27 & 28 Septembre 1972
TOULOUSE - FRANCE

colloque international

Conception et Maintenance des
Automatismes Logiques

(NASA-CR-130388) DESIGN OF AUTOMATA THEORY OF CUBICAL COMPLEXES WITH APPLICATIONS TO DIAGNOSIS AND ALGORITHMIC J.P. Roth (International Business Machines Corp.) 11 Apr. 1972 8 p CSCL 09B

N73-14201

Unclas G3/08 91519

SURVEY PAPER

J.P. ROTH
International Business Machines Corporation
IBM Thomas J. Watson Research Center
Yortown Heights, New York 10598

Design of Automata Theory of Cubical Complexes with Applications to Diagnosis and Algorithmic Description.

NTIS HC $3.00

DESIGN OF AUTOMATA
THEORY OF CUBICAL COMPLEXES WITH APPLICATIONS TO DIAGNOSIS AND ALGORITHMIC DESCRIPTION

by

J. Paul Roth

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: This investigation has been concerned with the following problems: (1) methods for development of logic design together with algorithms such that it is possible by means of these algorithms to compute a test for any failure in the logic design, if such a test exists; it is also concerned with developing algorithms and heuristics for the purpose of minimizing the computation for tests. (2) a method of design of logic for ultra LSI (Large Scale Integration) which seems to make LSI technologically feasible (failures may exist on the chip with correct operations still taking place) This method of design, called the Universal Function Schema, allows for instant Engineering Changes EC's as well as for complete functional changes in milliseconds of time. In addition this scheme provides a "universal card" which solves the so-called "stocking problem" as well as reduces the cost of production. (One must pay the penalty of about a 10 times increase in circuit count) (3) It has been discovered that the so-called quantum calculus can be extended to render it possible: 1) to describe the functional behavior of a mechanism component-by-component and 2) to compute tests for failures, in the mechanism, using the Diagnosis algorithm. In view of the large amount of extant electro-mechanical gear, this result may be of basic importance. (4) One of the original motivations for this investigation was the development of an algorithm for the multi-output 2-level minimization problem. A program MIN 360 (its new name) has been written for this algorithm. MIN 360 has options of mode (exact minimum or various approximations), cost function, cost bound, etc., providing flexibility. MIN 360 has been applied to some of the Jet Propulsion Laboratory's problems; its solutions seem already to have been incorporated into some of JPL's hardware.

This work has been sponsored in part by the Jet Propulsion Laboratory, California Institute of Technology with the support, under Contract NAS 7-100, of the National Aeronautics and Space Administration.

RC 3814 (#17297)
April 11, 1972
Computer Sciences
Mathematics

Section 1. Diagnosible Design Form and the D-algorithm

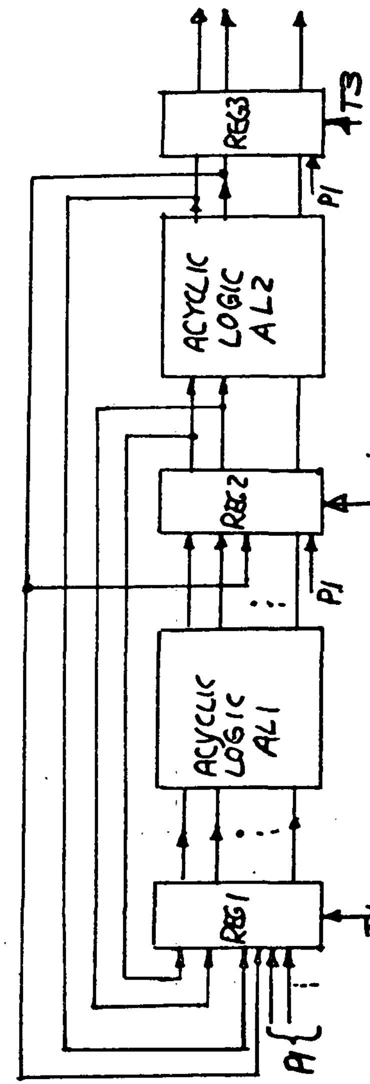
If one designs random asynchronous logic and attempts to develop tests for failures in this logic design, one is doomed to failure if the circuit is at all large. The reason for this is as follows: The D-algorithm (as with all the others of which we know) when confronted with asynchronous cyclic designs attempts to cut certain feedback loops in an adroit manner, so that the cuts appear to be the natural places in which to insert unit delays for all the cuts. One then generates tests for this simplified model of the original circuit, simplified in the sense of its timing behavior. Does the test for failure in the model also represent a test in the original more realistic design? In order to find out, one runs simulators which locate races and hazards. If the test, which in general is a sequence of patterns to the primary inputs of the circuit, has a race then it must be discarded and another attempt to compute a test for another failure. As the size of the circuit increases the frequency with which a "potential test" harbors races increases. The method of design, called Diagnosible Design Form, described in this section, is a procedure for the the design of logic in which it possible to realize any function; furthermore, one can prove that the D-algorithm (extended to these sequential circuits) will always compute a test for a failure if such exists.

System/360 model 40 and the system/370 Model 195 both used almost exclusively diagnosible design form.

Diagnosible Design Form may be described with reference to the figure below. The basic configuration consists of a bank of latches - call them registers - R1 which are gated at time T1. This R1 feeds acyclic logic L1 (hence combinational logic) which is followed by a bank of registers R2 gated at time T2. Feedback is allowed from R2 to R1. R2 in turn is followed by a section of acyclic logic L2, followed by registers R3 gated at time T3 with feedback allowed from R3 to R2 or R1. The difference T2 - T1 is chosen large enough so that all changes in the acyclic logic have taken place from T1 gating R1 before T2 is activated etc. Thus all races hazards are, by design, eliminated.

Our only remaining task is to show that the D-algorithm slightly modified may be easily adapted to computing tests (a deterministic test) for a given testable failure F1. In the first place, it may be noted that the obvious place to cut the feedback loops is from one bank of registers to another, thus there is no complicated choice which must be made.

It was stated that it could be guaranteed that the D-algorithm would compute a "deterministic" test for a given failure if one existed. By deterministic is meant that the tests, in general of sequences of input patterns, do not depend upon the signal in the feedback loops (state variables) at the time of application of the tests. One would have also to compute "D-sequences" and "singular cube sequences" for the latches, a trivial one-shot computation, and make sure that only such tests were used in the generation of a test.



PI denotes primary inputs

Figure 1 Diagnosible Design Form

The D-algorithm for the DDF will be termed DALGS.

Section 2. Heuristics for DALG

The computation for a "covering" ensemble for tests for very large circuits (say the order of 10,000 devices) become significant. The following two heuristics are offered with the purpose of reducing the computation. The first step in the procedure to compute an ensemble of tests to detect any failure of a given category in a logic design is to select a failure out of this given category. The first heuristic is a method for making such a selection. The procedure starts as follows: a failure for a primary input for the circuit is first chosen (we do not have a criterion for selecting among the primary input failures). One then uses the D-algorithm (or some variant or equivalent thereof) to compute a test for this failure. One then uses TESTDETECT to ascertain all failures detected by this test (in general a sequence of input patterns). One would wish to maximize the number of failures detected by a given test for this tends towards minimizing the total number of tests computed and hence minimizing the total computation. Thinking of the D-algorithm in terms of the D-chains developed in generating the tests, for a failure originating in the primary input (a PDCF) this chain drives through the entire logic and it can be shown that any line on this D-chain will be tested in one mode or another by this particular test being generated: for this choice the D-chain tends to be as large as possible since it is generated for a primary input. Next one selects some primary input

failure not yet tested if such exists, and repeats the same procedure. If no such primary input failure untested by the tests computed to date exists, one next selects a device fed by primary inputs and goes through the same procedure etc., until an ensemble of tests has been computed which covers all failures of the category agreed upon in advance.

It is estimated that the number of tests required by this mode of computation as opposed to the "counter strategy" of starting with the primary output failures and proceeding backwards to the primary input failures would be about 1 to 2.

The second strategy is a simple one which in complicated control circuits might be extremely helpful. For some of these circuits a sequence of reset input patterns is originally applied in order to render the design into a known and desired state. The strategy followed is thus to first apply the sequence of reset inputs which is known, and then to derive the D-chain emanating from the failing device in the usual way: this imposes a constraint on the development of the D-chain. It is likely that the D-algorithm even with this constraint will compute a test if it exists for the failure. Indeed it may be that it is necessary to apply the reset input sequence before one is able to get a test; the D-algorithm would not know this in advance and thus might take a great deal of time in generating the test if indeed it had enough computation time so to do. It is expected that this short-cut will substantially speed up computation time. The exact algorithm would then be followed by this and used only if the heuristic did not yield a test.

Section 3. Cyclic TESTDETECT

In the computation of an ensemble of tests to detect any failure of a given category in a logic design the procedure is usually as follows: (1) select a failure from a given category of failures and compute a test for it (e.g. by the D-algorithm) (2) use a simulator to determine all failures of the given category which are detected by this test. This procedure is continued until a test ensemble has been obtained which detects all the prescribed failures (or one has reached a certain modicum of completeness or "coverage", say 85%).

The bulk of computation time is in the simulation, perhaps by a ratio of at least 100 to 1.

TESTDETECT was developed as a "1-pass" simulator to determine all failures detected by a given test. It was designed originally for acyclic (combinational) logic. It had not been seen how to the generalize TESTDETECT to the general cyclic circuit case. A study of the Diagnosible Design Form, however, shows that if the logic design is clocked by a clock having at least two phases then it is rather simple to extend TESTDETECT to cover DDF-logic.

In order to describe cyclic TESTDETECT we will first review how it works for acyclic logic. Assume that we have a test consisting

of a single wave of patterns applied to the primary inputs of the circuit. First one computes the signal on each line (device) in the logic design. Let us now look at the primary output. If a given output has a "0" assigned to it for this input pattern, then clearly it is tested for stuck-at-1. Likewise if it has a 1, then it is tested for stuck-at-0. Assume that a given primary output is a 2-input OR with inputs 1 and 0 respectively. It is clear that the line with input 1 is tested for stuck at 0 whereas the input with value 0 will not be tested by this input pattern. We proceed in this way from the primary outputs ascertaining whether or not the line (device) is detected in its failure by this test, passing through the entire circuit to the primary input. This is thus a 1-pass-type of simulator. TESTDETECT for the acyclic case has been programmed in APL and run extensively. (Roth, Bouricius & Schneider, 1967)

TESTDETECT takes the order of n steps for a piece of logic having n devices. A simulator requires n^2 steps: for each failure one must determine the signal on each line of the circuit: n failures multiplied by n devices equals n^2.

Now we will outline the procedure for the case of the cyclic circuit and use the figure 2 as an illustration. Initially we assume that all lines have a signal which is unknown - let this condition by represented by the symbol x. In the case of cyclic logic the test will in general be a sequence. Assume that we apply the first input pattern to the primary inputs with all other lines being x. In general, some lines will be determined immediately by this primary input pattern. On the other hand there will be some for which this is not the case, for example, suppose that a given line is the output of an OR with two inputs, one having the value x and other, the value 0. Figure 2 shows a cyclic circuit (the registers required for DDF are omitted for simplicity). Figure 3 shows cyclic TESTDETECT operating for the first input pattern t1. Figure 4 shows it for the second t2.

The general procedure is as follows:

(1) The resulting signal 1 or 0, on each line if determined by t1 is computed; if not determined, it is assigned to value x. Clearly the input pattern can test no line assigned x.

(2) One then reasons backwards from the primary Output lines (here only line 10): if the test pattern causes a signal a, a = 0 or 1, on the PO line, then clearly this line is tested for the failure stuck-at-$\bar{a}$.

(3) Having determined failure detection for the PO lines, we next examine the lines ℓ feeding PO blocks (lines). If the PO block is an OR or NOR then ℓ is tested only if the signals on all other inputs to the block is 0; if an AND or a NAND, then all other lines feeding the block must be 1. If such a line is testable then it is tested for $\bar{a}$ if t induces a signal a on ℓ.

(4) This procedure is pushed level by level. If a line fans out to

more than one block then one must, as with acyclic TESTDETECT, proceed forward from the line to the point(s) of reconvergence. Nevertheless the process is very rapid.

(5) Having made the determination for the signals 1, 0 or x which the first input pattern causes, plus the ensemble of failures detected by t1, one similarly applies the second input pattern t2 to determine, together with the signals established on the feedback loops by t1, the signals induced on the lines, together with as above, the failure detected by t2-preceeded-by-t1.

(6) One continues similarly through the entire given sequence of test patterns, t1, t2,..., to determine all failures detected by this test.

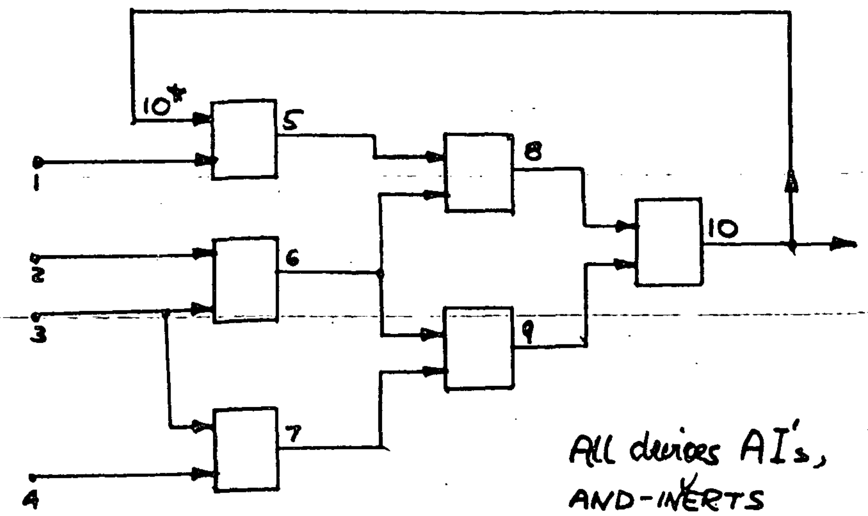


Fig 2 CYCLIC TESTDETECT – example

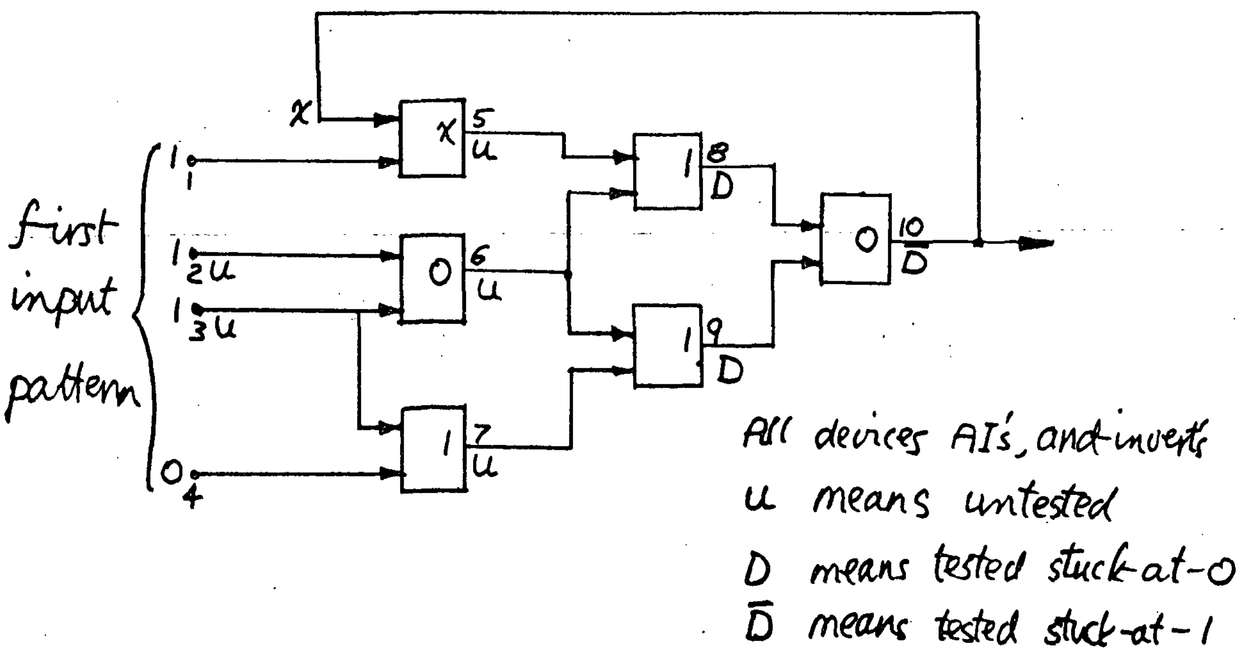


Fig 3 CYCLIC TESTDETECT – example.

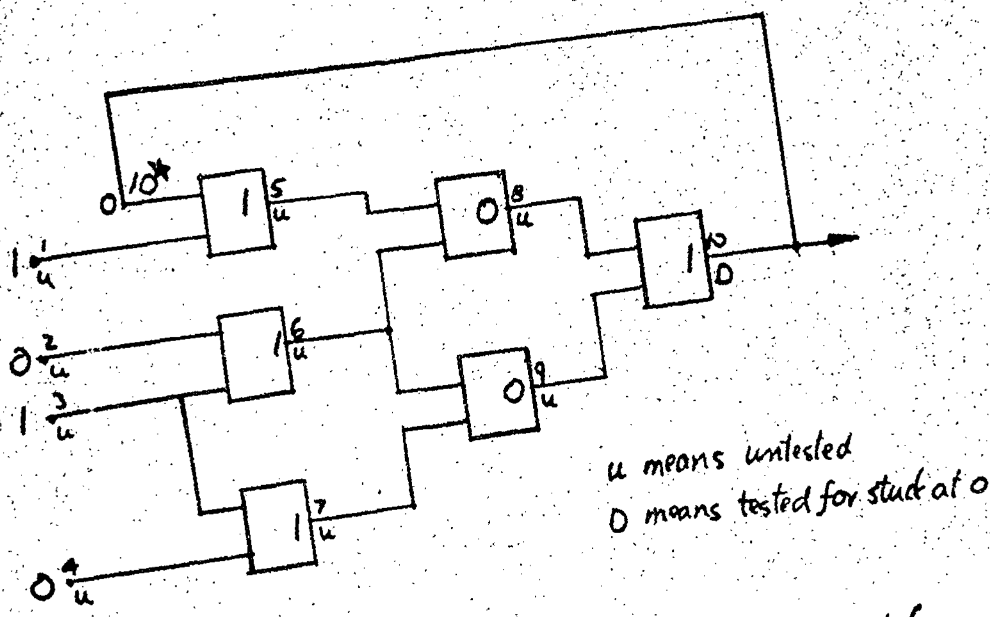


Fig 4 CYCLIC TESTDETECT Values assumed by second input pattern