ATPG for Ultra-Large Structured Designs

John A. Waicukauski, Paul A. Shupe, David J. Giramma, and Arshad Matin

Mentor Graphics Corporation
8500 S.W. Creekside Place
Beaverton, OR 97005-7191

Abstract

A fast fault simulator is optimally combined with a powerful test generator to form an ATPG system that can efficiently create a high coverage test for extremely large scan designs. This ATPG system successfully tested or proved redundant all faults of the ISCAS85 and ISCAS89 benchmark designs with significantly better performance than previously published results.

Introduction

The creation of a high quality test for the rapidly increasing size of todays designs is straining the capabilities of current Automatic Test Pattern Generation (ATPG) systems. ASIC chips may contain 100,000 logic gates which require a test coverage of over 99% to achieve the necessary quality. Test strategies which were adequate in the past can no longer handle this problem, and test has become a limiting factor in the ability to manufacture chips.

Faced with the problem of satisfying the demands of increased quality for larger designs, many designers are accepting design-for-testability features which improve the ability of ATPG systems to create a test. One of the most common design-for-test strategies is scan design [1,2] which structures a general design to behave as combinational logic. While this improves the ATPG process, current ATPG systems can not adequately handle designs that will soon be approaching a million logic gates. Current ATPG systems either do not fully exploit the combinational nature of a scan design or effectively utilize the new technology developments in fault simulation and test generation.

This paper describes an ATPG system that has both the performance to handle these large designs and the capability to consistently achieve greater than 99.9% test coverage (excluding the proven redundant faults). This is accomplished by optimally combining a fast fault simulator with a powerful test generator that implements the most effective features described in recent literature. The results of an experiment showing the effect of using random pattern fault simulation as the primary source of test patterns are given to identify the optimum relationship between fault simulation and test generation. In developing this ATPG system, the objective was to successfully create a test for all non-redundant faults and identify all redundant faults with minimum CPU time, memory requirements, and number of test patterns.

The ATPG process begins with gathering information that will be used to make the test generation as intelligent as possible. This eliminates much of the unproductive test generation effort, and results in fewer aborted faults and improved performance. The creation of the test patterns is divided into two parts. First, based on the results of the experiment, an appropriate amount of random pattern fault simulation is performed identifying the detected faults and selecting only those patterns required to detect faults. For the faults which remain, test generation followed by fault simulation is repeated until all faults are either detected, proven redundant, or aborted. The ATPG process is completed by resimulating the patterns in reverse order to reduce the number of patterns in the test.

This ATPG system has been implemented on a 32-bit Apollo workstation and has successfully tested or proved redundant all faults of the ISCAS85 and ISCAS89 benchmark designs [3,4] with significantly better performance than previously published results. In obtaining these results, the flipflops contained in the ISCAS89 designs were assumed to be fully scannable. For the ISCAS89 designs, this marks the first time such results have been achieved where no faults were aborted.

Fault simulation

Parallel Pattern Single Fault Propagation

An efficient fault simulator is a critical part of an effective ATPG system. Its primary role is to accurately determine the faults which are detected by a given set of patterns. This allows the undetected faults to be selected for test generation to ultimately achieve the desired test coverage. Non-exact fault grading (such as probabalistic) is not adequate because it inevitably results in a significant number of undetected faults and wasted patterns. For large designs which require high test coverage, the cost of this inaccuracy cannot be tolerated. Furthermore, the performance of non-exact fault grading for scan designs is no better than fast fault simulation and frequently much worse.

The Parallel Pattern Single Fault Propagate (PPSFP) fault simulator [5] was chosen for this ATPG system. This simulator is optimized for the combinational nature of scan designs taking advantage of pattern independence and the ability to do zero-delay simulation. PPSFP combines parallel pattern evaluation with single fault propagation to perform highly efficient fault simulation. The fault simulator exploits the natural parallelism available when performing computer operations to simultaneously simulate multiple patterns. The number of parallel patterns depends on the length of a computer word. For 32 bit machines, the number of patterns simulated in parallel would be 32. In addition to its high performance, its memory requirements are low compared to concurrent and other traditional fault simulation techniques. This makes it an ideal choice for very large structured designs.

PPSFP 3-Value Simulation

The original PPSFP implementation provided only for a 2-value simulation. This optimized performance and memory but limited the designs which could be accurately simulated to those whose gates were always at a logical '0' or '1'. Gates for some combinational designs may sometimes be at an unknown state (X), especially those which contain tri-state devices. To make this ATPG system more general, the PPSFP fault simulation has been enhanced to perform a full 3-value (0, 1, X) simulation in a manner similar to that developed by Schulz and Pellkofer [6].

For 2-value PPSFP simulation, the coding of values is direct. A '0' logic value is represented by a value of 0 on a single bit of a 32 bit computer word. Similarly, a '1' logic value is represented by a bit whose value is 1. The logic operations are also direct, using the AND, OR, EXCLUSIVE-OR, and COMPLEMENT bitwise operations to perform the corresponding gate calculations for the 32 patterns in parallel. For 3-value simulation, each logic value requires 2 bits, one bit of which will be placed in a given position in each of 2 computer words of 32 bits each. The same number of patterns will be simulated in parallel, but there will be an extra word to store simulated values, and extra effort will be required to calculate gate values.

The following encoding [7,8] of the 3 states allows for maximum simulation performance for standard AND, NAND, OR, NOR, and INVERTER gates:

1. For a '0' logic value, the bit value in word1=0 and the bit value in word2=1.

2. For a '1' logic value, the bit value in word1=1 and the bit value in word2=0.

3. For an 'X' logic value, the bit value in word1=0 and the bit value in word2=0.

With this representation of the logic values, it is quite easy to perform the gate calculations. The values of an inverter can be calculated by simply swapping word1 and word2 of its input gate. For an AND gate, word1 is calculated by ANDing word1 of all inputs together, and word2 is calculated by ORing word2 of all inputs. For OR gates, word1 is calculated by ORing word1 of all inputs, and word2 is calculated by ANDing word2 of all inputs. NAND (NOR) gates can be calculated by first performing the AND (OR) calculation and then swapping word1 and word2.

PPSFP Algorithm

The PPSFP algorithm is shown in Figure 1 and consists of the following steps:

1. A zero-delay, rank-order, three-value good machine simulation is performed in parallel for 32 patterns (assuming a 32 bit machine such as an Apollo 3550). The good machine values of each logic gate for the 32 patterns are stored so that they can be compared with fault values.

2. For each group of single stuck faults that reside in the same fanout-free network (FFN) [9] still remaining in the fault list, a single fault propagate fault simulation is performed. For each fault in the FFN fault group, the fault values for all 32 patterns are first propagated forward to the FFN output and the results are recorded. The fault values of the FFN output are then assigned its good machine values with bit positions inverted that correspond to patterns that can detect any fault. Finally, the fault values for all 32 patterns are propagated forward beginning at the FFN output and continuing through gates that have fault values different from good-machine values. If the fault values are different than good-machine values at a primary output, a fault is detected if it was also detected at the FFN output for a given pattern. When a fault becomes detected, the simulation of that fault is stopped, the fault is recorded as detected, and it is removed from the fault list.

3. Steps 1 and 2 are repeated until all faults are detected or until all patterns have been simulated.

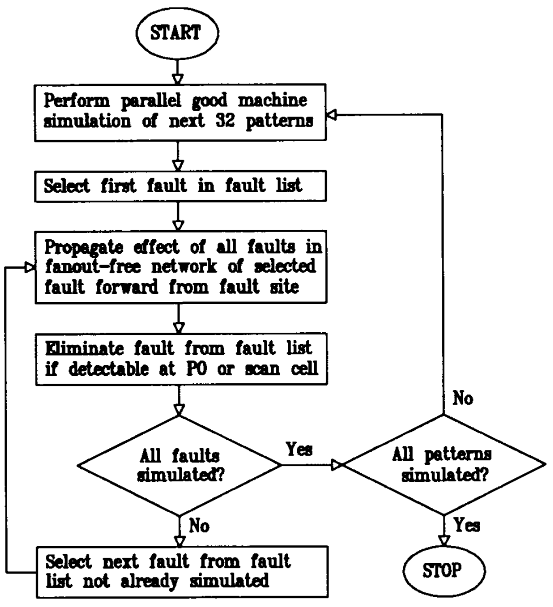
START
Perform parallel good machine simulation of next 32 patterns
Select first fault in fault list
Propagate effect of all faults in fanout-free network of selected fault forward from fault site
Eliminate fault from fault list if detectable at PO or scan cell
All faults simulated?
Yes
No
All patterns simulated?
No
Yes
Select next fault from fault list not already simulated
STOP

Figure 1 PPSFP algorithm

Deterministic Test Generation

General Considerations

Deterministic test generation is the process of explicitly creating a pattern that can detect a given fault. This is accomplished by determining a set of values to be assigned to primary inputs that places the opposing fault signal value at the point of the fault and propagates the fault signal to an observable point [10]. In satisfying the conditions necessary to create a test pattern, test generation is often faced with making decisions where there is more than one choice to place a desired value on a gate. If the choice of a decision ultimately results in a conflicting assignment on a gate, then a decision must be remade in an attempt to find a successful test. To guarantee a test will be found if one exists, the test generator must be capable of trying all possible choices of the decisions. If all possible choices are exhausted before a test is found, then no test for that fault exists and the fault is proven redundant. This identification of redundant faults is an important by-product of test generation and is critical to achieving a high test coverage. If a fault is identified as redundant it can be removed from the fault list. Otherwise, it must be considered a potentially detectable fault and counted as an untested fault when calculating test coverage.

A test generator is measured not only by its ability to create a test for a fault (or prove it is redundant) but also by its performance. Most test generators can claim the ability to create a test for all testable faults but the ability is limited by the amount of computer time that is practical to create the test. Exhausting all possible choices for decisions can require an enormous amount of time since the number of choices increases exponentially with the number of decisions. Left uncontrolled, there is no limit to the amount of time that test generation may consume - it could require years. Therefore, a test generator must place a limit on the amount of effort allowed for a given fault. Faults which remain undetected after this limit is exceeded are termed "aborted" faults.

Recent advances in test generation [11,12,13,14,15] strive to make the decision process more intelligent to minimize the number of faults that are aborted. Information is gathered to identify restrictions on decisions that help prevent making unsuccessful choices or restrict the decision to a single choice so that the associated gate can immediately be assigned the required value. This is normally accomplished by determining all possible implications that result when gates are assigned values. Some implications are direct and can be easily determined by propagating the effect forward and backwards through the direct logical connections in the circuit. Other implications are indirect and are more difficult to identify. They require an *apriori* analysis, the results of which are then used during actual test generation.

Preprocessing Steps

Prior to test generation, the following analyses are performed to make the test generation effort more efficient and minimize the number of faults that must be aborted.

Ordering of Gate Fanin and Fanout Whenever a choice must be made to satisfy a decision, the first choice should be the one that is easiest to satisfy. There are two distinct kinds of decisions which shall be called controllability decisions and observability decisions. A controllability decision on a gate implies there is more than one assignment of its input values that can satisfy its assigned value. An observability decision on a gate implies that the gate has more than one fanout stem which may be used to propagate a fault signal to an observable point.

The selected criteria for easiest to satisfy will be the minimum number of primary inputs (PIs) required to satisfy the condition. To calculate this number, the minimum number of PIs required to set each gate to both a '0' and a '1' is first calculated. This calculation is only approximate since the effect of reconvergent fanout is ignored. Using these values, the minimum number of PIs necessary to observe each gate can then be calculated, again ignoring reconvergent fanout effects.

For controllability decisions, only gates which are at values that allow a choice are of interest. A decision on AND and NAND gates implies that a '0' on any input is desired and the input which requires the fewest number of PIs to achieve a '0' state will be the first choice. OR and NOR gate are similar with the '1' state being the desired state. To minimize the test generation effort, the fanin list of each AND, NAND, OR, and NOR will be ordered so that the inputs easiest to control at the appropriate state are placed first. The test generator will then make the choices for controllability decisions in this fanin order.

For observability decisions, a calculation is made for each multiple-fanout gate to determine which fanout provides an observation path requiring the fewest number of PIs to be set. These values can be calculated directly from the previously calculated minimum observability and controllability values. The fanout list will then be ordered so that the fanout which provides the easiest observability appears first in the fanout list. The test generator will make choices for observability decisions in this fanout order.

Conflict-Free Gate Assignments Some gates may be assigned values such that there exists a way to satisfy the assignment without any possibility of creating conflicts [12]. Whenever these gates are assigned these values, the test generator can safely stop the backward implication process at these points. At the end of a successful test generation, these gate assignments which were used may then be satisfied by a simple backward-only implication.

The primary motivation to delay satisfying these gate assignments is to prevent unnecessary decisions. Remaking decisions that result from a conflict-free gate assignment cannot help to resolve a conflict since it cannot create a conflict. However, these decisions will contribute to the exponential process of exhausting all decision choices. Removing just two decisions each having four choices will reduce the maximum number of potential decision choices by a factor of 16.

The minimum requirement for a conflict-free gate assignment is the existence of a way to satisfy the assignment by a backward implication that does not place a value on any gate which is the source of reconvergent fanout. An analysis is performed to identify all gate assignments which satisfy this condition. This information is stored and made available for test generation.

Static Learning The implication process that determines the effect of a gate assignment can be enhanced beyond what can be calculated by backward and forward implication through the direct logical connections of the circuit. To expose these indirect implications of a gate assignment, it is necessary to perform a separate implication process for other gate assignments. Examples of these hidden implications and details on how to identify them are described in a paper by Shulz et al. [13]. The process of calculating these indirect implications is called static learning. Using the results of static learning significantly improves the ability of the test generator to avoid unproductive choices for decisions.

Immediate Dominators Another example of an indirect implication occurs when all paths from an observation decision gate reconverge to a common gate [13,14]. If the common gate is an AND or OR type gate and has additional inputs that are not in the fault path, then the additional inputs may be immediately assigned the non-controlling value of the gate. To efficiently identify this condition, the concept of immediate dominators is used. Gate A is defined to be an immediate dominator of gate B if all paths from gate B go through gate A, and gate A is the first gate along the path that satisfies this condition. During test generation, the immediate dominator (if one exists) of the observation decision gate and all immediate dominators of the immediate dominators are inspected. Inputs to these gates which are not in a possible fault propagation path will be assigned the non-controlling value of the gate.

Test Generation Strategy

After the completion of the preprocessing steps, the test generator has all the data it needs to efficiently create a test for a given fault. To begin the test generation process, the fault signal value is assigned to the point of the fault. Whenever a value is assigned to a gate, a complete implication process is performed. This implication process calculates all possible backward and forward implicated assignments, including those determined by static learning. If a non-conflictable assignment occurs, it will not be implicated backwards and it will be recorded so it may be resolved at the end of a successful test. Controllability decision points that result from the assignment are identified and stored. If an assignment restricts any decision to a single choice, then the assignments that are required to satisfy the decision are also made. If any gate assignment results in a conflicting condition then control is passed to the procedure that resolves conflicts.

After assigning the fault signal to the point of the fault, the fault signal is then driven forward to an observation decision point making all necessary assignments. Using the immediate dominators, all possible additional assignments are made. A choice is then made for the first controllability decision and the appropriate gate assignment is made. If this assignment results in additional decisions, they will all be satisfied before the previously remaining unsatisfied decisions. Choices continue to be made for decisions and the implication process performed on the resulting assignments until all controllability decisions are satisfied. If the fault signal has propagated to an observable point the test is complete. If not, a choice is made for the last observation decision and the fault signal is driven forward to the next observation decision point making all necessary gate assignments. The controllability decisions that result from the implication of these gate assignments are then satisfied as before. This will continue until the fault signal has reached an observable point or until a conflicting condition occurs.

If a conflict occurs and the number of conflicts exceed the maximum number allowed, then the fault is aborted. Otherwise, the next available choice for a decision is made in reverse order in which the decisions were satisfied. The states for all gates are restored to their value at the time of the decision. The previous choice of this decision which can not be satisfied is then assigned the opposite value. Although not apparent by the data which existed at the time of the decision, conditions exist which force that gate to be at the value opposite to what was desired. Assigning the value and performing the implication process will help prevent future unproductive decisions. The test generation process then continues as before. If there are no further available choices for decisions, then no test exists for the fault and the fault is declared redundant.

Decision Strategy Switching

The test generation strategy described in the previous section orders the decision making process so that all controllability decisions are satisfied before making observability decisions. In general, this strategy is the most likely to be successful but occasionally some faults must be aborted. Many of these aborted faults can be tested or proven redundant by simply switching the decision strategy so that all observation decisions are made before any controllability decisions [15]. Whenever a fault is aborted by one decision strategy, test generation will be repeated using the other decision strategy. If the fault is not aborted with the second strategy, then that decision strategy will become the initial strategy for future test generations. If a fault is aborted with both decision strategies, then it is considered an aborted fault.

Random Pattern Fault Simulation

The fault simulation of a large number of random patterns is an alternative to deterministic test generation in creating test patterns [16]. Only, those random patterns identified by the fault simulator as necessary to detect faults are stored in the test pattern set. This fault simulation cannot replace test generation since it can never identify redundant faults and can only create tests for faults which can be detected by a reasonable number of random patterns. However, it is possible that total ATPG performance can be improved by creating tests for some faults by random pattern fault simulation.

An experiment was conducted to identify what amount of random pattern fault simulation resulted in optimum ATPG performance. In this experiment a variable amount of random pattern fault simulation is performed followed by explicit test generation with fault simulation on the remaining faults. The details of the ATPG process are given in the next section. The results are a function of the relative performance of the fault simulator and test generator. For the fault simulator and test generator used in this ATPG system, the results are given in Table 1. Three designs were chosen from the ISCAS benchmark designs representing a small, medium, and large circuit. The performance of the ATPG system was measured over a range between no random pattern simulation up to continuing the random pattern simulation until 16 consecutive 32-pattern intervals failed to detect a fault. The columns in Table 1 are:

1. Design Name - The assigned name for the design.

2. Min. #dets. per interval - The minimum number of detected faults per 32-pattern interval required to continue random pattern fault simulation.

3. Max. #ints without dets. - The maximum number of consecutive 32-pattern intervals that fail to detect a new fault allowed before simulation is terminated.

4. #patterns (orig.) - The total number of patterns created by the ATPG process including those created by both random pattern fault simulation and test generation. Test coverage in all cases was 100% of testable faults.

5. #patterns (red.) - The number of patterns created by the ATPG process after pattern reduction.

6. DN3550 CPU secs - The CPU times for an Apollo DN3550 which was required for fault simulation, test generation, and total test creation. The CPU times do not include the pattern reduction fault simulation time.

Table 1 Random Pattern Fault Simulation Data

Design name	Min. #dets. per interval	Max. #ints without dets.	#patterns		DN3550 CPU secs		
			orig.	red.	flt_sim	test_gen	total
C3540	–	0	217	173	3.1	26.7	29.8
	32	1	234	162	4.1	13.6	17.7
	16	1	234	162	4.2	13.4	17.6
	8	1	243	166	4.3	13.3	17.6
	4	1	249	169	4.7	9.2	13.9
	2	1	249	169	4.8	9.2	14.0
	0	1	249	169	5.0	9.0	14.0
	0	2	256	172	5.8	6.6	12.4
	0	4	257	173	6.3	5.8	12.2
	0	8	257	169	7.1	4.8	12.0
	0	16	257	168	8.1	4.5	12.6
S15850	–	0	555	460	14.6	40.8	55.4
	32	1	643	472	18.2	36.2	54.4
	16	1	657	465	19.6	36.0	55.5
	8	1	682	463	24.8	30.1	54.9
	4	1	682	463	24.6	30.3	54.9
	2	1	693	475	26.3	29.3	55.6
	0	1	697	472	26.5	30.2	56.7
	0	2	717	483	35.0	25.1	60.1
	0	4	719	467	45.6	22.5	68.1
	0	8	726	475	60.9	20.4	81.3
	0	16	736	473	80.8	19.5	100.4
S38417	–	0	1232	967	44.4	57.6	102.0
	32	1	1442	987	58.2	49.9	108.1
	16	1	1451	980	59.3	49.6	108.9
	8	1	1458	988	60.6	49.1	109.7
	4	1	1545	985	76.6	46.9	123.5
	2	1	1545	985	76.6	47.1	123.7
	0	1	1525	955	89.3	43.4	132.7
	0	2	1569	997	119.3	40.9	160.1
	0	4	1568	988	120.4	40.8	161.2
	0	8	1555	983	232.1	32.2	264.3
	0	16	1579	970	331.6	28.5	360.2

The results in Table 1 are arranged in order of increasing amount of random pattern fault simulation. As expected, an increase in the amount of random pattern fault simulation caused the fault simulation CPU time to increase and the test generation time to decrease. However, it is the sum of the two times which is of interest and its relationship with the amount of random patterns is more complex. For the small design (C3540), the optimal APTG time occurred with a large number of random patterns, while the optimal ATPG time for the large design (S38417) occurred with no random patterns. This effect can be explained by noting how the performance of fault simulation and test generation is affected by increasing gate count. PPSFP fault simulation has been shown to increase almost linearly with the number of gates in a circuit [5]. Test generation is a local effect and the time required for a single test generation is relatively constant. The test generation cost of a set of 32 patterns stays constant while the cost of fault simulating a set of 32 random patterns keeps increasing with the number of gates.

The number of patterns created by the ATPG system was also significantly affected by the amount of random pattern fault simulation. Increasing the number of random patterns increased the number of patterns in the test set by as much as 30 percent. However, this difference totally disappeared after pattern reduction was performed.

ATPG Algorithm

An algorithm has been developed to perform the ATPG process that has been optimized for maximum test coverage and performance, especially for large designs. All reasonable effort is made so that the test generation decision-making process is as intelligent as possible to avoid aborted faults. This is a small added expense to test generation but for faults which experience conflicts, the reduction in remade decisions results in significantly better overall performance. The algorithm of the ATPG process is shown in Figure 2 and consists of the following steps:

1. A fault list is created choosing a representative fault from each fault equivalence class.

2. The learning process is performed. This includes static learning and the identification of non-conflictable assignments and immediate dominators.

3. Random pattern fault simulation is performed for sets of 32 patterns. Faults which become detected are recorded and removed from the fault list. The patterns which were required to detect these faults are added to the test pattern set. The random pattern simulation continues until the termination criteria is reached. Designs less than 5000 gates are stopped when 4 consecutive sets of 32 patterns fail to detect a new fault. For designs of 5000 gates or greater, the simulation stops when a set of 32 patterns detects less than 16 faults. Although the experiment suggests that no random patterns results in maximum ATPG performance for large designs, the existence of aborted faults could significantly effect the optimum point. The additional cost of this amount of simulation was small, and for designs which contain aborted faults, this could provide a significant benefit if they could be detected with random patterns.

4. Deterministic test generation is performed to create patterns for 32 randomly selected faults that remain in the fault list. Primary inputs which were not assigned values by test generation are assigned random values. Faults which are identified as redundant are recorded and removed from the fault list. Aborted faults are recorded and prevented from further test generation, but remain in the fault list and continue to be fault simulated. Aborted faults which are not redundant may be accidentally detected by later patterns.

5. Fault simulation is performed for the 32 patterns created by test generation. Detected faults are recorded and removed from the fault list. Patterns required to detect these faults are added to the test pattern set.

6. Steps 4 and 5 are repeated until no faults remain except those that were aborted by test generation.

7. If the faults that were detected by early patterns are also detected at later patterns, then the early patterns are not required. A resimulation of the patterns in reverse order can identify when this occurs and significantly reduce the number of patterns in the test pattern set. Fault simulation of the non-redundant faults is performed for all patterns in the test pattern set in reverse order for sets of 32 patterns. Detected faults are recorded and removed from the fault list, and the patterns necessary to detect these faults are added to the reduced pattern set.

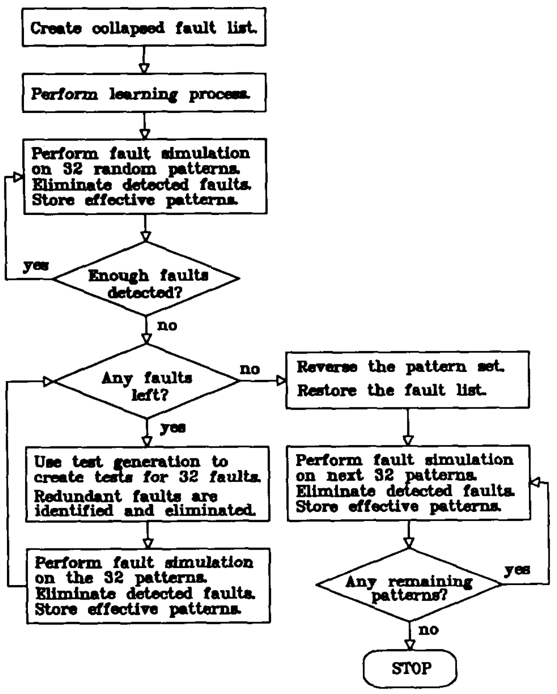
Create collapsed fault list.
Perform learning process.
Perform fault simulation on 32 random patterns. Eliminate detected faults. Store effective patterns.
yes
Enough faults detected?
no
Any faults left?
no
yes
Reverse the pattern set. Restore the fault list.
Use test generation to create tests for 32 faults. Redundant faults are identified and eliminated.
Perform fault simulation on the 32 patterns. Eliminate detected faults. Store effective patterns.
Perform fault simulation on next 32 patterns. Eliminate detected faults. Store effective patterns.
Any remaining patterns?
yes
no
STOP

Figure 2 ATPG algorithm

Benchmark Results

This ATPG system has been implemented on an Apollo 32-bit workstation. The results of performing ATPG for all ISCAS85 and ISCAS89 benchmark designs [3,4] have been obtained and are given in Table 2 and Table 3. The ISCAS89 designs contain flipflops which are assumed to be fully scannable. Each flipflop was remodeled into a primary input and primary output to reflect its ability to be both a controllable and an observable point. The following columns are contained in the tables:

1. Design Name - The assigned name for the design.

2. #gates - The number of logic gates in the simulation model. Primary inputs and primary outputs are considered gates and each flipflop was expanded to a separate primary input and primary output.

3. #DFFs - The number of D flipflops which were contained in the design.

4. #faults (total) - The total number of fault equivalence classes generated from the circuit model.

5. #faults (redund) - The number of redundant faults identified by the test generation process.

6. #faults (abort) - The number of faults aborted by the test generator.

7. #patterns (orig.) - The number of patterns in the original test pattern set.

8. #patterns (reduce) - The number of patterns in the test after reduction.

9. DN3550 CPU secs - The CPU time in seconds for an Apollo DN3550 required to perform the entire ATPG process as shown in Figure 2.

For all of the ISCAS85 and ISCAS89 designs, the ATPG process successfully tested all testable faults and proved all redundant faults to be redundant. There were no faults that were aborted. The performance results are equally impressive. The largest designs which contained about 25 thousand logic gates required only about 3 minutes of CPU time on an Apollo DN3550 workstation. A graph (Figure 3) of circuit gate count versus ATPG CPU time shows that the CPU time increases linearly with circuit size for the ISCAS designs.

Table 2 ISCAS85 ATPG Results

Design name	#gates	#DFFs	#faults			#patterns		DN3550 CPU secs
			total	redund	abort	orig.	reduce	
C432	211	0	524	4	0	78	60	2.3
C499	283	0	758	8	0	69	55	1.2
C880	469	0	942	0	0	111	63	1.7
C1355	627	0	1574	8	0	116	91	4.7
C1908	990	0	1879	9	0	176	122	8.1
C2670	1575	0	2747	117	0	181	122	17.3
C3540	1775	0	3428	137	0	257	173	18.9
C5315	2631	0	5350	59	0	228	150	10.8
C6288	2480	0	7744	34	0	44	32	23.8
C7552	3883	0	7550	131	0	380	235	41.5

Table 3 ISCAS89 ATPG Results

Design name	#gates	#DFFs	#faults total	redund	abort	#patterns orig.	reduce	DN3550 CPU secs
S27	24	3	32	0	0	9	8	0.1
S208	133	8	215	0	0	44	31	0.5
S298	170	14	308	0	0	49	32	0.5
S344	225	15	342	0	0	37	23	0.5
S349	226	15	350	2	0	37	23	0.5
S382	232	21	399	0	0	54	37	0.5
S386	199	6	384	0	0	95	76	1.2
S400	239	21	428	6	0	52	38	0.6
S420	265	16	430	0	0	91	61	1.7
S444	255	21	474	14	0	53	38	0.8
S510	255	6	564	0	0	80	65	1.4
S526	265	21	555	1	0	102	69	1.2
S526n	266	21	553	0	0	102	69	1.1
S641	495	19	467	0	0	93	58	1.3
S713	508	19	581	38	0	94	58	1.5
S820	368	5	850	0	0	208	126	4.1
S832	367	5	870	14	0	203	125	4.3
S838	523	32	857	0	0	144	105	4.2
S953	521	29	1079	0	0	141	100	4.6
S1196	615	18	1242	0	0	238	149	6.1
S1238	598	18	1355	69	0	244	160	9.0
S1423	906	74	1515	14	0	125	67	3.2
S1488	741	6	1486	0	0	209	135	6.8
S1494	735	6	1506	12	0	208	135	7.0
S5378	3400	179	4603	40	0	422	274	17.6
S9234	6326	228	6927	452	0	608	426	66.5
S13207	10167	669	9815	151	0	672	490	76.6
S15850	11739	597	11725	389	0	657	465	84.4
S35932	21903	1728	39094	3984	0	81	74	71.7
S38417	27379	1636	31180	165	0	1451	980	186.8
S38584	24173	1452	36303	1506	0	672	490	172.6

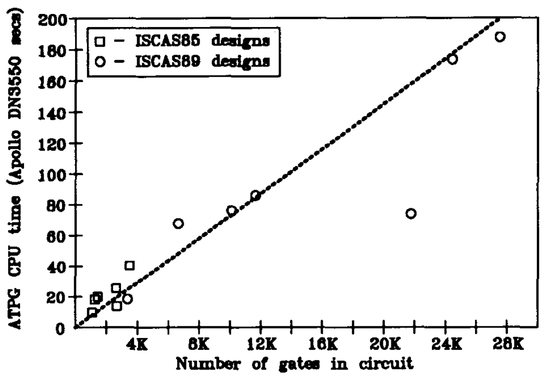

Figure 3 Graph of circuit size versus ATPG CPU time.

Conclusions

An automatic test pattern generation system has been developed using many of the features contained in recent literature and optimized for performance of large structures. For the ISCAS85 and ISCAS89 circuits, this ATPG system created a test for all testable faults and identified all redundant faults without a single aborted fault. This represents the first time this has been achieved for the ISCAS89 designs and the performance of this ATPG system is significantly better than published results [4,13]. Performing ATPG for the largest ISCAS89 designs which contained about 25 thousand gates required only 3 minutes of CPU time on an Apollo DN3550 workstation.

The data collected for the ISCAS designs showed that the ATPG CPU time increased linearly with gate count. This strongly suggests that ATPG can now be efficiently performed for circuits of 100 thousand gates and even one million gates. Some aborted faults are inevitable, but the test coverage of testable faults would be expected to exceed 99.9% and almost all redundant faults would be identified. For designers who are considering the costs versus benefits of a scan design, the ability to create a high quality test with so little effort should be a strong motivation for scan.

Acknowledgements

The authors wish to thank Dr. Eun Sei Park for the many helpful suggestions in the development of this ATPG system. We also would like to thank Clayton Mohr, Hemanth Kumar, and Mike Ausec for assisting in the preparation of the manuscript. Finally, we wish to give special thanks to Dr. Franc Brglez for his role in creating and distributing the ISCAS benchmark designs. These designs have significantly contributed to many of the recent advances in both test generation and fault simulation.

References

[1] E. J. McCluskey, Logic Design Principles, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[2] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability", Proc. Design Automation Conf., 1977, pp. 462-468.

[3] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Designs and a Special Translator in Fortran", International Symposium on Circuits and Systems, June 1985.

[4] F. Brglez et al., "Combination Profiles of Sequential Benchmark Circuits", International Symposium on Circuits and Systems, May 1989, pp. 1929-1934.

[5] J. A. Waicukauski et al., "Fault Simulation for Structured VLSI", VLSI Systems Design, Dec. 1985, pp. 20-32.

[6] M. H. Schulz and D. Pellkofer, "Fast Three-Valued Fault Simulation in Combinational Circuits", to be published.

[7] D. K. Pradhan, Fault-Tolerant Computing, Theory and Techniques, Vol. 1, Prentiss-Hall, Englewood Cliffs, New Jersey, 1986.

[8] N. Ishiura et al., "High-Speed Logic Simulation on Vector Processors", IEEE Transactions on Computer -Aided Design, May 1987, pp. 305-321.

[9] K. J. Antreich and M. H. Schulz, "Fast Fault Simulation for Scan-Based VLSI-Logic", IEEE Transactions on Computer-Aided Design, Sept. 1987, pp. 704-712.

[10]J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method", IBM Journal of Research and Development, vol. 10, July 1966, pp. 278-291.

[11]P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Transactions on Computers, vol. C-30, March 1981, pp. 215-222.

[12]H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms", IEEE Transactions on Computers, vol. C-32, No. 12, Dec. 1983, pp. 1137-1144.

[13]M. H. Schulz et al., "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE Transactions on Computer-Aided Design, vol. CAD-7, No. 1, Jan. 1988, pp. 126-137.

[14]T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG", Design Automation Conference, June 1987, pp. 502-508.

[15]H. B. Min and W. A. Rogers, "Search Strategy Switching: An Alternative to Increased Backtracking", International Test Conference, Aug. 1989, pp. 803-811.

[16]J. L. Carter et al., "ATPG via Random Pattern Simulation", International Symposium on Circuits and Systems, June 1985, pp. 683-686.