\[
A_3(k) = \text{Maj}_k(A_4, A_1 \oplus A_2 \oplus B_1, A_2 \oplus B_1 \oplus B_2, A_2 \oplus A_4 \oplus B_3, A_1 \oplus B_1 \oplus B_3)
\]
\[
A_4(k) = \text{Maj}_k(A_4, A_1 \oplus A_2 \oplus B_1, A_1 \oplus A_3 \oplus B_2, A_2 \oplus A_3 \oplus B_3, B_1 \oplus B_2 \oplus B_3).
\]

Each \(A_i(k)\) is determined by a 3-out-of-5 majority element. These majority elements still give the correct output if at most one of the terms \(A_i, A_j, \ldots\) has a component failure. One can apply these majority elements to the design of a four-stage single-failure-tolerant counter in a manner similar to that for the previous three-stage counter.

**Functional Partitioning and Simulation of Digital Circuits**

MELVIN A. BREUER, MEMBER, IEEE

Abstract—In this paper we present a method for obtaining a functional partitioning of the logic of a computer. It is shown that given a basic function to be performed, such as addition, the computer logic can be partitioned into four disjoint sets, namely the active information logic \(\mathcal{I}\), the semiactive flip-flops \(\mathcal{F}\), the activated control logic \(\mathcal{C}\), and the dormant logic \(\mathcal{D}\). Techniques involved in implementing the partitioning algorithm such as an event-directed simulator and three-value simulation are discussed. An application of this partitioning scheme as part of a large logic simulation system is described.

Index Terms—Background simulation, design automation, event-directed simulation, functional logic partitioning, simulation, three-value simulation.

I. INTRODUCTION

PROCEDURES for the automated partitioning of computer logic have previously been discussed [1]–[5]. Most of these techniques carry out the partitioning process based upon the number of wires between the various components and ignore the functional aspects of the design. It has long been a desirable goal to develop a procedure for automatically carrying out functional partitioning. A number of applications for this type of partitioning exists, such as in simulation systems, in specifying the contents of the circuit cards in a computer, and in the retrieval of specific subcircuits of a large design whose description is stored in a mass memory.

In this paper we will present an algorithmic procedure for obtaining a functional partitioning of a circuit with respect to a given partial state, referred to as a partially specified initial condition, and a given subset of the flip-flops and primary outputs of the circuit. In particular, the logic is partitioned into four disjoint sets, called the active information logic \(\mathcal{I}\), the activated control logic \(\mathcal{C}\), the semiactive flip-flops \(\mathcal{F}\), and the dormant logic \(\mathcal{D}\).

To illustrate and informally define these various partitions, consider the logical circuit of a digital computer. Assume the current state of the computer is such that when the next clock pulse occurs, the sum of two registers, for example \(A\) and \(B\), will be stored in register \(A\). Then the \(A\) register, and that logic that actually implements the addition, is referred to as the active information logic. For example, this may consist of a 32-bit parallel adder. Note that the actual information being added does not influence the selection of what constitutes this active information logic.

The sum is placed into register \(A\) at this clock time because the operation code and phase counter are in the appropriate state to allow this information to be gated into this register. That control logic that decodes the operation code and phase counter and allows this transfer of information to take place is referred to as the active control logic.

REFERENCES

to the area of logic simulation. In Section VI we give some computer results, and in the Appendix we describe the event-directed simulator used in the partitioning algorithm.

The three-value logic scheme used in the partitioning algorithm. In Section III we present the partitioning algorithm, and in Section IV we expand upon the techniques used for determining the functional relationship between the next state of a flip-flop based upon the current state of the circuit. In Section V we describe an application of this partitioning procedure.

II. PRELIMINARY REMARKS

Notation

In Fig. 2 we indicate a portion of a logic circuit. Each gate is denoted by a unique integer, and the output signal (line) of each gate is denoted by the same integer. If is a flip-flop (F/F), its “true” output signal is labeled and its “false” output signal is labeled , where . The logic value or state of any line is denoted by , where . The assignment implies that the value of line is unspecified. In general, there are signal lines, namely . is the set of all gates having an input signal generated by element , e.g., . denotes the set of all inputs to element , e.g., . External inputs to a circuit are called primary input (PI) lines. In Fig. 2, the set of primary input lines is . Those lines whose value can be observed via external measurements are called primary output (PO) lines, e.g., line 7. Flip-flop types can be identified by their input labels; R–S for set–reset; T for trigger; D for delay. The flip-flops are assumed to be clocked. In Fig. 3 we illustrate the general model for the circuit to be partitioned.

Three-Value Simulation

It is possible to simulate the operation of a circuit even when the inputs to the gates have values from the set . A flip-flop logic scheme is used in the partitioning algorithm. For the operation code , those control lines set to “1” are ADD and HOLD, while all other control lines are set to “0.” For this case, we have and . When carrying out an addition, is said to be an active flip-flop because its next state cannot be determined from its present state. This is not true of flip-flops ; hence, it is not considered to be active. To determine the next state of , one must determine the output of gates 1, 2, and 9 based upon the value of , , , . Hence, these gates are also active. Gates 3, 4, 5, 10, 11, 12, and 13 are considered to be active control gates since their output is specified by the operation code, and each of these gates is an element of some input network of at least one active gate.

For the operation code , which is used to load register , we have and .

Section II of this paper covers some preliminary information on notation as well as a description of the three-value logic scheme used in the partitioning algorithm. In Section III we present the partitioning algorithm, and in Section IV we expand upon the techniques used for determining the functional relationship between the next state of a flip-flop based upon the current state of the circuit. In Section V we describe an application of this partitioning procedure.

1 The concept of a three-value logic simulator has been independently formulated and reported in [6], [7]. Such a system is used for design verification and hazard detection.
Lemma 1: If \( i \) is a gate and \( v(i) = 0 \) or \( 1 \), then changing any inputs to \( i \) from \( x \) to \( 0 \) or \( x \) to \( 1 \) will not change the value of the output \( v(i) \).

Theorem 1: Given two partitions \( F_0, F_1, F_x \) and \( F_0, F_1, F_x \) on \( F \), where \( F_0 \supseteq F_0 \) and \( F_1 \supseteq F_1 \). If, for both cases, the gates are simulated in the same order and starting with the same initial conditions, then \( G_0 \supseteq G_0 \) and \( G_1 \supseteq G_1 \).

Proof: If the logic is acyclic, then this result is trivial. For cyclic logic the problem is more involved since in simulating a gate \( i \), it is not always possible to have first simulated all the gates in \( U^{-1}(i) \). Hence, the simulation results can be a function of the initial conditions and the order in which gates are simulated. In any event, since \( F_0 \supseteq F_0 \) and \( F_1 \supseteq F_1 \), by Lemma 1 we have that \( i \in G_i \) implies that \( i \in G' \), where \( e \in \{0, 1\} \), and \( i \in G_x \) implies that \( i \in G_0 \cup G_1 \cup G_2 \).

Note that this result includes the case where \( F_1 = \phi \), i.e., where \( F_0, F_1, F_x \) represents a TSIC with respect to \( F_0, F_1, F_x \).

It should be noted that in some cases carrying out a three-value logic simulation produces unspecified gate values when logically the value of a gate is either 0 or 1. One such case is illustrated in Fig. 5, where simulation would produce \( v(3) = x \). Using normal rules of binary Boolean algebra, the actual value of \( v(3) \) would be 1, since the inputs to this gate are complements of one another.

Stack Storage

Since the algorithms to be presented employ stacks, we shall now briefly define our stack notation.

A first in last out push down stack (PDS) is a particular storage structure. The operation \( \text{push}(i; S) \) will place the integer whose label is \( i \) on stack \( S \). The operation \( \text{pop}(i; S) \) will remove the last integer placed on stack \( S \), and label that integer \( i \). For example, if 5 is the last integer placed on \( S \), then we would denote the result of \( \text{pop}(i; S) \) by writing \( i = 5 \). The operation \( \text{push}(i; S) \) would then replace 5 on the (top of the) stack. If the stack is empty, we would obtain \( i = \phi \). In some cases we desire to put an element on a stack only if it is not already on the stack, or only if it has not previously been placed on the stack and subsequently removed. To accomplish this, we employ the operators \( \text{push}^*(i; S) \) and \( \text{pop}^*(i; S) \) as well as a table of markers (flags), one flag per stack per integer. If the flag corresponding to the integer labeled by \( i \) for stack \( S \) is reset, then \( \text{push}^*(i; S) \) will set this flag and execute \( \text{push}(i; S) \); otherwise \( \text{push}^*(i; S) \) does nothing. \( \text{pop}^*(i; S) \) executes a \( \text{pop}(i; S) \) and then resets the stack flag of the integer labeled by \( i \).

III. Functional Logic Partitioning

Let \( F \) be a subset of the total set of flip-flops and primary output lines in a given circuit. The partitioning procedure to be described in this section is said to carry out a functional partitioning with respect to the pair \((F, \mathcal{F})\).

Assume we have carried out a PSS with respect to some given PSIC, say \( \tilde{F} \). For an arbitrary TSIC with respect to \( \tilde{F} \), we wish to determine the minimal amount of logic that must be considered in order to determine the next stable state of all elements in \( \mathcal{F} \), given the current state and the results of the PSS. The active information logic, obtained by the functional partitioning algorithm, will constitute this minimal set of logic.

We will first illustrate the concepts involved on the circuit shown in Fig. 6. In this figure we have also indicated the re-
sults of carry out a PSS for the PSIC \( \mathcal{F} \) specified by \( F_0 = \emptyset \), \( F_1 = \{3, 4, 5\} \), and \( F_2 = \{1, 2, 6, 7, 8, 9, 21\} \). Let \( \mathcal{F} = \{6, 7, 8, 9\} \). We will now analyze each flip-flop in the set \( \mathcal{F} \).

Consider flip-flop 6 whose state is unspecified by \( \mathcal{F} \). Its set input is at 0, while its reset input is \( x \). For any TSIC with respect to \( \mathcal{F} \), if \( v(6) = 0 \), then flip-flop 6 will not change state. If \( v(6) = 1 \), then the input logic to the reset line must be evaluated in order to determine the next state of flip-flop 6, since for some TSIC it may be possible for the reset condition \( v(17) = 1 \) to occur. Since the output of gates 10 and 17 must be evaluated in order to determine the value of this reset line, these two gates along with flip-flop 6 are considered to be active.

The input to trigger flip-flop 7 is 1; hence, this flip-flop will change state independently of which TSIC with respect to \( \mathcal{F} \) is applied. Such a flip-flop is considered to be semiactive, since it undergoes a state transition, and its next state is completely specified by its current state and \( \mathcal{F} \).

The input to delay flip-flop 8 is \( x \), and hence it appears that it may be an active flip-flop. Note, however, that if \( v(8) = 1 \), then \( v(20) = 1 \), and if \( v(9) = 0 \), then \( v(20) = 0 \). Hence, for any TSIC with respect to \( \mathcal{F} \), the next state of flip-flop 8 is completely specified by its current state. Hence, this flip-flop is considered to be dormant. This technique of determining the functional dependency of the next state of a flip-flop based upon its current state will be the subject of Section IV.

Since the input to trigger flip-flop 9 is 0, for any TSIC with respect to \( \mathcal{F} \), this flip-flop will not change state, and hence is also considered to be dormant.

Finally, gate 12, whose output is 1, is considered to be the active control logic.

For this example we have that \( \mathcal{F} \subseteq F_x \). In general, this need not be the case. If flip-flop \( i \in \mathcal{F} \cap (F_0 \cup F_1) \), then by considering the value of the inputs to this flip-flop, it can be classified as being either active (\( \mathcal{I} \)), semiactive (\( \mathcal{S} \)), or dormant (\( \mathcal{D} \)). This classification is given in Table I.

In general, the pair \( (\mathcal{F}, \mathcal{S}) \) generates a natural partitioning of the logic circuit \( C \) into four components, labeled \( \mathcal{I}, \mathcal{S}, \mathcal{P}, \mathcal{D} \). The active information logic with respect to \( (\mathcal{F}, \mathcal{S}) \), denoted by \( \mathcal{A} \), consists of the following elements: 1) \( i \in \mathcal{I} \) if \( i \in \mathcal{F} \), and either a) \( i \) is a PO and not the output of a flip-flop,
and \(v(i) = x\) (such a PO line is said to be active), or b) \(i\) is a flip-flop, \(i \in F_i\), and the next state of \(i\) implied by some TSIC \(F' = (F_0, F_1, \phi)\) with respect to \(F\) cannot be determined from the partially specified initial conditions

\[
F_0, F_1 \cup \{i\}, F_x = \{i\} \quad i \in F_1
\]

or from

\[
F_0 \cup \{i\}, F_1, F_x = \{i\} \quad i \in F_0,
\]

or c) \(i\) is a flip-flop, \(i \in F_i U F_1\), and the next state of \(i\) cannot be determined from the results of the PSS. (Such a flip-flop is said to be active.)

2) \(j \in S\) if \(j\) is not a flip-flop, \(v(j) = x\), and \(j \in U^{-1}(k)\) where \(k \in S\). That is, there exists a path through the circuit from gate \(j\) to the input of a flip-flop \(i \in S\), or to a PO \(i \in S\), such that the output of each gate along this path is an \(x\).

The activated control logic with respect to \((F, S, S)\), denoted by \(\mathcal{G}\), consists of all elements \(i\) such that \(i\) is not a flip-flop, \(v(i) \in \{0, 1\}\), and \(i \in U^{-1}(j)\) where \(j \in S \cup \mathcal{G}\).

The semiactive logic with respect to \((F, S)\), denoted by \(\mathcal{E}\), consists of all flip-flops \(i \in S\), which may change state for some TSIC with respect to \(F\), but whose next state can be determined either by \(F\) alone, and/or by the knowledge of the present state of \(i\) alone.

The remaining logic in circuit \(C\) is considered to be dormant, and is denoted by \(\mathcal{D}\).

Having precisely defined the various partitions of the circuit \(C\), we will now present an algorithm, shown in Fig. 7, for generating these sets. The process begins by first carrying out a PSS with respect to a PSIC \(F\). This is followed by a number of steps that determine whether or not a flip-flop is active, semiactive, or dormant. With the set of active flip-flops and PO lines stored in stack \(S\), we enter label \(1\) and determine \(\mathcal{G}\), which is stored in stack \(T\) when we are done. For our example, we get the following results for each transition through the loop beginning with the block labeled \(1\) and \(S = (6)\).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Stack Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(S = (6), T = (\emptyset))</td>
</tr>
<tr>
<td>1</td>
<td>(S = (17), T = (6))</td>
</tr>
<tr>
<td>3</td>
<td>(S = (10), T = (17, 6))</td>
</tr>
<tr>
<td>4</td>
<td>(S = (\emptyset), T = (10, 17, 6))</td>
</tr>
</tbody>
</table>

If we execute the algorithm starting at \(1\), with stack \(S\) consisting of all \(i\) such that \(v(i) \in \{0, 1\}\) and \(i \in U^{-1}(j)\) for some \(j \in S\), and with \(v(i) = x\) in block \(A\) changed to \(v(i) = \bar{x}\),
then we would end up with \( \mathcal{C} \) stored on stack \( T \). For our case we would have the following.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Stack Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( S=(12), T=(\phi) )</td>
</tr>
<tr>
<td>1</td>
<td>( S=(\phi), T=(12) )</td>
</tr>
</tbody>
</table>

Finally, the algorithm also produces the set \( \mathcal{J} \) of semi-active flip-flops.

Let \( F_0, F_1 \) and \( F_x \) and \( F_0', F_1', F_x' \), be two partitions on \( F \), and let \( \mathcal{J} \) and \( \mathcal{J}' \) be their associated active information logic sets.

**Theorem 2:** For a given \( \mathcal{J} \), if \( F_0 \supseteq F_0' \) and \( F_1 \supseteq F_1' \), then \( \mathcal{J}' \subseteq \mathcal{J} \).

**Proof:** Note that if \( F_0 = F_1 = \phi, \mathcal{J} \) is the entire circuit. If \( F_x = \phi \), then \( \mathcal{J} = \phi \). In general, we are given \( F_0, F_1 \) and \( F_x \neq \phi \), and the associated \( \mathcal{J} \) set. If we take an element \( k \) out of \( F_x \) and place it in \( F_1 \) or \( F_0 \) and carry out a new PSS, then by Lemma 1 either no gate outputs change state, or else the output of one or more gates goes from \( x \) to \( 0 \) or \( 1 \). It is not possible for an output of a gate to go from \( 0 \) to \( 1 \), \( 1 \) to \( 0 \), or \( x \) to \( x \). Hence, when we execute the algorithm to obtain the corresponding \( \mathcal{J}' \), we get a subset of \( \mathcal{J} \). We can repeat this process of removing elements from \( F_x \) until we have transformed \( F_0, F_1, F_x \) to \( F_0', F_1', F_x' \).

It follows, for these same conditions, that \( \mathcal{J}' \subseteq \mathcal{J} \) and \( \mathcal{J}' \subseteq \mathcal{J} \).

**IV. Identification of Active, Semiactive, and Dormant Flip-Flops**

In this section we will expand upon the technique employed in identifying the active, semiactive, and dormant flip-flops \( i \), where \( \mathcal{J} \subseteq \mathcal{F} \cap F_x \). This decision can be made by analyzing the results of the PSS associated with the given PSIC \( \mathcal{F} \), as well as two related PSIC's referred to as extensions of \( \mathcal{F} \).

Given the PSIC \( \mathcal{F} = \{ F_0, F_1, F_x \} \), where \( i \in F_x \), then the \( v(i)=1 \) extension of \( F_0, F_1, F_x \) is the PSIC \( F_0^{(1)}, F_1^{(1)}, F_x^{(1)} \), where \( F_0^{(1)} = F_0, F_1^{(1)} = F_1 \cup \{ i \} \), and \( F_x^{(1)} = F_x - \{ i \} \). Similarly, the \( v(i)=0 \) extension of \( F_0, F_1, F_x \) is the PSIC \( F_0^{(0)}, F_1^{(0)}, F_x^{(0)} \), where \( F_0^{(0)} \cup \{ i \}, F_1^{(0)} = F_1 \), and \( F_x^{(0)} = F_x - \{ i \} \). Note that due to Theorem 1, \( G^{(0)} \supseteq G_x \) and \( G^{(1)} \supseteq G_x \), for \( e \in \{ 0, 1 \} \).

When we carry out a PSS for say a delay flip-flop \( i \), \( i \in F_x \), we must obtain one of the three following results, where \( Q \) is the present (initial) state, \( Q' \) the next state, and \( D \) the value of the input to the flip-flop.

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( D = Q' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x ) unknown next state</td>
</tr>
<tr>
<td>( x )</td>
<td>( 0 ) known next state (( \mathcal{J} ))</td>
</tr>
<tr>
<td>( x )</td>
<td>( 1 ) known next state (( \mathcal{J} ))</td>
</tr>
</tbody>
</table>

For the second and third cases, the flip-flop is either reset to \( 0 \) or set to \( 1 \); hence, its next state is known. These two cases both identify the flip-flop as being semiactive, since it is possible to select \( Q = D \); hence, the flip-flop will change state for some TSIC with respect to \( \mathcal{F} \).

If the results of the PSS produce \( D = x \), then the next state of the flip-flop is unknown. To determine the proper classification for this flip-flop, we repeat the PSS process for the \( v(i)=0 \) and \( v(i)=1 \) extensions of \( \mathcal{F} \). For each case we can classify the flip-flop according to the entries in Table I. The final classification for this flip-flop is then given by the "max" of these two classifications, where max is a commutative binary operator defined as follows.

\[
\text{max} (\mathcal{J}, \mathcal{J}') = \mathcal{J} \\
\text{max} (\mathcal{J}, \mathcal{D}) = \mathcal{J} \\
\text{max} (\mathcal{J}, \mathcal{D}) = \mathcal{J}
\]

For example, if the \( v(i)=0 \) extension produces \( D=0(\mathcal{D}) \) and the \( v(i)=1 \) extension produces \( D=x(\mathcal{J}) \), then \( i \) is classified as \( \mathcal{J} = \text{max} (\mathcal{J}, \mathcal{D}) \), i.e., \( i \) is active.

For a trigger flip-flop \( i \in F_x \), the results of the PSS will produce one of the following three results.

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x ) unknown next state</td>
</tr>
<tr>
<td>( x )</td>
<td>( 0 ) known next state (( \mathcal{J} ))</td>
</tr>
<tr>
<td>( x )</td>
<td>( 1 ) known next state (( \mathcal{J} ))</td>
</tr>
</tbody>
</table>

The unknown next state condition occurs for the case \( T=x \), where \( v(i)=x \). For this case, we can classify this flip-flop by employing the max operator to the classifications obtained from the \( v(i)=0 \) and \( v(i)=1 \) extensions of \( \mathcal{F} \).

For an \( R-S \) flip-flop \( i \in F_x \), the results of the PSS will produce one of the following five results, where we assume that \( R \cdot S = 0 \) and \( \tilde{I} \in \{ 0, x \} \).

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( R )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( 0 )</td>
<td>( x ) unknown next state</td>
</tr>
<tr>
<td>( x )</td>
<td>( x )</td>
<td>( 1 ) set (( \mathcal{J} ))</td>
</tr>
<tr>
<td>( x )</td>
<td>( 1 )</td>
<td>( 1 ) reset (( \mathcal{J} )) known next state</td>
</tr>
<tr>
<td>( x )</td>
<td>( 0 )</td>
<td>( 0 ) hold (( \mathcal{J} ))</td>
</tr>
</tbody>
</table>

The second result corresponds to \( xx0 \) and \( xxx \). For these three unknown next state cases, the classification of flip-flop \( i \) is again determined by employing the \( v(i)=0 \) and \( v(i)=1 \) extensions of \( \mathcal{F} \).

1) For the case

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( R )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( 0 )</td>
<td>( x )</td>
</tr>
</tbody>
</table>
only the \( v(i)=0 \) extension need be simulated, since the \( v(i)=1 \) extension has a classification of \( \mathcal{S} \). For the \( v(i)=0 \) extension, \( i \) is classified as either \( \mathcal{I} \), \( \mathcal{S} \), or \( \mathcal{D} \) according to whether \( S \) equals \( x \), 1, or 0, respectively.

2) For the case

\[
\begin{array}{ccc}
Q & R & S \\
X & X & 0 \\
\end{array}
\]

only the \( v(i)=1 \) extension need be simulated, since the \( v(i)=0 \) extension has a classification of \( \mathcal{D} \). For the \( v(i)=1 \) extension, \( i \) is classified as either \( \mathcal{I} \), \( \mathcal{S} \), or \( \mathcal{D} \) according to whether \( R \) equals \( x \), 1, or 0, respectively.

3) For the case

\[
\begin{array}{ccc}
Q & R & S \\
X & X & X \\
\end{array}
\]

both the \( v(i)=0 \) and \( v(i)=1 \) extensions of \( \mathcal{F} \) may need be simulated in order to properly classify \( i \). Both extensions need not always be simulated, since for either extension we classify \( i \) as \( \mathcal{S} \), then the final classification of \( i \) is also \( \mathcal{S} \) since \( \max(\mathcal{S}, \xi) = \mathcal{S} \) for \( \xi \in \mathcal{I}, \mathcal{S}, \mathcal{D} \).

In the Appendix we describe the simulator used to carry out the partially specified simulations used in this partitioning algorithm.

V. An Application

Given the logic description of a digital circuit, it may be desirable to have the facilities to simulate the operation of this circuit on a digital computer \([12]–[18]\). Some applications of such a simulator are

1) verification of the design prior to hardware implementation,
2) analysis of circuit operation as a function of delays, loading, and hazards,
3) generation and verification of component fault-testing sequences.

One common type of simulation system is a compiled driven one. Here the description of the logic is first compiled into machine code and then simulated. There are a number of advantages to employing such a technique. For example, by employing fault injection \([12]\), the operation of a machine under component failure can be easily simulated. Also, many different test cases can be simulated simultaneously \([17]–[18]\). This technique greatly improves the efficiency of the system. One disadvantage of this technique is that usually the entire code is evaluated each simulated clock period. We can employ the functional partitioning algorithm to partially alleviate this source of inefficiency.

We can intuitively motivate the applicability of this procedure as follows. At any given instant of time, only a small portion of the logic of a machine is usually being activated \([5], [12]\). That logic that is active is generally specified by the value of various control and timing signals. Hence, to achieve a highly efficient logic simulator one could partition the logic such that for a given set of control and timing signals, only that logic that can possibly affect the next state of the machine when these signals are active be simulated.

We will now illustrate this concept on the circuit shown in Fig. 6. Assume we wish to simulate this circuit for the following conditions.

1) \( v(3)=v(4)=v(5)=1 \).
2) A set of initial state conditions for the as yet unspecified flip-flops, and a set of input sequences for the primary input lines. For each case we wish to determine the resulting sequence of flip-flop and primary output states.

Since flip-flops 3, 4, and 5 are not allowed to change state, we let \( \mathcal{F} = \{6, 7, 8, 9\} \). Using the PSIC specified by \( F_0 = \phi, F_1 = \{3, 4, 5\} \), and \( F_2 = \{1, 2, 6, 7, 8, 9, 21\} \), we can carry out a partially specified simulation. From the previous discussion (Section III) of this circuit for this \( (\mathcal{F}, \mathcal{F}) \), it is seen that if \( v(6)=0 \), no simulation is required since the state of the circuit will not change. If \( v(6)=1 \), then the only signal values that need be specified are for lines 1 and 2. All other signal values will not cause any flip-flops to change state. Finally, only gates 10 and 17 need be simulated. The compiled logic for this case is given below, where we assume that \( v(i) \) is stored in word \( Q+i \).

\[
\begin{align*}
\text{CAL} & \quad Q+1 \\
\text{ORA} & \quad Q+2 \\
\text{SLW} & \quad Q+10 \\
\text{ANA} & \quad Q+12 \\
\text{ANA} & \quad Q+5 \\
\text{SLW} & \quad Q+17 \\
\text{CAL} & \quad Q+16 \\
\text{BNZ} & \quad *+6 \quad \text{(branch on nonzero)} \\
\text{CAL} & \quad Q+17 \\
\text{BZE} & \quad \text{EXIT} \quad \text{(branch on zero)} \\
\text{CAL} & \quad \text{ZERO} \\
\text{SLW} & \quad Q+6 \quad \text{(store 0 in } F/F \text{ 6)} \\
\text{TRA} & \quad \text{EXIT} \\
\text{SLW} & \quad Q+6 \quad \text{(store 1 in } F/F \text{ 6)} \\
\text{EXIT} & \quad \cdots \\
\text{CAL} & \quad Q+6 \\
\text{BZE} & \quad \text{EXIT} \\
\text{CAL} & \quad Q+1 \\
\text{ORA} & \quad Q+2 \\
\text{BZE} & \quad \text{EXIT} \\
\text{COM} & \quad \text{(complement)} \\
\text{SLW} & \quad Q+6 \\
\text{EXIT} & \quad \cdots
\end{align*}
\]

A more efficient code, which makes use of all the information known, such as \( v(12)=v(5)=1 \), would be

\[
\begin{align*}
\text{CAL} & \quad Q+6 \\
\text{BZE} & \quad \text{EXIT} \\
\text{CAL} & \quad Q+1 \\
\text{ORA} & \quad Q+2 \\
\text{BZE} & \quad \text{EXIT} \\
\text{COM} & \quad \text{(complement)} \\
\text{SLW} & \quad Q+6 \\
\text{EXIT} & \quad \cdots
\end{align*}
\]

It is seen that the code required to simulate the logic for the case being considered is a small subset of the code required to simulate all the logic shown in Fig. 6. Note that this simulation technique is effective even when input sequences are applied, as long as the state of the machine at each clock time is a TSIC with respect to \( \mathcal{F} \). Note that if \( i \in \mathcal{F} \cap (F_0 \cup F_1) \), then flip-flop \( i \) may change state \( (i \in \mathcal{I} \cup \mathcal{S}) \).
and hence the sets $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, and $\mathcal{D}$ may be modified. During the actual simulation process, the semiactive logic is handled separate from the active logic. For example, after determining the new state for each active flip-flop, one must then go through the list of semiactive flip-flops and determine their new states. These new states can be directly determined from the results of the PSS.

For the active logic, much simulation time can be saved by using the concept of “instruction bypassing” [12]. For example, consider an active delay flip-flop whose 0 extension with respect to $\mathcal{F}$ is classified as $\mathcal{D}$. During the simulation of this circuit, once this flip-flop is reset to the zero state, it will remain in this state during the remainder of the simulation period. Hence, by first testing for the zero state, the program can determine when to branch over the logic associated with this gate.

In Fig. 8 we illustrate a three-stage simulation system based upon this partitioning procedure. In the first stage, the circuit is partitioned, and during the second stage the resulting subset of logic is then compiled into executable code. During the third stage, the logic is simulated.

If many test cases are to be simulated for a large circuit, the time saved in not stimulating all the logic can more than compensate for the time spent in carrying out the partitioning process.

VI. COMPUTATIONAL RESULTS

The functional partitioning algorithm, as shown in Fig. 7, has been programmed in FORTRAN IV. The example problem presented in Fig. 6 was partitioned in less than 0.5 second on an IBM 360/44. In general, the time increases linearly with the size of the circuit and with the number of flip-flop elements in the set $\mathcal{F}_0 \cap \mathcal{F}$, since it is these elements that require the evaluation of the 0 and 1 extensions of $\mathcal{F}$.

APPENDIX

Event-Directed Simulation

In this section we will describe one of the simulators used in the partitioning system. In order to make the simulation process as efficient as possible, especially for evaluating the 0 and 1 extensions of $\mathcal{F}$, we have employed the concept of event-directed simulation [8], [9]. The virtues of such a system are efficiency and also the ease with which cyclic logic can be handled.

An event is said to occur when in simulating a gate, it is found that its new output value (state) is different from its previous value (state). Initially, all gates are assigned the value $x$. An event-directed simulator operates according to the following simple principle. If, after simulating gate $i$ we find that it changes state, then simulate all gates in $U(i)$. This technique is efficient because we only simulate a gate when there is a possibility that it may change state, i.e., when at least one of its inputs has changed state. In general, only a small percentage of the logic of a computer will have this property during any one interval of time between two successive clock pulses. We assume all flip-flops are clocked. State devices that are not clocked, such as latches, are treated in the same way as the rest of the gates in the circuit.

We call the operation of simulating all the gates in a given push-down stack a simulation scan. Assume that in a stack $S_0$ we have the names of every gate $j$ such that during the last simulation scan there existed a gate $i$ that changed state, and $j \in U(i)$. We then simulate each gate $j$ in $S_0$, i.e., we execute a simulation scan. If gate $j$ changes state, we push $U(j)$ onto a stack $S_1$. When $S_0$ is empty, stack $S_1$ now has the properties that we assumed initially for $S_0$. We then execute another simulation scan, popping $S_1$ and pushing onto $S_0$. This procedure continues until both $S_0$ and $S_1$ are empty, i.e., $S_0 = S_1 = \emptyset$. This will not occur if we have an unstable cyclic circuit such as in inverter having the same input and output signal name. Such situations are terminated by specifying an upper limit to the number of simulation scans. The stacks $S_0$ and $S_1$ are usually referred to as $S_a$ and $S_b$, where $a, b \in \{0, 1\}$, and $a = 1 - b$. Stack $S_a$ is said to represent the current event vector, and is the stack which always gets popped. We push onto stack $S_b$.

We will now describe a complete event-directed simulation system. By complete, we mean that for any initial set of values for the flip-flops, it is capable of simulating the operation of a circuit for any sequence of inputs on the primary input lines. To begin the simulation, we place on stack $S_a$ all $j \in U(i)$, where $i$ is either a flip-flop or a PI line. Specified initial values for flip-flops, gates, and PI lines are also recorded. All signal values not defined are assigned an initial value of $x$.

Beside stacks $S_0$ and $S_1$, stacks $S$ and $S'$ are also employed. Stack $S$ contains the name of all flip-flops that will have to be simulated to determine whether or not they will change state. Since the flip-flops are clocked, we do not simulate the flip-flops until the circuit has reached steady state, i.e., $S_0$ and $S_1$ are empty. Stack $S'$ is used to store the name of all flip-flops that have changed state.

The simulation algorithm is shown in Fig. 9. Only one output command is shown. For studying races and gate state transitions, more output commands would be required.

This simulation procedure can be easily modified to handle the case where different gates have different amount of delay.

To modify this algorithm for use in carrying out the PSS required in the partitioning algorithms, we just exit when label $\Box$ is reached. For example, to carry out the $v(8) = 1$
extension of the circuit shown in Fig. 6, we first set \( v(8) = 1 \) and push \( U(8) = \{11, 19\} \) on stack \( S_a \). We then carry out a simulation scan. We assume that the present value of each line corresponds to the results obtained from the PSS starting with the given PSIC. We obtain \( v'(19) = 1 \neq v(19) = x \) and \( v'(11) = 1 \neq v(11) = x \); hence, we push 20 and 18 onto stack \( S_a \). During the next simulation scan we obtain \( v'(20) = 0 \neq v(20) = x \) and \( v'(18) = v(18) = 1 \). This concludes the simulation process. In general, only a small percentage of the entire circuit is simulated in determining the next state of the computer.

**ACKNOWLEDGMENT**

The author would like to thank H. Huang and R. Klement for programming the partitioning algorithm.

**REFERENCES**


