Advanced microprocessor test strategy and methodology

by W. V. Huott
T. J. Koprowski
B. J. Robbins
M. P. Kusko
S. V. Pateras
D. E. Hoffman
T. G. McNamara
T. J. Snethen

This paper describes the overall test methodology used in implementing the S/390® microprocessor and the associated L2 cache array in shared multiprocessor designs, the design-for-test implementations, and the test software used in creating the test patterns and in measuring test effectiveness. Microprocessor advances in architectural complexity, circuit density, cycle time, and technology-related issues, coupled with IBM's high requirements for quality, reliability, and diagnosability, have made it necessary to develop testing methods and attain quality levels that far exceed what others have approached.

Introduction

The advent of deep-submicron technology has given rise to integrated circuits containing hundreds of thousands of logic gates, embedded memories approaching the megabit range, I/O counts in the thousands, and operating frequencies in the hundreds of MHz. Along with the benefits of such characteristics and the design flexibility necessary to achieve them come severe design and test challenges. In particular, traditional methods of testing semiconductor devices are quickly becoming obsolete. The use of functional patterns derived for design verification as manufacturing test patterns is becoming increasingly unacceptable. Some of the most severe problems associated with this approach are high test development times, defect coverages that are low or hard to measure, and poor diagnosability. As far back as fifteen to twenty years ago, test techniques were developed within IBM and in industry which based analysis on the design structure rather than on functionality [1]. Within IBM, these techniques have been evolving from the 308x testing in the early 1980s to the 3090* testing in the later '80s, to high-density CMOS parts in the early '90s [2–13]. These techniques have led to the development of automatic test-pattern generation (ATPG) algorithms and tools [14–19]. Although ATPG-based approaches to digital testing have met with some success, they also are becoming increasingly ineffective as chip sizes increase. Indeed, time requirements for ATPG algorithms grow nonlinearly in relation to the size of the circuit under test [20].

However, the largest problem with both the functional and ATPG-based test techniques is their reliance on the use of automatic test equipment to apply the test patterns to the device's external inputs and measure responses on the device's external outputs. This approach does not provide a means to adequately detect all of the device's internal defects. Direct access to the internal structures of a device is necessary. This requirement has led to the development of design-for-test (DFT) and built-in self-test (BIST) techniques and methods [21–27].

DFT techniques consist of design rules and constraints aimed at increasing the testability of a design through increased internal controllability and observability. The most popular form of DFT is scan design, which involves

0018-8646/97/$5.00 © 1997 IBM

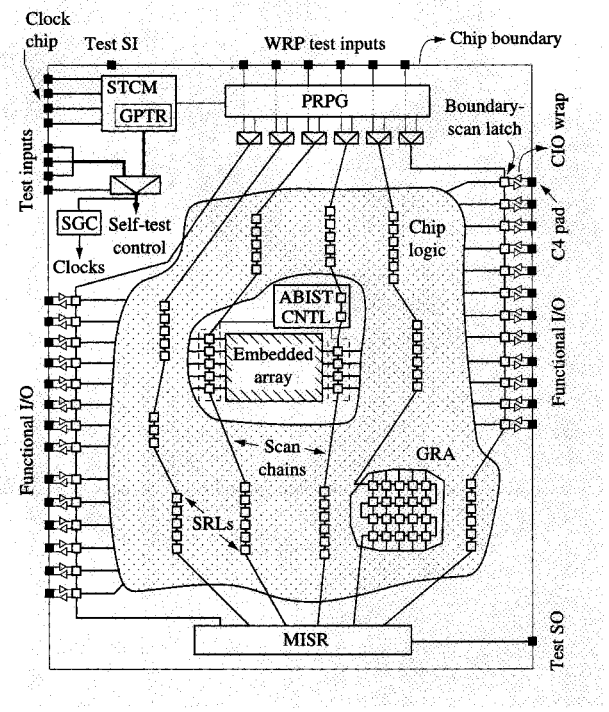

Figure 1

DFT overview.

modifying all internal storage elements such that in test mode they form individual stages of a shift register for scanning test data stimuli and scanning out test responses.

Conceptually, the BIST approach is very simple. It is based on the realization that much of a circuit tester's electronics is semiconductor-based, just like the products it is testing, and that the challenge in ATE design, and many of the emerging limitations in ATE-based testing, lie in the interface to the DUT. In light of this fact, the BIST approach can be described as an attempt to move many of the already semiconductor-based test equipment functions into the products under test and eliminate the complex interfacing. This embedding of functionality has many benefits; some of the more important ones are the following:

- The burden on and complexity of external test and dynamic stress equipment are drastically reduced.
- The cost of product interface equipment, interface boards, space transformers, probes, etc. is reduced.
- Embedded memories and other structures can easily be accessed for testing purposes.
- All tests can be run at-speed, i.e., at the system operating frequency, which provides for better coverage of delay-related defects.
- The approach can be used after product assembly for system and field testing.

The clear benefits of DFT and BIST encouraged the extensive use of these techniques in the design of the currently developed S/390* microprocessor and associated L2 cache chip. Indeed, the high complexity and density of these submicron CMOS-based devices, coupled with the need to optimize testing across all levels, minimize device silicon, optimize test equipment, decrease time to market, and achieve an extremely high shipped-product quality level, made the use of DFT and BIST essential. Novel and innovative BIST techniques were developed to address some of the unique test challenges that arose from these state-of-the-art designs.

This paper gives an overview of the test methodology and describes the various DFT and BIST techniques used. It then discusses some of the benefits of these DFT features in hardware debug. The paper concludes with a brief description of the test-generation software and fault-model build, and the use of the fault-model/test software in generating the test data.

Overview of test methodology

The test methodology was optimized across multiple IBM divisions. This included a combination of DFT options that dramatically reduced the required capital equipment investment in test equipment. Potentially, the cost of a full-speed full-input/output tester for this product could have exceeded $8 million per tester. Instead, the devices were tested and their performance verified using a low-cost tester. **Figure 1** shows the DFT and BIST techniques which were used.

To reduce the number of full-speed tester channels required, a boundary-scan DFT technique was implemented. It consists in placing a scannable memory element adjacent to each chip I/O so that signals at the chip boundaries can be controlled and observed using scan operations and without direct contact. This boundary-scan chain is also needed for the logic BIST technique. Access to the boundary-scan chain as well as to most of the DFT and BIST circuitry is achieved through a custom five-wire interface similar to the standard IEEE 1149.1 TAP approach [28]. This interface is used to initialize and control the various on-chip BIST controllers and other DFT hardware during both system test and manufacturing test. A state machine within each chip, referred to as the self-test control macro (STCM), is used to control internal-test-mode signals and the sequencing of all test and system clocks while in test mode. Instead of testing the performance of the device at full speed through the

pins, an on-chip phase-locked loop (PLL) was used to multiply the incoming tester frequency to bring it up to the operating frequency of the chip. Additional self-generated clock (SGC) [22, 29, 30] circuitry is then used to generate the various system clock sequences needed to properly exercise all portions of the chip.

The BIST techniques can be divided into two major categories: logic BIST (LBIST) to test at-speed the logic in the devices, and array BIST (ABIST) to provide at-speed testing of the embedded arrays (i.e., RAMs). The basic idea in LBIST is to add a pseudorandom-pattern generator (PRPG) to the inputs and a multiple-input signature register (MISR) to the outputs of the device's internal scan chains. A BIST controller generates all necessary waveforms for repeatedly loading pseudorandom patterns into the scan chains, initiating a functional cycle (capture cycle), and logging the captured responses out into the MISR. The MISR compresses the accumulated responses into a code known as a signature [31]. Any corruption in the final signature at the end of the test indicates a defect in the chip. This LBIST architecture is known as a STUMPS [6] architecture (self-test using MISR and parallel shift register sequence generator), and the scan chains connecting the PRPG and MISR are defined as the STUMPS channels.

Although pseudorandom patterns achieve high test coverage for most scan-based designs, some areas within the design may be inherently resistant to testing with such patterns. Supplemental patterns designated as weighted random patterns (WRP) [15, 16] are therefore used during manufacturing test. WRP avoids the large test data volume that would be needed to drive conventional stored-pattern logic tests. External tester hardware is used to force individual bits in scan-based random test patterns to be statistically weighted toward a logic one or zero. Compared with LBIST alone, this method greatly reduces the number of random patterns needed for obtaining high test coverage, thereby greatly reducing test time.

For the ABIST test, a controller based on a programmable-state machine is used to algorithmically generate a variety of memory test sequences. As with LBIST, test patterns can be applied to the embedded array at cycle speeds. Because of the regular structure of arrays, an ABIST controller can be shared among several arrays. This not only reduces the overhead per array, but allows for decreased test times, since the arrays can be tested in parallel.

To ensure good-quality chips, several IBM standard manufacturing tests are applied. In addition to those already mentioned, the types of tests include a boundary-scan I/O test, parametrics, IDDQ, excessive voltage and temperature stressing, and dynamic burn-in [13]. Stressing the devices with voltage stressing and burn-in conditions helps guarantee the very high quality and reliability necessary for the mission-critical applications for S/390 servers.

The DFT implementation requires chip area dedicated to test functions; however, the bulk of the logic is also used for system initialization, recovery, and system failure analysis. The amount of logic dedicated to manufacturing test on these high-density CMOS parts utilizes less than 1% of the overall chip real estate.

Design for test—LBIST

LBIST is used for manufacturing test at all package levels and for system self-test. The main LBIST components are a PRPG and a MISR. These two components are connected to chip scan chains to form the overall LBIST structure.

The basic LBIST logic test sequence used to apply test patterns is as follows:

1. The PRPG and MISR are initialized to a predetermined state known as a *seed*. Then, the circuitry loops on Steps 2 and 3 for n patterns.
2. Scan clocks are applied to the PRPG, MISR, and system latches so that a pseudorandom pattern is generated by the PRPG and loaded into the system latches. Simultaneously, the result of the previously applied test pattern is compressed from the system latches into the MISR.
3. System clocks are applied to the system latches to test the logic paths between the latches. Test patterns are both launched and captured by the latches in the scan chains.
4. After n repetitions of Steps 2 and 3, the signature in the MISR is compared against an expected predetermined signature that was calculated during the test-pattern generation and simulation process.

Although the LBIST sequence is straightforward, there are multiple means to apply the sequence to perform different categories of logic tests. If the test is required to verify only that the logic structure between the latches is correct and has no stuck-at faults, the LBIST test can be applied with static, nontransitional patterns. The time between launch of data from one system latch and data capture in another system latch is irrelevant, so data are scanned into the latches in a nonskewed state such that the master and slave latches contain the same data. When system clocks are applied, there is no transition of data on the launching latches.

If the LBIST test is to determine not only that the logic between system latches is correct, but also that the propagation delay from one system latch to another occurs within a predetermined delay, a transition test is applied. In this test, data are scanned into the latches in a skewed state such that the master and slave latches potentially have different values so that the launch clock will create

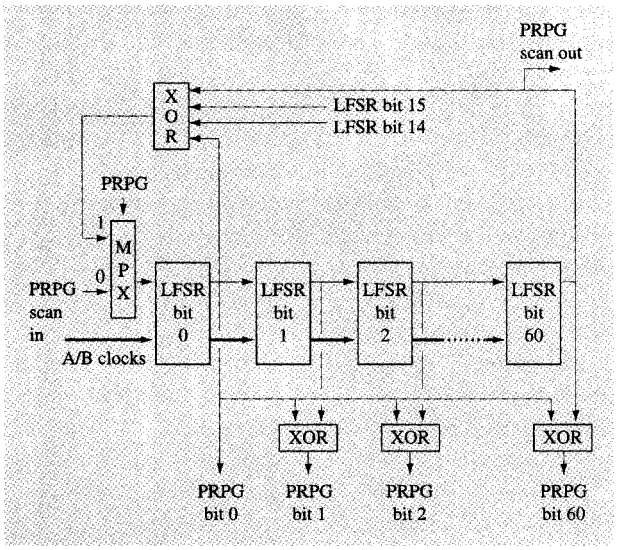

Figure 2

LBIST PRPG architecture.

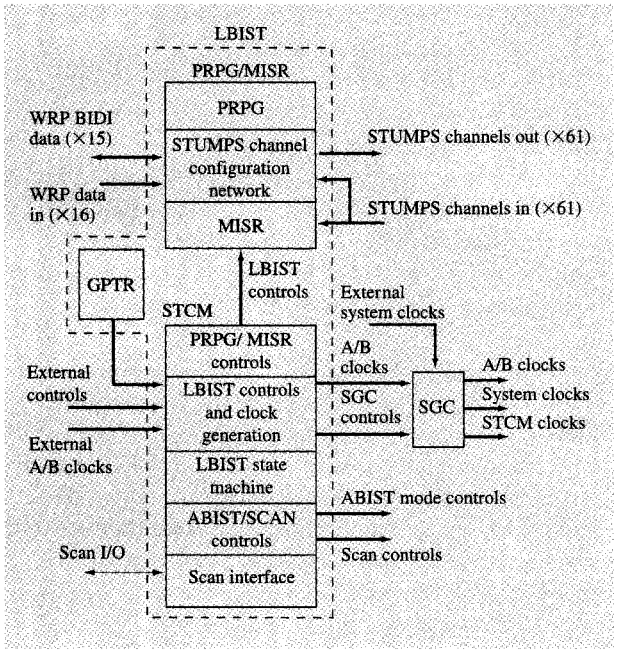

Figure 3

LBIST high-level architecture.

transitions at the latch outputs. Then precisely timed launch and capture clocks are applied to the system latches via the SGC circuitry.

LBIST is used on the tester during manufacturing test and during system self-test. During manufacturing test, the tester can apply the necessary signals to scan the shift-register chains, cycle the PRPG and MISR, and apply the system clocks at the proper time. In the system, there are no available resources external to the chip to control the LBIST circuitry. These controls are generated on-chip by the self-test control macro (STCM). The STCM executes the LBIST test sequence in a stand-alone manner. In fact, an entire self-test sequence of the entire system can be initiated at a customer office via modem/service processor controller.

- *LBIST design implementation*

Several unique features were required in the logic implementation to support the various aspects of the LBIST methodology. The PRPG shown in **Figure 2** is a 61-bit linear-feedback shift register (LFSR) with a feedback configuration utilizing taps 0, 14, 15, and 60.

To minimize data dependencies, the outputs of the LFSR are passed through an XOR spreading network before being applied to the logic. This spreading network is used to minimize latch adjacency dependencies between subsequent stages of the LFSR. Each stage of the LFSR has an associated two-input XOR which is fed from that stage and bit 0 of the LFSR. The output of the LFSR is applied to the appropriate STUMPS channel scan input.

The MISR is also 61 bits long and has a feedback configuration similar to that of the PRPG. Unlike the PRPG, the MISR has a two-input XOR between each of the latch stages, which allows for 61 bits of data from the STUMPS channel scan outputs to be clocked into the MISR on each LBIST scan cycle in the process of generating the signature.

The primary purpose of the STCM in **Figure 3** is to control the on-chip LBIST test operation; however, it also functions as the main interface and controller for all other test functions, with the exception of ABIST execution, which has its own independent test engine. The functions of the STCM are as follows:

- LBIST scan-clock generation and sequence controls.
- Scan-chain configuration controls.
- ABIST initialization.
- External clock controls.

The mode of operation, or *test mode*, is determined by four bits of the general-purpose test register (GPTR). The GPTR is a register that is initialized prior to each test with static control information. The GPTR consists of scan-only latches, and its state remains constant throughout the application of a given set of tests. Details of the GPTR are discussed later.

• *LBIST controls and scan-clock generator*

The LBIST controls and scan-clock generator in **Figure 4** consist of a state machine and static control latches that are initialized by a scan operation. It contains the following major components:

1. *A/B clock counter* The A/B clock counter is loaded with the number of A/B clock pairs to be generated during a STUMPS channel load/unload sequence. Usually this number is the length of the longest STUMPS channel.
2. *A/B clock-generation logic* The A/B clocks required during the LBIST test sequence are generated within the STCM. These clocks are derived from the system clocks that run the LBIST engine. Static latch controls are provided to suppress the first A clock and/or suppress the last B clock so all possible nonskewed or skewed load/unload scan sequences can be applied.
3. *Pattern counter* The pattern counter specifies the number of patterns to be applied. A pattern is defined as a channel load/unload sequence followed by the application of system clocks.
4. *SGC sequence controls* The SGC sequence controls consist of a system clock-sequence register and an SGC-go generator that indicates to the SGC circuitry when to apply the specified system clocks.
5. *State-machine start-up and completion logic* The STCM is initialized by a scan operation, and upon receiving a *start-test* pulse, it executes to completion without external intervention. The start-up and completion logic generates initial STCM reset and enable signals and controls for clocks, the PRPG, and the MISR. At test completion it produces an *LBIST-done* signal.
6. *GPTR* The GPTR is a register that provides static control signals to the chip. In the IBM manufacturing test environment, the number of test I/Os contacted at wafer test is limited to 64. If more than 64 static control lines are required, they are provided by the GPTR. The GPTR is on a separate A/B clock distribution network so that it can be scanned without affecting the state of other system latches.
7. *Static-test-control logic* This logic uses the states of the test inputs and the GPTR to generate high-level chip control signals that configure the functional logic to the proper state for test or system execution.

• *Scan-chain configuration controls*

The scan chains can be configured in several ways depending on the specified test mode:

- *LSSD (level-sensitive-scan-design) [3] mode* All of the chip latches are configured in one long scan chain. This mode is used whenever all latches of the chip must be initialized to a specific state. It is the primary chip interface for both initialization and observation during test and system operation.
- *Boundary-scan mode* This mode is a subset of the LSSD mode in which only the boundary-scan [24] latches and GPTR are contained in the scan chain. This mode is used during I/O, parametric, and interconnect modes of testing.
- *GPTR mode* This mode is a subset of the LSSD mode in which only the GPTR latches are contained in the scan chain. This mode is used to modify the state of the GPTR without changing the state of the chip system latches.
- *WRP mode* This design implementation of WRP uses fifteen scan chains. In this mode, the scan latches are configured into the fifteen independent chains. This mode is used for applying WRP and deterministic test patterns during manufacturing test.
- *LBIST mode* In this mode, the scan chains are configured as 61 STUMPS channels connected to the PRPG and MISR.

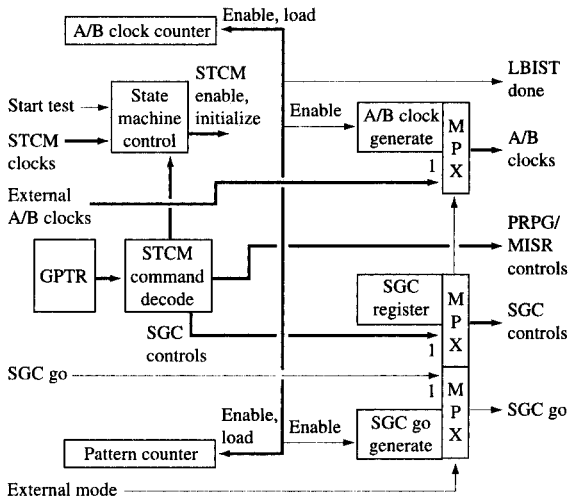

Figure 4

LBIST STCM high-level architecture.

• *ABIST initialization*

The ABIST engines are local to the different arrays on the chip, so there is no central ABIST state machine. ABIST engines are initialized by a scan-load operation. The function of the STCM is to specify the appropriate ABIST mode on the basis of the state of the test-mode bits in the GPTR and start the clocks to the ABIST engines. The STCM starts the ABIST tests by generating an *SGC-go* to the SGC circuitry when a *start-test* signal is received. Each

of the ABIST engines runs to completion and issues an *ABIST-done*. The STCM combines these done signals into one *ABIST-done* for the chip and propagates that signal to a chip output.

- *External clock controls*

In all cases where internally generated clocks are used, external clocks can also be applied to generate the same test sequences. LBIST, WRP, or deterministic stored patterns can be applied with internal clocks or external tester-generated clocks without modifying the stimulate/measure latch data. This feature allows for the debugging of timing or data problems at slow speeds and is used quite often to verify the integrity of the patterns before attempting to apply the patterns at fast internal chip cycle times.

- *LBIST circuit design implementation*

The LBIST macro is designed to communicate with the rest of the chip in an asynchronous manner, since there are no critical timing signals to or from the LBIST macro. This clocking approach is possible because of the asynchronous nature of the design; it allowed the macro to be synthesized and physically designed with no custom circuit layout techniques.

Design for test—ABIST

Memory array built-in self-test (ABIST) has traditionally consisted of a finite-state machine logic engine designed to apply a prescribed fixed set of memory test patterns to the memory array(s) under test. These tests typically include data blankets (all 0s and all 1s), word-line stripes, bit-line stripes, and checkerboards. Although these are performed in multiple addressing modes, including unique addressing (read before write), this pattern set is the limit of most finite-state-machine ABIST engines and is mostly unchangeable. A simplified logic model has registers of latches set up as counters in nested loops to perform the series of array addressing and reading/writing, and a two-latch data register that fans out to all of the array even data bits and odd data bits, respectively. Data patterns are limited by the combinations of this two-bit data register. The data out of the array are compared against this data-in register, and the pass/fail results are latched. Execution of finite-state-machine ABIST involves initializing the chip for ABIST, usually through the scan chain, and applying a sufficient number of system clocks, either externally or through the SCG, for the finite-state machine to reach its final state. The ABIST pass/fail results (and repairable array addresses for arrays with redundancy) are scanned out through the scan chain.

For the S/390 microprocessor and L2 cache designs there are several different custom embedded memory array designs, often having different test requirements. The tight cycle-time and access-time requirements on these arrays caused their designs to be quite aggressive. Dynamic and self-resetting circuit techniques were used extensively. These aggressive arrays dictated a need not just for high-speed ABIST engines, but for a testing scheme that was flexible enough to help diagnose potential problems, stress array performance, and provide production-level testing ability.

A programmable ABIST design was implemented for these high-performance arrays [32, 33], with microprocessor-like function. The ABIST program to be applied is scanned into a custom microcode array, and each instruction is decoded, executed, and applied to the array by the ABIST microprocessor. The programmable ABIST design comprises eight basic components, as shown in **Figure 5**:

1. *Microcode array* Scannable register array that contains the ABIST program to be executed (typically eight instructions).
2. *Pointer control macro* Register that controls the address of the ABIST instruction being executed.
3. *Address control macro* Register that controls the address to be applied to the array.
4. *Data control macro* Register that controls the data to be applied to the array.
5. *Read/write register* Register that controls the read/write mode to be applied to the array.
6. *Result compression macro* Registers that either log pass/fail results and failing address data or store a passing/failing signature on the basis of the array data outputs.
7. *Test control interface* Logic which communicates to the STCM and controls the test modes of the array.
8. *Access timer macro* Digitally programmable timer which measures the access time of the array.

The core blocks of the programmable ABIST design are the microcode array, pointer control macro, and test control interface. For nearly all of the different memory array designs, each ABIST engine uses the same basic design for these core blocks. The address, data, and read/write macros need to vary only in size according to the address/data widths and read/write configurations of the arrays under test. The result compression macro generally consists of either a MISR (multiple-input signature register) on the arrays without redundancy or a data comparator and failed address registers on the arrays with redundant repairable addresses. Redundancy is a feature sometimes used on larger arrays to improve yield by replacing failing word lines with redundant (spare) word lines.

The microcode array is custom-designed as a dense, scannable, read-only register array, with fast access. It

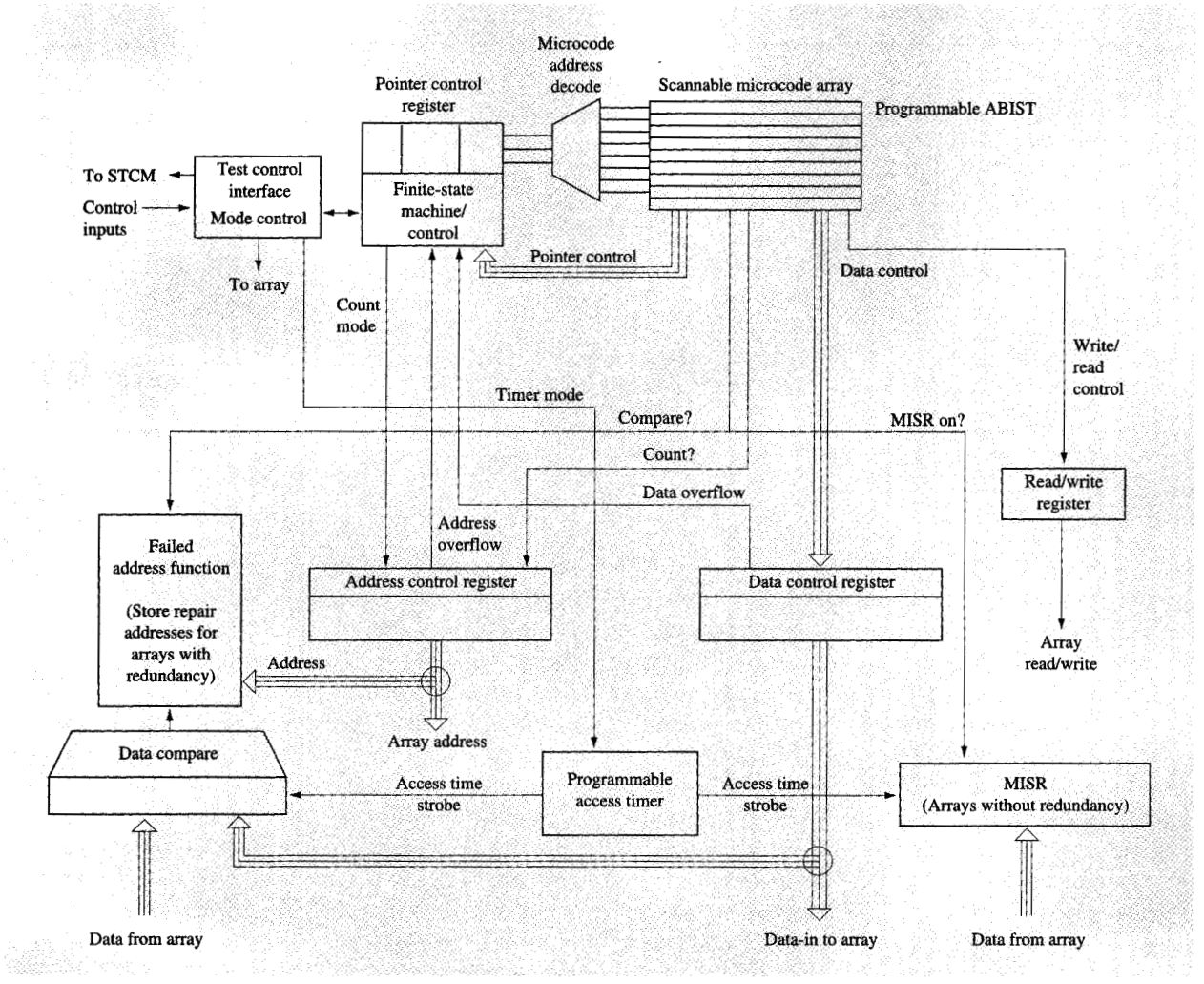

Figure 5

ABIST block diagram.

generally contains eight ABIST microcode instructions. Its scannability gives the ABIST great flexibility in that it can so easily be reprogrammed. Since it consists mainly of scan-only latches, the microcode array is implemented in a very area-efficient manner.

The programmable ABIST instruction set is small but quite powerful. A microcode instruction is basically broken down into five command fields:

Field 1: Pointer control macro branch commands.
- Unconditional branch
- Conditional branch/loop based on address overflow condition
- Conditional branch/loop based on data overflow condition

Field 2: Address macro commands.
- Increment/decrement/hold

Field 3: Data macro commands.
- Hold/reset/invert/rotate/rotate with invert/etc.

Field 4: Read/write register commands.
- Read/write

Field 5: Result compression macro commands.
- Compress/do not compress

During the execution of an ABIST program, the current microcode instruction is accessed, each of the command fields are decoded by the appropriate ABIST macros, and the resulting pattern is applied to the array under test. This process continues until the ABIST program has

Table 1 ABIST programming examples

	Field 1	*Field 2*	*Field 3*	*Field 4*	*Field 5*
A background of blanket 0s can be written into the memory array using just one instruction:	Loop until address overflow	Increment	Reset	Write	Compress
Reading back the blanket 0s is just as easy:	Loop until address overflow	Increment	Reset	Read	Compress
Now to convert the blanket 0s to 1s in a unique address mode (read before write) would take instructions:	Unconditional increment Increment on address else branch −1	Increment Hold	Reset Invert	Read Write	Compress Compress
Now to march a data pattern of 0s across the background of 1s using a nested loop:	Unconditional increment Increment on data else −1 Increment on address else −1	Hold Hold Increment	Rotate w/inv Hold Invert	Write Read Read	Compress Compress Compress

Note: In the above examples, 'Loop until address overflow' is shorthand for Loop on the current microcode instruction address until an overflow is received from the address control macro. 'Increment on data else −1' is shorthand for Increment the microcode instruction address pointer on a data control macro overflow or else decrement the microcode instruction address pointer by 1.

completed, final results are latched, and an ABIST_DONE signal is sent back to the self-test control macro (STCM).

For example, as shown in **Table 1**, a background of blanket 0s can be written into the memory array using just one instruction; several other examples are included in Table 1. Only a few instructions are needed to perform some very powerful operations.

While it may sound like an expensive proposition to include the full array data input width in the data control macro, it actually can be done quite cheaply with the programmable ABIST design. The data, address, and read/write input fields to the S/390 arrays are usually logical system cycle boundaries. This means that a system series latch, or at least a listening latch for LBIST, must exist at the data, address, and read/write ports of the arrays. Either system data or ABIST data are multiplexed into these latches depending on the test mode of the array. The programmable ABIST design is able to use these latches to form its data, address, and read/write control macros. The cost of these macros becomes just combinational logic for these macros, with a few control lines from the ABIST microcode array to tell the macro what to do and when. With the ability to economically provide full-data-width testing to the memory arrays, not only can a multitude of new test patterns be thrown at the arrays (such as marching and walking patterns, which are typically not done by finite-machine ABIST engines), but there are system benefits as well. For example, the logic in the ABIST engine can now be used to initialize the data in the arrays to good parity and ECC upon power-on reset or even during on-the-fly reset recovery after error hits in the system.

The use of MISRs for signature compression on the outputs of memory arrays without redundancy in conjunction with a programmable ABIST has many advantages. Many of these arrays have very wide data-out buses, which make a data-comparator type of compression difficult to do at speed without complex circuitry and wiring. These arrays are usually placed in some of the most congested locations of the chip, where silicon area and wiring channels are at a premium. A MISR lends itself very well to very wide data words while keeping wiring channel usage and circuit complexity to a minimum. A properly implemented MISR can also be operated at very high rates of speed because of its relative logic simplicity. The MISR function on the nonredundant arrays is actually integrated into the data output register of these arrays. Since the data output register is already a necessary component of the array function, the integration of the MISR logic becomes even more economical compared to a full data comparator.

Another limitation of a data-comparator type of compression is that the ABIST engine always has the burden of calculating the expected array data output for the comparator. This puts limitations on the complexity of the test patterns that can be applied to the array, especially with a finite-state-machine ABIST engine. With the MISR approach, the expected result is never calculated by the ABIST engine. All of the responses are merely compressed into the MISR final signature. The ABIST engine can generate patterns of any complexity while writing and reading them back in any order whatsoever. A programmable ABIST engine is well suited to generate patterns at these levels of complexity. The net results of a MISR approach with a programmable ABIST are more thorough pattern coverage and flexibility while maintaining high performance and minimum design overhead.

In addition to maximizing pattern coverage with the programmable ABIST, there is also much emphasis on

performance characterization of the memory arrays. Each of the ABIST engines is designed to cycle well below the specified system cycle-time limit of the chip. In fact, the ABIST is able to cycle below the expected cycle-time limits of many of the individual arrays. This is no small feat, since ABIST operating frequencies in some cases go beyond 500 MHz. The benefit of this is not just the guarantee of chip system cycle performance from the arrays; cycle-time results can also be examined to point of failure on a per-array basis. Point-of-failure results enable the quantification of cycle-time guardbanding for the arrays as well as possible qualification of the failing circuitry for future improvements. Because the access time of a memory array may occupy as little as 50% of a full system cycle, access-time measurements can be even more important than cycle-time measurements. Each ABIST engine is equipped with a digitally programmable access timer macro [34] which is able to measure the access time of each array to a resolution of nearly 100 ps. The desired access-time setting is scanned into the timer, and on each array clock the timer supplies an access-time strobe, delayed by the scanned setting, to the data-compression macro. An access-time strobe is applied to the data-compression macro for every cycle of the entire ABIST program, yielding a worst-case access-time measurement across every pattern and every address in the array. When the pass/fail settings of the timer for a particular array have been determined, the timer is then configured in a recirculating-loop mode which allows the timer to oscillate at a rate corresponding to the access time of the array under test. This frequency is divided to produce a lower rate and multiplexed off the chip for an easily measured, very accurate access time of the array, based on this oscillator frequency. The access timer macro also has a static mode which completely removes the access-time measurement from the ABIST test and allows for static functional debug.

The design and implementation of programmable ABIST engines for the S/390 microprocessor and L2 cache chips achieved high function at minimal cost. This approach provided high-speed testing capability, the flexibility needed for diagnosing difficult problems, and the ability to stress and measure array and redundancy performance, along with production-level testing capability. All this was accomplished in a design-efficient manner, minimizing real estate and functional timing impact.

Chip testing debug, analysis, and diagnosis

Although rigorous checking and verification of the design [35] is performed, the complexity of microprocessors often leads to some technology-related problems that are found during test of the hardware. The flexibility built into the BIST designs was invaluable in chip bring-up, both at the tester and in debugging the system.

One unique use of the BIST hardware was in isolating a hardware problem caused by coupled noise. A problem was first suspected when initial test runs showed that LBIST passed only in a very narrow (~100-mV) voltage range. At voltages above and below the narrow band, the LBIST signatures were intermittent and nonrepeating, and varied with voltage. First an attempt was made to find a single failing pattern that caused the fail, using a binary search with LBIST patterns. Since the MISR signature was known to be correct after X patterns, it must also have been good for all patterns less than X. With this knowledge, one simply changed the bits programmed in the pattern counter in a binary search fashion and ran LBIST in the known good-voltage range. The signature was not checked but was saved and used as the *golden signature* for additional analysis. (The benefit of this approach is that a new signature can be obtained in less than a minute, and no additional pattern generation is required.) Then the voltage was varied to see whether the passing voltage window remained the same.

It was found that certain patterns caused the good-voltage window to narrow from the previous pattern. These were identified as noisy patterns, and the state of the system latches was extracted from the LBIST sequence before and after these patterns were applied. The extracted patterns were then applied in a deterministic fashion and used to narrow down the source of the coupled noise.

Another unique use of LBIST was determining power-supply noise problems. LBIST can be programmed to apply a skewed or nonskewed load/unload sequence with or without system clocks. This feature was used to measure the power-supply noise at different levels of switching activity. Since LBIST runs in a continuous loop, it was straightforward to trace the V_{dd} supply and determine the delta-I noise and power-supply droop with different levels of switching activity based on the scan and system clock sequences applied. Worst case is a skewed load/unload sequence with system clocks applied. Best case is a nonskewed load/unload with no system clocks applied.

Within a complex microprocessor, delay measurements are very complicated to determine. Again, LBIST was used to isolate the worst-case delay paths between scan chains, using a technique similar to that for the coupled-noise analysis. A binary search was performed to find the failing pattern, but using cycle time as the search variable rather than the voltage. The LBIST patterns were narrowed down to those that failed at the slowest cycle time; these patterns were then extracted and used for deterministic analysis of timing-critical paths.

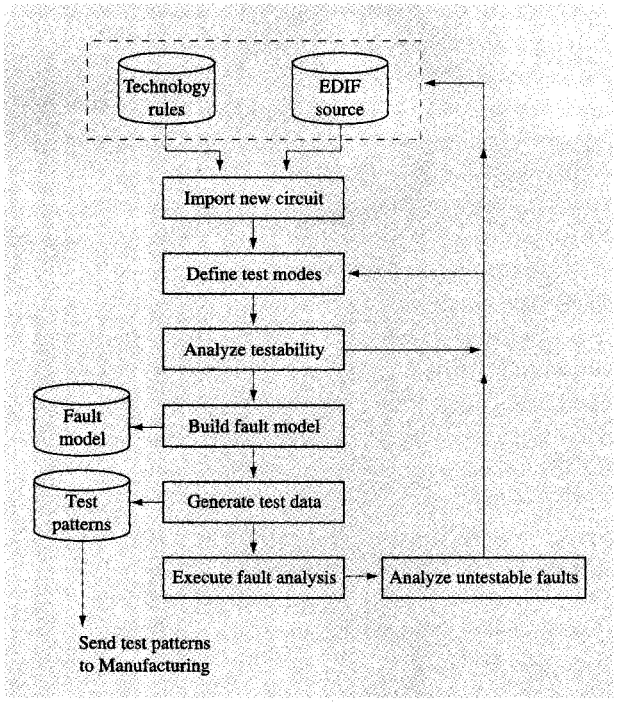

Figure 6

High-level TestBench methodology flow.

In the above cases, LBIST was able to be used to diagnose problems at the tester because of the flexibility designed into the LBIST circuitry. The analysis was performed, without requiring any pattern generation beyond the original LBIST patterns, and by simple edits to the initialization state of the LBIST scan setup.

Test software strategy

The wide variety of test techniques discussed in the preceding sections and the complexity of the processor design require powerful and flexible test analysis, generation, and diagnostic support. TestBench*, the IBM-developed test-generation tool, was selected to fill this role because of its state-of-the-art capabilities. It supports various design styles, encompassing several different clocking and scan approaches. Along with supporting efficient and varied test-generation and simulation techniques, TestBench understands the interaction of these test types. This support is provided via multiple test modes, where a *test mode* is the set of conditions required for test. In addition, there is a close working relationship between the TestBench development team and both the processor development and internal technology groups. An outcome of these partnerships is correct-by-construction test data, a critical component of any test methodology.

Figure 6 shows a high-level TestBench flow through the test-generation methodology. TestBench requires both a structural description of the processor design and a description of each technology cell or macro used in the design. The structural description is provided in EDIF (Electronic Design Interchange Format) or other languages. A model is built for the technology cells, and test functions are specified for the pertinent signals to define the various test modes. The steps are described in the following sections.

- *Analyze testability*

Analyze testability is a step in TestBench which checks the design for conformance to the established DFT guidelines. The DFT guidelines ensure that the required DFT structures are properly implemented, the clocks can be controlled as needed, and the logic is free of races which might not be correctly predicted by simulation and could result in invalid test data. In the system mode of operation, the design uses edge-triggered flip-flops, so that, viewed from a structural perspective, there appear to be race conditions throughout the design. The TestBench structure verification (TSV) takes this into account, with a simple assumption that clocks win all races. Example error conditions that TSV looks for are consecutive level-sensitive latches controlled by the same phase of a common clock, and edge-sensitive flip-flops gating other branches of their own clocks.

- *Generate test data*

Generate test data supports the several different types of tests-mentioned earlier: LBIST, ABIST, stored-pattern stuck-fault tests, I/O wrap, parametrics, IDDQ, and burn-in. The most familiar of these is stored-pattern tests for the logic, where the process consists in automatically generating input stimuli and performing fault simulation to produce the output responses and grade the fault coverage of the tests, keeping track of the faults that have been tested and the ones that remain. As another example, the test-generation process for LBIST consists in reading in the clock sequence for the LBIST tests and the initial seeds for PRPG and MISR, and performing fault simulation to grade the tests (as in stored-pattern simulation) and produce the expected MISR signature, which corresponds to the output responses for stored-pattern test. Another test type previously mentioned is burn-in. TestBench has no specific support for burn-in, but by using the TestBench multiple-test-mode support, the logic environment for burn-in test can be described so that tests can be generated for burn-in.

Most TestBench applications work with the circuit in a single test mode at a time, but there are a few instances in which there is some interaction among various test modes. LBIST is a prime example. The STUMPS configuration is

built upon a scan design, but the shift registers are fed by a PRPG and the shift-register outputs feed an on-product MISR. This scan mode is clearly different from the standard LSSD scan mode, in which the shift registers are connected to chip pins. A full-scan (LSSD) mode is used to initialize the design, including the PRPG and MISR. Then the design is switched to the LBIST mode, in which the PRPG and MISR are not scannable, but perform their intended test functions. In manufacturing test, the design is switched back to the LSSD scan mode at the end of the test to observe the signature. Thus, LBIST application involves a combination of test modes.

Also bringing together different test modes is the cross-mode mark-off aspect of multiple test modes. Faults marked off (detected) in one test mode may be automatically marked off in other test modes to avoid wasting time in redundantly testing the same faults. This cross-mark-off ability is one of the techniques which guarantee an efficient, compact test-vector set.

The clock circuitry and control designs presented a challenge to TestBench, since the clock block contains nonscannable latches. These latches control the scan operation, and TestBench does not support sequential logic in the clock-generation and scan controls. This tool limitation was circumvented by removing these latches from the model and replacing them with equivalent combinational logic and "pseudo primary inputs" that can be sequenced in such a way as to mimic the real operation. Of course, this necessitated additional steps. All of the test patterns had to be converted from the TestBench sequences, which use the pseudo primary inputs, to a form that could be applied to the hardware without the pseudo primary inputs, and with the appropriate clocking on the real inputs to produce the desired effect. This conversion was straightforward for LBIST and WRP data because these TestBench processes accept user-specified clocking sequences. Some other test-generation processes, such as I/O wrap, do not support clocking constraints, and for those processes the conversion was complicated by the need to look for and eliminate any tests that could not be applied on the real hardware. Fortunately, as it turned out, there were few automatically generated tests that could not be easily converted.

- *Analyze untestable faults*

Because of the mission-critical nature of the S/390 servers, the chips had to be tested to the highest product quality. The test coverage goal was greater than 99.9%. To understand the fault coverage, test generation with user-defined clock sequences was simulated on each functional subunit of the chips. TestBench fault analysis was then simulated on the remaining untested faults. Detailed analysis was performed to understand the nature of each untested fault.

The TestBench fault analysis identified testable and untestable faults and split the untestable faults into various categories: redundant faults, untestable due to test inhibits, multi-time-frame untestable, and test generated but simulation failed.

Redundant logic (corresponding to the "redundant" faults) was analyzed and the redundancy often removed. Custom logic has a higher percentage of redundant faults than synthesized logic. The causes for the redundancies were understood and often removed.

Test inhibits hold a fixed value on a pin throughout the test generation in a particular mode. Faults untestable due to these constraints cannot be tested in the respective test mode, but must be tested in some other mode if they are to be tested at all. Such faults would often cause a system failure if they were to occur. Each fault of this type was analyzed to ensure that it was tested in some other test mode, usually employing some other type of test. TestBench provides a "global" test coverage which reflects the union of all of the tests for which fault simulation is performed, across all of the various test modes.

The term *multi-time-frame untestable* refers to faults which require a series of clock pulses to be detected. TestBench fault analysis is based on its own automatically generated single-clock sequences. This means that faults which require a series of clock pulses to detect will remain undetected by the automatic analysis program. Usually these faults were tested when user-defined clock sequences were applied.

The last category of untestable fault, representing a disagreement between the test-pattern generator and the fault simulator, is usually symptomatic of a software problem. Either the generated test was incorrect, or the fault simulation was in error. The robust simulation capabilities in TestBench were valuable in the analysis of these faults. Test patterns could be rerun using a different simulator. When the two simulators agreed, this pointed to a problem in the test-pattern generator. Another useful feature of TestBench is the capability to simulate a subunit as if the fault existed and display the waveforms showing either the good or faulty behavior of the design.

Along with identifying untestable faults, TestBench also identified faults which, while not tested, were testable. Typically one might not need to consider these faults, but since the designs supported a limited set of clock sequences, these faults had to be examined. This analysis prompted model changes to support an additional clock sequence. Also, some groups of faults were understood to be untestable with the models used, but either did not exist or were actually being tested in the hardware.

Custom design techniques which stretch or bend accepted DFT guidelines and tool capabilities are often

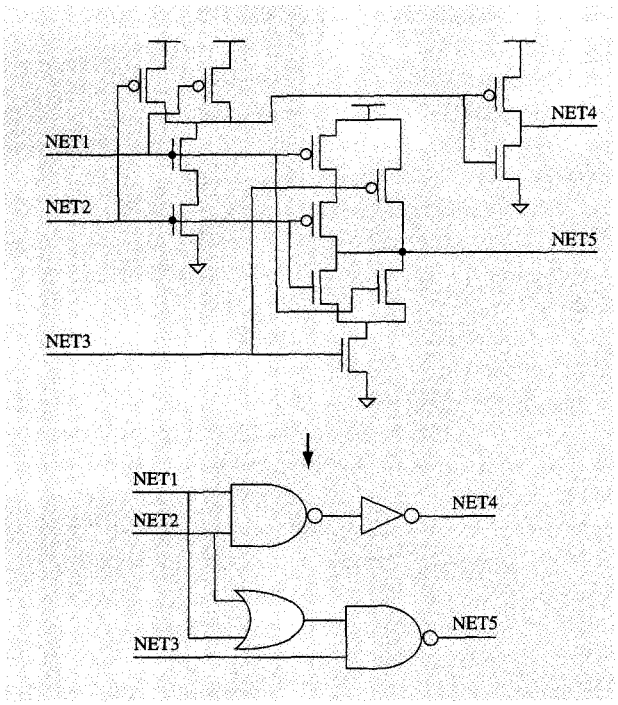

Figure 7

Example of Modgen fault-model rule generation.

necessary in today's competitive environment. A close partnership between the product designer and the tool provider is mandatory for survival, as more demands are being placed on the tools, both for additional function and for added flexibility to run existing functions with fewer constraints on the design. This has come about primarily from the natural forces of VLSI: larger circuits demanding higher tool performance and integration of more functions on a single chip, requiring techniques such as self-generated clocking and self-test.

Fault modeling

TestBench operates on a design's fault model, which is based on a gate-level representation of the circuit. In this section, we refer to this gate-level circuit model as a *fault-model rule*. For standard-cell (ASIC) designs, a TestBench-compatible fault-model rule set was developed and stored in a library. A predefined set of fault-model rules cannot be created for custom designs. Custom designs are modeled by an application in TestBench called Modgen, which automatically generates fault-model rules directly from a transistor-level schematic.

Modgen takes as input a netlist of a transistor circuit and uses a path-tracing algorithm to produce a structurally equivalent gate-level circuit. The gate-level circuit is composed of logic blocks (called logic primitives) that TestBench understands, such as AND, OR, LATCH, and TSD (tristate device). An example of this is shown in **Figure 7**.

Modgen creates the fault-model rules for combinational logic, but not for sequential logic such as latches, clock blocks, and arrays. Fault-model rules for sequential logic were created by hand. The fault-model rule-generation process is shown in **Figure 8**.

Modgen's output is in the form of EDIF (Electronic Design Interchange Format), which contains TestBench logic primitives. After the EDIF model is built, it is imported into TestBench, and a TestBench model is created. The EDIF is also used by E2V (EDIF to Verilog) to create a fault-model cell view stored in the designer's library. The fault-model cell view is compared to the schematic using Verity [36] to ensure that the fault-model rule correctly predicts the circuit behavior.

Building a fault-model rule from a hierarchical schematic consists of running Modgen with all of the schematic's instantiated cells treated as black boxes. If the fault-model rule for a given black-boxed cell has already been created, no further processing is required for the cell. If a fault-model rule has not been created for a given cell, or if the schematic for the cell has been updated

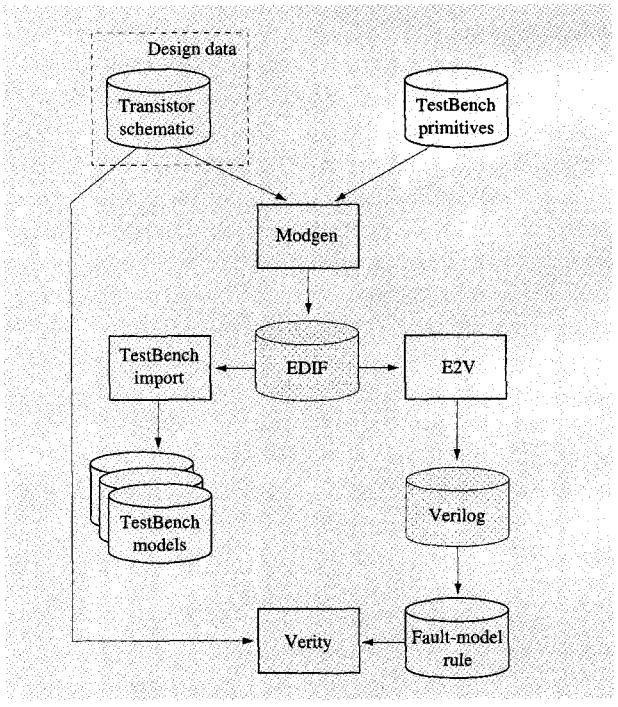

Figure 8

Flow for fault-model generation.

since the fault-model rule was created, a new fault-model rule is created. This hierarchical traversal of the design continues until updated fault-model rules have been created for all unique cells in the design.

- *Modeling sequential logic*

Creating fault-model rules for sequential logic is done manually, since Modgen does not handle sequential logic. The schematic must be studied and understood, and then a fault-model rule and corresponding EDIF file can be built using TestBench logic primitives.

One example of modeling sequential logic is the clock block that is used throughout the chip to provide local C2/C1 clocks to latches. Its fault-model rule is shown in **Figure 9(a)**.

TestBench treats a circuit using this clock block as an unconstrained sequential design, because the latches that are used in the clock generation confuse the TestBench analysis programs, which look for simple means of controlling all of the system latch clocks and the scan data path. To allow the TestBench highly efficient stored-pattern, LBIST, and WRP support to process the chips, the clock-block model was simplified. The challenge

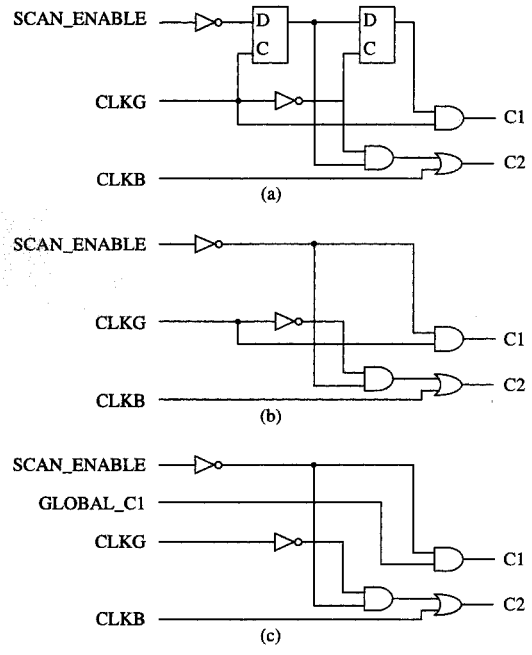

Figure 9

Clock-block fault-model rules: (a) True clock block; (b) first modification to remove nonscannable latches; (c) final fault-model rule for clock block.

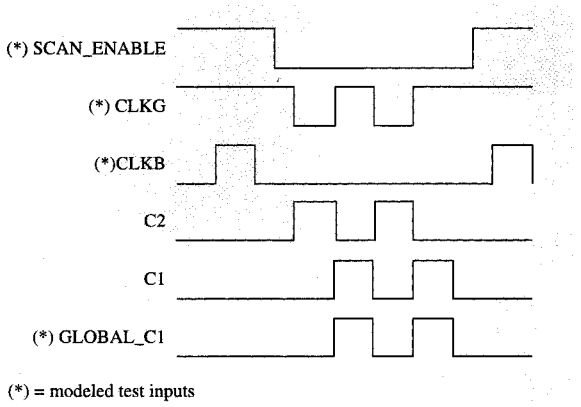

Figure 10

Clock sequence for clock-block fault-model rule.

was to simplify the model while ensuring that the tests generated with this simplified model would work on the hardware.

The first change to the model is to remove the nonscannable latches, as shown in **Figure 9(b)**. There is still a problem with this model, because TestBench requires the existence of a primary input *stability state* defined by setting all test inhibits (constant-value inputs) and all clock primary inputs to their inactive states. Even for an edge-triggered design, the clock primary inputs must have a defined stability state. It is a requirement that the stability state must set all clock inputs to all latches to known values. In this case, there was no way to define the clock stability state so that just setting the clock primary inputs (there are no test inhibits in this picture) would force the derived clock signal to a known value. This problem was solved by adding to the model an extra pin called GLOBAL_C1. **Figure 9(c)** shows the new model with the extra "pseudo" primary input.

Adding the pseudo pin to the model has several ramifications. Since the pseudo pin does not exist in the hardware, TestBench cannot be allowed to control it in a random manner; instead, a user-specified clock sequence must be used so that the model will behave like the hardware. **Figure 10** shows the user-specified clock sequence that is used. Note that signals C2 and C1 behave identically in Figures 9(a) and 9(b) using the user-specified clock sequence, so the model will behave like the hardware if the clock sequence used in Figure 10 is used during test-pattern simulation.

Another ramification of using a pseudo pin is that the test patterns must be changed prior to applying them at the tester. In addition, using pseudo pins increases the risk

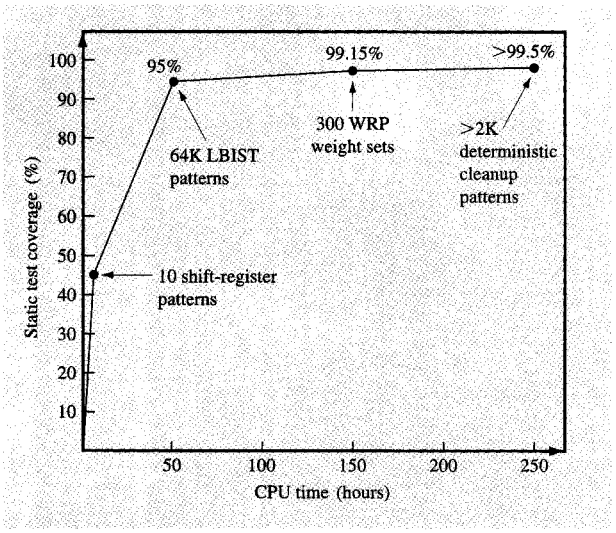

Figure 11

Test coverage vs. CPU time.

of modeling the behavior of the circuit incorrectly, so extensive model verification must be done.

- *Model verification*

In order to confirm that the TestBench model correctly predicts the behavior of the chip, the model must be compared to the circuit for a variety of patterns. Verification was done at different levels of hierarchy in the design: leaf-cell-, macro-, and chip-level verification.

Leaf-cell verification

A leaf cell is defined as any cell that contains transistors. For each leaf cell in the design containing combinational logic, the fault-model rule was compared to the schematic using Verity. Verity performs exhaustive verification—that is, the circuit is verified for all possible input stimuli. Verity cannot be used to verify sequential logic, so verification was also done at the macro level and chip level, where another verification method was used.

Macro-level verification

A macro is a functional logic group that can contain combinational and sequential logic. In order to verify the fault-model rule used for a macro, TestBench was run on the macro to create a set of patterns. These patterns were then simulated on the VHDL model of the macro, and the output of TestBench was compared to the output of the VHDL. This process proved to be quite valuable. Since it is time-consuming to create a set of patterns to use for verification, using TestBench to create the patterns saved considerable time. Also, TestBench created patterns that a designer might have overlooked.

Chip-level verification

For the chip-level verification, LBIST and ABIST were run using TestBench and also using the chip simulation model [37], and the signatures were compared. This level of verification disclosed problems existing between macros that were not caught by the macro-level verification. It also found problems in the behavior of TestBench, and VHDL problems in the simulation model. This extensive model verification resulted in easing debugging of the test patterns at the tester, where no model or tool problems were found.

Test-pattern generation and coverage

Many test techniques have been discussed. They all come together in the final test-data generation for the product. The goal in test generation is to maximize test coverage as quickly and as efficiently as possible. With limited tester buffer sizes, test-data volume is critical. Also, the total CPU time to generate the test patterns had to be kept to a reasonable length, since pattern regeneration occurred often because of code bugs, model updates, last-minute logic changes, and efforts to optimize the pattern set.

The approach used was to target static stuck-faults first and then resimulate the pattern set with dynamic fault simulation turned on. This was done because dynamic fault simulation for the CP chip required more than two CPU weeks on an RS/6000* Model 590 with two gigabytes of real memory (the number of flat model blocks is 1.3 million, and the combined number of static and dynamic faults is 9.2 million), and static test-pattern generation required several iterations. To speed dynamic test-coverage growth when targeting the static stuck-faults, the dynamic-type clock sequences were used first. Then the remaining static-only-type clock sequences were used to complete the test generation. During dynamic resimulation of the static patterns, this approach enabled quicker dynamic test-coverage gain, which ultimately improved the test coverage. Note that even though targeting was done in a static manner, these patterns were executed on the product in a dynamic mode.

Figure 11 shows an overview of the test patterns that were created, the time required to create them, and the number of each type that were created.

The first test generated is the shift-register test, which detects about 45% of the static stuck-faults (static stuck-faults total about 4.5 million). This runs quickly, in about ten CPU minutes, and generates only ten patterns.

Next, 256 000 LBIST patterns were generated, but only the first 64 000 were fault-simulated. Fault simulation of 64 000 patterns required 50 hours of CPU time, but 256 000 would have required about 200 hours of CPU

time. The test-coverage number reflects only the first 64 000 patterns; however, in order to get the benefit of the patterns, the full 256 000 patterns were applied at the tester. Since LBIST is inexpensive from a tester buffer and tester run-time perspective, the number of patterns was extended to detect unmodeled defects as well. Unmodeled defects are defects which can occur in hardware that are not modeled in the fault-model rule for a circuit. In addition, it is beneficial to have a large number of LBIST patterns, since LBIST is the only logic test that is run on the higher levels of system testing.

With the *easy-to-detect* faults out of the way, WRP generation was invoked next. WRP generates weight sets, and patterns are simulated with these weight sets until a marginal test-coverage threshold is reached. The weight sets target specific faults that are typically random-pattern-resistant and are difficult to detect using LBIST patterns. As shown in Figure 11, creating WRP patterns is CPU-intensive, and 300 weight sets required 103 CPU hours. When the effectiveness of WRP declined substantially, the remaining faults were tested with deterministic patterns.

The deterministic patterns were created in the WRP test mode, since the scan-chain length in WRP mode is 15 times shorter than the full scan-chain length of the chip. This allowed 15 times more deterministic patterns to fit in the tester buffer. Because of the large number of deterministic patterns required to achieve a high test coverage, this method greatly reduced the tester buffer and test time.

Once the bulk of static faults were detected, the other tests were created to test the I/Os, the PLL, and the arrays (using ABIST).

After the static tests were created, the patterns were resimulated with dynamic fault simulation turned on. Then the remaining untested dynamic faults were targeted with WRP and deterministic patterns, and the new patterns were appended to the total pattern set. Greater than 90% dynamic test coverage was achieved.

Conclusions

The S/390 custom microprocessor design created many test challenges. The test methodology required several enhancements to the test-generation process and specialized fault-model development because of the complexity and unique clocking structure of the design. BIST required unique design development to support the complexity and high-performance aspects of the design, while SGC was instrumental in verifying performance and allowed the chips to be tested at cycle time during the manufacturing test process.

As we look to the future, several key test issues must be addressed: Dynamic test must be enhanced to keep up with increasing device performance; dynamic faults must be targeted and accurately measured, while on-chip techniques must be developed to provide more detailed analysis of device performance. BIST must be optimized to detect dynamic faults.

Test-generation techniques must be developed to reduce test-generation time and test-application time in an effort to drive down test costs. Larger and more complex designs will exceed the capability of today's test-generation tools and will increase test times so that test becomes a greater portion of the overall product cost.

Test is and will always be key to future microprocessor designs. Investments in test methodology development, test generation, and advanced BIST techniques will ensure the success of future programs.

Acknowledgments

Many individuals contributed to the success of the Test team: Larry Lange for writing several test tools and enhancements; Tom Foote for test generation on the L2 design; Pradip Patel and Phil Shephard for help in the ABIST design; Rick Rizzolo for sorting and test-characterization specification for manufacturing; Stelio Tsapepas for his interfacing to manufacturing; and Jim Allen, Jordy Asher, Carolyn Asher, Paul Chang, Bob Gongleski, Brion Keller, Rod MacLean, Kevin McCauley, Ed Orosz, Leon Palmer, and Jon Tischer for their ability to respond with quick updates on the TestBench programs.

Also, several teams were involved in the testing and characterization of the chips: Dave Heidel, Mike Immediato, Keith Jenkins, Kevin Stawiasz, and Steve Wilson for their efforts on characterization; Barry Butkus, Bill St. George, Mark Olive, Dana Santerre, Pierre Thivierge, Darren Childress, Owen Farnsworth, Deborah Hamm, and Ed Leahy for their efforts on pattern generation and hardware bring-up in manufacturing; and Micrus Corporation personnel Franco Motika, Donato Forlenza, Orazio Forlenza, Ray Kurtulick, Joe Sheridan, Wendy Chong, and Adrian Anderson for hardware bring-up and characterization at the tester.

References

1. D. R. Schertz and G. Metze, "A New Representation for Faults in Combinational Digital Circuits," *IEEE Trans. Computers* **C-21,** 858–866 (August 1972).
2. E. B. Eichelberger, "Method of Level Sensitive Testing a Functional Logic System," U.S. Patent 3,761,695, September 25, 1973.
3. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 14th Design Automation Conference,* New Orleans, 1977, pp. 462–468.
4. L. A. Stolte and N. C. Berglund, "Design for Testability of the IBM System/38," *Proceedings of the IEEE International Test Conference*, 1979, pp. 29–36.

5. S. DasGupta, P. Goel, and T. W. Williams, "Level-Sensitive Scan Design System," U.S. Patent 4,293,919, October 6, 1981.
6. P. H. Bardell and W. H. McAnney, "Self-Testing of Multichip Modules," *Proceedings of the IEEE International Test Conference*, 1982, pp. 200–204.
7. John Reilly, Arthur Sutton, Robert Nasser, and Robert Griscom, "Processor Controller for the IBM 3081," *IBM J. Res. Develop.* **26,** No. 1, 22–29 (January 1982).
8. I. M. Ratiu and H. B. Bakoglu, "Pseudorandom Built-In Self-Test Methodology and Implementation for the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** No. 1, 78–84 (January 1990).
9. Cordt W. Starke, "Design for Testability and Diagnosis in a VLSI CMOS System/370 Processor," *IBM J. Res. Develop.* **34,** No. 2/3, 355–362 (March/May 1990).
10. S. F. Oakland, J. L. Corr, J. D. Blair, R. D. Norman, and W. J. DeGuise, "Self-Testing the 16-Mbps Adapter Chip for the IBM Token-Ring Local Area Network," *IBM J. Res. Develop.* **34,** No. 2/3, 416–427 (March/May 1990).
11. R. W. Bassett, P. S. Gillis, and J. J. Shushereba, "High-Density CMOS Multichip-Module Testing and Diagnosis," *Proceedings of the IEEE International Test Conference*, 1991, pp. 530–539.
12. B. Konemann, J. Barlow, P. Chang, R. Gabrielson, C. Goertz, B. Keller, K. McCauley, J. Tischer, V. Iyenger, B. Rosen, and T. Williams, "Delay Test: The Next Frontier for LSSD Test Systems," *Proceedings of the IEEE International Test Conference*, 1992, pp. 578–587.
13. P. Gillis, T. Guzowski, B. Keller, and R. Kerr, "Test Methodologies and Design Automation for IBM ASICs," *IBM J. Res. Develop.* **40,** No. 4, 461–474 (July 1996).
14. P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. Computers* **C-30,** 215–222 (March 1981).
15. F. Motika and J. Waicukauski, "Weighted Random Pattern Testing Apparatus and Method," U.S. Patent 4,688,233, August 18, 1987.
16. J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger, and O. P. Forlenza, "A Method for Generating Weighted Random Test Patterns," *IBM J. Res. Develop.* **33,** No. 2, 149–161 (March 1989).
17. M. Schulz, E. Trishler, and T. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *Proceedings of the International Test Conference*, 1987, pp. 1016–1026.
18. H. K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test Generation for Sequential Circuits," *IEEE Trans. Computer-Aided Design* **7,** 1081–1093 (October 1988).
19. B. L. Keller and T. J. Snethen, "Built-In Self-Test Support in the IBM Engineering Design System," *IBM J. Res. Develop.* **34,** No. 2/3, 406–415 (March/May 1990).
20. P. Goel, "Test Generation Costs Analysis and Projections," *Proceedings of the Design Automation Conference*, June 1980, pp. 77–84.
21. T. W. Williams and K. P. Parker, "Design for Testability—A Survey," *Proc. IEEE* **31,** No. 1, 18–22 (January 1983).
22. J. J. LeBlanc, "LOCST: A Built-In Self-Test Technique," *IEEE Design and Test*, pp. 45–52 (November 1984).
23. P. H. Bardell and W. H. McAnney, "Self-Test of Random Access Memories," *Proceedings of the IEEE International Test Conference*, 1985, pp. 352–355.
24. R. W. Bassett, M. E. Turner, J. H. Panner, P. S. Gillis, S. F. Oakland, and D. W. Stout, "Boundary-Scan Design Principles for Efficient LSSD ASIC Testing," *IBM J. Res. Develop.* **34,** No. 2/3, 339–354 (1990).
25. P. H. Bardell, W. H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, John Wiley, New York, 1987.
26. B. Koenemann, J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Technique," *Proceedings of the IEEE International Test Conference*, October 1979, pp. 37–41.
27. H. Drukerman, M. Kusko, S. Pateras, and P. Shephard, "Cost Trade-Offs of Various Design-For-Test Techniques," *Proceedings of the Third International Workshop on the Economics of Design, Test and Manufacturing*, IEEE Computer Society, Austin, TX, May 1994, pp. 45–50.
28. *IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1-1990, IEEE Standards Board, 345 East 47th Street, New York, NY 10017, 1990.
29. P. J. Camporese, P. J. Meaney, B. J. O'Leary, and R. R. Rizzolo, "Programmable Clock Tuning System and Method," U.S. Patent 5,455,931, October 3, 1995.
30. A. M. Cady and G. A. Hughes, "Off-Chip Module Clock Controller," *IBM Tech. Disclosure Bull.* **32,** No. 4A, 77–78 (September 1989).
31. M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, New York, pp. 421–452.
32. P. G. Shephard III, W. Huott, P. R. Turgeon, R. W. Berry, Jr., P. Patel, G. Yasar, J. Hanley, and F. J. Cox, "Programmable Built-In Self Test Method and Controller for Arrays," U.S. Patent 5,633,877, May 1996.
33. W. Huott, T. J. Slegel, T. Lo, and P. Patel, "Programmable Computer System Element with Built-In Self Test Method and Apparatus for Repair During Power-On," U.S. Patent 5,659,551, May 1966.
34. W. Huott and T. McNamara, "Programmable Delay Clock Chopper/Stretcher With Fast Recovery," U.S. Patent 5,420,467, February 2, 1996.
35. K. L. Shepard, S. M. Carey, E. K. Cho, B. W. Curran, R. F. Hatch, D. E. Hoffman, S. A. McCabe, G. A. Northrop, and R. Seigler, "Design Methodology for the S/390 Parallel Enterprise Server G4 Microprocessors," *IBM J. Res. Develop.* **41,** No. 4/5, 515–547 (July/September 1997, this issue).
36. A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, "Verity—A Formal Verification Program for Custom CMOS Circuits," *IBM J. Res. Develop.* **39,** No. 1/2, 149–165 (January/March 1995).
37. B. Wile, M. P. Mullen, C. Hansen, D. G. Bair, K. M. Lasko, P. J. Duffy, E. J. Kaminski, Jr., T. E. Gilbert, S. M. Licker, R. G. Sheldon, W. D. Wollyung, W. J. Lewis, and R. J. Adkins, "Functional Verification of the S/390 Parallel Enterprise Server G4 System," *IBM J. Res. Develop.* **41,** No. 4/5, 549–566 (July/September 1997, this issue).

Received December 4, 1996; accepted for publication July 21, 1997

William V. Huott *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (ace@vnet.ibm.com).* Mr. Huott is an Advisory Engineer in S/390 custom microprocessor design. He received a B.S. in electrical engineering from Syracuse University in 1984. Mr. Huott is a design engineer responsible for the definition and design of the memory array built-in self-test (ABIST) circuitry and interfaces for high-frequency custom microprocessors. He is also responsible for the engineering test debug of high-frequency custom macros and support chips. Mr. Huott joined IBM in 1983 at the East Fishkill facility designing, testing, and debugging high-speed logic gate arrays and memory arrays. He holds four U.S. patents and several publications.

Timothy J. Koprowski *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (tkoprowski@vnet. ibm.com).* Mr. Koprowski is an Advisory Engineer in S/390 custom microprocessor design. He received his B.S. in electrical engineering from Lehigh University in 1981 and is currently pursuing an M.S. degree in computer engineering at National Technological University. He is a design engineer responsible for the definition and design of the LBIST circuitry for high-frequency custom microprocessors. Mr. Koprowski joined IBM in 1981 at the East Fishkill facility, where he designed advanced VLSI logic and memory test systems for eleven years. He holds one U.S. patent.

Bryan J. Robbins *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (brobbins@vnet. ibm.com).* Mr. Robbins received the B.S. degree in electrical engineering from Purdue University in 1986, and the M.S. degree in computer and electrical engineering from Purdue University in 1988. He is an Advisory Engineer responsible for test modeling and test generation for high-frequency custom microprocessors. Mr. Robbins joined IBM in 1988 at Poughkeepsie, New York.

Mary P. Kusko *IBM Microelectronics Division, East Fishkill facility, Route 52, Hopewell Junction, New York 12533 (mkusko@vnet.ibm.com).* Ms. Kusko is an Advisory Engineer in EDA (electronic design automation). She received a B.S. in electrical engineering from the University of Delaware in 1982, joining the IBM S/390 Division that same year. From 1982 to 1993 she held several positions in processor design. Since 1993, she has worked in the areas of test strategy definition and implementation, model generation, test analysis, test generation, and internal test tool direction. Ms. Kusko is a member of the IEEE, SWE, and TTTC.

Stephen V. Pateras *LogicVision Inc., 101 Metro Drive, Third Floor, San Jose, California 95110 (pateras@lvision.com).* Dr. Pateras received his Ph.D. degree in electrical engineering from McGill University in 1991. While at IBM, he served as test team leader in the S/390 custom microprocessor group. He is now Director of System BIST Product Engineering at LogicVision. Dr. Pateras is a member of the IEEE and has authored or coauthored several papers and articles in the fields of test generation and BIST.

Dale E. Hoffman *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (daleh@vnet.ibm.com).* Mr. Hoffman received a B.S. in electrical engineering from Pennsylvania State University in 1981, and an M.S. in electrical engineering from Syracuse University in 1984; he is currently pursuing an M.S. in computer engineering at National Technological University. He is a Senior Engineer and a design manager responsible for test, chip integration, physical design, and back-end design methodology for high-frequency custom microprocessors. Mr. Hoffman joined IBM in 1981 at the East Fishkill facility, where he designed and managed advanced VLSI logic and memory test systems for eleven years. He holds five U.S. patents and has several publications.

Timothy G. McNamara *IBM System/390 Division, 522 South Road, Poughkeepsie, New York 12601 (tmcnamera@vnet. ibm.com).* Mr. McNamara is an Advisory Engineer in S/390 custom microprocessor design. He received his B.E. in electrical engineering from the State University of New York at Stony Brook in 1983 and his M.S. in computer engineering from Syracuse University in 1990. He is currently working on high-performance clock system designs for S/390 CMOS microprocessors. Mr. McNamara holds two U.S. patents.

Tom J. Snethen *IBM Microelectronics Division, 1701 North Street, Endicott, New York 13760 (snethent@endvm5.vnet. ibm.com).* Mr. Snethen is a Senior Engineer in the IBM Test Design Automation project. He received the B.S.E.E. degree from the University of Missouri and the M.S.E.E. from MIT before joining IBM in 1965. Mr. Snethen is a member of the IEEE. He holds one patent and has authored or coauthored several papers in the field of fault modeling and test generation.