NASA CR-172,159

NASA Contractor Report 172159

NASA-CR-172159
19840002697

Feasibility Study For A Generalized Gate Logic Software Simulator

John G. McGough
Flight Systems Division
Bendix Corporation

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NF02035

TABLE OF CONTENTS

N84-10765#

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (CONT'D)

LIST OF TABLES

FEASIBILITY STUDY FOR A GENERALIZED GATE LOGIC SOFTWARE SIMULATOR

1.0 SUMMARY AND CONCLUSIONS

1.1 SUMMARY

Past experience has shown that commercial gate logic software simulators are several orders of magnitude too slow to perform practicable fault experiments which require simulation of digital hardware and software. In 1979 Bendix developed a very high speed gate-level simulation of the Bendix BDX-930 digital computer cpu (hereafter referred to as "BGLOSS"). The simulation was used to perform fault experiments to determine fault latency and to validate coverage of self-test programs. The success of BGLOSS in these programs resulted in a follow-on contract to determine the feasibility of developing a generalized version of BGLOSS (hereafter referred to as "GGLOSS") which would retain the high speed feature of BGLOSS and, in addition, be applicable to a wide variety of digital circuits and computers.

SECTION SUMMARY

Section 2.

The objectives and requirements of BGLOSS are reviewed and extended to GGLOSS.

Objectives of BGLOSS

- o To conduct failure modes and effects analyses
- o To assist in the design and validation of self-test
- o To obtain fault latency data such as was obtained in (refs.1,2)

Requirements of BGLOSS

- o Capable of simulating software
- o High speed
- o Capable of simulating multiple cpu's operating concurrently

Section 3.

A hypothetical network is defined which represents the prototype of all networks to be simulated. The network features a fictitious clock which, effectively, discretizes the time scale. The User constructs a prototype network by

- o defining the gate operations of the network
- o selecting a fictitious clock period
- o designating which branches are clocked

5

It is shown that this procedure produces a network of combinational circuits whose inputs are clocked. GGLOSS then evaluates each combinational circuit once in every clock cycle. It is shown that the clocked combinational circuits can be evaluated in any order but the gates within each combinational circuit must be evaluated in a definite order. GGLOSS computes the correct order.

Other key design issues are examined, including

- o 2-valued versus multi-valued logic
- o network initialization
- o gate operations and alternate network representations
- o parallel versus serial mode simulation
- o fault modelling
- o extension to multiprocessor systems

It is concluded that, to obtain high speed, GGLOSS should employ

- o 2-valued logic
- o parallel mode simulation
- o a low-level programming language for gate evaluations

This section concludes with a discussion of simulation speed and efficiency. It is shown that BGLOSS achieved an effective speed of 2.86×10^{6} gates/sec as compared with 1000 gates/sec for commercially available simulators.

Section 4.

The salient characteristics of BGLOSS are reviewed for possible incorporation into GGLOSS.

Characteristics of BGLOSS

- o Gate logic software simulator
- o Programmed in Bliss
- o Vax 11/780 host computer
- o parallel mode simulation, exclusively
- o fixed order of node evaluations
- o 2-valued logic, exclusively
- o Requires User-initialization of the network
- o Stuck-at faults, exclusively
- o Collapses faults
- o Capable of simulating software
- o 2.86×10^{6} gates/sec

6

Section 5.

After a review of BGLOSS an overview of GGLOSS is given.

Proposed Characteristics of GGLOSS

    o gate logic software simulator
    o hosted on Vax 11/780
    o parallel mode simulation
    o serial mode simulation, optional
    o fixed order of node evaluations
    o 2-valued logic, exclusively
    o requires User-initialization of the network
    o stuck-at faults, exclusively
    o collapses faults

GGLOSS will be implemented in such a way as to make maximum use of an existing computer-aided circuit design facility. In particular, the method of defining gate-level circuits, peculiar to the existing database, will be retained. GGLOSS will be designed to interface with existing netlist and partslist formats. Simulation specifications and data not normally on file in the database, such as failure rates and input/output options, will be input via the menu.

Appendix A

A brief discussion of an algorithm for computing the proper order of gate evaluations is given.

Appendix B

Techniques of modelling basic digital devices, such as flip flops, tristate busses and Rom memory, are described.

Appendix C

To illustrate the methodology of GGLOSS a simulation of a hypothetical Computer Control Unit is described.

1.2 Conclusions

    o A high speed generalized gate logic software simulator is feasible. Speeds of the order of 2×10^{6} gates/sec can be achieved.

    o The fidelity of fault simulation is at least as good as that of an event-driven commercial simulator.

    o The simulator can, albeit at reduced speed, simulate circuit timing.

o The simulator can be designed to utilize the existing resources of a computer-aided circuit design facility. It will not be necessary to reprogram existing gate-level circuits which are on file in the database.

o A reasonable degree of transportability can be obtained by programming the control and executive functions of GGLOSS in Fortran.

o For high speed it is necessary to use a low-level programming language for gate evaluations. The recommended language is Bliss.

2.0 Introduction

In the Fall of 1979 Bendix was awarded a contract by NASA Langley Research Center to perform a fault simulation study to determine fault latency in a digital avionics processor(ref.1). Prior to the award, Bendix had developed a gate logic software simulator(BGLOSS) for its BDX-930 digital computer. The study provided the opportunity to not only establish fault latency statistics but to test BGLOSS in a variety of scenarios, not the least of which included the simulation of software programs and their interaction with hardware faults. Prior simulation experience lead to the conclusion that, next to reasonable accuracy, simulation speed was the most important characteristic of a simulator. Current, commercially available varieties are too slow for the types of fault experiments envisioned, being of the order of 1000 gates/sec of host computer cpu time. The impact of such speeds can readily be appreciated by considering an application in which it is desired to determine the detectability of a fault by a self-test program consisting, typically, of 1000 assembly language instructions. In the BDX-930, an assembly language instruction requires, on the average, four passes through the cpu, which consists of 5000 equivalent gates. Thus, a single fault, together with a complete execution of self-test, requires the simulation of 20 million gates. At 1000 gates/sec it would require 5.55 hours of cpu time! Subsequently, after thousands of simulated faults, the speed of BGLOSS was established at 2.86 million gates/sec on a Vax 11/780 host computer. As a consequence of the success of BGLOSS in the context of the Fault Latency Study, NASA Langley Research Center awarded Bendix a follow-on contract to determine the feasibility of developing a generalized gate logic software simulator (GGLOSS) based upon the BGLOSS model.

2.1 Objectives of BGLOSS

BGLOSS was designed to simulate faults in large scale digital systems for the purpose of

* conducting failure modes and effects analyses
* designing and validating self-test programs
* obtaining fault latency statistics such as were obtained in (refs. 1, 2).

2.2 BGLOSS Requirements

Based on the above objectives and intended applications a set of minimal design requirements was established for the planned simulator:

A) The simulator must be capable of simulating software. This was a basic requirement since the objectives included the design and validation of self-test and the evaluation of fault latency when comparison-monitoring is the method of fault detection.

B) The simulator must yield results in a timely manner. The simulation of self-test and flight control applications programs requires many passes through the cpu. Considering the quantity of faults that were to be simulated it was the judgement of the BGLOSS design team that the simulation time, on whatever computer BGLOSS was hosted on, should not exceed 25000 times real time. Assuming 5000 gates in the cpu, at a clock cycle of 250 nanoseconds, this would be equivalent to simulating 801,753 gates/second (In any event, BGLOSS , simulated on a Vax 11/780, did not exceed 7000 times real time, which was equivalent to 2.86x10**6 gates/second).

C) The simulator must be capable of simulating multiple cpu's, with different software programs, concurrently. Because many of the envisioned simulation experiments involved redundant channels of a flight control system it was desired that the simulation should be capable of modelling the concurrent operation of synchronous and asychronous channels, i.e., processors which, effectively, execute different software programs concurrently.

2.3 Objectives of GGLOSS

The eventual success of BGLOSS was an indication that a more general simulator could be designed which met the same design requirements. In summary, these are:

1) The simulator must be capable of simulating software
2) The simulator must yield results in a timely manner
3) The simulator must be capable of simulating multiple cpu's, operating sycnchronously or asynchronously with each, possibly, executing different software programs.

In concluding this Introduction, it must be emphasized that GGLOSS is intended solely as a fault simulator. In particular, it is not intended as a circuit design tool. As a consequence, the reader will observe,

subsequently, that it lacks certain features that are normally contained in state-of-the-art commercial simulators. These features and the rationale for their ommission will be described in later sections.

3.0 Simulation Techniques

During the early phase of the study it became evident that a prerequisite for understanding the issues involved in the design of BGLOSS and the rationale for selecting many of its features was a broad knowledge of gate logic simulation techniques and the key issues involved in the selection of a simulator. Thus, this Report begins with a general survey of gate logic simulation techniques and related design issues. It is hoped that the subsequent treatment of BGLOSS is more comprehensible, as a result.

3.1 The Prototype Network

Before discussing simulation techniques it is desirable to define a hypothetical network which will represent the prototype of all non-faulted networks to be simulated.

Terminology

A graph is a set, (V,E,f), where

    V = a set of elements called "vertices"
    E = a set of elements called "edges"
    f = a mapping from E to the set of unordered pairs of elements of V.

A graph is "directed" if the edges are given a direction. The vertices of a directed graph are called "nodes" and the edges, "branches". To be more precise, a directed graph is a set, (N,B,f), where

    N = a set of elements called "nodes"
    B = a set of elements called "branches"
    f = a mapping from B to NxN.

The node pair

$$f(b)=(u,v), \text{ b in B, } (u,v) \text{ in NxN}$$

directs the branch from node u to node v. Nodes u and v are the "initial" and "terminal" nodes of b, respectively.

The number of branches which terminate at a node is called the "in-degree" of the node.

A "path" is a sequence of branches b(1),b(2),....,b(n) such that there is a sequence of nodes N(0),N(1),....,N(m) with the property that the initial and terminal nodes of b(k) are N(k-1), N(k), respectively. The path is said to "join" N(0) to N(m).

The "length" of a path is the number of branches in the path.

A path is a "loop" if it joins a path to itself.

A "network" is a directed graph. In our applications the nodes will represent logic elements (e.g., AND, OR, INVERT) and the branches, transmitted signals. Signals will assume the values of logic 0 and logic 1, exclusively. Since a branch is always associated with a node we associate a fictitious node with each external input. We call such nodes "E-nodes".

A "primitive" network is a network whose nodes consist of single logic elements, e.g., AND, OR, INVERT. A node of a primitive network is called a "simple" node. A node which is not simple is called a "compound" node.

A "combinational" network is a primitive network with no loops.

A node N is called a "predecessor" of node M if there exists a path joining N to M (If the network has no loops then N and M cannot be predecessors of each other). An ordering, N(1), N(2),...,N(m), of the network nodes is called a "p-ordering" if N(k) is not the predecessor of any node N(j) where j<k. We note that, since some nodes are not related by the predecessor relation, there will generally exist many p-orderings of a network, In this connection we have

Theorem. In a network with no loops there exists at least one p-ordering of the nodes (see Appendix A).

Clocked Networks

The prototype network will consist of compound nodes with each node representing a combinational network. In order to model the network for digital simulation it is necessary to introduce a discrete time scale. For this purpose we introduce a fictitious clock. The clock generates a periodic train of pulses which are transmitted to all nodes without delay. Upon receipt of a pulse a node activates its inputs and computes a set of outputs. The outputs, however, are only transmitted to other nodes at the descending edge of the pulse. As a consequence, node inputs and outputs do not change while a clock pulse is present. A realization of a simple clocked node, using D-flip flops, is shown in Figure 1. A similar realization of a compound clocked node is shown in Figure 2.

11

In connection with a clocked node it will be assumed that

1) all gates have the same propagation delay. d;
2) each clock cycle is a multiple of d;
3) the propagation delay through the combinational network does not exceed the clock period.

As a consequence of this latter assumption, for all intents and purposes, the propagation delay of the combinational network may as well be zero.

Constructing The Prototype Network

We assume, for the present, that the network is a primitive network which may contain feedback loops. In our principal applications the network will represent a digital processor, in which case, the network is a true, clocked network. In constructing the prototype network we take the position that the User fully understands the dynamics of the constituent circuits in regard to normal and faulted performance. It is our view that a Simulator cannot be expected to model a universal circuit. It will be shown, subsequently, through numerous examples, that a network model can be constructed at many hierarchical levels, depending upon the objectives of the User. In addition, the failure modes of digital components can be exceedingly complex and, in any case, will generally be a function of the unique characteristics of the circuit. Only the knowledgeable User can supply this information.

Briefly, the procedure for constructing a prototype network is as follows:

1) Select a fictitious clock cycle. The clock cycle represents the smallest distinguishable time increment. As a consequence, the clock frequency is the maximum frequency of any network input, internal state or output. For example, if it is desired to model a flip flop and observe potential high frequency oscillations then the clock cycle should be equal to the delay of a single gate. In a true, clocked network, on the other hand, the clock cycle should not be greater than the true clock cycle.

2) Identify clocked signals. The User must designate which signals, i.e., branches, are to be clocked. Effectively, this is the equivalent of inserting a fictitious D-flip flop in the branch. It is not necessary to identify the output flip flops, as shown in Figures 1 and 2. These are superfluous, if it is assumed that an output flip flop always terminates at a clocked node. In designating clocked branches the following rules must be followed:

12

1) All external inputs are clocked.
2) All loops must contain at least one clocked branch.
3) If A and B are the initial and final branches of a path and
   if A and B are clocked and no clocked branches intervene,
   then the propagation delay between A and B must not exceed
   the clock period.

If each clocked branch is cut the result is a set of disconnected, com-
binational subnetworks. The inputs to each subnetwork are clocked and
the outputs are either inputs to a fictitious D-flip flop or network
outputs that are terminated. As a result of this construction the net-
work will consist of clocked, compound nodes of the type shown in Figure
2.

Multiple Clocks

The preceding network employed a single clock. It is desirable, either
to improve computational efficiency (see Example 6, following) or to
model certain high speed networks, to employ multiple clocks or a single
clock with multiple phases. In these arrangements the User associates a
clock, clock frequency and phase with branches. The rules governing this
designation are the same as before, i.e.,(1), (2) and (3).

As a result of this construction the network will consist of clocked,
compound nodes except that the fictitious D-flip flops are triggered by
different clocks. If we make the assumption, as we do, that all clock
cycles are multiples of some primary clock cycle with period equal to
unity, then the network is evaluated as follows:

1) Each clocked branch is cut, as before, to form combinational
   subnetworks. The inputs to each subnetwork are clocked, possibly
   by different clocks.

2) Each subnetwork is evaluated every time at least one input
   D-flip flop is triggered, irrespective of whether or not the
   input changed at this time. The triggering is activated by the
   associated fictitious clock. Thus, for example, if all inputs
   are clocked by clocks of period m and in phase with each other
   then the subnetwork will be evaluated once in every m primary
   clock cycles.

Observations

Despite its appearance as a clocked network the prototype can model a
wide variety of networks, as the following examples demonstrate:

13

Example 1.   Asynchronous Network With Unit Gate Delays

The network is modelled by a primitive clocked network in which the clock cycle is equal to the proagation delay of a single gate.

Example 2.   Asynchronous Network With Variable Delays

All gate delays are assumed to be multiples of the delay, d, of a single gate. If the gate delay is viewed as a transport lag then a gate with delay md can be modelled by inserting m-1 fictitious gates, each with delay, d, in series. The network is otherwise modelled as in Example 1.

Example 3.   Realization of an R-S Flip Flop

The circuit is shown in Figure 3. The prototype network consists of two nodes, N(0), N(1), also shown in the figure.

Example 4. A Sequential Network

It is shown in (ref.3) that every sequential network can be realized by a combinational network with unit delay feedback. A realization, using a prototype network, is shown in Figure 4.

Example 5. Simulating a True, Clocked Network

In a true, clocked network the network may experience a change of state following the descending edge of a clock pulse. One example of this is the D-flip flop, which requires that the clock return to zero before the next data input----otherwise no output change is possible. There are two approaches that can be taken when simulating a true, clocked network, depending on the desired accuracy and level of detail of the circuit model:

1) The nodes are evaluated only at the rising edge of the true clock. In this case the circuit models implicitly assume that a clock transition has occurred prior to each rising edge(see, for Example, the D-flip flop model of Figure B-7).

2) The nodes are evaluated at both the rising and descending edges of the true clock. The circuit models should be constructed to reflect differences in the responses, otherwise the extra net evaluation in each cycle is wasted.

In (1) the fictitious clock may coincide with the true clock. In (2) the fictitious clock should be twice the frequency of the true clock, e.g., the nodes should be evaluated once at the rising edge and once at the descending edge. Node outputs are transferred to the global memory after all nodes have been evaluated, irresective of whether the evaluation corresponds to a rising or descending edge.

Example 6. Use of multiple clocks to improve efficiency

Consider a true clocked network which is combinational except that it contains an R-S flip flop. While the inputs are clocked at, possibly, true clock cycles it may be desired to observe the outputs of the flip flop at higher frequencies. This could be achieved by evaluating the total network at a higher frequency but this would be computationally very inefficient since the combinational circuit inputs do not change between true clock cycles. A better approach would be to clock the inputs of the flip flop at the higher frequency. This creates an independent subnetwork which can be evaluated at high frequencies without requiring a corresponding evaluation of the total network.

3.2 SIMULATION TECHNIQUES

In this section it will be assumed that the network to be simulated is a primitive network. The results, however, can be extrapolated to networks with compound nodes in an obvious way. For illustrative purposes we consider the primitive network of Figure 5A.

There are essentially two techniques used to simulate the prototype network: the "Unit-Delay" and "Zero-Delay" simulations hereafter referred to as the "U" and "Z" simulations, respectively. These differ in the way that data is transmitted between simulated nodes. Specifically, each node accesses a dedicated, "local" memory into which is stored computed outputs.This memory can only be accessed by its associated node. When data is to be made available to other nodes it is transferred, by the simulation executive program, to a "global" memory whence it can be accessed by all nodes.Thus, at the start of each node evaluation the inputs are fetched from the global memory and the resultant, computed outputs are stored in local memory.

1) In the U simulation the contents of all local memories are transferred to the global memory after all nodes have been evaluated.

2) In the Z simulation, on the other hand, upon completion of each node evaluation the associated local memory is transferred immediately to the global set where it is used as input data in subsequent node evaluations.

3.2.1 Unit-Delay Simulation

The procedure is as follows:

1) Define an arbitrary ordering of the nodes.
2) Initialize the network (which is necessary because of the use of 2-valued logic).

15

Then, at each clock pulse,

3) Evaluate every node in the selected order.
4) Transfer the computed output to the global set from whence it can be accessed by other nodes.
5) Repeat (3),(4) at the next clock cycle.

Observations

1) The U simulation evaluates all nodes in every clock cycle. Thus, if the network consisted of n nodes(excluding E-nodes) it could require $n**2$ node evaluations to propagate a signal through the network.

2) The U simulation gives correct node outputs for any ordering of node evaluations.

3) The U simulation is recommended when

   a) it is desired to simulate the effects of propagation delays, as in a feedback circuit; or
   b) it is desired to observe the response of each node in each clock cycle, as in a true, clocked network; or
   c) it is inconvenient to determine a p-ordering of the nodes.

The disadvantages of the U simulation technique are:

1) It requires a large number of node evaluations to propagate a signal through the network. This is wasteful if the propagation delays are small relative to the time between successive input events. For example, consider a combinational network of n nodes within a higher-level node, as in a true , clocked network. In a good design the propagation delay of the circuit is never larger than the clock cycle and, in most cases, very much smaller. Consequently, the circuit is always stable before the onset of the next clock pulse. In this scenario the U simulation would require $n**2$ node evaluations to propagate each input event when actually n would suffice. Moreover, most of the $n**2$ evaluations result in no change in the circuit elements.

2) Even when it is desired to observe the outputs of each node in each cycle, in many applications only a subset of nodes see changing inputs or outputs and, hence, require evaluation in each clock cycle. The U simulation, however, evaluates all nodes irrespective of whether or not the evaluation yields different results.

16

3.2.2 Event-Driven Simulation

To overcome this last disadvantage a variation of the U simulation can be used: the Event-Driven simulation. In this approach only those nodes are evaluated whose inputs, outputs or internal state have changed from those of the preceding clock cycle. This is the most commonly employed simulation technique. Unfortunately, it requires considerable software overhead to maintain and this frequently offsets any advantage gained in reducing the number of node evaluations. In addition, it does not lend itself to parallel simulation (See Section 3.3.4).

3.2.3 Zero-Delay Simulation

This technique is used to overcome the former disadvantage of the U simulation. The procedure is as follows:

1) External input branches are clocked.

2) A p-ordering of node evaluation is selected, e.g., N(0),N(1), N(2),....,N(n-1).

3) The network is initialized.

   Then, at the kth clock pulse, k=0,1,2,...., DO

4) Evaluate N(k(modn)) and transfer its local outputs to the global set.

Observations

1) The order of node evaluations is critical: the ordering must be a p-ordering.

2) In a combinational network with n nodes(excluding E-nodes) the Z simulation can propagate a signal through the network after, at most, n node evaluations.

3) The Z simulation is recommended when

   a) the outputs of intermediate nodes are of no interest; or
   b) the propagation delays of the network are small relative to the time between successive input events; or
   c) the circuit is a subset of a node of a true, clocked network.

4) The Z simulation employs an invariant order of node evaluations. This is a benefit in parallel mode simulation.

The disadvantages of the Z simulation are:

1) It requires a p-ordering of the nodes.

2) It assumes that the propagation delays of the network are small relative to the time between successive input events, i.e., a node, once evaluated, does not see a change in input or internal state until all nodes have been evaluated.

3) It is not easily applicable to sequential circuits unless the time delay associated with feedback elements is large relative to the propagation delay of the straight-through elements.

As a comparison of the U and Z simulation techniques consider the simulation of a simple network consisting of tandem gates, as shown in Figure 6A. The resultant node evaluations of the U and Z simulations are shown in Figures 6B,6C, respectively. From Figure 6B it can be seen that the U simulation evaluates n nodes (excluding E-nodes) in each clock cycle, for a total of $n^{**}2$ to propagate the input.The Z simulation, on the other hand, evaluates a single node in each clock cycle (see Figure 6C), requiring only n node evaluations to propagate the input to the output. In so doing, however, it assumes implicitly that the input did not change during the intervening n cycles. If the input were subject to change during these cycles then both techniques would require the same number of node evaluations.We note that an Event-Driven simulation of this network would have resulted in the same order and quantity of node evaluations as in the Z simulation.

3.2.4 Summary of Simulation Techniques

The U simulation technique is conceptually simple, easy to program, independent of the order of node evaluations and yields the states and outputs of each node in each clock cycle. It is ideal for parallel mode simulation. The technique is recommended for simulating the compound nodes in a clocked network, as in a cpu. Its main disadvantage is the large number of node evaluations required to propagate a signal through the network.

The Z simulation technique is extremely efficient computationally when the time between successive inputs is large compared with the propagation delay of the network. It is ideal for parallel mode and combinational circuit simulations.

The Event-Driven simulation is potentially computationally more efficient than the U simulation.However, the overhead may offset this advantage. In any case, the Event-Driven simulation does not lend itself to parallel mode simulation.

3.2.5 Application To GGLOSS

Both the U and Z simulation techniques will be used in GGLOSS. Briefly, the procedure will be as follows:

1) Starting with a given network the User creates a clocked, Prototype Network model consisting of compound nodes of the type shown in Figure 2.

2) The compound nodes are evaluated at each clock cycle, concurrently, using the U simulation technique.

3) The combinational networks within the compound nodes are p-ordered and evaluated using the Z simulation technique.

ER127

3.3 OTHER KEY DESIGN ISSUES

Other key design issues associated with the selection of the simulator are:

- o 2-valued versus multi-valued logic
- o Network initialization
- o Gate operations and alternate network representatons
- o Parallel versus serial mode simulation
- o Fault modelling
- o Extension to multiprocessor systems

3.3.1 2-Valued Versus Multi-Valued Logic

In the prototype network all signals assume binary values, 0 and 1. In many commercial simulators, however, signals are permitted to assume, in addition, pseudo values such as

X: unknown state
E: error state
Z: high impedance state

The rationale for this multi-valued logic is

X: Under certain conditions circuits may assume unknown states and outputs. A signal value is denoted by an "X" until its value is known without ambiguity.

There are, principally, three scenarios in which the X designation has been found useful:

1) Power-On. At power-on the circuit elements may assume random bit patterns. These are designated by X's until they stabilize to unambiguous values.

2) Entry Into Self-Test. At the initiation of a self-test program the contents of accumulators and scratchpad memory may be unknown. The constituent bits are designated by X's until they are explicitly set by the test.

3) Failure Conditions. Certain failures may result in unknown or ambiguous conditions such as when all of the transmitters of a tristate bus are in the high impedance state. The resultant bus value is designated by X's.

E: In some devices, such as flip flops, certain combinations of inputs and internal states are not permissible since they may result in ambiguous or unintended outputs. When such a condition arises the affected output is denoted by "E".

Z: This state distinguishes between high/low impedance states of a transfer gate and the binary values of logic signals.

It will be shown, subsequently, that the use of multi-valued logic greatly reduces the real time efficiency of parallel mode operations. However, the issue , at present, is whether or not multi-valued logic is necessary.

The Unknown State, X

GGLOSS is not intended as a circuit design tool: it will always be assumed that the non-faulted circuit is well-designed. As a consequence, a standard input sequence will always result in a correct and unambiguous state, independently of the initial state. In a clock synchronized cpu, for example, the circuits stabilize to unambiguous values after power-on. To obtain the correct initial conditions it is only necessary to simulate the power-on conditions for a random selection of initial states of the network. In the fault scenarios it will always be assumed that the network has stabilized to an unambiguous state prior to the injection of a fault. It will be shown, subsequently, that fault models are, themselves, logic networks which merely replace the non-failed network. As a consequence, unless the fault model is, itself, ambiguous a stuck-at fault cannot introduce an ambiguous signal in the network. The absence of the X designation requires that memory elements be initialized to known values. If this is inconvenient or impracticable random bit patterns can be selected. Upon entry into self-test or any program, for that matter, it is good design procedure to initialize all pertinent memory elements. The X state alerts the User to non-initialized memory elements which may influence the outcome of the test. In summary, the unknown state provides the following benefits:

1) It relieves the User of the burden of determining initial network states and outputs.
2) It flags non-initialized states whose values may influence the outcome of a test procedure.
3) It can be used to flag unknown or ambiguous signal values due to failures. In the context of a fault simulator it does not appear to be unreasonable to expect the User to provide the initial conditions for the network especially since these conditions will have a unique effect on failure modes. Replacing the initial states and outputs by X's defeats the purpose of the simulation which is, after all, to identify failure modes and effects. The case of unknown or ambiguous effects of failures is a different matter. Here it may not be practicable to ascertain the precise effects of a failure. However, even in this context the use of the X state merely alerts the User to this uncertainty. In any case the detectability of the fault cannot be determined until the uncertainty is removed.

The Error State, E

It will always be assumed that the prototype network is defined at the level of detail necessary to unambiguously model the network's response to all combinations of inputs and internal states. Admittedly, this imposes a heavy burden on the user, requiring a detailed knowledge of the fault modes of all devices in the circuit including those represented at the functional-level. The use of "E" values alerts the user to an error condition without detailing the resultant response. However, since the purpose of GGLOSS is fault simulation and the determination of detection coverage the use of "E" values gives very little information regarding the subsequent detection of this condition and, consequently, should be eliminated from the simulation.

The High Impedance State, Z

It will be demonstrated in Appendix B that a special state to denote high impedance is unnecessary. In general, high/low impedance devices can be modelled by gates with the high/low impedance states replaced by binary valus of 0 and 1.

3.3.2 Network Initialization

In a well-designed network means are provided to stabilize the internal states, usually at power-on. By simulating this condition the proper network initialization can always be obtained and stored for future reference. Proper network initialization can always be obtained by simulating a sample program of sufficient length to arrive at an unambiguous network state.

3.3.3 Gate Operations And Alternate Network Representations.

In Section 1 the prototype network was defined in terms of AND, OR and INVERT gates. In this section we will consider alternate gate representations. Figure 7 identifies the most common gates used to either simulate or implement a digital network. It is well-known (see (ref. 3), for example) that a sequential network can be realized by a combinational network with loops. As a consequence, the sufficiency of a gate set depends upon its ability to represent arbitrary combinational networks. In the prototype network the gates are assumed to be n-input gates. This, however, is merely a convenience since each n-input gate could have been defined, recursively, in terms of binary gates of the same type by making use of the associative properties of the AND and OR operations. Thus, for example, an n-input OR gate could have been represented by n-1, binary OR gates, as shown in Figure 8. As a consequence, we may say that the gate operations of the prototype, combinational network constitute a two-element(i.e.,0 and 1) Boolean Algebra with binary operations of multiplication and addition and the unary operation of complementation. The devices which perform these operations are called "AND" gates, "OR" gates and "INVERT" gates, respectively.

22

Definition 1. A Boolean function of n variables is a mapping of n-tuples of binary variables into the set {0,1}.

Every Boolean function can be realized by a combinational network consisting of AND, OR and INVERT gates, exclusively. However, this is not the only set of gates that can realize a Boolean function.

Definition 2. A set of gates (i.e., gate-types) is "complete" iff. every Boolean function can be realized by a combinational network constructed from gates of the set.

It can be shown (ref.4) that the following sets are complete:

S1) (AND, OR, INVERT)
S2) (AND, INVERT)
S3) (OR, INVERT)
S4) (NAND)
S5) (NOR)

Since S1 is a complete set the prototype network of Section 1 is complete in the sense that it can realize any Boolean function and, consequently, any sequential network. On the other hand, since { NOR } is also a complete set, a complete prototype network could be realized by employing NOR gates, exclusively.

Example. A prototype network is realized by binary gates of the set { AND, OR, INVERT }. It is desired to realize the network using binary NOR gates. The network can be transformed as follows: Let the symbol, "/", represent the NOR operation. Then

replace x' by x/x
replace x+y by (x/y)/(x/y)
replace xy by (x/x)/(y/y)

In summary, we may say that any prototype network can be realized by any one or a combination of the gate sets, S1 through S5. The only requirement is that the selected gate-set must be complete.

3.3.4 Parallel Versus Serial Mode Simulation.

In order to achieve good simulation speed and efficiency careful attention must be given to the efficiency of the logical constructs of the programming language. Typically, the host computer will contain, at the assembly language level, logical constructs of the form, AB, A+B, and A', where A and B are whole words. Because they constitute the basic logical operations of the computer these constructs are among the most efficient in the instruction repertoire. Moreover, and equally important from the standpoint of simulation, these instructions perform the same logical operation on corresponding bits of the operands. Thus, if A and B are the words

$$A = (a(1),a(2),...,a(m))$$
$$B = (b(1),b(2),....,b(m))$$

where $a(k),b(k)$ = kth bit values
and   m = number of bits in a word

then the construct , A+B, yields the result

$$C = (c(1),c(2),...,c(m))$$

where $c(k) = a(k) + b(k)$, k=1,2,...,m.

Effectively, m gate evaluations are performed, in parallel, at the cost of a single instruction. By exploiting this property the speed and efficiency of the Simulator can be increased by a factor of m, approximately. Specifically, the simulation is that of m, identical networks which differ from each other in that they have different fault sets or they are responding to different software programs. Each gate node is represented by a full, m-bit word of the host computer, with the kth bit value representing the kth network. A single pass through the network yields the responses of the m networks in their respective scenarios. This method of simulation is called "parallel mode simulation".

Hereafter, logical constructs which are

1) highly efficient in real time and
2) perform the same operation on corresponding bits of the operands will be referred to as "primitive logical constructs".

High-Level Programming Language. Typically, high-level languages such as Fortran and Pascal avoid the use of primitive constructs. In these languages the logical operands are Boolean variables(i.e.,TRUE or FALSE) so that, in effect, an entire word of the host computer is reserved for a single bit value. Thus, in these languages, to obtain the output of a binary gate for m sets of inputs requires the execution of m, logical constructs. This method of simulation (i.e.,one gate at a time) is called "serial mode simulation". Parallel and serial mode simulations are illustrated in Figure 9 for the construct, A+B.

24

Macro Language. Macro languages provide the user with the basic capabilities of assembly language while reducing some of the tediousness of assembly language programming. They do this, typically, by allowing the user to call-up strings of commonly used assembly language instructions with a single instruction(i.e., a macro instruction). Macro languages almost always retain the primitive logical constructs of assembly language and, consequently, they are ideal for parallel mode simulation.

Example. It is desired to simulate a multiprocessor system with m, identical processors, each with a different software program. The cpu is simulated in the parallel mode using a simulation technique that employs an invariant order of node evaluations. Each processor is represented by a fixed bit position in the words of the host computer, e.g., processor #k is represented by bit position #k, k=1,2,...,m. The fact that the resultant event sequences may be different for the m processors is immaterial since the order of node evaluations is invariant for any event sequence. As a consequence, a single pass through the cpu (in a clock cycle) gives the network states for the m processors.

Disadvantages of Parallel Mode Simulation

The use of parallel mode simulation imposes certain constraints on the Simulator: It limits the use of

1) high-level programming languages
2) multi-valued logic
3) Event-Driven simulation
4) and restricts modularization.

1) To fully capitalize on the benefits of parallel mode simulation the programming language must contain primitive constructs. This precludes the use of most high-level languages such as Fortran and Pascal. The use of low-level languages(e.g., assembly and macro languages)severely limits the transportability of the Simulator.

2) The use of multi-valued logic values, 0,1,X,E or Z, requires two or more bits to represent a signal value. Moreover, and more significantly, the logical operations involving these bits are not simple, logical functions of pairwise bits of the input words. This negates the advantage of primitive constructs, which operate on pairwise bits.

3) The efficient use of parallel mode simulation demands an invariant order of node evaluations. If m networks are simulated in parallel, each subject to different software, inputs or faults, it is to be expected that they will experience different event sequences. As a consequence, an event-directed simulator would eventually reach a condition that required, at the same instant of time, a different order of node evaluations for two or more of the m networks.

4) To achieve maximum simulation speed and efficiency the network should be modelled at the gate-level, since these operations corresond directly to primitive constructs of the programming language. Functional-level representations, such as registers, memory devices or IC chips, must be evaluated serially. Thus, when a gate interfaces with a functional-level device it is necessary to convert from parallel to serial mode formats or vice versa, depending upon the device. In addition, the functional-level device must be evaluated repeatedly to obtain the appropriate parallel oututs. The process is illustrated in Figure 10. Referring to the figure, the inputs to the functional-level device are in parallel mode format. The functional-level device, however, must be evaluated serially, perhaps using a truth table or some other high-level evaluation procedure. The procedure can be summarized as follows:

1) The parallel inputs are converted to serial formats, $(a(k), b(k))$, $k=1,2,...,m$.

2) The functional-level device is evaluated serially to obtain the outputs, $c(k)=f((a(k),b(k)))$, $k=1,2,...,m$.

3) The outputs are converted to parallel format, i.e., to the whole word $(c(1),c(2),...,c(m))$.

Despite the relative inefficiency of this procedure it is unreasonable and impracticable to proscribe functional-level representations. Since some functional-level representation is unavoidable it is recommended that the number of such devices be kept to a minimum and efficient conversion algorithms be used.

3.3.5 Fault Modelling

It has been demonstrated, thus far, that the prototype network has the flexibility and capability to model a wide variety of digital networks under non-faulted conditions. In this section techniques of modeling failure modes are considered. At the present time there is little or no data available regarding either the mode or frequency of failures of MSI or LSI devices. Despite this deficiency of data, failure mode and effects analyses are regularly performed for avionics and flight control systems. The conventional approach is to assume a set of failure modes for each device. These are usually restricted to faults at single pins although, occasionally, multiple faults may be considered. In most cases the failure rate of a device is assumed to be uniformly distributed over the pins or over the set of postulated failure modes. Except for special devices, faults are assumed to be static, being either S-a-0 or S-a-1. The point to be made here is that failure modes and their frequency of occurrence are necessarily conjectural and the credibility of any fault model proposed here will suffer no less from this deficiency of data than the conventional "model".

26

3.3.5.1 Proposed Fault Model

The proposed fault model is essentially the same as that used in BGLOSS. The following assumptions regarding failure modes of digital devices are assumed:

1) Every device can be represented, from the standpoint of performance and failure modes, by the manufacturer-supplied, gate-level equivalent circuit (a " gate-equivalent" circuit is a logic circuit that models the non-faulted performance of the device).

2) Every fault can be represented as either a S-a-0 or S-a-1 fault at a gate node.

3) The failure rate of the device is uniformly distributed over the gates of the equivalent circuit.

4) The failure rate of a gate is uniformly distributed over the nodes of the gate.

5) S-a-0 and S-a-1 faults are equally likely.

6) Faults remain active indefinitely.

7) A fault at an output node propagates to all nodes and devices that are physically connected to the failed node.

8) A fault at an input node does not propagate back to the driving node.

This latter assumption, while unrealistic from the standpoint of modeling shorts, provides a wider variety of failure modes than would otherwise be possible if propagation were allowed.

The method of selecting faults is implicit in the above model. Each S-a-0 and S-a-1 is assigned a probability of occurrence proportional to the prescribed failure rate. The resultant fault set is then randomly sampled with each fault weighted by its probability of occurrence. Thus, faults in devices with high failure rates will be selected more frequently than faults in devices with lower failure rates. The above procedure does not distinguish between gate-level and pin-level faults; the method automatically assigns failure rates to pins. If only pin-level faults are considered an alternative selection procedure is to assume that

"the failure rate of the device is uniformly distributed over the pins".

While this assumption violates the prescribed fault model it is consistent with the conventional method of assigning a probability of occurrence to pin faults.

27

In summary, the essential assumptions of the proposed fault model are:

o All faults can be represented by single S-a-0 and S-a-1 faults of gate nodes of the non-faulted, gate-equivalent circuit.
o The failure rate of the digital device is uniformly distributed over the gates of the gate-equivalent circuit.

Advantages of the Proposed Model.

o Faults are single, stuck-at faults.
o The model provides a simple and systematic procedure for modeling a wide variety of faults including data-dependent faults.
o The model provides a systematic procedure for associating a probability of occurrence with each fault.
o The model accommodates parallel mode simulation.

3.3.5.2 Validity of the Proposed Fault Model

The proposed fault model is based on the fundamental assumption that the failure modes of the digital device can be modelled by stuck-at faults of a gate-equivalent circuit. The fact that it may be necessary to inject multiple faults, whereas the proposed model employs only single stuck-at faults, is a relatively minor difference since the proposed model could just as easily accommodate multiple faults, as well. The major effect of introducing multiple faults is the difficulty of assigning probabilities of occurrence to the multiple fault sets.

The point at issue here is

o Does a gate-equivalent circuit, which was selected to model non-faulted performance, correctly model failure modes when subjected to stuck-at faults?
o More specifically, can every failure mode be produced by a set of stuck-at faults in the gate-equivalent circuit? Conversely, does every set of stuck-at faults produce a realizable failure mode?

Example. Consider a circuit that implements the logic, S=(A+B)(C+D)+EF. Figure 11 illustrates two possible gate-level representations. In the non-faulted case both representations are equivalent. It is apparent, however, that the failure modes of stuck-at faults are quite different in the two representations.

From this example it may be concluded that a gate-equivalent circuit does not necessarily model the failure modes of the device. The question is, does any gate-equivalent circuit do so?

28

In several recent publications (refs. 5,6) the assumption that failure modes can be modelled by stuck-at faults of a gate-equivalent circuit is challenged. The authors assert that

1) All failures cannot be modelled by stuck-at faults.
2) Stuck-at faults are satisfactory for modelling small-scale circuits but inadequate for large-scale circuits.
3) Faults should be opens and shorts, referenced to the physical layout of the circuit.

In support of this thesis the authors exhibit the gate circuit of Figure 12. The switch-like network consists of a load transistor and a set of command transistors which act like switches. By applying input patterns,e.g.,A,B,C,D,E,F, a set of conduction paths are determined between the output node and the VSS power supply node. A conduction path is activated when all of its command transistors are on; a conduction path is blocked when at least one of its command transistors is blocked. When a conduction path between output node and the VSS node is activated the output is at the VSS potential(logic state 0); when all conduction paths are blocked the output is at the VDD potential(logic state 1). The network realizes the function S=(A+B)(C+D)+EF.

Two possible gate-level representations of the function were given previously in Figure 11. Figure 13 shows a set of open and short faults of the transistor network and the corresponding stuck-at faults in the gate circuits. Of the seven faults selected, gate circuit #1 can model four as stuck-ats; gate circuit #2 can model six, provided that multiple stuck-ats are allowed. Fault #7, which shorts two paths, cannot be modeled by either gate circuit. The reason is that this fault creates a new logic circuit which realizes the function

S=(A+B+E)(C+D+F).

From this example it becomes clear why the failure modes of small-scale devices can be modelled by gate-equivalent circuits with stuck-at faults: they consist of relatively few multiple conduction paths.

Based on the above observations we conclude that::

1) The proposed model is adequate to model failure modes of SSI devices.

2) The proposed model should accommodate multiple stuck-at faults.

3) The selection of an appropriate gate-equivalent circuit, which most closely models the failure modes of the device, should be based on a failure modes analysis of the transistor circuit. Based on this analysis the user must select a realistic set of open and short faults and their associated stuck-at faults. The User should be cognizant, however, that not every combination of stuck-at faults corresponds to a realizable failure mode.

29

The above procedure will result in a gate-equivalent circuit that will model normal performance and a subset of realizable failure modes. Each failure mode not modellable by the gate-equivalent circuit will undoubtedly require a unique gate circuit to model. Thus,

4) The user must select a set of "representative" faults and determine, for each fault, a unique gate circuit to model the resultant failure mode.

In summary, the fault model will consist of

1) A gate-equivalent circuit which models normal performance and some failure modes.

2) A set of gate circuits which model the failure modes of representative faults not modellable by the gate-equivalent circuit.

3.3.5.3 Implementing the Fault Model

The Simulator must be capable of simulating faulted and non-faulted network performance in either the serial or parallel modes of simulation. When a device is represented at the functional-level it is necessarily simulated in the serial mode. Implementing fault modes associated with a functional-level device is a relatively straightforward procedure: The database contains a non-faulted device representation at the functional-level and a set of fault models (see Figure 14). When a fault is to be injected the Simulator Executive identifies 1) the fault and 2) the network,i.e.,one of the m, parallel networks to be faulted. The Simulator then evaluates the appropriate network responses and, after all parallel inputs have been serviced, converts the outputs to the appropriate parallel format.

The procedure is more delicate when the device is represented at the gate-level. Here, the concern is speed and to achieve speed it is necessary to maintain parallel mode simulation at all times. The efficient use of parallel mode simulation requires:

R1) A single gate circuit which models both faulted and non-faulted performance.

R2) Every gate must be evaluated for each of the m, parallel networks and in an invariant order.

The major advantage of the proposed fault model in this context is that it employs a single gate circuit which models both faulted and non-faulted performance. As we have seen, however, it may be necessary to employ several different circuits for this purpose. In any case it is necessary to comply with requirements R1 and R2 if efficient parallel mode simulation is to be achieved.

Example. A gate-equivalent circuit is given which models normal performance and some failure modes. It is necessary to create an additional circuit to model a special failure mode. To comply with R1 and R2 the circuits are simulated as shown in Figure 15. During each clock cycle both circuits are evaluated. The special failure mode is activated by a discrete qualifier,F. In the parallel mode F is represented by an m-bit word in the host computer. When it is desired to activate this failure mode in, say network #k, the kth bit is set to a logic 1, all other bits remaining at logic 0. The logic 1 state causes the transfer of the outputs of gate circuit #1 to the outputs of the device. Naturally the extra gates introduced for this purpose are never faulted.

Modelling Failure Modes in The Gate-Equivalent Circuit

The method of parallel mode simulation depicted in Figure 15 can be relatively costly in simulation speed and efficiency since it requires the evaluation of the gates in both the gate-equivalent and special gate circuits during each pass. In this section more efficient simulations are described.

The gate-equivalent circuit is used to model non-faulted performance and some failure modes which are the result of stuck-at faults. To model stuck-at faults each gate is expanded in a manner illustrated in Figure 16, for an AND gate. Referring to the figure

a = 1 models a S-a-1 of A
b = 1 models a S-a-1 of B
c = 0 models a S-a-0 of A or B or AB
d = 1 models a S-a-1 of AB

It is noted that stuck-at faults are modelled by selecting appropriate values of inputs, a,b,c, and d. Moreover,when a=b=d=0 and c=1 the circuit models non-faulted performance. Thus, in the parallel mode the circuit of Figure 16B can be used in all m networks, whether faulted or not. The circuit, however, requires 5 gate evaluations whereas the non-faulted gate required a single gate evaluation. If every gate where replaced by a similar circuit the simulation speed would decrease five-fold! To avoid this penalty it was found expedient, in BGLOSS, to partition the network and insert faults in one partition at a time.

Modelling Special Failure Modes

When it is neccessary to model special failure modes a special gate circuit must be included, as illustrated in Figure 15. However, the gate-equivalent circuit then consists of non-faulted gates, only. Thus, there is relatively little reduction in simulation speed. Naturally it is advisable to incorporate a single special gate circuit, at a time.

3.3.5.4 FAULT COLLAPSING

Another technique used to increase simulation speed in BGLOSS was "fault collapsing". It was recognized that not all stuck-at faults were distinguishable at a gate output. For example, a S-a-0 fault on an input node of an AND gate is indistinguishable from a S-a-0 fault on the output node. As a consequence, if two indistinguishable faults of the same gate were selected, only one fault was simulated.

3.3.6 Extension To Multiprocessor Systems

Parallel mode simulation is ideal for simulating multiprocessor systems although it may, under certain conditions, result in a decrease in simulation speed. As an illustration, consider a multiprocessor system such as SIFT (ref. 7), consisting of 8 processors. Each SIFT processor is assigned a set of computing tasks, some of which are redundantly executed, depending upon the criticality of the task. Non-critical tasks are normally assigned to a single processor. As a consequence, it may be assumed that, for the most part, the processors are executing different programs. Assuming that a 32-bit word of the host computer represents a single gate node, the bits are subdivided into 4 segments, of 8 bits. Each segment represents a different version of the multiprocessor system. A fault would then be injected into one of the 8 bits in each segment (If faults are limited to the same processor then it is only required to simulate the 7 memories of the non-faulted processors and 4 memories corresponding to the faulted processors). In this approach only 4 faults can be simulated in a single run as compared with 31 if a single processor system is simulated. This results in an 8-fold decrease, approximately, in simulation speed. An apparently attractive alternative approach would be to simulate 7, non-faulted processors and 25, faulted processors in a single run. The problem here is that the action taken by the multiprocessor system may be different for different faults. Effectively, the system operates like an event-driven simulator. As a consequence, when a fault is injected it is necessary to simulate the non-faulted processors for the duration of the run in order to determine the resultant action taken by the system. In summary, when the action taken by a multiprocessor system (e.g., dual, triplex, etc.) is dependent on the fault then it is necessary to simulate all of the non-faulted processors during a run.

3.4 SIMULATION TIMING

An important figure of merit of a simulator is the cpu time required by the host computer to simulate a network. This is particularly important when it is desired to simulate a software program in a miniprocessor. In these applications a typical miniprocessor might consist of 5k or more equivalent gates and a single instruction could require 4 or more passes through the cpu. If the entire software program is executed for a single fault, as in self-test, then it is obvious that even a modest number of faults would require a very large number of gate evaluations.

3.4.1 Simulation Speed

Definition 1.

Given a class of combinational networks, each with n gates. If the host computer requires T seconds of cpu time, on the average, to propagate a single input vector through a network then we define

"simulation speed" = n/T gates per second.

Example. In the parallel mode of operation, if

> m = number of bits per word
> and T = cpu time for a single pass
> then simulation speed = nm/T gates /sec

Thus, parallel mode operations improve simulation speed by a factor of m.

Observations

1) The figure of merit is independent of the number of gates actually evaluated by the simulator.

   Example. In a tandem combinational network of n gates a Z simulation will propagate a signal after n node evaluations whereas a U simulation would require n**2 node evaluations. If the host computer cpu time was 1 second for each then the simulation speed would be n gates/sec for each simulation.

2) Simulation speed is a function of the speed of the host computer. For our purposes all simulation speeds will be referenced to the Vax 11/780 computer(0.8 MIPS).

3) The figure of merit, when applied to either a U or Z simulation is independent of the structure of the network since each of these simulations evaluates a fixed number of gates, e. g., n**2 and n, respectively. However, when applied to the Event-driven simulator the figure of merit is a strong function of network structure. Consider, for example, the two networks of Figure 17. Each contains n gates but the figure of merit applied to network #2 will tend to favor the Event-Driven simulation over the Z simulation since the former could propagate some inputs after only a single node evaluation. In general "simulation speed" will always favor the Event-Driven over the Z simulation.

33

4) It is desirable to obtain the relative simulation speeds of the U and Z simulations. While each method evaluates a fixed number of gates the problem is that some gates have more than two inputs. Thus, the desired relationship is a function of the

1) average number of inputs/gate;
2) cpu time required for each gate;
3) average number of gates in the network.

If the gate constructs correspond to 2-input gates and if

   n = average number of gates in the network
   i = average number of inputs/gate,
then
   n(i-1) = average number of equivalent, 2-input gates
and
   Z simulation speed = n(i-1)x(U simulation speed).

3.4.2 Simulation Efficiency

Definition 2.

Let t = real time to propagate a signal through a network
     T = average cpu time required by the host computer to
         simulate a single pass through the network.

We define
     "simulation efficiency" = t/T.

This figure of merit will be applied principally to a miniprocessor cpu where t = true, clocked cycle.

The definition is independent of the number of gates in the network. This omission was intentional in order to preclude an incorrect computation of simulation efficiency based on simulation speed. For example, suppose that simulation speeds for the Z and Event-Driven simulations are known for the collection of combinational circuits within each node of a true, clocked network. If the speeds are A and B, respectively, and if n = number of gates in the network then it might be concluded that At/n, Bt/n were the respective simulation efficiencies. The problem is that these favor the Z simulation since they do not account for the fact that not all nodes are exercised in a clock cycle, e.g., the Event-Driven may only need to evaluate a reduced set of nodes in each cycle (see Example 4 for an estimate of the proportion of nodes actually evaluated in a large-scale network). Thus, when computing simulation efficiency from simulation speed it is necessary to ascertain the average number of gates exercised in a single pass through the network.

34

Example 1. In the parallel mode of operation, if

        m = number of bits per word
        T = cpu time for a single pass
and t = real time for single pass

then simulation efficiency = mt/T.

Thus, parallel mode operations improve efficiency by a factor of m.

Example 2. The simulation speed and efficiency of BGLOSS was determined as follows:

        n = 5000 equivalent gates of the BDX-930 cpu, excluding
           main memory
        (i = 4 = average number of inputs/gate)
        t = clock cycle of the BDX-930 = 250 nanosec.

Based on a large number of runs on the Vax 11/780 it was determined that

    a) Non-faulted Case

        simulation efficiency = 1/5000

        (i.e., a single pass through the cpu required $5000 \times 250/10^{6}$ = .00125 sec of host computer cpu time).

        Since there were 5000 gates in the network,

        simulation speed = 5000/.00125 = 4×10^{6} gates/sec.

    b) Faulted Case

        simulation efficiency = 1/7000.

        Thus, simulation speed = 5000/.00175 = 2.86×10^{6} gates/sec.

It is emphasized that these parameters reflect the use of parallel mode simulation with a word size of 32 bits. Moreover, the nodes were only evaluated on the rising edge of the clock pulse.

Example 3. As an indication of the relative simulation speed and efficiency of BGLOSS these parameters were estimated for an existing commercial software simulator. The simulator was event-driven and required 300 seconds of Vax 11/780 cpu time to propagate 100 input vectors through a non-faulted network consisting of 2300 equivalent gates. Thus,

simulation speed = $100 \times 2300/300$ = 767 gates/sec.

The network represented the cpu of a miniprocessor with a clock cycle of 250 nanoseconds. Thus, a single pass through the network was equivalent to 250 nanoseconds of real time. Since a single, simulated pass required 3 seconds of host computer cpu time

simulation efficiency $= 250 \times 10^{**}-9/3 = 1/12 \times 10^{**}6$

i.e., the simulator was 12 million times slower than the real processor.

When the simulator was used to simulate faults it was estimated that

simulation speed $= 430$ gates/sec

based on a network of 214 equivalent gates and 367 simulated faults. It is emphasized that, being event-driven, the simulator did not neccessarily evaluate all of the gates in the network in each pass.

Example 4. In (ref. 8) the authors describe a design for a high speed logic and fault software simulator, VOTE. The simulator employs the event-driven technique, 4-valued logic and parallel simulation wherever possible. The authors estimate that, in the non-faulted case, for a processor consisting of 50,000 equivalent gates

1) 1500 gates are evaluated, on the average (3% of total), in a single pass using the event-driven simulation;
2) simulation on a Vax 11/780 would require one cpu second to evaluate 7000 gates.

Thus, if the prediction is realized, the effective speed would be

simulation speed $= 7000 \times 50000/1500 = 2.33 \times 10^{**}5$ gates/sec.

In the faulted case simulation speed is estimated to be 8 times slower. Of particular interest, in this example, is the estimate that, in a large network, only 3% of the gates require evaluation in a single pass.

ER128

4.0 BGLOSS CHARACTERISTICS AND APPROACH

4.1 SALIENT CHARACTERISTICS

Based on these discussions of Section 3 we are now in a position to enumerate the characteristics and features of BGLOSS. These are:

o Gate logic software simulator
o Programmed in BLISS
o Vax 11/780 host computer
o Parallel mode simulation, exclusively
o Fixed order of node evaluations
o 2-valued logic, exclusively
o Requires User-initialization of the network
o Stuck-at faults, exclusively
o Collapses faults
o Capable of simulating software as well as hardware
o Limited input capability
o Limited output capability
o 2.8 million gates/sec

4.2 RATIONALE

Based on the discussions of Section 3 the rationale for selecting the features and techniques of BGLOSS can now be given:

1) Parallel Mode Simulation. For high speed, simulation via parallel simulation and efficient primitive constructs. This required, however, a fixed order of node evaluations.

2) BLISS. A very efficient macro language that contains primitive constructs.

3) 2-Valued Logic. For high speed simulation. Furthermore, all simulated faults resulted in unambiguous states.

4) User-Initialization. Eliminated the need for "unknown" logic states.

5) Stuck-At Faults. Easy to simulate, especially in the parallel mode. Moreover, stuck-at fault models appeared to be adequate.

6) U,Z Simulation Techniques. The overhead of an Event-Driven simulation was expected to be unacceptable. In addition, it was anticipated that most of the devices could be simulated by the Z simulation technique. Problems associated with the proper ordering of the network were discovered, belatedly.

7) Simulation Speed. Preliminary analysis of fault scenarios indicated that a simulation efficiency of 1/25,000 would have been acceptable. It was a pleasant surprise to achieve 1/7,000.

37

8) Fault Collapsing. For increased simulation speed and efficiency.

4.3 SIMULATION TECHNIQUES

4.3.1 Functional-Level Networks

In BGLOSS the only devices that were simulated at the functional-level were

- o Microprogram memory (54S473)
- o Microprogram control prom (54S288)
- o Macroinstruction start address prom (54S472)
- o Main memory, Rom and Ram
- o Ram memory (accumulators) of the ALU (2901A)

These devices were defined by arrays.

4.3.2 Gate-Equivalent Circuits

A gate-equivalent circuit was obtained from the manufacturer for all IC devices, including the 2901A ALU. In several cases it was expedient to modify the gate circuit either for convenience or to increase simulation speed.

Gate-Types: The simulation imposed no restrictions on gate-types. In fact, all of the common gates of Figure 4 were used somewhere in the network.

Flip Flops: As indicated in Appendix B, flip flops, even clocked flip flops, can present problems in simulation. Because of the large quantity of D-flip flops in the cpu the equivalent circuit was modified to that of Figure B-7. Observe that the feedback branch is clocked.

Busses: With one exception all bus transceivers of the cpu are unidirectional and all are tristate. For BGLOSS, it was determined that the tristate busses of the BDX-930 operate like wired AND logic and, hence, are functionally equivalent to connecting the transmitted signals to the input of an AND gate. Moreover, when all impedance levels are high the bus value is a logic 1. The gate-equivalent circuit of the bus, used in BGLOSS, is shown in Figure B-9. The only bidirectional transceivers were between the memory bus and the cpu data bus and between the memory bus and memory. A detailed description of bus simulation techniques is given in Appendix B.

38

4.3.3 The Prototype BDX-930 Network Model

The complete BDX-930 cpu circuitry is shown in Figure 18. The network was modularized by manually partitioning the circuit into subsets of components, primarily for convenience of computation. The combinational networks within each module were manually p-ordered. The modules were then evaluated once in every clock cycle. In retrospect, it is obvious that the modules should have corresponded to clocked, compound nodes. They did not: some inputs were clocked and others were not. As a result, some module boundaries separated the inputs and outputs of combinational circuits which should have been modelled with zero delays. This created two problems:

1) The non-clocked inputs, instead of propagating to a clocked node with zero delay, were delayed by at least one clock cycle, depending upon the number of intervening modules. This problem was resolved by identifying the outputs of these circuits and outputting their values immediately to the global memory. This solution required, in addition, that the modules be evaluated in a p-order.

2) In some cases two modules contained different inputs to the same combinational, zero delay circuit. As a consequence, it was not always possible to evaluate the modules in an invariant order and obtain the correct results for all combinations of inputs. This problem was resolved by reassigning these inputs to the same module.

With the above changes the Prototype Network Model had the following structure:

1) Modularized, some inputs clocked, some non-clocked.
2) Non-clocked outputs transferred, immediately, to global memory; clocked outputs transferred to global memory after all modules were evaluated.
3) Modules were evaluated in an invariant p-order to insure correct simulation of combinational, zero delay circuits cut by module boundaries.

4.3.4 Fault Models

The underlying assumptions of the BGLOSS fault model were exactly those described in Section 3.2.5.3. BGLOSS employed, essentially, two failure models:

1) Gates were faulted by the method described in Section 3.2.5.3 and as shown in Figure 13.
2) Bits in memory devices, e.g., the micromemory, were faulted to the complement of the non-faulted value.

39

In all cases the faults remained for the duration of each experiment.

BGLOSS did not model special failure modes.

4.3.5 Method of Identifying Detected Faults

As indicated in the Introduction, BGLOSS was developed primarily for the purpose of validating self-test programs and estimating fault latency in a comparison-monitored system. In these applications it was essential that the meaning of "detected fault", as assessed by BGLOSS, exactly corresponded to that of the target system. The following discussion anticipates the need for a simple, concise and general definition of "fault detection" to be used by GGLOSS. Typically, a self-test program outputs an encoded set of discretes which signifies "pass" or "fail". Usually the "fail" discrete is output at that point in the program at which the fault was detected, but this is not essential. It is almost always the case, however, that the output occurs, when it occurs at all, no later than in the time required by a non-faulted, but identical, processor to complete the entire program, assuming that the program consists of an invariant sequence of instructions. When a fault occurs there are three possible responses of self-test:

1) The fault is detected, explicitly, and the program attempts to output a "fail" discrete.
2) Self-test cannot be completed nor can the "fail" discrete be output.
3) The processor "successfully" completes self-test and outputs a "pass" discrete.

Thus, a fault is recognizable, in principle, if

1) self-test outputs a "fail" discrete or
2) self-test does not respond with either a "pass" or "fail" discrete in some period of time determined by the particular test. (This "non-response" is usually monitored by a watchdog timer).

In BGLOSS this " period of time" was the time required by the non-faulted processor to complete its self-test.

In the experiments involving comparison-monitoring certain computed variables were designated as "monitored variables" and stored in memory after each computation. Any difference, at the completion of a frame of computations, between corresponding monitored variables in the faulted and non-faulted processors signified a "detected failure". This approach, although a good approximation, was not a correct model of comparison-monitoring. In a real system the monitored variables are exchanged, usually over data links, and the processor which detects a miscompare takes some action as a result. The BGLOSS approach had the merit of not requiring an explicit software program for the monitoring, or a simulation of the data links and the hardware associated with the output discretes.

Extrapolating from the above scenarios, we propose, tentatively, the following definition of "fault detection" for use in GGLOSS:

A "detected fault" is any fault which, during a predetermined interval of time, results in a difference between the corresponding outputs of certain designated components of the faulted and non-faulted processors. As a consequence, the User of GGLOSS need only identify the monitored outputs and the associated time intervals.

4.4 PREPROCESSOR/POSTPROCESSOR CHARACTERISTICS

Preprocessor Tasks

- o Computed probability of stuck-at faults from device failure rates
- o Queried User for number of faults to be simulated
- o Randomly selected faults

Postprocessor Tasks

- o Determined which faults were detected and the clock cycle in which detected
- o Computed cumulative fault detection statistics
- o Outputted results

BGLOSS did not compute p-orderings. This was done manually.

4.4.1 Statistical Methods

The techniques of statistical analysis are described, in detail, in (ref.1). It is sufficient to note that the method is based on random, stratified sampling techniques.

4.5 SIMULATION TIMING

The simulation speed and efficiency of BGLOSS was given in Example 2 of Section 3.4.1, i.e.,

simulation speed = 2.86×10^6 gates/sec

simulation efficiency = 1/7000

using a Vax 11/780 host computer.

41

4.6 PROBLEM AREAS

The problem areas of BGLOSS have been described throughout this report. In summary, the problem areas were:

1) Incorrect designation of compound nodes. Node boundaries, in some cases, cut zero delay circuits. Consequently, some node values had to be passed to global memory immediately while others had to be delayed until all nodes were evaluated. Distinguishing between these outputs was a tedious and time consuming exercise. In addition, these mixed nodes had to be evaluated in a definite order.

2) The network had to be p-ordered, manually.

3) The simulation was entirely Bliss-coded. Changes in a circuit had to be reprogrammed in Bliss.

4) Initially, inefficient parallel/serial conversion algorithms were used.

5) Considerable effort was expended in determining prototype models of flip flops, busses, etc.

4.7 BGLOSS IN RETROSPECT

Despite initial problems BGLOSS was a resounding success from the standpoint of simulation speed, efficiency and fidelity of modelling stuck-at faults. More than 10,000 faults were eventually simulated and every undetected fault (i.e., by Self-Test) was analyzed to determine why it was not detected. In only one case was it determined that the simulation was in error in modelling a fault.

Despite this success, however, BGLOSS was a disappointment because it could not be used to simulate a different cpu. BGLOSS was customized for the BDX-930 and the format did not permit an extrapolation to another computer.

In retrospect it is now apparent that a generalized BGLOSS could have been developed at less cost.

5.0 GGLOSS

5.1 INTRODUCTION

5.1.1 Summary Review of BGLOSS

The success of BGLOSS can be attributed to

o The use of a low-level programming language (Bliss) and parallel simulation and

o a fixed order of node evaluations.

The result was a very high speed gate logic software simulator.

5.1.2 BGLOSS's Deficiencies

BGLOSS suffered from several deficiencies:

1) It required virtually a complete reprogramming to apply the simulation to another cpu.

2) It was not easily transportable, being programmed entirely in Bliss.

3) It was not User-friendly: circuit changes could only be implemented by directly reprogramming in Bliss.

5.1.3 Conclusions

In retrospect it is now clear that, had it been a design goal, BGLOSS could have been developed with greater applicability and at less cost. It is to be hoped that the proposed GGLOSS design embodies the good features of BGLOSS and none of its deficiencies.

5.2 OVERVIEW OF GGLOSS

This section gives a brief overview of GGLOSS, its essential characteristics and the factors and considerations that influenced the design. As explained in Section 3.3.4, the high speed simulation requirement of GGLOSS imposes certain constraints on the design, notably in the use of a higher order programming language. Subject to this constraint the objectives of the GGloss design are:

o maximal transportability and adaptability, with minimal modifications, to an existing, computer-aided circuit design facility
o ease of use

To maximize transportability it is proposed to host GGLOSS on a Vax 11/780 computer; to code the control and executive functions in Fortran and to code the high speed arithmetic and logical constructs in Bliss.

To achieve adaptability and ease of use GGLOSS will be" menu-driven" and will utilize, to the greatest extent possible, existing circuit specification formats.

It is envisioned that a given network will be defined in a manner already familiar to the User, e.g., via standard partslists, netlists and component specifications at the gate-level. The menu will be reserved exclusively for selecting desired output data and procedural options available to the User. In order to simplify the implementation and reduce the cost of GGLOSS it is intended that the User will provide a database system for preparing, editing, storing and displaying hierarchical network designs. Such systems are commercially available and relatively inexpensive. This approach offers other advantages in addition to those cited above:

1) It does not require a redefinition of a previously defined network. It recognizes that a fault simulator is only one of many tools associated with the design, development and validation of digital circuits and, consequently, a network definition will most likely exist in the database long before a fault simulation is undertaken.

2) It allows the User flexibility in planning a database system independently of GGLOSS.

3) The database system can be expected to provide a far greater capability than one developed exclusively for GGLOSS. For example, it can be expected to contain a library of standard digital devices, graphics capability, computer-aided design programs such as Tegas, Spice and a User-friendly design language.

Essentially, then, the database system will provide the capability for preparing, editing, storing and displaying hierarchical networks. From the standpoint of GGLOSS, it is only required that the circuit be defined in terms of a standard partslist and netlist in database. The network definition may include the failure rates of components but this is not an essential requirement.

In summary, we may say that the general characteristics of GGLOSS are

5.2.1 General Characteristics

* High speed gate logic software simulator
* Hosted on a Vax 11/780 computer
* Control and executive functions coded in Fortran
* Arithmetical and logical constructs coded in Bliss
* Utilizes existing database system
* Menu-driven

The specific characteristics of GGLOSS have been alluded to in previous sections and are summarized here:

5.2.2 Specific Characteristics

* Conducts limited interaction with User via the Menu
* Preprocessor/postprocessor features
* Gate-level and component-level simulation
* Parallel mode of operation; serial mode, optional
* Fixed order of node evaluations
* Includes parallel/serial transformations for interfacing between gate-level and component-level modules
* 2-valued logic, exclusively
* Requires user-initialization of network
* Stuck-at faults, only
* Collapses faults
* Speed = 1 to 2 M gates/sec on Vax 11/780
* Unit-Delay simulation of clocked nodes
* Zero-Delay simulation for combinational circuits

5.2.3 GGLOSS Modes of Operation

GGLOSS will be designed to simulate any User-defined Prototype Network which is constructed according to the rules of Section 3.1. This is the only mode of GGLOSS. However, as noted in Section 3.1, a Prototype network can model a wide variety of networks. By simply designating clocked branches the User completely determines the simulation.

Example 1. It is desired to simulate a primitive combinational network. It is known, a priori, that the propagation delay through the network is small relative to the successive changes of input events. The User defines the network by specifying a Partslist and Netlist. The parts are assumed to be defined in a standard library of components. The User then designates all external input branches as "clocked branches". Apart from selecting the output data the non-faulted simulation is now completely specified. GGLOSS will determine a p-order of gate evaluations and proceed to evaluate the combinational network once in every clock cycle. The User has specified, in effect, a Z simulation.

Example 2. It is desired to simulate the same network as in Example 1. However, in this case the time between successive changes of input events is small compared with the propagation delay through the network. The User defines the network as in Example 1 but designates all branches as "clocked branches". The non-faulted simulation is now completely specified. GGLOSS proceeds to evaluate each gate in every clock cycle, transmitting the results to the global memory after all gates have been evaluated. The User has specified, in effect, a U simulation.

5.3 REQUIRED TASKS

5.3.1 User Tasks

As a minimum, the User will be required to:

* Define network
    o Supply partslist/netlist
    o Functional-level modules
    o Memory, accumulators, registers
    o Edit and add to standard circuit library
    o Fault models, if non-standard (See Section 3.3.5.3 and Figure 15, for example)

No matter which mode of operation is ultimately selected by the User it is the User's responsibility to define the appropriate prototype network.

* Define contents of program and microprogram memories.
* Supply: network initialization; start location; test vector sequence, if required.
* Supply failure rates of components.
* Supply definition of "fault detection".
* Supply fault list.
* Designate clocked branches, clock frequencies and phases.
* Designate I/O options: outputs, formats, etc.

A more detailed description of a typical User-procedure for setting-up a simulation is illustrated in Appendix C.

5.3.2 GGLOSS Tasks

As a minimum, GGLOSS is required to:

* Translate the circuit specifications to executable code.
* Debug User-defined network for completeness(e.g.,identify dead-ended pins) and continuity.
* Compute probability of faults in fault list or select faults probabilistically.
* Collapse faults.
* Partition the network for efficient fault injection.
* Supply fault models for standard gates.
* Create simulated network.
* Execute and control simulation.
* Identify detected faults and the clock cycle in which the fault was detected.
* Compute cumulative fault detection statistics.
* Output results; store, if required.
* Conduct limited interaction with User(via menu)

5.4 STRUCTURE OF GGLOSS

The proposed structure of GGLOSS is shown in Figure 19. Excluding the Bliss compiler, GGLOSS is comprised of three programs: Preprocessor, Executive and Postprocessor. In addition, GGLOSS will provide a library of primitive, Bliss-coded macros and the capability to expand the library.

5.4.1 Preprocessor Tasks

* Translate circuit specifications to executable Bliss code
* Create standard fault models for gates (Special failure models are defined in the Prototype Network Model. See, for example, Figure 15)
* Collapse faults
* Partition network for efficient fault injection and assign fault sets to partitions
* Compute probability of faults in fault list
* Compute p-orderings
* Debug network
* Conduct limited interaction with User(via menu)

5.4.2 Postprocessor Tasks

* Identify detected faults and the clock cycle in which the fault was detected
* Compute cumulative fault detection statistics
* Output results in appropriate formats

47

5.4.3 Executive Tasks

* Execute and control simulation
* Pass data to the Postprocessor

5.4.4 Library of Bliss-Coded Macros

Initially, GGLOSS will provide a library of primitive logical constructs, coded in Bliss syntax. These will include:

AND, OR, NAND, NOR, XOR, INVERT, D-Flip Flop, etc.

The names of these constructs (i.e., macros) will correspond directly to parts names in the partslist and in the library of standard components. As a consequence, the translation from a part to its Bliss-coded counterpart is relatively straightforward. The User can create higher-level, Bliss-coded macros in exactly the same way that any higher-level part is constructed, i.e., by designating a collection of previously defined parts as a new component. It is emphasized that the translation to Bliss-coded macros is completely transparent to the User. Neither a knowledge of Bliss or the role it plays in the simulation process are required of the User: The User defines the network in the User-oriented language of the database system, e.g., in SDL, then GGLOSS makes the translation.

The translation process requires standard formats for partslists, netlists and component definitions. These formats, however, can be expected to vary, somewhat, from one facility to another. As a consequence, it will be necessary to develop standard formats for use in conjunction with GGLOSS. It will be the User's responsibility to translate existing formats to these standards. This is expected to be a minor task.

5.4.5 Overview of the Simulation Process

Referring to Figure 19:

1) The User creates a partslist and netlist using the User-oriented language of the existing database system.

2) The Preprocessor conducts the interaction with the User, via the Menu, and creates a node evaluation program in Bliss syntax. The Bliss-coded macros are created as described in Section 5.4.3. The node evaluation program is then compiled, eventually to be called by the Executive Program as a Fortran subroutine. Since the nodes are coded in Bliss their execution will be at very high speed and in the parallel mode. The Preprocessor is coded in Fortran.

48

3) The Executive Program controls the execution of the simulation and passes data to the Postprocessor for statistical analysis and outputting. Outputs can be placed on disc, printed or displayed graphically. Both the Executive and Postprocessor Programs are coded in Fortran.

5.5 I/O OPTIONS

The input and output options will be selectable , via the menu, and will include the following:

* Network?
* Functional-level components?
* Fault models?
* Fault list?
* Fictitious nodes?
* Failure rates?
* Network initialization, start location?
* Test vector sequence?
* Compute number of clock cycles to completion of simulated program?
* Use fault models for standard gates?
* Compute network partition?
* Compute collapsed faults?
* Clock frequencies, phases?
* Clocked branches?
* Fault detection statistics?
* Definition of fault detection?
* Print network/subnetwork states at specified clock cycle?
* Print network partition?
* Print contents of simulated memory?
* Print undetected fault list?
* Print "time to detect" faults?
* Print indistinguishable faults(i.e., dead-ended pins)?
* Print p-orderings?
* Print number of clock cycles to completion of program?

5.6 ESTIMATED TASKS

Preprocessor Tasks

1. Create standard netlist for translation to Bliss code
2. Create standard partslist for translation to Bliss code
3. Create standard library of primitive constructs, coded in Bliss
4. Create higher-level macro generation program
5. Define menu and create translation program
6. Conduct limited interaction with User

49

7. Create fictitious clocks and identify clocked branches
8. Create standard fault models
9. Create network partition
10. Compute p-ordering of network
11. Create Bliss-coded circuit for use by the Executive Program as Fortran subroutines
12. Create program to compute probability of faults
13. Create program to select faults (using statified sampling)
14. Create fault list
15. Collapse faults
16. Debug network
17. Create table of instructions for Executive Program

Executive Tasks

1. Create program to execute and control simulation, using instructions obtained from Preprocessor
2. Pass data to Postprocessor

Postprocessor Tasks

1. Identify detected faults and the clock cycle in which the fault was detected
2. Create program to compute cumulative fault statistics
3. Create program to output to disc, printer, display

GGLOSS Trials

1. Run sample simulations

Documentation Tasks

1. User's manual
2. Simulation techniques manual

6.0 REFERENCES

1. Mc Gough,J.,Swern,F., "Measurement of Fault Latency in a Digital Avionic Mini Processor", NASA CR-3462, NASA Langley Research Center, Hampton, Va, October 1981.

2. McGough,J.,Swern,F., "Measurement of Fault Latency in a Digital Avionic Mini Processor", NASA CR-3651, NASA Langley Research Center, Hampton, Va, January 1983.

3. Wood,P.,Jr., "Switching Theory", McGraw-Hill, New York, 1968.

4. Lee,S., "Modern Switching Theory and Digital Design", Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

5. Galiay,J.,Crouzet,Y.,Vergniault,M., "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability", IEEE Trans. Computers, vol.c-29, No.6, June 1980, pp. 527-531.

6. Chiang,K.,Vranesic,Z., "Test Generation for MOS Complex Gate Networks", Proc. FTCS-12, IEEE Comp. Soc., June 1982, pp. 149-157.

7. Wensley, J., Lamport, L., Goldberg, J., et.al., "Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proc. IEEE. Vol. 66, October 1978, pp. 1240-1254.

8. Ulrich,E.,Lacey,D.,Phillips,N.,et.al., "High Speed Concurrent Fault Simulation With Vectors and Scalars", ACM Communications, June 1980, pp. 374-380.

ER129
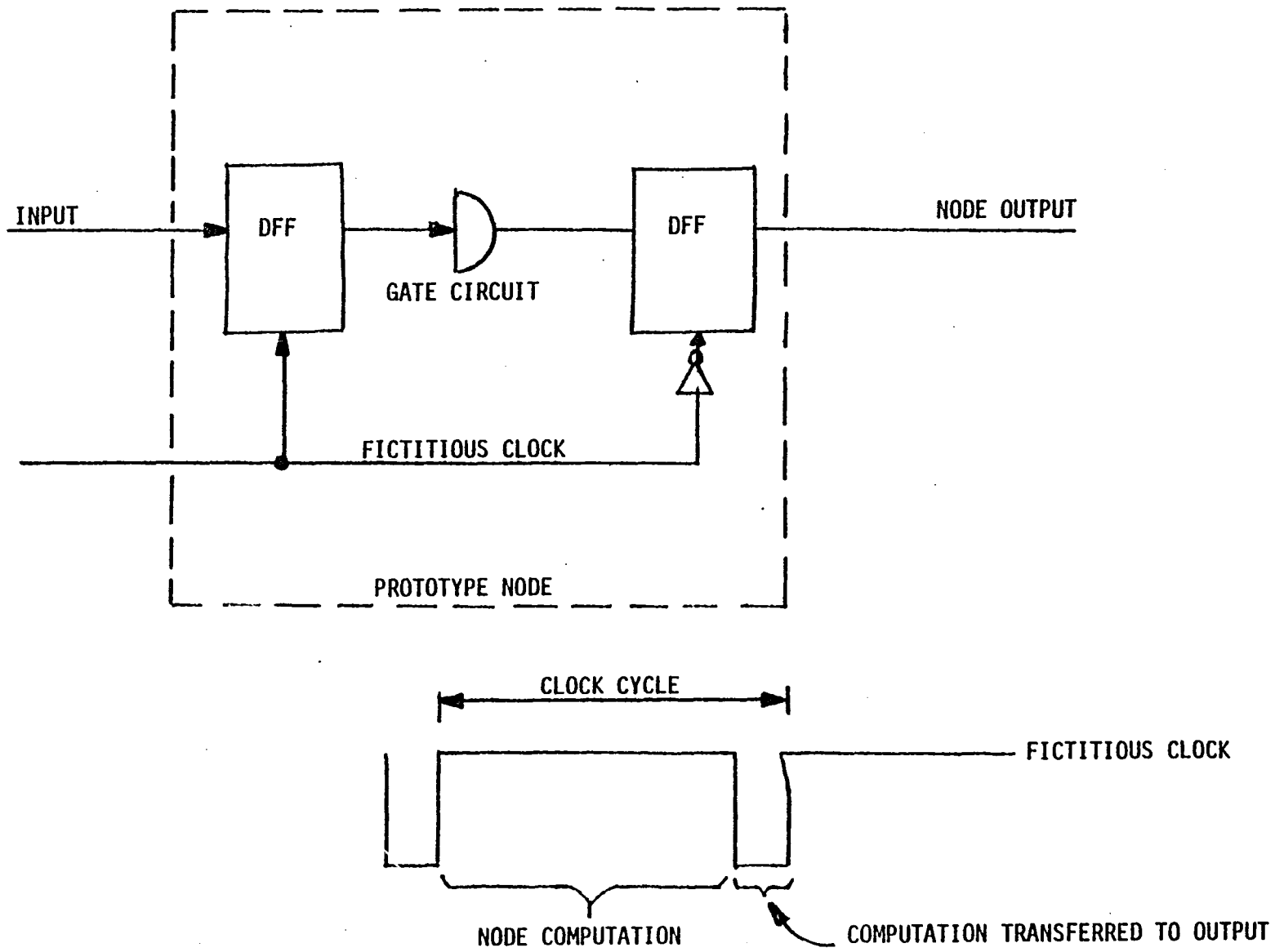
FIGURE 1   REALIZATION OF A SIMPLE CLOCKED NODE
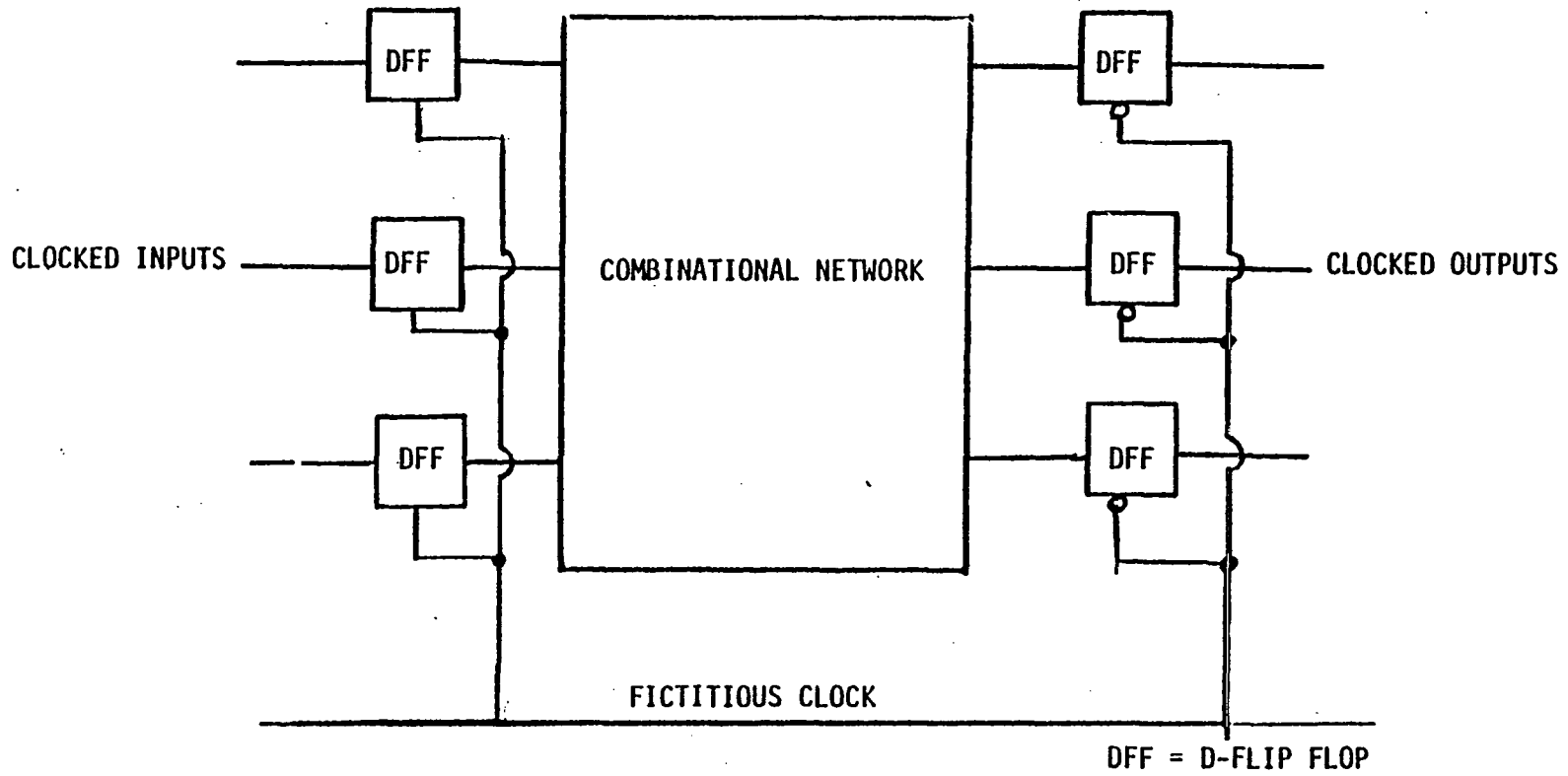
FIGURE 2   REALIZATION OF A COMPOUND, CLOCKED NODE

53

R-S FLIP FLOP

NODE (0)

= FICTITIOUS DFF

NODE (1)

REALIZATION OF R-S FLIP FLOP
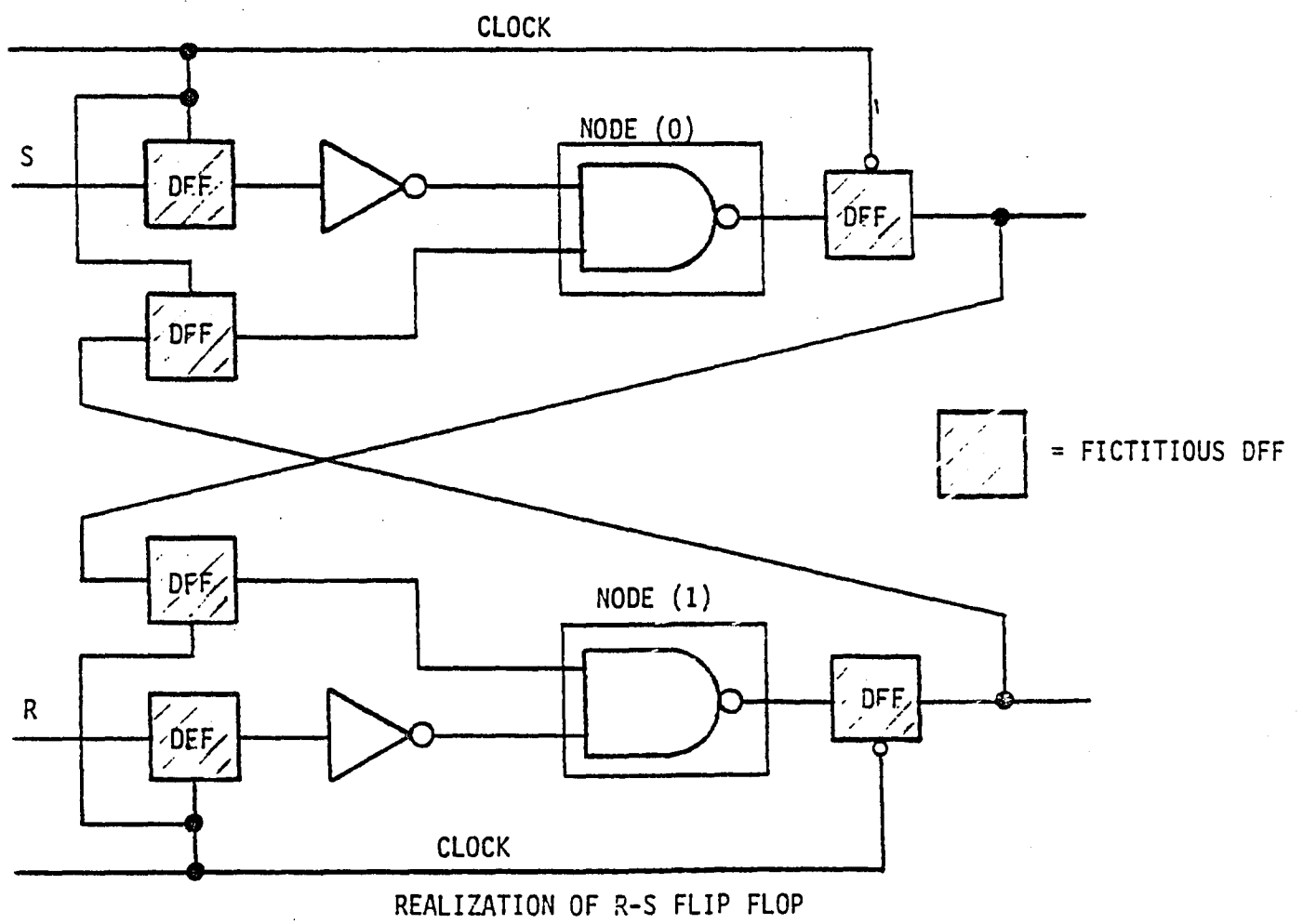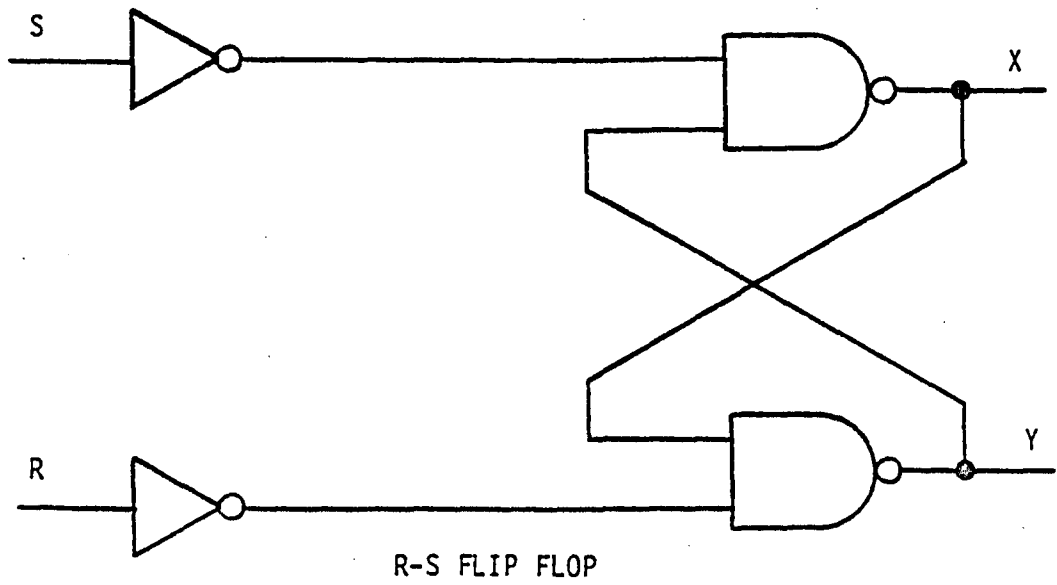
FIGURE 3   R-S FLIP FLOP
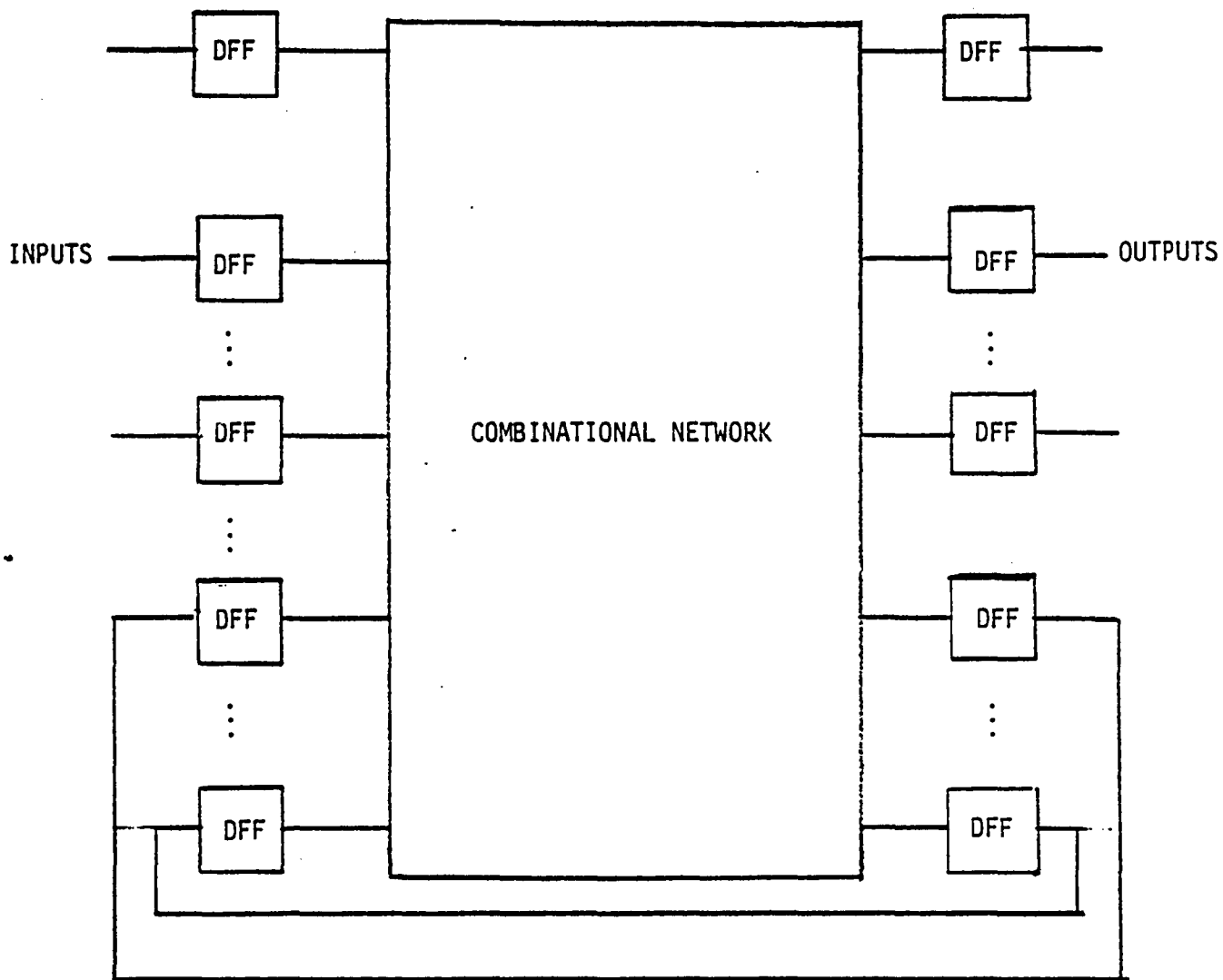
FIGURE 4   REALIZATION OF A SEQUENTIAL NETWORK
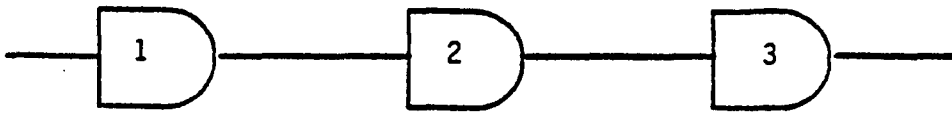
FIGURE 5A   PRIMITIVE NETWORK

FICTITIOUS CLOCK
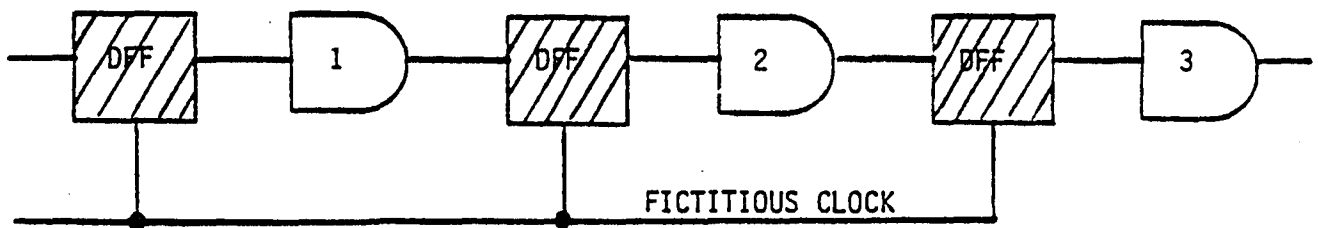
FIGURE 5B   PROTOTYPE NETWORK MODEL FOR UNIT-DELAY SIMULATION

FICTITIOUS CLOCK
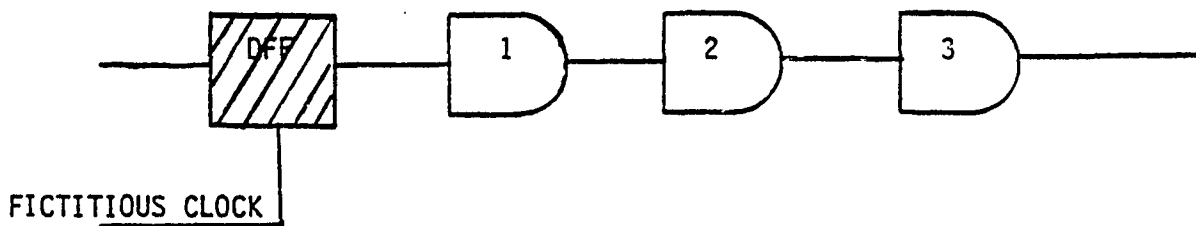
FIGURE 5C   PROTOTYPE NETWORK MODEL FOR ZERO-DELAY
SIMULATION   = FICITIOUS DFF
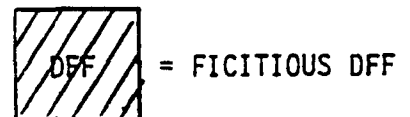
FIGURE 5   PROTOTYPE NETWORK MODELS FOR UNIT-DELAY AND ZERO-DELAY SIMULATIONS
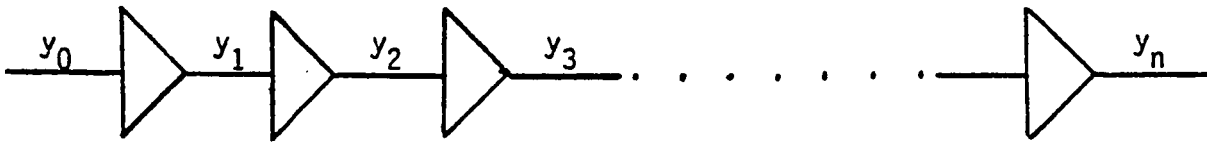
FIGURE 6A   TANDEM NETWORK

CLK	y_0	y_1	y_2	y_3		y_n
0	1	1	0	0	· · · · · · · · · · · ·	0
1	1	1	1	0		0
2	1	1	1	1		0
n-1	1	1	1	1	· · · · · · · · · · · ·	1

FIGURE 6B   OUTPUTS OF UNIT-DELAY SIMULATION

CLK	y_0	y_1	y_2	y_3		y_n
0	1	1				
1	1*	1*	1			
2	1*	1*	1*	1		
n-1	1*	1*	1*	1*		1

FIGURE 6C   OUTPUTS OF ZERO-DELAY SIMULATION

1* = NOT EVALUATED BUT ASSUMED = 1

FIGURE 6   NODE EVALUATIONS IN U AND Z SIMULATIONS

57

$$x_1 + x_2 + \ldots + x_n \qquad \text{OR}$$

$$x_1 \, x_2 \, \ldots \, x_n \qquad \text{AND}$$

$$(x_1 + x_2 + \ldots + x_n)' \qquad \text{NOR}$$

$$(x_1 \, x_2 \, x_3 \, \ldots \, x_n)' \qquad \text{NAND}$$

$$x \qquad x' \qquad \text{INVERT}$$
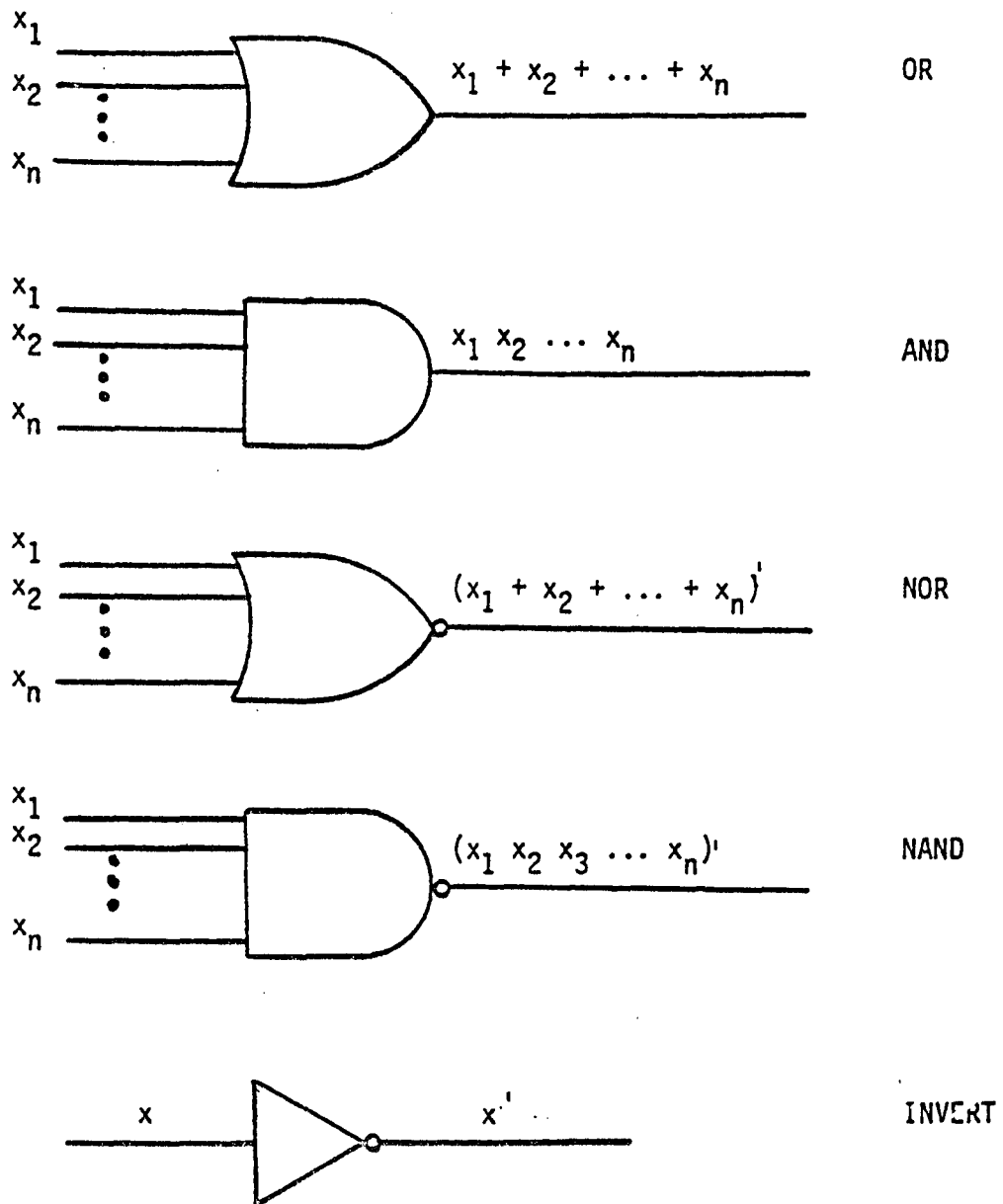
FIGURE 7   COMMON GATES

58
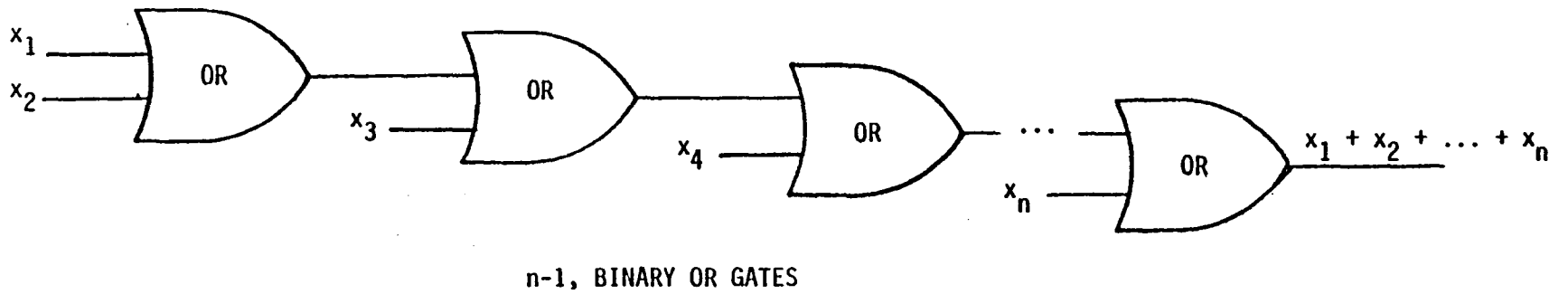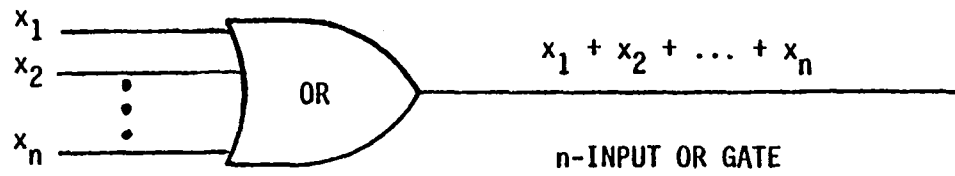
FIGURE 8   EQUIVALENCE OF AN n-INPUT "OR" GATE AND n-1, BINARY "OR" GATES

SERIAL MODE SIMULATION

$c = (a_1 + b_1,\ a_2 + b_2,\ \ldots,\ a_m + b_n)$
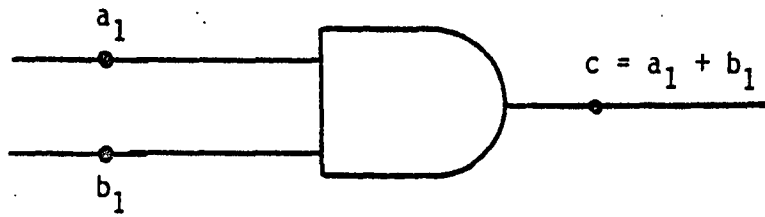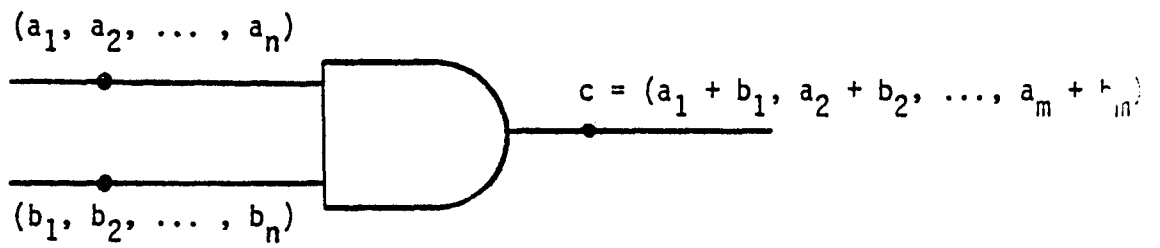
PARALLEL MODE SIMULATION

FIGURE 9   PARALLEL/SERIAL MODE SIMULATION
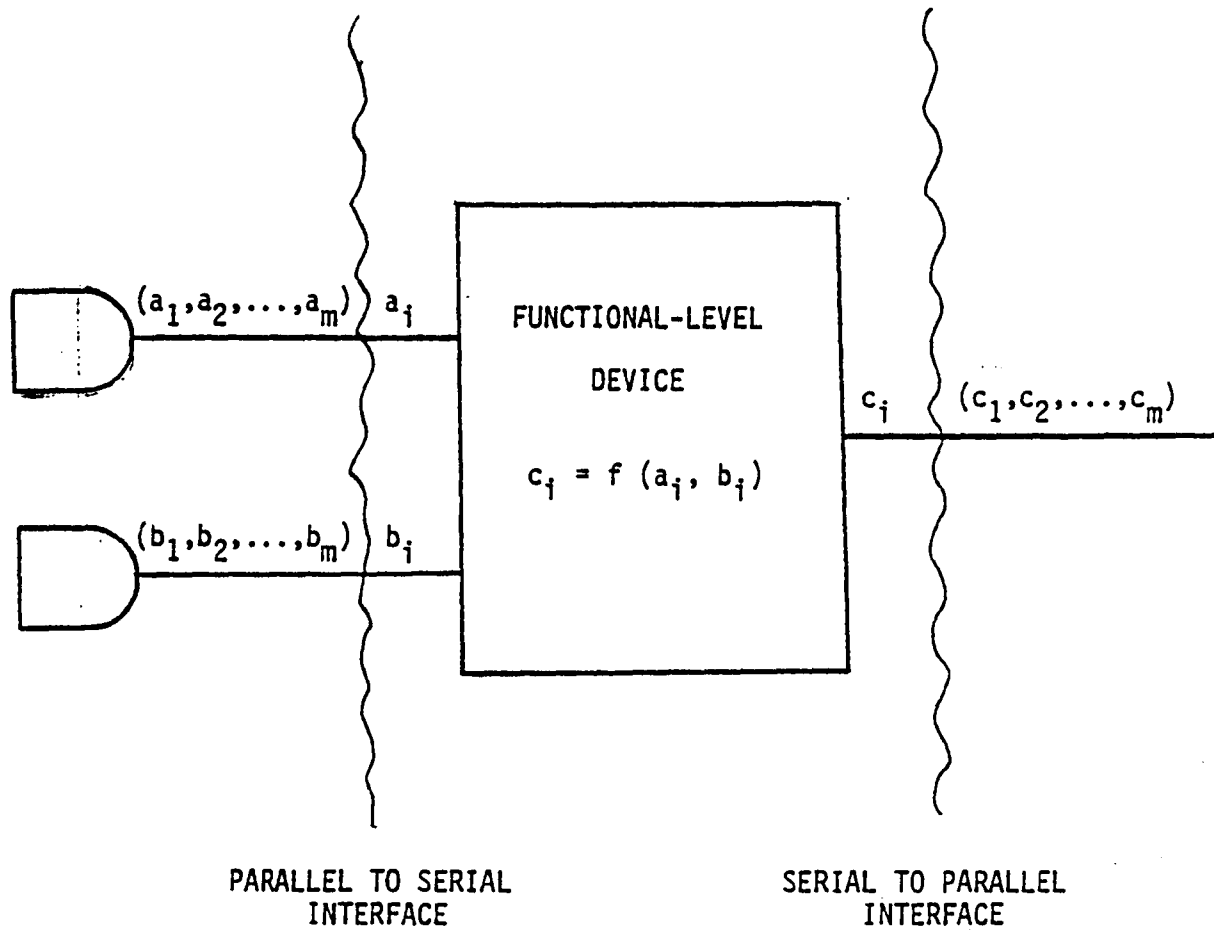
FIGURE 10   TYPICAL PARALLEL/SERIAL INTERFACES

IMPLEMENTATION #1

S = (A+B)(C+D)+EF

IMPLEMENTATION #2
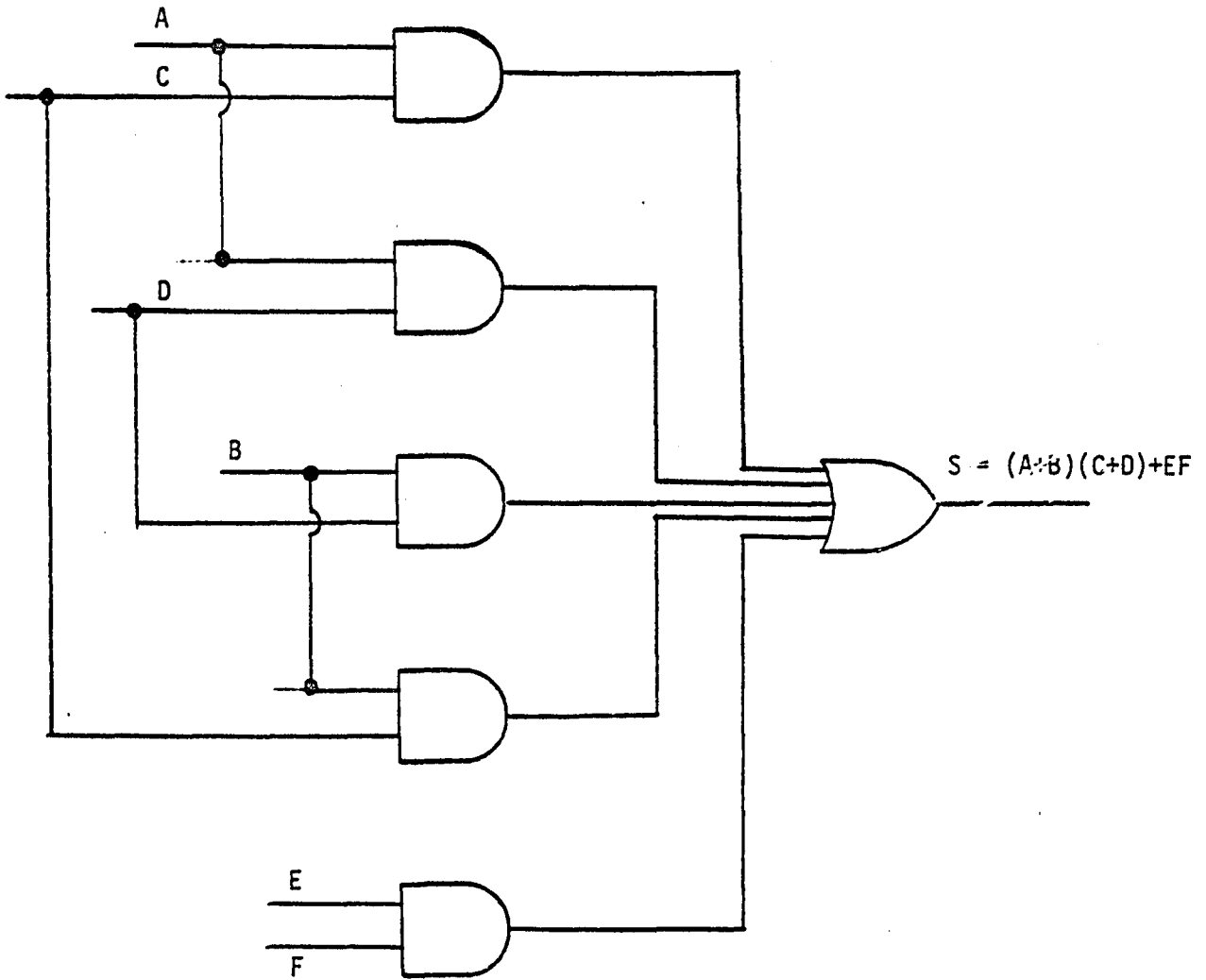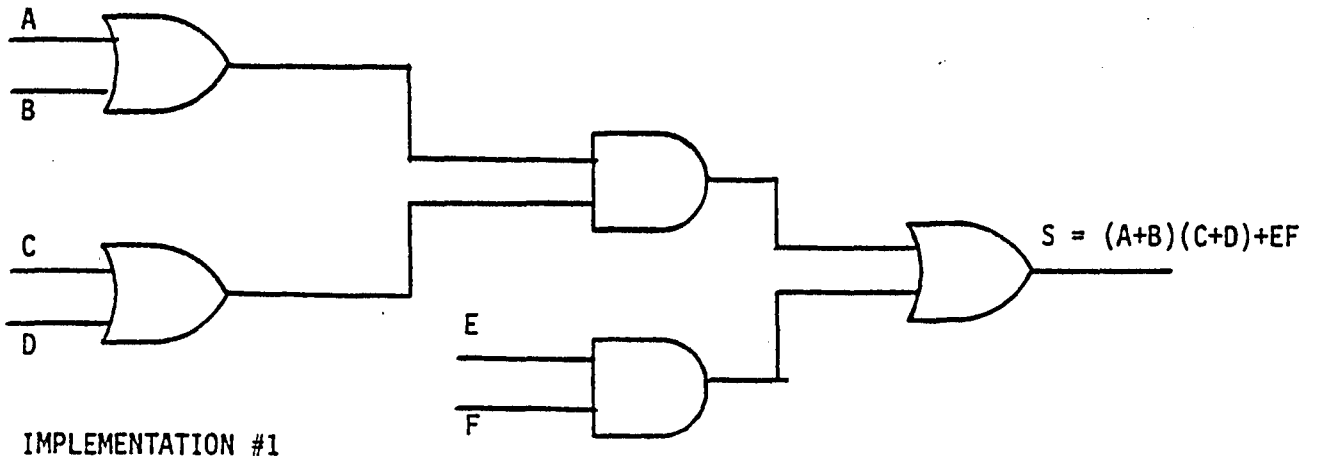
S = (A+B)(C+D)+EF

FIGURE 11   GATE-EQUIVALENT IMPLEMENTATIONS OF S = (A+B)(C+D)+EF
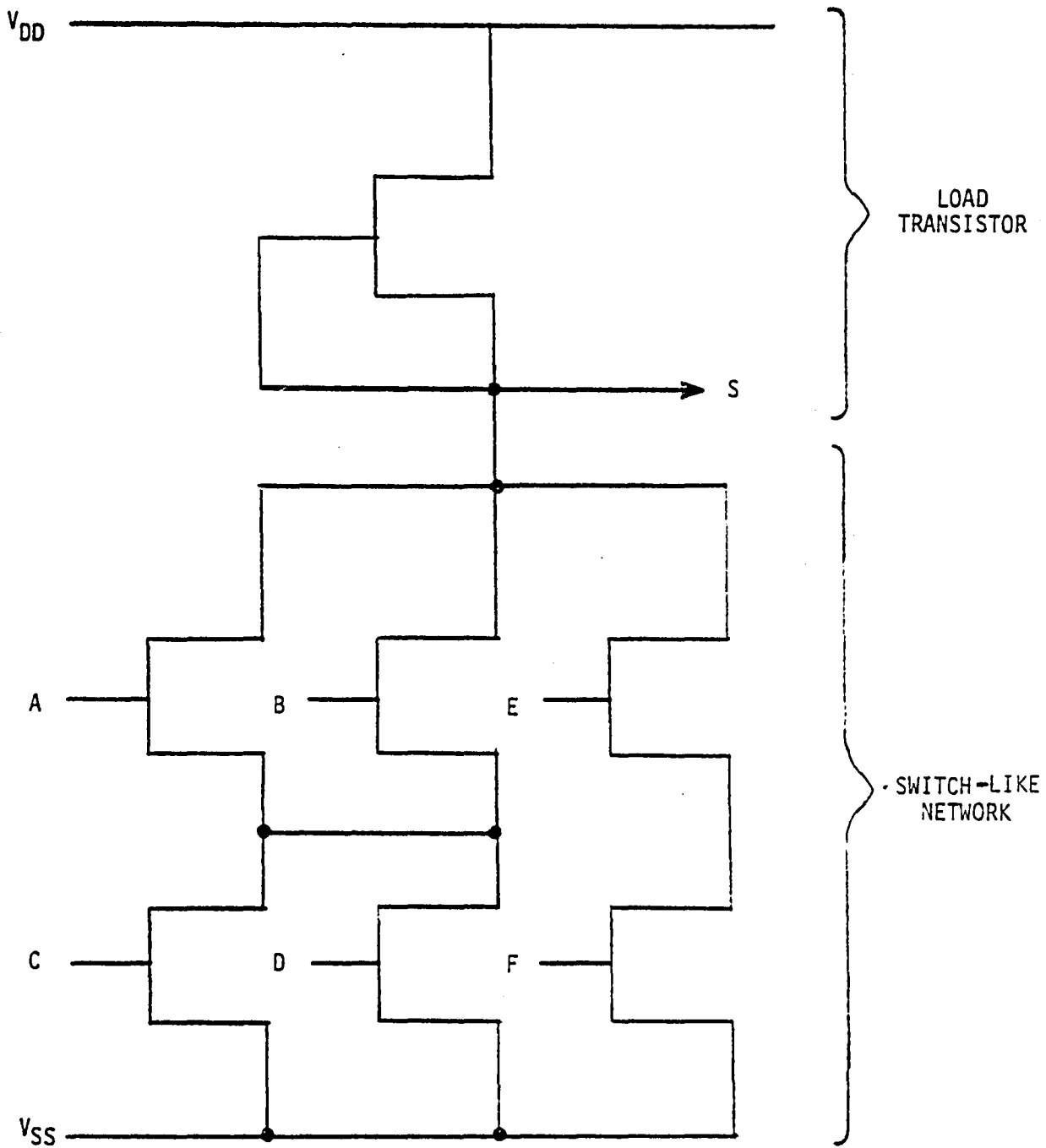
$$S = (A + B) \ (C + D) + EF$$

FIGURE 12   TRANSISTOR NETWORK REALIZING S = (A + B) (C + D) + EF

VDD

S

#4

#1   #3

A

#6   B   #7   E

#2   <----> = SHORT FAULT
|| = OPEN FAULT

C

D   F

VSS

#5

TRANSISTOR CIRCUIT

#1
s-a-o

A

#3
C s-a-o
D

#2
s-a-1
E
F

S=(A+B) (C+D) + EF

#4
s-a-o

FAULTS #5, #6, #7
NOT COVERED

GATE - EQUIVALENT CIRCUIT #1

#1   #6
A s-a-o   s-a-o

D

B   s-a-o

#3
s-a-o   s-a-o

#2
E s-a-1
F

S=(A+B) (C+D) + EF

#4
s-a-o

FAULT #7
NOT COVERED

#5
s-a-o
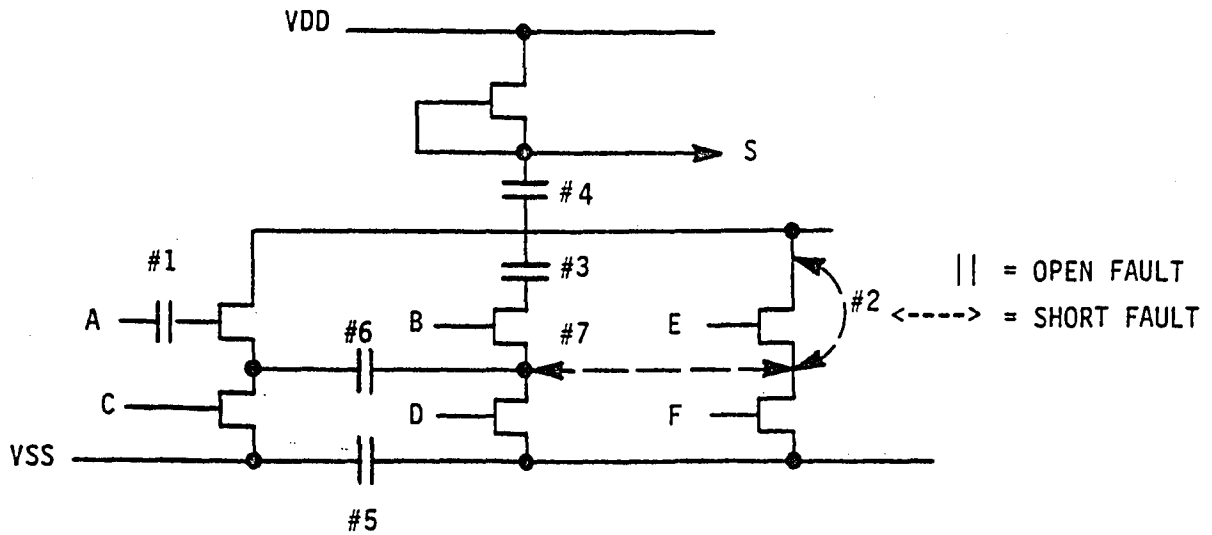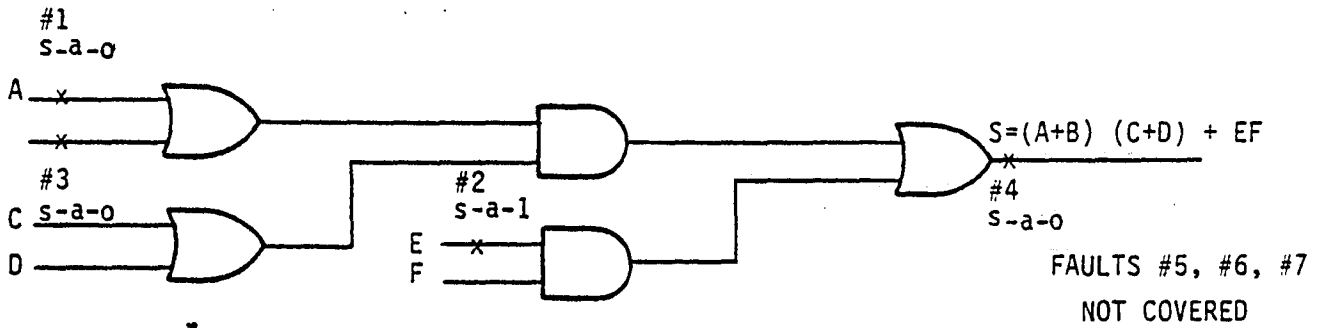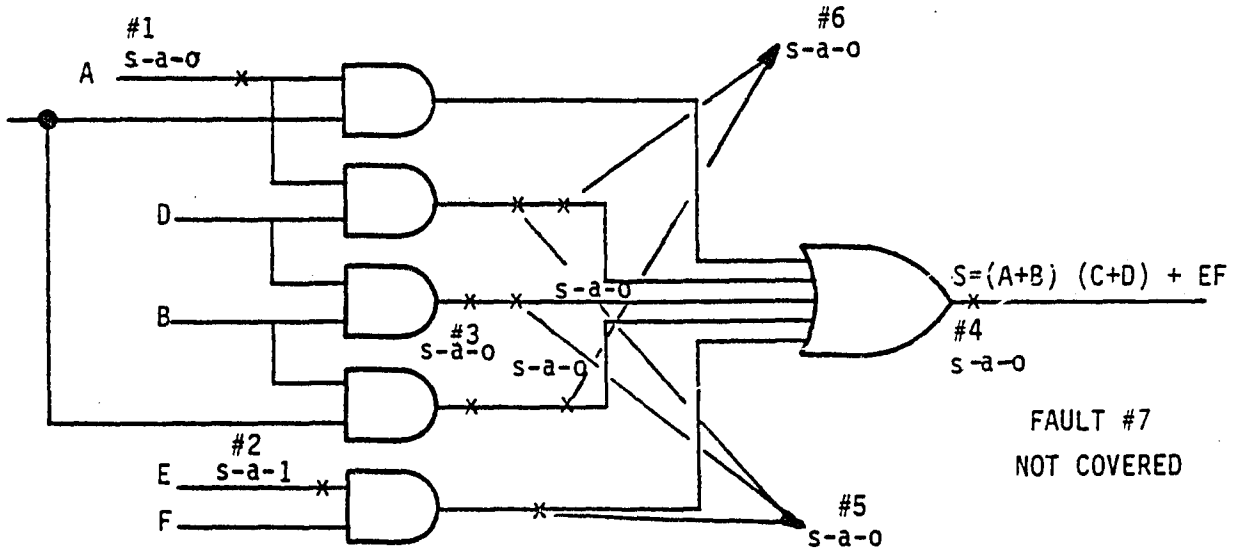
GATE - EQUIVALENT CIRCUIT #2

FIGURE 13   FAULT CORRESPONDENCES IN GATE - EQUIVALENT CIRCUITS

64
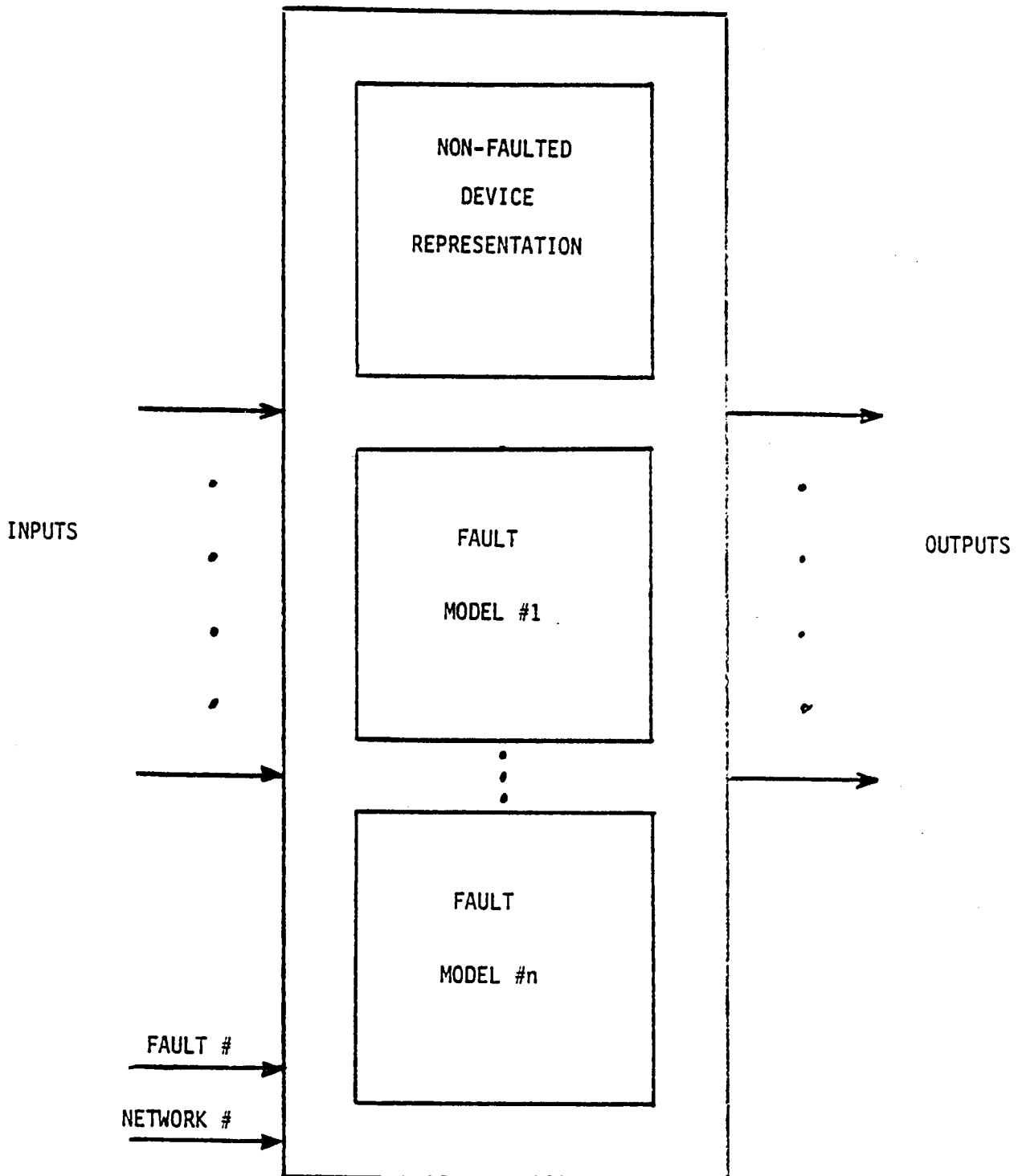
FIGURE 14   PROCEDURE FOR SIMULATING FAULTS IN A FUNCTIONAL - LEVEL DEVICE

F = SPECIAL FAULT QUALIFIER
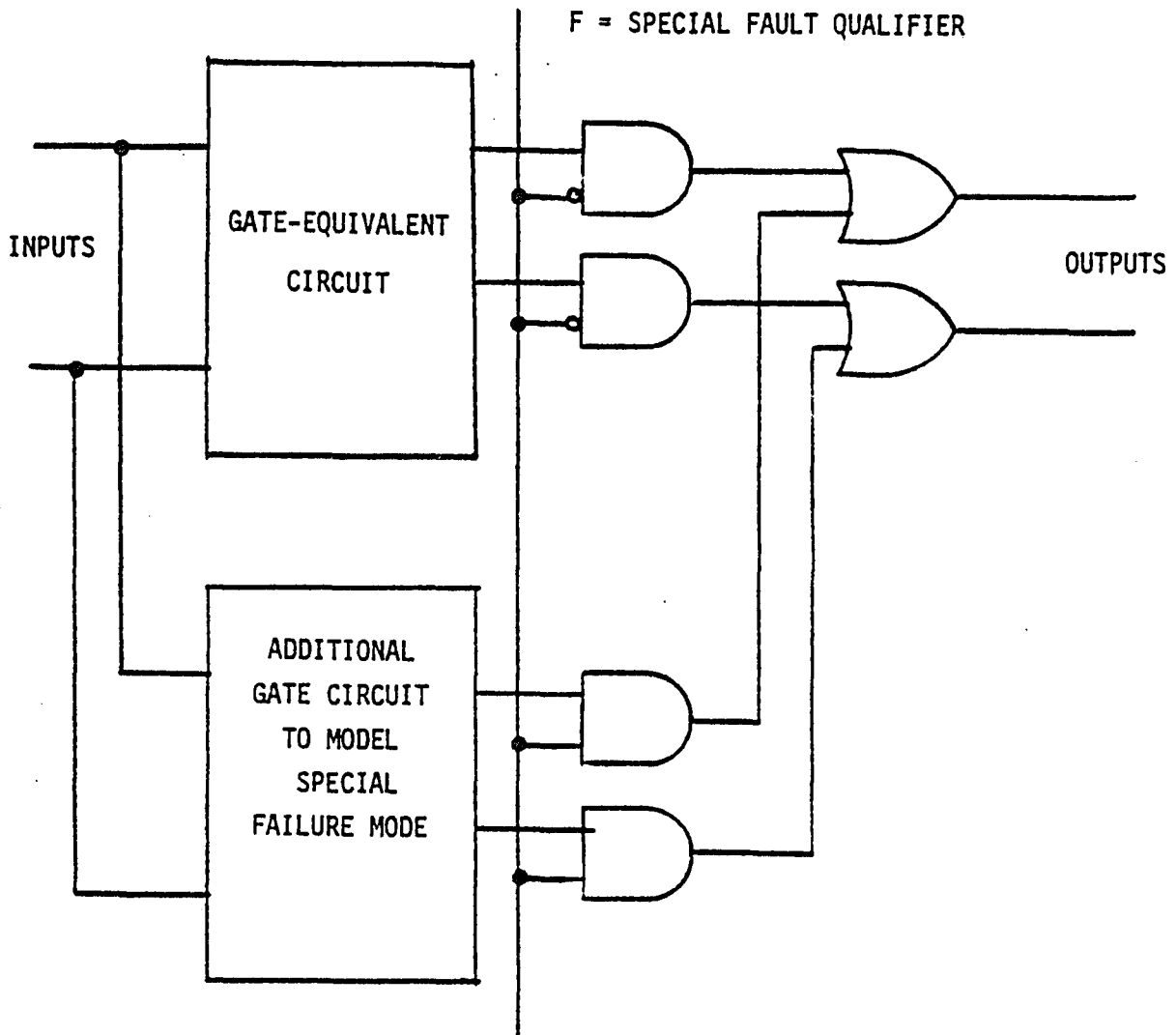
FIGURE 15   A METHOD OF PARALLEL SIMULATION OF SPECIAL FAILURE MODES

66
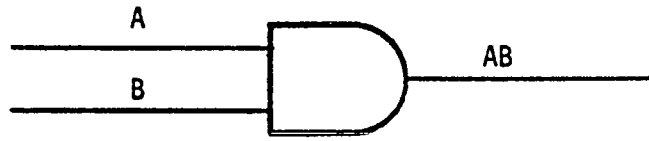
FIGURE 16A   NON-FAULTED AND GATE
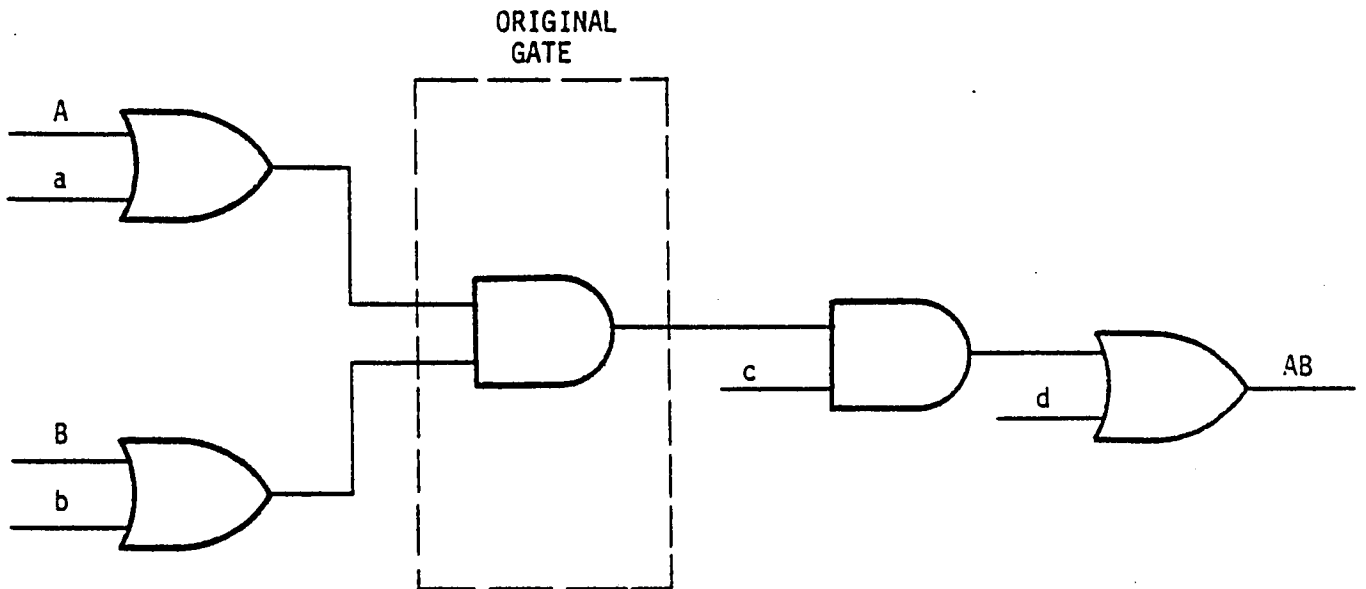
FIGURE 16B   AND GATE FAULT MODEL

FIGURE 16   STANDARD FAULT MODEL OF A GATE

67

n GATES

NETWORK #1

n GATES
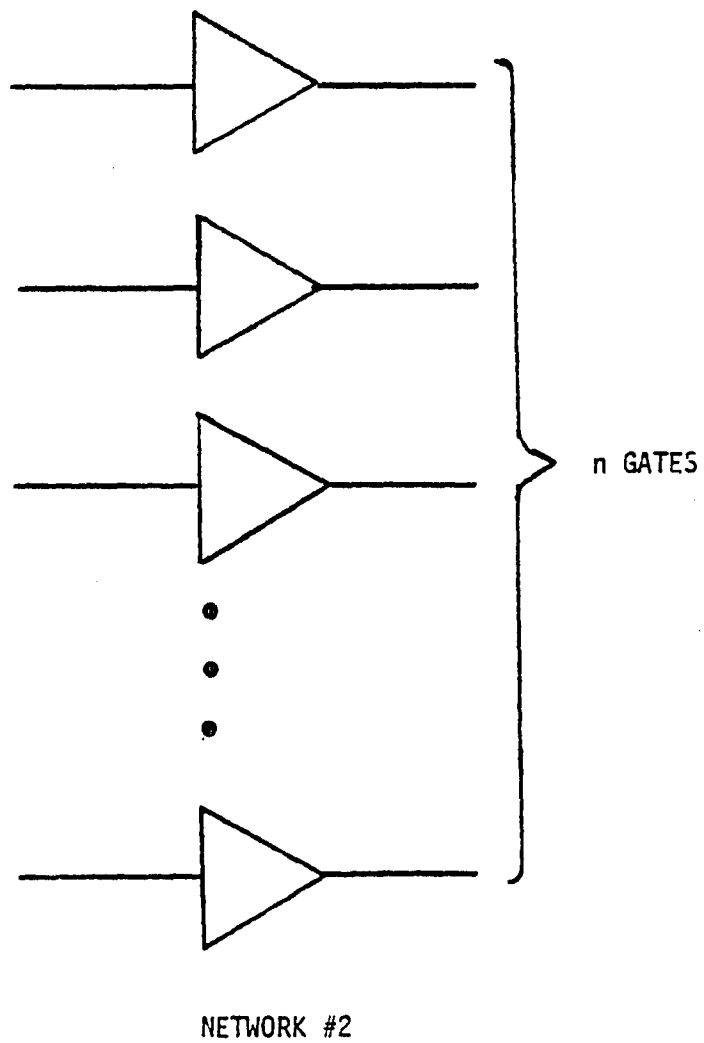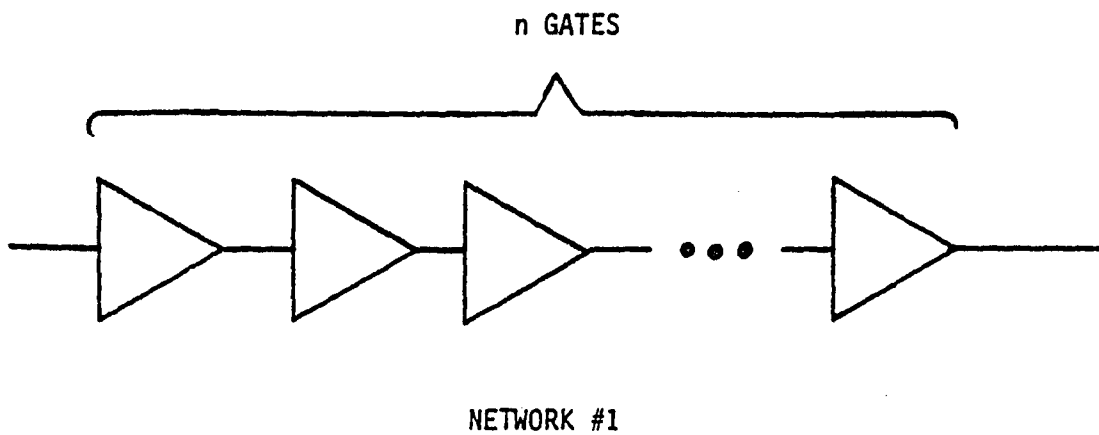
NETWORK #2

FIGURE 17   COMBINATIONAL NETWORKS OF DIFFERENT STRUCTURE
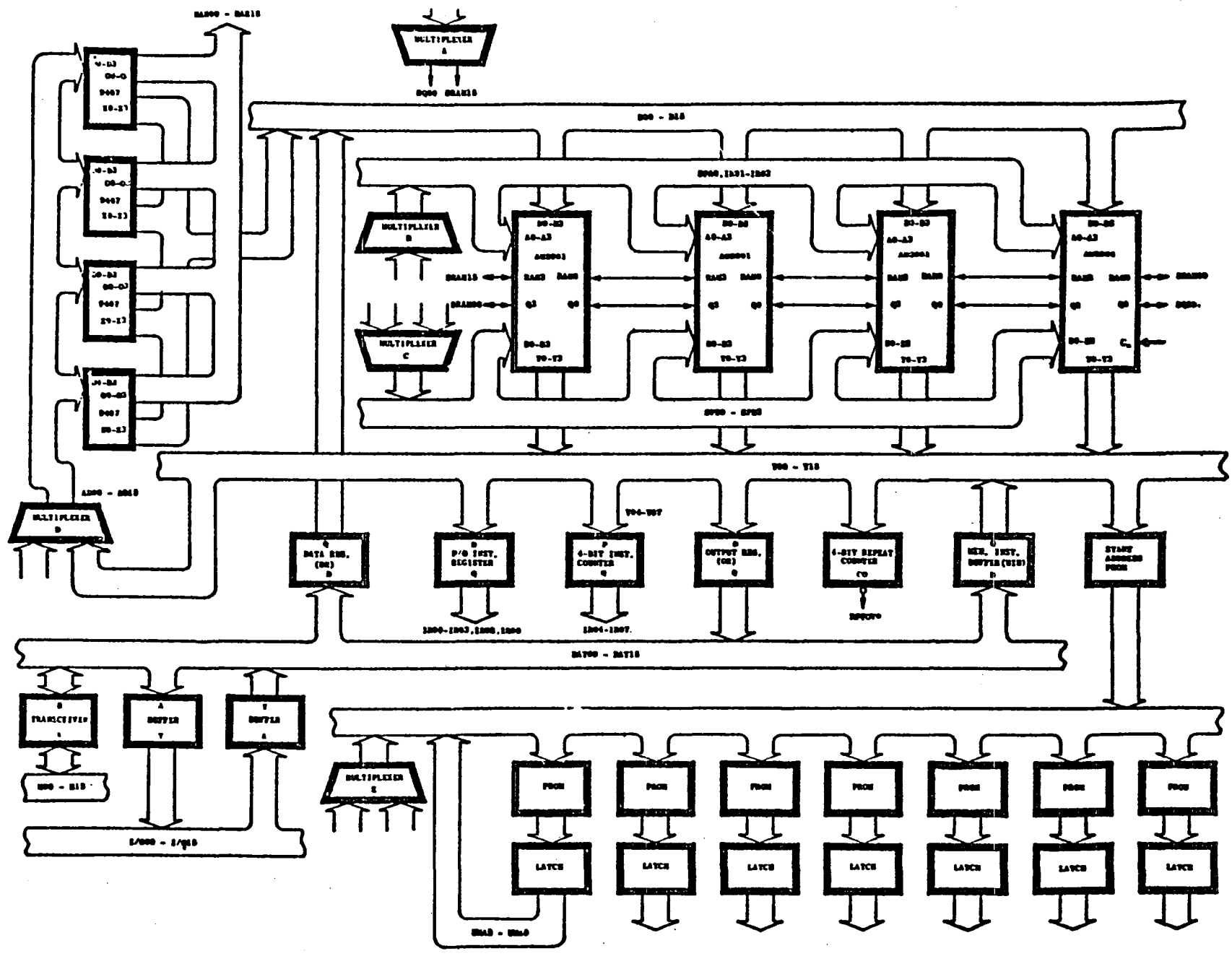
FIGURE 18   BDX-930 PROCESSOR
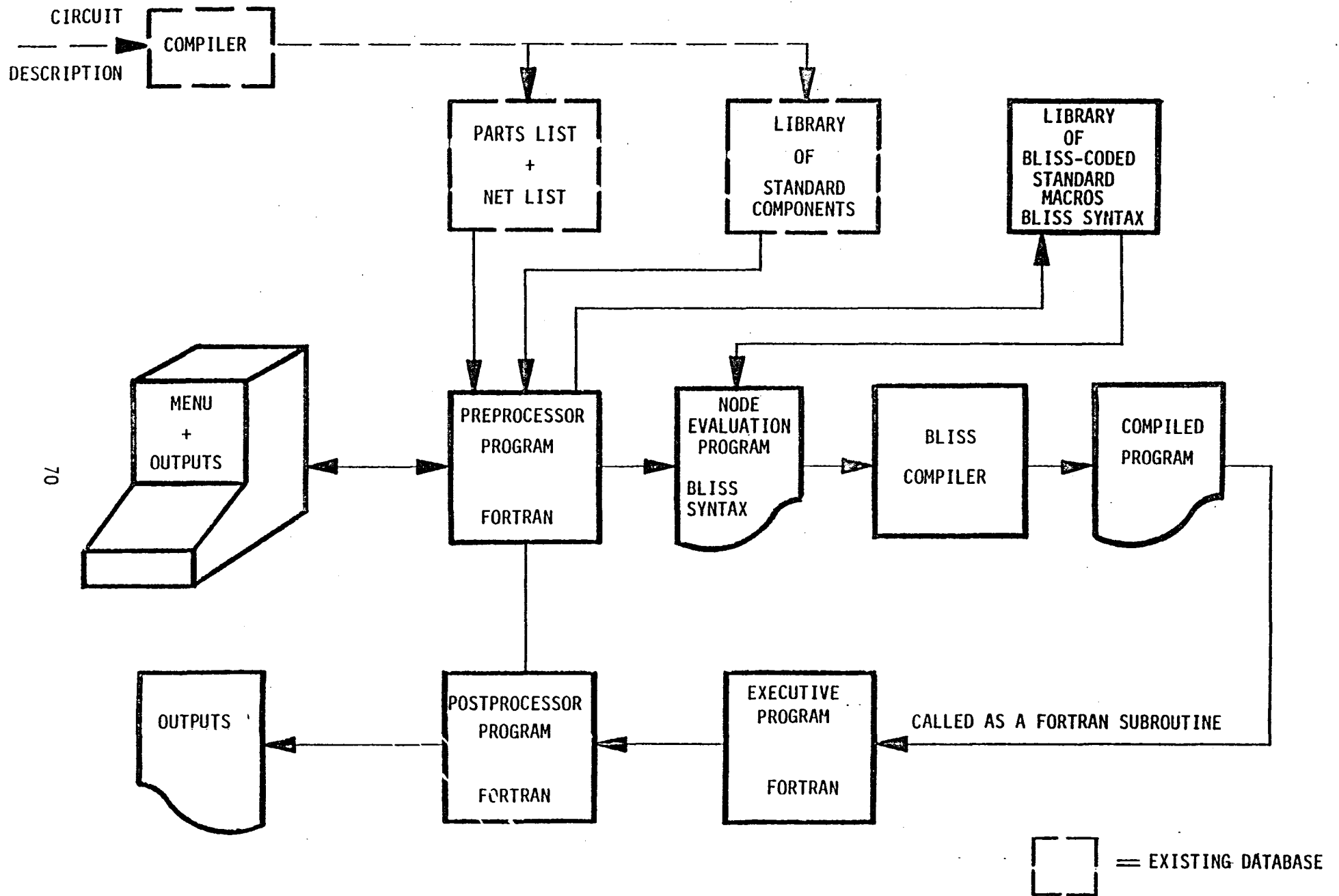
FIGURE 19   PROPOSED STRUCTURE OF GGLOSS

70

APPENDIX A
PROPERTIES OF LOOP-FREE NETWORKS

We state several important properties of loop-free networks. These are given in the form of lemmas. We note that, since combinational networks are primitive loop-free networks, all of the properties of loop-free networks apply to combinational networks, as well.

Given a loop-free network with n nodes (including E-nodes).

Lemma 1.

There exists a unique sequence of disjoint and non-empty sets, $R(0),R(1),R(2),....,R(m)$, $m \leq n-1$, such that

   1) each node is a member of exactly one set

   2) $R(0)$ contains exactly those nodes with no inputs

   3) $R(k)$, $k=1,2,....,m$, contains those and only those nodes whose inputs originate in nodes belonging to the union of $R(0),R(1),....,R(k-1)$.

Nodes in $R(k)$ are said to have "rank order k".

If m is the largest rank order of any node then m is called the "depth" of the network.

Lemma 2.

If $m =$ depth of the network then there exists at least one path of length m and no path exceeds m in length. (As a consequence, a signal can be propagated through a network after, at most, m gate delays.)

Lemma 3.

If nodes in $R(0)$ are selected in any order, followed by nodes in $R(1)$, in any order, etc., then the resultant sequence is a p-ordering of the nodes.

Lemma 4.

There exists at least one p-ordering of nodes of a network without loops.

Example. Figure A-1 shows a rank-ordered network. Using the node designations of the figure a p-ordering is $N(0)$, $N(1)$,....,$N(11)$. The network has depth $= 4$.

71

An Algorithm For Determining a P-Ordering Of Nodes

Lemmas 1 and 3 embody a procedure for determining a p-ordering of a network without loops.

Step 1.

Identify all nodes with no inputs and place these in R(0)(If R(0) is empty and at least one node remains then the network has a loop).

Step 2.

Remove the nodes of R(0) and their outputs. In the reduced network identify all nodes with no inputs and place these in R(1) (If R(1) is empty and at least one node remains then the network has a loop).

Repeat the above process until all nodes are exhausted. The desired p-ordering is obtained as prescribed in Lemma 3.
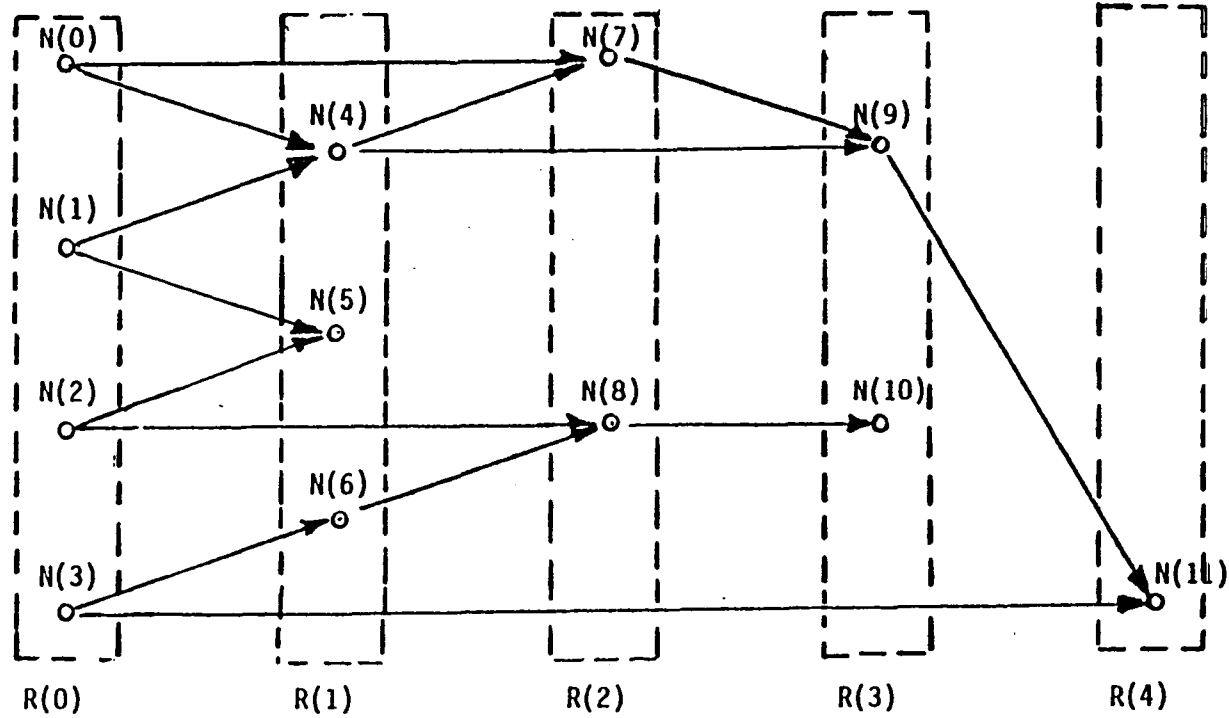
FIGURE A-1   EXAMPLE OF RANK-ORDERING

APPENDIX B
SPECIAL DIGITAL DEVICES

The techniques of modelling and simulation described in the previous sections will be illustrated by means of several basic digital devices:

    I) Flip flops
    II) Tristate busses
    III) Functional-Level Program Memory

These devices were selected because they

    o present unique problems of modelling and simulation;
    o comprise the basic components of digital processors;
    o constitute the most frequently simulated digital devices.

It is important to emphasize that the techniques of modelling and simulation described herein are presented to assist the User in setting up appropriate circuit models. GGLOSS, after all, merely evaluates gates. The selection of an appropriate network model, to which these techniques refer, remains the responsibility of the User.

I. FLIP FLOPS

1. R-S Flip Flop

A gate-equivalent circuit which uses the NAND gate implementation is shown in Figure B-1A. An alternate implementation, using NOR gates, is shown in Figure B-2. Since the essential dynamics are identical in both representations we will consider only the NAND gate circuit. From the figure we obtain the characteristic function of the flip flop:

1)
$$x(n) = S(n-1) + y'(n-1)$$
$$y(n) = R(n-1) + x'(n-1)$$

where   n denotes time t=nd
                d=propagation delay of a gate
                $(\ \ )' = $ complement of $(\ \ )$.

Nominally, y is the complement of x except when the forbidden pair, S=R=1, occurs. The state diagram and truth table are shown in Figure B-1B and B-1C, respectively.

It is recalled that the U simulation will evaluate both NAND gates in every primitive clock cycle (i.e.,a cycle equal to the delay of a gate). From the state diagram it can be seen that, starting in one of

the states (x=0,y=1) or (x=1,y=0), it could require two clock cycles to reach the final state. The operation of the flip flop is described as follows:

S=0,R=0: The flip flop does not change state and one clock cycle is sufficient to stabilize the outputs. While this is a normal input combination it could, nevertheless, result in an oscillation. From the state diagram of Figure B-1B it can be seen that the pairs (S=0,R=1) and (S=1,R=0) result in a transition through the state (x=0,y=0). If, while in this state, (S=0,R=0), then the oscillation will occur. To avoid this we require that the time between successive changes in the pair, (S,R), must exceed the propagation delay of the flip flop, i.e., at least one gate delay.

S=0,R=1: If the previous state is (x=0,y=1) the state will remain unchanged. Again, one clock cycle is sufficient to stabilize. If the previous state is (x=1,y=0) the state will change to (x=0,y=1) after two clock cycles.

S=1,R=0: If the previous is (x=1,y=0) the state will remain unchanged. Again, one clock cycle is sufficient to stabilize. If the previous state is (x=0,y=1) the state will change to (x=1,y=0) after two clock cycles.

S=1,R=1: This is the forbidden combination. After, at most, two clock cycles the final state is (x=1,y=1). This presents two problems:

1) y is not the complement of x;
2) while in state (x=1,y=1) the occurrence of the normal pair, S=R=0, will result in an oscillation.

In a simulation in which gate delays are multiples of the primitive clock cycle,d, it is probable that the combination S=R=1 followed by S=R=0 will occur frequently. To avoid this and the resultant oscillation it is recommended that the SET and RESET inputs be modified as shown in Figure B-3. This circuit prevents the occurrence of S=R=1, replacing it with S=R=0. All other combinations of S and R remain unchanged.

From this example it can be seen that the R-S flip flop can be simulated with two NAND gates(excluding the fictitious circuit of Figure B-3) and, to reach a stable value, requires four gate evaluations.

2. D-Flip Flop

The conventional gate-equivalent representation of the D-flip flop is shown in Figure B-4A. The corresponding truth table is shown in Figure B-4B. It will be shown, subsequently, that the dynamics of the D-flip

flop, as determined by the gate-equivalent circuit, are more complicated than the truth table indicates. For the present we give a textbook description of the flip flop:

1) When C=0 then S'=R'=1, independently of D. The x and y outputs remain unchanged.

2) If D=0 and C=0 and C changes to a 1 then S'=1 and R'=0 and, hence, x=0,y=1. If now, while C=1, there are any subsequent changes in D, S' and R' will not change states nor will x and y. When C returns to 0, S'=R'=1, so that the flip flop remembers its previous state.

3) If D=1 and C=0 and C changes to a 1 then S'=0 and R'=1 and, hence, x=1,y=0. When C returns to 0, S'=R'=1, so that the flip flop remembers its previous state.

This description is presumably based on the gate-equivalent circuit of Figure B-4A. If so, it is misleading. It can be shown, in fact, that these characteristics are as described if the following conditions hold:

C1)The positive and zero clock intervals each exceed the maximum propagation delay of the flip flop.

C2)A change in the data input, D, and an edge of the clock pulse are separated,in time, by at least one gate delay.

While it may be true that these conditions are normally satisfied in a well-designed network, they may not apply in a simulation, particularly if the user is careless. If either condition is not satisfied the flip flop could oscillate.

Example. If conditions (C1) and (C2) do not apply then the complete excitation table for S and R is given in Table B-1. The corresponding state diagram is shown in Figure B-5. As an illustration, assume that condition (C2) does not apply. Let

        D=0,C=0 at time=0
        D=1,C=0 at time=d
        D=0,C=1 at time=2d,3d,4d,....

From the excitation table the successive states are seen to be

        (0,1,1,1) at time=0
        (0,1,1,1) at time=d
        (0,1,1,0) at time=2d
        (1,1,1,1) at time=3d
        (0,0,0,0) at time=4d
        (1,1,1,1) at time=5d
        (0,0,0,0) at time=6d
              etc.

where a state is defined as (a,S',R',b), the parameters being referenced to Figure B-4A.

After time=3d, S' and R' oscillate between S'=R'=1 and S'=R'=0, causing an oscillation in the x and y outputs of the D-flip flop.

Under non-faulted conditions if the input does not change more than once in a clock cycle, it could require 2 gate delays to stabilize the x and y outputs at both the leading edge and descending edge of the clock pulse. Using the U-Simulation it would require 24 gate evaluations in each clock cycle(since there are 6 gates). However, if it is known, a priori, that conditions (C1) and (C2) hold and the input will not change more than once in a complete clock cycle, then a much simpler D-flip flop model can be used. Under these conditions the characteristic function of the flip flop is

1)
$$x(1) = x(0)*(C(1)*C'(0))' + D(1)*C(1)*C'(0)$$

$$y(1) = y(0)*(C(1)*C'(0))' + D(1)*C(1)*C'(0)$$

where
$C(0)$ = past clock value (0 or 1)
$C(1)$ = present clock value (0 or 1)
$x(0),y(0)$ = past values of x and y
$x(1),y(1)$ = present values of x and y.

It is important to observe that the D-flip flop requires that the clock return to 0 before the next data input----otherwise no output change is possible. The time between the "past" and "present" is the time between the rising and descending edges of the clock pulse or between the descending and rising edges. To properly evaluate the model it is neccessary to evaluate it, once at the leading edge of the clock pulse and, again, at the descending edge. A non-faulted gate-equivalent circuit of this model is shown in Figure B-6. Observe that, in the non-faulted case, $y(1)=x'(1)$. One advantage of this model is that it also models clock faults which prevent the clock from oscillating. If, however, these clock faults, i.e., failure to transition through zero, are ruled-out then the D-flip flop admits of an even simpler model. Its characteristic function is

2)
$$x(1) = x(0)*C'(1) + D(1)*C(1)$$

$$y(1) = y(0)*C'(1) + D(1)*C(1).$$

This model can be evaluated at the rising edges of the clock, exclusively, or at both the rising and descending edges. This was the model of the D-flip flop used in BGLOSS (with the additional assumption that $y(1)=x'(1)$). A non-faulted gate-equivalent model of this circuit is shown in Figure B-7. Observe the clocked feedback branches in both models.

An additional advantage conferred by these simpler models is that they can be simulated by the Z simulation. This is made possible because the time delay of the feedback signals, $x(0)$, $y(0)$, $c(0)$, can be set equal to the clock cycle (or half of the cycle, in the case of $c(0)$) and, hence, is large relative to the propagation delay of the straight-through elements.

II. TRISTATE BUSSES

Figure B-8 shows five typical bus interface arrangements. Unless the contrary is stated the following discussion applies to all arrangements.

At any given time one and only one terminal is transmitting (assuming no failures). In this case the transmitting terminal sets its impedance level to

Z=low (=logic level 0).

This transfers the signal to the bus. A transmitted output may be read by one or more receiving terminals. The high impedance state effectively cuts-off its associated gate while the low impedance state activates it. Under prescribed operating conditions the bus value is determined as follows:

1) If only one transmitter is in its low impedance state the bus value is that of the transmitting terminal.
2) If two or more transmitters are in their low impedance state and if they are transmitting the same value then the bus is set to this value.

A potential problem arises when

a) two or more transmitters are in their low impedance state and are transmitting different values;
b) all transmitters are in their high impedance states.

In the first case the bus could assume a 0 or 1 value, depending upon the particular bus interface. In the second case the bus value could depend upon its prior value, e.g., if bus=1 previously, then bus=1, now; if bus=0 previously, then it could take many gate delays before the bus finally assumes the value 1. The correct model requires a careful analysis of the bus interface circuitry.

For BGLOSS it was determined that, in the context of the BDX-930 cpu, the tristate bus operates like wired AND logic, i.e., it was functionally equivalent to connecting the transmitted outputs to the input of an AND gate. Moreover, when all impedance levels were high, the bus value was a logic 1. Thus, if

b(k)=transmitted signal of terminal #k
Z(k)=impedance level of the transmitter
   n =number of transmitters

78

then the bus value was given by

$$BUS=(b(1)+Z(1))AND(b(2)+Z(2))AND...AND(b(n)+Z(n)).$$

Observe that all transmitters must be evaluated before the bus value can be determined. The gate-equivalent circuit of the transmitter/bus used in BGLOSS is shown in Figure B-9. The figure also shows the fault set used in the simulation. Observe that the impedance states are represented by either a logic 0 or logic 1.

Transmitter/Receiver Models

With one exception, no device of the BDX-930 cpu transmitted and received data over the same bus. Thus, the bus interface configurations of these devices correspond to arrangements #1 and #2 of Figure B-8. A typical BGLOSS model of this transmitter/receiver arrangement is shown in Figure B-10, including the placement of stuck-at faults. The one exception is the bidirectional transceiver connecting the DATA and Memory busses (the memory, itself, contains a bidirectional receiver but this was not simulated in BGLOSS). This arrangement corresponds to #5 of Figure B-8. The BGLOSS model of this tranceiver is shown in Figure B-11. Faults were injected, as shown.

Modelling a Bidirectional Transceiver

As indicated previously, a correct model of a transceiver requires a detailed analysis of the interface circuitry. Consequently, we do not presume here to offer such a model. We offer, instead, a candidate model which could be correct given the assumption that the bus operates like wired AND logic.

[It cannot be overemphasized that a correct transceiver model, as well as other circuit models, is the responsibility of the User. If such a model can be represented by a prototype network, as described in Section 3.1, then the model can be simulated.]

The bidirectional transceiver is that connecting the Memory Bus and the Ram Memory, as shown in Figure B-12A. If the memory matrix is represented at the functional-level then the transceiver and memory can be represented as a combinational network as shown in Figure B-12B. A prototype network model of the device is shown in Figure B-13. This model also includes stuck-at faults.

III. FUNCTIONAL-LEVEL PROGRAM MEMORY

As indicated in Section 4.3.1, the only functional-level devices used in BGLOSS were memory devices. To illustrate some of the considerations involved in modelling these devices we will examine the BGLOSS simulation of Main Program Memory(ROM).

79

Referring to Figure 18 the 9407 loads the memory address bus. This is a unidirectional bus and presents no special problems of simulation. The memory data bus is bidirectional and the techniques used to simulate it have been described in Appendix B (see Figure B-11). At the start of a clock cycle the program memory is commanded to write a word on the memory data bus (at this time data is stable on both busses).

Of the 16 bits of the address bus only 15 are actually used to address memory. Consequently, the BDX-930 can address 32k words. In general it would have been necessary to allocate 32k words of the host computer to simulate the BDX-930 memory. However, only enough memory was allocated to store the simulated program—which never exceeded 2200 words. A memory address outside of this range(as a result of a fault) caused BGLOSS to write a "HALT" instruction on the memory bus. The rationale for this was that a jump out of the program would have been detected by any monitoring strategy.

In BGLOSS it was assumed that no faults occurred in main memory. Thus it was only necessary to simulate a single memory for all 32 parallel processors. The contents of the simulated memory was then loaded with the bit patterns of the simulated software program. Thus, memory was represented by the array

$$(16\text{-bit address word, }16\text{-bit memory word}).$$

As indicated in Section 3.3.4, when a device is represented at the func-tional-level in a parallel mode simulation it is necessary to perform a parallel-to-serial or serial-to-parallel conversion when crossing its boundary. This is a time consuming operation and, if inefficiently im-plemented, can result in a significant decrease in simulation speed. Because of its importance we will describe the conversion algorithm used by BGLOSS, in detail.

In the parallel mode the address bus is represented by the 16 words:

$A(1) = bit\#1 = (a(1,1), a(1,2),...., a(1,32))$

$A(2) = bit\#2 = (a(2,1), a(2,2),...., a(2,32))$

$...$

$A(16) = bit\#16=(a(16,1), a(16,2),...., a(16,32))$

where it is assumed that each word of the host computer is 32 bits.

After the parallel-to-serial conversion we obtain 32 addresses, each corresponding to that of a different cpu:

B(1) = Address#1 = (a(1,1), a(2,1),...., a(16,1), x, x,...,x)

B(2) = Address#2 = (a(1,2), a(2,2),...., a(16,2), x, x,...,x)

...

B(32)= Address#32= (a(1,32), a(2,32),...., a(16,32), x, x,...,x)

where x = "don't care" bit.

The host computer uses each of the 32 words to fetch a word from memory. The conversion is repeated, in reverse, when loading the memory data bus.

The BGLOSS conversion algorithms are shown in Figures B-14 and B-15.
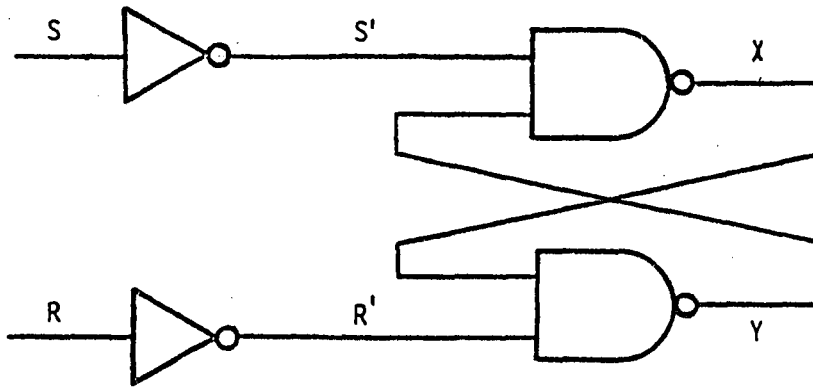
FIGURE B-1A   NAND GATE REPRESENTATION OF THE R-S FLIP FLOP
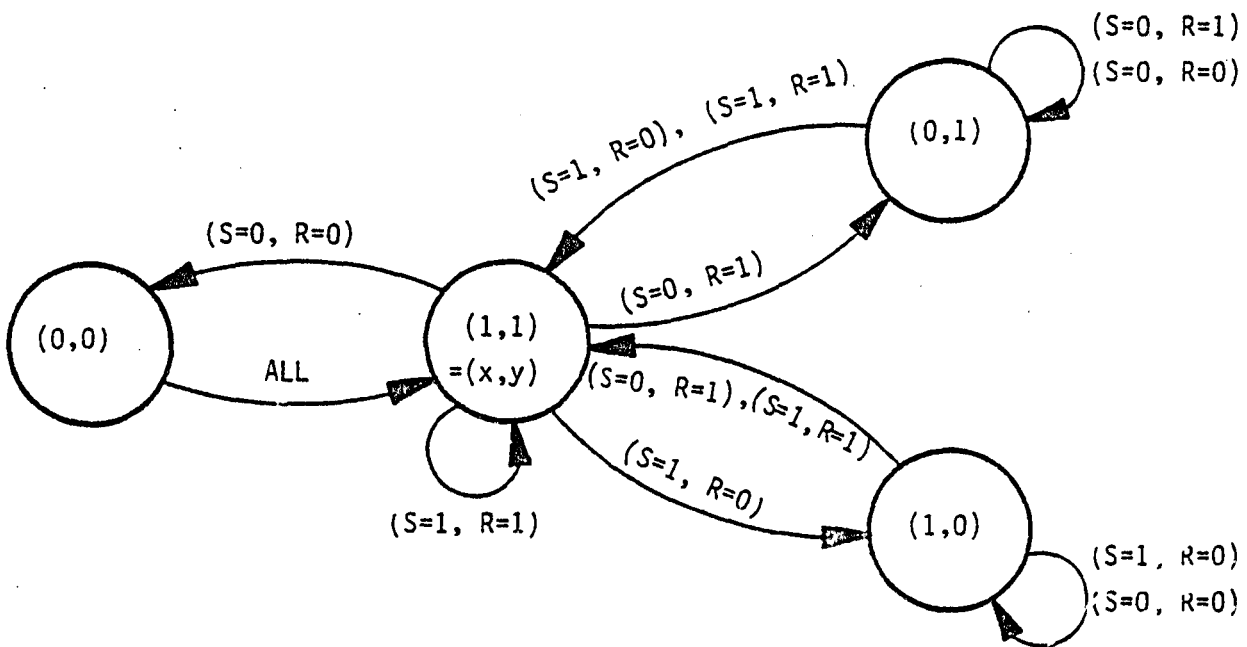
FIGURE B-1B   STATE DIAGRAM OF R-S FLIP FLOP (NAND GATE)

@ t=nd		@ t=(n+1)d	
S	R	x	y
0	0	x_n	y_n
0	1	0	1
1	0	1	0
1	1	INDETERMINATE	

FIGURE B-1C TRUTH TABLE FOR R-S FLIP FLOP

FIGURE B-1 R-S FLIP FLOP NAND GATE REPRESENTATION
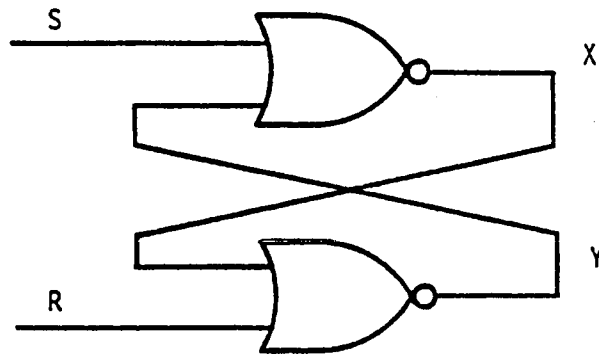
FIGURE B-2A   NOR GATE REPRESENTATION OF THE R-S FLIP FLOP
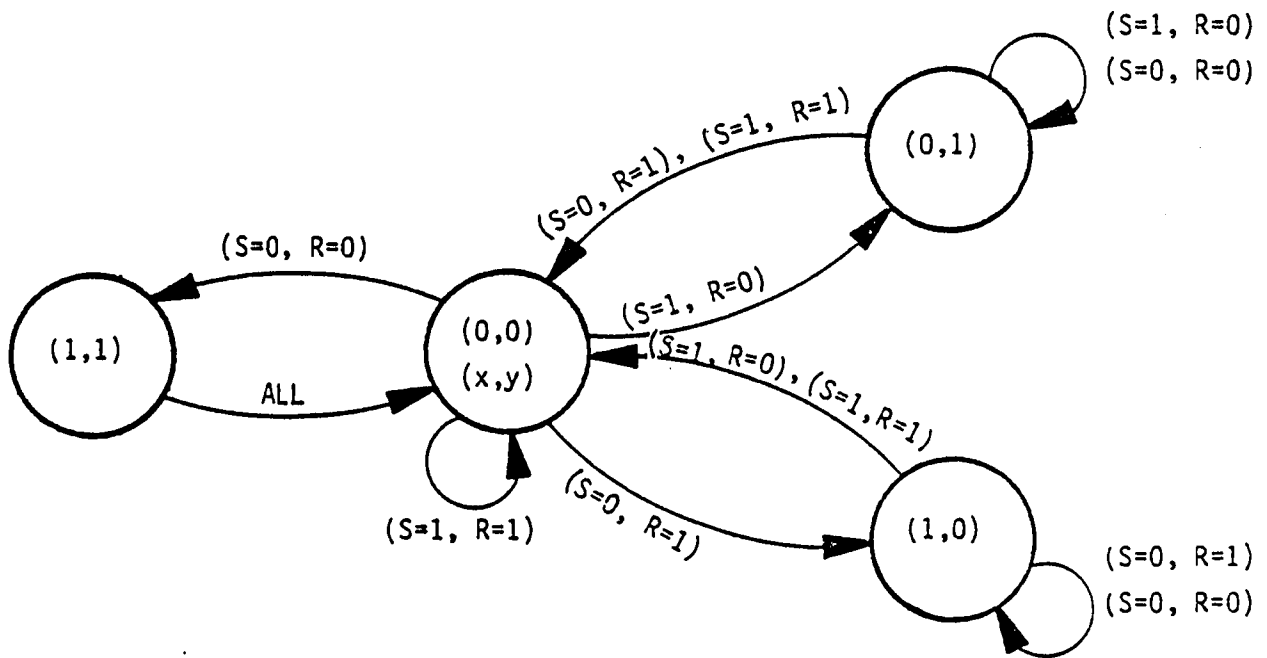
FIGURE B-2B   STATE DIAGRAM OF R-S FLIP FLOP (NOR GATES)

FIGURE B-2   R-S FLIP FLOP NOR GATE REPRESENTATION
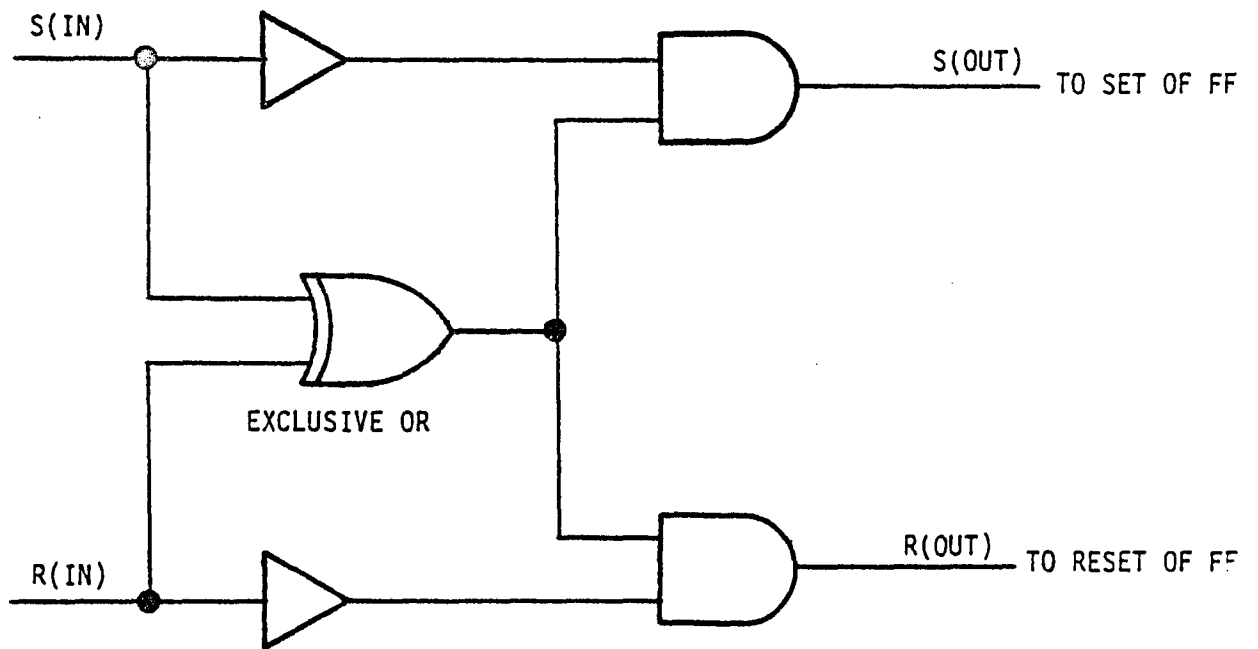
FIGURE B-3   LOGIC TO INHIBIT THE OCCURRENCE OF S=R=1

FIGURE B-4A
D - FLIP FLOP (NAND GATE)
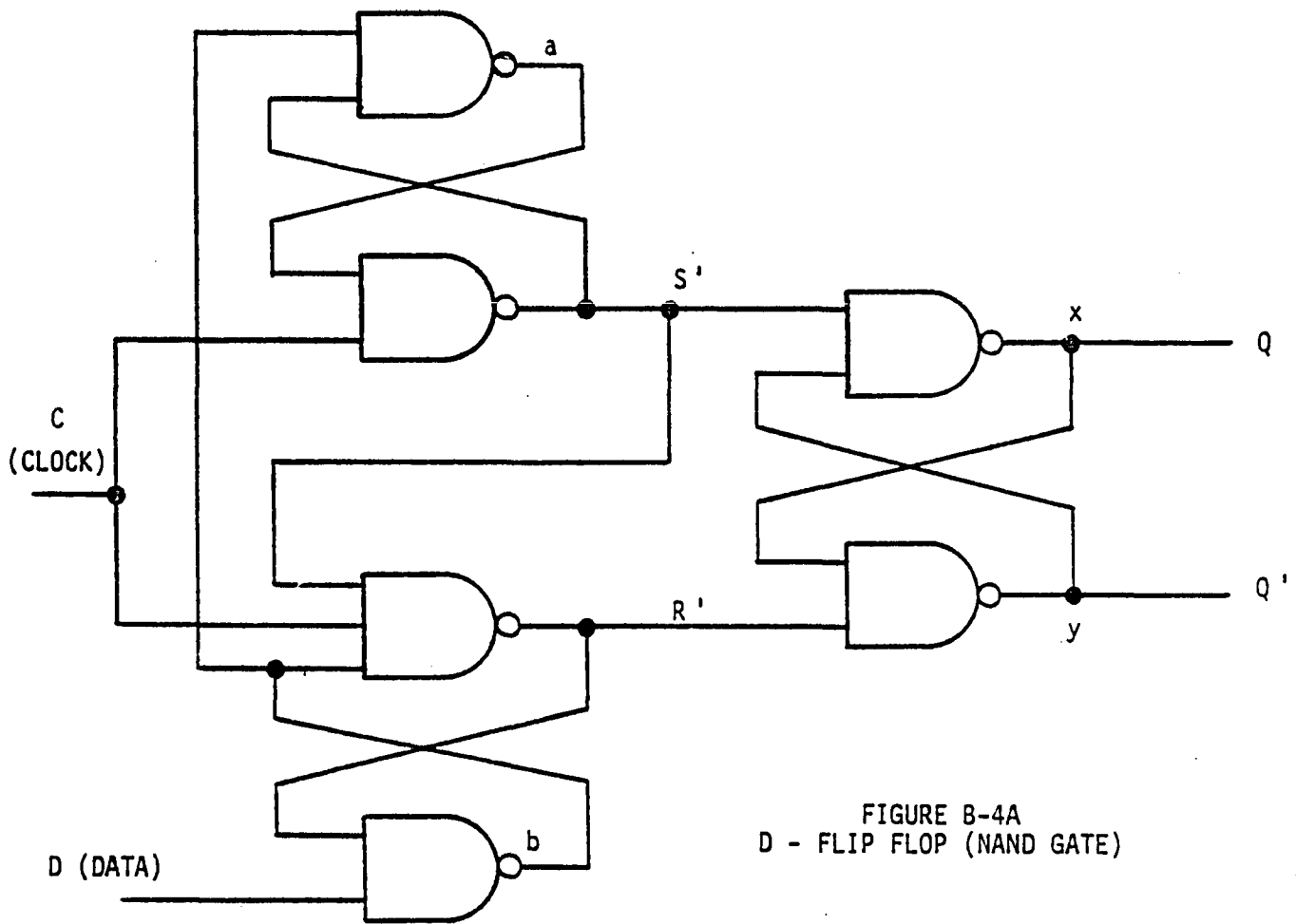
@ t_o		@ t_f		@ t_f	
CLOCK	D	S'	R'	x	y
0(↓)	dc	1	1	Q_0	Q_0'
1(↑)	0	1	0	0	1
1(↑)	1	0	1	1	0

dc = don't care

FIGURE B-4B   TRUTH TABLE D - FLIP FLOP

FIGURE B-4   D- FLIP FLOP
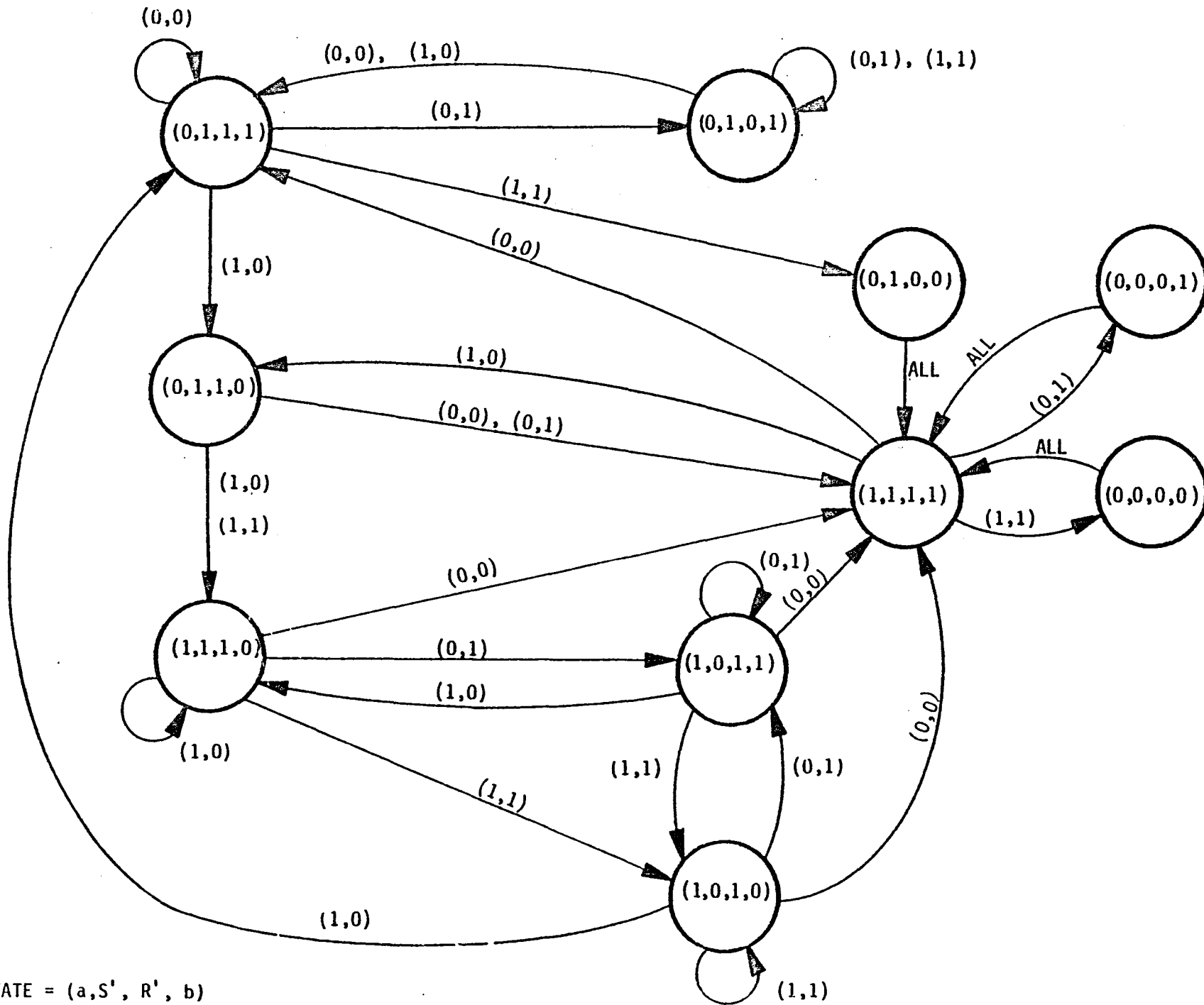
STATE = (a, S', R', b)
TRANSITION = (D, CLOCK)

FIGURE B-5 SATE DIAGRAM D- FLIP FLOP

86
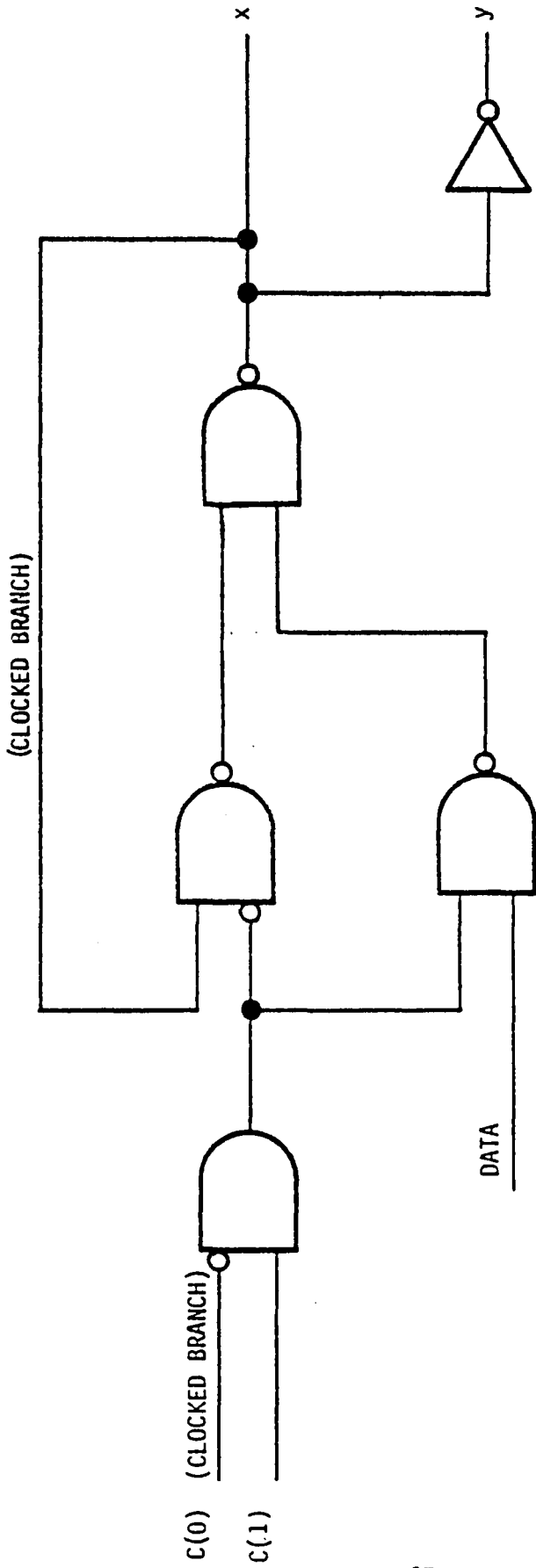
FIGURE B-6   SIMPLIFIED MODEL OF D-FLIP FLOP
(that requires clock transition through zero)

(CLOCKED BRANCH)

CLK

CLK'

x = Q

y = Q'

DATA

88
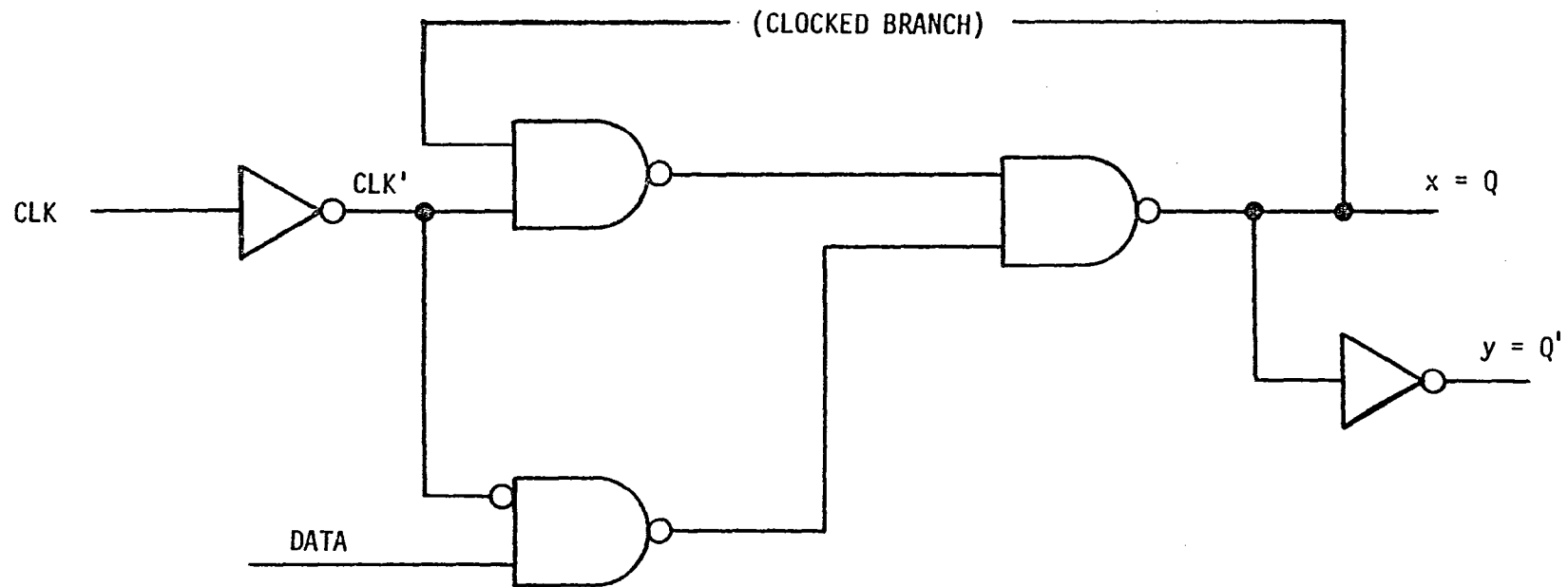
FIGURE B-7 SIMPLIFIED MODEL OF D-FLIP FLOP

(assuming failure-to-transition clock faults are prohibited)
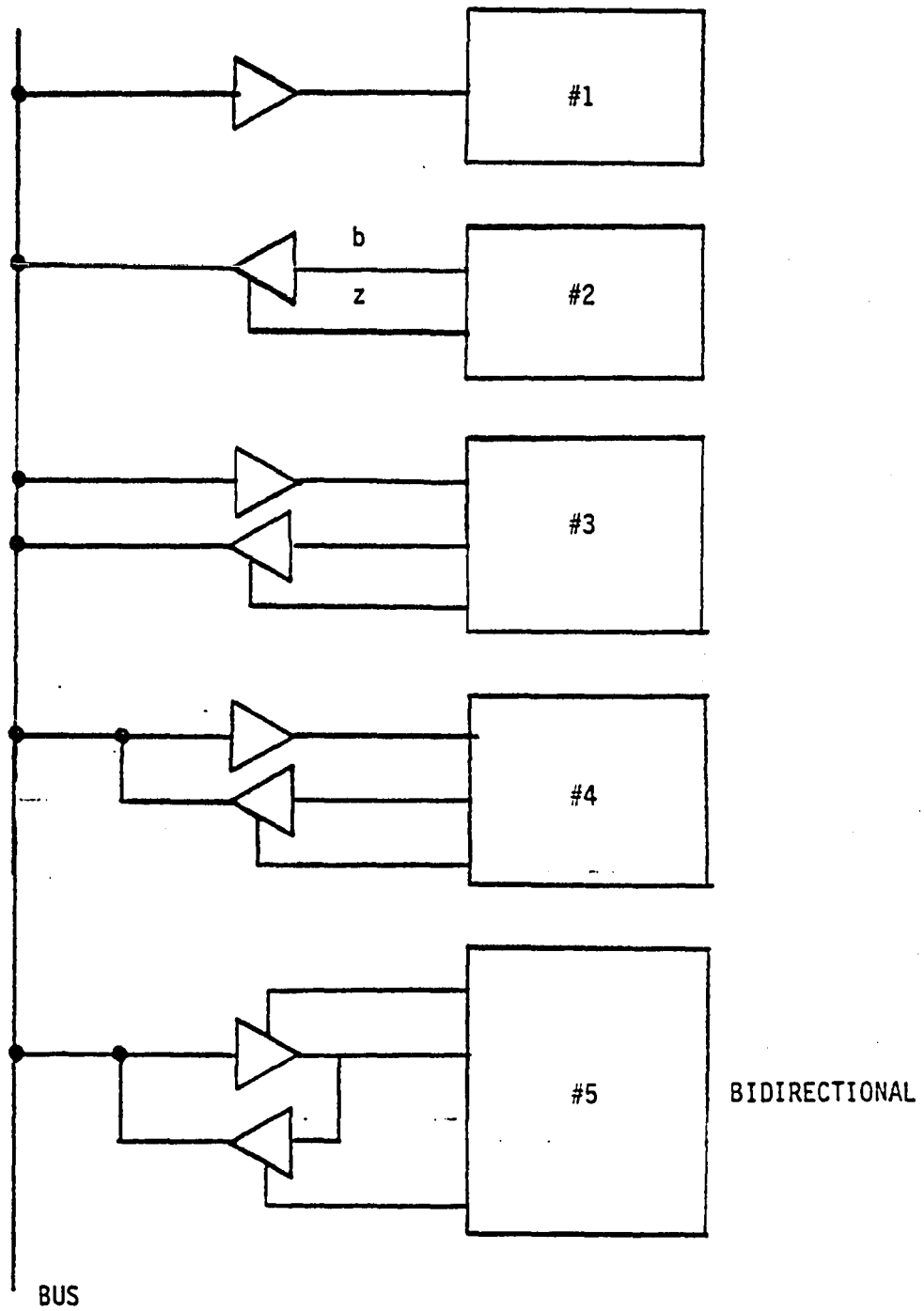
FIGURE B-8   TYPICAL TRISTATE TRANSMITTER/RECEIVER ARRANGEMENTS

BUS

NON-FAULTED
AND GATE

X = STUCK-AT FAULTS
$z(i)$ = IMPEDANCE STATE
$b(i)$ = SIGNAL

$z(1)$
$b(1)$

$z(2)$
$b(2)$

$z(n)$
$b(n)$
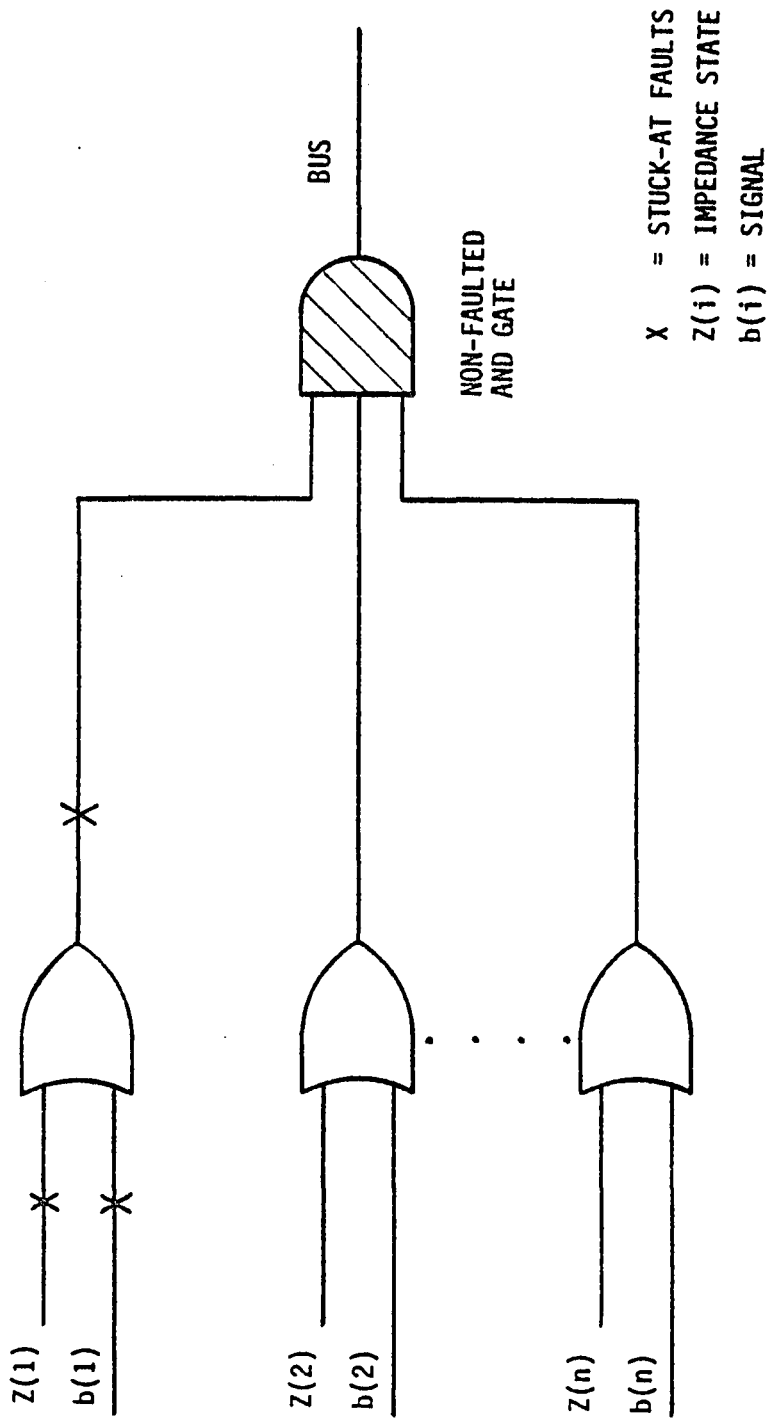
FIGURE B-9

GATE-EQUIVALENT CIRCUIT TRISTATE BUS OF BGLOSS

90
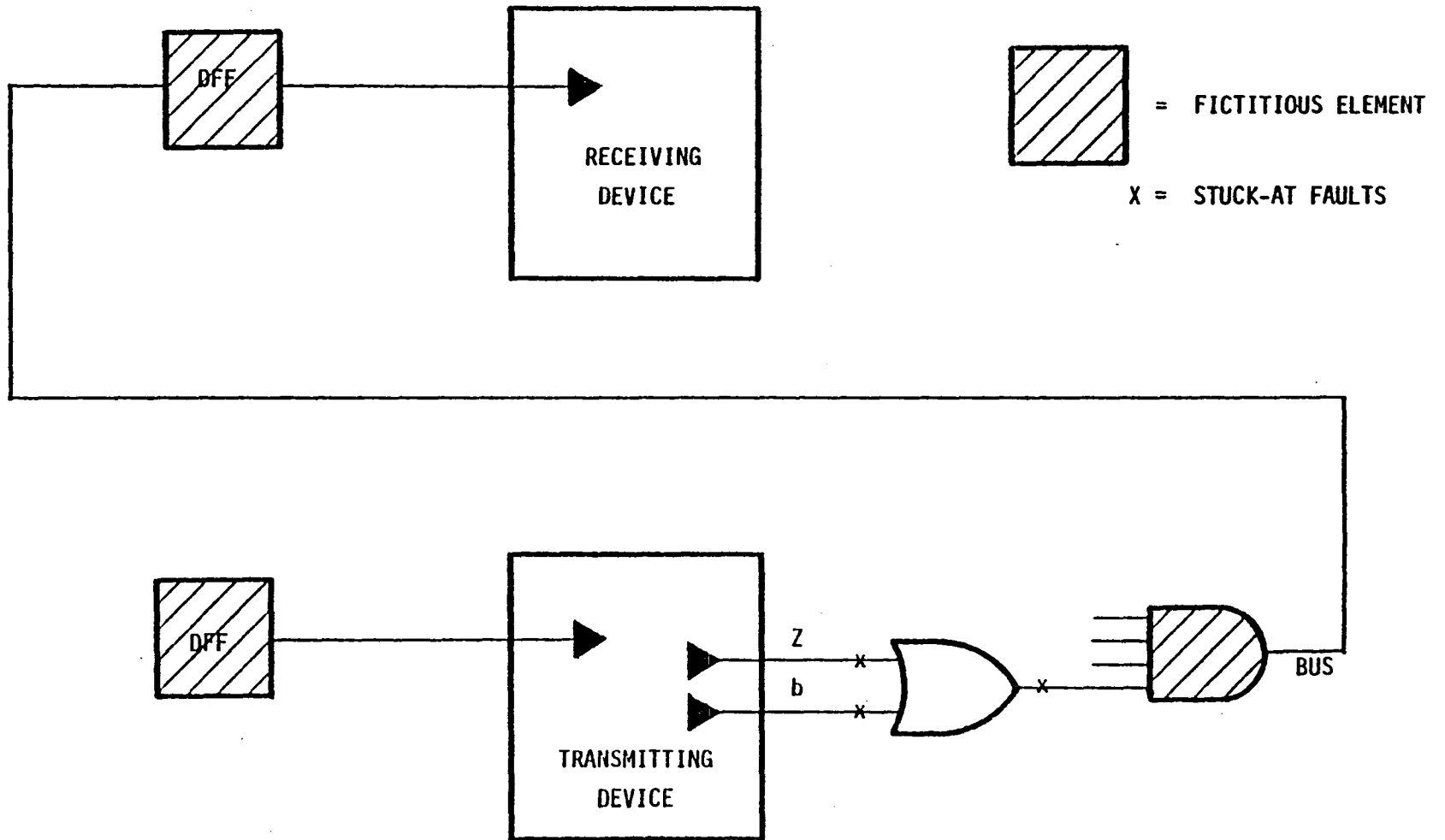
FIGURE B-10   TYPICAL BGLOSS MODEL OF A TRANSMITTER/RECEIVER

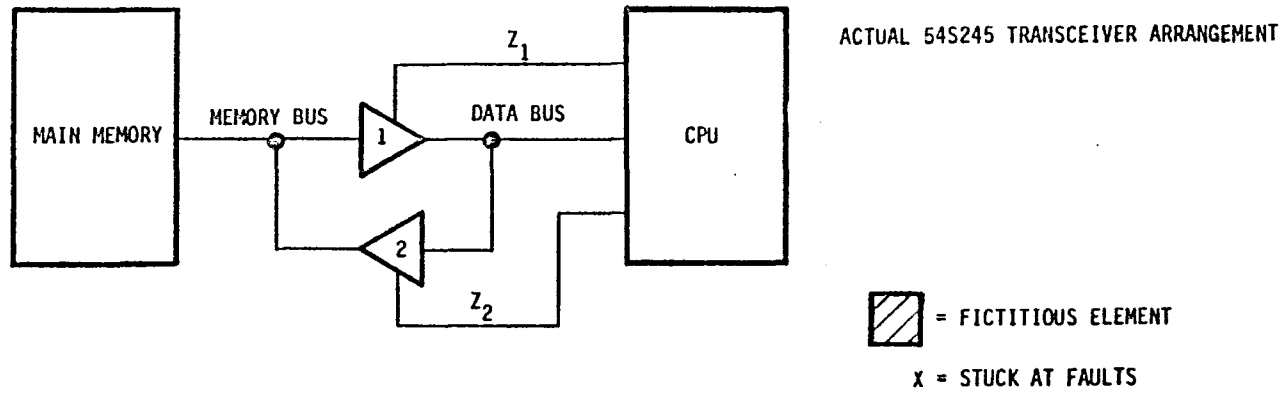ACTUAL 54S245 TRANSCEIVER ARRANGEMENT

Z_1

MAIN MEMORY

MEMORY BUS

1

DATA BUS

CPU

2

Z_2

= FICTITIOUS ELEMENT

X = STUCK AT FAULTS

DFF

MAIN
MEMORY

BGLOSS MODEL OF
54S245 TRANSCEIVER
(MEMORY FAULTS PROHIBITED)

MEMORY
BUS

DFF

CPU

Z_1

1

DATA

BUS

2

MEM
BUS

Z_2
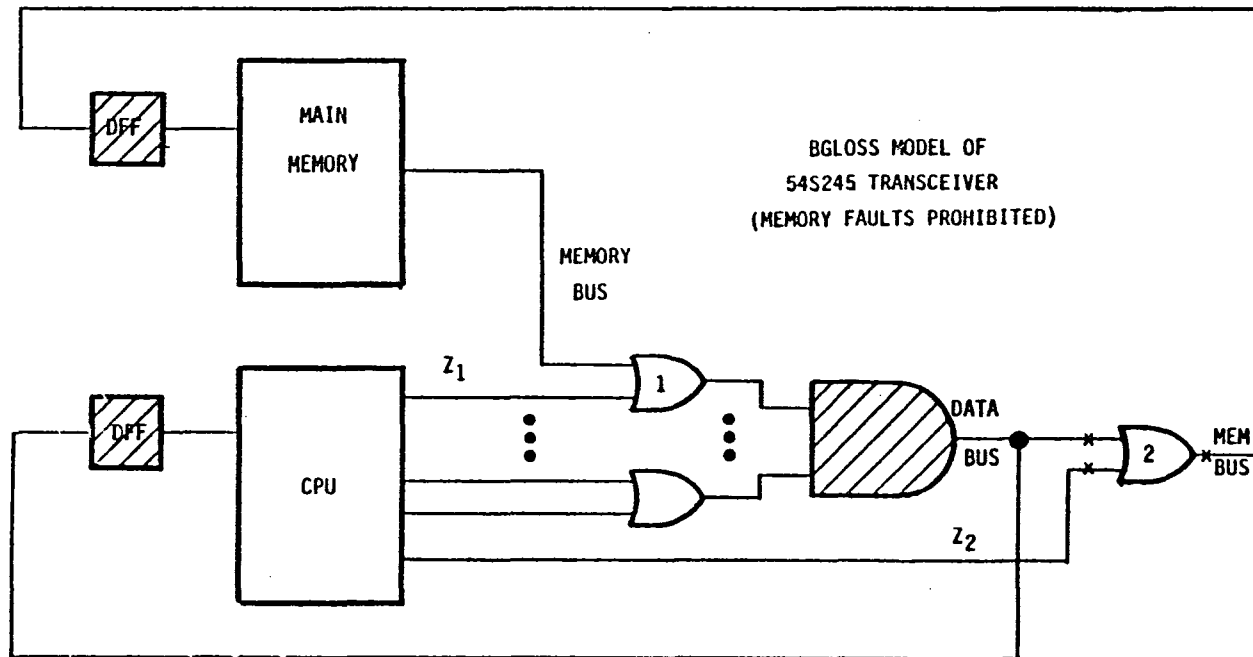
FIGURE B-11   BIDIRECTIONAL TRANSCEIVER MODEL USED IN BGLOSS
(DATA BUS TO MEMORY BUS)
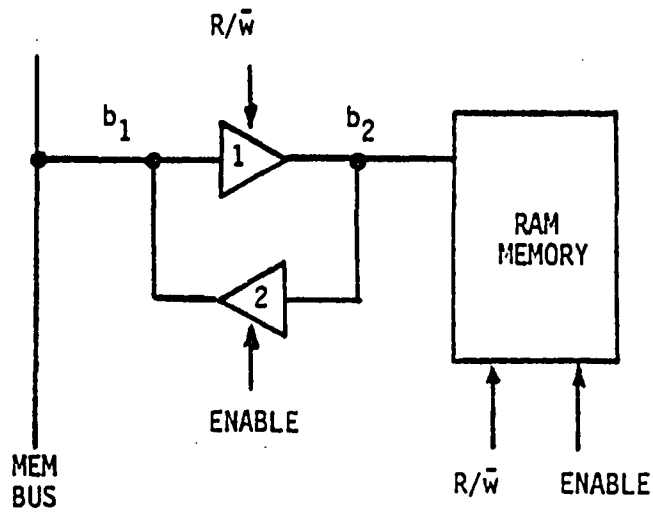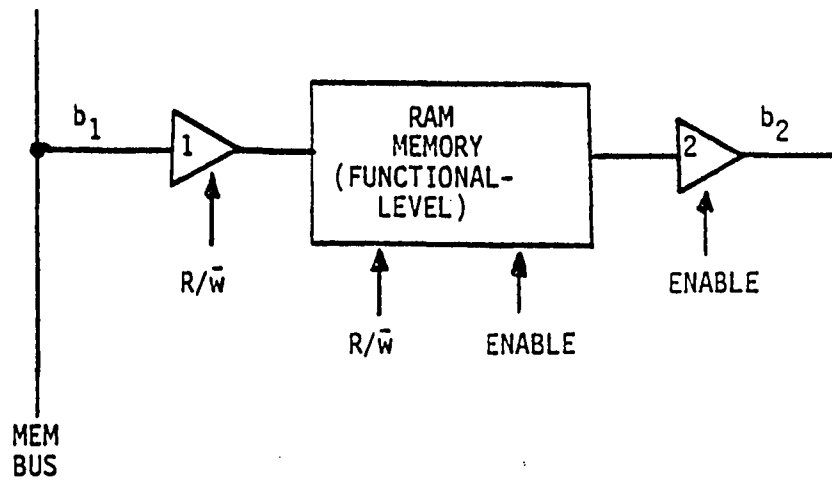
FIGURE B-12A   MEMORY TO MEMORY BUS TRANSCEIVER

FIGURE B-12B   COMBINATIONAL NETWORK REPRESENTATION OF TRANSCEIVER/MEMORY
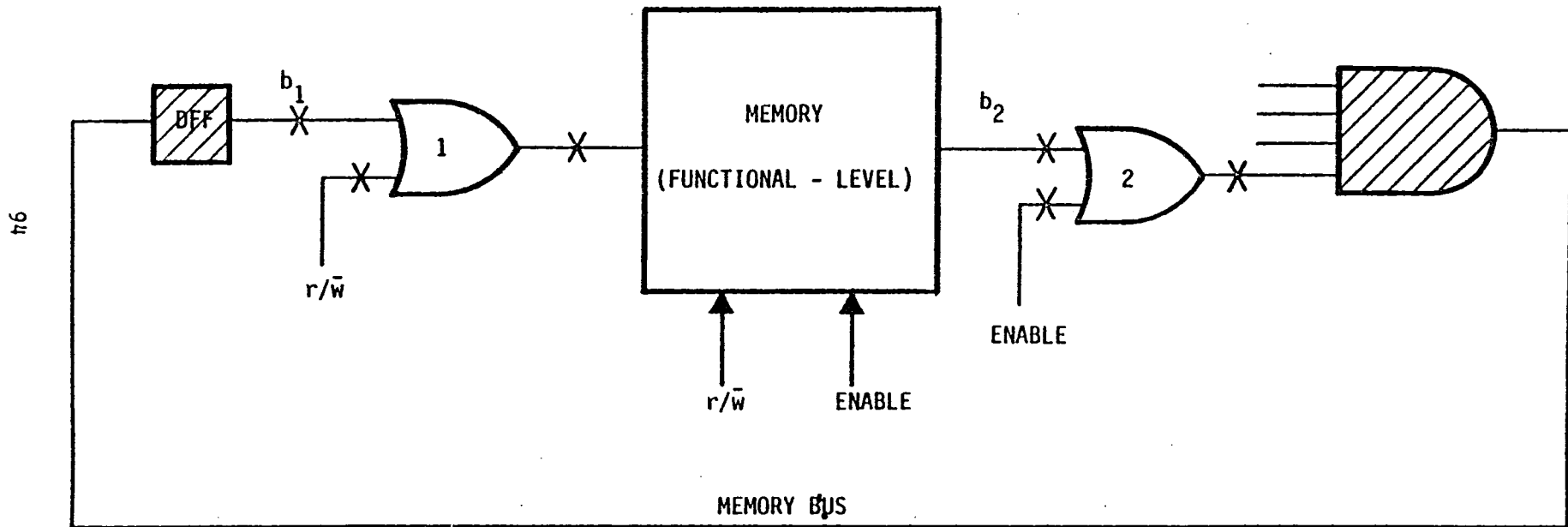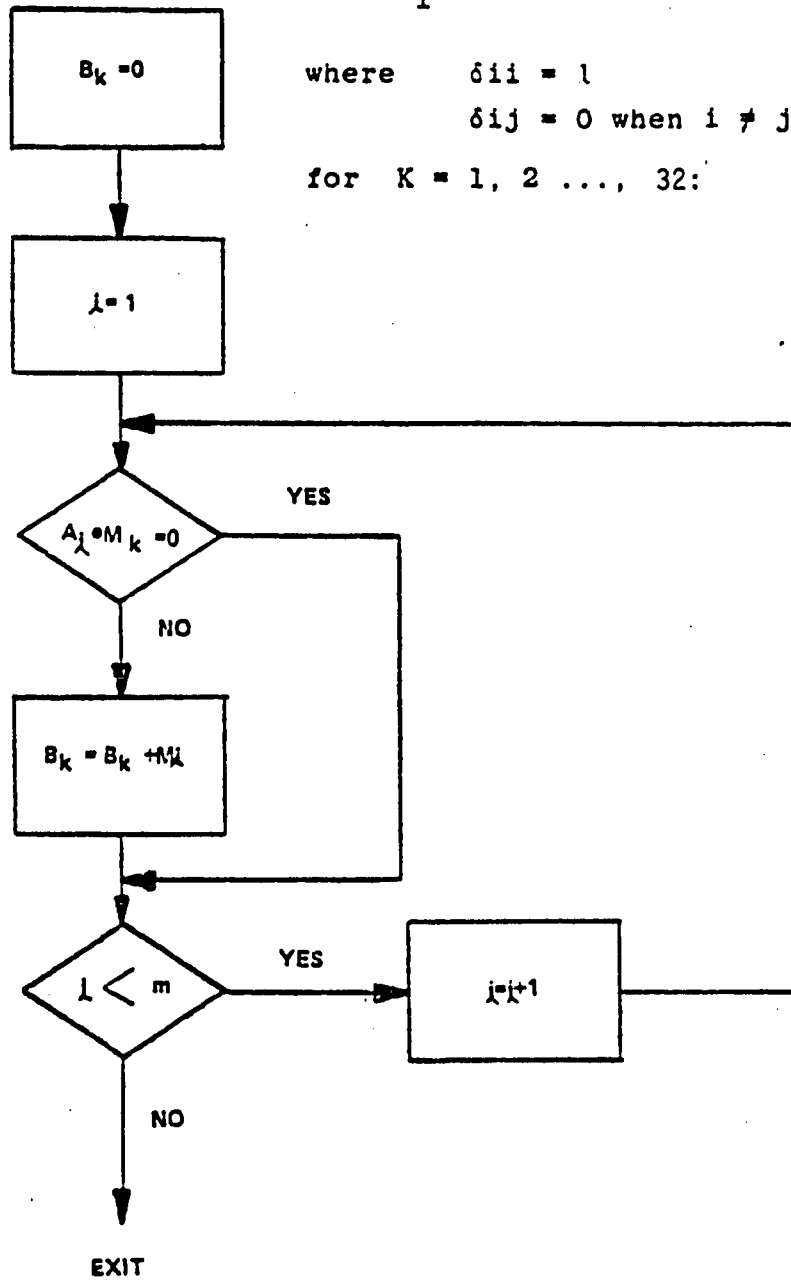
FIGURE B-12   MEMORY TO MEMORY BUS TRANSCEIVER MODEL

FIGURE B-13   PROTOTYPE NETWORK MODEL OF MEMORY TO MEMORY BUS TRANSCEIVER

Set $M_i = (\delta i1, \delta i2, \ldots, \delta i,32)$, $i=1,2, \ldots, 16$

where $\delta ii = 1$

$\delta ij = 0$ when $i \neq j$

for $K = 1, 2 \ldots, 32:$

$B_k = 0$

$i = 1$

$A_i \cdot M_k = 0$ YES

NO

$B_k = B_k + M_i$

$i < m$ YES $i = i + 1$

NO

EXIT

NOTE: "." = logical "AND", "+" = logical "OR"

FIGURE B-14   GATE-LEVEL TO REGISTER-LEVEL CONVERSION ALGORITHM
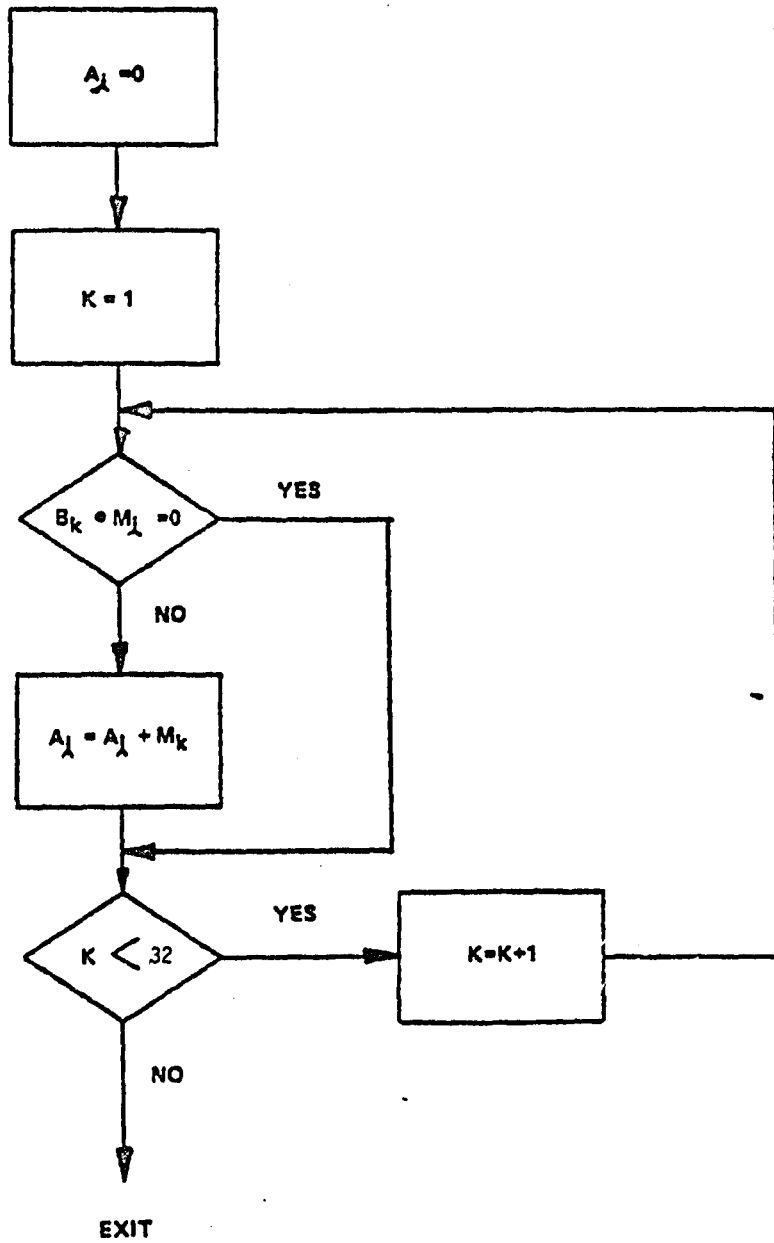
For i = 1,2, ..., 16:

FIGURE B-15 REGISTER-LEVEL TO GATE-LEVEL CONVERSION ALGORITHM

TABLE B-1
EXCITATION TABLE FOR THE D-FLIP FLOP

(a,S',R',b)**	D=0,C=0	D=1,C=0	D=0,C=1	D=1,C=1**
(0,1,1,1)	(0,1,1,1)	(0,1,1,0)	(0,1,0,1)	(0,1,0,0)
(0,1,1,0)	(1,1,1,1)	(1,1,1,0)	(1,1,1,1)	(1,1,1,0)
(1,1,1,0)	(1,1,1,1)	(1,1,1,0)	(1,0,1,1)	(1,0,1,0)
(1,1,1,1)	(0,1,1,1)	(0,1,1,0)	(0,0,0,1)	(0,0,0,0)
(0,1,0,1)	(0,1,1,1)	(0,1,1,1)	(0,1,0,1)	(0,1,0,1)
(0,1,0,0)	(1,1,1,1)	(1,1,1,1)	(1,1,1,1)	(1,1,1,1)
(0,0,0,0)	(1,1,1,1)	(1,1,1,1)	(1,1,1,1)	(1,1,1,1)
(0,0,0,1)	(1,1,1,1)	(1,1,1,1)	(1,1,1,1)	(1,1,1,1)
(1,0,1,0)	(1,1,1,1)	(1,1,1,0)	(1,0,1,1)	(1,0,1,0)
(1,0,1,1)	(1,1,1,1)	(1,1,1,0)	(1,0,1,1)	(1,0,1,0)
(0,0,1,0)*				
(1,0,0,0)*				
(0,0,1,1)*				
(1,0,0,1)*				
(1,1,0,0)*				
(1,1,0,1)*				

* = unreachable state (under non-faulted conditions)
** D = data input, C = Clock, S = Set, R = Reset

97

APPENDIX C
A HYPOTHETICAL SIMULATION

To illustrate the methodology of GGLOSS we will outline the procedures employed by GGLOSS and the User in setting up and executing a hypothetical simulation.

Let it be desired to simulate a non-faulted digital computer such as the BDX-930. A typical computer architecture is shown in Figure C-1. Since it is desirable to describe the procedure in some detail we will concentrate, exclusively, on the Computer Control Unit (CCU). The extrapolation of the procedure to the other devices will be apparent. A block diagram of a typical CCU is shown in Figure C-2.

We will omit a description of the sequence of operations of the CCU. This can be obtained in any text on computer design. In any case it is not necessary for the User to understand exactly, or even approximately, how the CCU performs its function.

User Procedure

Step #1

The User identifies each device of Figure C-2, its inputs and outputs, and interconnections with other devices.

Step #2

The User obtains a logic diagram for each device.

Step #3

The User determines which devices are to be simulated at the gate and functional-levels. [Functional-level devices are memories, registers or any device whose outputs can be defined as an array. To improve simulation speed it is recommended that simple devices, such as matrix decoders, be represented by fictitious, gate-equivalent circuits instead of at the functional-level. This eliminates a pair of parallel/ serial transformations.]

Step #4

The User determines which logic circuits already exist in the library of standard components. Employing a standard format, the User defines any new circuits and stores these in the library. At this point all gate-level and functional-level devices are defined(including the contents

of all memories) and are resident in the library of standard components. Each such device is designated a "part". The User then defines, via the Menu,

1) a Partslist(devices)
2) a Netlist(interconnections)

for the CCU. This defines the network.

Step #5

In general, the system may employ several clock signals, usually of the same frequency but with different phases. The User identifies all devices which are activated by one of these signals, e.g., edge-triggered flip flops.

Step #6

In Step #7 the User will be required to specify when each node of the Prototype Network is to be evaluated. To do this it is necessary to relate the fictitious and true clock periods. This relationship is User-selectable, depending upon the desired number of node evaluations in each true clock period. In our hypothetical simulation we will assume that it is desired to evaluate each node at both the rising and descending edges of a true clock pulse. In this case the true clock period is specified as twice that of the fictitious clock. Referring to Figure C-2, it is seen that the CCU does not contain a true clock oscillator. Consequently, we invented one (shown in Figure C-3). The output of the oscillator is a periodic train of alternating 1's and 0's, corresponding to the rising and descending edges of the true clock pulses.

Step #7

The User now defines the Prototype Network. The rules of Section 3.1 must be followed, e.g.,

1) all external inputs are clocked
2) all loops must contain at least one clocked branch
3) throughput delays are small relative to the true clock cycle (this is assured in a well-designed network).

The Prototype Network model is defined simply by designating which branches are clocked (by the fictitious clock). These branches represent inputs to the true clocked devices. The resultant Prototype Network is shown in Figure C-3. The clocked branches are those denoted by the fictitious D-flip flops.

99

Step #8

The User defines an initialization of the CCU. In practice the CCU would include a power-on external which would correctly initialize the CCU, independently of its prior state. This input was omitted from Figure C-2. Had it been included, it would have been multiplexed with the output of the Instruction Register and gated to the Starting Address Decoder.

Step #9

The User selects the desired output formats, via the Menu.

Treatment of Flip Flops

To improve simulation speed we elect to simulate flip flops by the circuit model of Figure B-7 rather than at the functional-level. Since all feedback paths must be clocked we must associate a fictitious clock with every feedback branch of every flip flop. We use the same clock as in Step #7.

Simulator Procedure

Step #1

The Simulator interrogates the User, via the Menu, to obtain the items enumerated in Section 5. These include the Partslist, Netlist and the clocked branches.

Step #2

The Simulator (Preprocessor) constructs the appropriate network for simulation. It does this by the method described in Section 3.1, i.e., the clocked branches are cut and the resultant combinational network is p-ordered. The cut network is shown in Figure C-4.

Step #3

Since the devices are non-faulted the Simulator is now ready to begin the simulation.
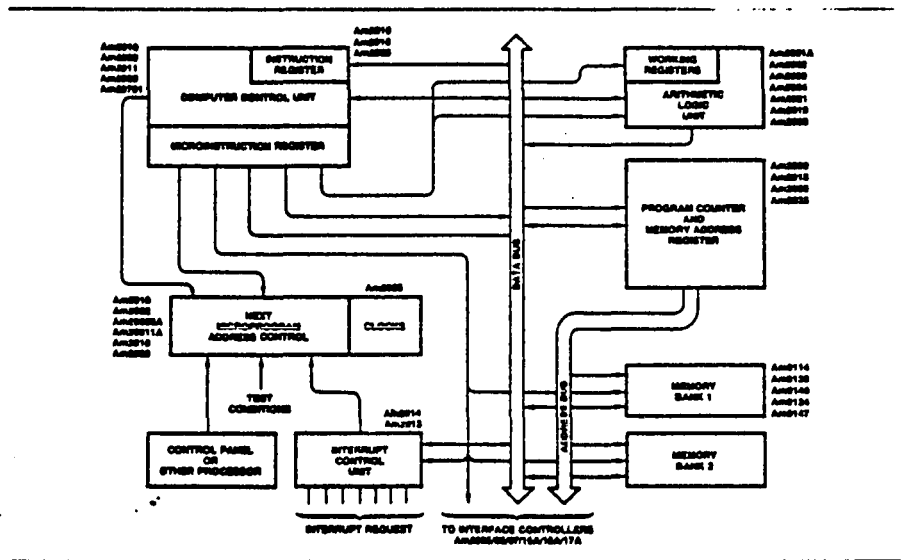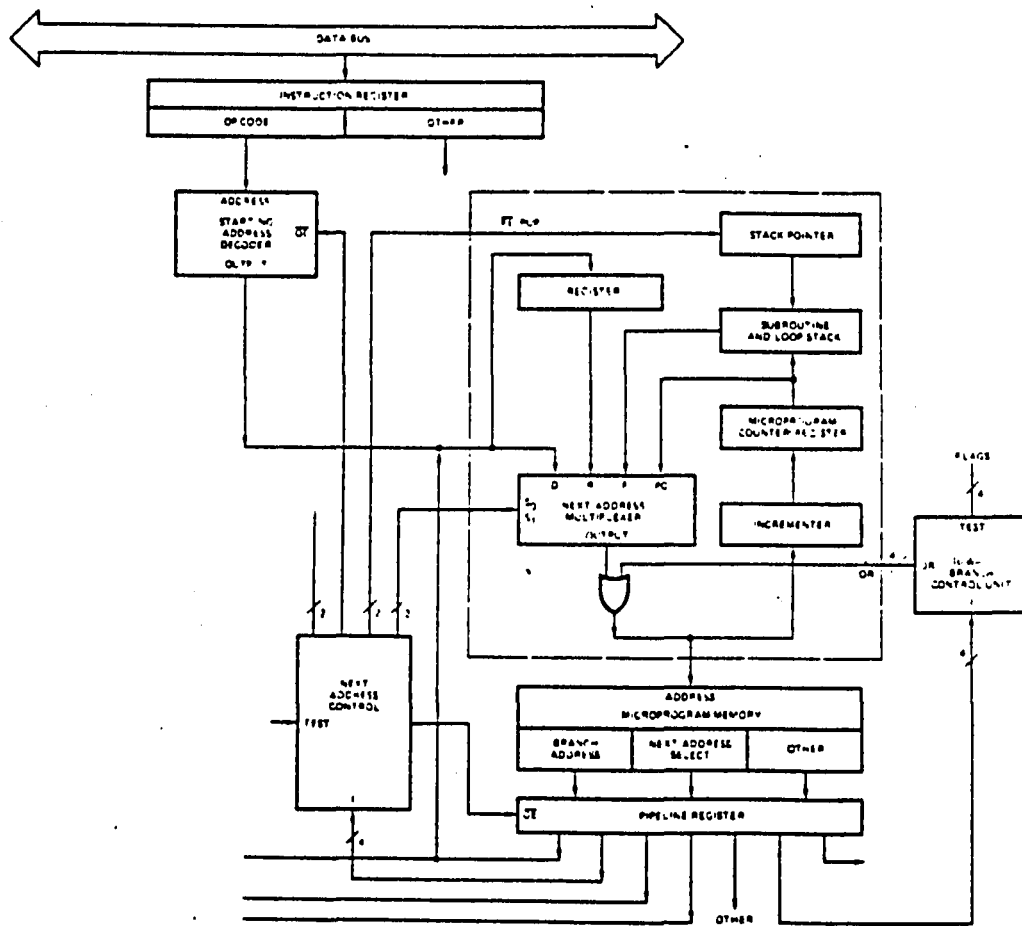

ER130

FIGURE C-1   A GENERALIZED COMPUTER ARCHITECTURE
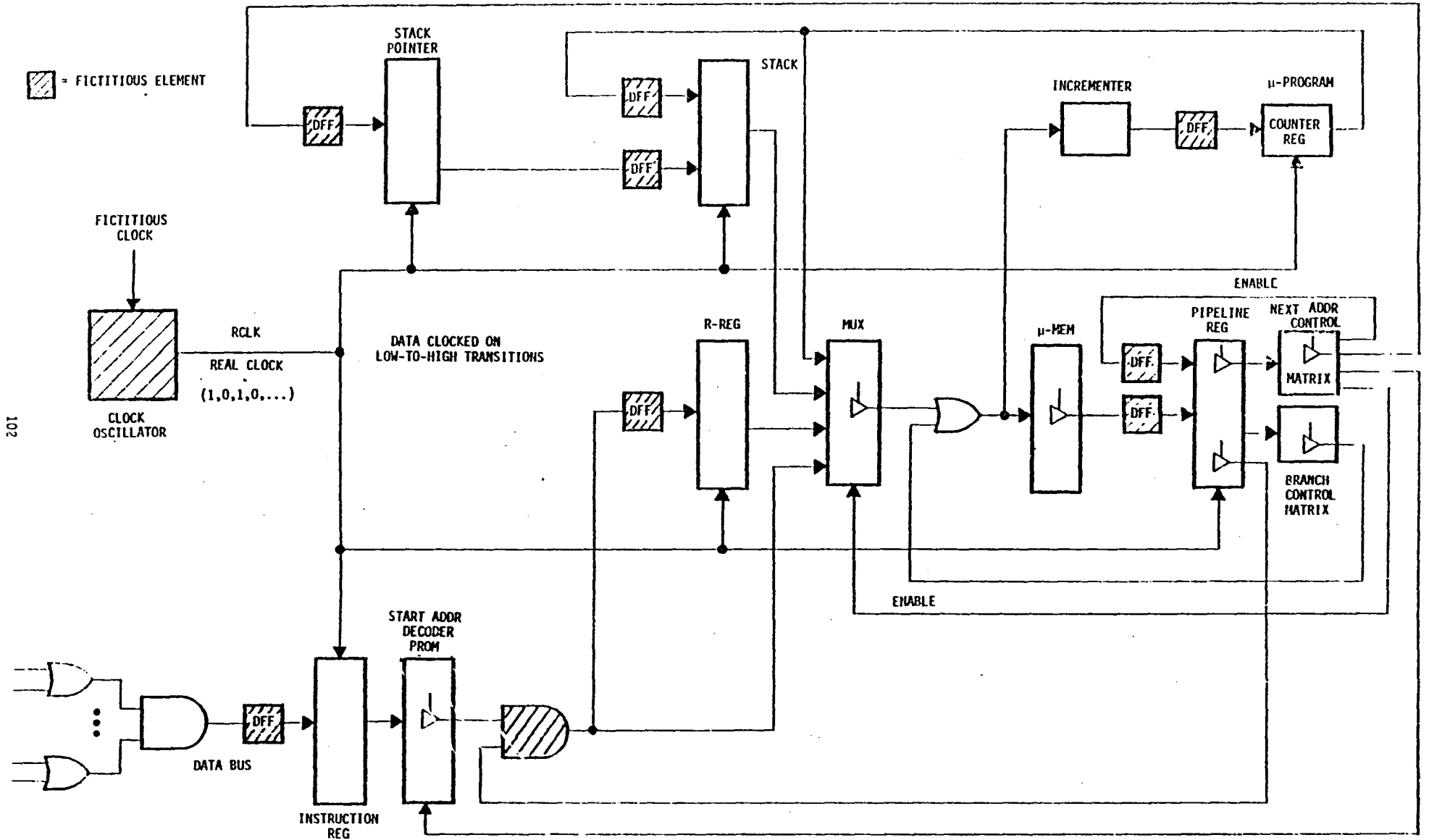

FIGURE C-2   A TYPICAL COMPUTER CONTROL UNIT
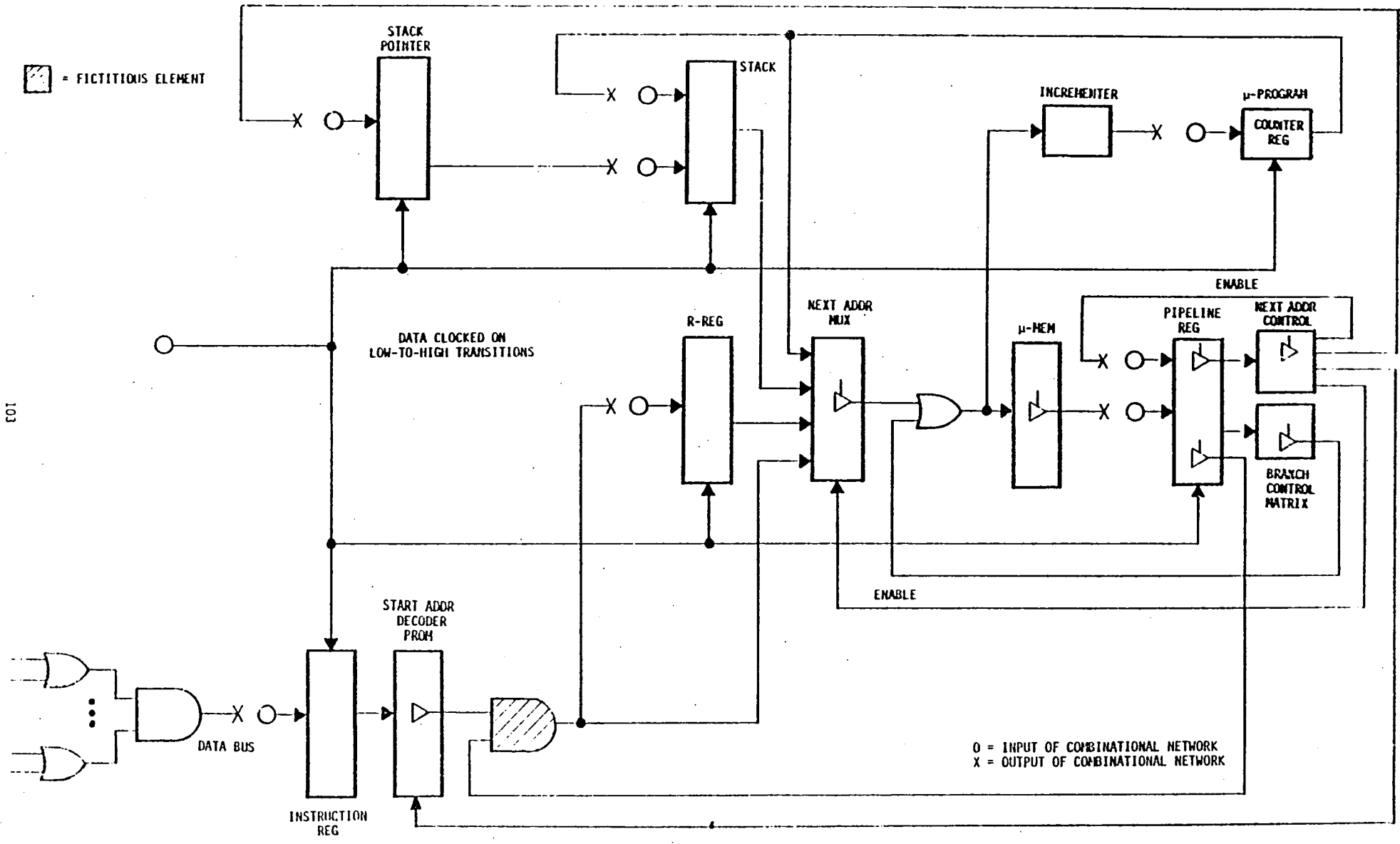
101

FIGURE C-3 PROTO.YPE NETWORK MODEL TYPICAL COMPUTER CONTROL UNIT

102

FIGURE C-4   CONSTRUCTION OF COMBINATIONAL NETWORKS WITHIN COMPOUND NODES

End of Document