithmetic, slower and core expensive checker

Code	Low-cost residue, check base $A = 2^b-1$
Data part	k b-bit bytes
Check symbol	1 b-bit byte
Fault-secure	Single byte slice faults except all stuck-at-0 or all stuck-at-1
Self-testing	Faults affecting fewer than all bits
Checker	k-byte tree of b-bit modulo 2^b-1 adders and totally self-checking equality checker
Comments	Direct implementation of arithmetic operations, not fault-secure for some single byte slice faults, slightly slower checker than checksum codes.

3. CHECKER PLACEMENT

Sellars et. al. [1968] give criteria for the placement of checkers in coded data paths:

1. The checkers should minimize undetected errors and faults.
2. The checkers should locate the error before it corrupts further data (so that the error indications are meaningful for maintenance, and so that the erroneous data can be reconstructed).
3. The checkers should locate the error to as small an amount of hardware as necessary to ensure easy servicing.
4. The checker cost should be minimized (usually this means using the minimum number of checkers that will do the job).

There are obviously trade-offs between satisfying the first three criteria and satisfying the last (examples are given in [Sellars, et. al., 1968]). Rather than dwell on these, we would like to give formal rules for checker placements that guarantee to detect all detectable errors (criterion 1 above). The rules apply to networks of functional blocks, such as the one shown in Fig. 3. In addition to control inputs, each block has data inputs and outputsin some error-detecting code S. A <u>detectable error</u> occurs when a noncode word is produced at the output of a block because of a fault. We would like to guarantee that every detectable error is eventually either corrected or propagated to a checker. In order to do so, we must first characterize the behavior of functional blocks in response to noncode inputs. We do this with the "detection lossless" property defined below. This property is similar to the code disjoint property of self-checking checkers [Anderson and Metze, 1973].

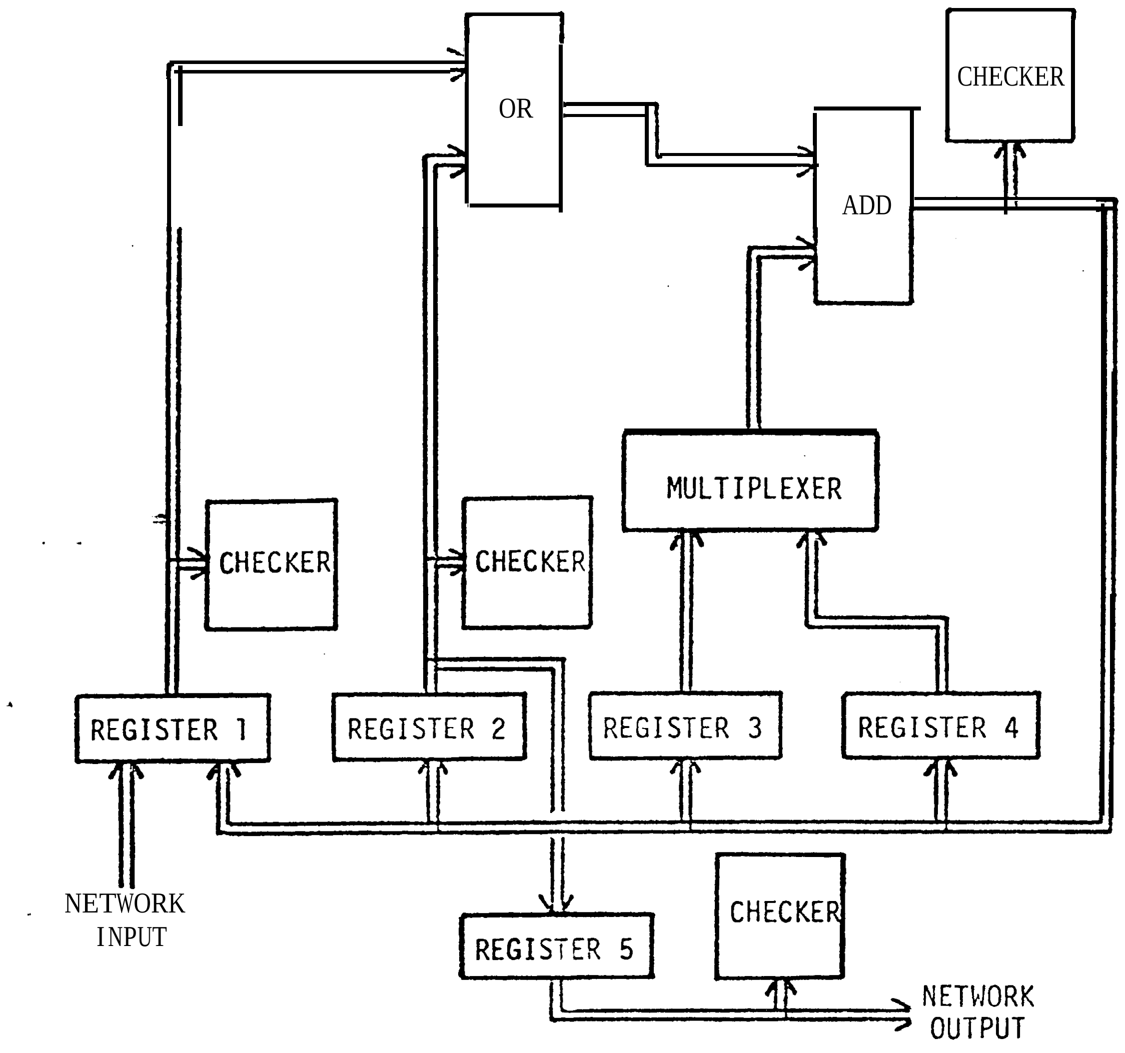


Fig. 3 A network of functional blocks and checkers.

Definition: Let a functional circuit have a control input vector C, data input vectors $X_1,\ldots,X_m$, and a single output vector Z. The circuit is detection lossless for a data code S and a set R of control inputs if

1) $C \in R$ and $X_1,\ldots,X_m \in S$ implies $Z \in S$; and
2) $C \in R$ and exactly one $X_i \notin S$ implies $Z \notin S$ or the X_i input has no effect on Z for this choice of C.

It should be apparent from the definition above that if a circuit is detection lossless, then a detectable error in one of its input vectors produces either the correct output or a detectable error. An an example, consider an adder for code words in an arithmetic code. If one of the input operands (summands) is a code word and the other is not, then the sum will be a noncode word and the presence of the erroneous summand is detected. An another example, consider a multiplexer for two coded operands. If one input is a code word and the other is not, the output will be either correct or a detectable error, depending on which input operand is selected. As an example of a circuit that is not detection lossless, consider a circuit for performing the logical OR operation by duplication, as shown in Fig. 4. The check symbols (A_c,B_c) of the *two* input operands (A_d,B_d) are discarded, and the logical OR is computed by two independent circuits. The check symbol (T_c) of the result is computed from the output of one of the OR circuits, while the output of the other becomes the data part (T_d) of the result. The duplicated circuit is self-checking because failures in either duplicated OR circuit produce a mismatch between the data part of the result and its check symbol. (The tested and secure fault sets depend on the code and the circuit implementation of OR.) However, the circuit is not detection

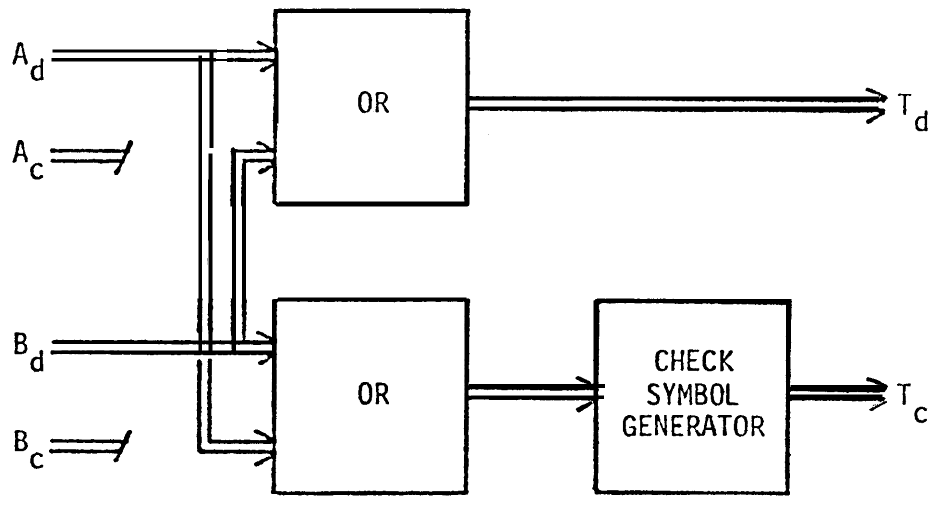


Fig. 4 Totally self-checking OR circuit.

lossless because the check symbols of the inputs are thrown away. A noncodc input operand is never detected because both duplicate OR circuits compute the same erroneous result.

We can now state general requirements for the placement of checkers that guarantee the eventual detection of all detectable errors produced in a single block of a network:

1. A checker should be placed at each network output.
2. There should be at least one checker in every loop in-the data paths.
3. There should be checkers at all the inputs of blocks that are not detection lossless.
4. If a data path fanout reconverges at the inputs of a detection lossless block, then there should be a checker in at least all but one of the paths from the fanout point to the inputs of the block.

Application of these rules to the network of Fig. 3 results in the indicated checker placement (all blocks except OR are assumed to be detection lossless). Requirement 1 above ensures that all data is checked before it leaves the system, Requirement 2 prevents repeated-use multiple errors, and Requirement 3 prevents undetected errors at the outputs of blocks that are not detection lossless. Requirement 4 is necessary so that a single error at the output of a block that fans out does not become an undetected multiple error. This *requirement, although sufficient, is not necessary for the detection of single errors that fan out. For example, consider the network shown in Fig. 5 and assume that data are encoded in arithmetic code. A weight-one arithmetic error at the output of the multiplexer has an error value of $\pm 2^i$ for some i. Because of fanout, both R1 and R2 *may* also receive errors of $\pm 2^i$, and produce a double error at the adder input. However, the sum will have an error value of $\pm 2^{i+1}$, which is also a detectable error value. Hence the network will detect all single errors even though Requirement 4 is not satisfied. On the other hand, consider the network of Fig. 6 and assume data are coded in a low-cost code with check base A=3. An error value of $\pm 2^i$ at R1's output can produce an error value of $\pm 2^i \pm 2^{i+1} = \pm 3 \cdot 2^i$ at the adder output, which is undetectable. In this network Requirement 4 is obviously necessary.

The data paths of the example processor of Section 8 are designed so that a single checker is sufficient for detecting all data path errors.

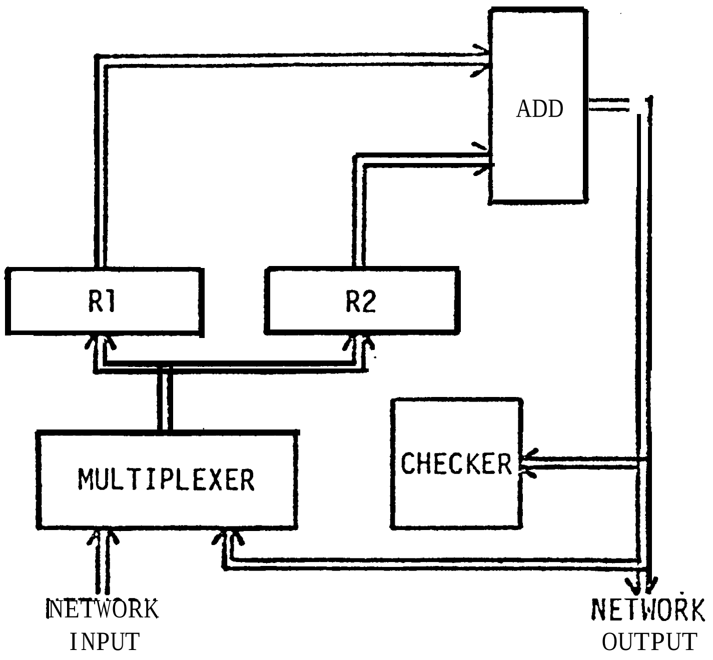


Fig. 5 A network for which Requirement 4 is not necessary.

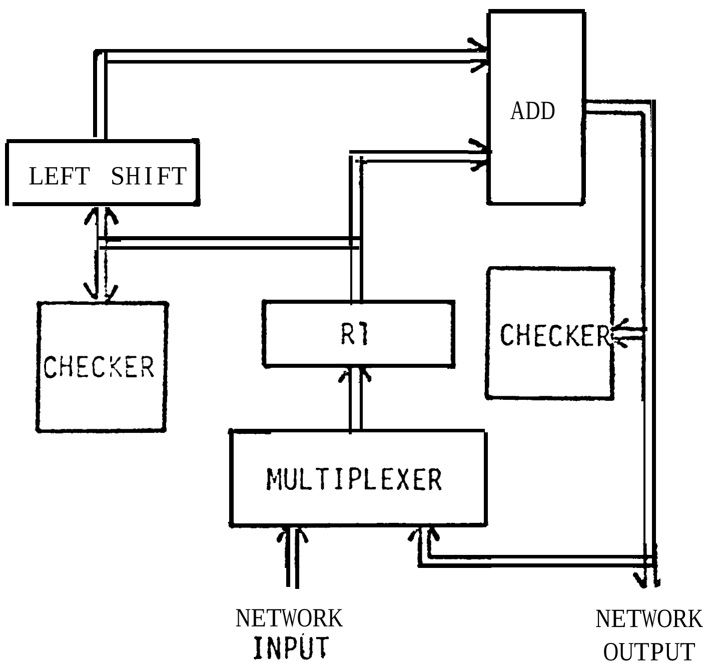


Fig. 6 A network for which Requirement 4 is necessary.

4. ARITHMETIC OPERATIONS

The basic arithmetic operations in a computer are addition, sub-
-a
traction and shifting. Operations such as multiplication and division may be accomplished in software or firmware by iterative use of the basic operations. In this section we discuss methods of detecting errors in the basic arithmetic operations; errors in the iterative operations are manifested either as errors in the basic operations or as control errors detectable by the techniques of Sections 6 and 7.

Both arithmetic and non-arithmetic codes can be used to detect errors in arithmetic operations. The non-arithmetic codes such as parity and check&m codes can be used with check symbol prediction techniques to preserve the code under arithmetic operations [Sellars, et. al., 1968; Wakerly, 1975]. These techniques predict the check symbol of the sum of two operands as a function of the check symbols of the two operands and carries generated during the addition. The main advantage of check symbol prediction techniques is that they can be used with codes such as b-adjacent and checksum codes that guarantet to detect any error in a single b-bit byte. However, parity prediction techniques require duplication of the adder carry. generation circuitry to avoid undetectable failures.

Arithmetic codes such as low-cost residue codes [Avizienis, 1971] are better suited for performing arithmetic operations. A code word in a low-cost residue code has a b-bit check symbol which is the modulo 2^b-1 residue of the data part. The main disadvantage of low-cost residue codes is that they allow some undetected b-bit errors, namely all l's in a b-bit byte changed to all. O's or vice versa. However, they allow direct inplementation'of addition, subtraction, and rotation operations in the one's complement system using standard MSI and LSI IC's. Two's complement operations and logical and arithmetic shifts require the check symbol of a result to be modified as a function of carries into and out of the low order and high order bits of the data part operation. The circuitry for performing this fix-up does not destroy or degrade the self-checking properties of a two's complement ALU, but it does potentially slow down the ALU's operation. The fix-up operations necessary for various add, subtract, and shift operations are described in detail in [Wakerly, 1976a]. The fix-ups for one's complement operations are given in [Monteiro and Rao, 1972).

The example design of Section 8 uses a modulo 2^4-1 low-cost residue code for the data paths. Arithmetic is performed in the two's complement system, and check symbol fix-up circuitry is provided at the output of the ALU.

5. LOGICAL OPERATIONS

It is well known that no code short of duplication is preserved by logical operations such as AND and OR [Peterson and Rabin, 1959]. Hence all schemes for self-checking logical operations must use a form of duplication. The STAR computer uses duplicate logic units [Avizienis, et. al., 1971], and a processor designed by Monteiro and Rao [1972] duplicates the AND operation and performs the other logical operations using a combination of AND and checked arithmetic operations.

While no code short of duplication detects errors in logical operations, it is possible to perform logical operations in *a* self-testing manner without duplication. The concept of partially self-checking circuits [Wakerly, 1973] allows the non-code-preserving logical operations to be performed by circuits that are tested in normal operation by other, code-preserving, operations (such as addition and subtraction using a residue code).

Simple duplication (such as shown in Fig. 4) and partially self-checking circuits for logical operations have a common characteristic: they are not detection lossless since the input check symbols are not used in the computation of the result. Thus a system using such a method will require checkers at the inputs of the logic unit. In the remainder of this section we describe the design of logic units that avoid this problem: totally self-checking detection lossless logic units. A logic unit for residue-coded operands that was detection lossless for OR and EXCLUSIVE OR (XOR) operations but not for AND was designed by Montciro and Rao [1972]. The designs reported here are extensions of their design.

Consider any systematic code, where the check symbol of a data part A can be written as a function F(A). The specifications for a detection lossless AND circuit can be given as follows:

a) Inputs: A_d, B_d (data parts)

A_c, B_c (check symbols)

b) outputs: T_d (data part)

T_c (check symbol)

c) $T_d = A_d \wedge B_d$

d) $A_c = F(A_d)$ and $B_c = F(B_d)$ implies $T_c = F(T_d)$.

e) $A_c \neq F(A_d)$ or $B_c \neq F(B_d)$ (but not both) implies $T_c \neq F(T_d)$.

Specifications for other logic functions are obtained by simply changing item (c) above. These specifications can be met for linear block codes, residue codes, and checksum codes as described below.

For example, consider the linear block codes. Since

$$A_d \wedge B_d = Ad \oplus B_d \oplus (A_d \vee B_d) \quad ,$$

we may write

$$T = F[A_d \oplus B_d \oplus (A_d \vee B_d)] \quad .$$

But due to the linearly of F with respect to $\oplus$, this nay also be written as

$$T_c = F(A_d) \oplus F(B_d) \oplus F(A_d \vee B_d)$$

$$= A_c \oplus B_c \oplus F(A_d \vee B_d)$$

Thus a detection lossless AND circuit can be designed as shown in Fig. 7. In view of the above equation for Tc , it is quite apparent that the design satisfies specifications (a) through (d). It also satisfies specification (e) due to the use of A_c and B_c in computing Tc . The circuit is totally self-checking, since it is obviously fault secure and self-testing for all non-redundant faults in F and $\oplus$, and for a class of faults in the AND and OR circuits that depends on the code (exactly the clases given in Table 1).

A totally self-checking single-error detection lossless OR circuit can be designed using the relation,

$$A_d \vee B_d = A_d \oplus B_d \oplus (A_d \wedge B_d) \quad .$$

Thus the OR circuit is obtained simply by swapping the AND and OR blocks in Fig. 7.

Detection lossless logic circuits for operands in a residue code with check base A can be designed using relations such as

$$Ad \wedge B_d = A_d \text{ plus } B_d \text{ minus } (A_d \vee B_d) \quad ,$$

which implies

$$T_c = A_c +_A B_c -_A F(A_d \vee B_d).$$

Defining "$\dotplus$" as the componentwise $+_{2^b}$ addition operation for $(n, 2^b)$

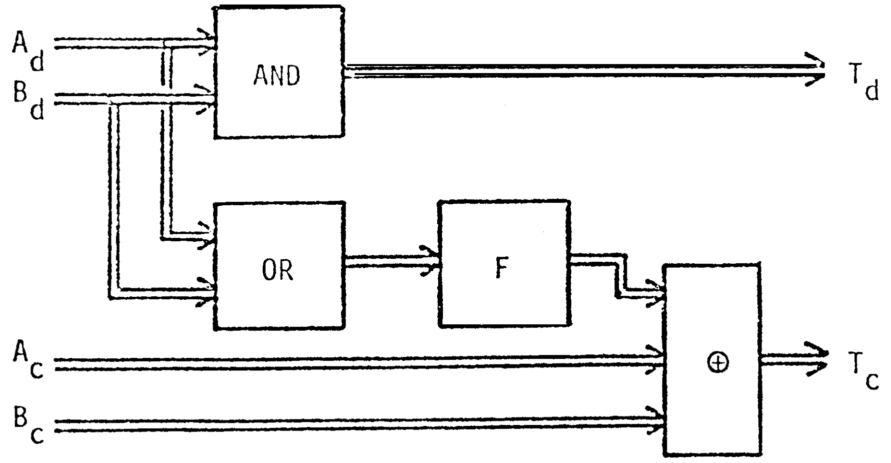


Fig. 7 Totally self-checking detection lossless AND circuit.

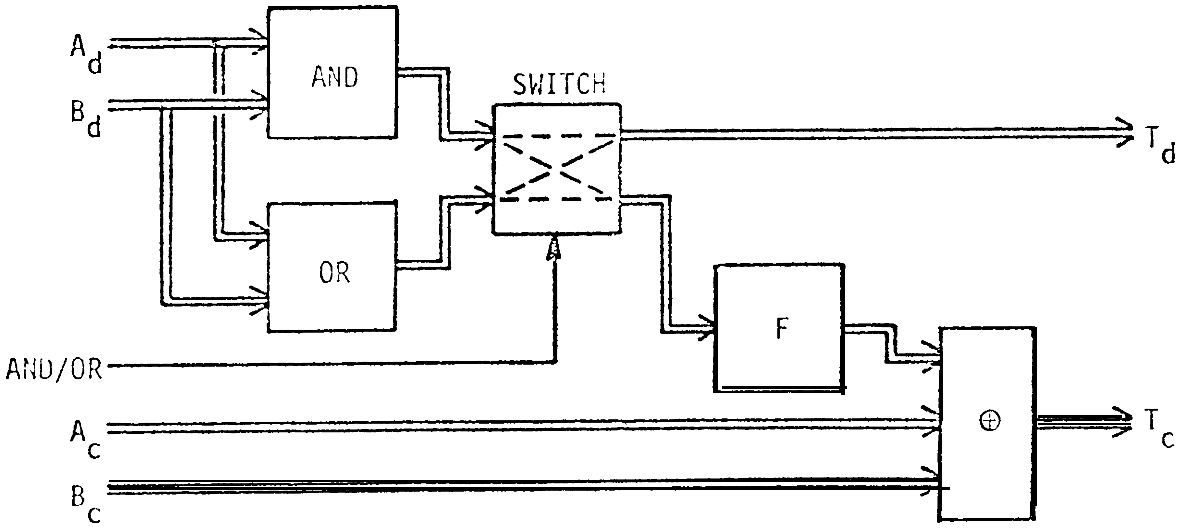


Fig. 8 Totally self-checking detection lossless AND/OR circuit.

checksum code vectors and "$-$" as subtraction, we can write

$$A_d \wedge B_d = A_d + B_d - (A_d \vee B_d) ,$$

so that for a checksum code AND circuit,

$$T_c = A_c +_{2^b} B_c -_{2^b} F(A_d \vee B_d) .$$

Since linear block codes are preserved by the XOR operation, a detection lossless XOR can be implemented directly. For residue and checksum codes, XOR can be implemented by making use of the identity

$$A_d \oplus B_d = A_d \text{ plus } B_d \text{ minus } 2(A_d \wedge B_d)$$

If both the AND and OR operations are required in a processor, then it[3] is possible to design a detection lossless totally self-checking logic unit in which neither operation is duplicated. As shown in Fig. 8 for linear block codes, a switch is used to swap the AND and OR blocks depending on which operation is desired. Of course, the switch may be more costly than an approach that uses two separate logic units. At worst the switch consists of two conventional 2-input multiplexers for each data bit. At best the switch could use a pair of CMOS transmission gates per bit [Hnatek, 1973] or some scheme involving multiple output I^2L gates and three-state logic [Horton et. al., 1975]. Another alternative is to use a conventional logic unit that has circuitry to select either the AND or OR operation, and to perform the checked computation in two steps. This is the approach used in the design example of Section 8, The two-step approach works only if there are no faults that can produce an erroneous 1 in the AND operation and an erroneous 0 in the same bit position in the OR operation (or vice versa).

6. MICROPROGRAM MEMORY AND DECODERS

Detection of errors in microinstructions fetched from microprogram memory is generally accomplished by encoding each microinstruction in an error-detecting code [Cook et. al., 1973; Chang et. al., 1973a,b]. A checker then monitors the microprogram data register and non-code words indicate errors. While this scheme detects errors in fetching microinstructions, it does not guarantee that the control signals reach their destinations correctly. For example, suppose one bit of the microinstruction is used to gate bus A onto bus T, as shown in Fig. 9. Assume that data on the buses is encoded in an error-detecting code. A break in the "T $\leftarrow$ A" control line affects multiple bits of the output on the T bus and thus may not be detected by the data path error-detecting code. Errors caused by broken wires do not seem likely when one thinks in terms of #14 house wiring, but bad connections in plug-in cards and breaks in the wires between bonding pads and pins in integrated circuit packages are quite common. Therefore means of detecting errors caused by these failure modes must be provided.

A simple modification of the basic microinstruction coding scheme allows control line failures to be detected. Instead of checking that the microprogram data register is a code word, all of the control lines are routed to their destinations and then collected in a "sink register". The combination of the sink register and the check symbol in the microprogram

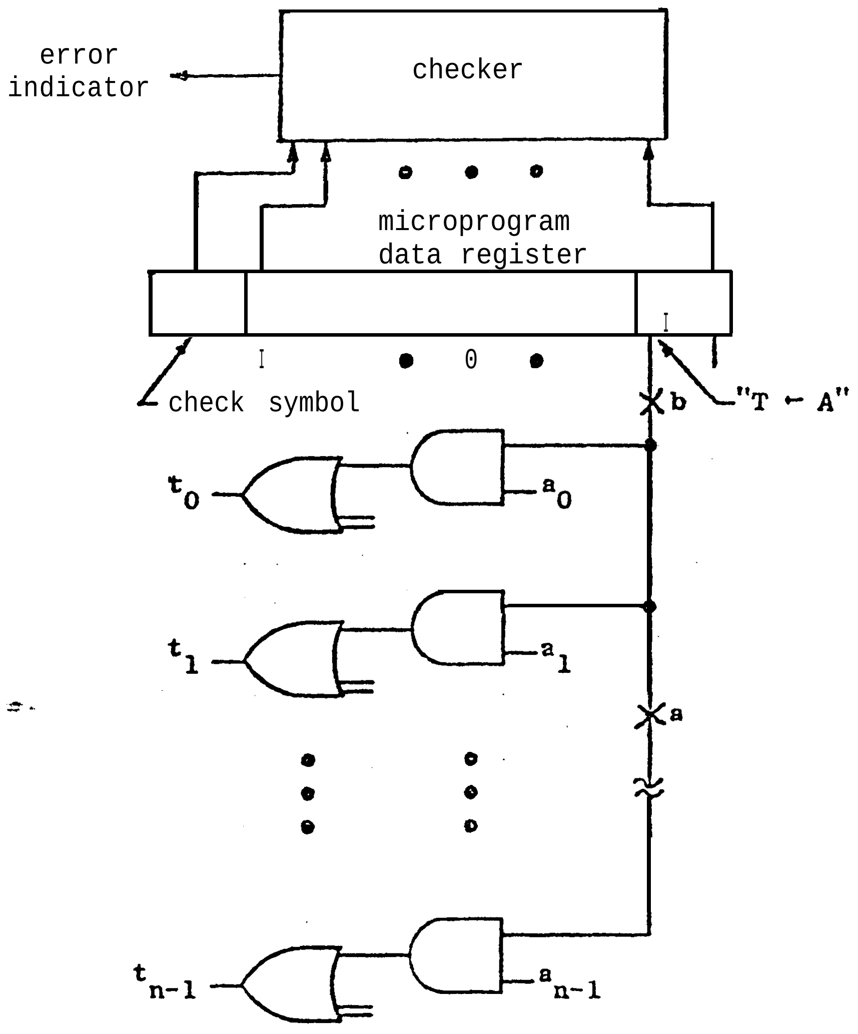


Fig. 9 Example of ineffective control line checking.

data register is then monitored by the checker, as shown in Fig. 10. With this scheme, a failure in the main control line is detected by the sink register checker, while a failure in a line to an individual bit (byte) slice is detected by the data path error-detecting code.

The previous example showed a single bit of the microinstruction used to enable the operation "$T \leftarrow A$." In a typical processor there are several possible sources for each bus, and several possible destinations. Only one source and generally only one destination is used for each bus during a single microinstruction. Therefore the length of the microinstructions can be shortened by consolidating the n one-bit fields such as "$T \leftarrow A$," "$T \leftarrow B$," etc. into a single field of length $\lceil \log_2 n \rceil$ "T source." The "T source" field is used as the input to a l-out-of-n decoder to activate the proper source during each microinstruction. Failures in a source decoder can cause no source or two or more sources to be gated to the same bus, and an erroneous code word on the bus may be produced. Thus some means must be provided for detecting decoder failures.

One method of detecting decoder failures is simply to monitor the decoder outputs with a checker. Self-testing and totally self-checking decoder checker designs for l-out-of-n and general k-out-of-n codes have been given by Toy [1971], Carter, et. al. [1971], Anderson and Metze [1973], and Reddy [1974].

Another method of detecting decoder failures is based on the use of the sink register. Decoder outputs can be re-encoded into their original form after being routed to their destinations, and then loaded into the sink register. If the re-encoded field in the sink register is different

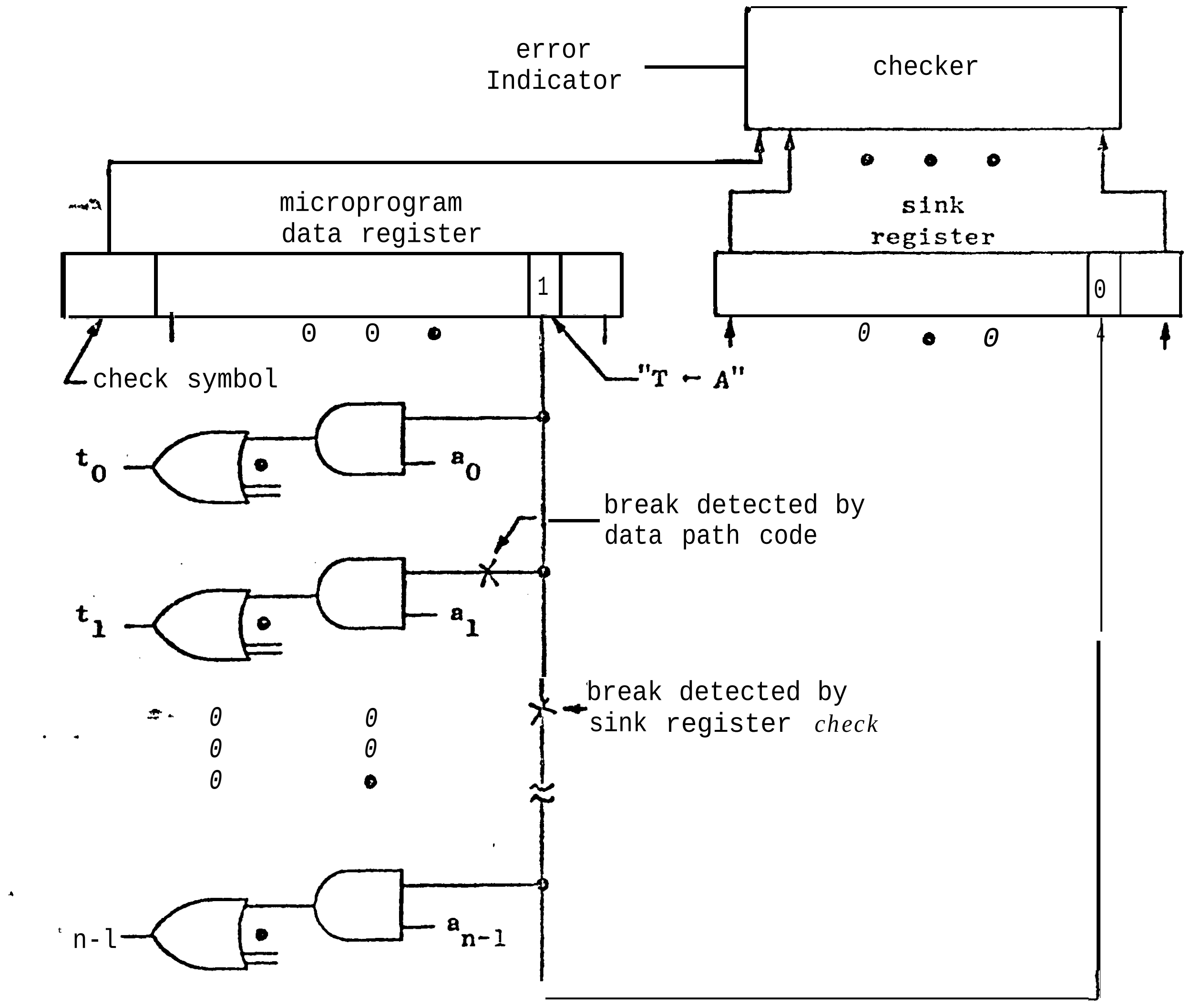

error Indicator
checker
microprogram data register
sink register
1
0
check symbol
"T ← A"
t_0
a_0
break detected by data path code
t_1
a_1
break detected by sink register check
n-1
a_{n-1}

Fig. 10 Control line checking using sink register.

from the original field in the microprogram data register, then the error in the decoder can be detected by the checker for the microprogram data. This assumes that the field is no wider than b bits, where b is the number of adjacent bit errors that can be detected by the microprogram data error-detecting code. An m-bit field in the microprogram can be decoded into a 1-out-of-(2^m-1) code and re-encoded into m bits in the, sink register so that the circuit is self-testing but not fault secure for all single stuck-at faults in the decoder and encoder. If the m-bit field is already encoded in a k-out-of-m code and decoded into a 1-out-of-$\binom{m}{k}$ code, then the circuit is both self-testing and fault-secure for single faults [Wakerly, 1976 a].

In commercially available LSI data path slices, there are already source and destination field decoders on each slice. A failure in one of these decoders affects only a single slice and so decoder failures do not have to be detected explicitly. The example processor of Section 8 is designed so that there are no decoders between the microprogram data register and the control inputs of the slices. However, the sink register concept is used, so that the microprogram data is checked after it has been routed to the slices.

7. MICROPROGRAM SEQUENCER AND CLOCK

A microprogram sequencer is a circuit which, along with ROM, comprises a microprogrammed control unit. As a minimum, a sequencer must contain a microprogram address register and logic to update the address register. It may also contain such features as a pushdown stack for micro-subroutines, condition code latches, and instruction register. The inputs of a se- quencer are: 1) a command indicating how to compute the next microprogram address (e.g., increment, subroutine return, conditional branch); 2) status bits for conditional branching; and 3) a microprogram address for reloading the address register for branches, subroutine jumps, instruction decoding, etc. The outputs of a sequencer are simply the microprogram address (which is connected directly to the ROM) and possibly a few control bits.

A number of "tricks" for checking microprogram sequencer operations without complete duplication have been described in the literature [Chang, et. al., 1973; Wakerly, 1973, 1976a]. For example, errors in a microprogram counter can be detected by an arithmetic code, and errors in the fetching of linked microinstrcctions can be detected by address parity bits. However many operations of a sequencer, such as intepreting conditional branches, decoding instructions, and setting and storing individual condition code bits, can be checked only by duplication [Wakerly, 1973, 1976a]. Since the present technology allows a complete microprogram sequencer to be implemented on a single LSI package, it is best to duplicate the entire microprogram sequencer.

The example processor in the next section uses a duplicated microprogram sequencer. Both copies of the sequencer receive identical inputs. The output of one of the sequencers is used to address the microprogram ROM, and the outputs of both sequencers are compared using a totally self-checking equality checker.

Both the microprogram sequencer and the data path slices of a processor operate from a common clock. Clock errors can be detected by a totally self-checking periodic signal checker as described by Usas [1975].

8. DESICN EXAMPLE -- SCAMP

A self-checking processor based on the foregoing principles is being designed and constructed at Stanford University. Dubbed SCAMP (Self-Checking And Maintainable Processor), the processor has a microprogrammed control unit and 16-bit data paths, registers, and two's complement arithmetic. Using standard speed TTL logic and main memory with 600 ns access time, the target machine will perform a 16-bit register-to-register add in about 1 us.

SC&? instructions are one or two words (16 or 32 bits); the second word of a double word instruction is an immediate operand or an address. The first 8 bits of any instruction is always an opcode that can be uniformly decoded as in the IBM 360/370. There are 16 general-purpose registers, one of which is a stack pointer. Register-to-register and register-to-memory instructions are available, and operations and formats are similar to those of an Interdata 7/16 [Interdata, 1971]. Two 4-bit fields in most instructions specify two general registers. For memory reference instructions a variety of addressing modes can be used for the memory operand, similar to those of a PDP-11 [DEC, 1973]. There are no input/output instructions; memory-mapped I/O is used as in the PDP-11 and the M6800 [Motorola, 1975].

The processor is partitioned along the lines of Fig. 1 into data paths, microprogram memory, and microprogram sequencer. The data paths are implemented using five identical 4-bit slices, four for the data part and one for the check symbol. Associated with the check slice is fix-up circuitry for arithmetic and logical operations. The microprogram memory is implemented with 1K word by 4-bit ROM's, and has one extra ROM for checking. The microprogram sequencer is duplicated. There are only four checkers in the system, monitoring the clock, the data path bus (D-bus), the microprogram memory, and the duplicate sequencers.

The organization of the 4-bit data path slice is shown in Fig. 11. Most of the control lines have been omitted. Each slice contains 16 general purpose registers, four scratchpad registers, an ALU, and a shifter. The RW and RX registers can be loaded from an external source; the contents of either one of these registers can be used to select one of the general registers. The selected general register may be used as the ALU A input and/or written into. The scratchpad is a two-port register file, so that one scratchpad may be used as the ALU A input and another can be used as the B input. The RW and RX registers can also be used as ALU B inputs. This is useful for operations in which RW and RX contain not register numbers but short operands. The shifted ALU output may be placed on the D-bus, However, the D-bus is a three-state bus, and the shifter output may be disabled so that external data may be placed on the bus. A general register or scratchpad can be loaded with either the D-bus or the K-bus, an auxiliary input bus.

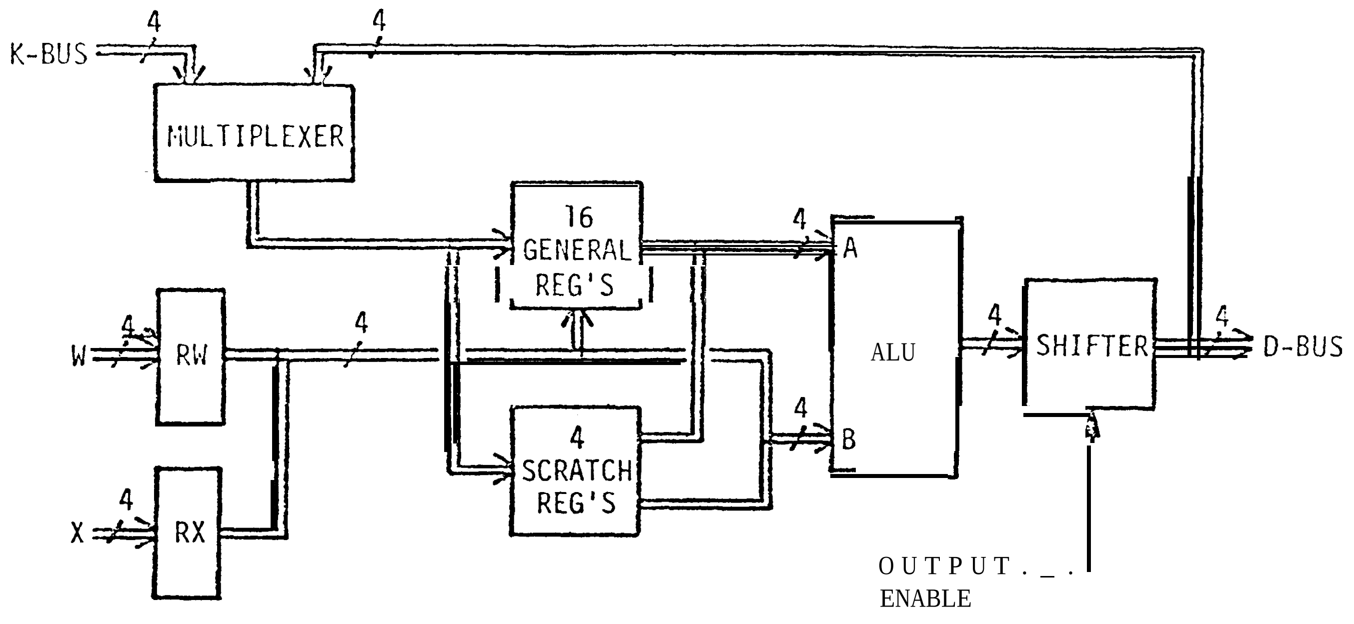


Fig. 11 4-bit slice organization.

The complete data paths are obtained using five 4-bit slices as shown in Fig. 12. There are four slices for the data parts of operands ($D_{15\text{-}0}$, D_0 least significant) and one slice for check symbols- With a few exceptions, identical control signals go to all of the slices. The W and X inputs of all the slices are connected to D_{7-4} and D_{3-0} respectively, since the register-select fields of instructions appear at these positions. In the data part, the K-bus is connected directly to the D-bus as a 4-bit right rotation. In the check slice, the K-bus is connected to the output of the check symbol fixup circuitry. For most operations that require no check symbol fixup, the registers of both the data part and the check slice are loaded from the D-bus. For an operation that requires a fixup, the R-bus is used to load the check symbol slice's register only. A 4-bit right rotation of the data part using the K-bus should leave the check symbol unchanged; therefore in the check slice the D-bus is selected for such operations.

The contents of the D-bus are latched into a check register at the end of each micro-cycle. Checking is overlapped with the next microcycle. The generated check symbol may be subtracted by the fixup unit from the output of the check slice for performing detection lossless logical operations in two steps as described in Section 5.

Every data transaction in the system makes use of the D-bus, and so a single checker on the D-bus is sufficient for all data checking. Microprogram constants are loaded into the data paths by disabling the slices' D-bus outputs and gating the appropriate microprogram field onto the D-bus. Opcodes are loaded into the sequencer by placing the instruction register

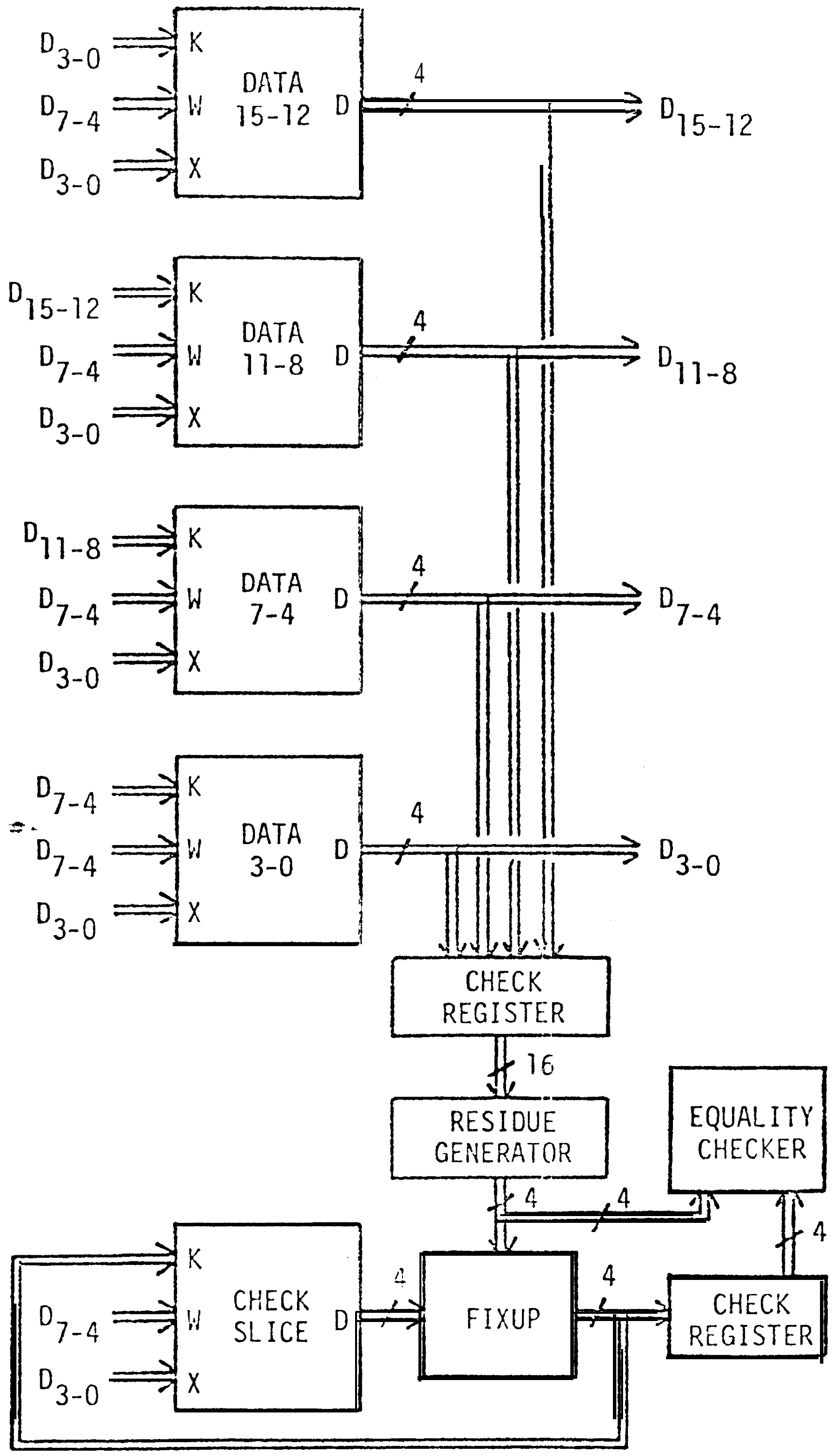


Fig. 12 SCAMP data path checking.

(one of the scratchpads) on the D-bus and enabling an 8-bit sequencer input bus that is connected to the high-order bits of the D-bus. The only limitation that arises from having only a single data path checker is in multiplication and division. Since these operations require essentially a double precision shift at each iteration, the iteration cannot be done in one microcycle. Some LSI data path slices have a "Q register" that makes possible one-microcycle iterations, but a Q register in a self-checking machine would require an additional checker. SCAMP requires two microcycles per iteration of the multiplication loop.

There are no I/O instructions in SCAMP; memory-mapped I/O similar to a PDP-11 or M6800 is used. The I/O controller consists of a bus address register (BAR), a bus data register (BDR), and a few control signals (MSYNC, SSYNC, R/W). The BAR can be loaded from the D-bus and its output is the I/O address bus. The BDR can be loaded from the D-bus and its output can drive the bidirectional I/O data bus. Alternatively, the I/O data bus can drive the D-bus.

In order to simplify I/O control and to facilitate checking with the existing checking mechanism, I/O is performed strictly under microprogram control. For a read operation, the microprogram loads an address into the BAR and sets the read/write (R/W) bit. Then it asserts MSYNC and proceeds with any operations that are to be overlapped with the read. Meanwhile the I/O device asserts SSYNC after placing its data on the I/O data bus. When the microprogram is ready for the input data it does a conditional branch on SSYNC. A loop counter in the sequencer (also used for multiply and divide loops) is used to force an exit if the device does not respond within a fixed time. Write operations proceed in a similar manner.

Although the present design allows detection of errors in loading the BDR and BAR and in the assertion of control'signals, it does not detect errors further down the I/O bus. The means of detecting I/O controller and device errors has not yet been determined, although the methods of Usas [1976] are applicable.

The preceding is meant to be only a brief overview of SCAMP's organiation. The detailed design of the processor will be discussed in another paper [Wakerly, 1976b]. This paper will examine the cost and performance penalties incurred by providing complete error detection.

9. DIAGNOSIS

Self-checking processors following the design principles of the previous sections have a number of totally self-checking checkers whose outputs can be combined to produce a signal that indicates the presence of any detectable error in the processor. As a minimum, this global error signal should be used for error-logging or perhaps to halt the processor. However, it would be more desirable for the processor to be self-diagnosing. Upon recognizing an error signal, the processor would initiate steps to determine the source of the failure. A trustworthy diagnostic would either give the processor a clean bill of health (the failure was a transient) or it would isolate the failure to a replaceable package for manual repair. There are three important diagnostic areas for processors following our design principles: data paths, microprogram memory, and microprogram sequencer. We discuss each of these briefly.

If a data path failure is detected, then we can use the microprogram memory and sequencer as tools for diagnosis. Microdiagnostics are used in non-self-checking commercial machines and they have been studied in the literature [Johnson, 1971; Guffin, 1971; Ramamoorthy and Chang, 1972). In a non-self-checking machine, a microdiagnostic must provide a stimulus for each fault, observe the output, and compare the output with the expected output. In a self-checking machine, microdiagnostics are much easier. Because of the built-in checkers and the self-testing nature of the circuits, the micro-diagnostics need only provide a sufficient set of stimuli and monitor the checker output. The usual problems of interdependence among the units being tested and the units being used to make comparisons and decisions are eliminated.

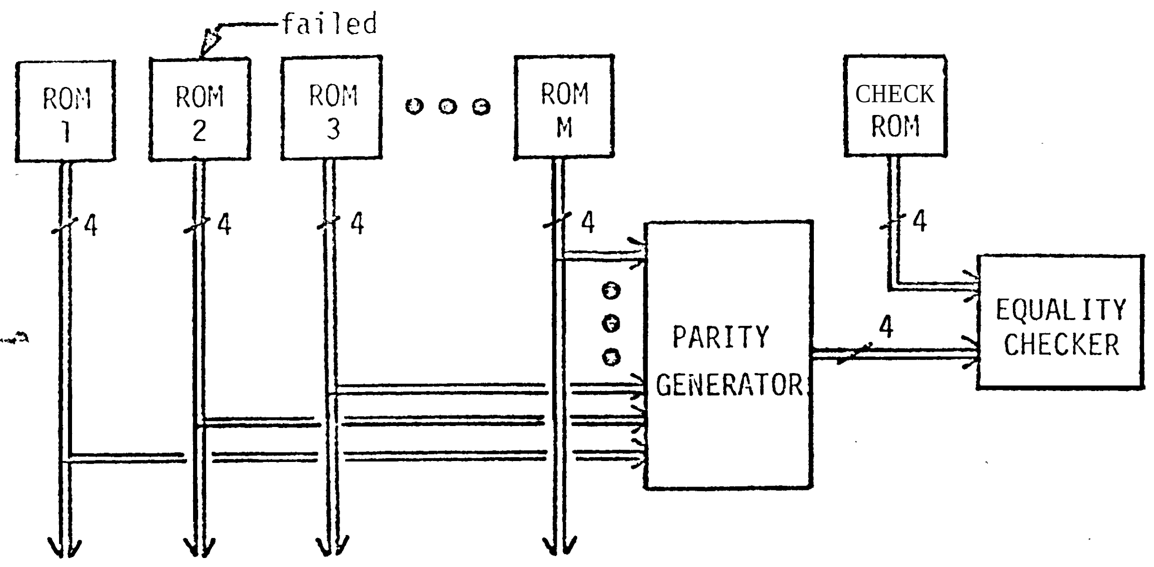
A failure in microprogram memory is detected by the memory's error-detecting code. Once the failure is detected, it should be isolated to a single ROM or microprogram data register package. Unfortunately a simple distance-two code gives no diagnostic information, and due to the read-only nature of ROM it is not possible to apply selective-patterns that sensitize one ROM package at a time. A distance-3 or greater code is needed to isolate ROM failures. Rather than use a wider ROM word (more check bits), it is possible to store a checksum one word of RON, effectively encoding the entire ROM in a distance-4 code. The checksum word should be a b-bit bytewise check over the ROM words, where b is the word length of the individual RON packages. En this way, a single package failure will result in an easily identifiable error in the corresponding byte of the checksum. However, note that the code must be chosen carefully to detect the failures that cause errors in all of the words of a ROY package. For example, a simple vertical parity check will. not detect a ROM output bit stuck-at zero.

Although the microprogram sequencer is duplicated, representing the highest level redundancy in a self-checking processor, it is nonetheless difficult to diagnose. If a sequencer failure occurs, then we observe a mismatch between the outputs of the duplicated sequencers, but it is impossible without additional information to determine which is the incorrect one. One possible approach to sequencer diagnosis is to write a microprogram that tests a single sequencer and gives an indication of the sequencer's correctness independent of the checking hardware. For example, the program could be written so that it exercised all of the sequencer's features and halted at a certain location only if all of the steps had been executed properly. Diagnosis would consist of running this test on both sequencers and tossing out the one that fails the test. We note that the complexity of sequencer testing is comparable to that of microprocessors.

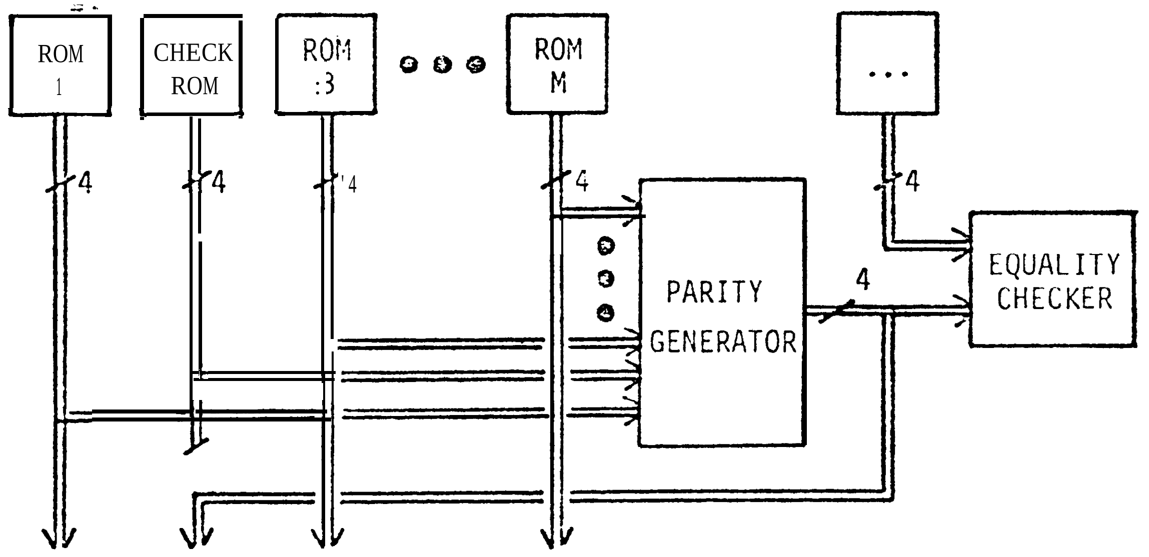
Since all errors in a self-checking machine are detected with self-checking checkers, there is always a chance that an error signal indicates a failure in the checker. The diagnostic technique must take this into account. For example, in SCAMP it is possible for the microprogram to place constants on the D-bus that exercise the data path checker without using the data path slices. Another example is in sequencer testing -- if both sequencers pass the test then it can be assumed that the sequencer equality checker is bad.

10. REPAIR

It is interesting to note that a self-checking processor with the structure of Fig. 2 has in it almost all of the hardware needed to continue operation in a unchecked mode after a failure. If it is diagnosed that a failure affects only a data path or microprogram memory check slice or a checker, then obviously it is possible to ignore the checker outputs and continue correct operation. If a failure affects a data path slice that is processing data parts, then the failed slice can be removed and replaced with the check slice since it is identical. If there is a failure in one of the duplicate microprogram sequencers, then the non-failed sequencer must be made the one whose outputs are connected to the ROM. The only failure that does not allow a direct swap is a ROM package failure, since the contents of the check ROM are obviously different from the contents of the other ROM's. However, if we know which ROM package is faulty, then it can he removed and its contents can be computed as a function of the other ROM contents and the check ROM. This computation can be carried out by the existing checking circuitry, as shown in Fig. 13 for a system that uses 4-bit ROM packages and a 4-adjacent code. In practice, this example would require that the RON's be mounted in sockets and some small jumpers would be needed for the change.



(a) Before repair.



(b) After repair.

Fig. 13 ROM failures.

11. CONCLUSION

We have described checking methods for each of the major elements of a minicomputer-like processor. These methods can guarantee concurrent detection of all single integrated circuit failures. The redundancy required for checking consists mainly of checkers, a duplicate microprogram sequencer, one extra microprogram memory slice, one extra data path slice and fixup circuitry (the complexity of the fixup circuitry is much less than that of a data path slice). Additional costs are one or two microprogram bits that must be provided separately for the data path slices and the-check slice, a few microprogram bits to control the checkers and fixup circuitry, and a few extra microprogram words for certain checking operations. For most operations no performance penalty is incurred by checking, since the check slice is just as fast as the data path slices and operates in parallel, and checking can be overlapped with execution. However, a penalty is incurred in operations that require extra steps for checking (such as two-step logical operations) and in arithmetic operations that require a fisup (in SCAMP the fixup takes less than 1/3 microcycle). When the design of SCAMP is completed, we will be able to discuss these factors in detail, but it appears that the total redundancy for checking will be somewhat less than 50% and the performance degradation will be minimal (about 10%).

References

AMD, 1975. Am2900 Bipolar-Microprocessor Family, Advanced Micro Devices, Sunnyvale, California.

Anderson, D. A., 1971. "Design of self-checking digital networks using coding techniques," Tech. Rep. R-527, Univ. of Illinois Coordinated Science Laboratory, Urbana, Illinois.

Anderson, D. A., and G. Metze, 1973. "Design of totally self-checking check circuits for _m_-out-of-_n_ codes," IEEE Trans. Comput. C-22(3): 263-269.

Avizienis, A., 1971. "Arithmetic codes: Cost and effectiveness studies for application in digital systems design," IEEE Trans. Comput. C-20(11): 1322-1331.

Avizienis, A., G. C. Giley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, 1971. "The STAR computer: An investigation of the theory and practice of fault-tolerant computer design," IEEE Trans. Comput. C-20(11): 1312-1321.

Bossen, D. C., 1970. "_b_-Adjacent error correction,' IBM J. Res. Develop. 14(7): 402-408.

Carter, W. C., and P. R. Schneider, 1968. "Design of dynamically checked computers," IFIP Conf. Proc. 2: 878-883, Amsterdam: North-Holland Publishing Company.

Carter, W. C., K. A. Duke, and D. C. Jessep, 1971. "A simple self-testing decoder checking circuit," IEEE Trans. Comput. C-20(11): 1413-1414.

Chang, H. Y., R. C. Dorr, and D. J. Senese, 1973. "The design of a microprogrammed self-checking processor of an electronic switching system," IEEE Trans. Comput. C-22(5): 489-500.

Cook, R. W., W. H. Sisson, T. F. Storey, and W. N. Toy, 1973. "Design of a self-checking microprogram control." IEEE Trans. Comput. C-22(3): 255-262.

Flynn, M. J., 1975. "Microprogramming - Another look at internal computer control," Proc. IEEE 63(11): 1554-1567.

Garner, H. L., 1958. "Generalized parity checking," IRE Trans. Electron. Comput. EC-7(9): 207-213.

Guffin, R. M., 1971. "Microdiagnostics for the Standard Computer MLP-900 processor," IEEE Trans. Comput. C-20(7): 803-808.

Hnatek, E. R., 1973. A User's Handbook of Integrated Circuits, New York: John Wiley & Sons.

Hoff, M. E. Jr., J. Sugg, and R. Yara, 1975. Central Processor Designs Using the Intel Series 3000 Computing Elements, Application Note AP-16, Santa Clara, Calif: Intel Corporation.

Horton, R. L., J. Englade, and G. McGee, 1975. "I^2L takes bipolar integration a significant step forward," Electronics 48(3): 83-90.

IBM, 1973. IBM System/3 Model 15 Processing Unit Theory-Maintenance Diagrams, Form SY31-0367-1, White Plains, N. Y.

Johnson, A. M., 1971. "The microdiagnostics for the IBM 360/30," IEEE Trans. Comput. C-20(7): 798-803.

Peterson, W. W., and M. 0. Rabin, 1959. "On codes for checking logical operations," IBM J. Res. Develop. 3(2): 163-168.

Peterson, W. W., and E. J. Weldon, 1972. Error-Correcting Codes, Cambridge, MIT Press.

Ramamoorthy, C. V. and L. C. Chang, 1972. "System modeling and testing procedures for microdiagnostics," IEEE Trans. Comput. C-21(11): 1169-1182.

Rao, T. R. N., and P. Monteiro, 1972. "A residue checker for arithmetic and logical operations," Dig. 1972 Int'l. Symp. Fault-Tolerant Computing, IEEE pub. 73CH0772-4C, 79-84.

Reddy, S. M., 1974. "A note on self-checking checkers," IEEE Trans. Comput. c-23(10): 1100-1102.

Sellars, F. F., M. Y. Hsaio, and L. W. Bearnson, 1968. Error Detecting Logic for Digital Computers, New York: McGraw-Hill.

Toy, W. N., 1971. "Modular LSI control logic design with error detection," IEEE Trans. Comput. C-20(2): 161-166.

Usas, A. M., 1975. "A totally self-checking checker design for the detection of errors in periodic signals," IEEE Trans. Comput. C-24(5): 483-488.

Usas, A. M., 1976. "Error management in digital computer input/output systems," Ph.D. thesis, Stanford Cniversity, Stanford, California.

Wakerly, J. F., 1973. "Low-cost error detection techniques for small computers," Ph. D. Dissertation, Department of Electrical Engineering, Digital Systems Taboratory, Stanford University, Stanford, CA.

Wakerly, J. F., 1974. "Partially self-checking circuits and their use in performing logical operations," IEEE Trans. Comput. C-23(7): 658-666.

Wakerly, J. F., 1975. "Checked binary addition using parity prediction and checksum codes," Technical Note No. 39, Digital Systems Laboratory, Stanford University, Stanford, California.

Wakerly, J. F., C. R. Hollander, D. Davies, V. Coleman, K. Rallapalli, J. R. Mick, and C. Ghest, 1975. "LSI microprogram sequencers," Proc. Eighth Annual Workshop on Microprogramming, IEEE publ. CH1053-8C, Session 4, pp. 46-68.

Wakerly, J. F., 1976a. Error-Detecting Codes, Self-Checking Circuits, and Applications. American Elsevier, New York, New York (in preparation).

Wakerly, J. F., 1976b. "SCAM? - A self-checking and maintainable processor," in preparation.