

 **TANDEM** COMPUTERS

Design Methodology for System Correctness: Lessons From The Tandem NonStop CLX

Peter L. Fu

Technical Report 87.7
November 1987
Part Number 11642

**Design Methodology For System Correctness:
Lessons From The Tandem NonStop CLX**

Peter L. Fu

Technical Report 87.7

November 1987

Part Number 11642

Table of Contents

Abstract	1
Introduction	1
System Design Correctness	1
Areas of Correctness	1
Correctness Responsibilities.	1
Common Design Errors	2
Design Methodology for the CLX CPU	2
Conceptual Design	2
Logic Design	2
Microcode Design	2
Integration	3
Two Levels of Modeling	3
Verification	3
Design Commitment	3
Hardware Debugging	3
Production Tests	4
A Fault-Tolerant Design Methodology.	4
Graceful Degradation.	4
Design for Testability	4
High Fault Coverage Self-Tests	4
Modularity	4
Fault Avoidance	4
Fault Masking	4
Fault Secureness	5
Self-checking Design	5

DESIGN METHODOLOGY FOR SYSTEM CORRECTNESS: LESSONS FROM THE TANDEM NONSTOP CLX

Peter L. Fu

Tandem Computers Incorporated
19333 Vallco Parkway
Cupertino, CA 95014

Abstract

This paper presents a methodology which allows a small design group to quickly produce a highly integrated processor. The methodology presented here is based on the design experience of the CPU for the Tandem NonStop CLX. The first section focuses on system design correctness, who shares the correctness responsibilities and some of the likely design errors. The second section gives an account of the design and verification process of the NonStop CLX CPU. The last section draws lessons from this process and introduces parallels between the techniques for implementing highly reliable or fault-tolerant system and the design methodology for system correctness.

Introduction

Most of today's system projects, though modest in size and scope, use aggressive circuit technology for performance and competitiveness. To be successful, these projects must seek an optimal investment in time, people and design tool resources. There is a need for system design environments which enable a small number of system designers to define, implement, verify and deliver such a product on time. Most of today's silicon compilation, standard cell and gate-array vendors are striving to provide such an environment.

The methodology presented here is based on the design experience of the Tandem NonStop CLX™ CPU [1]. It grew out of planning, on-the-job decisions and lessons learned in retrospect. Examples, taken from the design and verification process of the CLX CPU, are used for illustration. Although the processor was designed using silicon compilation, the methodology applies equally well to other forms of ASIC designs.

System Design Correctness

This section of the paper focuses first on the particulars of what design correctness encompasses. This is followed by a discussion on whom design correctness responsibilities fall, in the context of a layered technology partnership between vendors or groups within a company. Finally, a discussion on common design errors concludes this section.

Areas of Correctness

The designer should be aware of the following areas of correctness:

- **Feature correctness and completeness.** It is obvious that in a fully functional design, each feature must be implemented and perform as specified.
- **Correctness should also be viewed in context.** External interface correctness is as important as internal design correctness.
- **Cost, Performance and Flexibility.** A design is not correct if it costs too much to manufacture or does not meet performance or capacity requirements. Beyond mere meeting specifications to the letter, a good design should also consider some flexibilities in its implementation. Such flexibility might be needed later on in the project to make up for some lost performance in other areas.
- **A design is not correct if it exceeds specified physical constraints** such as size, weight and power consumption.
- **A design is also not correct if the intended testability coverage is not implemented.** This is important for manufacturability and system maintenance.
- **Data Integrity Coverage.** System designer must consider the appropriate amount of protection from component failures as a correctness issue, since error detection and recovery techniques are difficult to be retrofitted into systems.

Correctness Responsibilities

Any state-of-the-art system project requires expertise from multiple disciplines. From project management and efficiency points of view, it is desirable to partition project responsibilities according to these disciplines, where there can be well-defined interfaces and accountable shares of design correctness responsibilities. A multi-vendor partnership may involve a system manufacturer doing the system design, a tools vendor providing silicon compilation or other ASIC tools and a semiconductor foundry providing fabrication and testing services. An inter-group partnership may consist of a design group, a CAD group and the fab-line of the same company.

System designers must have a full understanding of the system requirements and the technology base. The full flexibility and limitations of the tools and technology capabilities must be taken into account in assessing design implementation

and design process options. Appropriate aggressiveness and conservativeness need to be weighed against system correctness in the final product.

Technology/CAD tools suppliers must fully understand and be able to exploit the capabilities of the underlying circuit technology. The overall responsibility to the system designer is to present a level of abstraction, which facilitates the system design process. To maintain design correctness, all tools must be carefully verified. To *enhance* design correctness, all tools should be made extensible where appropriate, especially simulation tools. Tool development must stay ahead of intended usage. Size and other limits in the tools should be aggressively tested in advance of usage.

Semiconductor foundries or fab-line services form the third partner in creating correct systems. Besides maintaining correctness internally in their manufacturing process, they must ensure correctness in their interfaces with the CAD tools. Further responsibilities include correlating the performance of their process with the tools supplier so that the latter can calibrate their performance prediction tools for the system designers.

Common Design Errors

Given the capabilities of the current CAD tools and fab-line automation, many of the more obvious design errors such as inconsistent netlists, electrical and physical design rule violations are automatically screened out. Aside from errors within CAD tools or the fabrication process itself, imperfect designs tend to be due to errors such as:

- mis-communicated specifications
- interface/protocol mismatch
- incomplete or erroneous error/exception handling
- mishandling of boundary conditions for some operations or algorithms
- missing feature
- incorrect initialization or unresettable state
- no source on bus under certain conditions
- unintended and undesirable side effects of some functions
- missed testability coverage

Design Methodology for the CLX CPU

System design methodologies to ensure correctly designed products are evolving. Not long ago, the standard procedure was to first produce a careful paper design, reducing concepts to good practical implementations. Prototypes were then built, debugged, revised and debugged again. The advent of VLSI made this obsolete. With its inherent complexities and high cost of making revisions to a design, it is paramount that the confidence in design correctness be high before commitment to silicon.

The current approaches span a spectrum ranging from full system modeling and simulation [2][3], to hierarchically structured design validation and correct-by-construction techniques [4], to proof of hardware design correctness [5]. But for most projects with few designers, using highly leveraged tools to manage complexity, the former two approaches may be

too costly, in terms of time and resources. On the other hand, the proof of design correctness tools are not yet available commercially. The methodology used at Tandem for the design of the NonStop CLX CPU is an example of utilizing available tools to the fullest, under the constraints of time, human and machine resources. This methodology is based on the GENESIL[®] design system from Silicon Compilers Systems Corporation. In addition, complementary tools for design verification, firmware development, lab debug and chip test support were developed by Tandem.

Conceptual Design

The design goal of the NonStop CLX was to build a fault-tolerant, high performance and low-cost minicomputer compatible with the existing line of Tandem NonStop computers. The investigation of the micro-architecture and the target technology was an interactive process, started in early 1985. Various ASIC alternatives were evaluated, with key sections of logic serving as test design vehicles. It was decided that silicon compilation based on GENESIL provided the best leverage for integration, performance and design efficiency from a rich set of high-level functional blocks. Much effort was applied to understand the capabilities and limitations of the tools before logic design was begun. The result was an architecture that matches well with the technology, thus enhancing the probability of a correct implementation.

Logic Design

A careful top-down paper design was made, resulting in a detailed block diagram of the functions of a chip-set, the major blocks within, and all the major buses. Since there were few designers — three for the most complex of the four chips — communication problems on design interfaces were minimal. Actual logic implementation and data entry proceeded bottom-up.

Micro-code Design

When the micro-architecture crystallized, the micro-word field definitions were documented. Key micro-instruction sequences were written to verify the capabilities of the micro-engine to support the functionality and performance of the macro-architecture. Firmware designers then customized a universal macro-assembler to generate CLX micro-code. In order to manage the inter-field dependencies, parameterized templates were used to ensure consistency. The multi-stage pipeline also involved many multi-line interactions, and a post-processing constraint checker was built to monitor possible conflicts between sequentially executed lines of micro-code. This was possible with the help of a network builder program, which finds all paths of execution the micro-code could take.

Special hardware features deserved special attention. The micro-ROM within the CPU chip cannot be changed once the design was committed to fab. Besides having a need to be correct (and complete), it needed to avoid hard-coded references to relocatable sections of code or data. Also, as an optimizing feature, the first lines of each macro-instruction were to reside in a skipping region (see [1]) and do not sequence like normal micro-instructions. Instead of fragmenting the code, source reorganization was chosen as a postprocessing step, relieving

firmware designers from this complexity. Overall, automation was used wherever possible to enhance code correctness.

Integration

The integration of sub-blocks into chips happened rather rapidly, partly due to many of the high-level functional blocks available from the GENESIL. A tradeoff was made in the areas of control and decoding logic, which would take more time to optimize. Many of these design details such as logic partitioning and state assignment problems (for size and timing), are still best handled by human designers. Rather than waiting for time consuming optimizations, preliminary functional designs were made such that the whole system could be modeled to obtain global functionality, timing analysis and chip sizing feedback as soon as possible.

In retrospect, this was instrumental for functional simulation, but these early estimates for size and timing were too optimistic.

Two Levels of Modeling

There were two simulation efforts, one for hardware and the other for firmware development. The hardware simulation modeled as much of the processor as possible at the functional level with logic strengths and don't-cares. This included full functional models of all custom chips, full control store, address and data caches, full-sized main memory and interface logic, which were all done on the GENESIL functional simulator. Models for the chips were automatically generated when the designs were entered into the system. Off-chip TTL interface logic was also constructed within GENESIL, as if they were other chips.

This modeling effort was notably streamlined by the full programming language and underlying operating system interface support in the very extensible GENESIL simulator. For example, the full 16 MBytes address space of main memory with ECC and nibble-wide don't-cares was modeled without too much effort (see Appendix A).

The firmware simulation model was extracted from the hardware structure and was kept up-to-date as the design changed. The firmware simulator used behavioral models of hardware blocks, optimized for speed. Its primary purpose was for low-level diagnostic program development and verification. The firmware simulator ran on a Tandem system and was more than 100 times faster than the GENESIL simulator. Given that the firmware model tracked that of the hardware, a translation tool was built such that the extraction and updating could be automated and errors minimized. This was possible because the GENESIL design information was accessible and presented in a well-defined and parsable table format. This eliminated the problem of the two groups verifying their design to an inconsistent model.

Verification

Extensive simulation is key to design verification. Limited interactive simulation of individual functional areas was done to flush out local design errors. Exhaustive testing was used for some critical areas. For example, all 64K possible macro-instructions were simulated to check the address

formation logic against a behavioral model. The most thorough verification were achieved when the whole or at least a major part of the system was simulated using the hardware model. The logic was then simulated in context, revealing a number of interface problems. Micro-coded diagnostics were written to test the low-level functionality of specific portions of the chips. These tests were first checked out using the firmware simulator. Since these were self-contained micro-programs, they were loaded directly into the full control store model, and were executed by the simulated processor.

Once instruction-set micro-code was loaded into the simulated control store, macro-code could be directly simulated. Simple programs or segments of compiled object code were used as macro-instruction based tests. The program and data segments were loaded into either the cache or main memory models. At this point, the ability to generate meaningful tests that simultaneously stressed many area of the design was greatly improved. For example, the many cache operations and memory interactions, such as cache fill, write-through, address translation and cache conflicts, were exercised. Throughout each stage of the verification process, and especially when many memory elements were involved, the use of the don't-care or indeterminate state proved to be extremely helpful. Many errors, from reset or initialization problems, to unintentional operations, to subtle non-deterministic behavior showed up as indeterminate outputs. Still more design errors were caught early because they accessed uninitialized locations in control store, cache or memory.

Design Commitment

Aside from functional verification, other aspects of design correctness needed attention before committing chips to silicon. Much time was spent on optimizing the speed performance of the CPU chip. GENESIL provided a static timing analyzer, as opposed to timing simulation. Indeed, worst-case timing under all logic conditions, not just the timing under normal operations, was required due to pseudo-random testing.

The NonStop CLX includes many fault-detection mechanisms such as parity, ECC, cross-coupled checking, and testability mechanisms such as non-destructive freeze and scan, single-stepping, built-in pseudo-random self-test support [6]. All of these were tested before design commitment. Freeze and scan were especially critical for hardware debugging (see below). Hence, in addition to free-running mode, most tests were executed in freeze-step mode and some were executed in freeze-scan-step mode. Routines written in the GENESIL simulation environment simulated this debugging environment.

The length of the regression test suite totaled over 250,000 clocks. Before the design commitment of the second revision of the CPU chip, this regression test was done in one and a half weeks.

Hardware Debugging

The CLX CPU contains a maintenance microprocessor which can interrogate and set all register and memory elements through the freeze, scan and single-step mechanisms. When all the components were realized in hardware, the exact same tests created earlier were loaded and used for initial system bring-up.

Without exception, all tests ran exactly as simulated. All four chips were functional on first pass, but there were indeed minor design errors that escaped simulation, forcing small changes in micro-code. Only the CPU chip was revised, primarily for speed improvements.

Production Tests

The system simulation model was leveraged to generate test vectors for production chip tests. Micro-coded diagnostics and segments of macro-instruction tests were executed on the simulator and signals at the chip boundaries were captured to use as test vectors. Even hand-generated and specialized scan-related test vectors were simulated and recaptured, giving them full chip pin coverage on every vector.

Finally, the GENESIL simulator's list-processing capability (a la LISP) even allowed partial fault-coverage analysis. A toggle analysis of most simulator visible nodes ran only a few times slower than normal (see Appendix B). After the chips were released, idle time on the GENESIL CAD system was used to run stuck-at-fault simulation on the same nodes. It took a few months, but the results provided the first basis for improving fault coverage of the test vectors (see Appendix C).

A "Fault-Tolerant" Design Methodology

It is every designer's ideal to achieve perfection, to do it right the first time and on time. But given the time pressure of the commercial competitive environment, it is practically impossible to be 100% thorough in the design verification process. The task of building "perfect" systems is likened to the reliability of computer systems. Each single Tandem processor is designed to be no less failure-prone than other vendors' processors, being built with similar components. But because of two key features of Tandem NonStop systems — data integrity and fault-tolerance — the probability of a Tandem NonStop system being unavailable or delivering an erroneous result is substantially reduced. Likewise, providing some degree of "fault-tolerance" in the design process points to solutions to deal with real-life design for correctness problems, just as a fault-tolerant computer deals with real life component failures.

It is perhaps wiser to address the system design correctness problem in terms of a cost measure of design errors in each feature. A priority for verification can be established based on this cost. This error cost is often not binary — complete success versus catastrophic failure. In many cases, especially for computer systems, there are alternate means or workarounds to do a certain function, possibly causing some performance or cost penalty, but without requiring the complete removal of the error.

Following are known techniques used in building highly reliable systems, and their parallels applied to a "fault-tolerant" design for correctness methodology. [7][8]

Graceful Degradation

The ability to isolate a failure, while operating the system at reduced performance, is called graceful degradation.

In the context of design correctness, wherever possible and especially for complex areas of the design, some fall-back strategy could make the difference between a non-functional system or a reduced performance system. Examples include patchable ROMs or optimized sequences that could be run non-optimized. In the NonStop CLX, most of the micro-code lines in ROM within the CPU chip can be run in the external control store, with reduced performance.

Design for Testability

Just as a physical system that is highly testable enhances manufacturing efficiency, if a designer considers how the unfinished design is to be verified, overall design time is reduced. There is often some flexibility in partitioning and structuring a design. Choosing the more intuitive alternatives could help the design debug effort, thus enhancing correctness.

High Fault Coverage Self-Tests

Periodic self-tests improve system reliability and availability, by spotting failures before they actually cause the system to shut down or run at reduced performance or reduced fault-tolerance. Similarly, in system design verification, it helps to have a set of high fault coverage regression tests, such that as the design moves closer to full implementation, the validity of earlier designed modules can be assured. In general, the sooner lower level models can be integrated into a more complete model, the sooner more leveraged tests can be written for the subsystem, in terms of fault coverage and future conversion into production system tests. For this reason, integration happened early in the design cycle of the CLX CPU.

Modularity

Modularity encourages localization of complexity, simplification of inter-module interfaces and reduction of error detection latency. The same applies to the design process.

Fault Avoidance

Fault avoidance seeks to increase system reliability by reducing the possibility of failures. If a particular fault can cause a total system failure, it deserves critical analysis. In the design for correctness realm, selective exhaustive testing should be applied in critical areas intolerant of design errors. In the CLX, macro-instructions are dispatched to starting micro-addresses with a hard-coded PLA in the CPU chip. A serious performance penalty would be incurred if certain high frequency instructions were involved in any incorrect mapping. Consequently, this PLA was exhaustively tested with all possible opcodes.

Fault Masking

Fault masking as used constructively in a fault-tolerant system is employed to prevent erroneous outputs. Conversely, fault masking is to be avoided in system verification. Care must be taken not to certify correctness of a design prematurely just because a program or diagnostic has run successfully. Consider a fault which causes a whole section of the test to be skipped. The test may not be designed with such a fault in mind, and indicates no errors found at the end. At least for the first time such a test is run, the entire execution should be followed in

detail. In many cases, it is true that such tests have coverage of faults beyond its intended fault set. Careful tracing may reveal such design flaws in logic unrelated to a particular test.

Another potentially more serious type of fault masking is unintended side-effects. If possible, as much state information in the system as practical should be checked after each diagnostic test.

Fault Secureness

Fault secure techniques used in reliable systems ensure that the outputs are correct unless an error is indicated. Error-detecting and correcting codes are typically used to implement a fault-secure system. In the context of design verification, system tests and diagnostics are most often included in a regression test suite for future retesting as the design progresses. When these are rerun, there will not be an opportunity to trace the execution in detail again, as suggested above. It would be highly desirable to strengthen these tests with error detection or watchdog mechanisms separate from the actual tests. One effective yet simple mechanism is to associate with each test a measure of reasonableness. For example, the number of simulated clock cycles as well as the final system state for each test can be captured. These can then be compared against the carefully monitored first run.

Self-checking Design

A self-checking circuit produces an error indication when there is an internal fault or inconsistency. In data or transaction processing, data integrity is just as important as availability. Many systems today try to provide both features, particularly in memory systems or communication subsystems where parity, error-detecting and correcting codes are used. Including these built-in fault detection mechanism in the modeling during system verification will improve design error sensitivity. For example, enabling the checksum mechanism in a receiver may reveal errors in a data packet early, before further error propagation, saving the time necessary to backtrack. In the CLX lab debug phase, the totally self-checked duplicate and compare cross-coupled checking scheme was very sensitive to uninitialized state in the CPU chips. Some otherwise difficult to trace operating system support micro-code problems were uncovered early.

Conclusions

As computer-aided design technology progresses, a greater portion of the task of mapping design concepts to appropriate implementations is being automated. The focus of design correctness issues are moving away from the mechanical aspects of the design process into more global and complex aspects. This paper presented the strategy taken by the Tandem NonStop CLX design team to manage these issues, resulting in the successful introduction of the system. Further analysis of this process revealed many parallels with the techniques used in fault-tolerant computing. It is hoped that lessons learned here will result in further improvements of future design methodologies for system design correctness.

References

- [1] Lenoski, D. E., "A Highly Integrated, Fault-Tolerant Minicomputer: The NonStop CLX," *Digest of Papers, Compcon Spring 1988*.
- [2] Bak, D. T. and C. Wiecek, "VAX 8800 Design Tools and Methodology," *Digest of Papers, Compcon Spring 1987*, pp. 329-332.
- [3] Ohno, Y., et al., "Principles of Design Automation System for Very Large Scale Computer Design," *The 23rd Design Automation Conference Proceedings, 1986*, pp. 354-359.
- [4] Ryan, R. J., "The CLIPPER™ CAD System Integrated Hierarchical VLSI Design," *Digest of Papers, Compcon Spring 1986*, pp. 186-190.
- [5] Barrow, H. G., "Proving the Correctness of Digital Hardware Design," *VLSI Design, July 1984*, pp. 64-77.
- [6] Garcia, D. J., "Built-In-Self-Test for the Tandem NonStop CLX Processor," *Digest of Papers, Compcon Spring 1988*.
- [7] McCluskey, E. J., "Hardware Fault Tolerance", *Digest of Papers, Compcon Spring 1985*, pp. 260-263.
- [8] Siewiorek, D. P., et al., *The Theory and Practice of Reliable System Design*, 1982.

Appendices

The following GENIE™ code segments were developed at Tandem and used for design verification of the CLX. GENIE is the programmatic interface of the GENESIL functional simulator for advanced simulation. It has C-like statements and LISP-like prefix operators or commands and list-processing support. Binary numeric values have 3 states: 0, 1 and indeterminate.

Appendix A

```
/*
** Functions to Model Arbitrary Sized RAM
** Written by: Peter Fu, Charles Spirakis.
**           Tandem Computers, Inc.
**
** These functions use a UNIX™ file to model
** a 22 bit wide RAM, each RAM location takes up
** exactly 10 bytes, as modeled by the C string:
** " 0xhhhhh\n", where h is a hex digit or 'X'
** for a nibble-wide indeterminate value.
** The file seek function allows random access.
** In most UNIX systems, locations that have not
** yet been written to are "holes" in the file
** and do not occupy unnecessary disk space.
** If read, these unwritten locations return
** ASCII NULs. Remember to remove apparently
** huge files when done!
**
** Ram_Init: initialize the details of the RAM
*/
func Ram_Init {
    set ram_width 22
    set ram_hexwidth 6
    set ram_offset 10

    /* Now open RAM file for reading and writing. */
    open ram_file "RAM_model" r+
```

```

/* Define a 22-bit indeterminate value to */
/* present on errors. */
set ram_indeterminates 0biiiii.....
set ram_status 1 /* 1 == status ok */
return @ram_status
}
/*
** Ram_Write: writes to the ram file. Accepts an
** address and a value, and deposits the value
** in the location defined by the address.
** If the address has any indeterminate bits,
** no write is performed, and RAM status is set
** to corrupted.
*/
func Ram_Write { args addr value
/* make sure the address does not contain */
/* any indeterminate bits */
/* an XOR between 2 indeterminate bits */
/* does not result in a 0 */
if (== 0 (bitxor @addr @addr)) {
seek ram_file (* @ram_offset @addr)
writeto @ram_file " 0x%s\n" (hex @value
@ram_hexwidth)
} /* else */ {
println "diskram zapped!!"
set ram_status 0
return @ram_indeterminates
}
return @value
}
/*
** Ram_Read: reads from the RAM file. Accepts an
** address. If it has no indeterminate bits,
** and the status of the RAM is good,
** and there is a value for that address, that
** value is returned. Otherwise, indeterminates
** are returned.
*/
func Ram_Read { args addr
vars tstChr retval
if (== 1 @ram_status) /* see if corrupted */ {

/* if all address bits are determinate */
if (== 0 (bitxor @addr @addr)) {

seek ram_file (* @ram_offset @addr)

/* Look for an ASCII space to indicate if */
/* there is real data here */
if (== 1 (readfrom @ram_file "%c" tstchr)) {

if (== (ord " ") (ord @tstchr)) {
/* Yes, there is data */
if (== 1 (readfrom @ram_file "%x" \
retval)) {
return @retval
}
}
}
}
return @ram_indeterminate
}
}

```

Appendix B

```

/*
** Simplified version of Activity Simulation
** Written by: Peter Fu, Tandem Computers, Inc.
**
** This core routine samples the nets in the
** node_list and updates in actv_list the
** corresponding list of triplets for each net.
** Each net has as many triplets as its bit width.

```

```

** Called once each clock, Actv_Update will
** increment one of three counters, depending on
** the value of the bit being 0, 1 or indeterminate
** The following is an associative list, mapping
** each logic strength to the three values.
*/
set actv_ref '( ("0" 0) ("1" 1) ("i" 2) ("p" 2) \
("L" 0) ("H" 1) ("I" 2) \
("l" 0) ("h" 1) ("x" 2) ("z" 2) )

func Actv_Update {
vars n nw bn bv lp la bc nc
setptr la actv_list 0
foreach n @node_list {
picknet @n
set nw (netwidth)
set nv (bin (shownetvalue) @nw)
for bn 0 (- @nw 1) {
set bv (getref @actv_ref (substr @nv @bn 1))
setptr lp la 0 @bn @bv
set nc (+ 1 (first @lp))
deleat lp
insertat lp @nc
}
++ la
}
return
}

```

Appendix C


```

/*
** Poor Man's Fault Simulation (simplified)
** Written by: Peter Fu, Tandem Computers, Inc.
*/
func FaultSim {
/* Process ALL simulator visible nets */
foreach n net {
if (StuckVec @n "L") { /* found errors */
println "Net" @n "stuck-at-0 covered."
} /* else */ {
println "Net" @n "stuck-at-0 NOT covered!"
}
if (StuckVec @n "H") { /* found errors */
println "Net" @n "stuck-at-1 covered."
} /* else */ {
println "Net" @n "stuck-at-1 NOT covered!"
}
}
}
func StuckVec { args node val
/* Force a particular net hi or lo, then run */
/* the available tests. If there is no error */
/* the net is not covered for stuck at faults */
bindnet @net @val
/* Insert code here to run test vectors */
/* until error or done */
return (geterrcnt) /* return # of errors */
}

```

™ Tandem, NonStop and CLX are trademarks of Tandem Computers Incorporated. UNIX is a trademark of AT&T Bell Laboratories. GENIE is a trademark of Silicon Compiler Systems Corporation.

© GENESIL is a registered trademark of Silicon Compiler Systems Corporation.

Distributed by
 **TANDEM COMPUTERS**
Corporate Information Center
19333 Vallco Parkway MS3-07
Cupertino, CA 95014-2599

