Built-in self-test support in the IBM Engineering Design System

by B. L. Keller
T. J. Snethen

To evaluate the effectiveness of built-in self-test (BIST) for logic circuits, the test design automation (TDA) group within the IBM Engineering Design System (EDS) has developed tools to support BIST. This paper is an overview of that support. The specific hardware approaches taken are described briefly, and a short description is given of the major tools that have been developed and the methodology for using them. The performance of the system is shown for two sample circuits.

Introduction

Many different implementations of built-in self-test (BIST) have been used throughout the industry and within IBM as well [1–3]. Previous BIST designs within IBM have been accomplished without much design automation support. However, as IBM and the industry in general have become more interested in BIST, the need for design automation support of BIST has become more important.

BIST design automation support has been developed by the test design automation (TDA) mission of the IBM Engineering Design System (EDS). The tools provided by this support and described in this paper assume that linear feedback shift registers (LFSRs) are used to generate random patterns and to compress output responses into signatures.

The STUMPS self-test architecture

BIST tools were developed to support the STUMPS architecture[1] as shown in **Figure 1**; however, they were also defined to include support for the more general random-pattern testing environment [4] (**Figure 2**). This paper deals with the BIST support based upon the STUMPS circuit architecture.

The architecture for STUMPS is built upon the IBM level-sensitive scan design (LSSD) approach for designing testable circuits. In fact, the support requires that a circuit have (at least) two states—the LSSD state and the self-test state.

LSSD circuit design concepts are well known [5]. The self-test design structure discussed in this paper refers to the general BIST architecture supported by EDS. The LSSD design rules ensure that every latch in the circuit is scannable. The only memory elements not required to be scannable are those inside a random-access memory (RAM). The self-test design rules allow certain latches to have specific functions in the BIST operation; these latches are not scannable in the self-test state. Also, certain latches are allowed to be fixed in value (once they

[1] STUMPS is a multilevel acronym which stands for Self-Test Using MISRs and Parallel SRSGs; it was defined by Paul Bardell and William McAnney in 1982 [1]. MISR and SRSG stand respectively for Multiple-Input Signature Register and Shift Register Sequence Generator (see acronym definition list).

are initialized). These latches, called "fixed-value" latches, are loaded with the appropriate value during an initializing LSSD load before entering the self-test state of the circuit. Channel gating latches (required for generation of channel signatures) are an example of why one would use such latches.

The basic STUMPS circuit comprises three types of shift registers:

1. STUMPS channels—ordinary LSSD shift registers with channel scan inputs and outputs. Each channel can span a portion of a chip, an entire chip, or multiple chips.
2. Pseudorandom pattern generator (PRPG)—a linear feedback shift register (LFSR) used to generate the random patterns to be shifted into the channels. The original STUMPS design called this a parallel shift register sequence generator (PSRSG).
3. Multiple-input signature register (MISR)—an LFSR used to compress all channel outputs into a single signature.

The EDS system requires that there be an LSSD state in addition to the self-test (STUMPS) state, because the initial values for all latches are loaded using the LSSD scan operation (including any fixed-value latches). Once the initial values are loaded, the self-test state is entered, and the self-test operations begin. Because we define the self-test state also to be a boundary-scan internal state, no external inputs to the circuit (except for the clocks and other primary inputs which establish and run the self-test) can influence the final signature in the MISR. This is the essence of the STUMPS approach to BIST.

The STUMPS architecture as shown in Figure 1 provides for an optional exclusive-OR tree network between the PRPG and the channel inputs. The exclusive-OR trees provide each channel input with a different exclusive-OR function of some number of the PRPG outputs. This can be used to prevent channels from having highly correlated values and is desirable when more than one channel can feed the same logic.

The PRPGs and MISRs are implemented using LFSRs. The EDS tools support two different kinds of LFSRs: multiplier and divisor, as shown in **Figure 3** and **Figure 4**. The size and number of feedback taps used can be arbitrary; however, a maximal-length LFSR should be used. A maximal-length LFSR of n bits will cycle through $2^n - 1$ different states before repeating. Failure to use a maximal-length LFSR for the PRPG may cause a drop in test coverage which would be shown by simulation. Failure to use a maximal-length LFSR for the MISR may cause faults to be masked out as the signatures are being generated; however, simulation will not detect such occurrences, since it assumes that a fault is detected as soon as its effect has been shifted into the MISR.

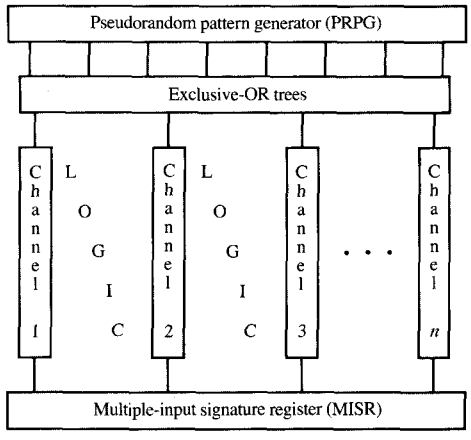

Figure 1

Simplified view of the STUMPS channel architecture.

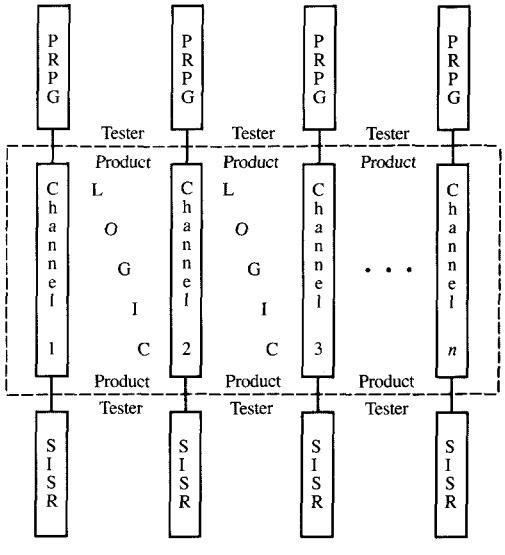

Figure 2

Simplified view of tester-assisted pseudorandom-pattern testing.

Throughout the tool design, an effort was made to be as general as possible. Whereas the tools require a

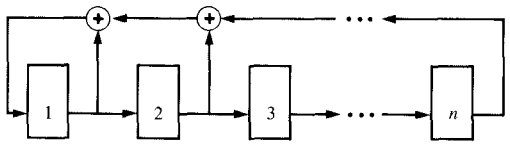

Figure 3

Multiplier type of linear feedback shift register. Several bits of the LFSR are exclusive-ORed and fed back into the leftmost bit as they shift.

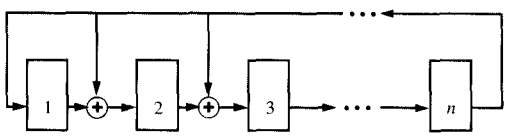

Figure 4

Divisor type of linear feedback shift register. The rightmost bit of the LFSR is fed back into the leftmost bit and exclusive-ORed into several other bits as they shift.

STUMPS circuit in the self-test state, there are no packaging restrictions as to how this is implemented. The STUMPS architecture may be implemented on a chip, module, card, or board. The tools specifically support processing of a circuit containing several independent STUMPS circuits (e.g., a module of ten chips each with STUMPS built into them).

In addition, when the circuit being processed is in the boundary-scan internal state, the resulting signatures and test coverage are valid regardless of where the tests are being applied. BIST may be applied in manufacturing (during module, card, board, or system test), after manufacturing (during power-on self-test in the customer's office), or both.

Requirements

As mentioned earlier, IBM has used many different self-test designs. Before the start of design of the EDS tools, requirements were gathered from design laboratories and manufacturing sites throughout IBM. The following list contains requirements deemed necessary for design automation support tools.

- *Design rules checking* This is required to ensure that the logic design follows all of the applicable rules. If the rules are not followed, many things can go wrong—including having signature mismatches even though the part is good.
- *Testability analysis* IBM requires high test coverage of stuck faults. It is therefore required that design automation tools indicate what test coverage will be attainable for a given design and a given number of applied patterns.
- *Signature generation* Since it is very difficult to obtain the response signatures for a correct part, it is desirable that design automation tools provide them. If a part is already known to be good, it can be used to determine the expected signatures; however, finding the first good part is not easy and is somewhat exposed to error (i.e., a bad part may be assumed good if it happens to be one of several parts which all yield the same signature).

In addition, the following design automation support items were deemed to provide a significant benefit.

- *Support for individual channel signatures* This is desirable when there is a need to diagnose a failure as belonging to individual channels or individual latches within a channel. This support requires that it be possible to gate out each channel independently of the other channels feeding the MISR. This is usually implemented by using a fixed-value latch for each channel to gate its connection to the MISR input. These latches are called "channel gating latches."
- *Interactive analysis of untested faults* Although a circuit may be intrinsically testable with deterministic test patterns, it may be very difficult to test with random patterns. An interactive analysis tool can help determine the reasons for low testability. Once the reasons are known, circuit modifications can be made to enhance the testability to any desired level.
- *Support for RAM initialization and RAM test* To generate repeatable signatures, a self-tested circuit must either have all its memory elements initialized, or mask the uninitialized memory elements to prevent their being observed during the logic test. For many designs, gating the RAMs in this way is not practical; therefore, support is useful in ensuring that the RAMs are initialized—preferably to random values. Allowing RAMs to be tested via self-test is a natural extension of the logic self-test support. It is not required that RAMs be tested in this manner—instead, they can be tested via a functional self-test or by some other deterministic method.

System overview

The major software functions supplied for self-test support are the following:

1. Checking the logic design for adherence to the self-test design rules and limitations.
2. Accepting user input of clocking sequences used for self-test.
3. Checking the validity of user-entered array[2] initialization and array test sequences.
4. Analyzing the logic for its random-pattern testability.
5. Calculating good-machine signatures.

These and other key steps shown in the system flow diagram in **Figure 5** are discussed in the following sections.

• *Structure processing*

The structure-processing step extracts the logic description from the EDS data base, expands the technology-specific books into the primitive blocks which are recognized by TDA programs, and assembles all chip or module data to create the current packaging-level view of the circuit. These programs extract other pertinent information such as PRPG and MISR identification. Information on how to enter the LSSD, self-test, or array-scan state is derived from the data base and stored in tables associated with the logic model.

• *Design rules checking*

The design rules checking function is required in order to ensure that specific testability design rules have been followed. Many rules must be checked to ensure that a circuit satisfies LSSD requirements: No races may exist; all latches must be included in a shift register; the shift registers must be configured correctly, etc. In addition to checking design rules, the checker provides information in its output files which is useful to subsequent programs.

A major addition to the existing design rules checking function was checking for adherence to the self-test rules and producing information about the SRL (STUMPS) channels. This information includes on-product LFSR definitions (i.e., polynomials used) and tester LFSR definitions [to support the more general random-pattern testing environment where the tester provides PRPGs for each primary input and a single-input signature register (SISR) for each primary output]. The information also defines each shift register channel and notes the length of the longest channel.

The self-test rules being checked include (but are not limited to) the following:

[2] In this paper, the terms "array" and RAM are used synonymously.

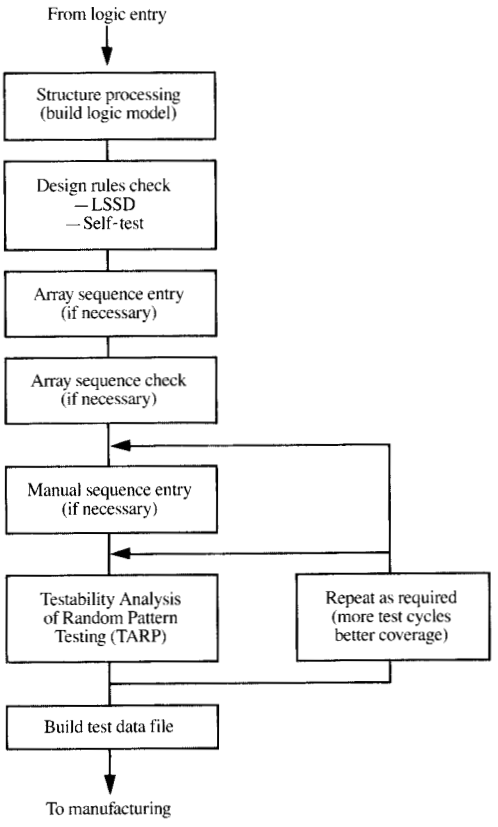

Figure 5

Simplified view of flow through pseudorandom-pattern test system.

- Ensure that no unknown values can propagate into a signature register (no X-state propagation). If the self-test state is also a boundary-scan internal state, all primary inputs which are not test-function primary inputs are considered to be sources of unknown values, and therefore are checked to ensure that their values cannot be observed.
- Ensure that each channel is fed by a PRPG and feeds to a signature register.
- Ensure that any shift-register latches (SRLs) defined to be "fixed-value" SRLs do in fact maintain their initial value once the self-test circuit state is entered.
- Optionally check for correct channel gating logic (between the channel outputs and an on-product MISR) for those designs which require the ability to collect individual channel signatures for better diagnostics.
- Ensure predictable operation of all three-state drivers. Random patterns are not permitted to cause a three-

state driver to "burn out," or to produce an observable, unpredictable state. This check verifies, for all internal three-state drivers that are not inhibited during the self-test state, that they cannot be driven to both zero and one simultaneously. It also verifies that if a three-state driver can go to high impedance, it must be terminated to produce either a soft 0 or 1, or it cannot be observed when it is in a high-impedance state.

If the circuit has embedded random-access memory (RAM), the design rules checker determines whether or not the RAM can be observed during the self-test. The design rules checker indicates that each RAM that is not blocked from observability must be initialized (via a RAM initialization sequence). If at least one RAM must be initialized, the design rules checker looks for the existence of an array-scan state for the circuit, because any array-initialization sequence applied to the circuit for the purpose of initializing RAM must be applied in the array-scan state.

If the circuit possesses an array-scan state, the self-test rules checker performs most of the same checks applied to the self-test state in the array-scan state. The main difference between the two states is that the array-scan state may have certain SRLs removed from the channels for use as address steppers. The address steppers are required to guarantee that every address will be initialized (or tested).

The most important check performed by the self-test rules checker is to prohibit unknown values from propagating into the signature. Most designs work well given their functional input patterns, but some do not behave as well when given a nonfunctional pattern. Because random patterns generated during BIST are almost certain to include nonfunctional patterns, the design rules checker must point out any potential problems that could be caused by nonfunctional patterns.

- *Fault model build*

The test coverage for BIST is based upon the same stuck-fault model used for deterministic test generation, implying that ignorable[3] faults and equivalence classes have been identified. However, the fault model build function performs some fault analysis uniquely for circuits under self-test. This analysis is called "*a priori* fault mark-off." Because of the high confidence that faults in some classes of logic are certain to be detected, and because there is no diagnostic fault dictionary built for random-pattern self-test, these faults are flagged as "detected *a priori*." This has the advantage of saving time in the more expensive steps of testability analysis (such as fault simulation).

[3] An ignorable fault is one that would obviously not affect the function of the circuit. This includes faults blocked by TIEd logic or faults on dangling logic. It does not include identification of redundant faults.

The faults marked *a priori* belong to the following classes of logic:

1. Most of the faults within PRPGs and MISRs.
2. Faults along the BIST channel scan data paths.
3. Some faults on the BIST shift clock paths.
4. Faults in any exclusive-OR logic between the channels and LFSRs.
5. Faults between RAM inputs and the last fan-in blocks that feed the array.

The percentage of the total faults marked "*a priori* tested" differs from part to part and is dependent upon the percentage of the circuit in the shift registers. This percentage may range from as low as 5 percent to as high as 30 percent (for circuits made up mostly of latches).

- *Pattern sequence entry*

BIST support provides an automatic pattern-sequence-generation tool for those designs not constrained as to how the clocks to the circuit can be applied. However, some self-test designs can apply only a limited set of clocking sequences; for these situations, manually entered sequences must be generated. Array initialization and array-test sequences are also manually entered.

Pattern sequences are entered using a full-screen CRT interface. Nets to be included in operations are listed on the left side of each row, while the operations to be applied are listed across the top of the columns. As an example, the "pulse clocks" operation can be coded in a column, and the desired clocks can be identified by placing "P" (for pulse) in the row pertaining to each clock.

Besides the BIST circuit design rules, some clocking sequence rules have also been established; these are used to ensure that

1. No race conditions occur.
2. The TDA Compiled Logic Simulator can be used for high-speed good-machine and faulty-machine simulation.
3. Consecutive patterns can be simulated in parallel.
4. Post-test diagnostic simulation can be performed efficiently [6–8].

The pattern sequence entry program checks that the sequences do not violate constraints 1 or 2. If a sequence violates any of the other constraints, the sequence is so flagged; it may cause less efficient processing by TDA simulators or diagnostic simulators.

- *Array sequence checking*

As mentioned above, there is a tool for verifying whether an array initialization sequence does in fact initialize

every RAM cell that can be observed during the logic self-test. This program can also verify whether an array-test sequence will test every port and all the cells of every RAM in the circuit. These checks are performed by a type of simulation and can be quite expensive to run.

If all (or even just the "large") RAMs are gated from being observed, an array initialization sequence is not required (or can be much shorter in length).

• *Testability analysis*

BIST testability analysis is performed under the control of the Testability Analysis for Random Patterns (TARP) control program. The TARP control program was designed to allow many different testability analysis tools to "plug in" as they became available. It provides the basic functions that are required of any analysis tool; those functions not generic across tools must be implemented by the tool requiring the function.

Four basic functions run under TARP:

1. Automatic test-sequence generation.
2. Signature generation.
3. Test-coverage calculation.
4. Testability analysis.

The main tool currently in use is the Complied Logic Simulator (CLS), used in conjunction with the LFSR Simulation Monitor (LFSRSIM). This is a hybrid simulator implementing the low-level logic simulation via compiled object code (CLS) and the simulation of the channel-scanning and signature-generation operations via high-level (LFSR) simulation. This hybrid simulator, called CLS-LFSRSIM, is discussed in more detail in the next section.

The CLS-LFSRSIM simulator performs good-machine simulation for signature generation and fault simulation to calculate test coverage. Signature generation and fault simulation are normally done simultaneously, although the two functions can be selected independently by the user.

Another tool available under TARP is used when it would be prohibitively expensive to fault-simulate the total pattern set to be applied to the circuit. This tool does not perform pattern simulation—rather, it takes a more analytical approach to determining the random-pattern testability of the circuit. The tool is named CARPET (Circuit Analysis for Random-Pattern Extensive Testing).

CARPET first determines the detection probability for any given fault to be considered detected (with 99.999% confidence), assuming that all the patterns requested to be applied to the circuit are in fact applied. Then it estimates the detection probability for each undetected fault. If the detection probability is high enough, CARPET marks the fault as detected. Since CARPET assumes that the patterns are completely random, it may not detect any problems associated with correlated (not completely random) inputs.

Because CARPET may not detect these problems, because it is expensive to run, and also because it can be quite pessimistic in its assertions, CARPET is not utilized as often as CLS-LFSRSIM.

TARP is normally used to process several different logic test sequences. These sequences can be generated automatically by TARP, or can be entered manually.

CLS-LFSRSIM simulator

The LFSRSIM portion of the CLS-LFSRSIM simulator is responsible for interpretively executing the current clocking sequence. If the logic test were to be preceded by an array initialization, that too would be simulated.

The main responsibility of LFSRSIM is to perform the channel-scanning operation at a high level as efficiently as possible. It computes the new (PRPG-generated) values scanned into the channels. It also generates the MISR signature by scanning the current channel contents into the MISR. If channel signatures are requested, LFSRSIM computes a separate signature for each channel. In addition to the final full MISR signature and individual channel signatures, LFSRSIM can produce intermediate signatures for use in detecting failures before all tests have been applied. The number of tests between intermediate signatures is specified by the user.

When a particular clocking event occurs, LFSRSIM calls the CLS-compiled object code to simulate that event at the logic gate level. CLS has good-machine object code to calculate the good-machine response to a clocking event. It also has faulty-machine object code to calculate the response of a fault or group of faults to a clocking event. The compiled object code provides very high simulation rates (equivalent to more than 500 million gate evaluations per second for good-machine simulation and even higher for faulty-machine simulation). The compiled code simulation is similar to and was derived from that of the High Speed Simulator (HSS) [9].

The use of compiled code for low-level logic and an LFSR simulator for high-level channel-scan simulation provides an efficient means of simulating the (potentially) large number of random patterns usually applied to a BIST circuit. To make simulation even more efficient, CLS-LFSRSIM can simulate up to 32 patterns in parallel for about the cost of simulating one pattern.[4]

CLS-LFSRSIM assumes that a fault is detected once its fault effect is latched into a channel. It assumes that the channel-scan operation will capture the fault effect into

[4] The CLS-LFSRSIM parallel pattern simulation breaks down when RAMs are calculated. For this reason, if many RAMs must be calculated (many fault effects must propagate through RAMs), the cost of simulating patterns in parallel goes up—but never as high as the cost of simulating the patterns one at a time.

Table 1 Statistics for the Hardware Test Vehicle.

277,746	Logic gates	205	RAMs
3,180	SRLs	32	Channels
166	SRLs in longest channel	93,840	RAM cells
689,070	Stuck faults	443,664	Fault equivalence classes
8192	Array initialization cycles	29	Clocks per array initialization cycle
32	Total array initializations		
262,144	Total array initialization cycles	7,602,176	Total array initialization clock events (GM simulated)
28,672	Logic test cycles	29	Clocks per logic test cycle
		831,488	Total logic test clock events (FM simulated)
94.4	% Fault coverage		

CPU minutes	*Activity*	*CPU minutes*	*Activity*
17	Check design rules	26	Compiling circuit for simulation
29	Check array sequences	7	CLS-LFSRSIM GM simulation of 7,602,176 array initialization clock pulses
15	For miscellaneous	54	CLS GM and FM simulation of 831,488 test clock pulses
87	Total for all simulation and signature generation	87	Total for all simulation and signature generation
148	Total for all steps		

Note: CPU times given are for an IBM 3090-600 E processor.

the MISR and will result in a signature different from that of the good machine.

The output from CLS-LFSRSIM comprises the following:

1. The test sequence used. This includes the identification and order of clocks to be pulsed and any other necessary control signals, as well as the number of cycles for which the test is to be run.
2. Good-machine signature(s) for the sequence.
3. Cumulative and incremental fault coverage attained by this simulation.
4. Optionally, good-machine net activity counts. These are used by the Self-Test Fault Analyzer (discussed next) in determining the signal probabilities observed on the nets.
5. Complete good-machine circuit state (including RAM cells). These can be used as inputs to a subsequent simulation to be performed after this current sequence is applied. The ability to link each sequence with the results of the previous sequence removes the need to re-initialize the RAMs, which in turn speeds up the simulation and the test application.

Self-Test Fault Analyzer

Attaining adequate test coverage when random patterns are being applied is not always easy. Some designs are inherently random-pattern testable, while others are random-pattern resistant. For very large circuits, there may exist some portions of the circuit which are easily tested, whereas other portions are very hard to test with random patterns. To aid the user in determining where his logic is random-pattern resistant, EDS provides the Self-Test Fault Analyzer (STFA).

STFA uses the results of CLS-LFSRSIM fault simulation to determine where faults are being left behind in "clusters." A "fault cluster" is simply a group of faults in a connected area of logic that are not detected, apparently due to the inability of the random patterns to generate the appropriate sensitizing conditions. STFA attempts to find a single net which may be the largest factor contributing to the existence of the cluster. For example, STFA would point out that a certain high-fan-in AND gate (or equivalent) was inhibiting the detection of (n) faults.

STFA produces a list of clusters in order of decreasing number of affected faults. The user can evaluate this list and determine which areas of logic (if any) should be modified in order to improve its random-pattern testability. Often just fixing the top ten entries on this list solves most of the testability problems. Instruction decode logic and comparator outputs are often found at the top of cluster lists.

• *Saving test results*

Each TARP run is considered an experiment. Using his own criteria, the user can decide whether the experiment was a success or a failure. If an experiment is considered a success, the user can save the results permanently by running the Save Test Results step. All previously saved experiments are accumulated, and the test coverage reported is for the combination of all such experiments.

• *Building the test data file*

When adequate test coverage is attained, the test data file (TDF) is built; it includes

1. The logic model used to derive the test data.
2. The test sequences to be applied.
3. The signatures.
4. The test coverage attained.
5. Audit information indicating which steps of the flow failed or which rules failed.

Performance and results

The performance of the BIST support system depends on the particular circuit being processed. Circuits that have embedded RAMs gated from observability during the logic test (and therefore do not require array initialization) generally require less CPU time to process than an equivalent circuit with the RAMs participating in the logic test. These designs usually depend on functional self-testing means to test the RAMs.

Performance is shown for a benchmark circuit which was built to test these tools when they were being developed. The statistics for this circuit (called the "Hardware Test Vehicle," or HTV) are shown in **Table 1**.

One of the reasons why a circuit such as the HTV can take so long to process is that it has relatively large logical RAMs (8K addresses). Since it requires 8192 cycles to initialize all of the RAMs (some of the smaller RAMs are initialized multiple times), with each cycle containing 29 clock pulses and a channel-scan operation, it can take a long time to check the array sequences and to simulate them.

Another performance factor is the number of RAMs and how often they must be calculated during simulation. The RAM simulation within CLS-LFSRSIM is performed essentially by behavior, *not* by compiled code. The more often RAMs must be simulated, the slower the simulation. The number of faults which must propagate through RAMs in order to be detected can have a major influence on the number of times RAMs must be calculated. Approximately 90% of the fault-simulation time for the HTV circuit was spent calculating RAMs due to faults which were fed into them.

The system supports two different ways of circumventing this problem:

1. Break up the RAM into smaller RAMs—initialize in parallel. If a large logical RAM, e.g., 32K bits by 36 bits, is implemented using thirty-six 32K-bit by 1-bit RAM macros, the array initialization requires 32K cycles. If, however, the RAM was implemented using thirty-six 8K-bit by 4-bit RAM macros, the array initialization may require only 8K cycles, since it may be possible to write 144 bits simultaneously. The more cells that can be written simultaneously, the shorter the array initialization time.
2. Gate the RAM from being observed during logic self-test. Circuits with no RAM are simulated very rapidly through the BIST system. The same performance can be obtained for circuits with embedded RAMs by blocking the RAMs from being observed in the self-test state. If a RAM can be tested using a functional self-test or by a separate, algorithmic self-test, there is no need to include it in the logic self-test—it just gets in the way. The best approach is to have two self-tests, a logic self-test and an array self-test, and never overlap them.

Table 2 Statistics for the BIST chip.

39,616	Logic gates	0	RAMs
774	SRLs	4	Channels
416	SRLs in longest channel	0	RAM cells
104,690	Stuck faults	58,787	Fault equivalence classes
1,048,576	Logic test cycles	1	Clock per logic test cycle
		1,048,576	Total logic test clock events (FM simulated)
94.1	% Fault coverage		

CPU minutes	*Activity*	*CPU minutes*	*Activity*
1	Check design rules	3	Compiling circuit for simulation
0	Check array sequences	0	CLS-LFSRSIM GM simulation of 0 array initialization clock pulses
1	For miscellaneous	13	CLS GM and FM simulation of 1,048,576 test clock pulses
16	Total for all simulation and signature generation	16	Total for all simulation and signature generation
18	Total for all steps		

Note: CPU times given are for an IBM 3090-600 E processor.

Another, relatively small, circuit was processed through this system. It is a chip with fewer than 40K logic gates, only four STUMPS channels, and boundary scan. The channels are longer than the ones on the HTV circuit. A significant difference between this chip and the previous circuit is that the chip has no RAMs. It also has significantly fewer clocks (8 compared to 29), and they were only clocked one per test cycle with rotation (i.e., a different clock was pulsed in each subset of eight test cycles). This type of clocking was used because it was to be implemented in the higher-level package used to test the chip. The statistics for this circuit (identified here as the "BIST chip") are shown in **Table 2**.

The BIST chip required many more cycles to achieve even 94% test coverage; however, it was being penalized, since no faults in the I/O logic were detected because it was in boundary-scan internal state. If a boundary-scan external test were added to ensure that the I/O pins and logic were tested, the test coverage would be much higher (about 5%) [10, 11]. It is important to note that an entire 40K-gate chip was processed in about 18 CPU minutes on an IBM 3090-600 E—including fault simulation of

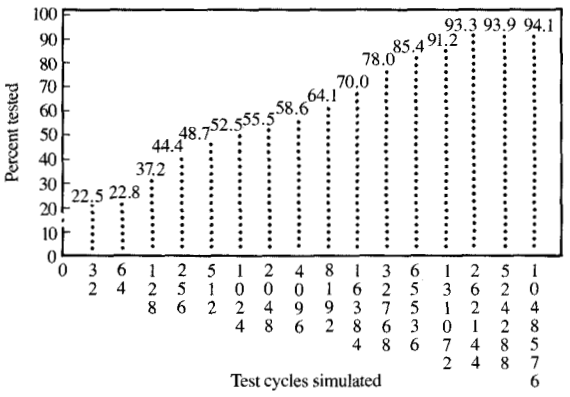

Figure 6

Test coverage curve for the BIST chip. This logarithmic coverage chart is output from the CLS-LFSRSIM simulator. Note that coverage started at 18% from *a priori* fault analysis.

over one million random patterns—with signatures also being generated. This processing time includes the effective simulation of more than 430 million single-bit shifts of the STUMPS PRPG, channels, and MISR in order to generate the patterns and signatures.

The test coverage curve for the BIST chip (**Figure 6**) shows some interesting features. A normal (linear) coordinate system would only show that test coverage climbs very fast and then flattens out. The logarithmic chart shows that there are some faults that require a certain base level of patterns to be applied before they begin to be detected in any great numbers. Notice that the chart is steeper in some areas than in others (between test cycles of 1024 and 2048, less than 3% of the faults are detected; between test cycles 16 384 and 32 767, 8% of the faults are detected). This may indicate that there are many faults clustered around some blocking logic which requires that a certain threshold of patterns be applied before those faults can propagate through the blocking logic.

For example, a ten-input AND gate with completely random inputs requires about 4000 patterns as a base before any reasonable number of fault effects can flow through that gate. A twelve-input AND requires about four times that many patterns. High-fan-in OR gates cause the same problem. The coverage chart produced by CLS-LFSRSIM gives the user a feel for how often high-fan-in gates (or equivalent logic) are blocking the testability of the circuit. Often the high-fan-in function is distributed over many gates and is not easily found by inspection of the logic structure. The STFA tool helps pinpoint the areas that may be causing problems and thus helps a designer to identify where circuit modifications would be effective in improving the circuit's random-pattern testability.

Conclusions

The overall performance of the BIST support system is good. The main bottleneck for the system is large logical RAMs, which require many initialization cycles to fill them with random data. The system supports two different ways of circumventing this problem:

1. Break up a large logical RAM into smaller physical RAMs.
2. Gate the RAM from being observed during logic self-test.

When random-pattern self-test is used, the random patterns may not provide adequate test coverage when the target number of patterns is applied. Even when the design passes all of the LSSD and self-test rules, it may still not meet the test-coverage requirements without some modification. This aspect of designing circuits for random-pattern testability is a new and unfamiliar problem for logic designers.

The use of the Self-Test Fault Analyzer (STFA) helps when the circuit must be modified to make it more random-pattern testable. The STFA helps point out certain nets that impede the detection of many faults. By using this tool, it is possible to identify a few key sections of the logic that must be modified to improve the random-pattern testability of the circuit.

As logic designers (and logic synthesis programs) become familiar with the ramifications of random-pattern BIST, they may be able to foresee problems and fix them before even running BIST tools. Until then, however, it will be an ongoing learning experience.

Appendix: Glossary

The following definitions are provided for the acronyms found in this paper.

- BIST Built-In Self-Test. Features designed into the circuit that enable it to test itself using pseudorandom test-pattern generators and signature analysis.
- CARPET Circuit Analysis for Random-Pattern Extensive Testing. A tool which estimates test coverage without actually simulating any patterns.
- CLS-LFSRSIM The Compiled Logic Simulator and the LFSR Simulation Monitor. The high-speed simulator used to compute signatures and perform fault simulation of the self-test patterns.
- EDS The IBM Engineering Design System. This is a system of design automation tools which includes

support for logic entry, design verification, physical design, timing analysis, design rules checking, automatic test generation, and BIST testability analysis.

- LFSR Linear Feedback Shift Register. Used either as a pseudorandom pattern source or as a signature analyzer.
- LSSD Level-Sensitive Scan Design. A circuit design technique which makes all latches scannable and ensures that no race conditions will exist in the testing environment. This technique has been used both within and outside IBM with much success.
- MISR Multiple-Input Signature Register. This is an LFSR used as a signature register with multiple parallel inputs.
- PRPG Pseudo-Random Pattern Generator. This is an LFSR used as a pseudorandom pattern source, perhaps with multiple parallel outputs.
- SISR Single-Input Signature Register. This is an LFSR used as a signature register with a single input into its leftmost bit.
- SRSG Shift Register Sequence Generator. Another name for a PRPG.
- STFA Self-Test Fault Analyzer. A tool which helps locate the causes for low test coverage of random-pattern-resistant logic.
- STUMPS Self-Test Using MISRs and Parallel SRSGs. A self-test circuit design which connects latch scan strings between an LFSR pattern source and an LFSR signature register.
- TARP Testability Analysis for Random Patterns. The TARP controller provides a single user interface to several random-pattern-testability analysis tools.
- TDA The Test Design Automation mission within the EDS organization of IBM. It provides design automation tools for deterministic test generation, fault simulation, and self-test.
- TDF Test Data File. This file is the main output from the TDA system. It contains several subfiles, including the logic model, fault model, and simulation results. It is later combined with other data (e.g., physical design data) before being sent to manufacturing.

References

1. P. H. Bardell and W. H. McAnney, "Self-Testing of Multichip Logic Modules," *Proceedings of the 1982 IEEE International Test Conference*, November 1982, pp. 200–204.
2. B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Techniques," *Digest of Papers, 1979 IEEE International Test Conference*, October 1979, pp. 37–41.
3. J. J. LeBlanc, "LOCST: A Built-In Self-Test Technique." *IEEE Design and Test of Computers* **1**, 45–42 (1984).
4. F. Motika, J. A. Waicukauski, E. Lindbloom, and E. B. Eichelberger, "An LSSD Pseudo-Random Pattern Test System," *Proceedings of the 1983 IEEE International Test Conference*, November 1983, pp. 283–288.
5. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 14th Design Automation Conference*, June 1977, pp. 462–468.
6. E. M. Barna, "Self-Test Diagnosis Using Parallel Superpositioning," *IBM Tech. Disclosure Bull.* **28,** 609–614 (1985).
7. V. P. Gupta, S. T. Patel, and J. A. Waicukauski, "Diagnosis of Array Failures in a Self-Test Environment," *IBM Tech. Disclosure Bull.* **31,** 430–433 (1989).
8. J. A. Waicukauski, V. P. Gupta, and S. T. Patel, "Diagnosis of BIST Failures by PPSFP Simulation," *Proceedings of the 1987 IEEE International Test Conference*, September 1987, pp. 480–484.
9. Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge, "HSS—A High Speed Simulator," *IEEE Trans. Computer Aided Design Int. Circ. & Syst.* **CAD-6,** 601–617 (1987).
10. P. T. Wagner, "Interconnect Testing with Boundary Scan," *Proceedings of the 1987 IEEE International Test Conference*, September 1987, pp. 52–57.
11. P. Goel and M. T. McMahon, "Electronic Chip-in-Place Test," *Proceedings of the 1982 IEEE International Test Conference*, pp. 83–90.

Received May 16, 1989; accepted for publication August 15, 1989

Brion L. Keller *IBM Data Systems Division, P.O. Box 8008, Endicott, New York 13760.* Mr. Keller received a B.S. in computer science from Pennsylvania State University in 1979, and is working to obtain an M.S. in computer engineering from Syracuse University. He joined IBM in 1979, developing a software paging and data base access method for use with very large design automation systems. Mr. Keller is currently an advisory programmer in the Test Design Automation mission of the Engineering Design System. He is responsible for the design of future deterministic and self-test systems. He has authored several technical disclosures and papers presented within IBM, and received an IBM Outstanding Technical Achievement Award for his technical leadership in the development of the self-test design automation system. Mr. Keller is a member of the ACM and is chairperson of the local chapter.

Thomas J. Snethen *IBM Data Systems Division, P.O. Box 8008, Endicott, New York 13760.* Mr. Snethen holds a B.S. in electrical engineering from the University of Missouri and an M.S. in electrical engineering from M.I.T. He is a senior engineer in the Test Design Automation mission of the IBM Engineering Design System. His responsibilities include requirements definition and system design of test-generation and fault-simulation software. Mr. Snethen joined IBM in 1965 and has held several positions in Endicott and East Fishkill in the area of logic test generation. He is the author and co-author of several papers presented at various IEEE-sponsored conferences related to logic testing. Mr. Snethen was awarded a patent for the Simulator-Oriented Fault Test Generator and is the author of several IBM technical disclosures. He received an IBM Outstanding Innovation Award for his software architecture work in support of built-in self-test. Mr. Snethen is a member of Eta Kappa Nu, Tau Beta Pi, Sigma Xi, and the Institute of Electrical and Electronics Engineers.