Fault Simulation Basics

Zycad Corporation
1380 Willow Road
Menlo Park, CA 94025
(415) 688 - 7400

ABSTRACT

Fault simulation is a powerful yet not well understood tool for generating test vectors. This tutorial describes the principles of fault simulation, fault modeling techniques and the economic benefits of fault simulation. The section on principles of fault simulation includes serial, parallel, and concurrent fault simulation algorithms along with descriptions of digital faults, fault coverage, and fault mechanisms typically found in digital circuits. The modeling technique section presents the trade-offs of fault placement, fault collapsing algorithms, and the ability to discover different physical defects through fault simulation. Economic benefits are shown through empirical examples and through correlating fault coverage to average defect levels resulting after manufacturing test.

FAULT SIMULATION OVERVIEW

The primary types of simulation used in Computer Aided Engineering (CAE) are logic and fault simulation. Logic simulation uses computer models of an electronic design to imitate the behavior of a physical system. It allows the engineer to know if the design will work the way it is supposed to work before the product is manufactured. Logic simulation is the only practical alternative to the time-consuming and costly practice of actually building and debugging physical prototypes. Once a design has been properly logic simulated, fault simulation is the next critical step.

All manufacturing processes create good and bad parts. In fact, it is not unusual for yields to average between 30% and 50% for integrated circuits. As Figure 1 shows, it is up to the test process to screen out these bad parts so they do not get to your customer.

Fault simulation helps you screen out bad parts. It is a tool that defines how well the test vectors used to verify a finished product will catch manufacturing defects. A perfect test vector set would be able to detect any defect in the physical chip.

When you have verified a circuit and fully completed its logic simulation, you have a full description of how a defect-free version of the circuit operates. However, will you be able to know when a part is defective? As Figure 2 shows, fault simulation uses a good version of the circuit as a reference, and systematically inserts faults into an identical copy of the circuit to check whether the test stimulus can detect differences between the two circuits' output.

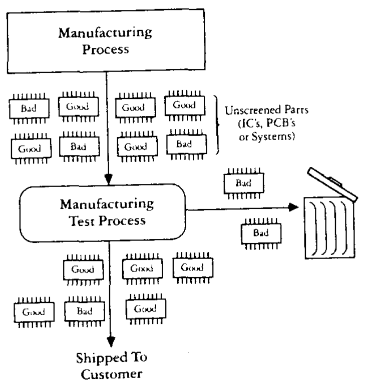
Manufacturing Process
Bad
Good
Unscreened Parts (IC's, PCB's or Systems)
Manufacturing Test Process
Shipped To Customer

Figure 1. The Problem: How to screen out all bad parts?

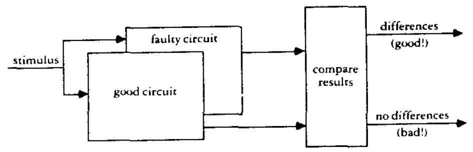

Figure 2. The Fault Detection Process

The ratio between the number of faults detected by the test vectors and the total number of faults is called Fault Coverage Percentage. As this number approaches 100%, the potential for defective circuits to pass manufacturing tests approaches zero. Thus, a set of test vectors that catches less than 100% of the possible manufacturing faults allows for defective parts to pass through the testing cycle and potentially create expensive consequences when failures occur in the field.

There are usually several stages of the testing cycle in-house, including I.C., board, and system test. Each successive stage provides an additional screen to catch defective parts. The better the test vectors, the earlier a defective part will be discovered, and as Figure 3 shows, the less costly it will be.

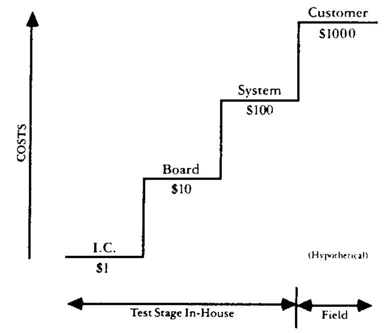

Figure 3. The Cost vs. Time Relationship for finding Manufacturing Defects

For commercial-quality TTL circuits, the typical level for defective parts shipped, is 0.01%. This means that one of every ten thousand parts shipped is defective. It is not unusual for IC's and PCB's to have a defect level of 1%, 5%, or even higher. A higher defect level results in a greater number of field failures which can translate into high repair costs and adversely affect customer confidence. Table 1 shows estimated field-repair savings at selected defect levels for a run of 10,000 parts based on an average repair cost of $800 per defective part.

Total Parts Shipped	Defect Level	Defective Parts	Repair Cost Per Part	Total Cost
10,000	5%	500	$800	$400,000
10,000	1%	100	$800	$80,000
10,000	0.1%	10	$800	$8,000
10,000	0.01%	1	$800	$800

Table 1 Estimated Field-Repair Costs for Selected defect level Percentages

In this example each 1% improvement in defect level results in an $80,000 savings. This illustrates the significant savings which can result from reducing the number of defects which reach your customers. The question is, how does increased fault coverage affect field defect levels?

A joint study done by Motorola and Delco Corporation[1] shows that 50% fault coverage (the norm for good hand-generated tests) resulted in a defect level of 7%, that 90% fault coverage yielded a 3% defect level, and that it took a full 99.9% fault coverage to achieve 0.01% defect level. Figure 4, taken from the Motorola/Delco paper, shows these relationships graphically.

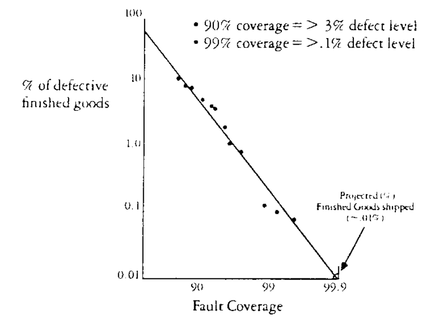

Figure 4. The Relationship of Fault Coverage to Defect Level

Figure 5 shows the results of an experiment used to determine the correlation between fault coverage and defect level. In this experiment, 10,000 parts were tested from a process with a 30% yield. In the first case, only part of the test vector corresponding to a 50% fault coverage was run. Of the 7,000 bad parts, 6,773 were detected. The remaining 227 undetected defective parts yielded a 7% defect level for the 3,227 parts actually shipped. The experiment was repeated for 90%, 99%, and 99.9% fault coverage. These results agree with the Delco/Motorola results—a 0.1% defect level corresponds to 99% fault coverage, and a 0.01% defect level corresponds to 99.9% fault coverage.

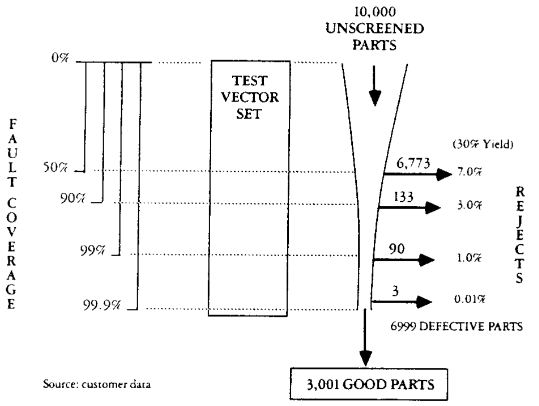

Figure 5. Increasing Fault Coverage to get Lower Defect Levels

Although it is desirable to achieve the highest possible fault coverage on your designs, optimally 100%, the costs and time associated with this type of analysis are often prohibitive. It is not unusual for a serial fault simulator running on a 1-MIP mainframe to take months to fault simulate a 10,000 gate circuit. Today, with advanced algorithms and specialized hardware, months can be reduced to days or even hours.

FAULT SIMULATION IN THE DESIGN AUTOMATION PROCESS

There are several uses of fault simulation in the design automation process. Three are listed below:

- Fault simulation can be used as an accurate testability metric early in the design process. Assuming comprehensive test patterns, "grading" the design verification patterns can provide a useful insight into the testability of the circuit.
- The most common use for fault simulation, the one described in this handbook, is using fault simulation to define how well your test vectors will catch manufacturing defects.
- Another use of fault simulation is to "grade" a vendors test suite. For example, a 99% detection rate, could justify less rigorous incoming inspection on the parts.

FAULT ALGORITHMS

There are three common algorithms used for fault simulation—serial, parallel, and concurrent.

Serial Fault Simulation

Serial fault simulation, illustrated in Figure 6, is the simplest technique for fault simulation. Two complete copies of the circuit are stored in memory. A single fault is then inserted into one circuit, both circuits are simulated, and their output results are compared. If the output results differ (which is what we want) then the test vectors are said to have "detected" that fault. A fault is considered "undetected" if none of the test vectors create a difference in output response between the good and faulty circuits. Upon detection, the fault is categorized, a new fault is chosen, and the process above is repeated. Serial fault simulation is a very simple but slow method of fault simulation. Even with a hardware accelerator, runtimes for circuits with greater than 10,000 primitives can be prohibitive.

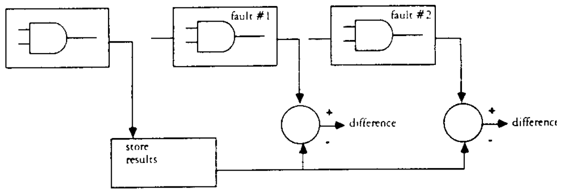

Figure 6. Serial Fault Simulation

Parallel Fault Simulation

Parallel fault simulation is used by many software-based fault simulators. Figure 7 shows how parallel fault simulation uses several *complete* copies of the circuit. One copy is the good machine and each of the other copies is a particular faulty machine. Each faulty machine is simply a complete copy of the good circuit with one unique fault placed in it. The number of parallel machines is usually small, for example 4 to 16 machines, depending on the word size of the computer, and the required accuracy in modeling state and strengths. This algorithm is typically faster than serial faulting because multiple faults are run at the same time. However, the parallel algorithm must continue the simulation until every parallel fault is detected; whereas, the serial algorithm can stop each fault simulation as soon as the fault is detected.

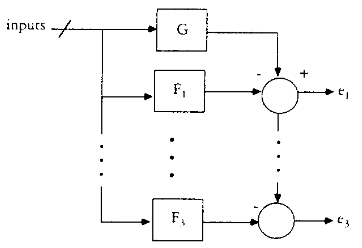

Figure 7. Parallel Fault Simulation

Concurrent Fault Simulation

Concurrent fault simulation is the most powerful of the three algorithms. It is based on the concept that a particular fault placed in a circuit does not affect the behavior of the majority of the rest of the circuit. Thus, multiple copies of a complete circuit need not be placed into memory at one time. Instead, memory is allocated on an "as needed" basis. This allows hundreds or thousands of faults to be simulated at once.

The concurrent algorithm simulates the circuit for the good machine and, when the inserted faults make the results of the good machine and the faulty machines differ, it diverges (copies) those devices and simulates them separately. Figure 8 shows an illustration of the concurrent fault algorithm. The small solid squares shown inside each of the faulty machines, $F_1 \ldots F_{100}$ indicate that only that portion of the faulty machine behaves differently from the good machine. For example, if a fault were inserted near the end of a chain of inverters, only the last few inverters would behave differently from the good machine, and consequently only those affected inverters would be diverged and simulated separately from the good machine.

The concurrent algorithm presents a very complex bookkeeping problem that fortunately is well suited to a special-purpose computer. It's great advantage is that it is a very fast method for fault

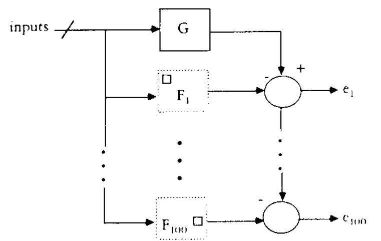

Figure 8. Concurrent Fault Simulation

simulation, and requires less memory than either the serial or parallel algorithms. Figure 9 provides an illustration of the relative memory and runtime requirements for these three methods.

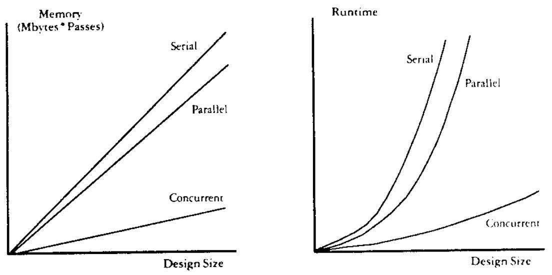

Figure 9. Memory and Runtime Requirements for Fault Simulations by Algorithm Type

The concurrent algorithm is the most efficient way of running fault simulations. This algorithm can be embedded into hardware creating an unbeatable combination. Hardware simulation accelerators run hundreds of times faster than software running on a general-purpose computer. Even running against a workstation rated at 10-MIPs, a fault simulation might run for 2 days versus only 3 hours on a hardware accelerator. This speed improvement is especially important because fault simulation is an iterative process, and quick turn-around time is essential for maximizing engineering productivity.

Simulations do not require the traditional computations that general-purpose computers excel at, like mathematical operations or floating point processes. Instead, it is more of a data manipulation problem that needs large memory resources and high data bandwidth. Simulation is easily adapted to parallelism and thus is a great candidate to accelerate with specialized hardware.

FAULT SIMULATION IN-DEPTH

This section contains more details than the first. It is intended to provide an overview of the finer points of fault simulation by defining often-used terms and algorithms. It describes different methodologies for fault modeling, fault seeding, and fault collapsing. Efficiency tools for fault simulation, are discussed, as well as defining how to interpret fault simulation results.

Circuit Defects

Circuit defects in integrated circuits are commonly caused by missing implants, oxide defects, metal shorts and opens, junction defects, and lithographic defects. As Figure 10 shows these defects are physical in nature. Trying to test the actual physical defects would be too complex a procedure. Empirical evidence has shown that an effective way to test circuits is to observe the circuit's behavior and test the physical character using a logic model of the circuit's performance.

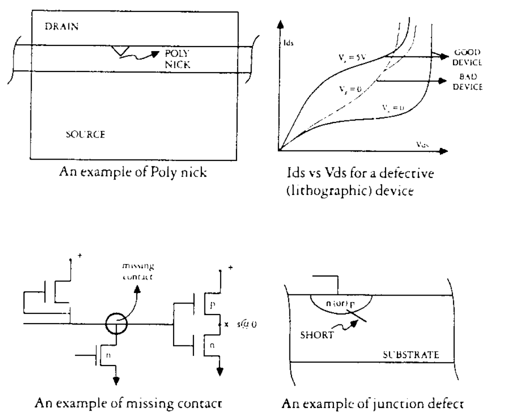

Figure 10. Sources of Circuit Defects[2]

In short, the key to fault modeling is not whether the model matches a particular physical failure, but that there is a high correlation between detecting defects with fault simulation and the actual test applied to the parts. Central to fault modeling is the concept of stuck-at faults.

Stuck-At Modeling

Most fault simulators use the logic stuck-at-0 and the stuck-at-1 model to represent physical defects. These faults can be independently placed on every input, output, or node in a system, and then simulated with a test vector as input to determine if their effect on the behavior of the circuit is observable at the designated test points. Other models exist, including the stuck-open and stuck-short models. However, these models are very compute intensive and studies have shown that there is little marginal gain from their use. Figure 11 depicts the standard stuck-at faults.

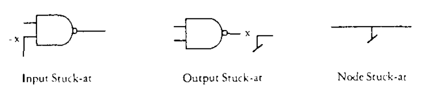

Figure 11. Standard Stuck-at Faults

The stuck-at models for input and node faults are simple. An input fault sticks the input pin to a logic one or logic zero but has no other effect on the node (a node is also called a signal on a net). A node fault sticks the whole node and each input it drives, but the outputs it is driven by have no effect on it.

There is no general agreement on how output faults should behave. There are several possible drive strengths available, which we shall call—Fixed, Rail, and Strength. These are summarized in Table 2. For a Fixed output fault model, the nodes will be driven with a fixed strength value. This type of output fault can fight with other outputs driving the same node. For the Rail output fault model, the output behaves as if it were shorted to power or ground. Like node faults, other outputs driving this node will have no effect. For the Strength fault model, the node will be driven with a strength value derived from the gate. For example, if the gate can pull up with a resistive strength, and pull down with an active strength, the output s@1 value would be "resistive-1" and the output s@0 value would be "active-0." In either case, the output fault can fight with other outputs driving the same node.

Model Name	Node Behavior	Result
Fixed	Nodes are driven with a fixed strength value.	Outputs driving the same node can fight.
Rail	Nodes behave as if shorted to power or ground.	Other outputs driving the node have no effect.
Strength	Nodes are driven with a strength value derived from the gate type drive strength.	Outputs driving the same node can fight.

Table 2 Output Fault Models

Levels of Faulting

For any given design, there can be many levels of logic simulation including architectural, behavioral, functional, gate, and switch. For fault simulation, it only makes sense to fault structural implementations of a circuit, since faults are actually models of structural, or physical, defects within a system. For mixed-level simulations, faults can be placed at the boundaries of a behavioral model, but not usua'ly inside one.

Ideally, faults should be placed at the lowest level where there is doubt that the physical implementation of the device is perfect. In practical terms, the lowest level faults could be placed in a circuit is at the switch level. Faulting at the package level would be appropriate with printed circuit boards assuming that the packages have been thoroughly screened. Faulting at the gate level would be appropriate for gate arrays, or where the exact switch-level implementation of a gate was unknown. This is the level that most vendors require their customers to fault simulate at before accepting a design for fabrication. However, faulting at high level cells or macros can yield overly optimistic fault coverage numbers. Vectors that provide a 95% hard detection rate at the macro level might only provide a 50% detection rate at the switch level. Therefore, faults should be placed at the level where you expect to have failures.

In any case, a fault simulator should allow you to easily define the level that faults are seeded within a design.

FAULT COLLAPSING

Fault collapsing is an automatic feature provided by many fault simulators. Faults seeded at different points in a circuit may produce identical behavior. These faults are said to be equivalent to each other. (There also exists a class of dominant faults that aren't equivalent.) To save simulation time, equivalent faults are collapsed together, and only one fault from each equivalence group is actually simulated.

There are two methods of identifying equivalent faults—gate collapsing and node collapsing. Gate collapsing defines when an input fault can be collapsed to an output fault on the same gate. Node collapsing governs when faults on the same node can collapse together. In other words, node collapsing determines when an output fault can collapse to a node fault, or a node fault can collapse to an input fault. Figure 12 shows two buffers and how the input, output, and node faults can collapse together.

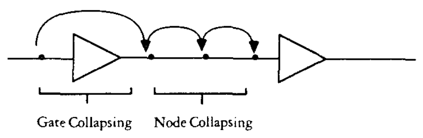

Figure 12. Gate and Node Collapsing

Gate Collapsing

Gate collapsing is based on the logical nature of particular gates. For example, Figure 13(a) shows a two-input AND gate that has 6 possible stuck-at faults—s@0, s@1 on each input, and s@0, s@1 on the output. Since a 0 on any AND-gate input forces the output to a 0, the s@0 input faults are said to be equivalent to the s@0 output fault. In other words, the AND gate behaves the same whether an input is s@0, or the output is s@0, so there is no need to simulate the input faults. The two s@0 input faults shown in Figure 13(a) can be collapsed to the output s@0 fault, which results in the 4 faults shown in 13(b).

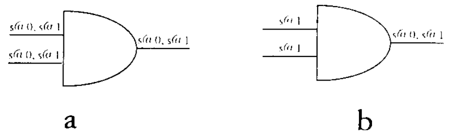

Figure 13. Gate Collapsing. (a) An AND gate with six stuck-at faults. (b) The four equivalent faults.

In this example, fault collapsing decreases the number of faults to be simulated by 33% (6 faults have been reduced to 4). Each primitive gate has its own rules for fault collapsing. Although collapsing on a single primitive only slightly decreases the number of faults, total savings from collapsing all primitives can significantly reduce run times.

Node Collapsing
Node collapsing is governed by rules of how a fault will affect other faults on the same node. Figure 14 shows two simple cases. In Figure 14(a), the output fault of the left buffer can collapse with the node fault, which in turn can collapse with the input fault of the right buffer. In Figure 14(b), the output fault of the left buffer can collapse with the node fault, but the node fault cannot collapse with the two input faults.

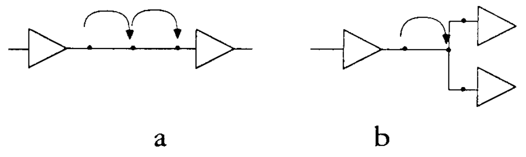

Figure 14. Node Collapsing. (a) The output and node faults can collapse to the input fault. (b) The output fault can collapse to the node fault only.

Node collapsing is controlled by the fan-in/fan-out of each node. In general, a single fan-in node will allow the single output fault to collapse to the node fault, and single fan-out nodes will allow the node fault to collapse to the single input fault.

Strength Sensitivity
Device drive strengths also become a consideration in fault collapsing when designing with transistors, and when more than one device drives a node. Figure 15 shows a buffer driving a transistor. Since the transistor output depends on its input strength value, the strength of the output stuck-at at point B, or the node stuck-at at point C could have an effect on the circuit behavior. For example, the transistor output might fight with other transistor outputs. The strength of the transistor output will determine how the contention is resolved, and thus the final output value. If the faults at B or C have different strengths they cannot be collapsed together. Moreover, the input fault at A cannot be collapsed with B unless the output fault has the same strength as the normal gate output.

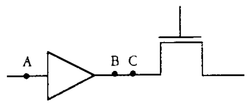

Figure 15. Fault Collapsing and Strength Sensitive Devices

The complete rules for collapsing are a function of the gate types involved, the fan-in/fan-out at each node, the output fault type, and strength sensitivities.

For the circuit in Figure 16 the following rules apply: inputs collapse to outputs, outputs collapse to nodes, and nodes collapse to inputs. However, not all circuits are as simple as this. Besides a one-to-one (1:1) fan-in/fan-out, there may also be a one-to-multiple (1:M), a multiple-to-one (M:1), or a multiple-to-multiple (M:M) fan-in/fan-out.

Figure 16 illustrates these possibilities and each of the points at which a fault origin can be placed.

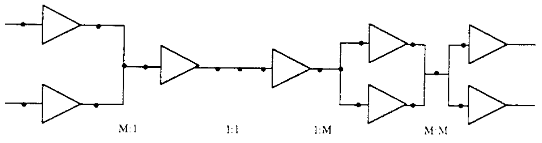

Figure 16. Possible Circuit Fan-In/Fan-Out Combinations

Accurate fault collapsing requires consideration of circuit topology and output fault type. If the output fault type (discussed earlier in this section) is Rail then an output fault can collapse with a node fault regardless of the fan-in/fan-out of the node. If the output fault type is Strength then an input fault can always collapse to the output regardless of the fan-in/fan-out of the node. Finally, if the output fault type is Fixed then an input fault can only collapse to the output if there is a single fan-in. The number of faults that collapse can depend on the output fault type.

Deterministic Versus Probabalistic Fault Simulation
Deterministic fault simulation uses the process described earlier in this handbook. Each node in a circuit has faults placed on it, test vectors are applied against the good circuit and the faulty circuit, and test points are observed to determine if there are differences in the output. Next, each fault is categorized as being detected or not. If unhappy with the coverage, the user writes additional test vectors specifically targeted towards undetected faults, then reruns the fault simulation in an effort to increase the hard detection rate. Deterministic fault simulation provides the tools to systematically improve fault coverage. Its drawback is that it is a very compute-intensive task.

Probabalistic fault simulation takes an indirect approach to defining fault coverage. Instead of placing faults on each possible node in a circuit, and then simulating, certain types of analysis are performed on the circuit instead. Probabalistic fault simulation runs faster than deterministic and works well for rough estimates of coverage. The drawbacks are that if the user needs to know exactly what faults are undetected, or how to compress and optimize the test suite, it cannot be done. This algorithm also breaks down at higher fault coverages. The variance in a probabalistic fault result will make the effort to extend coverage to 99% meaningless.

Efficiency Tools for Fault Simulation

Reducing the time it takes to do fault simulation on a circuit is important. Fault simulations, even using hardware acceleration, can take days to complete, and must be run multiple times to refine the vector set. Fault collapsing, described above, is one means to compress the simulation time. Other time saving tools can also be used, as discussed below.

Fault Simulation using Statistical Sample
Statistical fault simulation is an alternative to a full simulation run that exhaustively simulates every fault origin in a circuit. Using statistical simulation, one can simulate a small random sample of faults for a circuit and extrapolate the results. This assumes that faults are chosen randomly and that there is a uniform fault density.

Toggle Test
A toggle test applies the set of vectors to the circuit and watches for nodal activity. If there are nodes that do not change state during the simulation run, they are placed on a separate list. These unactive nodes should be addressed before starting a fault simulation.

Observability and Controllability
Observability is the ability to observe a circuit's behavior from the primary outputs. Controllability is the ability to control a circuit's behavior from primary inputs. These attributes have a significant affect on how hard or easy it is to develop test vectors that result in good fault coverage. If a given node is unobservable, there is no point in simulating faults on that node because they will never be detected.

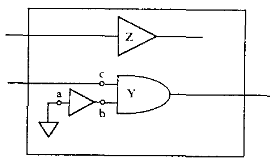

Figure 17. An Example of Non-Observable and Non-Controllable Nodes

For example, device **Z** in Figure 17 is not observable because its output does not connect to any primary output. Any faults on device Z are impossible to detect. Node **c** in Figure 17 is also not observable because the other AND gate input is always 0. Nodes **a** and **b** are not controllable to 1, since they will always be 0. Therefore there is no point in simulating a stuck-at-0 fault at **a** or **b** because the faulty machine and the good machine would always have identical behavior, only stuck-at-1 faults are interesting at those nodes.

Unobservable nodes are common in many circuits. For example, circuits often use only one of the **Q** or **Qbar** outputs from a D flip-flop. One output may be unused and should not be faulted. Faults which are not observable or controllable should be eliminated from the list of faults. This increases fault simulation speed, plus gives a more meaningful fault coverage result.

Incremental Simulation
Building a good test vector set takes several fault simulations. It is a process of iteratively fault simulating, adding vectors, fault simulating, adding more vectors, fault simulating, and so forth. To build efficiency into this process it is advantageous to reuse previous fault simulation results, and only run certain classes of fault results (like previously undetected faults), or certain portions of the circuit. This reduces the fault simulation task to an incre-

mental process of testing for only those faults not previously detected.

FAULT RESULTS

The results of a fault simulation tell if a faulty machine propagated a different output result than the good machine. A fault simulation generates one of seven possible fault detection results for each fault:

Definite (or hard) detect. This means that the fault was detected during the simulation—the faulty machine had the opposite logic value at an output pad as compared to the good machine.

Possible (or potential) detect. This means that the fault may have been detected. At some point in the simulation the logic level for the good machine was a 0 or 1 and the logic level for the faulty machine was an unknown (X). Since an X represents a 0 or a 1 it is not clear if the fault was really detected or not.

Soft detect. This is a form of possible detect, in which the good machine had both a 0 and 1 logic level while the faulty machine was unknown (X). The implication is that on a tester, a real part with that fault would likely be found.

Undetected. This means that the fault was not detected during the simulation.

Oscillatory. This means that introducing the fault caused the faulty machine to oscillate. The fault is neither detected nor undetected, but rather marked as a problem area to be addressed separately from the other fault results.

Hyperactive. This means that the fault diverged more than a specified percentage of the circuit. This type of result typically occurs when an X gets on to some global control line like a clock line and causes large portions of the circuit to have Xs. Thus the fault is no longer worth simulating because it would take too long and further simulation is probably uninteresting. Like the oscillatory result, hyperactive implies that the fault is neither detected nor undetected, but rather a problem area to be addressed separately from the other fault results.

Impossible fault. This means that the fault cannot be detected. For example, it is impossible to detect a stuck-at-0 fault on a grounded input pin or an unconnected output of a logic element. This type of result comes from a static analysis of the circuit, and is not strictly the result of a fault simulation.

Fault Dictionaries

Fault dictionaries are a valuable tool for locating defective portions of a product, especially for technologies that allow repair to a defective part, such as printed circuit boards. They are also useful for failure analysis of ASIC's and Full-custom IC's.
A fault dictionary is generated after a fault simulation is complete. It is a listing of every detected fault in ascending order of the time it was detected, along with its expected and actual testpoint outputs. This dictionary can become quite voluminous.

Besides their bulkiness, the effectiveness of fault dictionaries is limited by the models used. The stuck-at model is used not because it exactly represents physical reality, but because it directly correlates with locating failures in a system. A fault dictionary, therefore, cannot contain a list of all the possible defects that could occur within a printed circuit board or integrated circuit. It should, however, be able to localize where a problem with the circuit may be.

Fault Simulation Process

There are many valid fault simulation methodologies. One recommended top-down approach is described below.

1) The first step in any fault simulation is to be sure that the logic simulation is successfully completed. Examine the logic simulation results, or run a toggle test, to make sure all nodes in your circuit were active at least some time during the simulation. If not, these nodes are undetectable.

2) Follow this with a statistical run. Since fault simulations usually take several passes, and since faults are seeded randomly per pass, a statistical test could simply comprise of monitoring the results of the first few passes of a full deterministic run. If unhappy with the results, stop the simulation and add more vectors. If satisfied, let the full run continue.

3) If the circuit is large (>64,000 primitives depending on the circuit) the circuit and the test suite should be partitioned. By seeding only one section of the circuit at a time and directing a set of test vectors specifically targeted for that section, the sum total of fault simulating all the sections will be a lot less than if the whole circuit was simulated at once.

4) A full deterministic fault simulation run could start by seeding node faults, unless accuracy is important, then input and output faults should be seeded. Once the run is complete, move on to step 5 if the coverage results are acceptable, if not, look at the potential and undetected lists and write vectors specifically targeted to detect those faults. (This is the hard part, and where test engineers really earn their pay.) Rerun the fault simulation with this larger test suite against the potential and undetected faults from the previous run. Continue with this cycle until the percentage of definitely detected faults is at your specified target level.

5) Optimize your test vectors. Histograms will plot out the number of faults found per test vector. If there are groups of vectors that do not detect any faults and are not setting up the circuit to a particular state, then they can be eliminated. This will save time on the manufacturing floor during functional testing. Empirical evidence has shown that some test suites can be reduced by as much as 50% without affecting fault coverage. If there are several test suites, there are programs available to define their optimal order for functional testing, this again will save time in manufacturing during testing.

Automatic Test Pattern Generation (ATPG)

Automatic Test Generation (ATG) is the computer driven synthesis of the test vectors used to screen out the defective parts created during the manufacturing process. Manual test vector generation by a test engineer is difficult, time-consuming and tedious. There has been a great deal of interest in the prospect of automating this step. Unfortunately, there have been several roadblocks delaying the acceptance of ATG systems.

When the first ATG packages were developed, designs were typically very small and simple. These early attempts at automation ran very poorly when designs grew and became more complex. Test generators were ineffective with most designs because they could not traverse through the time-layers of sequential logic and soon fell out of favor.

More recently, new approaches for test generation have been developed to handle larger designs with sequential logic. These fall into two categories; those that require changes to the design structure (SCAN-SET) and those employing new ATG algorithms.

SCAN-SET implementation effectively reduces complex sequential structures to combinational planes of logic. This requires additional logic and can cause some speed contraints but this technique has been reasonably successful.

The most significant changes to the ATG algorithms come from the research done at Westinghouse Electric, in the mid 1970's. This new approach effectively handles layers of deep sequential logic by a clever technique that works backwards through the design, setting up the needed sequential states as they are required.

The addition of hardware simulation accelerators to the ATG process further enhanced these techniques. Vector generation and fault simulation, performed quickly in a closed iterative loop, combine to produce more efficient test vectors in a much shorter time.

[1] "Logic Fault Verification of LSI: How It Benefits the User," Richard A. Harrison, Ronald W. Holzwarth, and Philip R. Motz of Delco Electronics Division, General Motors Corporation; and R. Gary Daniels, James S. Thomas, and Warren H. Wiemann of MOS IC Division, Motorola, Inc., *Proceedings of the WESCON Professional Program* (September, 1980).

[2] "Defect Analysis and Fault Modeling in MOS Technology," R. Chandramouli and H. Sucar, *International Test Conference Proceedings* (November, 1985).