A Top-Down Methodology for Microprocessor Validation

Prabhat Mishra and Nikil Dutt
University of California, Irvine

Narayanan Krishnamurthy and Magdy S. Abadir
Motorola

Editor's note:
A major challenge in today's functional verification is the lack of a formal specification with which to compare the RTL model. The authors propose a novel top-down verification approach that allows specification of a design above the RTL. From this specification, it is possible to automatically generate assertion models and RTL reference models. The authors also demonstrate that symbolic simulation and equivalence checking can be applied to verify an RTL design against its specification.
—Li-C. Wang, University of California, Santa Barbara

THE COMBINATION of shrinking time to market and short product lifetimes makes it critical to drastically reduce microprocessor design cycle time. Because verification and design analysis are major components of this cycle time, 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 state space explosion, are not suitable for the complete design. 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 validation of microprocessors using a combination of symbolic simulation and equivalence checking.

Figure 1 shows a traditional architecture validation flow. In current validation methodology, the architect prepares an informal specification of the microprocessor in the form of an English document. Logic designers implement the modules at the 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 pseudorandom 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 processor validation techniques fall into two categories, depending on the models used for specification and implementation. Those in the first category don't verify the actual RTL design implemented by the logic designers. Instead, they verify an implementation that's described in a formal language either written manually or generated using design abstraction techniques. Hence, this verification doesn't uncover the bugs in the actual design. Traditional formal verification techniques fall into this category. Techniques in the second category focus on the actual hardware description language (HDL) implementation. However, because they lack a golden reference model, these techniques use reverse engineering to derive the specification model from the actual implementation. Simulation-based techniques fall into this category. Existing validation techniques use reverse engineering to derive a processor's functionality from its RTL implementation.

0740-7475/04/$20.00 © 2004 IEEE Copublished by the IEEE CS and the IEEE CASS

Our validation technique complements these bottom-up approaches. We leverage the system architect's knowledge of the processor's behavior through architecture description language (ADL) constructs, thus providing a powerful top-down approach to microprocessor validation. Our framework generates behaviors of the intended properties to enable model checking, and generates a complete description of the processor to enable equivalence checking. We've applied our methodology in two validation scenarios: property checking of a memory management unit for a microprocessor compliant with the PowerPC instruction set, and verification of the DLX processor.

Top-down validation methodology

Figure 2 shows our validation methodology. Validation engineers specify the processor's structure and behavior using an ADL. Validating the ADL description ensures that it specifies a well-formed architecture.[1] Our framework uses the Expression ADL,[2] and automatically generates the reference model (HDL description) from the ADL specification. To verify that the RTL design implementation satisfies certain properties, our framework generates behaviors for those intended properties. We use the Versys2 symbolic simulator to per-

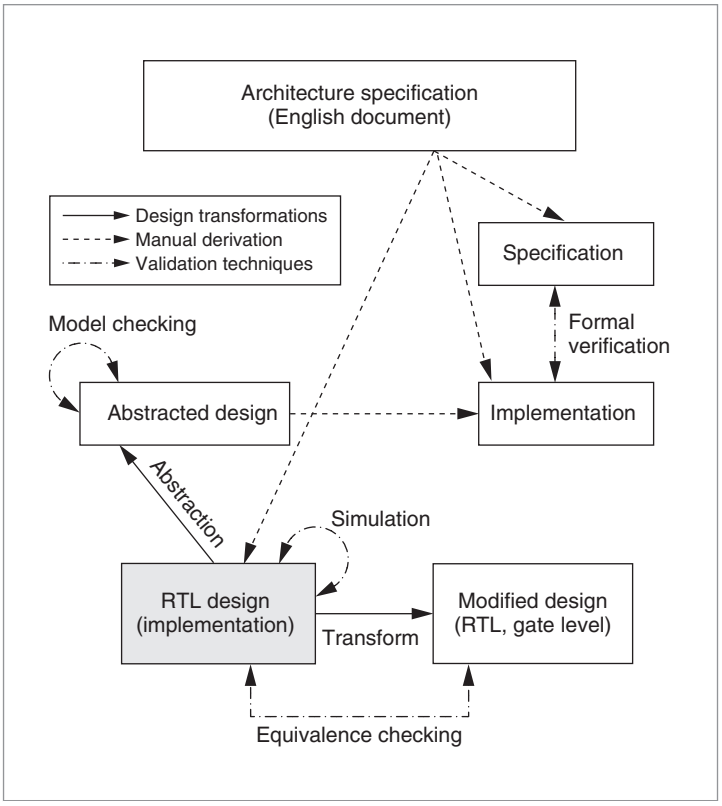


Figure 1. Traditional bottom-up validation flow.

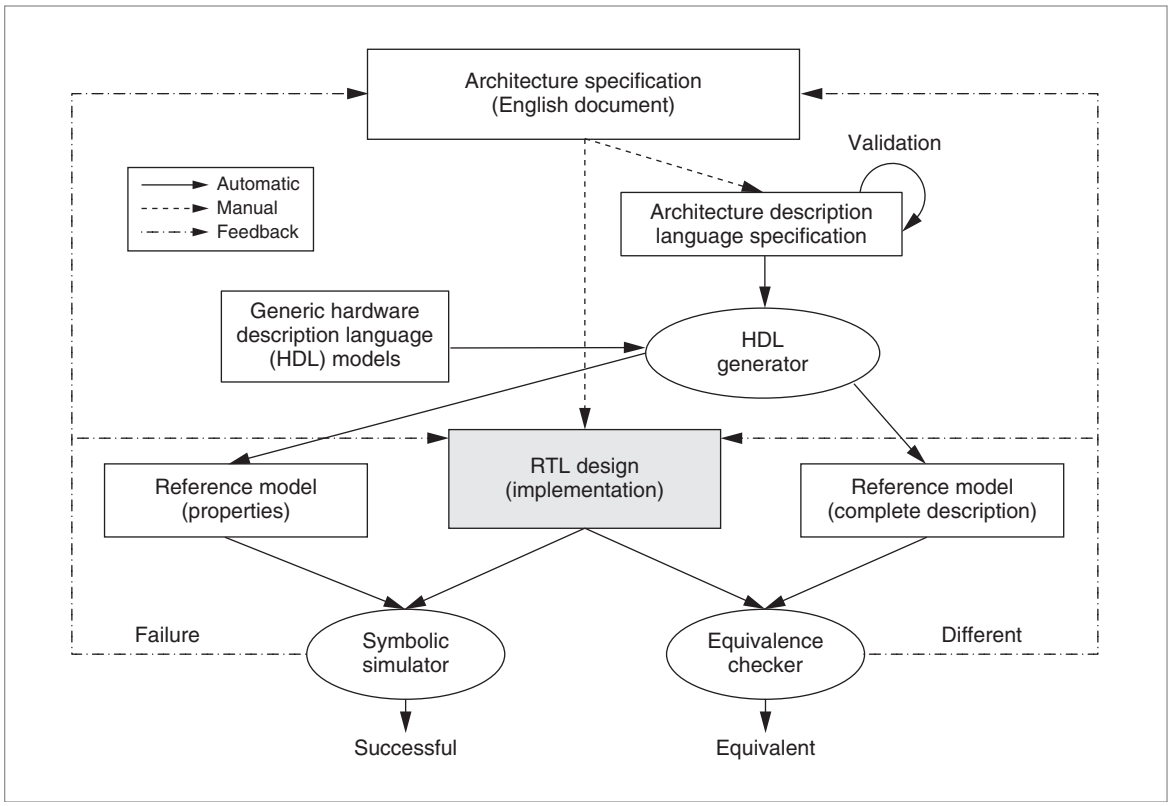


Figure 2. Top-down validation flow.

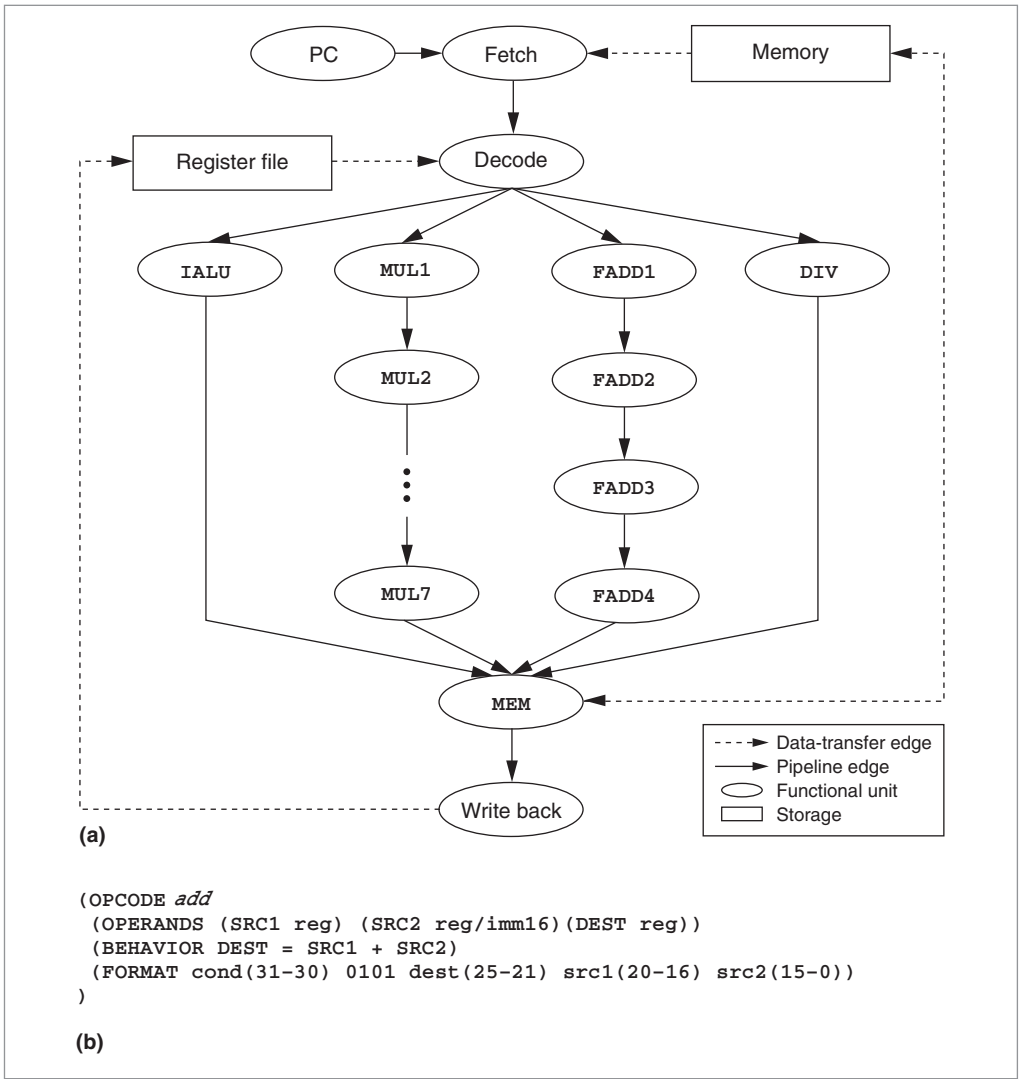


Figure 3. Specification of DLX architecture using the Expression architecture description language (ADL): DLX architecture structure (a), and an ADD operation (b).

form property checking.[3] Our framework also generates the processor's complete description to enable equivalence checking using Synopsys' Formality (http://www.synopsys.com). When a failure occurs, validation engineers use the feedback (the error generated by Formality) to modify the RTL design. If an ambiguity in the original description led to the mismatch, the architecture specification must be updated.

Our methodology involves three important steps:

- capturing the architecture using an ADL specification;
- generating the reference model from the architecture specification; and
- performing design validation using symbolic simulation and/or equivalence checking, depending on the generated reference model.

ADL specification

ADL-driven frameworks typically enable rapid design space exploration of programmable embedded systems. The framework captures the processor and memory architecture and, from the ADL specification, generates a software toolkit, including a compiler and a simulator. After compilation and simulation of the application programs, the feedback (information about performance and code size) is helpful for modifying the architecture's ADL specification. The goal is to find the best possible architecture for the given set of application programs under area, power, and performance constraints. ADLs traditionally fall into two categories, depending on whether they primarily capture the processor's behavior (instruction set) or its structure. Several recently proposed ADLs—including Expression, the ADL our framework uses—capture both the structure and the behavior.

Expression captures all the architectural components and their connectivity as a netlist. It considers two types of components: units (for example, ALUs) 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. For example, in Figure 3a, the oval (unit) and rectangular (storage)

boxes represent components, and the solid (pipeline) and dotted (data transfer) lines represent edges. The behavior is organized into operation groups, with each group containing a set of operations having some common characteristics. Each operation is then described in terms of its opcode, operands, behavior, and instruction format. Each operand is classified either as source or as 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's fields in binary and assembly codes. For example, Figure 3b shows the description of an ADD operation.

Reference model generation

Our framework uses functional abstraction to generate the reference model (VHDL description) from the ADL specification. Mishra, Dutt, and Nicolau first introduced their functional-abstraction technique to generate simulation models for a wide variety of architectures.[4]

Functional abstraction. 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. We can eliminate the first difference by defining generic functions with appropriate parameters. We can eliminate the second one by defining generic subfunctions, which different architectures can use at different times. The last difference is difficult to eliminate, because it's new, unless we can create this new functionality using existing subfunctions—for example, by combining MUL and ADD operations to create a multiply-accumulate (MAC) operation.

Functional abstraction captures each functional unit's structure using parameterized functions. For instance, the fetch unit's functionality contains several parameters, such as number of operations read per cycle, number of operations written per cycle, reservation station size, and branch prediction scheme. Figure 4a shows a specific example of a fetch unit described using subfunctions. We define each subfunction using appropriate parameters. For example, `ReadInstMemory` reads *n* operations from the instruction cache using the current PC address (returned by `ReadPC`) and writes them to the reservation station. Having generic subfunctions provides the flexibility to specify the system in finer detail and allows for component reuse.

```
FetchUnit ( # of read/cycle, res-station size, ....)
{
      address = ReadPC();
      instructions = ReadInstMemory(address, n);
      WriteToReservationStation(instructions, n);
      outInst = ReadFromReservationStation(m);
      WriteLatch(decode_latch, outInst);

      pred = QueryPredictor(address);
      if pred {
        nextPC = QueryBTB(address);
        SetPC(nextPC);
      } else
        IncrementPC(x);
}
(a)

ADD (src1, src2) {
      return (src1 + src2);
}

MUL (src1, src2) {
      return (src1 * src2);
}

MAC (src1, src2, src3) {
      return (ADD(MUL(src1, src2), src3));
}
(b)
```

Figure 4. Modeling architectural components using functional abstraction: subfunctions describing a fetch unit (a); modeling of multiply-accumulate (MAC) function using two independent existing operations, ADD and MUL (b).

We capture a generic processor's behavior through the definition of operations and define each operation as a function with a generic set of parameters that performs an intended functionality. The parameter list includes source and destination operands, necessary control, and data type information. Some common subfunctions are ADD, SUB, MUL, and SHIFT. Opcode functions can use one or more subfunctions. For example, the MAC function uses two subfunctions (ADD and MUL), as Figure 4b shows. Mishra, Dutt, and Nicolau provide a detailed description of generic abstractions for all microarchitectural components.[4]

We implemented all the generic functions and subfunctions using VHDL. Our framework generates a VHDL description from the processor's ADL specification by composing functional-abstraction primitives. Here, we describe how to generate three major processor components—the instruction decoder, the data path, and the controller—using the generic VHDL models.[5]

In decoding a given instruction, a generic instruction

decoder uses information regarding an individual instruction's format and opcode mapping to each functional unit. The instruction format information is available in Expression's operations section. The decoder uses the instruction format to extract information regarding opcode, operands, and so on from the input instruction. Expression's mapping section captures the information regarding the mapping of opcodes to the functional units. The decoder uses this information to perform necessary tasks (such as operand read) and to decide where to send instructions.

The data path implementation has two parts: First, the HDL generator in our framework composes each component in the structure. Second, the HDL generator instantiates components (for example, fetch, decode, ALU, load/store, write back, caches, register files, and memories) and establishes connectivity using the appropriate number of latches, ports, and connections from the structural information available in the ADL. To compose components, the generator uses the information available in the ADL regarding the components' functionality and parameters. For example, to compose an execution unit, it instantiates all opcode functionalities (ADD, SUB, and so on, for an ALU) supported by that unit. If the execution unit requires a read connection, the generator must instantiate the appropriate number of operand read functionalities unless several units can share the same read functionality using multiplexers. Similarly, if this execution unit has to write the data back to the register file, its instantiation must include the functionality for writing the result. The actual implementation of an execution unit could contain many more functionalities—for example, read latch and write latch.

The controller implementation has two parts. First, using the generic controller function with the appropriate parameters, the HDL generator produces a centralized controller that maintains the information (busy, stalled, and so on) for each functional unit. This centralized controller computes hazard information based on the list of instructions currently in the pipeline. On the basis of these bits and the information available in the ADL, this controller stalls and flushes the necessary pipeline units. Second, each functional unit in the pipeline has a local controller, which generates certain control signals and sets necessary bits on the basis of the input instruction. For example, the local controller in an execution unit activates the ADD operation if the opcode is add, or marks the unit as busy if it is executing a multicycle operation.

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

$$output = \sum_{i=1}^{n} input_i$$

The 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 verification complexity. However, this raises an important question: How do we 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. They can then choose the properties to uncover 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 generating properties to validate each of the controller's outputs. To measure these properties' effectiveness, designers can use certain coverage measures during property checking.[6]

Design validation

Here we describe the two validation techniques used in our framework: property checking using symbolic simulation and equivalence checking.

Property checking. In our methodology, property checking uses symbolic simulation, which combines traditional simulation with formal symbolic manipulation.[7] Each symbolic value represents a signal value for different operating conditions, parameterized in terms of a set of symbolic Boolean variables. By this encoding, a single symbolic-simulation run can cover many conditions that would require multiple runs on a traditional simulator. Figure 5a shows a simple *n*-input AND gate. Exhaustive simulation of the AND gate requires $2n$ 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 5b shows the vectors: *n* vectors with one input set to 1 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 ($S_1, S_2, \ldots, S_n$), as Figure 5c shows.

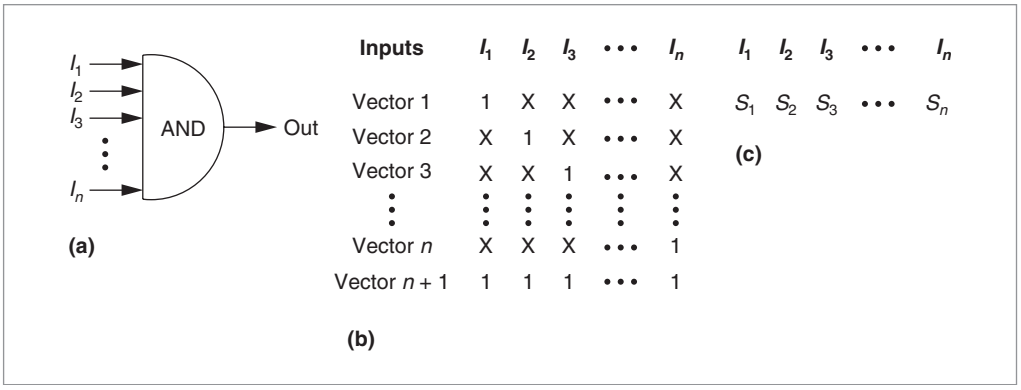


Figure 5. Test vectors for validating an AND gate: *n*-input AND gate (a), and test vectors for ternary (b) and symbolic (c) simulations.

Researchers at IBM first introduced symbolic simulation to reason about properties of circuits described at the RTL. With the advent of binary decision diagrams, 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's 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.[8]

STE is a modified form of symbolic simulation that operates over the quaternary logic domain (0, 1, X, and T), where T can be viewed either as a state vector in which each node is simultaneously at 0 and 1 or as an unconstrained state.[8] A circuit state is the set of all node values at a particular time instant. The value domain is partially ordered and forms a complete lattice, $X \sqsubseteq 0$, which means X has less information than 0, or X is weaker than 0. The information content of 0 and 1 are not comparable. If $r \sqsubseteq q$ and $r \sqsubseteq t$, where q, r, and t are states, r effectively represents both q and t. Any property that holds for state r will also hold for all states above it in the lattice—for example, q and t.

STE differs from symbolic simulation in that it provides a mathematically rigorous method for establishing that properties (assertions) of the form antecedent $A \Rightarrow$ consequent C hold for a given simulation model of a circuit. For the test vector shown in Figure 5c, 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 AND s_2 AND … 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 assertion $A \Rightarrow C$ 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.[9] If the implementation is correct, these assertions should hold during symbolic simulation of the RTL design. Assertion $A \Rightarrow C$ 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).

To verify that the RTL design implementation satisfies certain properties, our framework generates behaviors for the intended properties instead of generating the complete reference design. We use the Versys2 symbolic simulator, which uses STE to perform property checking.[3] This 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.[9] Versys2 symbolically simulates the RTL design 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. We then use this counterexample to modify the RTL design.

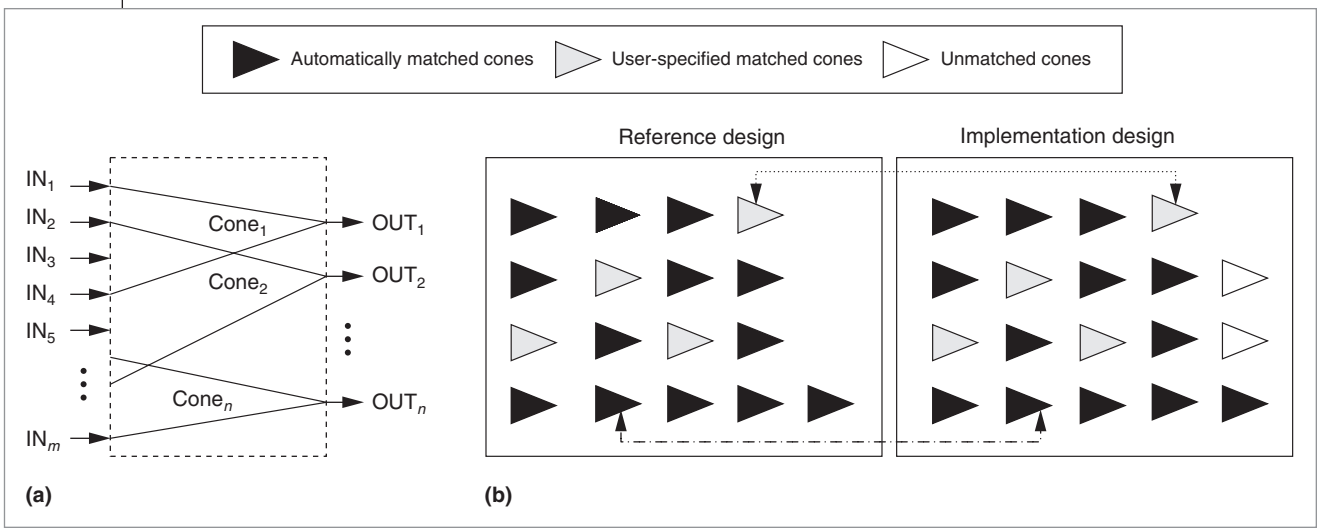


Figure 6. Matching of compare points between a reference design and an implementation design: logic cones in a design block (a), and compare-point matching (b).

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 stages that design transformations affect most. During the reading stage, the equivalence-checking tool reads both design versions, and segments them into manageable sections called logic cones. Logic cones are groups of logic bordered by registers, ports, or black boxes. Figure 6a shows the cones for a typical design block. A logic cone's output border is the *compare point*. For example, in Figure 6a, OUT_1 is the compare point of $Cone_1$.

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.[10] Two types of matching techniques are name based (nonfunction) and function (signature analysis) based. Figure 6b 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 by the name-based techniques, slowing the matching performance. 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 isn't possible to write the design rules, and it's necessary to either manually match compare points for better performance or use 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 functionally nonequivalent.[11] 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.

Our framework generates the complete description of the processor to enable equivalence checking using Synopsys' Formality. The tool reads both the reference design and the implementation, and tries to match the compare points between them. We must map the unmatched compare points manually. The tool 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 (in case of a false negative) or to modify the RTL design implementation.

Experiments

An important aspect of our methodology is that it

can perform both property and equivalence checking. To verify that the RTL design implementation satisfies certain properties, our framework generates behaviors for the intended properties rather than generating the complete reference design. On the other hand, if the generated reference model contains a complete description of the design, our framework performs equivalence checking between the implementation and the generated reference model.

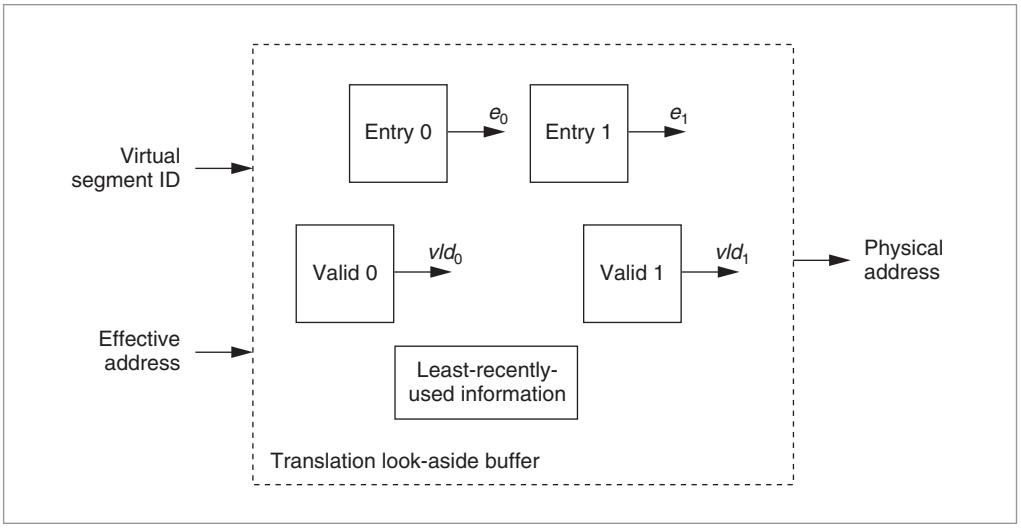


Figure 7. Translation look-aside buffer (TLB) block diagram.

Property checking of an MMU

The memory management unit supports demand-paged virtual memory. The MMU consists of blocks such as segment registers, translation look-aside buffers (TLBs), and block address translation (BAT) arrays. Each memory block contains subblocks. For example, the TLB has three subblocks—entry (data information), valid (information regarding data validity), and least-recently-used information, as Figure 7 shows. The TLB implements each subblock as SRAM. The typical operations in SRAM are read and write. The generated reference model to verify each SRAM cell's write and read properties contains the Verilog code segment shown in Figure 8a and 8b.

We modified the Versys2 configuration file to give the node mapping between the reference model and the implementation. For example, we mapped the reference model's **`wrClk`** to the implementation's **`sramWrClk`**. An interesting feature of this validation approach is that the same properties, without any modification, apply to all MMU memory blocks. However, in each case, the node mapping must be modified.

To verify whether the RTL design correctly implemented the TLB miss detection, our framework generated the Verilog code shown in Figure 8c. The information needed to build this property is directly available from the MMU specification. This property verifies miss detection for a two-way, set-associative TLB. Generating this property for an *n*-way, set-associative TLB would require only a simple extension. In Figure 8c, the TLB block's inputs are **`vsid`** (virtual segment ID) and **`ea`** (effective address), and the output is the physical address. The **`e`** and **`vld`** variables are the outputs from the entry and valid blocks shown in Figure 7.

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

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

assign inp = ({1'b1, vsid[0:23], ea[4:9], ea[10:13]});
assign out0 = ({vld0, e0[0:23], e0[24:29], e0[54:57]});
assign out1 = ({vld1, e1[0:23], e1[24:29], e1[54:57]});
assign hit0 = (inp = = out0);
assign hit1 = (inp = = out1);
assign miss = ~(hit0 | hit1);
(c)
```

Figure 8. Verilog code for verifying an SRAM cell's write property (a), an SRAM cell's read property (b), and miss detection for a two-way, set-associative TLB (c).

Similarly, we generated and validated the property for the BAT array's miss detection. We found several mismatches during property checking. In most cases, the architecture specification document doesn't provide the else condition's value (for example, a signal's default value). As a result, the property description doesn't have the default value of a signal, whereas the signal's implementation always has a definite value. In such cases, symbolic simulation produces mismatches. Consider an SRAM cell's read implementation, shown

in Figure 8b. This implementation assigns `32'b0` to the output signal when the (`rdClk & rdEn`) condition is false. However, the architecture document doesn't specify the value in the default case (condition false). Therefore, the generated property doesn't have the value that caused the mismatch.

To avoid the detection of false negatives, we can update the architecture document to add the values in all cases or impose certain constraints in the Versys2 configuration file. Thus, for the example in Figure 8b, we could set condition (`rdClk & rdEn`) as true in the Versys2 configuration file to avoid the detection of the mismatch just described.

Equivalence checking of the DLX architecture

We successfully applied the proposed methodology in a case study to validate the DLX processor,[12] using the Formality equivalence checker. We chose the DLX processor because it's been well studied in academia and its HDL implementations are available. We obtained a VHDL description of the synthesizable 32-bit RISC DLX from http://www.eda.org/rassp/vhdl/models/processor.html and used it as the implementation. We captured the DLX architecture's structure and behavior using the Expression ADL. Our framework generated the VHDL description from the ADL specification using the method described earlier. The generated VHDL description served as the reference model (specification) for the validation.

Regardless of the implementation style, the equivalence checker can verify the design based on the correct behavior in the reference model. For example, our HDL generation framework generates a 32-bit adder module that uses a carry-look-ahead principle. The equivalence checker verifies that this design is equivalent to the 32-bit adder implementation, which uses a ripple-carry adder principle. Equivalence checking took 4 seconds for this adder example on a 300-MHz Sun Ultra-250 with 1,024 Mbytes of RAM. Similarly, we generated a structural model of a 32 × 32 register file and used it as a reference model to verify the RISC DLX's behavioral register file implementation. In this case, equivalence checking took 432 seconds. The majority of this time (347 seconds) was for the behavioral implementation's elaboration (linking) phase.

Our framework also generated synthesizable RTL for a 32-bit RISC DLX that supports signed operations. To avoid memory explosion, we guided the RTL generation process to have a structure similar to the DLX implementation. Equivalence checking took 397 seconds on the same Sun Ultra-250. The verification process uncovered many mismatches. For example, there was a mismatch in the output data bus at clock cycle 2,500. The analysis revealed that the problem was in the adder's overflow bit. The DLX's ripple-carry adder implementation had incorrectly computed the overflow bit.

Design analysis in our framework is very fast once we discover which module has caused the problem. For example, once we know that the adder is causing the problem, we can verify the adder implementation of the DLX by generating an adder specification (HDL description) from our framework and applying equivalence checking.

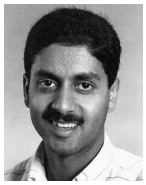
SPECIFICATION-DRIVEN HARDWARE generation and validation of design implementation using equivalence checking has one limitation: The structure of the generated hardware (reference) model needs to be similar to that of the implementation. This requirement is primarily due to the capability of the equivalence 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. ■

References

1. P. Mishra et al., "Automatic Modeling and Validation of Pipeline Specifications Driven by an Architecture Description Language," *Proc. 7th Asia South Pacific Design Automation Conf. / 15th Int'l Conf. VLSI Design* (ASPDAC/VLSI 02), ACM Press, 2002, pp. 458-463.
2. A. Halambi et al., "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability," *Proc. Design Automation and Test in Europe* (DATE 99), IEEE CS Press, 1999, pp. 485-490.
3. N. Krishnamurthy et al., "Design and Development Paradigm for Industrial Formal Verification Tools," *IEEE Design & Test*, vol. 18, no. 4, July-Aug. 2001, pp. 26-35.
4. P. Mishra, N. Dutt, and A. Nicolau, "Functional Abstraction Driven Design Space Exploration of Heterogeneous Programmable Architectures," *Proc. Int'l Symp. System*

Synthesis (ISSS 01), ACM Press, 2001, pp. 256-261.
5. P. Mishra, A. Kejariwal, and N. Dutt, "Rapid Exploration of Pipelined Processors through Automatic Generation of Synthesizable RTL Models," *Proc. 14th Int'l Workshop Rapid System Prototyping* (RSP 03), IEEE CS Press, 2003, pp. 226-232.
6. H. Chockler et al., "A Practical Approach to Coverage in Model Checking," *Proc. 13th Conf. Computer Aided Verification* (CAV 01), Lecture Notes in Computer Science 2102, Springer-Verlag, 2001, pp. 66-78.
7. R.E. Bryant, "Symbolic Simulation—Techniques and Applications," *Proc. 27th ACM/IEEE Design Automation Conf.* (DAC 90), IEEE Press, 1990, pp. 517-521.
8. C. Seger and R. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, Mar. 1995, pp. 147-189.
9. L. Wang, M. Abadir, and N. Krishnamurthy, "Automatic Generation of Assertions for Formal Verification of PowerPC Microprocessor Arrays Using Symbolic Trajectory Evaluation," *Proc. 35th Design Automation Conf.* (DAC 98), ACM Press, 1998, pp. 534-537.
10. D. Anastasakis et al., "A Practical and Efficient Method for Compare-Point Matching," *Proc. 39th Design Automation Conf.* (DAC 02), ACM Press, 2002, pp. 305-310.
11. J. Marques-Silva and T. Glass, "Combinational Equivalence Checking Using Satisfiability and Recursive Learning," *Proc. Design Automation and Test in Europe* (DATE 99), IEEE CS Press, 1999, pp. 145-149.
12. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

Prabhat Mishra is a PhD candidate at the University of California, Irvine. His research interests include system-level modeling, architecture synthesis, design space exploration, and validation of embedded systems. Mishra has a BE in computer science from Jadavpur University, India, and an MTech in computer science from the Indian Institute of Technology, Kharagpur. He is a student member of the IEEE and ACM SIGDA.



Nikil Dutt is a professor in the School of Information and Computer Science at the University of California, Irvine. His research interests include embedded-systems design automation, computer architecture, optimizing compilers, system specification techniques, and distributed embedded systems. Dutt has a BE in mechanical engineering from the Birla Institute of Technology and Science, India; an MS in computer science from Penn State University; and a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a senior member of the IEEE, serves on the advisory boards of ACM SIGBED and ACM SIGDA, and is vice chair of International Federation for Information Processing (IFIP) Working Group 10.5.

Narayanan Krishnamurthy is a principal software staff engineer and CAD tools/methodology developer in the High Performance Tools and Methodology Group at Motorola's PowerPC Design Center in Austin, Texas. His research interests include CAD for VLSI and digital systems, formal methods and verification, hardware-software design methodologies for microprocessors and SoCs, and software implementation techniques for application in digital and analog design automation. Krishnamurthy has a BTech in instrumentation engineering from the Indian Institute of Technology, Kharagpur, and an MS and PhD in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE.

The biography of **Magdy S. Abadir** appears on p. 81 of this issue.

■ Direct questions and comments about this article to Prabhat Mishra, Center for Embedded Computer Systems, 444 Computer Science Bldg., University of California, Irvine, CA 92697; pmishra@cecs.uci.edu.