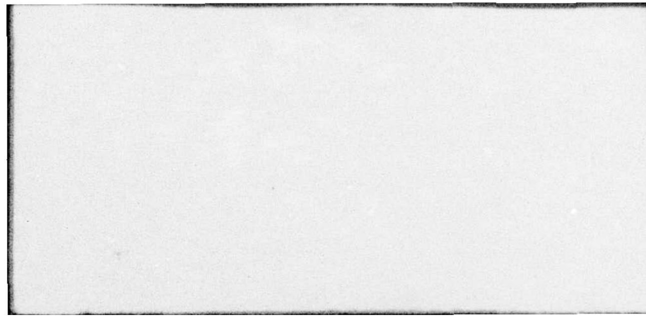
AD-A040 271 BREUER AND ASSOCIATES ENCINO CALIF F/G 9/5
AN ADVANCED AUTOMATIC TEST GENERATION SYSTEM FOR DIGITAL CIRCUI--ETC(U)
MAR 77 M A BREUER N00014-75-C-1053
UNCLASSIFIED TEST/80-1-77 NL



BREUER & ASSOCIATES

SOFTWARE SYSTEMS AND
CONSULTATION IN CAD/CAM

16857 BOSQUE DRIVE
ENCINO, CALIF. 91436

(213) 981-7583

(12)

(14) TEST/80-
Report 1-77

(6) An Advanced Automatic Test Generation
System for Digital Circuits*

Submitted to:

Department of the Navy
Office of Naval Research
Arlington, Virginia 22217

DDC
RECEIVED
JUN 7 1977
C

Prepared by:

Breuer and Associates

392 045

(11) 29 Mar 77

(12) 111p.

Project Director:

Melvin A. Breuer
(10) Melvin A. Breuer
March 29, 1977

(15)
*This work was carried out under Contract No. N00014-75-C-1053, NR 048-631.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

bpg

TABLE OF CONTENTS

Section		Page
	ABSTRACT	iv
1	INTRODUCTION	1
2	PREPROCESSING	6
	2.1 Rate Analysis	6
	2.2 Cost Assignments	18
3	TIMING	43
	3.1 Basic Definitions on Time	43
	3.2 Problem/Solution Space	50
	3.3 Dynamic Assignment of Phases and Periods	54
	3.4 Data Structures	58
4	TEST GENERATION ALGORITHM	69
	4.1 Data Structure	69
	4.2 Functional Element Routines	70
	4.3 Implication Routine	74
	4.4 D-Drive Routine	75
	4.5 Justification Routine	76
	4.6 Initialization Routine	77
	4.7 Consistency Routine	77
	4.8 Definition of D-Frontier	77
	4.9 Initialization Module	77
	4.10 Preprocessing	79
	4.11 Unsolvable States	80
	4.12 Solvable States	81
	4.13 Backtrack State Event	82
	4.14 Flow Charts for STIM GEN	83
5	FAULT SIMULATION	92
	5.1 Introduction	92
	5.2 Simulation System Structure	95
	5.3 Data Structure for Circuit	96

Section **Page**

5.4 Simulator Data Structure 97

5.5 Simulator Operation 101

5.6 Fault Insertion 105

5.7 Initialization 106

5.8 Data for Primary Inputs 106

5.9 Input Skew Buffering 106

5.10 Miscellaneous Items 106

NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

ABSTRACT

This is the third and final report documenting our research into the design of a new and powerful automatic test generation system for digital logic, called TEST/80. This system is designed around the following six concepts.

1. A powerful circuit preprocessing analysis which leads to greater efficiency during stimulus generation.
2. An effective initialization algorithm.
3. The use of time frames, phases and periods so that asynchronous circuits can be accurately processed during stimulus generation.
4. The use of functional level primitives so that complex circuits including shift registers, counters and RAM's, can now be effectively processed.
5. The use of a stimulus generation algorithm which incorporates the concepts in 1 - 4.
6. The use of a functional level concurrent fault simulator, used to "grade" a test and produce a fault dictionary.

This system is intended to replace existing test generation systems used by the United States Navy, and meet the Navy's needs for automatic test generation during the next six to ten years. At that time it is believed that circuit densities will have made the use of such a program infeasible. The report consists of the results of our research into the solution of the six concepts previously presented, and does not represent a detailed design of TEST/80.

1. INTRODUCTION

This report documents our results on the conceptual design of a very powerful and efficient automatic test generation program (ATGP) software system for digital circuits. The general structure of such a system is shown in Figure 1.1. This structure is quite universal, and is used by most existing systems such as LOGOS, LASAR, FAIRTEST, L-LOGIC, and INDICATES. The input to these systems is a manually generated circuit model. The model is then preprocessed, where syntax errors are detected, library models are substituted for calls, and fault collapsing is carried out, etc.

In the STIM GEN module test vectors are generated; this can be done either manually, by random technique, or via some heuristic or algorithmic procedure.

In the simulation module the test just generated is evaluated, i.e., the set of all faults detected is determined. If the percent detection exceeds some bound, then a dictionary can be generated and a test tape produced. If the percent fault detection is too low, more tests are generated.

There are numerous reasons why existing systems fail to be effective for present day circuits. These problems will be even more severe as more and more LSI devices continue to be used.

Some of these problems are:

1. Existing ATGP systems fail to initialize circuits which are initializable.
2. Existing ATGP systems cannot effectively process circuits containing RAM's and large counters and shift registers. This

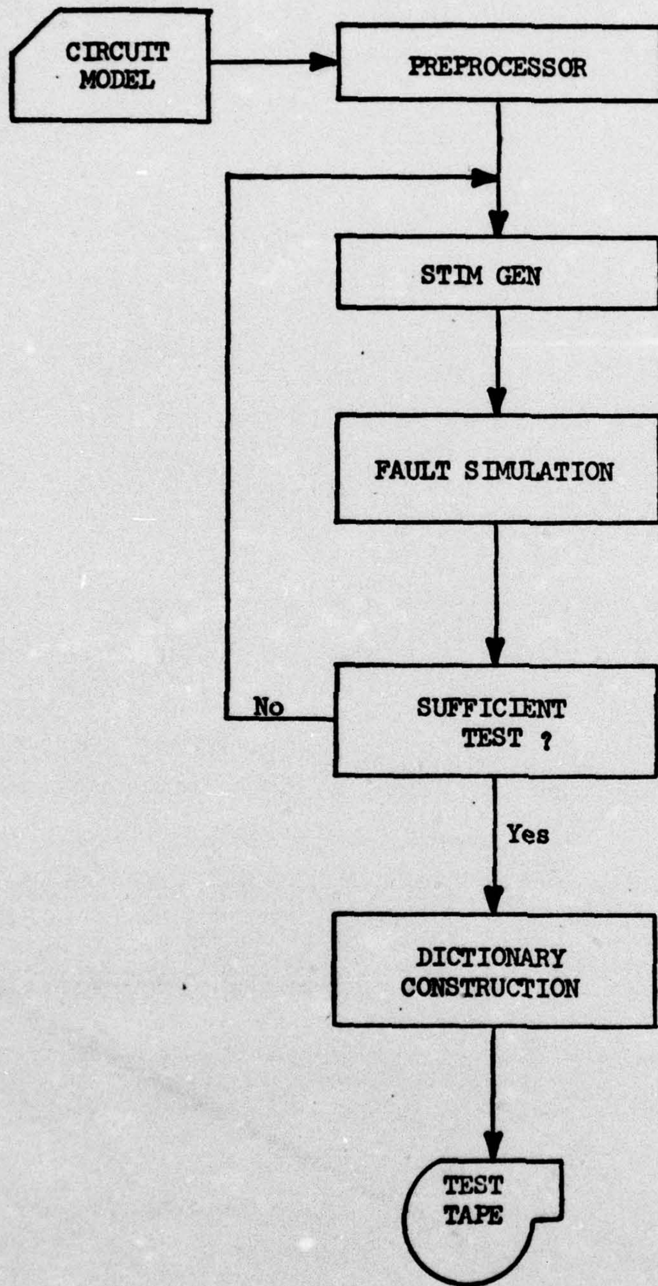


FIGURE 1.1
GENERAL STRUCTURE OF AN 'ATGP' SYSTEM

problem manifests itself by: a) not generating tests for faults, and simultaneously b) using up a tremendous amount of CPU time in searching for a test.

3. Existing ATGP systems cannot effectively handle asynchronous circuits.

4. Some existing ATGP systems use low-level logic primitives and this, together with 3 above, require complex modeling and work-arounds in order to generate the input circuit model.

TEST/80 has been designed to eliminate and/or drastically reduce the problems listed above. The basic concepts employed in TEST/80 are:

1. Functional level modeling.
2. The use of time frames, phases and periods to handle the generation of tests for asynchronous circuits.
3. The use of a powerful initialization process so that complex circuits can be automatically initialized.
4. The use of new sophisticated preprocessing concepts to reduce CPU time during stimulus generation.
5. The use of a functional-level concurrent-driven fault simulator.

Preliminary hand calculations indicate that for complex circuits the potential performance of this system will be significantly greater than existing systems. The improvement in performance is a function of the structure of the circuit to be tested, and hence, no absolute number can be stated. For combinational logic, our system will actually be a little slower than existing systems because of the overhead in our preprocessing

and complex data structures. For circuits having counters and complex decoding circuitry, our system can reduce a 5-hour (1108) test generation problem to a 30-minute (1108) test generation problem.

This report is the third and final report on TEST/80 to be published under the present contract. The previous two reports are:

1. TEST/80, Report No. 1-76: "Functional Level Modeling of Complex Elements in TEST/80."
2. TEST/80, Report No. 2-76: "Initialization of Digital Logic Circuits."

The first report presents our results for developing functional level models for complex sequential devices to be used by our stimulus generation and simulation modules.

The use of these high-level primitive models reduces CPU time during test generation. If one is interested in an internal IC fault, the module containing the fault is modeled at a lower level such that the fault is external to the primitive elements in the model. The remaining elements in the circuit can still be modeled at the higher level.

The second report deals with the development of new and powerful algorithms for carrying out automatic initialization of circuits. Unfortunately, for some complex circuits these procedures may not be computationally feasible. We, in these instances, plan to use classical initialization techniques based upon three (0,1,u)-valued logic.

The present report deals with the following four major areas:

** Preprocessing

Rate analysis

Cost analysis

** Timing

** Test Generation Algorithm

** Functional level concurrent fault simulation.

We only deal with the "research aspects" of the TEST/80 system, i.e., with those problems investigated and solved during the course of our research.

We have not included any discussion of the classical components of an ATGP system whose solutions are well known, such as fault collapsing, dictionary construction, etc.

2. PREPROCESSING

2.1 Rate Analysis

Rate analysis is a preprocessing technique in which certain lines in a circuit are assigned labels, called rates. These labels indicate the maximum rate at which a line can change values. These rates are used during test generation in one of two ways.

1) Logic values can be assigned to lines consistent with the maximum rate, and

2) sequences which cannot be justified can be identified.

Let a and b be rates. Then the following are rates:

1) 0 and 1

2) ab (concatenation)

3) $a^n = \underbrace{a.a \cdots a}_{n \text{ times}}$

4) (a)

5) $a^* = a.a \cdots$ (indefinite)

6) r (a random bit)

7) r^n (a random sequence of 0's and 1's).

Examples:

a) $(01)^* = 010101\cdots$

b) $(0^2 1^3)^* = 001110011100111\cdots$

c) $(0^2(01^2)^2)^* = 0001101100011011\cdots$

Normally, when no confusion will occur, we do not indicate the *.

If rate a has length ℓ then we denote this fact by $a[\ell]$.

Examples:

a) $0^3 1^4 [7]$

b) $(0^2(01^2)^3[9])^2$ [22].

A rate of the form $e^{n_1}\bar{e}^{n_2}$ is of order 1, where $e \in \{0,1\}$. If rates a and b are of order i and j respectively, then rate ab is of order i + j. A rate indicates the maximum rate at which a signal on a line can change values. We desire to approximate these maximum rates by simple rates of low order, say, of order 1, 2, or 3.

If A is a signal line, then the rate associated with the line is denoted by A_r.

Expansion:

Let b be a rate. Then the expansion of b, denoted by E(b), is also a rate, and is obtained by replacing any element $x \in \{0,1\}$ in b by x^n for an arbitrary $n > 0$. Note that the expansion operator can be used repeatedly.

Example: $$E(0^2 1) = 0^3 1^4 .$$

We say that b is less than E(b), denoted by $b \leq E(b)$, for all expansions of b.

<u>Compatible</u>:

Let a and b be two rates. Then a is <u>compatible</u> with b if there exists an expansion of b, denoted by E(b), such that a = E(b). If a is compatible with b, we write $b \leq a$.

Examples:

a) $a = 0\,0\,1\,0\,1$ is not compatible with $b = 0^2 1^3$.

b) $a = 0^2 1^3$ is compatible with $b = 0\,0\,1\,0\,1$. This follows because b can be expanded as follows:

$$b = 0\,0\,1\,0\,1$$

$$0\,0\,1\,1\,1\,0\,0\,1\,1\,1 = (0^2 1^3)^2 \quad .$$

Given a rate a, then b is the greatest lower bound of order i of a if

a) b is of degree i

b) a is compatible with b

c) there exists no other rate c of degree i such that b is compatible with c, i.e., there exists no c such that

$$c < b \leq a \quad .$$

Consider the rate A_r shown below for line A.

$$A_r = (000\underbrace{11}_{2}\underbrace{000}_{3}111000011100011000 11) \quad .$$

Since the minimum number of consecutive 1's and 0's is 2 and 3,

respectively, the greatest lower bound for A_r is

$$A_r' = 0^2 1^3 \ .$$

If one tries to justify the sequence $\underline{A} = 0\ 0\ 1\ 0\ 1$, since $E(A_r') \neq \underline{A}$ for any expansion, $\underline{A}$ cannot be justified.

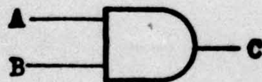
The <u>maximum rates</u> are 01 and 10, and the minimum rates are 0^* and 1^*, namely a constant. We assume all primary imputs have maximum rate, unless specified otherwise.

The <u>complement</u> of a rate a is obtained by changing all 0's to 1's and 1's to 0's, and is denoted by $\bar{a}$.

The AND and OR of two rates a and b are denoted by $a \cdot b$ and $a + b$, respectively, and are also rates.

<u>Rules for Propagation of Rates Through Gates</u>

1. AND

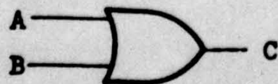


$$C_r = A_r \cdot B_r$$

If $A_r = 1$ then $C_r = B_r$.

If $A_r = 0$ then $C_r = 0$.

2. OR

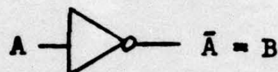


$$C_r = A_r + B_r$$

If $A_r = 0$ then $C_r = B_r$

If $A_r = 1$ then $C_r = 1$.

3. Inverter



$B_r = (\overline{A_r})$

Procedure AND: Computing $Z_r = X_r \cdot Y_r$

Let

$$X_r = x^{n_1} \bar{x}^{n_2} x^{n_3} \cdots$$

and

$$Y_r = y^{m_1} \bar{y}^{m_2} y^{m_3} \cdots$$

where

$$n_1 \geq m_1 .$$

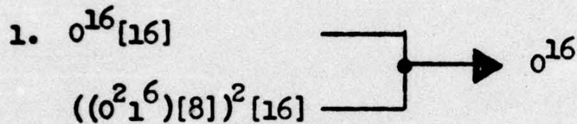
Find a prefix of Y_r of length n_1. If $x = 0$ then output 0^{n_1}, else output the prefix. Delete x^{n_1} from X_r and the prefix from Y_r. If processing a term of the form $(\cdots)^n$, then take the result produced, say p, and replace it by $(p)^n$.

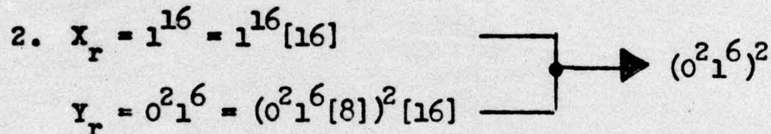
Return to the new X_r and Y_r rates, relabel if necessary, and repeat the procedure until both have been processed to the end. In order to generate low order rates, one can restrict this procedure to the case where $n_1 = m_1$ or $m_1 + m_2$ or $m_1 + m_2 + m_3, \ldots$

Example: Given

$$X_r = 0^{16}1^{16} \quad \text{and} \quad Y_r = 0^2 1^6$$

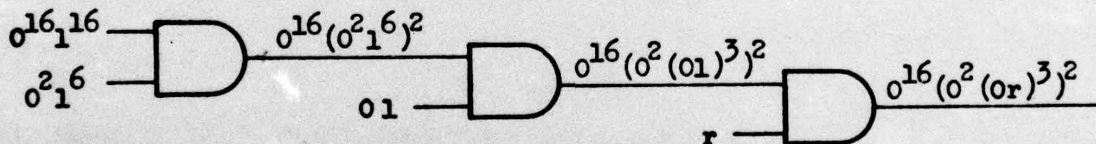
compute $$Z_r = X_r \cdot Y_r \ .$$





$$\therefore \ Z_r = 0^{16}(0^2 1^6)^2 \ .$$

Example:



The output can be approximated by the first-order rate $0^{16}\ r^{16}$.

Creation of Rates

Rates are usually created at the output of sequential devices, such as flip-flops, counters, shift registers, etc. Normally, we select the input to the device to have rates such that the outputs are maximum.

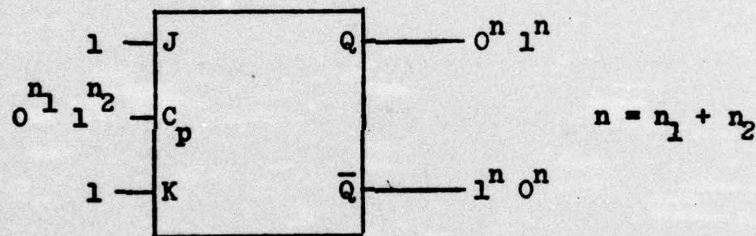
Consider a device triggered by a positive edge on the clock line. We denote this condition by $C_p(\uparrow)$. If $(C_p)_r = 0^{n_1} 1^{n_2}$, then the rate of the positive edge is

$$((C_p(\uparrow))_r = 0^n 1^n \quad \text{where} \quad n = n_1 + n_2$$

is the period of the clock pulse.

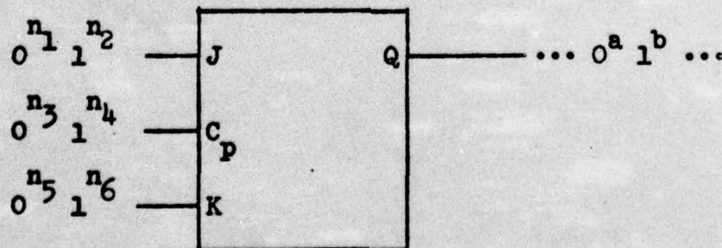
JK edge triggered flip-flop:

1. Maximum rate configuration



Note: For $n_1 = n_2 = 1$, $n = 2$ and $Q_r = 0^2 1^2$. This illustrates the fact that this configuration represents a divide by 2 counter.

2. General configuration:

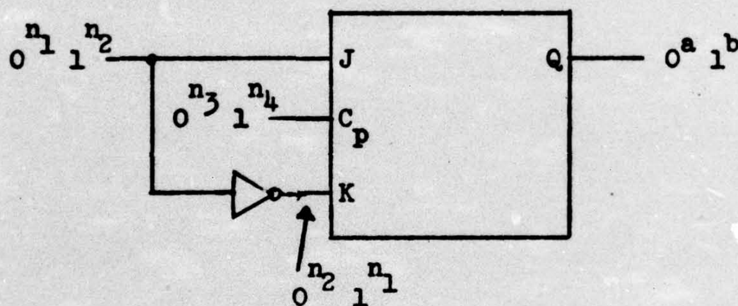


1) $a,b \geq n_3 + n_4$, i.e., the output cannot change faster than the input period

2) $a \geq n_1$, i.e., as long as J = 0 the F/F cannot be set

3) $b \geq n_5$, i.e., as long as K = 0 the F/F cannot be reset.

3. Special case (JK F/F acting as a delay flip/flop):

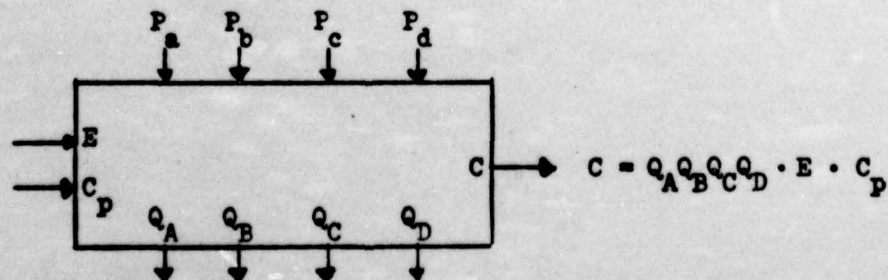


$a,b \geq n_3 + n_4$

$a \geq n_1$

$b \geq n_2$.

Counter:



1. Count mode

$$E_r = 1$$

$$(C_p)_r = 0^n 1^m$$

$$q = n_1 + n_2$$

$$(Q_A)_r = 0^q 1^q$$

$$(Q_B)_r = 0^{2q} 1^{2q}$$

$$(Q_C)_r = 0^{4q} 1^{4q}$$

$$(Q_D)_r = 0^{8q} 1^{8q}$$

$$C = 0^{16q-m} 1^m \quad .$$

2. Maximum count mode

Set $n = m = 1, \quad q = 2.$

3. Special case

$$E_r = 0^{n_1} 1^{n_2} \quad .$$

$$(C_p)_r = 01$$

$$n = n_1 + n_2, \quad n_2 \geq 2 \quad .$$

Then

$$(Q_A)_r = 0^n 1^n$$

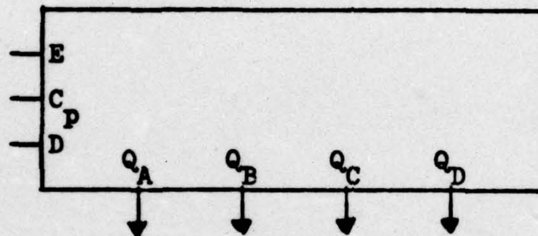
$$(Q_B)_r = 0^{2n} 1^{2n}$$

etc.

4. Parallel load mode: $(i \in \{A, B, C, D\})$

$(Q_i)_r = \overline{L}_r \cdot Z_r$ where Z_r is the rate of Q_i when operated as a delay flip-flop with input P_i and clock C_p.

Shift register:



$(Q_i)_r = E_r \cdot Z_r$ where Z_r is the rate of a delay flip-flop having input D and clock C_p.

Rate Assignment Procedure

The following procedure assigns rates to the lines in a circuit:

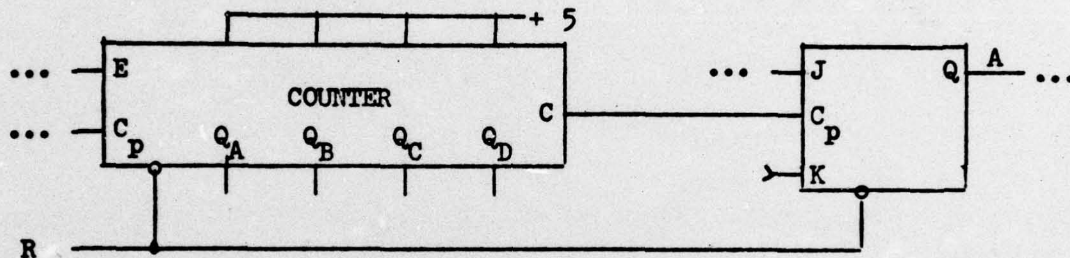
1. Sequentially process clocked elements.
2. Given an element, for those inputs not yet assigned a rate, assign rates to achieve maximum output rates.
3. Compute output rate of device.
4. Propagate rates through combinational logic as long as lower order rates can be preserved.
5. Return to the beginning and process the rest of the devices.

If the input to a sequential device is given a new rate, reprocess the element.

Applications of Rate Analysis

The major application of rate analysis is the identification of allowable sequences on lines.

Example:



Assume E, C_p and J are driven by large combinational circuits and hence their rates are not known. Let K be a primary input.

We assign rates as follows: For the counter, the maximum rate inputs are $E_r = 1$ and $(C_p)_r = 01$. This gives us $C_r = 0^{31}\,1$. For the JK flip-flop, the maximum rate inputs are $J = K = 1$, and since $(C_p)_r = 0^{31}\,1$, we have that $Q_r = 0^{32}\,1^{32}$.

Assume that during the generation we desire to test for the fault A s-a-0. To solve this problem we have the following subproblems:

1. Set $A = 1$
2. To solve 1 we have subproblems:

   a) t: $C_p = \uparrow$ $(C_p(t) = 1, C_p(t-1) = 0)$

   b) t-1: $J = K = 1$, $Q = 0$, $C_p = 0$.

3. To solve the problem $Q = 0$ we have:

   c) $t' < t-1$: $R = 0$.

4. By implication, we see that $R = 0$ also resets the counter and we have $C = 0$.

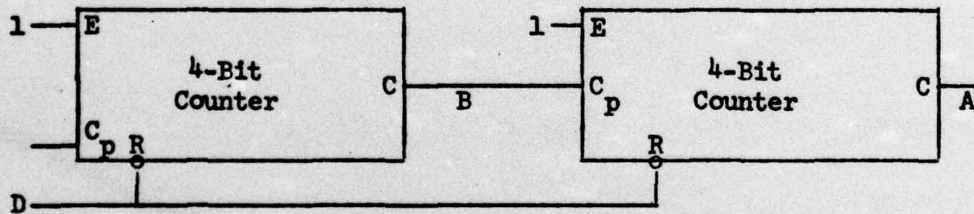
In summary we have,

Time:	t'	...	(t - 1)	t
Line C:	0	...	0	1

But rate analysis implies that $C_r = 0^{31}\,1$. Therefore, for $t' = 1$, we have $(t-1) = 31$ and $t = 32$. Therefore a 32 clock time test is required.

Example:

This example shows how our functional modeling language interacts with rate analysis to give information for test construction.



(T = translator)

To set $A = 1$ we require:

$R(H^n\,U)^4$ (reset and count up 4 times) (1)

$U \xrightarrow{T} \{E = 1,\, C_p = \uparrow\}$

$$H \xrightarrow{T} \{E = 0 \text{ (impossible)}; \; E = 1, \; E_p = \overline{0\,1}\,\}$$

$$R \xrightarrow{T} \{D = 0\}$$

Substituting back into (1), we get for the line C_p:

$$C_p : R(H^n\,U)^4 \;=\; \times((\overline{0\,1})^n \uparrow)^4$$

Since $\uparrow = (0\,1)$ and by rate analysis we have that $(C_p)_r = 0^{31}\,1$, we obtain $(\overline{0\,1})^n = 0^{30}$ and hence, $C_p = \times\, 0^{30}\, 0\, 1$.

Another application of rate analysis is the following: Assume that in order to get a test for a fault we assign line A the sequence $\underline{A}$. Now, if $\underline{A}$ is not compatible with A_r, then $\underline{A}$ cannot be justified and the test generation procedure should backtrack and take another choice.

2.2 Cost Assignments

In preprocessing a circuit we assign to each line A in the circuit three costs, namely,

cA - the cost of setting line A to a 1

c$\bar{A}$ - the cost of setting line A to a 0

and

dA - the cost of driving a D ($\bar{D}$) on line A to a primary output.

These costs are used by our test generation algorithm during D-drive and line justification. In this section we discuss how the preprocessor

assigns these costs. In general, problems of higher cost are more difficult to solve.

The cost cA is given by the equation

$$cA = \min (cfA + csA + cdA , K)$$

where A is the output of an element of type t and K is an input parameter usually taken to be 32,000. The term cfA is that cost contribution due to the logical properties of t; csA is that cost distribution due to the side effects of setting A to 1, and cdA is a constant cost associated with the element type t. Usually we set cdA to the values shown below:

cdA	t
2	gates
4	flip-flops
4n	n-bit registers

A similar equation holds for $c\bar{A}$.

Calculation of cfA and $cf\bar{A}$

Gate functions:

Consider a gate having input lines $i = 1,2,\ldots,n$. Let the output be A. Then we have the following results.

AND gate

$$A = 1 \cdot 2 \cdots n$$

$$cfA = \sum_{i=1}^{n} ci$$

$$\bar{A} = \bar{1} + \bar{2} + \cdots + \bar{n}$$

$$cf\bar{A} = \min_{i} \{c\bar{i}\}$$

NAND gate

Interchange A and $\bar{A}$

OR gate

$$A = 1 + 2 + \cdots + n$$

$$cfA = \min_{i} \{ci\}$$

$$\bar{A} = \bar{1} \cdot \bar{2} \cdots \bar{n}$$

$$cf\bar{A} = \sum_{i=1}^{n} c\bar{i}$$

NOR gate

Interchange A and $\bar{A}$

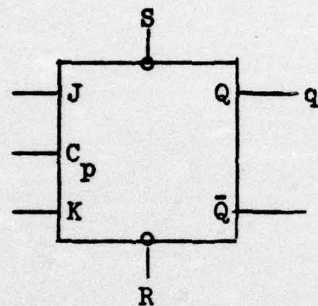
We represent a sequence of input vectors by $X(1), X(2), X(3), \ldots$.

Then the cost of this sequence is the sum of the cost of each vector.

Also, since $C_p(\uparrow) = (C_p = 0),(C_p = 1)$, then

$$cC_p(\uparrow) = c\bar{C}_p + cC_p \ .$$

Flip-flop (JK positive edge triggered; S,R asynchronous):



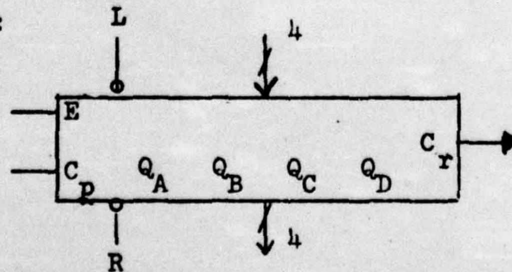
$$Q^+ = J(\bar{Q} + \bar{K})\ SR\ C_p(\uparrow) + \bar{S}R$$

$$\bar{Q}^+ = K(Q + \bar{J})\ SR\ C_p(\uparrow) + S\bar{R}$$

$$\begin{aligned} cfQ = \min\ (&c\bar{S} + cR, && \text{direct set} \\ &cJ + c\bar{K} + 2cS + 2cR + cC_p + c\bar{C}_p, && \text{set F/F} \\ &cJ + c\bar{Q} + 2cS + 2cR + cC_p + c\bar{C}_p) && \text{set or trigger} \end{aligned}$$

$$\begin{aligned} cf\bar{Q} = \min\ (&c\bar{R} + cS, \\ &cK + c\bar{J} + 2cS + 2cR + cC_p + c\bar{C}_p, \\ &cK + cQ + 2cS + 2cR + cC_p + c\bar{C}_p) \ . \end{aligned}$$

Counter:



$$Q_A = (\bar{R}L, RLE\ C_p(\uparrow))$$ reset and increment

$$+ R\bar{L}\ P_A$$ load a 1

$$+ RLE\ C_p(\uparrow)\bar{Q}_A$$ $Q_A = 0$ and increment

$$\bar{Q}_A = \bar{R}L$$ reset

$$+ R\bar{L}\ \bar{P}_A$$ load a 0

$$+ RLE\ C_p(\uparrow)Q_A$$ $Q_A = 1$ and increment .

Therefore

$$cfQ_A = \min\ (c\bar{R} + 3cL + 2(cR + cE) + cC_p + c\bar{C}_p,$$

$$cR + c\bar{L} + cP_A,$$

$$2(cR + cL + cE) + cC_p + c\bar{C}_p + c\bar{Q}_A)\ ;$$

$$Q_B = R\bar{L}\ P_B$$ load a 1

$$+ RLE\ C_p(\uparrow)Q_A$$ $Q_A = 1$ and increment ;

$$\bar{Q}_B = \bar{R}L$$ reset

$$+ R\bar{L}\ \bar{P}_B$$ load a 0

$$+ RLE\ C_p(\uparrow)(Q_A Q_B)$$ $Q_A = Q_B = 1$ and increment;

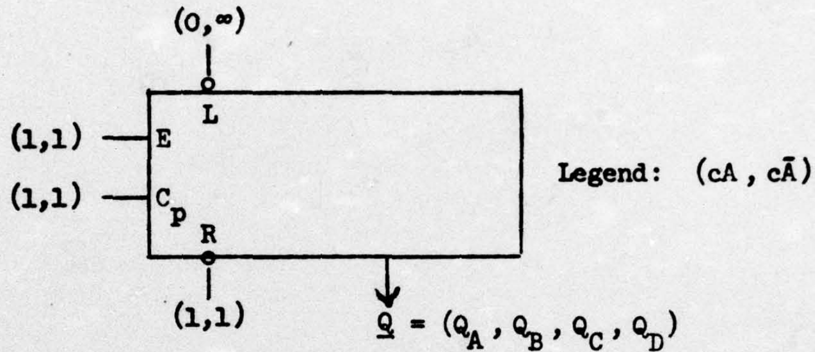
etc.

$$C_r = E\ C_p\ Q_A\ Q_B\ Q_C\ Q_D$$

$$cfC_r = cE + cC_p + cQ_A + cQ_B + cQ_C + cQ_D\ .$$

Example:

Consider a counter having the input costs as shown below. For simplicity we let $cQ = cfQ$ for $Q = A$ and $\bar{A}$.



$$cQ_A = \min(1 + 3\cdot 0 + 2(1 + 1) + 1 + 1,\ 1 + \infty \cdots,\ 2(1 + 0 + 1) + 1 + 1 + c\bar{Q}_A)$$

$$cQ_A = \min(7, 6 + c\bar{Q}_A)$$

$$c\bar{Q}_A = \min(1 + 0,\ 1 + \infty + \cdots,\ 2(1 + 1 + 1) + 1 + 1 + cQ_A)$$

$$= \min(1, 8 + cQ_A)$$

$$= 1$$

$$cQ_B = \min(1 + \infty + \cdots,\ 2(1 + 0 + 1) + 1 + 1 + cQ_A)$$

$$= 6 + cQ_A$$

$$c\bar{Q}_B = \min(1 + 0,\ 1 + \infty + \cdots,\ 2(1 + 0 + 1) + 1 + 1 + cQ_A + cQ_B) = 1 .$$

In summary, we have

$$cQ_A = \min(7, 6 + c\bar{Q}_A) \quad (1)$$

$$c\bar{Q}_A = 1 \quad (2)$$

$$cQ_B = 6 + cQ_A \quad (3)$$

and

$$c\bar{Q}_B = 1 . \quad (4)$$

Substituting (2) into (1) we get

$$cQ_A = 7 \quad (5)$$

and substituting (5) into (3) we obtain

$$cQ_B = 13 . \quad (6)$$

To verify this result, we see that to set $Q_B = 1$ we must reset the counter $(\bar{R}L)$, and count up two times, i.e.,

$$(RLE\ C_p(\uparrow))^2 .$$

This gives us a cost of

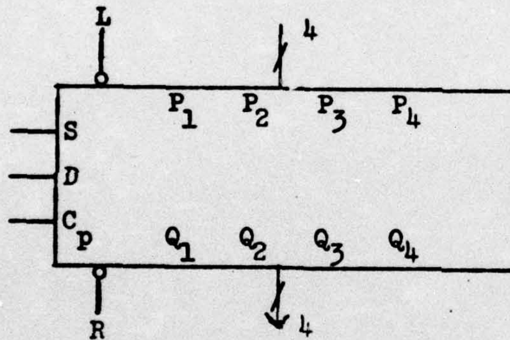
$$c(\bar{R}L) = 1 + 0 = 1$$

and

$$c(RLE\ C_p(\uparrow))^2 = 2(2(1 + 0 + 1) + 2)$$
$$= 12\ ,$$

hence a total cost of 13.

Shift register:



Legend:

Device resets on R = 0.

Device loads on L = 0.

Device shifts right on S = 1 and $C_p(\uparrow)$.

Data in is D.

$Q_i = RLS\ C_p(\uparrow)Q_{i-1}$ shift in a 1

$R\bar{L}\ P_i$ load a 1

$\bar{Q}_i = \bar{R}L$ reset

$RLS\ C_p(\uparrow)\bar{Q}_{i-1}$ shift in a 0

$R\bar{L}\ \bar{P}_i$ load a 0

where

$Q_0 = D.$

$$cfQ_i = \min\left(2(cR + cL + cS) + cC_p + c\bar{C}_p + cQ_{i-1},\ cR + c\bar{L} + cP_i\right)$$

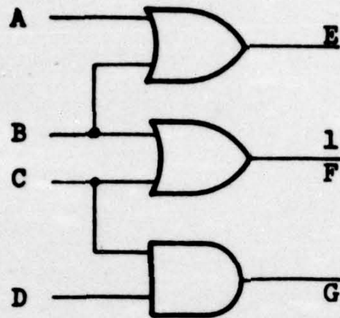
$$cf\bar{Q}_i = \min\left(c\bar{R} + cL,\ 2(cR + cL + cS) + cC_p + c\bar{C}_p + c\bar{Q}_{i-1},\ cR + c\bar{L} + c\bar{P}_i\right) .$$

Computation of Side Effects Factor csA

When a line is set to a 0 or 1, due to its fan-out this line setting may affect many parts of a circuit. We refer to this phenomenon as side-effects.

Example:

Consider the circuit shown below.



To justify $F = 1$, if we set $B = 1$ we force $E = 1$. But, if we justify $F = 1$ by setting $C = 1$, we do not force G to a logic value. Hence, we desire $cB > cC$. That is, we penalize B because of its side-effects.

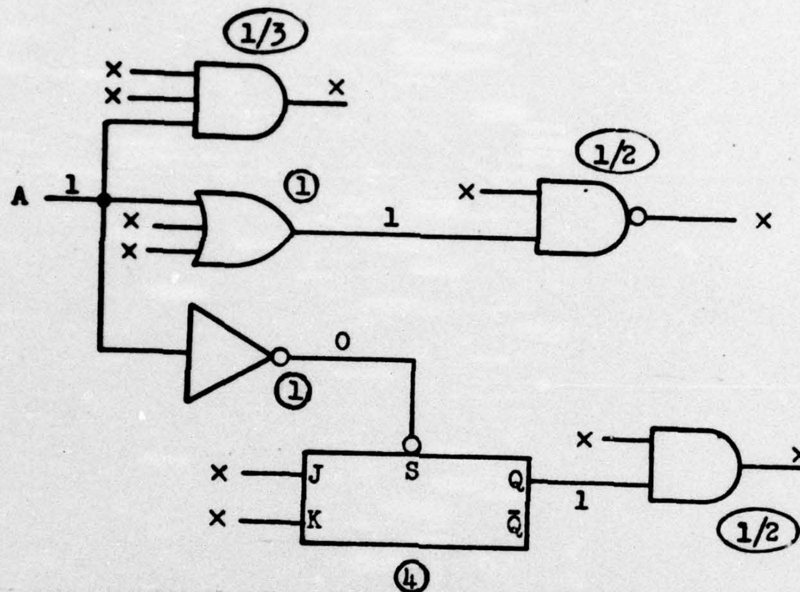
The value of csA can be computed as follows. Let line A have logic value $\delta \in \{0,1\}$. Simulate the circuit with the initial condition $A = \delta$ and all other lines at $\times$. The contributions to csA are as follows:

1) a line B (output of a gate) set to 0 or 1 contributes 1;

2) a line B at $\times$ having m binary input has a contribution of m/n, where n is the number of inputs to gate B;

3) if a flip-flop B is set, or reset, its contribution is 4.

Legend: (s) - contribution to side-effect factor csA



$$csA = \frac{1}{3} + 1 + 1 + \frac{1}{2} + 4 + \frac{1}{2} = 7\frac{1}{3} .$$

Side Effects in an n-Bit Counter

THEOREM 1: When an n-bit counter is incremented, the expected number $E_I(n)$ of flip-flops which change state is given by the expression:

$$E_I(n) = E_I(n-1) + \frac{1}{2^{n-1}}$$

$$= \frac{2^n - 1}{2^{n-1}} = 2 - \frac{1}{2^{n-1}} .$$

We tabulate this result below:

n	$E_I(n)$
1	1
2	3/2
3	7/4
4	15/8
5	31/16
⋮	⋮
n	$(2^n - 1)/2^{n-1}$

Note that:

$$\lim_{n \to \infty} E(n) = 2 \quad ;$$

R = 0: Resets all Flip-Flops

We assume that half the flip-flops change state when the counter is reset, hence the cost contribution to csR is

$$\frac{n}{2} (4 + G)$$

where G is the average side effect associated with an output line of the counter, and 4 is the cost associated with changing the state of a flip-flop.

L = 0: Parallel Load

We assume that on the average, half of the flip-flops change state when a parallel load occurs. The side-effect's contribution to cL is thus

$$csL = \frac{n}{2}(4 + G) .$$

$E \cdot C_p(\uparrow)RL$: Increments a Counter

When a counter is incremented, $E(n)$ flip-flops may change state. Hence, the side-effect's cost is

$$E(n)(4 + G) .$$

Side-Effects in an n-Bit Shift Register

THEOREM 2: When an n-bit shift register is shifted, the expected number $E_s(n)$ of flip-flops which change state is given by the expression

$$E_s(n) = \frac{n}{2} .$$

The results of 'reset' and 'load' are the same as for the counter. For the case of a right-shift, the side effect's cost is

$$\frac{n}{2}(4 + G) .$$

Using these results, the cost given earlier for the counters and shift registers can now be modified to include the side effect's cost.

The side effects of R and L are already in cR and cL, but we must add the side effects due to the increment process, since this is a function of both E and $C_p(\uparrow)$. Hence, whenever we increment we add the cost, $2(4 + g)$, to the cost given.

Note that primary inputs only have a side effect's cost. Their function cost, cf, is 0.

The costs for a circuit are computed twice, as explained below.

Case 1: Neglect side effects in registers due to master set, reset and load.

Case 2: Use all side effects.

The reason for employing these two cases is the following. When initializing a circuit we do not really care what state the flip-flops are in as long as they are not u, hence using master reset and set is a good strategy, and we should not penalize their use because of their large side effect's cost.

However, in the middle of test generation, if we need some flip-flop to be in the state 0, it could be disastrous to achieve this state via global reset line , since it would also effect all other flip-flop settings. Hence, for this case, side effects are used.

A flow chart for assigning cA and $c\bar{A}$ to all lines in a circuit is given in Figure 2.1, where we assume the rules for calculating the costs for each element type are known.

Computation of D-Drive Costs dA

The cost dA associated with a line A is an estimate of the difficulty of driving an error signal (D) on line A to a primary output (po). The basic strategy for computing dA is shown next.

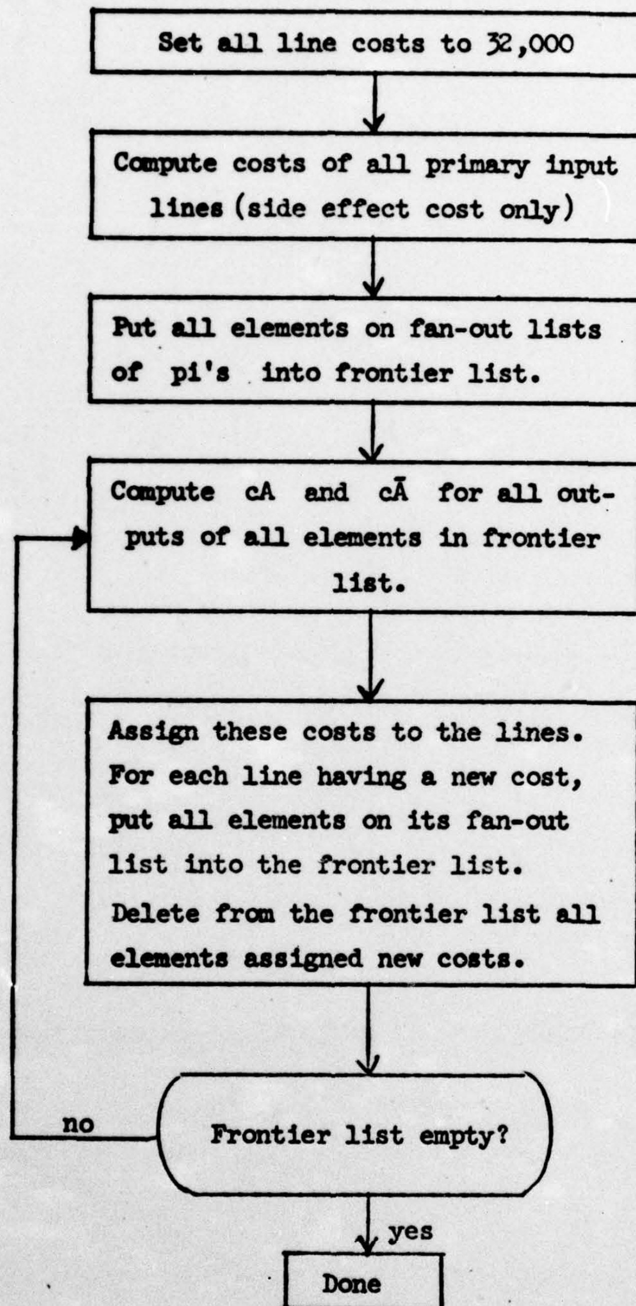
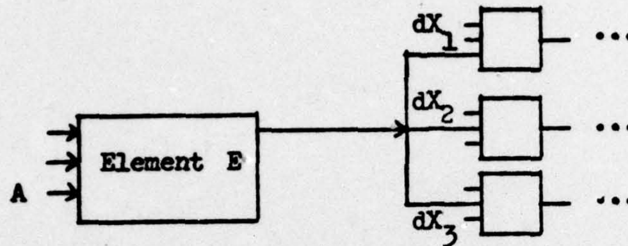


FIGURE 2.1 : Computation of Line Costs cA and cĀ



We have that

$$dOUT = \min(dX_1, dX_2, dX_3) \quad . \tag{1}$$

Then

$$dA = dpA + dOUT + dt \quad . \tag{2}$$

Here we see that A is an input to element E whose output fans-out to three elements.

The D drive costs associated with the output line X from E to these three elements is dX_1, dX_2, and dX_3, respectively. Using formula (2) above, then the cost to drive a D on line A to an output consists of three components, namely:

1) dpA - the cost of propagating a D from A to X, the output of E;

2) dOUT - the minimum cost of propagating the D at X to a po, and,

3) dt - a cost associated with the element type t of E. Usually, dt increases with the number of clock times

required to drive the D at A to X .

1. Primary outputs:

   dA = 0 if A is a primary output .

2. Gate functions:

   AND gate - assume inputs 1,2,...,n, and a D on line i.

Then

$$di = \sum_{j=1,\, j \neq i}^{n} ci + dOUT + 1 \quad .$$

This follows since all other inputs to the gate must be a 1.

   NAND gate - same result

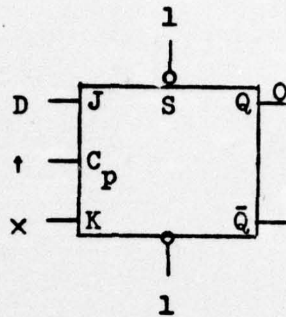
   OR gate

$$di = \sum_{j=1,\, j \neq i}^{n} c\bar{i} + dOUT + 1 \quad .$$

3. JK flip-flop with direct set and reset (negative logic) and positive edge triggering:

Case A) Propagation of J = D to output.

   Below we see



that to propagate a D (or $\bar{D}$) from J to Q or $\bar{Q}$ we need S = R = 1, Q = 0, and a clock pulse. Then if J = 0 we have $Q^+ = 0$, and if J = 1 we have $Q^+ = 1$. Therefore,

$$dJ = \min(32000, c\bar{Q} + 2cS + 2cR + cC_p(\uparrow) + dOUT + 2) .$$

Similarly, we have

$$dK = \min(32000, cQ + 2cS + 2cR + cC_p + dOUT + 2)$$

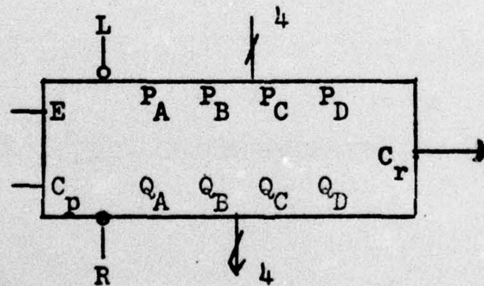
$$\begin{aligned} dS = \min(32000, & \ dOUT + 2 + cR + cS \\ & + \min(c\bar{Q} + cC_p, \\ & \quad c\bar{Q} + 2\bar{C}_p, \\ & \quad c\bar{R}, \\ & \quad cC_p(\uparrow)) \\ & + \min(c\bar{J} + cK, \\ & \quad c\bar{J} + c\bar{Q}, \\ & \quad cR + cK + cQ))) \end{aligned}$$

$$
\begin{aligned}
dR = \min(32000, & \ dOUT + 2 + cR + cS \\
& + \min(cQ + cC, \\
& \qquad cQ + 2cC_p(\uparrow), \\
& \qquad c\bar{\bar{S}}, \\
& \qquad 2C_p(\uparrow)) \\
& + \min(cJ + c\bar{K}, \\
& \qquad cK + cQ, \\
& \qquad cS + cJ + c\bar{Q})))
\end{aligned}
$$

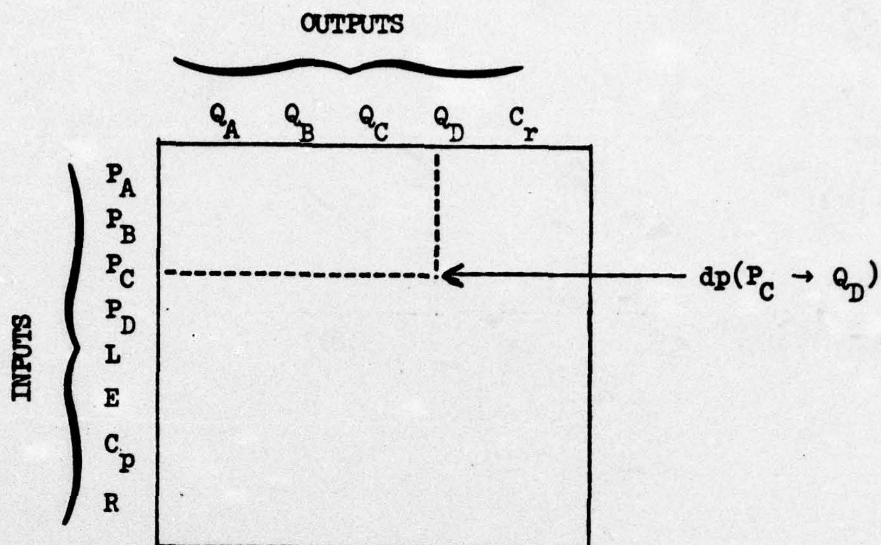
and

$$
\begin{aligned}
dC_p = \min(32000, & \ 2cS + 2cR \\
& + \min(cC_p, c\bar{C}_p) + dOUT + 2 \\
& + \min(cJ + c\bar{Q}, cK + cQ)) \ .
\end{aligned}
$$

4. Counter:



A D on any input line can be propagated to any output. Hence, we can form a matrix of propagation costs shown below. The entry shown is the cost to propagate a D on line P_c to the output Q_D. Once all entries in this matrix have been calculated, a dOUT for each row can be calculated.



The following are illustrations of the derivation of a few of these costs:

$dp(P_A \rightarrow Q_A)$: To propagate a D from P_A to Q_A we need the algorithm LOAD, i.e.

$$C_p(\uparrow)\bar{E}\ R\ \bar{L} \ .$$

Hence

$$dp(P_A \rightarrow Q_A) = cC_p + c\bar{E} + cR + C\bar{L} \ .$$

$dp(P_A \rightarrow Q_B)$: To propagate a D from P_A to Q_B we need the algorithm LOAD, INCREMENT, i.e.

$$C_p(\uparrow)\ R(\bar{E}\bar{L}, EL) \ .$$

Hence,

$$dp(P_A \rightarrow Q_B) = 2cC_p(\uparrow) + 2R + c\bar{E} + c\bar{L}$$
$$+ cE + cL \ .$$

$dp(L \rightarrow Q_i)$: To propagate a D from L to Q_i, $i \in \{A,B,C,D\}$, we must set $Q_i = \delta$, $P_i = \bar{\delta}$, $L = D$, and clock, i.e.

$$C_p(\uparrow)\ \bar{E}\ R(Q_i \neq P_i) \ .$$

Hence,

$$dp(L \rightarrow Q_i)$$
$$= cC_p(\uparrow) + c\bar{E} + cR$$
$$+ \min\ (cQ_i + c\bar{P}_i\ ,\ c\bar{Q}_i + cP_i) \ .$$

$dp(E \rightarrow Q_A)$: Algorithm - INCREMENT, i.e.,

$$C_p(\uparrow)\ \text{"}E = D\text{"}\ RL \ .$$

Hence,

$$dp(E \rightarrow Q_A) = cC_p(\uparrow) + cR + cL \ .$$

Note that once we have $Q_A = D$, we can propagate this D to Q_B via an INCREMENT command, hence:

$$dp(Q_A \rightarrow Q_B) = cC_p + cE + cR + cL \ .$$

Similarly,

$$dp(Q_B \rightarrow Q_C) = cC_p + cE + cR + cL + cQ_A$$

$$dp(Q_C \rightarrow Q_D) = cC_p + cE + cR + cL + cQ_A + cQ_B$$

$$dp(Q_D \rightarrow Q_A) = cC_p + cE + cR + cL + cQ_A + cQ_B + cQ_C \; .$$

These results thus allow us to calculate the cost of

a) propagating a D from an input line into the counter as well as

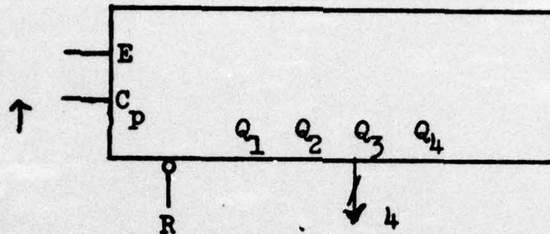
b) propagating a D from one flip-flop to another.

The component dt of the cost dA is given below:

	Q_A	Q_B	Q_C	Q_D	C_r
P_A	2	3	4	5	5
P_B	5	2	3	4	5
P_C	4	5	2	3	5
P_D	3	4	5	2	5
L	2	2	2	2	5
E	2	3	4	5	5
C_p	2	3	4	5	5
R	2	2	2	2	2

dt

5. **Shift register:**



For a shift register we propagate D's into the register via E, C_p, or R. We can propagate a D from Q_i to Q_{i+1} via a shift operation.

$dp(R \to Q_i)$: Algorithm:: First set $Q_i = 1$ and then reset, i.e., $Q_i = 1 ; R$.

$$dp(R \to Q_i) = cQ_i + c\bar{R} \ .$$

$dp(E \to Q_1)$: Algorithm:: For $E = D$, clock, i.e., $C_p(\uparrow) \cdot R$.

$$dp(E \to Q_1) = 2c\bar{R} + cC_p + c\bar{C}_p \ .$$

$dp(C_p \to E)$: Algorithm:: For a D or C_p, enable shift, i.e., $E\bar{R}$, hence

$$dp(C_p \to E) = 2(cE + c\bar{R}) \ .$$

Note that we can propagate a D from Q_i to Q_{i+1} via the

algorithm,

$$E\ C_p(\uparrow)\ R\ ,$$

hence,

$$dp(Q_i \rightarrow Q_{i+1}) = c\bar{C}_p + cC_p + 2(cE + cR)\ .$$

When driving a D from E or C_p to Q_i we use the dt cost given by the equation $dt = i$. Driving a D from R to Q_i has a dt value of 2.

In Figure 2.2 we show the flow chart for computing dA for an entire circuit, where we assume the rules for calculating the dA costs for each element are known.

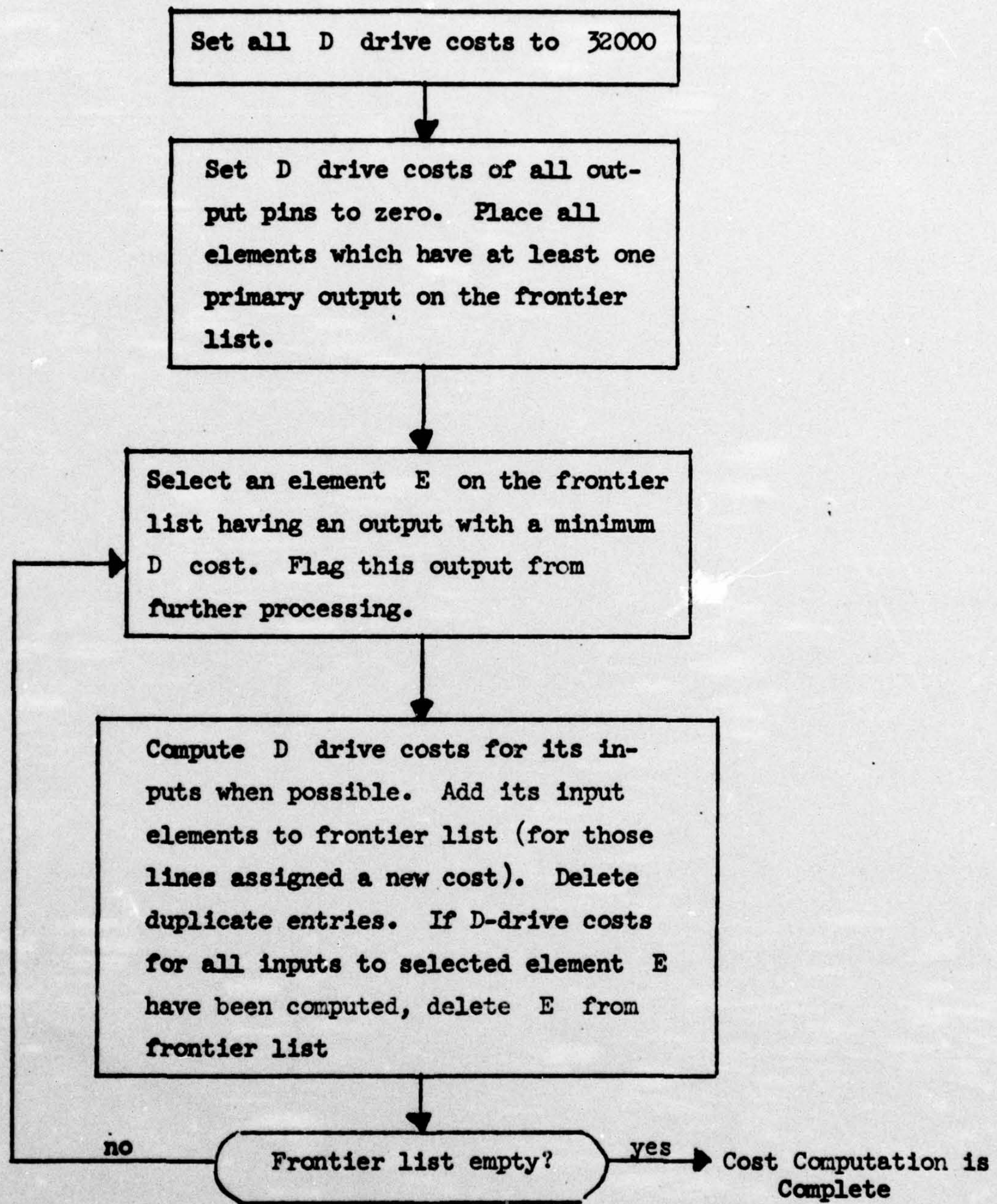


FIGURE 2.2: Computation of D-Drive Costs

3. TIMING

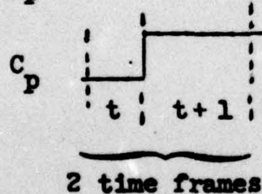
3.1 Basic Definitions on Time

An essential component in TEST/80 is the way time is handled. There are three major units of time used in TEST/80, namely a time frame (TF), a phase (φ), and a period (P). In addition, the concept of a computation step (s) is closely related to time.

Whenever a new input vector is applied to the circuit a new time frame (TF) is defined. We denote consecutive time frames by t, $t+1$, $t+2, \ldots$. Time frames which need not necessarily be consecutive are denoted by t_i, t_j.

Example:

$C_p = \uparrow$ is equivalent to



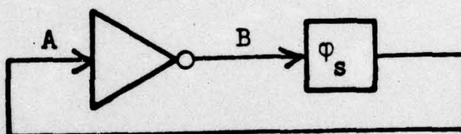
A time frame consists of one or more phases. For practical reasons we limit the number of phases in a single time frame to 1, 2, or 3. The number of phases in a TF is determined dynamically, and changes from time frame to time frame. A line in a circuit can have only one value (static) during any one phase. We denote the phases 1, 2, 3 by φ_1, φ_2, φ_3, and $\varphi_i(t)$ is phase i of TF t.

A phase is the smallest unit of time. During a single phase, elements are assumed to have 0-delay.

A phase element, denoted by —[φ]—, is a device such that if a signal enters it during phase φ_i, the signal exits at phase φ_{i+1}.

Every circuit must be modeled such that each asynchronous feedback loop contains a phase element. There are two types of phase elements, denoted by φ_d and φ_s. A φ_d element introduces delay only. A φ_s element is in a feedback line and represents a state variable, such as in a latch.

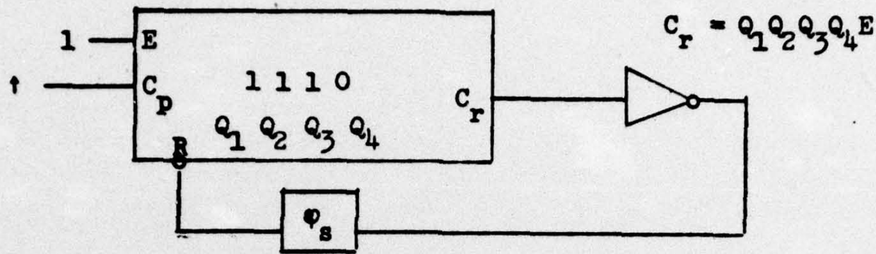
Example 3.1:



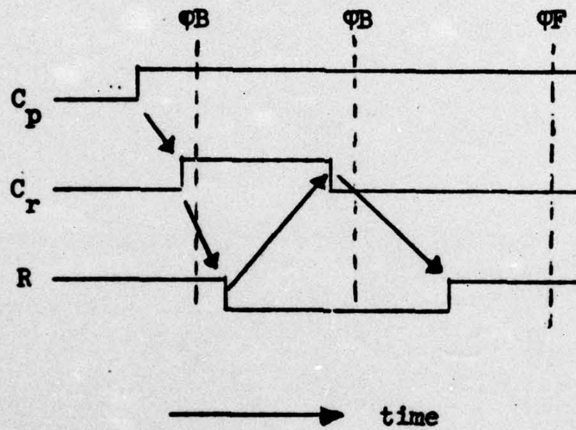
A	B	Phase
0	1	φ_1
1	0	φ_2
0	1	φ_3
1	0	φ_4
	⋮	

This situation (oscillation) is not allowed for in TEST/80 STIM GEN.

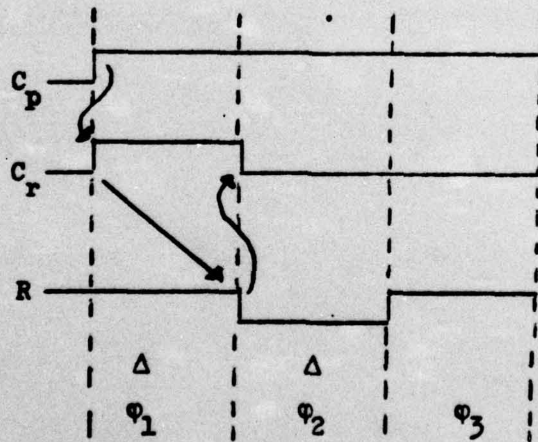
The TEST/80 STIM GEN algorithm allows for a maximum of three phases per TF, hence, for two changes in the value of a line in response to a primary input change. We illustrate this "worse" case situation by the circuit shown in Figure 3.1. Let the initial state of the counter be $Q_1 Q_2 Q_3 Q_4(lsb) = 1110$, $C_r = 0$, $E = 1$ and $C_p = 0$. Let C_p now go to a 1, hence, defining a new time frame.



(a)



(b)



(c)

FIGURE 3.1: Circuit Operation Illustrating the Need for Three Phases.

The following events occur:

The clock causes the counter to increment to state 1111. This forces $C_r = 1$, hence $R = 0$. This, in turn, resets the counter to the state 0000 and hence, $C_r = 0$ and R becomes a 1.

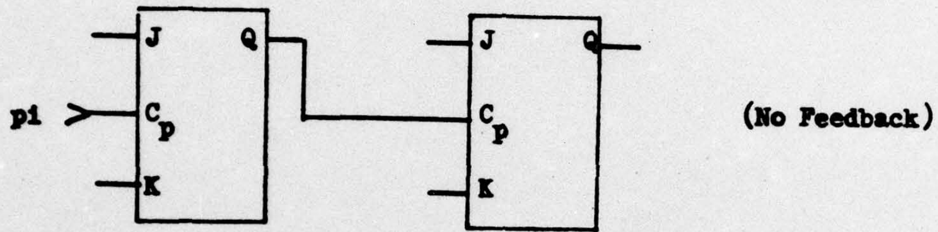
Note the need for the phase element in the feedback loop.

In Figure 3.1(b) we show how the signals may propagate through the circuit as a function of time. Whenever a signal propagates through a phase element, we have what is known as a phase boundary, denoted by φB. The final steady state value defines the phase boundary φF, which, in turn, denotes the end of the time frame. Assume a phase element has a delay of Δ units. Recall that all other elements have 0-delay/phase, hence the "real time" timing diagram of Figure 3.1(b) can be mapped into our model "phase" timing diagram of Figure 3.1(c). Note here that transitions during the same phase occur in 0 time. We see that during the time frame in question C_p changes value once, while C_r and R change twice.

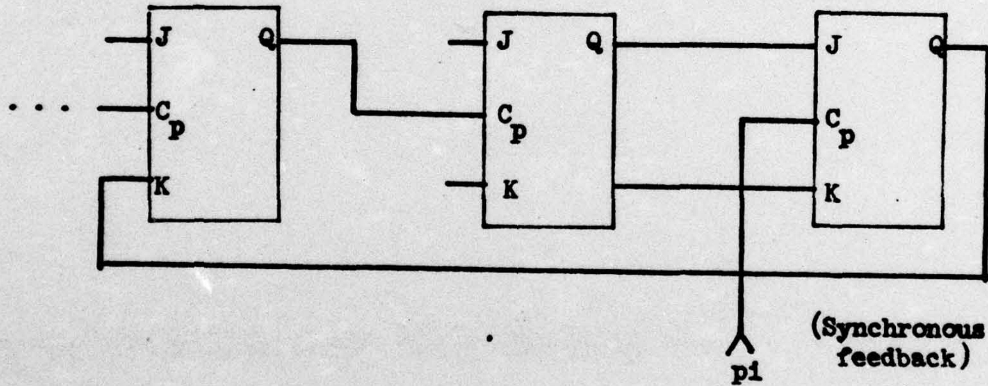
For a synchronous circuit, there is only one phase per TF, and hence, we can ignore the concept of phases and deal only with time frames.

To help clarify the modeling process, in Figure 3.2 we illustrate a few examples where no phase element is needed, and in Figure 3.3 we illustrate a few examples where phase elements are needed. In these examples, C represents an arbitrary combinational circuit.

Phase elements are also used to introduce delays, and hence, change a critical race situation to a noncritical race (see Figure 3.4).

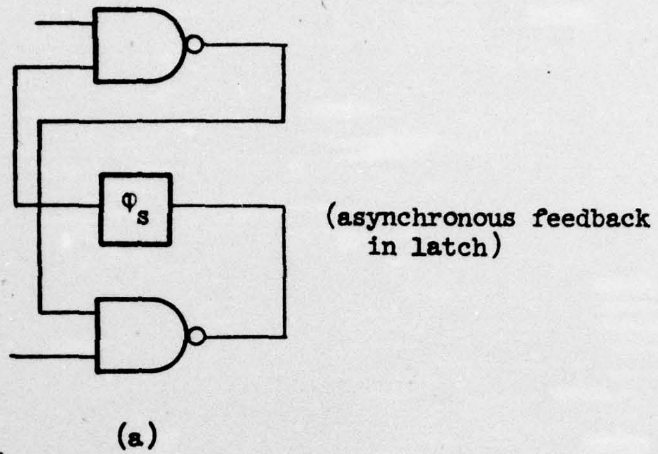


(a)

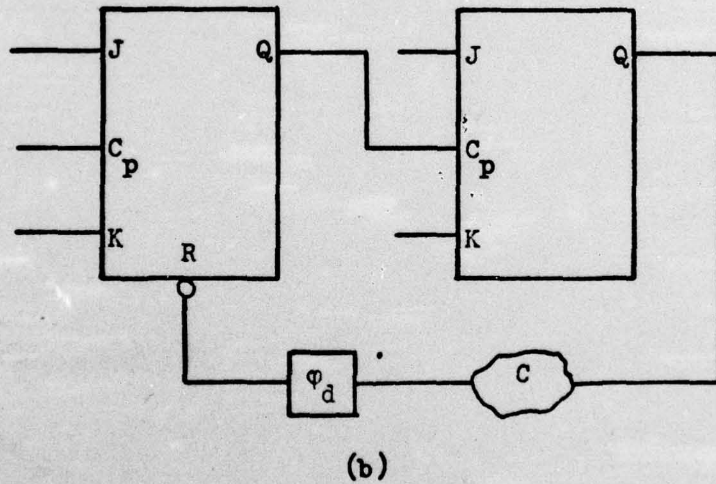


(b)

FIGURE 3.2: Circuit Configurations Where no Phase Elements are Required.

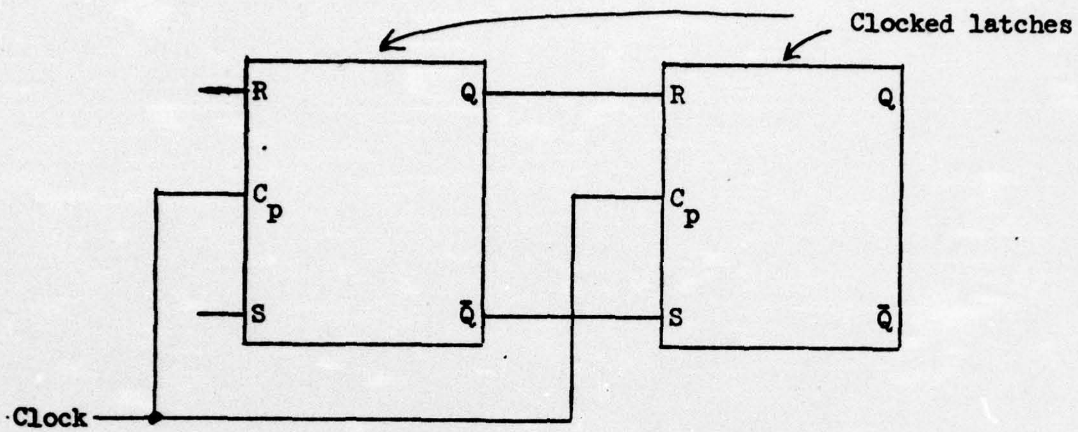


(a)

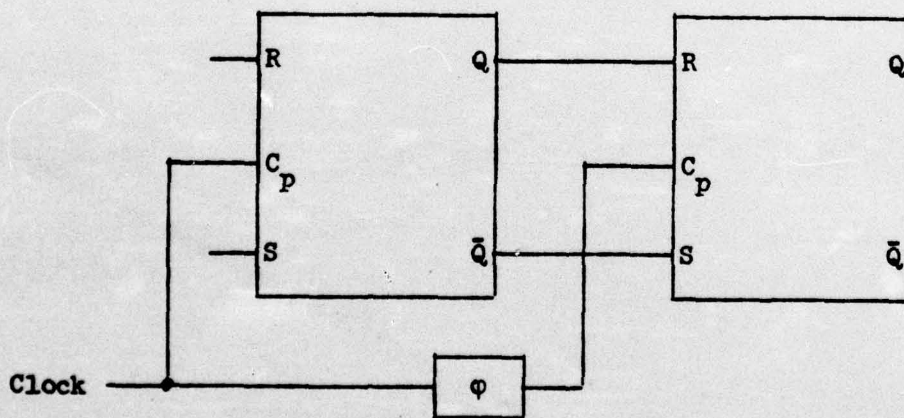


(b)

FIGURE 3.3: Circuit Configurations Where Phase Elements are Required.



(a)



(b)

FIGURE 3.4: (a) Original Circuit; (b) Model.

A period is a unit of time which corresponds to a phase. Sequences in our solution library are stored by period, i.e., if $\underline{X} = x_1, x_2, \ldots$ is a sequence, then x_1 corresponds to period p_i, x_2 to period p_{i+1}, x_3 to period p_{i+3}, etc. During test generation we map periods into phases, which, in turn, get mapped to time frames.

3.2 Problem/Solution-Space

During a phase a line can have only one value, namely, 0,1,u,×,D or $\bar{D}$. Initially, most lines are assigned the value × at the beginning of a phase and as computations are carried out the ×'s are assigned values from the set $\{0,1,u,D,\bar{D}\}$.

In deriving a test of a fault in a circuit, four major operations are carried out, namely, selecting an initial error generating stimuli, D-drive, line justification and implication. Implication is actually carried out simultaneously with the other three concepts.

For every primitive element we have functional algorithms or tables for carrying out these four basic operations. Except for establishing the initial D in the circuit, test generation consists of a sequence of solving two basic problems, namely,

1) drive a 0 through an element, and
2) justify a 0 or 1 at the output of an element.

We denote the problem of justifying line i at value $v \in \{0,1\}$ by the notation $J : i = v$.

We denote the problem of driving a D (or $\bar{D}$) from line i to j by the notation $D : i \rightarrow j$.

If line i is set to the value $v \in \{0,1,u,D,\bar{D}\}$ due to

implication, we denote this fact by $I : i \leftarrow v$.

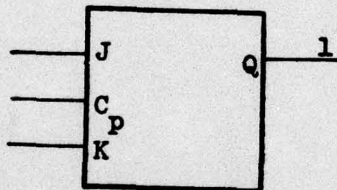
We next illustrate some problem/solution pairs.

A. Example of one-period solution:

Problem: $J : C = 0$

Solution: $A = 0 \quad p_i$

B. Example of two-period solution:



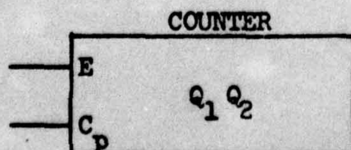
Problem: $J : Q = 1$

Solution:

J	K	C_p	
×	×	1	p_i
1	0	0	p_{i-1}

The quantities p_i and p_{i-1} can be either part of one time frame, or p_{i-1} can be the last phase of time frame t_{j-1}, and p_i the first phase of t_j.

C. Example of multiple-period solution:



Problem : $J : Q_1 = 1, Q_2 = 1$ (assume initial state is $Q_1 = Q_2 = 0$)

Solution: $(E(1\times)\ C_p(\uparrow), H^n)^3$ (functional notation)

Solution with no "holds":

$$(E(1\times)\ C_p(\uparrow))^3$$

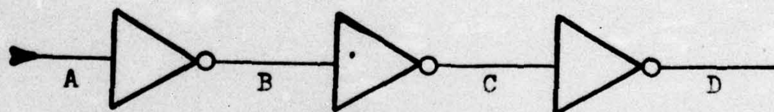
Solution by periods (no holds):

Q_1	Q_2	E	C_p	Period
×	×	×	1	p_i
×	×	1	0	p_{i-1}
×	×	×	1	p_{i-2}
×	×	1	0	p_{i-3}
×	×	×	1	p_{i-4}
0	0	1	0	p_{i-5}

(periods p_i through p_{i-5}: six period solution)

Finally, it is important to note that a solution to a problem usually defines a new set of problems.

Example:



Problem 1: $J : D = 1$

Solution : $C = 0$

Problem 2: $J : C = 0$

Solution : $B = 1$

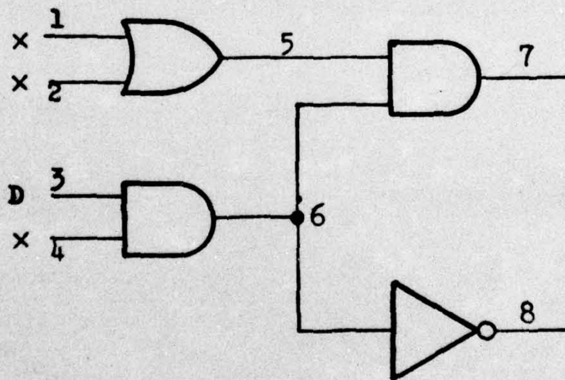
Problem 3: $B = 1$

Solution : $A = 0$ (solved since A is a primary output.

By transitivity, solving Problem 3 in turn solves Problem 2, which in turn solves Problem 1. (For simplicity we have ignored implication.)

Let P be a problem, e.g., a J or a D. A computation steps s consists of retrieving from the functional modeling library a prestored solution for this problem, and applying this solution to the circuit. This application either leads to an inconsistency, in which case, back-tracking occurs, or it is consistent. If consistent then some new J problems may be created; new D-drive frontiers exist; and some lines are set to 0, 1, u, D, $\bar{D}$ via implication. The total set of new line values so defined are all related to what is known as the <u>step</u> s computation. We number our computation steps s = 0,1,2,...

<u>Example</u>:



Step	Line Number 1	2	3	4	5	6	7	8	Comment
1	×	×	D	×	×	×	×	×	Initial state
2	×	×	D	1	×	D	×	D	D-Drive & implication
3	×	×	D	1	1	D	D	D	D-Drive
4	1	×	D	1	1	D	D	D	Justify
⋮									

Step 2: Problem solved $D:3 \rightarrow 6$

Step 3: Problem solved $D:3 \rightarrow 7$

Step 4: Problem solved $J:5 = 1$

3.3 Dynamic Assignment of Phases and Periods

When carrying out D-drive and/or forward implication, signals are propagated forward in time, hence, there is no problem in assigning problem solutions to phases, namely φ_2 follows φ_1 and φ_3 follows φ_2.

However, in line justification a sequence of line assignments for periods p_{i-1}, p_i may be mapped as follows:

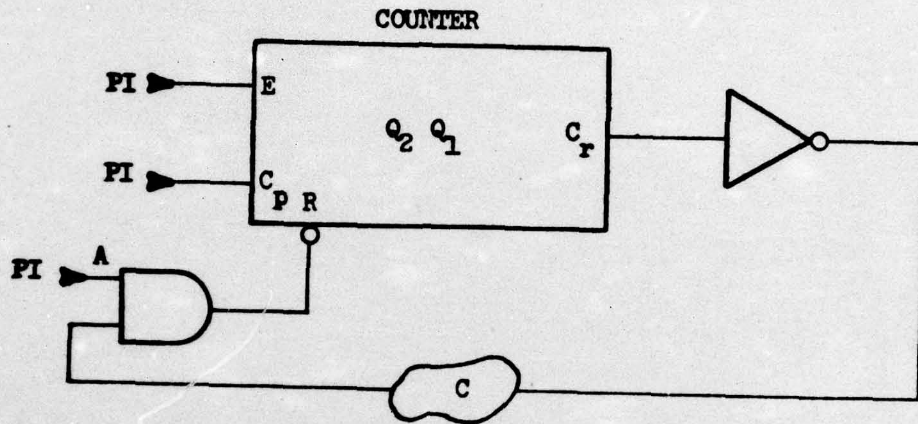
$$t_k \leftarrow \varphi_1 \leftarrow p_i$$
$$t_{k-1} \leftarrow \varphi_j \leftarrow p_{i-1} \quad (j=1,2,3)$$

or

$$t_k \leftarrow \varphi_j \leftarrow p_i \quad (j=2,3)$$
$$\nwarrow \varphi_{j-1} \leftarrow p_{i-1}$$

The test generation algorithm carries out this assignment process automatically. We illustrate the need for this type of operation in the next example.

Example: Consider the following circuit configuration.



Problem: $J : Q_2 = 0,\ Q_1 = 1$

Solution: $R\ ,\ E(1\times)C_p(\uparrow)$ (functional notation)

Period	E	C_p	R	Q_2	Q_1
P_i	×	1	1	0	1
P_{i-1}	1	0	1	0	0
P_{i-2}	×	×	0	0	0
				×	×

For this case we can use the following mapping:

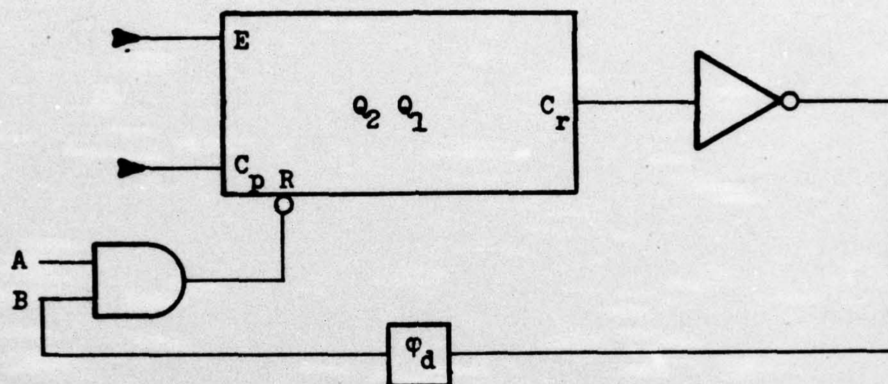
$$t_i \leftarrow \varphi_1 \leftarrow P_i$$

$$t_{j-1} \leftarrow \varphi_1 \leftarrow P_{i-1}$$

$$t_{j-2} \leftarrow \varphi_1 \leftarrow P_{i-2}$$

We solve $R = 0$ by setting $A = 0$.

Now we consider the following circuit configuration, and assume the initial state is $Q_2 Q_1 = 10$ and we desire to justify the state $Q_2 Q_1 = 01$.



The solution computation is shown in Table 3.1. Ignore the time and phase demarcations.

To justify $Q_2 Q_1 = 01$ we have the two period solution shown in lines 1 and 2, and the new problem $Q_2 Q_1 = 00$. To solve this problem we have the one period solution $R = 0$ (line 4). By implication, we get line 3 (forward in time) and line 5 (backward in time). The result from line 5 gives us the new problem $Q_2 Q_1 = 11$. We solve this on lines 6 and 7. We now scan this solution from line 7 up to line 1, assigning lines to phases and time frames. Whenever a primary input changes we get a new TF; whenever we go up a line due to implication (forward or

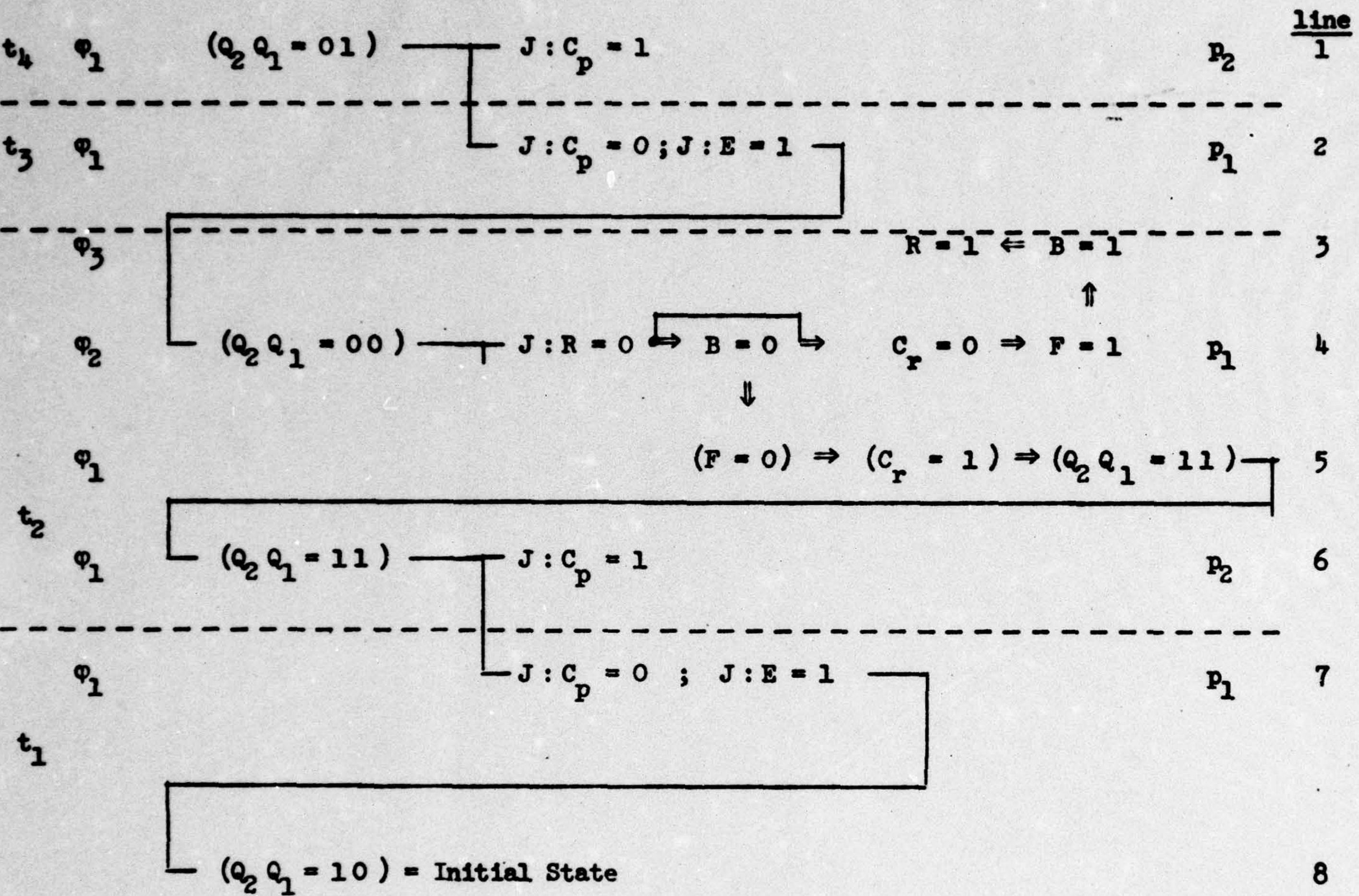


TABLE 3.1

reverse) we get a new phase. Note that the solution $R = 0$ (p_1) gets assigned to φ_2 of time frame t_2.

3.4 Data Structures

3.4.1 The Value Matrices V(L,S)

The value of each line in the circuit at each step in the computation is stored in a set of matrices $V(L,S)$, where $\ell = 1,2,\ldots,L$ is the index of a line and $s = 1,2,\ldots,S$ is a step number. L is the largest line number and S is the current step being processed. Hence, V is of fixed width (L) but of dynamic length (S). There is one V matrix for each unique phase. The V's are constructed dynamically on demand, i.e., whenever a new phase is created. Figure 3.5 indicates the general structure for locating the value $V(\ell,s) \in \{0, 1, D, \bar{D}, \times, u\}$ of line ℓ at step s for time frame t_j and phase φ_i.

3.4.2 The Problem/Solution Tree (P/S - T)

Every solution to a problem can be expressed as a sequence of line settings (J and D subproblems) specified to occur in a certain order, usually by phase and time frame.

The general format for a solution can be illustrated (approximately) by the data structure shown in Figure 3.6.

Initially the φ and t entries are null. They are assigned values dynamically by the test generation algorithm.

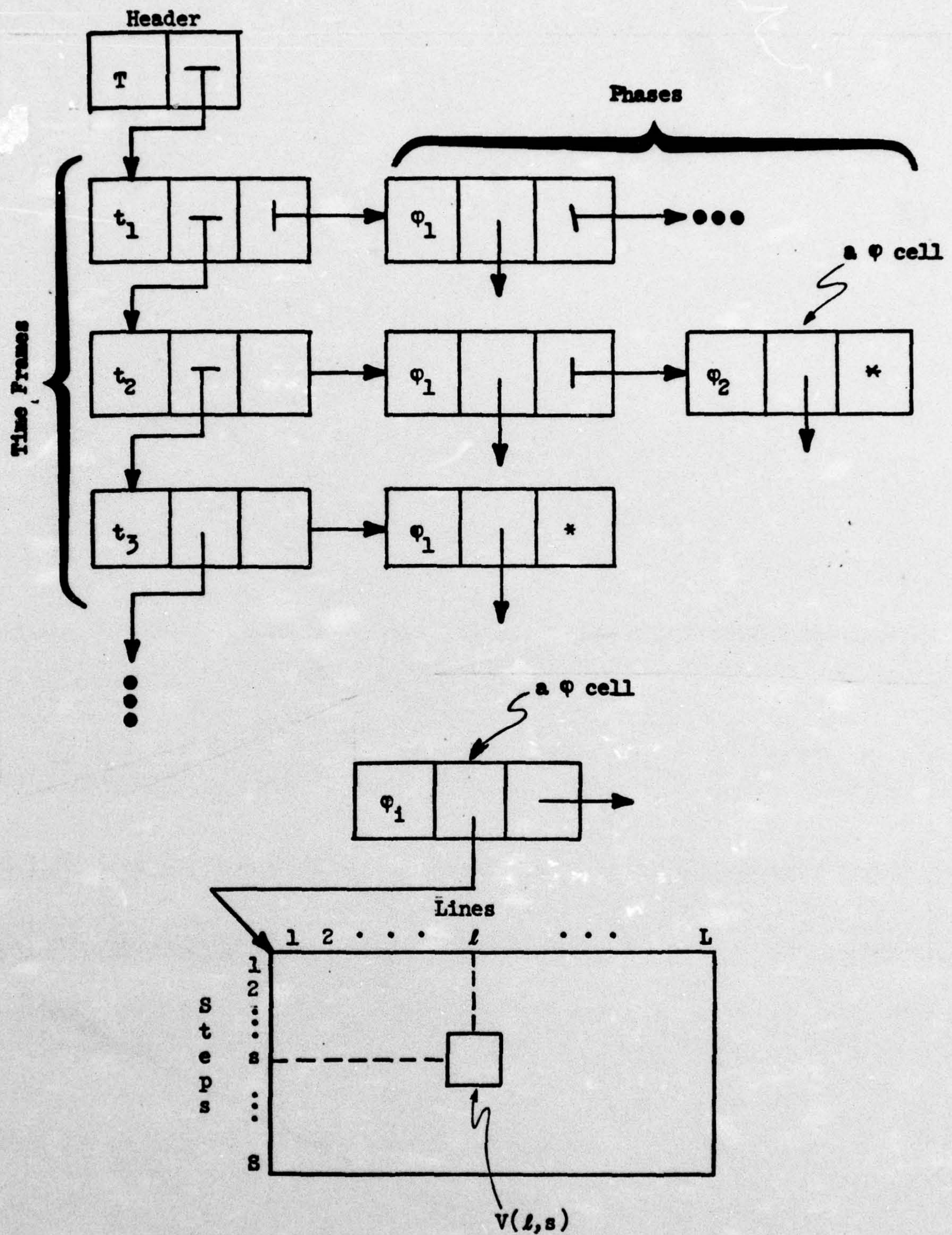


FIGURE 3.5: Value Matrices V(L,S).

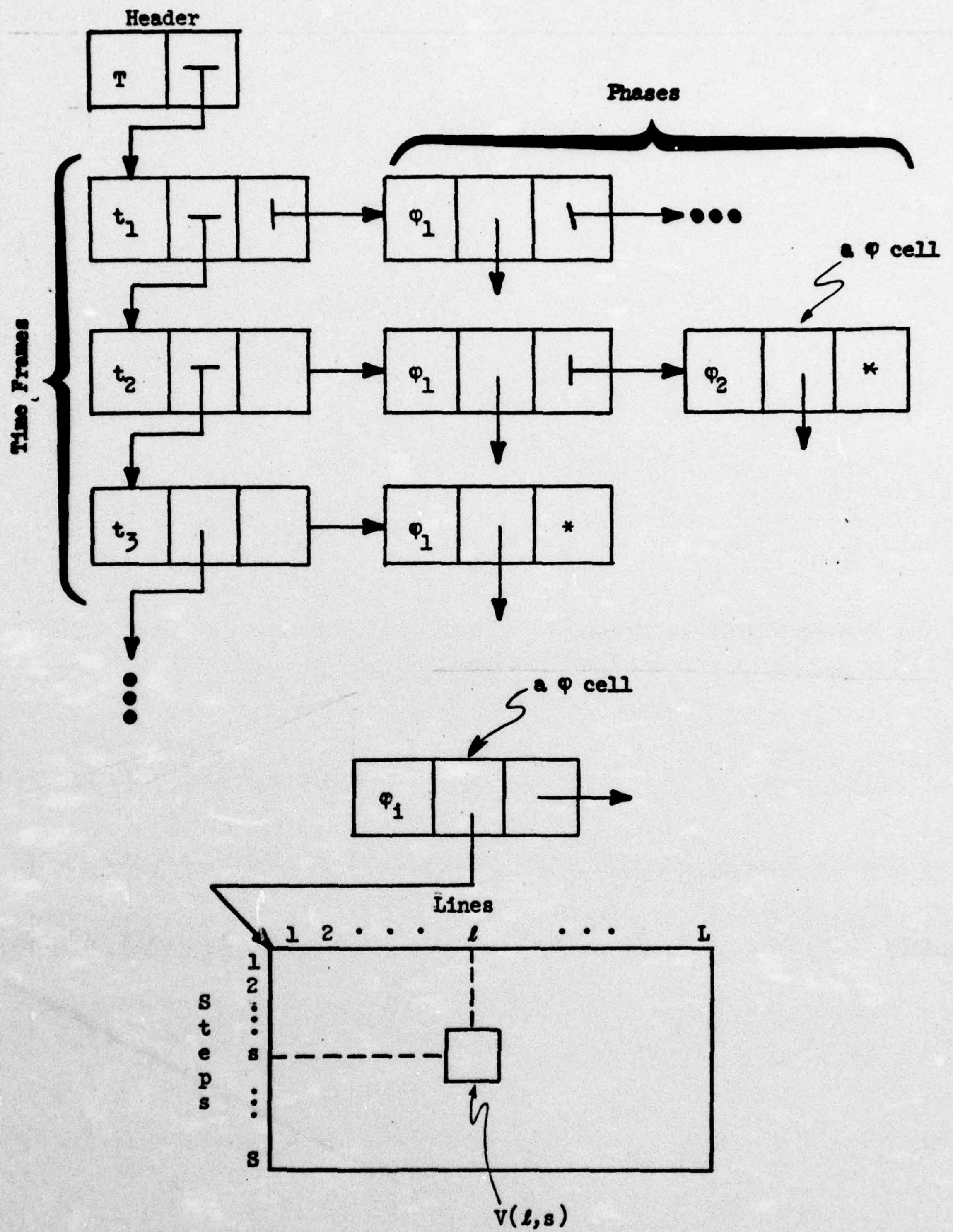


FIGURE 3.5: Value Matrices V(L,S).

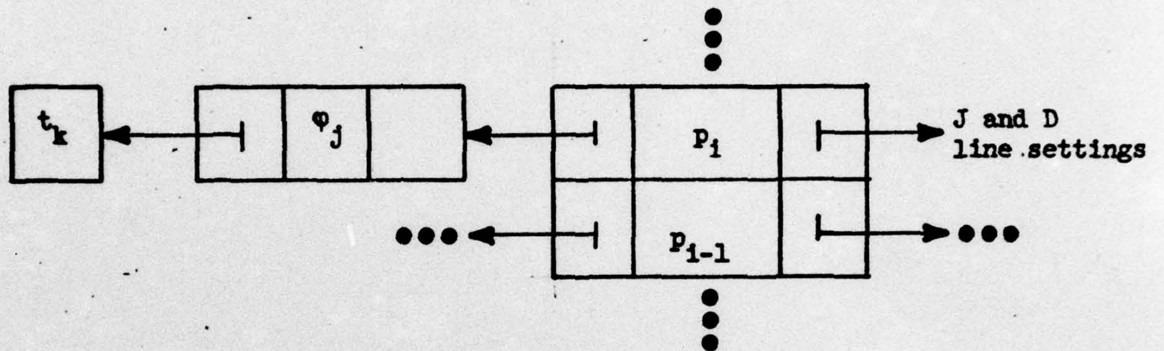


Figure 3.6

Example:

The solution shown below:

A	B	C_p	TF	Phase
0	0	1	t_1	φ_1
1	0	0	t_2	φ_1
1	0	1	t_3	φ_1
1	1	0	t_4	φ_2
0	1	x	t_4	φ_1

would be denoted by the structure shown in Figure 3.7.

Recall that a step consists of solving a D-drive line justification problem. The effect of the resulting assignments set some lines to 0 and 1 via implication. We add these line settings to the corresponding data structure as additional line settings at the appropriate period.

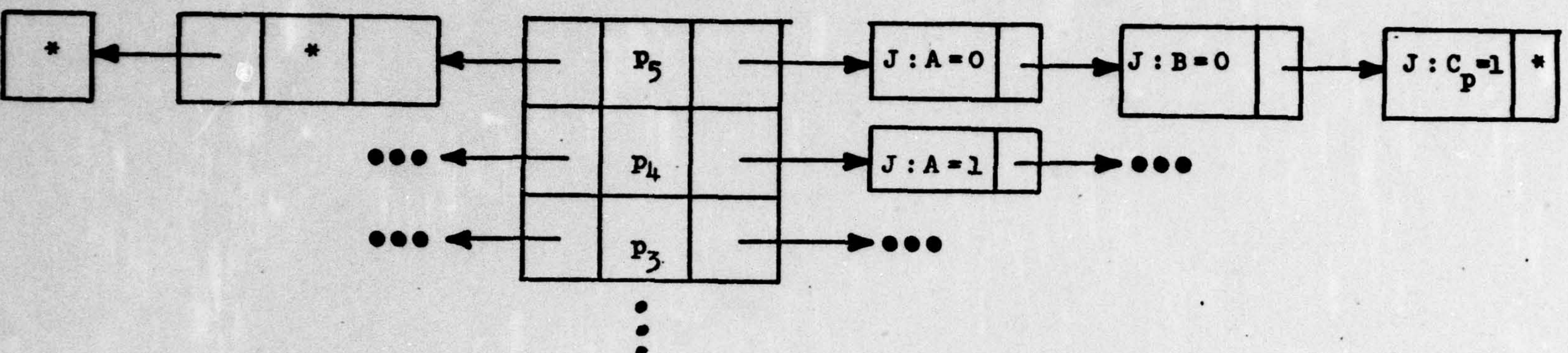
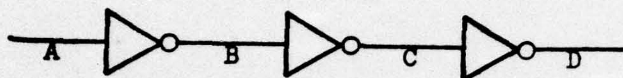


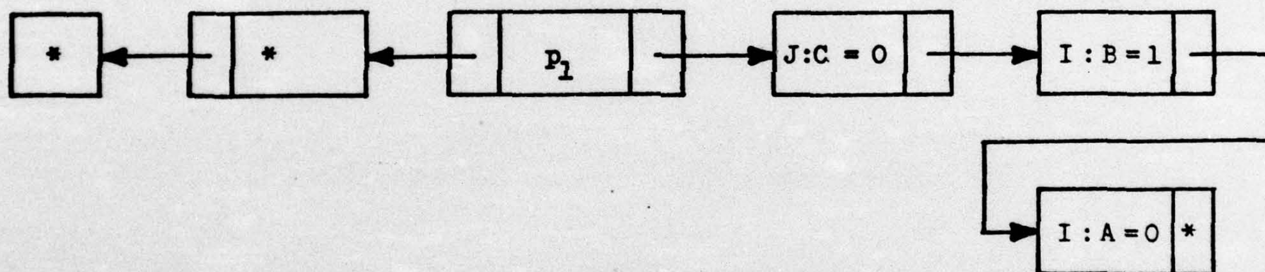
Figure 3.7

Example:



Problem : J : D = 1

Solution: C = 0



Note that the implications cannot be stored as part of the function problem solution; they are a function of the circuit and not the element being processed.

Given a problem, we can go to one of our functional algorithms to seek a solution subtree. This subtree is then appended to the problem tree at the node representing the problem. Each node in the appended subtree now becomes a new problem. We illustrate this in Figure 3.8. The header of each subtree (solution to a problem) is a step number.

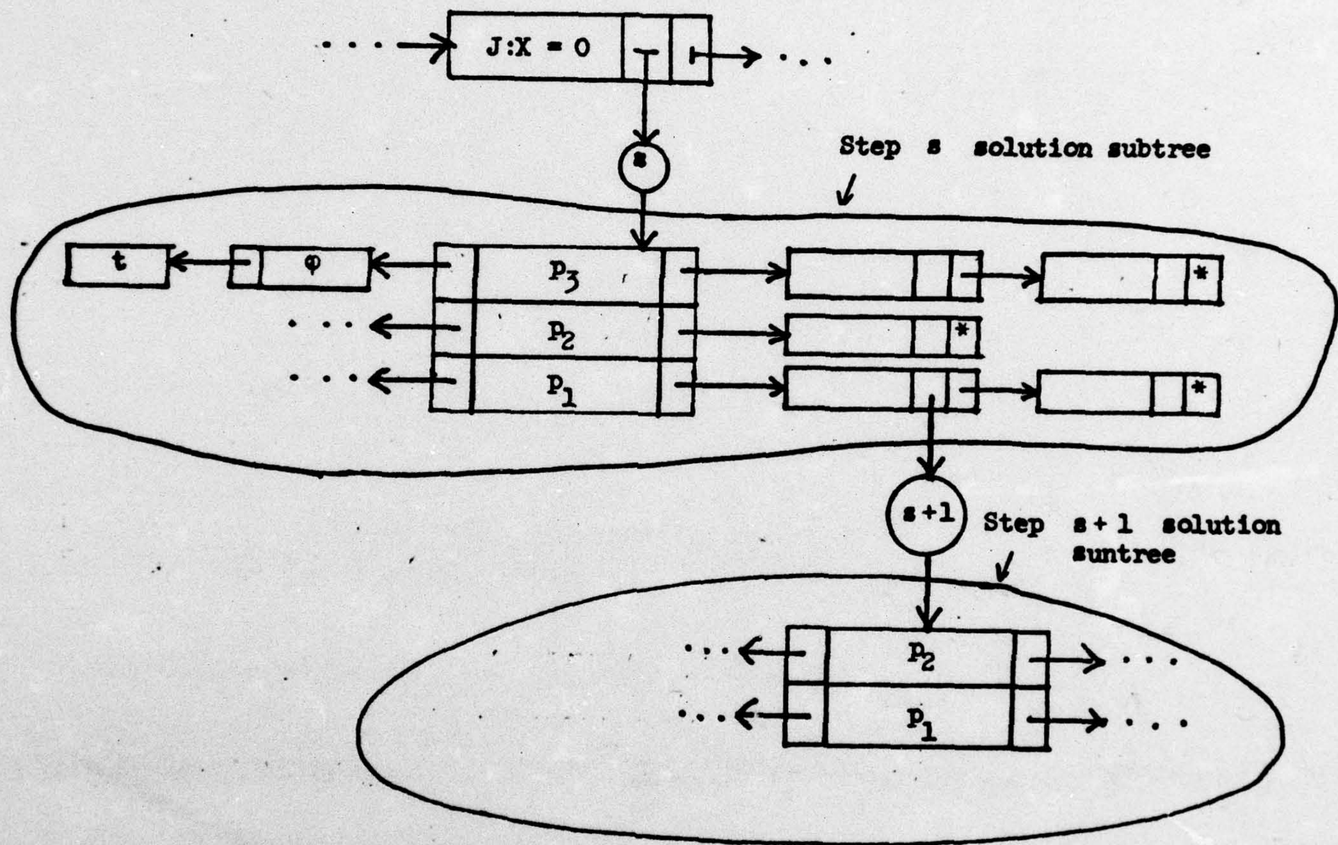


FIGURE 3.8

Note: If a solution subtree has a problem of the form $J : Y = v$, for time frame t_j and phase φ_i, but Y already has the value v for time frame t_j and phase φ_i, then this problem is deleted from the problem subtree.

The preceding description is only illustrative of the concept involved in storing problems/solutions. The actual data structure consists of basic records having the form shown in Figure 3.9.

The data structure for storing the implications (I) created by step i is similar to that used for J and D. However, since I does not lead to any problems, we do not require the fields, S, SP, T and NS.

Part of solution to problem A. This defines the problem B.										Data for solution C to problem B.			
F	R	N	PT	LN	V	P	TF	PR	PPT	S	SP	T	NS

where:

F - flag;

F = 1 - first cell in solution;

F = 0 - not first cell;

R - reverse pointer to son or father (son refers to another part of the same solution; father refers to the calling problem)

N - points to next son in current solution for a different line (Last entry points to first implication cell.)

PT - problem type: D for D-drive, J for line justification;

LN - line number (s) for D-drive or justify;

V - value (to justify);

P - phase (1, 2 or 3);

PR - period;

PPT - pointer to rest of solution associated with this line at previous phase (or time phase);

S - step number;

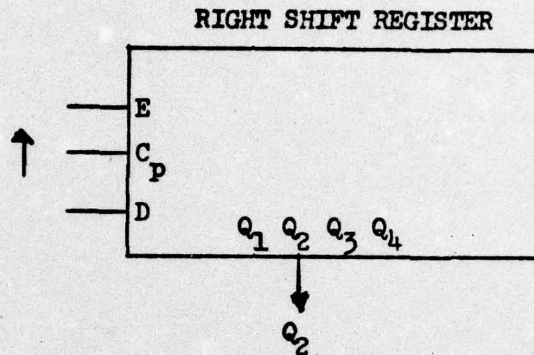
SP - pointer to solution to this problem;

T - total time required so far in solving this problem;

NS - number of this solution; if backtracking is required, this data used to get next solution.

FIGURE 3.9

Example: Consider the shift register shown below.



Problem: Justify $Q_2 = 1$ (t_n)

Solution No. 1 : Algorithm (functional)

$(E=1\times)(D=1\times, C_p(\uparrow); (C_p(\uparrow))$

ALGORITHM (Translated)

TF t_i*	E	D	C_p
i = n	×	×	1
= n - 1	1	1	0
= n - 2	×	×	1
= n - 3	1	×	0

In Figure 3.10 we illustrate the data structure for the solution to this problem.

Let J(s) be a list of all (unsolved) line justification problems which exist at step s. This list consists of pointers to problems in the

---

*We assume a synchronous circuit where each period is a unique time frame.

	F	R	N	PT	LN	V	P	TF	PR	PPT	S	SP	T	NS
Problem	0			J	Q_2	1	1	n + 1	*	*	24		-	1
Solution	1			J	C_p	1	1	n	1		—	*	0	—
	0		*	J	C_p	0	1	n-1	1		—	*	0	—
	0		*	J	C_p	1	1	n-2	1		—	*	0	—
	0		*	J	C_p	0	1	n-3	1	*	—	*	0	—
	0			J	D	1	1	n-1	1	*	—	*	0	—
	0		*	J	E	1	1	n-1	1	*	—	*	0	—
	0		*	J	E	1	1	n-3	1	*	—	*	0	—

FIGURE 3.10: Data Structure of Problem/Solution in P/S-Tree.

P/S-tree. The list is sorted in descending order of cost.

When going from $J(s)$ to $J(s + 1)$, new entries can occur due to D-drive or carrying out line justification; entries disappear due to line justification and implication. If S is the final step, then $J(S)$ is empty.

$D(s)$ is a list of D-drive problems existing at step s. This list consists of pointers to D-drive problems in the P/S-tree. The list is sorted in ascending order of cost.

Consider a problem created at step j. Assume, during step S $(S > j)$ we attempt to solve this problem. If all solutions lead to an inconsistency, then problem j has no solution and all entries in the P/S-tree and V matrices corresponding to steps $j, j+1, \ldots, S$ must be erased.

4. TEST GENERATION ALGORITHM

In this section we will describe the basic structure of the test generation algorithm used in TEST/80. In essence, it is an extension of the D-algorithm and employs path sensitization. Its unique features are:

1. the use of high level functional primitives;
2. the use of preprocessing, namely, rate and cost analysis;
3. the use of a powerful initialization routine;
4. the handling of asynchronous circuits via phases and time frames; and
5. various miscellaneous techniques used to reduce CPU time.

4.1 Data Structure

The data structure shown in Figure 4.1 is symbolic, i.e., it shows the data representation of the circuit to be tested. The structure is based upon the concept of a descriptor. There is one descriptor for every primitive element in the circuit.

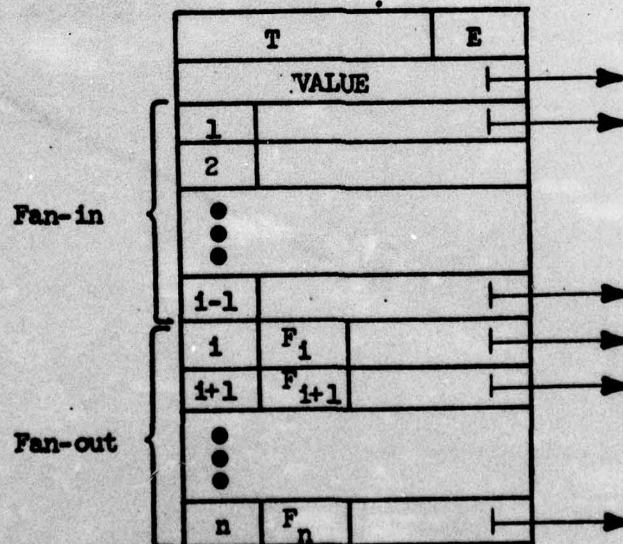


FIGURE 4.1: Data Descriptor for a Primitive Element

The fields in this descriptor are described next.

T - type of element; T is an integer and is used to identify the functional subroutine associated with the element type.

E - unique integer name for element being defined.

VALUE - a vector having n components representing the I/O lines and state of the device; each line can take on the value 0, 1, x, D, $\bar{D}$ or u, hence, there are 3 bits per element.

The next i - 1 fields have the format (j, address) where the address points to the j-th fan-in of this device. We assume i - 1 inputs. The fields i,...,n are associated with the outputs of the device. The format for these fields is $(k, F_k, \text{address})$. F_k is the degree of fan-out of the k-th signal. If $F_k = 1$ the address points to the element where k fans-out; if $F_k > 1$ the address points to the fan-out list.

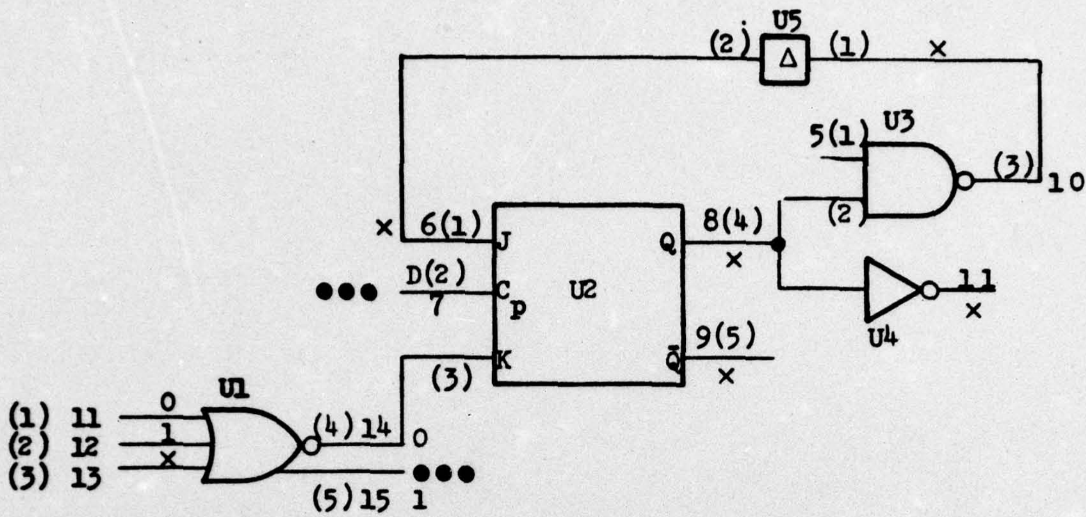
Figure 4.2(a) indicates a portion of a simple circuit and Figure 4.2(b) indicates the corresponding data structure.

The fields in the vector VALUE are ordered as follows:

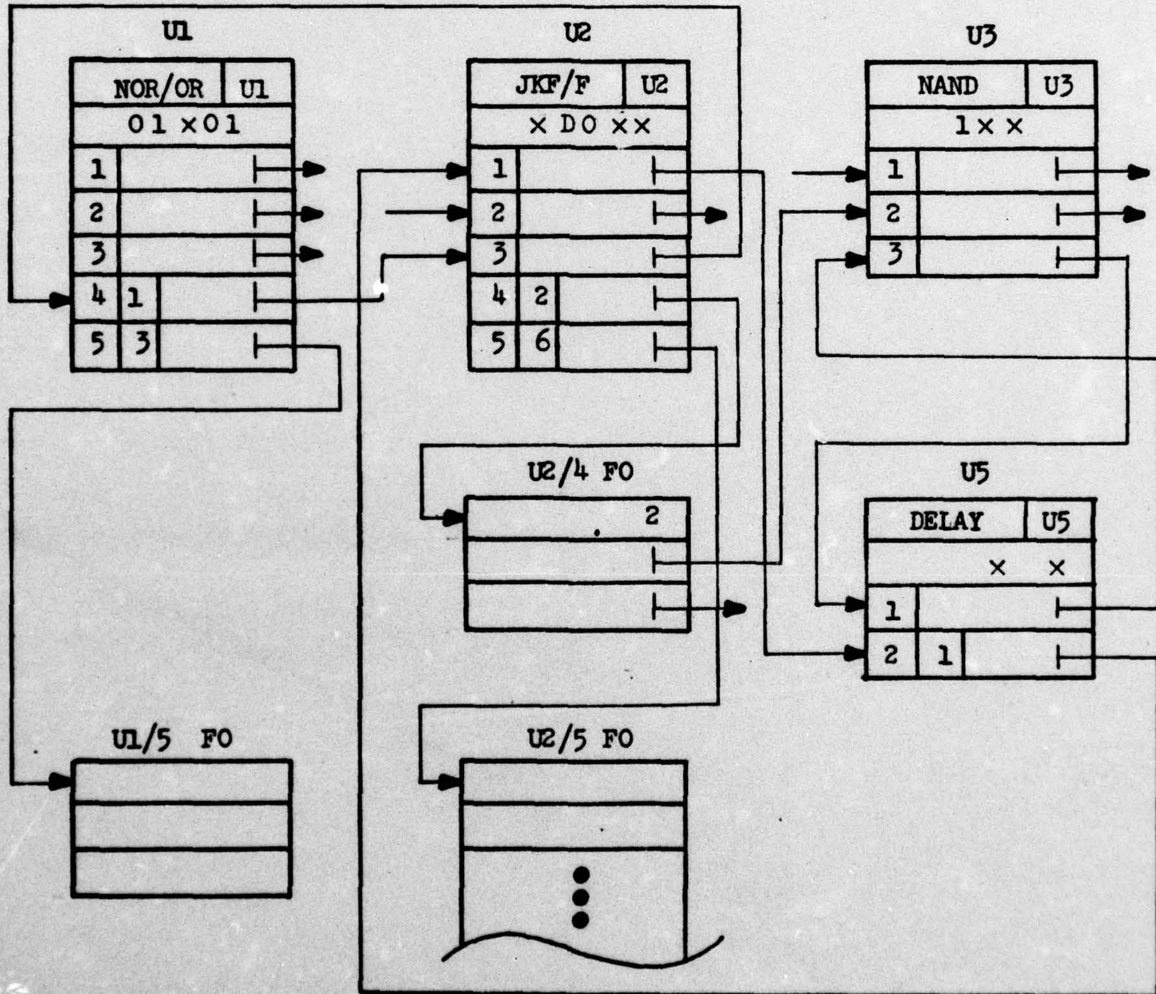
The j-th field, $1 \leq j \leq i - 1$, corresponds to the j-th input; the next fields (ordered) correspond to the outputs; and the final fields correspond to the state variables (if any exist).

4.2 Functional Element Routines

There are five functional level routines related to every primitive element type. These routines carry out the process of initial error generation, implication, D-drive, line justification and initialization. The



(a)



(b)

FIGURE 4.2: (a) Simple Circuit; (b) Data Representation.

concept of implication, D-drive, and line justification are discussed in our Report No. 1-76, "Functional Level Modeling of Complex Elements in TEST/80," and initialization is discussed in our Report No. 2-76, "Initialization of Digital Logic Circuits."

The routines are usually called via a generic global operation, e.g., D-drive from J input of F/F to Q output. The tables respond with generic sequences for carrying out the desired operation. The translator maps these generic solutions into binary strings.

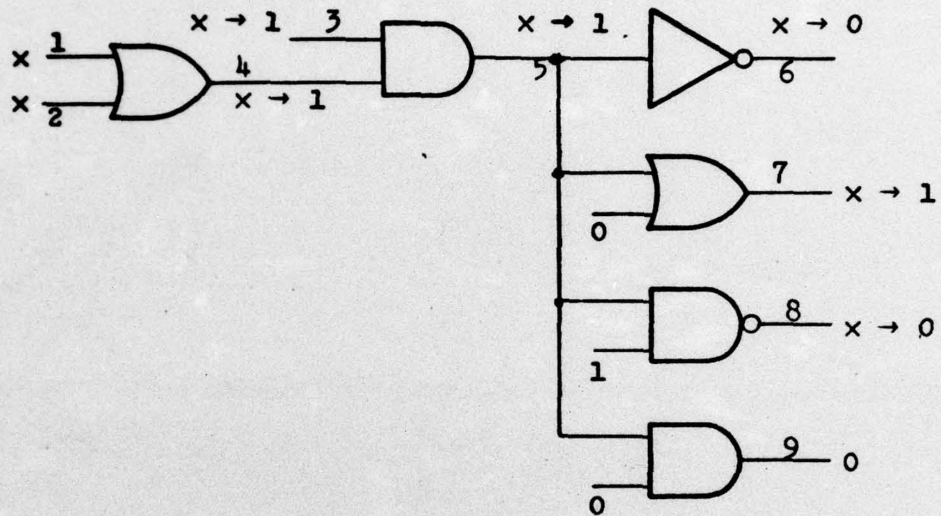
4.2.1 Initial Error Generation

To generate a test for line i s-a-1(0), we place a $D(\bar{D})$ into the line value table for time t_n phase φ_1 and store the problem, "Justify line i at the value α" in the P/S-tree, where $\alpha = 0(1)$ for the fault s-a-1(0).

4.2.2 Implication

Recall that every element type is associated with an implication table or routine. Whenever a line (input or output) of an element is changed from $\times$ to $v \in \{0, 1, u, D, \bar{D}\}$ we check to see if any other lines, currently at $\times$, are forced to a value due to this change.

Example:



When line 6 changes from x to 0, denoted by $6 : x \rightarrow 0$, we have, by implication (forward $\overrightarrow{I}$, reverse $\overleftarrow{I}$)

a) $\overleftarrow{I} : 5 : x \rightarrow 1$

b) $\overrightarrow{I} : 7 : x \rightarrow 1$

c) $\overrightarrow{I} : 8 : x \rightarrow 0$

d) $\overleftarrow{I} : 3 : x \rightarrow 1$

e) $\overleftarrow{I} : 4 : x \rightarrow 1$

f) $\overleftarrow{I} : 2 : x \rightarrow 1$

g) $\overleftarrow{I} : 1 : x \rightarrow 1$

The process of making all line assignments due to implication is carried out by the implication routine in conjunction with the implication tables for each element type.

This routine is recursive. It is entered with an implication event which consists of

a) a line value $v \in \{0,1,\bar{D},D,u\}$ which was formally an $\times$;

b) the name of the line;

c) an element E; and

d) the period p.

Implication then hands this data over to the implication tables (forward or backward) associated with element E. These tables then return with new implications if they exist.

The implication events to be processed are stored on a stack. They can be processed in any order. New implications are placed on the top of the stack. They are removed one at a time and processed. Implication terminates when the stack is empty.

4.3 Implication Routine

1. Pop implication event from implication stack. If empty exit, else continue.
2. Send event to appropriate implication functional routine.
3. Process response. Identify all lines changed from $\times$ due to implication event.
4. For all new line settings create appropriate implication events. If event created by forward implication on line i,

then create a new implication event for all fan-out elements of i. If event created by backward implication on line i through element E, create new implication events on the signal source for i and all fan-outs of i except for E. Put all these new line events onto the implication stack. Go to 1.

4.4 D-Drive Routine

D-drive is a simple routine which selects a D-drive event and passes the event to an element D-drive functional routine. The function returns with the appropriate line settings required to solve the D-drive event problem. The D-drive event consists of

a) the element being processed;

b) the source of the current D;

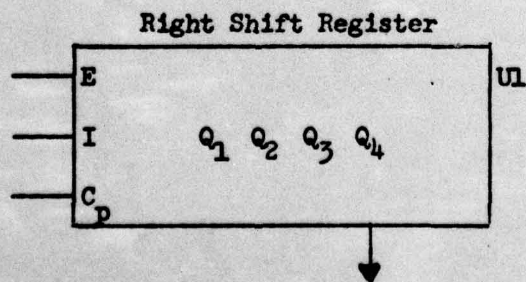
c) the target for the D; and

d) the current period and step number.

The source can be either an input to E or one of its internal state variables. The target can be either a state variable or an output line.

Example

Right Shift Register



D-drive event: a) U1

b) I (input)

c) Q_4 (output)

d) P_4

Solution number i: $(E = (1\times)\ C_p(\uparrow))^4$

D-drive event: a) U1

b) Q_2

c) Q_3

d) P_4

Solution number j: $(E = 1\times)\ C_p(\uparrow)$

4.5 Justification Routine

Justification is a simple routine which selects a line justification event and passes the event to an element line justification function. The function returns with the appropriate line settings required to solve the justification event problem. A line justification event consists of

a) the element being processed;

b) the line or state variable to be justified;

c) the value to be justified; and

d) the current period and step number.

As illustrated previously in the section on timing, for asynchronous feedback loops, line justification requires dynamic assignment of line settings to phases.

4.6 Initialization Routine

This routine is similar to D-drive and line justification. Its function is to obtain from the functional element routines an input sequence for initializing an element E, i.e., to take the element out of the u state.

4.7 Consistency Routine

This is a simple routine which, when given a new line value $v(i) \in \{0, 1, u, \bar{D}, D\}$ for line i and its associated time frame and φ, checks with the $V(L,S)$ matrices to see that the line does not already have a value $w(i) = \{0, 1, u, \bar{D}, D\}$, where $v(i) \neq w(i)$. If this is the case, the new line setting is said to be inconsistent with previous line settings and the algorithm backtracks.

4.8 Definition of D-Frontier

The D-frontier is the set of all elements having a D at a source and an $\times$ at a target. The source is either an input line or state variable; the target is either a state variable or an output line.

4.9 Initialization Module

Most test generation algorithms generate a self-initializing test for a fault f, i.e., a test which works no matter what the initial state of the circuit. In our system, we use an initialization module which derives a sequence $\underline{R}$ for driving the fault-free circuit to a known state S. We assume $\underline{R}$ drives C_f to the state S_f. We then derive tests for faults

with respect to the resulting composite state S/S_f. S_f is determined via simulation.

Case 1: $S = S_f$. For this case S/S is our initial state, or goal state.

Case 2: $S \neq S_f$ but S_f is completely specified, i.e., has no u's.

For this case S/S_f has D entries and test generation is relatively simple.

Case 3: $S \neq S_f$ and S_f is incompletely specified, i.e., has u entries.

This situation occurs when the fault f inhibits the initialization of some state variables. We detect this type of fault by the following strategy. Let $Y = (y_1, y_2, \ldots, y_n)$ be the state variables in the circuit, and $Y_1 = (a_1, a_2, \ldots, a_n)$ be a state, i.e. $a_j \in \{0,1\}$ for all j. Let R be an initialization sequence that drives the fault-free circuit C_0 to the state $Y_1 = (a_1, a_2, \ldots, a_n)$. Assume, due to a "reset" fault f, that the circuit C_f is driven to the state $Y_2 = (u, a_2, \ldots, a_n)$ by R, i.e., y_1 fails to initialize.

Case 3a: Assume the actual circuit C_f initializes to the state $y_1 = \bar{a}_1$.

We can detect this situation by forming the composite initial state $S/S_f = (D, a_2, a_3, \ldots, a_n)$. Using this as the initial state, we can construct an input sequence X for driving a D to a po .

Case 3b: Assume the actual circuit C_f initializes to the state Y_1.

In this case $\underline{X}$ may not detect this "reset" fault. After $\underline{R}\,\underline{X}$ has been applied, assume the composite circuit C_0/C_f is in the state Y_3/Y_3. Let $\underline{T}$ be an input sequence which drives C_0/C_f from Y_3/Y_3 to a state Y_4/Y_4, where $Y_4 = (\bar{a}_1, b_2, b_3, \ldots, b_n)$, $b_j \in \{0,1\}$. Now, if we apply $\underline{R}$ the resulting state of C_f will be $Y_1 = (\bar{a}_1, a_2, \ldots, a_n)$, and $\underline{X}$ will detect the fault (Case 3a). Hence, a test for a reset fault f is

$$\underline{R}\,\underline{X}\,\underline{T}\,\underline{R}\,\underline{X} \quad .$$

If in trying to get to Y_3/Y_3 or Y_4/Y_4 we get to a state Y_i/Y_j, $Y_i \neq Y_j$, we again have a D and thus propagate it to a po.

4.10 Preprocessing

There are three major preprocessing routines employed by TEST/80, namely,

1. Rate analysis;
2. Cost analysis; and
3. Fault collapsing.

Rate and cost analysis have been discussed previously. The results are stored in appropriate matrices for access by the stimulus generation modules.

As the third phase of the preprocessing of a circuit, faults are collapsed based upon the classical concepts of fault equivalence and fault dominance.

4.11 Unsolvable States

It is often the case that certain states in a circuit cannot be reached. For example, a 4-bit counter may be used in a mod 10 counter, in which case the states

```
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

cannot be realized. Often, however, the test generation algorithm may try to justify one of these states. Such a problem is unsolvable, but may require a great deal of computation before the algorithm exhausts all possibilities and comes up with this correct conclusion. To save CPU time we create a table called UNSOLVABLE which stores all such problems. The general form for an entry in UNSOLVABLE is (S_1, S_2) where S_1 and S_2 are partial states (some $\times$ entries) of C_0, and for which there exists no sequence for driving C_0 from S_1 to S_2. For our example, we would write

$$(\times\times\times\times, 101\times)$$

and

$$(\times\times\times\times, 11\times\times) .$$

Entries are placed into UNSOLVABLE in two ways. Initially, the test engineer can enter states which are unsolvable. After that, whenever TEST/80 fails to justify a state configuration, it can enter this configuration into the table.

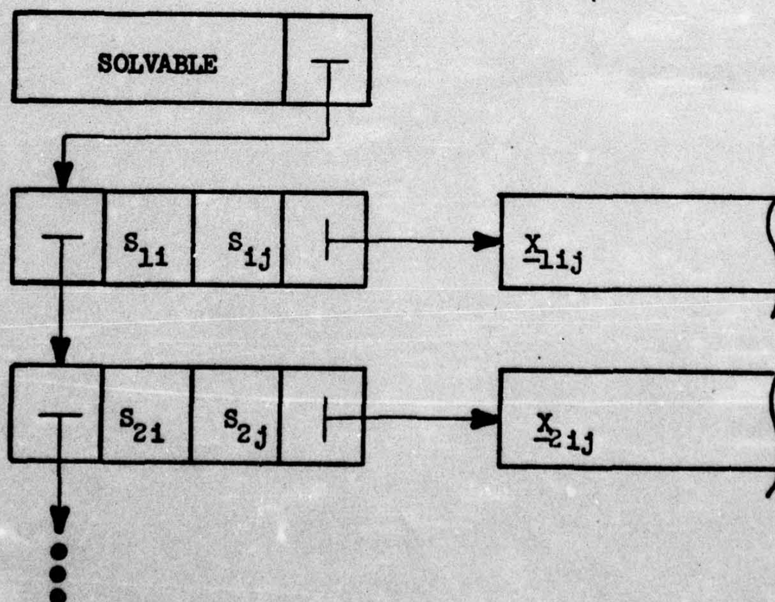
The table is used as follows. When a new time frame is being processed, the current state S of the system is computed, and compared with entries in UNSOLVABLE. If S is "contained" in an entry in this table, then the state is unsolvable. (We say 1 1 0 × is contained in 1 1 × × since 1 1 × × covers 1 1 0 ×.)

4.12 Solvable States

Once the TEST/80 algorithm has computed a sequence $\underline{X}_{ij}$ for driving C_0 from S_i to S_j, it stores this sequence in a table SOLVE for possible future use. If this problem is ever encountered again, the sequence $\underline{X}_{ij}$ can be retrieved and used.

Entries

Entries are made into this table automatically by the program. In an interactive mode, the test engineer can also make entries to this table. The format for this table is shown next.

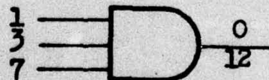


4.13 Backtrack State Event

There are usually numerous ways (solutions) to justify a line i at a value $v \in \{0,1\}$, or to D-drive through an element. If a selected solution leads to an inconsistency, one must backtrack to try another alternative. The mechanism for backtracking and selecting alternatives is called a backtract state event. There is one such program module for each element type. These modules are part of the functional element type routines and when given information on the last solution, produce the next solution.

There are numerous ways for carrying out this process, and different element types may employ different techniques. We will illustrate next a few examples of how backtrack state events can be handled.

Example:



Justify : 12 = 0.

Solutions:

1	3	7
0	×	×
×	0	×
×	×	0

1. Stack backtract statement:

   Place the three solutions

   0 × ×
   × 0 ×
   × × 0

   on a stack when the

problem first occurs. Whenever we need a new solution we pop the stack. When the stack is empty we have exhausted all solutions.

2. Counting backtrack state event:
   Solution i consists of placing a 0 on the i-th input line to the gate. We enter the routine originally with $i = 1$. Each time we backtrack we increment i. When $i = 4$, we have exhausted all solutions.

In summary, we associated with each element type a backtrack state event routine such that given the last solution, the routine allows the functional element routine to produce the next solution, if one exists.

4.14 Flow Charts for STIM GEN

In the flow charts presented, R stands for return, S for success and F for failure. When we call a routine with an argument R, the routine, upon finishing its execution, sets R to F or S depending on the result obtained.

Figure 4.3 shows the general structure of TEST/80.

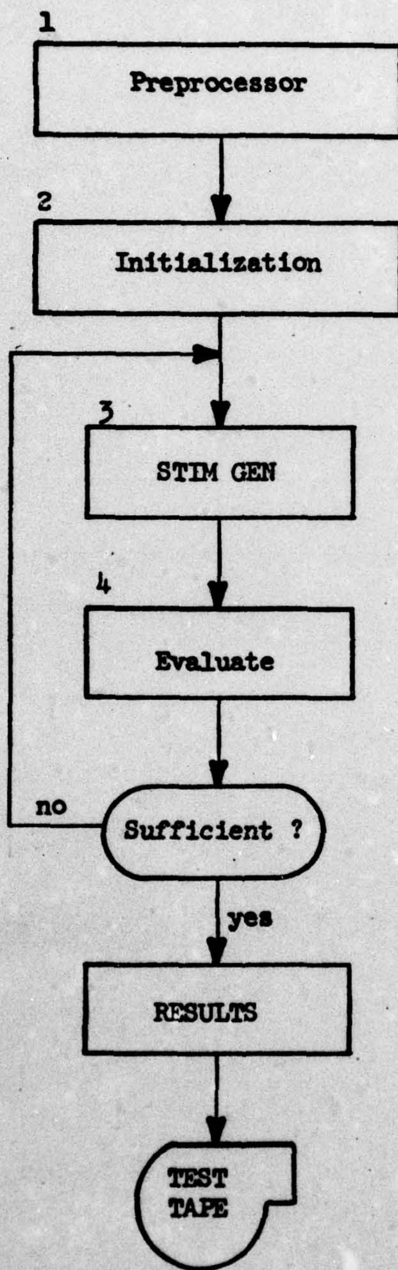
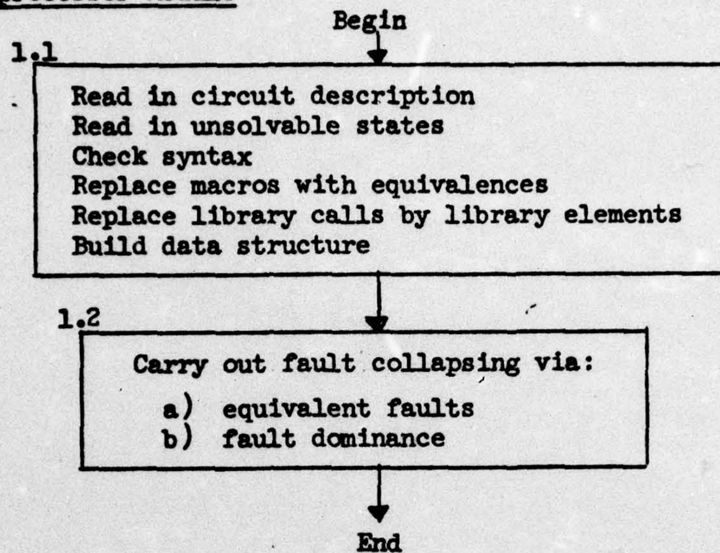


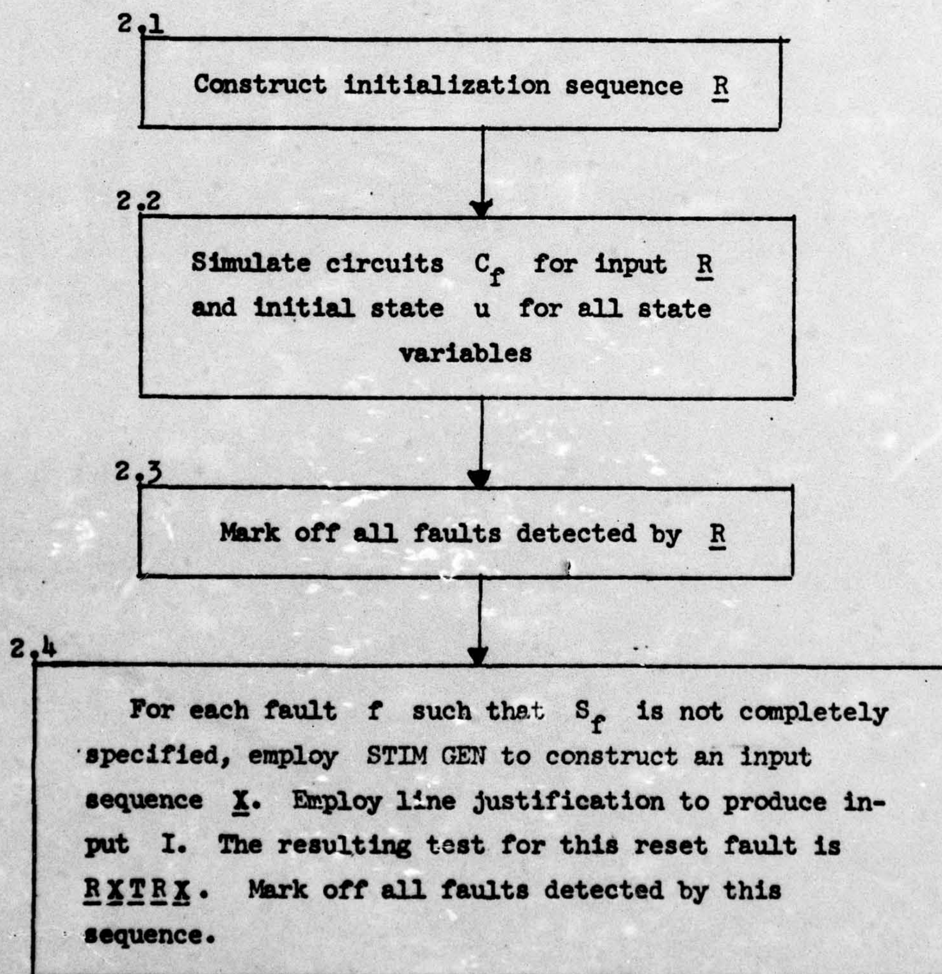
FIGURE 8.3: TEST/80 Flow Chart.

No. 1: Preprocessor Module

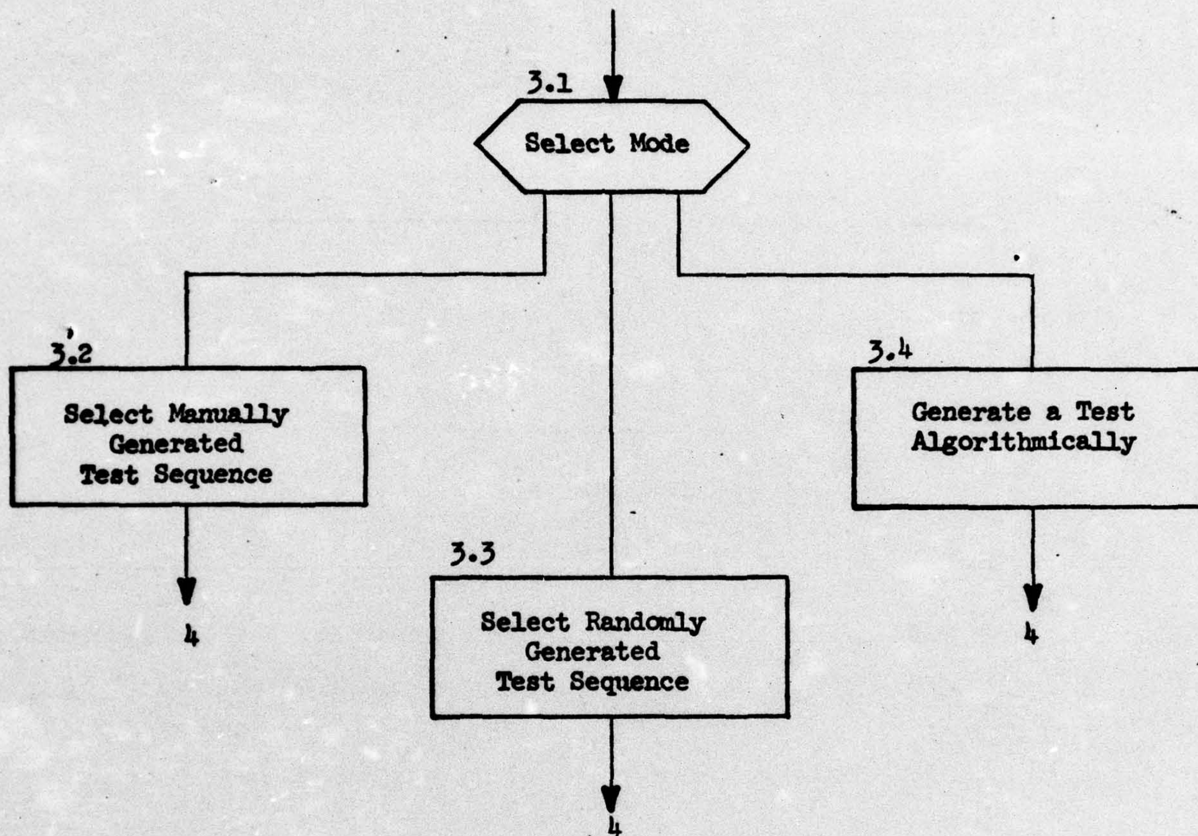


No. 2: Initialization Module

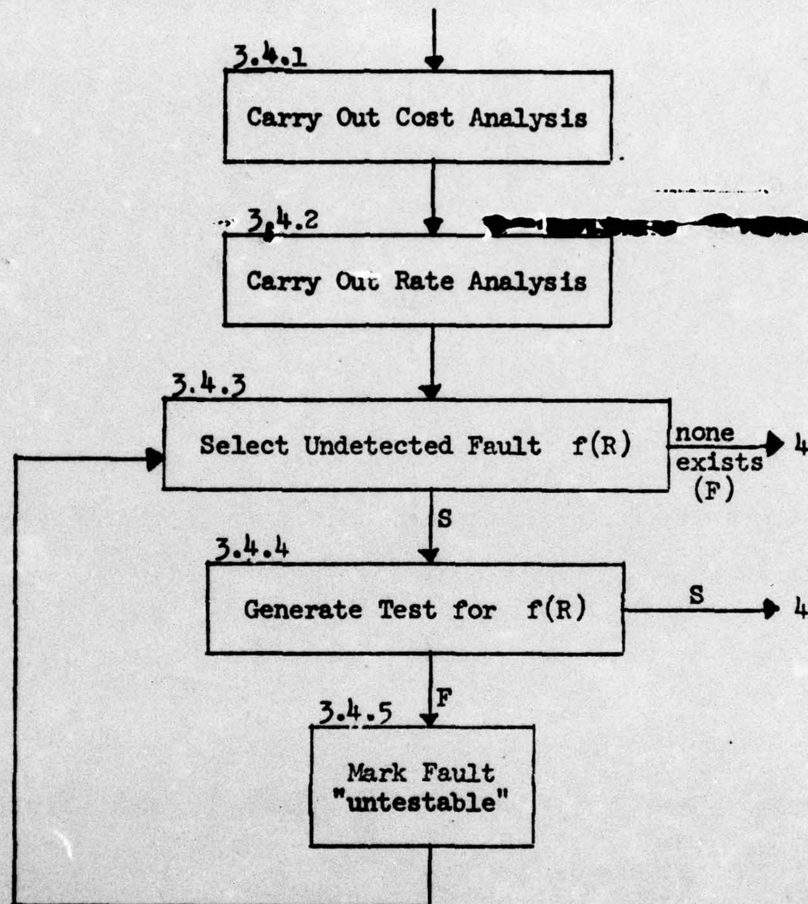
The algorithms for this module are discussed in our Report No. 2-76. They employ the line justification and functional initialization routines associated with each element type. At the conclusion of this routine all initializable circuits will be in a given state. Faults which inhibit initialization are called reset faults and are processed as discussed previously.



No. 3: STIM GEN



No. 3.4: Algorithmic Test Generation



No. 3.4.4: **Generate Test for f(R)**

1. Clear P/S-tree; clear V(L,S) tables; S ← 0; set initial state to IS (result of initialization module); TF ← 1; φ ← 1.
2. Set up initial D on line with fault; create D-frontier; create associated line justification problem .

3. D on ps ?

Yes: Call justification routine (s, R_1)

R_1 = S — R = S and return with test

R_1 = F — go to 8

No : Continue

4. S ← S+1

5. Construct D-frontier A(s) for this step.

6. Select cheapest D-drive problem in A(s) and delete from A(S)·(R)

R = S — continue

R = F — go to 14

7. Set up backtrack state event for D-drive.

8. D-drive through selected element (R).

R = F — go to 6

R = S — continue

9. Carry out implication due to D-drive assignment.

10. Set up backtrack state event for timing assignment.

11. Assign periods to phases, and phases to time frames wherever necessary (R).

R = F — (no more assignments exist) backtract to 7

R = S — continue

12. Check for line consistency due to D-drive and implication (R).

R = F — (inconsistent) backtrack to 10

R = S — continue

13. Carry out a check on rate analysis (R).

R = F — (inconsistent) backtrack to 10

R = S — continue

14. S ← S - 1

15. S = 0 ?

Yes: R = F and return

No : backtrack to 7

Justification Routine (S^*,R)

"S^* is current step number"

1. Set S ← S^*

2. Select most costly line not yet justified at step S.

   None exists: R ← S and return

   Exist: Continue

3. Create line justification event J(S) = (E, i, v, p),

4. Set up backtrack state event B for J(S).

5. Pass J,B to functional element justification routine (R).

   R = F — go to 12

   R = S — continue

6. Carry out all implications due to line justification event,

7. Set up backtrack state event for timing assignment .

8. Assign periods to phases, and phases to time frames wherever necessary (R).

   R = F — (no more assignments exist) backtrack to 4

   R = S — continue

9. Check for line consistency due to justification and implication (R),

   R = F — (inconsistent) backtrack to 7

   R = S — continue

AD-A040 271 BREUER AND ASSOCIATES ENCINO CALIF F/G 9/5
AN ADVANCED AUTOMATIC TEST GENERATION SYSTEM FOR DIGITAL CIRCUI--ETC(U)
MAR 77 M A BREUER N00014-75-C-1053
UNCLASSIFIED TEST/80-1-77 NL

2 OF 2
AD
A040271

END
DATE
FILMED
6-77

10. Carry out a check on rate analysis (R)

$R = F$ — (inconsistent) backtrack to 7

$R = S$ — continue

11. Set $S \leftarrow S+1$ and go to 2

12. Is $S = S^*$?

No : $S \leftarrow S-1$ and go to 4

Yes: $R \leftarrow F$ and return.

5. FAULT SIMULATION

5.1 Introduction

The function of the fault simulator is to determine the set of faults detected by a given test sequence. The attributes of the simulator for TEST/80 are listed below:

a) Concurrent fault simulation;

b) Functional modeling;

c) Three-valued simulation;

d) Arbitrary propagational delay;

3) Input skew buffering (optional);

f) Event-directed, table-driven; and

g) Dynamic (real-time) inputs.

The rationale for using functional level modeling is the same as for its use in the stimulus generation module, namely, efficiency (reduction in CPU time) and the fact that for some element types, such as LSI's, logic level descriptions are not available. The rationale for using concurrent fault simulation is that it appears to be the most efficient fault simulation technique compatable with functional level modeling. This occurs because the same routine used to simulate an element in the good circuit is used to simulate the same element in the faulty circuit.

To be compatable with our test generation algorithm, we can employ functional level models in our fault simulator. Hence, even though standard gate types and flip-flops are primitives, so also are most MSI's. Finally, some LSI's are primitives, such as RAM's and ROM's. We refer to

a primitive as an element type, e.g., a 3-input AND gate, and a specific instance of an element type in a circuit is called an element.

As in the test generation module, MSI's and flip-flops can be modeled either functionally or at the gate level. For example, a counter can be modeled via gates and flip-flops, and the flip-flops can be modeled, if desired, by gates. Hence, if an internal fault in a specific IC is of interest, this IC can be modeled at a level so that the fault appears external to a primitive. Hence, this fault can now be dealt with explicitly. Note that all other IC's can still be modeled at their functional level.

We assume each element type is associated with an element simulation module such that given its current internal state, input and output state (either an input or internal state event), its next internal and output state can be determined. We denote the element simulation module for element type t_i by $S(t_i)$.

State events occur when an element must be processed even though no input event has occurred. As an example, consider a 4-bit ripple counter. Assume its initial state is 0111 and an input event occurs at time t_0 which causes the counter to increment. Assume the desired internal state sequence is shown below. Here, we assume that each bit of the register has Δ units of delay. The state at t_1 occurs because of the input event at t_0. At time t_1 we compute the new state which is to occur at time t_2 and enter this event into the schedulor. This is called a state event. This process continues until the device stabilizes.

lsb → Q_1

Q_4	Q_3	Q_2	Q_1	t
0	1	1	1	$t' \leq t_0$
0	1	1	0	$t_0 + \Delta = t_1$
0	1	0	0	$t_1 + \Delta = t_2$
0	0	0	0	$t_2 + \Delta = t_3$
1	0	0	0	$t_3 + \Delta = t_4$

The simulator will employ three-valued logic, namely 0, 1 and u. The u has the following meaning:

a) steady state value - unknown;

b) input transition - unknown ; and

c) transient - race or hazard.

All $S(t_i)$ must be capable of handling this logic. For simple elements, such as gates and flip-flops, $S(t_i)$ can be implemented via zoom tables. For more complex elements the techniques discussed in our TEST/80 report on functional level modeling can be used. (See also, section on implication.)

5.2 Simulation System Structure

The simulation system structure is shown in Figure 5.1.

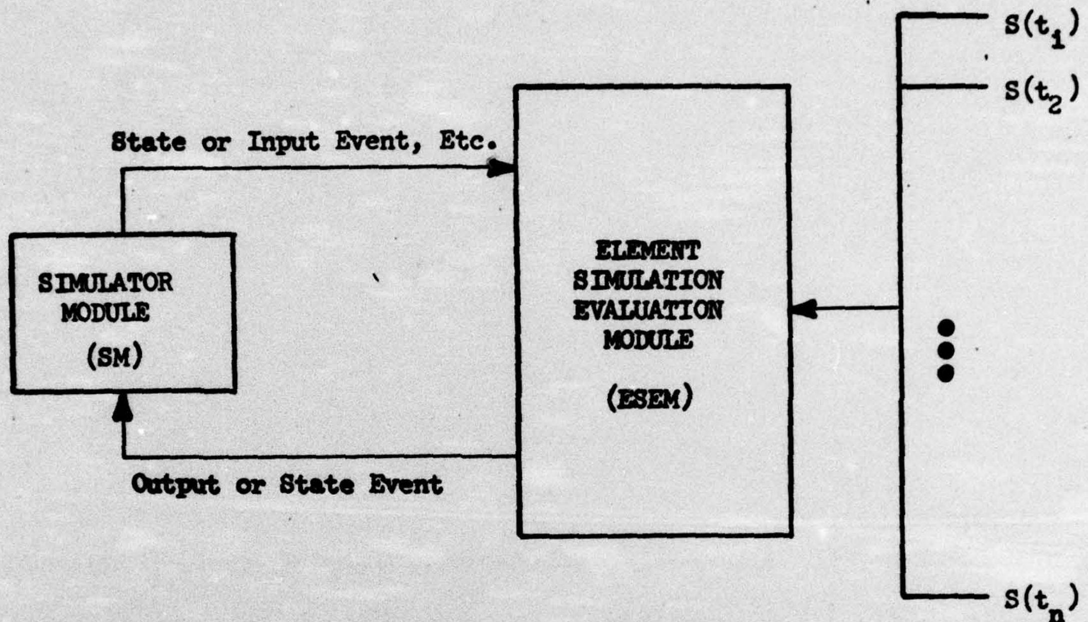


FIGURE 5.1: Structure of Simulation System

The simulator module (SM) passes the following data to the ESEM, namely,

1) name of element;
2) type of element;
3) input or state event (value and line);
4) current total state for this element; and
5) fault number or good circuit event.

Using this data and the corresponding $S(t_1)$, the ESEM executes $S(t_1)$ and passes the following data back to the SM.

1) output events, and

2) the time at which they occur.

The main modules in the SM are shown in Figure 5.2.

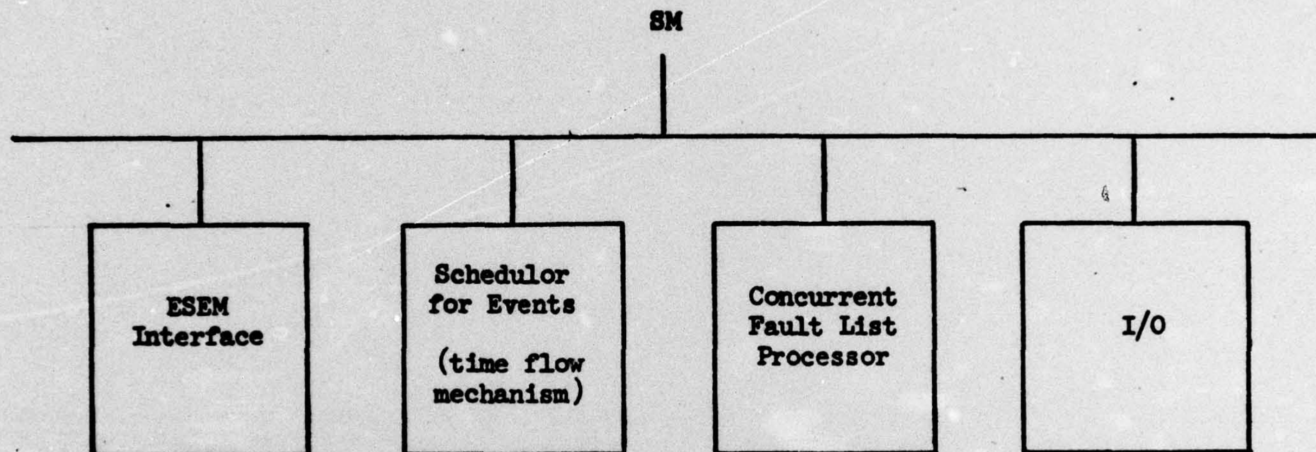


FIGURE 5.2: Main Modules in Simulator Module.

5.3 Data Structure for Circuit

The data structure used by the simulator is a subset of that used by the stimulus generation module, i.e., it is based upon the concept of descriptors associated with each element. Each element type is associated with a descriptor type. It may be the case, however, that several of the elements modeled functionally during stimulus generation are modeled at the gate/flip-flop level during simulation. This may be done in order to obtain more detailed timing analysis, or else to treat internal IC faults.

The major components in this data structure are:

a) the fan-out list associated with each element output (the fan-in list is not explicitly needed);

b) a pointer to the concurrent fault list (CFL) for this element; and

c) element information consisting of i) type of element, ii) delay parameters.

The actual value for the lines associated with this element for both the good and faulty circuits is stored in the concurrent fault list.

5.4 Simulator Data Structure

There are six basic line events which can occur, namely 0/1, 1/0, 0/u, 1/u, u/0 and u/1. In general, we will denote a line event by $e = v_1/v_2$. Because our elements have multiple inputs as well as multiple outputs, events at elements are actually vector events, e.g., $(e_1, e_2, \ldots, e_n)$ would represent a vector input event to a n input element. For simplicity of presentation, we will restrict our attention to scalar events, i.e., just one line event/element will be processed at one instance of time.

The total state of an element will be denoted by (I, S, Θ), where I = inputs, S = internal state, and Θ = outputs. The current total state of a good element is denoted by CGTS, and that of an "error element" by CETS, where CGTS $\neq$ CETS.

The concurrent fault list associated with an element E has the format shown in Figure 5.4.

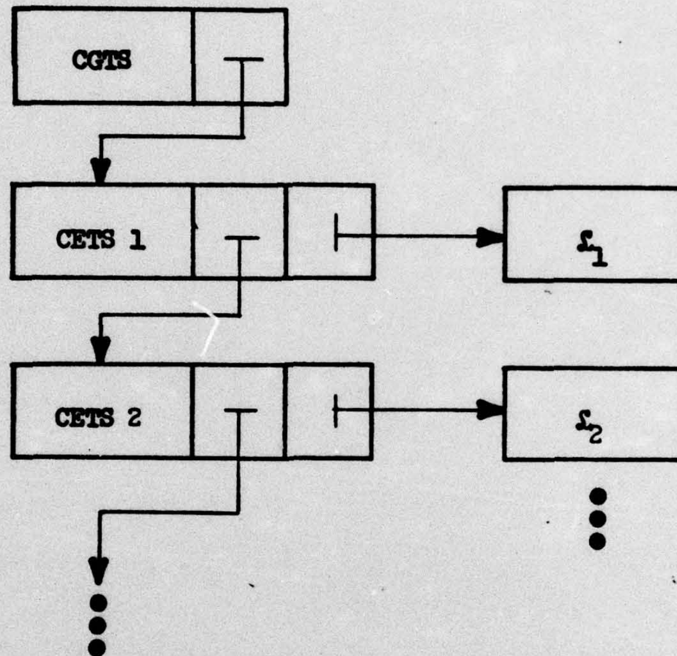


FIGURE 5.4: Concurrent Fault List Associated with Element E

$\mathcal{L}_i$ is a list of faults, each of which has the current error total state CETS i . Note that CGTS $\neq$ CETS i for each i, and CGTS i $\neq$ CETS j for i $\neq$ j.

The abstract format of the events list which resides in the scheduler, is shown in Figure 5.5. Here, $t_i < t_{i+1}$. Each event in list t_i represents a signal change v_1/v_2 on some line ℓ in either the good and/or faulty circuit. The actual structure of an event entry <u>event</u> j is quite complex.

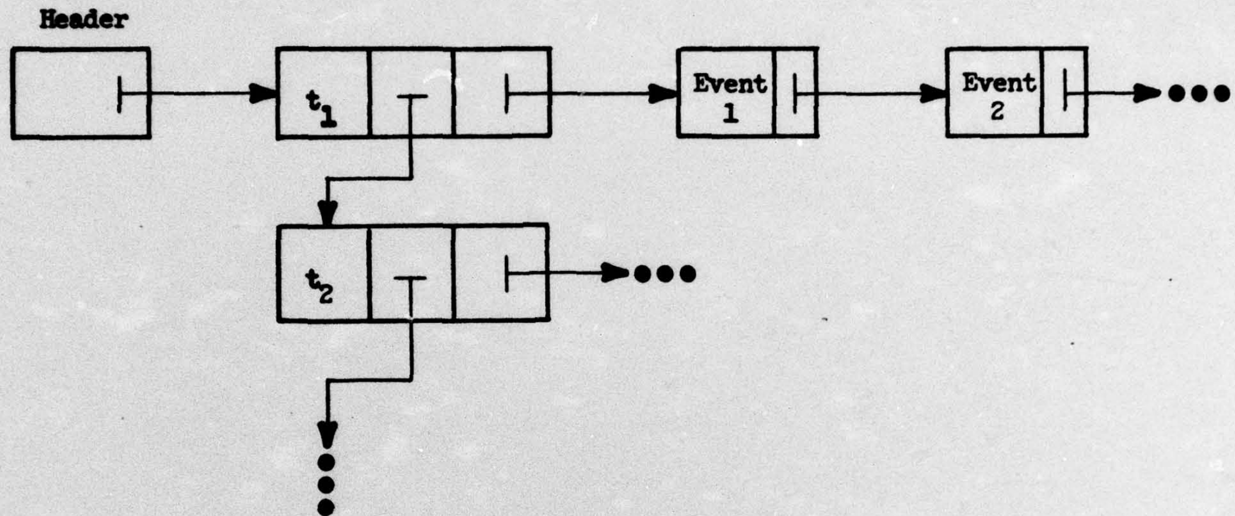


FIGURE 5.5: Abstract Structure of the Events List

If in the good circuit an event $e_0 = v_1/v_2$ occurs at an input A of element E, the same event can be applied to all the entries in the concurrent fault list associated with E, with the following exceptions, namely, faulty circuits in which the signal A is static at the value:

a) v_1 - these correspond to the "new visible " faults. They form 'add candidates" for the CFL of E.

b) v_2 - these correspond to the "new invisible" faults. They form "delete candidates" for the CFL of E.

c) v_3 - these faults have been and will remain "visible."

The following notation is used:

e_0 = good event = v_1/v_2

e_0 = 0 $\Rightarrow$ no activity in the good circuit

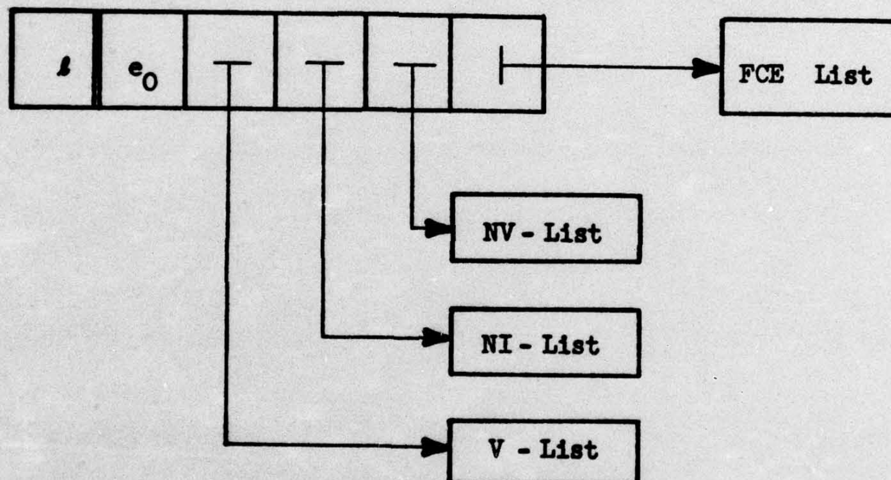
NV = new visibles $\{f_i\}$

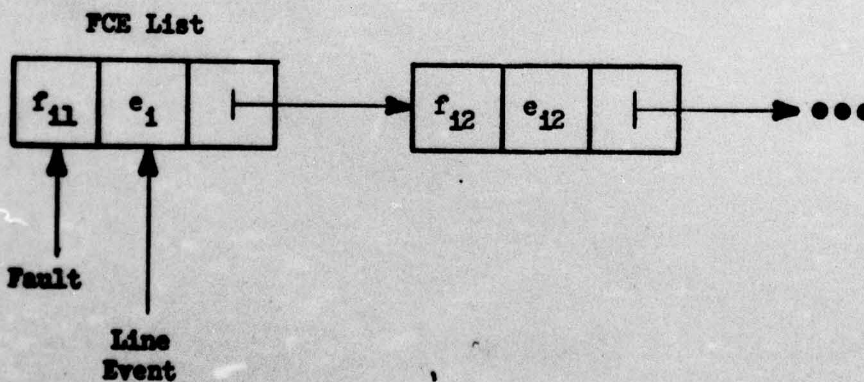
NI = new invisibles $\{f_j\}$

V = visible $\{f_k\}$

FE = faulty circuit event.

An event in a concurrent simulation takes on the following structure.





Here, ℓ is the line on which the event occurs; e_0 is the event in the good circuit. The FCE list accumulates all the events associated with this line but are created by faults.

The following functions access and/or modify the events list:

1. NEXT-TIME (time, nomore)

   If we are currently working on the header for time t_i, then this function returns the header for t_{i+1} for the argument time. If no such header exists, nomore = 1, else nomore = 0.

2. NEXT-EVENT (event, nomore)

   This function extracts the next event for the current time t_i. If no such event exists, then nomore = 1.

3. SCHEDULE (time, event)

   This function inserts the new event, event, into the list for time = current time + delay.

5.5 Simulator Operation

The basic structure of the simulator module is shown in Figure 5.6. The main concept here is to take events off the events list, process them, update the CFL, and schedule new events.

Notation

CGTS - current good total state

NGTS - next good total state

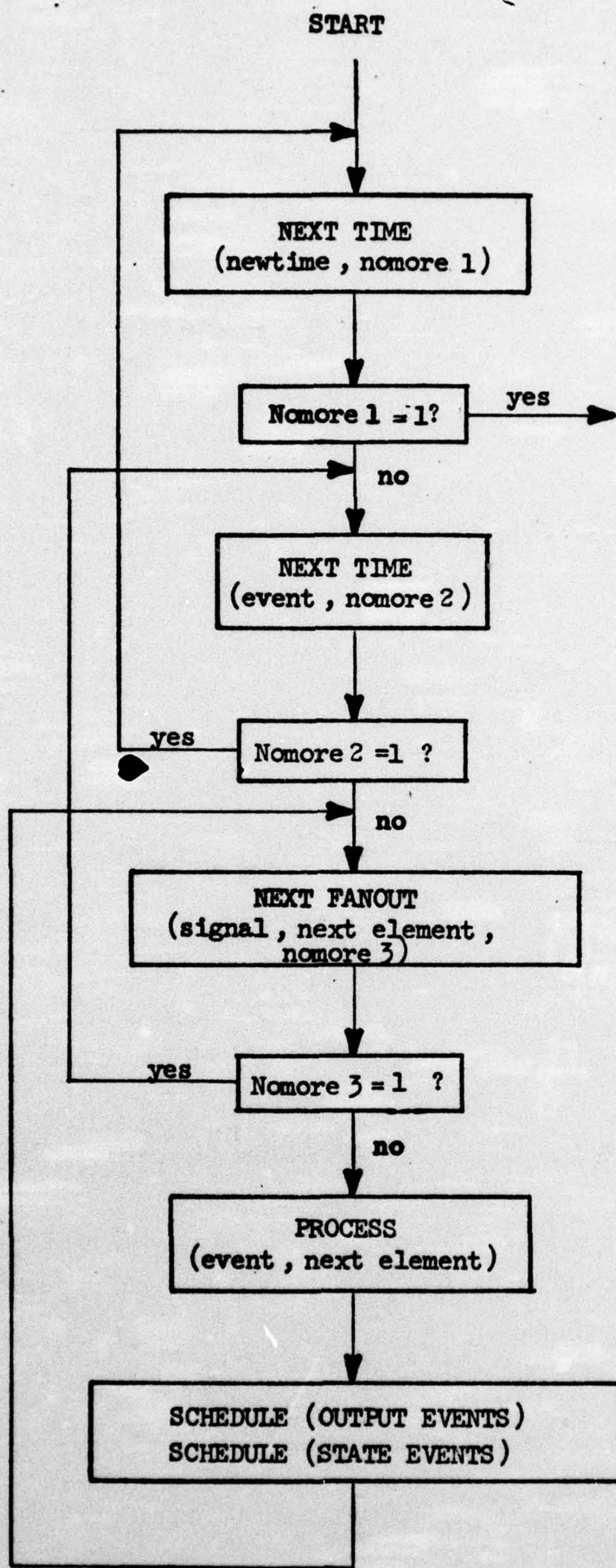


FIGURE 5.6: Basic Simulation Flowchart.

CETS - current erroneous total state

NETS - new erroneous total state

OES - current erroneous output value.

The program for processing an event (IEV) given the current concurrent fault list (CFL) in order to product a new event (OEV) and the new concurrent fault list (NCFL) is described next.

<u>Concurrent Fault List Processor</u>

```
begin 1  initialize  OEV = empty,  NCFL = empty
  if 1  e_0 ≠ 0  then 1
        e'_0 = evaluate  (E , CGS , e_0)
        update  OEV·f_0·event = e'_0          "e'_0 = v'_1/v'_2"
        for every  f_k ∈ IEV·V do
           begin 2
                remove  (f_k)  from  CFL
                add  (f_k , CFTS)  to  NCFL
           end 2
        for every  f_j ∈ IEV·NI do
           begin 3
                remove  (f_j)  from  CFL
                  if  CETS ≠ NGTS  then add  (f_j , CETS)  to  NCFL
                  if  e'_0 ≠ 0  and  OEV = v'_2  then add  (f_j)  to
                        OEV·NI
           end 3
```

```
for every f_i ∈ IEV•NV do
   begin 4
      if f_i ∈ CFL then
         begin
            remove (f_i) from CFL
            add (f_i , CETS) to NCEL
         end
      else
         add (t_i , CGTS) to NCFL
      if e'_0 ≠ 0 and OEV ≠ v'_2
         then add (f_i) to OEV•NV
      end 4
else "nothing"
for every f_ℓ ∈ IEV•DE do
   begin 5
      if f_ℓ ∈ CFL then
         begin 6
            remove (f_ℓ) from CFL
            e'_ℓ = evaluate (CETS , e_ℓ)
         end 6
      else e' = evaluate (CGTS , e_ℓ)
      if NETS ≠ NGTS then add (f_ℓ , NETS) to NCFL
      if e'_ℓ ≠ 0 then
         if e'_ℓ ≠ e'_0 then add (f_ℓ , e'_ℓ) to OEV•DE
            else add (f_ℓ) to OEV•V
         else "e'_ℓ = 0" .
            if e'_0 ≠ 0 then
```

if $v_1' = v_\ell$ **then** add (f_ℓ) to OEV·NV

if $v_2' = v_\ell$ **then** add (f_ℓ) to OEV·NI

if $v_3 = v_\ell$ **then** add (f_ℓ) to OEV·V

end 5

for every $\mathcal{L}_1 \in$ CFL do

begin 6

$e_1' =$ evaluate (CETS i , e_0) add $(\mathcal{L}_1)$ to NCFL

if $e_1' \neq 0$ **then**

if $e_1' \neq e_0'$ **then** add $(\mathcal{L}_1, e_1')$ to OEV·DE

else add $(\mathcal{L}_1)$ to OEV·V

else "$e_1 = 0$"

if $e_0' \neq 0$ **then**

if $v_1' = v_1$ **then** add $(\mathcal{L}_1)$ to OEV·NV

if $v_2' = v_1$ **then** add $(\mathcal{L}_1)$ to OEV·NI

if $v_3' = v_1$ **then** add $(\mathcal{L}_1)$ to OEV·V

end 6

end 1

5.6 Fault Insertion

When an output event (good circuit) occurs on a line, the associated complementary s-a-fault $(s-a-\bar{v}_2)$ entry is added to the NCFL, if it is not already in the list, and an entry made into OEV·DE.

When an input event (good circuit) is processed, then the associated s-a-fault $(s-a-\bar{v}_2)$ becomes a "new visible."

When an input faulty event (IEV•DE) occurs at E, we check this event to see if it is compatible with E, e.g., the fault line i s - a - δ cannot place the value $\bar{\delta}$ on line i.

In addition to line s-a - fault, we can extend our faults to include state s - a - values in functional models of devices such as shift registers, counters, etc.

5.7 Initialization

We assume the initial value of all lines not specified to be 0 or 1 by the user, have the value u. To initialize the circuit the fault free simulator is run in a zero-delay mode, i.e. $\Delta = 0$ for all elements. Only events of the form u/0 and u/1 are scheduled.

5.8 Data for Primary Inputs

We assume that the data for the primary inputs have been previously generated, e.g., via the test generation algorithm. We also assume that real-time inputs are possible. These inputs are assumed to be pre-placed into the events scheduler.

5.9 Input Skew Buffering

If it is desired to process input skew, then input events of the form $v/\bar{v}$ should be modified to take the form v/u (Δ), $u/\bar{v}$ (Δ).

5.10 Miscellaneous Items

All classical aspects dealing with simulation, such as fault collapsing, oscillation control and supression, etc., will be handled using classical techniques.