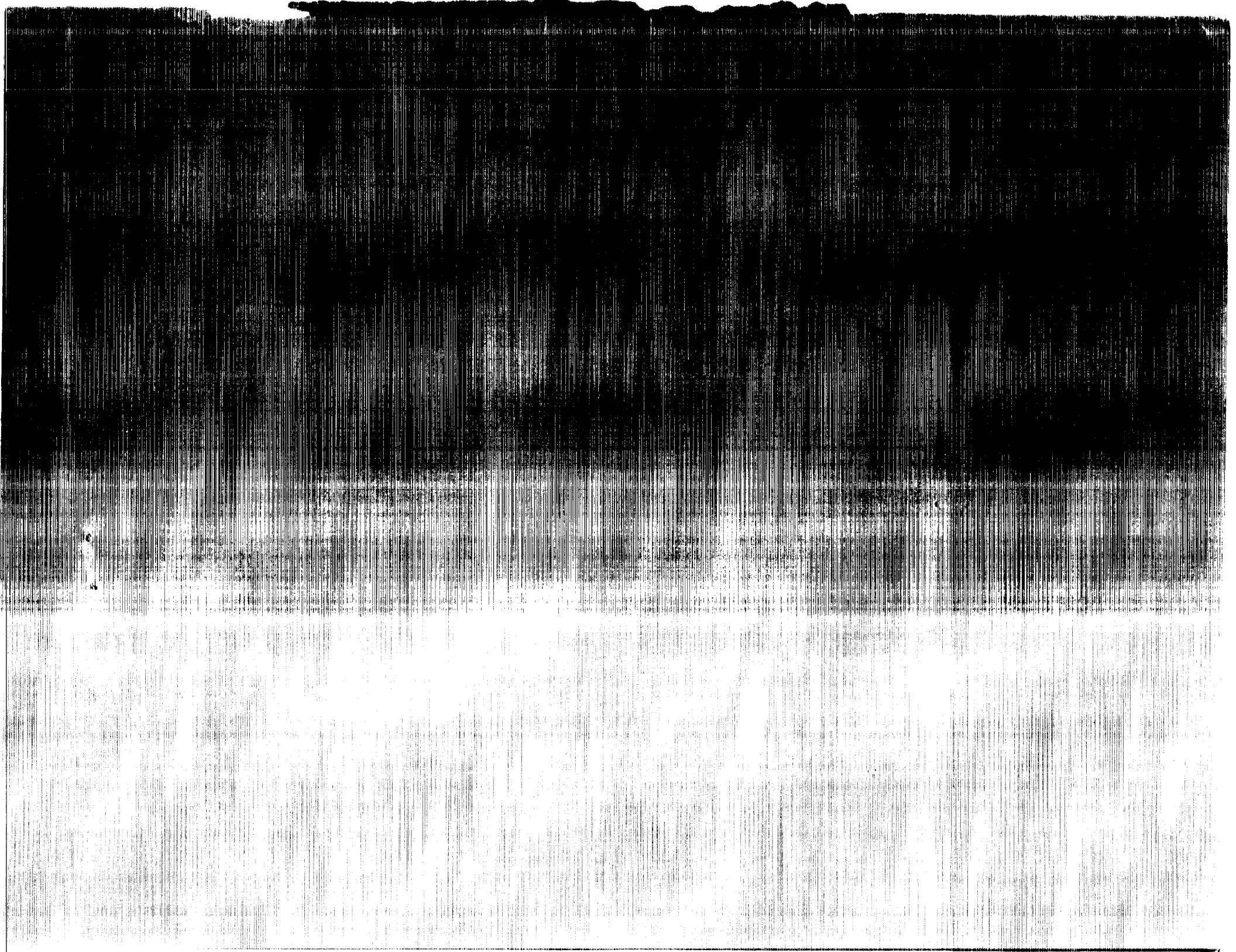
NASA Technical Memorandum 4027

On the Diagnostic Emulation Technique and Its Use in the AIRLAB

Gerard E. Migneault

OCTOBER 1988



NASA Technical Memorandum 4027

On the Diagnostic Emulation Technique and Its Use in the AIRLAB

Gerard E. Migneault
Langley Research Center
Hampton, Virginia

National Aeronautics
and Space Administration

Scientific and Technical
Information Division

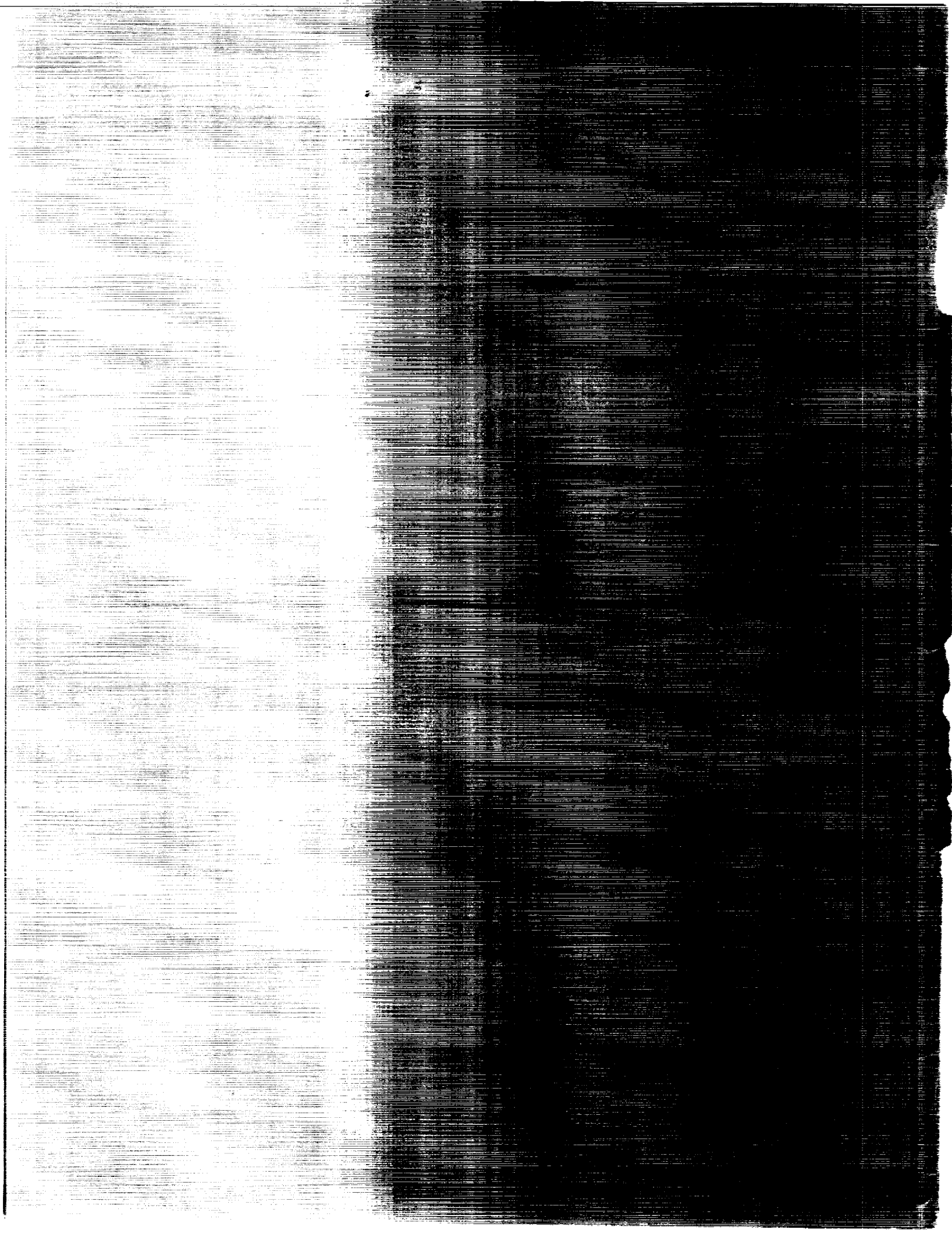
1988

NASA Technical Memorandum 4027

On the Diagnostic Emulation Technique and Its Use in the AIRLAB

Gerard E. Migneault

OCTOBER 1988



NASA Technical Memorandum 4027

On the Diagnostic Emulation Technique and Its Use in the AIRLAB

Gerard E. Migneault
Langley Research Center
Hampton, Virginia

National Aeronautics
and Space Administration

Scientific and Technical
Information Division

1988

Preface

This document is intended primarily as an aid for understanding and judging the relevance of the "diagnostic emulation technique," developed at the Langley Research Center, to studies of highly reliable, digital computing systems for aircraft. Therefore, the document contains a short review of the need for and the use of the technique as well as an explanation of its principle of operation and its implementation. However, details that would be needed for operational control or modification of existing versions of the technique are described in NASA CR-178391 by Becher (1987).

The Introductory section contains a description of the need (and origin of the need) for an emulation technique in the Langley Avionics Integration Research Lab (AIRLAB). The section on *The Technique* contains a concise description of a conceptual algorithm for emulating a computational process, a more lengthy description of the actual implementation of the conceptual algorithm, and a summation of the implemented features of the technique. Appendixes A to D contain details of the technique.

The document has been organized to be read at the following levels of detail:

1. The main points of the document can be found in the main text alone (presented in the left column of each page), without reference to the footnotes or appendixes.

2. The footnotes to the main text (presented in the right column of each page) are intended to either expand upon or clarify a point in the main text, but not to provide more detail.

3. The appendixes explain how and why specific gate logic algorithms work.

4. There are footnotes to the appendixes (presented at the bottoms of the pages) describing more specific details.

Summary

This document is intended primarily as an aid for understanding and judging the relevance of the "diagnostic emulation technique," developed at the Langley Research Center, to studies of highly reliable, digital computing systems for aircraft. Therefore, the document contains a short review of the need for and the use of the technique as well as an explanation of its principle of operation and its implementation. However, details that would be needed for operational control or modification of existing versions of the technique are described in NASA CR-178391 by Becher (1987).

Introduction

The validation of highly reliable, fault-tolerant computers is an important and, at the very least, difficult task in the development of digital computing systems for aircraft. The reasons are evident. On the one hand, demand for safety imposes an elevated, minimally acceptable level of reliability on any system whose failure could cause fatalities; on the other hand, uncertainty about the relationship between malfunctions of an aircraft's computing system and catastrophes involving the aircraft makes any precise specification of a minimally acceptable level of reliability problematic. To be generally acceptable, any posited, minimum level of reliability of a computing system must be conservatively elevated to such an extent that the reliability of currently available components is inadequate for constructing conventional (i.e., fault-intolerant) computing systems. Consequently, a need arises naturally for computing systems designed to tolerate their own (i.e., internal) faults and malfunctions.[1]

However, fault tolerance does not obviate the need for assessing reliability. "Fault tolerant" is but a cachet for systems designed to continue functioning after some internal (and necessarily superfluous) components cease functioning properly; that is, the systems are designed to contain and make advantageous use of redundancies to provide alternative means of performing their functions.[2] With limited resources in a limited interval of time there can be only a limited number of available alternatives. Clearly, when the set of alternatives, each fault intolerant, has been depleted, the system fails.[3] Thus, and also clearly, a fault-tolerant system has an associated level of reliability that must be evaluated to ensure that it equals at least the minimally acceptable level.

There is also an added complication that accompanies the use of fault tolerance. A fault-tolerant system could be less reliable than straightforward

[1] In the literature, the value $1 - 10^{-9}$ has become a somewhat *de facto* standard for the minimum acceptable level of reliability of a civil aircraft's computing system (for a flight duration of 10 hours) if it performs tasks crucial to flight. For military aircraft, requirements for reliability of computing systems hover about the value $1 - 10^{-7}$ for a mission. However, the need for innovative designs for systems does not arise only from such extreme requirements; less stringent levels of reliability would still be beyond the sure reach of computing systems that are intolerant of faults.

To be sure, some critics consider the attainment of *ultrareliability* to be infeasible at best and the resulting schemes for fault-tolerant computers as "redundancy "run amok" " (e.g., see pp. 220–222 of Goldberg 1981). For an abbreviated rationale for the value $1 - 10^{-9}$, see Migneault (1980).

[2] Redundancy can be physical, temporal, logical, or any combination of these three—physical, as in parallel, replicated modules; temporal, as in repeated attempts of the same computational process; and logical, as in dissimilar algorithms for the same functions or processes, whether they are implemented physically, temporally, or in combination. Each form of redundancy addresses some causes of failure better than others.

Of course, once elements are called into play to tolerate faults, they can no longer be considered redundant; that is, here the meaning of redundancy is conditional on the absence or presence of faults. For an exposition of fault tolerance—one which, however, lacks a satisfactory discussion of the relationship to reliability—see Anderson and Lee (1981).

[3] Casually stated, the probability that a fault-tolerant system fails because of the depletion of its set of alternatives in a stated interval of time is equal to the nonzero product of the probabilities of failure of the separate, needed, redundant resources, with the assumption of uncorrelated failures. ("Casually stated," because the word failure is used loosely here to encompass both an event (i.e., a component becoming a failed component at a particular instant) and a state (i.e., a particular, failed condition in which the component is assumed to operate only improperly). More correctly stated, however, a failure is only an event. Moreover, in general a failed state does not imply necessarily improper behavior; it is, rather, the unfailed state that implies necessarily proper behavior.) Note also that the probability of failure of an alternative includes the possibility that its function is not performed within specified intervals of time, even in the absence of any tangible failures. That is, consistent with footnote 2, an interval of time can be considered to be effectively a consumable resource.

consideration of its set of redundant resources might reveal because, in addition to the redundant resources, a fault-tolerant system contains a means (here called "the mechanism") for composing the system's output from the outputs of its parts, whether or not some are faulty. In effect, the mechanism embodies the intelligence by which redundancy is put to effective use to provide the fault tolerance that is intended to increase reliability. Yet, the mechanism also contributes to lessened reliability in two ways; one of these is difficult to evaluate. First, there is the conventional effect upon reliability of an added, fault-intolerant component, that is, the mechanism. Evaluation of this effect poses no different challenge from that due to any other component of the system; indeed, fault tolerance can be designed into the mechanism also.[4] More difficult to evaluate, however, is the probability that a mechanism's design is flawed, so that in some situations it will not respond properly to some failures of components despite the availability of functioning, alternative resources. That is, what is at issue here is the possibility that there are modes of failure that have been overlooked or incompletely understood by designers of the system. But how does one estimate the probability of things unthought of?[5]

Ordinarily (i.e., for pedestrian, fault-intolerant systems for which reliability requirements are sufficiently benign), enough testing of the systems can be performed without knowledge of internal structure (i.e., "black-box" testing) to justify an assumption that the effect of any residual design flaws on reliability is negligible compared with chance failures resulting from degradation of components. Thus, ordinarily, the problem of design flaws is handled by the elimination of such flaws as they are discovered—that is, by the removal of all but the (assumed) statistically insignificant, unobserved flaws from the system. If, however, they are to be even only approximately achieved, the elevated levels of reliability required, the *raisons d'être*, of fault-tolerant systems preclude the possibility of generating enough data by black-box testing of systems to justify the same assumption with the same credibility, with the same statistical methods.[6]

As a substitute for sufficient, complete, and exhaustive—hence, infeasible—testing of such systems, more sophisticated methods of analysis are used, namely, techniques of modelling and analyzing the behavior of complex systems as stochastic processes. Models are constructed to represent the behavior of systems by an identification of the possible, significant states systems could assume and by a similar identification of significant transitions among the states. (In particular, transitions representing

[4]Here is an instance of what might be called "redundancy run amok" (see footnote 1), for redundancy within the mechanism appears to call for either a recursively functioning mechanism or a meta-mechanism overseeing use of the redundancy within the mechanism—and where is fault tolerance to end? For a resolution, see the use of local and global executive functions in the SIFT (Software Implemented Fault-Tolerant) computer system described by Wensley et al. (1978).

[5]A design flaw in the mechanism is more pernicious than a simple, latent fault that, be it a design flaw or a physically degraded component, lies dormant until an unlikely situation awakens it. Such latent faults can by chance surface effectively simultaneously with other faults and become part of a simultaneous-failure (i.e., multifailure) event—but not necessarily. Design flaws in the mechanism, however, belong to the more special class of latent faults that become active when, and only when, certain activating failures occur; therefore, such flaws necessarily are part of multifailure events—events for which, by the nature of the design flaws, no fault-tolerating mechanism has been specifically designed. Thus, although the rates of occurrence of failures may have been estimated and turned into estimates of the rate of depletion of viable alternatives, estimates of the probability that a fault-tolerant system will continue to function after simple failures remain problematic and involve an additional analysis of a conditional probability called a coverage function.

[6]A test rig containing 1000 independently operating, *highly reliable* fault-tolerant systems serviced approximately every 10 hours to replace failed components can be expected to function for decades (the number depending upon the specific reliability criterion) before even the first system failure will be seen. Clearly, neither such a number of systems nor the time on test would be economically feasible, and any feasible amount of testing would be inconclusive—unless, of course, the systems failed miserably.

malfunctions of the fault-tolerance mechanisms can be included). The stochastic nature of a system's behavior due to the chance presence or absence of faults is then represented in the form of parametric, functional expressions associated with the (conditional) probabilities of the various transitions. One benefit gained by the use of the models is evident. Assessment of a systems's reliability is reduced to the solution of mathematical equations implied by the corresponding model[7]—but only after a leap of faith has been made. One must believe that a model faithfully expresses the stochastic nature of the system in question, for although precise statements can be made about the accuracy and precision of the mathematical techniques, the credibility of any reliability assessment hangs upon the inaccuracies and imprecisions of the models.

Thus, the determination of the inaccuracies and imprecisions of a model becomes an important task to which testing and experimentation can (indeed must) be applied. Now, however, testing can be selectively focused to obtain, verify, or clarify knowledge of those states, transitions, and associated functional expressions that are most significant or uncertain in the model. The nature and number of faults to be considered in the analysis of highly reliable systems, however, still make testing on actual systems impractical. Consequently, it is convenient to use abstract representations, such as computer simulations, as surrogates for the real systems.[8] Clearly it is advantageous that a surrogate truly represent the internal structure of the system being analyzed, for then, if instrumented enough, a surrogate can serve not only as a generator of data from which the parameters of the functional expressions can be estimated but also as a serendipitous means of unearthing unanticipated behavior (i.e., a means of finding unforeseen transitions not included in the stochastic process model).

Some History of This Implementation

The core of the technique, the gate logic level algorithm described in the following sections, was first implemented in 1976 as a computer program in FORTRAN on mainframe computers in order to verify (and demonstrate) the correctness of the algorithm in emulating a complete processor running its own program. A (hypothetical) 16-bit microprocessor was specially defined to play the role of a computer being emulated, and a program was defined and coded in the instruction language of the hypothetical microprocessor. After the successful verification of the algorithm, the computer program and the hypothetical microprocessor were immediately used to perform a small pilot experiment in "stuck-at" fault detection.[9] Subsequently, the feasibility of

[7]CARE III and SURE, for example, are computer codes developed for NASA for analyzing a large class of such models. See Bavuso and Petersen (1985) or Butler (1984).

[8]There are two obvious advantages to the use of *abstract* surrogates: they involve no physically destructive failure actions, and they can be automated on computers (and thus require less setup time between failure cases—possibly the most time-consuming aspect of laboratory experimentation). The disadvantage, of course, is that they require an additional level of verification to be sure that they are truly surrogates for the systems under study.

[9]For a description of the experiment and its results, see Nagel (1978).

speeding up the algorithm by the use of a horizontally microprogrammed computer was examined in a contractor facility having a horizontally microprogrammable computer, and a survey was made of candidate microprogrammable computer systems.[10]

The technique has also been used to emulate the CPU of a Bendix BDX-930 processor,[11] the processor used in the SIFT system. (See Wensley et al. 1978.) As the emulation was an in-house project to evaluate implementation of the technique on the horizontally microprogrammable computer (a Nanodata Computer Corporation QM-1a computer) at the Langley Research Center (LaRC) rather than a study of the BDX-930 processor, it was not documented for public dissemination. The project demonstrated that the version of the emulation technique implemented on the horizontally microprogrammable computer is about 35 to 40 times faster than an equivalent version implemented on a Digital Equipment Corporation VAX-11/750 computer (the general purpose computer in AIRLAB) and is approximately 2×10^4 times slower than the actual BDX-930 processor.

The version of the emulation technique on the VAX-11/750 computer has since been used in a continuing university grant study of gate-logic fault behavior, a study actually performed by graduate students using the technique remotely.[12] Recently, the technique has been used to evaluate the self-diagnostic capability of the fault-tolerant voter module of a fault-tolerant processor design study.[13]

The Technique

General Concept

The computational process in a logic network can be considered to be a process of successive, related perturbations in a medium that is almost in equilibrium.[14] As a result, the emulation task can be viewed as a task of generating the successive states of such a perturbation process. This view has two useful properties. First, since elements that remain in equilibrium at an instant in a perturbed medium are not active components of the perturbation process at that same instant, an algorithmic representation of the process is an efficient scheme for simulating behavior of a network—if most of the network's pattern of signals is in equilibrium at most instants of time, as is usually true for a logic network. Second, since the data anomalies caused by external agents and the effects of physical failures also appear as perturbations (albeit seemingly spontaneous from the viewpoint of a computational process[15] and, in the case of physical failures, more precisely considered as meta-perturbations[16]), an algorithmic representation of a

[10] The feasibility study and survey are documented in Martin Marietta (1981). The survey of candidate microprogrammable computer systems appeared in the form of an interim report several years earlier and served as a basis for procurement of the microprogrammable computer used at the Langley Research Center (LaRC). The interim report appears as attachment 1 of Martin Marietta (1981).

[11] The BDX-930 CPU contains 3000 to 4000 gate equivalent devices. The gate-logic description of the processor was obtained from Bendix Corp. and is documented in Swern and McGough (1982). Of course, the gate level description is now also available directly from the computer files that serve to input to the emulation technique.

[12] Initially the study concentrated upon the error propagation from the gate to the pin level on a chip. The study is described in Lomelino and Iyer (1986). In the continuing study, the emulation technique was interfaced with a circuit-level simulation in order to examine error behavior in finer detail.

[13] The analysis of the self-diagnostic capability was done at Research Triangle Institute and is documented in Baker, Mangum, and Scheper (1988).

[14] The notion that computation in a logic network is analogous to a perturbation process comes from the following observations. First, it is possible to describe a logic network in terms of elementary devices (e.g., logic gates or transistors) which constantly (attempt to) maintain an output signal consistent, in some prescribed manner, with their input signal(s). That is, each device acts as a local equilibrium-restoring force. Moreover, there is some time delay, however small, between the occurrence of a change in an input signal and a related change to an output signal—in light of which the value of a device's output signal may be considered to be a reaction after a discrete interval of time to a perceived inconsistency between its output and input signals. Consequently, a pattern of output signals of the devices in a network is clearly in equilibrium when the output signal of each (and every) device is consistent with its input signal(s). Furthermore, in the absence of seemingly spontaneous effects or arbitrary changes to signals caused by external agents or devices, since every input signal is an output signal of some device, there is no mechanism to disturb equilibrium. Second, the notion of computation within a network is incompatible with the notion of undisturbed equilibrium. Therefore, computation implies that there must be at least some local instance(s) of disequilibrium (i.e., inconsistency) somewhere in the pattern of signals. However, as noted, the nature of the elementary devices is to react to every local instance of inconsistency, eliminating the inconsistency but necessarily further perturbing the pattern of signals in the process. Thus, computation proceeds as a process of successive, related perturbations about an elusive condition of equilibrium.

[15] While the term "spontaneity" of perturbations due to failures and anomalies implies that they do not arise from mechanisms within the computational process, it is only the initial, "spontaneous" perturbations that are the doings of a *deus ex machina*. Subsequent perturbations, if any, arise from the mechanisms of the computational process.

[16] "Meta-perturbations" correspond to sudden, spontaneous changes to the rules of the process; perturbations, whether spontaneous or not, do not. Meta-perturbations occur because the logic network is suddenly, "spontaneously" different as the result of a physical failure (or failures). There may or may not be accompanying perturbations.

perturbation process easily accommodates the inclusion of failures and anomalies into a simulation.

Algorithmic representation of the perturbation process is conceptually straightforward:

> From two lists of future perturbations (one list, constructed by this very algorithm, in which each item describes a perturbation and its time of occurrence; a second list, created externally, in which each item describes a "spontaneous" perturbation, or meta-perturbation, and its time of occurrence) all items with a common, earliest time are extracted. This time is recognized to be, by construction, the present moment, the "now," of the perturbation process. The extracted items represent the state of the process at the moment (*plus* any meta-perturbations representing effects of physical failures also occurring at the moment).
>
> Then, all consequences of any meta-perturbations just extracted are determined—from rules[17] that are either contained in a separate list of rules governing perturbations or specified within the identification of the meta-perturbation itself. Since meta-perturbations represent changes to the rules governing the perturbation process, their primary consequences are modifications to some of the items in the list of rules. Secondary consequences of meta-perturbations are related perturbations that are added, as a function of their time of occurrence, either to the list of future perturbations or to the set of perturbations just extracted. (Some of these new perturbations may simply cancel out perturbations already listed.)
>
> Finally, all consequences of the perturbations in the set just extracted (and possibly just augmented) are determined from the rules (possibly just modified) and added to the list of future perturbations. (Again, some of these new perturbations may simply cancel out perturbations already listed.)

Repeated use of this algorithm can generate the successive states of the perturbation process. Conceptually! Note that the algorithm only *extracts* items from the list of spontaneous perturbations. However, there is nothing to prevent items from being added by another mechanism, say, another algorithm playing the role of a *deus ex machina*. (See footnote 15.)

The algorithm accommodates as fine a fidelity of representation as desired—at a cost, of course. Indeed, the algorithm includes, as a special case,

[17] Each rule is simply an algorithm for modifying itself or other rules in the list of rules governing perturbations, or it is an algorithm for computing whether or not the current output signal of a connected, elementary device is consistent with its (newly changed) input or will be perturbed at some future time.

Note that the logical values of the output signals of elementary devices (i.e., as seen by a computational process) correspond to mutually exclusive ranges of some physical property (e.g., an output voltage) of the devices. By construction, the number of logical values of a device's output signal is finite. For example, in Boolean logic the number is nominally two, corresponding to TRUE and FALSE.

Of course, real hardware is somewhat more involved than the unequivocal definitions of Boolean logic imply. Since in any real device there will be some internal signal noise, there must be an allowance made for a finite separation between two ranges that carry the logical values—if the meaning of the ranges is to be unequivocal. From the point of view of Boolean logic, such an intermediate range has no meaning and its existence is not recognized. Yet it is there. Clearly, in changing from one logical state to another (say TRUE to FALSE), the output signal of a device must, however fleetingly, take on values in such a region of limbo between TRUE and FALSE. Hence, a more faithful model of the physical behavior of an elementary device (say a logic gate) would have to account for such a "meaningless" range. Some simulation schemes do. Indeed, some simulation schemes consider subdivisions of this "meaningless" range in order to represent different, possible characteristics of transitions between the ranges that carry the logical values.

Although the conceptual algorithm accommodates simulation details down to the level of differential equations (actually, representation by difference equations) of electrical circuit analysis, the implemented versions of the gate logic algorithm within the diagnostic emulation technique do not model the transition phase between the two (Boolean) ranges of the output signals of properly functioning devices. That is, a device's output signal is considered to be either greater than a certain threshold level, representing one of the logical values, or less than a lower level, representing the complementary logical value. With the exception of one device, the tri-state, the possible occurrence of an output signal that corresponds to neither TRUE nor FALSE is considered faulty behavior and accommodated separately. For the modelling of the tri-state device, see appendix C.

The preceding discussion does not mean that the implemented versions cannot accommodate fine detail. They can, but to do so they require additional "user definable" functions, as described in the next section.

simulation according to the differential equations of electrical circuit analysis. (See footnote 17, especially the last two paragraphs.) That is, fidelity can be increased by subdivision of the elementary devices into ever more elementary subdevices (i.e., by identification of more possible perturbations and more perturbation rules). Such a process of refinement leads eventually to a set of relationships corresponding to difference equations representing the differential equations for currents and voltages in elements of the electrical circuits that compose the network. But, of course, such a refinement would mean that an interval of simulated time would contain more events (perturbations) whose consequences would need to be determined, and more resources would be required for the simulation. In general, therefore, simulation at the level of differential equations is feasible only for networks that are quite small.

Note, however, that from the perspective of a perturbation process there is no need to impose a uniform level of fidelity of representation over an entire logic network. The trade-off between fidelity and resources (mainly real simulation time) can differ for different portions of the network being simulated simply by appropriate definition of the perturbation rules that apply to the various devices of the network. Moreover, the rules can be varied by the meta-perturbation mechanism. That is, there is naturally embedded within the algorithm a means of dynamic control of multiple levels of fidelity across a network in order to minimize resources needed at every moment during the simulation.

Implementation

The diagnostic emulation technique actually consists of several algorithmic schemes which, although highly intertwined, are separately describable according to their distinct functions. Similarly, there are several data structures that are shared by the cooperating algorithms. The technique derives its name from one of the schemes, admittedly not the most central of the algorithms, that simply emulates a network of logic gates and flip-flops (and a few other devices). At the gate level of description, local manifestations of many isolated physical defects can be represented as logical symptoms[18] that can then be inserted into an emulation to determine their subsequent effects upon the rest of a digital system without having to increase the level of detail. The identification of potential physical defects and their correspondence to logical symptoms is not, however, an element of the diagnostic emulation technique; it is, rather, a precursory task.

An equally significant feature of the technique is the algorithmic scheme for linking combinations

[18]Possibly the most widely discussed type of fault is the "stuck-at" fault, that is, when a physical defect is determined to affect a gate by causing its outputted quantity to be stuck at some value and is interpreted, on the logical level, as a particular logical value. The stuck-at fault is popular because it is convenient to handle, simulate, analyze, etc. Admittedly it is not adequate for representing the manifestations of all physical defects of digital systems. For example, logical representations of some (not necessarily extensive) defects can call for redefinitions of significant portions of the networks of the logic devices.

of emulation and simulation algorithms.[19] From the previous discussion of the conceptual algorithm, it should be clear that various parts of a system can be represented at different levels of detail (as appropriate to an efficient use of the emulation technique). However, the efficiency of this multilevel feature depends upon the use of particular data structures,[20] called "actions,"[21] that govern the scheduling and occurrence of events. While some actions are predefined (e.g., those for simulating fetching from and storing into memory), the data structures have been chosen, among other reasons, to allow the creation of new actions by users and, thus, the creation of new fault and failure types and linkages for interfacing other emulation and simulation schemes when desired.

Figure 1 shows the major functional groupings of the algorithms that comprise the technique. Their titles are clearly descriptive. Also clearly, emulation at the level of gate logic is only one element; it is capable of playing a principal or secondary or even minor role in emulations and simulations.

Notice that in addition to the algorithms that function during the emulation process, the technique includes preprocessing functions—for translating descriptions of systems to be analyzed into the formats of the necessary data structures, for automatically generating consistent sets of initial values for the outputs of the devices of systems, and for generating the data structures used to map (i.e., associate) emulated and simulated hardware to the hardware of the real host system.

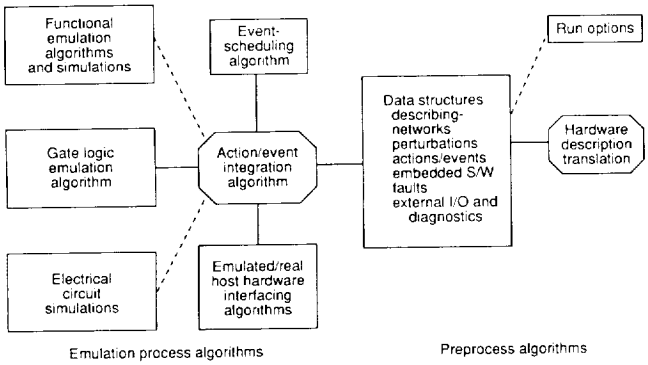


Figure 1. Components of diagnostic emulation technique.

The gate logic algorithm uses five data structures to represent a logic network and its behavior: two contain perturbations, one contains output signals, one contains network structure, and one contains pseudo registers. The two smallest structures each contain a list of perturbations occurring at an instant

[19] Questions often arise about the nuances of the words emulation and simulation. In general usage, both words denote the imitation of one person, thing, etc., by another; simulation, however, especially connotes likeness, whereas emulation connotes betterment. In computing matters, simulation is the general, non specific term. For example, in a general sense, CARE III (a reliability analysis code; see Bavuso and Peterson 1985) simulates the failure process of a system. Obviously, no real physical failures occur; there is a limit to the verisimilitude of a simulation. Similarly, when a computer code written for one computer (A) is interpreted by code in another computer (B), so that computer B produces the same results (when viewed by an external observer) as would computer A, then computer B is said to "simulate" computer A. However, there is a special case in which computer B performs in an algorithmically identical fashion each internal operation that computer A would perform. This special case is called "emulation." It is most efficiently achieved by microcoding.

Other things being equal, one would expect *microcoded* emulation to provide for faster operation and simulation to provide for easier implementation. Other things are not always equal.

Modern computers whose instruction sets are defined by microcode in the hardware are simply emulations, simply roles acted out to a microcode script by the underlying hardware. At a moment's notice, the script could change and the actors could take on other roles. Thus, a manufacturer can offer a seemingly new computer by merely changing the internal microcode—without having modified one element of the underlying hardware.

[20] For items of information in a computer to be manipulated meaningfully, they must be organized (i.e., logically ordered) in a manner consistent with the rules, the algorithms, that govern their manipulation. The rigid orderliness of the items of information in a computer is what is referred to by the term "data structure" in the jargon of computer science. There is always a consideration of efficiency (of operating time, of memory space, etc.) in the joint choice of data structures and algorithms.

[21] "Actions" are particular data structures of the diagnostic emulation technique that specify mappings (i.e., correspondences) between data structures (that may themselves be actions) and control the sequences of algorithms that transform and transfer data among the data structures. Such data transformations and transfers constitute "events," both emulated and real. Actions schedule actions to occur at future times. (And so, indirectly, they schedule the occurrence of events.) Actions control the transfer of information among different data structures of the emulation and the real, host hardware; they control the insertion (or removal) of faults and control the recording of diagnostic observations. For example, the mapping of one or more words of emulated memory to the real host memory is information specified by an action that performs the fetching of data from an emulated computer's simulated memory. Thus, a set of actions controls the running of an emulation.

of time; one corresponds to the present moment t_0, while the other corresponds to a future time $t_0 + \delta t$, where δt represents the time required to propagate the perturbations identified in the first list through the bank of devices identified in the second list. The data are, of course, simply the identifications of devices whose output signals change value at that instant in time. In terms of the general perturbation algorithm previously described, these two data structures represent subsets, respectively, of the list of extracted items and the list of future perturbations.[22] The third data structure contains a description of the complete pattern of output signals, that is, the state of the computational process in the logic network. The fourth data structure is the largest and contains a description of the network's structure, its devices, and their interconnections. In terms of the conceptual algorithm, this fourth data structure contains some (but not all) of the rules of the perturbation process. In particular, it contains those related to the topology of the network.

The gate level algorithm contains the animus for the process; that is, it contains, within its construction, the definitions of the equilibrating action of the different types of elementary, scalar-valued devices, regardless of where they appear in a logic network. The set of such devices consists of the various logic gate types (AND, OR, NAND, NOR, XOR, XNOR), inverters, buffers, flip-flops,[23] and tri-state devices.[24]

It is clear that the gate level algorithm *and* the fourth data structure contain a portion of the list of the perturbation rules (i.e., those pertaining to gate logic level devices) of the conceptual algorithm. From the perspective of the gate logic algorithm, the rules are unchanging since a network's structure is ordinarily unchanging; accordingly, the gate logic algorithm alters only (dynamic) signal information and contains no logic for altering static network-structure information.[25]

The gate logic algorithm operates only upon devices within a prescribed logic network (i.e., that identified by the fourth data structure)—which may (or may not) represent the entire system being emulated. Perturbations (signals) that come from outside the prescribed network (as in fig. 2) appear "spontaneously" in the first data structure (i.e., the one containing items of time t_0), and the gate logic algorithm operates in the manner of that portion of the conceptual algorithm that is ignorant of spontaneous perturbations. (Appendixes A to D contain finer details of the algorithm and the modelling of the elementary devices.)

On the other hand, signals leaving the prescribed network go into a fifth data structure that contains fictitious (register) devices. These registers serve to

[22]Here are two constraints on—and therefore, two possible enhancements of—emulation imposed by the existing versions of the diagnostic emulation technique. First, the existing versions (i.e., on the VAX-11/750 and the QM-1/A computer) treat the instant $t_0 + \delta t$ as the next earliest time in the list of future perturbations and, hence, constrain the granularity of simulated time. But the fidelity of a simulation can be adversely affected when the granularity is too great (namely, when the order of some events that are in reality sequential is of some consequence, but the events appear to be simultaneous because of the granularity of simulated time). This is indeed possible since the logical values of the input signals of the devices correspond to ranges of some physical property, as do the values of output signals (see footnote 17), and it is virtually impossible that any two devices having an input signal from a common source could recognize exactly the same value of the physical property to be the edge of a range. That is, when examined at a fine enough granularity of time, one of the two devices will be seen to recognize a change in the input signal common to both devices before the other device does. Hence, at a fine enough granularity of time, no perturbations are simultaneous. In some circumstances, therefore, it can be necessary to consider events intermediate in time between t_0 and $t_0 + \delta t$. In the existing versions, this is only possible by the introduction of fictitious devices—admittedly an inefficient ploy. However, the list of future perturbations exists also in another data structure not used directly by the gate level algorithms. It would be possible to modify the manner of use of that data structure in conjunction with the gate logic algorithms to obtain a finer granularity of time—without having to introduce fictitious devices.

Second, the restriction to these two data structures means that all devices are assumed to have the same propagation time, namely δt. Although this assumption can be relaxed in existing versions of the technique (again by the insertion of fictitious devices), this is again a cumbersome method, and a more efficient method for avoiding the assumption of a single, common propagation time would be to increase the number of data structures containing the lists of perturbations from two to some larger number that would be unlikely to be exceeded. Of course, the most general method would dynamically determine the precise number of data structures needed.

[23]More precisely stated—as implemented the flip-flop portion of the algorithm handles "halves" of flip-flops. See appendix B (footnote 34) for an explanation of the manner and reason.

[24]Another basic logic device is the bidirectional switch element. Because of an oversight in the implementation of the diagnostic emulation technique, no definition of the equilibrating action of the switch element was included in the gate logic algorithm. However, such switch elements can be represented—in either of two ways. Within the gate logic algorithm, a switch can be represented by a fictitious network constructed from the available elementary devices. Or, external to and interfacing with the gate logic algorithm, a switch can be represented by a specially created action. Admittedly, both representations are inefficient, especially if many instances of such switches occur in the system being emulated. Here is another possible enhancement.

[25]Of course, the data are only static from the viewpoint of the gate logic algorithm portion of the computational process. As stated previously, physical failures can (and usually do) manifest themselves as redefinitions of the perturbation rules and as seemingly spontaneous perturbations (i.e., changes to signal values).

As constructed, the gate logic algorithm and its data structures do not embody all the possible rules of the perturbation process. Consequently, only those changes to the rules which express new network configurations are accommodated. However, this is a fairly large set. For example, it includes all stuck-at faults.

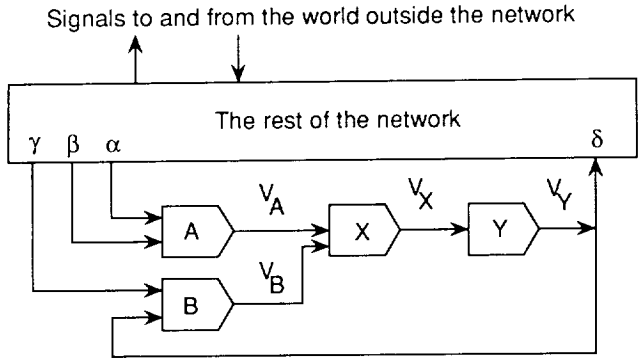


Figure 2. Sample network of logic devices.

collect and compress the information to be extracted from the prescribed logic network. Packed information is then available for use by other algorithms. Note that this information may be an integral part of the emulated system, or it may be diagnostic information having no inherent use within the perturbation process. For example, a register may contain data which trigger an "action" outside the gate level algorithm to simulate (at a functional simulation level) a fetch of data from a simulated memory location. On the other hand, a register may contain data to be printed (by an action devoted to that purpose) for use in analyzing the emulated system or, perhaps, for use in analyzing the operation of the diagnostic emulation process.

A means for controlling the sequence of events is central to the diagnostic emulation algorithm. Since each event is the occurrence of a number of actions (performed by algorithms), control is performed by naming the actions to be performed at a given time, the order in which the actions are to be performed, the identification of the algorithms that perform the actions, identification of the parameters and variables to be used by the algorithms, and by the activation of the algorithms accordingly. But note that this control process is precisely the process described in the conceptual algorithm. That is, the central control algorithm is merely the implementation of the portion of the conceptual algorithm that examines each item within the extracted set of perturbations—with specific data structures in mind.[26] Note that the meaning of a perturbation item is expanded here to include functions that control the emulation process and that the particular data structure used is merely a list of actions, each one of which contains information of the sort stated in the preceding paragraph.[27] The central control algorithm invokes each action in turn. The last action in the list is (almost) always one that advances simulated time to the next earliest time taken from a list of future events (i.e., it

[26]The data structures consist, in general, of linked lists embedded within linked lists embedded within linked lists, etc. It is instructive to note that the algorithmic and data structuring techniques used here, when taken in combinations, form compound algorithms (and data structures) that are sometimes recursive, sometimes merely (if maddeningly) convoluted, and very often tedious. In other contexts (e.g., in artificial intelligence, in expert and knowledge-based systems, etc.), complex combinations of these same techniques often take on an aura of mystique. But individually they remain quite straightforward.

[27]The data structure of an action is akin to that of a computer instruction. That is, it has a tag by which it can be addressed (usually a time tag), an operation code that specifies the action it is calling for, and operands that identify which variables are to be involved in the action. In this general format, an action is capable of many things. For example, there is an action that examines fictitious registers in order to determine which other actions are being invoked by, say, the gate logic, by an external signal originating, say, from an operator's keyboard, or even by another action.

in effect supplies a new, *extracted* set of actions for the central control algorithm to invoke). Thus, the central control algorithm appears to find an endless list.

The event-scheduling algorithm is invoked by its corresponding action. Its function is to insert other actions into a list of future events. The data structure is that of a linked list. It accommodates the insertion of events at times not yet in the list and the insertion of events at times for which other events are already identified.

The emulated/real host hardware interfacing algorithms provide the means of using the real hardware of the host computer to perform emulated functions. They map data structures that contain descriptions of (portions of) the emulated system onto real registers of the host computer. For example, responding to their corresponding actions, these algorithms transfer data (e.g., diagnostic data) to buffers and registers for display to users.

Functional-level simulations are provided within the diagnostic emulation technique to represent the fetching of data from and the storing of data into emulated memory. The simulations allow for the inclusion of time delays and lags. Any number of additional desired functions can be added by the definition and addition of appropriate actions to the list of available actions and the addition of simulation code to perform the new actions.

No circuit-level simulations are currently provided within the diagnostic emulation technique. They must be provided by the users. Of course, they require the definition of appropriate actions.[28]

Summary

There are, of course, many ways of implementing an emulation or simulation. The diagnostic emulation technique is but one particular, algorithmic process organized about particular data structures. It has been sub-microcoded on a horizontally microprogrammable computer as well as coded in FORTRAN for a general purpose computer. It may be compared with other techniques as follows:

1. It contains an *emulation* scheme at the level of gate logic—in contrast to most available computer programs for simulating digital networks at the gate logic level. The diagnostic emulation technique does accommodate the (add-on) use of simulations (e.g., for representing portions of a system at a higher functional level of detail, such as the instruction and register transfer level, or, if needed, at a lower functional level of detail).

2. It is a *generic* scheme—in the sense that it uses a fixed algorithm for all networks emulated at the gate logic level. Indeed, a network's de-

[28]This capability has been used once in support of an experiment for a doctoral dissertation. A transistor level model of a gate logic level device was used as a means of examining the effects of specific faults at the transistor level (see footnote 12).

scription appears as an input data set. Contrast a generic scheme to what might be called an embedded network scheme, a scheme that compiles (i.e., embeds) a system's network description into a computer program.[29]

3. It is a *hybrid* scheme. That is, although the nominal level of detail is the gate logic level, the scheme facilitates the representation of different parts of a digital system at different levels of detail—usually, for computational efficiency, at a less detailed and more functional level (e.g., memories do not usually need to be emulated at the gate level), but also, if needed, at a more detailed level (e.g., a transistor level description of a device to be faulted). Flexible extension to other levels of emulation and simulation is possible by means of user-definable (and also some predefined) "actions." (See footnote 21.) This hybrid extension scheme appears to be a special feature of the diagnostic emulation technique.

4. It is a *quasi event-driven* scheme. Following the completion of an event, simulated time is usually advanced to the moment of the next scheduled event, but in time steps of predetermined amount that correspond to the interval of time for logically (but not necessarily physically) propagating a signal through a device. For example, in the emulation scheme, rather than being evaluated at each possible increment in time, the output of a gate is evaluated only when at least one of the gate's inputs changes. A similar (scheduling) process is also applied to more global events. Efficiency is lost, however, if the scheme is applied to devices with vector-valued output; for this reason, such devices are better emulated at a more functional level.[30]

5. It is a *unit-propagation-delay* scheme, for the most part, in its handling of gates. That is, in its normal mode of use, all gates are presumed to share the same propagation delay. However, varied delays can be accommodated by means of the user-definable actions.

6. It is mainly a *binary-valued* scheme, since its principal data structures accommodate only TRUE and FALSE values for the outputs of its nominal primitive devices. However, more complex, multi-state behavior can be accommodated, at a cost in running time, by the use of the user-definable actions.

7. Finally, it is a *sequential processing* scheme as it has been implemented (in FORTRAN for general purpose computers and in sub-microcode for a horizontally microprogrammable computer)—because of the limitations of the host hardware. However, there is *latent parallelism* corresponding to concurrency among events in the system being emulated. Therefore, with some modifications to the algorithms, advantages could be gained from special host hardware.

[29]For analyzing failure events, generic schemes have several useful characteristics that embedded network schemes do not. The algorithms of generic schemes are effectively independent of the network being emulated and can be implemented directly in special hardware—under some conditions to an advantage. For example, special purpose computers (for verifying system designs by emulation at the gate logic level) have been proposed (e.g., see Abramovici et al., 1983), have been developed as one-of-a-kind machines (e.g., see Pfister, 1982 or Koike et al. 1985), and have even become commercially available.

Also, since a generic scheme accommodates different network definitions by changes to an input data set, the scheme can readily accommodate hardware failures that effectively cause a redefinition of the network being emulated during the course of an emulation. Moreover, the network description as a separate data structure that is directly driving the emulation can always be readily visible.

An embedded network scheme, on the other hand, would presumably require a compilation for each separate network and, therefore, additional validation—or its credibility would be strained. Clearly, embedded network schemes march to a different drummer; they appear to be more suitable to verification and validation of software, for which there is no consideration of failures in the underlying hardware. As an example, see the TRW (1980) report.

[30]In event-driven schemes, following the simulation of an event, simulated time is advanced abruptly to the next (possibly dynamically) scheduled, relevant event(s). No host computer time (or resource) is used simulating irrelevant events that occur during the intervening period of time—but at a cost of maintaining a (dynamic) schedule of events. Obviously, there must be some means of having foreknowledge of the next event(s). When such foreknowledge is not possible, in the simplest scheme that is not event-driven, simulated time is incremented by predetermined amounts. The consequence is that the complexity of dynamic scheduling is avoided—but at a cost, namely, the cost of simulating each device at each step in time to see whether or not its output has changed (i.e., to see whether or not an event has occurred) in the last increment of time.

The diagnostic emulation technique falls between the two extremes. Events can be forecast; almost all (fixed) increments of time contain at least one event, but most increments contain relatively few (of all possible) events, and some kinds of events can only be forecast in the immediately adjacent interval of time.

Appendix A

Modelling Gates

The algorithm for evaluating and updating the logical states of gates in the diagnostic emulation technique uses counters to represent gates. A network of gates of various types is represented by an identical network of counters—akin to a directed graph. The algorithm increments (or decrements) the counters of the network as their inputs change. The inputs, of course, come from other counters and represent the changing output values of gates (and other devices). However, only a change of a counter's value to or from zero corresponds to a change in the output value of the gate it represents—and vice versa. Therefore, only when a transition of a counter's value into or out of zero occurs does the algorithm propagate the event as an increment (or decrement) to other counters indicated by the network. Consequently, the task of the algorithm is simply to increment (or decrement) indicated counters, and to indicate which, if any, other counters are to be incremented (or decremented) in turn. Repeated use of the algorithm emulates the behavior, as it varies with the passage of time, of a network of interacting gates.

The modelling of gates as counters to be incremented (or decremented) as the inputs change is based on the observation that gates of all types (namely, AND, OR, NAND, NOR, XOR, NXOR, NOT, and the simple buffer) may be considered special cases of a hypothetical gate (the ${}^{m}_{n}$XOR gate[31] and its negation, N${}^{m}_{n}$XOR). The equivalence of the ${}^{m}_{n}$XOR gate (and its negation), for selected values of the m and n parameters, to the various types of gates follows from their definitions. For example, the output value of an AND gate having n inputs is defined to be TRUE when and only when all n of its input values are TRUE; otherwise it is FALSE. The output of an ${}^{n}_{n}$XOR gate behaves similarly. As another example, the output value of an XOR gate is TRUE when one or the other, but not both, of its two inputs is TRUE, and otherwise it is FALSE; this is also the behavior of the output value of a ${}^{1}_{2}$XOR gate. Thus, a network of the hardware gate types can be conceptually replaced by a network of properly chosen ${}^{m}_{n}$XOR gates. (See last paragraph of footnote 17.) Table A1 contains the set of useful equivalences.

Table A1. Equivalence of Hardware Gates to ${}^{m}_{n}$XOR Gates

Hardware gate	No. of inputs	${}^{m}_{n}$XOR gate
AND	n	${}^{n}_{n}$XOR
OR	n	N${}^{0}_{n}$XOR
NAND	n	N${}^{n}_{n}$XOR
NOR	n	${}^{0}_{n}$XOR
XOR	2	${}^{1}_{2}$XOR
NXOR	2	N${}^{1}_{2}$XOR
NOT	1	${}^{0}_{1}$XOR
Buffer	1	${}^{1}_{1}$XOR

In turn, an ${}^{m}_{n}$XOR gate can be represented by a counter whose value is the count of the ${}^{m}_{n}$XOR gate's inputs that are TRUE at any given moment. Consequently, when a gate's output value is TRUE, the corresponding counter's value is m, and vice versa; when a gate's output value is FALSE, the counter's value is some quantity which is not m, and again vice versa. Clearly, the gate's output value changes whenever the counter's value changes to or from m. And again, the converse is true. It is also convenient to represent the current value of a gate by an extra bit attached to the corresponding counter. As long as the bit's value is changed every time the counter's value changes to or from the critical value m, the bit's value will always correctly represent the gate's output value—assuming, of course, that it was correct initially. Thus, for example, a simple 9-bit counter can represent an ${}^{m}_{n}$XOR gate with up to $n = 15$ inputs (1 bit holds the output value, 4 bits hold the count, and 4 bits hold the value of the parameter m, which can be any integer from 0 to 15).

Note that in 2's complement notation a biased counter (i.e., a counter from which the value of the parameter m has been subtracted) also can represent an ${}^{m}_{n}$XOR gate. Unlike unbiased counters, each having its own critical value for the parameter m, biased counters share a common critical value (in particular, zero) for signalling a change in a gate's output value. Biased counters do not need to carry along the value of a parameter m (since it is always zero) and

[31] The ${}^{m}_{n}$XOR gate, read "m out of n exclusive or" gate, has an output value defined to be TRUE when and only when any m (but exactly and only m) of its n inputs are TRUE; otherwise its output value is FALSE.

therefore need only half as many bits, not counting the extra bit that holds the gate output value. Consequently, a 5-bit biased counter (now counting the extra bit) suffices to represent an ${}^{m}_{n}$XOR gate with up to $n = 15$ inputs—clearly an improvement on an unbiased counter. A 7-bit biased counter would accommodate up to 63 inputs, a number not often exceeded in real hardware.

In addition, since the value of the extra bit corresponds to the gate's output value (which is either 0 or 1), the quantity to be added to the extra bit to change it to its new value is precisely the increment (or decrement) to be made in other counters to which, as indicated by the network, the change must be propagated. For example, to an extra bit's value that is 0 (i.e., FALSE) when its biased counter changes to or from zero the quantity 1 must be added to yield a new value of 1 (i.e., TRUE). Thus, 1 is exactly the increment to be made in other connected counters. If the extra bit's value had been 1 (i.e., TRUE) and thus needed to change to 0 (i.e., FALSE), the quantity to be added would have been -1, a ready-made decrement. Note that the only operation on a counter is one of addition; decrementing a counter happens by the *addition* of a negative increment to the 2's complement quantity in the counter.

The preceding paragraphs apply equally to N^{m}_{n}XOR gates when ${}^{m}_{n}$XOR is replaced by N^{m}_{n}XOR, and TRUE, FALSE, 1, and 0 are appropriately interchanged. The two types of gates are naturally distinguished because the possible combinations of the values of a counter and its extra bit are mutually exclusive—as shown in table A2. Moreover, since the operation on the counters is the same for ${}^{m}_{n}$XOR and N^{m}_{n}XOR gates, knowledge of a gate's type is not needed after the initial values for the counter and extra bit are determined.

Table A2. Gate Type as Function of Extra Bit and Counter Values

Extra bit value	Gate type for counter value of—	
	Zero	Nonzero
0	N^{m}_{n}XOR	${}^{m}_{n}$XOR
1	${}^{m}_{n}$XOR	N^{m}_{n}XOR

As an illustrative example, consider an OR gate with the three inputs X, Y, and Z. An OR gate, of course, is supposed to take on the output value TRUE when any one or more of its inputs takes on the value TRUE, as happens, for example, when the input set (XYZ) = (TRUE,TRUE,FALSE). Only when all the inputs have the value FALSE, that is, (XYZ) = (FALSE,FALSE,FALSE), does the OR gate take on the output value FALSE. Of the eight possible combinations of values that the input set can assume, there is only one that corresponds to the logical output value of FALSE, a behavior equivalent to the N^{m}_{n}XOR gate with parameters $m = 0$ and $n = 3$. With the notation 1 and 0 for TRUE and FALSE (for then the count of inputs is simply the sum of their values interpreted as decimal digits), the counter value $C(XYZ)$, as a function of the inputs that have the value 1 and the bias amount m is

$$C(XYZ) = X + Y + Z - m$$

Note that the counter can only take on the values 0, 1, 2, or 3, and only one value of the counter corresponds to the output value 0 (and, in fact, it is the count value 0), while all other counter (and count) values correspond to the output value 1. Therefore, knowledge of the count is equivalent to knowledge of the value of the OR gate's output. As the counter changes from one nonzero value to another (e.g., from 2 to 1), the output value remains unchanged; it only changes value when the counter transitions to or from 0.

Consider that initially $X = Z$ = TRUE and Y = FALSE (i.e., $(XYZ) = (101)$). Then,

$$C(XYZ) = C(101) = 2$$

and, since the output value of the OR gate is TRUE, the extra bit must be 1.

Now imagine that the input set changes to $(XYZ) = (100)$ (i.e., $\Delta Z = -1$). The counter takes on the value

$$\begin{aligned}\text{New count} &= \text{Old count} + \Delta X + \Delta Y + \Delta Z \\ &= C(101) + 0 + 0 + (-1) \\ &= 2 + 0 + 0 - 1\end{aligned}$$

Since the counter has not transitioned to or from 0, the extra bit is not changed. But this is exactly the result that would have been determined from

$$\begin{aligned}C(100) &= X + Y + Z - m \\ &= 1 + 0 + 0 - 0 \\ &= 1\end{aligned}$$

where also the extra bit would be 1 since the output value of the OR gate is TRUE.

Imagine again that the input set changes, this time to $(XYZ) = (000)$ (i.e., $\Delta X = -1$). Then,

$$\begin{aligned}\text{New count} &= \text{Old count} + \Delta X + \Delta Y + \Delta Z \\ &= C(100) + (-1) + 0 + 0 \\ &= 1 + 0 + 0 - 1 \\ &= 0\end{aligned}$$

Since the count has changed to zero, the extra bit must change. It was 1; it becomes 0. The biased counter's value becomes 0 with extra bit 0. A moment's reflection reveals that this is the proper value. And since the extra bit has changed, an increment of -1 will be propagated to other counters.

In summary, a real hardware gate can be represented by an ${}^{m}_{n}$XOR gate (or its negation) with properly selected values of the parameters m and n. In turn, an ${}^{m}_{n}$XOR gate (and its negation) can be represented by a biased counter (in 2's complement notation) with an extra bit. After initial values for the counter and the extra bit have been determined, changes in the counter sometimes generate increments (or decrements) to propagate on to other counters in a network. There is only one operation performed on a counter, and it is common to all counters (i.e., adding the increments). As a result there is little complexity in the data structure and processing algorithm needed to emulate behavior of a gate.

Appendix B

Modelling Flip-Flops

The algorithm for evaluating and updating the logical output values of flip-flops in the diagnostic emulation technique uses counters to represent flip-flops and represents a network of flip-flops by an identical network of counters. Of course, to distinguish the input from the output connections of each counter, the network of counters contains information about the direction of signal flow. As its inputs change, a counter's value is changed as a function of its current value as well as the input changes. However, only some of the possible changes of a counter correspond to a change of a flip-flop's output value. When such changes occur, their effects are propagated to other counters indicated by the network. Therefore, the algorithm's task is simply to change (according to a transition rule) the values of indicated counters and to indicate which, if any, other counters are to be changed in turn. Thus, repetition of the algorithm provides an emulation of the behavior, varying with the passage of time, of a network of interacting flip-flops.

The reader should notice the wording here is similar to that in the introductory paragraph of appendix A on modelling gates. The similarity is intended to emphasize the commonality of the algorithms and the fact that the two modelling schemes can readily be combined to emulate networks composed of a mixture of gates and flip-flops (and other types of devices).

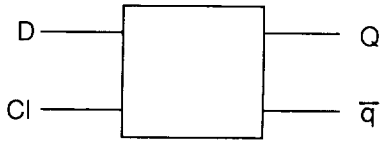
[32] The state machine is defined to have $n+4$ distinct, internal aspects. Each aspect can be only TRUE or FALSE (but not both simultaneously); hence, each state is uniquely described by a binary integer with $n+4$ bits. Obviously, the integer also serves as an index on the states. The input variable to the state machine is defined to be a binary integer of n bits. The state transition rule is defined in two steps. First, each of the first n aspects (i.e., the equivalent bits) of the state index is changed (i.e., either TRUE → FALSE or FALSE → TRUE, as appropriate) if and only if the corresponding bit of the input variable is TRUE. (Recall that a signal transmitted to a device indicates whether or not a change has occurred to the variable in question and not the value of the variable.) Secondly, the possible change in each of the remaining four aspects (i.e., the remaining 4 bits of the state index) is determined as a function of the integer values of the state index and the input variable. Thus, the values of the input variable and the state index together determine a new value for the index.

The representation of flip-flops as counters is based on the observation that a flip-flop (whether J-K, D, R-S, master-slave, with or without preset and/or clear, or triggered on the leading or trailing edge of a clock (Cl) signal) can be considered as a special case of a hypothetical flip-flop with n inputs whose behavior can be described by a comprehensive truth table and, in turn, modelled as a finite-state machine with 2^{n+4} states and a compact state transition rule.[32] That is, given a hypothetical flip-flop type, a state machine can be used to model the behavior of a flip-flop by (1) equating the n inputs, the two outputs, and two internal, binary-valued quantities (which have meaning in the case of a master-slave flip-flop) to the $n+4$ aspects of the flip-flop state machine; (2) equating the occurrence of changes (whether they appear as increments or decrements) of the flip-flop's inputs to TRUE (i.e., 1) values of the corresponding bits of the machine's n-bit input variable; and (3) using the flip-flop's truth table to determine the function that maps the state index and input variable into new values for the remaining four aspects. All the useful edge-triggered types of flip-flops appear to be included in the truth table in table B1; the truth table thus defines a single flip-flop state machine with $n = 5$.[33] Specific flip-flop types are generated from the combined truth table by an appropriate choice of initial values for the unused input variables and the option parameters.[34]

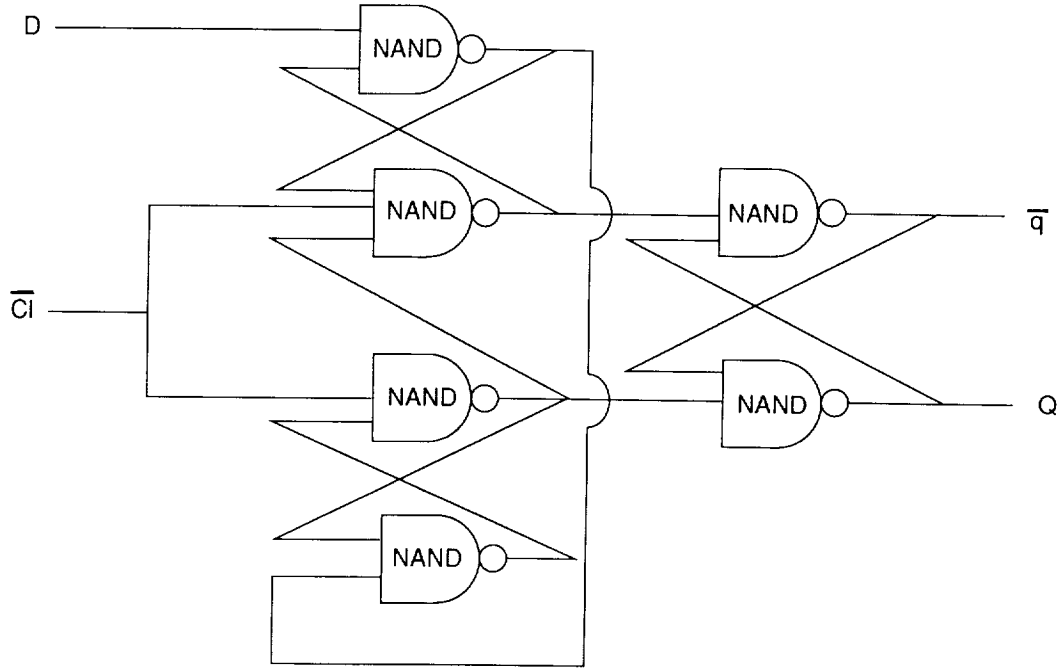
For example, consider the simple D flip-flop in figure B1. (Of course, the flip-flop could be implemented as a network of gates as in figure B1(b)—if

[33] For practical purposes $n = 8$, since three additional quantities, option selection parameters, are needed. Although not inputs, they can be considered inputs which do not change from their initial condition. They indicate when flip-flops are (or are not) master-slave types and/or types that have outputs that are indeterminate in some circumstances.

[34] Actually, the basic flip-flop device modelled in the diagnostic emulation technique corresponds to only *half* of a flip-flop, i.e., it corresponds to only one output's value, *either* the Q or the $\bar{q}$ value. The lowercase $\bar{q}$ is used instead of $\overline{Q}$ to emphasize this point. Either output can be obtained by the same algorithm by simply reversing the P and C input lines and, similarly, the J and K input lines—as the symmetry in the truth table in table B1 reveals. When needed, the other output's value is *usually* obtained by simply using the complement of the computed output's value. (The data structure available in the diagnostic emulation scheme for describing a network of devices contains a convenient capability for inverting (i.e., complementing) a signal going from one device to another.) For flip-flop types for which there is a possibility of output values that are not complements of each other (e.g., see row 1 in table B1), the two output values of the flip-flop must be separately computed through the use of two "halves" of a flip-flop.



(a) D flip-flop.



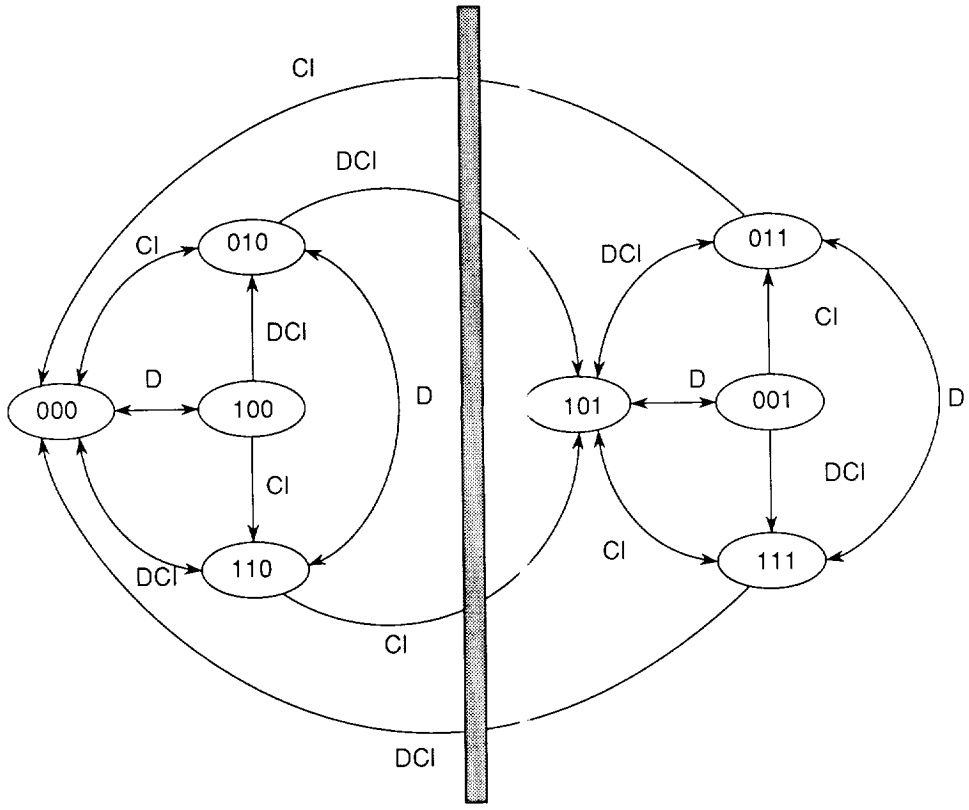
(b) Gate network for D flip-flop.

Figure B1. Components of D flip-flop.

D	Cl	Q	$\overline{q}$
*	↑	†	†
0	↓	0	1
1	↓	1	0

*Don't care.
↑Inactive edge of clock signal.
†No change.
↓Active edge of clock signal.

(c) Truth table.



(d) State transition diagram.

Figure B1. Concluded.

Table B1. Combined Truth Table for Flip-Flops

	Inputs					Parameters			Internals		Outputs	
Row	P	C	J (or D)	K (or $\overline{D}$)	Cl	σ	α	β	iQ_{n+1}	$i\overline{q}_{n+1}$	Q_{n+1}	$\overline{q}_{n+1}$
1	0	0	*	*	*	*	*	*	1	1	1	1
2	0	1	*	*	*	*	*	*	1	0	1	0
3	1	0	*	*	*	*	*	*	0	1	0	1
4	1	1	*	*	↑	0	0	0	*	*	†	†
5	1	1	0	0	↓	0	0	0	*	*	†	†
6	1	1	0	1	↓	0	0	0	*	*	0	1
7	1	1	1	0	↓	0	0	0	*	*	1	0
8	1	1	1	1	↓	0	0	0	*	*	Q_n	$\overline{q}_n$
9	1	1	1	1	↓	0	0	1	*	*	**	**
10	1	1	*	*	↑	1	0	0	†	†	iQ_n	$i\overline{q}_n$
11	1	1	*	*	↑	1	1	0	‡	‡	iQ_n	$i\overline{q}_n$
12	1	1	0	0	↓	1	0	0	†	†	†	†
13	1	1	0	1	↓	1	0	0	0	1	†	†
14	1	1	1	0	↓	1	0	0	1	0	†	†
15	1	1	1	1	↓	1	0	0	iQ_n	$i\overline{q}_n$	†	†
16	1	1	1	1	↓	1	0	1	**	**	†	†

*Don't care.
↑Inactive edge of clock signal.
†No change.
↓Active edge of clock signal.
**Indeterminate.
‡No change—indeterminate if J and/or K changed during time interval δt before active edge of clock.

there were a need.)[35] The truth table in table B1 generates the same limited, state-machine behavior as the D flip-flop's truth table (fig. B1(c)) if the σ, α, and β parameters are preset to 0, the P and C inputs are preset to 1, and the K input is considered to always equal $\overline{D}$ in the absence of an actual $\overline{D}$ input. Also note that for a D flip-flop that is not a master-slave type, the internal binary quantities are irrelevant. Hence, only rows 4, 6, and 7 in table B1 are applicable and, for the D, Cl, Q, and $\overline{q}$ variables, they are clearly equivalent to the D flip-flop's truth table. Furthermore, with the assumption that the $\overline{q}$ output can only be the complement of the Q output, $\overline{q}$ is redundant and only 2^{n+1} states are needed to describe the D flip-flop state machine. Moreover, since $n = 2$ (i.e., the only inputs are D and Cl; $\overline{D}$ is redundant),

$$2^{n+1} = 2^3$$

Hence, a 3-bit integer, corresponding to the triplet of signals D, Cl, and Q, suffices to index (and describe) the states. Since it is irrelevant (i.e., a matter of convention) whether the leading or the trailing edge of the Cl signal is the active edge, assume for the purpose of this example that the $1 \rightarrow 0$ transition is the active edge. From the truth table, one can construct the diagram in figure B1(d), depicting all the states and transitions among the states. The input signals whose changes caused the transitions are indicated along the arcs connecting the states. In effect, the "causes" correspond to the values of the input variable of the machine.[36] (Note that only those transitions which cross the vertical barrier in the center of the diagram correspond to changes of the flip-flop's output value).

In a similar manner, a J-K flip-flop with direct clear input (see truth table in table B2) can be represented by presetting the σ, α, and β parameters to 0 and the P variable to 1 and by choosing the $0 \rightarrow 1$ transition to be the active edge of the clock signal. Rows 3 to 8 (excluding 4) of the combined truth table (table B1) become the only applicable rows and have the same behavior as the J-K flip-flop's truth table. The behavior of other useful flip-flop types can be modelled from the hypothetical flip-flop's truth table. With such equivalences in mind, a network of flip-flops can be conceptually replaced by a network of properly initialized flip-flop state machines corresponding to the combined truth table.

Table B2. Truth Table for J-K Flip-Flop With Clear Direct Input

C	J	K	Cl	Q_{n+1}	$\overline{q}_{n+1}$
0	*	*	*	0	1
1	0	0	↑	Q_n	$\overline{q}_n$
1	0	1	↑	0	1
1	1	0	↑	1	0
1	1	1	↑	$\overline{q}_n$	Q_n

*Don't care.
↑Inactive edge of clock signal.

A flip-flop state machine, in turn, can be represented by a counter whose content is the value of the state index (or, equivalently, the state descriptor) at any given moment. Changes to the counter value need (indeed must) only be made when the value of the input variable is nonzero (i.e., only when a transition event occurs; see footnote 36 for exceptions) and are then made in accordance with the state transition rules of the flip-flop state machine. That is, changes to the first n (the input) bits of the state index are determined directly by the input variable; each is simply complemented when the corresponding bit of the input variable indicates a change. Changes to the bits corresponding to the output (and internal) quantities are determined as a function of the value of the counter and the input variable, and when a change to an output quantity does occur, that change is propagated to other devices as indicated by the network, appearing either as an input variable indicating a change to a particular input (if the device is another flip-flop) or as an increment or decrement (if the device is a gate). The function is determined, of course, from the flip-flop state transition diagram; moreover, it only needs to indicate which transitions correspond to a change. For example, the state transition diagram for the D flip-flop illustrated in figure B1(d) can be transformed into the computationally more convenient matrix form in figure B2(a), in which transitions corresponding to output changes are marked

[35] If there were a need to examine the behavior of the flip-flop in greater detail (e.g., to observe faulty behavior), the network of gates in figure B1(b) could be emulated by only the gate modelling capability (described in appendix A) of the diagnostic emulation scheme. Of course, in a network of flip-flops this would be done with discretion, since it increases the number of primitive devices being emulated—in this case of a D flip-flop, for example, by a factor of 6 to 1.

[36] Of course, for any interval of time during which no change occurs to the inputs, the state can be considered either to remain unchanged or to transition to itself. That is, there is no behavior to be modelled between transition events; otherwise, there would be additional recognizable states of the finite-state machine and something, perhaps the passage of time, causing changes from these states. If one wished to describe the process of transition from one state to another in greater detail, one could define additional tiers of states with different transition rules. For example, the process of a device failing in the time interval *between* normal, state transitions could be modelled this way.

State	Input Change Cl	D	DCl
000	010	100	110
001	011	101	111
010	000	110	101+
011	000-	111	101
100	110	000	010
101	111	001	011
110	101+	010	000
111	101	011	000-

(a) Device.

State	Cl	DCl	ΔQ
	011	111	-
	110	010	+

(b) Truth table.

Figure B2. State transition matrix and compacted function for D flip-flop.

by a "+" or a "−" (i.e., increments or decrements). Hence, in this case, all the information needed to specify the function is contained in figure B2(b)—a useful reduction in data structure. Further, if the devices (i.e., counters) which provide the inputs to a flip-flop are evaluated singly and their outputs propagated singly, then an input variable to a state machine's counter will contain an indication of change to only one bit of the counter. That is, only n columns (one for each input signal) of the reduced matrix are needed. In the D flip-flop example, this corresponds to saying that *none* of the transitions labeled DCl will ever occur. Hence, only the Cl column in figure B2 is needed to specify the transition function. Thus, the state transition function reduces to a search of a one-column table. If the value of the counter is found in the table, the output (i.e., Q) bit of the counter is complemented and the ΔQ propagated; if not, no change is made and no propagation occurs. In a similar manner, the combined truth table in table B1 implies a complex flip-flop state transition diagram with an equivalent state transition matrix and a compact function statement that can be implemented as a simple table search.

In summary, a flip-flop can be modelled as a flip-flop state machine. In turn, the flip-flop state machine is naturally represented as a counter with $n+4$ bits. After an initial value for the counter has been determined, changes (as a function of the counter value and an input variable) to the counter sometimes are accompanied by a need to propagate a signal to other devices in a network. The only operation performed on the counter is bit complementing; the function consists of scanning one of a half dozen tables of few entries. As a result, there is little complexity in the data structure and processing algorithm needed to emulate behavior of a flip-flop.

Appendix C

Modelling Tri-State Devices

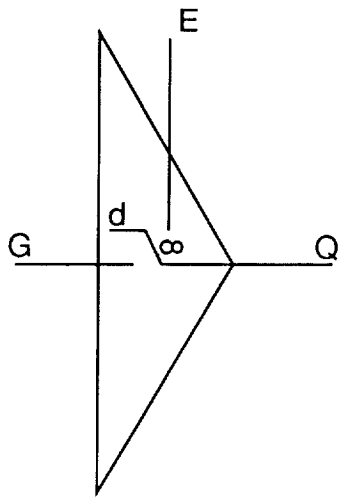
The algorithm for emulating tri-state devices in the diagnostic emulation technique is a combination of the algorithm for emulating gates (described in appendix A) and, with a slight modification, the algorithm for emulating flip-flops (described in appendix B). Accordingly, tri-state devices are represented by counters (as are gates and flip-flops), and a network of tri-state devices (and, of course, of gates, flip-flops, and tri-state devices) is represented by an identical network of counters. Also similarly, only some of the changes to the counter, caused by changes of its inputs, correspond to changes of the tri-state device's output values; when such changes occur their effects are propagated to other devices (i.e., counters) indicated by the network. Thus, the function of the algorithm is the same as that of the gate and flip-flop algorithms, namely, to change the value of a counter according to a transition rule and to indicate which, if any, other counters of the network are to change as a consequence.

Modelling of a tri-state device in the diagnostic emulation technique is based on the observation that the third state of a tri-state device effectively disconnects the tri-state device from the device(s) to which its output signal would otherwise propagate, and that forcing and holding the tri-state device's output to TRUE (or FALSE, depending upon the particular device) logically achieves the same (disconnection) effect. Thus, for modelling purposes, the third state corresponds to the condition of being *stuck* at one of the normal, binary values. There are two methods for doing this in the diagnostic emulation technique. One method consists of substituting a two-input gate (an AND or OR gate, depending upon the desired value of the disabled output) for a tri-state device in a network description. The second method replaces a tri-state device with a hypothetical gate–flip-flop device (for which the disabled output value d is a parameter).

The two-input gate is the simpler and relatively obvious method. The two inputs to the tri-state device (inputs G and E in fig. C1(a)) become the inputs to the gate that replaces the tri-state device. No special algorithm is needed since the normal gate modelling algorithm (see appendix A) already suffices to emulate a gate. An OR gate is chosen when the output of the modelled tri-state device must be TRUE to simulate disconnection. If the tri-state device is of a sort that disconnects when its *enabling* input signal is TRUE, then it is immediately consistent with modelling by an OR gate and the enabling input need only be connected as an input to the OR gate. If, however, the tri-state device is of a sort that disconnects when its enabling input signal is FALSE, then to be consistent with modelling by an OR gate the enabling input signal must be inverted before connection to the OR gate. (The data structure available in the diagnostic emulation technique for describing a network of devices contains a convenient capability for inverting (i.e., complementing) a signal at the input to a device gate or otherwise.) On the other hand, if the output of the tri-state device is to be FALSE when the device is disconnected, then the tri-state device is represented by an AND gate. In this case and in a manner analogous to that of the OR gate just mentioned, a tri-state device that is disconnected when its enabling input signal value is FALSE is consistent with representation by an AND gate and needs no inversion of the enabling input signal, while one that disconnects when the enabling input signal value is TRUE is inconsistent and requires that the enabling input signal be inverted.

The hypothetical gate–flip-flop method is somewhat more complex, but it provides an additional capability—modelling of stuck-at faults. The method is based on the observation that a tri-state device can be described by a truth table (as in figs. C1(a) and C1(b)) and, therefore, modelled by a state transition diagram as are flip-flops (as described in appendix B). As a consequence, the algorithm developed for flip-flops can be used by adjoining a tri-state device's enabling input signal as simply another input signal to the hypothetical flip-flop's comprehensive truth table. Since all the other inputs to a flip-flop are available to represent the G input of a tri-state device, the counter for holding the state index (see appendix B) is available for use as the counter for the gate algorithm (see appendix A). That is, the implementation details of the algorithms allow representation of a device which operates as if it were a gate (AND, OR, XOR, etc.) with an enabling input signal. Such a device can then be used to represent a conventional tri-state device (e.g., a one-input OR gate with an extra input signal, the enabling signal)—or it can be used to represent a conventional gate which can be forced, by the extra signal, to a predetermined output value (i.e., to act like a stuck-at gate).[37]

[37] Of course, the enabling input signal that is used to emulate a stuck-at fault of a conventional gate does not exist in the network being emulated. It comes from a fictitious connection added by the diagnostic emulation technique in order to introduce the fault.



G	E	Q
g	e	g
*	$\bar{e}$	d

*Don't care

(a) Device. (b) Truth table.

Figure C1. Components of tri-state device.

Appendix D

Modelling Busses

In the diagnostic emulation technique, a bus line is modelled as a gate.[38] Therefore, in a network of devices to be emulated, each bus line is replaced by a gate whose inputs are the signals that flowed into the bus line. The output of the gate, representing the signal seen on the bus line, becomes the signal source for all those device inputs that were connected to the bus line. The gate, of course, does not exist in the network being emulated; however, it is modelled just as any real gate of the network. Since the gate modelling method already described in appendix A suffices, no additional algorithmic or data structure complexity is needed for modelling bus lines.

[38] The term"bus line" can be misleading. A bus line is a junction point, an elongated point to be sure, through which signals travel from the outputs of some devices to the inputs of others. The use of a bus line requires that devices attached to it share the property that the signal representing one of the logical (binary) values is always dominant over any number of signals representing either the other logical value or other possible output states of devices (in particular, the signal corresponding to the third state of a tri-state device; see appendix C). Given this property, a network is ordinarily designed so that at any time *all but one* of the devices that output to a bus line (i.e., junction point) are in a *nondominant* (and, hence, effectively disconnected) state. Thus, seen as a signal source, a bus line appears to track the output value of the one device that is not necessarily in a nondominant state. Consequently, a bus line can be represented by either an AND or an OR gate, depending upon whether the FALSE or the TRUE logic value is dominant.

References

Abramovici, Miron; Levendel, Ytzhak H.; and Menon, Premachandran R. 1983: A Logic Simulation Machine. *IEEE Trans. Comput.-Aided Design Integrated Circuits & Syst.*, vol. CAD-2, no.2, Apr., pp. 82–94.

Anderson, T.; and Lee, P. A. 1981: *Fault Tolerance—Principles and Practice.* Prentice/Hall International, Inc.

Baker, Robert; Mangum, Scott; and Scheper, Charlotte 1988: *A Fault Injection Experiment Using the AIRLAB Diagnostic Emulation Facility.* NASA CR-178390.

Bavuso, S. J.; and Petersen, P. L. 1985: *CARE III Model Overview and User's Guide (First Revision).* NASA TM-86404.

Becher, Bernice 1987: *Diagnostic Emulation: Implementation and User's Guide.* NASA CR-178391.

Butler, Ricky W. 1984: *The Semi-Markov Unreliability Range Evaluator (SURE) Program.* NASA TM-86261.

Goldberg, Harold 1981: *Extending the Limits of Reliability Theory.* John Wiley & Sons, Inc.

Koike, Nobuhiko; Ohmori, Kenji; and Sasaki, Tohru 1985: HAL: A High-Speed Logic Simulation Machine. *IEEE Design & Test Comput.*, vol. 2, no. 5, Oct., pp. 61–73.

Lomelino, D.; and Iyer, R. K. 1986: Error Propagation in a Digital Avionic Processor—A Simulation-Based Study. *Proceedings Real-Time Systems Symposium,* Inst. of Electrical & Electronic Engineers, pp. 218–225.

Martin Marietta 1981: *Digital Avionics Design and Reliability Analyzer.* NASA CR-181641.

Migneault, Gerard E. 1980: Software Reliability and Advanced Avionics. *AFIPS Conference Proceedings, Volume 49–1980 National Computer Conference,* AFIPS Press, pp. 715–720.

Nagel, Phyllis M. 1978: *Modeling of a Latent Fault Detector in a Digital System.* NASA CR-145371.

Pfister, Gregory F. 1982: The Yorktown Simulation Engine: Introduction. *ACM IEEE Nineteenth Design Automation Conference Proceedings,* IEEE Catalog No. 82CH1759-0, Inst. of Electrical and Electronics Engineers, Inc., pp. 51–54.

Swern, F.; and McGough, J. 1982: *Description of the Box 930 Processor at the Gate Logic Level.* NASA CR-181642.

TRW Defense & Space Systems Group 1980: *Advanced SMITE Reference Manual.* RADC-TR-80-66, U.S. Air Force, Feb. (Available from DTIC as AD A087 743.)

Wensley, John H.; Lamport, Leslie; Goldberg, Jack; Green, Milton W.; Levitt, Karl N.; Melliar-Smith, P. M.; Shostak, Robert E.; and Weinstock, Charles B. 1978: SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proc. IEEE,* vol. 66, no. 10, Oct., pp. 1240–1255.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-4027		

4. Title and Subtitle	5. Report Date
On the Diagnostic Emulation Technique and Its Use in the AIRLAB	October 1988
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Gerard E. Migneault	L-16397
9. Performing Organization Name and Address	10. Work Unit No.
NASA Langley Research Center Hampton, VA 23665-5225	505-66-21-03
	11. Contract or Grant No.
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546-0001	Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This document is intended primarily as an aid for understanding and judging the relevance of the diagnostic emulation technique to studies of highly reliable, digital computing systems for aircraft. Therefore, the document contains a short review of the need for and the use of the technique as well as an explanation of its principles of operation and implementation. However, details that would be needed for operational control or modification of existing versions of the technique are not described.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Emulation Simulation Digital simulation Reliability analysis Failure analysis	Unclassified–Unlimited Subject Category 60

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	27	A02

NASA-Langley, 1988
For sale by the National Technical Information Service, Springfield, Virginia 22161-2171