EASILY TESTABLE BIT-SLICED DIGITAL SYSTEMS

T. SRIDHAR

DIGITAL INTEGRATED SYSTEMS CENTER REPORT
DISC/81-4

DEPARTMENT OF ELECTRICAL ENGINEERING—SYSTEMS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CALIFORNIA 90007

OCTOBER 1981

This research was supported in part by the Joint Services Electronics Program under an AFOSR Contract No. F49620-81-C-0070 and by the Naval Electronics Systems Command under Contract No. N00039-80-C-0641 (VHSIC Program).

DEDICATION

To my parents

ACKNOWLEDGEMENTS

It gives me great pleasure to acknowledge numerous supports received by me during the course of my research work. It is very difficult to express in simple words the help and support provided by Professor John P. Hayes. I am very grateful for his excellent guidance and constant encouragement. His invaluable constructive criticisms have greatly enhanced the quality and presentation of this thesis. I would like to thank Professor Melvin A. Breuer and Professor Dennis R. Estes for serving on my dissertation committee.

I am also thankful to many of my friends and colleagues in the Department of Electrical Engineering and elsewhere for their help throughout my graduate study. The financial assistance received from the Department of Electrical Engineering, the Joint Services Electronics Program and the International Business Machines Corporation in the form of teaching assistantship, research assistantship and graduate fellowship, respectively, is gratefully acknowledged.

Finally, I am also grateful to Kristin Pendleton for her excellent typing job and understanding.

ABSTRACT

Bit-sliced systems are an important class of (complex) digital systems that have a regular array-like structure, which is particularly attractive for very large-scale integration (VLSI) technology. A bit-sliced system is realized by interconnecting identical slices or cells to form a one-dimensional iterative logic array (ILA). In this thesis the design of bit-sliced systems that are easily testable is investigated. First an analytic test generation methodology for bit-sliced and related systems is developed. For this purpose a high-level (register-level) circuit model and a corresponding functional fault model are defined. A 1-bit processor slice C having the main features of commercial slices is defined as a test case. Using the high-level circuit and fault models a technique for deriving a complete and near-minimal length test sequence for C is presented. The cell C is then extended to form general k-bit slices whose internal structures more closely resemble commercial products. It is shown that test patterns for an array of N identical processor cells can be easily derived from the tests for an individual cell. Furthermore, the number of test patterns needed

for the processor array is independent of the array length. It is therefore concluded that for test generation purposes, bit-sliced processors can be usefully modeled as C-testable ILAs, which require a constant number of test patterns independent of array size.

The property of C-testability in one-dimensional ILAs is studied in detail. Basic concepts of C-testability in unilateral combinational arrays are discussed first. C-testable arrays are characterized and procedures to construct test patterns for such arrays are given. Design methods to make an ILA C-testable are proposed. C-testable arrays of bilateral and sequential cells are analyzed next. A characterization of C-testable bilateral combinational ILAs is presented, and a design modification scheme to make a bilateral ILA C-testable is given. It is shown that the results on C-testable combinational arrays can be applied directly to a useful class of sequential arrays.

A new testability criterion for ILAs called I-testability is introduced. I-testability ensures that identical test responses can be obtained from every cell of an ILA, and thus simplifies response verification. I-testable combinational ILAs are characterized, as are CI-testable ILAs that are simultaneously C- and I-testable. A design scheme for making an arbitrary ILA CI-testable

is also presented. Finally, a design technique for realizing self-testing bit-sliced computers based on I-testing is developed. It is established that the family of processor arrays constructed of cell C and its extensions is CI-testable. Using CI-testable processor arrays and other I-testable bit-sliced circuits, a self-testing computer is designed. The advantages and limitations of the proposed design are discussed, and compared to more conventional self-checking approaches that are based on coding techniques.

CHAPTER 1

INTRODUCTION

The problems addressed in this thesis are defined here and the reasons for considering them are discussed. The goals of the thesis are outlined and a brief summary of the contents of each chapter is given.

1.1 Background

The testing of digital systems is concerned with the detection and location of hardware failures. It involves the application of appropriate test signals to the input lines of a system under test, and the verification of the response signals appearing at the system's output lines. The advent of large-scale integration (LSI) technology has greatly increased the complexity of digital integrated circuits (ICs). This increase in complexity has made testing ICs more difficult and expensive. The need for efficient low-cost testing methods has grown rapidly in recent years.

Conventional testing methods have two serious limitations. (1) They only recognize low-level devices such as gates and flip-flops as primitive elements. (2) They use low-level fault models such as the line stuck-at-zero/

one (s-a-0/1) model, which associates faults with all lines interconnecting the primitive elements. In systems containing tens of thousands of gates and lines, the amount of computation required to construct a comprehensive test set can be enormous. Furthermore, a gate-level description of the system components may be unavailable to the test designers. Manufacturers of devices such as microprocessors often only supply designers with register or function-level circuit diagrams.

Relatively little work has been done on function-level testing methods [McCaskill 1976, Chiang and McCaskill 1976, Breuer and Friedman 1980, Thatte and Abraham 1980], which are applicable to LSI elements like microprocessors. In practice, heuristic test generation methods are usually employed for microprocessor-based systems. Typically, test programs are written using the instruction set of the microprocessor under test. These programs are designed to exercise in ad hoc fashion the major functional units of the system being tested. Heuristic testing methods of the foregoing kind have two main disadvantages. (1) The fault-coverage they achieve is often low and hard to determine. (2) They may yield excessively large test sets, e.g.,a million or more test patterns for an Intel 8080 8-bit microprocessor [Chiang and McCaskill 1976], while a typical 4-bit bit-sliced microprocessor, the Advanced Micro Devices 2901, was found to require about 12,500 test patterns [McCaskill 1976].

It is well-known that a regular structure, such as a linear one-dimensional cascade or array, facilitates fault diagnosis. The testing problems for arrays of simple components implemented using small-scale integration (SSI) have been extensively studied [Kautz 1967, Friedman and Menon 1971]. Bit-sliced microcomputers [Hayes 1981] are representative of digital systems that are realized by interconnecting complex LSI components in the form of a regular one-dimensional array. A bit-sliced system to process operands of any desired size is built by connecting an appropriate number of identical units, called slices or cells, in cascade. Therefore, for analysis purposes a bit-sliced system can be modeled as an iterative logic array (ILA). Typical examples of bit slices are the microprocessor slices in the 2900 and 3000 microprocessor series [Intel 1976, Advanced Micro Devices 1979]. Because of their great design flexibility, which is in part due to their use in microprogramming, bit-sliced computers have a particularly wide range of applications [Hayes 1981]. The interconnection regularity of bit slicing can also be used to advantage in the design of complex IC chips [Mead and Conway 1980, Clark 1980]. For example, it is used extensively in the design of the data path (processor) chip for the Caltech OM-2 microcomputer [Mead and Conway 1980].

1.2 Problem Statement

The main objective of this thesis is to develop design guidelines for constructing bit-sliced digital systems that are easy to test. An easily testable system is defined here as one with the following attribute. There exists a relatively short and easily generated sequence of tests that can detect all faults of interest.

In order to overcome the two limitations of the conventional test generation methods noted at the beginning, it is necessary to employ high-level circuit and fault models. In this thesis, we propose to treat relatively complex functional units such as registers and multiplexers as primitives, and to test these primitive components for very general functional failures. Using this high-level approach we will examine the following problems.

1. Construction of non-heuristic (analytic) test generation methods that can yield short and complete test sequences.

2. Determination of efficient ways to construct test sequences for a bit-sliced array.

3. Identification and characterization of ILA properties that can be used to enhance testability.

4. Application of desirable ILA properties to practical bit-sliced systems such as bit-sliced computers.

5. Development of design modification schemes that can simplify the testing of individual slices and arrays of slices.

6. Development of new efficient and inexpensive design techniques for constructing self-testing bit-sliced systems.

1.3 Thesis Outline

Bit-sliced systems are formally defined and their salient features discussed in Chapter 2. The organization and structure of a typical bit-sliced central processing unit (CPU) are studied. Some important commercial bit-slice families are discussed and compared. The high-level circuit and fault models which are used throughout this work are introduced. Functional register-level units like registers, multiplexers, arithmetic-logic units and counters, are defined as primitive circuit modules. A functional fault model is introduced which allows a fault to change arbitrarily the function realized by a primitive module, provided the number of internal states is not increased. Thus a complete test sequence for a combinational (sequential) module must verify its entire truth (state)

table. Although the individual modules are tested exhaustively, a network of such modules is tested in an efficient nonexhaustive way.

In Chapter 3 an analytic test generation methodology for bit-sliced systems is developed. A formal model C is defined for a 1-bit microprocessor slice that has the main features of many commercially-available processor slices. Using the above-mentioned high-level circuit and fault models, a complete and near-minimal sequence of tests T_C for C is derived. The basic cell C is then extended to obtain two more general cells C^k and $C^{k,n}$. C^k is a k-bit version of C with provision for high-speed carry circuits within its k-bit arithmetic-logic unit (ALU) module, while $C^{k,n}$ has an additional $n \times$ k-bit scratchpad random access memory (RAM). Test sequences for these cells are also derived.

A family of processor arrays obtained by cascading C, C^k or $C^{k,n}$ cells is analyzed in Chapter 4. It is shown that the test patterns for the individual cells can be extended to obtain test patterns for an array of such cells. It is observed that for test generation purposes bit-sliced processor arrays may be viewed as C-testable ILAs, which require a constant number of test patterns independent of array size. The effects of using a high-speed carry-lookahead scheme between the cells instead of a ripple-carry propagation are also examined.

The property of C-testability in various kinds of one-dimensional iterative logic arrays is studied in detail in Chapters 5, 6, 7 and 8. The basic concepts of C-testability are discussed in Chapter 5 for the case of unilateral ILAs of combinational cells. (An ILA is unilateral if the signal flow between cells is in one direction only.) Chapters 6 and 7 deal with combinational unilateral arrays without and with direct outputs from cells, respectively. Necessary and sufficient conditions for an array to be C-testable are derived. Procedures to construct a set of test patterns, called C-tests, for C-testable ILAs with at most one faulty cell are presented. In the case of arrays with direct cell outputs, it is shown that a complete set of C-tests, obtained under the single-cell fault assumption, can also verify the truth-table of the entire array. In other words, a single fault test set can also detect all (detectable) multiple faults in the array. Thus the test procedure can also verify the validity of the logic design of the array. It is established that an arbitrary array can be made C-testable by the addition of a small amount of logic to each cell. The cell modification schemes proposed here are shown to be better than the existing methods [Friedman 1973, Dias 1976].

In Chapter 8 the basic concepts of C-testability are extended to one-dimensional arrays of bilateral and sequential cells. The results of Chapters 6 and 7 are shown to

be easily extended to bilateral arrays. These results include the characterization of C-testability, and the design modification schemes for achieving C-testability. A class of sequential arrays in which the internal states of cells are directly observable is considered. It is shown that all the results obtained for combinational arrays are directly applicable to this special class.

A new property of ILAs that simplifies test response verification is introduced in Chapter 9. This property, called I-testability, ensures that the test responses from every cell of the array are identical. Hence these identical responses can easily be verified internally using equality checkers. ILAs with the I-testability property are called I-testable, and the test patterns that induce identical responses are called I-tests. I-testable unilateral arrays of combinational cells are characterized. Arrays that can be completely tested with a constant number of I-tests independent of the array size are called CI-testable. CI-testable ILAs have the advantages of both C-testability and I-testability. Again such ILAs are characterized. A design procedure to make any combinational ILA CI-testable is proposed. The foregoing concepts are illustrated by several examples, including ripple-carry adders. It is shown that a ripple-carry adder array is C-testable and I-testable, but not CI-testable.

In Chapter 10 a new built-in testing approach for bit-sliced systems based on I-testing is presented. I-testing involves the application of a complete sequence of I-tests to a bit-sliced array and the verification of the resulting identical responses from every slice. The verification is achieved here using equality checkers. The checkers can themselves be realized as ILAs and hence they can easily be integrated into the arrays being tested. The practical implications of this self-testing approach are examined. It is shown that the processor arrays of C, C^k or $C^{k,n}$ cells discussed in Chapter 4, are all CI-testable. Furthermore, the I-tests for an array are readily obtained from the tests for the individual cells derived in Chapter 3. Another practical bit-sliced device, a microprogram sequencer, is analyzed and found to be I-testable. Using these I-testable processor and microprogram sequencer arrays, a self-testing bit-sliced CPU design is proposed. The test patterns for the bit-sliced arrays are stored in the control store of the CPU. The stored test patterns are applied periodically, and the outputs of the bit-sliced arrays are checked by their respective equality checkers. The non-bit-sliced units of the CPU are duplicated and are also monitored by equality checkers. The proposed CPU design is then compared in detail with a more conventional self-checking CPU designed using coding techniques.

The results presented in this thesis are summarized in the concluding chapter and some suggestions for further research are presented.

CHAPTER 2

SYSTEM MODELS AND TESTING

Digital systems are usually represented as networks of primitive components, where the complexity of the primitives determines the level of modeling being used. For example, if logic gates are used as primitive components, then we obtain a gate-level model. This chapter is concerned with modeling bit-sliced systems at the register level. First bit-sliced systems are formally defined, and the motivations for their use in designing various types of digital systems are discussed. Several commercially-available bit-sliced circuits are examined, and it is observed that they consist of a small number of register-level components interconnected in a simple way. To model such circuits, a register-level network model is introduced that treats components like adders, multiplexers, decoders and registers as primitives. A corresponding functional fault model is then defined. A method is outlined for generating tests using these circuit and fault models.

2.1 Bit-sliced Systems

A digital system S^n is said to be *bit-sliced* if it can be realized as a one-dimensional array or cascade of

N identical modules of type S^k as shown in Figure 2.1. Such an array of identical modules is referred to as an *iterative logic array* or ILA. S^k is a basic module or *(bit) slice* that performs a fixed set of operations I on k-bit operands or data words. The bit-sliced system S^n performs the same set of operations I on Nk-bit words where $n = Nk$. Well-known examples of S^n are the commerical bit-sliced microprocessors in which S^k is a k-bit microprocessor slice, and k is typically 2, 4 or 8 [Hayes 1981]. Other bit-sliced devices include memories, high-speed multiplier/divider chips, shift registers and up-down counters [Advanced Micro Devices 1975].

A bit-sliced system is called a *bit-sliced (micro-) processor* if it can perform the functions of the *execution unit* or E-unit of a computer's central processing unit (CPU). An E-unit consists of the arithmetic-logic circuits and registers required to execute a set of (micro-) instructions. The microinstructions for the E-unit are provided by another part of the CPU, anmely the *instruction unit* or I-unit. To enhance the system flexibility, this I-unit is usually designed to be *microprogrammable.* Microprogrammed control allows the users to define their own sets of microprograms, thereby achieving greater control over the CPU operations than is possible in the case of non-microprogrammed or hardwired machines.

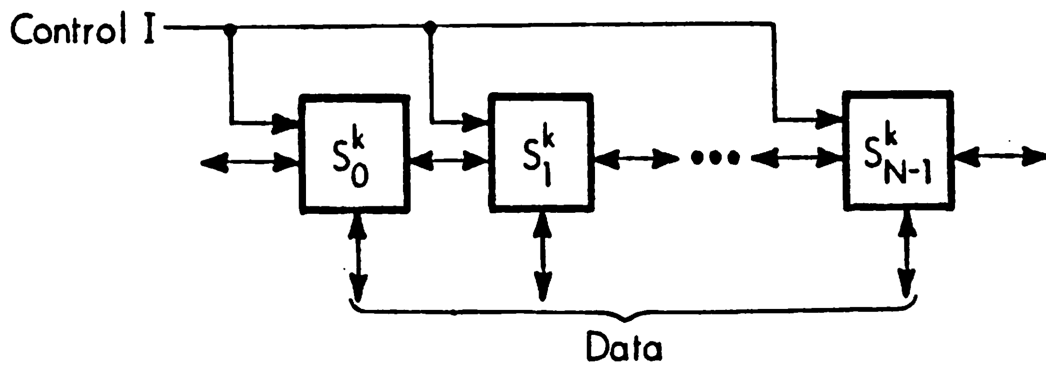


Figure 2.1. General structure of a bit-sliced system S^n composed of N identical k-bit slices S^k

Figure 2.2 shows a block diagram of a typical microprogrammable bit-sliced CPU. It consists of two main units: a bit sliced E-unit and a microprogrammable I-unit. A bit-sliced processor array of identical cells of type P forms the E-unit. The I-unit comprises two parts: a *control memory* for storing microinstructions, and a *microprogram sequencer* for generating the addresses of the microinstructions to be executed by the E-unit. Bit slicing may also be applied to microprogram sequencer design. This can be done by partitioning the microinstruction address into N k-bit words processed by N cascaded slices such as the 2909 slice in the 2900 bit-slice family [Advanced Micro Devices 1979]. Bit-sliced microprogram sequencers are discussed in more detail later in this chapter, as well as in Chapter 10. The system bus of Figure 2.2 is used to connect the CPU to an external main memory and input-output (IO) circuits to form a complete computing system. Such a computer is usually referred to as a *bit-sliced computer*.

Bit-sliced devices became commercially available only in the early 1970s. However, the use of bit slicing can be found in several computers designed as early as 1962. One such computer is the IBM DX-1, an experimental self-diagnosable computer in which the CPU is partitioned into two identical slices each capable of testing the other [Forbes et al. 1965]. Bit-sliced design was also used by Levitt et al. to realize an easily diagnosable and main-

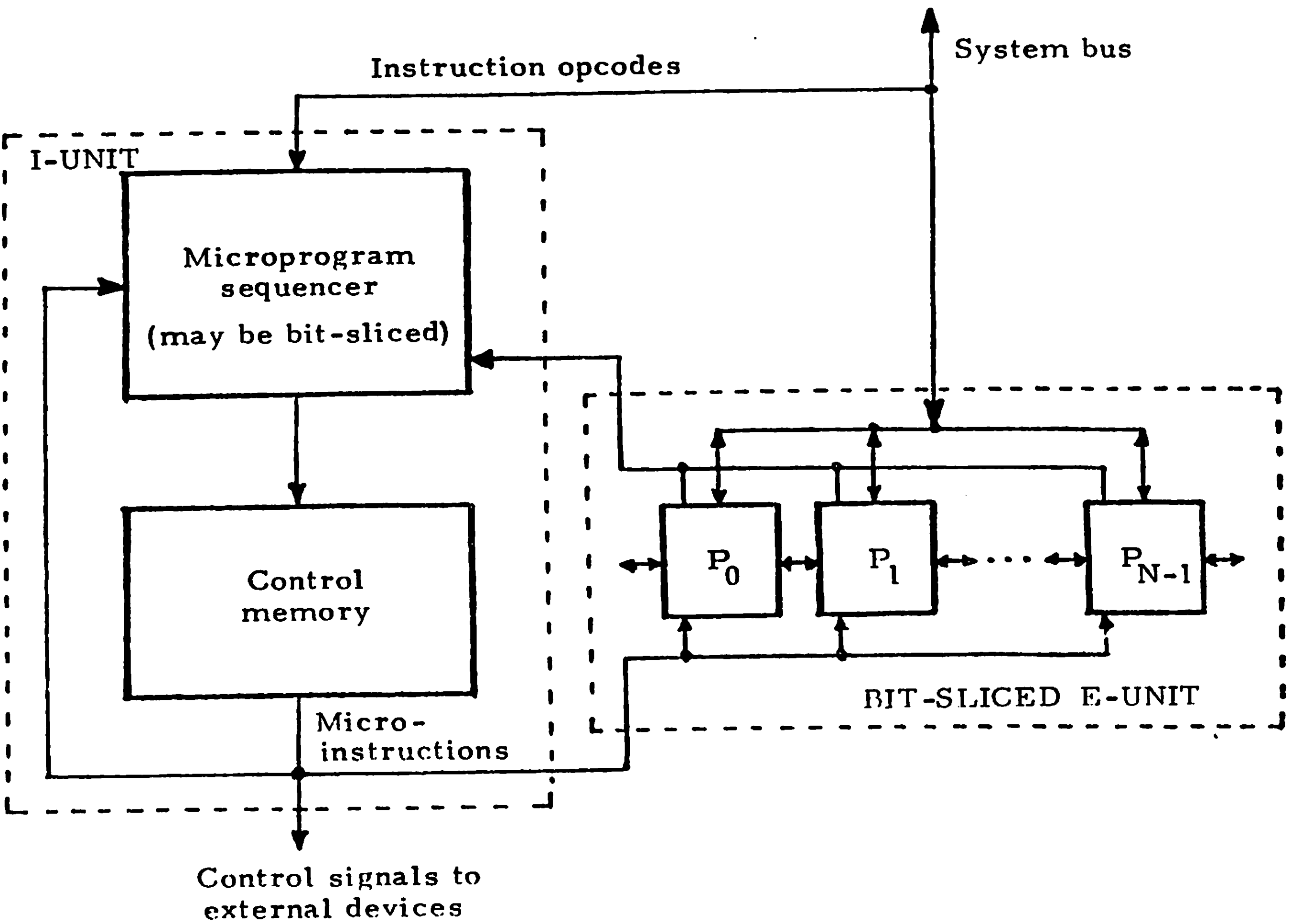


Figure 2.2. Simplified block diagram of a bit-sliced CPU with multi-programmed control

tainable computer [Levitt et al. 1968]. Thus testability was an early motivation for bit-sliced design, although it has received little attention in this context recently. Other motivations include the need for special word sizes, and flexibility in system design. Typical applications of bit-sliced devices include special-purpose processing and control tasks such as real-time signal and image processing, and peripheral device control. Bit-sliced computers are also frequently used to emulate the instruction sets of existing computer families [Hayes 1981].

Bit slicing allows a designer to customize the word size of a system. A typical 4-bit processor slice can be used to build a family of 8-, 16-, or 32-bit processors. These processors execute the same set of microinstructions; thus bit slicing makes it possible to design computers in which the same basic software is executed by processors of different sizes. It should also be noted that the instruction sets of many computers require words of different sizes to be used as operands. Employing a bit-sliced design it is possible to process such operands in a fairly uniform way.

Bit slicing has been successfully used both at the printed-circuit (PC) board level and at the IC chip level. At the board level a bit-sliced component is usually a single IC chip. Such single-chip components are typically

realized by high-speed bipolar technologies such as Schottky TTL and ECL. An important reason for the commercial introduction of bit slicing was to overcome the low gate densities associated with bipolar ICs. With the improvements in IC manufacturing technology that have taken place in recent years, much more sophisticated bit slices with relatively high gate densities are now commercially available. Although the high speeds attainable with bipolar devices is the primary reason for the use of bit-slice components in many applications, the design flexibility offered by bit slicing and microprogramming is also an important consideration.

Recently bit slicing has also been applied to design at the chip level. Integrated circuit layout design is becoming very difficult due to the thousands of active elements being integrated on a single VLSI chip. Bit slicing, due to its structural regularity and simple interconnection requirements, reduces the time needed for chip design. This is clearly demonstrated in two university VLSI chip design projects: the Caltech OM-2 computer [Mead and Conway 1980] and the Stanford Geometry Processor [Clark 1980].

As noted before, the use of uniform components and interconnection structures can simplify testing and maintenance of bit-sliced systems. This has provided motiva-

tion for several early computer design projects [Forbes et al. 1965, Levitt et al. 1968] as well as some recent research on self-testing computers [Ciompi and Simoncini 1977, Wakerly 1978].

2.2 Commercial Bit-slice Families

Microprocessors based on the bit-slice concept were first introduced commercially in 1973 by National Semiconductor. Its GPC/P (General Purpose Controllers/Processors) system design kit is based on a 4-bit PMOS processor slice called a RALU (register and arithmetic-logic unit) [National Semiconductor 1973]. In 1974, Intel introduced the 3000 series which is a family of bipolar bit-slice components [Intel 1976]. Like the 3000 series, most of the subsequent bit-slice families use bipolar instead of MOS technology to achieve higher operating speeds. Also in 1974 the Monolithic Memories 5701/6701 4-bit processor slice was introduced [Wyland 1975]. This was superceded in 1976 by the very similar AMD 2901 slice [Advanced Micro Devices 1979]. Currently the 2900 family is one of the most widely used. Some of the more recent products include the Texas Instruments 481 series, the Motorola 10800 series and the Fairchild 100K series [Adams 1978, Myers 1980].

A typical commercial bit slice family consists of a processor slice, called a bit-sliced microprocessor, a microprogram sequencer, which may also be bit-sliced,

and various support circuits such as memories, interrupt controllers, timers and IO interface circuits. We next examine the key members of five representative bit-slice families: the Advanced Micro Devices 2900, the Intel 3000, the Texas Instruments 481, the Motorola 10800 and the Fairchild 100K series. Of these families, the first three employ Schottky TTL technology, while the last two use very fast ECL technology. We will first discuss in some detail the processor slice and the microprogram sequencer of the 2900 series, and then briefly summarize those of the other four families.

Processor Slices

The 2901 processor slice of the AMD 2900 family is shown in Figure 2.3. It is a 4-bit slice having a 4-bit arithmetic-logic unit (ALU), a two-port 16×4-bit scratchpad random access memory (RAM), two shifters, a general-purpose register T, and various (micro-) instruction decoding circuits. It is capable of performing a set of standard arithmetic and logical operations on the two 4-bit operands R and S. The arithmetic operations include addition, subtraction and negation; while the logical operations include AND, OR, EXCLUSIVE-OR and complementation. The two operands R and S can be obtained via the source multiplexer from several sources: an external input data line D, the 16×4-bit scratchpad RAM, the register T or a

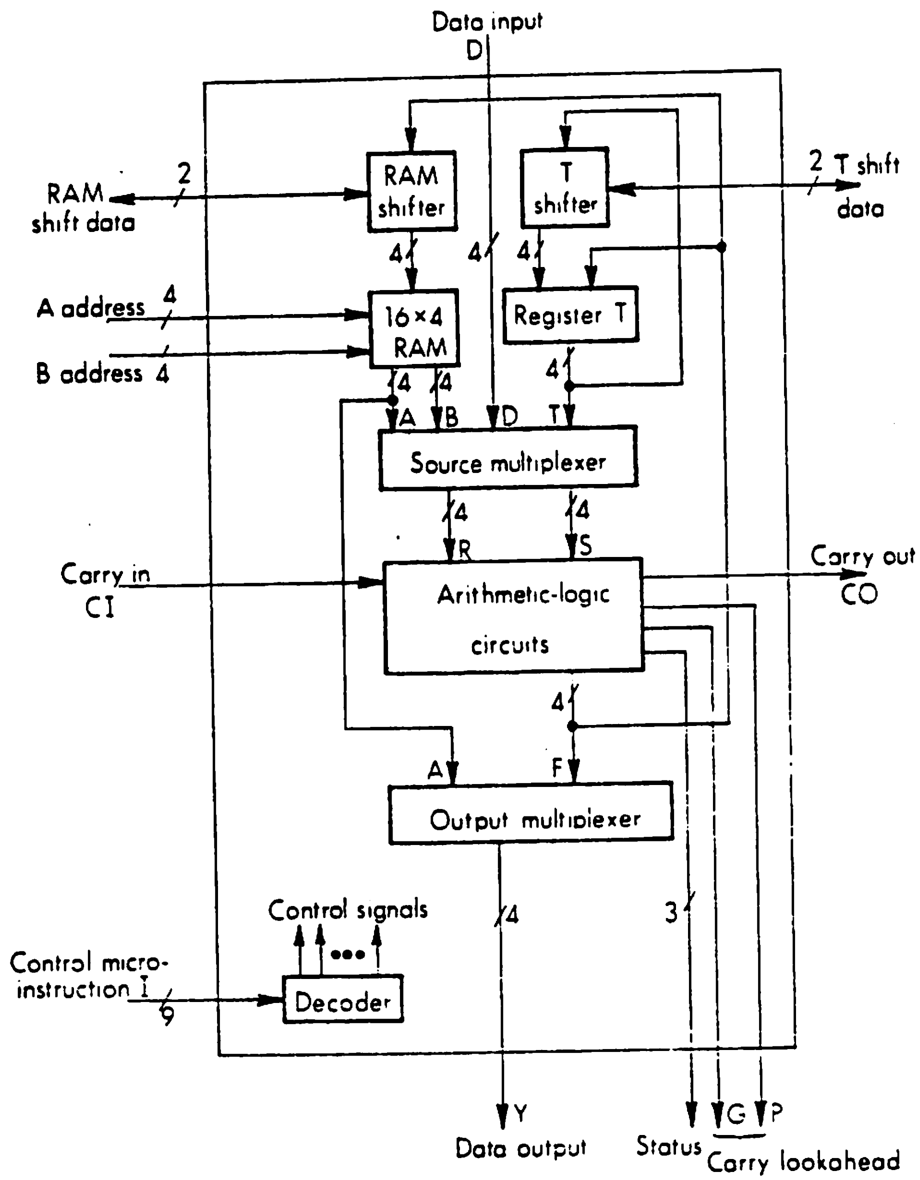


Figure 2.3. Block diagram of the 2901 4-bit processor slice

logical 0. The final result of an operation performed by the arithmetic-logic circuits can be transferred to the scratchpad RAM, the register T or the output data line Y. Shift operations (both right and left) can also be performed on the data before writing into the scratchpad or the register T. The 2901 generates several status or flag signals including sign, overflow, and zero-result signals.

A bit-sliced processor based on the 2901 has the ILA structure depicted in Figure 2.4. To allow arithmetic operations to be extended to operands of arbitrary length, neighboring cells communicate via carry (borrow) signals. Each cell generates a carry output signal CO which is connected to the carry input line CI of the cell on its right. This allows ripple carry propagation through the entire array. Similar left-shift and right-shift connections between the adjacent cells allow shift operations to take place across the ILA. No communication between cells is needed by the logical operations.

In addition to the ripple-carry signal CO, the 2901 has circuits which produce two signals called carry generation (G) and carry propagation (P). G and P are used to implement a scheme called carry lookahead which can be used in place of ripple-carry propagation to speed up arithmetic operations. Carry lookahead requires the use of

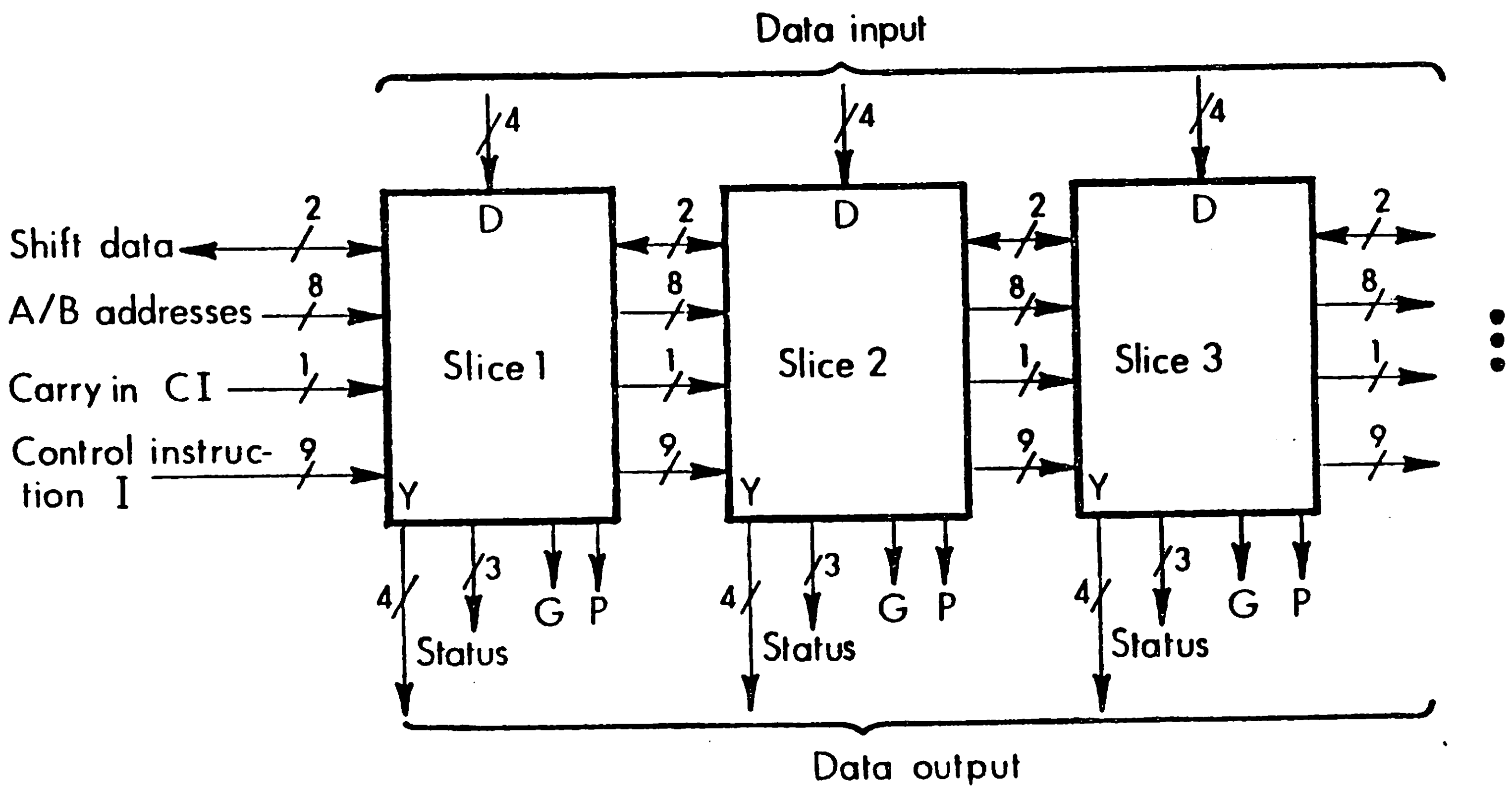


Figure 2.4. An array of 2901 4-bit slices

an additional IC, for example, the 2902 [Advanced Micro Devices 1979], which generates from G and P the signals to be applied to the CI lines. Note, however, that if carry lookahead is implemented, the microprocessor no longer has the structure of a simple ILA.

Most commercial processor slices are similar in general organization and functional capability to the 2901. The Intel 3002 is unique in its use of a 2-bit slice with an unusually large number of IO data buses [Intel 1976]. The buses include two input data buses, an output address bus, and a separate output data bus. In addition to its 11 general-purpose registers, the 3002 slice has a separate main memory address register. It also has right-shift logic which is implemented in the ALU itself, and no explicit left-shift logic. Unlike the 2901, the 3002 slice does not provide explicit status signals such as the zero, sign and overflow flags. The Texas Instruments 74S481 is a 4-bit slice having a double-length accumulator and a dual memory address register [Texas Instruments 1979]. It has only one other temporary register, and the usual CPU working registers are maintained externally in the main memory. Another important feature of the 74S481 is its built-in microprogrammed multiply and divide algorithms. These algorithms are of the shift-and-add/subtract type, hence they can be extended to operands of arbitrary length using

the basic ILA cell interconnection structure. The 74S481, like the 2901, provides the usual CPU status signals. The Motorola 10800 processor slice is a 4-bit ECL slice with a 4-bit ALU [Motorola 1976]. It has one accumulator and only only other temporary register. Hence, the main memory must provide the CPU scratchpad or working registers, as in the 74S481. The ALU of the 10800 can perform decimal (BCD) arithmetic operations as well as the usual binary (two's complement) arithmetic. A more recent ECL processor slice, the Fairchild 100220, has an 8-bit word size [Chu 1979]. This slice uses a 9-bit wide data path which includes one parity bit. The parity bit along with the appropriate encoding and decoding logic provide on-chip error-checking. Like the 10800, it has numerous data paths and multiplexers, but only one working register. An interesting feature of the 100200 is that its ALU can perform both packed and unpacked decimal arithmetic. The key characteristics of the foregoing processor slices are summarized in Table 2.1.

Microprogram Sequencers

A microprogram sequencer contains the logic circuitry required to select the address of the next microinstruction to be executed. This address can be obtained from various sources such as a microprogram counter, a stack,

Table 2.1. Comparison of the key features of five representative commercial processor slices

	Processor Slice				
Characteristics	2901	3002	74S481	10800	100220
Primary manufacturer	Advanced Micro Devices	Intel	Texas Instruments	Motorola	Fairchild
IC technology	Low-power Schottky TTL	Schottky TTL	Schottky TTL	ECL	ECL
Slice width (bits)	4	2	4	4	8
Maximum clock rate (MHz)	10	10	10	20	25
Number of pins	40	28	48	48	68
Number of basic ALU operations	16	40	41	64	27
Number of registers	17	13	4	1	2
On-chip error correction present?	no	no	yes	no	yes
Number of input-output buses	2	5	5	5	6

or an external bus. Figure 2.5 shows the 2909 4-bit microprogram sequencer slice of the 2900 family [Advanced Micro Devices 1979]. It consists of five functional blocks: a 4-to-1 multiplexer, an input register AR, a 4 × 4-bit stack, a microprogram counter and some OR-AND logic. The output address Y is 4 bits wide. It can be increased to any multiple of 4 by simply cascading the 2909 slices in the same way as the processor slices are cascaded in Figure 2.4. Y can be selected from any of four sources: the microprogram counter, the address input bus D, the stack, or the register AR. The microprogram counter can be incremented to generate a sequence of consecutive addresses. The address input bus D usually provides the branch address field of the current microinstruction, or the starting address of a microprogram derived from the opcode field of an external instruction. The stack is used to hold the return addresses of microprogram subroutines. It allows up to four levels of subroutine nesting. The register AR is useful for storing the starting address of a frequently-used microprogram. Finally, the OR-AND logic is used for selectively changing bits of Y in response to external signals such as interrupts.

The microprogram sequencers of the other bit slice families are functionally similar to the 2909. The Intel 3001 is a 9-bit non-bit-sliced microprogram sequencer

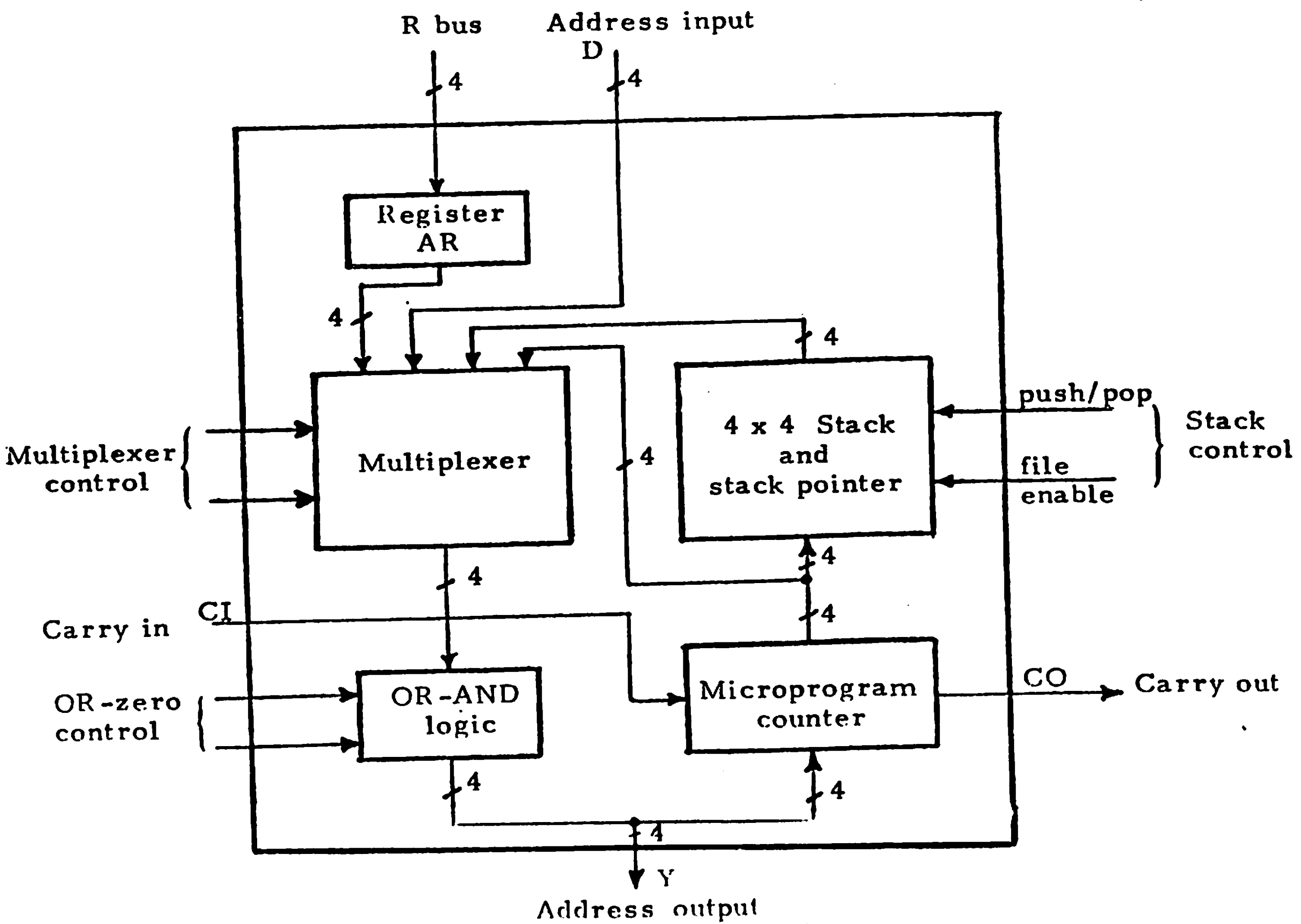


Figure 2.5. Block diagram of the 2909 4-bit microprogram sequencer slice

chip. Unlike the 2909, it does not include a microprogram counter or a stack. Thus the next microinstruction address must be explicitly calculated or specified in some field of the current microinstruction via special "jump commands." This rather complicated microprogram sequencing scheme has made the 3001 obsolete. The 74S482 microprogram sequencer in the Texas Instruments 481 series is a 4-bit slice that is similar to the 2909. It includes a 4-bit adder unit instead of a microprogram counter, which allows more complex microinstruction addressing modes. Like the 2909, it also has a 4-word stack. Compared to the foregoing microprogram sequencers, the 4-bit 10801 bit-sliced ECL chip in the Motorola 10800 series has a rather complicated internal structure [Motorola 1976]. It includes a 4-word stack, four miscellaneous registers, an incrementer and several multiplexers. It has all the main features of both the 2909 and the 3001. Like the 2909 it can increment the present microinstruction address, and can support subroutine nesting in the microprograms. It has a status register and associated status control logic not present in the 2901. The 10801 slice, like the 3001, has its address selection signals organized into a set of jump commands. The microprogram sequencer chip in the Fairchild 100K family has not yet been announced. Table 2.2 compares some key features of the above microprogram sequencers.

Table 2.2. Comparison of the key features of four representative commercial microprogram sequencers

	Microprogram Sequencer Slice			
Characteristics	2909	3001	74S482	10801
Primary manufacturer	Advanced Micro Devices	Intel	Texas Instruments	Motorola
IC technology	Low-power Schottky TTL	Schottky TTL	Schottky TTL	ECL
Bit-sliced?	yes	no	yes	yes
Slice width (bits)	4	9	4	4
Number of pins	28	40	20	48
Stick size (words)	4	0	4	4
Microprogram counter present?	yes	no	no	yes
Number of registers	2	1	1	3
Number of input-output buses	4	3	2	6

2.3 Circuit Model

As mentioned before we are interested in representing digital circuits at a level higher than the usual gate level. We therefore propose to treat a circuit U as a collection of small well-defined register-level modules interconnected in a known way. The modules themselves are regarded as black boxes or primitives whose input-output or functional behavior is completely defined. For example, functional units like multiplexers, adders, decoders and comparators are treated as primitive modules in our analysis. In the case of sequential logic, only simple synchronous machines like registers and counters are considered as primitive. A list of representative primitive components is given in Table 2.3. The internal structure of these primitive modules is not considered. Typical modules of the foregoing type contain from ten to a hundred gates, a complexity level associated with medium-scale integration (MSI). Such register-level modules form the basic functional building blocks of many useful digital systems [Blakeslee 1979]. Hence this kind of register-level modeling is often referred to as functional modeling.

It is relatively easy to model bit-sliced devices at the register level, since the number of modules involved and the number of input-output (IO) lines of the modules are relatively small. For example, the block diagram of

Table 2.3. List of some representative register-level primitive modules

Logic Type	Primitive Modules
Combinational	multiplexers encoders decoders arithmetic-logic circuits parity generators comparators code converters carry-lookahead generators
Synchronous Sequential	registers shift registers counters timers

the 2901 slice shown in Figure 2.3 is a register-level circuit having eight primitive modules. The corresponding gate-level circuit contains hundreds of primitives (gates). Thus with register-level models it is possible to use simpler representations of complex circuits. Furthermore, we no longer require gate-level circuit descriptions which are usually not provided by the manufacturers of commercial LSI/VLSI ICs.

Corresponding to the above functional view of the primitive module behavior, we next define a fault model based on similar functional considerations.

2.4 Fault Model

Faults in a digital circuit can in general be classified as either logical or parametric [Breuer and Friedman 1976]. A *logical fault* changes the logic function realized by the circuit. On the other hand, a *parametric fault* affects electrical parameters such as voltage, current or time delay. In this thesis we are only concerned with logical faults.

Let M be a primitive combinational or synchronous sequential logic module in a circuit U under test. Let z denote the function realized by M, and let s be the number of internal states of M; $s = 1$ if M is combinational. A malfunction F of M is called a *(functional) fault* of M if F permanently changes M to a module M^F realizing z^F, where

$z \neq z^F$ and the number of states s^F of M^F is not greater than s. M and M^F are assumed to have the same input-output connection lines and signal names.

Thus faults in a combinational module can induce arbitrary changes in the truth table of the module, but cannot convert it to a sequential circuit. To detect these faults, it is necessary and sufficient to apply all 2^n input vectors to an n-input module. This fault model is relatively powerful. It includes as a proper subset all single and multiple faults generated by the standard stuck-line fault model. The restriction excluding sequential behavior appears to be relatively minor. It does, however, exclude faults that convert a combinational to a sequential circuit. Such faults are uncommon, but are possible in certain IC technologies like CMOS [Wadsack 1978].

When M is a sequential circuit, we allow faults to cause any change in the state table of M that does not increase the number of states. This is quite realistic in the case of modules in which there are k binary memory elements and exactly 2^k states; most microcomputer components are of this type. To detect all faults in a sequential module it is therefore necessary and sufficient to verify its state table. The restriction on the number of states allows fault detection using the classical checking sequence approach [Friedman and Menon 1971]. In this

approach an input sequence called a checking sequence is used to test each sequential module M for functional faults. More formally a *checking sequence* for M is a sequence of inputs which distinguishes the state table of M from all other state tables with the same number of states, and IO lines. Clearly such a sequence will also detect all possible single and multiple stuck-line faults in M. The verification of the state table of M involves the initialization of M to desired internal states, and the identification of the states reached by M in response to all possible external input combinations. Thus, in general, checking sequences tend to be long and difficult to construct. But as will be seen later, checking sequences are relatively easy to derive for the relatively simple kinds of sequential elements encountered in practical bit-sliced circuits.

As is customary when analyzing faults in synchronous sequential circuits, the clock circuitry is assumed to be fault-free and the clock lines are not shown explicitly in circuit diagrams. It is further assumed that only one module in the circuit U is faulty at any time. This *single fault assumption* is included in most fault models. It is justified if the module failures are independent, and if U is tested relatively often. The single fault assumption covers some but not all stuck-line faults on module inter-

connections. In particular, it does not include all s-a-0/1 faults on all intermodule lines that fan out, since such faults may affect the behavior of more than one module. However, as we show later, stuck-line faults on these lines are often detected by the tests for the functional failures.

It should be emphasized that the foregoing model will only be applied to n-input s-state modules where n and s are relatively small. Such small modules are characteristic of bit-sliced devices. The small size of the modules is necessary to make practical the essentially exhaustive testing methods required by the fault model. Although individual modules are tested exhaustively, networks of these modules are tested in an efficient non-exhaustive manner.

The high-level circuit and fault models defined above are independent of the logic design and the IC technology used. This fact can be used to advantage during the original circuit design process. Exhaustive functional testing of the individual modules in a circuit U can also detect any design errors in them. Hence the test sequences for U derived using the functional fault model are valid for verifying the correctness of the logic design of U. Such design verification tests are extremely useful in practice, especially when U is a complex VLSI chip.

2.5 Test Generation

Here we outline a general approach used to generate test sequences for a network of register-level modules. Let U be a network of k interconnected modules $M_1, M_2, \ldots, M_k$. Let T_i be a complete test sequence that detects all faults in M_i allowed by the fault model. T_i consists of all possible input patterns of M_i if M_i is combinational. If M_i is a sequential module then T_i must be a checking sequence. It will be shown that for the small sequential modules encountered in bit-sliced devices, checking sequences of minimal or near-minimal length are easily derived. When M_i is an internal component of U, faults in M_i are detected by test sequence T'_i, which when applied to the primary inputs of U

(1) causes T_i to be applied to M_i, and

(2) causes the responses of M_i to be propagated to the observable outputs of U.

A composite test sequence T_U for the entire circuit U is obtained by concatenating the various T'_i test sequences thus

$$T_U = T'_1 \, T'_2 \cdots T'_k \, . \qquad (2.1)$$

Since only one module in U can be faulty at a time, the

tests for several modules can often be merged, thus reducing the length of T_U.

The problem of constructing T'_i of Equation (2.1) is often non-trivial. It involves the application of a desired input pattern to the module inputs, and at the same time, the propagation of the module outputs to the primary observable outputs. These operations are very similar to the line justification and D-drive operations employed in the classical D-algorithm for test generation [Breuer and Friedman 1976].

The above requirements for constructing the T'_i sequences may make it very difficult to generate test patterns for arbitrary unstructured register-level circuits where detection of all functional faults, i.e., 100 percent fault coverage, is desired. For such circuits heuristic methods of test generation are normally used, as mentioned in the previous chapter. However, we are dealing here with a class of well-structured circuits, namely bit-sliced systems, for which T'_i can be constructed fairly easily. As demonstrated by the commecial bit slices discussed in Section 2.2, bit-sliced circuits are composed of relatively simple modules like registers, multiplexers, adders, etc. Furthermore, the interconnections between the modules are relatively simple, and the module IO lines are easily accessed via the primary IO lines of the bit-sliced circuit.

In most cases the only sequential modules present are registers or small scratchpad RAMs. These modules are directly accessible and can readily be initialized to any desired state during testing. Also the regular array-like interconnection structure of a bit-sliced system can be used to advantage in constructing array tests from the tests for an individual slice. These considerations are analyzed in detail in the next chapter for a family of easily testable bit-sliced processors.

CHAPTER 3

EASILY TESTABLE PROCESSOR SLICES

In the previous chapter we discussed the main features of commercial bit-sliced processors. We also introduced high-level circuit and fault models, and outlined a general testing approach employing these models. In this chapter, a detailed processor cell model C that has all the main features of commercial processor slices is developed. By applying our circuit and fault models to C, we demonstrate how they simplify the test generation problem for C. The basic cell C is then extended to make it more closely resemble commercial bit slices like the AMD 2901. Test generation for an array of such processor cells is discussed in Chapter 4.

3.1 Processor Cell Model C

In order to obtain a manageable yet reasonably realistic processor cell model we make the following preliminary assumptions.

(i) The operand size of the cell is one bit;

(ii) The cell contains only two scratchpad registers, an accumulator A and one additional

temporary register T;

(iii) Only ripple-carry propagation is used between the cells of a bit-sliced array.

The above assumptions may be justified as follows. The use of 1-bit operands makes it feasible and relatively simple to use the powerful functional fault model of Section 2.4. Furthermore, we will show that tests based on a 1-bit cell can easily be extended to larger cells. It is also worth noting that 1-bit processors may be useful in themselves; the Motorola 14500 is an example of a commercially-available 1-bit microprocessor which, however, is not bit-sliced [Motorola 1977]. The use of only two general-purpose registers is mainly to simplify the test generation process. Again, direct extension of the model to cells having a larger set of registers is possible, and is discussed later. Several important commercial microprocessor cells such as the Motorola 10800 and Texas Instruments 74S481 have just two working registers like our model; see Section 2.2. Ripple-carry propagation only is allowed because, as noted earlier, carry lookahead cannot be implemented without destroying the basic ILA structure. The OM-2 data path chip is also designed using 1-bit processor cells with ripple-carry interconnections. Commenting on this aspect of the OM-2 design, Mead and Conway state: "Simulation of several lookahead carry circuits

indicated that they would add a great deal of complexity to the system without much gain in performance" [Mead and Conway 1980, p. 150].

A block diagram of the basic 1-bit processor cell model C is shown in Figure 3.1. Its major external control lines are defined in Table 3.1. C consists of six basic functional modules: an ALU M_F, three multiplexers M_L, M_R and M_S, and two registers M_A and M_T. The ALU module performs eight different operations on the two 1-bit operands R and S. These include arithmetic operations such as addition, subtraction and negation; and logical operations such as AND, OR, EXCLUSIVE-OR and complementation. Note that the widely-used commercial 4-bit processor slice, the AMD 2901 performs essentially the same operations as C [Advanced Micro Devices 1979]. The operands R and S can be obtained from several sources: an external data line D, the registers A and T, or a logical 0, as shown in Figure 3.1. The final result of the ALU operation can be transferred to the registers A and T through the shifter M_L, and also to the output data line Y. M_L allows a 1-bit left or right shift operation to be combined with any ALU operation. To facilitate the microinstruction decoding process, we have distributed it among the various functional blocks; for example, the decoding logic for the lines I_3, I_4 and I_5 that select the ALU function is included in the ALU module M_F. To expand the word size of

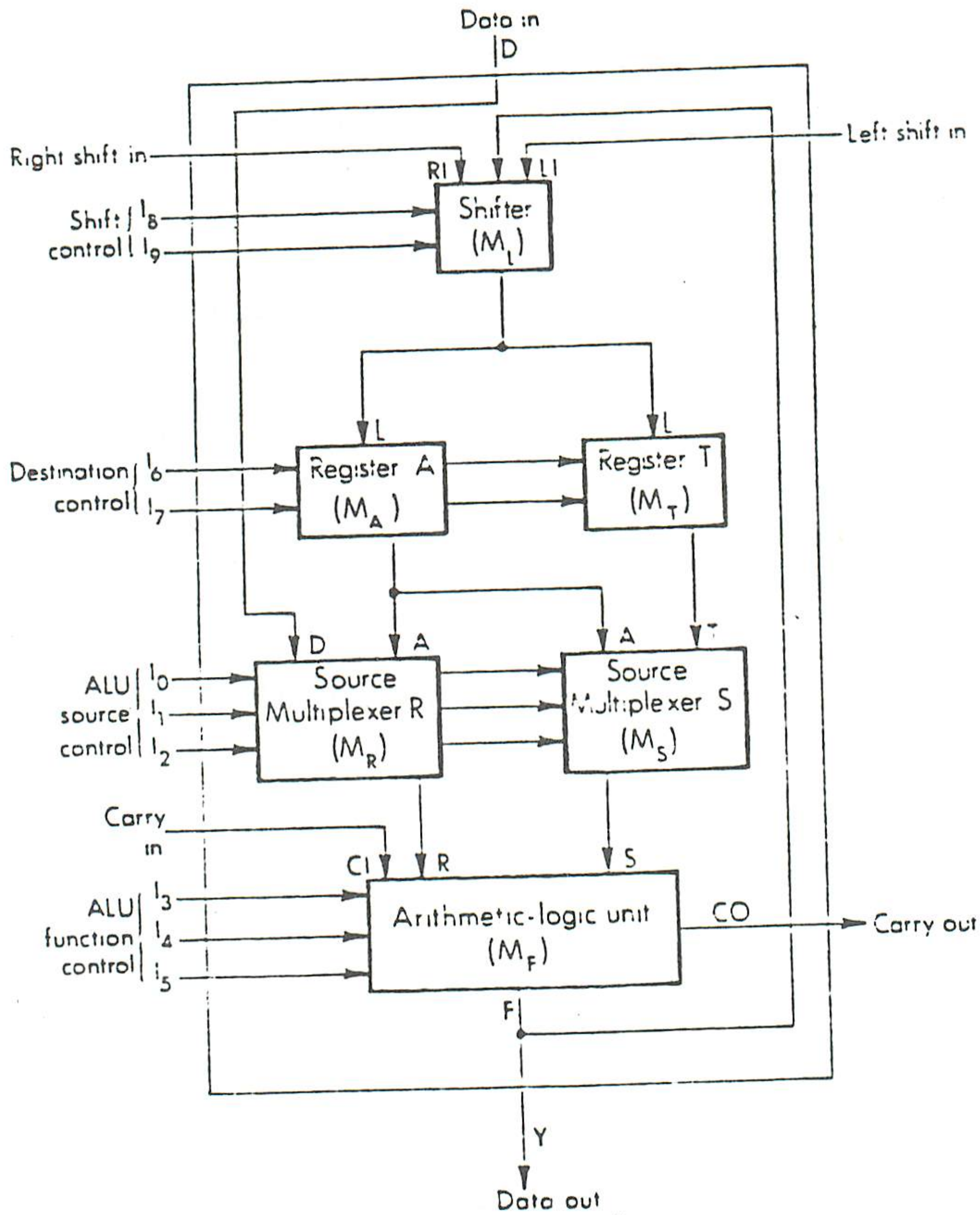


Figure 3.1. 1-bit processor cell C

Table 3.1. Definitions of the major control lines of C

I_2	I_1	I_0	R	S
0	0	0	0	A
0	0	1	0	T
0	1	0	D	A
0	1	1	D	T
1	0	0	A	A
1	0	1	A	T
1	1	0	D	0
1	1	1	A	0

ALU source control

Fcn.	I_5	I_4	I_3	F output
f_0	0	0	0	R plus S
f_1	0	0	1	S minus R
f_2	0	1	0	R minus S
f_3	0	1	1	R OR S
f_4	1	0	0	R AND S
f_5	1	0	1	$\overline{R}$ AND S
f_6	1	1	0	R $\oplus$ S
f_7	1	1	1	R $\overline{\oplus}$ S

ALU function control

I_7	I_6	Function
0	0	A←L
0	1	T←L
1	0	A,T←L
1	1	No op

ALU destination control

I_9	I_8	L output
0	0	F
0	1	RI
1	0	LI
1	1	Not used

Shift control

the operands being processed from one to N bits, N copies of C are cascaded. Ripple-carry propagation between the ALUs of the adjacent cells is realized by means of the carry-in (CI) and carry-out (CO) lines. During logical operations the carry lines are not used. However, for testing purposes, we define CO = CI for the ALU logic functions f_3, f_4, f_5, f_6 and f_7. Thus C includes all the basic features of the 2901 (Figure 2.3) except its carry-lookahead logic.

Later in Section 3.5 we will discuss certain extensions to the cell C which remove the above assumptions (i) and (ii). These extended cells operate on k-bit operands (k = 2,4,8 etc.) and include a scratchpad memory and carry-lookahead logic inside each cell. The use of carry-lookahead logic between the cells of an array is also considered later.

We now outline a test generation method for the processor cell C using the general approach described in Section 2.5. Applying our high-level circuit model, the cell C can be divided into six modules: four combinational modules M_F, M_R, M_S and M_L, and two sequential modules M_A and M_T as shown in Figure 3.1. Each of these modules except M_F has only one output signal which must be identified during testing. The output of every module is directly or indirectly observable at the Y or CO primary outputs of the cell.

A sequence T_i is said to be a *complete test sequence* for module M_i if it can detect any fault in M_i allowed by the functional fault model introduced in Section 2.4. T_F, T_R, T_S, T_L, T_A and T_T denote complete test sequences for the six modules of C. Let T_i' be a test sequence which when applied to the primary inputs of C, applies T_i to the module M_i, and at the same time propagates the output signals of M_i to the observable outputs of C. Then a test sequence T_C for C can be expressed as a simple concatenation of the six T_i' test sequences for the six modules of C (see Section 2.5). In the next two sections, the construction of test patterns for various combinational and sequential modules of C is discussed in detail.

3.2 Testing Combinational Modules of C

Consider first the source multiplexer module M_R. It has five input lines and one output line whose state has to be observed during testing. Thus T_R must contain all $2^5 = 32$ 5-bit binary patterns as shown in Table 3.2, which is the same as the truth-table of M_R. It is required to construct a corresponding test sequence T_R' which when applied to C, causes all 32 input test patterns of T_R to be applied to M_R. T_R' must also propagate the response signal R to the primary output lines of C. Four of the five input lines of M_R, namely the control lines I_0, I_1, I_2 and

Table 3.2. Test sequence T_R for the module M_R of C.

No.	Inputs of M_R					Output of M_R
	I_2	I_1	I_0	D	A	R
1	0	0	0	0	0	0
2	0	0	0	0	1	0
3	0	0	0	1	0	0
4	0	0	0	1	1	0
5	0	0	1	0	0	0
6	0	0	1	0	1	0
7	0	0	1	1	0	0
8	0	0	1	1	1	0
9	0	1	0	0	0	0
10	0	1	0	0	1	0
11	0	1	0	1	0	1
12	0	1	0	1	1	1
13	0	1	1	0	0	0
14	0	1	1	0	1	0
15	0	1	1	1	0	1
16	0	1	1	1	1	1
17	1	0	0	0	0	0
18	1	0	0	0	1	1
19	1	0	0	1	0	0
20	1	0	0	1	1	1
21	1	0	1	0	0	0
22	1	0	1	0	1	1
23	1	0	1	1	0	0
24	1	0	1	1	1	1
25	1	1	0	0	0	0
26	1	1	0	0	1	0
27	1	1	0	1	0	1
28	1	1	0	1	1	1
29	1	1	1	0	0	0
30	1	1	1	0	1	1
31	1	1	1	1	0	0
32	1	1	1	1	1	1

the input data line D, are also primary input lines of C. The remaining input A coming from the register module M_A can be easily controlled by manipulating the state of M_A. This can be done by applying the appropriate logic value 0 or 1 to the D input of C, and then transferring the value of D to the register M_A via the source multiplexer M_R, the ALU M_F, and the shifter M_L. The responses of M_R to T_R can be observed at the Y output of C by propagating the output signal R of M_R to Y through the ALU module M_F. One way of achieving this propagation is by setting the control signals I_3, I_4 and I_5 of M_F to the logic values 0, 1 and 1, respectively, so that the ALU performs an EXCLUSIVE-OR operation resulting in $Y = F = R \oplus S$. Thus any change in the signal R due to a fault in M_R is detected at Y. Table 3.3 gives a test sequence T'_R of length $|T'_R| = 36$, which completely tests M_R when applied to the primary inputs of C. As an example, consider the following input vector t of M_R.

$$t: \quad (I_0, I_1, I_2, D, A) = (1,0,1,1,1)$$

Let the register A be initially set to 1. Using the above method of propagating R to the Y output, a corresponding input vector t′ of C which applies t to M_R, is

$$t': \quad (I_0{:}I_9, D, RI, LI, CI) = (1,0,1,0,1,1,1,1,0,0,1,d,d,d) \tag{3.1}$$

Table 3.3. Test sequence T'_R for the multiplexer modules M_R and M_S of C

No.	ALU Destination Control		ALU Function Control			ALU Source Control			Data In	Register Outputs		Mux. Outputs		Carry In	Cell Outputs	
	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	D	A	T	R	S	CI	Y	CO
1	1	0	0	1	1	1	1	0	0	d	d	0	0	d	0	d
2	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
3	1	1	0	0	1	0	0	1	1	0	0	0	0	0	1	0
4	1	1	0	0	0	0	1	0	1	0	0	1	0	0	1	0
5	1	1	0	0	0	0	1	1	1	0	0	1	0	1	0	1
6	1	1	0	0	1	1	0	0	1	0	0	0	0	1	0	1
7	1	1	0	1	0	1	0	1	1	0	0	0	0	0	1	0
8	1	1	0	0	1	1	1	0	1	0	0	1	0	0	0	0
9	1	1	0	1	0	1	1	1	1	0	0	0	0	1	0	1
10	0	0	0	1	1	1	1	0	1	0	0	1	0	d	1	d
11	1	1	0	0	0	0	0	0	1	1	0	0	1	0	1	0
12	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0
13	1	1	1	1	0	0	1	0	1	1	0	1	1	0	0	0
14	1	1	0	1	0	0	1	1	1	1	0	1	0	1	1	1
15	1	1	1	1	0	1	0	0	1	1	0	1	1	1	0	1
16	1	1	1	1	0	1	0	1	1	1	0	1	0	0	1	0
17	1	1	1	1	0	1	1	0	1	1	0	1	0	1	1	1
18	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0
19	0	1	0	1	1	1	1	0	1	1	0	1	0	d	1	d
20	1	1	0	0	0	0	0	0	0	1	1	0	1	1	0	1
21	1	1	0	0	1	0	0	1	0	1	1	0	1	1	1	1
22	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	0
23	1	1	1	1	0	0	1	1	0	1	1	0	1	0	1	0
24	1	1	1	1	1	1	0	0	0	1	1	1	1	0	1	0
25	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1
26	1	1	1	1	0	1	1	0	0	1	1	0	0	1	0	1
27	1	1	1	1	1	1	1	1	0	1	1	1	0	1	0	1
28	0	0	0	1	1	1	1	0	0	1	1	0	0	d	0	d
29	1	1	1	1	1	0	0	0	0	0	1	0	0	0	1	0
30	1	1	1	1	0	0	0	1	0	0	1	0	1	1	1	1
31	1	1	1	1	1	0	1	0	0	0	1	0	0	1	1	1
32	1	1	1	1	1	0	1	1	0	0	1	0	1	0	0	0
33	1	1	1	1	0	1	0	0	0	0	1	0	0	d	0	d
34	1	1	1	1	1	1	0	1	0	0	1	0	1	1	0	1
35	1	1	1	1	0	1	1	0	0	0	1	0	0	d	0	d
36	1	1	1	1	0	1	1	1	0	0	1	0	0	d	0	d

Note: In all cases $(I_9, I_8, RI, LI) = (0,0,d,d)$.

where $I_0:I_9$ denotes $I_0, I_1, \ldots, I_9$, and "d" denotes a don't care value which may be 0 or 1. Figure 3.2 shows a set-up of C for testing M_R along these lines. It also illustrates the application to C of the above test pattern t', which is the 16th input test pattern in the T'_R sequence of Table 3.3. As depicted in Figure 3.2, t' applies t to the input lines of M_R and at the same time propagates the response to t of M_R to the Y output of C. The path created by t' from the fault site M_R to the observable output Y is shown in heavy lines. The input patterns t'_1, t'_{10}, t'_{19} and t'_{28} in the sequence T'_R are used to set the initial state of the registers A and T, where t'_i is the ith input pattern in T'_R.

The other multiplexer module in C, namely M_S, is very similar to M_R. It also has five input lines that are easily accessed, and its output signal S can be propagated to Y in the way discussed above for the output signal R of M_R. The configuration of C for testing M_S is also similar to that used for M_R; see Figure 3.2. Furthermore, a change in any one of the two signals R and S can be observed at the Y output using an ALU EXCLUSIVE-OR operation such as $F = R \oplus S$. Since at most one module is assumed to be faulty, an ALU EXCLUSIVE-OR function makes it possible to observe the output signals of M_R and M_S at the same time. It is therefore possible to test M_R and M_S simultaneously. For example, the input t' of C given by Equa-

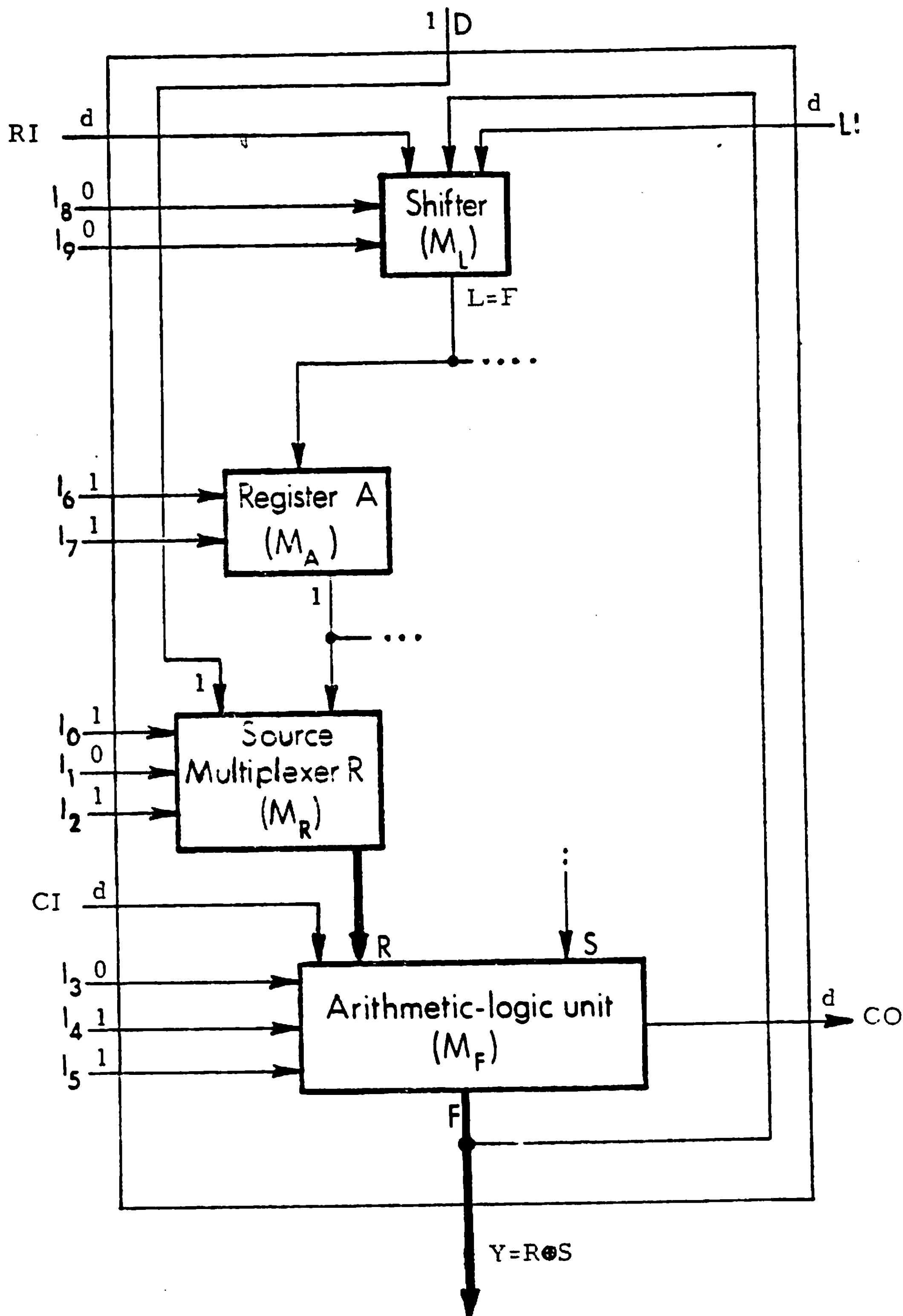


Figure 3.2. A set-up of C for testing the multiplexer module M_R

tion (3.1) also applies the test pattern $(I_0, I_1, I_2, A, T) = (1,0,1,1,0)$ to M_S. Thus t′ can detect a fault in any one of the two modules M_R and M_S. In the same way, the input patterns in the sequence T'_R of Table 3.3 apply all 32 test patterns to M_S. Hence T'_R is also a complete test sequence for M_S.

Now the ALU module M_F has six input lines, hence any complete test sequence for M_F must contain all 64 6-bit binary input patterns. Four of M_F's six input lines, namely I_3, I_4, I_5 and CI, are also primary inputs of C. The remaining two input lines, R and S, can be set to any desired values using the multiplexers M_R and M_S. For example, $R = 1$ and $S = 0$ can be realized by applying the logic values 0, 1, 1, 1 to the input lines I_0, I_1, I_2, D respectively. The two outputs Y and CO of the ALU are also the two primary outputs of C, and hence are directly observable. Thus a test sequence T'_F can be constructed very easily. Again many test patterns are common to the multiplexer modules and the ALU. The four combinations 00, 01, 10 and 11 of the signal pair RS occur respectively 12, 8, 8 and 4 times in the test sequence T'_R for M_R and M_S; see Table 3.3. Of the 12 occurrences of $RS = 00$ during the application of T'_R to C, nine can be used for partial testing of the ALU functions f_0, f_1, f_2, f_6 and f_7. The corresponding nine input patterns in T'_R are t'_2, t'_3, t'_6,

t'_7, t'_9, t'_{12}, t'_{26}, t'_{29} and t'_{31}. Similarly, the 20 occurrences of RS = 01, 10 and 11 in T'_R can also be used for testing the ALU. Thus T'_R provides 29 of the 64 input patterns needed to test the ALU module M_F. The remaining 35 test patterns for M_F are provided by the test sequence T'_a given in Table 3.4. Thus the concatenation of T'_R from Table 3.3 and T'_a from Table 3.4 forms a complete test sequence T'_F for M_F.

Test generation for the remaining combinational module M_L of C is similar to the above. The resulting test sequence T'_L of length 33 for the module M_L is given in Table 3.5. Thus we have the following result.

Lemma 3.1: There exists a test sequence $T'_b = T'_R T'_a T'_L$ of length 105 that completely tests the combinational modules M_R, M_S, M_F and M_L of the processor cell C. T'_b is defined by Tables 3.3, 3.4 and 3.5.

3.3 Testing Sequential Modules of C

The register modules M_A and M_T are Moore-type synchronous sequential machines [Friedman and Menon 1971], since their direct output signals A and T are the same as their internal states. M_A and M_T have essentially similar state tables and state diagrams as shown by Figures 3.3 and 3.4. The rows of the tables represent the

Table 3.4. Test sequence T'_a for the ALU module M_F of C.

No.	ALU Destination Control		ALU Function Control			ALU Source Control			Data In	Multiplexer Outputs		Carry In	Cell Outputs	
	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	D	R	S	CI	Y	CO
1	1	0	1	1	1	1	1	0	0	0	0	d	0	d
2	1	1	0	0	0	0	1	1	1	1	1	1	1	1
3	1	1	0	0	0	0	1	1	1	1	1	0	0	1
4	1	1	0	0	0	0	1	0	0	0	0	1	1	0
5	1	1	0	0	1	0	1	1	1	1	1	0	1	0
6	1	1	0	0	1	0	1	1	1	1	1	1	0	1
7	1	1	0	0	1	0	1	1	0	0	1	0	0	1
8	1	1	0	0	1	0	1	0	1	1	0	1	1	0
9	1	1	0	1	0	0	1	1	1	1	1	0	1	0
10	1	1	0	1	0	0	1	1	1	1	1	1	0	1
11	1	1	0	1	0	0	1	0	1	1	0	0	0	1
12	1	1	0	1	0	0	1	1	0	0	1	1	1	0
13	1	1	0	1	1	0	1	0	0	0	0	0	0	0
14	1	1	0	1	1	0	1	1	0	0	1	0	1	0
15	1	1	0	1	1	0	1	0	1	1	0	0	1	0
16	1	1	0	1	1	0	1	1	1	1	1	0	1	0
17	1	1	0	1	1	0	1	0	0	0	0	1	0	1
18	1	1	0	1	1	0	1	1	0	0	1	1	1	1
19	1	1	0	1	1	0	1	0	1	1	0	1	1	1
20	1	1	0	1	1	0	1	1	1	1	1	1	1	1
21	1	1	1	0	0	0	1	0	0	0	0	0	0	0
22	1	1	1	0	0	0	1	1	0	0	1	0	0	0
23	1	1	1	0	0	0	1	0	1	1	0	0	0	0
24	1	1	1	0	0	0	1	1	1	1	1	0	1	0
25	1	1	1	0	0	0	1	0	0	0	0	1	0	1
26	1	1	1	0	0	0	1	1	0	0	1	1	0	1
27	1	1	1	0	0	0	1	0	1	1	0	1	0	1
28	1	1	1	0	0	0	1	1	1	1	1	1	1	1
29	1	1	1	0	1	0	1	0	0	0	0	0	0	0
30	1	1	1	0	1	0	1	1	0	0	1	0	1	0
31	1	1	1	0	1	0	1	0	1	1	0	0	0	0
32	1	1	1	0	1	0	1	1	1	1	1	0	0	0
33	1	1	1	0	1	0	1	0	0	0	0	1	0	1
34	1	1	1	0	1	0	1	1	0	0	1	1	1	1
35	1	1	1	0	1	0	1	0	1	1	0	1	0	1
36	1	1	1	0	1	0	1	1	1	1	1	1	0	1

Note: In all cases $(I_9, I_8, RI, LI) = (0,0,d,d)$.

Table 3.5. Test sequence T'_L for the shifter module M_L of C

No.	Shift Control I_9	I_8	Shift Inputs RI	LI	Data In D	Cell Output Y
1	0	0	d	d	0	0
2	0	0	0	0	0	0
3	0	0	0	0	1	1
4	0	0	1	0	1	0
5	0	0	1	0	1	1
6	0	0	0	1	1	0
7	0	0	0	1	1	1
8	0	0	1	1	1	0
9	0	0	1	1	1	1
10	0	1	0	0	1	0
11	0	1	0	0	1	1
12	0	1	1	0	0	0
13	0	1	1	0	0	1
14	0	1	0	1	1	0
15	0	1	0	1	1	1
16	0	1	1	1	0	0
17	0	1	1	1	0	1
18	1	0	0	0	1	0
19	1	0	0	0	1	1
20	1	0	1	0	0	0
21	1	0	1	0	1	1
22	1	0	0	1	0	0
23	1	0	0	1	0	1
24	1	0	1	1	1	0
25	1	0	1	1	0	1
26	1	1	0	0	1	0
27	1	1	0	0	1	1
28	1	1	1	0	0	0
29	1	1	1	0	1	1
30	1	1	0	1	0	0
31	1	1	0	1	1	1
32	1	1	1	1	0	0
33	1	1	1	1	1	1

Notes: (i) $(I_7,I_6,I_5,I_4,I_3,I_2,I_1,I_0,CI,CO) = (0,0,1,1,0,1,1,0,d,d)$ for the first test pattern, t_1.

(ii) $(I_7,I_6,I_5,I_4,I_3,I_2,I_1,I_0,CI,CO) = (0,0,1,1,0,0,1,0,d,d)$ for the remaining test patterns, t_2:t_{33}.

Present State A	Input I_7I_6L Next state entry								Output A
	000	010	100	110	001	011	101	111	
0	0	0	0	0	1	0	1	0	0
1	0	1	0	1	1	1	1	1	1

(a)

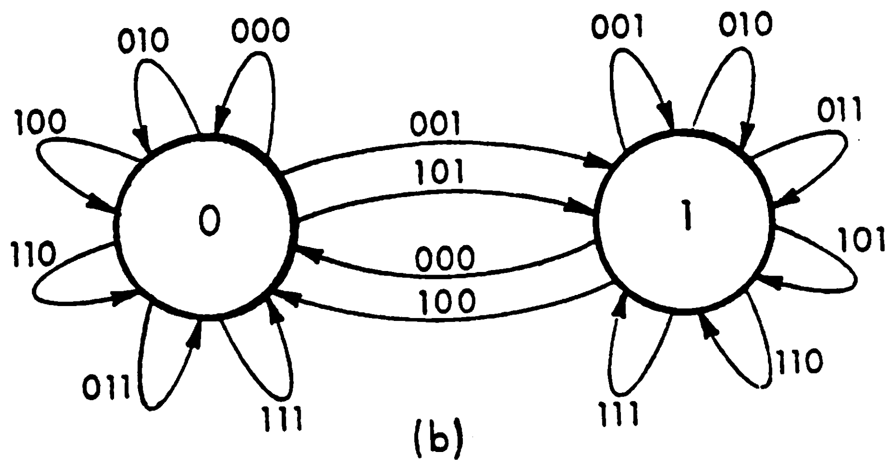


Figure 3.3. (a) State table and (b) state diagram for the register module M_A

Present State T	Input $I_7 I_6 L$ Next state entry								Output T
	000	010	100	110	001	011	101	111	
0	0	0	0	0	0	1	1	0	0
1	1	0	0	1	1	1	1	1	1

(a)

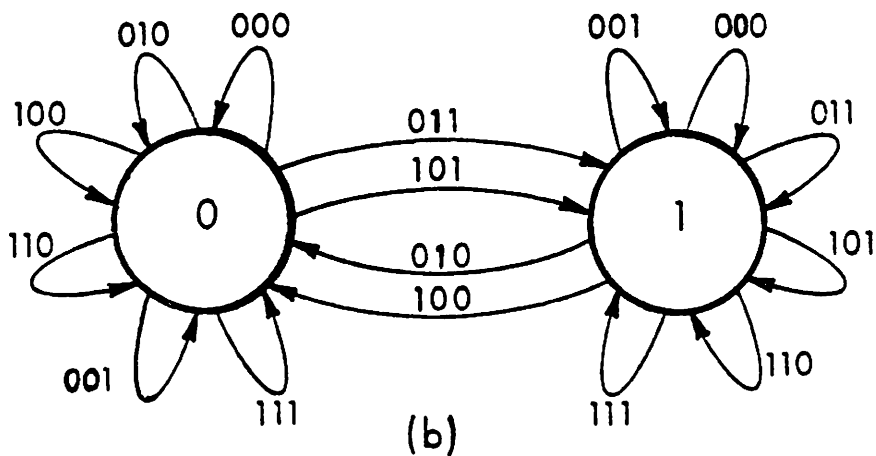


Figure 3.4 (a) State table and (b) state diagram for the register module M_T

present (internal) states 0 and 1, while the columns represent all eight possible binary input patterns. The entries in the tables, which are also referred to as *transitions*, indicate the next state reached. For example an entry x_1 in row x_0 and column y implies that the machine undergoes a transition τ from state x_0 to a next state x_1 when $I_7 I_6 L = y$; τ is denoted by $x_0 \xrightarrow{y} x_1$. The state diagrams of M_A and M_T represent their state tables in graphical form as shown in Figures 3.3b and 3.4b. A node x in a state diagram corresponds to a state or row of the state table, and a directed edge y from a node x_0 to a node x_1 represents a transition from the present state x_0 to the next state x_1 under the direct input y. Hence there are as many edges in a state diagram as there are entries (transitions) in the state table.

To test a sequential machine like M_A according to our fault model, it is necessary and sufficient to verify its state table. As discussed in Section 2.4, this involves designing a checking sequence for the machine [Friedman and Menon 1971]. A checking sequence systematically verifies all the transitions in a state table. A transition τ: $x_0 \xrightarrow{y} x_1$ is said to be *verified* if τ is induced in the machine by applying y to its input lines with x_0 as its present state, and then the resulting next state x_1 reached by the machine is observed. However, in the case of the Moore-type sequential modules like M_A and

M_T in C, the output signals are the same as the internal states. Therefore the next state x_1 reached after a state transition takes place can be verified merely by observing the output lines. Hence to verify a transition τ in M_A or M_T, it is necessary and sufficient to produce this transition in the sequential module in question, and then observe the resulting output signal from the module. Any input sequence Q for M_A or M_T that includes an input pattern corresponding to each edge in the corresponding state diagram is a checking sequence for the modules. Of course, the output signals A and T must be propagated to the primary outputs of C, but this presents no difficulty. Q will be a minimum-length checking sequence if it contains exactly one input pattern for every edge of the state diagram.

Now the state diagrams of Figures 3.3b and 3.4b are both *Eulerian directed graphs*, i.e., there exists a path P passing through every edge of the diagram exactly once. P is referred to as an *Eulerian path* and corresponds to a minimum-length checking sequence. One such path in Figure 3.3b is P_1 = 100 110 111 101 001 011 101 110 111 100 001 010 000 011 000 010 with node 0 as the starting node. The same path P_1 is also Eulerian in Figure 3.4b. The following input sequence T_e for sequential modules M_A and M_T corresponds to the above path P_1, and hence is a minimum-length checking sequence for both these modules.

$$\text{Inputs of } M_A \text{ and } M_T \begin{cases} I_7: & 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0 \\ I_6: & 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1 \\ L\ : & 0\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0 \end{cases}$$

$$\text{State of } M_A\text{: } 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0$$

$$\text{State of } M_T\text{: } 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0$$

Thus the sequential modules M_A and M_T can both be tested completely by applying T_e to their inputs and then observing their output signals A and T at the Y output of C. Again both A and T can be observed simultaneously at the Y output by propagating them to the R and S outputs of the multiplexers M_R and M_S respectively, and selecting an EXCLUSIVE-OR operation in the ALU. One way of doing this is to set the control signals $(I_0, I_1, I_2) = (1,0,1)$, and then select the ALU function f_6 or f_7 (see Table 3.1) so that the Y output signal is either $R \oplus S = A \oplus T$ or $R \bar{\oplus} S = A \bar{\oplus} T$. Using this scheme a single test sequence for C that tests both M_A and M_T completely, can be constructed. Table 3.6 gives the sequence T_e' of length 18 which applies the above sequence T_e to both M_A and M_T and also propagates their responses to the Y output. The first input pattern t_1' of T_e' is an initializing input that sets the states of A and T to 0. T_e' generates the next desired L input signal for M_A and M_T in the sequence T_e by feeding back the present F output to the L input line via the shifter M_L.

Table 3.6. Test sequence T'_e for the sequential modules M_A and M_T of C.

No.	ALU Function Control			ALU Source Control			ALU Destination Control		Shifter Output	Present State		Next State		Cell Output
	I_5	I_4	I_3	I_2	I_1	I_0	I_7	I_6	L	A	T	A	T	Y
1	1	0	0	0	0	0	1	0	0	d	d	0	0	0
2	1	1	0	1	0	1	1	0	0	0	0	0	0	0
3	1	1	0	1	0	1	1	1	0	0	0	0	0	0
4	1	1	1	1	0	1	1	1	1	0	0	0	0	1
5	1	1	1	1	0	1	1	0	1	0	0	1	1	1
6	1	1	1	1	0	1	0	0	1	1	1	1	1	1
7	1	1	1	1	0	1	0	1	1	1	1	1	1	1
8	1	1	1	1	0	1	1	0	1	1	1	1	1	1
9	1	1	0	1	0	1	1	1	0	1	1	1	1	0
10	1	1	1	1	0	1	1	1	1	1	1	1	1	1
11	1	1	0	1	0	1	1	0	0	1	1	0	0	0
12	1	1	0	1	0	1	0	0	1	0	0	1	0	1
13	1	1	1	1	0	1	0	1	0	1	0	1	0	0
14	1	1	0	1	0	1	0	0	0	1	0	0	0	0
15	1	1	0	1	0	1	0	1	1	0	0	0	1	1
16	1	1	1	1	0	1	0	0	0	0	1	0	1	0
17	1	1	0	1	0	1	0	1	0	0	1	0	0	0
18	1	1	0	1	0	1	d	d	0	0	0	d	d	0

Note: In all cases (I_9, I_8, RI, LI, D, CI, CO) = (0,0,d,d,d,d,d).

Lemma 3.2: The sequential modules M_A and M_T of C are both tested completely by the test sequence T'_e which is defined in Table 3.6 and has the minimum possible length 18.

A complete test sequence T_C for the entire processor cell C can now be obtained by concatenating the two sequences T'_b of Lemma 3.1 and T'_e of Lemma 3.2, i.e., $T_C = T'_bT'_e$. The length of T_C is $|T_C| = |T'_b|+|T'_e| = 105+18 = 123$. Note that at least 64 test patterns are required to test the module M_F alone, hence T_C is less than twice the minimum possible length.

Theorem 3.1: The processor cell C of Figure 3.1 is completely testable with the sequence $T_C = T'_bT'_e$ of length 123. T_C is within a factor of two of the minimum possible length.

3.4 Multiple-module Faults

Although the foregoing test sequences are constructed under the assumption that at most one module is faulty, they can also detect many multiple faults where two or more modules are faulty. Consider, for example, the test sequence T'_R of Table 3.3 which verifies the truth tables of the combinational modules M_R and M_S of C. Suppose that M_R and M_S contain some functional faults F_1 and F_2, respectively, at the same time. Since T'_R propagates the R

and S output signals simultaneously to Y, this multiple fault will be detected as long as F_1 and F_2 do not change the signals R and S simultaneously when T'_R is applied to C. This is because T'_R employs a 2-input EXCLUSIVE-OR ALU function which can detect changes involving only one of its input operands. For example, an important class of multiple faults that are detected by T'_R are all the 3^3-1 stuck-line faults on the three control lines I_0, I_1 and I_2 that are common to both M_R and M_S. This is proved in Appendix A. Similarly the test sequence T'_e of Table 3.6, in addition to detecting all faults in the modules M_A and M_T, also detects all 3^2-1 stuck-line faults on their common control lines I_6 and I_7 (see Appendix A). This leads to the following corollary of Theorem 3.1.

Corollary 3.1: The test sequence T_C for the processor cell C also detects all multiple-module faults resulting from (multiple) stuck-line faults on the common control lines $I_0{:}I_9$ of C.

3.5 Extensions to the Basic Cell

In this section the 1-bit processor cell C of Figure 3.1 is extended to a more general k-bit cell C^k. The test patterns for C obtained in the previous section are easily extended to C^k. Incorporating scratchpad memories in C or C^k to increase the number of working registers is

also considered. We first discuss a simple component expansion technique called replication. This technique is later used in expanding the word-size of C to k bits.

Consider any 1-bit module M such as a 1-bit multiplexer or register, with a data input J, a control input I and a (data) output V. To expand the word size of the operands processed by M from one bit to k bits, k identical copies of M can be juxtaposed to form a system M^k as shown in Figure 3.5. The control lines I are made common to all k modules. The k input signals $J_0,J_1,\ldots,J_{k-1} = J_0:J_{k-1}$ together form the new k-bit input operands, while $V_0:V_{k-1}$ is the k-bit result produced by the k modules of M^k. It should be noted that in the expanded system M^k there is no communication between the replicated modules. Such an expansion of operand size is usually referred to as *replication* [Hayes 1978]. The structure of Figure 3.5 is also that of a bit-sliced system with no inter-slice communication; cf. Figure 2.1. Hence a system expanded via replication may be regarded as a *restricted* bit-sliced system or a restricted ILA.

A complete test sequence T_M' for the replicated system M^k can easily be constructed from a complete test sequence T_M for the 1-bit module M. To test M^k completely according to the fault model used here, it is necessary and sufficient to apply T_M to every module M in M^k. Any input pattern t_i applied to M consists of two parts: a control

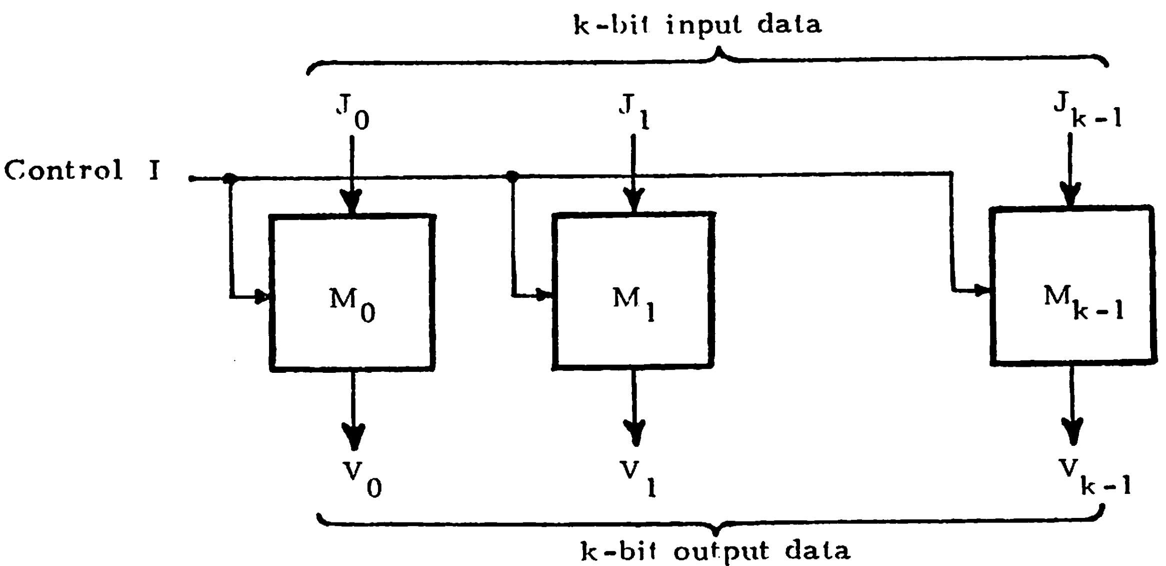


Figure 3.5. A k-bit system M^k obtained by replicating k identical 1-bit modules M

input pattern t_{iI} applied to the control input lines I, and a data input pattern t_{iJ} applied to the data input lines J. Therefore t_i can be represented as (t_{iI}, t_{iJ}). The common control input t_{iI} is applied simultaneously to every module in M^k. Since the data input lines of the k modules of Figure 3.5 are all independent of one another, it is obvious that t_{iJ} can be applied to the data input lines of every module simultaneously. Thus all k modules can be tested in parallel by a single test sequence T'_M of the same length as T_M. The ith pattern t'_i in the sequence T'_M, corresponding to t_i in T_M, is given by $t'_i = (t_{iI}, t^k_{iJ})$ where t^k_{iJ} represents t_{iJ} repeated k times. The response of M^k to t'_i is directly observable at the k-bit data output $V_0{:}V_{k-1}$. In M^k there are no reconverging signals, hence T'_M detects all (detectable) multiple-module faults among the k modules of M^k. Such multiple faults include all stuck-line faults on the common control lines of M^k. The foregoing approach to testing M^k is used in testing the extended processor cells discussed below.

k-bit Processor Cell C^k

We now extend the 1-bit cell C of Figure 3.1 into a k-bit cell C^k, so that the cell complexity can more closely approximate that of such commercial bit slices as the Intel 3002 $(k = 2)$, the AMD 2901 $(k = 4)$, or the Fairchild 100220 $(k = 8)$. We still restrict our attention to ripple-

carry propagation between the cells so that we can retain the basic ILA interconnection structure.

Figure 3.6 shows a cell C^k, which is a k-bit version of the original 1-bit cell C of Figure 3.1. In expanding the word size from one to k, we use the straightforward replication technique discussed above. In most cases, a module M_i in C is replaced by k copies of M_i. For example, consider the multiplexer module M_R. M_R is replicated k times to form the k multiplexer modules $M_{R_0}:M_{R_{k-1}}$ of C^k shown in Figure 3.6. The module M_{R_i}, where $0 \leqq i \leqq k-1$, operates on the ith bit of the k-bit word obtained from the direct input bus $D_0:D_{k-1}$, and the corresponding i^{th} bit obtained from the registers $M_{A_0}:M_{A_{k-1}}$. The same technique has been used for expanding the modules M_L, M_A M_T and M_S of C, as depicted in Figure 3.6. In the case of the ALU module M_F, however, we have retained it as a single module operating on the k-bit operands $R_0:R_{k-1}$ and $S_0:S_{k-1}$. The reason for this is to permit the use of any fast carry-propagation scheme such as carry-lookahead within C^k. If such speed-up techniques are not required, then ripple-carry propagation can be used within the k-bit ALU module M_F, in which case C^k consists of k identical cells of type C in cascade. Such an array of cells is examined in detail in the next chapter.

The fault model and test generation approach defined for C are also used with C^k. Consider, for example, mul-

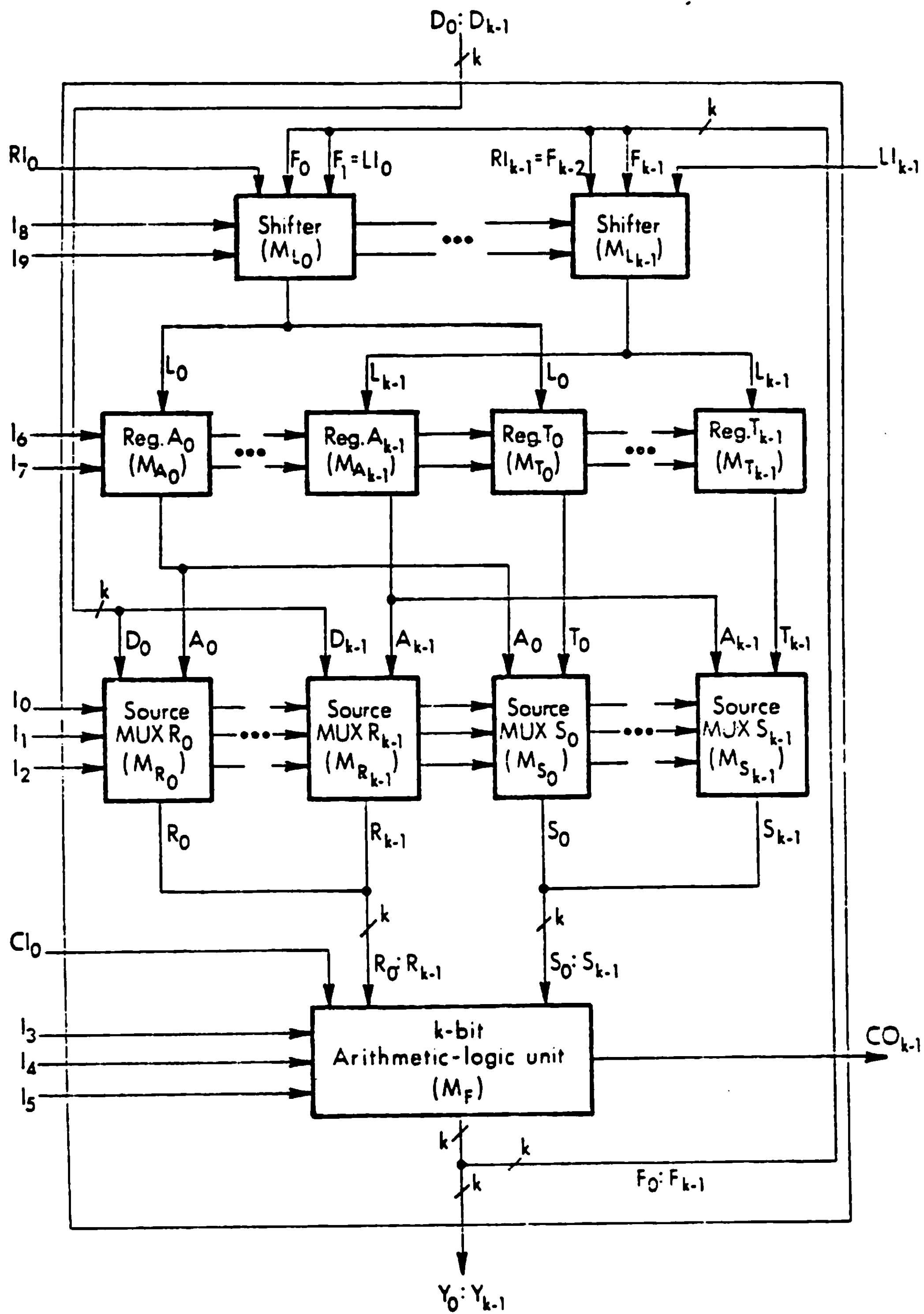


Figure 3.6. k-bit version C^k of cell C

tiplexer modules $M_{R_0}:M_{R_{k-1}}$ and $M_{S_0}:M_{S_{k-1}}$. A complete test sequence for the modules M_{R_i} and M_{S_i} is the same as T_R' (Table 3.3) for the modules M_R and M_S of C with D, A and T replaced by D_i, A_i and T_i respectively. A further consequence of the simple replication scheme used to form C^k, is that the test sequence T_R' for the pair of multiplexers M_{R_i} and M_{S_i} can be applied at the same time (in parallel) to every other pair M_{R_j} and M_{S_j}, where $i \neq j$. Thus the modules $M_{R_0}:M_{R_{k-1}}$ and $M_{S_0}:M_{S_{k-1}}$ can be completely tested by a sequence of only 36 test patterns obtained by extending T_R' to all k module pairs. Similarly a test sequence of length 18 for the modules $M_{A_0}:M_{A_{k-1}}$ and $M_{T_0}:M_{T_{k-1}}$ is obtained from the sequence T_e' of Table 3.6 for the modules M_A and M_T of C. The modules $M_{L_0}:M_{L_{k-1}}$ require 33 test patterns derived from the test sequence of Table 3.5. Now the remaining module M_F of C^k is combinational and has 2k+4 input lines. Hence the number of tests for M_F is 2^{2k+4}. As in the case of C, these tests are very easy to derive. Thus C^k can be completely tested with $36+18+33+2^{2k+4} = 87+2^{2k+4}$ tests. These tests for C^k also detect all multiple-module faults in any one of the replicated module groups $M_{R_0}:M_{R_{k-1}}$, $M_{S_0}:M_{S_{k-1}}$, $M_{A_0}:M_{A_{k-1}}$, $M_{T_0}:M_{T_{k-1}}$, or $M_{L_0}:M_{L_{k-1}}$.

Processor Cells with Scratchpad Memories

Next we consider the tasks of modeling and testing the scratchpad RAMs encountered in bit-sliced processors such as the 2901 and the 3002. Figure 3.7a shows a general scratchpad RAM SR of size $n \times 1$ bits. Any of the existing heuristic methods for memory testing [Breuer and Friedman 1976] may be used to test SR, but the fault coverage obtained cannot readily be estimated. We now show how the circuit and fault models developed in Sections 2.3 and 2.4 may be used to obtain tests for SR.

SR is modeled as n independent 1-bit memory (or register) modules $G_0{:}G_{n-1}$ as shown in Figure 3.7b where the G_i's are treated as primitives. It is assumed that the address decoding logic and the read/write logic are distributed among the n memory modules $G_0{:}G_{n-1}$; see Figure 3.7b. Interaction between the memory modules is not considered, thus excluding some types of pattern sensitive faults [Hayes 1975]. The foregoing assumptions are quite reasonable for memories with small values of n, say $n \leq 16$ which is typical of commercial bit-sliced processors. Also the exhaustive testing of the individual memory modules required by the fault model ensures the detection of any (permanent) functional fault in the memory modules independent of their internal structure.

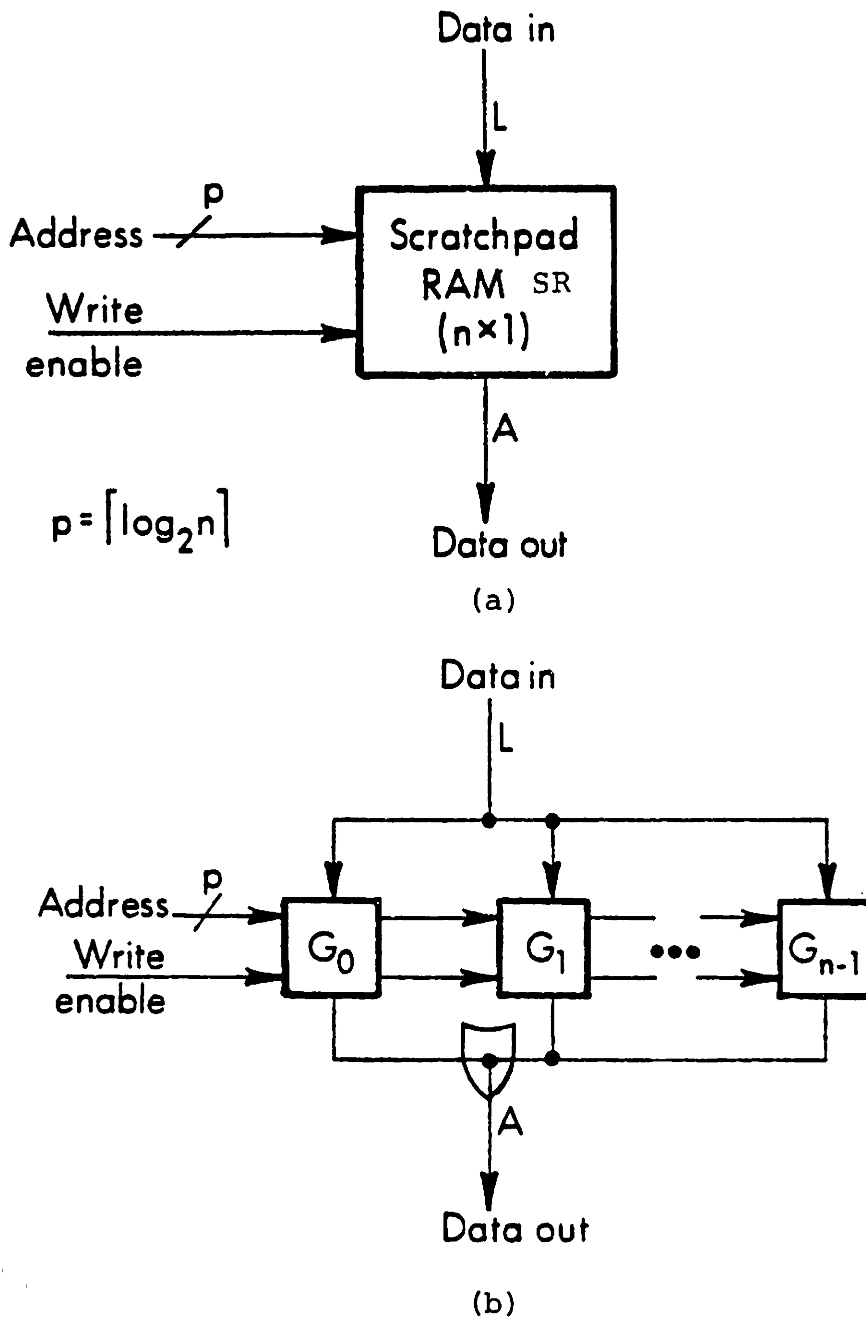


Figure 3.7. (a) An $n \times 1$-bit scratchpad RAM SR
(b) A circuit model for SR

The 1-bit memory module G_i of Figure 3.7b is similar to the 1-bit register modules M_A or M_T of C. Thus the behavior of G_i can be completely described by a state table resembling that of Figure 3.3a. Now G_i has two internal states (0 and 1) and p+2 input lines, where $p = \lceil \log_2 n \rceil$, hence its state table has two rows and $2^{p+2} = 4n$ columns. Using our fault model which treats each G_i as a separate module, the RAM can be tested by verifying separately the state tables of each of the n memory modules. Each state table has 8n entries, therefore a total of 16n test patterns is required, including 8n propagating input patterns for reading the next state reached by G_i. Since at A we can only observe the output signal from one memory module at a time, we require $16n^2$ tests to test an $n \times 1$ RAM. This is not an unreasonably large number of tests in view of the small values of n typically encountered in bit-sliced processors.

The module replication and testing schemes described above can also be extended to a more general $n \times r$ RAM. Such a RAM SR^r can be constructed by replicating the $n \times 1$ RAM circuit SR appearing in Figure 3.7a r times, with address lines connected in common to all r circuits $SR_0{:}SR_{r-1}$. The structure of SR^r is the same as that of the general replicated system M^k shown in Figure 3.5. Again the r individual RAMs $SR_0{:}SR_{r-1}$ can be tested simultaneously with $16n^2$ test patterns.

The foregoing scratchpad RAMs can be used to increase the number of general-purpose registers in the processor cells C or C^k from 1 to n. For example, the modules $M_{A_0}:M_{A_{k-1}}$ of C^k can be replaced by a single $n \times k$-bit scratchpad RAM SR^k. Let $C^{k,n}$ denote the resulting cell. $C^{k,n}$ can be tested in much the same way as C^k. The main difference is that the 18 tests for the modules $M_{A_0}:M_{A_{k-1}}$ of C^k are replaced by $16n^2$ tests for the $n \times k$ scratchpad RAM of $C^{k,n}$. Thus the number of test patterns for $C^{k,n}$ will be $87+2^{2k+4}+16n^2$.

It is interesting to note that the structure of $C^{4,16}$ is quite similar to that of the 4-bit 2901 processor slice. The number of tests for $C^{4,16}$ is 8,279 according to the foregoing testing approach. This is about 35 percent less than the 12,500 test patterns obtained by McCaskill for the 2901 [McCaskill 1976]. His test generation method is heuristic and so cannot guarantee the detection of all single-module faults. On the other hand, the fault coverage achieved here is well-defined by the underlying functional fault model. The 8,279 test patterns obtained for $C^{4,16}$ detect all single-module functional faults and many multiple-module faults.

CHAPTER 4

ARRAYS OF PROCESSOR SLICES

In this chapter arrays of processor cells or slices of type C, C^k and $C^{k,n}$ are examined. It is shown how the tests for the individual cells obtained in the previous chapter can easily be extended to an array. Initially we assume that ripple-carry propagation is used between the cells of an array. The analysis presented here strongly suggests that bit-sliced processors can be modeled as C-testable iterative logic arrays that use ripple-carry propagation. In Section 4.3 we discuss the effects of using a fast carry-propagation scheme like carry lookahead (CLA).

4.1 Arrays of C Cells

Figure 4.1 shows an N-bit processor array PA consisting of N identical cells of type C from Figure 3.1 in cascade. The interconnections between the cells are as follows. The carry lines are cascaded so that $CI_{i+1} = CO_i$ for $i = 0,1,2,\ldots,N-2$, while the 10-bit control bus I, consisting of the lines $I_0:I_9$, is common to every cell in the array. For realizing the shift operations the Y_i output of C_i is connected to the right shift input RI_{i+1} of cell C_{i+1}, and also to the left shift input LI_{i-1} of C_{i-1}, as

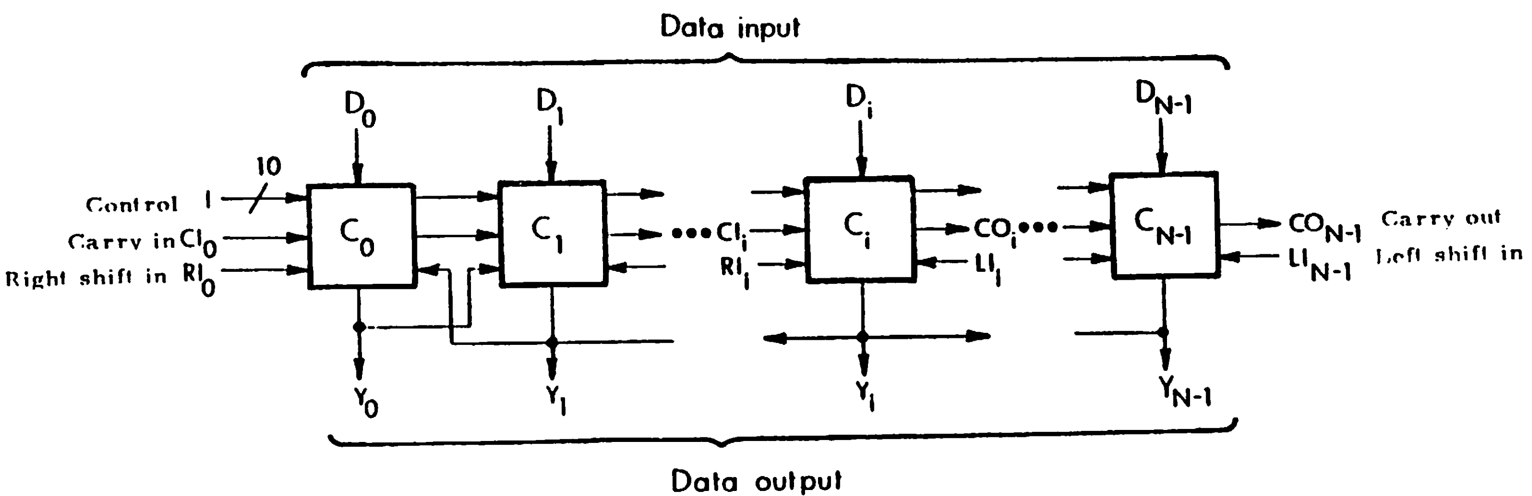


Figure 4.1. Processor array PA composed of N cells of type C

shown in Figure 4.1. The primary inputs of PA are RI_0, LI_{N-1}, CI_0, $I_0{:}I_9$ and $D_0{:}D_{N-1}$, while the primary outputs are $Y_0{:}Y_{N-1}$ and CO_{N-1}. Clearly the interconnections of the array in Figure 4.1 are very similar to those of the 2901 array depicted in Figure 2.4.

Testing the Processor Array

We now discuss test generation for the array PA. The fault model for an individual cell is the same as before, with at most one cell in PA assumed faulty. In testing the array two basic requirements must be met.

(1) It should be possible to apply a complete test sequence for C, such as the sequence T_C obtained in Section 3.3, to every cell in the array irrespective of its position.

(2) It should be possible to propagate the output signals of any cell under test to the primary outputs of the array.

Our approach to testing the array PA is to extend the test sequence T_C for a single cell C to a test sequence T_{PA} for the array, such that T_{PA} satisfies the above two requirements.

The test patterns in the test sequence T_C for C may be grouped into the following two broad classes based on the signal values on the cell interconnection lines,

namely the carry and the shift lines.

1. Test patterns for which the carry-in (CI) and carry-out (CO) signal values are the same, and the shift-in (RI and LI) signal values are don't cares; such test patterns are referred to as *parallel (or P-) tests.*

2. Test patterns called *nonparallel (or $\bar{P}$-) tests*, for which either the CI and CO signal values are different or the shift-in signal values are not don't cares.

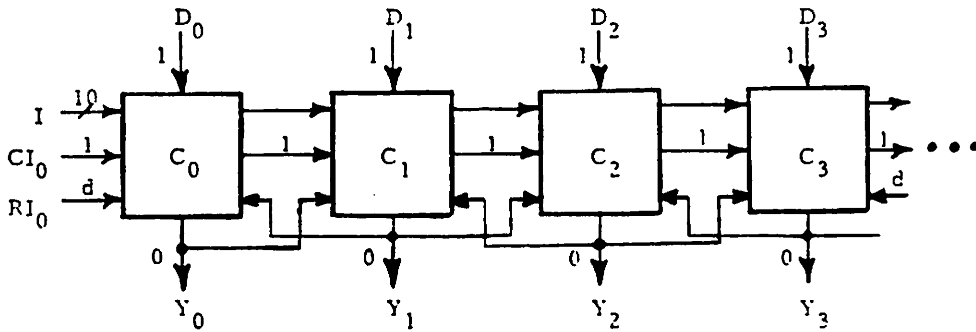
A P-test pattern t does not affect the intercell carry and shift lines. Therefore, when t is applied to any cell C_i of PA, it is possible to apply the same input pattern t to the adjacent cells C_{i-1} and C_{i+1} as well, by making their data input signals identical, i.e., $D_{i-1} = D_i = D_{i+1}$. In other words, it is possible to apply any P-test pattern simultaneously to every cell of PA; hence the name, parallel tests. On the other hand, a $\bar{P}$-test pattern t affects the intercell signals, and hence it is not possible to apply t to every cell at the same time. However, as will be seen later, such test patterns may be applied simultaneously to cells $C_0, C_p, C_{2p}, \ldots$, spaced at regular intervals of p cells along the array. Therefore only p array input patterns are sufficient to apply t to every cell in

PA. (Note that p becomes one for a P-test pattern.) We now explain below how the actual test patterns for the various modules in C given in Tables 3.3 to 3.6 are extended to form the corresponding test patterns for PA.

First consider the test sequence T'_R for the multiplexer modules M_R and M_S of C given in Table 3.3. It is evident that all the input patterns in T'_R are P-test patterns. Hence T'_R can be applied simultaneously to every cell in PA; the corresponding test patterns for the array are easily derived. For example consider the fifth pattern t_5 of T'_R. The corresponding primary input pattern t'_5 that applies t_5 to every cell of the array, is obtained by applying identical D signals to the data input line of every cell.

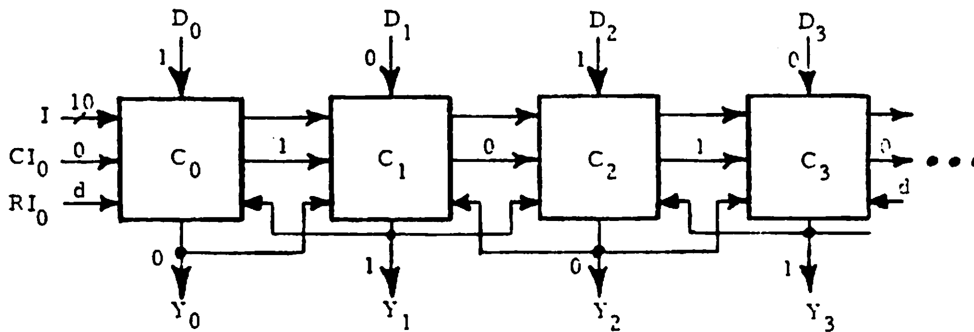
$$t'_5: (RI_0, LI_{N-1}, CI_0, I_0:I_9, D_0:D_{N-1})$$
$$= (d,d,1,1,1,0,0,0,0,1,1,0,0,1^N)$$

The application of t'_5 to the array PA is illustrated in Figure 4.2a. Thus the 36 array test patterns for the array corresponding to the 36 cell test patterns in T'_R are the same as those given in Table 3.3 with CI_0 and $D_0:D_{N-1}$ replacing CI and D, respectively. Similarly, the test sequence T'_e of Table 3.6 that tests completely the sequential modules M_A and M_T of C, can be applied at the same time to every cell of PA, since all its input patterns are



Note: I-bus I_0:I_9 is set to (1, 1, 0, 0, 0, 0, 1, 1, 0, 0).

(a)



Note: I-bus I_0:I_9 is set to (1, 1, 0, 0, 0, 0, 1, 1, 0, 0).

(b)

Figure 4.2. Application of two kinds of test patterns to the cells of the processor array PA: (a) a P-test pattern and (b) a $\bar{P}$-test pattern

P-tests.

Now consider the test patterns for the ALU module M_F given in Tables 3.3 and 3.4. The test patterns of Table 3.3 that are common to the modules M_R, M_S and M_F are P-tests, and are extended to array input patterns in the manner just described. All but six of the input patterns in Table 3.4 are P-tests, and are again easily extended to array input patterns. The remaining six patterns, t_3, t_4, t_7, t_8, t_{11} and t_{12}, are $\bar{P}$-test patterns since $CI \neq CO$. These tests are used for verifying the ALU arithmetic operations f_0, f_1 and f_2; see Table 3.1. However, during the execution of these arithmetic operations it is possible to make the carry signals identical at the carry input lines of every second cell. Hence these $\bar{P}$-test patterns can be applied to alternating cells simultaneously. For example, consider the test patterns t_3 and t_4 in Table 3.4. t_3 can be applied to the even-numbered cells $C_0, C_2, C_4, \ldots$, while at the same time t_4 is applied to the odd-numbered cells $C_1, C_3, C_5, \ldots$, and vice versa. The corresponding array input patterns t'_a and t'_b are as given below.

$$t'_a: (RI_0, LI_{N-1}, CI_0, I_0{:}I_9, D_0{:}D_{N-1})$$
$$= (d,d,0,1,1,0,0,0,0,1,1,0,0,(10)^{N/2})$$

$$t'_b: (RI_0, LI_{N-1}, CI_0, I_0:I_9, D_0:D_{N-1})$$
$$= (d,d,1,0,1,0,0,0,0,1,1,0,0,(01)^{N/2})$$

Figure 4.2b shows how t'_a applies the test patterns t_3 and t_4 to alternating cells. Here it is assumed that the registers $A_0:A_{N-1}$ and $T_0:T_{N-1}$ of cells $C_0:C_{N-1}$ are initialized to $(01)^{N/2}$ and $(10)^{N/2}$, respectively. A similar set of array test patterns can be constructed for the remaining four $\bar{P}$-test patterns t_7, t_8, t_{11} and t_{12}. The resulting eight input patterns for PA are given in Table 4.1. The first two patterns in this table are for initializing the registers $A_0:A_{N-1}$ and $T_0:T_{N-1}$ to the above mentioned values.

Now the remaining tests in T_C, namely the test patterns in Table 3.5 for the shifter module M_L, are all $\bar{P}$-tests so they cannot be applied simultaneously to every cell. For example, to apply test t_3 from Table 3.5 to every cell, we require $(RI_i, LI_i, F_i) = (0,0,1)$ for all i. But by the construction of PA, the line $Y_i = F_i$ is connected to the line LI_{i-1}, i.e., $F_i = LI_{i-1}$, and hence $LI_{i-1} = 0$ and $F_i = 1$ is impossible. Therefore t_3 cannot be applied to every cell at the same time. However, it is possible to apply the $\bar{P}$-tests in Table 3.5 to cells spaced periodically in PA. Consider the three tests t_3, t_4 and t_6 of Table 3.5. A corresponding array input pattern t'_2 given in Table 4.2 applies t_3 to every third cell,

Table 4.1. Test patterns for the processor array PA that apply the $\bar{P}$-test patterns of the ALU module M_F to every cell

No.	Shift Inputs		Carry In	I-bus										D-bus				
	RI_0	LI_{N-1}	CI_0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	D_0	D_1	$D_2 \cdots$	D_{N-2}	D_{N-1}
1	d	d	d	0	1	1	1	1	0	0	0	0	0	0	1	0...	0	1
2	d	d	d	0	1	1	1	1	0	1	0	0	0	1	0	1...	1	0
3	d	d	0	1	1	0	0	0	0	1	1	0	0	1	0	1...	1	0
4	d	d	1	0	1	0	0	0	0	1	1	0	0	0	1	0...	0	1
5	d	d	0	1	1	0	1	0	0	1	1	0	0	0	1	0...	0	1
6	d	d	1	0	1	0	1	0	0	1	1	0	0	1	0	1...	1	0
7	d	d	0	0	1	0	0	1	0	1	1	0	0	1	0	1...	1	0
8	d	d	1	1	1	0	0	1	0	1	1	0	0	0	1	0...	0	1

Table 4.2 Array input patterns for testing the shifter module M_L in every cell of the processor array PA

No.	Shift Control		Shifter Inputs			Data in		
	I_9	I_8	RI_i	F_i	LI_i	D_{i-1}	D_i	D_{i+1}
1	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	1	0
3	0	0	1	0	0	1	1	0
4	0	0	1	1	0	0	1	0
5	0	0	0	0	1	1	1	1
6	0	0	0	1	1	0	1	0
7	0	0	1	0	1	1	1	0
8	0	0	1	1	1	0	1	0
9	0	1	0	0	0	1	1	1
10	0	1	0	1	0	0	1	0
11	0	1	1	0	0	1	0	1
12	0	1	1	1	0	1	0	0
13	0	1	0	0	1	0	1	0
14	0	1	0	1	1	1	1	1
15	0	1	1	0	1	0	0	0
16	0	1	1	1	1	0	0	1
17	1	0	0	0	0	1	1	1
18	1	0	0	1	0	0	1	0
19	1	0	1	0	0	0	0	0
20	1	0	1	1	0	1	1	1
21	1	0	0	0	1	1	0	0
22	1	0	0	1	1	0	0	1
23	1	0	1	0	1	0	1	1
24	1	0	1	1	1	1	0	0
25	1	1	0	0	0	1	1	1
26	1	1	0	1	0	0	1	0
27	1	1	1	0	0	1	0	0
28	1	1	1	1	0	1	1	0
29	1	1	0	0	1	0	0	1
30	1	1	0	1	1	0	1	1
31	1	1	1	0	1	1	0	1
32	1	1	1	1	1	1	1	1

Notes: In all cases

(i) $(I_7, I_6, I_5, I_4, I_3, I_2, I_1, I_0, CI_0) =$ $(0,0,1,1,0,0,1,0,d)$

(ii) $RI_i = F_{i-1} = LI_{i-2}$ and $D_i = D_{i+3}$ for all i

i.e., to cells $C_i, C_{i+3}, C_{i+6}, \ldots$. At the same time it applies tests t_4 and t_6 to cells $C_{i+1}, C_{i+4}, C_{i+7}, \ldots$, and to cells $C_{i+2}, C_{i+5}, C_{i+8}, \ldots$, respectively. In the same way all the other array input patterns of Table 4.2 each apply three test patterns of Table 3.5 to consecutive cells.

We have now accounted for every test pattern in T_C. Hence the processor array PA can be completely tested by a test sequence T_{PA}, of length 125, which is a concatenation of all the foregoing array input patterns.

<u>Theorem 4.1</u>: An N-bit processor array of type C cells is completely testable by the test sequence T_{PA} of length 125 independent of the array size N. Hence the processor array is C-testable.

The test sequence T_{PA} derived under the single-cell fault assumption can also detect many multiple-cell faults. Since the multiplexer modules $M_{R_0}:M_{R_{N-1}}$ in the cells $C_0:C_{N-1}$, respectively, are all tested simultaneously by T_{PA}, any multiple fault affecting these modules is readily detected. Similarly, all multiple-cell faults affecting the modules $M_{S_0}:M_{S_{N-1}}$, $M_{A_0}:M_{A_{N-1}}$ and $M_{T_0}:M_{T_{N-1}}$ in the cells $C_0:C_{N-1}$ are also detected by T_{PA}. Finally, T_{PA} can detect all stuck-line faults on the common I-bus lines.

IC Implementation of a Processor Array

A 4-bit version of the above C-testable processor array PA was implemented on an NMOS integrated circuit chip using the Caltech/Xerox VLSI design software [Mead and Conway 1980]. Figure 4.3 shows the circuit layout of the processor chip. The four identical 1-bit processor cells in cascade can be easily identified in the layout. The silicon area consumed by the chip is 2.75 mm × 3.25 mm = 8.94 sq.mm and it has about 1,250 transistors. The chip was easily tested using the above test sequence T_{PA}. Details of the logic and circuit design of the processor chip are given in Appendix B.

4.2 Arrays of C^k and $C^{k,n}$ Cells

The foregoing analysis of the processor array PA of type C cells is equally applicable to arrays of C^k or $C^{k,n}$ cells. Let PA^k and $PA^{k,n}$ denote processor array of type C^k and $C^{k,n}$ cells, respectively. Now compare the structures of PA with $N = k$ (Figure 4.1), and the cell C^k (Figure 3.6). In C^k, the five modules M_L, M_A, M_T, M_R and M_S of C have been replicated k times as shown in Figure 3.6. These five modules are also replicated in the cascade structure of PA. Hence the test patterns for these replicated modules in the array PA^k of size N_1 are the

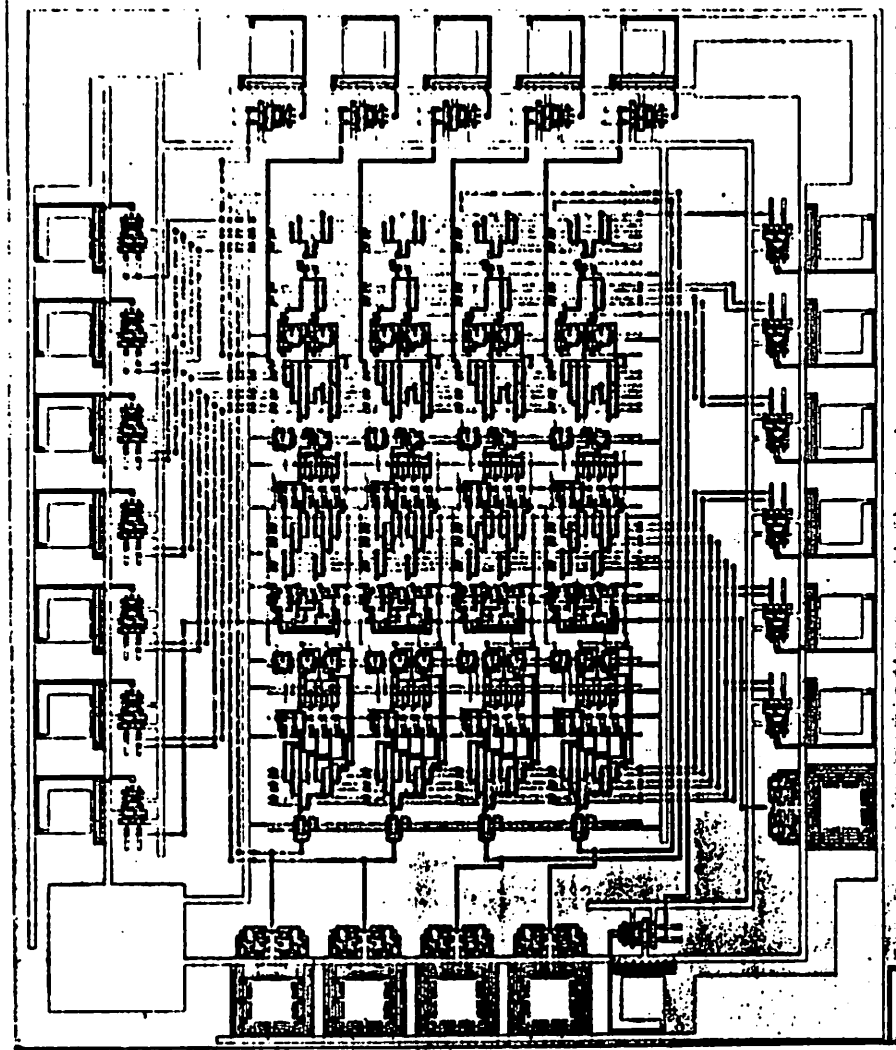


Figure 4.3. Chip layout of a 4-bit processor array of type C cells.

same as the corresponding test patterns in T_{PA} for PA of size N_1k. The tests for the remaining module, namely the k-bit ALU M_F in C^k, can also be extended into array input patterns for PA^k. This extension is similar to that of the ALU test patterns for PA discussed in the previous section. Hence a complete test seqeunce T_{PA^k} for the array PA^k is easily derived. A test sequence $T_{PA^{k,n}}$ for the array $PA^{k,n}$ is very similar to T_{PA^k}. The main difference is that the tests for the register modules $M_{A_0}:M_{A_{k-1}}$ of C^k are replaced by the tests for the scratchpad RAM modules $SR_0:SR_{k-1}$ of $C^{k,n}$ (see Section 3.5). Again the lengths of the T_{PA^k} and $T_{PA^{k,n}}$ test sequences are constant, independent of the array sizes.

Theorem 4.2: Processor arrays of identical cells of type C, C^k or $C^{k,n}$ are C-testable.

It is not unreasonable to expect any bit-sliced processor array, having basic features similar to the foregoing processor arrays, to be C-testable also. In fact, as will be evident in the subsequent chapters, if only addition and shift operations are responsible for the intercell communication in a processor array, then it can be concluded that such an array is C-testable. Thus in analyzing the testing requirements of bit-sliced processors, it is extremely useful to model them as C-testable iterative logic arrays.

4.3 Processors with Carry-Lookahead

The processor arrays studied in the previous sections use ripple-carry propagation between the cells. It is well-known that this type of carry propagation is relatively slow. To alleviate this problem the extended processor cells C^k and $C^{k,n}$ developed in Section 3.5 include speed-up techniques for handling carry propagation within a single cell. We now discuss the inclusion of a fast carry propagation scheme between the cells of an ILA.

Several high-speed carry propagation schemes have been proposed, including the conditional-sum, carry-completion and carry-lookahead schemes [Hwang 1978]. Carry lookahead (CLA) is probably the most widely used of these techniques. Commercial IC chips implementing CLA are available, e.g., the 2902 carry-lookahead generator chip of the AMD 2900 family [Advanced Micro Devices 1979].

The CLA approach to reducing the carry propagation time generates the carry input signal to a cell directly from the data inputs of the preceding cells, rather than allowing carry signals to ripple through every cell. This can be understood with the help of Figure 4.4 which shows a simple 4-bit adder array that uses CLA. The cells U_0, U_1, U_2 and U_3 are 1-bit adder cells containing some additional logic Q (not explicitly shown) that computes the carry "generate" and the carry "propagate" signals

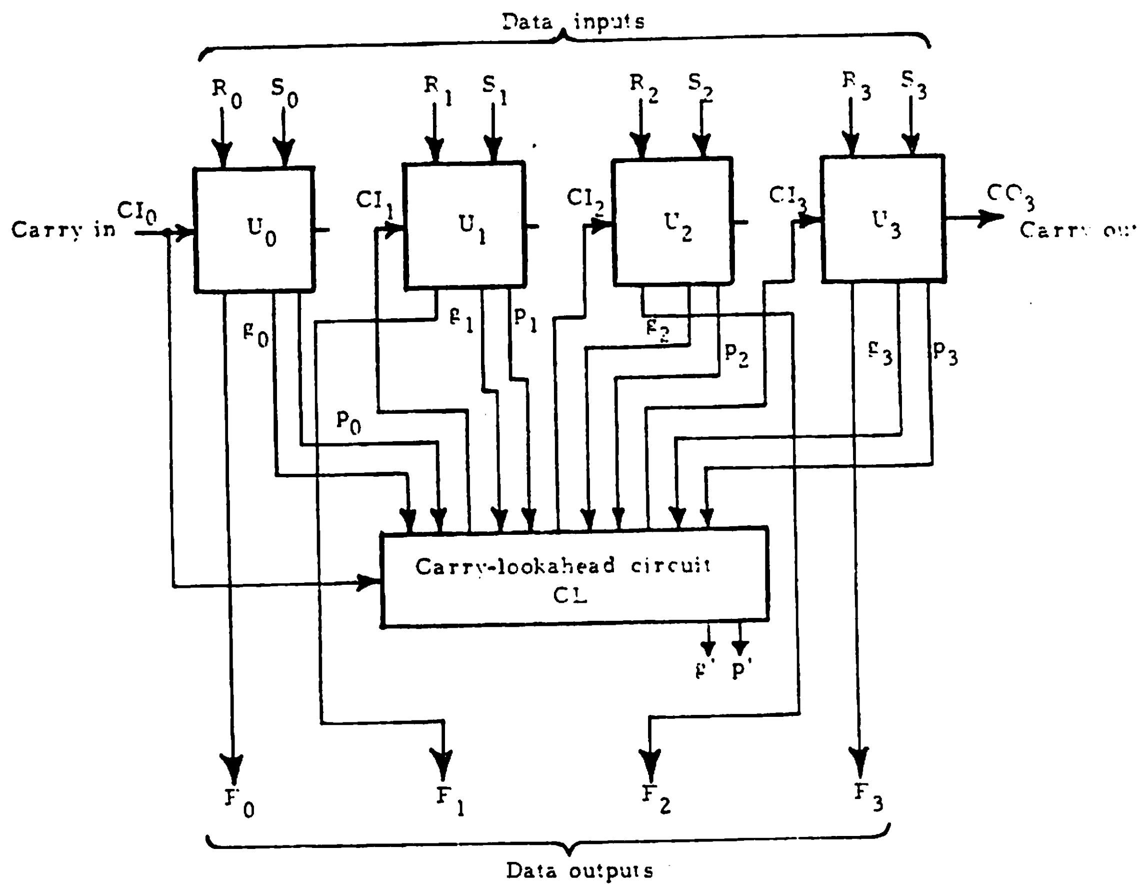


Figure 4.4. A 4-bit adder array using carry-lookahead

g_i and p_i. The cell U_i can be viewed as the portion of the ALU module M_F of the processor cell C (Section 3.1) that performs addition; the signal names in Figure 4.4 are selected to reflect this view. The carry input signals to all adder cells except U_0 are generated separately by an external carry-lookahead circuit CL which uses the g_i and p_i signals shown in Figure 4.4.

As their names suggest, the signals g_i and p_i represent the logical conditions under which a carry signal is generated in and propagated through, respectively, the ith bit position. They are defined by the following logic equations.

$$g_i = R_i S_i$$
$$p_i = R_i \oplus S_i \quad (4.1)$$

The carry input signal CI_{i+1} of cell U_{i+1} is then defined by

$$CI_{i+1} = g_i + p_i CI_i \quad (4.2)$$

Applying this iterative equation to the 4-bit adder circuit of Figure 4.4 we get the following equations for the carry-in signals.

$$CI_1 = g_0 + p_0 CI_0$$

$$CI_2 = g_1 + p_1 g_0 + p_1 p_0 CI_0 \tag{4.3}$$

$$CI_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 CI_0$$

$$CI_4 = CO_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 CI_0$$

The signals g_i and p_i, and CI_i can all be generated by means of 2-level logic circuits in U_i and CL, respectively. If t_d is the propagation delay of a 2-level logic circuit then the total delay of the 4-bit CLA adder of Figure 4.4 is $3t_d$, compared to $4t_d$ in the case of a 4-bit ripple-carry adder.

Although, in principle, a CLA adder can be of any word size n, the size of the CLA circuit CL is limited in practice by the complexity of Equations (4.3). Usually no more than $k = 8$ consecutive carry signals $CI_1{:}CI_k$ may be generated in a single CLA circuit. To handle larger values of n, we can partition n into N k-bit groups, each containing a CLA circuit. Ripple-carry propagation is then used between the N groups. The worst-case delay time involved here is $(N+2)t_d$ [Hwang 1978], which is better than the delay nt_d of an n-bit ripple-carry adder. The processor array PA^k of type C^k cells discussed in the previous section, uses this approach. For large N, the delay due to the ripple carry between the cells or groups, can

be quite significant. Again CLA can be used to speed up the carry signal generation between the k-bit groups as well. For this purpose, the k-bit (adder) groups and the CLA circuit CL (see Figure 4.4) generate group carry-generate and carry-propagate signals g′ and p′ according to the equations

$$g' = g_{k-1}+p_{k-1}g_{k-2}+p_{k-1}p_{k-2}g_{k-3}+\cdots+p_{k-1}p_{k-2}\cdots p_1 g_0$$
$$p' = p_{k-1}p_{k-2}\cdots p_1 p_0 \qquad (4.4)$$

Consider a new processor cell D^k which is the earlier processor cell C^k of Section 3.5 with an additional logic circuit Q′ that generates the CLA signals g′ and p′. The g_i and p_i signals that produce g′ and p′ according to Equations (4.4) are in turn obtained from the ALU operands $R_0{:}R_{k-1}$ and $S_0{:}S_{k-1}$ using Equations (4.1). Therefore Q′ is a combinational logic circuit with 2k inputs and two outputs. This is illustrated in Figure 4.5 which shows a block diagram of the processor cell D^k.

Figure 4.6 depicts a 16-bit processor PD^4 realized using four D^4 cells, and the CLA circuit CL of Figure 4.4 for generating the carry input signals CI_4, CI_8 and CI_{12}. The worst-case processing time here is $5t_d$, compared to $(4+2)t_d = 6t_d$ in the case of the processor array PA^4 composed of four C^4 cells in cascade. We now show how tests are generated for PD^4.

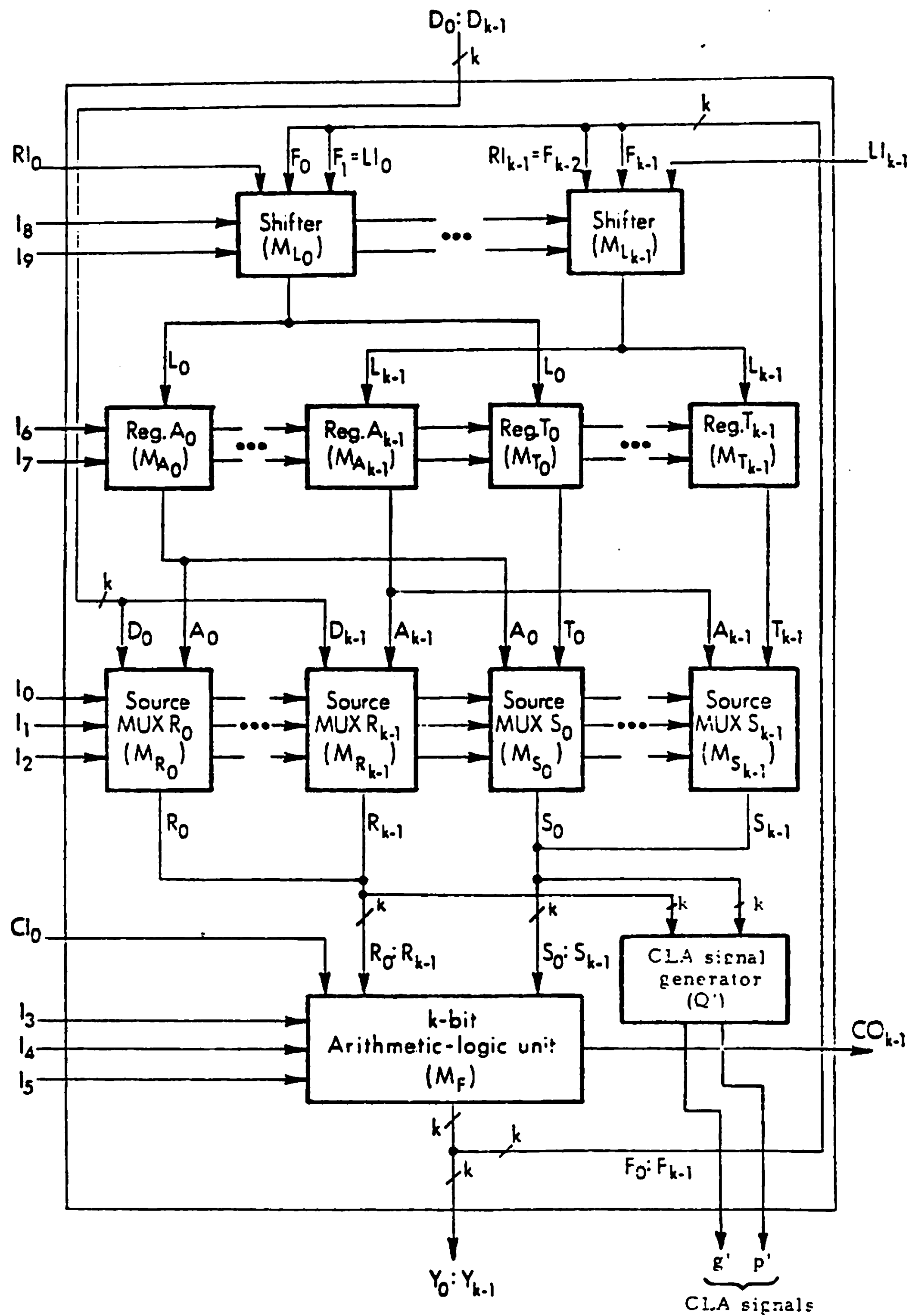


Figure 4.5. Processor cell D^k containing carry-lookahead logic

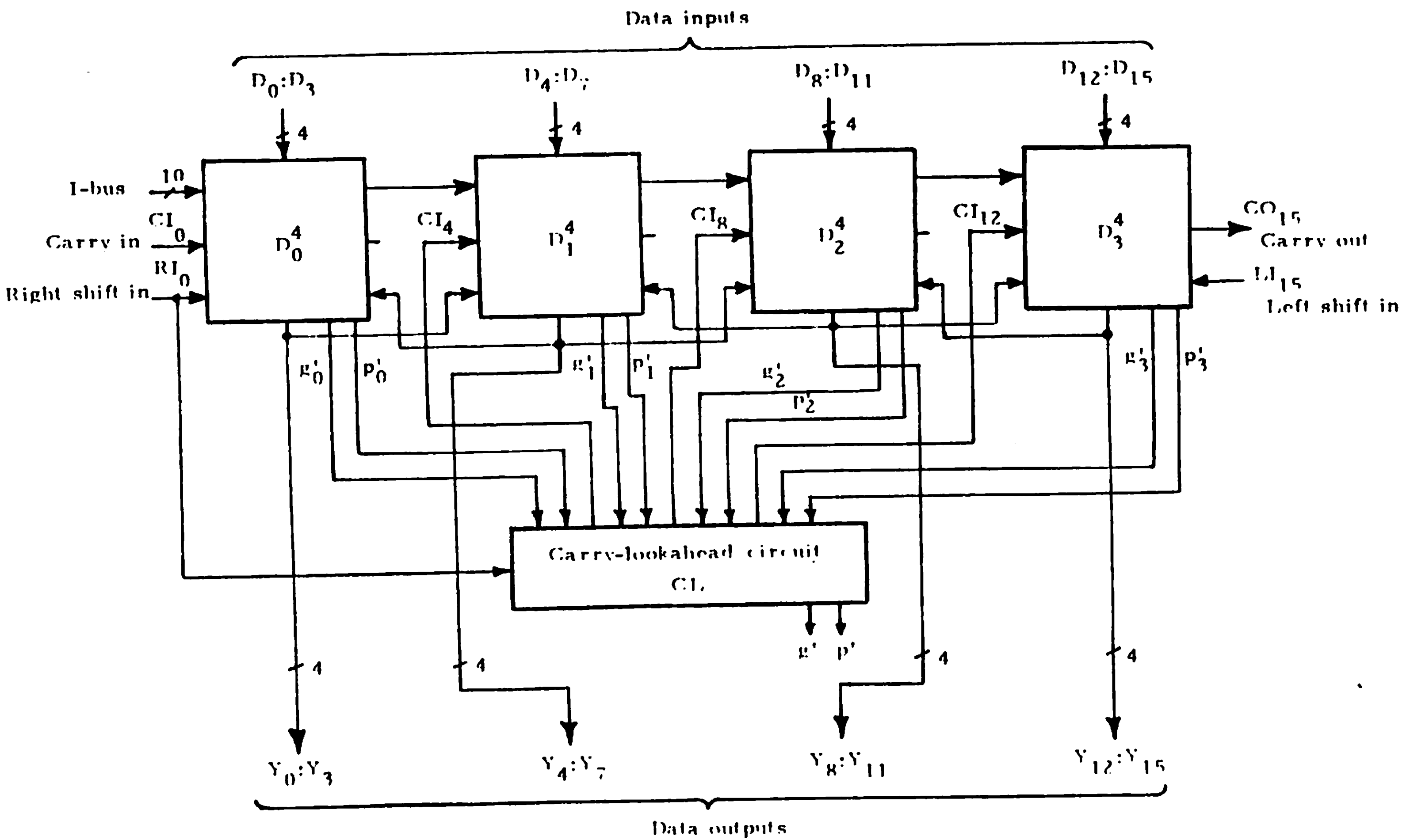


Figure 4.6. A 16-bit processor using carry-lookahead

Testing the Processor PD^4

The cell D^4 used in PD^4 consists of C^4 along with a new module Q'; see Figure 4.5. Therefore the processor array PD^4 of Figure 4.6 is PA^4 with the addition of five new modules, namely, four Q' modules in the four D^4 cells, and the CLA module CL. Note that both PD^4 and PA^4 have identical input-output behavior; their only difference is in the implementation of the carry propagation between the individual cells. Therefore the 8,279 test patterns for the C-testable array PA^4 obtained in Section 4.2 can test completely all modules of PD^4 except the above-mentioned five new modules. It is interesting to note here that although the use of a CLA scheme destroys the ILA structure of the processor PD^4, it allows the original array test patterns to be used for testing PD^4. The five additional modules are handled separately as follows.

Consider the module Q' of the leftmost cell D_0^4 of PD^4. It has eight inputs $R_0{:}R_3$ and $S_0{:}S_3$ as shown in Figure 4.5, and hence is completely tested by a test sequence of length 2^8. These 2^8 input patterns are easily applied to the input lines $R_0{:}R_3$ and $S_0{:}S_3$ which also provide the ALU operands in D^4 or C^4. The two output signals g'_0 and p'_0 of Q' are propagated simultaneously to the two outputs g' and p' of CL, respectively, by having $p'_1 = p'_2 = p'_3 = 1$ and $g'_1 = g'_2 = g'_3 = 0$ (cf. Equations (4.4) with $k = 4$). We are assuming

here that g' and p' are observable primary outputs. Similarly the Q' modules of the remaining three D^4 cells are completely tested one by one by three different test sequences of length 256 each. The remaining module of PD^4, namely the CLA module CL, has nine easily controlled inputs CI_0, $g'_0:g'_3$ and $p'_0:p'_3$. The carry signals CI_4, CI_8 and CI_{12} generated by CL are made observable at Y output lines Y_4, Y_8 and Y_{12} respectively, by selecting any arithmetic function of the ALU. The other two outputs g' and p' are directly observable primary outputs of PD^4. Thus CL can be tested completely by a sequence of length 2^9. Let T_{PD^4} denote the test sequence obtained by concatenating the foregoing sequences for PA^4, the Q' modules, and CL. We therefore have the following result.

<u>Theorem 4.3</u>: The 16-bit processor PD^4 of Figure 4.6 with CLA propagation between the cells is completely testable by the above test sequence T_{PD^4} of length 9,815.

The foregoing approach to CLA design and test generation can be extended to processor arrays larger than PD^4. In general, test sequences for such systems are obtained by simply concatenating appropriate test sequences for the new logic modules to those of the (C-testable) processor arrays PA^k or $PA^{k,n}$ examined in Section 4.2.

CHAPTER 5

C-TESTABLE ARRAYS

The analysis of bit-sliced processors presented in Chapter 4 indicates that for testing purposes such bit-sliced systems can be usefully modeled as C-testable arrays, i.e., arrays that are testable with a constant number of test patterns irrespective of the number of cells. The basic concepts of C-testable arrays are examined in this chapter by analyzing general unilateral arrays of combinational cells. A detailed study of C-testability in different classes of arrays is presented in subsequent chapters. Throughout this thesis only one-dimensional iterative logic arrays (ILAs) are considered since thay are sufficient to model the bit-sliced systems of interest here.

5.1 Classification of Arrays

We now classify one-dimensional ILAs based on their structure and certain functional properties. The general structure of the arrays under consideration is depicted in Figure 5.1. The various signals associated with a basic cell L of this array are identified as follows. Y

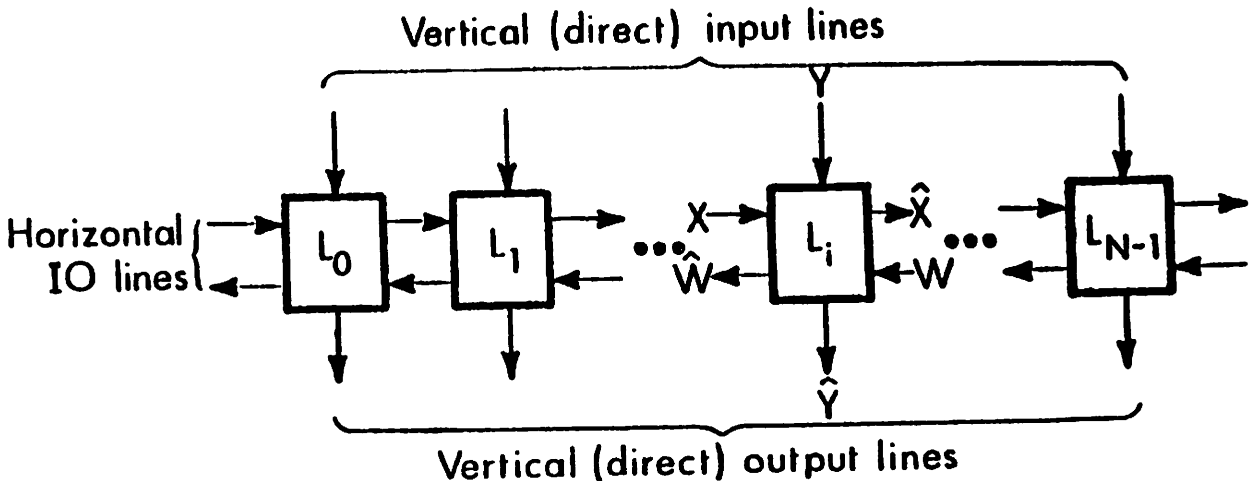


Figure 5.1. General one-dimensional iterative logic array (ILA) structure

and $\hat{Y}$ are the vertical direct input and output signal sets respectively. $X = \hat{X}$ and $W = \hat{W}$ are the right and left horizontal signal sets respectively. S denotes the set of internal states of L. Arbitrary members of X, $\hat{X}$, W, $\hat{W}$, Y, $\hat{Y}$ and S are denoted by x, $\hat{x}$, w, $\hat{w}$, y, $\hat{y}$ and s, respectively. Let $|X| = |\hat{X}| = n$, $|W| = |\hat{W}| = p$, and $|Y| = m$. These notations are very similar to those used in the past by Kautz and others [Kautz 1967, Friedman and Menon 1971]. The cells can communicate in two ways: left to right via the X lines, and right to left via the W lines.

We use the following three characteristics of the basic cell L to classify ILAs.

1. Input-output behavior: L is combinational if its present output signals depend on its present input signals only, otherwise it is sequential. An array is said to be *combinational (sequential)* if its basic cell is combinational (sequential).

2. Direction of horizontal signal flow: L can have just one of the two horizontal sets of lines X and W, or it can have both. In the first case, the signal flow between the cells of the array is in only one direction, either to the left or to the right, and the array is

called *unilateral*. In the latter case signal flow exists in both the directions and the array is said to be *bilateral*.

3. Existence of direct (vertical) output lines: An array may or may not possess direct output lines depending on whether L has $\hat{Y}$ output lines. As will be seen later, the presence or absence of such directly observable output lines plays an important role in testing an array.

Table 5.1 lists all eight ILA classes obtained from the above three characteristics. It also includes references to prior research on the testability aspects of these arrays. This research is reviewed in the next section.

5.2 Previous Work

Various authors have studied the testing of iterative logic arrays since the 1960s, but only a few have considered C-testable arrays as indicated in Table 5.1. In his classic 1967 paper Kautz first presented some fundamental results that are very useful in analyzing C-testability in ILAs [Kautz 1967]. For example, he has characterized optimally testable combinational arrays, i.e., arrays for which there exist fault-detecting test sets that are of minimal size independent of the array

Table 5.1. A classification of iterative logic arrays

Array Class No.	Combinational or Sequential	Unilateral or Bilateral	Direct Outputs	Prior Work
1	Combinational	Unilateral	Absent	Kautz 1967, Landgraff and Yau 1971, Friedman and Menon 1971, Friedman 1973*, Tamamoto and Takaba 1974*, Prasad and Gray 1975, Parthasarathy and Reddy 1979*
2	Combinational	Bilateral	Absent	None known
3	Sequential	Unilateral	Absent	None known
4	Sequential	Bilateral	Absent	Gray and Thompson 1978
5	Combinational	Unilateral	Present	Kautz 1967, Landgraff and Yau 1971, Turcat and Verdillon 1976, Dias 1976*, Parthasarathy and Reddy 1979*
6	Combinational	Bilateral	Present	None known
7	Sequential	Unilateral	Present	Breuer 1968, Sung 1976
8	Sequential	Bilateral	Present	None known

Note: References marked by asterisks consider C-testable arrays.

size. This topic is discussed later in this chapter.

In 1973 Friedman defined the concept of C-testability [Friedman 1973]. He studied only unilateral combinational arrays without direct outputs, i.e., Class 1 arrays. General necessary and sufficient conditions are given in [Friedman 1973, Theorem 1], which directly reflect the basic requirements of C-testability. However, they are not linked directly to the functional properties of the basic cell. Friedman also gave some simple sufficient conditions for C-testability, which he used as the basis of a design modification scheme to make any Class 1 array C-testable. As will be discussed later in Section 6.3 this scheme has some minor errors and is relatively expensive, since it requires a large amount of additional hardware. Tamamoto and Takaba have also studied Class 1 arrays, and have proposed a different design modification scheme [Tamamoto and Takaba 1976] that is even more expensive than Friedman's technique.

Next Dias studied C-testability rather indirectly while developing a procedure to verify the truth-table of Class 5 arrays [Dias 1976]. Dias' method produces a test set for verifying the truth-table of the array such that the test set size is independent of the array size; hence the array is C-testable. He characterized the C-testability of certain array types, and also presented a modification method to make any Class 5 array C-testable.

More recently, Parthasarathy and Reddy have extended the results of Kautz and Friedman by introducing concepts they term one-step testability and one-step C-testability [Parthasarathy and Reddy 1979]. These new concepts are special cases of testability and C-testability, respectively, and are introduced mainly to simplify test pattern generation and to reduce the number of test patterns needed. They also discuss certain aspects of fault location in C-testable combinatinal arrays.

As far as we know, there is no published work on C-testable sequential arrays. In Chapter 8 we consider a special type of sequential array to which the results on combinational arrays can easily be extended.

In the remaining sections of this thesis, unless otherwise mentioned, the term array or ILA will refer to the particular array class or classes from Table 5.1 being discussed in that section.

5.3 Basic Approach to C-testing

In this section we discuss the basic properties of C-testability in unilateral combinational arrays, i.e., the Class 1 and Class 5 arrays of Table 5.1. The array structure considered here is the same as in Figure 5.1 with the horizontal W lines removed. In analyzing such combinational arrays we make use of the well-known analogy between a one-dimensional unilateral array of identical

combinational cells and a synchronous sequential machine viewed at successive clock periods [McCluskey 1958]. In the following subsection we examine this analogy and some basic definitions that follow from it.

Analogy with Synchronous Sequential Machines

Consider the conventional Huffman model of a synchronous sequential machine SM shown in Figure 5.2a. It consists of two modules: a combinational logic circuit L and a memory module M. L generates the present direct output signal $\hat{Y}$ as well as the next-state $\hat{X}$ based on the present direct input signal Y and the present internal state X of SM. The memory module M is a simple storage element, typically a set of flip-flops, that holds the present state information of SM. The output signal $\hat{X}$ from L is fed to M and is written into it by a clock pulse which, as is customary, is not shown in the figure. The output of M is then fed back to L as the X input; see Figure 5.2a. In other words, the $\hat{X}$ output signal of L is presented at its X input after a delay of one clock period. This feeding back of the $\hat{X}$ signal in successive clock periods can be viewed as (logically) equivalent to having a cascade of identical L-type combinational cells in space, as depicted in Figure 5.2b. The two circuits of Figure 5.2 are functionally equivalent in the following sense. Consider an

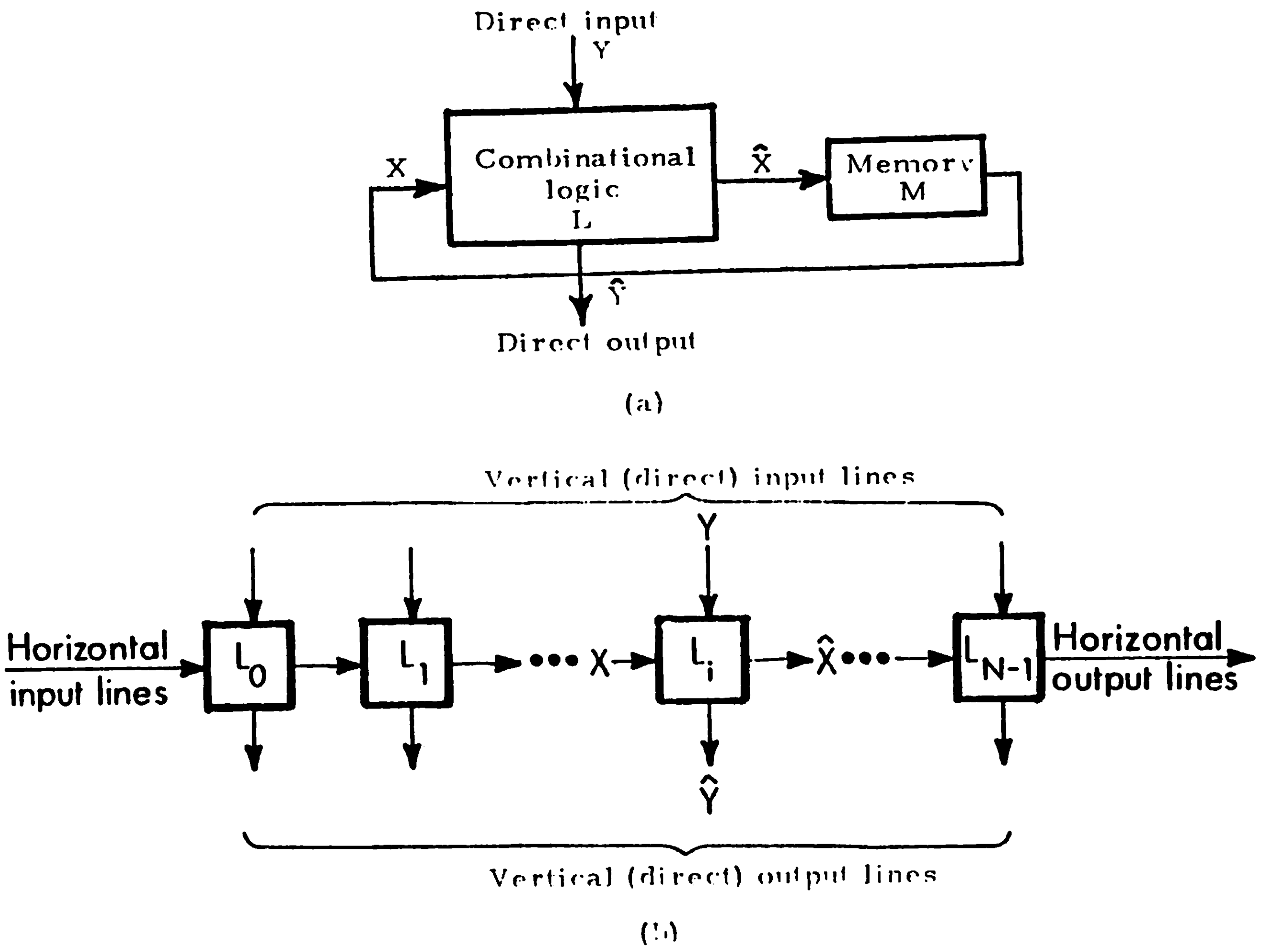


Figure 5.2. (a) Huffman model of a synchronous sequential machine. (b) The equivalent one-dimensional unilateral combinational array

input sequence $y(0)y(1) \cdots y(N-1)$ applied to the sequential machine SM whose initial state is $x(0)$. Let $\hat{y}(0)\hat{y}(1) \cdots \hat{y}(N-1)$ be the corresponding output sequence generated by SM, and let $x(N)$ be the final state reached. Then the cell L_i in the corresponding iterative combinational array has the signal $y(i)$ applied to its Y input, for $i = 0,1,\ldots, N-1$, while $x(0)$ is applied to the X input of L_0. The output signal $\hat{y}(i)$ is generated at the $\hat{Y}$ output lines of the cell L_i. In other words, the cell L_{i-1} in the array is the combinational module L of SM viewed between the $(i-1)^{th}$ and the ith clock pulses, i.e., during the $(i-1)^{th}$ clock period. The X input signal of L_0 is the initial internal state of SM, and the $\hat{X}$ output signal of L_{i-1} is the final state reached by SM after i clock pulses. The combinational module L is referred to as the *basic cell* of the array with X as its set of "states."

Example 5.1: Consider a 1-bit serial adder which is a synchronous sequential machine like that of Figure 5.2a. Its combinational module L is a full adder circuit whose X and $\hat{X}$ signals form the carry-in and carry-out signals c and $\hat{c}$, respectively. The Y input consists of the two 1-bit operands a and b, and the $\hat{Y}$ output is the sum s. The memory module M is a single flip-flop. The equivalent iterative array circuit of this serial adder is a ripple-carry adder with a full adder as the basic cell; such a

circuit appears in Figure 5.2b. A ripple-carry adder is a practical example of an ILA and will be referred to many times in this thesis. As will be seen later, it also has very good testability characteristics. □

We now define some basic terms associated with combinational arrays that are derived from the above analogy with sequential machines.

Definition 5.1: An $n \times m$ table whose rows represent X and are called the states of the basic cell, and whose columns represent Y as shown in Figure 5.3 is called the *flow table* of the basic cell. This is analogous to the state table of a synchronous sequential machine. A typical entry e in row x_i and column y_j in the table is denoted by $(\hat{x}_a, \hat{y}_b)$ where $\hat{x}_a$ is the $\hat{X}$ output or the next-state function value $\hat{x}(x_i, y_j)$ and $\hat{y}_b$ is the direct output function value $\hat{y}(x_i, y_j)$. The next-state entry also specifies a (state) transition in the basic cell. A *transition* τ: $x_i \xrightarrow{y_j} \hat{x}_a$ implies that under the input $(X,Y) = (x_i, y_j)$ the $\hat{X}$ output of the basic cell is $\hat{x}(x_i, y_j) = \hat{x}_a$.

Definition 5.2: A directed graph whose nodes represent the set of states X of a flow table, and whose edges represent the transitions defined by the flow table, is called a *flow diagram*. This is analogous to a state diagram of

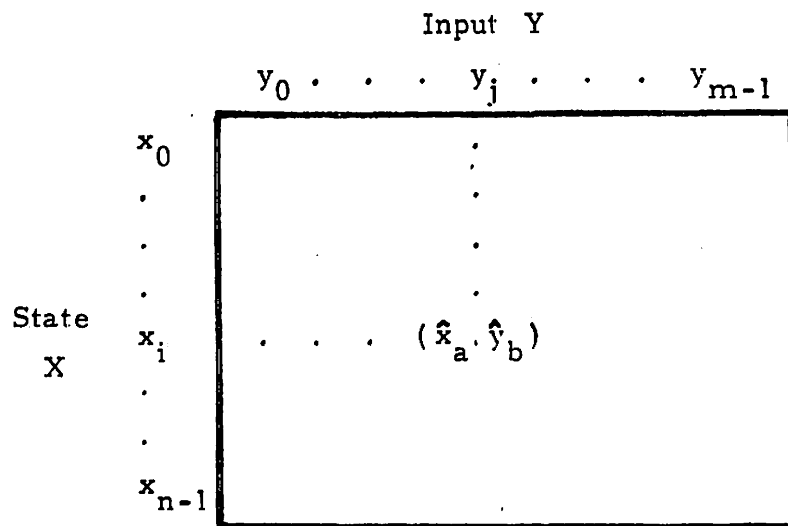


Figure 5.3. Flow table of a basic cell

a sequential machine. The edges of the flow diagram are labeled a/b where a is the Y input signal and b is the $\hat{Y}$ output signal associated with the transition:

Example 5.1 (cont'd): The basic cell of a ripple-carry adder, which is a full adder, its flow table and its flow diagram are shown in Figure 5.4. The edge labeled 00/1 from the node 1 to the node 0 denotes the transition $1 \xrightarrow{00} 0$ with the direct (sum) output of 1.

General Testing Approach

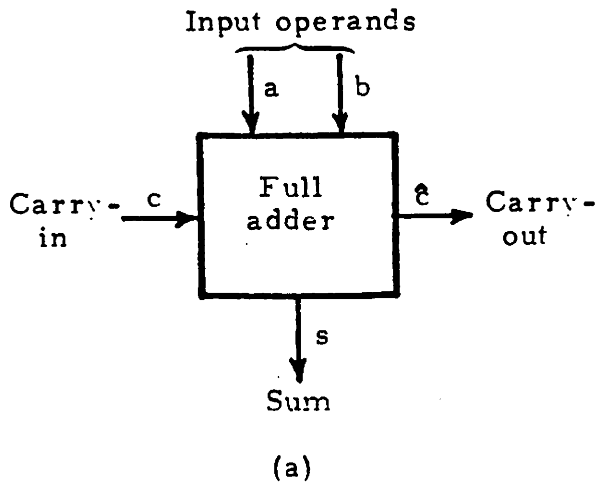
The fault model used here is that defined in Section 2.4 with every cell in the array treated as a single module. In addition to the usual single-module fault assumption, we assume that the Y input lines do not fan out to more than one cell in the ILA.

Definition 5.3 [Kautz 1967]: An array is *testable* if it is possible to detect the presence of any faulty cell in the array.

Definition 5.4 [Friedman 1973]: An N-cell array is *C-testable* if it is testable with a fixed number of test patterns, independent of N, the array size.

The general requirements for testing an ILA are:

(i) it should be possible to apply all input combinations,



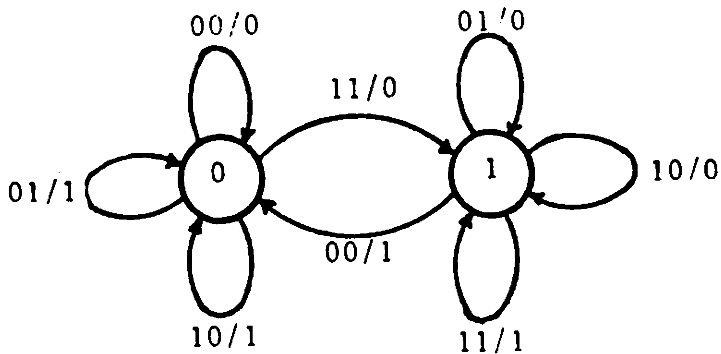
(a)

Input ab

State c	00	01	10	11
0	(0, 0)	(0, 1)	(0, 1)	(1, 0)
1	(0, 1)	(1, 0)	(1, 0)	(1, 1)

$(\hat{c}, s)$

(b)



Edge labels denote ab/s

(c)

Figure 5.4. (a) A full adder cell, (b) its flow table, and (c) its flow diagram

i.e., apply all transitions, to a cell independent of its position in the array, and (ii) at the same time, it should be possible to propagate the $\hat{X}$ output signal of a cell under test to the primary outputs of the array. The following two fundamental results on ILA testing are due to Kautz [Kautz 1967].

<u>Lemma 5.1</u>: An ILA is testable if and only if the cell flow table is such that no two rows are identical, and each state appears at least once as a next-state entry.

<u>Lemma 5.2</u>: A testable ILA of N cells with an $n \times m$ cell flow table can be completely tested by a test set whose size is at most $mn[(N-1)(n-1)+1]$.

We wish to derive test sets whose sizes are independent of the array length. Our approach is to design test inputs to the array such that a particular cell transition τ: $x_i \xrightarrow{y_j} \hat{x}_a$ can be verified in every cell of the ILA, by applying a constant number of k test input patterns to the ILA. By *verifying* a transition τ in a cell L_i we mean applying the corresponding input signals x_i and y_j, to the X and Y input lines of the cell L_i and then observing both $\hat{X}$ and $\hat{Y}$ output signals of L_i at the observable primary outputs of the array. For the array to be C-testable every transition should be verifiable in all cells of the array with a constant number of array input patterns.

5.4 A Necesary Condition

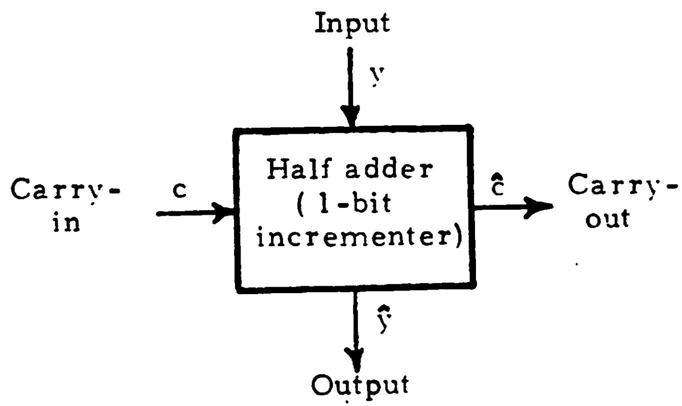
Next we derive an important necessary condition for an array to be C-testable.

Definition 5.5: Let τ: $x_i \xrightarrow{y_j} x_k$ be any transition in the flow table F of a cell L. τ is called a *repeatable transition* if there exists an input sequence that drives F from x_k back to x_i. F is called an *RT* (*r*epeatable *t*ransition) *flow table* if every transition of F is a repeatable transition. The corresponding flow diagram is called an *SC* (*s*trong *c*omponent) *flow diagram*.

Dias refers to an RT flow table as a table possessing strongly connected components [Dias 1976].

Example 5.1 (cont'd): The flow table of the full adder cell is an RT flow table and its flow diagram is an SC diagram; see Figures 5.4b and 5.4c.

Example 5.2: Consider a half adder cell which is defined in Figure 5.5a. The cell flow table is given in Figure 5.5b, and the flow diagram in Figure 5.5c. The output signal $\hat{y}$ of the half adder is y+1 if the carry input c is 1, and is y itself if c is 0; hence the half adder acts as a 1-bit incrementer. An array of N such cells forms an N-bit incrementer. Clearly the underlined transition in Figure 5.5b is a non-repeatable transition and hence the



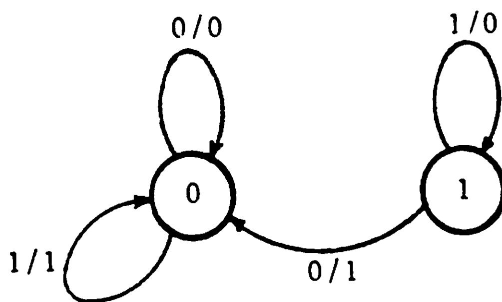
(a)

Input y

State c	0	1
0	(0, 0)	(0, 1)
1	(0, 1)	(1, 0)

$(\hat{c}, \hat{y})$

(b)



Edge labels denote $y/\hat{y}$

(c)

Figure 5.5. (a) A half adder or a 1-bit incrementer cell, (b) its flow table, and (c) its flow diagram

flow table is not an RT flow table. The corresponding flow diagram (Figure 5.5c) is a non-SC flow diagram. □

Suppose that the transition τ: $x_i \xrightarrow{y_j} x_k$ is not repeatable. For the array to be C-testable it is necessary that τ be verifiable in all its cells by applying k input patterns to the array, where k is independent of the array size N. If input signals x_i and y_j corresponding to τ are now applied to the X and Y inputs, respectively, of the leftmost cell L_0 in the array (see Figure 5.2b) then the $\hat{X}$ output signal of L_0 is x_k. Since τ is non-repeatable, it is by definition impossible to generate x_i again at the X input of any other cell in the array. This implies that the cell inputs x_i and y_j, corresponding to τ, cannot be applied to more than one cell at the same time. In other words, the number of array input patterns k required to verify τ in all N cells of the array must be proportional to N; hence the array is not C-testable. We therefore have the following necessary condition for C-testability.

<u>Theorem 5.1</u>: A unilateral combinational ILA is C-testable only if the cell flow table is an RT flow table.

Although Friedman has not mentioned the above condition explicitly it can be derived from his Theorem 1 in [Friedman 1973], which is a general characterization of

Class 1 C-testable ILAs. This is discussed in more detail later in Section 6.1. On the other hand, Dias has not given any result similar to the above. As will be seen later in Chapter 7, the necessary condition of Theorem 5.1 is also a sufficient condition for C-testability in certain types of ILAs.

In the next two chapters we examine in more detail the property of C-testability in Class 1 and Class 5 ILAs. In Chapter 8, arrays of bilateral and sequential cells are analyzed.

CHAPTER 6

ARRAYS WITHOUT DIRECT OUTPUTS

The structure of arrays that have no direct (vertical) outputs is depicted in Figure 6.1. Here the only primary output lines of the ILA are the $\hat{X}$ output lines of the rightmost cell. As can be expected, this limited observability complicates ILA testing, particularly in the case of C-testable arrays. The general requirements for C-testability in this type of arrays were first obtained by Friedman [Friedman 1973]. These results are reviewed here and some related new results, including the characterization of C-testable ILAs having a single horizontal line, are presented.

6.1 General Requirements for C-testability

Consider an ILA with an $n \times m$ cell flow table F. As shown in Figure 6.1, let X and Y be the sets of horizontal input and vertical input signals, respectively. We now introduce the concept of a periodic input pattern for an array of arbitrary size. A *periodic array input pattern* t denoted by $x_a, (b_0 b_1 \ldots b_{r-1})^*$ is an input combination that applies x_a to the X input of the leftmost cell L_0 in Fig-

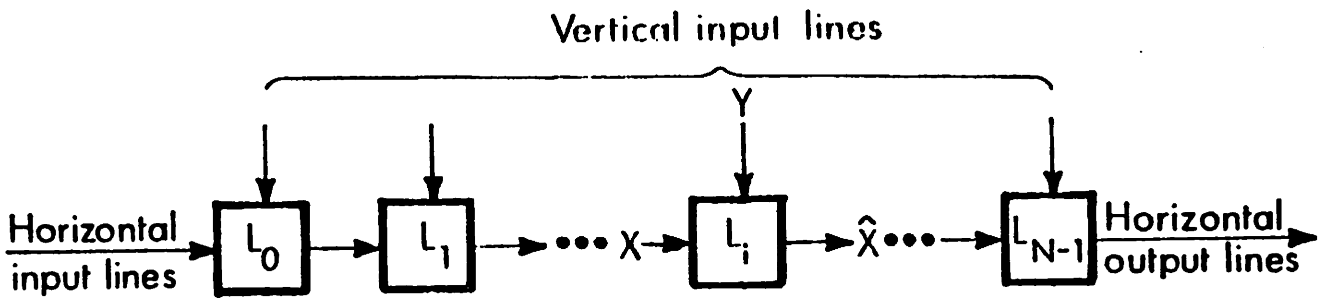


Figure 6.1. Structure of an ILA without direct outputs

ure 6.1, and applies $b_j \in Y$ to the Y input of the cell L_i, where $i \equiv j \pmod r$ for all i, and $\hat{x}(x_a, b_0 b_1 \ldots b_{r-1}) = x_a$. The asterisk superscript indicates that the Y input pattern $b_0 b_1 \ldots b_{r-1}$ is applied repeatedly to all the Y inputs of the array, thus $b_0 b_1 \ldots b_{r-1} b_0 b_1 \ldots b_{r-1} b_0 b_1 \ldots$. The number of repetitions depends on the array length; in general $(b_0 b_1 \ldots b_{r-1})^*$ is truncated to match the length of any particular ILA to which it is applied. Therefore t applies identical cell input patterns to the cells spaced at fixed intervals of r cells along the array, where r is the *periodicity* of t, i.e., the number of cells spanned by $b_0 b_1 \ldots b_{r-1}$. As will be seen later, such periodic patterns are very useful in constructing test patterns for C-testable ILAs.

Let τ: $x_a \xrightarrow{y_j} x_b$ be any transition of the cell flow table F. As discussed in Section 5.3, for the ILA to be C-testable it is necessary that τ be verifiable simultaneously in cells spaced periodically along the array. Thus input sequences for F should exist that satisfy the two general requirements given in the following theorem due to Friedman [Friedman 1973, Theorem 1].

Theorem 6.1: An ILA is C-testable if and only if for every transition τ: $x_a \xrightarrow{y_j} x_b$ of its cell flow table there exists an input sequence R for the table that satisfies the following two conditions:

(1) $\hat{x}(x_a, y_j R) = x_a$.

(2) The periodic input sequence $(Ry_j)^* = Ry_j RY_j \ldots$ is such that $\hat{x}(x_i, (Ry_j)^*) \neq \hat{x}(x_b, (Ry_j)^*)$ where $x_i \neq x_b$.

The first condition of Theorem 6.1 ensures that the same transition τ can be applied to cells spaced periodically at intervals of r cells, where r is the length of the sequence $y_j R$, i.e., $r = |y_j R|$. If the second condition is satisfied, then the periodic sequence $(Ry_j)^*$ will propagate a faulty $\hat{X}$ output signal x_i from the cell(s) in which τ is being verified, to the final observable $\hat{X}$ output of the rightmost cell of the ILA. Such a sequence R must exist for every faulty next state value $x_i \in X-\{x_b\}$. Note that R can be different for different faulty states. Thus the above Friedman conditions for C-testability are very general, and they do not indicate any specific functional property of the cells or their flow tables.

Condition (1) of Theorem 6.1 includes the necessary condition of Theorem 5.1 presented in Section 5.4, namely, an ILA is C-testable only if the cell flow table is an RT flow table. The array input pattern $x_a, (y_j R)^*$ of Theorem 6.1 is a periodic input pattern which is appropriately truncated to match the length N of the ILA.

Definition 6.1: Consider an arbitrary periodic array input pattern $t = x_a, (b_0 b_1 \ldots b_{r-1})^*$ of period r. The

shift operator s is defined by

$$s(t) = x_c, (b_1 b_2 \ldots b_{r-1} b_0)^* \qquad (6.1)$$

where $x_c = \hat{x}(x_a, b_0)$. If this shift operation is repeated $q < r$ times, then we get the periodic input pattern

$$s^q(t) = x_d, (b_q b_{q+1} \ldots b_{r-1} b_0 b_1 \ldots b_{q-1})^* \qquad (6.2)$$

where $x_d = \hat{x}(x_a, b_0 b_1 \ldots b_{q-1})$.

Clearly $s^q(t)$ of Equation (6.2) is the array input pattern obtained by shifting the periodic inputs applied by t to the cells of the array, by q cell positions to the left. In other words, if τ is the transition applied by t to cell L_i then $s^q(t)$ applies the same transition τ to cell L_{i-q}, for all i. The following identities are direct consequences of the definition of s.

(i) $s^0(t) = s^r(t) = t$, hence s^{ir} is the identity operator for any integer i,

(ii) $s^a(s^b(t)) = s^c(t)$ where $a+b \equiv c \pmod r$,

(iii) $s^{-i}(s^i(t)) = s^0(t) = t$, for any integer i.

Consequently the set of shift operators: $\{s^q : 0 \leq q \leq r-1\}$ forms an algebraic structure called a cyclic group [Herstein 1975]. $s^q(t)$ is referred to here as the *q-shifted version* of t.

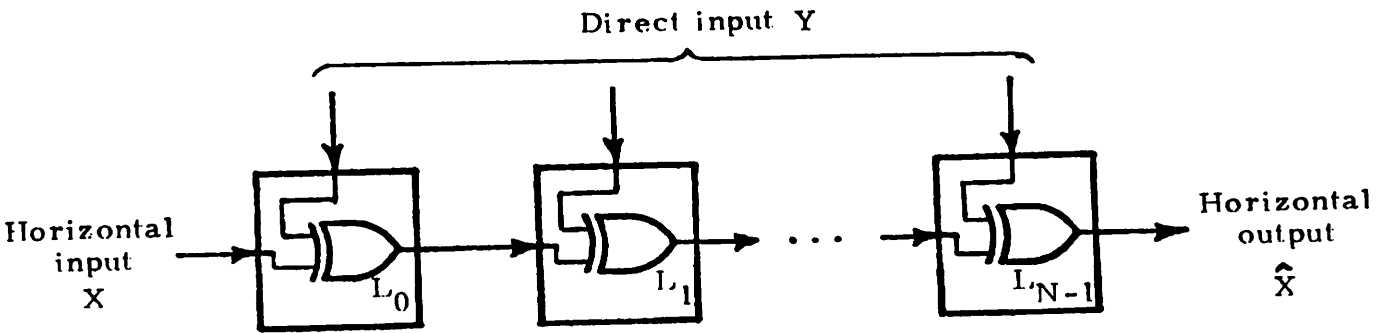
Example 6.1: Consider the N-bit parity-check ILA shown in Figure 6.2. The cell flow table defines the 2-input parity-check or EXCLUSIVE-OR function. Suppose that $N = 5$. The periodic array input pattern $t = 0,(101)^*$ applies 0 to the X input of cell L_0, and 1,0,1,1,0 to the Y inputs of the five cells L_0, L_1, L_2, L_3 and L_4 respectively. The three distinct shifted versions of t are then $s^0(t) = t$, $s^1(t) = s(t) = 1,(011)^*$ and $s^2(t) = s(s(t)) = 1,(110)^*$.

Defintion 6.2: Consider any transition τ: $x_a \xrightarrow{y_j} y_b$ of an $n \times m$ cell flow table. A C-test $C_i(\tau)$ associated with this transition is a periodic input pattern of the form

$$C_i(\tau) = x_a, (y_j R_i)^* \tag{6.3}$$

where R_i is a Y input sequence that satisfies the conditions of Theorem 6.1 and propagates a faulty next state signal $x_i \neq x_b$ appearing at the $\hat{X}$ output of a cell being tested for τ, to the observable $\hat{X}$ output of the array.

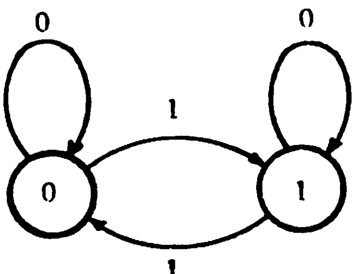
$C_i(\tau)$ of Equation (6.3) verifies the transition τ for any fault f that changes the next state from x_b to x_i in the leftmost cell L_0 of the array and in all cells spaced periodically at intervals of r_i cells along the array. r_i, which is the number of cells spanned by $y_j R_i$, is termed the periodicity of $C_i(\tau)$. Thus $C_i(\tau)$ tests cells L_0, L_{r_i}, L_{2r_i}, ... The q-shifted version $s^q(C_i(\tau))$ of $C_i(\tau)$ is a



(a)

State X	Input Y: 0	Input Y: 1
0	0	1
1	1	0

(b)



(c)

Figure 6.2. (a) Parity-check array, (b) its cell flow table, and (c) cell flow diagram

C-test which verifies τ for the same fault f in cells $L_{r_i-q}, L_{2r_i-q}, L_{3r_i-q}, \ldots$. Thus the r_i C-tests $\{s^q(C_i(\tau)): 0 \leq q \leq r_i-1\}$, collectively verify τ for fault f in all cells of the array. Let CT(τ) denote the set of $\sum r_i$ C-tests of the form $s^q(x_a,(y_jR_i)^*)$ obtained by considering every faulty next state $x_i \in X-\{x_b\}$, and by allowing q to range from 0 to r_i-1. Note that the sequence R_i may propagate more than one faulty next state signal at a time; for example, R_j may be the same as R_i for some faulty state $x_j \neq x_i$. The set CT(τ) can thus be written as

$$CT(\tau) = \{s^q(x_a,(y_jR_i)^*): 0 \leq q \leq r_i-1 \text{ and } R_i \text{ propagates a faulty state } x_i \in X-\{x_b\} \text{ to an ILA output}\} \quad (6.4)$$

This set of C-tests CT(τ) therefore, completely verifies τ in every cell of the ILA. Note that the size of CT(τ) is $|CT(\tau)| = \sum r_i$, where $r_i = |y_jR_i|$ is independent of the array size N, hence $|CT(\tau)|$ is also independent of N.

Example 6.1 (cont'd): Consider again the parity-check ILA of Figure 6.2. Its cell flow table given in Figure 6.2b is an RT flow table, and its flow diagram of Figure 6.2c is an SC flow diagram. Let the four transitions of the flow table be labeled τ_{00}, τ_{01}, τ_{10} and τ_{11}, where τ_{ij} is the transition in row i and column j. Possible C-tests

for these transitions are $CT(\tau_{00}) = \{0,0^*\}$, $CT(\tau_{01}) = \{0,1^*\}$, $CT(\tau_{10}) = \{1,0^*\}$ and $CT(\tau_{11}) = \{1,1^*\}$. Thus a parity-check ILA of arbitrary size can be completely tested with only four test patterns.

Definition 6.3: An ILA with an $n \times m$ cell flow table F is said to be *optimally testable* if it can be completely tested by a test set T_{ILA} of size mn.

It is obvious that T_{ILA} contains the smallest number of test patterns needed to test the ILA completely. This is because we need at least mn array input patterns to apply all mn transitions of F to any one cell. Since $|T_{ILA}|$ is independent of the array size, an optimally testable ILA is also optimally C-testable.

Definition 6.4: A column y_j of a cell flow table is a *permutation column* if every state appears exactly once as a next-state entry in that column.

The following characterization of optimally testable ILAs is due to Kautz.

Theorem 6.2 [Kautz 1967]: An ILA is optimally testable if and only if every column in the cell flow table is a permutation column.

In general, if y_j is a permutation column of an $n \times m$ cell flow table F, then the n C-test patterns: x_i, y_j^* for

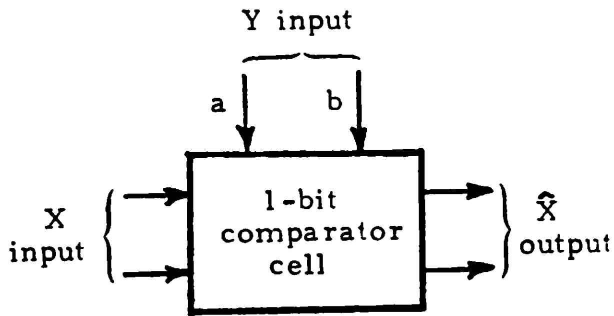
$i = 0,1,\ldots,n-1$, verify all n transitions of column y_j in every cell of the array. For example, both columns of the cell flow table of Figure 6.2b are permutation columns. Thus the parity-check ILA of Example 6.1 is optimally C-testable with only four tests as stated before.

Definition 6.5: A column y_j of a cell flow table is a *constant column* if the next-state entry in y_j is the same for every row.

Lemma 6.1 [Friedman 1973]: An ILA is C-testable only if there are no constant columns in the cell flow table.

Example 6.2: Figure 6.3a depicts a 1-bit comparator cell capable of comparing two 1-bit operands a and b. Its function is to generate the output signal 00 at the two $\hat{X}$ output lines if $a < b$, i.e., $a = 0$ and $b = 1$, and the output signal 10 if $a > b$. If $a = b$, then as indicated in the cell flow table F_c of Figure 6.3b, $x = \hat{x}$. It is evident that by cascading N identical cells of Figure 6.3a an N-bit comparator ILA is realized. The two columns 01 and 10 in F_c are both constant columns, consequently this comparator ILA is not C-testable. □

From Theorem 5.1 and Lemma 6.1 we get the following necessary conditions for an array without direct outputs to be C-testable.



(a)

Input Y

State X	ab: 00	01	10	11
00	00	00	10	00
01	01	00	10	01
10	10	00	10	10

(b)

Input Y

State X	ab: 00	01	10	11	y_a (Added column)
00	00	00	10	00	01
01	01	00	10	01	10
10	10	00	10	10	11
11 (Added row)	11	11	11	11	00

(c)

Figure 6.3. (a) A 1-bit comparator cell, (b) its flow table F_c and (c) the modified flow table F'_c

Theorem 6.3: A unilateral combinational array without direct outputs is C-testable only if the cell flow table

(1) is an RT flow table, and

(2) contains no constant columns.

The two conditions of the above theorem are not sufficient for an ILA to be C-testable, as demonstrated in the following example.

Example 6.2 (cont'd): The flow table F_c of Figure 6.3b is not an RT flow table since the transition $01 \xrightarrow{01} 00$ is not repeatable. Also F_c has two constant columns, hence it does not satisfy either condition of Theorem 6.3. A new row $X = 11$ and a new column y_a can be added to F_c, as shown in Figure 6.3c, such that the modified flow table F'_c is an RT flow table and has no constant columns. Therefore F'_c satisfies the two necessary conditions of Theorem 6.3 for C-testability. However, as is shown next, an array of modified comparator cells with F'_c as the new cell flow table, is not C-testable. Consider the transition τ: $01 \xrightarrow{y_j = 10} 10$ in F'_c and let 01 be the faulty next-state signal. According to Theorem 6.1, for C-testability it is necessary to be able to construct an input sequence R such that $\hat{x}(01, y_j R) = 01$ and $\hat{x}(01, (Ry_j)^*) \neq \hat{x}(10, (Ry_j)^*)$. Since in column 10 of F'_c the next-state entry is distinct only in

the new row 11, R should transfer the faulty next state 01 to the new state 11. At the same time R should transfer the fault-free state 10 to the initial state 01 of the transition τ. However, it can be shown that no such sequence R exists. Hence an array of cells with the flow table F'_c is not C-testable. □

The problem of obtaining a complete functional characterization of C-testable arrays without direct outputs, based on the properties of the cell flow table appears to be very difficult, and has not been solved. In the next section it is proved that the second necessary condition of Theorem 6.3 is also a sufficient condition for C-testability for the special but important case of arrays having only a single horizontal X line.

6.2 Arrays With a Single Horizontal Line

We now present a complete characterization of C-testable ILAs containing only a single horizontal line between the cells. Let $X = \hat{X} = \{0,1\}$ be the horizontal signal set. Consider any column y_j in a cell flow table F, and let x be the X input signal of the cell. Since F has only two states, $\hat{x}(0,y_j)$ is equal to either $\hat{x}(1,y_j)$ or $\bar{\hat{x}}(1,y_j)$. In the first case y_j must be a constant column (Definition 6.5) while in the second case it must be a permutation

column (Definition 6.4). If $\hat{x}(0,y_j) = \bar{\hat{x}}(1,y_j)$ then the $\hat{x}$ output function for column y_j can be concisely written as $\hat{x}(x,y_j) = x \oplus \hat{x}(0,y_j)$.

Theorem 6.4: An ILA with only one horizontal line is C-testable if and only if

$$\hat{x}(x,y_j) = x \oplus \hat{x}(0,y_j) \quad (6.5)$$

for all $y_j \in Y$ of the cell flow table.

Proof: *Necessity.* If the $\hat{x}$ output function of the basic cell does not satisfy Equation (6.5) then as discussed above, y_j must be a constant column. Hence by Lemma 6.1, the ILA is not C-testable.

Sufficiency. If $\hat{x}(x,y_j) = x \oplus \hat{x}(0,y_j)$, then $\hat{x}(0,y_j) = \bar{\hat{x}}(1,y_j)$ and y_j is a permutation column. Therefore if Equation (6.5) is true for every $y_j \in Y$, then the cell flow table contains only permutation columns. Hence according to Theorem 6.2 the ILA is optimally testable. The two array inputs $0,y_j^*$ and $1,y_j^*$ completely verify the two transitions in column y_j in every cell of the ILA. The ILA thus requires only 2m test patterns independent of its size; it is therefore optimally C-testable. □

The parity check ILA of Example 6.1 is a simple ILA that satisfies the condition of Theorem 6.4. The following

result is an immediate consequence of Theorem 6.4.

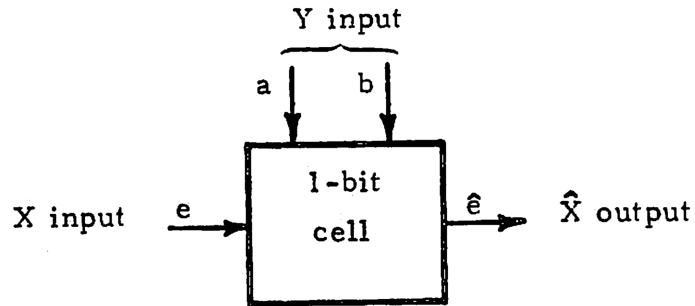
<u>Corollary 6.1</u>: If an ILA with only one horizontal line is C-testable, then it is optimally C-testable.

<u>Example 6.3</u>: Consider an equality checker ILA that determines if its two binary input operands are equal. A 1-bit cell and flow table for this array are given in Figure 6.4. The cell output $\hat{e}$ is 1 if $a \neq b$, and is the same as the input e if $a = b$. Therefore the $\hat{e}$ output of the rightmost cell of the array, which operates on the most significant bits, is 0 if the two operands are the same, and is 1 otherwise. It is evident that $\hat{x}(0,y)=\hat{x}(1,y) = 1$, where y is 01 or 10 in the flow table (Figure 6.4b), hence the equality checker ILA is not C-testable.

6.3 Making an Array C-testable

A general approach to making any ILA C-testable is to modify the cell flow table so that the requirements for C-testability discussed in Section 6.1, including the necessary conditions specified in Theorem 6.3, are satisfied. For this purpose additional rows and columns can be added to the original cell flow table F to form a new table F′ that meets the requirements for C-testability.

Let F be the $n \times m$ cell flow table of an ILA that is not C-testable. We first discuss a worst-case situation



(a)

State e	Input Y ab: 00	01	10	11
0	0	1	1	0
1	1	1	1	1

(b)

Figure 6.4. (a) A 1-bit equality checker cell, and (b) its flow table

in which F does not satisfy either of the two necessary conditions for C-testability given in Theorem 6.3. Thus F is not an RT flow table and contains constant columns. It is obvious that to eliminate the constant columns from F it suffices to add a new row x_n in which the next-state entries are different from those in the original n rows of the constant columns. Since F is a non-RT flow table in this worst-case situation, there exists at least one transition, say τ: $x_a \xrightarrow{y_j} x_b$, that is not repeatable. Then to make F′ an RT flow table it is necessary and sufficient to add new transitions to F that create a path from state x_b back to state x_a. This can be achieved only by adding a new column, say y_m, to F. Note that a new row will not serve the purpose here, since it will only introduce a new state and new transitions from this added state. The next-state entries in the new column y_m of F′ are selected so that all the states of F′ are linked via y_m to form a closed cycle in the cell flow diagram. One way of doing this is to make $\hat{x}(x_i, y_m) = x_{i+1}$, for all i.

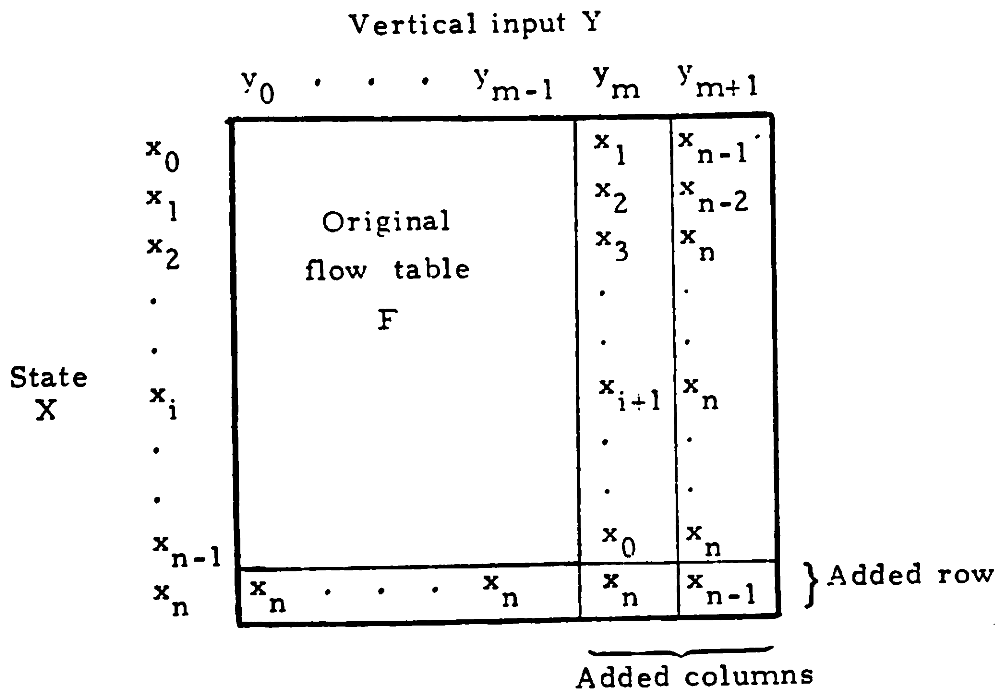
Thus in the above worst-case we need one new row x_n and one new column y_m to make F′ satisfy the necessary conditions for C-testability given in Theorem 6.3. However, since the conditions of Theorem 6.3 are not sufficient for C-testability, an ILA of cells having the modified flow table F′ may not be C-testable. For example, consider the

comparator ILA discussed in Example 6.2. Figure 6.3c shows the comparator cell flow table F_c of Figure 6.3a modified according to the above procedure, with $x_n = 11$ and $y_m = y_a$ as the new row and column. But it was shown in Example 6.2 that an array of cells with the flow table of Figure 6.3c is not C-testable. Hence, in general, there exist cases where the addition of one new row and one new column to F is necessary but not sufficient to achieve C-testability. Thus we have the following result.

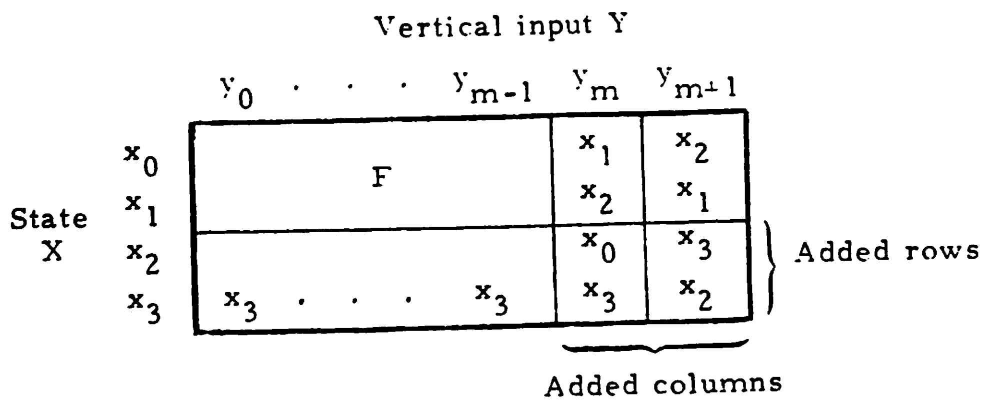
Lemma 6.2: A flow table modification scheme for C-testability requires in certain cases the addition of more than two rows and columns to the original cell flow table.

Proposed Method

We now propose a design scheme that makes any flow table C-testable by the addition of exactly one new row and two new columns. Let F be the original $n \times m$ flow table and let $n > 2$. Figure 6.5a shows the general structure of the modified flow table F' formed from F. The new row x_n eliminates constant columns. It is used as a "trap" state for any faulty $\hat{X}$ output signal while verifying the transitions in the original flow table F. Thus a faulty $\hat{X}$ output signal from any cell is observable as x_n at the $\hat{X}$ output of the rightmost cell of the array. The transfer of any faulty state to the trap state is easily achieved by con-



(a)



(b)

Figure 6.5. Flow table modification to make an array without direct outputs C-testable: (a) for $n > 2$; (b) for the special case $n = 2$

structing suitable transfer sequences from the entries in the new columns y_m and y_{m+1}. Using this general approach one can also verify the transitions in the new row and columns. For details on constructing the actual C-tests for the modified table, see Appendix A.3. Note that the entries in the new column y_m link all n states in a single loop to make F an RT flow table. In the foregoing modification scheme it was assumed that $n > 2$. For flow tables with $n = 2$ we can add a dummy row x_2 to make $n = 3 > 2$ and thus use the above scheme, as indicated in Figure 6.5b.

Theorem 6.5: ILAs whose flow tables are modified using the method of Figure 6.5, are C-testable.

For a proof of Theorem 6.5, see Appendix C.

From Lemma 6.2 and Theorem 6.5 it follows that for $n \geqq 3$, the proposed modification scheme is optimal with respect to the number of additional rows and columns used. Hence it appears to be least expensive modification scheme in terms of extra hardware required. A general hardware implementation of the modified basic cell L′ is shown in Figure 6.6. Since the original flow table entries are unaltered by our modification scheme, the logic design of the original cell L is not affected. The new logic circuits NL_1 and NL_2 of Figure 6.6 implement the new columns y_m and y_{m+1}, respectively; in other words, they generate

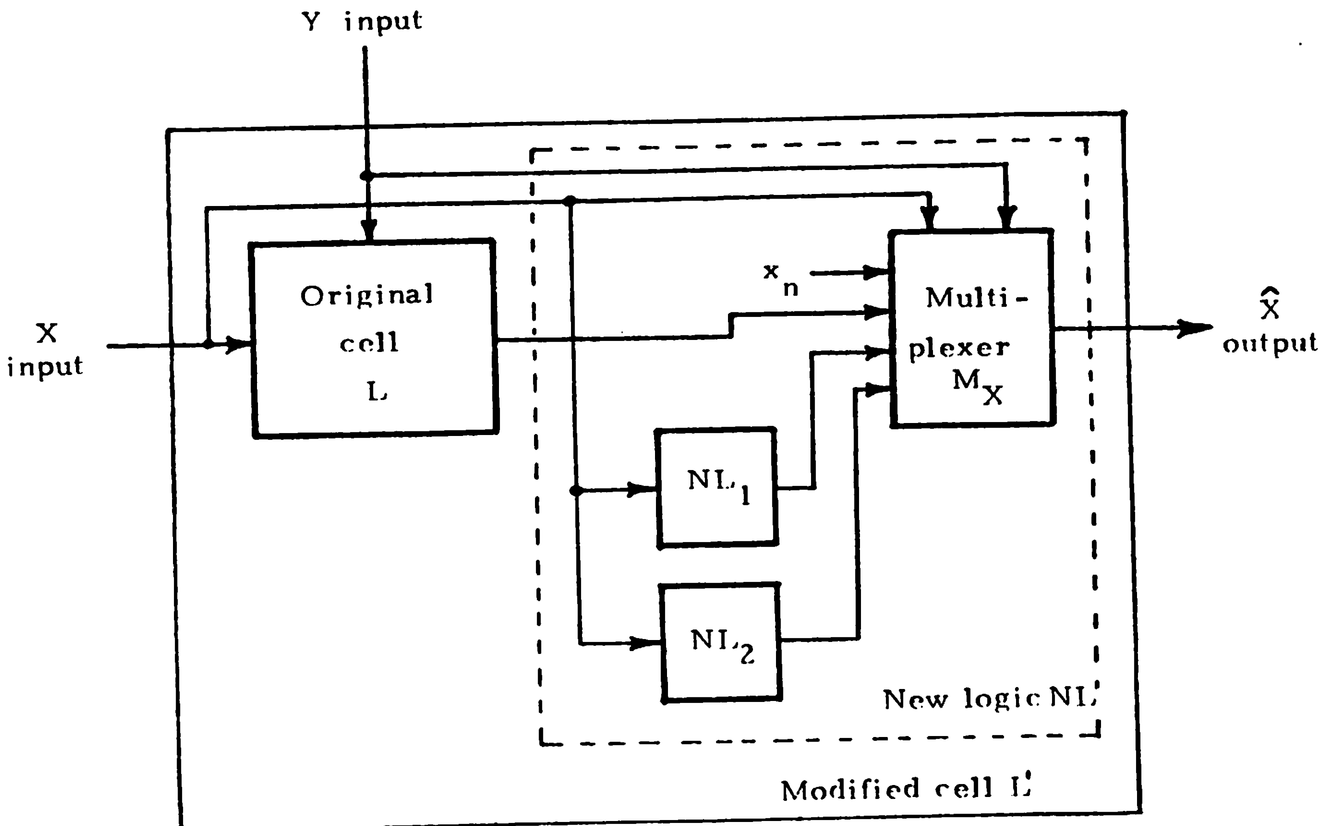


Figure 6.6. A hardware implementation of the cell modification scheme for C-testability

the next-state entries appearing in these columns. The multiplexer M_X selects the outputs of NL_1 and NL_2 whenever the Y input is y_m and y_{m+1} respectively, otherwise the $\hat{X}$ output of the original cell L is selected. Thus the new circuitry NL added to the original cell L consists of only three simple combinational modules NL_1, NL_2 and M_X. Since NL is independent of L, it is applicable to any given cell whose flow table is of size $n \times m$.

Several flow table modification methods for C-testability have been proposed in the past; all are non-optimal, however. Firedman's scheme [Friedman 1973] requires one additional row and $\frac{p}{2}(p+1)+2$ additional columns, where $p = \lceil \log_2 n \rceil$. Thus in his scheme the additional logic required increases rapidly with n, the number of states in the given cell flow table. Furthermore, there is a minor error in Friedman's modification method as pointed out recently [Parthasarathy and Reddy 1979]. Parthasarathy and Reddy have proposed a modification scheme similar to Friedman's based on their concept of one-step C-testability [Parthasarathy and Reddy 1979]. Their procedure requires one additional row and three additional columns.

In Table 6.1 upper bounds on the number of tests for the modified arrays obtained in the above three modification methods, are given. (For more details on the number of tests required in our method, see Appendix C.) It is evident that the upper bound UB_3 of the method proposed

Table 6.1. Comparison of three modification schemes to achieve C-testability

Method	Number of Added Rows	Number of Added Columns	Upper Bounds on the Number of Tests Required
Friedman	1	$\frac{p(p+1)}{2} + 2$	$UB_1 = (3n^2+n)(n+1)\left[m+\frac{1}{2}p(p+1)+2(n+1)\right]$ $= 6n^4+2mn^3+k_1$
Parthasarathy and Reddy	1	3	$UB_2 = (n+1)^3(m+1)+2(n+1)$ $= (m+1)n^3+3(m+1)n^2+(3m+5)n+(m+3)$
Scheme proposed here ($n \geqq 3$)	1	2	$UB_3 = mn^3+\left(\frac{7}{2}m+2\right)n^2+\left(14-\frac{5}{2}m\right)n+(m-17)$

Notes: 1. The original flow table has n rows and m columns.

2. $p = \lceil \log_2 n \rceil$

3. $k_1 > UB_2$ and UB_3

4. $UB_2 \cong UB_3+n^3+\left(1-\frac{m}{2}\right)n^2$

here is much better than that of Friedman's scheme, and is slightly better than that of Parthasarathy and Reddy. Also, as discussed earlier, our scheme is least expensive with respect to the additional logic required.

A completely different modification technique to achieve C-testability has been proposed by Tamamoto and Takaba [Tamamoto and Takaba 1976]. Their approach is to satisfy the condition for optimal C-testability given in Theorem 6.2, i.e., to make every column in the cell flow table, a permutation column. This requires excessive additional logic, and calls for a complete redesign of the original cell. Hence it does not appear to be of practical value.

CHAPTER 7

ARRAYS WITH DIRECT OUTPUTS

In this chapter the C-testability of unilateral combinational arrays with direct outputs, i.e., the Class 5 arrays of Table 5.1, is examined. Figure 7.1 shows the general structure of such arrays. ILA circuits like the ripple-carry adder of Example 5.1, and the incrementer of Example 5.2 belong to this class. Also ILAs composed of more sophisticated cells such as the multi-function ALU of of the 2901 (Figure 2.3) and the processor cell C (Figure 3.1) can be easily modeled as Class 5 ILAs. For example, consider the subarray formed by the eight-function ALUs in a processor array of C-type cells; see Figure 4.1. The ALU of this subarray is controlled by lines I_3, I_4 and I_5 which are common to every cell. Such a system can be viewed as consisting of eight Class 5 ILAs with eight different basic cells, one for each ALU function defined by the three common control lines.

Dias has studied the testability of Class 5 arrays, and some of his results are relevant to C-testing [Dias 1976]. His main goal was truth-table verification of Class 5 arrays, i.e., checking the complete input-output behavior of the array. He has developed a testing pro-

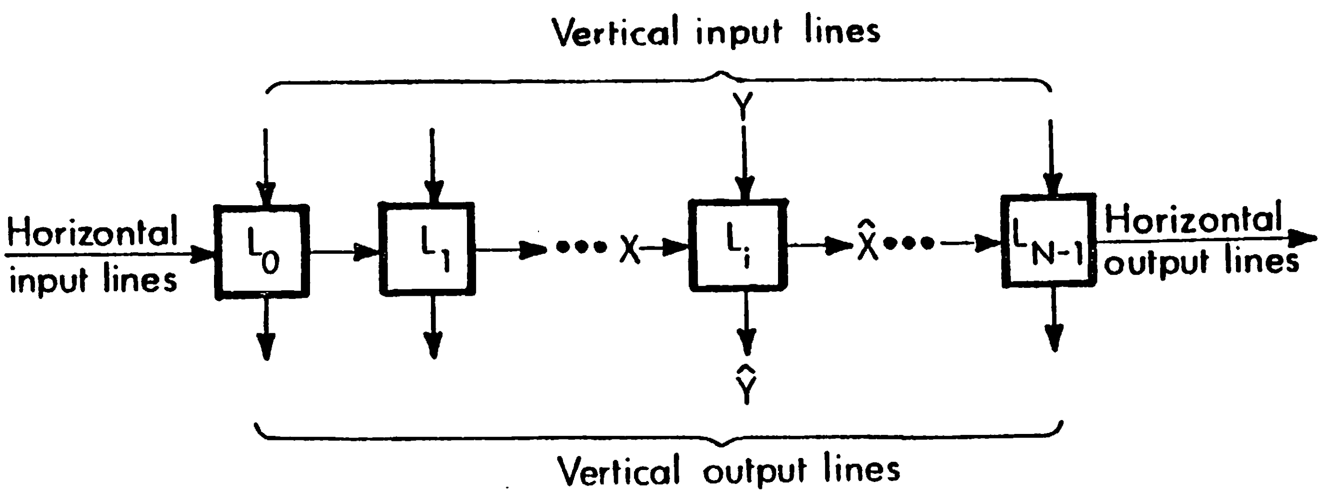


Figure 7.1. Structure of an ILA with direct outputs

cedure for this purpose which requires a constant number of array input patterns, independent of the array size. If such a procedure exists for a given array, the array must be C-testable. Dias' technique is valid only when the cell flow table is a reduced RT flow table [Dias 1976, Theorem 2]. He has also presented a modification method to make any basic cell possess these properties. Hence this work implicitly includes a scheme for making any Class 5 ILA C-testable.

The approach to C-testing employed here is similar to that used for Class 1 ILAs in the previous chapter. The difference is that here we make use of the available direct outputs of the cells to observe the internal horizontal signals. As will be seen later, this fact substantially simplifies array testing. First, certain special test sequences associated with a cell flow table are defined. These are based on the analogy between a cell flow table and a synchronous sequential machine discussed in Section 5.3. With the help of these sequences, array test sequences that are shorter in length than Dias' test sequences are constructed.

7.1 Distinguishing Sequences

We now define certain basic sequences derived from cell flow tables that are later used to construct test patterns for C-testable ILAs.

Definition 7.1: A *distinguishing sequence* $DS(x_a, x_b)$ for a pair of states x_a and x_b of a cell flow table F is a Y input sequence which, when applied to F, produces two different $\hat{Y}$ output sequences with x_a and x_b as the initial states.

Definition 7.2: A *set of distinguishing sequences* $SDS(x_a)$ for a state x_a of a cell flow table F is a set of Y input sequences which, when applied to F with x_a as the initial state, produces a set of $\hat{Y}$ output sequences different from the sets of output sequences produced with any other state as the initial state. Hence we have

$$SDS(x_a) = \{DS(x_a, x_b) : x_b \in X \text{ and } x_b \neq x_a\} \qquad (7.1)$$

In contrast, Dias uses more conventional sequences, namely, the set of characterizing sequences which he calls a *set of identifying sequences* (SIS), for identifying the state of a cell [Dias 1976, Definition 6]. A set of *characterizing* or *identifying sequences* for a flow table F is a set of Y input sequences whose application to F produces a set of $\hat{Y}$ output sequences that are different for each choice of the initial state [Friedman and Menon 1971]. An SIS of size one is a distinguishing sequence for the flow table. Both distinguishing and identifying sequences are useful in identifying the state, i.e., the $\hat{X}$ output signal,

of a cell L_i in an array. The $\hat{X}$ output signal of L_i can be identified by applying a member of either SDS or SIS to the Y inputs of the cells $L_{i+1}, L_{i+2}, \ldots$, and then observing the responses at their $\hat{Y}$ outputs. This is further explained later in this section.

SDS and SIS sequences differ in several ways. An SDS is defined for each state separately, whereas an SIS is defined for entire flow table. $SDS(x_a)$ can only determine whether the $\hat{X}$ output signal of the cell is x_a or not. On the other hand, an SIS determines the exact $\hat{X}$ output signal of a cell. For fault detection there is no need to identify the faulty $\hat{X}$ output signal of a cell, hence SDSs are sufficient for our purposes. As will be seen later, the use of SDSs instead of SISs reduces the number of test patterns needed to test an ILA. One reason for this is the fact that the distinguishing sequences of an SDS are usually shorter in length than the identifying sequences of an SIS, as demonstrated in the following example.

Example 7.1: Figure 7.2 shows flow tables for four different cells whose signal sets are $Y = \hat{Y} = \{0,1\}$ and $X = \hat{X} = \{00, 01,10,11\}$. The corresponding SISs and SDSs are also given in the figure, where $\emptyset$ denotes an empty set of sequences. As is evident here, the length of a distinguishing sequence is usually less than, and never greater than that of an identifying sequence. In the case of the flow table

State X	Input Y = 0	Input Y = 1
00	(01, 0)	(11, 1)
01	(01, 0)	(10, 0)
10	(00, 0)	(10, 0)
11	(00, 0)	(11, 1)

SIS = {1, 01}
SDS(x) = {1, 01}
for all x.

(a)

State X	Input Y = 0	Input Y = 1
00	(00, 0)	(01, 1)
01	(00, 0)	(10, 1)
10	(01, 0)	(11, 1)
11	(10, 1)	(00, 1)

SIS = {0, 10, 110}
SDS(00) = SDS(01) = SIS
SDS(10) = {0, 10}
SDS(11) = {0} .

(b)

State X	Input Y = 0	Input Y = 1
00	(00, 1)	(01, 0)
01	(01, 0)	(10, 0)
10	(10, 0)	(11, 0)
11	(00, 1)	(11, 1)

SIS = {111} or {110}
SDS(00) = {0, 1}
SDS(01) = SDS(10) = {0, 11}
SDS(11) = {1} .

(c)

State X	Input Y = 0	Input Y = 1
00	(01, 0)	(00, 1)
01	(00, 0)	(01, 1)
10	(10, 1)	(00, 0)
11	(10, 0)	(11, 0)

SIS = ϕ
SDS(00) = SDS(01) = ϕ
SDS(10) = {0}
SDS(11) = {1, 0} .

(d)

Figure 7.2. Flow tables and their identifying and distinguishing sequences for Example 7.1

of Figure 7.2d, the sets of sequences SDS(00), SDS(01) and SIS are empty since states 00 and 01 are not distinguishable at the $\hat{Y}$ output. □

It is interesting to note that an SIS is always an SDS(x_i) for any state x_i, and that the collection of SDS(x_i) sets for all n states together form an SIS for the flow table. This follows directly from the definitions of SIS and SDS(x_i).

<u>Definition 7.3</u>: Two states x_i and x_j of a cell flow table F are *equivalent* if every Y input sequence applied to F produces identical $\hat{Y}$ output sequences with x_i and x_j as the initial states. F is said to be *reduced* if no two states of F are equivalent.

For example, the three flow tables of Figures 7.2a, 7.2b and 7.2c are all reduced, while the flow table of Figure 7.2d is not, since its states 00 and 01 are equivalent. Note that any two states x_i and x_j of a reduced flow table are distinguishable, i.e., a distinguishing sequence DS(x_i,x_j) exists.

Reduced flow tables are very useful in ILA testing for the following two reasons. First, it is easy to propagate internal horizontal signals to the available direct outputs of the ILA using appropriate distinguishing sequences. On the other hand, if the flow table is not

reduced, then there exist at least two cell states that are equivalent. The horizontal signals corresponding to these equivalent states cannot be distinguished at the available $\hat{Y}$ outputs. Hence with reduced flow tables, the available ILA direct outputs can be used very efficiently during testing. A second important advantage of reduced tables is that design verification of ILAs with direct outputs is feasible. Note also that practical ILA circuits usually have reduced cell flow tables. In the remainder of this chapter we therefore concentrate on arrays with reduced flow tables.

7.2 C-testing and Test Generation

Consider an ILA with an $n \times m$ reduced cell flow table F. As discussed in Section 5.3, to test this ILA completely, it is necessary and sufficient to verify all mn transitions of F in every cell of the array. SDSs can be used to detect any fault in the next state reached by a cell being tested as explained below.

Let τ: $x_a \xrightarrow{y_j} x_b$ be any transition of F, and suppose that τ is verified in some cell L_i of the ILA; see Figure 7.1. It is required to determine whether the $\hat{X}$ output signal generated by L_i due to τ is x_b or not. For this purpose we use a set of distinguishing sequences $SDS(x_b)$ as defined in Equation (7.1). Let $SDS(x_b)$ consist of k input sequences $D_1, D_2, \ldots, D_k$ of lengths $a_1, a_2, \ldots, a_k$,

respectively. Each sequence $D_p = b_1 b_2 \ldots b_{a_p}$, where $b_i \in Y$, is applied to the Y input lines of the cells $L_{i+1}, L_{i+2}, \ldots, L_{i+a_p}$, to the right of L_i, i.e., the Y input signals $b_1, b_2, \ldots, b_{a_p}$ are applied to the Y input lines of the cells $L_{i+1}, L_{i+2}, \ldots, L_{i+a_p}$, respectively, and their $\hat{Y}$ output signals are observed. By definition of an SDS, it is evident that the $\hat{X}$ output signal x_b of L_i produces a set of $\hat{Y}$ output signals from cells $L_{i+1}, L_{i+2}, \ldots$, that is different from the one produced by any one of the other n-1 signals belonging to $\hat{X}$. Thus the application of $SDS(x_b)$ to the Y inputs of the cells to the right of L_i, detects any faulty $\hat{X}$ output signal generated by L_i due to the above transition τ. Unlike an SIS, an $SDS(x_b)$ does not determine the exact faulty state among the possible n-1 values. However, for fault detection in an ILA, the identification of the exact faulty value is not required. For example, consider the set $SDS(11) = \{0\}$ for the flow table of Figure 7.2b. This SDS produces 1 at the $\hat{Y}$ output of L_{i+1} if the $\hat{X}$ output signal of L_i is 11, and produces 0 if the $\hat{X}$ output signal is 00, 01 or 10.

We next construct a special class of array test sequences called C-tests, which are similar to the C-tests defined for arrays without direct outputs in Section 6.1.

<u>Definition 7.4</u>: A *C-test* $C_i(\tau)$ for any transition τ: $x_a \xrightarrow{y_j} x_b$ of the cell flow table F is a periodic input

pattern of the form

$$C_i(\tau) = x_a, (y_j D_i R_i)^* \qquad (7.2)$$

where D_i is a member of $SDS(x_b) = \{D_1, D_2, \ldots, D_k\}$ and R_i is an input sequence that takes the state of F from $\hat{x}(x_b, D_i)$ back to x_a. Such a sequence is usually referred to as a *transfer sequence* of F since it transfers the state of F from one value to another.

A C-test is similar to a loop test as defined by Dias [Dias 1976, Definition 7]. They differ primarily in the kinds of subsequences they contain. As mentioned before, Dias uses identifying sequences instead of distinguishing sequences for identifying the next state reached by a cell of the ILA. Therefore the above sequence D_i is replaced by a member of an SIS in a loop test. Also in a loop test, the sequences $R_1, R_2, \ldots, R_k$ are not only transfer sequences as here, but also synchronizing sequences, i.e., they synchronize the periodicities of the k loop tests for a transition τ, one for each member of SIS, to the same value. In other words, the R_i's are input sequences that take F back to state x_a and their lengths are such that the k sequences $D_1R_1, D_2R_2, \ldots, D_kR_k$ all have the same length r. This ensures that τ is verified in the same cells $L_0, L_r, L_{2r}, \ldots$ of the ILA with respect to every member of SIS. This synchronization property is used by Dias to prove that

loop tests verify the truth-table of the entire ILA. However, it will be shown later that the C-tests of Definition 7.4 also verify the ILA's truth-table without the need for synchronization. Due to their use of SISs and synchronizing sequences, loop tests tend to be longer than C-tests. This is demonstrated in Example 7.2 below.

The C-test $C_i(\tau)$ of Equation (7.2) verifies the transition τ with respect to distinguishing sequence D_i of $SDS(x_b)$ in the leftmost cell L_0, and in all cells spaced periodically at intervals of r_i cells along the ILA, where r_i is the periodicity of $C_i(\tau)$. Thus $C_i(\tau)$ partially tests cells $L_0, L_{r_i}, L_{2r_i}, \ldots$. The q-shifted version $s^q(C_i(\tau))$ of $C_i(\tau)$ (see Definition 6.1) is a C-test that verifies τ in cells $L_{r_i-q}, L_{2r_i-q}, \ldots$. The r_i C-tests $\{s^q(C_i(\tau)): 0 \leq q \leq r_i-1\}$ collectively verify τ with respect to D_i in all cells of the array. Let $CT(\tau)$ denote the set of $\sum_{i=1}^{k} r_i$ C-tests of the form $s^q(x_a,(y_jD_iR_i)^*)$ obtained by allowing D_i to range over all k members of $SDS(x_b)$, and by allowing q to range over $0,1,\ldots,r_i-1$. $CT(\tau)$ completely verifies τ in every cell of the ILA, and can be written thus

$$CT(\tau) = \{s^q(x_a,(y_jD_iR_i)^*) : 0 \leq q \leq r_i-1 \quad \text{and} \quad D_i \in SDS(x_b)\} \tag{7.3}$$

The size of $CT(\tau)$ is independent of the length of the ILA.

Example 7.2: This example serves to demonstrate the differences between Dias' testing scheme and ours. The flow table given in Figure 7.2a is taken from [Dias 1976]. This table has a common SDS for every state x, namely SDS(x) = {1,01}. Consider the transition τ_2: $00 \xrightarrow{1} 11$. The corresponding C-tests are $C_1(\tau_2) = 00,(110)^*$ and $C_2(\tau_2) = 00,(10)^*$, with periodicities of three and two, respectively. On the other hand, the loop tests for τ_2 given by Dias are $L_1(\tau_2) = 00,(1110)^*$ and $L_2(\tau_2) = 00,(1010)^*$. Note that $L_1(\tau_2)$ and $L_2(\tau_2)$ are synchronized, since each has periodicity four.

The C-tests for the remaining transitions of the flow table can be similarly constructed; they are given in Table 7.1. Using these C-tests an ILA of arbitrary size can be tested completely with only 56 test patterns. The number of test patterns required if loop tests are used is 74 [Dias 1976], about 30 percent more.

Theorem 7.1: There exists a complete set of C-tests CT(τ) for every transition of a cell flow table F, if F is a reduced RT flow table.

Proof: Definition 7.3 implies that an SDS(x_a) exists for every state x_a of a reduced flow table. Since F is an RT flow table, a transfer sequence R_i exists that takes F back to the initial state. Hence we can construct a com-

Table 7.1. C-tests for the ILA of Example 7.2

No. i	Transition τ_i	C-tests		$r_1(\tau_i)+r_2(\tau_i)$
		$C_1(\tau_i)$	$C_2(\tau_i)$	
1	$00 \xrightarrow{0} 01$	$00,(010)^*$	$00,(0010)^*$	3+4 = 7
2	$00 \xrightarrow{1} 11$	$00,(110)^*$	$00,(10)^*$	3+2 = 5
3	$01 \xrightarrow{0} 01$	$01,(0100)^*$	$01,(00100)^*$	4+5 = 9
4	$01 \xrightarrow{1} 10$	$01,(1100)^*$	$01,(10100)^*$	4+5 = 9
5	$10 \xrightarrow{0} 00$	$10,(01001)^*$	$10,(001)^*$	5+3 = 8
6	$10 \xrightarrow{1} 10$	$10,(1)^*$	$10,(101001)^*$	1+6 = 7
7	$11 \xrightarrow{0} 00$	$11,(01)^*$	$11,(00101)^*$	2+5 = 7
8	$11 \xrightarrow{1} 11$	$11,(1)^*$	$11,(101)^*$	1+3 = 4

Note: $r_1(\tau_i)$ and $r_2(\tau_i)$ are the periodicities of $C_1(\tau_i)$ and $C_2(\tau_i)$, respectively.

plete set of C-tests $CT(\tau)$, as defined in Equation (7.3), for any transition τ of a reduced RT flow table. □

Bounds on the size of a C-test set $CT(\tau)$ can be determined as follows. From Equation (7.1) we have $|SDS(x)| \leq n-1$ for every x. Now in any RT flow table, $|R_i| \leq n-1$, since the number of inputs required to go from a state to any other state of an n-state flow table is at most n-1. Hence the period r_i of $C_i(\tau)$ is bounded as follows.

$$1 \leq r_i \leq 1+(n-1)+(n-1) = 2n-1 \qquad (7.4)$$

The size of the C-test set $CT(\tau)$ is therefore at most $(n-1)(2n-1)$. Let CT_{ILA} be a test set consisting of C-tests for all mn transitions τ_1 to τ_{mn} of the cell flow table, that is,

$$CT_{ILA} = \bigcup_{i=1}^{mn} CT(\tau_i) \qquad (7.5)$$

The number of tests for an array of arbitrary size is therefore bounded thus

$$mn \leq |CT_{ILA}| \leq mn(n-1)(2n-1) \qquad (7.6)$$

The corresponding bounds obtained by Dias are

$$mn \leq |T_{Dias}| \leq mn(n-1)\left[2n-1+\frac{n(n-1)}{2}\log_2 n\right] \qquad (7.7)$$

Comparing Equations (7.6) and (7.7), it is seen that Dias' upper bound includes an additional term $mn(n-1)\frac{n(n-1)}{2}\log_2 n$ which is contributed by the synchronizing sequences used in the loop tests.

7.3 Characterization of C-testability

Next we characterize C-testable ILAs having reduced cell flow tables.

Theorem 7.2: An ILA of arbitrary size with a reduced cell flow table F is C-testable if and only if F is an RT flow table.

Proof: *Necessity.* The necessity of an RT flow table for C-testability is proved in Theorem 5.1 of Section 5.4.

Sufficiency. From Theorem 7.1 we know that a complete set of C-tests can be constructed for every transition of a reduced RT flow table. A complete set of C-tests CT_{ILA} for the ILA is then easily derived, as given in Equation (7.5). From Equation (7.6) it is evident that $|CT_{ILA}|$ is independent of the array size; hence the ILA is C-testable. □

Example 7.3: Consider again the ripple-carry adder ILA of Example 5.1. Figures 5.4a and 5.4b show the basic cell, which is a full adder, and its flow table F_a. Every direct

input (or column) in the cell flow table defines a distinguishing sequence, hence F_a is reduced. Also from the flow diagram of Figure 5.4c we see that F_a is an RT flow table. Therefore by Theorem 7.2, a ripple-carry adder ILA of arbitrary size is C-testable. Table 7.2 lists a complete set of C-tests for the adder array. The C-tests t_4 and t_5 for the transitions τ_4 and τ_5 actually verify these two transitions in alternating cells at the same time. This is because $t_4 = s(t_5)$ and $t_5 = s(t_4)$. Therefore the number of C-tests needed for an adder array is only eight, which is the minimum possible value.

7.4 Design Verification

We next consider the task of verifying the correctness of the logic design of a combinational array with direct outputs. A general approach to this problem is to check the complete input-output behavior, i.e., the truth-table, of the entire array. Following [Dias 1976], we refer to this as *truth-table verification*. Dias has shown that a complete set of his loop tests for a C-testable ILA verifies the truth-table of the ILA [Dias 1976]. We now consider the truth-table verification for a testable ILA (see Definition 5.3) which need not be C-testable. The results presented here are therefore a generalization of Dias' work.

Table 7.2. C-tests for a ripple-carry adder ILA

No. i	Transition τ_i	C-test for τ_i
1	$0 \xrightarrow{00} 0$	$0,(00)^*$
2	$0 \xrightarrow{01} 0$	$0,(01)^*$
3	$0 \xrightarrow{10} 0$	$0,(10)^*$
4	$0 \xrightarrow{11} 1$	$0,(1100)^*$
5	$1 \xrightarrow{00} 0$	$1,(0011)^*$
6	$1 \xrightarrow{01} 1$	$1,(01)^*$
7	$1 \xrightarrow{10} 1$	$1,(10)^*$
8	$1 \xrightarrow{11} 1$	$1,(11)^*$

Consider an ILA D with an $n \times m$ cell flow table F; Figure 7.1 shows the structure of D. Let T_D be a test set that verifies all mn transitions in every cell of D under the single-cell fault assumption. An example of T_D is CT_{ILA}, a set of C-tests for all flow table transitions when D is C-testable. It is assumed here that the flow table F is reduced, therefore SDS sequences (Definition 7.2) can be used to observe the internal horizontal signals of D at its available direct output lines. We now prove that T_D also completely verifies the input-output behavior of D independently of the internal structure or realization of the cells. In other words, T_D verifies not only the truth-table of the individual cells, but also that of the entire array D.

Definition 7.5: The ILA D *passes* T_D if the primary output signals produced by D in response to every test pattern in T_D are the same as expected from a fault-free ILA.

Lemma 7.1: If the ILA D passes the test set T_D, then all possible X input signals, i.e., all n states, are applied by T_D to the horizontal input lines of every cell of D.

Proof: (By contradiction.) Suppose that fewer than n states have been applied by T_D to some cell L_i of D. Let x_j be a state not applied by T_D to the X input line of L_i.

Now T_D should include at least m test patterns that apply x_j at the X input of L_i so that the m transitions in row x_j of the flow table F can be verified in L_i. Let the actual X input signal applied to L_i by any one of these m test patterns be $x_p \neq x_j$. This implies that the cell L_{i-1} to the left of L_i produces an incorrect $\hat{X}$ output signal x_p. But this $\hat{X}$ output signal from L_{i-1} is verified by T_D using the distinguishing sequence $DS(x_j,x_p)$ applied to the Y input lines of the cells $L_i, L_{i+1}, \ldots$. Since D passes T_D, the $\hat{Y}$ output sequence generated by D for this distinguishing sequence must be correct. Hence the $\hat{X}$ output signal of L_{i-1} is x_j, and not x_p as assumed, and all n X input signals must be applied by T_D to cell L_i. In other words, if D passes T_D then all n states are identified by T_D at the X input lines of every cell in D.

Lemma 7.2: Consider any two primary input patterns $PI = a_0, d_0 d_1 \ldots d_{N-1}$ and $PI' = a'_0, d'_0 d'_1 \ldots d'_{N-1}$, of D. Let b_k and b'_k be the respective X input signals at the X input lines of the cell L_k, for $k = 0,1,\ldots,N-1$, when PI and PI′ are applied to D. If D passes T_D then $b_k = b'_k$ if and only if $a_k = a'_k$, where a_k and a'_k are the X input signals of L_k in the fault-free case.

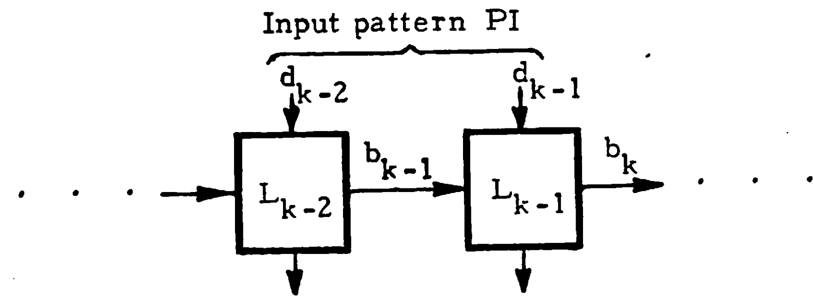
Proof: (By induction on k.)

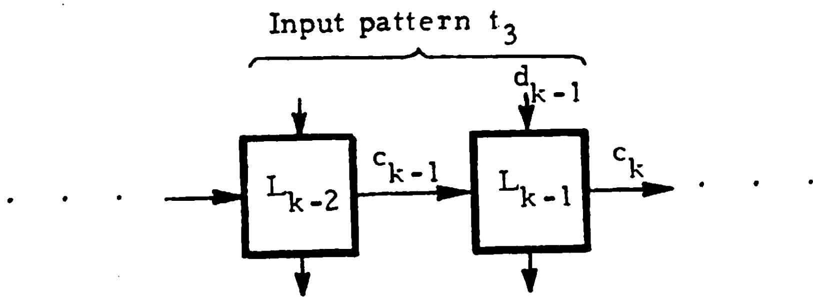
(a) $k = 1$: There exist two test patterns t_1 and t_2

in T_D that verify the transition τ_1: $a_0 \xrightarrow{d_0} a_1$ and τ_1': $a_0' \xrightarrow{d_0'} a_1'$ of F in cell L_0. If $a_1 \neq a_1'$ then $SDS(a_1) \neq SDS(a_1')$. Therefore D, which passes T_D, produces different $\hat{Y}$ output responses to t_1 and t_2, which contain $SDS(a_1)$ and $SDS(a_1')$, respectively. Thus $b_1 \neq b_1'$.

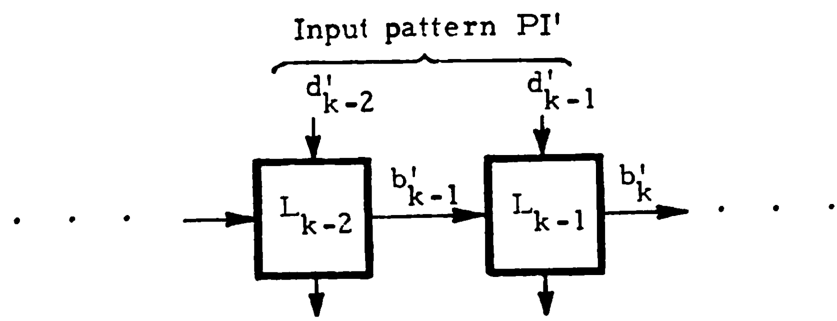
Suppose that $a_1 = a_1'$, but still $b_1 \neq b_1'$. This implies that two different states b_1 and b_1' produce the same $\hat{Y}$ response to $SDS(a_1) = SDS(a_1')$ in T_D, since D passes T_D. In other words, T_D identifies at most n-1 states at the X input lines of L_1. This cannot be true since T_D must apply all n states to the X input lines of L_1 if D passes T_D; see Lemma 7.1. Hence b_1 must equal b_1'. Therefore $b_1 = b_1'$ if and only if $a_1 = a_1'$.

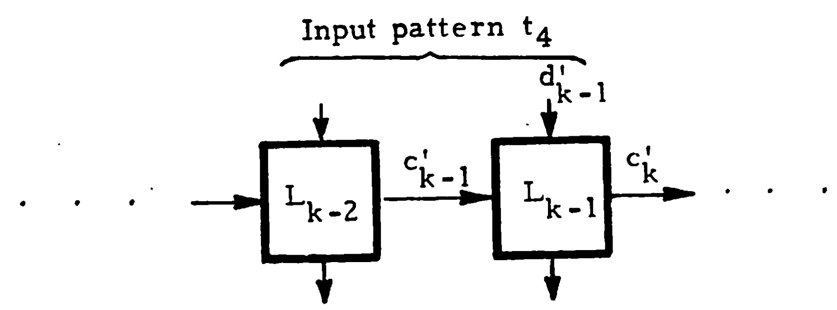
(b) $k > 1$: Assume that the lemma is true for k-1. Again there exist two test patterns, say t_3 and t_4, in T_D that verify the transitions τ_3: $a_{k-1} \xrightarrow{d_{k-1}} a_k$ and τ_3': $a_{k-1}' \xrightarrow{d_{k-1}'} a_k'$ in cell L_{k-1} of D. Let c_{k-1} (c_{k-1}') and c_k (c_k') be the horizontal signals produced by t_3 (t_4) at the X input lines of the cells L_{k-1} and L_k, respectively, as shown in Figure 7.3. Since the lemma is true for k-1, we have $c_{k-1} = b_{k-1}$ and $c_{k-1}' = b_{k-1}'$. Due to the application of the same transitions to the cell L_k by t_3 and PI (see Figure 7.3a) it is clear that $c_k = b_k$. Similarly $c_k' = b_k'$ (see Figure 7.3b). Using an argument similar to the case





(a)





(b)

Figure 7.3. Cell input signals produced by the array input patterns (a) PI and t_3, and (b) PI′ and t_4, of an array D of Lemma 7.2

of $k=1$ as above, we find that $b_k = b'_k$ if and only if $a_k = a'_k$ for any k.

<u>Theorem 7.3</u>: Let T_D be a complete test set for single-cell faults in an ILA D. D possesses the fault-free truth-table if and only if it passes T_D.

<u>Proof</u>: *Necessity.* If D does not pass T_D, then D produces an incorrect primary output signal for at least one test pattern, say t, of T_D. This implies that D possesses an incorrect truth-table entry corresponding to the primary input t. In other words D has a faulty truth-table.

Sufficiency. If D passes T_D, then from Lemma 7.2 the following is true. The X input signals generated at the X input lines of cells L_{k-1} and L_k by $PI = a_0, d_0 d_1 \ldots d_{N-1}$ and by a test pattern t_5 in T_D are the same, where t_5 verifies the transition $a_{k-1} \xrightarrow{d_{k-1}} a_k$ in cell L_{k-1}. This implies that the $\hat{Y}$ output signal of L_{k-1} produced by PI is the same as the one produced by t_5. It is also the same as that expected in a fault-free array, since D passes T_D. Since this is true for all k, the array D produces the expected primary output patterns for every primary input pattern. In other words, D has the truth-table of a fault-free ILA. Therefore, the test set T_D for single-cell faults also detects all detectable multiple-cell faults. □

The next result follows directly from Theorems 7.2 and 7.3.

Corollary 7.1: A set of C-tests for all flow table transitions of a C-testable ILA also verifies completely the logic design of the ILA.

The number of array input patterns required to verify the logic design of a C-testable ILA is therefore a constant, independent of the array size. Therefore C-testability has two main advantages. It simplifies ILA testing by providing a small number of easily-derived C-tests. Secondly, a complete set of C-tests can be used during the initial design process to verify not only the logic design of the individual cells, but also that of the entire ILA.

7.5 Design for C-testability

Consider a non-C-testable ILA with direct outputs. Let F be the $n \times m$ flow table of its basic cell L. We now present a method to modify F so that an array of modified L′ cells, with F′ as the new cell flow table, is C-testable. Our approach is to make use of Theorem 7.2 and change F to a reduced RT flow table.

The proposed flow table modification scheme is shown in Figure 7.4. It requires the addition of a new column y_a to the given flow table F. The next-state and the out-

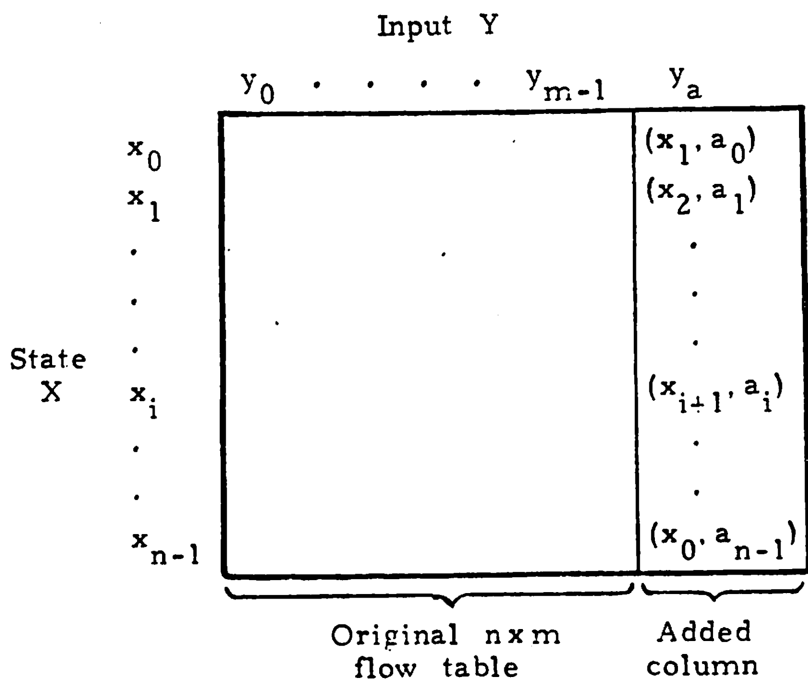


Figure 7.4. Flow table modification to make an array with direct outputs C-testable

put entries in y_a are designed to make F' an RT table and a reduced table, respectively. The next-state entry $\hat{x}(x_i,y_a)$ in row x_i and column y_a is $x_{i+1 \pmod n}$ for all i. Therefore y_a links all n states of F in a single loop in the flow diagram making it strongly connected. Hence F' is an RT flow table as required.

Next the output entries $a_0,a_1,\ldots,a_{n-1}$ in the new column y_a are selected to give F' a distinguishing sequence with respect to $\hat{Y}$. For simplicity, it is assumed that the cells considered here have only one $\hat{Y}$ output line. We therefore propose a procedure of assigning either 0 or 1 to the a_i's to make F' reduced. A similar procedure can be used in the general case where the basic cell has $q \geqq 1$ direct output lines, so that the a_i's can assume 2^q different values.

Let Q be an input sequence obtained by repeating y_a p times, i.e., $Q = y_a^p$. Our goal here is to make Q a minimum-length distinguishing sequence for F'. Note that the use of a minimum-length distinguishing sequence to construct C-tests for F' reduces the period of the C-tests, and hence the number of tests needed for the ILA. The $\hat{Y}$ output sequence produced by F' in response to Q with x_i as the initial state is

$$A_i = a_i a_{i+1 \pmod n} a_{i+2 \pmod n} \cdots a_{i+p-1 \pmod n} \tag{7.8}$$

Now F' is reduced if the output sequence A_i is different from any other output sequence A_j, where $i \neq j$, for $i = 0, 1, \ldots, n-1$. The input sequence Q is then a distinguishing sequence for the flow table. The minimum length of Q is $\lceil \log_2 n \rceil$ since a sequence of length p bits can produce at most 2^p distinct patterns.

In his flow table modification scheme [Dias 1976], which is similar to the above proposed method, Dias selects the output entries in y_a as follows: $a_i = 0$, for $0 \leqq i \leqq n-2$ and $a_{n-1} = 1$. This selection makes $Q = y_a^d$ a distinguishing sequence of length $d = n-1$, which is much greater than the minimum possible length, $\lceil \log_2 n \rceil$. Hence Dias' method is not optimal with respect to the length of Q. We now present a method of designing the a_i's using shift-register sequences so that Q becomes a minimum-length distinguishing sequence for F'.

Let $A = a_0 a_1 \ldots a_{n-1}$ be the sequence formed by the output entries in the new column y_a of Figure 7.4. Consider a p-bit serial-input parallel-output shift register SR where $p = \lceil \log_2 n \rceil$. Suppose that the sequence A is entered into SR repeatedly, i.e., the serial input of SR is $A^* = AAA\ldots$, as shown in Figure 7.5. Then at any instant of time t, the p-bit output word, i.e., the state of SR, is the A_i sequence of Equation (7.8) for some i. If A_i is the state of SR at time t, then the state of SR at time t+1 will be A_{i+1}. The state of SR viewed at successive

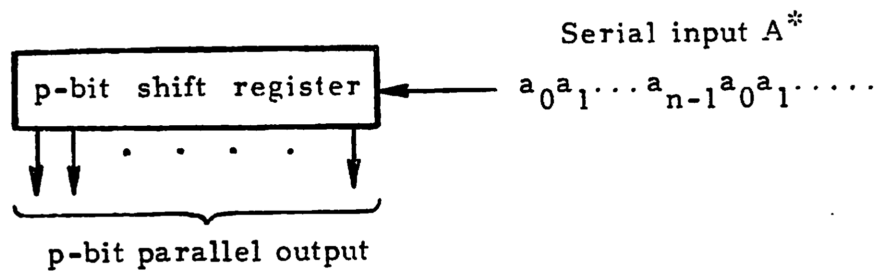


Figure 7.5. A p-bit serial-input parallel-output shift register

clock periods therefore forms the sequence $A_i A_{i+1} A_{i+2} \cdots$.

Since the input sequence A^* is periodic, the state sequence of SR is also periodic. If the period of the state sequence is the same as that of A^*, i.e., n, then the sequence A is referred to as a *shift-register sequence* of length n [Golomb 1967]. These concepts are illustrated in the following example.

Example 7.4: Let $n = 5$ and $A = 00111$. In a p-bit shift register, where $p = \lceil \log_2 5 \rceil = 3$, the input sequence A^* produces the following state sequence: 001, 011, 111, 110, 100, 001, 011,... . Therefore A is a shift register sequence of length 5. □

Suppose that $A = a_0 a_1 \ldots a_{n-1}$ is a shift-register sequence of length n. Then, by definition, the state sequence $A_i A_{i+1} A_{i+2} \cdots$ generated by SR is periodic with period n, and $A_i \neq A_j$ for $i \neq j$. Now the $\hat{Y}$ output sequence generated by applying $Q = y_a^p$ to the flow table F′ of Figure 7.4, is A_i of length p. As discussed before, for Q to be a distinguishing sequence for F′, $A_i \neq A_j$ for $i \neq j$. Hence Q will be a minimum-length distinguishing sequence if and only if A is a shift-register sequence of length n. For any given n, we are therefore interested in constructing a shift-register sequence A of length n. A method of doing it is given in the proof of the following theorem.

Theorem 7.4: For the modified flow table F′ of Figure 7.4, output entries $a_0, a_1, \ldots, a_{n-1}$ exist for any n, such that y_a^p, i.e., y_a repeated p times, is a minimum-length distinguishing sequence for F′.

Proof: We have to show that a shift-register sequence $A = a_0 a_1 \ldots a_{n-1}$ of length n exists for any n. It is well-known that for any positive integer p there exists a linear shift-register sequence LS of maximum length 2^p-1 [Golomb 1967, Peterson and Weldon 1972]. (A shift-register sequence is said to be linear if the present serial input signal is a linear function of the previous serial input signals.) Such a sequence generates all 2^p states of a p-bit shift-register except the all-0 state. One can derive shift-register sequences of other lengths from the sequence LS using the following method due to Golomb [Golomb 1967, pp. 192-193].

Case 1: $n = 2^p$.

Let $LS_0 = b_0 b_1 \ldots b_{n-2}$ be a maximum-length linear shift-register sequence. Since LS_0 contains every subsequence of length p, except the all-0 subsequence, there exists a subsequence B = 000...01 with p-1 leading zeros. To obtain a shift-register sequence A of length $n = 2^p$, it is required to introduce in LS_0 the all-0 state or the subsequence 000...00 of length p. This is achieved by inserting an additional 0 between any two of the p-1 leading zeros of B.

The resulting sequence of length n is the desired shift-register sequence A.

Case 2: $n < 2^p$.

Again let $LS_i = b_i b_{i+1} b_{i+2} \ldots b_{i+2^p-2}$ be a maximum-length shift-register sequence. Consider the bit-by-bit modulo-2 sum $LS_i \oplus LS_{i+n}$ of the two sequences LS_i and LS_{i+n}, where $LS_{i+n} = b_{i+n} b_{i+n+1} \ldots b_{i+2^p-2} b_i b_{i+1} \ldots b_{i+n-1}$. LS_{i+n} is LS_i rotated n bit positions to the left. Theorem 4.3 in [Golomb 1967] states that the 2^p-1 sequences $LS_i, LS_{i+1}, \ldots, LS_{i+2^p-2}$ form an Abelian group with respect to the operator $\oplus$. Therefore the modulo-2 sum of any two sequences in this group is a sequence belonging to the same group. Hence, for some integer M, $LS_i \oplus LS_{i+n} = LS_{i+M}$. Thus LS_{i+M} also possesses a subsequence $B = 00\ldots01$ having p-1 leading zeros. Hence for some j, we have $b_{j+M+1} = b_{j+M+2} = \cdots = b_{j+M+p-1} = 0$ and $b_{j+M+p} = 1$. This implies that $b_{j+1} = b_{j+n+1}$, $b_{j+2} = b_{j+n+2}, \ldots, b_{j+p-1} = b_{j+n+p-1}$, $b_{j+p} = \bar{b}_{j+n+p}$.

In sequence LS_i we have therefore found two subsequences of length p, spaced n bit positions apart, that are identical in their first p-1 bit positions and different in the pth bit position. The subsequence $b_{j+1} b_{j+2} \ldots b_{j+n}$ of length n falling between these two subsequences is then the desired shift-register sequence A. Thus for any positive integer n, it is possible to construct the output entries $a_0, a_1, \ldots, a_{n-1}$ in column y_a of Figure 7.4

such that F′ is reduced with a minimum-length distinguishing sequence y_a^p. □

Example 7.5: Let $p = 3$ and $n = 2^3 = 8$. A maximum-length linear sequence is $LS_0 = b_0b_1b_2b_3b_4b_5b_6 = 1001011$ of length $2^3 - 1 = 7$. The subsequence $b_1b_2b_3$ has $p-1 = 2$ leading zeros. Therefore a new 0 is introduced between b_2 and b_3 to form a shift-register sequence $A = b_0b_1b_2 0 b_3b_4b_5b_6 = 10001011$ of length $n = 8$. The corresponding periodic sequence of the 3-bit shift-register state is 100,000,001,010,101,011,111, 110,100,000,... .

Using LS_0 we can also construct shift-register sequences of lengths less than 8. For example, let the desired length be $n = 5$. Now $LS_n = LS_5 = b_5b_6b_0b_1b_2b_3b_4 = 1110010$, and hence $LS_0 \oplus LS_5 = 0111001$, which is the same as LS_4. The position of the subsequence 001 in LS_4 points to the two subsequences of length $p = 3$ in LS_0, that are identical in the first two bit positions and different in the third. These subsequences are therefore $b_4b_5b_6$ and $b_2b_3b_4$. Hence the desired shift-register sequence of length five is $b_4b_5b_6b_0b_1 = 01110$. The corresponding shift-register state sequence of period five is 011,111,110,100,001,011, 111,... . □

Theorem 7.5: An ILA of cells with the modified flow table F′ of Figure 7.4, is C-testable.

Theorem 7.5 follows directly from the foregoing discussions and Theorem 7.4.

C-tests for an array of modified cells are easily constructed as follows. If $\tau_{ij}: x_i \xrightarrow{y_j} \hat{x}(x_i, y_j) = x_k$ is any transition of the original flow table F, then a C-test for τ_{ij} is $C(\tau_{ij}) = x_i, (y_j y_a^p y_a^q)^*$, where y_a^p is the minimum-length distinguishing sequence, and y_a^q is the transfer sequence that takes F back to x_i. A C-test for any transition $\tau_{ia}: x_i \xrightarrow{y_a} x_{i+1}$ in the new column y_a is $C(\tau_{ia}) = x_i, (y_a y_a^p y_a^{n-1-p-1})^* = x_i, (y_a^n)^* = x_i, y_a^*$. Hence all n transitions in the new column can be verified in every cell with only n tests of the form x_i, y_a^*.

The number of tests V for an array of arbitrary size composed of modified cells is bounded as follows

$$mn(p+1)+n \leqq V \leqq mn(p+n)+n \qquad (7.9)$$

In Dias' modification scheme, a loop test $L(\tau_{ij})$ for the above transition τ_{ij} is $x_i, (y_j y_a^{n-1} y_a^d)^*$ where y_a^{n-1} is the distinguishing sequence and y_a^d is transfer sequence. Comparing the C-test $C(\tau_{ij})$ and the loop test $L(\tau_{ij})$, we can make the following observations. If $(p+q) < (n-1)$, then the periodicity of $L(\tau_{ij})$ is greater than that of $C(\tau_{ij})$, while if $(p+q) \geqq (n-1)$, then the periodicities are the same. Hence the length of the C-tests for the modified array never exceeds the length of Dias' corresponding loop tests.

The modification scheme given here can be expected to result in fewer tests. Equivalent bounds on the number of tests V' required by Dias' scheme are

$$mn^2+n \leq V' \leq 2n^2(m+1) \qquad (7.10)$$

<u>Example 7.6</u>: Consider again the incrementer ILA introduced in Example 5.2 of Section 5.4. Its cell flow table which appears in Figure 5.5b is not an RT flow table; hence the ILA is not C-testable. Applying our modification method to Figure 5.5b yields the flow table shown in Figure 7.6a. Clearly the modified flow table is a reduced RT flow table. The C-tests for the six possible state transitions are given in Figure 7.6b. To verify the transition τ_3 in every cell of the array, two array input patterns are required, while the remaining five transitions can each be tested with only one test pattern. Hence a modified incrementer ILA of arbitrary size can be completely tested with only 7 tests. □

Figure 7.7 shows a hardware realization of the modified cell L' having the general flow table F' of Figure 7.4. It consists of the original cell L and four new logic modules NL_1, NL_2, M_X and M_Y. The new circuit NL_1 realizes the next-state function in the new column y_a, while NL_2 generates the output entries of y_a. The two multiplexers

		Input Y		
		0	1	y_a
State X	0	(0, 0)	(0, 1)	(1, 0)
	1	(0, 1)	(1, 0)	(0, 1)
		Original flow table		Added column

(a)

No. i	Transition τ_i	C-test for τ_i
1	$0 \xrightarrow{0} 0$	$0, 0^*$
2	$0 \xrightarrow{1} 0$	$0, 1^*$
3	$1 \xrightarrow{0} 0$	$1, (0y_a)^*$
4	$1 \xrightarrow{1} 1$	$1, 1^*$
5	$0 \xrightarrow{y_a} 1$	$0, y_a^*$
6	$1 \xrightarrow{y_a} 0$	$1, y_a^*$

(b)

Figure 7.6. (a) Modified 1-bit cell flow table for an incrementer ILA and (b) its C-tests

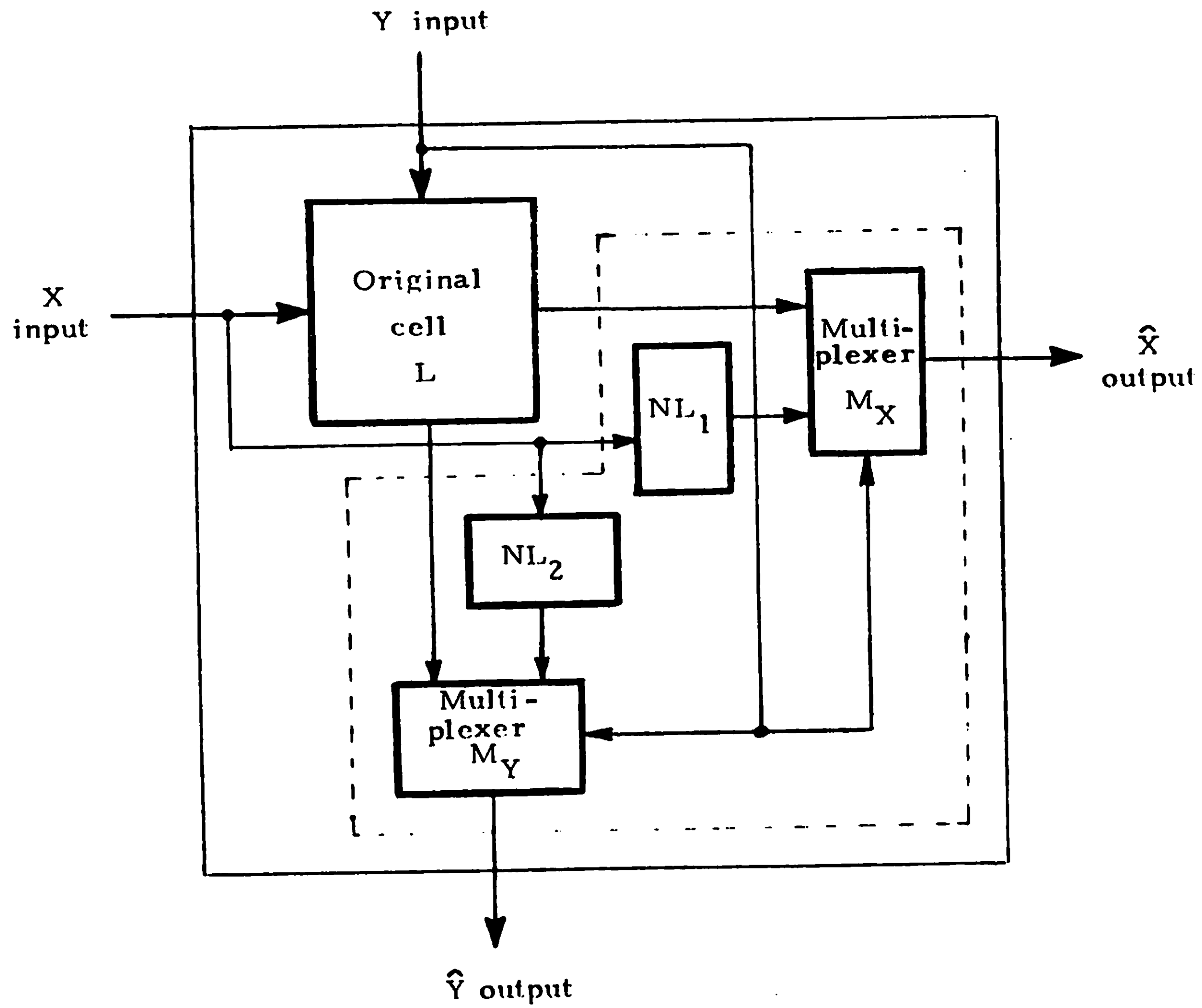


Figure 7.7. Implementation of the cell modification method for C-testability

M_X and M_Y select the outputs of NL_1 and NL_2, respectively, if the Y input is y_a; otherwise they select the $\hat{X}$ and $\hat{Y}$ outputs of the original cell. Therefore, the added logic circuits are simple combinational modules that are independent of the original cell L. Since the design of L is unaltered, the same modification circuitry can be used with any cell L that has an $n \times m$ flow table.

CHAPTER 8

BILATERAL AND SEQUENTIAL ARRAYS

The basic concepts and properties of C-testable arrays were discussed in detail in Chapters 5, 6 and 7. There only unilateral combinational ILAs, i.e., the Class 1 and 5 arrays of Table 5.1, were considered. In this chapter we analyze C-testability in more general ILAs. It is shown that most of the earlier results concerning unilateral ILAs can be extended to bilateral combinational ILAs. These results include the characterization of C-testability, and a design modification method to make any ILA C-testable. Also a useful class of ILAs composed of sequential cells is considered.

8.1 C-testability in Bilateral Arrays

In this section we consider bilateral ILAs of identical combinational cells with the structure shown in Figure 8.1. Each cell has independent vertical input and output lines, as well as horizontal input and output lines to neighboring cells. As before, the horizontal signal sets of each cell are denoted by X, W, $\hat{X}$ and $\hat{W}$, while the vertical input and output signal sets are Y and $\hat{Y}$, respec-

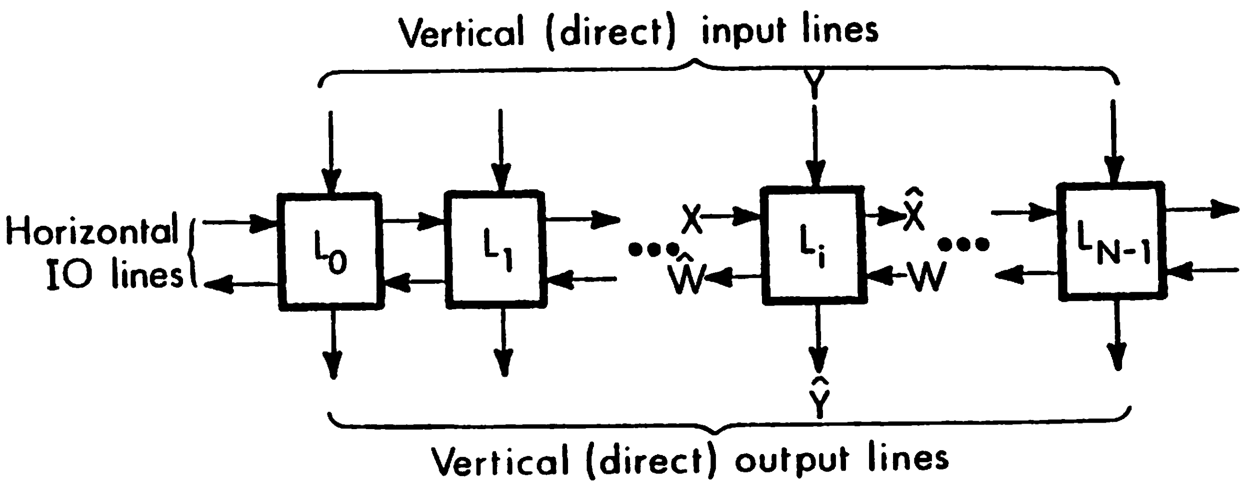


Figure 8.1. Structure of a bilateral ILA

tively; see Figure 8.1. Arbitrary members of X, $\hat{X}$, W, $\hat{W}$, Y and $\hat{Y}$ are denoted by x, $\hat{x}$, w, $\hat{w}$, y and $\hat{y}$, respectively. Because of the interconnection structure, $X = \hat{X}$ and $W = \hat{W}$. Let $|X| = |\hat{X}| = n$, $|W| = |\hat{W}| = p$, and $|Y| = m$. Flow tables can be defined in several ways for bilateral ILAs.

Definition 8.1: An $n \times mp$ table whose rows represent X and whose columns represent $Y \times W$ is called the *X flow table* of the basic cell of an array. For simplicity, a member (a,b) of the Cartesian product $A \times B$ will sometimes be written as ab. Thus a typical entry e in row x_i and column $y_j w_k$ of the X flow table is written as $(\hat{x}_a, \hat{y}_c \hat{w}_b)$, where $\hat{x}_a$ is the $\hat{X}$ output function $\hat{x}(x_i, y_j w_k)$, $\hat{w}_b$ is the $\hat{W}$ output function $\hat{w}(x_i, y_j w_k)$, and $\hat{y}_c$ is the direct output function $\hat{y}(x_i, y_j w_k)$.

The *W flow table* of the cell is defined similarly with X and W interchanged. The members of the set $X \times \hat{W}$ represent the input/output signal pairs appearing at the left boundary of a cell in the array, and are called *states* of a cell. A table in which $X \times \hat{W}$ is the set of rows or states, and Y is the set of columns is called the *$X\hat{W}$ flow table* of the cell. An entry in this table is a possibly empty set of pairs, each consisting of next state and output function values. The $X\hat{W}$ flow table is easily derived from the X flow table. For example, consider the entry $e = (\hat{x}_a, \hat{y}_c \hat{w}_b)$ in row x_i and column $y_j w_k$ of the X flow table. The cor-

responding entry in the $X\hat{W}$ flow table has the next state $\hat{x}_a w_k$ and the $\hat{Y}$ output value $\hat{y}_c$ in row $x_i \hat{w}_b$ and column y_j. Each next state entry in the $X\hat{W}$ flow table also specifies a (state) *transition* of the form τ: $x_i \hat{w}_b \xrightarrow{y_j} \hat{x}_a w_k$. Another table called the *$\hat{X}W$ flow table* in which the set $\hat{X} \times W$ is the set of rows (or states) and Y is the set of columns, is defined similarly. Figure 8.2 gives examples of the various flow tables. Here multiple entries are separated by semicolons, and the symbol $\emptyset$ indicates a null or empty entry.

Let $a_0, b_0 b_1 \ldots b_{N-1}, c_{N-1}$ denote the input pattern applied to a bilateral array, where a_0 is the X input signal to the leftmost cell L_0, b_i is the Y input signal to L_i, for $i = 0,1,\ldots,N-1$, and c_{N-1} is the W input to the rightmost cell L_{N-1}. The $\hat{Y}$ output pattern S of the array can easily be determined from the $X\hat{W}$ flow table of a single cell. S is the output sequence defined by the $X\hat{W}$ flow table for the input sequence $b_0 b_1 \ldots b_{N-1}$ with $a_0 c'$ and $a' c_{N-1}$ as the initial and final states, respectively. The corresponding sequence of next states gives the X and W signals appearing at the cell boundaries.

C-testing in bilateral ILAs can be analyzed using the approach developed earlier for unilateral ILAs. Again the fault model of Section 2.4 is used with each cell treated as a single module, and at most one cell assumed to be faulty. However, it will be shown later that for bilateral

X \ Y×W	00	01	10	11
0	(0,00)	(1,01)	(1,11)	(0,11)
1	(0,11)	(1,01)	(0,01)	(0,01)

(a)

State $X \times \hat{W}$ \ Vertical input Y	0	1
00	(00,0)	ϕ
01	(11,0)	(10,1);(01,1)
10	ϕ	ϕ
11	(00,1);(11,0)	(00,0);(01,0)

(b)

State $\hat{X} \times W$ \ Vertical input Y	0	1
00	(00,0);(11,1)	(11,0)
01	ϕ	(01,1);(11,0)
10	ϕ	(01,1)
11	(01,0);(11,0)	ϕ

(c)

Figure 8.2. (a) X flow table, (b) $X\hat{W}$ flow table, and (c) $\hat{X}W$ flow table of a bilateral cell

ILAs with direct outputs a single-fault test set also detects all detectable multiple-cell faults.

Consider a bilateral cell L with an $X\hat{W}$ flow table F of size $np \times m$. According to the fault model, to test completely an array of N L-type cells, it is necessary and sufficient to verify all npm transitions of F in every cell of the array. Consider some transition τ: $x_i\hat{w}_b \xrightarrow{y_j} \hat{x}_a w_k$ of F. To verify this transition in cell L_i of the array, it is necessary to apply the signals x_i, y_j and w_k to the respective input lines of L_i. At the same time, the $\hat{X}$, $\hat{W}$ and $\hat{Y}$ output signals of L_i should be observed at the primary outputs of the array. Let q be the number of array input patterns required to verify τ in every cell of the array. C-testability requires q to be a constant independent of the array size N. The concept of repeatable transitions (see Definition 5.5) introduced in Chatper 5 is useful here as well. Suppose that τ is not a repeatable transition. Then while verifying τ in cell L_0 say, it is not possible to verify τ in any other cell at the same time. This is because the initial state $x_i\hat{w}_b$ of τ is not reachable from the state $\hat{x}_a w_k$. Hence to verify τ in every cell we need at least N array input patterns, one for each cell. In other words, q depends on N. Hence we have the following necessary condition for C-testability.

Theorem 8.1: A bilateral ILA is C-testable only if the $X\hat{W}$ cell flow table is an RT flow table.

For example, the $X\hat{W}$ flow table of Figure 8.2b is not an RT flow table, since the underlined transition is not repeatable. On the other hand, the $X\hat{W}$ flow table of Figure 8.3a is an RT flow table. Since the $X\hat{W}$ and $\hat{X}W$ flow tables are complementary, an $\hat{X}W$ flow table is an RT flow table if and only if the corresponding $X\hat{W}$ flow table is an RT flow table.

Next we characterize C-testability in bilateral ILAs that have no direct outputs and only a single horizontal line in each direction. The characterization of C-testability in general bilateral ILAs without direct outputs is unsolved. The problems encountered here are very similar to those discussed in Sections 6.1 and 6.2 for unilateral ILAs. The following result is a direct extension of Theorem 6.4.

Theorem 8.2: A bilateral ILA with only a single horizontal line in each direction and with no direct outputs is C-testable if and only if there are no constant columns in the X and W cell flow tables. Furthermore, if the ILA is C-testable then it is optimally C-testable.

Proof: *Necessity.* To test the ILA it is necessary and sufficient to verify all next-state entries in both the X

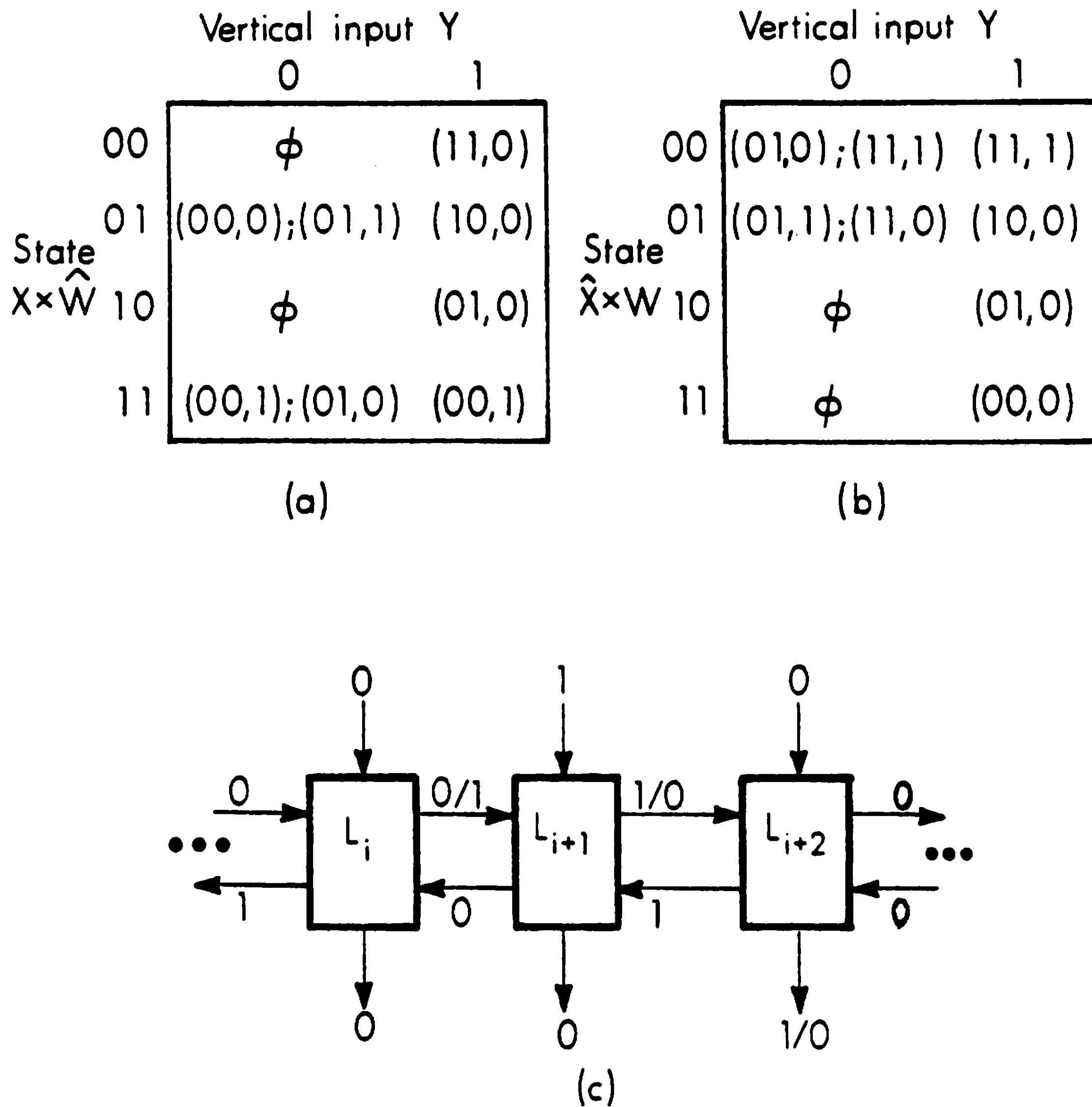


Figure 8.3. (a) The $X\hat{W}$ flow table of a cell L, (b) the $\hat{X}W$ flow table of L, and (c) the application of a distinguishing sequence to an array of L-type cells

and the W flow tables of the cell. If some column $y_j w_k$ in the X flow table is a constant column, then the two next-state entries $\hat{x}(x_0, y_j w_k)$ and $\hat{x}(x_1, y_j w_k)$ are identical. Therefore if any one of these next-state entries is being verified in some cell L_i, then the same entry cannot be verified in any other cell L_j for $j > i$. This is because the Y input signal corresponding to a constant column cannot be used to propagate a faulty X signal since its next-state entries are all identical. Therefore entries in a constant column cannot be verified in more than one cell at a time; hence there exists no C-test for such entries. Similarly the W flow table of a C-testable ILA must not possess any constant columns.

Sufficiency. Suppose that neither the X nor the W flow table has any constant column. Then a typical column y_j in the $X\hat{W}$ cell flow table is as shown in Figure 8.4, where $x_a, \bar{x}_a, x_b, \bar{x}_b \in X = \{x_0, x_1\}$ and $w_c, \bar{w}_c, w_d, \bar{w}_d \in W = \{w_0, w_1\}$. For all values of x_a, x_b, w_c and w_d, the column y_j is a permutation column, i.e., every state appears exactly once as a next-state entry. Therefore, from Theorem 6.2 for the unilateral case, the bilateral ILA is optimally C-testable. □

An optimal set of C-tests for the bilateral ILA can be constructed as follows. The entry in row $x_0 w_0$ and column y_j of the cell flow table of Figure 8.4 can be tested si-

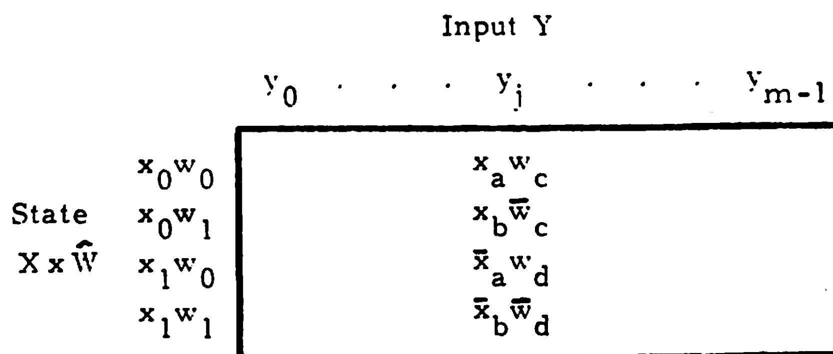


Figure 8.4. A typical column in the $X\hat{W}$ cell flow table of a bilateral ILA with no direct outputs and with only a single horizontal line in each direction

multaneously in every cell of the ILA by applying x_0 to the X input of the leftmost cell L_0, y_j to the Y input of every cell, and w_0 to the W input of the rightmost cell L_{N-1}. This test pattern is denoted by x_0, y_j^*, w_0. The tests for the remaining three entries of y_j, are x_0, y_j^*, w_1, x_1, y_j^*, w_0 and x_1, y_j^*, w_1. Thus the entries of each column can be verified in all cells with only four tests. Consequently the ILA is completely tested with only 4m test patterns and therefore is optimally C-testable.

8.2 Bilateral Arrays with Direct Outputs

The ILAs to be considered next are the Class 6 ILAs of Table 5.1 whose structure is depicted in Figure 8.1. A practical example of such an ILA is the subarray formed by the ALU modules in a processor array of the Intel 3002 slices [Intel 1976]. In this subarray the horizontal X and W lines between the cells are the carry and shift lines, respectively. In C-testing a Class 6 ILA, the available direct outputs are efficiently used for observing the internal X and W signals. Here the application and propagation of the array's internal X (W) signals are analyzed using the cell's $X\hat{W}$ ($\hat{X}W$) flow table.

We now extend the definitions of the conventional distinguishing sequences of a unilateral ILA presented in

Section 7.1 (see Definitions 7.1 and 7.2) to the bilateral case.

Definition 8.2: A *distinguishing sequence* $DS(x_a\hat{w}_c, x_b\hat{w}_c)$ for a pair of states $x_a\hat{w}_c$ and $x_b\hat{w}_c$ of an $X\hat{W}$ flow table F is a Y input sequence which, when applied to F, produces two different $\hat{Y}$ output sequences with $x_a\hat{w}_c$ and $x_b\hat{w}_c$ as the initial states.

Definition 8.3: A *set of distinguishing sequences* $SDS(x_a\hat{w}_c)$ for a state $x_a\hat{w}_c$ of an $X\hat{W}$ flow table F is a set of Y input sequences which, when applied to F with $x_a\hat{w}_c$ as the initial state, produces a set of $\hat{Y}$ output sequences different from the sets of output sequences produced with any other state $x_b\hat{w}_c$, for all $x_b \in X$, as the initial state. Hence we have

$$SDS(x_a\hat{w}_c) = \{DS(x_a\hat{w}_c, x_b\hat{w}_c): x_b \in X \text{ and } x_b \neq x_a\} \quad (8.1)$$

Definition 8.4: An $X\hat{W}$ flow table is *reduced* with respect to the $\hat{Y}$ output if there exists a $DS(x_a\hat{w}_c, x_b\hat{w}_c)$ for every $x_a, x_b \in X$ and for every $\hat{w}_c \in \hat{W}$.

Distinguishing sequences for an $\hat{X}W$ flow table are defined in the same way. These distinguishing sequences are useful in propagating faulty $\hat{X}$ and $\hat{W}$ output signals to the observable outputs of an ILA as demonstrated in the following example.

Example 8.1: Consider the $X\hat{W}$ flow table shown in Figure 8.3a. The possible distinguishing sequences are $DS(00,10) = 10$ and $DS(01,11) = 0$. Figure 8.3c demonstrates how the sequence $DS(00,10)$ can be used to verify the $\hat{X}$ output signal of any cell in an array. The $\hat{X}$ output signal of the cell L_i, in which the transition $\tau: 01 \xrightarrow{0} 00$ is being verified, is propagated to the $\hat{Y}$ outputs by applying signals 1 and 0, corresponding to the sequence $DS(00,10)$, to the Y input lines of the cells L_{i+1} and L_{i+2} as shown. Any change in the $\hat{X}$ output signal of L_i due to a fault f in L_i is detected at the $\hat{Y}$ output of L_{i+2}. A change from a to b, where $a,b \in \{0,1\}$, in the signal value on a line induced by fault f is indicated as a/b in Figure 8.3c. Figure 8.3b gives the $\hat{X}W$ flow table of the cell L. It is evident that this flow table also has distinguishing sequences to verify the $\hat{W}$ output signal of a cell, and hence is reduced.

Definition 8.5: Let D be an ILA whose $X\hat{W}$ flow table is a reduced RT table. Let $\tau: x_a\hat{w}_c \xrightarrow{y_j} \hat{x}_b w_d$ be any transition in the $X\hat{W}$ flow table. A *C-test* $C_{ix}(\tau)$ associated with this transition is a periodic input test pattern of the form

$$C_{ix}(\tau) = x_a\hat{w}_c, (y_j D_{ix} R_{ix})^* \qquad (8.2)$$

where D_{ix} is a distinguishing sequence in $SDS(\hat{x}_b w_d)$, and R_{ix} is a transfer sequence that takes the cell state back to $x_a\hat{w}_c$.

$C_{ix}(\tau)$ verifies the $\hat{X}$ part of τ with respect to D_{ix} in the leftmost cell of D, and in all cells spaced periodically at intervals k_i along D, where k_i is the periodicity of $C_{ix}(\tau)$, i.e., the number of cells spanned by $y_jD_{ix}R_{ix}$. Thus $C_{ix}(\tau)$ tests cells $L_0, L_{k_i}, L_{2k_i}, \ldots$. Let $s^q(C_{ix}(\tau))$ denote the input sequence obtained by shifting the test pattern applied to D by $C_{ix}(\tau)$ $q < k_i$ cell positions to the left; the operator s^q is formally defined in Definition 6.1. $s^q(C_{ix}(\tau))$ verifies the $\hat{X}$ part of τ in the cells $L_{k_i-q}, L_{2k_i-q}, L_{3k_i-q}, \ldots$ using the distinguishing sequence D_{ix}. Let $CT_x(\tau)$ denote the set of $\sum_{i=1}^{r} k_i$ C-tests of the form $s^q(x_a\hat{w}_c, (y_jD_{ix}R_{ix})^*)$ obtained by allowing D_{ix} to range over all r members of $SDS(\hat{x}_bw_d)$, and by allowing q to range over $0,1,\ldots,k_i-1$. $CT_x(\tau)$ can be formally written as

$$CT_x(\tau) = \{s^q(x_a\hat{w}_c, (y_jD_{ix}R_{ix})^*: \quad 0 \leq q \leq k_i-1 \text{ and } D_{ix} \in SDS(\hat{x}_bw_d)\} \tag{8.3}$$

$CT_x(\tau)$ completely verifies the $\hat{X}$ output signal produced by every cell of the array during the transition τ. Note that the size of $CT_x(\tau)$ is independent of the length of D. In a similar manner we can define the C-test $C_{jw}(\tau)$ and the set of C-tests $CT_w(\tau)$ which verify the $\hat{W}$ output signals produced by τ. Then a set of C-tests given by

$$CT(\tau) = CT_x(\tau) \cup CT_w(\tau) \tag{8.4}$$

completely tests τ in every cell of D. The size of $CT(\tau)$ is again independent of the length of D.

Theorem 8.3: There exist C-test sets $CT_x(\tau)$ and $CT_w(\tau)$ for every transition in the $X\hat{W}$ and $\hat{X}W$ flow tables of a cell, if both flow tables are reduced RT flow tables.

Proof: Let the $X\hat{W}$ and $\hat{X}W$ flow tables be reduced RT flow tables. Then by Definition 8.4 there exist SDSs for every state of the flow tables. Also since they are RT flow tables, the transfer sequences required in the C-tests $C_{ix}(\tau)$ and $C_{jw}(\tau)$ (see Equation (8.2)) exist. Hence there are C-test sets $CT_x(\tau)$ and $CT_w(\tau)$ for every transition in the $X\hat{W}$ and $\hat{X}W$ flow tables. □

Bounds on the length of C-tests are easily determined. The lengths of the transfer sequences R_{ix} and R_{jw} are each at most np-1, since the $X\hat{W}$ and $\hat{X}W$ flow tables are both RT tables with np states. Also for reduced flow tables $|D_{ix}|$ and $|D_{jw}|$ are both at most np-1. Hence from Definition 8.5 we have the following inequalities for the periodicities k_i and k'_j of $C_{ix}(\tau)$ and $C_{jw}(\tau)$, respectively.

$$1 \leqq k_i \leqq 1+(np-1)+(np-1) = 2np-1 \tag{8.5}$$

$$1 \leqq k'_j \leqq 1+(np-1)+(np-1) = 2np-1 \tag{8.6}$$

Consequently the size of the complete C-test set CT_D for a bilateral ILA D of arbitrary length is bounded as follows.

$$mnp \leq |CT_D| \leq mnp[(n-1)(2np-1)+(p-1)(2np-1)] \qquad (8.7)$$

The fixed-length C-test set CT_D is given by

$$CT_D = \bigcup_{i=1}^{mnp} CT(\tau_i) \qquad (8.8)$$

where τ_1 to τ_{mnp} are the mnp transitions of the $X\hat{W}$ flow table.

<u>Theorem 8.4</u>: A bilateral ILA with reduced $X\hat{W}$ and $\hat{X}W$ cell flow tables is C-testable if and only if the tables are RT flow tables.

<u>Proof</u>: *Necessity.* The necessity part follows directly from Theorem 8.1.

Sufficiency. If the $X\hat{W}$ and $\hat{X}W$ flow tables are reduced RT flow tables, then by Theorem 8.3 there exist C-test sets $CT_x(\tau)$ and $CT_w(\tau)$ for every transition in these tables. A complete C-test set CT_D for the ILA using these C-tests is defined by Equation (8.8) above. Since $|CT_D|$ is independent of the array size, the ILA is C-testable. □

<u>Example 8.1 (cont'd)</u>: The $X\hat{W}$ and $\hat{X}W$ flow tables of the cell L in Figure 8.3 are reduced RT flow tables. Hence an

array D of L-type cells is C-testable using C-tests. For example consider the transition τ: $01 \xrightarrow{0} 00$ in the $X\hat{W}$ flow table. A C-test for τ in $CT_x(\tau)$ is $C_{1x}(\tau) = 01,(01010)^*$; see Figure 8.3c. If the array length is 12, for example, then $C_{1x}(\tau) = 01,010100101001$ where the length of the periodic Y pattern is adjusted to match the number of cells present in D. C_{1x} applies the transition τ to the cells $L_0, L_5, L_{10}, \ldots$, and verifies the corresponding $\hat{X}$ output signals. The shifted version $s^1(C_{1x}(\tau))$ is $00,(10100)^*$ and it verifies the $\hat{X}$ output signal of cells $L_4, L_9, L_{14}, \ldots$. Similarly, the other three shifted versions of $C_{1x}(\tau)$, $s^2(C_{1x}(\tau))$, $s^3(C_{1x}(\tau))$, $s^4(C_{1x}(\tau))$, test τ in the remaining cells of the array. Thus the $\hat{X}$ output entry in τ is verified in every cell of D with only five array input patterns (C-tests) independent of the size of D.

8.3 Design Verification

In Section 7.4 the design verification of unilateral ILAs was discussed in detail. It was proved that a test set that verifies all flow table transitions in every cell of a unilateral ILA, also completely verifies the input-output behavior or the truth-table of the array; see Theorem 7.3. This result is extended here in a straightforward fashion to bilateral ILAs.

Let T_B be a test set that verifies all transitions of the $X\hat{W}$ and $\hat{X}W$ cell flow tables in every cell of a bilateral

array B. It is assumed that T_B is a test set for single-cell faults, and that it uses appropriate distinguishing sequences to observe the internal X and W signals at the vertical direct outputs of B. (Note that B need not be C-testable.) The ILA B *passes* the test set T_B if the primary output responses of B to T_B are the same as expected from a fault-free array.

Theorem 8.5: The bilateral ILA B possesses the fault-free truth-table if and only if it passes the test set T_B.

In other words, T_B also verifies completely the truth-table of B. Hence T_B is also a test set for multiple-cell faults.

The proof of the above theorem is similar to that of Theorem 7.3 with transition, states and SDSs now referring to $X\hat{W}$ and $\hat{X}W$ flow tables of the bilateral cell instead of the conventional flow table of a unilateral cell. As a consequence of Theorem 8.5, a test set such as CT_D of Equation (8.8), consisting of C-tests for a C-testable ILA D, also verifies completely the input-output behavior of D. The fixed-length test set CT_D can therefore efficiently verify the logic design of D during its initial development.

8.4 Design for C-testability

Theorem 8.4 suggests that any bilateral array can be

made C-testable by converting its $X\hat{W}$ and $\hat{X}W$ flow tables to reduced RT tables. Figure 8.5 indicates a proposed flow table modification scheme based on this theorem. Two new columns y_a and y_b are added to the original $X\hat{W}$ flow table F. Next state entries in the new columns are chosen so that every state of F is reachable from every other state. Consequently, the modified $X\hat{W}$ flow table is an RT flow table. The output entries a_i, for $i=0,1,\ldots,n-1$, and a'_j, for $j=0,1,\ldots,p-1$, are also chosen to make both $X\hat{W}$ and $\hat{X}W$ flow tables reduced with respect to $\hat{Y}$. Let Q_a be an input sequence obtained by repeating y_a n_1 times, that is, $Q_a = y_a^{n_1}$. Similarly let $Q_b = y_b^{n_2}$. The a_i and a'_j entries are then designed to make Q_a and Q_b minimum-length distinguishing sequences for the $X\hat{W}$ and $\hat{X}W$ flow tables respectively, with $n_1 = \lceil \log_2 n \rceil$ and $n_2 = \lceil \log_2 p \rceil$. The actual construction of a_i and a'_j entries using shift register sequences was described in the proof of Theorem 7.4 in Section 7.5.

Theorem 8.6: A bilateral array of cells whose $X\hat{W}$ and $\hat{X}W$ flow tables are modified as shown in Figure 8.5, is C-testable.

Proof: As mentioned above, the modified $X\hat{W}$ and $\hat{X}W$ flow tables are reduced, and have Q_a and Q_b as their respective distinguishing sequences. Now we show that using the new columns y_a and y_b, it is always possible to construct a

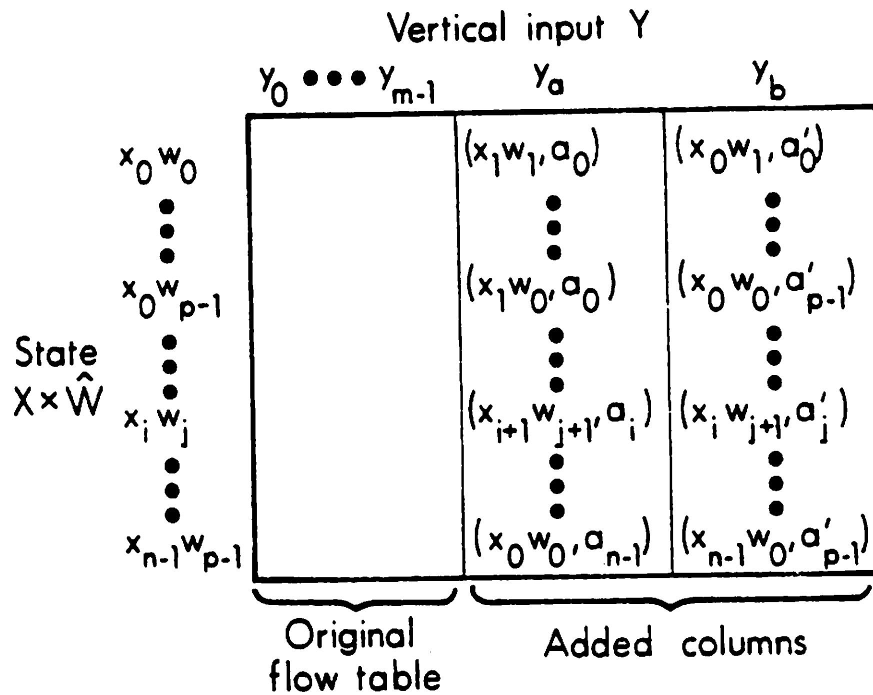


Figure 8.5. Flow table modification to make any bilateral ILA C-testable

transfer sequence R: $x_i w_j \longrightarrow x_e w_f$ that takes a cell from a state $x_i w_j$ to any other state $x_e w_f$. An input sequence $y_a^{m_1}$ takes $x_i w_j$ to a new state $x_e w_{j'}$, where m_1 is such that $i+m_1 \equiv e \pmod{n}$, and j' is given by $j+m_1 \equiv j' \pmod{p}$. A second input sequence $y_b^{m_2}$ can take the intermediate state $x_e w_{j'}$ to the final state $x_e w_f$, where m_2 is such that $j'+m_2 \equiv f \pmod{p}$. Thus the desired transfer sequence R is $y_a^{m_1} y_b^{m_2}$. Hence the modified $X\hat{W}$ and $\hat{X}W$ flow tables are both reduced RT flow tables. It follows from Theorem 8.4 that an array of the modified cells is C-testable. □

A possible hardware realization of the modified cell L′ is shown in Figure 8.6. The added logic modules NL_1 and NL_2 generate the entries in the new columns y_a and y_b as follows. NL_1 implements the $\hat{W}$ entries in columns y_a and y_b and also the a_j''s of column y_b. NL_2 realizes the X entries and the a_i's of column y_a. The multiplexers M_X, M_W and M_Y select the appropriate output signals determined by the Y input. For example, the multiplexer M_W selects the $\hat{W}$ output signal of the original cell L if the Y input is y_0 to y_{m-1}, and it selects the $\hat{W}$ output signal of NL_1 otherwise. Note that the circuit of Figure 8.6 applies to arrays composed of any cell L.

8.5 Sequential Arrays

Arrays of identical sequential cells are examined next. Their interconnection structure is the same as that of

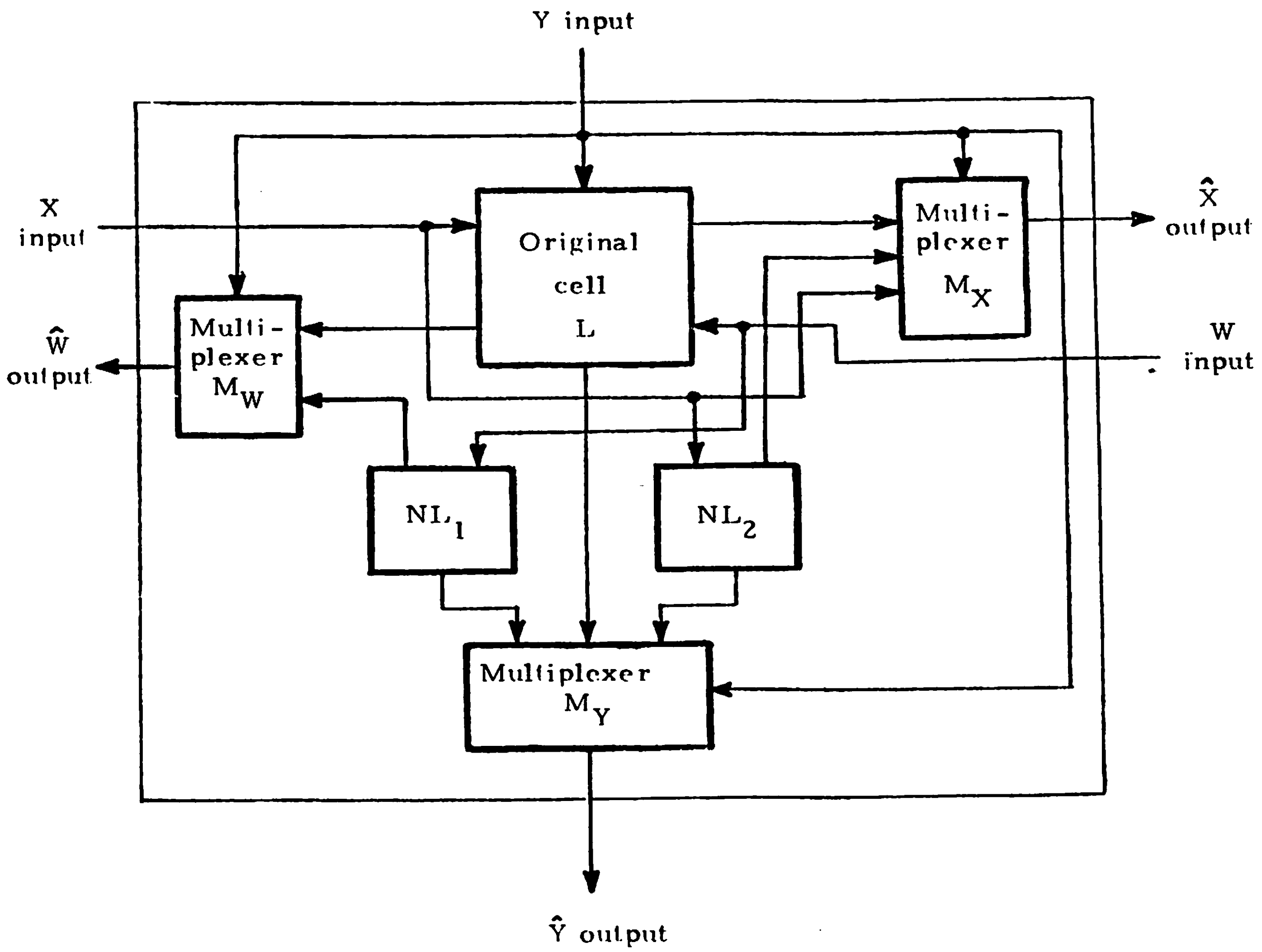


Figure 8.6. A hardware implementation of a bilateral cell modified for C-testability

Figure 8.1; the cells are now synchronous sequential machines, however. The problem of testing sequential ILAs in general is known to be extremely difficult [Breuer 1968, Sung 1976]. The main problems encountered include initialization of the memory elements, and identification of the state of a sequential cell independent of its position in the array. It appears that the analysis of C-testability in a general sequential ILA is even more complex. We now demonstrate that this problem is tractable in the case of a restricted but useful class of sequential ILAs.

Consider a sequential array with the basic cell L shown in Figure 8.7. L is a synchronous sequential machine containing two modules: a combinational module M_1 whose output is the next state $\hat{S}$, and a memory module M_2. The internal state S is directly presented at the $\hat{Y}$ output, thus L is a Moore-type machine [Friedman and Menon 1971]. Many practical ILA circuits such as registers, shift registers and counters used in commercial bit-sliced devices, belong to this class. Also,in the processor array of the C-type cells shown in Figure 4.1, the subarray formed by the shifter module M_L (representing the combinational module M_1 in L of Figure 8.7) and the register M_A or M_T (representing the memory module M_2 in L), can be viewed as a sequential ILA of the type under consideration.

The fault model of Section 2.4 is applied here to the 2-module sequential cell L. As before, we assume that at

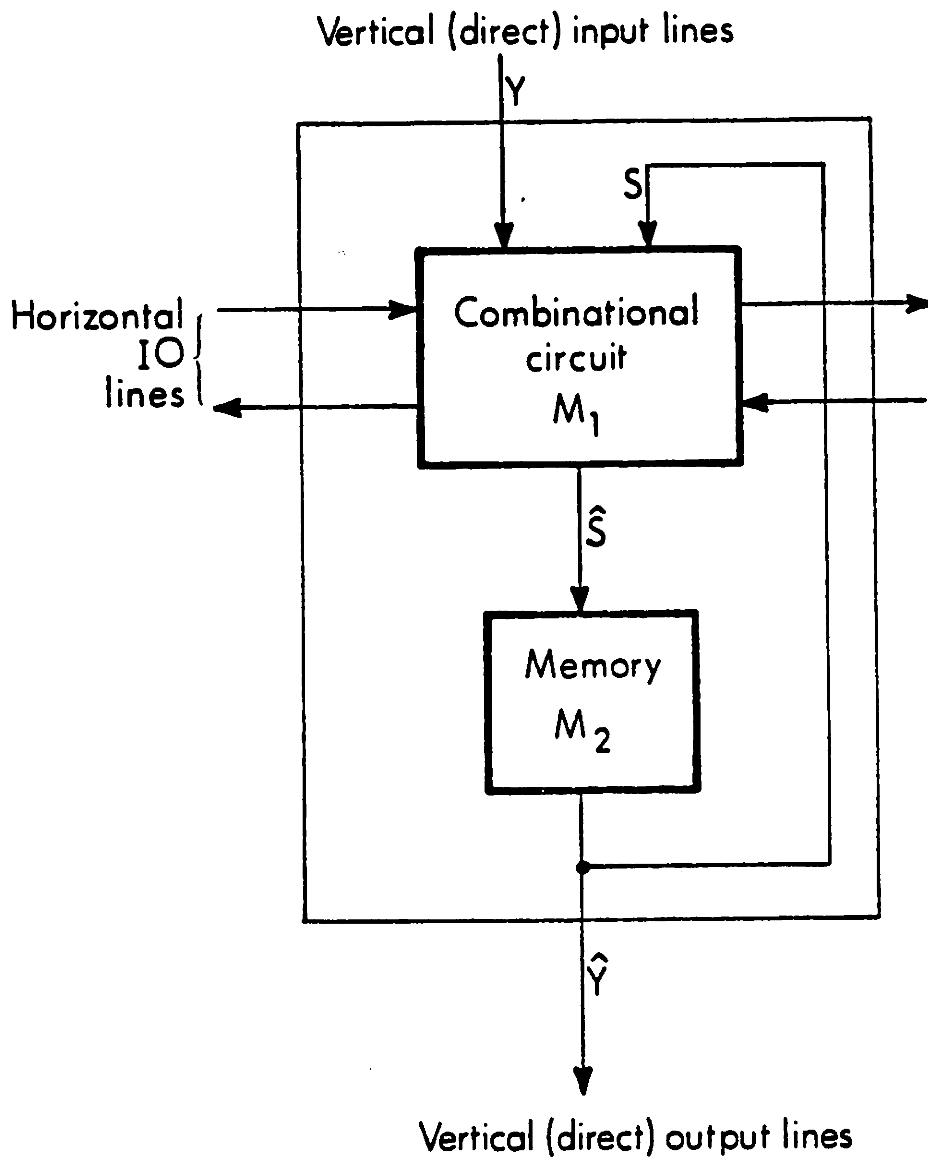


Figure 8.7. A synchronous sequential cell L

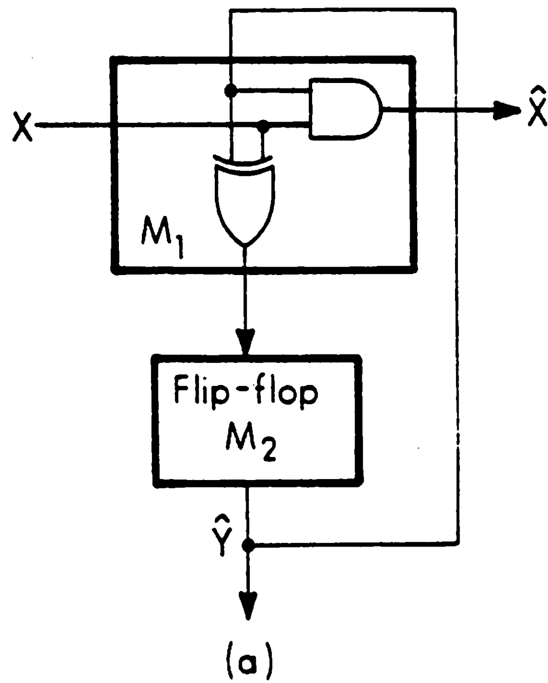
most one module is faulty in an array. Thus to test L, it is necessary and sufficient to verify the truth table of M_1 and the state table of M_2. We also assume that the memory module M_2 of any cell L can be independently initialized to any state $s \in S$ via the Y input lines or other external lines. This is a simplifying assumption commonly made in the theory of sequential machine testing [Breuer and Friedman 1976].

Let P be an array of L-type cells, and let D be the combinational subarray formed from the combinational modules M_1 in P. P can be tested completely by testing its subarray D and the memory modules in every cell of P. The subarray D is tested like any other combinational ILA. For this purpose, the necessary S input signals to M_1 in each cell L_j are obtained by appropriately initializing the state of M_2 in L_j. To test the memory module M_2, we need to verify its state table, in which each next-state entry should equal the corresponding present input $\hat{s} \in \hat{S}$. Most of these entries are automatically verified while testing D. This is because the output of M_2 is directly observable at $\hat{Y}$, and all possible $\hat{S}$ input signals to M_2 must be generated while testing M_1 exhaustively according to the above fault model. Any remaining untested entries of the state table of M_2 can readily be verified by applying appropriate initializing input patterns to P. Thus we have the following result.

Theorem 8.7: An array of L-type sequential cells is C-testable if and only if its combinational subarray is C-testable.

It follows that all results obtained previously for combinational arrays are directly applicable to this special class of sequential arrays.

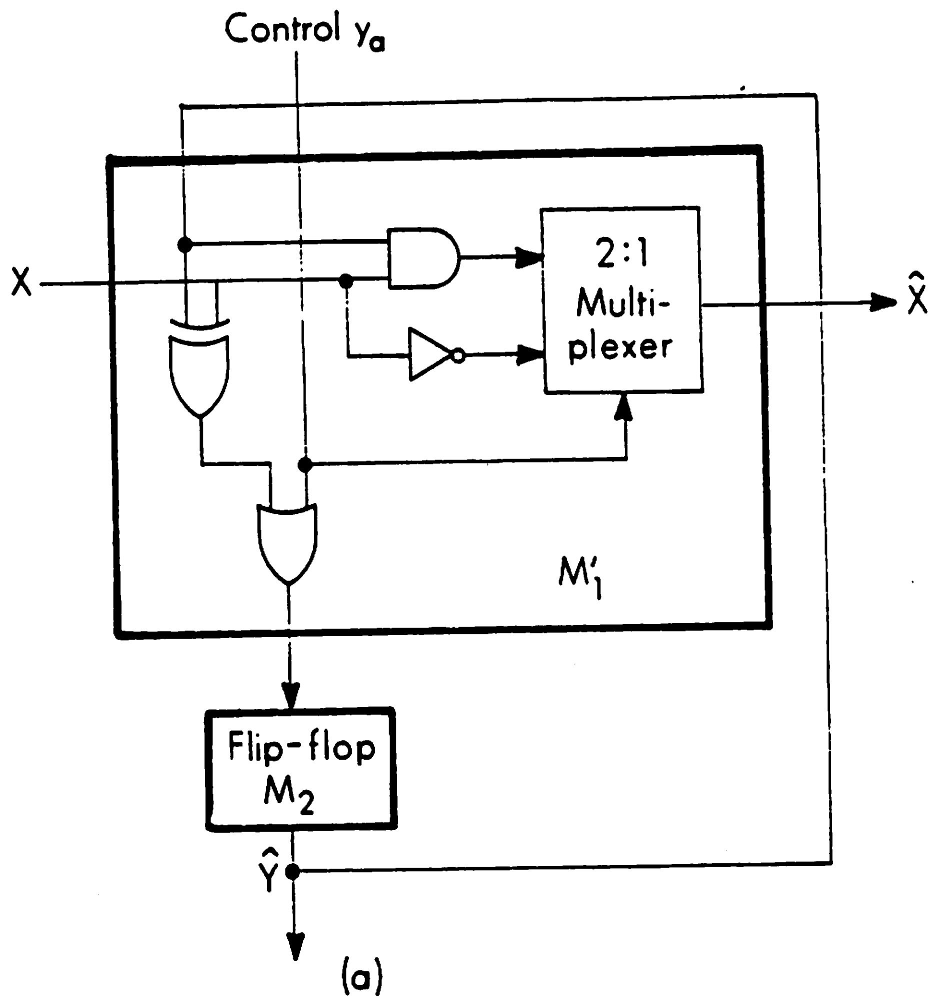
Example 8.2: Consider an N-bit binary counter array composed of N 1-bit counter cells with the structure of Figure 8.8a. The combinational module M_1 of L is a 1-bit incrementer, whose X flow table F appears in Figure 8.8b. F is a reduced but not RT flow table, since it contains the non-repeatable transition which is underlined. Hence the counter array is not C-testable. It can be made C-testable by modifying F to make it an RT flow table. This can be done by adding a new column y_a as shown in Figure 8.9b. Figure 8.9a shows the modifications needed in the counter cell itself in order to realize the flow table of Figure 8.9b. If the new control variable y_a is 1, then the entries in the added column are selected; whereas if y_a is 0, the entries in the original table are effective. Any sequential cell with the structure of Figure 8.7 can be similarly redesigned to make it C-testable.



State X	Vertical input 0	Vertical input 1
0	(0,0)	(0,1)
1	($\underline{0}$,1)	(1,0)

(b)

Figure 8.8. (a) A 1-bit counter cell. (b) The X flow table of the incrementer module M_1



Vertical input

State X	0	1	y_a
0	(0,0)	(0,1)	(1,1)
1	(0,1)	(1,0)	(0,1)

Original table (columns 0, 1) · Added column (y_a)

(b)

Figure 8.9. (a) Modified 1-bit counter cell
(b) The modified X flow table
for the incrementer module

CHAPTER 9

I-TESTABLE ARRAYS

Until now our study of ILAs has been primarily concerned with input test pattern generation. We have not discussed the issue of output test response verification. The output response of the circuit being tested is usually checked by comparing it with the fault-free response stored or generated elsewhere. This approach requires a relatively large memory M to store the complete input test patterns as well as the expected response patterns. M is typically part of the (external) test equipment. Additional hardware is also needed to compare the expected and observed responses. The amount of response data that must be stored can be reduced to some extent by using such data compression schemes as ones counting and transition counting [Breuer and Friedman 1976]. In this chapter, a new property of ILAs called I-testability is introduced which requires very little test equipment and greatly simplifies response checking. I-testability eliminates the need to construct or store response data by allowing identical responses to be generated from every cell of the array.

Definition 9.1: An input test pattern for an ILA D is called an *I-test* if the expected (fault-free) responses appearing at the vertical $\hat{Y}$ output lines of every cell of D are identical. D is called *I-testable* if all faults in D can be detected by a test set composed solely of I-tests.

The responses of D to I-tests can easily be verified by comparing the responses appearing on the vertical output lines of the individual cells. This comparison can be done by means of an equality checking circuit as shown in Figure 9.1. Thus I-testing, in which I-tests and equality checkers are used, eliminates the need to store explicitly the expected responses to T_D. The checking circuit of Figure 9.1 can itself be designed as an ILA and can easily be integrated into the original ILA D; this is discussed later in Section 9.2. Practical applications of I-testing in designing self-testing bit-sliced systems are discussed in the next chapter.

We first analyze I-testability in a basic class of ILAs, namely unilateral ILAs of identical combinational cells; see Figure 9.2. Practical array circuits such as ripple-carry adders, and the incrememters used in the processors and microprogram sequencers discussed in Section 2.2, belong to this class. Later the property of CI-testability in such unilateral arrays is also discussed. A *CI-testable ILA* is both C-testable and I-testable with

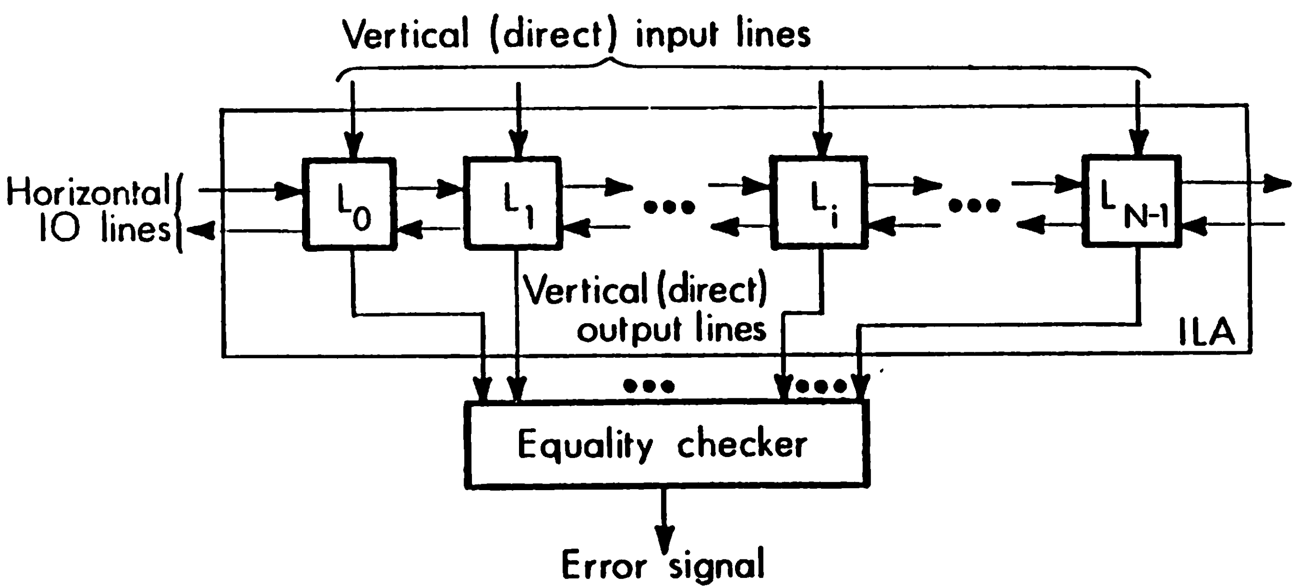


Figure 9.1. A scheme for response verification in a general I-testable ILA

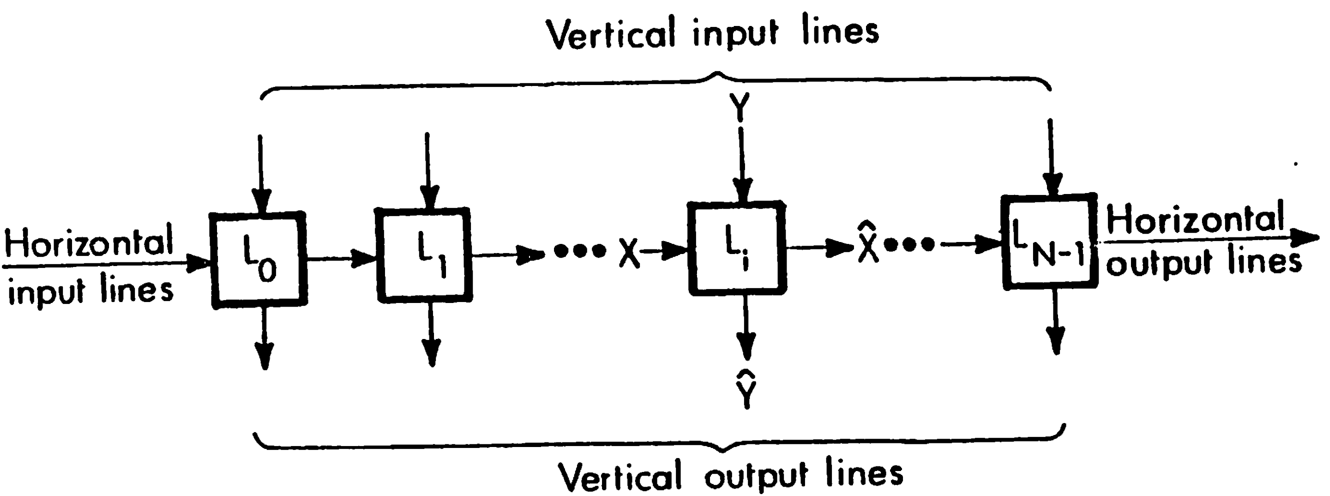


Figure 9.2. A combinational unilateral ILA

respect to some test set T_D, i.e., every test in T_D is a C-test as well as an I-test. The basic concepts and results presented here for combinational unilateral ILAs can readily be extended to more general bilateral sequential ILAs.

9.1 I-testability

Let L be a basic cell with the input-output signal sets X, $\hat{X}$, Y and $\hat{Y}$ shown in Figure 9.2, where $|X| = |\hat{X}| = n$, $|Y| = m$ and $|\hat{Y}| = q$. Let F be the $n \times m$ flow table of L, in which, as usual, rows represent X and columns represent Y. A typical entry $(\hat{x}_a, \hat{y}_b)$ in row i and column j of F is a pair consisting of the next state $\hat{x}(x_i, y_j) = \hat{x}_a$ and the vertical output signal $\hat{y}(x_i, y_j) = \hat{y}_b$. It is useful to decompose F into $|\hat{Y}|$ disjoint $n \times m$ flow tables $F_0, F_1, \ldots, F_{q-1}$, called *partitioned flow tables*, such that F_k contains all the entries (or transitions) of F for which $\hat{y}(x_i, y_j) = \hat{y}_k$. The null symbol $\emptyset$ is entered in F_k to denote each entry of F for which $\hat{y}(x_i, y_j) \neq \hat{y}_k$.

Example 9.1: The flow table F of a 1-bit full adder cell, and its two partitioned flow tables F_0 and F_1 are shown in Figure 9.3. X is the carry input, Y is the 2-bit data input, $\hat{X}$ is the carry output, and $\hat{Y}$ is the sum output. □

Vertical input Y

State X	00	01	10	11
0	(0,0)	(0,1)	(0,1)	(1,0)
1	(0,1)	(1,0)	(1,0)	(1,1)

(a)

Vertical input Y

	00	01	10	11
0	(0,0)	ϕ	ϕ	(1,0)
1	ϕ	(1,0)	(1,0)	ϕ

(b)

Vertical input Y

	00	01	10	11
0	ϕ	(0,1)	(0,1)	ϕ
1	(0,1)	ϕ	ϕ	(1,1)

(c)

Figure 9.3. (a) Flow table F of a 1-bit full adder cell (b) Partitioned flow table F_0 for $\hat{y} = 0$ (c) Partitioned flow table F_1 for $\hat{y} = 1$

Clearly, to test the ILA of Figure 9.2 completely with the basic cell L treated as a single module, it is necessary and sufficient to verify all mn entries of the flow table F for every cell of the ILA. In other words, all non-ø entries of each partitioned flow table F_k must be verified for every cell. While verifying an entry of F_k in some cell L_j using I-tests, every other cell in the ILA should be confined to non-ø entries appearing in F_k; otherwise the expected vertical output signals from every cell will not be identical.

We are interested in obtaining necessary and sufficient conditions for an ILA to be I-testable. Our approach is to modify the following well-known testability criteria of Kautz.

<u>Theorem 9.1 [Kautz 1967]</u>: An ILA is testable if and only if

(1) every state appears as a next state entry in the cell flow table F at least once, and

(2) no two rows of F are identical.

For the ILA to be I-testable, the conditions of the above theorem must be modified to apply individually to every partitioned flow table F_k, for $k = 0,1,2,\ldots,q-1$. The first condition of Theorem 9.1 ensures that every X input

signal is applicable to every cell of the ILA. In a partitioned flow table F_k it is not necessary to have every state appear as a next state entry because not all X input signals may produce the $\hat{Y}$ output signal $\hat{y}_k$. If, say, $x_a \in X$ does not produce the output signal $\hat{y}_k$, then the row in F_k corresponding to x_a contains ϕ entries only, and hence can be deleted from F_k. Thus Condition (1) of Theorem 9.1 is modified for I-testability as follows. Every undeleted state should appear at least once as a next state entry in F_k for all k.

The second condition of Theorem 9.1 states that any faulty $\hat{X}$ output signal of a cell must eventually be propagated to some observable output line of the ILA. This propagation condition may be modified for I-testability by introducing the following distinguishability criterion.

Definition 9.2: A (fault-free) state x_a is *I-distinguishable* from another (faulty) state x_b if and only if there exists a vertical input $y \in Y$ in flow table F_k such that the entries $e_a = (\hat{x}(x_a,y),\hat{y}(x_a,y))$ and $e_b = (\hat{x}(x_b,y),\hat{y}(x_b,y))$ satisfy either of the following conditions:

(1) $e_a \neq \phi$ and $e_b = \phi$

(2) $e_a \neq \phi$, $e_b \neq \phi$ and $\hat{x}(x_a,y) \neq \hat{x}(x_b,y)$.

Example 9.1 (cont'd): In the full adder flow table F_0 of Figure 9.3b, state $x_0 = 0$ is I-distinguishable from $x_1 = 1$

due to columns $y_0 = 00$ and $y_3 = 11$, and x_1 is I-distinguishable from x_0 because of columns $y_1 = 01$ and $y_2 = 10$. □

Note that cases exist where x_a is I-distinguishable from x_b, but x_b is not I-distinguishable from x_a. This is demonstrated in the following example.

Example 9.2: Consider the portion of an $n \times 4$ partitioned flow table F_k shown in Figure 9.4. The state x_a is I-distinguishable from the state x_b due to column y_0, while x_b is not I-distinguishable from x_a. Also x_c is I-distinguishable from both x_a and x_b in column y_1. □

Let x_a be I-distinguishable from x_b due to input y in F_k. Suppose that x_a is the expected $\hat{X}$ output signal from a cell L_j in the ILA during the verification of some entry of F_k for L_j. If we apply y to the Y input line of the cell L_{j+1} to the right of L_j, then a fault in L_j that changes x_a to x_b will be propagated either to the $\hat{Y}$ output line of L_{j+1} if $\hat{y}(x_b,y) \neq \hat{y}_k$, or to the $\hat{X}$ output line of L_{j+1} if $\hat{x}(x_a,y) \neq \hat{x}(x_b,y)$. Thus the above distinguishability property helps in propagating fault signals on the $\hat{X}$ output lines of a cell to an observable output of the ILA. I-distinguishability with respect to a fixed $\hat{y}$ value is also necessary to ensure that the expected vertical output signal in the propagating cell L_{j+1} is identical to that in the test cell L_j. Hence for I-testability, every state

Input Y

State X	y_0	y_1	y_2	y_3
	.	. .	.	.
x_a	(x_d, y_k)	(x_b, y_k)	(x_a, y_k)	$\emptyset$
x_b	$\emptyset$	(x_b, y_k)	$\emptyset$	$\emptyset$
x_c	(x_d, y_k)	(x_c, y_k)	$\emptyset$	$\emptyset$
	.	. .	.	.

Figure 9.4. Flow table F_k for Example 9.2

x_a that appears as a next state entry in F_k, for all k, should be I-distinguishable from every other (faulty) state.

From the foregoing discussion we obtain the following general characterization of I-testable ILAs.

Theorem 9.2: An ILA is I-testable if and only if in each partitioned cell flow table F_k,

(1) every (undeleted) state appears at least once as a next state entry, and

(2) every state that appears at least once as a next state entry is I-distinguishable from every other state.

We next establish some bounds on the number of I-tests $|IT_D|$ for an I-testable ILA D, where IT_D is a complete set of I-tests for D. Clearly, we need at least one test pattern for each of the mn transitions in the $n \times m$ cell flow table F. Hence the lower bound on $|IT_D|$ is mn. An upper bound on $|IT_D|$ is obtained as follows.

Let n_k be the number of non-$\emptyset$ entries in the partitioned flow table F_k. Clearly, $\sum_{k=0}^{q-1} n_k = mn$. For each non-$\emptyset$ entry in F_k being verified in some cell L_i, there are n-1 possible faulty $\hat{X}$ output signals. To propagate each of these n-1 faulty $\hat{X}$ signals to the observable outputs of D, a different combination of Y input signals may have

to be applied to the cells to the right of L_i. Thus the number of I-tests required to verify a non-$\emptyset$ entry of a partitioned flow table F_k in any internal cell L_i is at most n-1. In the case of the rightmost cell L_{N-1}, just one I-test is needed, since its $\hat{X}$ output line is also a primary observable output of D.

The number of I-tests required to verify all n_k entries of F_k in any cell L_i of D is therefore at most $(n-1)n_k$. To verify all entries of F for L_i requires at most

$$\sum_{k=0}^{q-1} (n-1)n_k = (n-1)\sum_{k=0}^{q-1} n_k = (n-1)mn$$

I-tests. Hence we have the following inequality

$$mn \leqq |IT_D| \leqq mn(n-1)(N-1)+mn \qquad (9.1)$$

where the last term mn in the upper bound is due to the rightmost cell L_{N-1}.

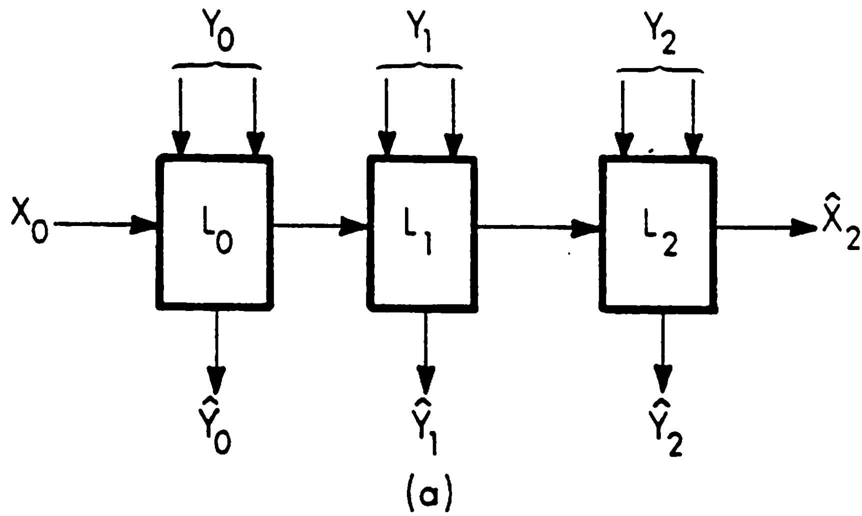
Example 9.1 (cont'd): Consider again the flow tables of an N-bit ripple-carry adder ILA shown in Figure 9.3. It is evident that the two partitioned flow tables F_0 and F_1 both satisfy the conditions of Theorem 9.2. Hence the adder array is I-testable. I-tests for verifying the entries in F_0 and F_1 can easily be constructed. For example, consider the entry (0,0) in row $x_0 = 0$ and column $y_0 = 00$ of

F_0. An I-test that checks this entry in all cells of the ILA is x_0, y_0^N, which applies x_0 to the horizontal input of the leftmost cell, and y_0 to the vertical inputs of all N cells. I-tests for all entries of the cell flow tables F_0 and F_1 are given in Table 9.1. The six tests t_1, t_2, t_3, t_6, t_7 and t_8 verify their respective transitions in every cell at the same time. The remaining tests t_4 and t_5 can verify transitions τ_4 and τ_5, respectively, in only one cell, namely L_i, at a time. This is because the two transitions in question are not repeatable in their respective partitioned flow tables. Hence they cannot be applied to more than one cell at a time. This is discussed further in Section 9.3. Therefore the number of I-tests required to verify τ_4 and τ_5 in all N cells is 2N. The minimum number of I-tests for an N-bit adder array is therefore 6+2N. Note that only eight tests suffice to form a complete set of C-tests; see Example 7.3. Figure 9.5 shows a 3-cell ripple-carry adder and the corresponding minimal set of 12 I-tests.

Example 9.3: Consider a 1-bit half-adder or incrementer cell with the flow tables shown in Figure 9.6. The partitioned flow table F_0 satisfies both conditions of Theorem 9.2, whereas F_1 satisfies only Condition (2). Hence an incrementer ILA is not I-testable.

Table 9.1. I-tests for an N-bit ripple-carry adder ILA

No. j	Transition τ_j	I-test t_j for τ_j
1	$0 \xrightarrow{00} 0$	$0,(00)^N$
2	$0 \xrightarrow{01} 0$	$0,(01)^N$
3	$0 \xrightarrow{10} 0$	$0,(10)^N$
4	$0 \xrightarrow{11} 1$	$0,(00)^i 11(01)^{N-i-1}$
5	$1 \xrightarrow{00} 0$	$1,(11)^i 00(01)^{N-i-1}$
6	$1 \xrightarrow{01} 1$	$1,(01)^N$
7	$1 \xrightarrow{10} 1$	$1,(10)^N$
8	$1 \xrightarrow{11} 1$	$1,(11)^N$



(a)

Cell transition $\tau : x_i \xrightarrow{y_k} \hat{x}_j$	I-test(s) for τ $X_0, Y_0Y_1Y_2$	Cells tested
$0 \xrightarrow{00} 0$	0,000000	L_0, L_1, L_2
$0 \xrightarrow{01} 0$	0,010101	L_0, L_1, L_2
$0 \xrightarrow{10} 0$	0,101010	L_0, L_1, L_2
$0 \xrightarrow{11} 1$	0,110101 0,001101 0,000011	L_0 L_1 L_2
$1 \xrightarrow{00} 0$	1,000101 1,110001 1,111100	L_0 L_1 L_2
$1 \xrightarrow{01} 1$	1,010101	L_0, L_1, L_2
$1 \xrightarrow{10} 1$	1,101010	L_0, L_1, L_2
$1 \xrightarrow{11} 1$	1,111111	L_0, L_1, L_2

(b)

Figure 9.5. (a) A 3-bit ripple-carry adder ILA
(b) A minimal set of I-tests for this adder

State X \ Vertical input Y	0	1
0	(0,0)	(0,1)
1	(0,1)	(1,0)

F

	0	1
0	(0,0)	ϕ
1	ϕ	(1,0)

F_0

	0	1
0	ϕ	(0,1)
1	(0,1)	ϕ

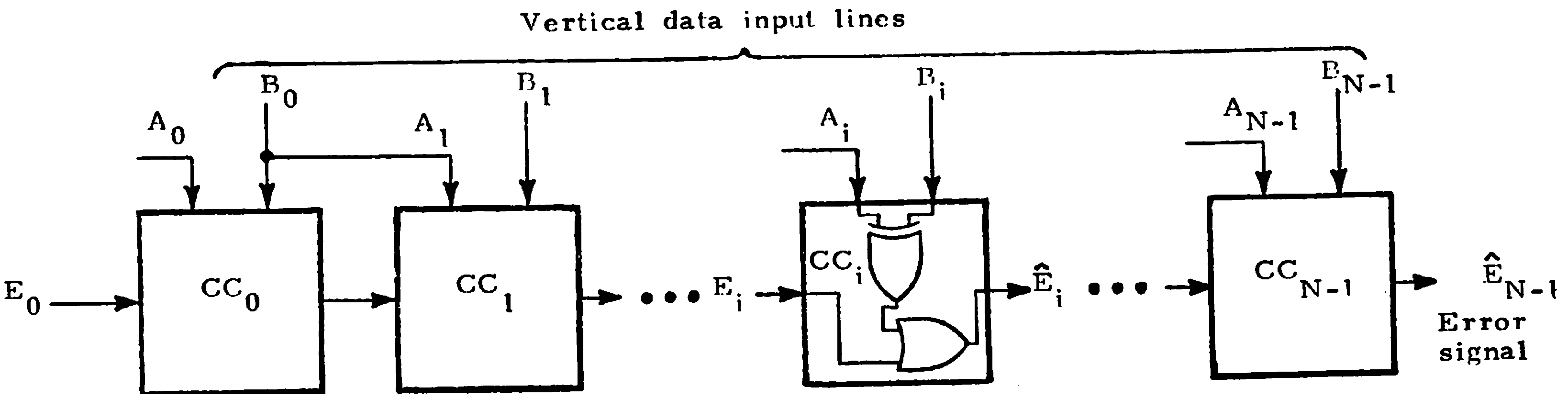
F_1

Figure 9.6. Flow tables F, F_0 and F_1 of 1-bit incrementer cell

9.2 Equality Checkers

We now examine the design and testing of equality checkers that can be used to verify the output responses of an I-testable ILA D to its I-tests; see Figure 9.1. The function of an N-bit equality checker here is to determine whether its N vertical input signals are all identical or not. If the input signals are all identical, then the output error signal is 0, otherwise it is 1 indicating the detection of an error.

Figure 9.7a depicts an N-bit equality checker EC implemented as an ILA with the cell flow table F_{CC} given in Figure 9.7b. The checker cell CC_i works as follows. Its horizontal output (error) signal $\hat{E}_i$ is 1 if the horizontal input signal E_i is 1; otherwise $\hat{E}_i = 1$ if and only if the two vertical input signals A_i and B_i are identical. Therefore if $A_i \neq B_i$ for any i, then a horizontal signal of value 1 is propagated through EC to the final output $\hat{E}_{N-1}$. With the input line A_i of CC_i connected to B_{i-1} of CC_{i-1}, as shown in Figure 9.7a, the primary output $\hat{E}_{N-1}$ of EC is 0 if and only if $B_0 = B_1 = B_2 = \cdots = B_{N-1}$ and $E_0 = 0$. Thus EC produces an error signal if for some i, $B_i \neq B_{i+1}$. This type of equality checker can be used to verify the expected identical $\hat{Y}$ output responses of any I-testable ILA D as indicated in Figure 9.1. Here we have assumed that the number of $\hat{Y}$ output lines from each cell is one. When a



(a)

State E_i	Input A_iB_i: 00	01	10	11
0	0	1	1	0
1	1	1	1	1

(b)

Figure 9.7. (a) An ILA implementation of an equality checker, and (b) its cell flow table F_{cc}

complete set of I-tests T_D is applied to the primary inputs of D, the $\hat{E}_{N-1}$ output of EC indicates whether the $\hat{Y}$ output responses of D to T_D are correct or not. The checker ILA EC can be easily integrated into D by incorporating CC_i into each cell L_i of D. This method of using ECs to verify the responses of an I-testable ILA D can be extended directly to the general case, where D has $k > 1$ $\hat{Y}$ output lines per cell. In that case k identical N-bit ECs, $EC_0:EC_{k-1}$, one for each $\hat{Y}$ output line are used. The k error output signals of $EC_0:EC_{k-1}$ can then be combined by an OR gate to yield a single composite error signal.

The checker EC of Figure 9.7a can itself be tested like any other combinational unilateral ILA without direct outputs. The cell flow table F_{cc} of Figure 9.7b is a non-RT flow table since the underlined transitions are not repeatable. Thus by Theorem 5.1, EC is not C-testable. Note that the question of I-testability does not arise, since EC has no direct outputs. The checker ILA is testable since F_{cc} satisfies the two testability conditions of Theorem 9.1. A set of test patterns for EC are given in Table 9.2. The four test patterns t_1, t_2, t_7 and t_8 of this table verify their respective transitions simultaneously in every cell of the array. This is because the transitions tested by these patterns belong to the permutation columns 00 and 11 of F_{cc}; see Theorem 6.2. The remaining four test patterns apply the entries in the two

Table 9.2. Representative test patterns for an N-bit equality checker ILA

j	Transition τ_j $E_i \xrightarrow{A_iB_i} \hat{E}_i$	Test t_j for τ_j
1	$0 \xrightarrow{00} 0$	$0,(00)^N$
2	$1 \xrightarrow{00} 1$	$1,(00)^N$
3	$0 \xrightarrow{01} 1$	$0,(00)^i01(11)^{N-i-1}$
4	$1 \xrightarrow{01} 1$	$1,(00)^i01(11)^{N-i-1}$
5	$0 \xrightarrow{10} 1$	$0,(11)^i10(00)^{N-i-1}$
6	$1 \xrightarrow{10} 1$	$1,(10)^i10(00)^{N-i-1}$
7	$0 \xrightarrow{11} 0$	$0,(11)^N$
8	$1 \xrightarrow{11} 1$	$1,(11)^N$

constant columns 01 and 10 of F_{CC}. Lemma 6.1 of Section 6.1 indicates that an entry in a constant column is verifiable in only one cell at a time. Therefore the test patterns t_3, t_4, t_5 and t_6 test only one cell CC_i, and so must be repeated for $i = 0,1,\ldots,N-1$. Hence the total number of tests required by EC is 4+4N. This is a relatively small number of tests for a non-C-testable array.

If N is very large, it may be desirable to reduce the number of tests for EC by making it C-testable using the modification scheme of Section 6.4 (see Figures 6.5 and 6.6). The structure of modified checker cell table F'_{CC} is indicated in Figure 6.5b, it is formed by adding two new rows and two new columns to the original flow table F_{CC}. The corresponding modified checker cell CC′ requires two additional horizontal and vertical input lines, and associated logic circuits. Hence there exists a trade-off between the additional logic and the number of tests. The number of C-tests for the modified C-testable checker ILA of arbitrary size is at most 251 as given in Table 6.1. Therefore, if the number of tests is of primary concern, the modified checker ILA can be used for $N \geqq 62$.

9.3 CI-testability

In this section the combined property of C- and I-testability, namely CI-testability is analyzed for combinational unilateral (Class 5) ILAs.

Consider the array D depicted in Figure 9.2. D is CI-testable if there exists a complete set of I-tests IT_D for D whose size is independent of the size N of D. A CI-testable ILA must, of course, satisfy all the conditions for C-testability and I-testability obtained earlier. During I-testing, the state transitions of every cell in D are confined to one partitioned flow table at a time. Also for the ILA to be C-testable the flow table being used must be an RT flow table as required by Theorem 5.1 of Section 5.4. Hence we have the following necessary condition for CI-testability.

Theorem 9.3: An ILA is CI-testable only if the partitioned flow table F_k, for $k = 0,1,\ldots,q-1$, is an RT flow table.

Example 9.1 (cont'd): The partitioned flow tables F_0 and F_1 of a full adder cell appearing in Figure 9.3 are not RT flow tables. Hence a ripple-carry adder ILA is not CI-testable. Nevertheless it is C-testable as proved in Example 7.3 of Section 7.3. It is also I-testable as shown earlier in this example. The problem is that there is no test set for the adder ILA that serves as a set of C-tests and I-tests simultaneously. □

We now modify slightly Definitions 7.1, 7.2 and 7.3 introduced for C-testable ILAs in Section 7.1, so that the

requirements of I-testability are also satisfied. The following three definitions refer to a partitioned flow table, F_k, but not to the complete flow table F of an array cell.

<u>Definition 9.3</u>: An *I-distinguishing sequence* $IDS(x_a, x_b)$ for states x_a and x_b of a partitioned flow table F_k is a Y input sequence which when applied to F_k produces a constant $\hat{Y}$ output sequence $\hat{y}_k^* = \hat{y}_k \hat{y}_k \hat{y}_k \ldots$, when x_a is the initial state, and some different $\hat{Y}$ output sequence when x_b is the initial state.

From Definitions 7.1 and 9.3, it follows that $IDS(x_a, x_b)$ always differs from $IDS(x_b, x_a)$, whereas the ordinary distinguishing sequences $DS(x_a, x_b)$ and $DS(x_b, x_a)$ are always the same.

<u>Definition 9.4</u>: A *set of I-distinguishing sequences* $SIDS(x_a)$ for a state x_a of F_k is a set of Y input sequences which, when applied to F_k with x_a as the initial state, produces a set of constant $\hat{Y}$ output sequences $\{\hat{y}_k^*, \hat{y}_k^*, \ldots, \hat{y}_k^*\}$ different from the sets of $\hat{Y}$ output sequences produced with any other state as the initial state. In general, we can write

$$SIDS(x_a) = \{IDS(x_a, x_b) : x_b \in X \text{ and } x_b \neq x_a\} \quad (9.2)$$

Definition 9.5: A partitioned flow table F_k is *I-reduced* with respect to $\hat{Y}$ if there exists an I-distinguishing sequence $IDS(x_a, x_b)$ for every pair of states (x_a, x_b), where x_a is any state that appears at least once as a next state entry in F_k, and x_b is any other state.

Example 9.1 (cont'd): Once more consider the full adder cell of Figure 9.3. Let the two states x_a and x_b in Definition 9.3 be $x_0 = 0$ and $x_1 = 1$. Either of the input patterns $y_0 = 00$ or $y_3 = 11$ can serve as $IDS(x_0, x_1)$ in F_0, while either $y_1 = 01$ or $y_2 = 10$ can be $IDS(x_1, x_0)$. Hence $SIDS(x_0) = \{IDS(x_0, x_1)\} = \{y_0\}$ or $\{y_3\}$, and $SIDS(x_1) = \{IDS(x_1, x_0)\} = \{y_1\}$ or $\{y_2\}$. Similarly in F_1, $SIDS(x_0) = \{y_1\}$ or $\{y_2\}$, and $SIDS(x_1) = \{y_0\}$ or $\{y_3\}$. Thus F_0 and F_1 are both I-reduced partitioned flow tables. □

Using the foregoing I-distinguishing sequences one can construct a class of fixed-length I-tests, called CI-tests, which are analogous to the C-tests of Definition 7.4.

Definition 9.6: Consider a transition $\tau: x_a \xrightarrow{y_j} x_c$ in a partitioned flow table F_k. A *CI-test* $CI_i(\tau)$ associated with this transition is a periodic input sequence of the form

$$CI_i(\tau) = x_a, (y_j D_i R_i)^* \qquad (9.3)$$

where D_i is a member of $SIDS(x_c) = \{D_1, D_2, \ldots, D_i, \ldots, D_r\}$ and R_i is a transfer sequence that drives F_k back to the state x_a.

The CI-test $CI_i(\tau)$ verifies τ with respect to D_i, in cells $L_0, L_{k_i}, L_{2k_i}, \ldots$, where k_i is its periodicity. Now consider the set of periodic input patterns

$$CIT(\tau) = \{s^u(x_a, (y_j D_i R_i)^*): 0 \leq u \leq k_i - 1 \text{ and } D_i \in SIDS(x_c)\} \tag{9.4}$$

where s^u is the shift operator introduced in Definition 6.1 of Section 6.1. Clearly, when the above input patterns of $CIT(\tau)$ are applied to the primary inputs of an ILA D, the transition τ is verified in every cell of the array. At the same time, the expected $\hat{Y}$ output responses to $CIT(\tau)$ from every cell are identical. Therefore a complete CI-test set CIT_D for D is given by

$$CIT_D = \bigcup_i CIT(\tau_i) \tag{9.5}$$

From Definition 9.5 and the properties of RT flow tables, it follows that $CIT(\tau)$ exists for any transition τ of F_k, if F_k is an I-reduced RT flow table. Hence the complete CI-test set CIT_D of Equation (9.5) also exists.

Example 9.1 (cont'd): Although an N-bit ripple-carry adder is not CI-testable some of its cell transitions can

be tested using CI-tests. For example, consider the transition τ_1: $1 \xrightarrow{01} 1$ of the partitioned flow table F_0 given in Figure 9.3b. A corresponding CI-test is $CI_1(\tau_1) = 1,(01)^*$ of period one. Hence a complete CI-test set for τ_1 is $CIT(\tau_1) = \{CI_1(\tau_1)\} = \{1,(01)^*\}$. In a similar way, CI-tests for five other repeatable transitions of F_0 and F_1 can be constructed. Obviously the non-repeatable transitions $0 \xrightarrow{11} 1$ in F_0 and $1 \xrightarrow{00} 0$ in F_1 do not possess CI-tests.

Theorem 9.4: A unilateral ILA with I-reduced partitioned cell flow tables, is CI-testable if and only if the partitioned flow tables are all RT flow tables.

Proof: The proof is similar to that of Theorem 7.2. The necessity of the above condition follows from Theorem 9.3. Suppose that the partitioned flow tables are all I-reduced RT flow tables. Then it is possible to construct CI-tests for every transition in these partitioned flow tables using Definition 9.6. A complete CI-test set CIT_D as defined by Equation (9.5) exists; hence the ILA is CI-testable. □

Bounds on the size of the foregoing CI-test set CIT_D can be determined in the following manner. The length of an I-distinguishing sequence $IDS(x_a,x_b)$ of Definition 9.3, like that of a conventional distinguishing sequence is at

most n-1. Again the length of the transfer sequence R_i of Equation (9.3) is no more than n-1. Hence the periodicity k_i of $CI_i(\tau)$ is bounded as follows.

$$1 \leq k_i \leq 2n-1 \qquad (9.6)$$

From Equation (9.2) it is evident that $|SIDS(x)| \leq n-1$, hence we obtain

$$mn \leq |CIT_D| \leq mn(n-1)(2n-1) \qquad (9.7)$$

Suppose that D is a CI-testable ILA with a complete CI-test set CIT_D of Equation (9.5). Then according to Theorem 7.3 of Section 7.4, CIT_D also completely verifies the truth-table of D. The output response of D to CIT_D is easily verified by means of an equality checker as shown in Figure 9.1.

9.4 Design for CI-testability

Let F be the $n \times m$ flow table of cell L in an ILA that is not CI-testable. It is possible to change F to F′ by adding new columns so that the modified partitioned flow tables $\{F'_k\}$ are I-reduced RT flow tables. By Theorem 9.4, an array of the modified cells is CI-testable. To change any given partitioned flow table F_k into an RT flow table, it is sufficient to add n new transitions to F_k so that all its n states are linked by a single loop of state

transitions. Hence all q partitioned flow tables can be made into RT flow tables by adding at most nq new transitions, i.e., q new columns. Note that in only the worst case is it necessary to add q new columns.

In the proposed design scheme q new columns, one for each partitioned flow table F_k, are introduced as in Figure 9.8. The next-state entries in the new columns are chosen so that all n states in every F_k are connected together in a single loop. Thus the modified flow tables $\{F'_k\}$ are all strongly connected and so they are RT flow tables. Each modified flow table F'_k has the following I-distinguishing sequences:

$$IDS(x_a,x_b) = y_{a_k}^{n-b} \quad \text{for all } a,b \text{ such that } b > a \text{ and } a \neq n-1,$$

$$IDS(x_a,x_b) = y_{a_k}^{n-1-a}\, y_{a_{k-1}} \quad \text{for all } a,b \text{ such that } b < a \text{ and } a \neq n-1,$$

$$IDS(x_{n-1},x_b) = y_{a_{k-1}} \quad \text{for all } b.$$

From the above equations it is evident that $IDS(x_a,x_b)$ exists for every pair of states (x_a,x_b) of F'_k, for all k. Thus by Definition 9.5, each F'_k is I-reduced. Using Theorem 9.4 we immediately obtain the following result.

<u>Theorem 9.5</u>: An ILA of cells with the modified flow table F′ of Figure 9.8 is CI-testable.

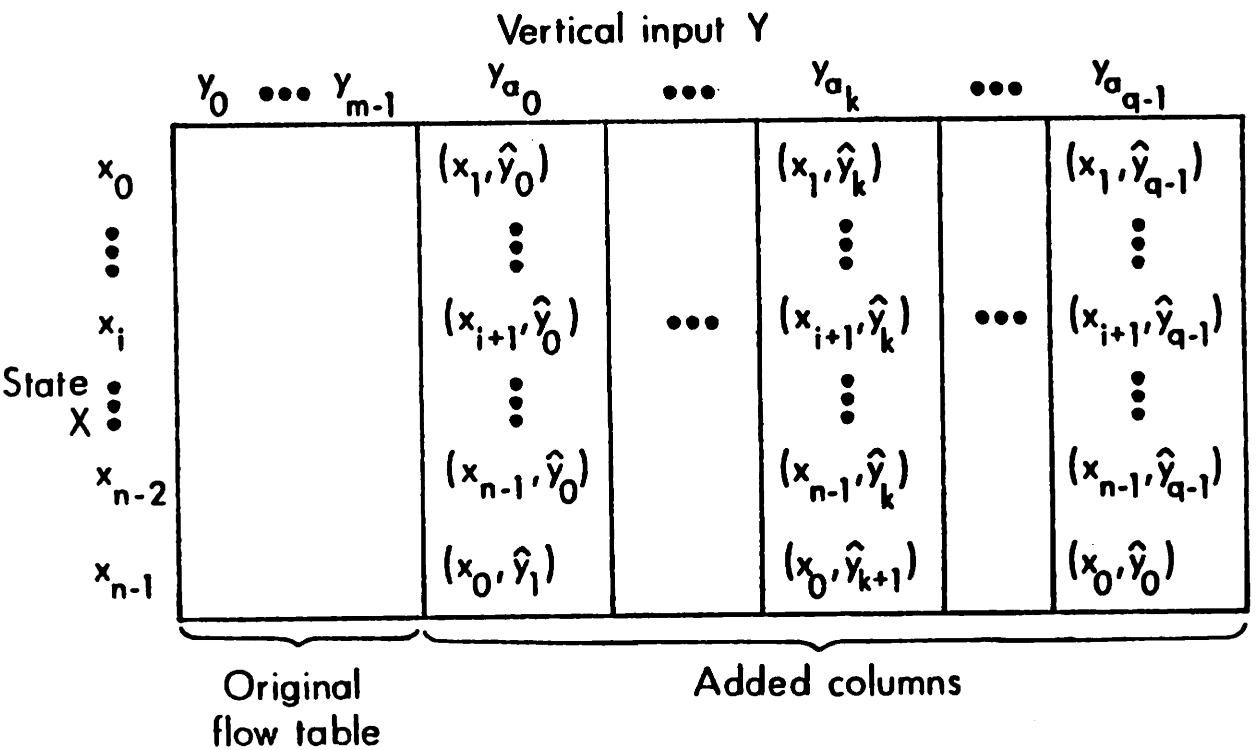


Figure 9.8. Flow table modification to make a unilateral ILA CI-testable

Example 9.1 (cont'd): It was shown already that the flow tables F_0 and F_1 of a 1-bit full adder cell are I-reduced non-RT flow tables. We now add a new column y_a to the original flow table F as depicted in Figure 9.9a. It is clear that the modified partitioned flow tables F_0' and F_1' shown in Figures 9.9b and 9.9c are RT flow tables. Hence an array of the modified full adder cells is CI-testable.

□

The foregoing cell modification can be implemented in hardware using much the same approach as used in Figure 7.7 for the case of C-testable ILAs.

9.5 Ripple-carry Adders

Ripple-carry adders constructed of 1-bit full adder cells in cascade were studied in Sections 7.3, 9.1 and 9.3; see Examples 7.3 and 9.1. It was shown that these adder ILAs are C- and I-testable but not CI-testable. Here we shall prove that this result is also true for a more general ripple-carry adder ILA RA^k composed of k-bit adder cells L^k. RA^k is also referred to as a group-carry adder, and is used in the processor arrays PA^k and $PA^{k,n}$ introduced in Section 4.2.

Let the k-bit adder cell L^k be as shown in Figure 9.10, where A and B are the two k-bit input operands to be added. The various signal sets associated with L^k are

Vertical input Y

State X	00	01	10	11	y_a
0	(0,0)	(0,1)	(0,1)	(1,0)	(1,1)
1	(0,1)	(1,0)	(1,0)	(1,1)	(0,0)

Original table (columns 00–11); Added column (y_a)

(a)

	00	01	10	11	y_a
0	(0,0)	ϕ	ϕ	(1,0)	ϕ
1	ϕ	(1,0)	(1,0)	ϕ	(0,0)

(b)

	00	01	10	11	y_a
0	ϕ	(0,1)	(0,1)	ϕ	(1,1)
1	(0,1)	ϕ	ϕ	(1,1)	ϕ

(c)

Figure 9.9. The modified flow tables (a) F', (b) F'_0 and (c) F'_1 of a 1-bit full adder cell

Input Y

A B

k k

Carry in X

k-bit adder cell L^k

$\hat{X}$ Carry out

k

Sum output $\hat{Y}$

(a)

Input $Y = A \times B$

State X	(0, 0)	. .	(0, 2^k-1)	(1, 0)	. .	(1, 2^k-1)	(2, 0)	. .	(2, 2^k-1)	. . .	(2^k-1, 0)	. .	(2^k-1, 2^k-1)
0	(0, 0)	. .	(0, 2^k-1)	(0, 1)	. .	(1, 0)	(0, 2)	. .	(1, 1)	. . .	(0, 2^k-1)	. .	(1, 2^k-2)
1	(0, 1)		(1, 0)	(0, 2)		(1, 1)	(0, 3)		(1, 2)		(1, 0)		(1, 2^k-1)
	Block 0			Block 1			Block 2				Block 2^k-1		

(b)

Figure 9.10. (a) A k-bit adder cell L^k, and (b) the flow table F

$X = \hat{X} = \{0,1\}$, $A = B = \hat{Y} = \{0,1,2,\ldots,2^k-1\}$ and $Y = A \times B$, the Cartesian product of A and B. For brevity, signals are denoted by decimal numbers. The cell flow table F for L^k is shown in Figure 9.10b. F is a reduced RT flow table, hence by Theorem 7.2 the adder ILA RA^k is C-testable.

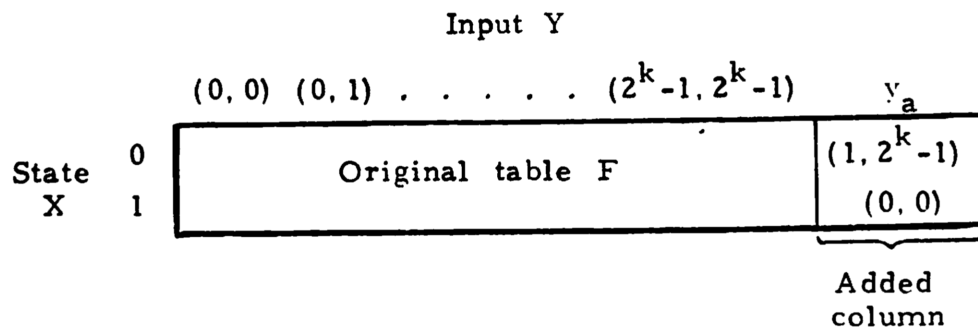
Now it is evident that in every block of F indicated in Figure 9.10b, the output entry $\hat{y}_j = j$, for $j = 0,1,\ldots,$ 2^k-1, appears exactly once in each row. This implies that each partitioned flow table F_j has exactly 2^k entries in each row, since there are 2^k blocks. The numbers of 0 and 1 next-state entries in row x_0 of F_j are $j+1$ and 2^k-j-1, respectively; in row x_1 they are j and 2^k-j, respectively. Therefore every state appears at least once as a next-state entry in each F_j. Since the $\hat{Y}$ entries in every column of F are different in each row, the two states 0 and 1 are I-distinguishable from each other in every partitioned flow table F_j. Hence RA^k is I-testable according to Theorem 9.2.

Now consider the partitioned flow tables F_0 and F_{2^k-1}. Neither one is an RT flow table, since the transitions from state 0 to 1 in F_0, and the transitions from state 1 to 0 in F_{2^k-1} are not repeatable. The remaining 2^k-2 partitioned flow tables are all RT flow tables. Therefore according to Theorem 9.3, the ILA RA^k is not CI-testable. Thus we have the following result for group-ripple carry adders.

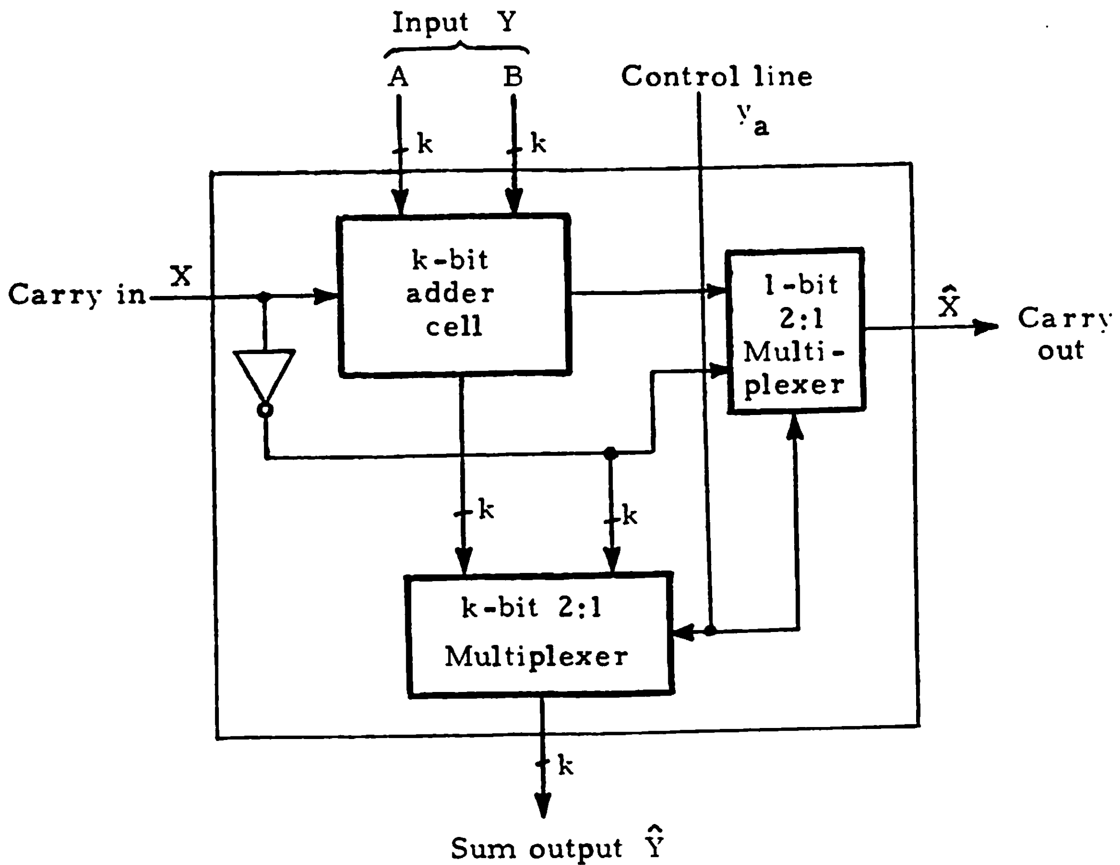
Theorem 9.6: A ripple-carry adder array composed of k-bit adder cells is C- and I-testable but not CI-testable.

For any value of k it is possible to make RA^k CI-testable by adding just one new column y_a to the flow table of L^k, as shown by Figure 9.11a. The new column y_a converts both F_0 and F_{2^k-1} into RT flow tables. Figure 9.11b shows a corresponding hardware modification to the original k-bit adder cell of Figure 9.10a.

It should be observed that although the number of tests for RA^k does not increase with the array size N, it grows exponentially with k, the word size of the basic cell.



(a)



(b)

Figure 9.11. (a) Flow table modification, and (b) cell design modification to make a ripple-carry adder ILA RA^k CI-testable

CHAPTER 10

SELF-TESTING BIT-SLICED MICROCOMPUTERS

The growing complexity of integrated circuits (ICs) has greatly increased the cost of external testing methods for digital systems. It has therefore become increasingly attractive to incorporate test methods into the system being designed. Relatively few machines such as electronic switching systems and space-borne computers, have been designed that are completely self-testing [Clary and Sacane 1979]. The cost of these systems is very high due to the large amounts of redundant hardware they contain. Most computers include some self-testing capability based on coding techniques. Various inexpensive software-implemented procedures have been used in microprocessor-based systems [Hayes and McCluskey 1980].

Although the techniques cited above are applicable to bit-sliced systems, the inherent array-like structure of these systems can be exploited to design very efficient self-testing systems. It is evident from the analysis of Chapter 3 that bit-sliced processors can be designed to be easily testable. Using high-level circuit and fault models it was shown there that complete fault coverage can be

achieved with a relatively small number of tests. In the previous chapter we demonstrated the use of I-testing to simplify test response verification for arrays. In this chapter we investigate the application of I-testing to the design of self-testing bit-sliced systems.

Only a few attempts have been made in the past to use bit slicing in the design of self-testing computers. In 1962 Forbes et al. at IBM designed an experimental microprogrammed self-diagnosable computer, the DX-1 [Forbes et al. 1965]. It consists of a CPU partitioned into two identical slices each capable of testing the other. Heuristic diagnostic programs are used for checking the various subunits of the partitioned CPU. The concept of every slice being tested by other slices has been further investigated by Ciompi and Simoncini [Ciompi and Simoncini 1977]. Wakerly has designed a self-checking minicomputer that makes extensive use of error-detecting codes in a bit-sliced framework [Wakerly 1978]. His design is discussed and compared with our proposed design later in this chapter. Recently, Könemann et al. have proposed the use of built-in pseudorandom test pattern generators and signature analyzers to make an individual bit slice self-testing [Könemann et al. 1979]. None of the foregoing approaches guarantees complete detection (100 percent fault coverage) of all likely physical faults in the systems under consideration. Furthermore,where explicit nonrandom test

patterns are used, considerable computational effort is required to construct these patterns and their responses.

10.1 Basic Self-testing Scheme

A brief outline of the proposed scheme for self-testing is now presented and later it is applied to some practical bit-sliced devices. The approach here is to construct a complete sequence of test patterns for the device D under test. This test data is then stored in D, which is also capable of applying the test patterns internally and verifying the corresponding responses. Appropriate logic is added to D for test data storage, application and verification. To simplify the test application and verification process, and to reduce the extra logic required, we exploit some special properties of bit-sliced devices. The circuit and fault models used here to generate test patterns for D are the same as defined in Chapter 2.

A straightforward self-testing method for bit-sliced devices based on the I-testability property introduced in the previous chapter will now be developed. Let D be an array of N identical cells as in Figure 9.1, and let T_D be a test sequence that detects all faults allowed by the fault model. If D is I-testable with respect to T_D, i.e., if T_D contains only I-tests, then the expected responses

to T_D appearing at the direct (vertical) outputs of D can be verified by means of equality checkers; see Figure 9.1. This kind of response verification eliminates the need to store explicitly the expected responses. As discussed in Section 9.2, the checker circuit of Figure 9.1 can itself be designed as an ILA and integrated into D.

If a complete sequence of I-tests for D cannot be found, then one of the following two approaches to I-testing may be used. The cells of D can be redesigned using the scheme proposed in Section 9.4 such that D is completely I-testable. A second approach is to use *multicycle testing*, which is implemented as follows. Any non-I-test response word from the cells of D is first stored in an internal or external register. In a subsequent clock cycle this stored data is compared internally with the expected response, thereby making the (fault-free) output signals from every cell of D identical. The expected response in this case must be stored internally or supplied via input test patterns. As will be seen later, many bit-sliced devices already possess the necessary hardware to store and compare the response data in the manner required for multicycle testing.

To demonstrate the practical applications of the foregoing self-testing philosophy, we next examine two important components of a bit-sliced (micro-) computer,

namely, processor arrays and microprogram sequencers.

10.2 CI-testable Processors

A family of processor cells, C, C^k and $C^{k,n}$ that closely resemble commercial bit-slices was introduced and analyzed in Chapter 3. Efficient test sequences that completely test these cells were derived. In Chapter 4 it was shown that an array of such cells is C-testable (Theorem 4.2), and that test patterns for the array are easily obtained from the tests for an individual cell. Here we will show that a uniform array of cells of type C, C^k or $C^{k,n}$ is also CI-testable.

C-type Processor Arrays

Consider the processor array PA, shown in Figure 4.1, which consists of N identical copies of C in cascade. To test PA for all faults it is necessary and sufficient to apply a complete test sequence for C, such as T_C derived previously in Sections 3.2 and 3.3, to every cell in the array. In addition, the responses of the cells to their individual test sequences must be propagated to the observable outputs of PA. Here we are interested in modifying the test sequence T_C for an individual cell C to form a complete sequence of I-tests or CI-tests for the array PA.

The processor cell C has six register-level modules: four combinational modules M_R, M_S, M_F and M_L, and two synchronous sequential modules M_A and M_T as shown in Figure 3.1. The test sequences for these modules were derived in Sections 3.2 and 3.3, and appear in Tables 3.3 to 3.6. The test sequence T'_R of Table 3.3 for the multiplexer modules M_R and M_S consists of parallel or P-tests that are simultaneously applicable to every cell of PA. As discussed in Section 4.1, T'_R can easily be extended into corresponding array input patterns. Since all cells of PA have identical input patterns, when P-tests are used, their expected Y output responses must be identical. Hence the extended test patterns of T'_R for PA that test the multiplexer modules M_R and M_S of every cell are CI-tests. Figure 4.2a shows the application of one such CI-test pattern to PA. Similarly, the test patterns of Table 3.6 for the sequential modules M_A and M_T of C are all P-tests, and hence the sequence T'_e of Table 3.6 is easily extended into a sequence of CI-tests for PA.

Next consider the tests for the ALU module M_F given in Tables 3.3 and 3.4. The P-test patterns of Table 3.3 are common to the modules M_R, M_S and M_F, so they can be extended into CI-tests as discussed above. As established in Section 4.1, all but six test patterns, namely t_3, t_4, t_7, t_8, t_{11} and t_{12} of Table 3.4 are P-tests. These six patterns are non-parallel or $\bar{P}$-tests in which $CO \neq CI$;

they are listed here in Table 10.1. These $\bar{P}$-tests are used for verifying the ALU arithmetic operations f_0, f_1 and f_2 of Table 3.1. All arithmetic operations use two's-complement ripple-carry addition or subtraction which is implemented by a 1-bit full adder that forms part of M_F in each cell.

Earlier in Sections 7.3 and 9.1 we studied in detail an N-bit ripple-carry adder array RA composed of 1-bit full adder cells. It was proved that RA is C-testable with only eight C-tests (Example 7.3), and is also I-testable with a minimum of 6+2N I-tests (Example 9.1). This implies that RA has only six tests that are both C- and I-tests. These CI-tests correspond to the six state transitions appearing in the full adder flow table of Figures 5.4b and 9.3a, specifically those transitions in which the carry input and output signals are the same, i.e., CO = CI. The remaining two transitions τ_1: $0 \xrightarrow{11} 1$ and τ_2: $1 \xrightarrow{00} 0$ of the adder flow table are such that CI $\neq$ CO. The test patterns t_1 and t_2 of Table 10.1, which are the same as t_3 and t_4 of Table 3.4, verify τ_1 and τ_2 of the adder flow table corresponding to the ALU function f_0, in a single processor cell C. Similarly the test pattern pairs $\{t_3, t_4\}$ and $\{t_5, t_6\}$ of Table 10.1 verify the transition pair of the flow tables in which CO $\neq$ CI, corresponding to the ALU functions f_1 and f_2, respectively. Table 4.1 shows how these

Table 10.1. The six non-parallel ($\bar{P}$) test patterns of the ALU module M_F of the processor cell C.

No.	ALU Destination Control		ALU Function Control			ALU Source Control			Data In	Multiplexer Outputs		Carry In	Cell Outputs	
	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	D	R	S	CI	Y	CO
1	1	1	0	0	0	0	1	1	1	1	1	0	0	1
2	1	1	0	0	0	0	1	0	0	0	0	1	1	0
3	1	1	0	0	1	0	1	1	0	0	1	0	0	1
4	1	1	0	0	1	0	1	0	1	1	0	1	1	0
5	1	1	0	1	0	0	1	0	1	1	0	0	0	1
6	1	1	0	1	0	0	1	1	0	0	1	1	1	0

Note: In all cases $(I_9, I_8, RI, LI) = (0,0,d,d)$.

six $\bar{P}$-test patterns for a single cell can be extended into C-tests for PA. These C-tests are, for convenience, repeated here in Table 10.2. For example, the test pattern t'_3, i.e., test no. 3 of Table 10.2, applies t_1 of Table 10.1 to even-numbered cells of PA. At the same time, t'_3 applies t_2 to the odd-numbered cells. On the other hand t'_4 of Table 10.2 applies t_2 to even-numbered cells and t_1 to odd-numbered cells. However, the C-tests of Table 10.2 are not I-tests since they produce non-identical Y signals from the cells. It is possible to convert these C-tests into CI-tests for PA by introducing additional test cycles as mentioned before. This multicycle approach is discussed below.

The non-identical output signals produced at the N-output lines $Y_0{:}Y_{N-1}$ of PA by a C-test that is not an I-test, such as t'_3 of Table 10.2, are first stored internally in a register, say M_A, of the cells. This is done by propagating the Y output signal of a cell back to register M_A via the shifter module M_L. In other words, the N-bit response of PA to a non-I-test is not checked during the present clock cycle; instead it is treated as a don't care output. During the next clock cycle this stored word is compared bit-by-bit with the expected response word, which is applied to the external D inputs $D_0{:}D_{N-1}$ as follows. The D input signals $D_0{:}D_{N-1}$ and the register

Table 10.2. Test patterns for the processor array PA that apply the $\bar{P}$-test patterns of the ALU module M_F to every cell

No.	Shift Inputs		Carry In	I-bus										D-bus				
	RI_0	LI_{N-1}	CI_0	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	D_0	D_1	$D_2 \cdots$	D_{N-2}	D_{N-1}
1	d	d	d	0	1	1	1	1	0	0	0	0	0	0	1	0...	0	1
2	d	d	d	0	1	1	1	1	0	1	0	0	0	1	0	1...	1	0
3	d	d	0	1	1	0	0	0	0	1	1	0	0	1	0	1...	1	0
4	d	d	1	0	1	0	0	0	0	1	1	0	0	0	1	0...	0	1
5	d	d	0	1	1	0	1	0	0	1	1	0	0	0	1	0...	0	1
6	d	d	1	0	1	0	1	0	0	1	1	0	0	1	0	1...	1	0
7	d	d	0	0	1	0	0	1	0	1	1	0	0	1	0	1...	1	0
8	d	d	1	1	1	0	0	1	0	1	1	0	0	0	1	0...	0	1

output signals $A_0{:}A_{N-1}$ are propagated to the two N-bit operand inputs $R_0{:}R_{N-1}$ and $S_0{:}S_{N-1}$ of the ALU module M_F via the source multiplexers. Then the previously verified ALU EXCLUSIVE-OR function f_6 is used to compare these two normally identical operands, producing the output signal $Y_i = R_i \oplus S_i = D_i \oplus A_i$ for all i. Hence the expected Y output signal from every cell should now be 0. For example, again consider the C-test pattern t_3' of Table 10.2. t_3' can be replaced by a sequence $t_3'' t_{p_3}' t_1'$ of length three, i.e., three clock cycles. Here t_3'' is t_3' with the register control inputs I_6 and I_7 set to 0 instead of 1 so that the present response word $Y_0{:}Y_{N-1}$ is stored in register A. t_{p_3}' is the propagating test input for the array PA used during the second clock cycle and setting $RI_0 = LI_{N-1} = d$, $CI_0 = d$, $I_0{:}I_9 = 0100111100$ and $D_0{:}D_{N-1} = 010101...01$; the latter is the expected response to t_3'. t_1' is a transfer input that restores the contents of register A to its original values. In the foregoing manner all test patterns for the ALU module M_F of C can be converted into single or multicycle CI-tests for the array PA. The number of array input patterns needed for the ALU modules of PA is 86, which includes the 36 test patterns that are common to M_R and M_S.

The remaining module of C, namely the shifter M_L, involves interaction between the adjacent cells of PA via

the shift lines. Hence the complete test sequence T'_L for M_L given in Table 3.5 of Section 3.2 contains only $\bar{P}$-tests. Again these $\bar{P}$-test patterns are extended in Section 4.1 into C-tests for the processor array PA. For convenience, these C-tests which were originally listed in Table 4.2, are repeated in Table 10.3. The L signals produced at the outputs of the shifter modules $M_{L_0}:M_{L_{N-1}}$ of PA by any C-test pattern of Table 10.3 are non-identical. As before, the non-identical shifter output signals $L_0:L_{N-1}$ are stored in registers $A_0:A_{N-1}$ of PA in the first clock cycle. During the next clock cycle, this stored word is compared with the expected L signals which are applied by the test pattern source to the data input lines $D_0:D_{N-1}$ of PA. Again the previously verified ALU EXCLUSIVE-OR function f_6 is used for this comparison. Consequently each C-test t'_i of Table 10.3 is replaced by a sequence $t'_i t'_{p_i}$, where t'_{p_i} is such that $RI_0 = LI_{N-1} = CI_0 = d$, $I_0:I_9 = 0100111100$ and $D_j = A_j = L_j$, for $j = 0$ to $N-1$. Hence the number of CI-tests required to completely test the shifter modules of PA is 64.

From the foregoing analysis it follows that every test pattern in T_C, the complete sequence of tests for a single cell C, can be extended to a CI-test sequence for PA using multicycle testing for some test patterns. The number of CI-test patterns needed for PA, includes 86 for

Table 10.3 Array input patterns for testing the shifter module M_L in every cell of the processor array PA

No.	Shift Control I_9	I_8	Shifter Inputs RI_i	F_i	LI_i	Data in D_{i-1}	D_i	D_{i+1}
1	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	1	0
3	0	0	1	0	0	1	1	0
4	0	0	1	1	0	0	1	0
5	0	0	0	0	1	1	1	1
6	0	0	0	1	1	0	1	0
7	0	0	1	0	1	1	1	0
8	0	0	1	1	1	0	1	0
9	0	1	0	0	0	1	1	1
10	0	1	0	1	0	0	1	0
11	0	1	1	0	0	1	0	1
12	0	1	1	1	0	1	0	0
13	0	1	0	0	1	0	1	0
14	0	1	0	1	1	1	1	1
15	0	1	1	0	1	0	0	0
16	0	1	1	1	1	0	0	1
17	1	0	0	0	0	1	1	1
18	1	0	0	1	0	0	1	0
19	1	0	1	0	0	0	0	0
20	1	0	1	1	0	1	1	1
21	1	0	0	0	1	1	0	0
22	1	0	0	1	1	0	0	1
23	1	0	1	0	1	0	1	1
24	1	0	1	1	1	1	0	0
25	1	1	0	0	0	1	1	1
26	1	1	0	1	0	0	1	0
27	1	1	1	0	0	1	0	0
28	1	1	1	1	0	1	1	0
29	1	1	0	0	1	0	0	1
30	1	1	0	1	1	0	1	1
31	1	1	1	0	1	1	0	1
32	1	1	1	1	1	1	1	1

Notes: In all cases

(i) $(I_7,I_6,I_5,I_4,I_3,I_2,I_1,I_0,CI_0) = (0,0,1,1,0,0,1,0,d)$

(ii) $RI_i = F_{i-1} = L_{i-2}$ and $D_i = D_{i+3}$ for all i

the modules M_R, M_S and M_F, 18 for the registers M_A and M_T, and 64 for the shifter M_L. Thus the processor array PA of C-type cells is CI-testable with at most 86+18+64 = 168 test patterns.

Other Processor Arrays

The foregoing method of constructing CI-tests for PA from the tests for an individual cell is equally applicable to an array of the extended cells C^k and $C^{k,n}$. C^k is a k-bit version of C, while $C^{k,n}$ is formed from C^k by replacing its single k-bit register A with an $n \times k$-bit register file. Testing these more general cells was discussed earlier in Section 3.5.

As before, let PA^k and $PA^{k,n}$ denote processor arrays of type C^k and $C^{k,n}$ cells, respectively. Consider first the testing of PA^k. A complete test sequence T_{PA^k} of C-tests for PA^k was derived in Section 4.2. In C^k the five modules M_R, M_S, M_L, M_A and M_T of C are replicated and interconnected as in the cascade structure of PA; see Figures 3.6 and 4.1. Hence the test sequences for these replicated modules in PA^k consisting of N_1 C^k cells, are the same as the test sequences for the corresponding modules in an array containing N_1k C-type cells. Thus CI-tests that test all replicated modules in PA^k are easily derived. The remaining modules of PA^k, namely the k-bit ALU modules, are tested in essentially the same way as the 1-bit ALU

modules of PA. The ALU logical operations do not involve any interaction between cells, and hence result in $CO = CI$. Therefore CI-tests for these operations are readily obtained. The ALU arithmetic operations use a ripple-carry adder array RA^k consisting of k-bit adder cells in cascade. RA^k was analyzed in Section 9.5, where it was shown (Theorem 9.6) that RA^k is C-testable but not CI-testable. The corresponding C-tests for PA^k are easily constructed. Most of these C-tests result in $CO = CI$ at the cell boundaries, and they are also CI-tests. The remaining C-tests that result in $CO \neq CI$, can be altered to CI-tests by introducing an extra test cycle after each C-test is applied. Therefore the ALU modules of PA^k can also be completely tested using CI-tests only. Thus PA^k is CI-testable.

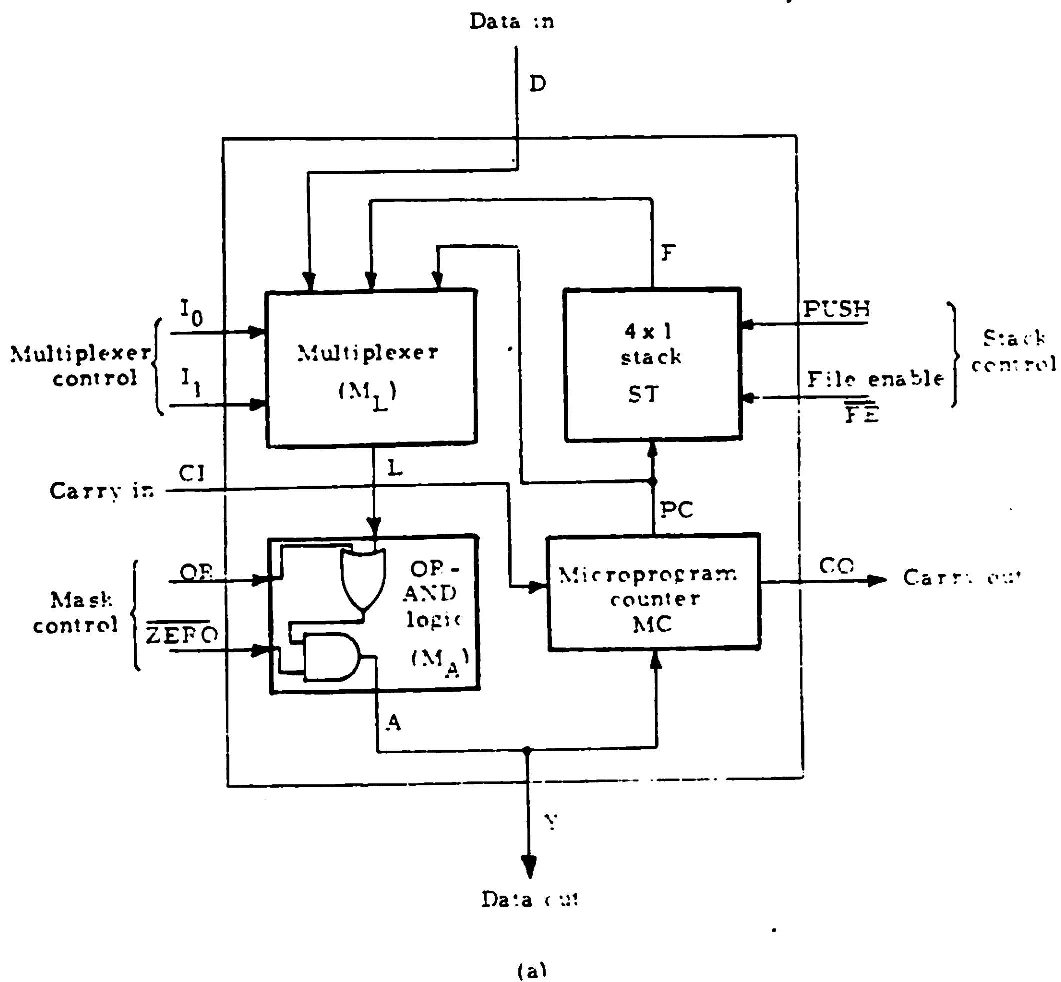
The array $PA^{k,n}$ is tested in essentially the same way as PA^k. The test requirements of $PA^{k,n}$ differ from those of PA^k only with respect to the $n \times k$-bit register file of $C^{k,n}$. (Testing the register file was discussed in Section 3.5.) However, since the register file is replicated in the cascade structure of $PA^{k,n}$, it is again possible to apply the test patterns for an individual register file to every cell of $PA^{k,n}$ simultaneously. This is similar to testing the replicated register modules in PA. Thus we have the following general result.

Theorem 10.1: A uniform processor array of cells of the type C, C^k or $C^{k,n}$ is CI-testable.

10.3 I-testable Microprogram Sequencers

A 1-bit microprogram sequencer S, based on the 2909 slice discussed in Section 2.2, appears in Figure 10.1. Its function is to generate the microinstruction addresses required in a microprogrammed CPU. S consists of two combinational circuits: a multiplexer module M_L and an OR-AND logic module M_A, and two synchronous sequential circuits: a microprogram counter MC and a 4×1-bit LIFO stack ST. The multiplexer M_L selects the next microinstruction address from one of three sources: the vertical external input D, the stack ST, or the counter MC. ST contains a 4×1-bit register file and a stack pointer which is a 2-bit up-down binary counter. Thus S is essentially a 1-bit version of the 4-bit 2909 slice of Figure 2.5, except that it does not include a latched data input line corresponding to the R line of the 2909.

Test patterns for S are derived in much the same way as described in Sections 3.2 and 3.3 for the processor cell C. (For a complete list of test patterns for S see Appendix D.) Testing the combinational modules M_L and M_A is straightforward; the corresponding test sequences T'_L and T'_A are given in Table D.1 of Appendix D. In this



(a)

Multiplexer control

I_1	I_0	L output
0	0	PC
0	1	D ⊕ PC
1	0	F
1	1	D

Stack control

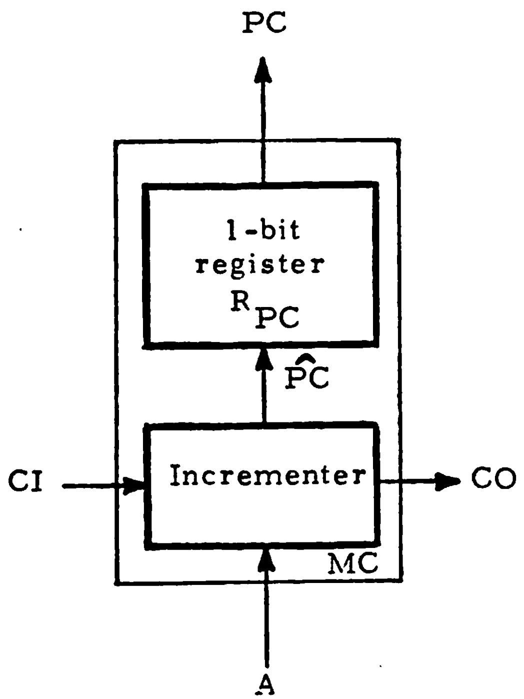
$\overline{FE}$	PUSH	Stack operation
1	d	No operation
0	1	Increment stack pointer and push PC onto stack
0	0	Pop stack and decrement stack pointer

(b)

Figure 10.1. (a) A 1-bit microprogram sequencer cell S, and (b) its control fields

table, tests $t_1:t_5$ are for initializing the stack, $t_6:t_{37}$ form the sequence T_L', and the sequence T_A' is formed by the remaining tests $t_{38}:t_{45}$. The microprogram counter MC is treated as a 1-bit register R_{PC} with an incrementer module as shown in Figure 10.2a. The flow table of the incrementer is given in Figure 10.2b. A sequence of tests that verifies the flow table of the incrementer as well as the state table of R_{PC} is given in Table D.2.

The remaining stack circuit ST of S is tested as follows. ST is modeled as four identical 1-bit register or memory cells SK_0, SK_1, SK_2 and SK_3, with a common stack pointer SP as in the 2909 slice. Figure 10.3a shows the stack modeled along these lines. SP is a 2-bit modulo four up-down binary counter with the state table given in Figure 10.3b. Data is written into or read from the memory cells $SK_0:SK_3$ by means of a push ($PUSH = 1$ and $\overline{FE} = 0$) or a pop ($PUSH = 0$ and $\overline{FE} = 0$) operation. The 2×32 state table of each SK_i has a very simple structure, since the contents of SK_i are changed only by a push operation when SP's output represents i-1. Also the output signal F of the stack ST is always equal to the present state of SK_i independent of the stack control inputs, where i is the state of SP. (For brevity the state table of SK_i is not shown explicitly.) Applying our fault model of Section 2.4 to the circuit of Figure 10.3a, it is evident that to test



(a)

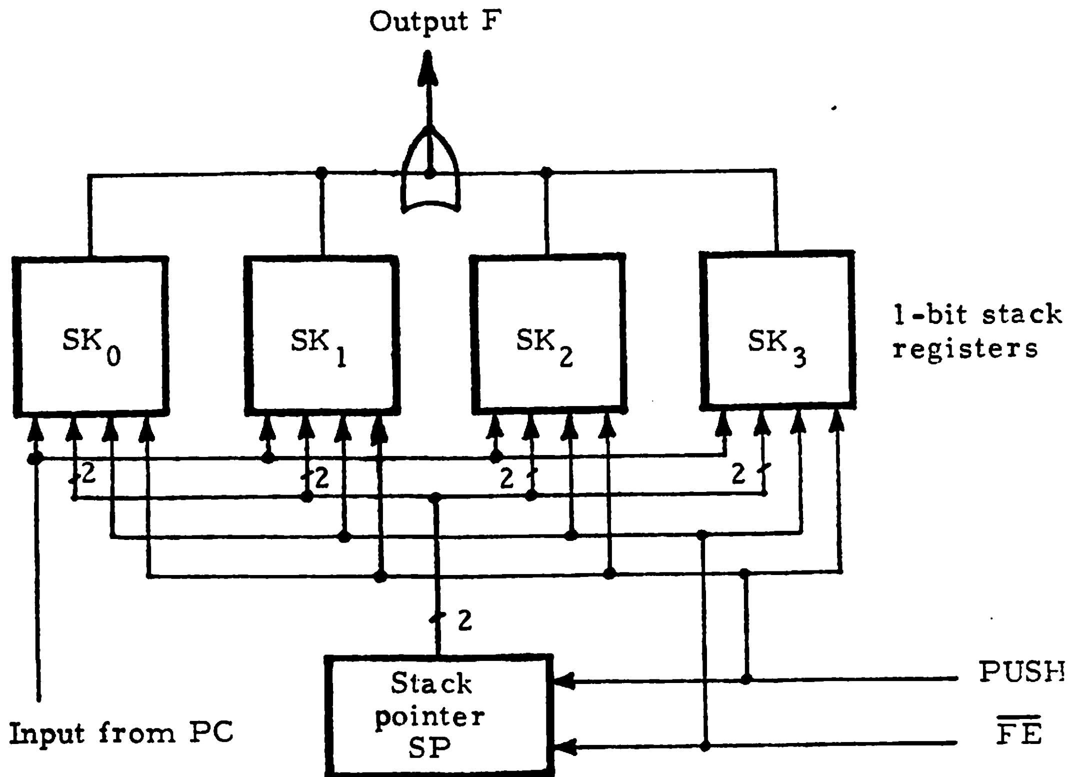
Input A

State CI	0	1
0	(0, 0)	(0, 1)
1	(0, 1)	(1, 0)

Each entry denotes (CO, $\hat{PC}$)

(b)

Figure 10.2. (a) A circuit model of the micro-program counter MC, and (b) the flow table of the incrementer part



(a)

State SP	Input PUSH $\overline{FE}$: 00	01	10	11
00	11	00	01	00
01	00	01	10	01
10	01	10	11	10
11	10	11	00	11

(b)

Figure 10.3. A circuit model of the 4 × 1-bit stack ST, and (b) the state table of its stack pointer SP

ST it is necessary and sufficient to verify the state tables of SP, SK_0, SK_1, SK_2 and SK_3. While verifying the state table of SP, the state of SP must be identified. One way of doing this is to construct a distinguishing sequence as follows. Let $SK_0:SK_3 = (0,0,1,1)$. Now consider two successive pop operations on the stack, and let Q be the corresponding output sequence of length two appearing at F. Q identifies the state of SP, which is 00, 01, 10 or 11 if Q is 01, 00, 10 or 11, respectively. Using two successive pop operations as a distinguishing sequence, a checking sequence that tests SP can easily be constructed. Table D.3 in Appendix D gives one such sequence of length 49.

The four cells $SK_0:SK_3$ all have the same five input lines, and it is therefore possible to merge their individual checking sequences. One way of doing this is as follows. The contents of all four cells are read after each one of the 32 binary input combinations is applied as a test vector to the inputs of the cells. The cell contents are read using four successive pop operations. A checking sequence CS_i constructed using this approach is given in Table D.4 of Appendix D. This test sequence verifies all the 16 entries in the eight columns corresponding to $SP = i$ in the four state tables of $SK_0:SK_3$. Thus by applying the four sequences CS_0, CS_1, CS_2 and CS_3 in succession, all 64 entries of their state tables can

be verified in all four cells SK_0:SK_3.

We conclude that the microprogram sequencer cell S can be completely tested with a test sequence T_S obtained by concatenating the various individual sequences derived above. T_S is of length $45+7+49+(6+4 \times 86) = 451$.

An N-bit microprogram sequencer MS is realized by connecting N identical copies of S in cascade as shown in Figure 10.4. Again we can extend the foregoing tests for S into I- or CI-tests for the array MS. The approach we use is similar to that used for processor arrays in the previous section.

Consider, for example, the subarrays formed by the two modules M_L and M_A, and the stack ST. Since there is no communciation between the cells of MS via M_L, M_A or ST, their test sequences given in Appendix D (see Tables D.1, D.3 and D.4) can be applied to every cell in parallel. The identical test responses of these three circuits appear at the observable outputs Y_0:Y_{N-1} of MS resulting in a sequence of CI-tests for M_L, M_A and ST. The remaining circuit MC of S involves communication between adjacent cells of MS via the carry lines. The flow table of the 1-bit incrementer module of MC shown in Figure 10.2b contains a non-repeatable transition $1 \xrightarrow{0} 0$. Hence the counter subarray CA formed by the counter circuits MC of every cell is not C-testable; see also Example 8.2 in Section 8.5. Moreover, CA is not I-testable as proved in Example 9.3

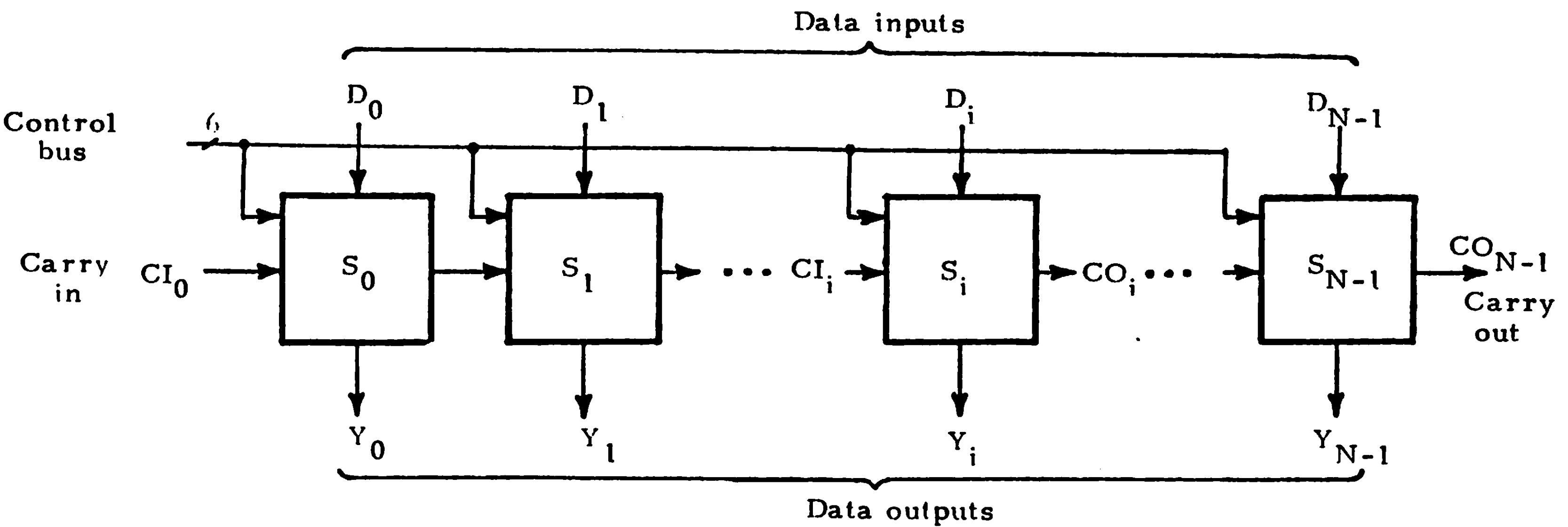


Figure 10.4. An N-bit microprogram sequencer array MS composed of N copies of the sequencer cell S

of Section 9.1. Consequently, MS is also neither C-testable nor I-testable. However, it can be made I-testable as follows.

Consider the seven test patterns for MC appearing in Table D.2. The first six patterns $t_1{:}t_6$ are P-tests that can be applied simultaneously to every cell of MS, and hence they easily extend to CI-tests for MS. The last test pattern t_7 corresponds to the above-mentioned non-repeatable transition and can only be applied to one cell of MS at a time. Let t′ be non-I-test for the subarray CA that applies t_7 to any cell S_i. The responses of CA to t′ at the $\hat{PC}$ outputs of MC in every cell (see Figure 10.2a) are not identical. We again use multicycle testing to make the responses from all cells to t′ identical in a subsequent test cycle. The N-bit response word at $\hat{PC}$ is first stored in the R_{PC} registers, and in the next cycle it is compared bit-by-bit in the M_L multiplexer modules of the N cells $S_0{:}S_{N-1}$ with the expected response applied externally to the data inputs $D_0{:}D_{N-1}$. For this comparison a special EXCLUSIVE-OR function $L = D \oplus PC$ has been introduced into M_L. It is selected by means of the fourth unused multiplexer control input combination $I_1 I_0 = 01$ as shown in Figure 10.1b. Thus all seven test patterns of MC can be extended to a sequence of 5+2N I-tests for MS. MS can be completely tested by a sequence T_{MS} of I-tests, formed by concatenating the foregoing array input sequences.

Theorem 10.2: The N-bit microprogram sequencer array MS of Figure 10.4 is I-testable, and has an I-test sequence T_{MS} of length 449+2N.

10.4 A Self-testing Microprocessor

In this section we describe the design of a bit-sliced CPU that uses I-testability to achieve self-testing in an efficient way. Our approach is to modify a conventional bit-sliced design, such as that discussed in Section 2.1, to use I-testing and equality checkers in all major subcircuits. Figure 10.5 shows the overall structure of a CPU designed along these lines. It consists of a bit-sliced execution unit (E-unit) and a microprogrammable instruction unit (I-unit). The various components of these two units and their test requirements are discussed in the sequel.

Execution Unit

The E-unit of the proposed CPU consists of an I-testable processor array, and some additional logic called the status logic SL which is used to generate the processor's status or flag signals. In the proposed design the N-bit CI-testable processor array PA of Section 10.2 is used; processors PA^k or $PA^{k,n}$ could also be used, if desired. The status logic SL generates five typical status

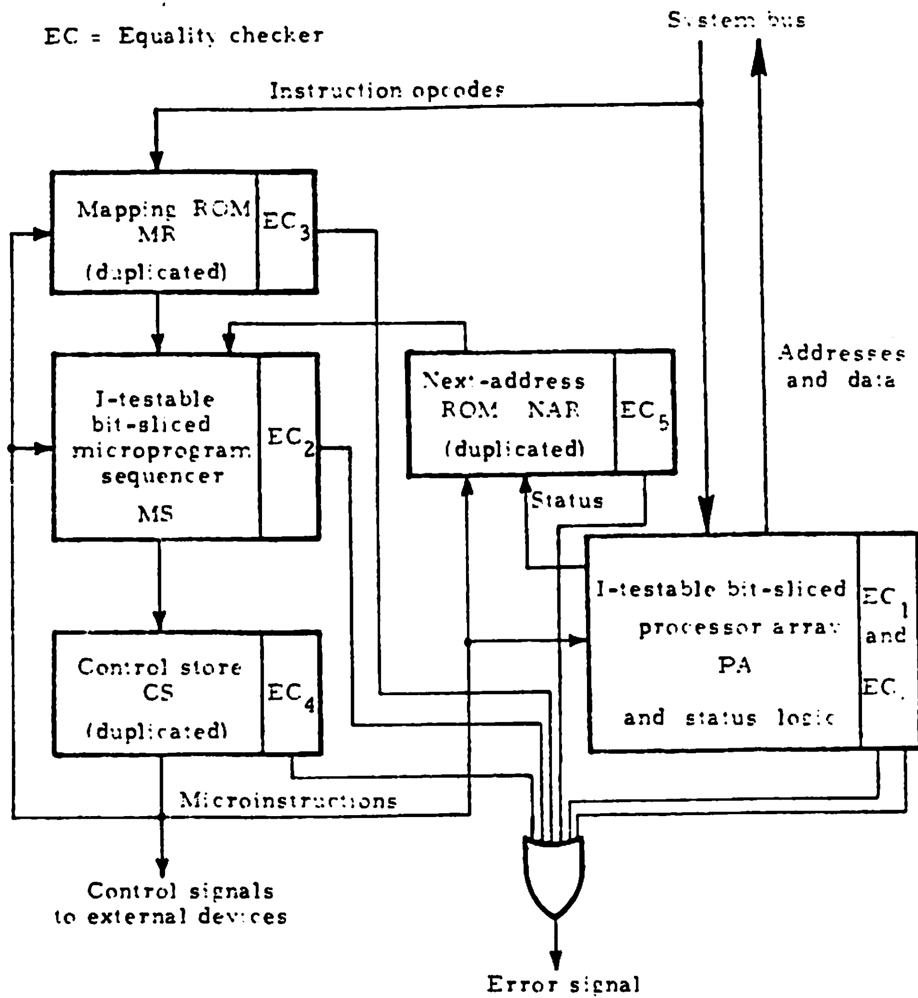


Figure 10.5. A self-testing bit-sliced CPU

bits: a carry bit CY, a sign bit S, an overflow bit OVF, a parity bit P, and a zero bit Z. These bits together form the processor status word, which is generally used for conditional branching during the execution of a program. Status bits are also useful for system diagnosis. The carry and sign signals are provided directly by the processor PA itself. The carry signal CY is the carry out signal CO_{N-1} of the rightmost cell C_{N-1} of PA, and the sign signal S is the same as the direct output signal Y_{N-1} of C_{N-1}. The remaining three status signals are obtained as follows. The parity and zero signals are generated by the two ILA circuits shown in Figure 10.6a. OVF is the EXCLUSIVE-OR function of the two carry signals CO_{N-1} and CO_{N-2} of PA; see Figure 10.6b. The ILA circuits of Figure 10.6a may be incorporated into PA. The processor status information can then be stored, at the end of every processor cycle, in a 5-bit status or flag register SR as depicted in Figure 10.6c. The outputs of SR are used by the I-unit to select the next microinstruction address.

The processor array PA requires 168 CI-test patterns as discussed in Section 10.2; these are conveniently placed in the control store CS of the I-unit. The output responses of PA to these CI-tests are verified by an N-bit equality checker EC_1 of the type discussed in Section 9.2.

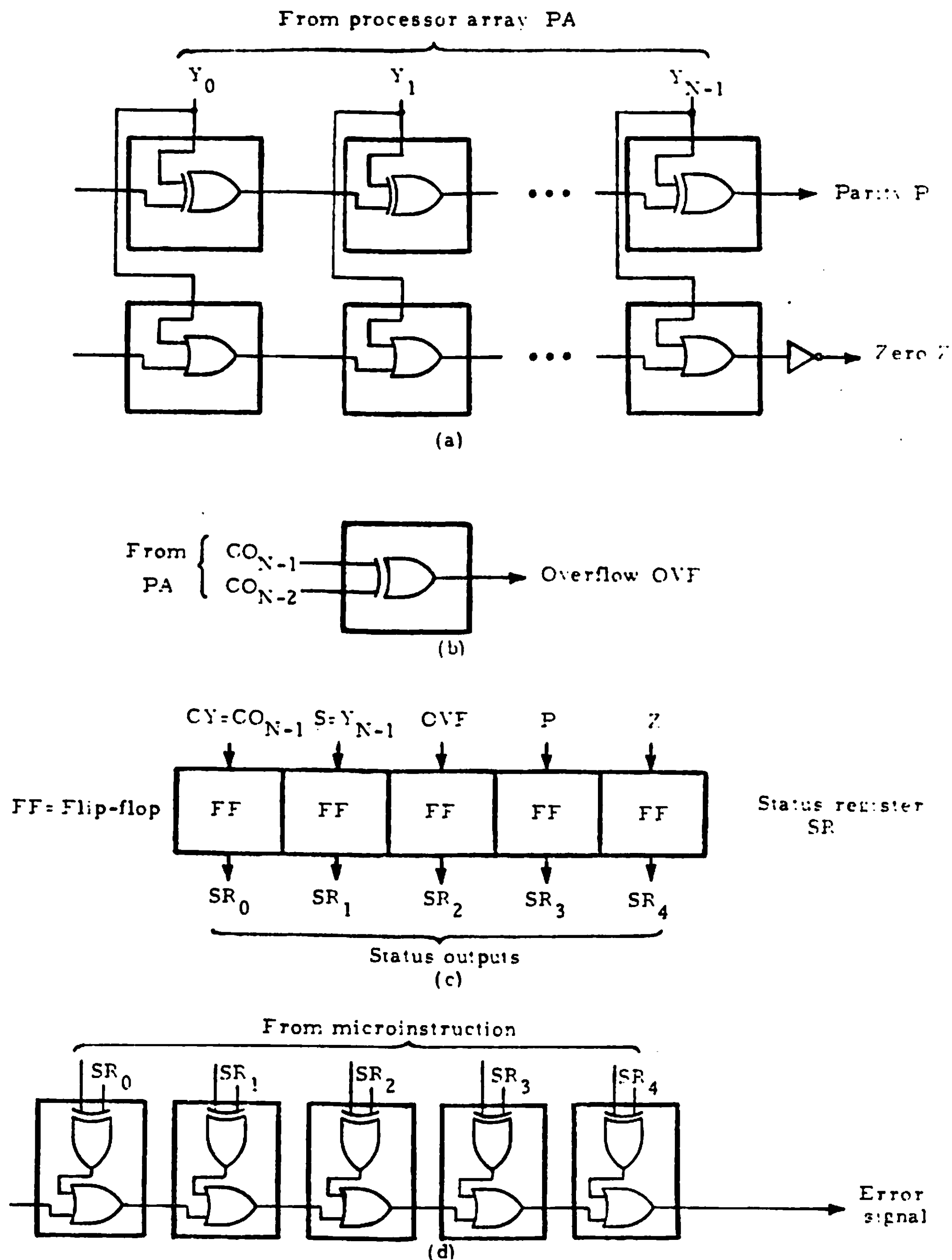


Figure 10.6. The status logic SL: (a) parity and zero flag circuits, (b) overflow circuit, (c) status register SR, and (d) checker EC_6 for SR

Note that multicycle testing is used in some cases. The responses of PA to some of its 168 test patterns are stored internally during a special test cycle and then checked during the next clock cycle. Such patterns are needed for testing the ALU and the shifter modules in PA; see Section 10.2 for details. Thus cell responses during certain clock cycles in multicycle testing are to be ignored by the checker EC_1. This can be specified by a single control bit in the microinstruction that indicates when EC_1 should ignore the current response of PA. As described earlier, EC_1 can be realized as an ILA and integrated into PA.

The test requirements of the status logic SL are determined as follows. The parity logic array of Figure 10.6a is optimally C-testable as proved in Section 6.1 (see Example 6.1 and Theorem 6.2) and requires only four tests. The zero logic array is not C-testable and reqires at least $2+2N$ test patterns. The 2-input EXCLUSIVE-OR gate of the overflow logic requires exactly four tests. The status register SR is modeled as five 1-bit registers or flip-flops in parallel as depicted in Figure 10.6c. Eight test patterns are sufficient to verify the state tables of all five flip-flops. The test responses of SL are observable at the five output lines $SR_0 : SR_4$ of SR. These responses are verified with the help of a 5-bit

equality checker ILA EC_6 as shown in Figure 10.6d. The expected responses of SL are stored in a microinstruction field. The 18+2N test patterns for SL are stored in the control store CS as a test microprogram.

Instruction Unit

The I-unit comprises a control store CS used for storing microinstructions, an I-testable bit-sliced microprogram sequencer MS of the kind discussed in the previous section, a mapping read-only memory (ROM) MR used to generate the starting address of a microprogram from the opcode of an external instruction, and a next-address ROM NAR, which controls the generation of the next microinstruction address by MS.

The I-testable microprogram sequencer MS requires $449+2N_1$ test patterns as specified in Theorem 10.2, where N_1 is the microinstruction address length. The N_1-bit equality checker EC_2 associated with MS verifies the test responses of MS. The remaining components MR, CS and NAR of the I-unit, which are typically ROMs, are tested by duplicating them and connecting duplicated units to the same input lines. The output lines from each of the duplicated pairs are connected to an equality checker of the type shown in Figure 10.6d. Thus the duplicated ROMs MR, CS and NAR are tested continuously by their respective

equality checkers EC_3, EC_4 and EC_5, and hence they do not require any specific test patterns or test mode.

The above approach of duplication and verification by equality checkers can be used for other unstructured or hard-to-handle parts of the system, including main memory and input-output interface circuits. Note that the replication-and-comparison test method used here for ROMs can be regarded as I-testing applied to duplicated non-ILA circuits. Conversely, I-testing may be viewed as a generalization of replication-and-comparison techniques to non-duplicated ILAs.

The six equality checkers EC_1:EC_6 used in the CPU of Figure 10.5 are ILAs of the type shown in Figure 9.7 and Figure 10.6d. They can all be tested like any other ILA. The actual test requirements for an N-bit equality checker were discussed in detail in Section 9.2. It was shown that the checker ILA is not C-testable, and requires a minimum of 4+4N test patterns.

Self-testing Procedure

As discussed above the test patterns for the processor array PA, its status logic SL, the microprogram sequencer MS and the equality checkers EC_1:EC_6 are stored in CS. They can be regarded as test (micro) programs at the microinstruction level. Two operating modes are

assumed for the CPU: a normal mode in which the CPU executes application programs, and a test mode where the CPU tests itself by executing the various test programs. Before the test mode begins, the contents of all CPU registers are saved in main memory. The test programs are then executed, and a fault is indicated by an error signal from any of the equality checkers $EC_1:EC_6$. It is possible to combine the individual error signals from the equality checkers into a single error signal by means of an OR gate as depicted in Figure 10.5. After all test programs have been applied, normal system operation is resumed.

The time spent by the CPU in the special test mode is quite small. For example, the number of test patterns required by PA, MS and $EC_1:EC_6$ can be expected to be less than 1000. With a CPU cycle time of 100 ns, these components can be tested in about 100 μs. Therefore we can afford to test the CPU very frequently, say once every second. Furthermore, in many microcomputer applications the CPU is often idle waiting for a memory or IO operation to be completed. Idle periods of this kind can be utilized for self-testing.

10.5 Comparison with Other Designs

Figure 10.7 shows a block diagram of Wakerly's somewhat more conventional self-checking bit-sliced CPU design

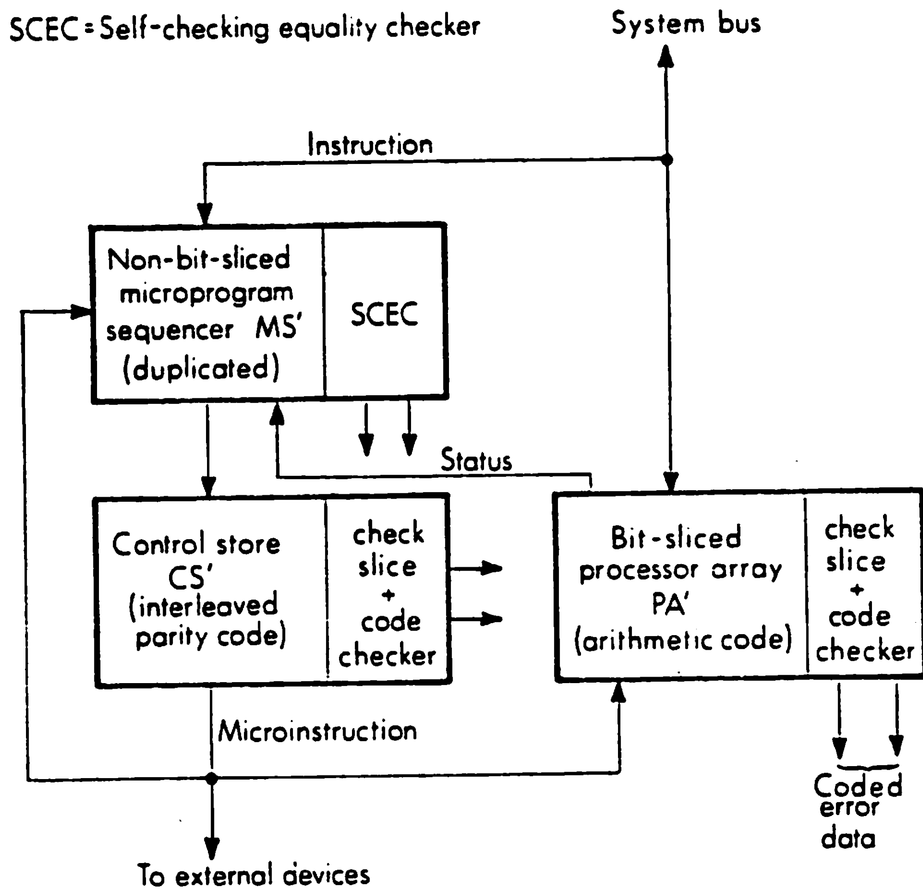


Figure 10.7. Wakerly's self-checking bit-sliced CPU

based on error-detecting codes. This design is representative of self-checking computers implemented via coding techniques, and is well-documented [Wakerly 1978, Chapter 8]. The E-unit of Wakerly's CPU consists of a 16-bit bit-sliced processor array PA′ realized as four 4-bit slices in cascade. The processor slices were designed by Wakerly but are similar in structure and functional capability to the 2901 and the 10800 slices discussed earlier (Section 2.2). The I-unit includes a 12-bit non-bit-sliced microprogram sequencer MS′ and a 4096 × 48-bit control store CS′. Thus the CPUs of Figures 10.5 and 10.7 have quite similar organizations. They differ mainly in the checking schemes used for the individual components. These differences are discussed below.

Checking the E-unit

The data path of the E-unit is checked by an arithmetic code, specifically a modulo 15 residue code. In genral, a *modulo m residue code* works as follows. An operand X is represented as two separate parts: an n-bit data part X_d and a k-bit code part X_c, also referred to as the residue or the check symbol. X_c is defined by the congruence relation $X_c \equiv X_d$ (modulo m). This encoding method has the property of preserving arithmetic operations. For example, consider the addition of two operands X and Y.

Let Z be the sum X+Y. The data part Z_d of Z is X_d+Y_d, and the code part of Z is given by $Z_c \equiv Z_d$ (modulo m) $= X_d+Y_d$ (modulo m). Now $X_d = n_1m+X_c$ and $Y_d = n_2m+Y_c$, where n_1 and n_2 are some integers. Therefore, $Z_c \equiv (n_1+n_2)m+(X_c+Y_c)$ (modulo m) $\equiv X_c+Y_c$ (modulo m). Thus the residue of the sum is the modulo-m sum of the residues of the operands.

This residue code is capable of detecting any change or error in k-1 or fewer adjacent bits of X, since the residue is k bits wide. In other words, if U is a circuit whose data is coded using the above residue code, then any physical fault in U that changes k-1 or fewer adjacent bits in the output word of U will be detected. The code can also detect many errors involving k or fewer adjacent bits. As will be discussed later, this kind of error detection does not guarantee the detection of all (physical) faults in U.

In the E-unit of Figure 10.7, m is 15, and the check symbol contains 4 bits. Thus the E-unit's main data path is 16+4 = 20 bits wide, comprising a 16-bit data word plus a 4-bit check symbol. A 16-bit data word is partitioned into four 4-bit words, each of which is processed by a separate 4-bit slice of PA′. The 4-bit check symbol is the modulo 15 sum of these four 4-bit words and is handled separately by an extra processor slice called the check slice. All five processor slices perform identical operations on their respective input data.

The E-unit is checked for its correct operation at the end of every microinstruction cycle by its code checker. The code checker actually verifies whether the residue of the 16-bit output data generated by PA′ matches that produced by the check slice. For this purpose it uses a circuit that generates the residue of the 16-bit output of PA′, and a 4-bit self-checking equality checker to compare the two residues as shown in Figure 10.8. Before this comparison is made, the check symbol generated by the check slice is passed through a rather elaborate "fixup" circuit which is explained later. The residue generator in Figure 10.8 uses three 4-bit adders in a tree structure to generate the 4-bit residue of the 16-bit result produced by PA′. The self-checking equality checker is also discussed in more detail later in this section.

The fixup circuit is essentially a 256 × 4-bit ROM as shown in Figure 10.8. It provides the necessary adjustments to the check symbol to compensate for carries that occur during arithmetic and shift operations. It is also used to force the check symbol to a desired value during system initialization and during logical operations. It is well known that logical operations do not preserve an arithmetic residue code. In fact, no single code is known that can be successfully used for both arithmetic and logical operations. This suggests that coding techniques are

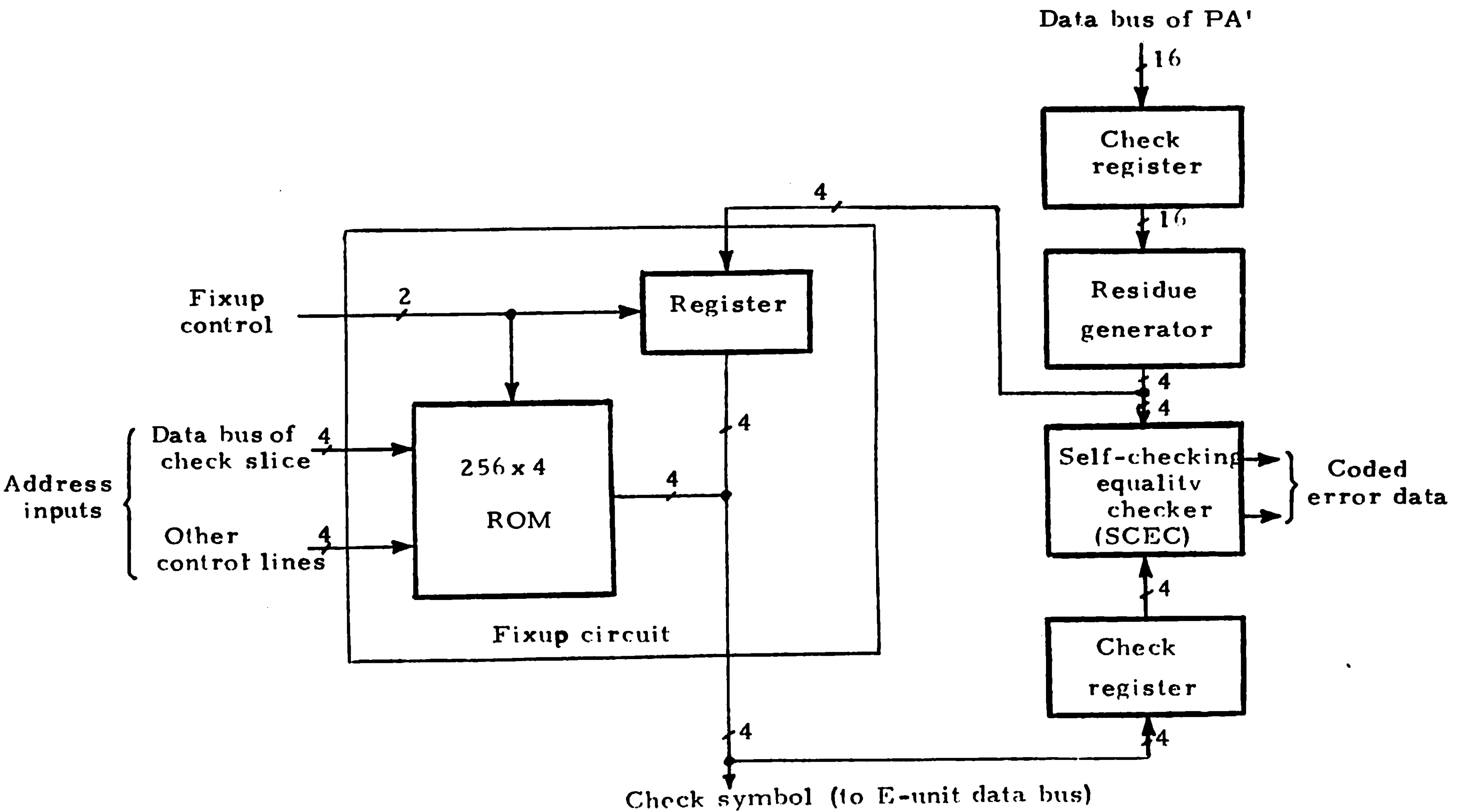


Figure 10.8. Code checker circuit for the E-unit of Figure 10.7

not suitable for realizing completely self-checking CPUs. The fixup circuit is itself checked by a self-checking equality checker (SCEC), which monitors the output of the fixup circuit as depicted in Figure 10.8. The fixup operation introduces a delay of about 40 nsec. in every 250 nsec. microinstruction cycle. In addition, one extra microinstruction cycle is required during each logical operation to correct the check symbol.

We now compare the above checking scheme for PA′ with the I-testing approach used in PA of Figure 10.5. A basic difference lies in the kinds of faults detected. The processor array PA is completely CI-tested for the well-defined set of functional faults. This includes, for example, all single stuck-line faults. On the other hand, in Wakerly's design only errors in the output data generated by the various self-checking subcircuits are detected. An *error* is an incorrect output, i.e., a non-code word, generated by the circuit due to the presence of a physical fault; the nature of the fault is unspecified. To be detectable, a fault must produce a detectable error signal at the circuit's output. The residue code used in PA′ detects any error affecting the 4-bit output word generated by any single processor slice. This kind of error detection cannot guarantee the detection of all functional or stuck-line faults in a single slice. For example, a single stuck-line fault that causes the output of a pro-

cessor slice to change from 0000 to 1111 or vice versa is not detected by the modulo 15 residue code, since both 0000 and 1111 are valid codewords. In general, it is difficult to identify the kinds of physical faults detected by any coding scheme. The relationship between physical faults and the errors induced by them depends on the circuit implementation. For these reasons the fault coverage achieved in PA′ is hard to determine, but it can be expected to be significantly less than the 100 percent functional fault coverage achieved in PA.

The additional logic to check PA′ consists of a 4-bit processor check slice and the fairly large code checker circuit of Figure 10.8. The extra logic needed for checking PA is just the equality checker EC_1 shown in Figure 10.5. Wakerly's computer provides continuous error detection thereby capturing intermittent errors, whereas PA requires a special test mode. Note, however, that the continuous error detection in PA′ increases the microinstruction cycle time by about 16 percent.

Checking the I-unit

Duplication and comparison is used for checking the non-bit-sliced microprogram sequencer MS′ of Figure 10.7, while the control store CS′ is checked by an interleaved parity code. The code used is the distance-2 b-adjacent code which works as follows. An n-bit word is divided

into k b-bit bytes, each of which is stored in a separate ROM slice. An extra ROM slice is used to store the b-bit parity byte or check symbol, where the ith bit p_i for $0 \leq i \leq b-1$ is the parity of the corresponding k i^{th} bits taken from the k bytes of the word. In Wakerly's design n is 48 and b is 4. Therefore CS′ is implemented as twelve 4096×4-bit ROM slices with an extra check slice for storing the check symbols of all 4096 words. The associated code checker for CS′ consists of four separate 12-bit parity circuits, and a 4-bit self-checking equality checker to compare the outputs of the parity checking circuits with the contents of the check slice; see Figure 10.9. The four processor status bits, namely, the carry, sign, zero and the overflow bits, are all stored in MS′, and hence are checked by the sequencer duplication scheme. MS′ also includes a next address selection ROM and a mapping ROM. Since these two ROMs are part of MS′, they are duplicated as in the CPU of Figure 10.5. Both MS′ and CS′ are then continuously monitored by their respective SCECs.

In the proposed design of Figure 10.5, I-testing is used to check the non-duplicated but bit-sliced microprogram sequencer MS, while the control store CS is duplicated. Thus the duplicated control store, the next-address ROM, and the mapping ROM of Figure 10.5 are all checked continuously as in Wakerly's design, while MS is tested

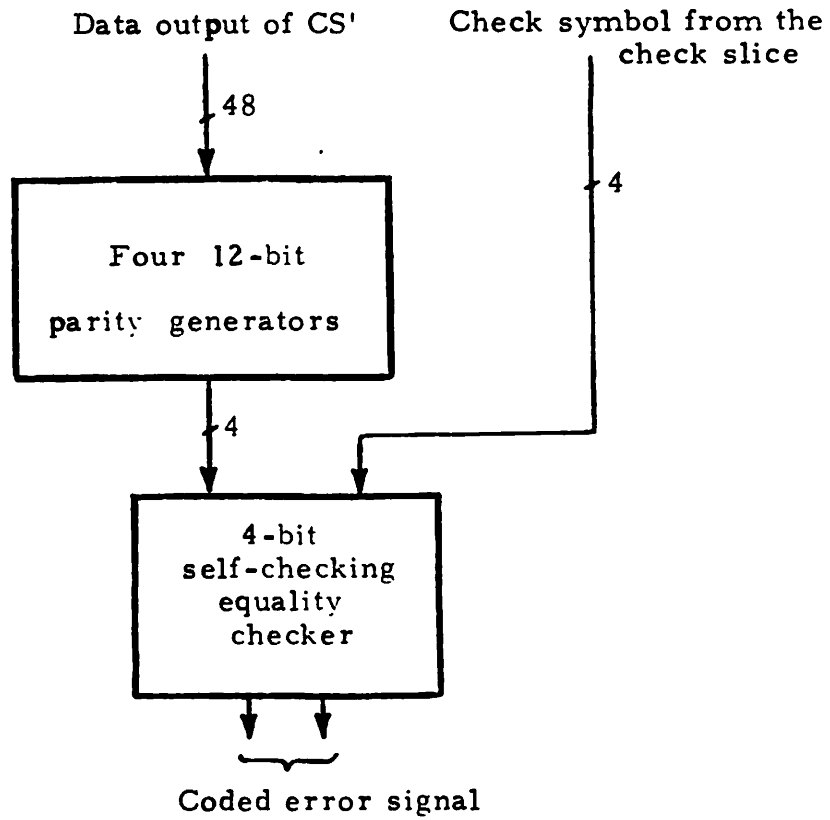


Figure 10.9. Code checker circuit for the control store CS′ of Figure 10.7

only during the special CPU test mode. Although the processor and the sequencer arrays PA and MS are not tested continuously, these units can be tested very frequently since their test sequences are quite short.

Self-checking Equality Checkers

The equality checkers used in Wakerly's design compare two n-bit words and generate an error signal whenever they are not identical. For the entire CPU of Figure 10.7 to be self-checking it is necessary that the equality checkers be self-checking. In other words, a fault of interest in a checker itself should produce a detectable error signal. For this purpose, the output of an equality checker is also encoded using an error-detecting code. Wakerly employs a conventional 1-out-of-2 code. Here the output of a self-checking equality checker is a 2-bit codeword such that 01 or 10 indicates error-free operation, i.e., exactly one out of the two output bits should always be 1. A 00 or 11 output word indicates an error. This code detects all unidirectional multiple errors, i.e., errors in which all erroneous bits are changed from 1 to 0 or vice versa.

The CPU of Figure 10.7 requires 4-bit and 16-bit SCECs. Two 4-bit SCECs are used in the code checking circuits of PA′ and CS′, and one 16-bit SCEC is used

for monitoring the duplicated MS'. Wakerly has proposed designing a 4-bit SCEC as a single custom MSI chip [Wakerly 1978]. Such chips can be interconnected in a tree structure to realize a 16-bit SCEC as illustrated in Figure 10.10. The 4-bit SCEC chip is implemented as a 4-bit *two-rail checker (TRC).* In general, a k-bit TRC is a logic circuit that maps k input signal pairs into one output signal pair such that the output signal pair is a 1-out-of-2 codeword if and only if each of the k input pairs is a 1-out-of-2 codeword. In the 16-bit SCEC of Figure 10.10 one of the 16-bit input operands, namely $B_0:B_{15}$, is inverted and compared bit-by-bit with the other operand $A_0:A_{15}$. A single 4-bit TRC chip is a 2-level combinational circuit having sixteen 4-input and two 8-input primitive gates. Therefore the SCEC of Figure 10.10 contains a total of $18 \times 5 + 16 = 106$ logic gates.

Comparing Wakerly's SCEC with the equality checkers ECs used in the proposed CPU of Figure 10.5, the following observations can be made. The equality checkers of Figure 10.5 need not be self-checking, since they are tested explicitly by applying a complete set of test patterns stored in CS. Therefore all functional faults within a single cell of an EC array (see Figure 9.7 or 10.6d) are detected. On the other hand, due to the use of the 1-out-of-2 code, an SCEC can only detect all unidirectional multiple stuck-line faults, i.e., faults in which all faulty

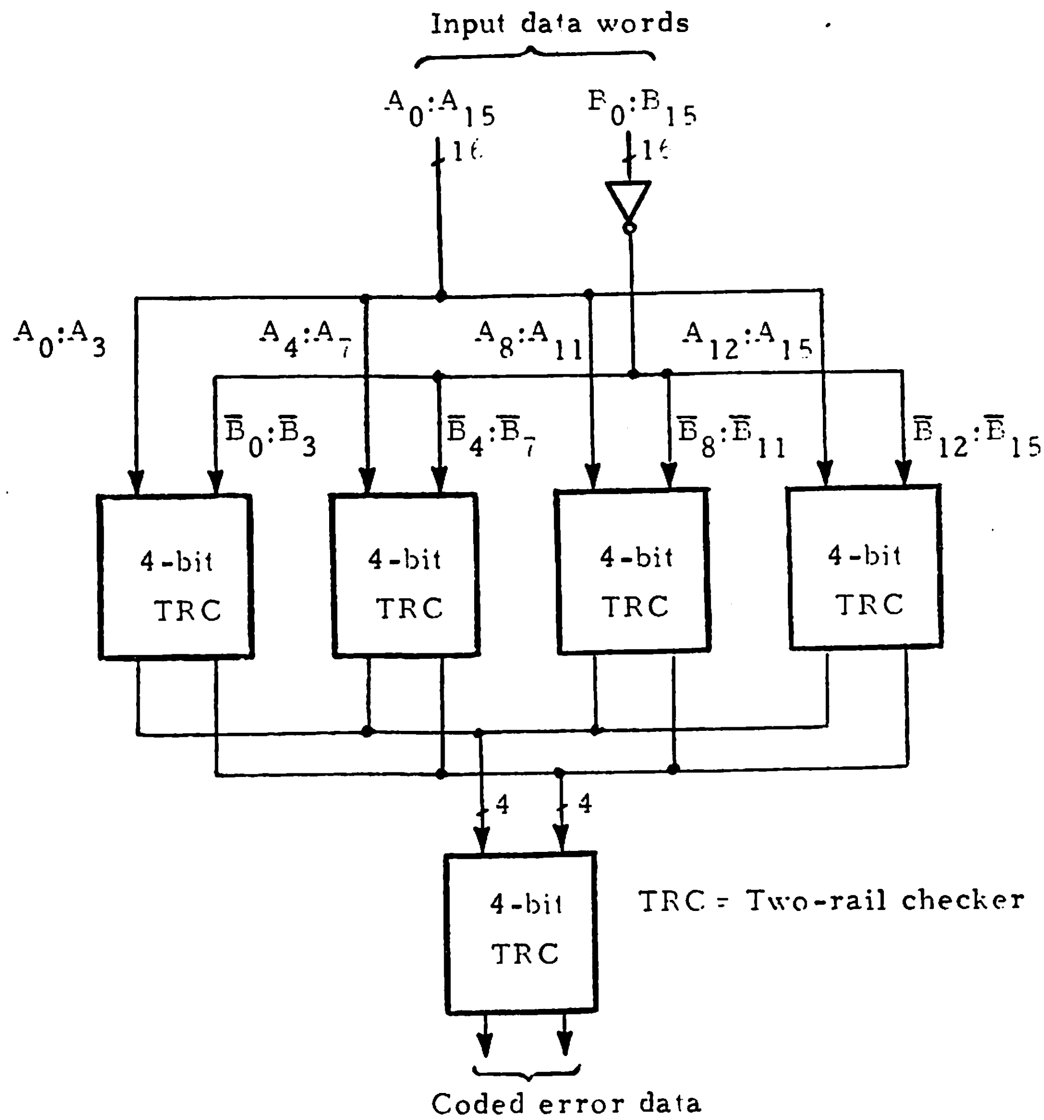


Figure 10.10. A 16-bit self-checking equality checker realized as a tree network of five 4-bit two-rail checkers

lines are stuck-at-zero or stuck-at-one. This includes faults in which multiple input lines are all stuck-at-zero or stuck-at-one, and faults in which many internal lines are stuck at the same logical value. However an SCEC cannot detect a fault consisting of any two lines carrying the 1-out-of-2 code that are stuck at different values, since 01 and 10 are valid codewords. For example, a fault that makes the two output lines of any of the TRC cells of Figure 10.10 stuck at 01 or 10 is not detected. Thus not all stuck-line faults within a single TRC cell of Figure 10.10 are detected. Therefore even the stuck-line fault coverage is not complete in an SCEC. A 16-bit equality checker ILA of Figure 9.7 contains 32 gates, in contrast with the 106 gates used in the 16-bit SCEC of Figure 10.10.

The main checking schemes used in the two CPU designs discussed here are summarized in Table 10.4. On the whole it appears that the amounts of additional logic used in both designs for checking the I-units are almost the same. Note however that more logic is used for checking the E-unit in Wakerly's design. Also the residue code used in the E-unit increases the data path width from 16 to 20 bits throughout Wakerly's computer system.

Table 10.4. Comparison of the checking schemes used in Wakerly's design and in the proposed design

System Component	Wakerly's Design		Proposed Design	
	Checking Scheme	Additional Logic	Checking Scheme	Additional Logic
Processor	Coding (modulo 15 residue code)	4-bit processor check slice 16-bit check register Three 4-bit adders Two 4-bit registers 256×4-bit ROM 4-bit self-checking equality checker (SCEC)	I-testing	One 16-bit equality checker (EC)
Microprogram Sequencer	Duplication and Comparison	An extra copy of the sequencer MS' 16-bit SCEC (MS' includes a next-address ROM and a mapping ROM)	I-testing	12-bit EC for MS 12-bit EC for mapping ROM (MR) 4-bit EC for next-address ROM (NAR) An extra copy of MR An extra copy of NAR
Control Store	Coding (interleaved parity code)	4096×4-bit check symbol ROM Four 12-bit parity generators 4-bit SCEC	Duplication and Comparison	An extra copy of CS 48-bit EC for CS

CHAPTER 11

CONCLUDING REMARKS

In this concluding chapter the principal results presented in this thesis are reviewed. Some possible extensions of this work are also discussed.

11.1 Thesis Summary

We have demonstrated that bit-sliced networks have certain properties which greatly facilitate testing. The relatively low complexity of the individual slices allows a powerful functional fault model to be employed. Unlike the classical line stuck-at-zero/one fault model, this model does not require a detailed gate-level description of the circuit under test; such descriptions are often unavailable in practice. Circuit modules are tested exhaustively, but a network of modules is tested in an efficient non-exhaustive manner. Test data generated according to our fault model has the added advantage of being useful for design verification during the initial logic design of the slice.

The regular array-like interconnections characteristic of bit-sliced systems also simplify test pattern generation. In many cases the tests for a single slice can

easily be extended to tests for a bit-sliced array of arbitrary length, as demonstrated in Chapter 4 for a family of realistic instruction-set processors. Particularly useful in this regard are C-tests which are constant in number and independent of array length. It is reasonable to expect that most bit-sliced processors employing only carry and shift signals for interslice communication will be C-testable. If a bit-sliced ILA is not already C-testable then the design modifications proposed in Sections 6.3, 7.5 and 8.4 can be used to make it C-testable. These procedures add a small amount of extra hardware to the original design.

C-testable combinational ILAs were analyzed in Chapters 5, 6 and 7. New theoretical results including characterizations of C-testable ILAs and procedures to generate small C-test sets were presented. Efficient design methods to make an ILA C-testable were developed. It was shown that these proposed methods use less additional logic and result in fewer C-tests for the modified arrays than previously known methods. In Chapter 8 C-testability in combinational bilateral ILAs and also in a useful class of sequential ILAs was considered. C-testability in more general sequential and bilateral ILAs is yet to be studied.

A new property in ILAs termed I-testability has been introduced which ensures that identical test output responses are obtained from every cell. Some basic theoreti-

cal properties of I-testable ILAs were obtained in Chapter 9. We have shown that complete sequences of I-tests or CI-tests can readily be derived for processor arrays and microprogram sequencers that closely resemble commercial products. I-tests and CI-tests seem particularly well-suited to the design of self-testing systems, since they allow test response verification using simple equality checkers. Thus the overhead required in testing time and built-in test equipment is relatively small. While I-testing can only be used directly in ILA-structured systems, it is a special type of replication-with-comparison testing, where the replication is inherent in the use of multiple slices. As a result, I-testing can readily be combined with conventional self-testing designs that are based on replication and comparison.

11.2 Further Research

An important general approach to simplify the design and testing of complex VLSI chips is to introduce regularity into their design. In recent years regular circuits such as programmable logic arrays (PLAs) and master slices are seeing increased use [Capece 1978, Davis et al. 1980]. Bit slicing is also making an impact on integrated circuit design as indicated by several recent university projects [Mead and Conway 1980, Clark 1980]. Bit slicing

may well become a basic design tool for design engineers. Various practical bit-sliced modules can form useful building blocks in digital IC design. In the future, related regular structures like trees and two-dimensional arrays may also be used to simplify circuit design. Using these structures one can build many practical circuits such as carry-save adders and multipliers, carry-lookahead trees, and multiplier/divider arrays [Advanced Micro Devices 1975, Hwang 1978]. Little is known about the testability of these structures, which therefore might be a fruitful area for further research.

Some specific open research problems related to this thesis are listed below.

1. A test generation methodology for networks of register-level modules was outlined in Section 2.5. This method was used in an ad hoc way to generate complete test sequences for the various processor and microprogram sequencer cells of interest here. To make this test pattern generation process into an algorithm requires formalizing operations that are similar to the line justification and D-drive operations of the classical D-algorithm [Breuer and Friedman 1976]. This is likely to be quite difficult for large random networks of register-level modules. However it appears to be quite tractable for bit-sliced circuits and other circuits having regular structure.

2. Because useful properties like C- and I-testability often exist in bit-sliced circuits, or can readily be introduced, it might be worthwhile to determine what kinds of functions besides addition, subtraction, shifting and logical operations, can be implemented via bit-sliced circuits. Thus there is a need to characterize bit-sliceable switching functions. At a slightly higher level one can also consider the characterization of bit-sliceable algorithms for operations like multiplication and division.

3. We have studied C-testability in some, but by no means all, classes of one-dimensional arrays. For example, Class 3 and Class 4 sequential arrays of Table 5.1 have not yet been considered. Also in the case of combinational arrays without direct outputs, i.e., Class 1 and Class 2 arrays, the problem of functional characterization is still unsolved. In Section 8.5 we have considered only a special class of sequential ILAs. The analysis of more general C-testable sequential ILAs appears to be very hard. This is due to the difficulty of initialization and identification of the internal states of a sequential cell. There is a need for further study in this area.

4. The scope of this thesis has been confined to fault detection. It is also of interest to perform fault location in bit-sliced ILAs. We can define a new property called *C-diagnosability* which enables the location of

faulty cells with a constant number of test patterns. Kautz has made a preliminary study of this problem [Kautz 1967] and, recently Parthasarathy and Reddy have studied some aspects of fault location in combinational unilateral ILAs [Parthasarathy and Reddy 1979]. C-diagnosability is yet to be explored completely. In the same vein, one can also define *I-diagnosability*, which ensures that the faulty cells be located by monitoring the expected identical responses from the cells. The study of C- and I-diagnosability in ILAs may lead to the design of easily diagnosable digital systems. Such systems are very attractive in realizing self-repairable systems.

5. The I-testing approach used in Chapter 10 to design a self-testing bit-sliced CPU takes advantage of the presence of identical slices in the system. Another way of testing a bit-sliced array is to allow every slice to test adjacent slices using a complete test sequence for a single slice. In such cases, obviously there is no need to generate test patterns for the entire bit-sliced array. But each slice should have the capability to generate or store the complete test sequence, and should also be capable of applying the test sequence to the adjacent slices and verifying their responses. The method of every slice testing its neighbors has been studied in the past to a limited extent [Forbes et al. 1965, Ciompi and

Simoncini 1977]. More detailed analysis is needed to determine its practical feasibility.

REFERENCES

Adams, P.M., "Microprogrammable microprocessor survey," *SIGMICRO Newsletter*, vol. 9, no. 1, pp. 23-49, March 1978 (Part 1), and *ibid.*, no. 2, pp. 7-38, June 1978 (Part 2).

Advanced Micro Devices, *Schottky and Low-power Schottky Bipolar Memory, Logic and Interface*, Sunnyvale, California, 1975.

Advanced Micro Devices, *The Am2900 Family Data Book*, Sunnyvale, California, 1979.

Blakeslee, T.R., *Digital Design with Standard MSI and LSI*, 2nd edition, Wiley-Interscience, New York, 1979.

Breuer, M.A., "Fault detection in a linear cascade of identical machines," *Proc. Ninth Symp. on Switching and Automata Theory*, pp. 235-243, 1968.

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

Breuer, M.A. and A.D. Friedman, "Functional level primitives in test generation," *IEEE Trans, Computers*, vol. C-29, no. 3, pp. 223-235, March 1980.

Capece, R.P., "Tackling the very large-scale problems of VLSI: a special report," *Electronics*, vol. 51, no. 2, pp. 111-125, November 23, 1978.

Chiang, A.C.L. and R. McCaskill, "Two new approaches to simplify testing of microprocessors," *Electronics*, vol. 49, no. 2, pp. 100-105, January 22, 1976.

Chu, P., "ECL accelerates to new system speeds with high-density byte-slice parts," *Electronics*, vol. 52, no. 16, pp. 120-125, August, 1979.

Ciompi, P. and L. Simoncini, "Design of self-diagnosable minicomputers using bit-sliced microprocessors," *Journal of Design Automation and Fault Tolerant Computing*, vol. 1, pp. 363-375, October 1977.

Clark, J. "A VLSI geometry processor for graphics," *Computer*, vol. 13, no. 7, pp. 59-68, July 1980.

Clary, J.B. and R.A. Sacane, "Self-testing computers," *Computer*, vol. 12, no. 10, pp. 49-59, October 1979.

Davis, C. et al., "Gate array embodies System/370 processor," *Electronics*, vol. 53, no. 22, pp. 140-143, October 9, 1980.

Dias, F.J.O., "Truth-table verification of an iterative logic array," *IEEE Trans. Computers*, vol. C-25, no. 6, pp. 605-613, June 1976.

Forbes, R.E. et al., "A self-diagnosable computer," *Proc. 1965 Fall Joint Computer Conference*, pp. 1073-1086, 1965.

Friedman, A.D., "Easily testable iterative systems," *IEEE Trans. Computers*, vol. C-22, no. 12, pp. 1061-1064, December 1973.

Friedman, A.D. and P.R. Menon, *Fault Detection in Digital Circuits*, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

Golomb, S.W., *Shift Register Sequences*, Holden-Day, Inc., San Francisco, California, 1967.

Gray, F.G. and R.A. Thompson, "Fault detection in bilateral arrays of combinational cells," *IEEE Trans. Computers*, vol. C-27, no. 12, pp. 1206-1213, December 1978.

Hayes, J.P., "Detection of pattern-sensitive faults in random-access memories," *IEEE Trans. Computers*, vol. C-24, no. 2, pp. 150-157, February 1975.

Hayes, J.P., "Component expansion techniques in computer design," *Digital Processes*, vol. 4, no. 3-4, pp. 295-312, 1978.

Hayes, J.P., "A survey of bit-sliced computer design," *Journal of Digital Systems*, vol. 5, 1981, to appear.

Hayes, J.P. and E.J. McCluskey, "Testability considerations in microprocessor-based design," *Computer*, vol. 13, no. 3, pp. 17-26, March 1980.

Herstein, I.N., *Topics in Algebra*, Xerox College Publishing, Lexington, Massachusetts, 1975.

Hwang, K., *Computer Arithmetic: Principles, Architecture and Design*, John Wiley and Sons, Inc., New York, 1978.

Intel Corp., *Intel Series 3000 Reference Manual*, Santa Clara, California, 1976.

Kautz, W.H., "Testing for faults in combinational cellular logic arrays," *Proc. Eighth Symp. on Switching and Automata Theory*, pp. 161-174, 1967.

Könemann, B. et al., "Built-in logic block observation techniques," *Digest 1979 Test Conference*, Cherry Hill, New Jersey, pp. 37-41, 1979.

Landgraff, R.W. and S.S. Yau, "Design of diagnosable iterative arrays," *IEEE Trans. Computers*, vol. C-20, no. 8, pp. 867-877, August 1971.

Levitt, K.N. et al., "A study of the data commutation problems in a self-repairable multiprocessor," *Proc. 1968 Spring Joint Computer Conference*, pp. 515-527, 1968.

McCaskill, R., "Test approaches for four-bit microprocessor slices," *Digest 1976 Test Conference*, Cherry Hill, New Jersey, pp. 22-26, 1976.

McCluskey, E.J., "Iterative combinational switching networks - general design considerations," *IRE Trans. Electronic Computers*, vol. EC-7, no. 12, pp. 285-291, December 1958.

Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

Motorola Inc., *M10800 High Performance MECL LSI Processor Family*, Phoenix, Arizona, 1976.

Motorola Inc., *MC14500B Industrial Control Unit Handbook*, Phoenix, Arizona, 1977.

Myers, G.J., *Digital System Design with LSI Bit-slice Logic*, Wiley-Interscience, New York, 1980.

National Semiconductor Corp., *GPC/P Product Description*, Pub. No. 4200005B, Santa Clara, California, October 1973.

Parthasarathy, R. and S.M. Reddy, "On fault diagnosis of iterative logic arrays," *Proc. Seventeenth Annual Allerton Conference*, October 1979.

Peterson, W.W. and E.J. Weldon, Jr., *Error-correcting Codes*, MIT Press, Cambridge, Massachusetts, 1972.

Prasad, B.A. and F.G. Gray, "Multiple fault detection in arrays of combinational cells," *IEEE Trans. Computers*, vol. C-24, no. 8, pp. 794-802, August 1975.

Sung, C.H., "Testable sequential cellular arrays," *IEEE Trans. Computers*, vol. C-25, no. 1, pp. 11-18, January 1976.

Tamamoto and H. and S. Takaba, "A design of fault-diagnosable cellular arrays," *Systems, Computers, Control*, vol. 7, no. 5, pp. 105-115, 1976.

Texas Instruments Inc., *The Bipolar Microcomputer Components Data Book*, 2nd edition, Dallas, Texas, 1979.

Thatte, S.M. and J.A. Abraham, "Test generation for microprocessors," *IEEE Trans. Computers*, vol. C-29, no. 6, pp. 429-441, June 1980.

Turcat, C. and A. Verdillon, "Recursion and testing of combinational circuits," *IEEE Trans. Computers*, vol. C-25, no. 6, pp. 652-655, June 1976.

Wadsack, R.L., "Fault modeling and logic simulation of CMOS and MOS integrated circuits," *Bell System Tech. Journal*, vol. 57, no. 5, pp. 1449-1474, May-June 1978.

Wakerly, J., *Error-detecting Codes, Self-checking Circuits and Applications*, North-Holland, New York, 1978.

Wyland, D.C., "Design your own computer by using bipolar/LSI processor slices," *Electronics*, vol. 23, no. 20, pp. 72-78, September 27, 1975.

APPENDIX A

MULTIPLE-MODULE FAULTS IN THE PROCESSOR CELL C

Consider the processor cell C of Figure 3.1. In Section 3.3, a complete test sequence T_C for C was derived under the assumption that at most one module of C is faulty. Here we establish that T_C also detects an important class of multiple-module faults, namely all stuck-line faults on the control lines $I_0, I_1, \ldots, I_9 = I_0{:}I_9$. For example, the line I_0 stuck-at-d denoted I_0^d, where $d \in \{0,1\}$, affects the two modules M_R and M_S. There are two sets of stuck-line faults affecting multiple modules of C, namely, the set F_1 of 3^3-1 stuck-line faults on the control lines I_0, I_1 and I_2 that are common to both M_R and M_S, and the set F_2 of 3^2-1 stuck-line faults on the two lines I_6 and I_7 that are common to M_A and M_T. The remaining five control lines I_3, I_4, I_5, I_8 and I_9 each control only one module of C, and hence any (single) stuck-line fault on them affects only one module of C. Such faults are therefore readily detected by the sequence T_C derived in Section 3.3. It will be shown below that T_C detects all faults belonging to F_1 and F_2.

Let f_1 be the fault in which line I_2 is stuck-at-zero, i.e., $f_1 = I_2^0$. Clearly $f_1 \in F_1$, and f_1 affects both

modules M_R and M_S of C; see Figure 3.1 and Table 3.1. Consider the test pattern t_{15} in the test sequence T'_R of Table 3.3 which completely verifies the truth-tables of both M_R and M_S. t_{15} detects the above fault f_1 since the presence of f_1 changes the Y output response of C to t_{15}. Similarly, all the other 25 faults in the set F_1 are detected by test patterns in T'_R. All 26 faults of F_1 and test patterns in T'_R that detect them are listed in Table A.1.

Now the eight faults in the set F_2 are detected by the test sequence T'_e of Table 3.6. T'_e verifies completely the state tables of the sequential modules M_A and M_T of C. For example, the fault I_6^0 is detected by the test pattern t_{13} in T'_e since the fault-free and faulty Y output signals are 0 and 1 respectively. Table A.2 lists all eight faults of F_2 and test patterns in T'_e that detect them.

Table A.1. Stuck-line faults on control lines I_0, I_1 and I_2 of C and their test patterns

Fault f	Test pattern in T'_R that detects f
I_0^0	t_{12}
I_0^1	t_{11}
I_1^0	t_4
I_1^1	t_2
I_2^0	t_{15}
I_2^1	t_{11}
I_0^0, I_1^0	t_{12}
I_0^0, I_1^1	t_2
I_0^1, I_1^0	t_{11}
I_0^1, I_1^1	t_3
I_1^0, I_2^0	t_4
I_1^0, I_2^1	t_{11}
I_1^1, I_2^0	t_2
I_1^1, I_2^1	t_{13}
I_0^0, I_2^0	t_{12}
I_0^0, I_2^1	t_{11}
I_0^1, I_2^0	t_{11}
I_0^1, I_2^1	t_{12}
I_0^0, I_1^0, I_2^0	t_{12}
I_0^0, I_1^0, I_2^1	t_{11}
I_0^0, I_1^1, I_2^0	t_2
I_0^0, I_1^1, I_2^1	t_{13}
I_0^1, I_1^0, I_2^0	t_{11}
I_0^1, I_1^0, I_2^1	t_{12}
I_0^1, I_1^1, I_2^0	t_3
I_0^0, I_1^1, I_2^1	t_{15}

Table A.2. Stuck-line faults on control lines I_6 and I_7 of C, and their test patterns

Fault f	Test pattern in T'_e that detects f
I_6^0	t_{13}
I_6^1	t_{14}
I_7^0	t_5
I_7^1	t_{12}
I_6^0, I_7^0	t_5
I_6^0, I_7^1	t_{12}
I_6^1, I_7^0	t_5
I_6^1, I_7^1	t_{12}

APPENDIX B

VLSI CHIP IMPLEMENTATION

In this appendix we briefly describe the process used to implement a 4-bit processor array of C-type cells on an NMOS integrated circuit chip. A photomicrograph of this chip appears in Figure 4.3. The design methodology and the SIMULA-based software tools used for generating the chip layout are outlined in [Mead and Conway 1980]. The chip was fabricated in the 1980 Multiproject Chip Set (MPC580) project.

First the 1-bit cell C of Figure 3.1 was designed and then replicated four times to realize the 4-bit array of Figure 4.1. To simplify the circuit design and layout process, a small number of easily implemented primitive cells or macros were identified, and used repeatedly to realize the register-level modules of C. Three kinds of macros, namely, pass transistors, inverters and 2:1 multiplexers were defined for this purpose. In addition, standard library macros called superbuffers [Mead and Conway 1980] were used to amplify signals driving long lines or several gates. Library macros were also used for the input-output pads and their associated circuits.

Figure B.1 depicts the circuit layout of the three macros employed to design C. The 2:1 multiplexer macro was used to construct various larger multiplexers such as M_R and M_S, and also to realize the two-variable Boolean switching functions used to implement the ALU module M_F. The "Manchester" carry-chain technique [Mead and Conway 1980, pp. 150-151] was employed to speed up carry propagation through the ALU modules of the processor array.

We next show how a representative module of C, the shifter M_L, is realized; the remaining modules are constructed in a similar way. M_L is functionally a 3:1 multiplexer with RI, F and LI as its inputs and L as its output; see Figure 3.1. The control lines I_8 and I_9 specify the source data to be selected by M_L. Therefore M_L is easily constructed using three 2:1 multiplexers as demonstrated in Figure B.2. The control lines I_8, $\bar{I}_8$, I_9 and $\bar{I}_9$ are obtained from the input pads of the chip via inverting and non-inverting superbuffers.

The modules of C are laid out in approximately the way they appear in the block diagram of Figure 3.1. The horizontal lines of the processor array, namely, the control lines I_0:I_9, the carry line and the shift lines, are implemented by horizontal metal lines on the chip; see Figure 4.3. The signals from the external world entering via the input pads are all buffered using appropriate superbuffers. The clock circuit is implemented by a two-phase clock and

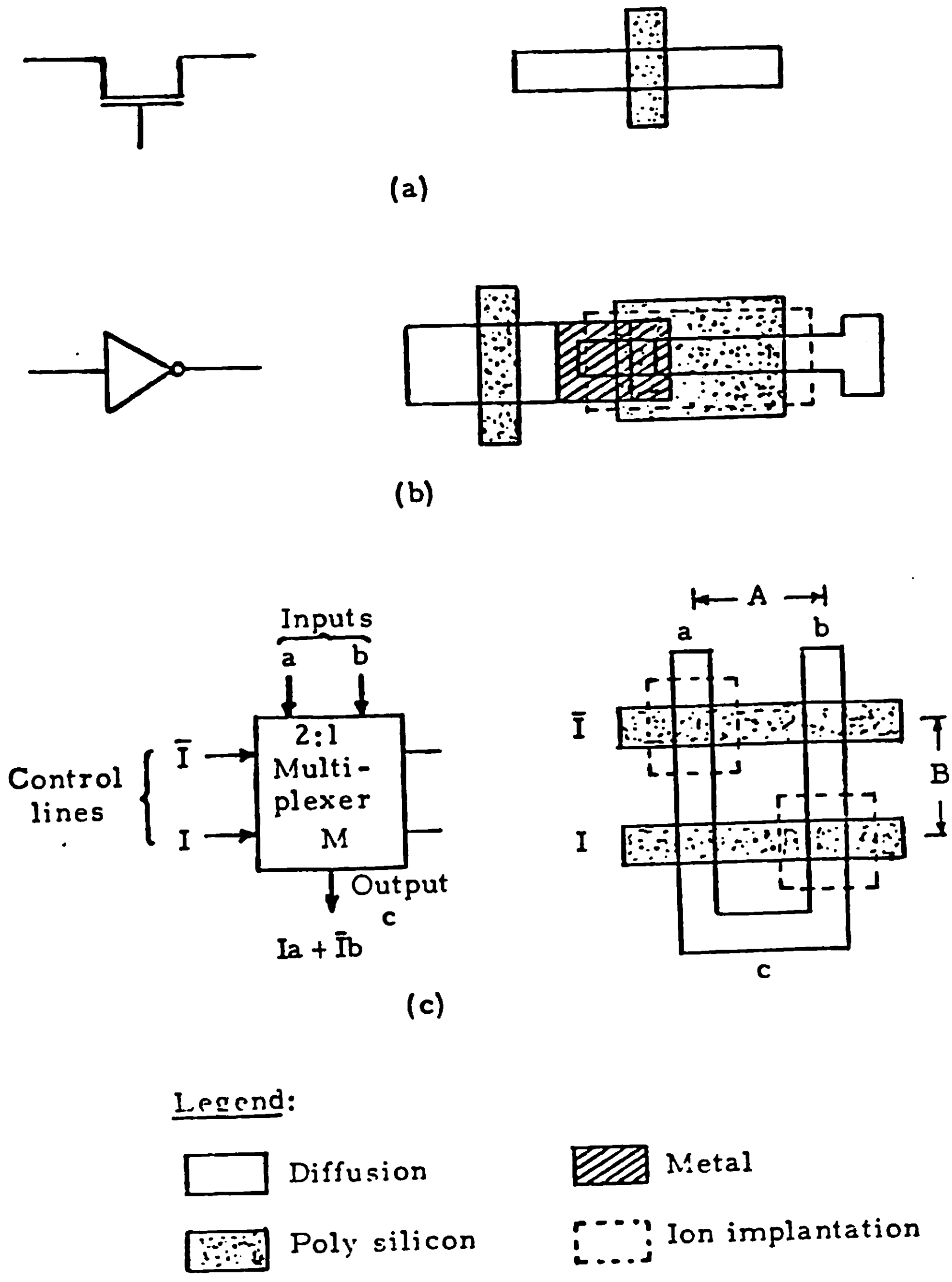


Figure B.1. Circuit representation and layout of the three basic macros: (a) pass transistor, (b) inverter and (c) 2:1 multiplexer M

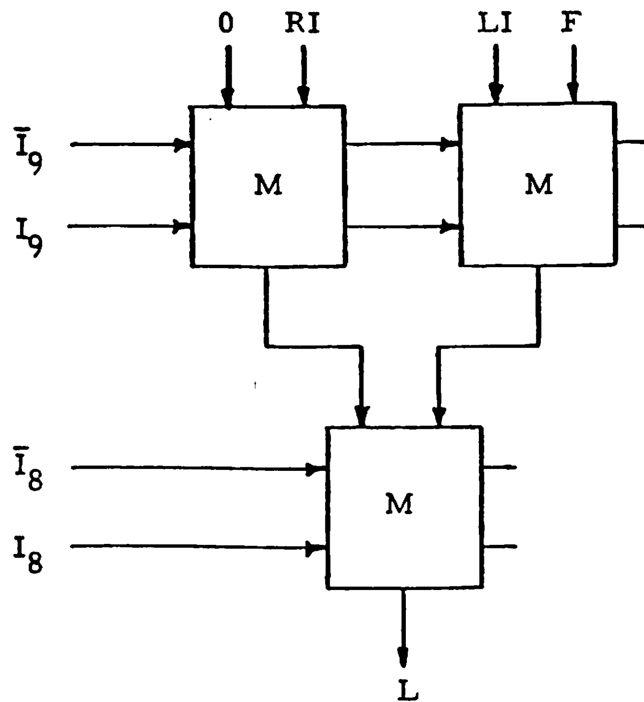


Figure B.2. Realization of the shifter module M_L of the processor cell C using the 2:1 multiplexer macro M

controls the transfer of data into the two sequential modules M_A and M_T of C. The complete chip layout appears in Figure 4.3.

Some of the SIMULA macro definitions and procedures used in generating the chip layout are given in Figures B.3, B.4 and B.5. These procedures use layout primitives such as WIRE and BOX, provided by the Caltech VLSI software [Mead and Conway 1980]. In Figure B.3, the inverter macro of Figure B.1b with two different sizes of pull-up transistor is defined. Procedures for generating the 2:1 multiplexer macro M of Figure B.1c, and for generating a row of two such multiplexers with common control lines are given in Figure B.4. The shifter module M_L is defined in Figure B.5. This SIMULA definition makes use of the SIMULA procedures of Figure B.4, and generates the circuit depicted in Figure B.6. This layout corresponds to the logic diagram of M_L given earlier in Figure B.2.

```
! The following two SIMULA definitions implement the
  inverter macro of Figure B.lb for two different
  sizes of the pull-up transistor ;

  ! INVERTER (k=8) CELL ; ! k is the ratio of L/W of
                            pull-up transistor to L/W
                            of pull-down transistor;

  DEFINE("INVERTER_K8"); ! Pull-up L/W = 4/1,
                           pull-down L/W = 1/2;
    LAYER(GREEN); ! Diffusion;
      WIRE(4,0,0).X(4); WIRE(2,11,0).X(20):
      BOX(21,-2,23,2);
    LAYER(RED); ! Poly silicon;
      BOX(2,-4,4,4); BOX(9,-3,18,3);
    LAYER(IMPLANT);
      BOX(7.5,-2.5,19.5,2.5),
    BE(7,0); ! Green-red butting contact facing east;
  ENDDEF: ! INVERTER_K8;

  ! INVERTER (k=4) CELL ;

  DEFINE("INVERTER_K4"); ! Pull-up L/W = 2/1,
                           pull-down L/W = 1/2;
    LAYER(GREEN);
      WIRE(4,0,0).X(4); WIRE(2,11,0).X(17);
      BOX(18,-2,20,2);
    LAYER(RED);
      BOX(2,-4,4,4); BOX(9,-3,15,3);
    LAYER(IMPLANT);
      BOX(7.5,-2.5,16.5,2.5);
    BE(7,0); ! Green-red butting contact facing east;
  ENDDEF; ! INVERTER_K4;
```

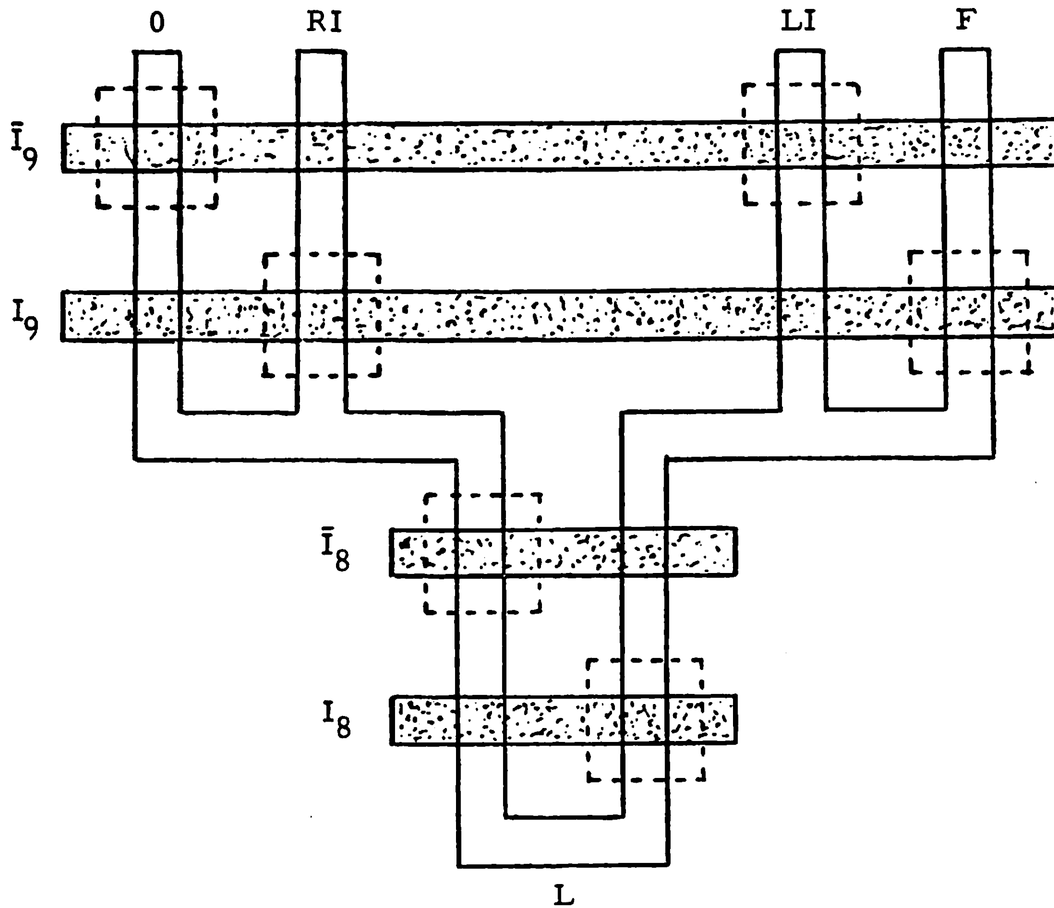
Figure B.3. SIMULA definition for the inverter macro of Figure B.1b

```
! The following SIMULA procedure generates the 2:1
  multiplexer macro M of Figure B.lc ;

  PROCEDURE DRAW_MUX2(X,Y,A,B);
    VALUE X,Y,A,B; REAL X,Y,A,B; ! (X,Y) is the origin,
                                   A and B are the
                                   horizontal and ver-
                                   tical spacings in
                                   the multiplexer,
                                   see Figure B.lc;
  BEGIN
    LAYER(GREEN);
      WIRE(2,X,Y+B+8).Y(Y).X(X+A).Y(Y+B+8);
    LAYER(RED);
      WIRE(2,X-3,Y+B+5).X(X+A+3);
      WIRE(2,X-3,Y+5).X(X+A+3);
    LAYER(IMPLANT);
      BOX(X-2.5,Y+B+2.5,X+2.5,Y+B+7.5);
      BOX(X+A-2.5,Y+2.5,X+A+2.5,Y+7.5);
  END OF DRAW_MUX2;


! The following procedure is for drawing two multiple-
  xer macros in a row with a spacing of 3A;

  PROCEDURE TWO_MUX2(X,Y,A,B);
    VALUE X,Y,A,B; REAL X,Y,A,B;

  BEGIN
    DRAW_MUX2(X,Y,A,B);
    DRAW_MUX2(X+4*A,Y,A,B);
    LAYER(RED); ! To draw the poly lines between the
                  two multiplexers;
      BOX(X+A+4,Y+4,X+4*A-4,Y+6);
      BOX(X+A+4,Y+B+4,X+4*A-4,Y+B+6);
  END OF TWO_MUX2;
```

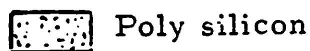
Figure B.4. SIMULA procedures for generating the 2:1 multiplexer macro M for Figure B.lc and a row of Ms

```
! The following procedure realizes the shifter module
  M_L of the processor cell C ;

  ! Define the shifter ;

  DEFINE("SHIFTER");
    A:= 10; B:= 7; ! Spacings in the multiplexers;
    OX:= 0; OY:= 0; ! Origin;

    OX:= OX + A; OY:= OY + 15;
    DRAW_MUX2(OX,OY,A,B); ! Bottom multiplexer, see
                            Figure B.2;
    OX:= OX - 2*A; OY:= OY+B+10;! Shift origin;
    TWO_MUX2(OX,OY,A,B);! Top row of multiplexers;

    ! Connections between the two rowsof multiplexers;

    LAYER(GREEN);
      WIRE(2,OX+A,OY).X(OX+2*A);
      WIRE(2,OX+4*A,OY).X(OX+3*A);

  ENDDEF;  ! SHIFTER;
```

Figure B.5. SIMULA definition for realizing the shifter module M_L of Figure B.2



Legend:



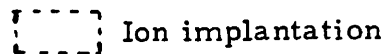


Figure B.6. Actual circuit layout of the shifter module M_L generated by the SIMULA definition of Figure B.5

APPENDIX C

PROOF OF THEOREM 6.5

Theorem 6.5 states that a combinational unilateral ILA without direct outputs that uses the modified flow table of Figure 6.5 is C-testable. We prove this result by construction, i.e., we construct C-tests for every transition of the modified cell flow table F′ of Figure 6.5a. Consider any transition τ: $x_i \xrightarrow{y_j} x_c$ of the original flow table F. We now construct an input sequence R for the flow table F′ that satisfies the two requirements for C-testability specified in Theorem 6.1. For this purpose we make use of the new columns y_m and y_{m+1} only. Let x_k be the faulty state to be propagated by R to the $\hat{X}$ output of the rightmost cell of the modified array, where $k \neq c-1$ and $k \neq n$. First x_k is transferred to the new state x_n which forms a trap state, since $\hat{x}(x_n, y_j) = x_n$ in all columns of F′, except y_{m+1}. To do this, the input y_m is repeated a times, where $a \equiv n-k \pmod{n}$, such that x_k is transferred to x_0, and then y_{m+1}^2, i.e., y_{m+1} repeated twice, will take x_0 to x_n. At the same time, the input sequence $y_m^a y_{m+1}^2$ transfers the fault-free state x_c to x_{n-1}. Next an input sequence y_m^b, where $b = i+1$, is used to trans-

fer x_{n-1} back to the initial state x_i of τ and the faulty state is trapped in state x_n. Thus the desired input sequence R is $y_m^a y_{m+1}^2 y_m^b$, and the corresponding C-test is $x_i, (y_j R)^*$, as indicated in Table C.1. The remaining two C-tests for the transitions in Group 1 are for the cases where $k = c-1$ and $k = n$, respectively.

In the foregoing manner R input sequences and C-tests for the remaining transitions of F′ can be constructed. Table C.1 gives possible C-tests for all transitions of F′. Hence an array of modified cells with the flow table of Figure 6.5a is C-testable.

The upper bounds, and in some cases the exact value, for the number of C-tests required by the transitions of F′ have been derived and appear in Table C.1. An upper bound UB_3 for the total number of tests needed to test a modified array is given by

$$UB_3 = mn^3 + \left(\frac{7}{2}m+2\right)n^2 + \left(14 - \frac{5}{2}m\right)n+m-17$$

as indicated in Table 6.1 of Section 6.3.

Table C.1. C-tests for the modified flow table F′ of Figure 6.5a

Group No.	Transition τ_{ij}		C-tests for τ_{ij}	Number of Tests Required for all Transitions in the Group
	row i	column j		
1	$0 \le i \le n-1$	$0 \le j \le m-1$	$x_i,(y_j y_m^a y_{m+1}^2 y_m^b)^*$ $x_i,(y_j y_m^a y_{m+1} y_m y_{m+1}^2 y_m^b)^*$ $x_i,(y_j y_m^a)^*$	$\le mn[3(n-1)+\frac{n(n-1)}{2}-1+\frac{(n-1)(n-2)}{2}]$ $\le mn[n+4+\frac{n-1}{2}]$ $\le mn^2$
2	n	$0 \le j \le m-1$	x_n,y_j^*	m
3	$0 \le i \le n$	m	x_i,y_m^*	n+1
4	$2 \le i \le n-1$	m+1	$x_i,(y_{m+1}^2 y_m^a)^*$ $x_i,(y_{m+1} y_m^3 y_{m+1} y_m^a)^*$	$2(n-2)+\frac{n(n-1)}{2}-3$ $5(n-2)+\frac{n(n-1)}{2}-3$
5	0	m+1	$x_0,(y_{m+1} y_m^a y_{m+1} y_m^2 y_{m+1} y_m)^*$	$6(n-1)+\frac{n(n-1)}{2}-1$
6	1	m+1	$x_1,(y_{m+1} y_m^3 y_{m+1} y_m^3)^*$	8
7	n	m+1	$x_n,(y_{m+1} y_m^a y_{m+1})^*$ $x_n,(y_{m+1} y_m^3 y_{m+1} y_m^3)^*$	$2(n-2)+\frac{n(n-1)}{2}-3$ 8

APPENDIX D

TEST PATTERNS FOR THE MICROPROGRAM SEQUENCER CELL S

In Section 10.3 a 1-bit microprogram sequencer cell S was introduced and test pattern generation for S was discussed. In this appendix complete test sequences for all the modules of S (see Figure 10.1) are given. Tables D.1, D.2, D.3 and D.4 give test sequences for modules M_L and M_A, the microprogram counter M_{PC}, the stack pointer SP, and the stack registers, respectively.

Table D.1. Test sequence for the modules M_L and M_A of the microprogram sequencer cell S

No.	Mask Control		Mux. Control		Stack Control		Data In	State of SP	Stack Output	Counter Output	Mux. Output	Data Out
	OR	$\overline{ZERO}$	I_1	I_0	$\overline{FE}$	PUSH	D		F	PC	L	Y
1	0	1	1	1	1	d	1	i	d	d	1	1
2	0	1	1	1	0	1	0	i	d	1	0	0
3	0	1	1	1	0	1	1	i+1	1	0	1	1
4	0	1	1	1	0	1	0	i+2	0	1	0	0
5	0	1	1	1	0	1	0	i+3	1	0	0	0
6	0	1	1	1	1	d	0	i	0	0	0	0
7	0	1	1	1	1	d	1	i	0	0	1	1
8	0	1	1	1	1	d	1	i	0	1	1	1
9	0	1	1	1	0	0	0	i	0	1	0	0
10	0	1	1	1	1	d	0	i-1	1	0	0	0
11	0	1	1	1	1	d	1	i-1	1	0	1	1
12	0	1	1	1	1	d	1	i-1	1	1	1	1
13	0	1	1	1	0	0	0	i-1	1	1	0	0
14	0	1	0	0	1	d	0	i-2	0	0	0	0
15	0	1	0	0	0	0	1	i-2	0	0	0	0
16	0	1	0	0	1	d	0	i-3	1	0	0	0
17	0	1	0	0	1	d	1	i-3	1	0	0	0
18	0	1	1	0	1	d	0	i-3	1	0	1	1
19	0	1	1	0	0	0	1	i-3	1	1	1	1
20	0	1	0	0	1	d	0	i	0	1	1	1
21	0	1	0	0	0	0	1	i	0	1	1	1
22	0	1	0	0	1	d	0	i-1	1	1	1	1
23	0	1	0	0	1	d	1	i-1	1	1	1	1
24	0	1	1	0	0	0	0	i-1	1	1	1	1
25	0	1	1	0	0	0	0	i-2	0	1	0	0
26	0	1	1	0	0	0	1	i-3	1	0	1	1
27	0	1	1	0	1	d	1	i	0	1	0	0
28	0	1	1	0	1	d	0	i	0	0	0	0
29	0	1	1	0	1	d	1	i	0	0	0	0
30	0	1	0	1	1	d	1	i	0	0	1	1
31	0	1	0	1	1	d	0	i	0	1	1	1
32	0	1	0	1	1	d	1	i	0	1	0	0
33	0	1	0	1	0	0	0	i	0	0	0	0
34	0	1	0	1	1	d	1	i-1	1	0	1	1
35	0	1	0	1	1	d	0	i-1	1	1	1	1
36	0	1	0	1	1	d	1	i-1	1	1	0	0
37	0	1	0	1	1	d	0	i-1	1	0	0	0
38	0	0	1	1	1	d	0	i-1	1	0	0	0
39	1	0	1	1	1	d	0	i-1	1	0	0	0
40	0	1	1	1	1	d	0	i-1	1	0	0	0
41	1	1	1	1	1	d	0	i-1	1	0	0	1
42	0	0	1	1	1	d	1	i-1	1	1	1	0
43	1	0	1	1	1	d	1	i-1	1	0	1	0
44	0	1	1	1	1	d	1	i-1	1	0	1	1
45	1	1	1	1	1	d	1	i-1	1	1	1	1

Note: In all cases CI = CO = 0.

Table D.2. Test sequence for the microprogram counter M_{PC} of the microprogram sequencer cells

No.	Mask Control		Mux. Control		Stack Control		Carry In	Data In	Cell Outputs	
	OR	$\overline{ZERO}$	I_1	I_0	$\overline{FE}$	PUSH	$\dot{C}I$	D	Y	$C\dot{O}$
1	0	1	1	1	1	d	0	0	0	0
2	0	1	0	1	1	d	0	0	0	0
3	0	1	0	1	1	d	0	1	1	0
4	0	1	0	1	1	d	1	0	1	1
5	0	1	0	1	1	d	0	1	1	0
6	0	1	0	0	1	d	0	d	1	0
7	0	1	0	1	1	d	1	1	0	0

Table D.3. Test sequence for the stack pointer SP of the stack ST of the microprogram sequencer cell S

No.	Mux. Control		Stack Control		Data In	State of	Stack Output	Counter Output	Data Out
	I_1	I_0	$\overline{FE}$	PUSH	D	SP	F	PC	Y
1	1	1	0	0	0	00	d	d	0
2	1	1	0	1	0	11	d	0	0
3	1	1	0	1	1	00	0	0	1
4	1	1	0	1	1	01	0	1	1
5	1	1	0	1	0	10	1	1	0
6	1	0	0	0	d	11	1	0	1
7	1	0	0	0	d	10	1	1	1
8	1	0	0	0	d	01	0	1	0
9	1	0	0	0	d	00	0	0	0
10	1	0	0	0	d	11	1	0	1
11	1	0	0	0	d	10	1	1	1
12	1	0	1	0	d	01	0	1	0
13	1	0	0	0	d	01	0	0	0
14	1	0	0	0	d	00	0	0	0
15	1	0	1	0	d	11	1	0	1
16	1	0	0	0	d	11	1	1	1
17	1	0	0	0	d	10	1	1	1
18	1	0	0	0	d	01	0	1	0
19	1	0	1	0	d	00	0	0	0
20	1	0	0	0	d	00	0	0	0
21	1	0	0	0	d	11	1	0	1
22	1	0	1	0	d	10	1	1	1
23	1	0	0	0	d	10	1	1	1
24	1	0	0	0	d	01	0	1	0
25	1	0	1	1	d	00	0	0	0
26	1	0	0	0	d	00	0	0	0
27	1	0	0	0	d	11	1	0	1
28	1	0	1	1	d	10	1	1	1
29	1	0	0	0	d	10	1	1	1
30	1	0	0	0	d	01	0	1	0
31	1	0	0	0	d	00	0	0	0
32	1	0	1	1	d	11	1	0	1
33	1	0	0	0	d	11	1	1	1
34	1	0	0	0	d	10	1	1	1
35	1	0	1	1	d	01	0	1	0
36	1	0	0	0	d	01	0	0	0
37	1	0	0	0	d	00	0	0	0
38	1	0	0	1	d	11	1	0	1
39	1	0	0	0	d	00	0	1	0
40	1	0	0	0	d	11	1	0	1
41	1	0	0	1	d	10	1	1	1
42	1	0	0	0	d	11	1	1	1
43	1	0	0	0	d	10	1	1	1
44	1	0	0	1	d	01	0	1	0
45	1	0	0	0	d	10	1	0	1
46	1	0	0	0	d	01	0	1	0
47	1	0	0	1	d	00	0	0	0
48	1	0	0	0	d	01	0	0	0
49	1	0	0	0	d	00	0	0	0

Note: In all cases $(CI, OR, \overline{ZERO}, CO) = (0,0,1,0)$

Table D.4. Checking sequence CS_i for the memory cells of the stack ST of the microprogram sequencer cell S

No.	Mux. Control I_1	I_0	Stack Control $\overline{FE}$	PUSH	Data In D	State of SP	Stack Output F	Counter Output PC	Data Out Y
1	1	1	1	d	0	i	d	d	0
2	1	1	0	1	0	i	d	0	0
3	1	1	0	1	0	i+1	0	0	0
4	1	1	0	1	0	i+2	0	0	0
5	1	1	0	1	0	i+3	0	0	0
6	1	0	1	d	d	i	0	0	0
7	1	0	1	0	d	i	0	0	0
8	1	0	0	0	d	i	0	0	0
9	1	0	0	0	d	i-1	0	0	0
10	1	0	0	0	d	i-2	0	0	0
11	1	0	0	0	d	i-3	0	0	0
12	1	0	1	1	d	i	0	0	0
13	1	0	0	0	d	i	0	0	0
14	1	0	0	0	d	i-1	0	0	0
15	1	0	0	0	d	i-2	0	0	0
16	1	0	0	0	d	i-3	0	0	0
17	1	1	1	d	1	i	0	0	1
18	1	0	1	0	d	i	0	1	0
19	1	0	0	0	d	i	0	0	0
20	1	0	0	0	d	i-1	0	0	0
21	1	0	0	0	d	i-2	0	0	0
22	1	0	0	0	d	i-3	0	0	0
23	1	1	1	d	1	i	0	0	1
24	1	0	1	1	d	i	0	1	0
25	1	0	0	0	d	i	0	0	0
26	1	0	0	0	d	i-1	0	0	0
27	1	0	0	0	d	i-2	0	0	0
28	1	0	0	0	d	i-3	0	0	0
29	1	0	0	1	d	i	0	0	0
30	1	0	0	0	d	i+1	0	0	0
31	1	0	0	0	d	i	0	0	0
32	1	0	0	0	d	i-1	0	0	0
33	1	0	0	0	d	i-2	0	0	0
34	1	1	0	0	1	i-3	0	0	1
35	1	0	0	0	d	i	0	1	0
36	1	0	0	0	d	i-1	0	0	0
37	1	0	0	0	d	i-2	0	0	0
38	1	0	0	0	d	i-3	0	0	0
39	1	0	0	0	d	i	0	0	0
40	1	1	0	1	1	i-1	0	0	1
41	1	0	0	1	d	i	0	1	0
42	1	0	0	0	d	i+1	1	0	1
43	1	0	0	0	d	i	0	1	0
44	1	0	0	0	d	i-1	0	0	0
45	1	0	0	0	d	i-2	0	0	0
46	1	1	1	d	1	i-3	1	0	1

Note: In all cases $(OR, \overline{ZERO}, CI, CO) = (0,1,0,0)$.

Table D.4. Continued

No.	Mux. Control I_1	I_0	Stack Control $\overline{FE}$	PUSH	Data In D	State of SP	Stack Output F	Counter Output PC	Data Out Y
47	1	0	0	1	d	i-3	1	1	1
48	1	0	0	1	d	i-2	1	1	1
49	1	0	0	1	d	i-1	1	1	1
50	1	0	1	0	d	i	1	1	1
51	1	0	0	0	d	i	1	1	1
52	1	0	0	0	d	i-1	1	1	1
53	1	0	0	0	d	i-2	1	1	1
54	1	0	0	0	d	i-3	1	1	1
55	1	0	1	1	d	i	1	1	1
56	1	0	0	0	d	i	1	1	1
57	1	0	0	0	d	i-1	1	1	1
58	1	0	0	0	d	i-2	1	1	1
59	1	0	0	0	d	i-3	1	1	1
60	1	1	1	d	0	i	1	1	0
61	1	0	1	0	d	i	1	0	1
62	1	0	0	0	d	i	1	1	1
63	1	0	0	0	d	i-1	1	1	1
64	1	0	0	0	d	i-2	1	1	1
65	1	0	0	0	d	i-3	1	1	1
66	1	1	1	d	0	i	1	1	0
67	1	0	1	1	d	i	1	0	1
68	1	0	0	0	d	i	1	1	1
69	1	0	0	0	d	i-1	1	1	1
70	1	0	0	0	d	i-2	1	1	1
71	1	0	0	0	d	i-3	1	1	1
72	1	0	0	1	d	i	1	1	1
73	1	0	0	0	d	i+1	1	1	1
74	1	0	0	0	d	i	1	1	1
75	1	0	0	0	d	i-1	1	1	1
76	1	0	0	0	d	i-2	1	1	1
77	1	1	0	0	0	i-3	1	1	0
78	1	0	0	0	d	i	1	0	1
79	1	0	0	0	d	i-1	1	1	1
80	1	0	0	0	d	i-2	1	1	1
81	1	0	0	0	d	i-3	1	1	1
82	1	0	0	0	d	i	1	1	1
83	1	1	0	1	0	i-1	1	1	0
84	1	0	0	1	d	i	1	0	1
85	1	0	0	0	d	i+1	0	1	0
86	1	0	0	0	d	i	1	0	1
87	1	0	0	0	d	i-1	1	1	1
88	1	0	0	0	d	i-2	1	1	1
89	1	1	1	d	0	i-3	0	1	0
90	1	0	0	1	d	i-3	0	0	0
91	1	0	0	1	d	i-2	0	0	0
92	1	0	0	1	d	i-1	0	0	0