Processor validation: a top-down approach

Prabhat Mishra

©EYEWIRE COMPOSITE: MKC

Shrinking time to market, coupled with short product lifetimes, has created a need to drastically shorten the design time for microprocessors. Verification and design analysis are major components of this cycle time. Thus, any effort that improves verification effectiveness and design quality is crucial for meeting customer deadlines and requirements.

Design validation techniques fall into two broad categories: simulation-based approaches and formal techniques. Because of the complexity of modern designs, validation using only traditional scalar simulation cannot be exhaustive. Formal techniques exhaustively analyze parts of the design but, because of the state space explosion, are not suitable for the complete design. (Formal techniques traverse the finite state machine (FSM) of a design during verification and perform necessary analysis and computations using algorithms. Each iteration of an algorithm creates new states (typically an exponential increase). As a result—even before completion, the number of states soon becomes intractable for the complete design (containing millions of gates). This phenomenon is known as state space explosion.)

Equivalence checking is a formal technique that is popular in industry today. Typically, this technique involves comparing the implementation to a set of Boolean equations or comparing an optimized circuit to the original circuit. Symbolic simulation is an efficient technique that bridges the gap between traditional simulation and full-fledged formal verification. This article presents a top-down methodology for validating microprocessors using a combination of symbolic simulation and equivalence checking.

Traditional processor validation flow

Figure 1 shows a traditional architecture validation flow. The architect prepares an informal specification of the microprocessor in a written document. Logic designers implement the modules at the register-transfer level (RTL). The verification team validates the design implementation using a combination of simulation techniques and formal methods.

Simulation, the most popular form of microprocessor validation, involves running millions of cycles using random or pseudo-random tests. The validation team applies model checking to a high-level description of the design abstracted from the RTL implementation.

Formal verification uses a formal language to describe the system. The specification for the formal verification comes from the architecture description; the implementation can come from either the architecture specification or the abstracted design. The validated RTL design serves as a golden reference model for future design modifications. After applying design transformations, including synthesis, to the RTL design, the validation team uses equivalence checking to validate the modified design against the RTL design.

Existing validation techniques use reverse engineering to derive processor functionality from its RTL implementation. This article presents a top-down validation methodology that complements existing bottom-up verification techniques.

Top-down validation

Figure 2 shows a top-down validation methodology. Validation engineers specify the processor's structure and behavior using an architecture description language (ADL) such as EXPRESSION ADL. The ADL description then needs to be validated to ensure that it specifies a well-formed architecture. To verify that the RTL design implementation satisfies certain properties (e.g., correct register read/write), the top-down validation framework generates behaviors for the intended properties. A symbolic simulator such as *Versys2* can be used to perform property checking. The framework also generates the processor's complete description to enable equivalence checking using tools such as *Formality* <http://www.synopsys.com>. When a failure occurs, validation engineers use the feedback (the error generated by the tool) to modify the RTL design. If there is an ambiguity in the original description that led to the mismatch, the architecture specification must be updated. This methodology has three important steps:

1. capturing the architecture using an ADL specification;
2. generating the reference model from the architecture specification; and
3. performing design validation using a combination of symbolic simulation and equivalence checking

ADL specification

ADL-driven frameworks are traditionally used to enable rapid design space exploration of programmable embedded systems. The framework captures the processor, coprocessor, and memory architectures. From the ADL specification, it then generates a software toolkit including a compiler and a simulator. The application programs are compiled and simulated. The feedback (information about performance and code size) is used for modifying the ADL specification of the architecture.

The goal is to find the best possible architecture for the given set of application programs under various design constraints such as area, power and performance. ADLs traditionally fall into two categories, depending on whether they primarily capture the processor behavior (instruction set) or its structure. Several recently proposed ADLs, such as EXPRESSION and LISA, capture both the structure and the behavior.

Box A

```
(OPCODE add
    (OPERANDS  (src1 reg) (src2 reg/imm16) (dest reg))
    (BEHAVIOR  dest = src1 + src2)
    (FORMAT    cond (31-30)  0101  dest (25-21)  src1 (20-16)  src2 (15-0))
)
```

EXPRESSION captures all the architectural components and their connectivity as a netlist. It considers two types of components: units (for example, arithmetic logic units) and storage locations (for example, register files). It also captures two types of connections in the netlist: pipeline and data transfer edges. Pipeline edges specify instruction transfer between units through pipeline latches; data transfer edges specify data transfer between components, typically between units and storage locations or between two storage locations.

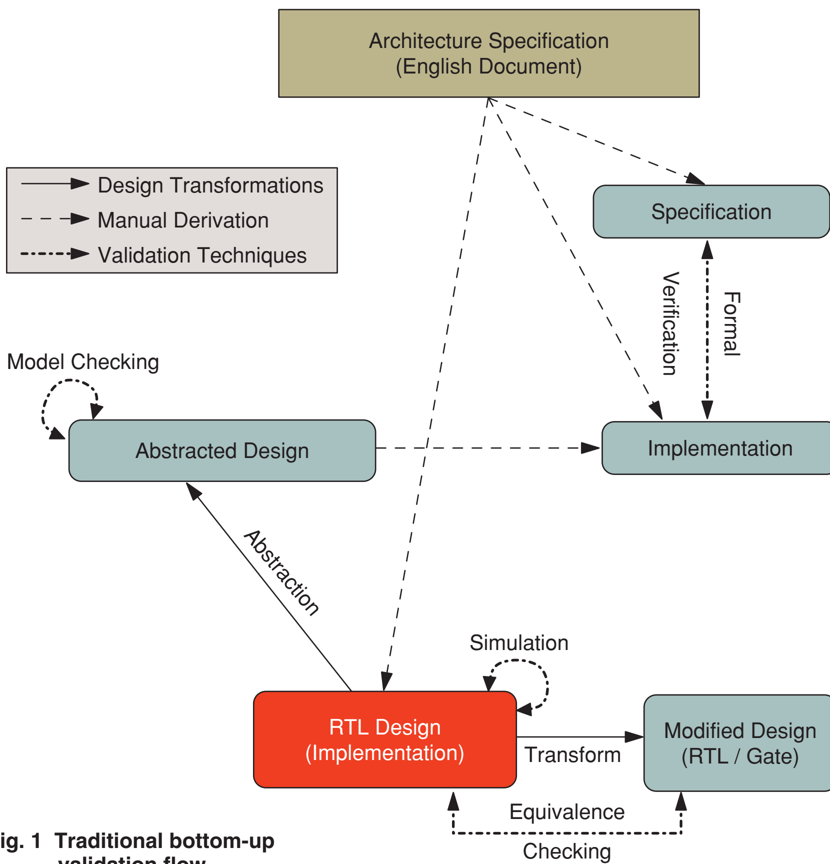

Fig. 1 Traditional bottom-up validation flow

For example, in Fig. 3, the oval (unit) and rectangular (storage) boxes represent components. The solid (pipeline) and dotted (data transfer) lines represent edges. The behavior is organized into operation groups. Each group has a set of operations with some common characteristics. Each operation is then described in terms of its opcode, operands, behavior and instruction format. Each operand is classified either as a source or as a destination. Furthermore, each operand is associated with a type that describes the type and size of the data it contains. The instruction format describes the operation fields in binary and assembly codes. For example, the description of an ADD operation is shown in Box A.

Reference model generation

This framework uses functional abstraction to generate the reference model [VHSIC Hardware Description Language (VHDL) description] from the ADL specification. The notion of functional abstraction comes from a simple observation: different architectures can use the same functional unit (such as a fetch unit) with different parameters; use the same functionality (for example, operand read) in different functional units; or have new architectural features.

Defining generic functions with appropriate parameters can eliminate the first difference. The second one is eliminated by defining generic sub-functions that different architectures can use at different times (for example, during fetch or decode). The last difference is difficult to eliminate because it is new. Unless, the new functionality can be composed using existing sub-functions; e.g., by combining MUL and ADD operations to create a multiply-accumulate (MAC) operation. The goal is to define a set of necessary abstraction primitives using generic functions and sub-functions. The framework generates a reference model from the ADL specification of the architecture by composing the functions and sub-functions.

The framework generates a complete description of the architecture as well as specific properties. The complete description is used to check for equivalence with the given implementation. However, having the specific properties would enable property checking. For example, for an n-input adder, the framework generates the following property:

$$output = \sum^{n}_{i=1} input_i$$

The adder design should satisfy this property regardless of the adder implementation— whether it is ripple-carry or carry look-ahead, for example.

The major advantage of property checking is that it reduces the verification complexity. However, this raises an important question: how to choose the set of properties? There are two ways. One way is for designers to decide which properties are important to verify based on their design knowledge and experience. Designers can then choose the properties to uncover those otherwise difficult-to-find bugs. Alternatively, designers can choose a set of behaviors and evaluate their effectiveness. For example, verifying a memory controller in a microprocessor requires generation

of properties to validate each of the controller's outputs. To measure these properties' effectiveness, designers can use certain coverage measures during property checking.

The top-down methodology uses two validation techniques: property checking using symbolic simulation and equivalence checking.

Property checking

Property checking can be performed using symbolic simulation, which combines traditional simulation with formal symbolic manipulation. Each symbolic value represents a signal value for different operating conditions, parameterized in terms of a set of symbolic Boolean variables. With this encoding, a single symbolic simulation run can cover many conditions that would require multiple runs on a traditional simulator.

Figure 4a shows a simple n-input *AND* gate. Exhaustive simulation of the *AND* gate requires 2^n binary test vectors. However, the ternary simulation, which uses 0, 1 and X (where X is "don't care"), requires (n+1) test vectors for the *AND* gate. Figure 4b shows the vectors: n vectors with one input set to 0 and the remaining set to X, and one vector with all inputs set to 1. Finally, symbolic simulation requires only one vector using n symbols (S_1, S_2, …, S_n), as Fig. 4c shows.

Researchers at IBM first introduced symbolic simulation to reason about properties of circuits described at the RTL. With the advent of binary decision diagrams (BDDs), the technique became far more practical. Providing a canonical representation for Boolean functions, BDDs enable the implementation of an efficient event-driven logic simulator operating over a symbolic domain. By encoding a model's finite domain with Boolean encoding, it is possible to symbolically simulate the model using BDDs. Seger and Bryant's work on symbolic trajectory evaluation (STE) helped to renew further interest in symbolic execution.

STE differs from symbolic simulation in that it provides a mathematically rigorous method for establishing that properties (assertions) of the form antecedent A ➨ consequent C hold. For the test vector shown in Fig. 4c, the antecedent is (I_1 is s_1, I_2 is s_2, …, I_n is s_n) from time 0 to 1, and the consequent is (*out* is s_1 & s_2 & …s_n) from time 1 to 2.

Symbolic values specified by the antecedent are used to initialize the circuit's state holders. A symbolic simulator then simulates the model, typically for one or two clock cycles, while driving the inputs with symbolic values. The simulator compares the resulting values—which appear on selected internal nodes and primary outputs—with the expected values expressed in the consequent.

In general, the values could be functions over a finite set of variables. A trajectory is a sequence of states such that each state has at least as much information as the next-state function applied to the previous one. Intuitively, a trajectory is a state sequence constrained by the system's next-state function. A successful simulation of the assertion A➨ C establishes that any sequence of value assignments to circuit nodes that is both consistent with the circuit behavior and consistent with antecedent *A* is also consistent with consequent *C*.

STE can verify that an implementation satisfies its specification (reference model). The assertion generator extracts necessary assertions from the reference model. If the implementation is correct, these assertions should hold during symbolic simulation of the RTL design. Assertion *A* ➨ *C* holds if the weakest antecedent trajectory that the implementation goes through during simulation (using *A*) is at least as strong as the weakest sequence satisfying consequent C. Informally, the outputs produced during simulation (using A) should be at least as strong as the expected outputs (given in *C*).

Consider a property-checking scenario for a memory management unit (MMU) of a typical microprocessor. A typical MMU consists of blocks such as segment registers, translation look-aside buffers (TLBs) and block address translation (BAT) arrays. Each block in turn consists of many subblocks. Each subblock is implemented using Static Random Access Memory (SRAM). Typical operations in SRAM are read and write. Therefore, it is natural to verify the

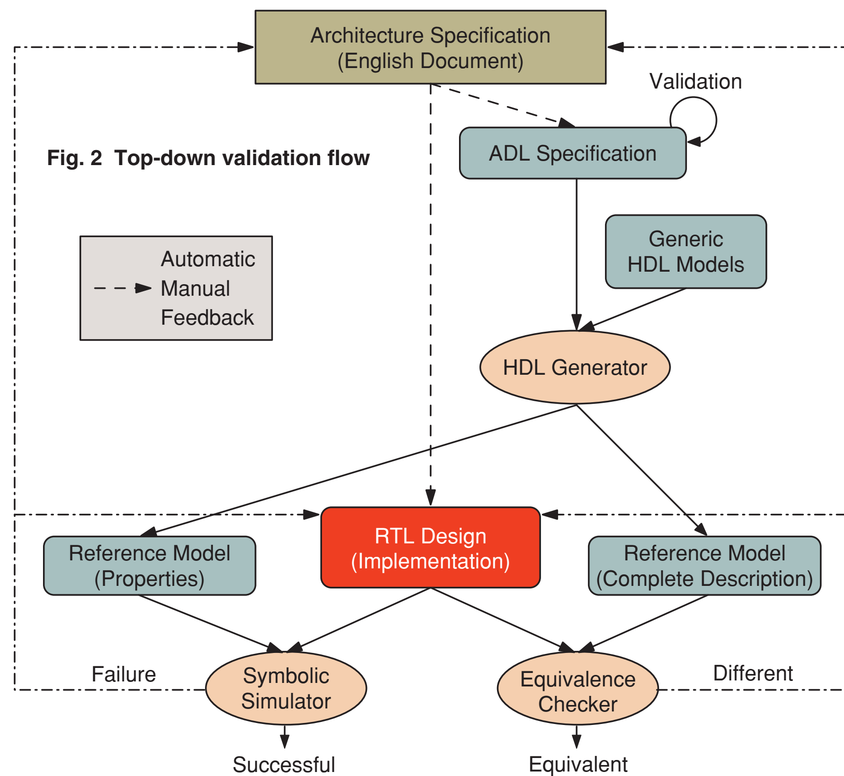

Fig. 2 Top-down validation flow

Box B

```
assign out = (rdClk & rdEn) ? ram[rdAddr] : 32'b0
```

Read Property

```
always @ (wrClk or wrEn or dIn or wrAddr)
begin
    if (wrClk & wrEn) ram[wrAddr] <= dIn;
end
```

Write Property

read and write properties of each subblock (SRAM). The generated reference model to verify each SRAM cell's read and write properties contains the Verilog code segment shown in Box B.

To verify that the MMU implementation satisfies these properties, the Versys2 symbolic simulator can be used to perform property checking. Versys2 accepts two inputs: MMU implementation and properties (the reference model). The simulator requires manual specification of the state mappings between the reference model and the implementation. This involves mapping both latches and bit cells by specifying their names. The assertion generator in Versys2 automatically generates the assertions from the reference model.

Versys2 symbolically simulates the MMU implementation by using the generated assertions to ensure that the design satisfies the reference model. Versys2 generates a counterexample if an assertion fails in the RTL design. This counterexample is used to modify the RTL design.

Equivalence checking

This branch of static verification uses formal techniques to prove whether two versions of a design are functionally equivalent. The equivalence checking flow has four stages: reading, matching, verification and debugging. Matching and verification are the two stages that design transformations affect the most.

During the reading stage, the equivalence-checking tool reads both design versions, and then segments them into manageable sections called logic cones. Logic cones are groups of logic bordered by registers, ports or black boxes. Figure 5a shows the cones for a typical design block. A logic cone's output border is the *compare point*. For example, in Fig. 5a, *OUT1* is the compare point of *Cone1*.

In the matching phase, the tool tries to match, or map, compare points from the reference (golden) design to their corresponding compare points within the implementation design. Two types of matching techniques are used: name based (non-function) and function (signature analysis) based. Figure 5b shows compare point matching for a typical reference design and implementation.

For better performance, name-based methods, which are more efficient, should complete most of the matching. Design transformations can result in the matching of fewer cones using the name-based techniques, which slows down performance. (It cannot preserve all the names since parts of the design are getting added/deleted/merged during design transformations.) Creating compare rules can assist name-based techniques, but determining and creating the rules can be time-consuming.

If the implementation differs drastically from the reference design, it is not possible to write the design rules. In such a scenario, the compare points can be matched manually for better performance or by using costly function-based techniques. Either way, this approach is impractical for designs with many unmatched points.

The verification stage proves whether each matched compare point is either functionally equivalent or not. Design transformations can affect a logic cone's structure in the implementation design. When logic cones are very dissimilar, performance suffers. In some cases, such as during retiming, logic cones can change so significantly that additional setup is necessary to successfully verify the designs. The debugging phase begins when the tool has returned a nonequivalent result. Unaccounted design transformations can cause a false-negative result, leading to a loss of valuable time spent debugging designs that are actually equivalent. The solution is to perform additional setup to guide the tool for the given designs.

Consider an equivalence-checking scenario for the DLX processor. The processor can be specified using ADL. The reference model (the synthesizable RTL description) of the DLX architecture can be generated from the ADL specification. The generated reference model can be used to verify a DLX implementation such as the synthesizable 32-bit RISC DLX implementation from <http://www.eda.org/rassp/vhdl/models/processor.html>.

An equivalence checker, such as Synopsys Formality, can be used to verify whether two designs are equivalent. The tool reads both the reference design and the implementation. It then tries to match the compare points between them. The unmatched compare points must be mapped manually. The tool next tries to establish equivalence for each matched compare point. When a failure occurs, the validation team needs to analyze the failing compare points to verify whether they are actual failures. The team can use the feedback to perform additional setup

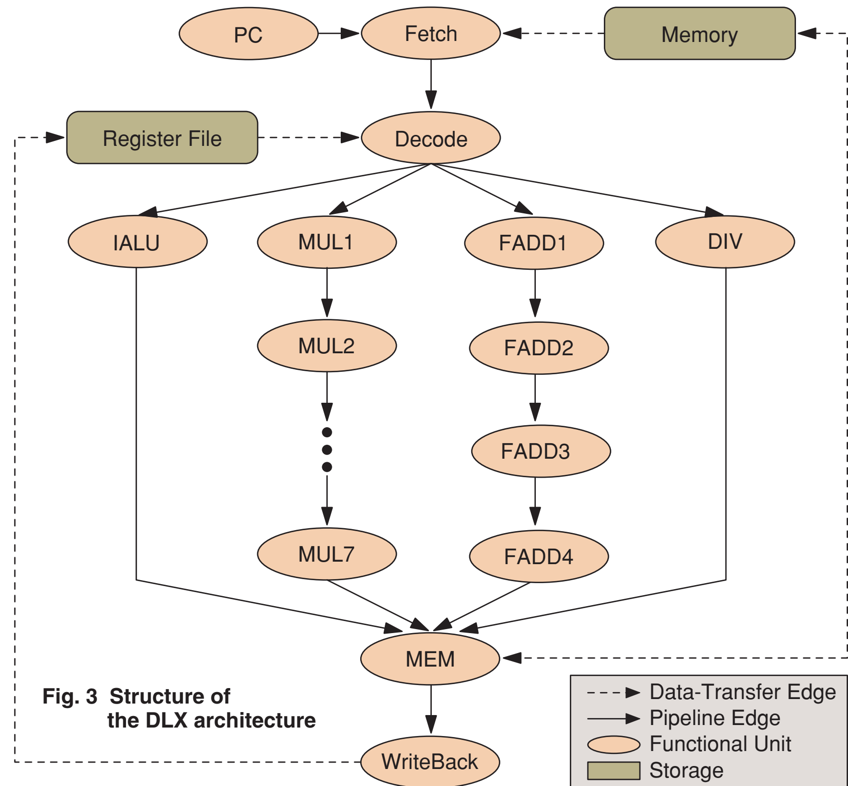

Fig. 3 Structure of the DLX architecture

(in case of a false negative) or to modify the RTL design implementation.

Conclusions

There is no argument that validation is one of the most important problems in today's processor design methodology. A significant bottleneck in the validation of such systems is the lack of a golden reference model. As a result, many existing approaches employ a bottom-up validation approach by using a combination of simulation techniques and formal methods. This article presented a top-down validation methodology that complements the existing bottom-up techniques.

Specification-driven hardware generation and validation of design implementation using equivalence checking has one limitation: the structure of the generated hardware (reference) model needs to be similar to that of the implementation. This requirement is primarily due to the limitation of the equivalence checkers available today. Equivalence checking is not possible using these tools if the reference and implementation designs are large and drastically different.

In reality, the implementation goes through several changes due to various requirements, such as area, cost, power and performance. As a result, the final implementation's structure might not be similar to that intended in the original specification. An improved methodology is needed that would enable reference model generation and design validation without any knowledge of the implementation details.

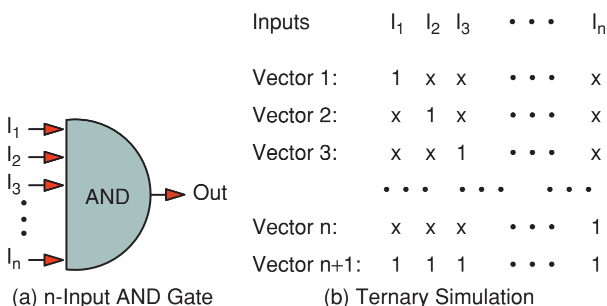

Fig. 4 Test vectors for validation of an AND gate

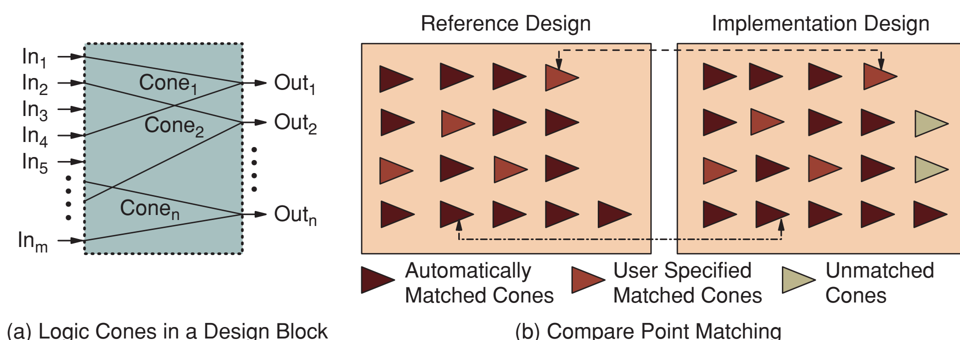

Fig. 5 Matching of compare points between reference design and implementation design

Read more about it

- P. Mishra, N. Dutt, N. Krishnamurthy, and M. Abadir, "A Top-Down Methodology for Microprocessor Validation," *IEEE Design and Test of Computers*, volume 21, number 2, pages 122-131, March-April, 2004.
- A. Halambi et al., "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," *Design Automation and Test in Europe (DATE)*, 1999, pages 485-490.
- N. Krishnamurthy et al., "Design and Development Paradigm for Industrial Formal Verification Tools," *IEEE Design & Test*, vol. 18, no. 4, July-Aug. 2001, pages 26-35.
- P. Mishra, N. Dutt, and A. Nicolau, "Functional Abstraction Driven Design Space Exploration of Heterogeneous Programmable Architectures," *Int'l Symposium on System Synthesis (ISSS)*, 2001, pages 256-261.
- C. Seger and R. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, March 1995, pages 147-189.
- L. Wang, M. Abadir, and N. Krishnamurthy, "Automatic Generation of Assertions for Formal Verification of PowerPC Microprocessor Arrays Using Symbolic Trajectory Evaluation," *Design Automation Conference (DAC)*, 1998, pages 534-537.
- J. Marques-Silva and T. Glass, "Combinational Equivalence Checking Using Satisfiability and Recursive Learning," *Design Automation and Test in Europe (DATE)*, 1999, pages 145-149.
- P. Mishra, A. Kejariwal and N. Dutt, "Synthesis-driven Exploration of Pipelined Embedded Processors," *International Conference on VLSI Design*, 2004, pages 921-926.

About the author

Prabhat Mishra is an assistant professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include design and verification of embedded systems, VLSI CAD, and computer architecture. Mishra received his B.E. from Jadavpur University, India, M.Tech. from the Indian Institute of Technology, Kharagpur, and Ph.D. from University of California, Irvine – all in Computer Science. He is a member of the IEEE and ACM.

Adapted from P. Mishra, N. Dutt, N. Krishnamurthy, and M. Abadir, "A Top-Down Methodology for Microprocessor Validation," *IEEE Design and Test of Computers*, volume 21, number 2, pages 122-131, March-April, 2004

Find the solutions to shape your future visit IEEE online at:

www.ieee.org/gold