Validating PowerPC Microprocessor Custom Memories

Narayanan Krishnamurthy

Andrew K. Martin

Magdy S. Abadir
Motorola

Jacob A. Abraham
University of Texas at Austin

Due to the high cost of correcting errors in a final product, there is a growing impetus in industry towards methodologies that can yield correct designs in the first manufacturing run. Design validation methodologies that combine simulation techniques with formal reasoning can be effective in ensuring correct operation of software and hardware systems. We show why simulation is necessary to complement formal mathematical reasoning in verifying certain classes of custom designed circuits. We present a validation methodology for PowerPC custom memories based on symbolic simulation.

■ **DESIGN IS AN ITERATION** of specification and implementation phases, performed either top-down or bottom-up before resulting in a product. Validation is performed at each of these design phases until the final manufacturing stage. Correctness of an implementation is not an autonomous concept, but rather a relation between a specification and an implementation. Design validation techniques attempt to establish a relation between the two entities.

There are a number of approaches to design validation. No matter how they are categorized, the ultimate objective of these different approaches is to ensure that the final product satisfies customer requirements and does not fail during operation. Design validation techniques can be broadly categorized into simulation-based approaches and formal techniques. Due to the complexity of modern designs, validation using only traditional scalar simulation cannot be exhaustive and has proved to be ineffective in exposing hard-to-find bugs. This is because of the combinatorial complexity of the number of states and input sequences possible for a nontrivial design. Formal techniques do an exhaustive analysis of the design but can check only small designs completely. As the sizes and complexity of the designs keep growing, formal validation techniques suffer from the state explosion problem. Unless drastic innovations in data structures and proof systems come about, validation methodologies purely based on formal methods are currently neither feasible nor economical. Symbolic simulation has proved to be an efficient technique, bridging the gap between traditional simulation and full-fledged formal verification.

0740-7475/00/$10.00 © 2000 IEEE

Symbolic simulation has been used to verify embedded on-chip memories of commercial microprocessors. Prior work focused on establishing that functional properties held for two different representations.[1–2] No notion of completeness was established by their methodology. To the best of our knowledge, this is the first time that a rigorous symbolic simulation methodology has been used to prove the correctness of switch-level models with respect to the register-transfer-level (RTL) models for all modes of operation (functional and nonfunctional).

This article shows why symbolic simulation is necessary and how it can be used in an effective validation and design analysis methodology. We survey the current state of the art in validation techniques and compare them with symbolic simulation. A key established result is that, for certain types of circuits, simulation is necessary and current Boolean equivalence-checking technology is fundamentally inadequate.[3] We also show how symbolic simulation can be used as part of a validation methodology for proving custom memory implementations correct with respect to a RTL specification. Finally, we present validation results on the latest PowerPC microprocessor and establish the need for a symbolic simulation validation methodology.

Validation Techniques

Errors uncovered by different validation approaches are classified as design or implementation errors. Design errors occur when a specification itself contains errors. Implementation errors are errors that occur in the realization (implementation) of the specification. Here, the specification is assumed to encode the correct set of behaviors of the design while the implementation is checked for errors. Our methodology seeks to uncover implementation errors.

Any validation approach is prone to two types of errors. A false-positive error occurs when the validation technique predicts that an incorrect implementation is correct. A false-negative error occurs when the validation technique predicts that a correct implementation is incorrect. Clearly, false positives must be avoided, and false negatives must be minimized. The desire to produce more reliable products free of design errors and the need to reduce time to market (for the same product quality) has resulted in a number of validation methodologies.

Simulation

The methodology used in the microprocessor and digital systems industry to validate designs has traditionally been simulation. Farms of workstations run simulations for months on the RTL and switch-level models. The product is taped out when the rate of discovering errors has dropped below a predetermined value. Due to the large combination of possible initial state and input sequences, it is not feasible to cover all cases exhaustively. Nonetheless, simulation is relied upon to catch the last (or thought to be last) pre-tape-out bugs simply because there are no other techniques that can handle these large designs and fit into existing design methodologies seamlessly.

With the introduction of ternary value *X*, it has been possible to simulate larger designs with fewer simulation vectors, resulting in fewer simulation runs. Ternary valued logic simulators use *X* as an unknown digital value in addition to the binary values 0 or 1. A ternary valued simulator can verify circuit behavior for many possible input and initial state combinations. If a simulation of a vector containing *X*s yields 0 or 1 on a node in the circuit, it is guaranteed that the value on the node will not change if the *X*s in the vector were replaced by any combination of 0's and 1's. Bryant proved the correctness of a static RAM design by logic simulation using ternary values.[4] For performance reasons, most ternary simulators are pessimistic and will produce an *X* even though it can be proved that the circuit produces a 0 or a 1 in all cases. As a result, a ternary logic verifier can produce many false-negatives, and the debug and re-runs can be quite time-consuming. However, this does not compromise the rigor of the verification.

Formal techniques

Formal validation establishes a mathematical relation between two different representations of a system. The nature of this mathematical relation varies according to the kind of validation approach. Formal techniques, when they are applicable, can establish universal properties about the design independent of any particular set of inputs.

Model checking is a verification technique in which properties are expressed in a temporal logic, and the design is modeled as a state-transition system.[5] A design is verified against such a temporal logic specification by proving that the design is a model of the specification formula. The temporal logic specification describes the ordering of events in time without introducing time explicitly.

This approach has a few drawbacks. Most realistic systems are still too large to be model checked. This is primarily due to the size of the transition relations that need to be built. Moreover, for certain classes of circuits (as shown later) an erroneous transition relation would be built. To avoid this, the model checker would have to work with an extracted state transition system whose transitions occur according to the timing of the underlying switch-level model, resulting in an explosion in the number of states.

Theorem proving is a technique that works within a framework of logic, with axioms representing known truths about the behavior of hardware and theorems representing newly inferred properties of the system's behavior.[6] To be amenable to mathematical proof, both the specification of the system and the model of its actual behavior need to be stated mathematically. Certain formal procedures in a specific deductive system are then used to operate on the symbolic statements. The main disadvantage of theorem proving is that complete automation is extremely hard and it may be necessary to guide the prover through a series of lemmas. Current hardware description languages such as Verilog do not have formalized semantics and rely on the user to embed their semantics in the proof-system's logic. It is not clear whether such proof systems can fit smoothly into existing methodologies for custom memory validation.

Language containment and trace algebra techniques are based on inclusion relations between sets of traces or sets of strings, where these are used to model system behavior. These techniques have primarily been applied to RTL and higher level behavioral models of designs, but have had limited success with transistor-level circuits. Another approach is forming the product machine between the specification and the implementation, and then searching the state space for reachable states at which the outputs differ. Here, the equivalence criterion is input-output behavior with each machine starting in a known state. As a result of the state explosion of the resulting product state machine graph, this technique can only be used on relatively small designs.

Boolean equivalence checking tools are used routinely to verify equivalence between RTL, gate-, and switch-level models of standard library cells and custom-designed circuits. These tools operate by extracting a Boolean function, representing cones of logic from these descriptions (which are at different levels of abstraction) and then comparing their stable outputs. Singh et al. have done equivalence checking by extracting RTL models from transistor netlists and then doing the comparison.[7] They require information about the clocking scheme and rely on manipulating the simulation relation obtained to derive the stable behavior of the circuit. However, current Boolean function extraction techniques using functional composition to extract logic functions, or techniques that try to derive the stable behavior, will not suffice for a certain class of self-timed custom circuits.

Symbolic Simulation

Symbolic simulation combines traditional simulation with formal symbolic manipulation.[8,9] 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 of a traditional simulator.

Researchers at IBM first introduced symbolic simulation to reason about properties of circuits described at the RTL.[10,11] Their approach drew on techniques developed for reasoning about software by symbolic execution. Darringer showed how to apply symbolic execution to combinational logic verification, by building a gate-level simulator and then simplifying the equations it implemented.[11] The D-algorithm for test generation is a different form of symbolic simulation

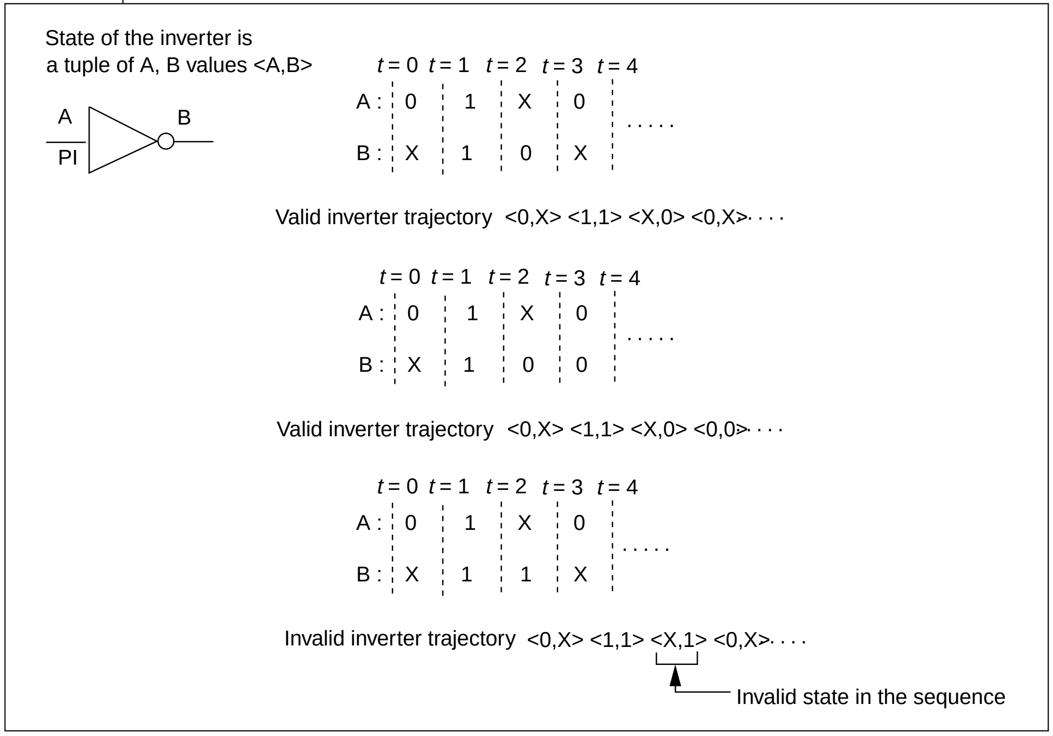

State of the inverter is
a tuple of A, B values <A,B>
A
B
PI
t = 0 t = 1 t = 2 t = 3 t = 4
A : 0 1 X 0
B : X 1 0 X
Valid inverter trajectory <0,X> <1,1> <X,0> <0,X> . . . .
A : 0 1 X 0
B : X 1 0 0
Valid inverter trajectory <0,X> <1,1> <X,0> <0,0> . . . .
A : 0 1 X 0
B : X 1 1 X
Invalid inverter trajectory <0,X> <1,1> <X,1> <0,X> . . . .
Invalid state in the sequence

Figure 1. Valid and invalid inverter trajectories.

where the functions of the basic logic gates are extended to operate over a larger value domain 0, 1, D, $\overline{\text{D}}$, and X.

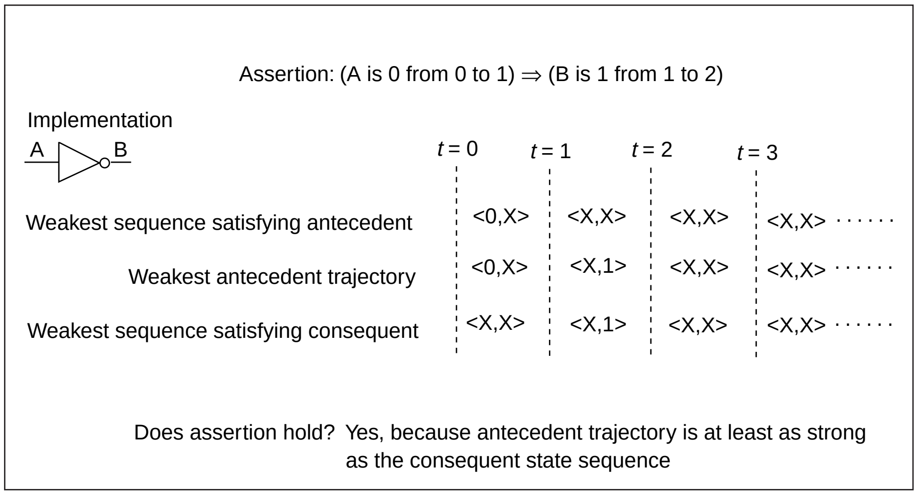
The success of this early work was limited by the weakness of the symbolic manipulation methods. With the advent of BDDs, the technique became much more practical.[12] Providing a canonical representation for Boolean functions, BDDs enabled the implementation of an efficient event-driven logic simulator that operated over a symbolic domain. By encoding a model's finite domain using a Boolean encoding, it is possible to symbolically simulate the model using BDDs. Bryant's formal state transition model for a ternary system and his work in the area of switch-level and memory symbolic simulation, and Seger's work on symbolic trajectory evaluation renewed further interest in symbolic execution.[4,9,13,14]

Symbolic Trajectory Evaluation. Symbolic trajectory evaluation (STE) is a modified form of symbolic simulation that operates over the quaternary logic domain 0, 1, X, and $\top$.[13,14] A state of the circuit is defined as the set of all node values at a particular time instant. The value domain is partially ordered and forms a complete lattice, $X \sqsubseteq 0$ indicates X has less information than 0, or X is weaker than 0. 0 is neither weaker nor stronger than 1 since their information contents are incomparable. If $r \sqsubseteq q$ and $r \sqsubseteq t$, we can think of r as representing both q and t. Any property that holds for a state such as 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 of the form antecedent (A) $\Rightarrow$ consequent (C) hold for a given simulation model of a circuit. Circuit state holders are initialized with symbolic values specified by the antecedent. The model is then simulated, typically for one or two clock cycles, while driving the inputs with symbolic values during simulation. The resulting values appearing on selected internal nodes and primary outputs are compared with the expected values expressed in the consequent. In the more general case, 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 state. Intuitively, a trajectory is a state sequence constrained by the system's next-state function.

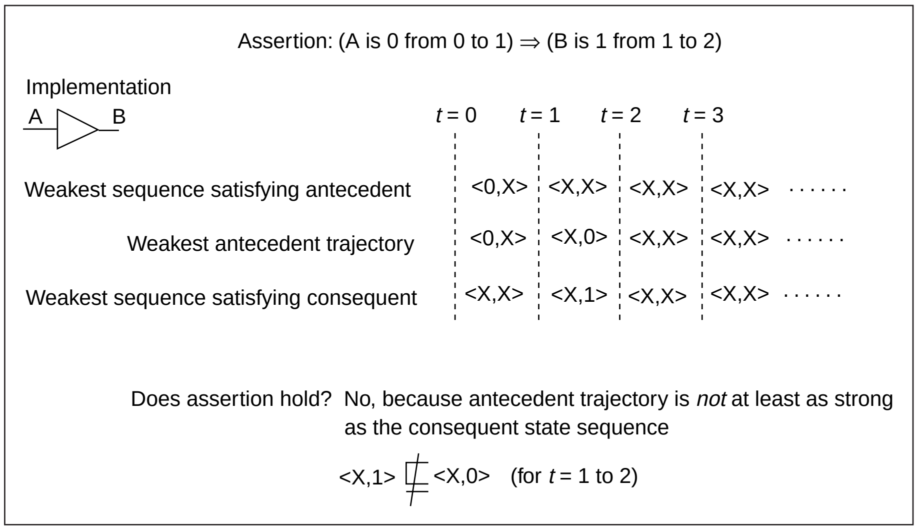
Consider an antecedent for an inverter stated as node A is 0 from 0 to 1, is 1 from 1 to 2, and is 0 from 3 to 4. Valid trajectories for the antecedent are the first and second state sequences shown in Figure 1. The third state sequence is an invalid trajectory. The value on node B is 1 instead of the correct value 0.

A successful simulation of $A \Rightarrow C$ establishes that any sequence of assignments of values to circuit nodes that is both consistent with the circuit behavior and consistent with antecedent A is also consistent with consequent *C*. For example, consider a specification for an inverter. An assertion that could be checked on an implementation for an inverter is "If node *A* is 0 from 0 to 1, then *B* is 1 from 1 to 2." If the implementation is correct, then the weakest antecedent trajectory that the circuit goes through is that shown in Figure 2. The trajectory that the circuit goes through is at least as strong as the weakest sequence satisfying the consequent, and the assertion holds. If the implementation is incorrect and a buffer is implemented instead of an inverter, the weakest antecedent trajectory that the circuit goes through is that shown in Figure 3. The trajectory that the circuit goes through is not at least as strong as the consequent's weakest state sequence, and the assertion fails.[14]



Assertion: (A is 0 from 0 to 1) ⇒ (B is 1 from 1 to 2)
Implementation
A
B
t = 0
t = 1
t = 2
t = 3
Weakest sequence satisfying antecedent
<0,X> <X,X> <X,X> <X,X> ······
Weakest antecedent trajectory
<0,X> <X,1> <X,X> <X,X> ······
Weakest sequence satisfying consequent
<X,X> <X,1> <X,X> <X,X> ······
Does assertion hold? Yes, because antecedent trajectory is at least as strong as the consequent state sequence

Figure 2. Trajectory for correct implementation.



Assertion: (A is 0 from 0 to 1) ⇒ (B is 1 from 1 to 2)
Implementation
A
B
t = 0
t = 1
t = 2
t = 3
Weakest sequence satisfying antecedent
<0,X> <X,X> <X,X> <X,X> ······
Weakest antecedent trajectory
<0,X> <X,0> <X,X> <X,X> ······
Weakest sequence satisfying consequent
<X,X> <X,1> <X,X> <X,X> ······
Does assertion hold? No, because antecedent trajectory is not at least as strong as the consequent state sequence
<X,1> ⋢ <X,0> (for t = 1 to 2)

Figure 3. Trajectory for incorrect implementation.

Benefits of symbolic simulation

Symbolic simulation has proved to be a practical and viable technique for validating behavioral, RTL, and switch-level models.[1,2,8-11,13-17] It combines switch-level accuracy with formalized reasoning to infer properties about the system.

The two main advantages of symbolic trajectory evaluation are that no monolithic transition relation of the design is built, and the symbolic simulation engine is sensitive to the dynamic behavior of the circuit. In most realistic designs, the transition relations required for other formal verification techniques such as model checking or language containment cannot be built. This is due to the large number of state-holding elements. By keeping time explicit and the temporal logic simple, the logic is less expressive but is more intuitive to phrase properties to be checked and debugged on the schematic. At any point during the symbolic simulation, only the present state and the previous state are kept in memory.

From an economic perspective, validation and analysis using symbolic simulation can fit neatly into existing design and validation methodologies. The potential benefits of improved product quality (fewer errors), reduced time to market, and lower validation costs (fewer highly skilled people) seem to warrant further investment and effort in this area. Whereas other approaches discussed earlier do not meet all three criteria, we hope symbolic simulation will fulfill the three metrics.

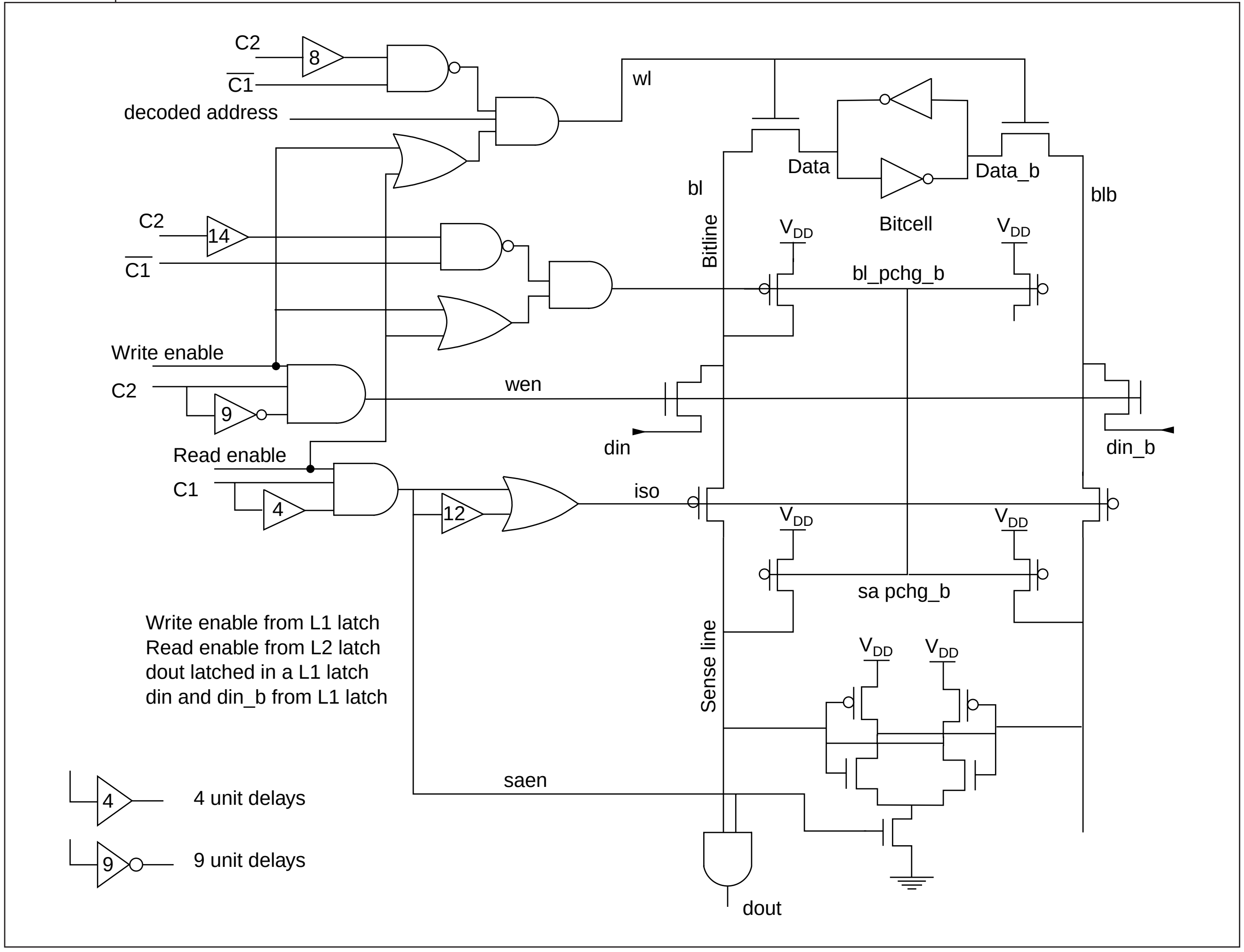


Figure 4. Read/write control logic for a bit cell.

We have two main technical objectives for symbolic simulation. First, we are interested in finding bugs and analyzing circuit-related problems such as races and glitches. Assuming a potential cost for each bug, it is easy to see the payoff for symbolic simulation. Secondly, we want to prove that the specification has been realized correctly. In cases where the RTL or behavioral model is annotated with timing (such as in event-driven simulation), proving correctness of implementation using symbolic simulation is easier than using other techniques. Symbolic simulation can benefit directly from the enhancements to the switch-level models and can utilize better delay models. In this work, we show that a symbolic simulation methodology can address both the economic and technical concerns mentioned earlier.

Do we have to simulate?

Designers are relying more and more on pulsed clocks and are incorporating self-timed circuit structures in their designs. Static analysis of such designs is no longer adequate.

Self-timed implementations

Figure 4 shows one bit cell of a custom memory and its associated read and write control logic. The memory has a number of carefully designed timing chains that ensure the correct temporal relationships between precharge, isolate, sense-amp enable, word-line assertion, and write enable operations. To simplify the presentation, each gate in the control logic is assumed to have a unit delay. The numbers on the inverters and buffers indicate the number of unit delays of the inverter buffer chain in that path. The read and write timing diagram for these signals is shown in Figure 5. For the purpose of illustrating the control signal sequence, the timing diagram takes into account only the delays introduced by the inverters and buffers.[15]

A static Boolean equivalence checker will not be able to verify such a circuit. For example, in Figure 4, the extracted static Boolean function of the logic cone feeding the wen signal of the write pass transistors is 0 even though the circuits write operation functions correctly. Static analysis of the bit cell node data generates a model that cannot be written to. Current Boolean equivalence-checking tools cannot work with such self-timed custom implementations. Being sensitive to the dynamic behavior of the circuits, symbolic simulation allows the validation of such self-timed and edge-triggered circuits.

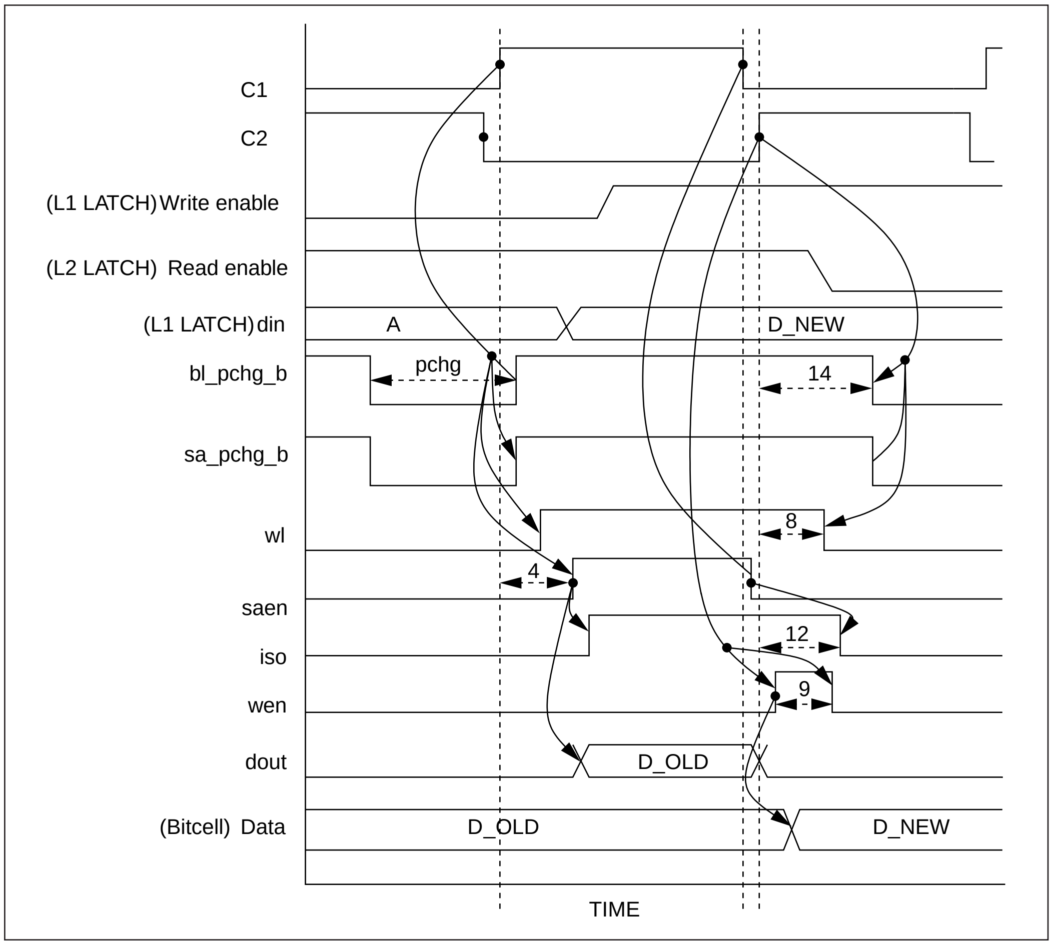


Figure 5. Read and write control sequence.

Table 1. Rise and fall delays for AND dynamic logic.

Primary input	Rise delay	Fall delay
A	5	2
B	4	4

Why accurate event orders are important

Consider the dynamic logic circuit and its RTL specification, shown in Figure 6. The circuit computes function CLK & *A* & *B*. Assume that the primary inputs are fed by dynamic logic that is precharged using CLK. Primary input *A* is precharged to 0, while *B* is precharged to 1. The primary input B is precharged to 1 to enable a faster rise time at the output during evaluation. A static Boolean equivalence-checking tool would verify that the dynamic circuit implements the RTL specification.

Now consider delay values for the primary inputs, as shown in Table 1. Our symbolic simulation tool will verify that the circuit implements the RTL specification,

RTL specification

```
module dyn_and (CLK, A, B, O);
  assign O = CLK & A & B
endmodule;
```

SCH implementation

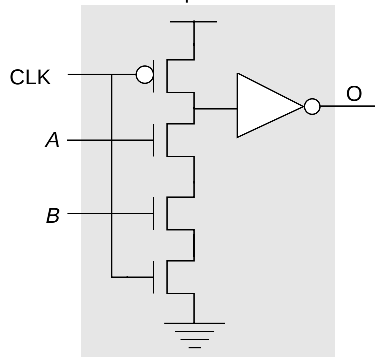


Figure 6. Dynamic logic circuit.

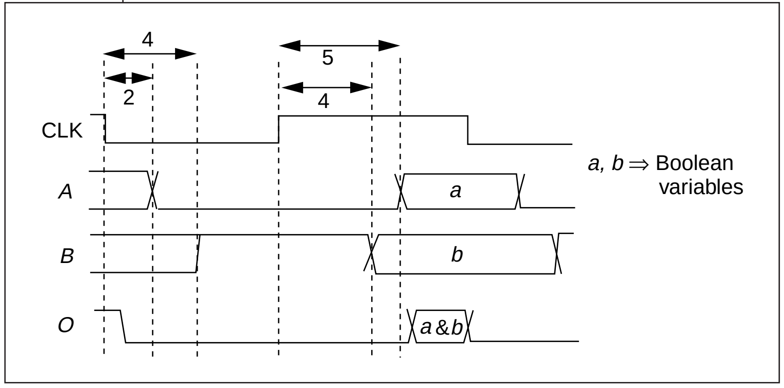


Figure 7. Timing diagram for user-specified rise/fall delays.

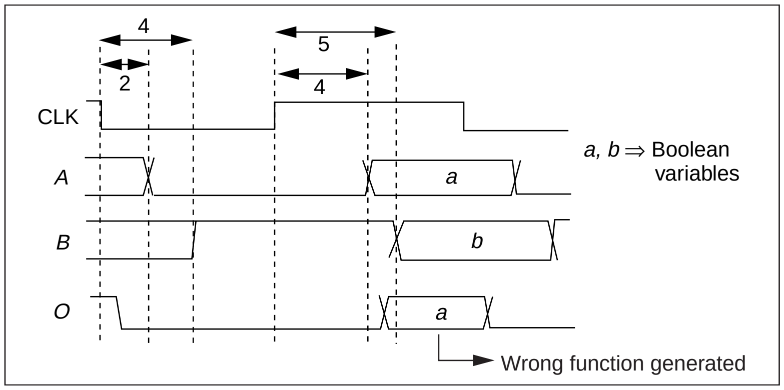
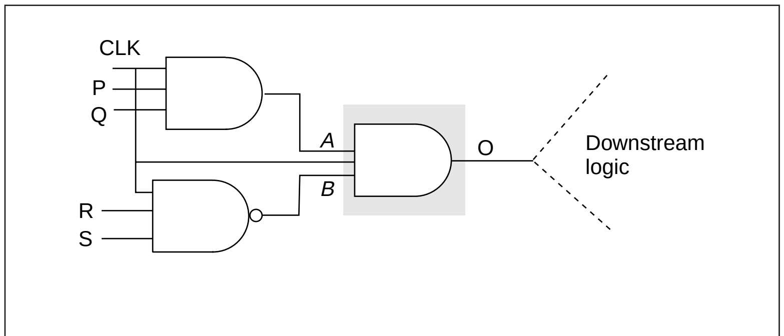


Figure 8. Timing diagram for modified rise/fall delays.



RTL specification

```
module (CLK, P, Q, R, S, .....);
   assign O = CLK ? P & Q & (~R I ~S) : 1'b0 '
   …
   …
end modules;
```

Figure 9. Dynamic logic instantiated in a combinational circuit.

as depicted in Figure 7. Consider a different set of delays, as shown in Table 2. The rise delay for *A* is now faster at 4 units, and the fall delay is slower for *B* at 5 units. Although the change in delays is a single unit, the dynamic circuit no longer implements the RTL specification, and the verification fails.

Table 2. Modified rise and fall delays for AND dynamic logic.

Primary input	Rise delay	Fall delay
A	4	2
B	4	5

The symbolic simulation tool will now produce the waveforms shown in Figure 8. Consider the circuit in Figure 6 instantiated in a larger design, as shown in Figure 9. The Boolean function computed at the outputs driven by *O* is actually dependent on the delays at internal nets *A* and *B*. Using a static Boolean equivalence checker is akin to validating the design using zero-delay simulation. This could result in a false positive since there could be real event orders that were not considered during the verification. By symbolically simulating the circuit with delays, we prove that the circuit implements the specification under a set of specified delays. This enables a more accurate verification and allows the designer to analyze the circuit for different event orders. Most design methodologies will have circuit design rules to ensure that such circuits with races are not designed. However, there will be circumstances where violation of those rules is warranted. Any validation technique must be capable of addressing these concerns.

Validation Using Symbolic Simulation

Our validation involves two different representations, the specification and the implementation, of the real system to be built. The specification is at the RTL. It is a finite-state cycle-accurate description of the required behavior of the circuit. The model is evaluated either once or twice every clock cycle. Typically, the RTL is written in a hardware description language such as Verilog.

The implementation is a netlist of transistors that is custom built. A switch-level simulation model is extracted from the transistor netlist.[18] Many important detailed effects arising from cir-

RTL model

```
module (C1, C2, Ain, Bin, Cout);
input Ain, Bin, C1, C2;
output Cout;
wire and_out;
reg A.L1, A.L2, B.L1, B.L2, C.L1, C.L2;
assign and_out = A.L2 & B.L2;
always @(C1 or Ain or Bin or and_out)
   if (C1)
      A.L1 = Ain;  B.L1 = Bin;  C.L1 = and_out;
always @(C2 or A.L1 or B.L1 or C.L1)
   if (C2)
      A.L2 = A.L1;  B.L2 = B.L1;  C.L2 = C.L1;
assign Cout = C.L2
```

Figure 10. RTL design representations.

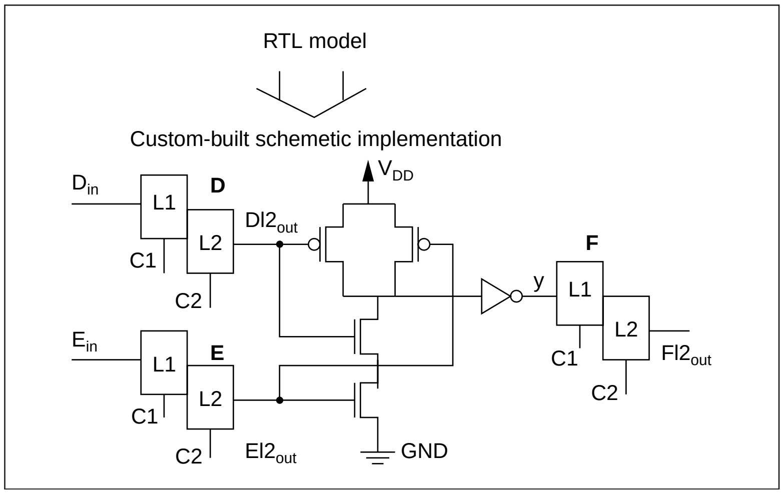


Figure 11. Custom schematic representation of design.

cuit structures such as ratioed and precharged logic, bidirectional pass transistors, and stored charge are accurately modeled. Node voltages are represented using the STE quaternary valued logic. The switch-level simulation model supports different rise and fall delay timing models and evaluates multiple times during a single clock cycle.

Illustration of an STE validation

For illustration, we present an example of an RTL compared with schematic validation using symbolic simulation for a simple design. Consider the RTL design shown in Figure 10. The RTL represents a two-phase clocked design that is to be custom built. The design models an AND gate driven by two slave latches A.L2 and B.L2 and whose output is latched in third master latch C.L1. With this RTL as a specification, a custom schematic of the RTL design is built, as shown in Figure 11. For example, the A.L1 master latch in the RTL corresponds to the D.L1 master latch in the schematic.

A typical schematic implementation of a master-slave latch pair is shown in Figure 12. The master latch consists of internal state-holding node L1 SN, data input D_{IN}, clock input C1, and data output L1 $_{OUT}$. To verify such a latch in STE, we set up antecedent A and consequent C, as illustrated in Figure 13 (next page), and then use STE to prove $A \Rightarrow C$. The gray-shaded portions represent places where no assumption is being made in the antecedent or no check is being performed in the consequent. A similar assertion would be run for the slave latch to verify the complete master-slave pair. This verification has only proved that the latches are correct with respect to the informal property we

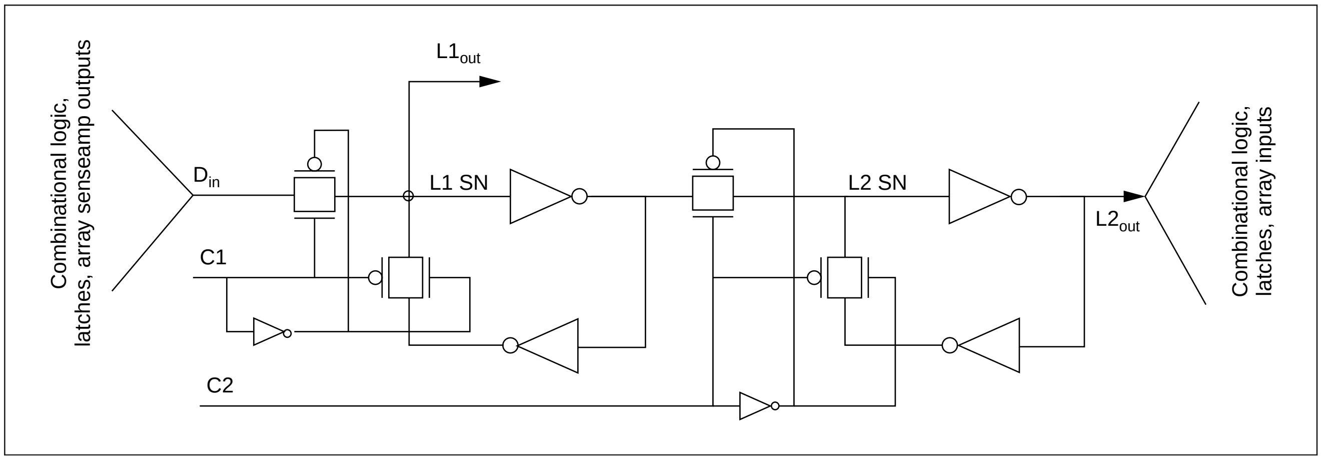

Combinational logic, latches, array senseamp outputs
$L1_{out}$
D_{in}
L1 SN
C1
C2
L2 SN
$L2_{out}$
Combinational logic, latches, array inputs

Figure 12. A master-slave latch.

have specified for a latch. We have not yet proved that schematic latch pair D behaves as predicted by the RTL. To achieve this, we derive the properties from the RTL automatically. For example, the antecedent and consequent for the schematic latch D.L2 are derived from corresponding RTL latch A.L2. This assertion is then mapped onto the schematic and given to the symbolic trajectory evaluator.

Does the schematic realize the RTL specification?

The design is first partitioned into a set of checkpoints. These checkpoints are nodes, such as latches and primary outputs, in the RTL about which properties can be stated and where states can be compared. The RTL encodes a set of next-state functions for the checkpoints in the design. These functions are captured as STE assertions that are then checked to hold on the schematic.

In Figure 10, the checkpoints are the storage nodes A.L1, A.L2, B.L1, B.L2, C.L1, C.L2, and the primary output C_{out}. Each checkpoint has its own assertion. For example, the next-state function, in terms of RTL node names, for latch C.L1 automatically derived from the RTL model is as follows:

$$\texttt{C.L1 = (C1\&(A.L2\&B.L2))V(}\overline{\texttt{C1}}\texttt{\&C.L1)}$$

This function forms the value part of the consequent for the assertion associated with it. The antecedent for the C.L1 assertion would involve driving inputs C1 and C2, and the checkpoints A.L2 and B.L2 with symbolic variables *c*1, *c*2, *a*, and *b* respectively. Having obtained the assertion for C.L1, we now have to verify that it holds on the schematic shown in Figure 11. The RTL assertion is mapped to a schematic assertion by mapping nodes and values in the RTL domain to nodes and waveforms in the schematic domain. The C.L1 latch RTL assertion is mapped into a schematic assertion, as shown in Figure 14. The schematic mapped assertion for the C.L1 latch involves

1. Identifying a F.L1SN corresponding checkpoint in the schematic where states can be compared.

2. Defining the timing window (t7, t8)for the consequent check of the C.L1 latch next-state function. This time window is also used in the antecedent when logic downstream of the C.L1 latch is checked.

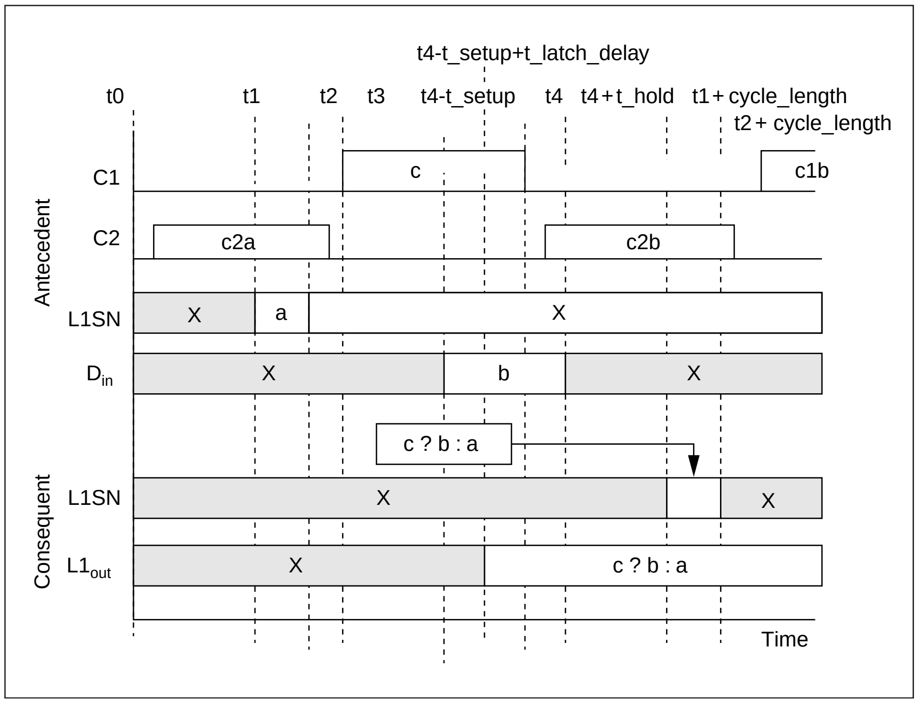


Figure 13. Latch verification using symbolic simulation.

The $D.L2_{OUT}$, $E.L2_{OUT}$ and F.L1SN latch nodes are initialized with independent variables a, b and c respectively, precisely at the time there is a stable feedback loop established in the latches after the corresponding clock goes to 0. The L2 latches in the schematic are initialized from t4 to t5 when the C2 clock is 0. The F.L1 latch is checked precisely one clock cycle after the time it was set up. The timing windows are defined by considering the setup and hold times of the latches.

Assertions are mapped for all checkpoints in the design. The schematic in Figure 11 is a correct implementation of the RTL in Figure 10 with respect to that mapping only if the conjunction of all the checkpoint assertions is satisfied by the schematic. The design decomposition into checkpoints and the definition of the mapping functions are carefully done so that the final verification result can be established by composing the results of the individual checkpoint assertions.

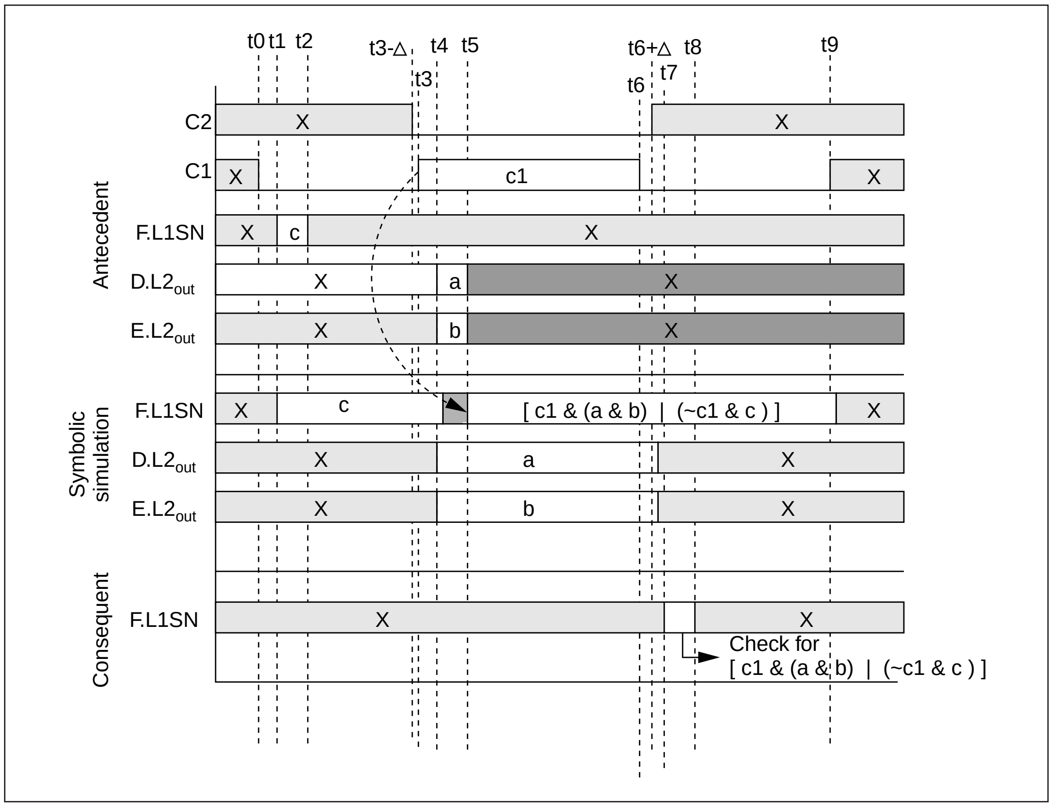


Figure 14. Antecedent and consequent pair for schematic.

Validation Methodology for Custom Memories

All custom memories on the latest PowerPC microprocessor were validated using STE. A typical custom memory consists of an array of bit cell storage nodes, as shown in Figure 15. The verification methodology, as shown in the later Figure 16, involves the first step of partitioning the design into checkpoints and then state mapping between the RTL and the schematic. In cases where the state machine encoding differed between the RTL and the schematic, the exact relationship between the nodes in the RTL and schematic was identified and specified. Next, the switch-level model is augmented with delay information, with the resulting event ordering between critical signals being consistent with Spice (simulation program with integrated circuit emphasis) simulations. The RTL model is then analyzed to obtain the assertions.

While verifying checkpoints other than the array bit cells, a schematic model without the array core is created. This reduces model load and debug times for assertion failures. Once all the primary outputs and latches pass, the array core is verified. Initial runs are done on a schematic model with only a few numbers of bit cells. Typically, a row or column of bit cells is selected. Once they pass, they are run on the full-size array. In cases, where assertions could not be run on the full-size array due to BDD blow-up, different size models were created and then verified using an orthogonal set of assertions. For more details on each of these steps, see Krishnamurthy et al.[16]

Completed Work and Results

Most earlier work focused on using STE to verify equivalence by proving that functional properties held on both the RTL and schematic.[1,2] These approaches relied on the user's knowledge of the RTL to come up with a complete set of assertions. However, bugs related to nonfunctional modes of operation such as debug and scan can be missed by such methods. The incorrect operation of these nonfunctional modes has a serious impact on the debug and test capability of these chips, thereby affecting profitability. We proposed a technique to generate the assertions from the RTL model automatically.[17] The automatically generated assertion set completely characterizes the RTL. The transistor schematic is then

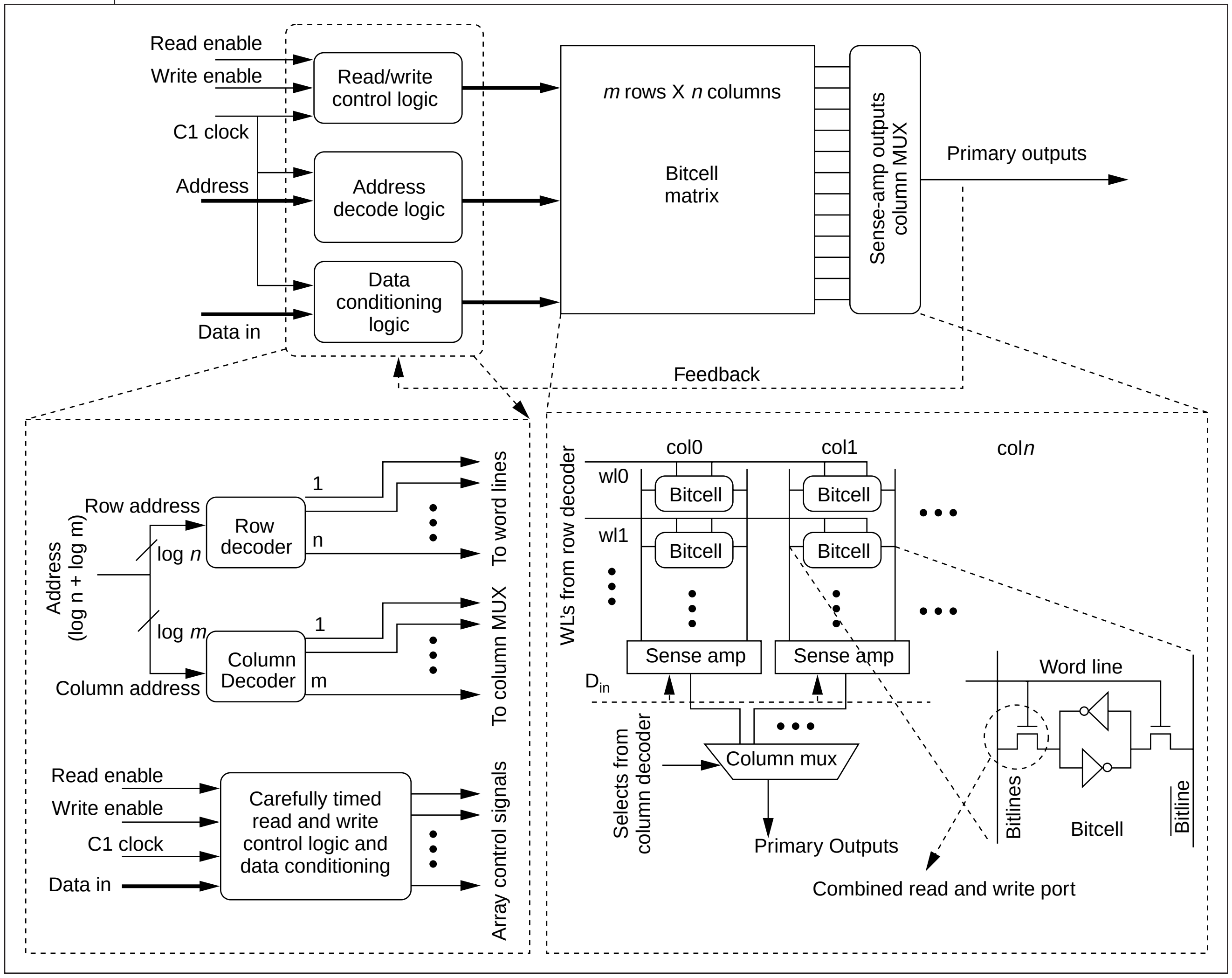


Figure 15. Custom memory.

checked to see if it satisfies this assertion set. By focusing on all modes of operation, our methodology is more rigorous and closer to equivalence checking than any of the earlier approaches.

Previous authors have attributed complex timing, multiple clock phases, complex sequential control logic, and the large number of state-holding elements (state explosion) as their reason for using a simulation-based methodology for equivalence checking custom memories.[1,2,17] While these reasons are valid, the fundamental reason for using a simulation-based approach for custom memories is the prevalence of a certain class of custom-designed self-timed logic structures for which no Boolean function can be extracted using existing techniques.

We also showed why accurate event orders are necessary for validation, giving an example of how symbolic simulation has the added benefit of uncovering potential races. Since it is sensitive to the dynamic behavior of the circuit under analysis, STE can verify such circuits that are prevalent in high-performance custom memory designs.

Results

Every custom array schematic in the latest PowerPC microprocessor designed at the Somerset PowerPC Design Center (Motorola, Austin, Texas) was validated with respect to its RTL specification using our methodology.

Table 3 lists some of the arrays that were verified. The control logic transistors are all the

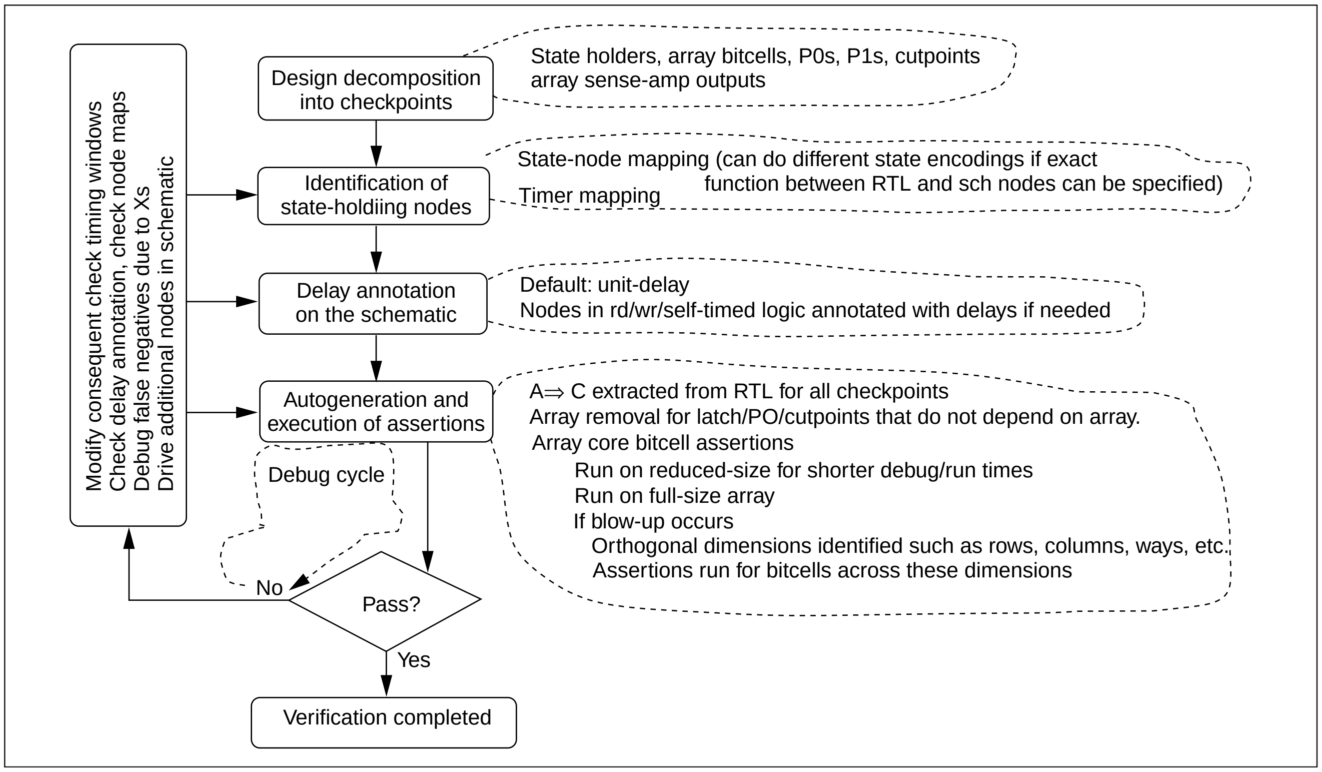

Modify consequent check timing windows
Check delay annotation, check node maps
Debug false negatives due to Xs
Drive additional nodes in schematic
Design decomposition into checkpoints
State holders, array bitcells, P0s, P1s, cutpoints array sense-amp outputs
Identification of state-holdiing nodes
State-node mapping (can do different state encodings if exact function between RTL and sch nodes can be specified)
Timer mapping
Delay annotation on the schematic
Default: unit-delay
Nodes in rd/wr/self-timed logic annotated with delays if needed
Autogeneration and execution of assertions
A⇒ C extracted from RTL for all checkpoints
Array removal for latch/PO/cutpoints that do not depend on array.
Array core bitcell assertions
Run on reduced-size for shorter debug/run times
Run on full-size array
If blow-up occurs
Orthogonal dimensions identified such as rows, columns, ways, etc.
Assertions run for bitcells across these dimensions
Debug cycle
No
Pass?
Yes
Verification completed

Figure 16. Verification methodology.

Table 3. Arrays that were verified.

Array block	Bit cells	Latches	Control logic transistors	Assertions runtime (hrs)
A	73,728	1,346	69,000	11–12
B	24,576	1,612	87,500	15–16
C	3,968	357	31,000	15–16
D	24,576	935	44,000	16–17
E	131,072	330	177,500	19–20
F	88,704	445	39,500	12–13
G	50,688	706	156,000	11–12
H	71,680	276	60,000	15–16
I	21,824	1,192	27,000	2–3
J	1,024	0	8,950	1.5–2
K	4,096	0	7,900	5–6
L	8,512	0	8,800	5–6
M	256	0	4,500	6–7
N	512	0	1,250	1–2
O	2,096	0	32,400	1–2
P	4,192	0	6,050	4–5

transistors in the array that do not comprise the bit cells and latches. The runtimes, measured on an IBM RS 6000 590 AIX machine with 512 M bytes of memory, represent the time that it takes for the assertion set to pass successfully on the schematic. We can infer from Table 3 that the complexity of the control logic is an important factor influencing assertion runtimes in addition to the bit cells and latches. For example, array B takes the same amount of time as array C even though array C has fewer bitcells, latches, and control logic transistors compared to array B. Moreover, all the assertion runtimes are less than a day. In contrast, random Verilog simulation of these arrays would take anywhere from weeks to months and be extremely difficult to quantify the coverage. From the per-

Table 4. Validation effort.

Array block	Validation time (person-months)	Discrepancies
A	3	4
B	3	6
C	3	5
D	3	5
E	3	9
F	2	6
G	2	5
H	3	4
I	2	2
J	1	1
K	2	4
L	2	5
M	2	5
N	1	3
O	1	0
P	1	2

spective of CPU usage and coverage, our technique is far better than conventional validation methods for these arrays.

Table 4 presents the validation time and discrepancies uncovered using our methodology. Approximately, three person-years were spent in the validation of over 20 custom arrays, uncovering 66 discrepancies using this validation methodology. The validation time comprised methodology development, partial implementation of the automatic assertion generator, manual state node and timer mapping, assertion generation for bit cells and sense-amp outputs, development of the verification support libraries, establishing the BDD variable order, and debugging assertion failures. About half the validation time was spent on debugging assertion failures and getting the timer and state node mapping right.

The array validation methodology discovered many discrepancies between the RTL and schematics and circuit-related problems. These included incorrect clock regenerators feeding the wrong sets of latches, control logic errors in the array read and write enables, incorrect modeling of sense-amp output precharge in the RTL, incorrect hookup of the scan chain in the schematic, and incorrect modeling of the primary outputs in the RTL when certain clocks were deasserted. Many of these bugs would manifest when switching between different modes of operation during test, debug, power-on-reset, and so on. These modes are essential for testing and debugging the chip. Validation techniques that only look at the functional behavior will not find these errors. In addition to logical discrepancies, a number of potential circuit-related problems such as glitches and races were identified and reported to the designers.

We have established a validation methodology based on symbolic simulation. No differentiation is made between functional and nonfunctional modes of operation (such as scan, POR, debug). The automatically generated assertions from the RTL model are superior to manually generated assertions. By employing this methodology, we removed the burden of generating these assertions from the verification engineer and eliminated the quality variation of the assertions. We have shown that it is essential to do equivalence checking between the RTL and schematic models rather than only verify functional properties. The focus of our effort was on custom memories, primarily due to the predominance of bugs in such circuits and the failure of standard equivalence checking tools to handle them. However, our methodology is applicable to other circuit classes as well.

We have also established why symbolic simulation models with accurate event orders are required for validating certain classes of transistor circuits and why existing Boolean equivalence tools cannot handle such circuits. To achieve higher clock speeds, circuit designers are using more self-timed structures and free-running finite-state machines to implement their logic. Methodologies based purely on static analysis will fail to meet the needs of a designer. Moreover, it is our experience that symbolic simulation often exposes potential circuit-related problems, hazards, and other logic functionality issues that would be completely missed by combinational Boolean equivalence checking.

Despite our success with this methodology, there were issues to overcome. For certain arrays, we were not able to symbolically simulate the full-size array due to the blow up of the BDDs. In such cases, we worked on reduced models of the array and relied on our knowledge of the schematic to validate them. For example, full-size arrays A and E were never verified. Nonetheless, based on the verification of our reduced-size schematic models, we are confident that no bug has escaped our methodology.

The entire verification is based on a specific assumption about the order of events that occur in the control logic. If the real delays in silicon contribute to different event orders contradicting our assumptions, then a false positive could occur. Our methodology attempts to minimize such false positives by deriving these event orders from Spice simulations of the control logic. We plan to research deriving more accurate delay models for switch-level simulation.

The transistors in the sense amplifiers were simulated by automatically generating simulation models that treated them as unidirectional. Although, the assertion runtimes are reasonable, the human effort involved in establishing the mapping functions is time-consuming. An incorrect mapping function results in a debugging fix cycle that can be quite long, although false positives cannot occur.

Since custom memories and their associated custom logic constitute a large percentage of the chip area (approximately 70 to 80%), equivalence between RTL and transistor schematic models must be established. We have shown that our verification methodology comes closer to equivalence checking than previous approaches. Exposing shortcomings of our approach will enable us to plug any loopholes this methodology may have. Symbolic simulation is a viable and practical technique for validation and analysis of custom designs, and can fit neatly into existing methodologies. It is capable of working with models at different levels of abstraction. This technology promises to be an exciting field of work with many areas of application. ■

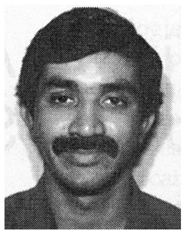
Acknowledgments

We thank the entire project and tools teams at Somerset, Motorola, for their cooperation and commitment to the successful conclusion of this project.

■ References

1. N. Ganguly, M. Abadir, and M. Pandey, “PowerPC Array Verification Methodology using Formal Techniques,” *In’l. Test Conf.,* Washington, D.C., 1996, pp. 857-864
2. M. Pandey et al., “Formal Verification of PowerPC Arrays Using Symbolic Trajectory Evaluation,” *Proc. 33rd ACM/IEEE Design Automation Conf.*, June 1996, pp. 649-654.
3. T. Kam and P.A. Subrahmanyam, “Comparing Layouts with HDL Models: A Formal Verification Technique,” *IEEE Trans. on Computer-Aided Design*, Apr. 1995, pp. 503-509.
4. R.E. Bryant, “Verifying a Static RAM Design by Logic Simulation,’’ *Proc. Fifth MIT Conf. on Advanced Research in VLSI*, 1988, pp. 335-349.
5. E.M. Clarke, E.A. Emerson, and A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,’’ *ACM Trans. on Programming Languages and Systems* Vol. 8, No. 2, 1986 pp. 244-263.
6. R.S. Boyer and J.S. Moore, “A Computational Logic,” ACM Monograph Series. Academic Press, New York, 1979.
7. K.J. Singh and P.A. Subrahmanyam “Extracting RTL Models from Transistor Netlists,*” IEEE/ACM Int’l Conf. on Computer-Aided Design*, Nov. 1995, pp. 11-17.
8. D. L. Beatty, R.E. Bryant, and C.J.H. Seger, “Synchronous Circuit Verification by Symbolic Simulation: An Illustration,” *Advanced Research in VLSI: Proc. Sixth MIT Conf.*, MIT Press, Mar, 1990, pp. 98-112.
9. R.E. Bryant and C.-J.H. Seger “Formal Verification of Digital Circuits Using Symbolic Ternary System Models,’’ *Computer-Aided Verification 1990*, E.M. Clarke, and R.P. Kurshan, eds., American Mathematical Society, 1991, pp. 121-146.
10. W.C. Carter, W.H. Joyner Jr., and D.Brand, “Symbolic Simulation for Correct Machine Design,’’ *Proc.16th ACM/IEEE Design Automation Conf.*, 1979, pp. 280-286.
11. J.A. Darringer “The Application of Program Verification Techniques to Hardware Verification,”

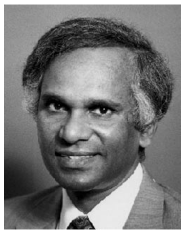
Proc. 16th ACM IEEE Design Automation Conf., 1979, pp. 375-381.
12. R.E. Bryant "Graph-Based Algorithms for Boolean Function Manipulation,'' *IEEE Trans. Computers*, Vol. C-35, No. 8, August 1986, pp. 677-691.
13. C.-J.H. Seger "Voss—a formal hardware verification system: user's guide," *Technical Report 93-45*, Dept. of Computer Science, University of British Columbia, 1993.
14. C.-J.H. Seger and R.E. Bryant, "Formal Verification by Symbolic Evaluation of Partially Ordered Trajectories," *Formal Methods in System Design*, vol. 6, 1995, pp. 147-189.
15. N. Krishnamurthy et al., "Equivalence Checking for PowerPC Custom Memories using Symbolic Trajectory Evaluation," *Proc. IEEE Int'l Workshop on Microprocessor Test and Verification*, (MTV99), 1999, pp.1-20.
16. N. Krishnamurthy et al., "Validation of PowerPC Custom Memories Using Symbolic Simulation," *Proc.18th IEEE VLSI Test Symp.*, Apr. 2000, pp. 9-14.
17. L.-C. 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.*, 1998.
18. R. E.Bryant, "Boolean Analysis of MOS Circuits," *IEEE Trans. CAD of Integrated Circuits*, CAD vol. 6, no. 4, July 1987.



Narayanan Krishnamurthy received a BTech in instrumentation engineering from the Indian Institute of Technology and a MS in electrical and computer engineering from the University of Texas at Austin, where he is currently pursuing a PhD. He is a CAD tools engineer in the ASP Advanced Tools and Methodologies Group at the Somerset PowerPC Design Center, Motorola, in Austin, Texas. His research interests include VLSI and dependable systems design, formal verification techniques and software engineering and testing.

Andrew K. Martin received his BSc in computing and information science from Queen's University at Kingston and his MSc and PhD in computer science from the University of British Columbia. Martin is a principal staff scientist for Motorola, Semiconductor Products Sector, Architecture and Systems Platforms Division in Austin, Texas. He is a member of the IEEE.

Magdy S. Abadir received his BS degree from the University of Alexandria and his MS degree from the University of Saskatchewan, both in computer science, and a PhD in electrical engineering from the University of Southern California. He works at Motorola as the test and verification manager in the Tools and Methodology Group at the Somerset PowerPC Design Center in Austin, Texas. He has published over eighty articles, as well as three books in his field of interest. He is a senior member of IEEE.



Jacob A. Abraham is a professor in the Departments of Computer Sciences and Electrical and Computer Engineering at the University of Texas at Austin. He received a BS degree in electrical engineering from the University of Kerala. He received both his MS and PhD from Stanford University in electrical engineering and computer science. Abraham's research interests are in VLSI design and test, formal verification, and fault-tolerant computing. He is a fellow of the IEEE.

■ For questions regarding this article, please contact Narayanan Krishnamurthy, e-mail narayanan.krishnamurthy@motorola.com.