On-Chip Controller Design for Built-in-Test[1]

Melvin A. Breuer
Department of Electrical Engineering-System
University of Southern California
Los Angeles, CA 90089-0781

Technical Report CRI-88-04
December 1985

[1]This work was supported in part by the Electro-Optical and Data Systems Group, Hughes Aircraft Company.

Contents

1 INTRODUCTION 1

2 BACKGROUND 1

3 TEST METHODOLOGIES 2

3.1 Set-Scan with Stored Vectors TDM 2

3.2 Pseudo Random BIT TDM 10

3.3 Exhaustive BIT TDMS 16

4 CONTROL UNIT DESIGN 16

4.1 Microprogram Control Unit Design 18

4.2 Hardwired Control Unit Designs 21

5 SUMMARY OF RESULTS 28

6 REFERENCES 28

Abstract

Many VLSI circuits are designed to be self-testing using one or more built-in-test techniques such as (1) set-scan registers and stored test patterns; (2) exhaustive tests produced by linear feedback shift registers and response compression using signature analysis; and (3) pseudo random tests and signature analysis. When applying these techniques to a circuit a test schedule is required which specifies how the test process is to be carried out in terms of the control signals issued to the circuit being tested and its associated built-in-test hardware. This paper deals with the design of the controller which issues these control signals. The controller is assumed to be implemented as part of a chip, though not necessarily on the chip being tested.

Two controller designs are presented – one a microprogram architecture, the other a hard-wired circuit. Both designs are quite simple, though the latter one appears to require less circuitry.

1 INTRODUCTION

This paper deals with the design of controllers for built-in-test (BIT) circuitry used in VLSI circuits. Before the design of these controllers is presented we will briefly review the types of BIT structures to be controlled and derive the requirements for the control circuitry. We will focus on three common test design methodologies ($TDMs$) [4], namely a scan-set design using precomputed test vectors, a random test vector approach, and an exhaustive testing approach for combinational logic. The latter two methods use linear feedback shift registers to generate test vectors as well as to calculate fault signatures. A good survey of self-testing techniques appears in [1], and some aspects of an on-chip monitor for partially controlling the test process appear in [2].

2 BACKGROUND

There are four related basic concepts associated with a test methodology, namely (1) test schemas, (2) test plans, (3) test schedules, and (4) I-paths. These concepts are discussed in some detail in [3, 4] and are briefly reviewed here for the sake of completeness. A *test schema* specifies how a test methodology is to execute. It can be represented in various forms, such as a state diagram, control graph or program. The main aspects specified by a test schema are: (1) the generation of test vectors, (2) the transfer of test vectors to the part of the circuit to be tested, referred to as the *kernel*, (3) the propagation of test data through the kernel, (4) the transfer of response data from the output of the kernel to a response evaluator or observation circuit, and (5) the processing of the response data.

When a test methodology is employed in an actual circuit many minor transformations may occur to the test schema. For example, the number of random vectors to generate is a function of the circuit to be tested. Also, there may be many ways to transfer data from the test vector generator to the kernel, and from the kernel to the response evaluator. These transfer paths are called identity-paths, or I-paths, since the data being transferred does not get transformed. Other forms of transfer paths are also possible. The result of this *mapping* or *embedding* of a test methodology into an actual circuit produces what we refer to as a *test plan* for the circuit. A test plan specifies how the test methodology is to be executed in the given circuit. In some cases, the processing of test vectors can be pipelined through a circuit, thus reducing total test time. A test plan which has been optimized to reduce test time is called a *test schedule*. From a test schedule, a control sequence for the BIT structure can be easily derived. From this control sequence, the controller for the BIT circuitry can be synthesized. Note that test schemas, test plans and test schedules are all very similar and can be represented in similar ways. Test schemas indicate potential parallelism, while test strategies indicate all actual concurrent operations.

3 TEST METHODOLOGIES

3.1 Set-Scan with Stored Vectors TDM

The first test methodology to be discussed is set-scan using stored test vectors. There are numerous variations of this TDM, known by such names as LSSD and scan-path. We will restrict our attention to where the kernel is a combinational circuit.

One form of the structure of a set-scan TDM is shown in Figure 1(a). Here $X1$ and $X2$ are controllable inputs; $Y1$ and $Y2$ are observable outputs; the kernel $C1$ is a combinational circuit; $R1$ is a clocked register having the following modes of operation: reset (R), parallel latch (L), and shift right (SR). In the normal mode, $R1$ operates as a latch register, and $C1$ and $R1$ together form a classical sequential circuit. In the test mode, $R1$ is placed in the shift register mode so that test data can be shifted into $R1$, and response data shifted out of $R1$. Assume the test set for $C1$ consists of K stored test vectors. Then one form of a test schema for this methodology is shown in Figure 2.

Briefly, the test schema operates as follows. There are three parts, namely a head and tail which are executed only once, and a body which is executed $(K-1)$ times. The first step in the head is to shift a test vector into $R1$. This requires n shift operations, hence one internal requirement for the controller is to count off the n shift operations. The interface between the BIT structure and the controller contains the shift control line $SR(R1)$ which is activated during this time. The actual test data is input from line $X2$, and this is denoted by $I : X2$. The second step in the head specifies that the parallel input data, which is still part of the first test vector, is to be read from input lines $X2$. The rest of the test schema is self explanatory.

Some comments on this test schema are required. First, several operations can be done in parallel, e.g., steps 1 and 2, as well as 3 and 4 in the head. Also, if we ignore the first output vector from $R1$, and the last input vector put into $R1$, then a test schema with just a body and no head or tail can be derived. Finally, an interface protocol between the external environment and the controller may be required so that the input and output operations at $X1$, $X2$, $Y1$, $Y2$ are synchronized.

Figure 3 indicates a control graph version of the test schema for the set-scan TDM. Nodes denoted by an f, such as node 1, are fork operators and essentially are used to initiate two potentially parallel operations. Node A is a state which occurs for n consecutive clock times. Node B is a state which occurs for just one clock time. While in a state, certain operations occur, e.g., when in state A register $R1$ is shifted right. These operations are indicated below the states. Nodes denoted by a j, such as node 2, represent a join operator and indicate that processing cannot continue until both states A and B have executed. The execution of the control graph remains in the initial state J until a pulse occurs on the $START$ line.

If the controller need not deal with the I/O operations, then the control graph of Figure 3 can be simplified, as shown in Figure 4(a). Finally, ignoring the head and tail, or buffering the corresponding test set with a pseudo first output vector and last input vector, we get the still simpler control graph shown in Figure 4(b).

Note that there are several different set-scan structures, each requiring a slightly different test schema.

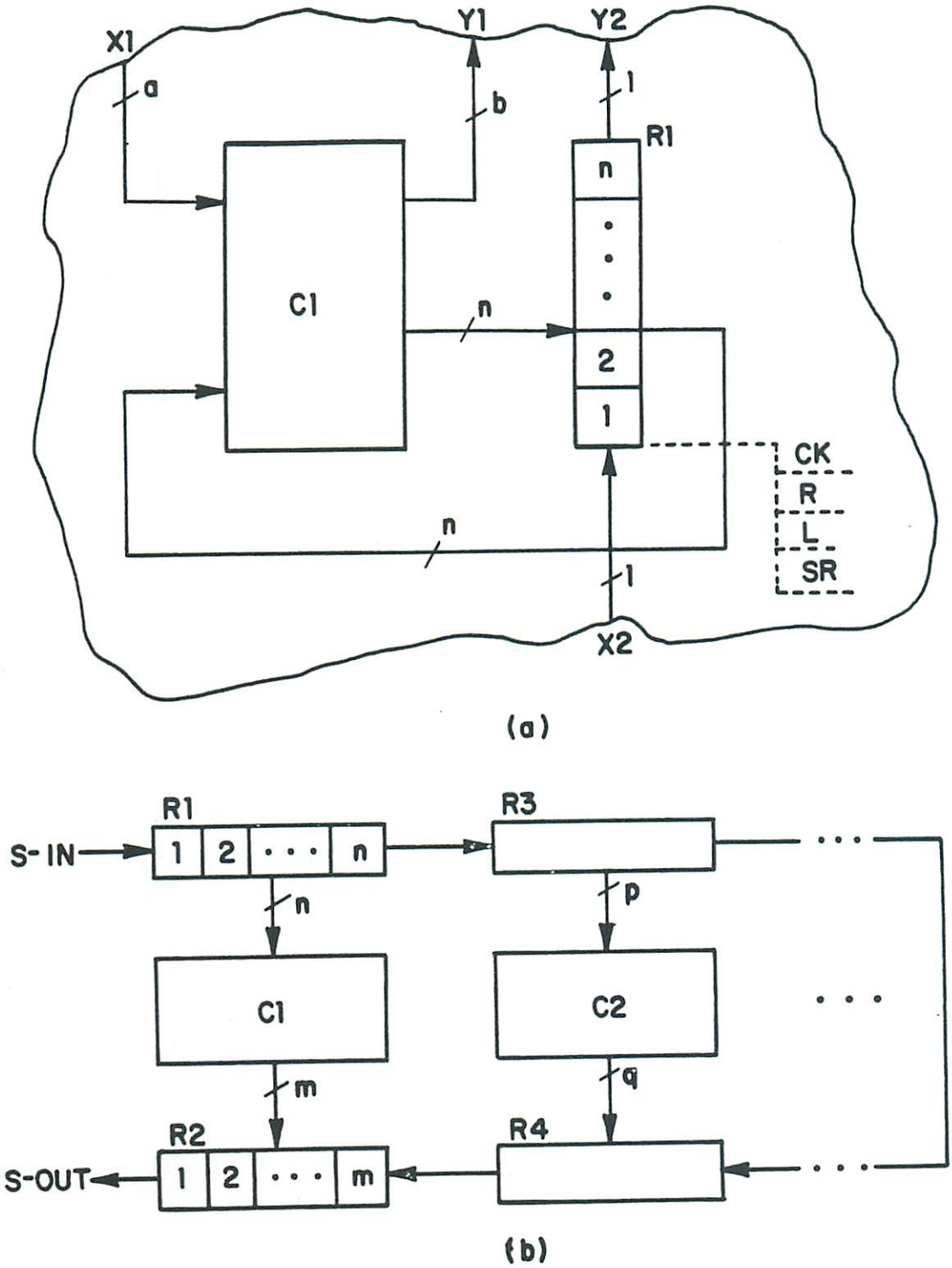
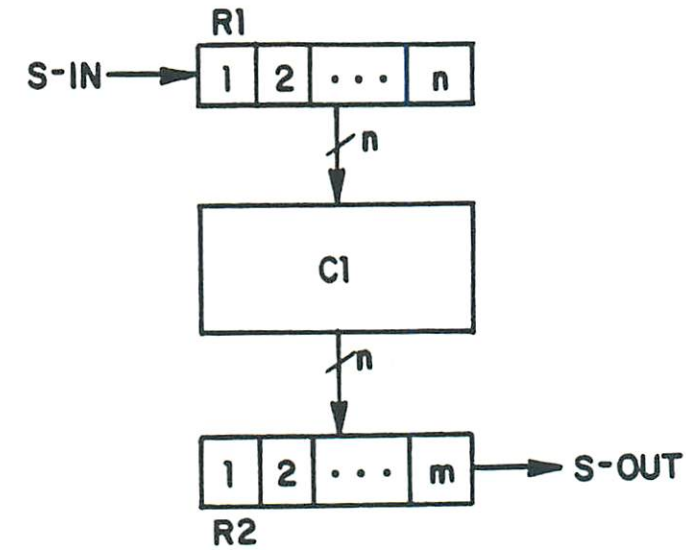
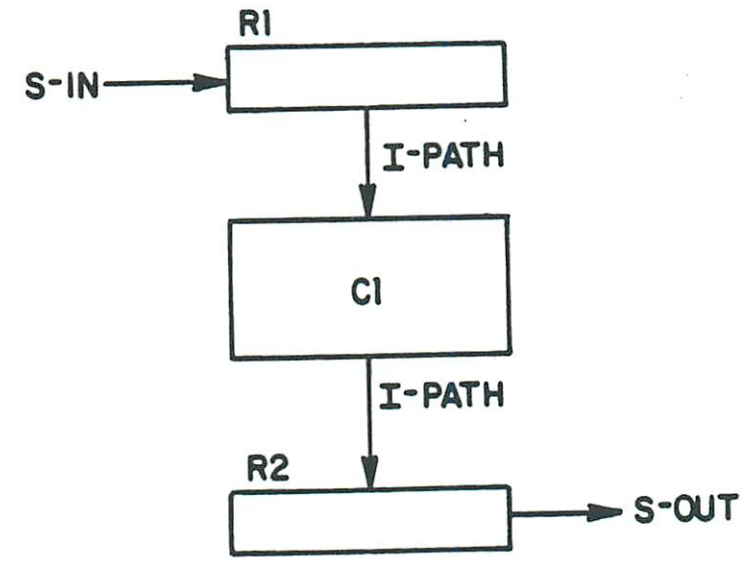


Figure 1: Set-scan TDM structures.
(a) Classical form for a sequential circuit
(b) Long shift register chain
(c) Separate input/output chains
(d) General form of (c)



(c)



(d)

Figure 1: Continued

Function	External Environment	Controller Requirements	
		Interface	Internal
HEAD			
1. Scan-in first test vector (n bits) into R1	I: X2	SR(R1)	Do n times
2. Input X1	I: X1	—	—
3. Output Y1	O: Y1	—	—
4. Latch R1	—	L(R1)	
BODY			
Repeat body (K-1) times			Do K-1 times
1. Scan-in next test vector and scan-out contents of R1	I: X2 O: Y2	SR(R1)	Do n times
2. Input X1	I: X1		
3. Output Y1	O: Y1	—	—
4. Latch R1		L(R1)	
TAIL			
1. Scan-out last test vector from R1	O: Y2	SR(R1)	Do n-1 times

Figure 2: Test schema for set-scan TDM.

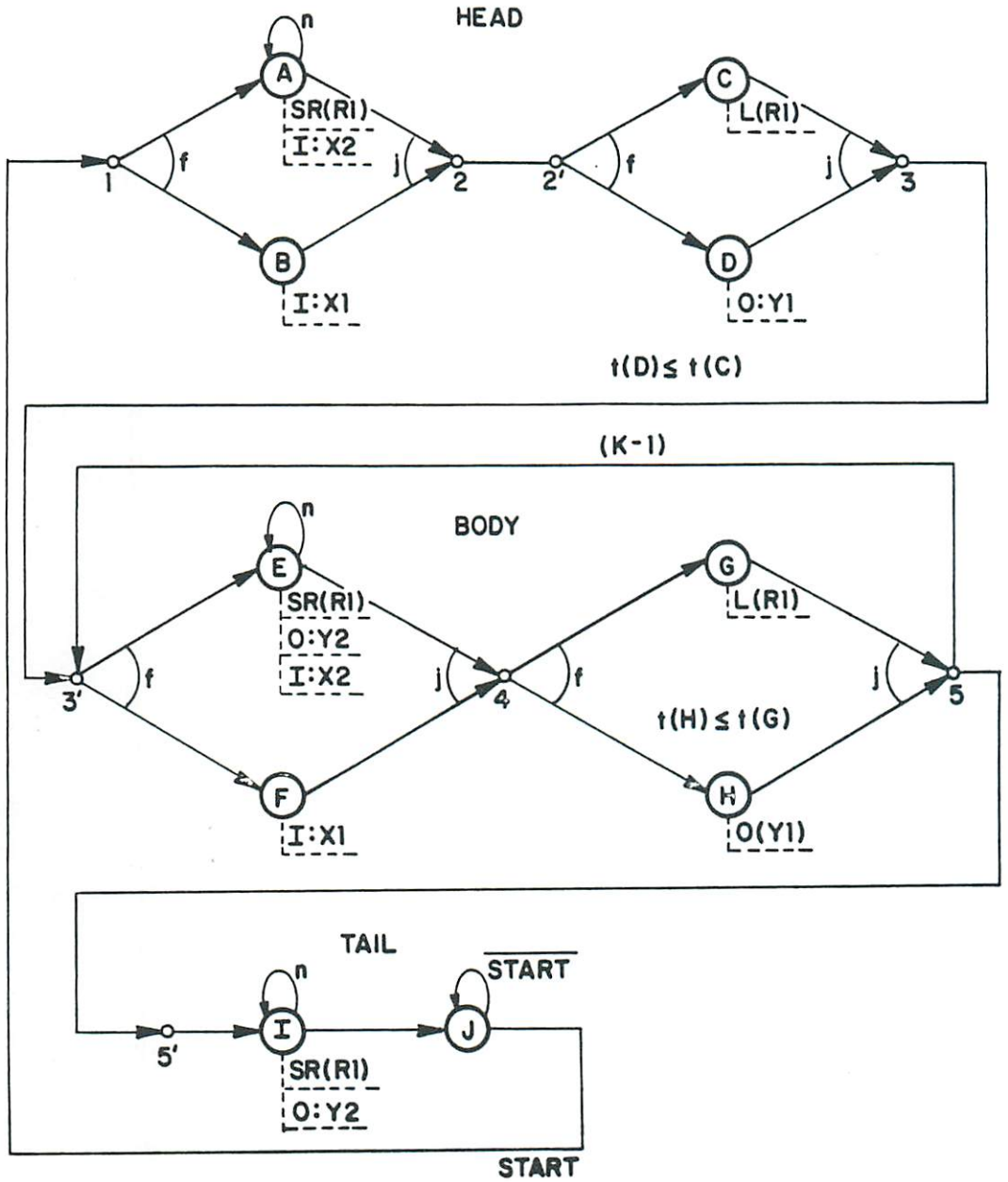


Figure 3: Control graph version of test schema for set-scan TDM.

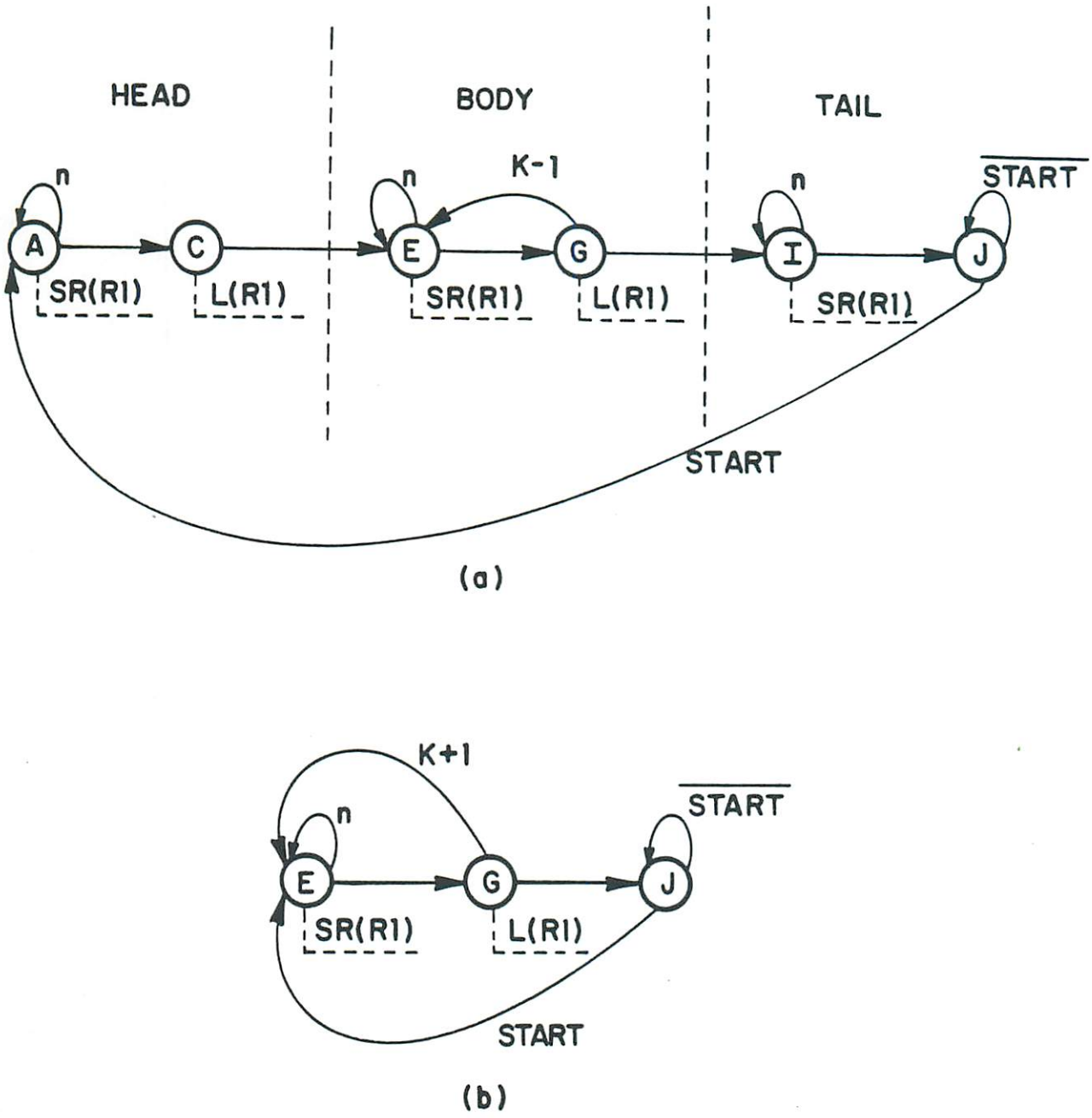
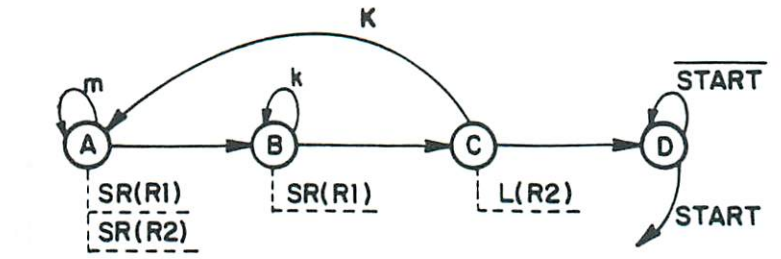
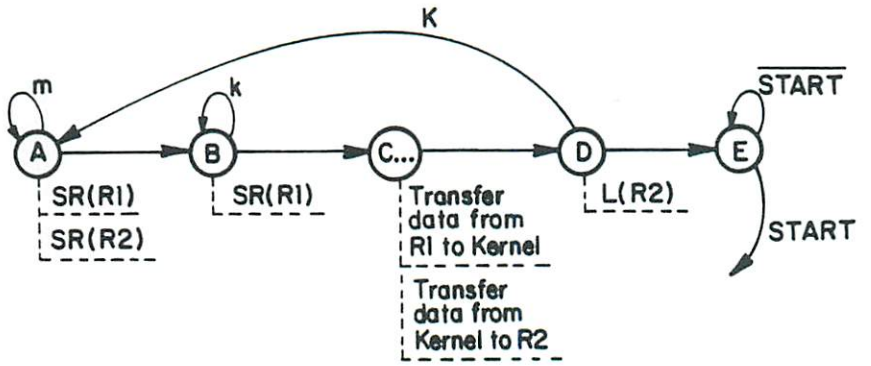


Figure 4: Simplified control graphs for the set-scan TDM structure.



(a)



(b)

Figure 5: Control graphs for independent I/O set-scan TDM structures.

Figure 1(b) indicates a different configuration for a set-scan structure. Here there may be several combinational circuits to test, and the scan-in and scan-out registers may form one long shift register chain. The simplest form for testing is to test all blocks of logic at the same time by scanning in a test for $C1, C2, \cdots$, latching the results into $R2$, $R4$..., and scanning out the results while scanning in the next test vector. Assume that it is desired to test each block individually. Then we require p shifts, where $p = max(n, m)$. Note that if $R2$ is in the middle of a shift register chain, then either we must shift the data completely out of the chain for each test vector, or else insure that no register latches results which will destroy the results of a test result. Hence individual latch controls may be required on the registers, which would probably be the normal way of designing a circuit.

The test schemas described earlier are adequate for this configuration, with appropriate choices made for the value of K and n.

In Figure 1(c) we indicate a third configuration for a set-scan TDM. Here the input and output registers can be controlled separately. Note that $R1$ has no serial output, and $R2$ has no serial input.

Assume $n > m$ and set $k = n - m$. Then both $R1$ and $R2$ can be shifted m times, followed by $R1$ shifted an additional k times. The control graph for this test schema is shown in Figure 5(a). Note that this test schema requires 3 counters, one for m, k and K.

Figure 1(d) indicates a more general form for a set-scan TDM structure, where now an I-

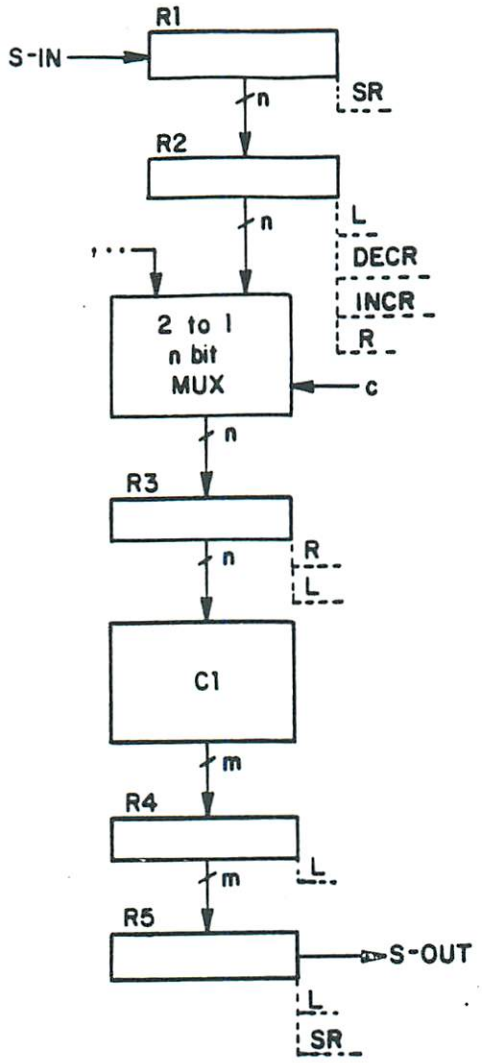


Figure 6: A portion of a circuit having kernel C1.

paths exist between the kernel and the set-scan registers. A control-graph for the test schema is shown in Figure 5(b). The state $C\cdots$ may correspond to a sequence of states. For simplicity, in Figure 6(a) and 6(b) we have indicated that $R1$ and $R2$ be simultaneously shifted m times, but this need not be the case.

Embedding a TDM

Often an actual design may require a non trivial I-path in order to implement a TDM. Consider the embedding of the TDM structure shown in Figure 1(d) into the circuit structure shown in Figure 6. The test plan of Figure 7 shows the flow of data for one test vector. Because test vectors can be overlapped to some extent, some pipeline action can be achieved. One test schedule is shown in Figure 8 for the case $n = 5$ and $m = 3$. Note that the third shift of $R5$ is not necessary. Here we have pipelined the tests to a maximum degree. We see that for the body of the test, 3 test vectors can be processed in parallel, indicated by a cycle. In our case,

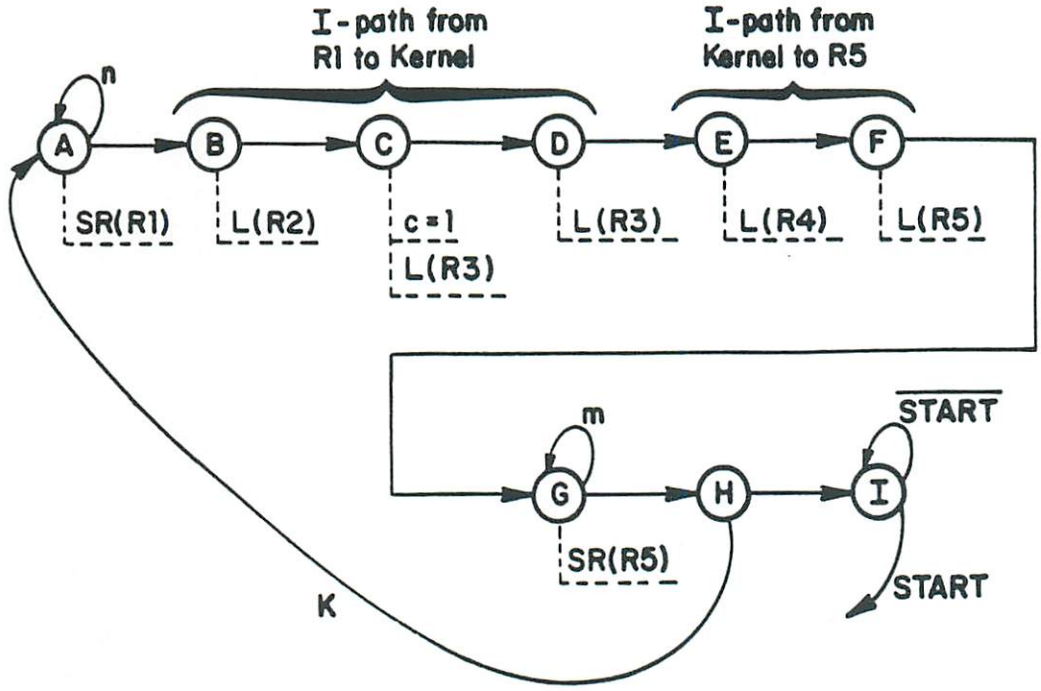


Figure 7: Test plan for set-scan TDM imbedded in circuit of Figure 6.

the cycle is of length 5. In general, the length of a cycle is the maximum of m and n. The control graph corresponding to Figure 8 is shown in Figure 9(a). A simple language form of this control graph is shown in Figure 9(b), where we have ignored the wait state F. In this procedural language, operators separated by a comma are executed in parallel. The end of a clock period is represented by a period.

Referring again to Figure 6, assume we set $n = 30$ and $m = 20$. Then $k = (n - m) = 20$. Also, the pipe is 4 registers deep, hence for this case we have a test schedule as shown in Figure 10.

In summary, the general form for a test schedule for the set-scan TDM appears to be a sequence of control states and three loops, two for the shifting of the scan-in and scan-out registers, and one for K.

In addition, if several independent set-scan test structures exist, then different test schedules may also have to be implemented, each for possibly different values of n, m and K. Also, the main effect imposed by I-paths is to add more steps to the test plan.

3.2 Pseudo Random BIT TDM

The pseudo random $BIT\ TDM$ employs a linear feedback shift register ($LFSR$) for random pattern generation (RPG), and another $LFSR$ for signature analysis (SA). There are four

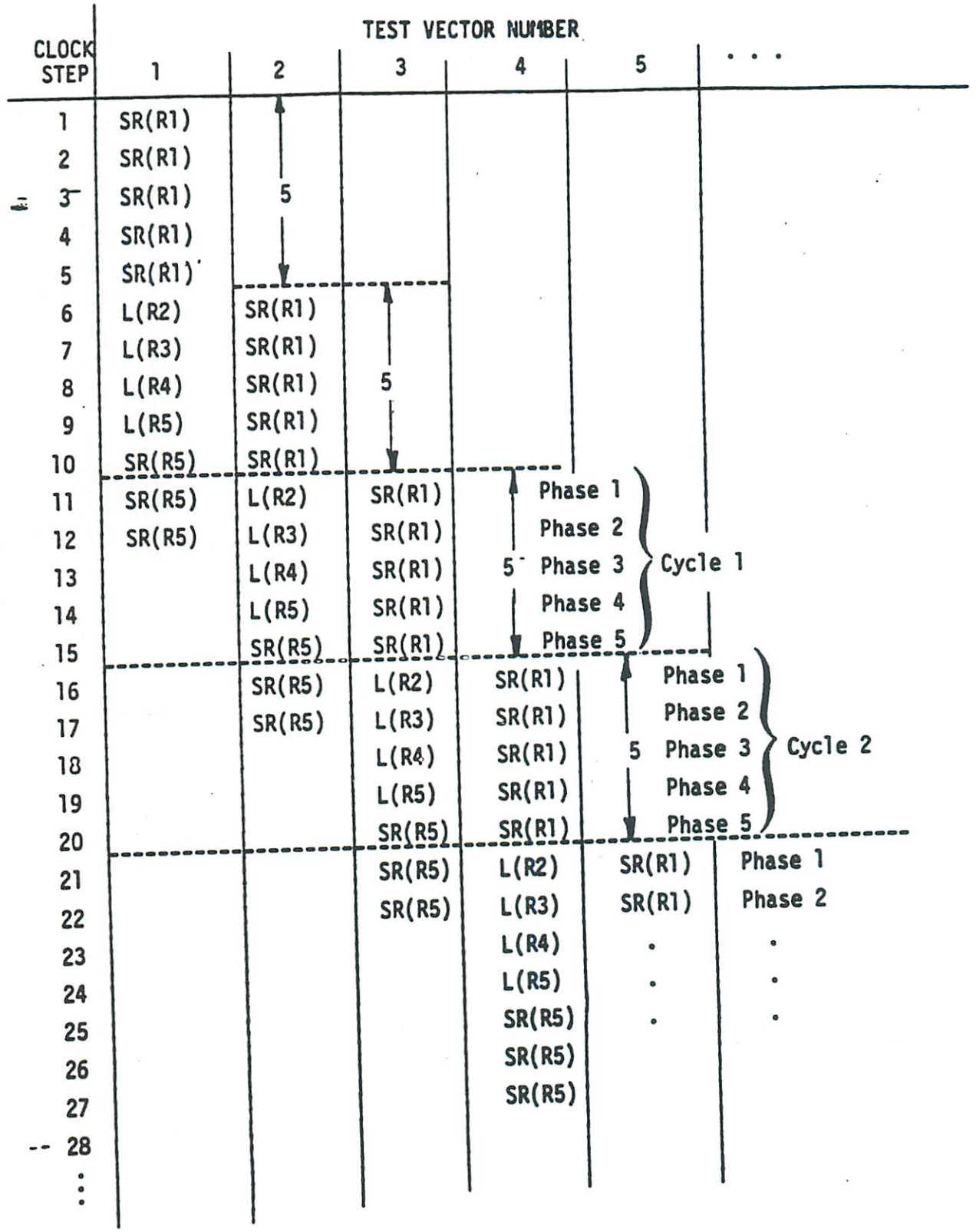
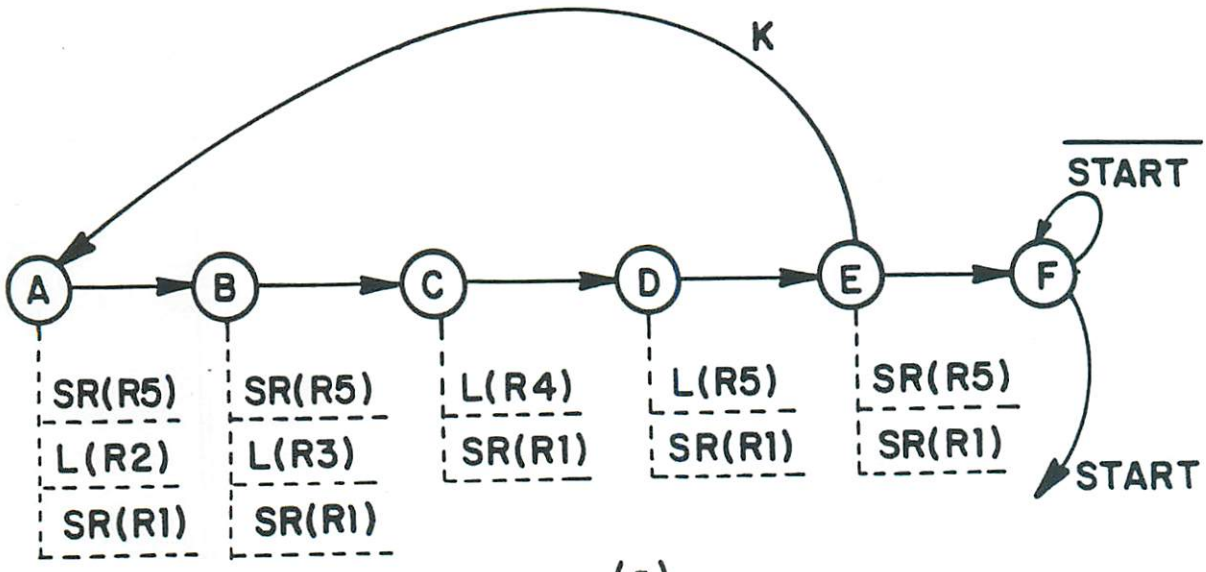


Figure 8: Test schedule.



(a)

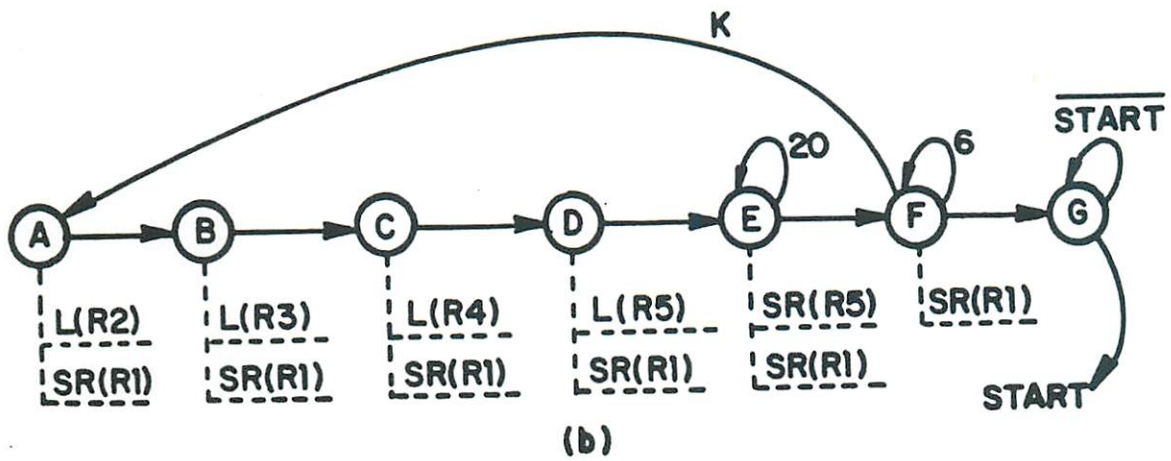
```
For each of K test vectors
        SR(R5), L(R2), SR(R1).
        SR(R5), L(R3), SR(R1).
        L(R4), SR(R1).
        L(R5), SR(R1).
        SR(R5), SR(R1).
end for
```

(b)

Figure 9: Test schedules.
(a) Graph form
(b) Program form

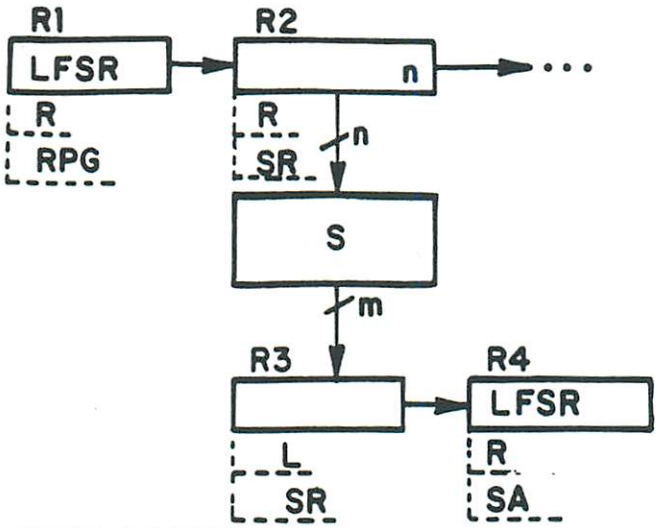
Phase #	i	Test # i+1	
1	L(R2)	SR(R1)	
2	L(R3)	SR(R1)	
3	L(R4)		
4	L(R5)		
5	SR(R5)		
6	SR(R5)	.	
7	.	.	
.	.	.	20
.	.		
.			
24	SR(R5)	SR(R1)	
25	—	SR(R1)	
26	—	.	
27	—	.	6
28	—	.	
29	—		
30	—	SR(R1)	

(a)



(b)

Figure 10: Test schedules for $n = 30, m = 20$.
(a) One cycle of test schedule
(b) Control graph



TEST SCHEMA

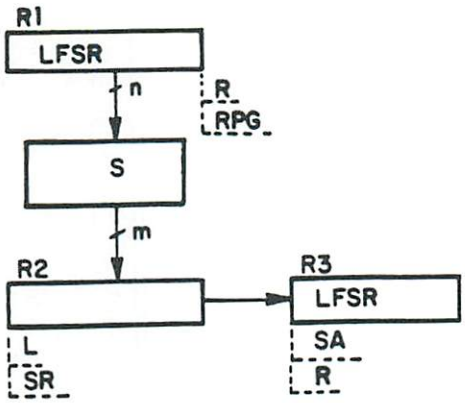
R(R1), R(R2), R(R4).
For each of K test vectors
    RPG(R1), SR(R2), L(R3)
    For each of m bits of R3
        SR(R3), SA(R4)
    end for.
end for.

(a)

Figure 11: Pseudo random BIT TDMs.
(a) S/I, S/O
(b) P/I, S/O
(c) S/I, P/O
(d) P/I, P/O

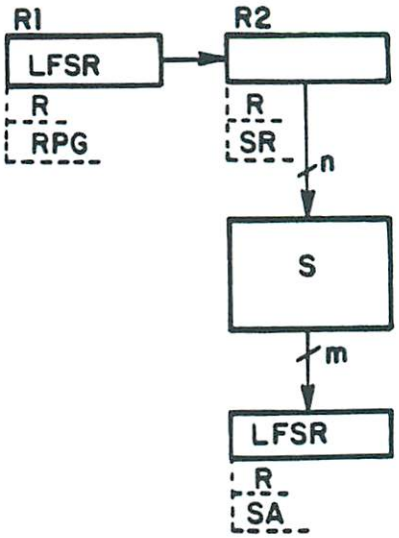
basic structures used, depending on whether or not the inputs and outputs from the $LFSRs$ are processed in series or in parallel. These structures and their corresponding test schemas are shown in Figure 11. Figure 11(a) illustrates the TDM for a serial input (S/I), serial output (S/O) BIT structure. The kernel S can be either a combinational or sequential circuit. The first step in the test schemas initialize or reset the registers specified. The results of the last test vector applied to the kernel are not processed by the test schemas shown. It is seen that for the two serial output $TDMs$, two "for" loops in the test schemas exist, while for the other two test schemas, only one "for" loop exists.

These $TDMs$ can be easily applied to complex circuits where non trivial I-paths exist. The result leads to test schedules similar to the test schemas shown in Figure 11, i.e. with one or two "for" loops, and possibly a few more steps used in transferring data from one register to another.



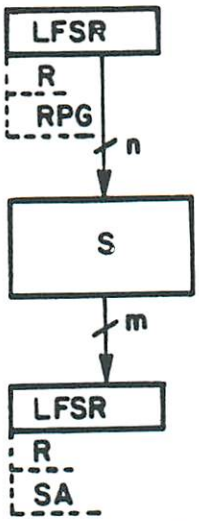
TEST SCHEMA
R(R1), R(R3).
For each of K test vectors
RPG(R1), L(R2).
For each of n bits of R2
SR(R2), SA(R3).
end for.
end for.

(b)



TEST SCHEMA
R(R1), R(R2), R(R3).
For each of K test vectors
RPG(R1), SR(R2), SA(R3).
end for.

(c)



TEST SCHEMA
R(R1), R(R2).
For each of K test vectors
RPG(R1), SA(R2).
end for.

(d)

Figure 11: Continued

3.3 Exhaustive BIT TDMS

The structures and test schemas for the exhaustive *BIT TDMs* are similar to those for the pseudo random *BIT TDMs*, and hence will not be discussed here in any length. The two main differences in these two techniques is that for exhaustive testing, (1) the kernel is assumed to be combinational logic, and (2) each output line of the kernel is implemented by a cone of logic, and this cone is tested exhaustively - hence the random pattern generator must be designed so that all $2^k - 1$ patterns are applied to each cone having k inputs. The all zero pattern is usually not included in this test.

4 CONTROL UNIT DESIGN

In the previous section we have discussed and illustrated the type of circuits which need be controlled. In general, the control graph consists of a sequence of states with one, two, and in some cases three "for" loops. The outer loop deals with $K-$ the number of test vectors to be applied to the kernel; the inner loops are for the shifting of data into or out of registers. In this section we will discuss microprogrammed and hardwired control units.

Figure 12 indicates a general interface configuration for a controller and a BIT structure to be controlled. Here, for example, $L(i)$ refers to the i^{th} load control line. PI refers to the data input ports, and PO to data output ports.

The main functions of the controller are the following.

1. To communicate with the external environment.
2. To select the desired test schedule to execute.
3. To execute a test schedule, including sequencing through loops when required.
4. To issue control signals to the structure to be tested.

In case the controller must be synchronized with other circuits, such as other chips or ATE in order to transmit and receive data, four "handshaking" lines are provided, namely the inputs and outputs shown next. An ETM bus interface can also be used.

Inputs

DRE	- used to indicate that "data was received at the environment"
DAE	- used to indicate that "data is ready to be read (available) from the test environment"

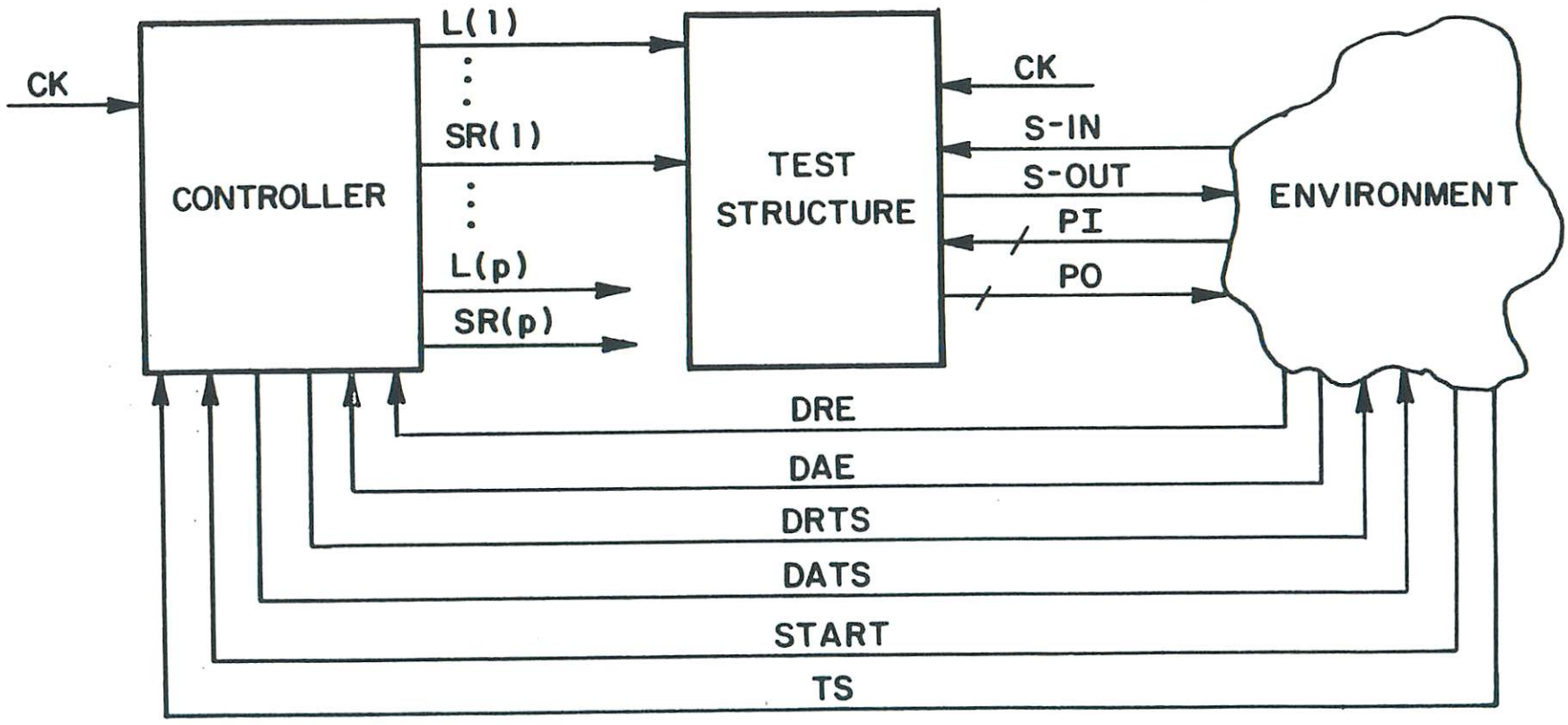


Figure 12: Generic structure of interface for controller/test structure/environment.

Outputs

DATS - used to indicate that "data is ready to be read (available) from the test structure"

DRTS - used to indicate that "data was received at the test structure"

The input TS can be used to transmit the name of a test schedule to be executed, when appropriate.

As a simple scenario, the environment E may send a pulse over the $START$ line to "wake up" the controller, followed by a k bit code over the TS line to inform the controller which test schedule is to be executed. The controller then executes this schedule, testing a particular portion of the chip. At the end of the test, a signature or simply a 0 or 1 signal to indicate pass or fail can be sent back to the environment over the S-OUT line.

It appears that a natural architecture for the controller consists of a unit for keeping count of the step numbers in a loop, such as a counter, and a finite state machine.

4.1 Microprogram Control Unit Design

The general architecture for a microprogrammable controller for test strategies is shown in Figure 13. The register stack contains constants, such as K, which can be loaded into the accumulator registers (ACC), decremented ($DECR$), and stored back into a temporary register in the stack. The logic C determines if the content of the accumulator is zero. The rest of the design is quite conventional. The fields of the instruction register (IR) are listed next.

Fields

1 - **opcode:** contains an encoding for which control lines should be active in the controller, or equivalently, specifies a micro-instruction to be executed by the controller.

2 - **control field:** specifies which control lines should be active in the test structure.

3 - **branch condition address:** used to specify the "name" of the variable on which a branch should be made dependent.

4 - **miscellaneous field:**

5 - **address:** specifies either an address for $ADR1$ or $ADR2$.

Fields 1 or 2 can be either horizontally or vertically encoded.

A few basic microinstructions are described next.

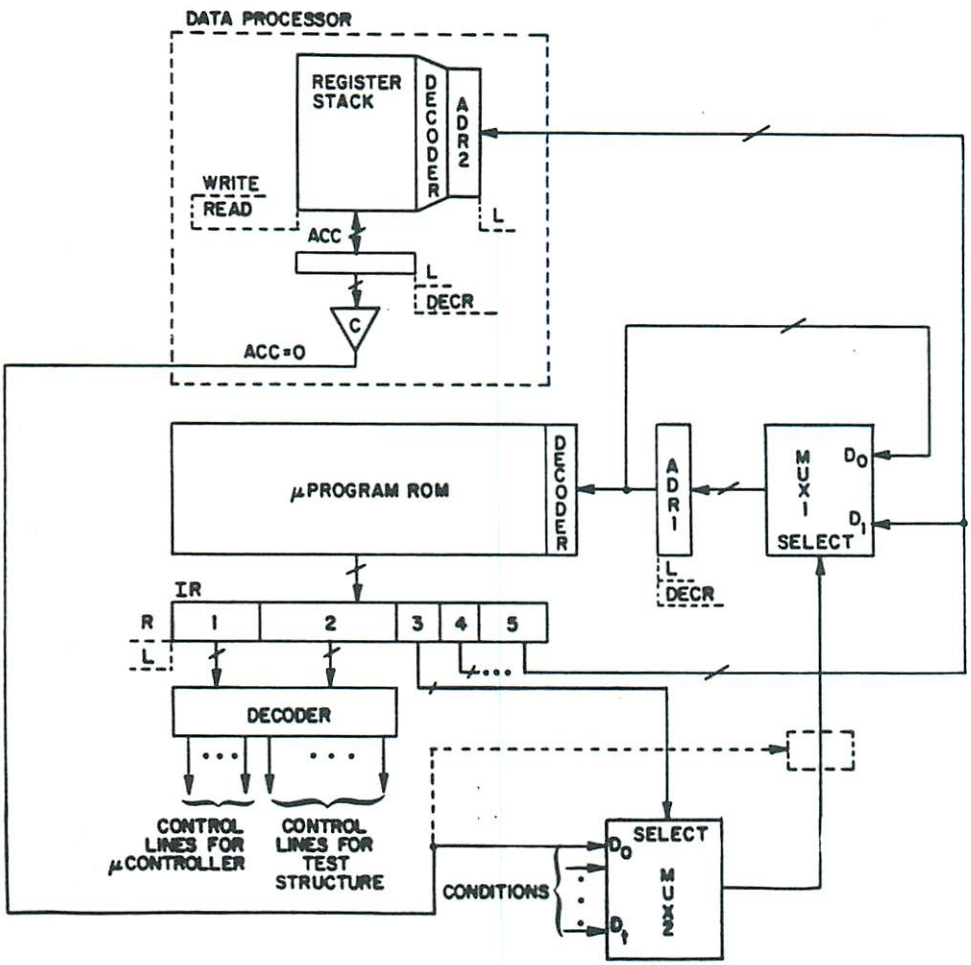


Figure 13: Microprogram controller.

Instruction No.	FIELDS			Comments
	1 opcode	2 controls	5 address	
1	LD	0	0	K-1 is in register 0
2	STO	0	1	Test vector count is in register 1
3	LD	0	2	n-1 is in register 2
4	DBNZ	SR(R1)	4	Shift right R1; execute this loop n times
5	NOP	L(R1)	0	Load R1
6	LD	1	0	Test vector count to ACC
7	DBNZ	0	2	Repeat major loop
8	...			end

Figure 14: Microprogram implementation of control graph of Figure 6(b).

Microinstruction List for Controller

field		
1 opcode	5 address	function
LD	I	load ACC from register I of stack
STO	I	load register I of stack with contents of ACC
DBNZ	N	decrement ACC; if current contents of $ACC \neq 0$, branch to N
NOP	-	no operation

Here, for simplicity, we have not used the branch condition address (field 3), hence for the $DBNZ$ instruction, if $ACC \neq 0$ then N is forced into $ADR1$, else $ADR1$ takes its normal next value which is $ADR1 + 1$.

The microprogram for the control graph of Figure 4(b) is shown in Figure 14. Here $(K-1)$ and $(n-1)$ are permanently stored in stack registers 0 and 2 respectively. The program requires only 7 instructions and 4 different operation codes.

Sizing a Typical Microprogram Control Configuration

Assume there are 4 separate test strategies to be executed, two requiring two loops and two requiring 3 loops. Hence 10 constants must be stored. Three words of working storage are required, hence the stack needs at least 13 words. Thus a 16 word stack and a 4 bit address are required. The width of each stack word should be approximately 20 bits, which would allow for an exhaustive or semi-exhaustive test of $K = 2^{20}$ patterns. Three microinstruction fields are required.

1 - op code: 3 bits provide for 8 operation codes.

2 - control field: If we assume a horizontal microcode configuration, with 2 shift and 3 load control lines per test strategy, then 20 bits of ROM are required for this field.

3 - address: 4 bits provide for addressing the stack as well as relative branches in the microcode of up to 16 words. However, a subtractor is now required so that $ADR1$ can be decreased by the specified amount.

The total length of a microinstruction word is thus $3 + 20 + 4 = 27$ bits. If we assume an average microprogram length for a test schedule is to be 10 words, then 40 words of ROM are required.

The control field width can be drastically reduced by (1) encoding the shift, load and reset controls, and (2) using a two BIT register to specify which test strategy is being executed.

Sizing a Near Minimal Microprogram Controller

A near minimal microprogrammable controller would probably deal with one test schedule having two loops, and would thus need 4 registers in the stack, 3 bits each for the operation code, control and address fields, resulting in a 9 bit wide ROM of 16 words.

If the chip being tested already has a microprogram control unit, then the control of the test procedure requires very little additional overhead. The latch and shift controls already exist. The data processor portion of the controller may need to be added to the chip if it does not already have an accumulator/ALU structure. An additional advantage of having a microprogrammed control is that one can implement microdiagnostic routines and functional tests.

4.2 Hardwired Control Unit Designs

Figure 15 indicates the design of a hardwired control unit consisting of a set of registers and the state diagram of a finite state machine (FSM). The machine contains 5 states and hence can be implemented with just 3 flip flops ($Q1, Q2, Q3$). The inputs to this unit are $D1, D2$ and $START$. The state ($Q1, Q2, Q3$) can be decoded to produce the control signals $L(R1)$, $L(R2)$, $L(R3)$, $SR(R1)$, $DECR(R2)$ and $DECR(R3)$. $D1$ and $D2$ can each be generated by a single "large input" $NAND$ gate.

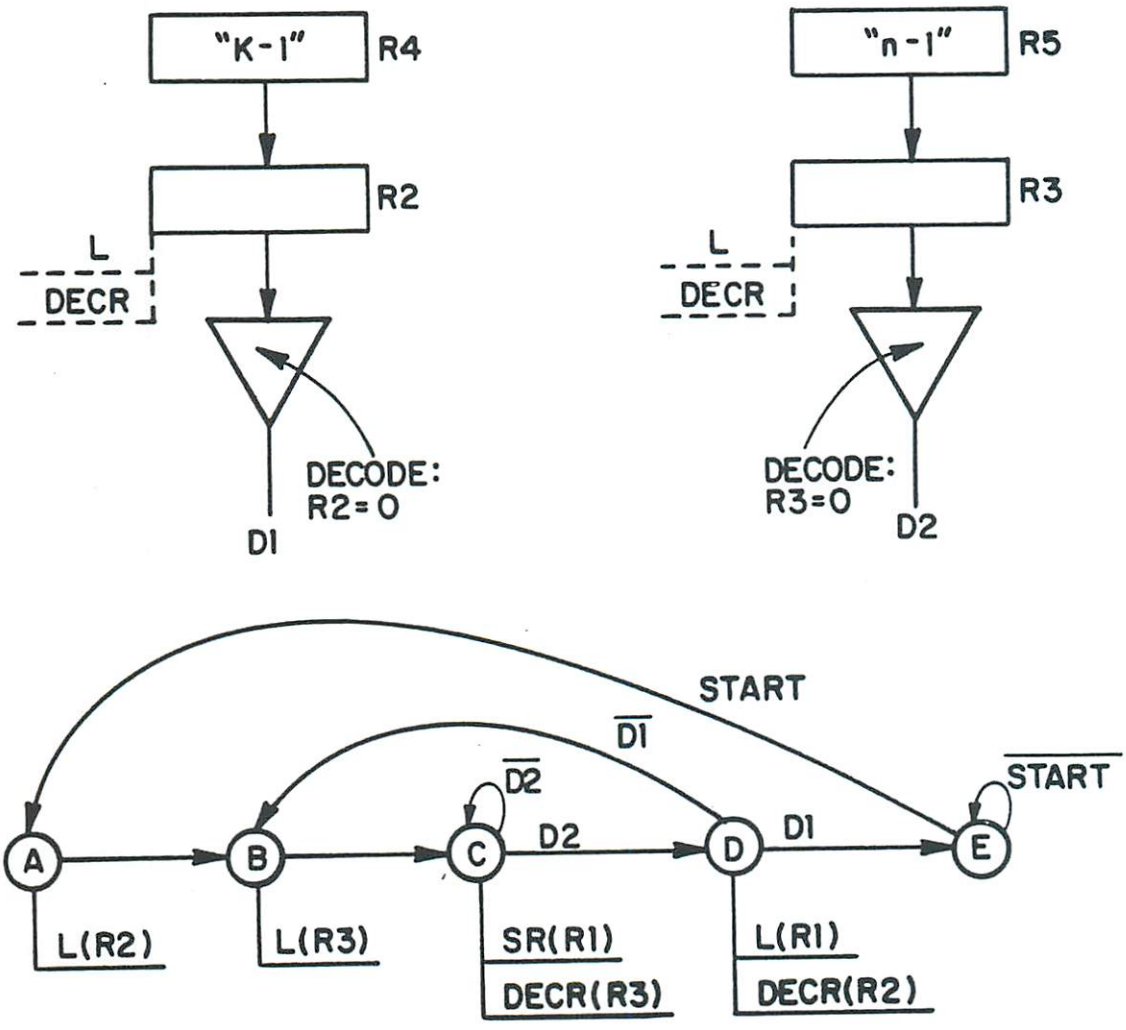


Figure 15: Hardwired control design of control graph of Figure 6(b).

If several test schedules exist, and each requires one or more loops, then numerous constants must be stored. Again a data processor as shown in Figure 13 is useful. A state in the state diagram can now be associated with a control signal of the form $L(ACC, i)$, which sets the accumulator to the latch mode, and register i of the stack is selected as the source register. The control command $S(ACC, i)$ is used to store the contents of the accumulator into register i of the stack. A design using this scheme is shown in Figure 16. The stack can be designed so that the constants are stored in read only memory, and only a few registers used for temporary storage have both read and write capability.

As we have seen, most test schedules consist of a main body repeated approximately K times. This body consists of one or more phases or steps, where some steps may contain a self loop required for shifting. Each test schedule can be implemented using a register stack and an FSM. To implement several test schedules, a programmable FSM can be used. That is, a register G can contain the number of the test schedule to be implemented, and this data used to modify the next state logic so that the desired state sequence is generated. As G counts through its various states, all test schedules will be implemented.

Figure 17(a) shows the general form of the state diagram for a test schedule. One simple way to implement such a diagram is to use a one-hot code, i.e., a pseudo shift register containing a single "1" bit. The register has the ability to either pass the "1" to the next flip flop in the register (shift), hold the "1" in a flip flop for several clock periods, or transfer it to a previous flip flop in the chain. Figure 17(b) indicates this concept. Figure 17(c) shows an $NMOS$ circuit for implementing a one-hot code, and Figure 17(d) shows how, by using two $MUXs$, the "1" is transferred to either a previous register cell or the next register cell. This appears to be a fairly simple implementation for a test schedule.

Figure 18(a), (b), (c) indicates three test schedules, denoted by I, II, III. Schedule I consists of states A through I. The edges are labeled for later reference. These three test schedules can be combined in two ways, namely by concatenation or by superposition.

To concatenate these schedules, one need only delete edges 12 and 19, and replace edge 11 by an unconditional edge $11'$ from state I to J, and replace edge 20 by an unconditional edge $20'$ from state N to O. The resulting schedule consists of the 20 states A through T and can be implemented in many ways.

The superposition of these three schedules produces the schedule shown in Figure 18(d), which consists of the 9 states a through i. Each of the states $A \cdots T$ is mapped into one of the states $a \cdots i$. We denote this mapping by $M : X \rightarrow y$. In Figure 18(a) we have indicated that state E is mapped into state e; in Figure 18(d) we indicate that states E, L and R are all mapped into state e. Once the states have been mapped, the edges in Figure 18(d) can also be easily determined, e.g., in schedule III there is an edge 22 between P and Q, and $M : \mathrm{P} \rightarrow b$, and $M : Q \rightarrow d$, hence we require an edge between b and d, and we label it III. Thus, if the machine is in state b and executing schedule III, then the next state is d. Edges 11 and 20 are modified to be $11'$ and $20'$, and 12 and 19 are ignored, as discussed previously. Note that a single edge leaving a state represents an unconditional transfer, and the schedule number(s) are not needed. Knowing the state and the schedule, control signals can be generated. Again a one-hot code can be used to implement the corresponding state diagram.

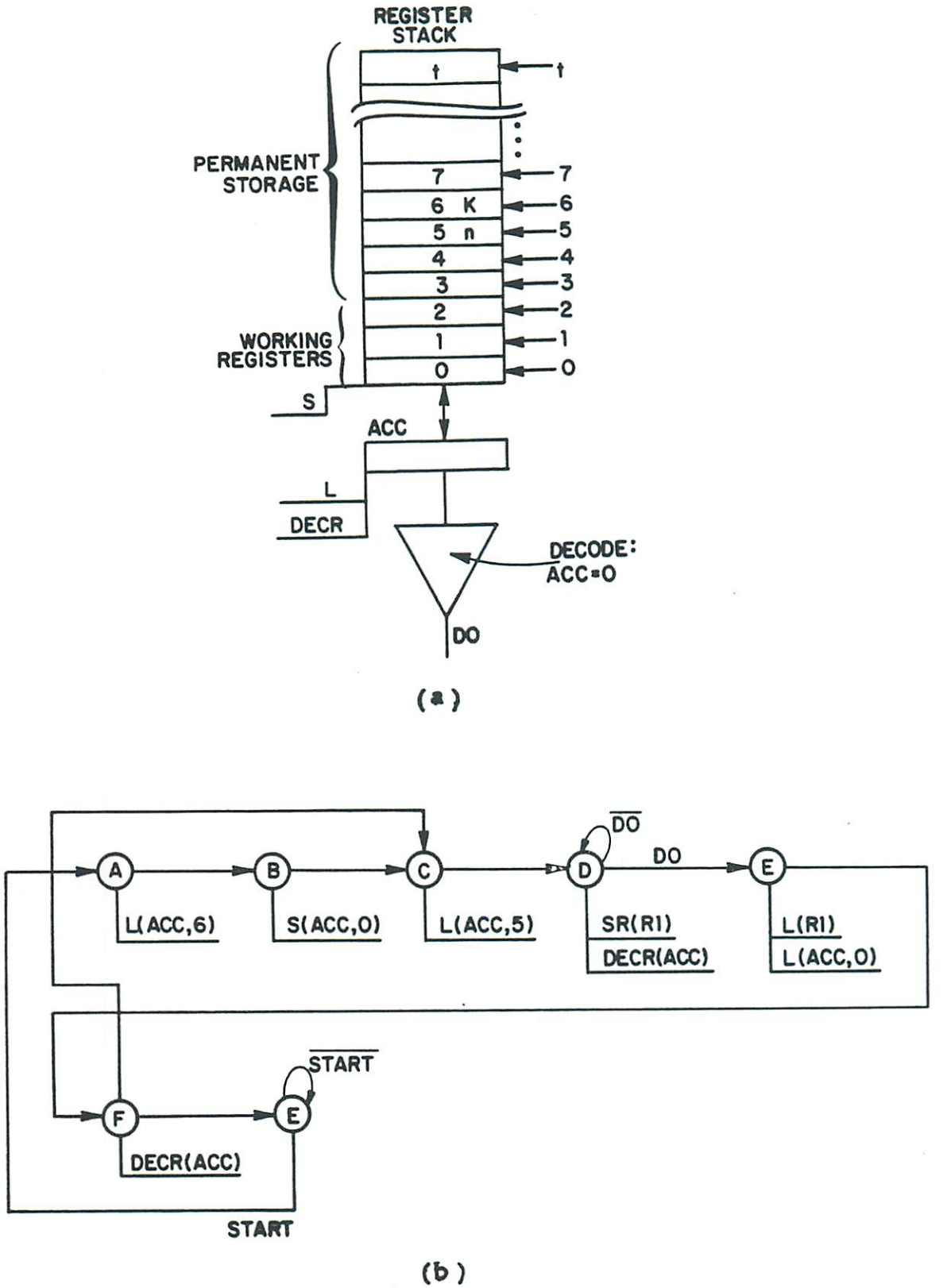
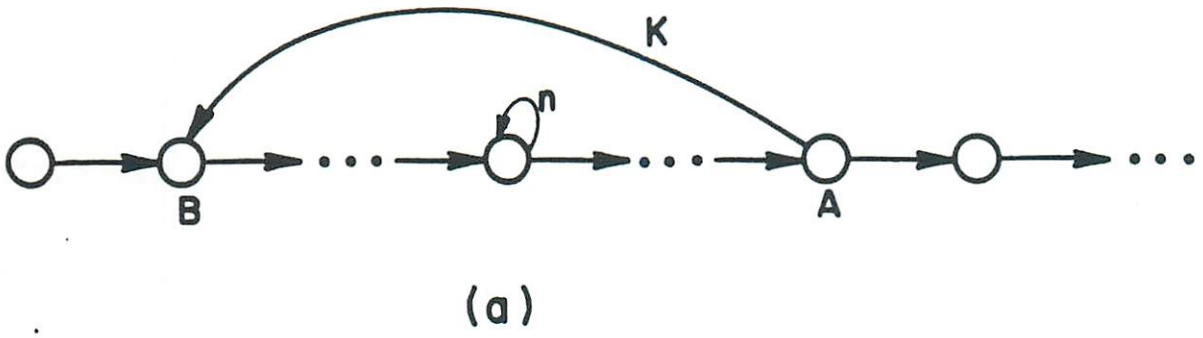
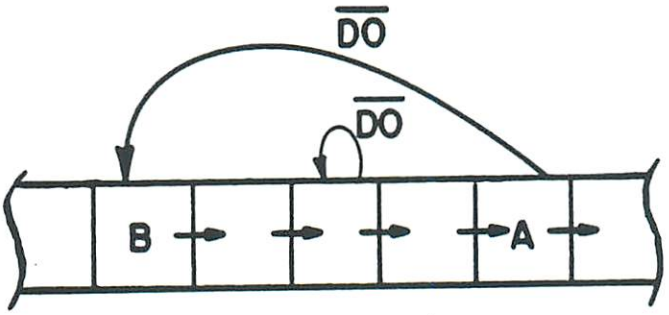


Figure 16: Hardwired control design using a register stack for control graph of Figure 6(b).
(a) Register stack
(b) Control flow

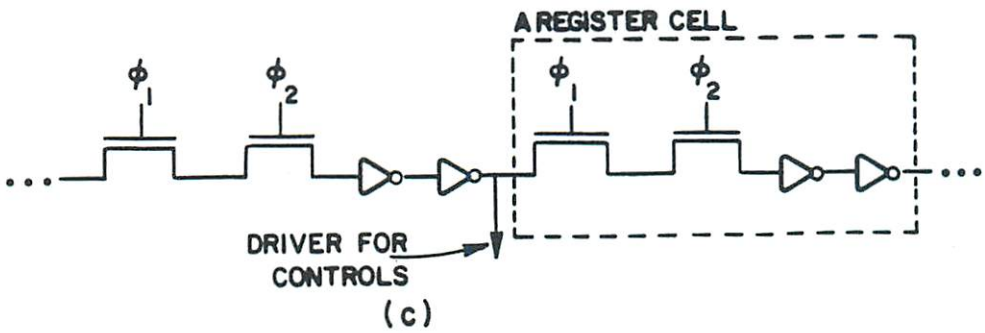


(a)

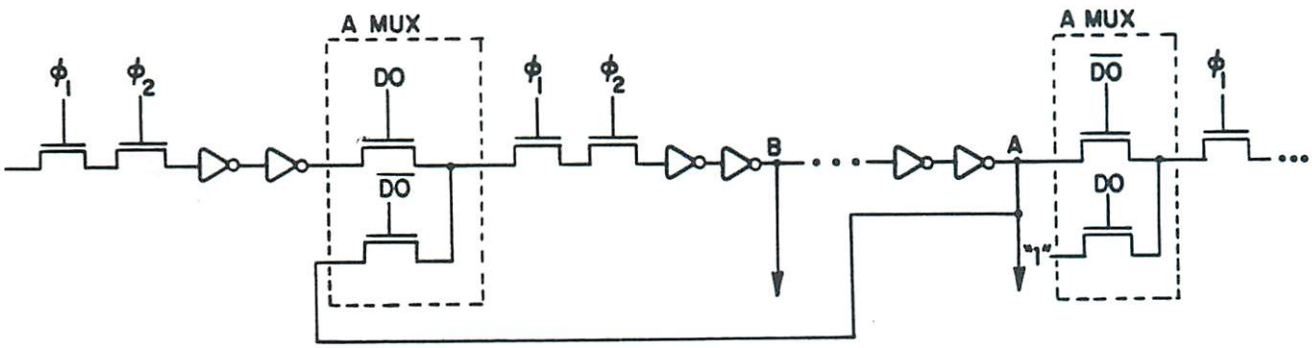


(b)

Figure 17: (a) General form of the state diagram control for a test schedule.
(b) One-hot code implementation
(c) NMOS shift register
(d) NMOS one-hot code circuit with loops



(c)



(d)

Figure 17: Continued

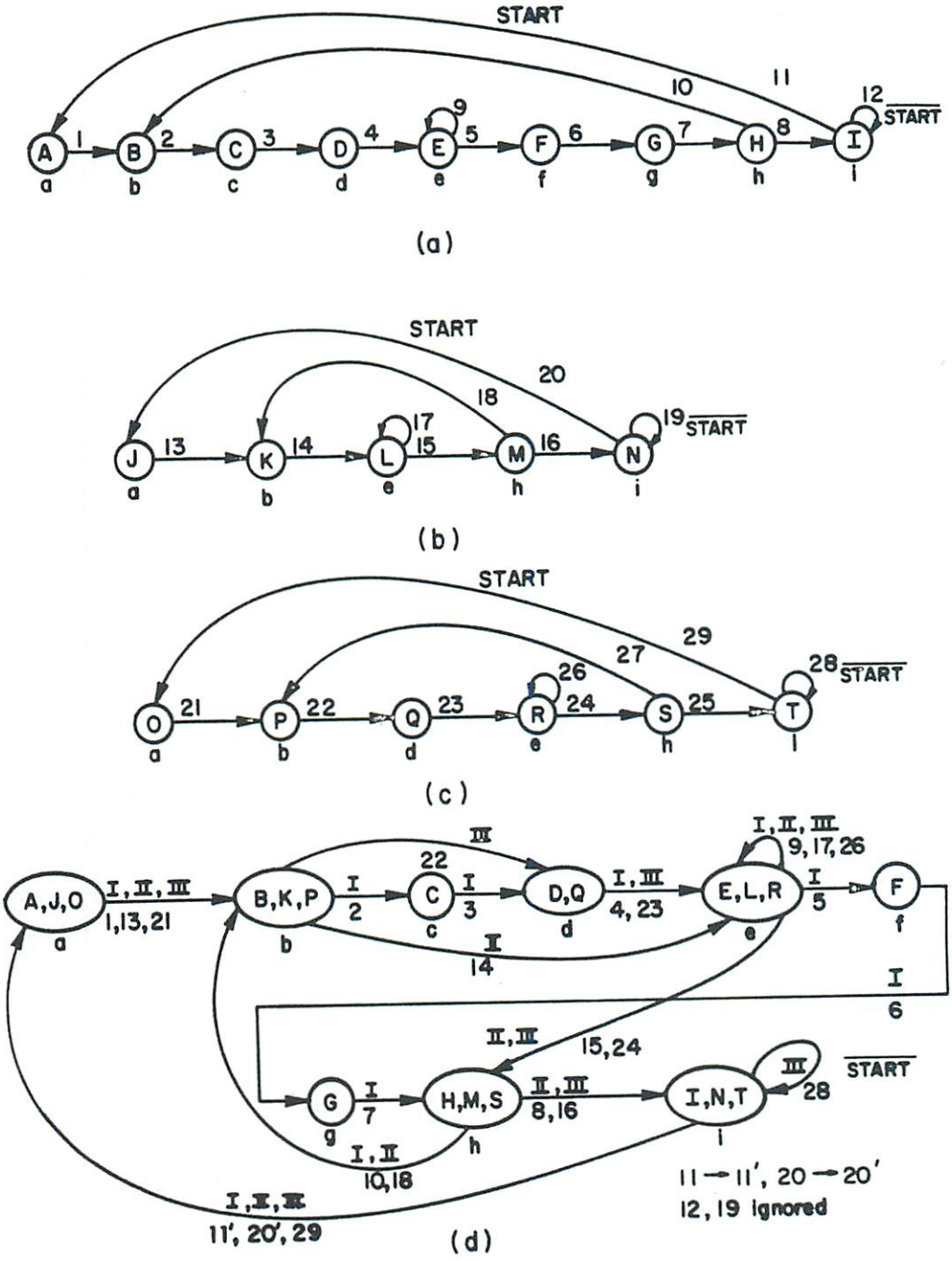


Figure 18: (a) Test schedule I.
(b) Test schedule II
(c) Test schedule III
(d) Combined test schedules

5 SUMMARY OF RESULTS

In this paper we have discussed three *BIT* techniques, namely set-scan, random and exhaustive testing. We have illustrated the concepts of test schemas, test plans, and test schedules. We have then presented several generic architectures for implementing machines to carry out the control function implied by the test schedules.

The microprogram control unit is very flexible, but appears to require a considerable amount of circuitry. If the chip already has a microprogrammable controller, then the overhead is much less. If such a chip also has a local store and *ALU*, then almost no overhead exists. However, there is a paradox here, in that the hardware being tested cannot be the same as that used to control the test process. Hence a careful partition is required between the portion of the microprogram hardware used to control the test process, the test structures used to enhance *BIT*, such as *LFSRs*, and the kernel structures being tested. To test the microprogram controller may require a small hardwired controller, or a self test program.

The hardwired controller appears to be quite simple, and a one-hot code using *MUXs* is sufficient for its implementation.

Both controller design methodologies make use of a register stack and accumulator for processing loops. It appears that most test schedules have from one to three loops, and required about 10 states to implement. The complexity increases somewhat if heads and tails of schedules are nontrivial, and if *I/O* protocols with the environment must be provided.

Finally, it is feasible to put the controller on one chip, and have this circuitry control the test process on other chips. In fact, even the *LFSRs* can exist off the chip being tested, and data sent to and from the chip under test via the serial scan-in and scan-out lines.

6 REFERENCES

1. T.W. Williams and K.P. Parker, "Design for testability - A survey," *IEEE Trans. on Computers*, vol. C-31, January 1982, pp. 2-15.

2. J.J. LeBlance, "LOCST: A built-in self-test technique," *IEEE Design and Test of Computers*, November 1984, pp. 45-52.

3. M.S. Abadir and M.A. Breuer, "Constructing optimal test schedules for VLSI circuits having built-in-test hardware," *Digest of Papers* 15^{th} *Int'l Conf. on Fault-Tolerant Computing*, June 1985, pp. 165-170.

4. M.S. Abadir and M.A. Breuer, "A knowledge-based system for designing testable VLSI chips," *IEEE Design and Test of Computers*, August 1985, pp. 56-68.

5. M.A. Breuer, "On-chip controller design for built-in-test", Hughes Technical Internal Correspondence, Ref. 7220-85-315, October 15, 1985.