COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY . STANFORD, CA 94305

TESTABILITY CONSIDERATIONS IN MICROPROCESSOR-BASED DESIGN

By

John P. Hayes
Departments of Electrical Engineering and Computer Science
University of Southern California
Los Angeles, California 90007

and

Edward J. McCluskey
Center for Reliable Computing
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Technical Report No. 179

November 1979

This work was sponsored in part by the Air Force Office of Scientific Research
under Grants AFOSR-77-3325 and AFOSR-77-3352,
the Joint Services Electronics Program
under Contract F44620-76-C-0061, and the
National Science Foundation under Grant MCS76-05327.

TESTABILITY CONSIDERATIONS IN MICROPROCESSOR-BASED DESIGN

by

John P. Hayes

Departments of Electrical Engineering and Computer Science
University of Southern California
Los Angeles, Ca. 90007

and

Edward J. McCluskey

Center for Reliable Computing
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, Ca. 94305

Technical Report No. 179

November 1979

ABSTRACT

This report contains a survey of testability conditions in microprocessor-based design. General issues of testability, testing methods, and fault modeling are presented. Specific techniques of testing and designing for testable microprocessor-based systems are discussed.

Table of Contents

Section	Subsection		Page
	List of Figures		iv
1.	INTRODUCTION		1
2.	TESTING METHODS		2
	2.1	Concurrent Testing	3
	2.2	Explicit Testing	3
	2.3	Compact Testing	6
3.	FAULTS AND TESTS		8
	3.1	Functional Faults	9
	3.2	Stuck-Line Faults	11
	3.3	Pattern-Sensitive Faults	13
4.	TESTING MICROPROCESSOR-BASED SYSTEMS		15
	4.1	Programmed Tests	16
	4.2	Testing tne Entire System	19
5.	DESIGN FOR TESTABILITY		21
	5.1	Test Point Insertion	22
	5.2	Logic Design Methods	23
6.	SELF-TESTING SYSTEMS		25

6.1 Coding Techniques 26

6.2 Hardware-Implemented Self-Testing 27

6.3 Programmed Self-Testing 29

6.4 Multimicroprocessor Systems 30

7. REFERENCES 33

List of Figures

Page

1. Two typical testing methods - (a) stored response testing and (b) comparison testing using a reference or gold unit. . . . 5

2. Two compact testing methods - (a) transition counting and (b) signature analysis. 7

3. A 1-bit processor slice C. 10

4. A 4-bit, bit sliced processor composed of four copies of C. 12

5. Representative patterns of RAM states created by two RAM tests - (a) checkerboard and (b) Galpat. 14

6. Architecture of the 8080 microprocessor. 18

7. External testing of a microprocessor-based system. . . . 20

8. A sequential (a) before and (b) after modification to introduce the scan-in scan-out testability feature. 24

9. A self-checking logic circuit. 28

10. Flowchart of a microprocessor system self-test program. . 31

11. CPU of a self-testing system. 32

12. A simple diagnostic graph. 34

1 INTRODUCTION

Testability refers to the ease with which the presence and perhaps the location of a fault or faults within a system can be discovered. It has come to be a significant factor influencing the total life-time cost of a digital system as well as the initial manufacturing cost. Current design trends emphasize the use of complex components employing large-scale integration (LSI). The key component in many such systems is a microprocessor, which is a programmable processor consisting of a small number of integrated circuits (ICs), often just a single IC. The entire digital system has the form of a microcomputer comprising a microprocessor which acts as the central processing unit (CPU) or system controller, memory circuits (ROMs and RAMs), and input/output (IO) circuits. The problems of testing microprocessor-based systems and designing them to be easily testable are surveyed in this paper.

A digital system is tested by applying a sequence of input patterns (tests) which produce erroneous responses when faults are present. *Fault detection* or go/no-go tests are intended to determine whether or not a system contains a fault. *Fault location* tests attempt to isolate a fault to a specific component; it is usually desired to isolate faults to easily replaceable components.

A system is considered to have good testability if a high level of fault coverage can be achieved at an acceptably low cost [50]. *Fault coverage* means the fraction of faults that can be detected or located within the unit under test (UUT). Several related factors contribute to the cost of achieving good testability.

1. The cost of the external test equipment and personnel needed to apply the various test procedures to the UUT.

2. The extra equipment built into the UUT to facilitate testing. This can include special logic circuits, extra IO connections used as test points, and memory space occupied by test programs.

3. The additional design and development costs needed to make the system testable. Included here is the cost of obtaining the test patterns needed to exercise the system.

4. The time required to apply the test patterns and to analyze the responses produced. The relative importance of each of the above cost factors depends on the system being designed, its intended applications, and the manner in which it is to be tested.

Microprocessor-based systems are difficult to test for several reasons.

1. The number of' possible faults is extremely large. An LSI circuit contains thousands of basic components (gates) and interconnecting lines which are individually subject to failure.

2. Access to internal components and lines is severely limited by the number of IO connections available. A typical microprocessor may contain 5,000 gates but only 40 IO pins. This means that many layers of potentially faulty logic circuits can separate a particular component from the points at which the test patterns can be applied and the responses observed.

3. A consequence of the large number of possible faults is the need for very large numbers of test patterns. This may require ways of encoding this information to reduce its size.

4. The system designer may not have a complete description of the ICs being used. Microprocessor specifications are typically register level descriptions consisting of block diagrams, a listing of the microprocessor's instruction set, and some information on system timing. Test procedures must often be designed using meager data of this kind.

5. New and complex failure modes such as pattern sensitivity occur. The impact of the foregoing difficulties can be greatly reduced by using design techniques that enhance testability, and by matching the test generation and application processes to these design techniques. The inherent ability of a microprocessor to execute complex programs can also be exploited.

Microprocessor testing is of interest in many different situations: semiconductor component manufacturering, test equipment design, system design, and system maintenance. We will be primarily interested in testing from the design viewpoint. We also restrict our attention to functional testing, which is only concerned with the logical behavior of the UUT. (This is often contrasted with parametric testing which deals with electrical properties such as voltage levels and signal delays.)

2 TESTING METHODS

Every testing procedure involves the generation of test data (input test patterns and output responses), application of the test patterns to the DUT, and evaluation of the responses obtained. Many different approaches to digital system testing have evolved, which are distinguished by the techniques used to generate and process the test data. They can be divided into two broad categories which we

term (1) concurrent or implicit and (2) explicit. In the concurrent testing approach, the data patterns appearing during normal computation serve as test patterns, and built-in monitoring circuits are used to detect faults. Thus testing and normal computation can proceed concurrently. Parity checking is the most common form of concurrent testing. In explicit testing special input patterns are used as tests, so that normal computation and testing occur at different times. Explicit tests can be applied by test equipment that is external to the UUT (external testing), or they can be applied internally (self-testing;). Most computer-based systems have little self-testing ability. Complete self-testing has hitherto been limited to applications such as spaceborne computers and electronic switching systems where stringent reliability requirements justified the extra cost of self-testing. The advent of low-cost microprocessors with significant processing abilities has greatly increased the number of systems where self-testing is both desirable and economically feasible.

2.1 Concurrent Testing

Concurrent testing is commonly implemented by coding techniques which allow the signal patterns generated during normal computation (with minor modification if necessary) to serve as test patterns. Thus a word to which a single parity bit is appended becomes a test for single-bit faults affecting the word in question. Parity generation logic circuits are used to produce the expected response pattern, which may then be compared with the actual response, the entire operation occurring in parallel with normal computation. Self-checking circuits, which are discussed in Sec. VI, also employ coding techniques to achieve concurrent testing. Very high fault coverage can be obtained by a concurrent testing technique in which the functional units are replicated and run in parallel with the outputs being compared and perhaps voted on. This technique is used in systems such as telephone offices in which very high availability is important. It is somewhat similar to the explicit testing technique called comparison testing to be discussed subsequently.

2.2 Explicit Testing

Explicit methods are used when the extra circuits and interconnections needed for concurrent testing are too costly, or when the signals produced during normal computation are inadequate for testing purposes. In the latter case, specific test pattern generation procedures are required. The test patterns are produced

either manually by a design or test engineer, or else special hardware- or software-implemented algorithms, called test-generation programs, are used. Manual test generation is widely used.

Test-generation programs typically make use of a simulation model of the target system. This model can range in complexity from a detailed logic model specifying every gate, to a crude heuristic model that approximates the behavior of a few major subsystems.

If the test patterns are difficult to generate, i.e., if substantial computation is necessary to construct them, then they are usually computed in advance (off-line), and stored in the tester along with the expected response to each test pattern. The stored set of test patterns and responses is called a fault dictionary. The fault dictionary may also identify the faults corresponding to specific test pattern and response combinations. Testing based on stored test data of this kind is often called stored response testing. It is typically implemented in three steps.

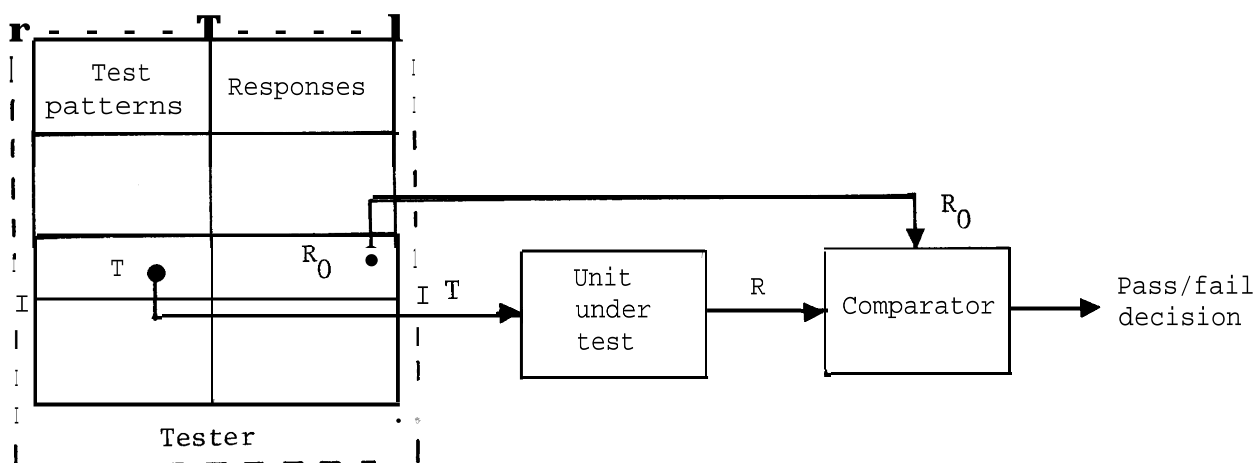
1. A sequence of one or more test patterns T is retrieved from the fault dictionary.

2. T is applied to the UUT and the response R is recorded.

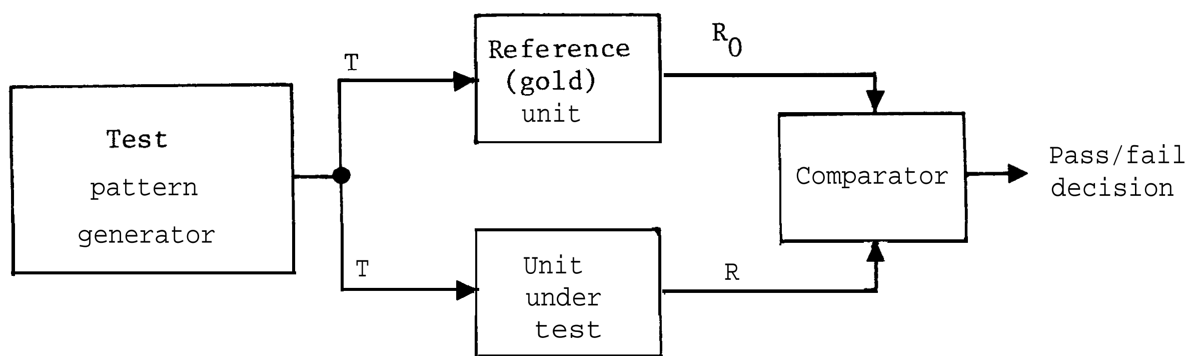
3. R is compared to the fault-free response R0 stored in the fault dictionary and appropriate action is taken if they differ. The main drawbacks of this approach are the cost of computing the fault dictionary, and the memory space needed to store it in the tester. However, the test patterns need only be computed once, an advantage when no simple way is known for obtaining suitable test patterns. Figure 1a illustrates stored response testing.

The cost of storing the the test pattern responses can be reduced by using a technique called comparison testing. Note that it may still be necessary to store the test patterns themselves. Comparison testing makes use of several copies of the UUT, each of which processes the same input signals; faults are detected by comparing the responses of all the units. A response differing from that of a known fault-free unit serves to pin-point a faulty unit. This technique can be implemented with as few as two copies of the UUT, one of which, the so-called gold unit, acts as a reference against which the other unit is compared. Figure 1b illustrates this type of testing. If there are three or more copies of the UUT, the majority response may be taken as the correct one, allowing voting circuitry to be used for concurrent fault diagnosis.

Stored response testing may be contrasted with methods in which the test data is computed each time the UUT is tested; we refer to these as algorithmic testing methods. The algorithmic approach



(a)



(b)

Fig. 1. Two typical testing methods

(a) Stored response testing
(b) Comparison testing using a reference or gold unit

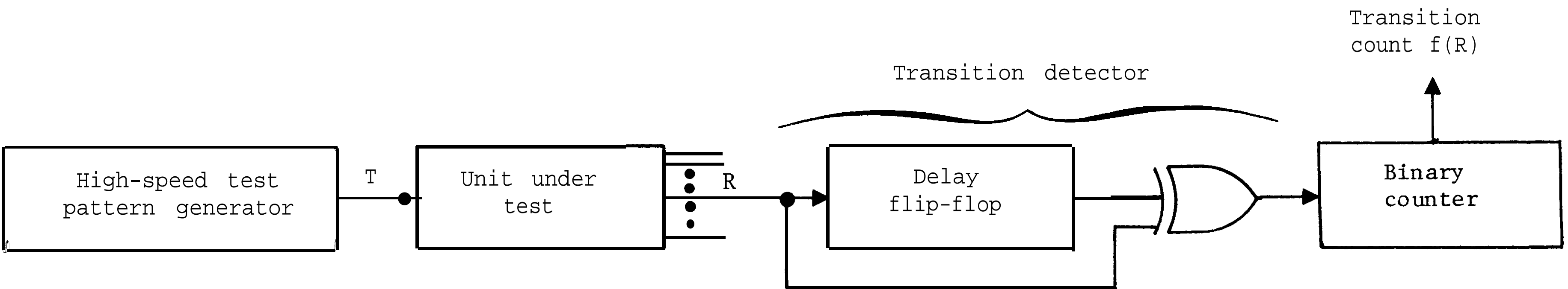
requires some rapid and therefore simple method for determining test data. A common test pattern source is a high-speed (hardware or software) counter that generates sequences of test patterns in a fixed or pseudo-random order. For example, an n-input combinational component can be tested by applying all possible 2^n binary patterns to it. These patterns can easily be obtained by incrementing an n-bit counter through all its 2^n states. The correct response RO to a particular test T may be determined rapidly by comparison testing using a gold unit (see Figure 1b). An example of a commercial tester for microprocessors based on this kind of comparison testing is the Megatest Q8000 [11].

Another way of obtaining the good response R_0 is for the test pattern generator to compute it. Such an approach is feasible if the test generator contains a computer (a mini- or microcomputer suffices) and there is a fairly simple function or algorithm relating T and R_0. For example, an adder may be tested with a test pattern comprising two numbers A and B that are to be added by the UUT. The correct response A+B is computed independently by the tester and compared to the sum computed by the UUT. This approach is well-suited to testing microprocessors, because many of the functions to be tested are defined by algorithms programmed into the UUT. Microprocessor testers are therefore often designed to emulate, i.e., to execute directly, the instruction set of the UUT's microprocessor. This allows the tester to control the UUT completely during testing, temporarily overriding the microprocessor in the UUT. This technique, which is very useful for design debugging, is called <u>in-circuit emulation</u>. It is a feature of many microcomputer development systems [44].

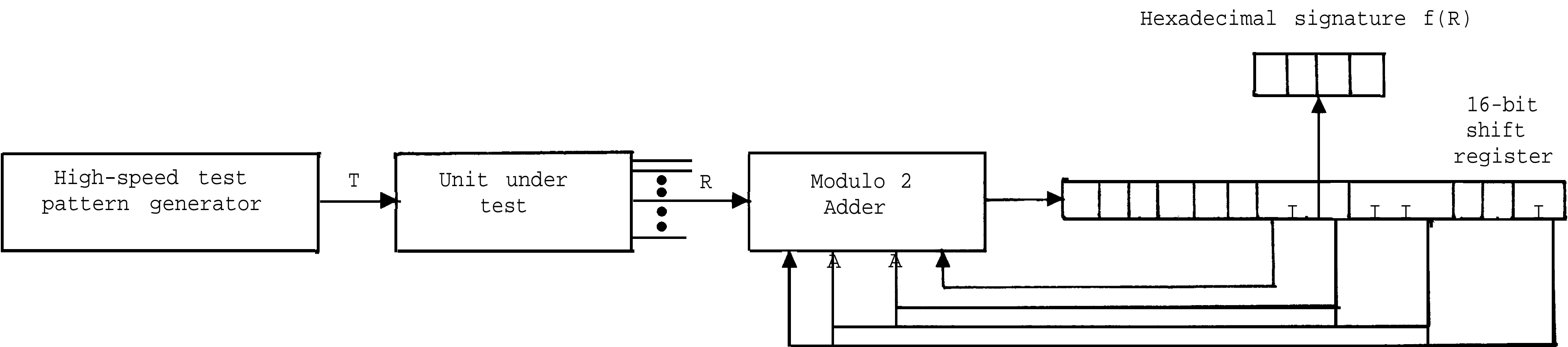
2.3 Compact Testing

High-speed test generators, particularly algorithmic testers, can produce huge amounts of response data whose analysis and storage can be quite difficult. Compact testing methods attempt to compress the response data R into a more compact form f(R), from which the information in R about the fault status of the UUT can be derived. The compression function f is one that can be implemented with simple circuitry. Thus compact testing entails little test equipment, and so is especially suitable for field maintenance.

Figure 2 illustrates two representative compact testing techniques. In each case, a fixed sequence T of tests is applied to the UUT and the output response is compacted to obtain f(R) which is recorded. The compacted response is then compared either automatically or manually with the precomputed compact response. By using a series of different test points and standard signal tracing techniques, a fault can be detected and located.



(a)



(b)

Fig. 2. Two compact testing methods: (a) Transition counting; and (b) Signature analysis.

The first method, called transition counting, computes the number of logical transitions (a 0 changing to a 1 and vice versa) occurring in the output response at the test point. All this requires is a simple transition detection circuit and a binary counter connected as shown in Figure 2a. Transition counting has been implemented in a number of commercial testers, and appears to provide fairly good fault coverage [27, 37, 46]. It has the advantage of being insensitive to normal fluctuations in signal duration, and so is especially useful for testing asynchronous circuits. Similar compact testing schemes such as ones counting have also been proposed [30, 38, 53].

Recently hewlett-Packard Corp. has proposed a compact testing scheme called signature analysis intended specifically for testing microprocessor-based systems [33, 65]. As illustrated in Figure 2b, the output response is passed through a 16-bit linear feedback shift register. The contents f(R) of this shift register after all the test patterns have been applied is called the (fault) signature, and is recorded or displayed as a 4-digit hexadecimal number. Thus an output response of arbitrary length is compressed into four digits. The compression function f implemented here is such that every single-bit error in the response stream, as well as many multiple-bit errors, cause f(R) to differ from $f(R_0)$, the precomputed correct compact response. A 99.998% probability of detecting a faulty output response has been claimed for signature analysis [33]. (Note that .99998 is essentially an estimate of the factor by which the fault coverage is reduced due to the use of the linear feedback shift register to compact the output response data). The correct compact responses for each test point are recorded on the system schematics; a maintenance person can compare the obtained response with that given on the diagrams to determine whether or not a fault has been detected.

3 FAULTS AND TESTS

Every testing procedure aims at diagnosing some particular class of faults, although in practice these faults are not always well-defined. An explicit fault model is necessary, however, if the fault coverage of a set of tests is to be determined. The complexity of LSI circuits makes it necessary to reconsider some of the fault models that have traditionally been employed for digital system testing.

3.1 Functional Faults

The UUT can be regarded as an n-input, m-output, s-state finite-state machine, an (n,m,s)-machine for short. Perhaps the most general of the useful fault models, which we may call the <u>functional fault</u> model, allows an (n,m,s)-machine to be changed by a fault to any (n,m,s')-machine, where s' does not exceed s. (Note that some restriction on the number of states of the faulty machine is necessary.) Under this model a combinational circuit, which is an (n,m,1)-machine, always remains combinational when faults are present. To test a combinational circuit M for all functional faults, it is necessary and sufficient to apply all 2^n possible input patterns to M which, in effect, exhaustively verifies its truth table, and thereby provides rather complete fault coverage. Although the number of tests is often very large, they can be easily and rapidly generated using the algorithmic technique mentioned earlier. This type of testing can sometimes be applied to the combinational subcircuits of a sequential UUT. When the circuit under test must be treated as sequential (s>1), complete detection of functional faults requires a special type of test called a checking sequence. The theory of checking sequences is well-developed [29, 39], but unless s is very small, checking sequences are extremely long and difficult to generate. We now illustrate an application of the functional fault model to a specific class of microprocessors.

<u>Example 1</u>: Testing a simple bit-sliced microprocessor [64]

A bit-sliced microprocessor is an array of n identical ICs called (bit) slices, each of which is a simple processor for operands of length k bits, where k is typically 2 or 4. The interconnections between the n slices are such that the entire array forms a processor for nk-bit operands. The simplicity of' the individual processors, and the regularity of the array interconnections make it feasible to use systematic methods for fault analysis and test generation. Unfortunately, the more widely used non-bit-sliced microprocessors do not share these properties.

Figure **3** shows a circuit model for a 1-bit processor slice which has most of the features of a commercial device such as the Am2901 [1]. (The main omissions are the logic circuits for implementing shift and carry-lookahead.) This circuit consists of five basic components or modules, two of which are sequential (registers A and T), and tnree combinational (the two multiplexers and the ALU). The ALU can perform addition, subtraction and the standard logical operations. Each module may fail according to the foregoing functional model, but only one module is allowed to be faulty at a time. A complete test set for this circuit must apply all possible input patterns to each combinational module, and a checking sequence to each sequential module. In addition, the

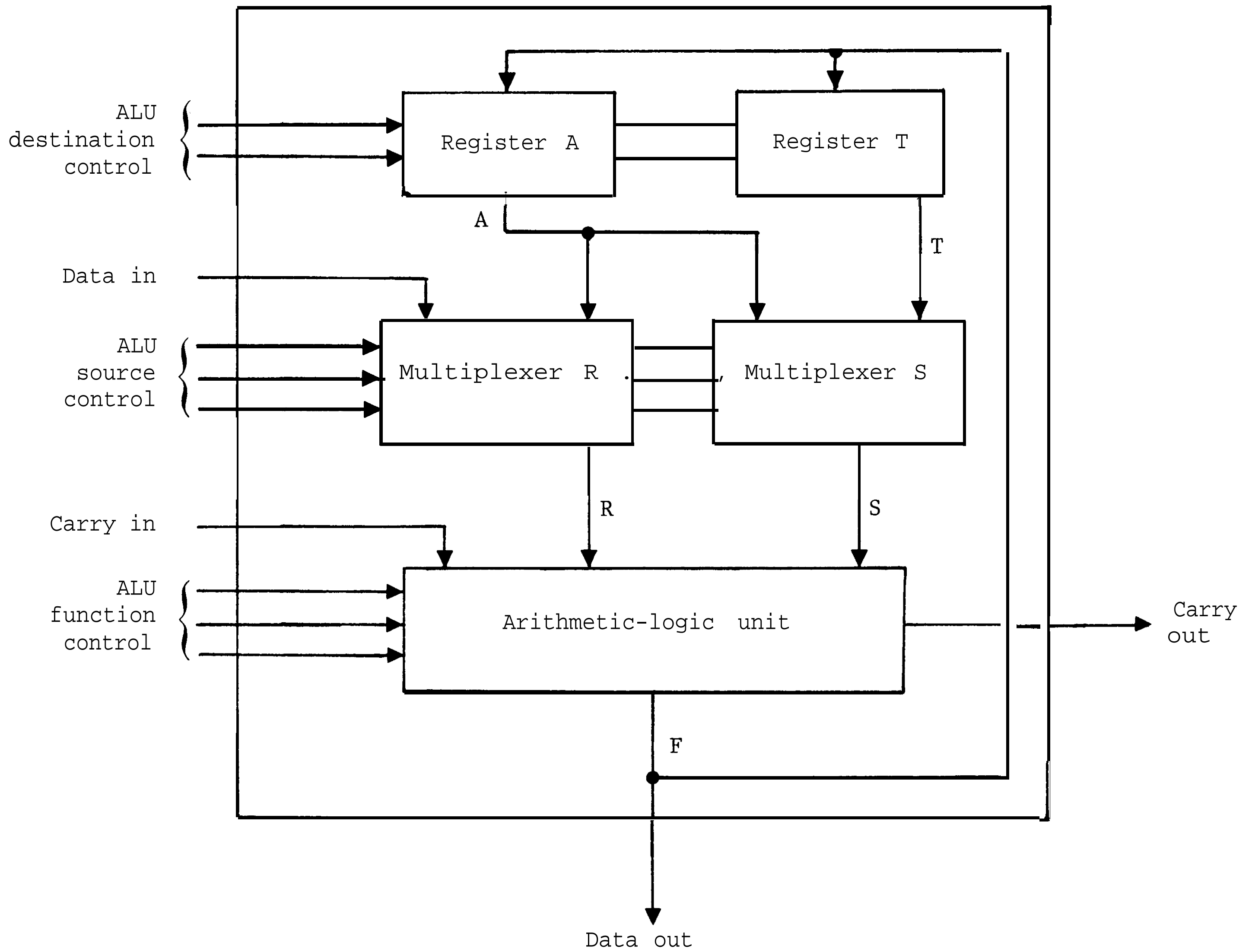


Fig. 3. (a) A 1-bit processor slice C.

responses of each module must be propagated to the two primary output lines. The tests required by the individual modules are easily generated because of the simplicity of the modules, a direct consequence of the small operand size (k=1). The module tests can be overlapped in such a way that **82** test patterns suffice for testing the entire circuit [64]. Note that the 6-input ALU alone requires **64** test patterns. The number of test patterns produced in this manner is considerably less than the number generated for comparable processors by conventional heuristic techniques [49].

An important property of this type of processor slice is the fact that the tests for a single slice can easily be extended to tests for an array of the slices, for example, the 4-bit array depicted in Figure 4. In fact, an array of arbitrary length can be tested by the same number of tests as a single slice, a property called <u>C-testabilitv</u> [28]. Other components of a computer system such as memories and microprogram sequencers [1] can also be bit-sliced, and thus share some of the good testability features of a bit-sliced processor.

3.2 Stuck-Line Faults

The most widely used fault model for logic circuits is the <u>single stuck-line (SSL)</u> model, which allows any interconnecting line to be stuck at logical 1 (s-a-1) or stuck at zero (s-a-0). Only one line is allowed to be faulty, and the circuit components, gates,flip-flops and the like, are assumed to be fault-free. Clearly SSL faults form a small subset of the functional faults. Many common physical faults are covered by this model, and several distinct methods have been developed for generating tests for SSL faults [13]. The best known test generation method for this purpose is Roth's D-algorithm [56,59]. Complete test sets of near-minimal size can be generated for SSL faults in combinational logic circuits. Sequential circuits, even those of moderate complexity, still present serious problems.

Since a microprocessor-based system is a very complex sequential circuit, it is generally not feasible to analyze it completely using the classical gate-level SSL model. The number of possible SSL faults of this kind is enormous. Furthermore, an adequate logic circuit of the UUT at the gate level may not be available. Nevertheless, because so many common faults, such as broken connections and short circuits to ground, can be be modeled accurately by s-a-O/l lines, it is desirable to generate specific tests for at least some of the SSL faults occurring in such systems.

In practice, tests for SSL faults are often restricted to the following cases.

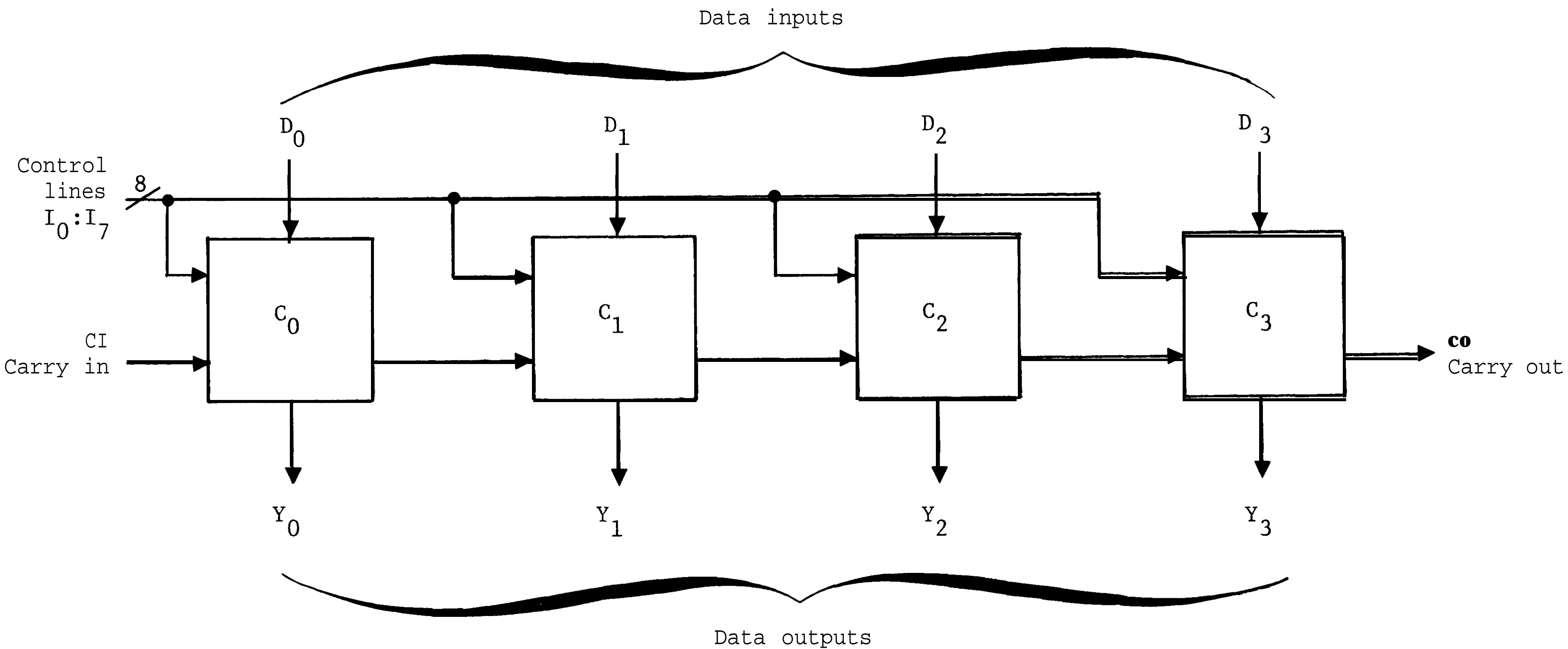


Fig. 4. A 4-bit, bit sliced processor composed of four copies of C.

1. Faults affecting the external IO pins of each IC, and the IO connections of the principal combinational or sequential modules within the IC.

2. Faults causing the main information transmission paths, e.g., buses, to become s-a-l/O.

3. Faults causing the major state variables to becomes s-a-l/O; such faults usually correspond directly to SSL faults in the associated registers and memory elements. Note that these SSL-type faults can be identified from a register-level description of the UUT. They define a restricted SSL fault model which is widely, if implicitly, used in testing complex digital systems. To detect these restricted faults, it is necessary to verify that the lines and variables in question can be set to both the 0 and the **1** values. Thus a basic test for a memory element such as a microprocessor register is to verify that a word of 0s and a word of 1s can be written into and read from it.

3.3 Pattern-Sensitive Faults

Another useful way to model faults in LSI circuits is to consider interactions between logical signals that are adjacent in space or time. Such a fault occurs when a signal x causes an adjacent signal y to assume an incorrect value. Faults of this type are termed pattern-sensitive faults (PSFs). There are many physical failure **modes** that result in pattern sensitivity. For example, electrical signals on conductors that are in close spatial proximity can interact with one another. This problem is aggravated **by** the very high component and connection densities characteristic of LSI. Another instance of pattern sensitivity is the failure of a device to recognize a single 0 **(1)** that follows a long sequence of Is (0s) on a particular line; this time-dependent PSF is a consequence of unwanted hysteresis effects. PSFs are particularly troublesome in high-density RAM ICs. Since microprocessors often contain moderately large RAMs, they too are subject to PSFs [34].

A variety of heuristic procedures have been developed to detect PSFs in memories [7, **13,** 26, 42]. Figure 5 illustrates two representative types. The RAM is viewed as a 2-dimensional array of N independent storage cells. The test called Checkerboard (Figure 5a) writes an alternating pattern of 0s and 1s into the RAM; this requires a total of N write operations. Then the contents of each RAM cell is read out and verified, requiring N read operations. These write and read steps are repeated with the positions of the 0s and Is reversed. If each l-cell read or write command is regarded as a test, then Checkerboard contains approximately 4N tests. Checkerboard

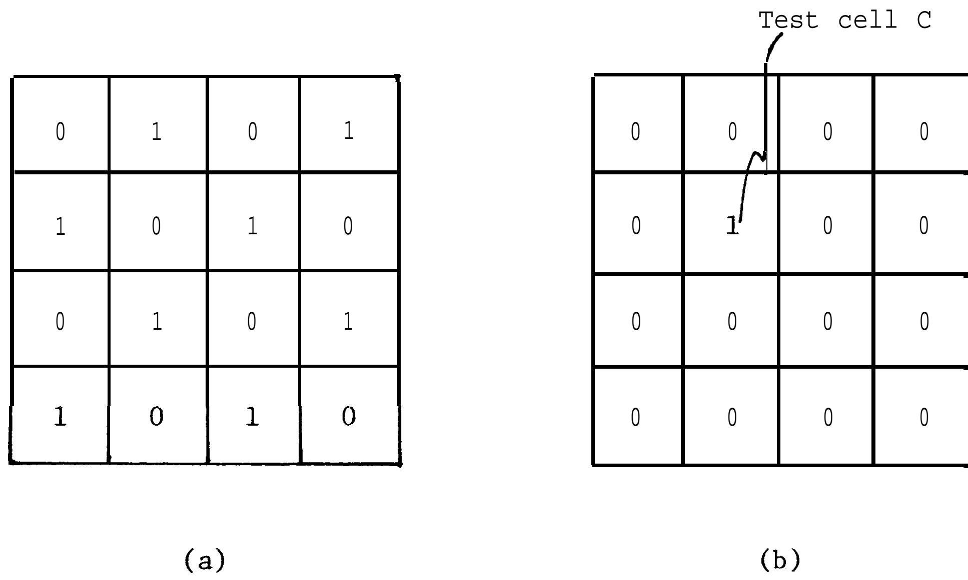


Fig. 5. Representative patterns of RAM states created by two RAM tests:

(a) Checkerboard
(b) Galpat

verifies that a 0 (1) can be transferred correctly to Or from a cell that is "surrounded" by 1s (0s), intuitively a worst-case condition for spatial pattern interference. The second representative PSF test is called Galloping 0s and 1s or <u>Galpat</u> (Figure 5b). Galpat initially writes a 0 (1) into all cells. A 1 (0) is then written into a test cell C. Each cell C' ≠ C is read in turn to determine whether its contents have beeen disturbed by the write operation addressed to C. After reading every C', C is again read to ensure that it is still correct. Thus Galpat moves back and forth or "ping-pongs" between the test cell C and all other cells, checking for unwanted interactions. The process is repeated with with every cell in the RAM playing the role of the test cell C.

Most PSF tests were derived empirically, and their underlying fault models are unclear, making it very difficult to determine the fault coverage of the tests. Attempts have been made to develop formal fault models for some kinds of PSFs [36, 51].

PSFs provide a good illustration of the testing problems caused by the rapidly increasing component densities in modern ICs. The Galpat test described above requires about $4N^2$ patterns to check an N-bit RAM. If each test pattern takes 100 ns to apply, then A 4K-bit (4096-bit) RAM can be tested by Galpat in about 2 s. However, a 1M-bit (2^{20}-bit) RAM, which is expected to appear on a single VLSI chip in the near future [14], would require about 30 hours for one application of Galpat at the same 100 ns-per-test rate.

4 TESTING MICHOPRGCESSOR-BASED SYSTEMS

In practice, tests for microprocessor-based systems are designed to exercise the UUT by applying a representative set of input patterns, and causing it to traverse a representative set of state transitions. In each case the decision on what constitutes a representative set is based on heuristic considerations. The faults being diagnosed may not be specifically identified, but they can often be related to the fault models discussed earlier. Two examples will serve to illustrate the heuristic nature of this approach.

The following list describes the 2n+4 test patterns used to test every data path and register in the Plessey System 250, a computer system designed for fault tolerance [45].

1. The all 0s pattern

2. A pattern of alternating 0s and 1s (a one-dimensional checkerboard pattern)

3. The n patterns containing a single 1 and n-l 0s

4. The logical complements of the preceding patterns. These tests detect all s-a-o/l faults associated with the data lines and state variables, as well as certain kinds of PSFs.

The program counter (PC) which generates the address of the next sequential instruction to be executed is a key component of a microprocessor. It can be tested by incrementing it sequentially through all its states, and verifying that the correct state is reached after each increment or, equivalently, that the proper address is generated [18]. This test thus checks for the existence of every possible state of the PC, and verifies the most common state transitions. Note that it does not verify all possible state transitions resulting from the execution of branch (jump) instructions.

4.1 Programmed Tests

Much of the uniqueness and power of a microprocessor-based system lies in the fact that it is program-controlled. Thus a natural tool for testing the system is a test program executed by the internal microprocessor of the UUT. This program is designed to apply appropriate test patterns to the major register-level modules of the UUT, all of which should be accessible via its instruction set. Typically, these modules are exercised by input patterns derived heuristically, and based on the functions performed by the modules under test. Note that in exercising these components, the microprocessor's instruction set is also exercised and therefore tested.

A disadvantage of this heuristic exercising approach to microprocessor testing is the absence of a suitable register-level fault model establishing a correspondence between instruction or module failures and the underlying physical faults. Consequently it is extremely difficult to determine the fault coverage of a test program. The need for a useful instruction-oriented fault model has long been recognized [48]. Recently some interesting work towards such a model has been done by Thatte and Abraham [67, 68].

A test program for a microprocessor is usually organized into a sequence of steps, each of which tests a related group of instructions or components. Once a group nas been proven fault-free, it may then be used to test other groups. The selection and sequencing of these steps is complicated by the fact that considerable overlap exists among the components affected by different instructions.

Example 2: Constructing a test program for the 8080 microprocessor [18]

The 8080, which was introduced by Intel Corp. in 1973, is perhaps the most widely used microprocessor. It is an 8-bit machine of fairly conventional design [43]. Figure 6 shows a block diagram of the 8080. This register-level description is adequate for applying heuristic fault models such as the restricted SSL model discussed in the preceding section. The 8080 contains a simple arithmetic-logic unit and six 8-bit general-purpose registers; the latter may be paired to form three 16-bit registers. (16 bits is the main memory address size.)

Table 1 lists the main steps in an 8080 test program [18]. The 8080 UUT is assumed to be connected to an external tester which has access to the IO lines comprising the data, address and control buses of the 8080. First, the tester resets the UUT. Then the 16-bit program counter PC is incremented through all its 65,536 states. This can be implemented by having the tester place a single instruction NOP (no operation) on the data (input) lines of the 8080 under test, and causing the 8080 to execute it repeatedly. The effect of NOP is to increment the PC and cause it to place its contents on the outgoing address lines where they can be observed and checked by the tester. This checking can be done rapidly by comparing the PC state to that of a hardware or software counter in the tester which is incremented on-line in step with the PC.

The next step is to test the various general-purpose registers by transferring 8-bit test patterns to and from them and checking the results. All possible test patterns may be used, because their number (256) is small, and they are easy to generate algorithmically. The tests are implemented by several data transfer instructions (MOV, LXI, PCHL), which are themselves also tested in this step. After a pattern is applied to a register r, the tester can inspect the contents of r by transferring it to the PC via the HL register. (The PCHL instruction which swaps the contents of PC and HL is used; the 8080 lacks instructions for transferring data directly between the PC and other registers.) Since the PC was tested in the first step, its contents can be taken to be correct , and they can be observed directly via the address bus. (Some pitfalls of testing 8080 registers in this way are discussed by Smith [63].) The remaining steps of the test program exercise the other components and instructions of the 8080 in a similar manner. Unfortunately, little data is available on the fault coverage of this type of test program.

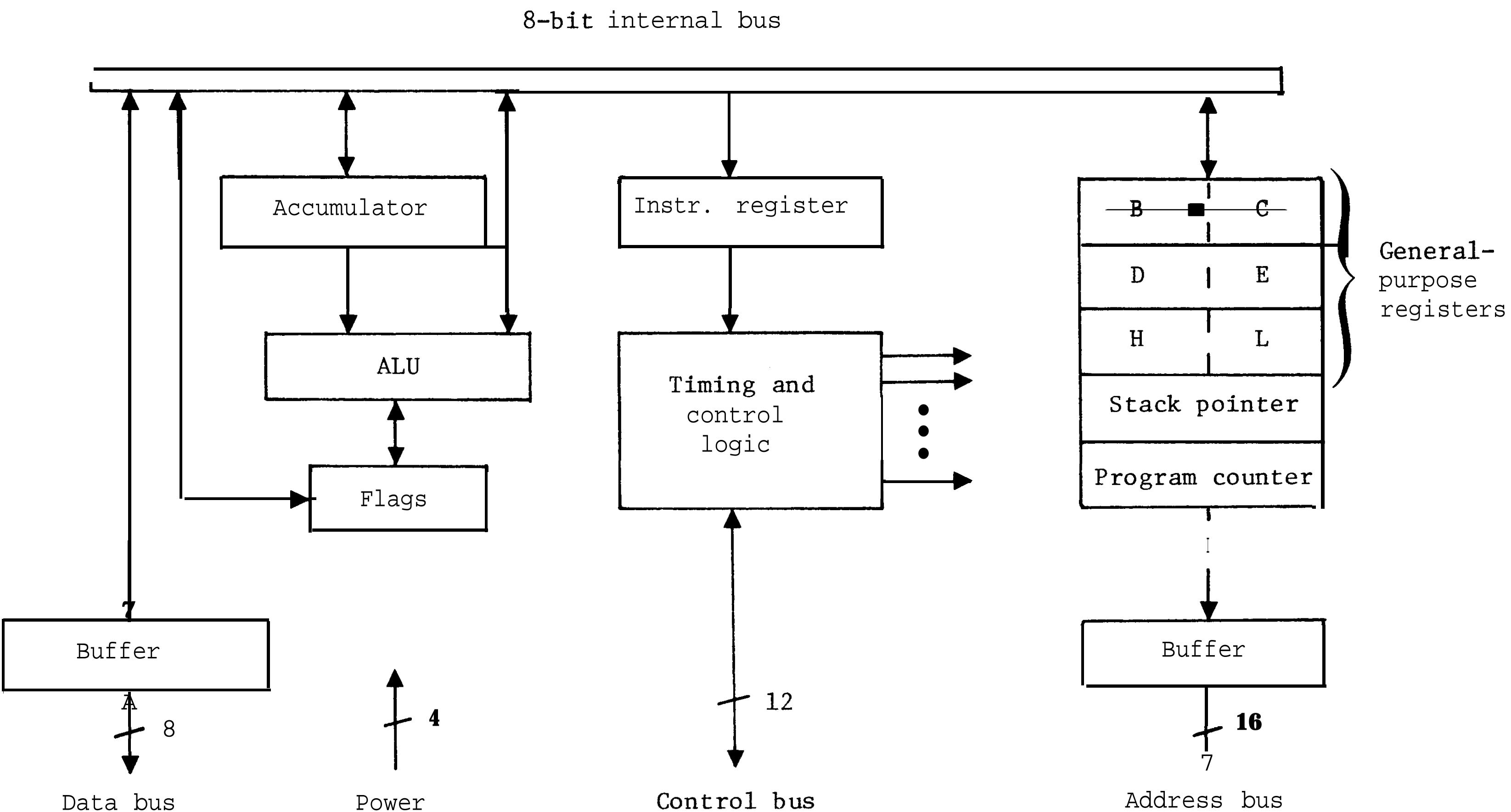


Fig. **6.** Architecture of the 8080 microprocessor.

Table 1. The Main Steps in a Program to Test the 8080 Microprocessor.

1. Reset the 8080 UUT.
2. Test the program counter PC by incrementing it through all its states via the NOP instruction.
3. Test the six 8-bit general-purpose registers by transferring all possible 256 test patterns to each register in turn via the PC.
4. Test the stack pointer register by incrementing and decrementing it through all its states; again access it via the PC.
5. Test the accumulator by transferring all possible test patterns to it via previously tested registers.
6. Test the ALU and flags by exercising all arithmetic, logical and conditional branch (flag-testing) instructions.
7. Exercise all previously untested instructions and control lines.

4.2 Testing the Entire System

A complete microprocessor-based system can be tested by using its microprocessor as the primary source of test patterns. Consider the problem of testing a system with the typical bus-oriented architecture shown in Figure 7. IO device testing is not considered here, since it varies from device to device. Again we assume that there is an external tester which has access to the various system buses. In addition, we require this external tester to be able to disconnect parts of the system from the buses during testing; this can often be done either electrically or mechanically. We now outline the main steps in a fairly general system testing procedure [41].

First a simple test is performed on the microprocessor to determine if one of its main components, the program counter PC, is operational. As discussed earlier, this can be done by making the PC traverse all its states causing it to place all possible address patterns on the system address bus. It is necessary to isolate the microprocessor from the data bus during this test so that the external tester can supply the instructions needed to increment the PC. In the case of the 8080 discussed earlier, the tester need only place a single instruction (NOP) on the microcomputer's data input lines in order to cause the PC to increment continuously. While the

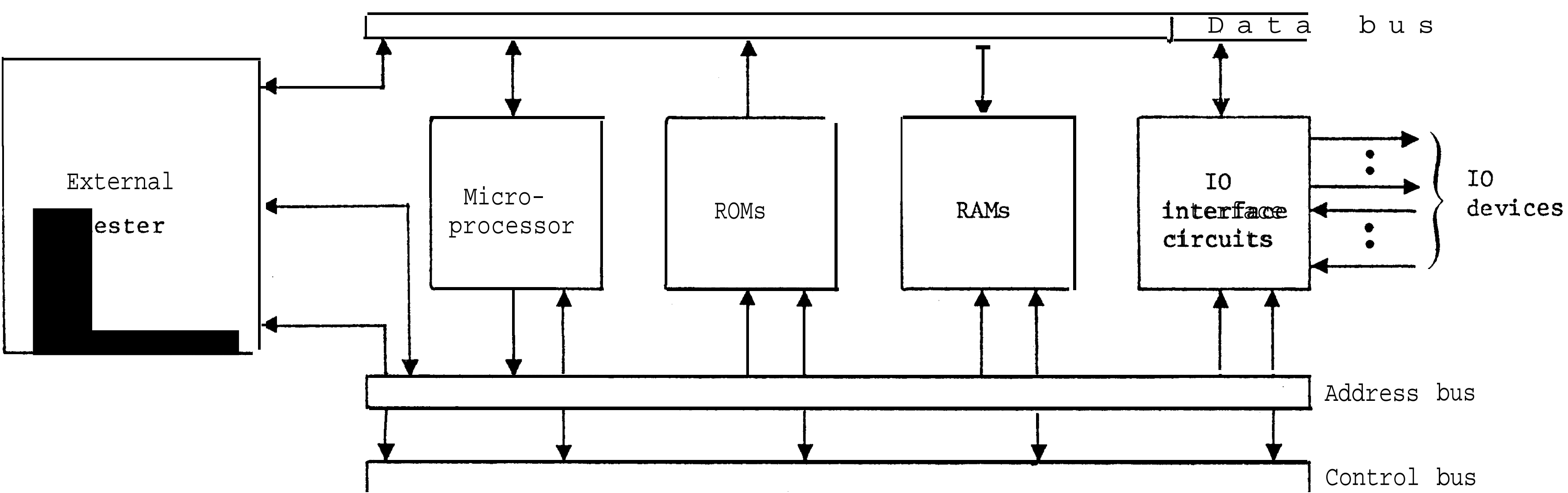


Fig. 7. External testing of a microprocessor-based system.

PC is being incremented, a mode of operation called free-running, the external tester monitors and checks the signals appearing on each of the system's address lines. It is relatively simple to do this via compact testing techniques like signature analysis.

Next the system ROMs are tested with the microprocessor still in free-running mode. During this test the RAMs are disconnected from the data bus. Since the microprocessor generates all memory addresses, it causes every ROM location to be accessed automatically. The tester monitors the signals which represent the ROM contents as they appear on the data bus. Since the ROM contents are fixed, a fixed signature can easily be associated with each ROM.

At this point the microprocessor, ROMs and system buses have been checked to determine if they are working. The remaining parts of the system are checked via specific exercising programs, which may be stored in the external tester or the UUT's ROMs. The RAMs can be tested by programs such as Checkerboard and Galpat described already. The IO interface circuits normally resemble memory devices, and can be tested by memory-oriented check programs. In order to do this under control of the microprocessor in the UUT, the output ports can be connected directly to the input ports using jumper connections, a technique called loop-back [22,31]. This allows the CPU to send a test pattern to an output port and read it back via an input port for checking.

All the tests outlined so far can be implemented with a very small subset of the microprocessor's instruction set, mostly NOP, LOAD and STORE. It remains to exercise the other instructions, which can be done along the lines of the 8080 tests described in Ex.2 above.

5 DESIGN FOR TESTABILITY

A fundamental requirement for testing any system S, be it a microcomputer or a large systems program, is the ability to be partitioned into easily testable subsystems S(1),S(2),...,S(k). Each S(i) should be small enough that it can be tested directly by the available testing procedures. While testing S(i), it must usually be isolated from the rest of the system so that a tester, either alone or in conjunction with other subsystems of the UUT (preferably those that are known to be fault-free), can control the inputs of S(i) and observe its responses. Thus the controlability and observability of the various parts of the system contribute in an important way to its testability.

Controllabilitv may be defined informally as the ease with which arbitrary input patterns can be applied to the inputs of S(i) by exercising the primary inputs of S. Observability is the ease with which the responses of S(i) can be determined by observing the primary outputs of S. These intuitive concepts have been quantified by Stephenson and Grason [66], so that measures of controlability and observability can be determined directly from a gate- or register-level circuit for S. These measures can be used to predict the difficulty of generating test patterns for S.

5.1 Test Point Insertion

Perhaps the most direct way to improve controllability and observability is to add extra input and output lines to S so that the logical distance between the IO lines of S(i) and those of S is reduced. Added IO access points of this kind are called test points. Usually a certain amount of extra logic is needed to insert the test points, particularly input test points. In the extreme case we can add enough test points to convert the IO lines of S(i) into IO lines of S during testing. If this is done for all S(i), then testing becomes trivial [35]. However, in LSI circuits space for test points is severely limited, hence testability improvement must be carried out using very few test points, often only one or two. Controllability and observability measures are useful in determining the best locations for a limited set of test points [6, 17, 661. Good locations for test points include:

1. The outputs of memory elements, particularly those used for control purposes

2. The IO connections of deeply buried components

3. Lines with large fan-in or fan-out, e.g., lines in the system bus.

Surprisingly, a high degree of testability can be achieved with only a very small number of test points, although usually at the expense of also adding significant amounts of extra logic to support the test points. Reddy has shown that every n-input combinational function can be realized by a circuit that requires only n+4 easily generated test patterns for complete fault detection [58]. Reddy's circuit, which consists of a cascade of EXCLUSIVE-OR gates driven by a set of AND gates, employs a single input test point, and at most two output points. However, the large propagation delay of this circuit makes it mainly of theoretical interest. A method called scan-in scan-out for making a sequential circuit easily testable using only a few test points was developed by Williams and Angell [70] and is described below.

5.2 Logic Design Methods

Logic designers have compiled a set of design guidelines to simplify testing which, in general, aim at increasing controllability and observability [9, 40, 47]. The following list is representative.

1. Allow all memory elements to be initialized before testing begins, preferably via a single reset line.

2. Provide a means for opening feedback loops during testing.

3. Allow external access to the UUT's clock circuits to permit the tester to synchronize with the UUT or to disable the UUT.

Design techniques that introduce regularity into a system's structure, e.g., the use of programmable logic arrays [52] also facilitate testing. The scan-in scan-out technique mentioned above allows a system that does not have a regular structure during normal operation to assume a regular easily testable structure during testing. This important design technique has been adopted recently by several computer manufacturers [21, 24, 32].

Example 3: Scan-in scan-out design [70]

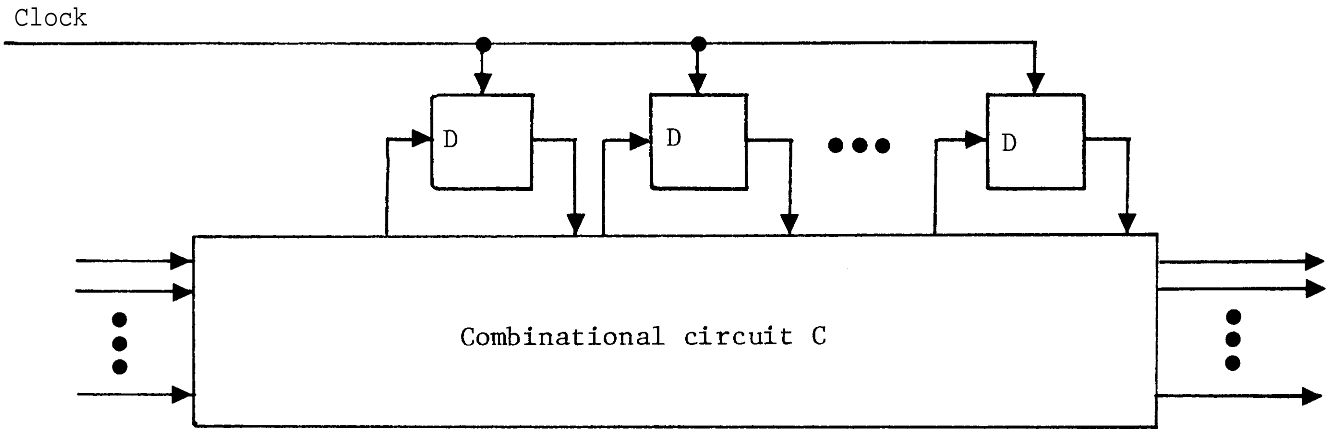
The key concept here is to design an LSI chip such as a microprocessor so that all memory elements can be connected together to form a single shift register SR during testing. This causes the entire chip to appear like a large combinational circuit whose IO lines are either chip IO lines or IO lines of SR. The system can then be tested by loading a test pattern into SR (scan-in) , allowing the system to respond to the test pattern , then reading out the response from SR (scan-out). The scan-in scan-out scheme has several advantages.

1. The test generation problem is largely reduced to the relatively easy one of testing a combinational circuit.

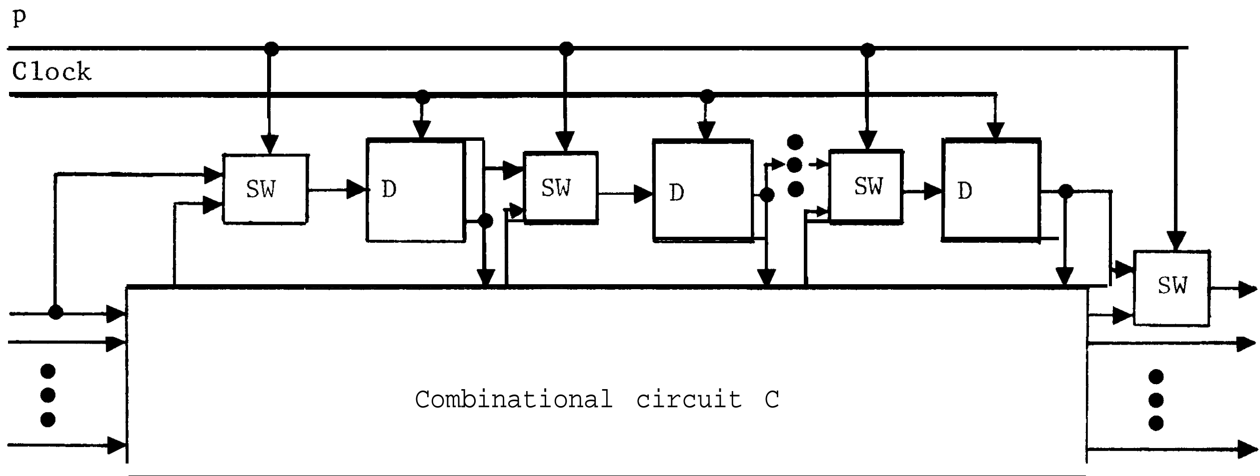
2. No matter what the length of SR, it can be accessed via one input and one output line which, in fact, may be normal IO lines of the system. A special control line is needed to switch the system between the normal mode of operation and the test mode. Thus at most three test points must be added to the basic system, and one extra test point may suffice.

3. Relatively little extra logic circuitry is added to the basic design, and its effect on system performance is minimal.

Figure 8 shows the design modifications needed to implement



(a)



(b)

Fig. 8. A sequential (a) before and (b) after modification to introduce the scan-in scan-out testability feature.

scan-in scan-out. The basic unmodified system, which appears in Figure 8a, is assumed to use clocked D (delay) flip-flops as its memory elements. It is modified for scan-in scan-out by adding a simple logic circuit (a multiplexer) denoted SW to the input of each flip-flop. SW acts as a switch which, under control of the special input signal p, connects the flip-flop input either to the combinational circuit C (normal mode), or to the output of another flip-flop (test mode). Copies of SW are also included which allow two normal IO lines to be used as IO lines of the shift register SR. Thus the only added test point is for the control signal p.

The circuit S of Figure 8b is tested as follows.

1. Switch to the test mode (p = 1).

2. Shift a test pattern into SR (scan-in).

3. Return to the normal mode (p = 0).

4. Clock the system to apply the input test pattern in SR and at the primary inputs of S. Some responses appear at the primary out-puts of s; the majority are stored in SR.

5. Switch to the test mode.

6. Shift the output responses out of SR (scan-out).

7. Verify all responses. Return to step 2 with the next test pattern.

A version of scan-in scan-out called LSSD (level-sensitive scan design) was developed at IBM which uses level-sensitive dual-clocked flip-flops in place of the D flip-flops of Figure 8b [24]. This results in circuits that are largely free of timing problems like hazards. The increase in the number of gates required to implement this scheme was found to range from 4% to 20%. LSSD is used in the design of the IBM System/38 computer [10].

6 SELF-TESTING SYSTEMS

So far, our discussion has been concerned with external testing methods, where the bulk of the test equipment is not part of the UUT. As digital systems grow more complex and more difficult to test, it becomes increasingly attractive to build test procedures into the UUT itself. Some self-testing ability is incorporated into most computers, mainly via coding techniques. A few machines have

been designed with essentially complete self-testing, notably telephone switching systems [23] and spaceborne computers [5]. In the last few years comprehensive self-testing features have become common in microprocessor-controlled instruments such as logic analyzers [25]. The Commodore PET, a personal computer based on the 6502 microprocessor, is delivered with a self-testing program which is considered to have sufficient fault coverage to serve as the sole go/no-go test used in manufacture [22].

Self-testing can be achieved using any of the techniques described in Sec. II. Often these methods are combined in various ways. For example, No. 1 ESS, an early telephone switching system with a high degree of self-testability, uses duplication with comparison (match) circuits, error-detecting codes, self-checking circuits, and diagnostic programs that are executed by No. 1 ESS itself [23]. In self-testing microprocessor-based systems, on-line or real-time self-testing procedures are generally the most practical. Comparison testing can be implemented by duplicating key subsystems, and comparing their output signals. Currently, such duplication is usually done at the board level, and individual ICs are duplicated. However, it is also possible to duplicate components within an IC, an approach that can take advantage of the high component densities of VLSI chips [14, 62]. Algorithmic self-testing can be achieved by having the microprocessor execute appropriate self-test programs. The use of special codes and checking circuits can be regarded as a form of hardware-implemented self-testing at a somewhat lower level.

6.1 Coding Techniques

Coding techniques for error detection have the great advantage that they provide a precisely-defined level of fault coverage with little overhead in extra hardware or processing time. Codes exist which can give almost any desired level of error detection or correction, although implementation costs generally increase with the fault coverage desired [54, 69].

The most widely used error-detecting and -correcting codes are the parity check codes. In these codes, a set of (parity) check bits are appended to the data item being tested (the information bits) to form a codeword. The covered faults cause a codeword to change to a non-codeword, an event that can be detected by a logic circuit called a checker. For serial data, a special class of parity check codes called cyclic codes are often used. These codes have the advantage of being easy to encode and decode using linear feedback shift registers of the type appearing in Figure 2b. Parity check codes are generally unsuitable for testing functional units; they are

mainly used for checking data transmission and storage devices. Special codes have been developed for some types of functional units, particularly arithmetic units [57].

6.2 Hardware-Implemented Self-Testing

The advantages of error coding techniques can be extended to general logic circuits by designing them to be self-checking [3, 4, 15, 69]. A self-checking logic circuit is one whose output responses constitute an error-detecting code. The encoded outputs may then be checked to detect errors as illustrated in Figure 9. A circuit C is formally defined to be self-checking if it has the following two properties [3]. Assume that we are concerned with detecting faults of a specified type F in C, and that all inputs of C are drawn from a set I of input patterns.

1. If any specified fault is present, the output of C is a non-codeword for at least one pattern in I (this is called the "self-testing" property).

2. If any specified fault is present, no member of I produces an output from C that is a codeword (fault secureness property). Note that if the input set I includes all patterns that are applied to C during normal operation, then C has the desirable property of being tested automatically during normal operation.

A variety of techniques for designing self-checking circuits are known, many of which are quite practical [69] . Indeed it is feasible to build a complete computer in which all testing is done in the hardware by self-checking circuits and similar coding techniques. A study by Carter et al. [16] shows that the cost of making a large computer self-testing in this way is relatively low using current LSI technology. They found that complete self-testing could be achieved for a System/360-type computer with an increase of less than 10% in the number of components used. The design for a VLSI fault tolerant computer proposed by Sedmak and Liebergot [62] also makes extensive use of self-checking circuits.

A very different approach to self-checking hardware design is the use of on-chip electrical monitors [20, 613. This technique, which has been applied to ECL-type LSI chips, uses special electrical circuits that can detect small changes in parameters such as current or resistance. A monitor circuit is typically connected to each IO line of the chip, and the combined output signals from the on-chip monitors are connected to an extra output pin. On-chip monitors of this kind are primarily intended to detect short-circuits, open circuits and similar interconnection faults. This promising testing method is new, and has seen little application so far.

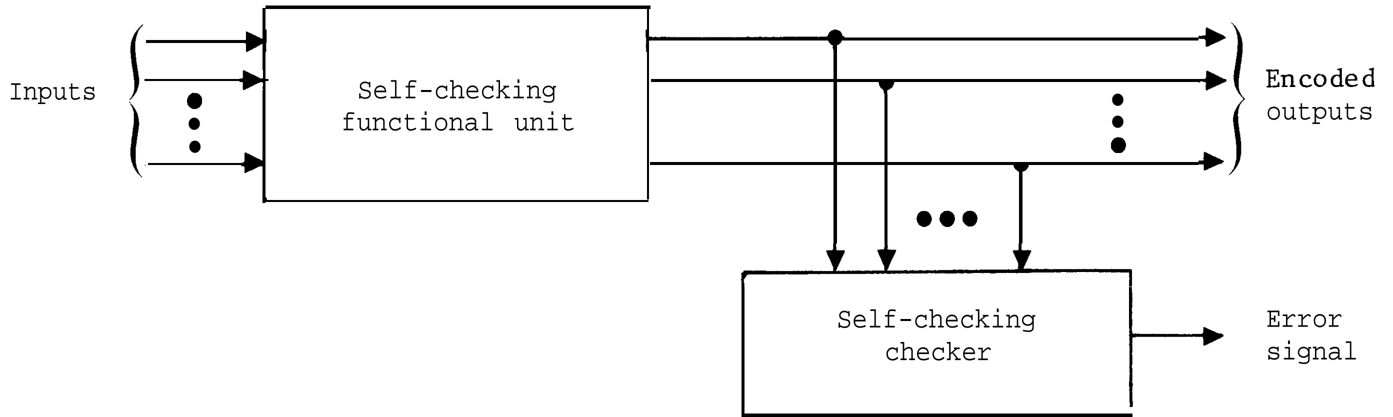


Fig. 9. A self-checking logic circuit.

It is also possible to make an IC self-testing by building into it all the circuitry required for a compact testing technique like signature analysis. As Figure 2 indicates, a relatively small amount of extra logic suffices, basically a counter for test pattern generation, and a feedback shift register for signature generation. The fault-free signatures may be stored in a ROM on the chip for comparison with the signatures produced during testing. An experiment simulating this approach using a modified version of the AND 2901 microprocessor slice is described by Zweihoff et al. [71].

6.3 Programmed Self-Testing

Although it is feasible to rely entirely on hardware checking circuits for self-testing, it is often more economical to use self-testing software, especially when off-the-shelf components with little or no built-in checking circuitry are used. The heuristic test programs for microprocessor-based systems discussed in Sec. V can readily be modified for self-testing. The role of the external tester is taken over by the microprocessor under test. Thus the microprocessor is responsible not only for executing the test programs, but also for scheduling their execution and interpreting the test results.

In self-testing systems, test program execution is usually interleaved with normal program execution, and is designed to interfere with the latter as little as possible. Test program execution is generally initiated at the following times.

1. When a fault is detected during normal operation, e.g., a parity check fails.

2. At regularly scheduled intervals.

3. At times when the system might otherwise be idle.

Several courses of action are possible when an error is detected. In the simplest case, the system sends an error signal to the outside world and halts. Alternatively, the system may be programmed to attempt to recover automatically by, for example, repeating the instructions being executed when the fault occurred. Little hardware or software is needed to implement this recovery scheme which is called program retry. It is mainly useful in the case of temporary or intermittent faults. Recovery from permanent faults can be attained by including spare or redundant units in the system, as well as mechanisms to switch from faulty to spare units. Recovery of this kind has been implemented in large fault-tolerant systems [5, 23]. We conclude with an example of a microprocessor system designed to achieve a high-level of self- testing and some fault tolerance at very low cost.

Example 4: A self-testing microprocessor-based system [31]

This machine, developed at E-Systems Inc., was designed as a communications controller. The system includes a CPU, ROMs, RAMs and IO interface circuits, all of which are tested automatically by a self-test program. This program is stored in a 1K-byte ROM within the CPU itself. It is executed in background mode, being invoked during normal processing by a low-priority interrupt signal. Figure 10 shows the organization of the self-test program. All major subsystems are tested in sequence, starting with the CPU. Detection of a fault causes an indicator light to be turned on in an LED display panel.

The CPU structure is shown in Figure 11. It contains two microprocessors, one of which serves as a standby spare in the event of failure of the active (controlling) microprocessor. The active microprocessor is required to access and reset a timer T at regular intervals. Failure to do so causes a time-out circuit to transfer control of the system to the back-up microprocessor, and turn on the CPU fault light. If the back-up microprocessor is working properly, it subsequently resets T causing the fault indicator to be turned off.

The memory and IO circuits are tested using the general approaches discussed earlier. The ROMs are tested by accessing a block of words from the ROM and summing them in the CPU. The accumulated word is then compared to a check word stored in the ROM. If they differ, the corresponding ROM fault indicator is switched on. If desired, the ROM status can be written into RAM , thus allowing the system to identify and bypass the faulty block in the ROM. This enables the system to operate even with a ROM fault present.

RAMs are tested in the following way. Each RAM location X is read in turn and its contents are saved in a CPU register. Then the two checkerboard patterns are applied to X in the standard way. If the test is passed, the original contents of X are restored from the temporary register, and the next RAM word is tested.

IO tests are performed using the loop-back procedure, whereby output ports are connected to input ports one at a time under CPU control. Test patterns are transmitted through the resulting closed data path and checked for accuracy.

6.4 Multimicroprocessor Systems

If a system contains a number of microprocessors in the form of a multiprocessor or computer network, then it may be possible -

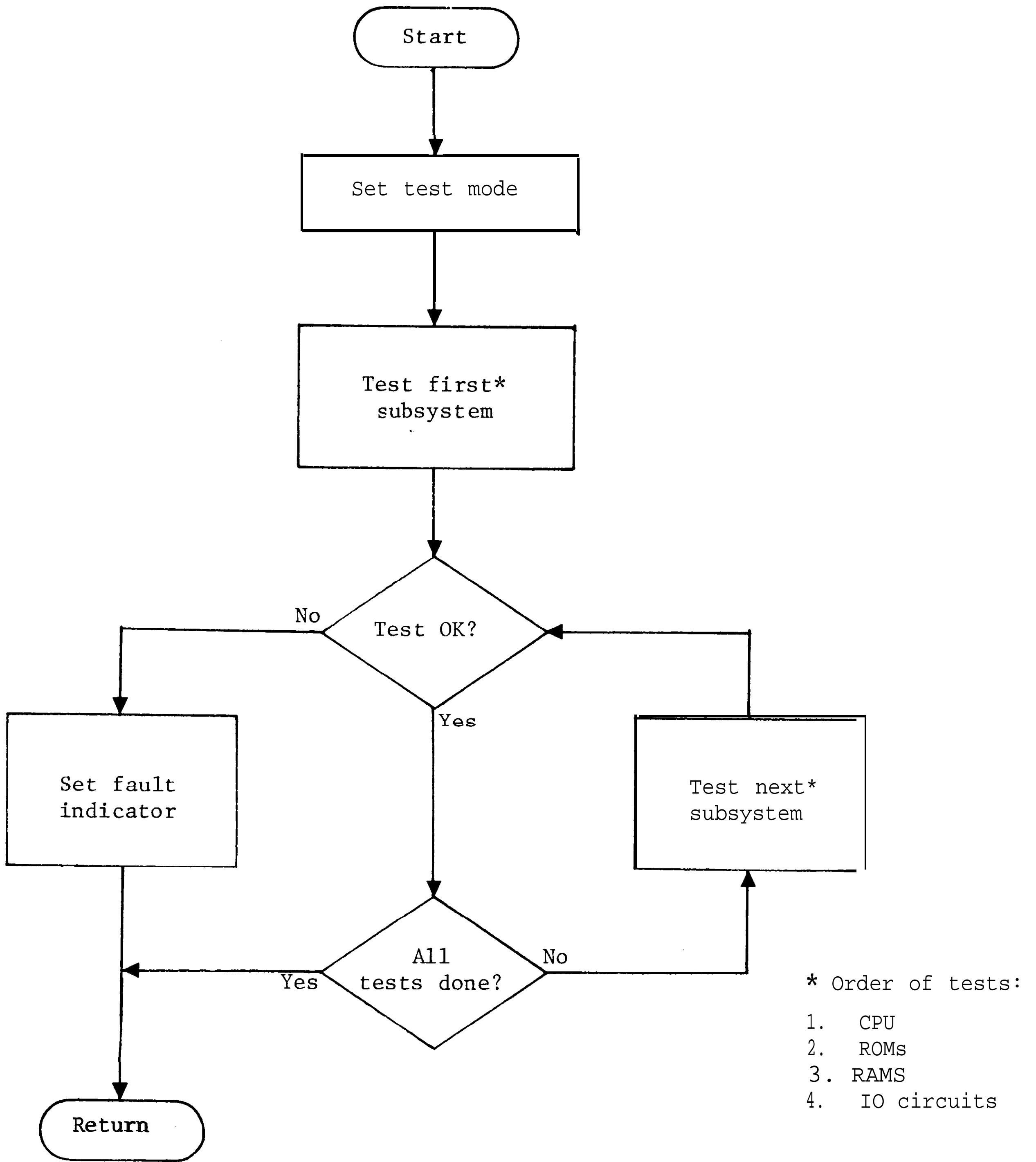


Fig. 10. Flowchart of a microprocessor system self-test program.

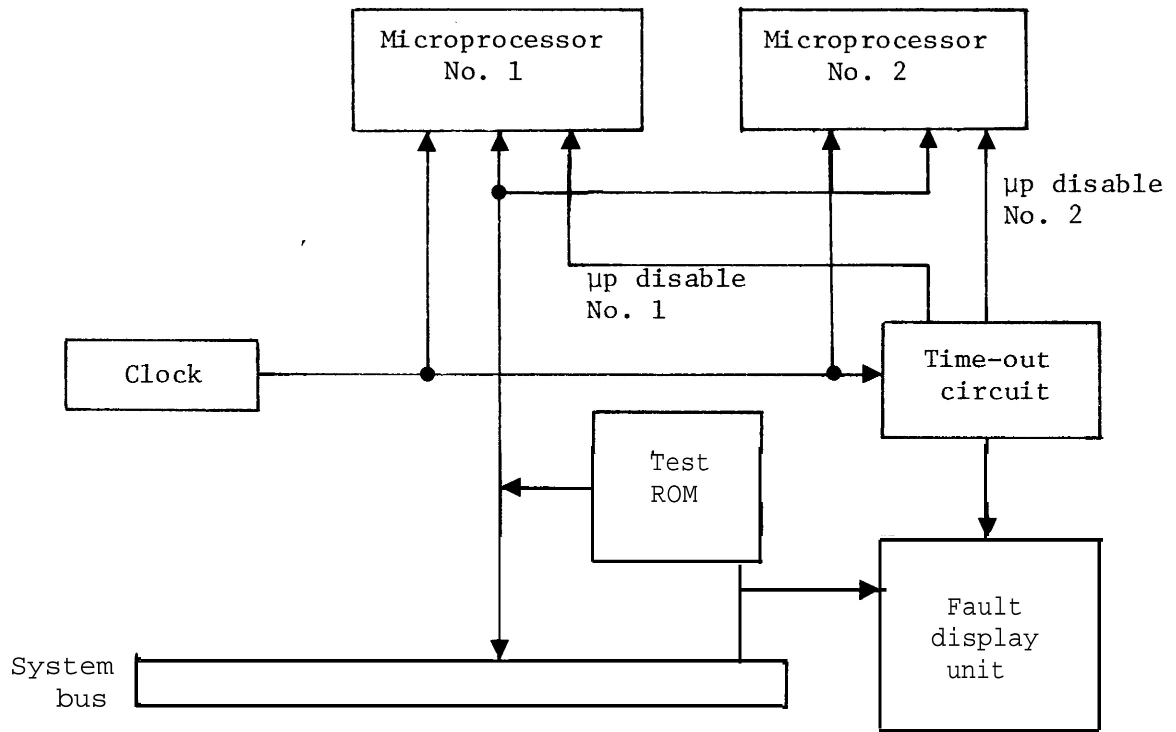


Fig. 11. CPU of a self-testing system.

and advantageous - to use the microprocessors to test one another. This approach to self-testing is employed in such fault-tolerant computers such as the UC-Berkeley PRIME system [8]. Although relatively few self-testing systems of this type have been built, some interesting relevant theory has been developed. Much of this theory is concerned with measuring system self-testability by means of graph-theoretical models [2,12,55,60]. In the widely studied Preparata-Metze-Chien model [13,55], the system is represented by a diagnostic graph containing a node for each subsystem that is capable of testing other subsystems. An arc goes from node S(i) to node S(j), if subsystem S(i) is capable of detecting all faults of interest in subsystem S(j). This arc is labeled with a boolean variable a(i,j) which is set to 1 (0) if S(i) diagnoses S(j) as faulty (fault-free). The set of values of all the a(i,j) variables constitutes a syndrome. Figure 12 shows a 2-node system with the syndrome (1,0). The extent to which faults can be located from the possible syndromes can be used as a measure of system self-diagnosability. A central difficulty with this type of analysis is that faults outside the subsystem being tested, e.g., a fault in the subsystem performing the test, may result in an indication of a fault where none exists, and vice versa. Thus the syndrome (1,1) for the system of Figure 12 indicates that S(1), S(2) or both are faulty.

The applicability of models of the foregoing type to existing systems is limited, mainly due to the fact that the testing function in most current systems is highly centralized. This situation is likely to change as multimicroprocessor systems become more common, allowing a high-level testing capability to be distributed throughout a system. An application of diagnostic graphs to a hypothetical bit-sliced microprocessor system is described by Ciompi and Simoncini [19].

7 REFERENCES

1. Advanced Micro Devices: The Am2900 Family Data Book, Sunnyvale, Calif., 1976.

2. P. w. Agnew, R. E. Forbes and C. B. Stieglitz: "An Approach to Self-Repairing Computers," Digest 1st Ann. IEEE Computer Conf., Chicago, Sept. 1967, pp. 60-63.

3. D. A. Anderson: Design of Self-Checking Digital Networks Using Coding Techniques," Tech. Rept. R-527, Univ. of Illinois, Coordinated Sci. Lab., Urbana, 1971.

4. D. A. Anderson and G. Metze: "Design of Totally Self-Checking Check Circuits for m-out-of-n Codes," IEEE Trans. Computers, Vol. C-22, March 1973, pp. 263-269.

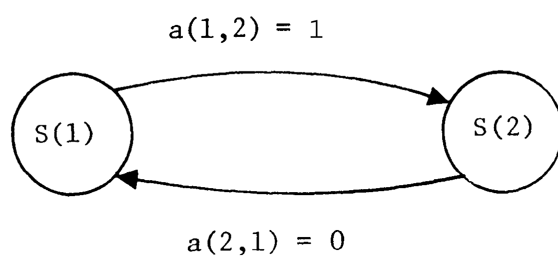


Fig. 12. A simple diagnostic graph.

5. A. Avizienis et al.: "The STAR (Self-Testing and Repairing) Computer: an Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Trans. Computers, Vol. c-20, Nov. 1971, PP. 1312-1321.

6. P. Azema, A. Lozes and M. Diaz: "Test Point Placement for Combinational Circuits," Digital Processes, Vol. 3, 1977, PP. 227-235.

7. W. Barraclough, A. C. L. Chiang and W. Sohl: "Techniques for Testing the Microprocessor Family," Proc. IEEE, vol. 64, June 1976, PP. 943-950.

8. H.B. Baskin, B. R. Borgerson and R. Roberts: "PRIME - a Modular Architecture for Terminal-Oriented Systems, AFIPS Conf. Proc., vol. 40, Spring 1972, pp. 431-437.

9. R.G. Bennetts and R.V. Scott: "Recent Developments in the Theory and Practice of Testable Logic Design," Computer, Vol. 9, No. 6, June 1976, pp. 47-63.

10. N.C. Berglund: "Level-Sensitive Scan Design Tests Chips, Boards, System," Electronics, Vol. 52, No. 6, March 15, 1979, PP. 108-110.

11. S.Bisset: "LSI Tester Gets Microprocessors to Generate Their Own Test Patterns," Electronics, Vol. 51, No. 11, May 25, 1978, PP. 141-145.

12. M. Blount: "Modeling and Diagnosis of Fail-Softly Computer Systems," Digest 8th Int. Conf. on Fault-Tolerant Computing, Toulouse, June 1978, pp. 53-58.

13. M.A. Breuer and A. D. Friedman: Diagnosis and Reliable Design of Digital Systems, Woodland Hills, Calif., Computer Science Press, 1976.

14. R. P. Capece: "Tackling the Very Large Scale Problems of VLSI," Electronics, Vol. 51, No. 24, Nov. 23 1978, pp. 111-125.

15. W.C. Carter and P. R. Schneider: "Design of Dynamically Checked Computers," Proc. IFIP Congress, Edinburgh, 1968, Vol. 2, pp. 878-883.

16. W. C. Carter et al.: "Cost Effectiveness of Self-Checking Computer Design," Proc. 7th Int Conf. on Fault-Tolerant Computing, Los Angeles, June 1977, pp. 117-123.

17. H. Y. Chang and G. W. Heimbigner: "LAMP: Controllability, Observability and Maintenance Engineering Techniques," Bell Sys. Tech. J., Vol. 53, Oct. 1974, pp. 1505-1534.

18. A.C.L. Chiang and R. McCaskill: "Two New Approaches Simplify Testing of Microprocessors," Electronics, Vol. 49, No. 2, Jan. 22 1976, pp. 100-105.

19. P. Ciompi and L. Simoncini: "Design of Self-Diagnosable Mini-Computers Using Bit Sliced Microprocessors," J. Design Autom. and Fault-Tolerant Computing, Vol. 1, Oct. 1977, pp. 363-375.

20. F. B. D'Ambra et al.: "On Chip Monitors for System Fault Isolation," Digest 1978 Int. Solid State Circuits Conf., Feb. 1978, pp.218-219.

21. S. DasGupta, E. B. Eichelberger and T. W. Williams: "LSI Chip Design for Testability," Digest 1978 Int. Solid-State Circuits Conf., Feb. 1978, pp. 216-217.

22. E.S. Donn and M. D. Lippman: 'Efficient and Effective uC Testing Requires Careful Planning," EDN, Vol. 24, No. 4, Feb 2 1979, pp. 97-107.

23. R. W. Downing, J. S. Novak and L. S. Tuomenoksa: "No. 1 ESS Maintenance Plan," Bell Sys. Tech. J., Vol. 43, Sept. 1964, pp. 1961-2019.

24. E.B. Eichelberger and T. W Williams: "A Logic Design Structure for LSI Testability," J. Design Autom. & Fault-Tolerant Computing, Vol. 2, May 1978, pp. 165-178.

25. B. Farly: "Logic analyzers aren't all alike," Electronics Design, Vol. 26, No. 3, Feb. 1 1978, pp. 70-76.

26. W. G. Fee: Tutorial: LSI Testing, Long Beach, Calif., IEEE Computer Soc., 1977.

27. Fluke Trendar: "FAULTRACK: Universal Fault Isolation Procedure for Digital Logic," Bulletin 122, Mountain View, Calif. 1973.

28. A.D. Friedman: "Easily Testable Iterative Systems," IEEE Trans. Computers, Vol. C-22, Dec. 1973, pp. 1061-1064.

29. A. D. Friedman and P. R. Menon: Fault Detection in Digital Circuits, Englewood Cliffs, NJ, Prentice-Hall, 1971.

30. H. Fujiwara and K. Kinoshita: "Testing Logic Circuits with Compressed Data," Proc. 1978 Int. Conf. on Fault-Tolerant Computing, Toulouse, June 1978, pp. 108-113.

31. D.P. Fulghum: "Automatic Self-Test of a Micro-Processor System," Proc. AUTOTESTCON '76, Arlington, Texas, Nov. 1976, pp. 47-52.

32. S. Funatsu, N. Wakatsuki and A. Yamada: "Designing Digital Circuits with Easily Testable Consideration," Digest 1978 Semicond. Test Conf., Cherry Hill, NJ, Oct. 1978, pp. 98-102.

33. G. Gordon and H. Nadig: "Hexadecimal Signatures Identify Troublespots in Microprocessor Systems," Electronics, Vol. 50, No. 5, Mar. 3 1977, pp. 89-96.

34. D. Hackmeister and A. C. L. Chiang: "Microprocessor Test Technique Reveals Instruction Pattern Sensitivity," Computer Design, Vol. 14, No. 12, Dec. 1975, pp. 81-85.

35 J.P. Hayes: "On Modifying Logic Circuits to lmprove Their Diagnosability," IEEE Trans. Computers, Vol. C-23, Jan. 1974, pp. 56-62.

36 J. P. Hayes; "Detection of Pattern Sensitive Faults in Random Access Memories,** IEEE Trans. Computers, Vol. C-24, Feb. 1975, pp. 150-157.

37. J. P. Hayes: "Transition Count Testing of Combinational Logic Circuits," IEEE Trans. Computers, Vol. C-25, June 1976, pp. 613-620.

38. J. P. Hayes: "Check Sum Methods for Test Data Compression," J. Design Autom. and Fault-Tolerant Computing, Vol. 1, Oct. 1976, pp. 3-17.

39. F. C. Hennie: "Fault Detecting Experiments for Sequential Circuits," Proc. 5th Ann. Symp. on Switching Theory and Logical Design, Nov. 1964, pp. 95-110.

40. Hewlett-Packard: "Designing Digital Circuits for Testability," Applic. Note 210-4, Palo Alto, Calif. Jan. 1977.

41. Hewlett-Packard: "A Designer's Guide to Signature Analysis," Applic. Note 222, Palo Alto, Calif. April 1977.

42. E. R. Hnatek: "4-kilobit Memories Present a Challenge to Testing," Computer Design, Vol. 14, No. 5, May 1875, pp. 117-125.

43. Intel: Intel 8080 Microcomputer Systems User's Manual, Santa Clara, Calif. Sept. 1975.

44. B. Kline, M. Maerz and P. Rosenfeld: "The In-Circuit Approach to Microcomputer Based Products," Proc. IEEE, Vol. 64, June 1976, pp. 937-942.

45. R. J. Leaman, H.H. Lloyd and C. S. Repton: "The Development and Testing of a Processor Self-Test Program," Computer J., Vol. 16, 1973, pp. 308-314.

46. J. Losq: "Efficiency of Random Compact Testing," IEEE Trans. Computers, Vol. C-27, June 1978, pp. 516-525.

47. J. Mancone: "Testability Guidelines," Electronics Test, Vol. 2, No. 3, March 1979, pp. 14-16.

48. R. A. Marlett: "On the Design and Testing of Self-Diagnosable Computers," Digest 1st Ann. IEEE Computer Conf., Chicago, Sept. 1967, pp. 14-15.

49. R. McCaskill: Test Approaches for Four Bit Microprocessor Slices," Digest 1976 Semicond. Test Symp., Cherry Hill, NJ, Oct. 1976, pp. 22-24.

50. E. J. McCluskey: "Design for Maintainability and Testability," Proc. Govt. Microcircuits Applic. Conf. (GOMAC), Monterey, Calif., Nov. 1978, pp. 44-47.

51. R. Nair, S. M. Thatte and J. A. Abraham: "Efficient Algorithms for Testing Semiconductor Random-Access Memory," IEEE Trans. Computers, Vol. C-27, June 1978, pp. 572-576.

52. D.L. Ostapko and S. J. Hong: "Fault Analysis and Test Generation for Programmable Logic Arrays," Digest 8th Int. Conf. on Fault-Tolerant Computing, Toulouse, June 1978, pp. 83-89.

53. K. P. Parker: "Compact Testing: Testing with Compressed Data," Proc. 6th Int. Conf. on Fault-Tolerant Computing, Pittsburgh, June 1976, pp.93-98.

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

55. F. P. Preparata, G. Metze and R. T. Chien: "On the Connection Assignment Problem of Diagnosable Systems," IEEE Trans. Electronic Computers, Vol. EC-16, Dec. 1967, pp. 848-854.

56. G. F. Putzolu and J. P. Roth: "A Heuristic Algorithm for the Testing of Asynchronous Circuits," IEEE Trans. Computers, Vol. C-20, June 1971, pp. 639-647.

57. T. R. N. Rao: Error Coding for Arithmetic Processes, New York, Academic Press, 1974.

58. S. M. Reddy: "Easily Testable Realizations for Logic Circuits," IEEE Trans. Computers, Vol. C-21, Nov. 1972, pp. 11851188.

59. J. P. Roth: "Diagnosis of Automata Failures: a Calculus and a Method," IBM J. Res. and Dev., Vol. 10, 1966, pp. 278-291.

60. J. D. Russell and C. R. Kime: "System Fault Diagnosis: Masking, Exposure, and Diagnosability without Repair," IEEE Trans. Computers, Vol. C-24, 1975, pp. 1155-1161.

61. S. H. Sangani and B. Valitski: "In-situ Testing of Combinational and Memory Circuits Using a Compact Tester," Proc. 8th Int. Conf. on Fault-Tolerant Computing, Toulouse, June 1978, p.214.

62. R. M. Sedmak and H. L. Liebergot: "Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," Proc. 8th Int. Conf. on Fault Tolerant Computing, Toulouse, June 1978, pp. 137-143.

63. D. H. Smith: "Exercising the Functional Structure Gives Microprocessors a Real Workout," Electronics, Vol. 50, No. 4, Feb. 17, 1977, pp.109-112.

64. T. Sridhar and J. P. Hayes: "Testing Bit-Sliced Microprocessors," Proc. 9th Int. Conf. on Fault-Tolerant Computing, Madison, June 1979, pp.211-218.

65. A. Stefanski: "Free Running Signature Analysis Simplifies Troubleshooting," EDN, Vol. 24, No. 3, Feb. 5 1979, PP. 103-105.

66. J. E. Stephenson and J. Grason: "A Testability Measure for Register Transfer Level Digital Circuits," Proc. 6th Int. Conf. on Fault-Tolerant Computing, Pittsburgh, June 1976, pp. 101-107.

67. S. M. Thatte and J. A. Abraham: "User Testing of Microprocessors," Proc. 18th IEEE Computer Soc. Int. Symp. (Spring COMPCON 79), San Francisco, Feb./March 1979, pp. 108-114.

68. S. M. Thatte and J. A. Abraham: "A Methodology for Functional Level Testing of Microprocessors," Proc. 8th Int. Conf. on Fault-Tolerant Computing, Toulouse, June 1978,pp. 90-95.

69. J. Wakerly: Error Detecting Codes, Self-Checking Circuits and Applications, New York, American Elsevier, 1978.

70. M. J. Y. Williams and J. B. Angell: "Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic,** IEEE Trans. Computers, Vol. C-22, Jan. 1973, pp. 46-60.

71. G. Zweihoff, B. Koenemann and J. Mucha: "Experimente mit einem Simulationsmodell fuer selbst-testende IC's," NTG-Fachberichte,. Band 68, April 1979, pp. 105-108.