

Controllability and Observability

(COT)

Control - ability to apply complete set of tests to a subsystem via external inputs (or via previously tested subunits).

Observation - ability to observe outputs of a subsystem via external outputs (or via previously tested subunits)

Control and Observability are significant issues in testing of sequential circuits as well as systems of interconnected units.

Sequential Circuit Design (cor) With Control + Observation Points

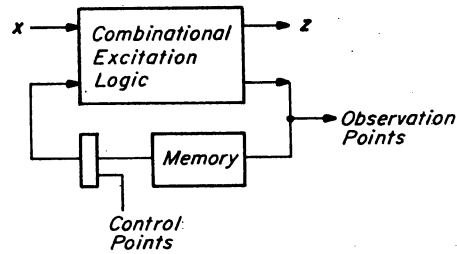


Figure 5.41 Sequential Circuit with Observable and Controllable Feedback Paths

Example 5.10: Consider the system represented by the graph of Figure 5.40. The observability and controllability relationships between modules are expressed by the arcs of the graph.

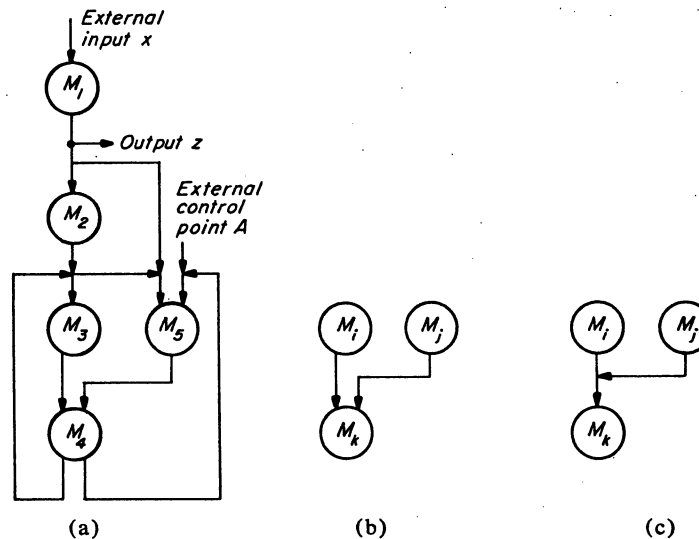


Figure 5.40

An arc from M_i to M_j indicates that M_i controls M_j . Separate arcs from M_i and M_j to M_k indicate that M_i and M_j collectively control M_k (Figure 5.40(b)). Arcs from M_i and M_j to M_k which combine (Figure 5.40(c)) indicate that M_k can be controlled by either M_i or M_j individually. Thus M_4 is controlled by the combined effects of M_3 and M_5 and is observed partially by M_3 and partially by M_5 . Logic associated with the test point A, the input x and the output z , is assumed to be previously verified or hardcore. M_1 can be diagnosed since it is controlled by x and observed by z , both of which are assumed verified. For each of the remaining unverified modules (M_2, M_3, M_4, M_5) it is not possible to both control and observe it through previously verified modules or hardcore. Hence additional test points are needed. By placing an external observation point B at the output of M_5 , this module can be verified since it can be controlled by M_1 and A which have been previously verified. At this stage M_2 can be controlled by M_1 and observed by M_5 , both of which are previously verified modules and thus M_2 can be verified. By placing an observation point C at the output of M_3 , M_3 can then be verified, following which M_4 can be verified since it can be controlled and observed by M_3 and M_5 , both of which have been previously verified. □

LOGIC LEVEL SIMULATION

Logic simulation is the process of building and exercising a model of a digital circuit on a digital computer. By exercising we mean the evaluation of signal values in the modeled circuit as a function of time for some applied input sequence.

There are two main applications for a logic simulator. The first is in the evaluation of a new design. Here, the logic designer is interested in testing for logical correctness, as well as timing and signal propagation characteristics. He may desire information related to race, hazard and oscillatory circuit conditions. For LSI circuits, where design errors are very costly and breadboarding is impractical, logic simulation is an invaluable aid.

A second application for logic simulation exists in the area of fault analysis. Here, the test engineer or logic designer may desire information related to what faults are detected by a proposed test sequence, what is the operational characteristic of the circuit under specific fault conditions, or what degree of fault resolution is obtainable with a given test sequence? These and other questions can be dealt with effectively by the process of fault simulation. The table of Figure 4.1 gives a detailed list of potential applications for a logic simulation system.

1. Hardware Design Verification
 - a. Verify logical correctness
 - b. Timing analysis
 - Delay models
 - Race and hazard analysis
 - c. Initialization analysis
2. Fault Analysis
 - a. Fault coverage
 - b. Timing analysis under fault conditions
 - c. Initialization under fault conditions
 - d. Fault induced hazards and races
 - e. Evaluation of test point effectiveness
 - f. Evaluation of self checking circuitry
 - g. Evaluation of fail-safe circuitry
 - h. Evaluation of roll back hardware-software
3. Software Development
 - a. Debugging software to run on hardware not yet implemented
 - b. Development of diagnostic software programs and microcode for computer systems.

Figure 4.1 Applications of simulation

4.1 AN OVERVIEW OF A LOGIC SIMULATION SYSTEM

The input information to a logic simulator usually consists of the following:

- (1) Description of the circuit to be simulated.
- (2) Input data to be simulated.
- (3) Initial value of memory states.
- (4) Faults to be simulated, if any.
- (5) Signals to be monitored.

The circuit description (1) consists of the topology of the circuit and the circuit element types, along with a list of primary inputs (pi's), and primary outputs (po's) including test points. The specification of delay parameters, as well as circuit restrictions such as fanin, and fanout restrictions may also be included (frequently on an optional basis). If the initial state of some devices is not known, or some input lines are not controllable, a don't know or unknown value, denoted by u , can be used. Faults may consist of lines being open, shorted to ground or power, signal lines shorted, adjacent pin shorts, etc. Permanent and intermittent faults may also be simulated. A high level language is often used for describing input sequences, desired output format, circuit description, etc.

The basic structure of a logic level simulation system is shown in Figure 4.4.

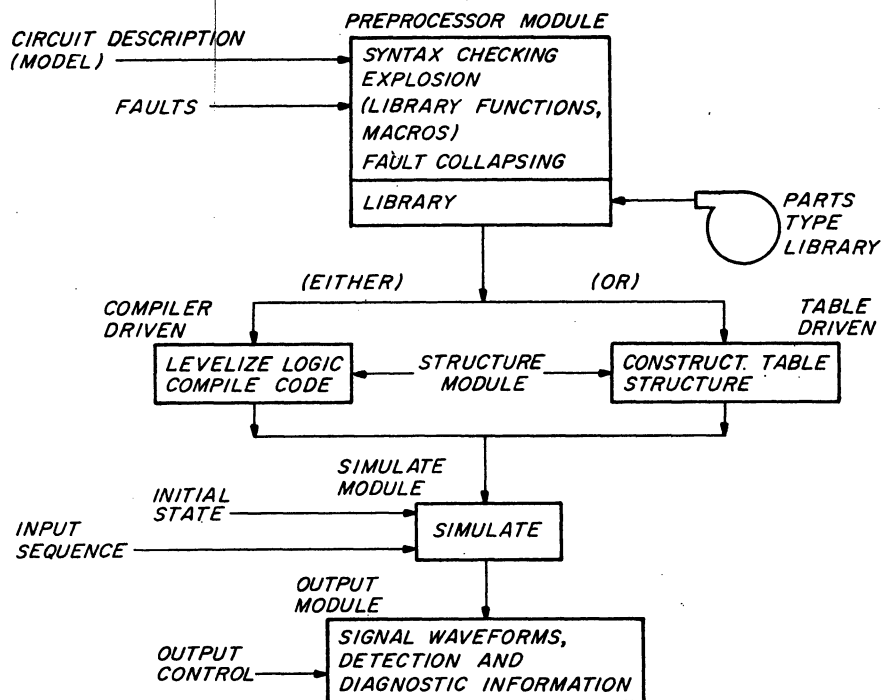


Figure 4.4 Basic structure of a logic level simulation system

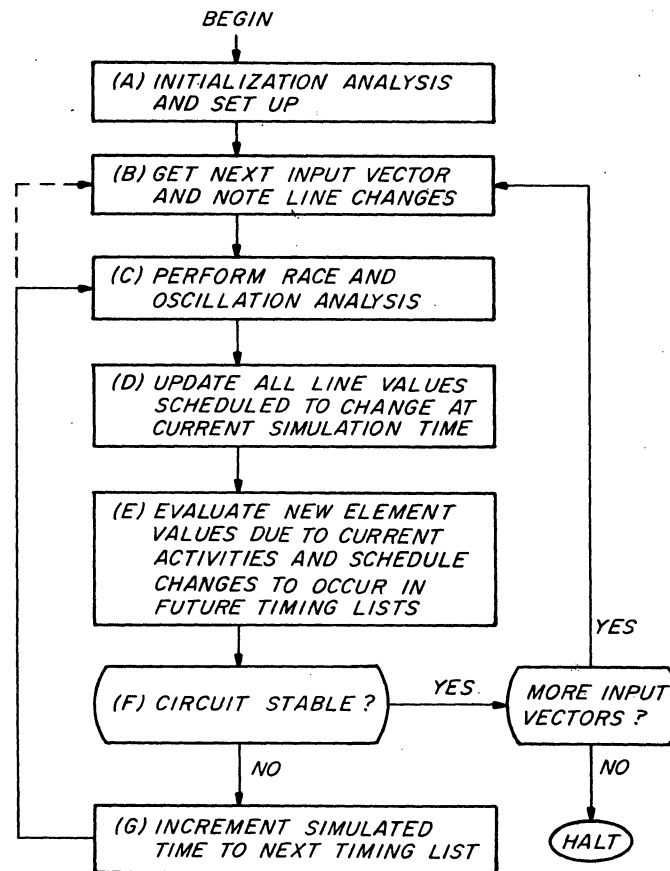


Figure 4.5 Basic structure of simulation module

4.1.6 Simulate Module

In a simulation system time is quantized into units, ~~by one unit~~. We will consider a table driven *activity directed* simulator. By *activity directed* we mean that an element is not simulated unless there is a signal change in at least one of its inputs. This follows from the fact that for the type of elements we are considering, all output value changes occur in response to some input value change.

Figure 4.5 shows the essential behavior of the Simulate Module for such a simulator. Block A deals with setting up the initial state of the circuit. In Block B a new input vector is read. Only line *changes* need be processed since only they define activity. Skipping over Block C temporarily, in Block D all changed line values caused by the new input are updated and in Block E elements affected by current activity are simulated. If any of these elements change states they are scheduled to change at some future time. All such activities are sequentially processed, after which the simulated time is incremented by one unit, and race and oscillation analysis is performed and processed accordingly (Block C). Simulators allowing for real time dynamic inputs would branch from Block G to B. For static simulation, the computation loop consisting of Blocks C, D, E, F, G is usually repeated until the circuit stabilizes, at which time a new input vector is processed.

4.1.8 CPU Simulation Time

Logic simulation is inherently a slow process. The primary reason for this is that a simulator processes elements sequentially, while in the actual circuit signals propagate along numerous paths simultaneously. Consider a synchronous circuit containing N elements and operating at a 1 MHz clock rate. If t seconds are required to simulate each element then we require $t \cdot N$ seconds to simulate the entire circuit for one input vector. Typically there are about 2-3 nonequivalent stuck-at faults per element (gate). Hence, to simulate this circuit under each fault condition would require at least $2tN^2$ seconds. Assuming that the length L of a test sequence for a circuit is proportional to N , i.e., $L = kN$, then to simulate the circuit for a given input test sequence for each fault would require $2tkN^3$ seconds. The table of Figure 4.6 indicates some representative calculations for $t = 10^{-5}$ seconds and $N = 10^4$. Since 10^{-1} seconds are required to simulate one clock cycle of this circuit ($1\mu\text{s}$ of actual operation), there is a "slow-up" of $10^{-1}/10^{-6} = 10^5$ in simulation with respect to real time operation.

	One Input Vector	$L = N$ Input Vectors ($k = 1$)
Good machine simulation	$tN = 10^{-1}$	$tN^2 = 10^3$
Faulty machine simulation	$2tN^2 = 2 \times 10^3$	$2tN^3 = 2 \times 10^7$

$$t = 10\mu\text{s} = 10^{-5} \text{ sec.}$$

$$N = 10^4$$

Figure 4.6 Simulation time

Because of this situation, much of the research in the area of simulation has been directed toward developing techniques to reduce simulation time. An example of such a technique is the process of activity directed simulation.

If elements are ordered so that an element is evaluated after all of its inputs have been evaluated,

The first step in ordering is called *levelizing*. Levelizing consists of assigning a level number (non-negative integer) to each element and signal line. The output signal of an element is assigned the same level number as the element itself. If k_i is the logic level of element i , and if element β has inputs from elements $i_1, i_2, \dots, i_a$ then

$$k_\beta = 1 + \max(k_{i_1}, k_{i_2}, \dots, k_{i_a}). \quad (1)$$

Logic level values are assigned according to the following procedure.

Procedure 4.5 (Circuit levelizing):

- (1) Assign all primary input lines (x) and feedback lines (y) the logic level 0.
- (2) For any element not yet assigned a logic level, and such that all its input lines have been assigned a logic level, assign this element and its output lines a level value as defined by equation (1). $\square$

Figure 4.23 shows a simple circuit and the logic levels associated with each element. Once logic levels have been assigned the elements can be

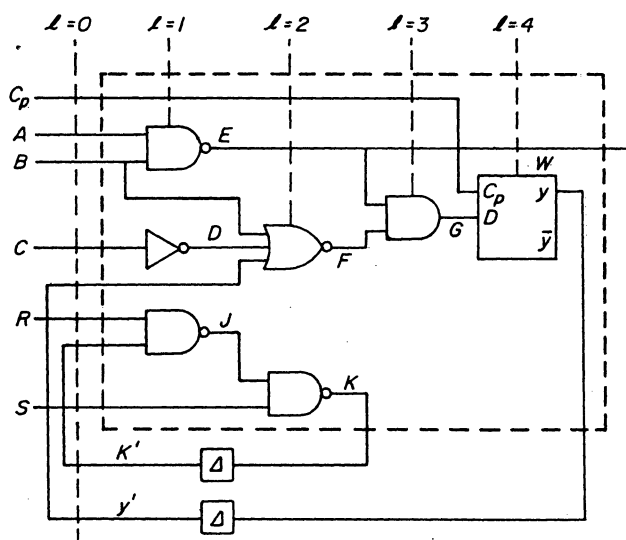


Figure 4.23 Simple circuit showing logic level for each element

sorted in ascending order, i.e., all i -th level elements are grouped together, followed by all $i + 1$ -st level gates, for $i = 1, 2, \dots$. For the circuit of Figure 4.23 the following levels are determined:

level 1: E, D, J

level 2: F, K

level 3: G

level 4: W

4.4.2 Event (Activity) Directed Simulation

During a single simulation computation scan frequently only a small percentage of the signal lines change values. The ratio of lines which change values to the total number of lines in the circuit is called the *activity* and is denoted by A . Usually A is between 2-10%. Because of this fact it appears wasteful to simulate all the logic when only a small portion of the circuit is going to change. This observation has led to the concept of *event (activity) directed simulation (selective trace)*.

An *event* is a change in value of a signal line. The output of a stable element i will only change value when one or more of its inputs change value. Hence element i need only be simulated when an event occurs at one of its inputs. When an event does occur, then every element to which this line fans-out is called *potentially active*. If we simulate these potentially active elements, we will find that some are active while others are not. A significant reduction in computation can often be achieved by simulating only potentially active elements rather than all the elements.

A table-driven event-directed simulator has two other advantages over a compiler-driven simulator, namely it handles combinational, synchronous and asynchronous circuits with equal ease, and delays can be processed in

4.4.3 Data Structure

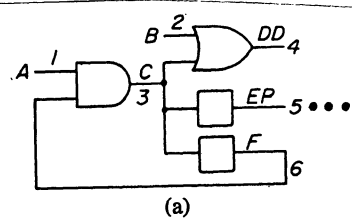
In a table-driven simulator the circuit is represented by a set of tables (vectors or matrices) indicating the important attributes of each element, such as:

- (1) the logic value of the output lines.
- (2) the type of element, e.g., AND, OR, etc.
- (3) the delay associated with this element.
- (4) the name of the inputs to this element (the fanin list).
- (5) the name of each element which is a load to this element, (the fanout list).
- (6) the state of the element, if it has memory.

Simple Data Structures

Numerous data structures exist for representing a circuit within a computer. We shall consider two representative structures. Figure 4.27(b) represents a portion of a simple data structure associated with the circuit shown in Figure 4.27(a). In this data structure we assume each element has a single output, hence the name of the element and the name of its output signal are the same. Also each element and signal has both an alphanumeric user's name, denoted by NAME, and an internal name, called its INDEX, which is a positive integer. This INDEX is used as the argument in vector arrays to address the piece of information desired. Gate C (INDEX = 3) has a fanin of two (lines A and F) and a fanout of three (gates F, EP, and DD). The structure shown consists of two tables, ELEMENT and IOLIST (input-output list). The ELEMENT i table has the following columns:

- (1) NAME(i) indicates the users name for the i -th element.
- (2) VALUE(i) is the current logic value for the i -th element or line.
- (3) NFO(i) is the fanout of line i .
- (4) NFI(i) is the fanin of element i .



ELEMENT TABLE

INDEX	NAME	VALUE	NFO	NFI	FIL	FOL	TYPE	DELAY	INTERNAL STATE	IOLIST TABLE
1	A									
2	B									4
3	C	0	3	2	1, 6	3	AND			5
4	DD						OR	2, 3		6
5	EP									
6	F									
7	.									1
8	.									6
										⋮

Figure 4.27 (a) Portion of a circuit (b) Data structure for table driven simulation

- (5) FOL(i) is a pointer to the IOLIST indicating the first element to which element i fans-out. The remaining NFO(i) - 1 elements are found in the next NFO(i) - 1 cells in this list.

Similarly there are fields specifying the fanin lines, type, delay characteristics, etc. of each element. Most simulation systems are limited by the number of elements and/or the size of IOLIST. The last factor is the most critical. Usually such factors as number of feedback lines, number of flip-flops, etc., are immaterial.

This data structure is quite versatile and we shall now indicate how it is used by the simulate module. Assume we desire to simulate gate i . Then $TYPE(i)$ gives us its type. $FIL(i)$ points to its first input, hence $VALUE(FIL(i))$ is the value of this input. By carrying out the same process on $FIL(i) + 1, \dots, FIL(i) + (NFI(i) - 1)$ all input values to i can be found. From this, the output value of i can be calculated. If an event has occurred the elements $FOL(i), FOL(i) + 1, \dots, FOL(i) + (NFO(i) - 1)$ can be placed onto the appropriate list or the scheduler.

4.5.1 Parallel Simulation

The data structure used for representing the input values of an element is closely related to how the element can best be evaluated. Consider an AND gate having two inputs A and B . If the value of A and B is stored in the i -th bit of words (or strings) A and B , respectively, then the AND of these two words will have the value of the output of the gate in the i -th bit of the resulting computer word. If the word (string) has W bits, then W different problems can be handled simultaneously (i.e., in parallel), each in a different bit of the word. For example, we may desire to simulate a circuit for a given input sequence and K different values for the initial state, or for K different input sequences, or for K different faults. In any case, W independent problems can be simultaneously simulated as long as all element evaluation operators employed process each bit independently of all others. Hence arithmetic operators cannot be used. A simulator operating in this manner is called a *parallel simulator*. Almost all compiler driven simulators are parallel simulators. In a compiler driven simulator, W cases can be simulated in parallel in about the same time that one case can be processed, where W is the number of bits in a word. Some time is lost in packing and unpacking the data words, and not using special

time-saving techniques such as stimulus bypass. We refer to the processing of W problems in parallel as a *pass*. If there are a total of K problems to be handled, e.g., K faults, then via parallel simulation we would require $\lceil K/W \rceil$ passes, where $\lceil x \rceil$ is the smallest integer greater than or equal to x .

Parallel simulation can also be employed in a table-driven event-directed simulator. In doing so, when $v(i)$ is compared with $v'(i)$, two vectors are compared. If they differ in any bit, then an event exists. Hence in parallel simulation it might be expected that the total activity of the circuit would substantially increase as a function of W . It is interesting to note that when simulating faults, this is not the case. That is, W faults can be selected in such a manner that the activity in the simulator is not substantially increased, i.e., it is much less than $A \cdot W$, where A is the normal activity in the circuit. Usually the set of W faults are selected from the same physical portion of the circuit.

4.6.2 Parallel Fault Simulation

In Section 4.5.1 we discussed the concept of parallel simulation and showed how W problems can be processed simultaneously. For parallel fault simulation these W problems refer to W different faults. The most common way to process these faults is via *fault injection*. That is, while simulating an element the logical effect of the faults are injected into the computation. For example, consider the two input AND gate $A = B \cdot C$, and the parallel simulation of the four faults associated with gate A , as specified in Figure 4.37.

	<u>fault</u>	<u>bit position</u>
output fault	$A\ s-a-0$	bit 1
output fault	$A\ s-a-1$	bit 2
input fault	$B\ s-a-1$	bit 3
input fault	$C\ s-a-1$	bit 4

Figure 4.37

We assume that the value of a signal X is stored in word X , and that a table-driven event-directed simulator is being employed. First gate A is flagged to denote that faults have been assigned to it. Masks are also associated with each of the terminals (A, B, C) of gate A and are used to inject faults. In the process of simulation, when it becomes necessary to simulate gate A , the flag is observed and special processing is carried out. This processing uses a mask M_1 to inject a logical 1 into bit 3 of word B forming B' , and a mask M_2 to inject a 1 into bit 4 of word C forming C' . Then $A = B' \cdot C'$ is computed, and a 0(1) is then injected into bit 1(2) of word A . Note that in the remaining bit positions of the data words for each signal other fault conditions in the circuit can be processed.

The masks used for fault injection are generated during the preprocessing of the circuit for fault simulation. Except for the overhead required to generate the masks and carry out fault injection, W faults can be processed almost as rapidly as one fault. That is, no time is lost in element evaluation due to the fact that W faults are being processed simultaneously. Of course we assume that element evaluation is carried out using logical word operators and is bit organized. By selecting the W faults from a common portion of the hardware, the activity in the simulator will often not be substantially greater than that for a single fault.

It is relatively easy to simulate multiple faults using this scheme. For example, two stuck faults can be handled by simply processing these two faults in the same bit position. Complex (non stuck type) faults can also be handled. For example, if a NAND gate becomes an AND gate, it is possible to write a special routine to inject the desired value, that is, to take the current output in bit i and complement it.

Processing W faults simultaneously is called a pass. If there are N_F faults to be processed, $\lceil N_F/W \rceil$ passes are required. It is possible to reduce the number of passes by simulating several *independent* faults (multiple or single) simultaneously. Let f be a fault defined on signal line i . Let S_f be the set of signal lines in a circuit which can be effected by the value of line i . Then faults f and g are said to be *independent* if and only if $S_f \cap S_g = \phi$, i.e., faults f and g cannot affect any common portion of the circuit. In a parallel fault simulator independent faults can be simulated in the same bit position. The least upper bound on the number of independent faults which can be processed simultaneously in the same bit position is the number of output lines p in the circuit. Hence the minimum number of passes is $\lceil N_F/(W \cdot p) \rceil$. The main disadvantage with this procedure is that of identifying the independent faults. If this cannot be done efficiently, then the time saved in simulation is lost.

In compiler driven simulators, faults are usually processed by logic level. If it is desired to simulate W faults affecting the output of elements having logic level L , then the simulation proceeds by first simulating all elements at logic levels 1 through L . At this time the faults are injected, and subsequently the elements at logic level $L + 1, L + 2, \dots$ are simulated. This procedure is quite efficient, especially for 2-valued logic and a zero delay model.

The efficiency of parallel simulation depends on the width W of a computer word as well as the instruction set and architecture of the host machine. Assuming a fixed memory cycle time and instruction execution time, by doubling W twice as many faults can be processed in the same time. If the host machine has string operators, then even further efficiencies can be achieved. For example some machines have memory to memory instructions of the form (OP, ADR, LG) where OP is an operator (op-code), ADR is the address of the first word (or byte) in a string of words (bytes), and LG is the length of this string. Then the instruction sequence

<i>NOP</i>	<i>A</i>	256
<i>AND</i>	<i>B</i>	256
<i>SLW</i>	<i>C</i>	256

would process 256 W bit words (bytes) in forming the quantity $C = A \cdot B$. Typically long string operations outperform accumulator type operations, even when register to register instructions are available.

Deductive Simulation

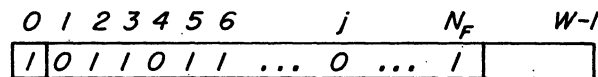
SIM

Deductive simulation is based upon the concept that if the inputs to a logic element and the effect of faults on the circuit for these input values are known the output of the element under all fault conditions can be deduced. This is accomplished by using the concept of *fault list propagation*.

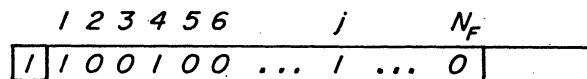
Recall that using the concept of parallel simulation, N_F faults require $\lceil N_F/W \rceil$ passes to process. If $W = N_F$, then only a single pass is required. Figure 4.38(a) illustrates the case where $W \geq N_F + 1$. Here bit 0 is associated with the fault free circuit, and bit j with fault number j . For signal X and time t , $X^0 = 1$, $X^1 = 0$, $X^2 = 1$, where X^j is the value of line X in the circuit under fault number j . These values are elements of the string X stored in the host computer memory, where X^j is stored in bit position j .

In general, only a small fraction of the faults will cause an error on a particular signal line X . Figure 4.38(b) illustrates the *characteristic vector* string corresponding to string X , in which bit j is 1 if and only if $X^j \neq X^0$.

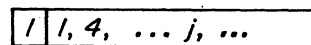
Since string C usually has very few 1's, it is more efficient to store C as a list containing only the indices $j > 0$ corresponding to faults for which $X^j \neq X^0$. This list is called a *fault list*.



(a)



(b)



(c)

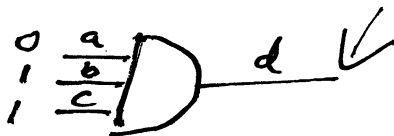
Figure 4.38 Evolution of a fault list (a) String X (at time t) used in parallel simulation (b) Characteristic string C associated with X (c) Fault list L associated with X

Two-valued Deductive Simulation

In this section we assume that all line values are binary. Using the concept of deductive simulation we can, in principle, process all faults via one pass simulation. This is accomplished by simulating the fault free circuit, and by associating with each line α in the circuit a set or list L_α which contains the name (index) of every fault which produces an error on line α . These fault lists are then propagated through the circuit. The simulator is typically table-driven event-directed. However, there are now two types of events, *logic events* which occur when a signal line takes on a new logic value, and, *list events* which occur when a fault list L_α changes, i.e., either gains or loses one or more entries.

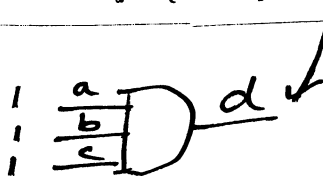
To illustrate this concept, consider an AND gate having inputs $a = 0$, $b = 1$, and $c = 1$. Then the output d has value 0. The faults which cause an error at line d , are the fault $d s-a-1$, and any fault in L_a which is not in L_b or L_c , i.e., any element in the set $L_a \cap \bar{L}_b \cap \bar{L}_c$, where $\bar{L}_a$ is the complement of L_a and consists of all faults not in L_a . We therefore have

$$L_d = \{d s-a-1\} \cup \{L_a \cap \bar{L}_b \cap \bar{L}_c\}.$$



Similarly if we consider the same gate but where $a = b = c = 1$, and hence $d = 1$, any fault which causes a 0 on any input line will cause an error at d , i.e., $d = 0$. Hence, we have

$$L_d = \{d s-a-0\} \cup L_a \cup L_b \cup L_c.$$



The following procedure can be used to construct the output fault list associated with an element realizing an arbitrary switching function ψ from the input fault lists.

Procedure 4.8 (*Propagation of fault lists*):

- (1) If the value of ψ in the normal circuit is 0(1), let $E(x_1, x_2, \dots, x_n)$ be a sum of products or product of sums Boolean expression for $\psi(\bar{\psi})$ in terms of the inputs to the element $x_1, x_2, \dots, x_n$. If the value of an input variable x_i in the normal circuit is 0, replace (all appearances of) x_i in E by L_{x_i} , and all appearances of $\bar{x}_i$ by $\bar{L}_{x_i}$; if the value of x_i in the normal circuit is 1, replace x_i by $\bar{L}_{x_i}$ and $\bar{x}_i$ by L_{x_i} . Replace $\cdot$ by $\cap$ and $+$ by $\cup$.
- (2) If d is the element output add $d_1(d_0)$ to the fault list obtained in (1). $\square$

To illustrate this result, again consider the case $a = 0, b = c = 1$, for a 3-input AND gate. Since $d = 0$, we have

$$E(a, b, c) = d = a \cdot b \cdot c \mid_{a=L_a, b=\bar{L}_b, c=\bar{L}_c}.$$

Hence the resulting fault list is

$$(L_a \cap \bar{L}_b \cap \bar{L}_c) \cup d_1.$$

For the case $a = b = c = 1$, since $d = 1$,

$$E(a, b, c) = \bar{d} = \bar{a} + \bar{b} + \bar{c} \mid_{\bar{a}=L_a, \bar{b}=L_b, \bar{c}=L_c}.$$

Hence the resulting fault list is

$$(L_a \cup L_b \cup L_c) \cup d_0.$$



In Figure 4.39 we illustrate the propagation of fault lists through a circuit. Let α_1 and α_0 denote, respectively, the faults line α s-a-1 and line α s-a-0. Assume the input test vector is $a = b = f = i = h = 1$ and $d = 0$.

$$L_a = \{a_0\}, L_b = \{b_0\}, L_c = \{L_a \cap L_b\} \cup \{c_1\} = \{c_1\}$$

$$L_d = \{d_1\}, L_e = L_d \cup L_c \cup \{e_1\} = \{c_1, d_1, e_1\}$$

$$L_f = \{f_0\}, L_g = L_e \cup \{g_1\} = \{c_1, d_1, e_1, g_1\}$$

$$L_h = \{h_0\}, L_i = \{i_0\}, L_j = L_e \cup \{j_1\} = \{c_1, d_1, e_1, j_1\}$$

$$L_k = L_i \cup L_h \cup \{k_1\} = \{i_0, h_0, k_1\}$$

$$L_l = L_j \cup \{l_0\} = \{c_1, d_1, e_1, j_1, l_0\}$$

$$L_m = \{L_k \cap \bar{L}_l\} \cup \{m_1\} = \{i_0, h_0, k_1, m_1\}$$

$$L_n = \{L_g \cap \bar{L}_f\} \cup \{n_1\} = \{c_1, d_1, e_1, g_1, n_1\}$$

$$L_p = L_n \cup L_m \cup \{p_1\} = \{c_1, d_1, e_1, g_1, n_1, i_0, h_0, k_1, m_1, p_1\}.$$

This result implies that any fault in L_p will be detected by the input test vector $(a, b, d, f, i, h) = (1, 1, 0, 1, 1, 1)$.

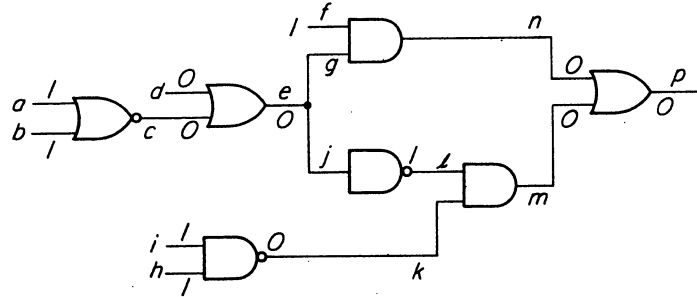


Figure 4.39 Propagation of fault lists through a circuit

Now assume that the input h changes

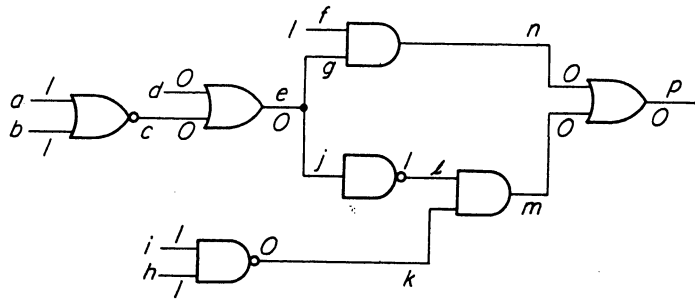


Figure 4.39 Propagation of fault lists through a circuit

to 0. The event directed approach can be used so that all lines need not be re-evaluated.

$$h: 1 \rightarrow 0$$

$$L_h = \{h_1\} \quad k: 0 \rightarrow 1$$

$$L_k = (L_h \cap \bar{L}_i) \cup k_0 = \{h_1, k_0\}$$

$$m: 0 \rightarrow 1 \quad L_m = (L_h \cup L_k) \cup m_0 \\ = \{h_1, c_1, d_1, e_1, f_1, l_0, k_0, m_0\}$$

$$p: 0 \rightarrow 1 \quad L_p = (L_m \cap \bar{L}_n) \cup p_0 \\ = \{h_1, k_0, m_0, f_1, l_0, p_0\}$$

Note that faults c_1, d_1, e_1 are not detected by this new input since they are related to reconvergent fanout lines. The other fault lists remain unchanged.

Now suppose $h: 1 \rightarrow 0$

$$L_h = \{h_1\}$$

$k: 1 \rightarrow 1$ (No event - no change)

$$L_k = \{h_1, k_0\} \quad (\text{a change} \rightarrow \text{list event}) - \text{delete } h$$

$$L_m \rightarrow \{c_1, d_1, e_1, f_1, l_0, k_0, m_0\} \quad (\text{delete } h_1)$$

$$L_p \rightarrow \{k_0, m_0, f_1, l_0, p_0\} \quad (\text{delete } h_1)$$

Note that when L_α is computed, to determine if a list event has occurred the new L_α must be compared with the old L_α , denoted by $\tilde{L}_\alpha$, before the latter is destroyed, i.e., we must determine whether or not $L_\alpha = \tilde{L}_\alpha$.

Sequential Circuit
Fault list propagation
is (much) more complex
due to

feedback (fault feeds back
on itself)

propagation through memory
elements

Initialization (three valued logic)

Example 4.6: For the gate and input conditions shown in Figure 4.42,

$$L_d^0 = (L_a^1 \cap L_b^1 \cap (\overline{L_c^0 \cup L_c^u})) \cup d_0$$

$$L_d^u = (L_a^u \cup L_a^1) \cap \overline{L_b^0} \cap \overline{L_c^0} -$$

$$\underbrace{L_a^1 \cap L_b^1 \cap \overline{L_c^0 \cup L_c^u}}_{111 \text{ input}}$$

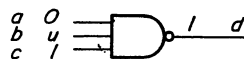


Figure 4.42

One disadvantage of this approach is that two lists exist for each line, and the complexity of the processing required to propagate the fault lists through an element is more than doubled. $\square$

Concurrent Simulation

Consider a single output n -input combinational element E . At a particular simulation time t for fault f this element has input values $a_1^f, a_2^f, \dots, a_n^f$ and output value b^f , where $f = 0$ refers to the fault free circuit. Consider the set of faults $F' \subset F$ (where F is the set of all single stuck faults) such that for each $f \in F'$, either $b^f \neq b^0$, or $a_j^f \neq a_j^0$ for some input j . That is, for each $f \in F'$ either an input or the output b of E is in error. We can now associate with b a super fault list (SFL) S_b , where each entry in this list is of the form

$$f; a_1^f a_2^f \dots a_n^f; b^f$$

for $f \in F'$. We illustrate this list in pictorial form in Figure 4.44. Here, the first element in S_b is 9;00;1 which implies that fault 9 causes inputs (0,0) and output 1. The elements are sorted by fault index, and can be stored in a sequential table or in a list structure as for L_b . We call S_b a super fault list since the deductive fault list L_b is a subset of this list in the sense that $f \in L_b$ if and only if $b^f \neq b^0$. It is therefore evident that concurrent simulation requires more storage than deductive simulation.

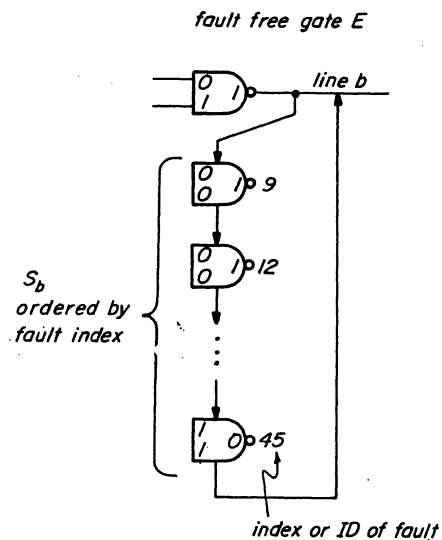


Figure 4.44 Super fault list S_b associated with a line b

In deductive simulation the fault free circuit is explicitly simulated and the faulty circuits are deductively simulated. In concurrent simulation only those elements in the faulty circuit which do not agree, in terms of input and/or output values, with the same element in the fault free circuit are explicitly simulated. In parallel simulation elements in both the fault free and faulty circuit are explicitly simulated even when they agree with each other.

Example 4.7: We will illustrate the computation of the super fault lists S_b , using the circuit shown in Figure 4.45(a) to show how these lists differ from L_b . List $\tilde{S}_b$ and $\tilde{L}_b$ refer to the present (initial) fault lists.

For gate c there is an entry $a_1; 10; 1$ in $\tilde{S}_c$ which implies that if a is $s-a-1$, the input to the gate will be $a = 1, b = 0$ rather than $a = 0, b = 0$, and the output is 1. Now assume that line a changes to a 1. The deductive simulator would re-evaluate c and see that no logic activity has occurred. However, since $L_c \neq \tilde{L}_c$ a list-event has occurred. Therefore $L_e = L_c \cup L_d \cup e_1$ is computed. Then, since $L_e \neq \tilde{L}_e, L_i = L_e \cup L_h \cup i_1$ is computed. Thus it is necessary to recompute 3 fault lists and check to see if they differed from their previous status. In general, with long lists this can be quite time consuming.

We will now consider how a concurrent simulator would handle this

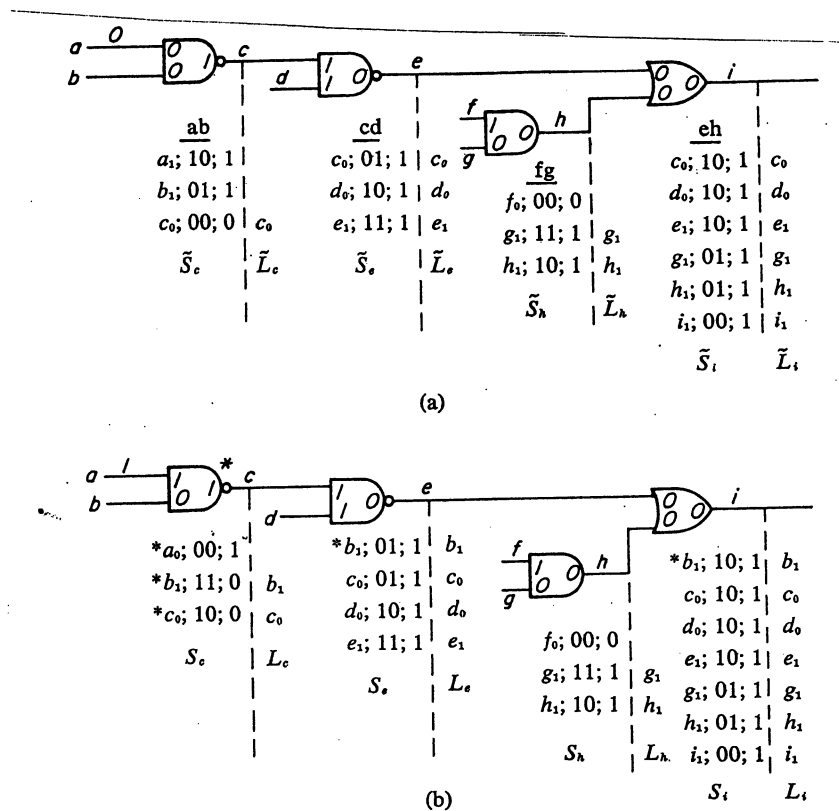


Figure 4.45 Comparison of lists for deductive and concurrent simulation

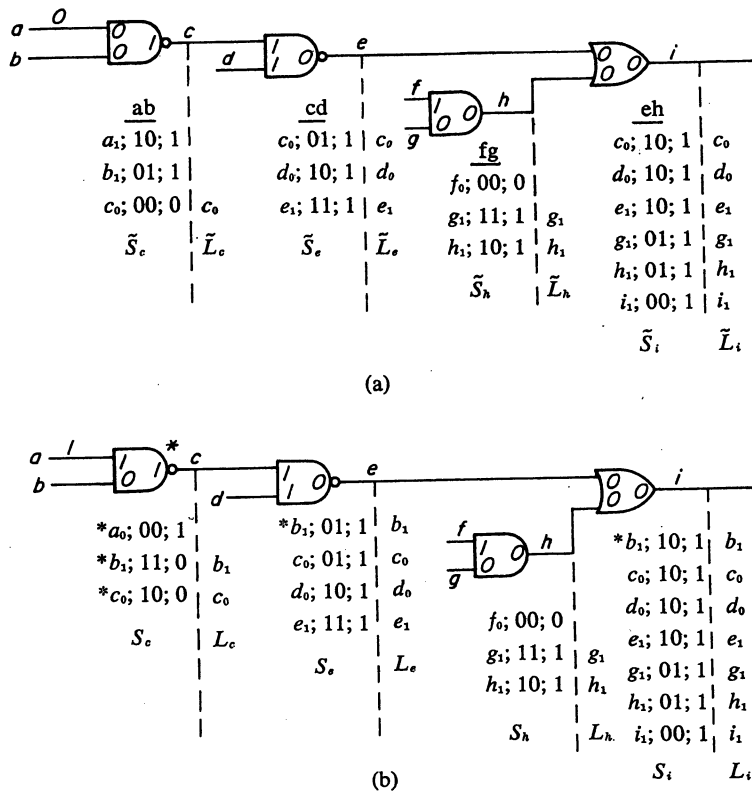


Figure 4.45 Comparison of lists for deductive and concurrent simulation

same problem. When line a changes to a 1 there is an activity and the entire list $\tilde{S}_c$ as well as gate c must be processed. For gate c , only its input changes. For the entry $a_1; 1; 0; 1$ since $a = 1$, this term is deleted and the term $a_0; 0; 0; 1$ is added to $\tilde{S}_c$. Similarly, $b_1; 0; 1; 1$ changes to $b_1; 1; 1; 0$ and $c_0; 0; 0; 0$ changes to $c_0; 1; 0; 0$. These computations produce the list S_c . Now *all* entries, and *only* these entries, which have either caused a change in the logic value of line c , or which create a new element to be added to a list due to an output error are processed. Such events are called *logic list events*. In this example, only entry b_1 qualifies for further processing. The list $\tilde{S}_e$ is scanned looking for a b_1 entry. (Note that all entries on a list S are ordered by fault index. In our example the order is alphabetic.) Since the first entry encountered on $\tilde{S}_e$ is c_0 , b_1 is not an entry on $\tilde{S}_e$ and therefore the entry $b_1; 0; 1; 1$ is added to $\tilde{S}_e$, forming S_e . Again this causes a logic list event and $\tilde{S}_i$ is processed next by adding $b_1; 1; 0; 1$ to it, thus forming S_i . Those gates (entries) which

have been processed are flagged in Figure 4.45(b) by an *. There are six such entries. For the fault list propagation technique three gates, (c , e , and i) would be processed and their associated fault lists recomputed. Hence the concurrent simulation process can potentially lead to a reduction in CPU time. This savings is related to the ratio of logic list events to the total length of the fault sets. $\square$

Comparison Concurrent vs Deductive

Sim 25

The reason the input vector $(a_1^f, a_2^f, \dots, a_n^f)$ is stored along with its entry in S is because each element E^f is actually simulated one at a time, as long as there is an activity associated with it. That is, if gate c has some activity due to faults i, j and k , then three separate copies of c are simulated and these entries are placed in S . Because of this, the input values for these three versions of gate c must be known.

Since faulty circuits and good circuits are processed (scheduled) concurrently, this technique is called *concurrent simulation*. Because each entry in S is processed separately, and its inputs are known, fast simulation techniques such as table look-up can be used. Thus one significant difference between deductive and concurrent simulation is related to the fact that a concurrent simulator only processes the active circuits. The difference in processing performance between deductive and concurrent simulation due to this fact becomes more apparent when a fault f causes a circuit to oscillate. We assume the fault free circuit has stabilized. In deductive simulation it is necessary to repeatedly process each gate in the circuit loop which is oscillating since a list event is present. Each gate processed requires a complex calculation over all its input lists, even though only one fault f is causing the oscillation. In concurrent simulation, only the single fault f is processed.

When an event occurs during concurrent simulation, an entry must be placed in the scheduler at the appropriate time. This entry is actually a 4-tuple consisting of (1) the name of the element E to be simulated; (2) the name of the fault causing simulation; (3) the name (index) of the input line on which an event occurred which caused E to be scheduled; and (4) the new value of this line.

The main disadvantages of the concurrent simulation procedure are that it requires more storage than the deductive procedure, and when an event occurs, due to a fault, the super fault list must be serially scanned to see if the fault is an entry on this list. However, this scan process is less time consuming than set intersection or union. The super fault list must also be processed when a logic event in the fault free circuit occurs, since it affects the input values to the element.

Since activity in both the fault free and faulty circuits must be scheduled, the length of the scheduler list is longer than in a deductive simulator. The same elements, however, are processed in both cases. That is, a change in an input causes an element E to be processed at time frame t in a deductive simulator if and only if it is scheduled in a concurrent simulator. In the deductive method, however, it will be scheduled by a simple entry such as E (multiple entries are possible but can be eliminated if desired), while in a concurrent simulator there will probably be multiple entries. These entries should be ordered by fault index (the second field) so they can be efficiently processed.

Finally, for multi-valued logic, such as the five valued system described earlier, concurrent simulation is ideal since each element in a super fault list is handled individually, hence all the tools available for fault free simulation are again available for fault simulation. In addition, complex delay models as well as functional modeled elements can be dealt with quite easily.

Deductive

Slower

Set Intersection
and Union

Concurrent

More storage
required

Super fault
list scan