

# Covering your HDL chip-design bets

JIM LIPMAN, TECHNICAL EDITOR

Better quality Verilog or VHDL code results in better, more testable chip designs. Code-quality-enhancement tools help you find coding errors and questionable code constructs before synthesis. Code-coverage tools help determine your simulator's efficiency in checking all parts of a design.

The term “garbage in, garbage out” applies to HDL code, Verilog, and VHDL, just as it applies to application software. With HDL-based chip designs, problems in source code may not show up until far into the design cycle, perhaps not even until first silicon. The farther down the design path you find such problems, the more expensive it is to fix them in terms of actual engineering time and in “lost-opportunity” time for product introduction. EDA tools that inspect behavioral-level or RTL code help you find coding errors, poorly designed constructs, and potential hardware-testability problems early in your chip-design cycle.

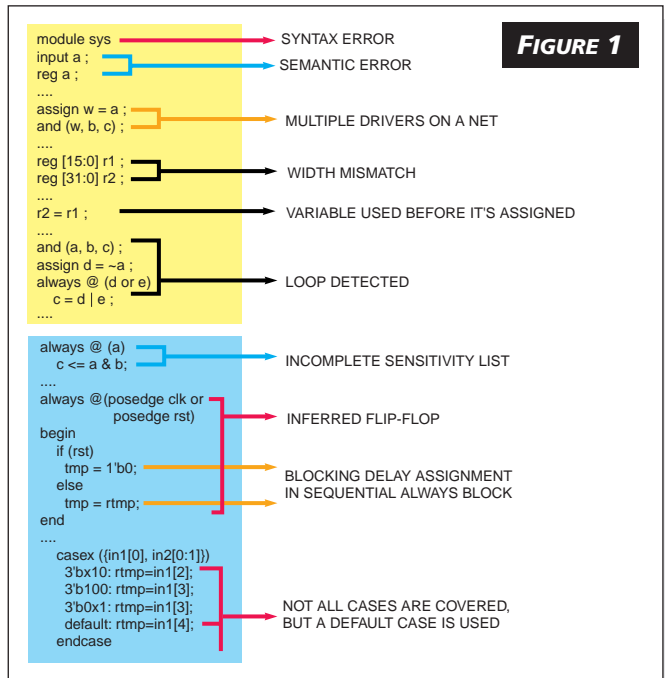
Another class of HDL-checking products, code-coverage tools, works on HDL designs before and after synthesis, during simulation. These tools check whether your simulation vectors are testing all portions of

the design with a minimum of redundancy during simulation. Code-coverage tools thus help you optimize your design testbench for high fault coverage and minimize your test-vector suite.

## Saving time and money

Unless you are an exceptionally skilled HDL-based-chip designer, your RTL code will be less than perfect. Non-ideal code can contain problems at different levels. Minor code inefficiencies may do nothing more than take extra space and run more slowly on simulation and logic-synthesis software. You may catch moderate errors during simulation or synthesis runs, but you then have to debug the code after wasting simulation and synthesis time. You may not catch serious errors until late in the chip-design cycle—sometimes not until after placement and routing. These errors largely impact engineering time and design costs; they give code-enhancement, or code-purification, tools a chance to earn their keep.

You use a code-enhancement tool early in your chip-design cycle, before any time-consuming simulations or logic synthesis. These tools inspect Verilog and VHDL RTL code, looking for syntax, semantic, synthesis-construct, and simulation-construct problems. Code-enhancement-tool checks include those contained in most logic-synthesis and simulation tools (see sidebar “In situ checking also en-



This RTL-code segment annotates examples of the kinds of Verilog-coding errors you can find by running a code-enhancement tool, such as Verilint, on RTL code.

## CODE-COVERAGE TOOLS

hances HDL code”). The benefit of code-enhancement tools is that you run these checks earlier, faster, and less expensively with these tools than with design tools that embed these checks. For example, interHDL’s Verilog-code purifier, Verilint, processes as many as 5000 lines of Verilog per second; its VHDL purifier, VHDLint, runs 4000 lines of VHDL in less than one minute.

RTL code-enhancement tools also have additional checks beyond those found in simulation and synthesis tools. **Figure 1** shows the types of errors that Verilint finds on a segment of RTL code. The errors include syntax and semantic errors as well as examples of poor design style. Verilint includes more than 600 design checks, and VHDLint has more than 300 checks.

Some code-enhancement tools go beyond syntax, semantic, and design-tool-construct checks. These tools help you create reusable designs—designs you develop with a consistent design methodology that can run on different vendors’ simulation and synthesis tools. Although many companies have HDL-coding guidelines for design portability between different design teams, a consistent set of industrywide coding guidelines would facilitate chip design with reusable soft cores coming from multiple vendors (see **sidebar** “The benefits of language independence”).

For information on a good set of general coding guidelines and techniques, check out the Reuse Methodology Manual (RMM) developed by Mentor

### @ a glance

- Code-enhancement tools spot syntax, semantic, and code-construct problems before simulation and synthesis. Some tools also check your design style.
- Code-coverage tools help you increase design testability by identifying nontested code.
- Code-coverage increases simulation time. The added time depends on code quality, coding style, the extensiveness of the coverage feature set, and the simulator interface.
- The increased use of imported soft cores and core testbenches in chip designs increases the value of using a code-coverage tool.
- By spotting problems early, code-enhancement and -coverage tools save valuable design time and resources.

Graphics ([www.mentorg.com](http://www.mentorg.com)) and Synopsys ([www.synopsys.com](http://www.synopsys.com)). Because one of the authors is from Synopsys, parts of the book dealing with coding guidelines may have some Synopsys-tool specificity. However, the book proposes a reasonable set of general-coding suggestions that can help you develop reusable HDL-based designs. Along with coding hints, the book’s authors address design flow and outline the proper use of tools and methodology to capture design information in a consistent, easy-to-read format. You’ll find an RMM description and ordering information at **Reference 1**. For consistent HDL design, EDA tools StyleCheck and Checkit help you check for good coding style.

StyleCheck, introduced by Quickturn to complement its emulation and

design-verification products, adds design-style checking for subsequent cycle-based simulation and emulation. The tool checks your RTL design for design features, such as asynchronous feedback loops (which cycle-based verification tools cannot handle), flip-flop-generated glitches, and weak clocking of state devices. Checkit, an interHDL tool that you use on RTL code, identifies asynchronous loops and identifies and analyzes mixed-clock domains in your design.

Although it’s not the focus of this article, a class of RTL and gate-level code-enhancement tools used to analyze a design’s testability before simulation, design-for-test (DFT) analysis tools, is worth a brief mention.

Examples of testability-analysis tools are interHDL’s Testit and SynTest’s TurboCheck-RTL. Testit, with more than 30 technology-independent testability checks, looks for DFT violations and other constructs that may result in questionable testability. These violations include uncontrollable clock, reset, and memory-control signals; unscannable flip-flops; unobservable logic; and other potential test problems. When Testit finds a problem, the tool recommends ways to improve DFT, allowing you to add scan logic and other design changes to meet your testability requirements. TurboCheck-RTL combines Verilog-testability analysis with some design-code error checking.

A code-coverage tool monitors an HDL simulation and collects data on how completely the simulation exe-

## IN SITU CHECKING ALSO ENHANCES HDL CODE

Many tools you use for HDL-based chip design include some code-enhancement features, as either separate “up-front” operations or integrated within the design tool. If available, such features almost always include Verilog- and VHDL-syntax checks. In addition, simulation and synthesis tools may contain checks to help determine whether you can successfully simulate or synthesize your code. A couple of examples helps show the kinds of code-enhancement features you will find in your design toolbox.

If you use Synplify, Synplicity’s ([www.synplicity.com](http://www.synplicity.com)) programmable-logic-synthesis tool, you can run HDL-code syntax and synthesis checks before actually running a synthesis. If there are Verilog and VHDL key-word and punctuation errors, you can correct them at that time. Synthesis checking includes warnings for incomplete flip-flop definitions; latch generation; unused input pins; and partially assigned

buses, in which the tool sets the unassigned bits to zero. Synplify also has a few language-specific checks, such as one for unused signals defined in a VHDL sensitivity list.

A number of Synopsys ([www.synopsys.com](http://www.synopsys.com)) tools have code-checking features integrated within their analysis, simulation, and synthesis products. Sample checks include one in the company’s VSS simulator that determines whether VSS can synthesize the HDL specification you are simulating; a “check\_design” option in the Design Compiler synthesis engine that catches RTL-coding errors, such as combinatorial feedback loops, before synthesis; and checks for incompletely specified CASE statements read by HDL Compiler.

Remember the limitations of code checking and code enhancement when embedded in other design tools. They are not a substitute for the stand-alone tools described in this article and listed in Table 1.

## CODE-COVERAGE TOOLS

cuts the HDL code (**Reference 2**). You use code-coverage tools to check the effectiveness of your simulation test suites at testing your design's HDL models (see **sidebar** "Code coverage versus fault coverage"). When developing a testbench, you want to be able to exercise all parts of the model. If the test suite does not check some parts of the design, a design bug may go undetected. Conversely, redundancy in HDL-code checking is inefficient and wastes simulation time. You run a code-coverage tool during simulation to track met-

rics, such as how often statements execute, how often each branch of a conditional statement (IF or CASE) executes, and which signals have changed state. Using HDL-code-coverage tools helps you develop better chip test suites and helps you focus your efforts on portions of the design that you are not sufficiently exercising (simulating). Other benefits of code-coverage-tool usage are a reduction in the number of simulation cycles and more confidence when signing off on the RTL phase of your design. However, this extra checking is

not free—it increases simulator time.

Running a code-coverage tool with your simulation increases simulator time because of code-coverage overhead. The overhead depends on a number of factors, including HDL-code quality (how error-free the code is), coding style, the extensiveness of the coverage feature set, and the efficiency of the coverage-tool/simulator interface. The amount of coverage overhead varies among coverage tools. Code-coverage-tool vendors each have their own claims of simulator-time overhead,

**TABLE 1—COMMON CODE-COVERAGE-TOOL CAPABILITIES**

General HDL coverage	VHDL/Verilog applicability	Description	Design level			Summit Design	TransEDA
			Behavioral	RTL	Gate		
Statement (block, line)	Both	Shows whether a simulation has executed an HDL statement	Yes	Yes		HDLScore	VHDLcover, VeriSure
Branch	Both	Shows whether a simulation has executed conditional branches of IF or CASE statements	Yes	Yes		HDLScore	VHDLcover, VeriSure
Condition (expression)	Both	Shows whether all input conditions to an IF or CASE statement have been executed	Yes	Yes		HDLScore	VHDLcover, VeriSure
Toggle	Both	Shows which signal bits have changed state	Yes	Yes	Yes		VHDLcover, VeriSure
Path	Both	Calculates sequential paths through IF and CASE statements	Yes	Yes		HDLScore	VHDLcover, VeriSure
Variable-trace coverage	Verilog	Shows how well specific variables, such as state variables or ROM addresses, have been tested with a range of values	Yes	Yes	Yes	HDLScore	VeriSure
Signal-trace coverage	VHDL	Shows how well specific signals, such as state variables or ROM addresses, have been tested with a range of values	Yes	Yes	Yes		VHDLcover
Triggering	VHDL	Shows whether one of the signals in the process' sensitivity list has uniquely triggered each process	Yes	Yes			VHDLcover
State-machine coverage							
State visitation	Both	Shows whether a valid state has been reached				HDLScore	StateSure
Arc (transition)	Both	Shows whether a state transition has occurred				HDLScore	StateSure
Expression	Both	Shows whether the expressions controlling a state transition have been fully tested				HDLScore	StateSure
Sequence	Both	Shows whether specific state sequences have been tested				HDLScore	StateSure
Pair arc	Both	Shows whether a pair of state machines has simultaneously executed specific transitions					StateSure
Pair and sequence combinations	Both	Shows which complex multiple state-machine interactions have been tested					StateSure

ranging from as low as 5% (for relatively clean code and basic statement and branch checks) to as high as 100%. As a rule, the more coverage checking the tool does, the higher the tool's overhead. For example, deeper levels of expression coverage require more time. Running code coverage with your simulations may affect other simulator and platform requirements as well.

Along with overhead time, coverage tools may require additional memory than that needed for stand-alone simulation. When asking about a code-cov-

## THE BENEFITS OF LANGUAGE INDEPENDENCE

### STEVE CARLSON, ESCALADE

The most powerful form of control you have over synthesis is HDL-composition style. You can consider RTL written for synthesis as a metanetlisting specification of the design's logic architecture. Technology mapping notwithstanding, "good" RTL code produces few surprises when you run it through logic synthesis. The problem with using RTL constructs as a basis for the functional design is that you constantly need to change the basis throughout the implementation process. Reverifying modified designs is costly and time-consuming.

Second-generation high-level design automation (HLDA) uses an RTL description that is language-construct-independent. This independence allows a single functional description to automatically generate multiple-construct formulations. Broader design-space coverage using multiple-construct formulations of a design gives you better results without the cost of design reverification.

Second-generation HLDA affords a language independence that provides at least a partial answer to the challenges of retargeting intellectual-property (IP) blocks for different speed and area goals. Language-independent design lets you capture a single description that you can use as the source for multiple-implementation targets. You can implement performance-oriented or area-efficient versions of a design without modifying the design source. The higher abstraction level of second-generation HLDA also offers better results with process-technology migration.

An extension of the retargeting issue is support of multiple EDA targets. Second-generation HLDA's language independence lets you generate VHDL and Verilog from one source. Language-independent design also reduces HDL- and synthesis-knowledge requirements. Designers have immediate access to high-quality

results without the estimated four to nine months of "learning" time for HDL and synthesis proficiency. In non-English-speaking countries, English-centric VHDL and Verilog languages further hamper the adoption of HLDA. Key words and semantic structures make sense to those well-versed in English but are difficult for others to understand. The universality of a language-independent HLDA makes a tremendous difference in the adoption of HLDA methodologies.

The style in which you write your HDL-design function is the most powerful form of control you have over synthesis. Mere synthesis-tool-subset compliance does not ensure the production of a high-quality post-synthesis design implementation. Without running synthesis on HDL code, how do you predict what is good code and what is bad code? You can apply a number of analysis levels to your HDL source with code-enhancement tools that yield correspondingly deeper pictures of coding-style quality. Level 1: Syntax analysis. Ensure that the HDL complies with the lexical rules of the language—for example, parentheses and semicolon use.

Level 2: HDL semantic analysis. Ensure that the HDL is consistent with the semantics of the language specification, such as operators and operands of compatible types and sizes.

Level 3: Subset compliance. Ensure that the HDL constructs and construct combinations you use fall within the target tool's guidelines. An example is use of the VHDL wait statement when targeting a synthesis tool. Level 4: Simulation robustness. Ensure that semantics are not simulation-language-dependent. Examples include blocking versus nonblocking assignments, net merge and value resolution, and event-queue manipulation.

Level 5: Hardware semantic analysis. Ensure that a code-enhancement tool reports basic hardware structure and resources, such as registers, state machines, and datapath elements. The tool should also report possible design flaws, including clock- and reset-edge misuse, dangling circuit nodes, and multiple driver nodes.

*Steve Carlson is the vice president of marketing at Escalade ([www.escalade.com](http://www.escalade.com)).*

Advanced Technology Center	interHDL	Design Acceleration
CoverMeter	CoverIt	Coverscan/ Statescan
CoverMeter	CoverIt	Coverscan/ Statescan
CoverMeter		Coverscan/ Statescan
CoverMeter	CoverIt	
	CoverIt	
CoverMeter	CoverIt	Coverscan/ Statescan
CoverMeter	CoverIt	Coverscan/ Statescan
CoverMeter	CoverIt	Coverscan/ Statescan
CoverMeter		Coverscan/ Statescan
	CoverIt	Coverscan/ Statescan
		Coverscan/ Statescan

## CODE-COVERAGE TOOLS

erage tool, ask the vendor if you need additional memory and, if so, how much you need. Another point to consider is simulator compatibility. Code-coverage tools differ in the range of HDL simulators they support. When talking to a code-coverage-tool vendor, make sure you find out if the tool is compatible with your simulator.

A good time to start using code-coverage tools is during early development of functional-verification vectors—not after you finish the verification test-bench and are ready to simulate your design. An early start lets you catch functional-verification inadequacies early, when they are still easy to fix. You work with code-coverage and simulation tools design module by design module, independently verifying each module's coverage. After you develop testbenches for each module, you then check the complete chip's coverage. The hierarchical features of a code-coverage tool, such as



By analyzing code-coverage results on different pieces of your chip, you can find where “holes” exist in simulation coverage and focus your efforts on increasing design testability. In this HDLScore example, instance MUX1 in module fifo\_tb.FIFO\_INST has 100% block and path coverage but only 85% expression coverage (indicated by “ok ok 85” in the center of the figure next to fifo\_tb.FIFO\_INST.MUX1). The bottom of this photograph indicates the cumulative coverage of module fifo\_tb.FIFO\_INST as 89% block, 85% path, and 83% expression.

Summit's HDLScore, give you quantitative indications on coverage effectiveness for different parts of your chip (Figure 2). By seeing which blocks have insufficient test coverage after assembly with other blocks, you can concentrate your efforts on the insufficiently tested

modules to bring full-chip coverage to an acceptable level.

Searching for insufficient test coverage on your HDL design description encompasses several types of coverage (Table 1). The simplest type, statement coverage, verifies that the simulator has

**TABLE 2—REPRESENTATIVE EDA-VENDOR TOOLS FOR CODE-ENHANCEMENT AND -COVERAGE ANALYSIS**

Company	Product	Function	Verilog or VHDL	Abstraction level	Unix or Windows	Starting price
Advanced Technology Center	CoverMeter	Test coverage analysis	Verilog	Behavioral, RTL, gate	Unix	\$10,000
Design Acceleration	DAI Coverscan/ Statescan	Test coverage analysis with state-machine extraction	Verilog	RTL	Unix	\$15,000
interHDL	Verilint	Semantics, synthesis, and coding style checker	Verilog	RTL	Both	\$9500
	VHDLint	Semantics, synthesis, and coding style checker	VHDL	RTL	Both	\$9500
	Checkit	Clock-domain analysis and asynchronous-loop identification	Both	RTL, gate	Both	\$20,900
	Testit	Testability analysis	Both	RTL, gate	Both	\$20,900
	Coverit	Test coverage analysis with power profiling	Verilog	RTL, gate	Both	\$14,900
Quickturn Design Systems	StyleCheck	Syntax and design-rule checking	Verilog	RTL	Both	\$28,000
Summit Design	HDLScore	Test coverage analysis with state-machine extraction	Verilog	Behavioral, RTL, gate	Unix	\$22,000
SynTest Technologies	TurboCheck-RTL	Testability analysis	Verilog	RTL	Both	\$20,000
	FCE-RTL	Fault-coverage enhancer	Verilog	RTL	Both	\$10,000
	TurboFault	Fault grading	Both	Gate	Unix	\$50,000
TransEDA	VHDLcover	Test coverage analysis	VHDL	Behavioral, RTL, gate	Both	\$20,000
	VeriSure	Test coverage analysis	Verilog	Behavioral, RTL, gate	Both	\$20,000
	CoverPlus	Regression suite optimization	Both	RTL, gate	Both	\$25,000
	StateSure	Finite-state-machine verification	Both	Behavioral, RTL	Both	\$25,000

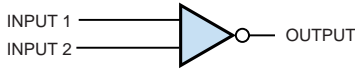
## CODE-COVERAGE TOOLS

## CODE COVERAGE VERSUS FAULT COVERAGE

Because fault- and code-coverage tools both deal with testability, you may be a little confused about the differences between these two types of EDA tools. You use a fault-coverage tool to check a test-vector suite's effectiveness in locating manufacturing defects, usually modeled as "stuck-at" faults. You use a code-coverage tool to check whether your simulations are effectively checking all parts of the design, which you've modeled in Verilog or VHDL. A simple example illustrates the differences between both types of analysis.

Figure A is the truth table for a two-input NAND gate. To test for a stuck-at-0 or stuck-at-1 condition on the output, you need only two input test vectors, (1,1) and (0,0). A fault-coverage tool expands on this basic stuck-at-fault principle by applying stimuli on various nodes in the circuit you are checking to propagate

FIGURE A



INPUT 1	INPUT 2	OUTPUT
1	1	0
0	1	1
1	0	1
0	0	1

Fault-coverage testing of this NAND's output pin requires only input vectors (1,1) and (0,0). To verify that the simulator has completely exercised the NAND gate, a code-coverage tool checks whether the simulator has applied all four possible input vectors.

stuck-at information to an observable point, generally one of the circuit's output pins. To completely verify NAND-gate operation, a

code-coverage tool checks whether a simulation tested all four input combinations. If the simulator applied only the same two input vectors used for fault coverage, the code-coverage tool recognizes that the gate's expression coverage was only 50%.

exercised, or checked, each executable line of Verilog or VHDL code. If you don't simulate a line of code, you don't know whether that line is correct, or even whether it's necessary. A more complex check, branch coverage, verifies that the simulator has taken every branch of an IF or CASE statement. Although statement coverage can fully check some CASE statements, there are some that need branch coverage to fully verify all conditional paths (**Reference 3**). Condition coverage expands an HDL expression, usually as Boolean-logic combinations, and checks whether the simulation executes the expression for each input combination. Unless you know that unverified

input combinations are illegal or "don't cares," if your simulation has not checked these combinations, you have an untested part of the design. To minimize the number of test patterns needed to fully test an expression, TransEDA uses a technology called focused expression coverage (FEC). FEC allows TransEDA's coverage tools to test a Boolean expression having N inputs using N+1 simulation vectors, instead of employing the exhaustive approach, which uses  $2^N$  vectors. Reducing the number of required simulation vectors helps you choose more effective tests for your expression coverage. A fourth type of coverage, path coverage, looks at sequential paths through IF and CASE

statements to see if particular sequences occur as expected. To check more complex design circuitry, you need additional types of coverage.

The complexity of finite-state machines (FSMs) makes the operation of these circuits difficult to verify. Checks you run on FSMs must answer the following questions:

- Can you reach all desired states without any states deadlocked?
- Are you reaching any illegal states?
- Has every desirable state transition occurred?
- Does your FSM have any "holes" (no response for a given input set)?
- Are there any conflicts (multiple responses for the same input set)?



StateSure's code-coverage analysis of state machines lets you see untested transitions and expressions either in a state diagram (a) or as annotated Verilog or VHDL code (b).

## CODE-COVERAGE TOOLS

- Has every condition triggering a desirable transition occurred?
- Has every desirable interaction between multiple state machines occurred?

Graphical-entry tools let you observe FSM operation and tell you the states visited and transitions made. However, these tools cannot measure FSM sequences, pair arcs, and pair sequences (Table 1). General code-coverage tools can measure states, transitions, and control values for the FSM, but they cannot verify sequences. Simulation tools don't let you look "inside" the FSM, they only let you observe output changes. To overcome these FSM-verification problems, you have FSM-specific coverage tools.

To check FSM code coverage, Design Acceleration, Summit Design, and TransEDA offer tools that specifically target these types of circuits. Design Acceleration's CoverScan/StateScan and Summit's HDLScore, both with automatic FSM extraction, combine some general and some FSM-specific checks. TransEDA's StateSure, a comple-

ment to the company's VeriSure and VHDLcover code-coverage products, is only for FSM-coverage checks (Figure 3). Both CoverScan/StateScan and StateSure have pair-arc capability, which lets you verify that state transitions in two state machines have occurred simultaneously.

Are these tools for you?

Even the best HDL designers let problems slip past the RTL-specification stage. If undetected, these problems may lead to time-consuming and expensive redesigns later in the design cycle. Code-enhancement tools are good insurance against certain syntax, semantic, and EDA-tool-construct errors that propagate with your HDL code. Pat Hefferan, CAD manager at Mitsubishi's (www.mitsubishi.com) Research Triangle Park (NC) design center, sums up Mitsubishi's need for code-coverage tools in one phrase: The tools help verify imported intellectual property (IP).

Hefferan and his colleagues work on a range of chips, including specialized memories,  $\mu$ Cs, ERAMs (embedded

logic in memories), and hard-disk controllers. Hefferan estimates that 25 to 50% of the circuitry on these chips is from third-party soft-IP blocks. Using a code-coverage tool, Mitsubishi finds functionality and code-quality problems in many imported-IP cores, especially in ones that the core vendor has recently released. In cases in which Mitsubishi buys a core and testbench from the same vendor, code coverage may be as low as 40% for some cores, indicating to Hefferan that the core vendor has done inadequate product checking. To check imported cores that the vendor has designed for standard functions, such as Universal Serial Bus or PCI, Mitsubishi often obtains a testbench from a vendor that specializes in testbench products. For these cases, Mitsubishi's designers check core and testbench coverage with a code-coverage tool. This method allows the designer to improve core and testbench quality before embedding the core in a chip.

As more system-on-chip (SOC) designs include IP not developed by the SOC vendor, the need for code-coverage and -enhancement tools increases (Table 2). If you think the price of these tools is a barrier to their purchase, remember that you can pay now, or you can pay much more later. **EDN**

## REPRESENTATIVE CODE-COVERAGE EDA COMPANIES

For more information on companies offering code-coverage tools, circle the appropriate numbers on the Information Retrieval Service card or use EDN's Express Request service. When you contact any of the following manufacturers directly, please let them know you read about their products in EDN.

Advanced Technology Center  
Laguna Hills, CA  
1-714-583-9119  
fax 1-714-583-9213  
www.covermeter.com  
Circle No. 358

Design Acceleration  
San Jose, CA  
1-408-885-1885  
fax 1-408-885-1886  
www.designacc.com  
Circle No. 359

interHDL  
Los Altos, CA  
1-650-428-4200  
fax 1-650-428-4201  
www.interhdl.com  
Circle No. 360

Quickturn Design Systems  
San Jose, CA  
1-408-914-6000  
fax 1-408-914-6001  
www.quickturn.com  
Circle No. 361

Summit Design  
Beaverton, OR  
1-503-643-9281  
fax 1-503-646-4954  
www.summit-design.com  
Circle No. 362

SynTest Technologies  
Sunnyvale, CA  
1-408-720-9956  
fax 1-408-720-9960  
www.syntest.com  
Circle No. 363

TransEDA  
Los Gatos, CA  
1-408-395-5014  
fax 1-408-395-4637  
www.transeda.com  
Circle No. 364

## VOTE . . .

Please also use the Information Retrieval Service card to rate this article (circle one):  
High Interest 570  
Medium Interest 571  
Low Interest 572

## Super Circle Number



For more information on the products available from all of the vendors listed in this box, circle one number on the reader service card.

Circle No. 365

## References

1. Keating, Mike and Pierre Bricaud, "Reuse Methodology Manual," Kluwer Academic Publishers, www.wkap.nl/book.htm/0-7923-8175-0.
2. Abrahams, Martin and Stuart Riches, "Optimize ASIC test suites using code-coverage analysis," EDN, May 21, 1998, pg 149.
3. Tempero, Thomas, "Introduction to HDL Verification Coverage," Design Acceleration white paper.



You can reach Technical Editor Jim Lipman at 1-925-606-1370, fax 1-925-606-1563, ednlipman@mcimail.com.

## VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest  
570

Medium Interest  
571

Low Interest  
572