Formal Verification of Memory Circuits by Switch-Level Simulation*

Randal E. Bryant
Carnegie Mellon University

July 1, 1999

Abstract

A logic simulator can prove the correctness of a digital circuit if it can be shown that only circuits implementing the system specification will produce a particular response to a sequence of simulation commands. Three-valued modeling, where the third state X indicates a signal with unknown digital value, can greatly reduce the number of patterns that need to be simulated for complete verification. As an extreme case, an N-bit random-access memory (RAM) can be verified by simulating just $O(N \log N)$ patterns. This approach to verification is fast, requires minimal attention on the part of the user to the circuit details, and can utilize more sophisticated circuit models than other approaches to formal verification. The technique has been applied to a CMOS static RAM design using the COSMOS switch-level simulator. By simulating many patterns in parallel, a massively-parallel computer can verify a 4K RAM in under 6 minutes.

1. Introduction

In this paper we present a new approach to verifying digital MOS systems by switch-level simulation. The concepts are developed using the verifi cation of a random access memory (RAM) as an illustrative example. We also discuss how the fundamental concepts apply to other classes of circuits.

1.1. Limitations of Conventional Simulation

Simulators provide a valuable tool for testing the correctness of digital circuits. Typically, however, only a limited set of test cases is simulated, and the circuit is presumed correct if the simulator yields the expected results for all cases. Unfortunately, this form of simulation provides no guarantee

*This research was supported by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, and by the Semiconductor Research Corporation under Contract 88-DC-068. Connection Machine time was provided by the Northeast Parallel Architectures Center at Syracuse University.

for $i \leftarrow 0$ to $N-1$:
    Write 1 at location i
for $i \leftarrow 0$ to $N-1$:
    Read at location i and test that output equals 1;
    Write 0 at location i
for $i \leftarrow N-1$ down to 0:
    Read at location i and test that output equals 0;
    Write 1 at location i

Figure 1: **Marching Test**. Tests such as this are commonly used to test for flaws in a RAM.

that all design errors have been eliminated. A successful simulation run can indicate either that the circuit design is correct, or that an insufficient set of test cases was tried. Designers often attempt to compensate by simulating very large numbers of test cases, consuming perhaps 1 or more weeks of CPU time on a mainframe computer. Such a brute force approach can only slightly increase the confidence in a circuit design.

Let us illustrate the limitations of brute force simulation in terms of an N-bit random access memory (RAM) circuit. Such a circuit has N binary state variables, and $n+2$ inputs, where $n = \log_2 N$. If we attempted to simulate memory operation for all possible combinations of input and initial state, we would need to simulate 2^{N+n+2} different cases. For even modest values of N, this is an unimaginably large number. For example, when $N = 256$, we would need to simulate around 10^{80} cases. To put this number on a cosmic scale, imagine if all the matter in our galaxy (around 10^{17} kilograms [20]) were used to build computers. Suppose furthermore that each of these computers were the size of a single electron (10^{-30} kg) and could simulate 10^{12} cases per second. If these computers had started simulating shortly after the universe was formed (about 10^{10} years ago [20]), then by now they would have simulated only around 0.05% of the total number of cases! Thus, whether we simulate a circuit for 1 second or for 1 year, we can only evaluate a miniscule fraction of the number of possible input and initial state combinations.

Conventional simulation techniques are surprisingly weak at uncovering design errors. Stories abound of systems containing fundamental design flaws that remain undetected despite extensive simulation. As an example, suppose we test a RAM design by simulating a standard memory test, such as a marching test [22]. This test proceeds as illustrated in Figure 1, marching up and down the addresses, reading the memory contents and writing the opposite digital value. Although such a test would detect many potential errors in a memory design, it does not guarantee correctness. In fact, if we examine the expected output sequence during a marching test, we see that it consists simply of N 1's, followed by N 0's. Consider the circuit illustrated in Figure 2, having the same external connections as a RAM, but consisting internally of an N-bit shift register in which shifting is enabled by a write operation. The address inputs have no effect on the circuit operation. This circuit cannot be used as a RAM, yet it would produce the desired response to the marching test! Clearly, it is dangerous to rely on a testing methodology that cannot even distinguish between a correct circuit and an obvious impostor.

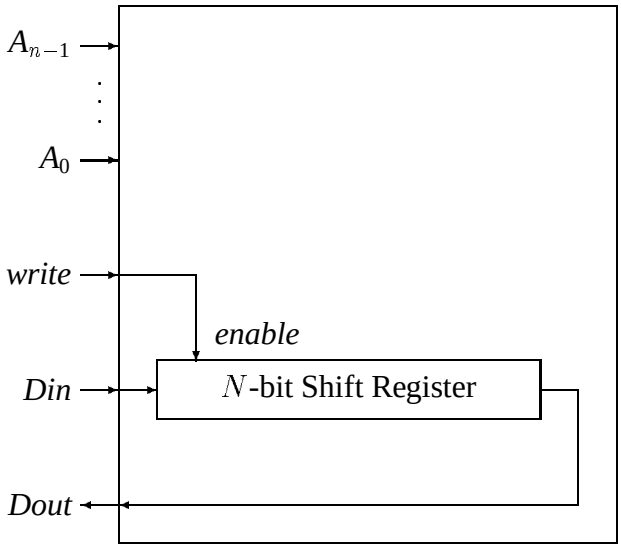

Figure 2: **Impostor RAM Circuit**. This circuit would pass a marching test, even though it does not remotely resemble a RAM.

Conventional wisdom holds that verifying a circuit by simulation is at best impractical and at worst impossible. As we have seen, the large number of possible input and initial state combinations would seem to require an overwhelming amount of simulation to test exhaustively. Furthermore, as Moore has shown [18], a sequential system cannot be fully characterized by observing its response to a sequence of stimuli. That is, for a given input sequence, we can always construct an impostor circuit that simply mimics the expected response of the system to this sequence, much as we did for the marching test.

1.2. Other Approaches to Formal Verification

Most researchers have turned to automated theorem provers [1, 12, 14, 16, 21] to demonstrate that a circuit meets the specification of its desired behavior. With the current state of the art, this process is only partially automated. The user must provide complete specifications of every component of the circuit and guide the program on proof strategies. Furthermore, these programs cannot operate with the detailed, transistor-level models required to verify complex MOS circuits. As an exception to this generalization, Weise [23] has developed a verifier that utilizes a very detailed electrical model. When composing circuits hierarchically, however, his program will at times resort to an exhaustive case analysis. This yields unsatisfactory performance for certain classes of circuits.

Other researchers have applied model checking programs [5] to construct a data structure representing the finite state behavior of the circuit, allowing the user to then prove assertions about the circuit behavior. This approach works well for small, controller circuits but is impractical for circuits, such as memories, having large numbers of possible states. More recently, these researchers have developed methods of representing state graphs symbolically [10]. By this means they can verify properties of state machines that are far too large (up to 10^{20} states) to represent explicitly.

Nonetheless, the memory circuits we verify are beyond the current capacity of even these newer methods.

1.3. Verification by Simulation

The conventional wisdom about logic simulation overlooks the capabilities provided by three-valued logic modeling, in which the state set $\{0, 1\}$ is augmented by a third value X indicating an unknown digital value. Most modern logic simulators provide this form of modeling, if for nothing more than to provide an initial value for the state variables at the start of simulation. Assuming the simulator obeys a relatively mild monotonicity property, a three-valued simulator can verify the circuit behavior for many possible input and initial state combinations simultaneously. That is, if the simulation of a pattern containing X's yields 0 or 1 on some node, the same result would occur if these X's were replaced by any combination of 0's and 1's. This technique is effective for cases where the behavior of the circuit for some operation is not supposed to depend on the values of some of the inputs or state variables. Three-valued modeling can also overcome the machine identification problem of Moore, assuming the user can command the simulator to set all state variables to X [6].

1.4. Limitations of Formal Verification

Formal verification involves proving that, under some abstract model of operation, the system will behave as specified for all possible input sequences. It does not guarantee, however, that the actual circuit will operate properly. The assumptions made in the abstract model may not hold in the physical implementation. For example, most methods of verifying digital systems assume that the circuit adheres to a logic abstraction whereby all signals can be represented by discrete values. Without such an abstraction, verification would be tedious, if not impossible. Design errors that cause marginal, nondigital circuit behavior may not be detected by verification against such a model.

For the case of verification by simulation, we must assume that the abstract model provided by the simulator faithfully captures the behavior of the actual circuit. When a circuit has been "verified" by a simulator, it simply means that any further simulation would not uncover any errors. We have obtained all of the useful information about the circuit that this particular simulator can provide. For example, once a RAM has passed our verification, we are assured that it contains no design errors that can be detected by switch-level modeling. Computer time that might have been spent simulating more test cases at the switch level should instead be used to analyze the circuit at more detailed modeling levels. It is important to maintain a perspective on just what formal verification provides. It reflects a limitation intrinsic to any approach to CAD modeling, and not to our method alone.

2. Overview of Paper

This paper develops the concept of verification by simulation in more detail, using as a case study the verification of a family of CMOS static RAM circuits by the switch-level simulator COSMOS

[7]. The circuit design was constructed solely as a benchmark for verification. However, it contains the same circuit structures found in actual CMOS static RAM's [13].

Random access memories are particularly amenable to verification by logic simulation. Although an N-bit memory has 2^N possible states, an operation on one memory location should not affect or be affected by the value at any other memory location. Thus many aspects of circuit operation can be verified by simulating the circuit with all, or all but one, bits set to X, covering a large number of circuit conditions with a single simulation operation. The resulting verification requires simulating only $O(N \log N)$ patterns. Even a minimal test of a memory design, such as a marching test, requires simulating $\Omega(N)$ patterns to make sure that each location can be written and read properly. The added $\log N$ factor seems a modest price to pay for a rigorous verification.

This case study provides a convincing demonstration of the advantages of verification by simulation. No other automatic verifiers are currently capable of verifying this design for nontrivial memory sizes. Most verifiers based on theorem provers do not provide a sufficiently detailed model of transistor operation to capture the behavior of the circuit. Weise's verifier would attempt an exhaustive case analysis of the circuitry forming the entire memory array due to the connections formed by the pass transistors in the column selector. Verifiers based on model checking would attempt to construct a finite automaton containing all 2^N possible memory states.

The patterns that must be simulated for the formal verification consist of many single cycle tests, each of which can be simulated independently. Thus the verifier can exploit *data parallelism* [9], in which the circuit is simulated for many patterns simultaneously. For example, running on a 32-bit machine, COSMOS is able to simulate up to 32 patterns simultaneously, speeding up the simulation by a factor of 10–30. In contrast, conventional simulation patterns, such as those in a marching test, must be simulated in sequence. Consequently, even on a conventional machine, we can formally verify a RAM faster than we can simulate a simple marching test. Running on a 32K processor Connection Machine reduces the verification time to a few minutes.

3. Verification Methodology

3.1. Simulation Model

For the purpose of verification, we view the circuit as a finite state machine, as illustrated in Figure 3. The circuit is modeled as an automaton, where each transition corresponds to the operation of the circuit for a single clock cycle. The automaton has n primary inputs and s state variables. Included in these state variables are the primary outputs as well as the internal state variables. We do not distinguish between these two classes of signals, since either can be observed during simulation. Furthermore, unlike in actual circuits, we assume that the values of the state variables can be altered by the user. In fact, our simulator treats every electrical node in the circuit as a state variable. With this abstract model, the process of simulating a circuit can be viewed as one of operating the automaton. Each cycle of simulation involves setting some of the inputs or state variables, operating the clocks, and computing new values for the state variables. We can then test for specified values on some of the state variables.

Since the simulator is assumed to faithfully model the actual circuit behavior, we make no attempt to distinguish between the circuit and its simulation model. Thus, we need not be concerned

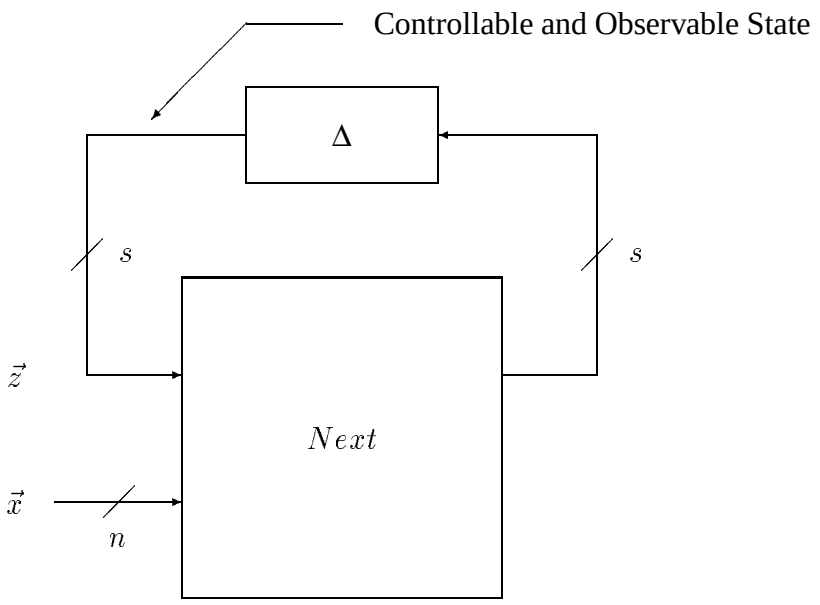

Figure 3: **Abstract Circuit Model Provided by Simulator**. The circuit is viewed as a finite state machine. The state, which can be both controlled and observed, consists of both the internal state variables and the primary outputs.

with many details of the actual circuit, such as the detailed transistor structure or the timing. Representing the circuit as a finite automaton corresponds to the view typically seen by the user of a logic simulator. The simulation program handles the details abstracted away by our model. It creates data structures representing the logic network and, via its simulation algorithm, implements the logic model. The clocking methodology is declared at the outset. From this point on, the user views the simulation task as one of validating a finite automaton. Hence, this circuit model is appropriate for the discussion here, since our goal in verification is simply to ensure that the user uncovers any design errors that can be detected by the simulator.

3.2. Three-Valued Modeling

Our simulation model extends the conventional state set for a digital system $\{0, 1\}$ with a third value X indicating an unknown or indeterminate value. In making this extension, the simulation model must obey certain mathematical properties to capture our intended interpretation of the value X.

Define the "information ordering" over the set $T = \{0, 1, X\}$ as the partial ordering where $X < 0$, and $X < 1$. That is, X indicates an absence of information, while 0 and 1 represent specific, fully-defined values. When speaking of domains ordered by information content, we say that values a and b are "consistent" if either $a \leq b$ or $b \leq a$, and "inconsistent" otherwise. Value a is "weaker" than b if $a < b$, i.e., $a \leq b$ and $a \neq b$.

The information ordering is extended to vectors pointwise. That is, two vectors $\vec{a}, \vec{b} \in T^m$, are ordered $\vec{a} \leq \vec{b}$ when $a_i \leq b_i$ for all $1 \leq i \leq m$. In other words, one vector value is less than or

b \ a	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

b \ a	0	1	X
0	0	1	X
1	1	1	1
X	X	X	X

b \ a	0	1	X
0	0	1	X
1	1	1	X
X	X	1	X

b \ a	0	1	X
0	0	1	X
1	1	1	X
X	X	X	X

Table 1: **All Monotonic Extensions of the** OR **Function.**

equal to another if each element of the first is less than or equal to the corresponding element of the second.

For partially ordered sets D_1, D_2 a *monotonic* function g: $D_1 \rightarrow D_2$ satisfies

$$a \leq b \Longrightarrow g(a) \leq g(b)$$

for all $a, b \in D_1$. Similarly, a monotonic function over multiple arguments satisfies this property for each argument.

For any program, such as a logic simulator, that processes data ordered by information content, monotonicity expresses an important property. Suppose the program is given a stimulus containing incomplete information, e.g., having some inputs equal to X. If the program obeys monotonicity, it will produce a response consistent with, but potentially weaker than, the response it would produce given a stronger stimulus.

For logic simulation, the three-valued behavior of a circuit is generally computed by monotonically extending functions defined over Boolean values to ones defined over ternary values. As an example, the two-input OR function can be extended monotonically from the Boolean to the ternary domain in the four different ways shown in Table 1. These extensions are listed from left to right in order of "pessimism", i.e., by the number different input combinations that are defined to yield X. A pessimistic extension is valid, but it will tend to produce X's on state variables even when there is no ambiguity in the actual circuit.

Our verification methodology holds for any simulator that obeys monotonicity. That is, the next state function $Next$: $T^s \times T^n \rightarrow T^s$ must be monotonic. The algorithms used by our switch-level simulator COSMOS guarantee this property [8]. Furthermore, any of a number of simulators could be used for formal verification. Most contemporary logic simulators provide a value X to avoid the need to find an initial Boolean state of the circuit that does not cause oscillations [17]. The monotonicity requirement simply expresses the desirable property that in the presence of X values, the simulator should not set an output or state variable to 0 or 1, when this would not have occurred had some of the X's been 0 or 1 instead. Any reasonable implementation satisfies this.

3.3. Interpretation of Results

Any approach to circuit verification can err in two different ways—it can accept an incorrect circuit or it can reject a correct one. We term the first form of error a *false positive response*, and the second

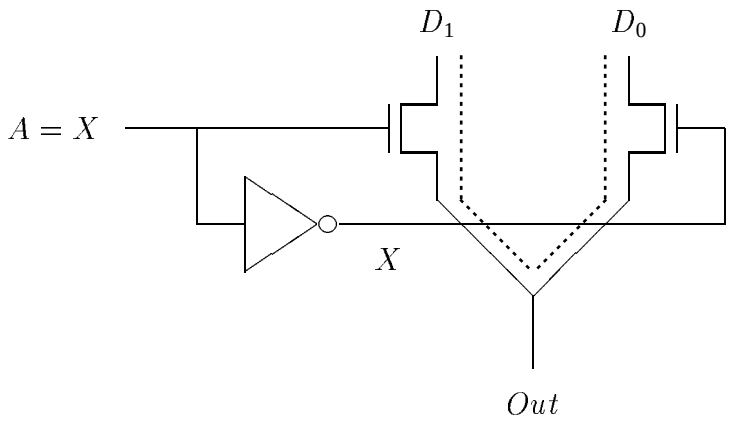

Figure 4: **False Sneak Path Example**. When A is X, the simulator views the two pass transistors as potentially forming a sneak path (shown as a dotted line) between data inputs D_1 and D_0.

a *false negative response*. As shall be shown, our verification methodology can never give a false positive response. If a circuit produces the expected results from the simulation patterns derived from the circuit specification, we are guaranteed that no simulation sequence would uncover any errors in the circuit.

On the other hand, our methodology can produce false negative responses. These responses are caused by pessimism in the simulation algorithm when modeling the effect of X values. An example of such pessimism is shown in Figure 4, illustrating part of a pass transistor multiplexor. Two trees of such multiplexors form the column decoder of our RAM circuit [13]. When the control signal A has value X, the simulator computes the inverter as having output X, and hence the two pass transistors form a potential sneak path (shown as a dotted line) between the data inputs D_0 and D_1. Unless D_0 and D_1 happen to be in the same state, the simulator will set both of them to X. In the actual circuit, the complementary signals on the pass transistor gates prevent both transistors from conducting simultaneously, and hence this sneak path never forms. Similar sources of pessimism occur in gate-level simulators, as well [4].

When the simulator produces an X when a 0 or 1 was expected, it could indicate a circuit design error, or it could simply be a false negative response. The more pessimistic the simulation algorithm, the greater is the likelihood of producing false negative responses. Unfortunately, computing the circuit state according to the least pessimistic monotonic function requires solving an NP-hard problem. Hence, most simulators err on the side of pessimism in the interest of efficiency, with a resultant tendency toward false negative responses. For example, logic gate simulators typically simulate each gate according to its least pessimistic monotonic extension [17].

3.4. Assertion Testing

We view the task of verifying a circuit as one of proving assertions about the next state behavior of the finite state machine illustrated in Figure 3. These assertions are expressed in a notation similar

to Floyd-Hoare assertions [11, 15]. Each assertion is an equation of the form

$$Initial \left\{ \ Action \ \right\} Result$$

where *Initial* specifies a precondition on the initial circuit state, *Action* specifies a condition on the circuit inputs, and *Result* specifies a postcondition on the resulting circuit state. All conditions are expressed as propositional formulas of the form $L_1 \wedge L_2 \wedge \cdots \wedge L_k$, where each L_i is a *literal* of the form $var = 1$ or $var = 0$, and var is some circuit input x_j or state variable z_j. Furthermore, no variable may occur in a formula more than once.

An assertion states that for any initial circuit state satisfying *Initial*, and any circuit operation satisfying *Action*, the resulting circuit state should satisfy *Result*. More formally, a circuit *satisfies* the assertion $Initial \ \{ \ Action \ \} \ Result$ if for any initial state $\vec{z} \in B^s$ satisfying *Initial* and any input $\vec{x} \in B^n$ satisfying *Action*, the state $Next(\vec{z}, \vec{x})$ satisfies *Result*.

Once a set of assertions has been devised, a simulator can verify that a particular circuit satisfies them. The restricted form of the assertion formulas guarantees that, if a vector $\vec{x} \in T^m$ satisfies a formula, then any vector $\vec{x}' \in T^m$ such that $\vec{x} \leq \vec{x}'$ must also satisfy the formula.

For a formula *Initial*, define the vector $\vec{z}_I \in T^s$

$$[\vec{z}_I]_i = \begin{cases} 1, & \text{if the term } z_i = 1 \text{ occurs in } \textit{Initial} \\ 0, & \text{if the term } z_i = 0 \text{ occurs in } \textit{Initial} \\ X, & \text{otherwise.} \end{cases}$$

In a similar fashion, define the vector $\vec{x}_A \in T^n$ according to the terms in *Action*. Observe that $\vec{z}_I$ is the minimum vector satisfying *Initial*, and that $\vec{x}_A$ is the minimum vector satisfying *Action*. These two vectors are equivalent to "cube" representations of the two formulas [19].

Theorem 1 *For an assertion $Initial \ \{ \ Action \ \} \ Result$ define corresponding vectors $\vec{z}_I$, $\vec{x}_A$ according to the formulas Initial and Action. If the vector Next$(\vec{z}_I, \vec{x}_A)$ satisfies Result, then the circuit satisfies the assertion.*

Proof: Let $\vec{z} \in T^s$ be any state satisfying *Initial*, and $\vec{x} \in B^n$ be any input satisfying *Action*. Clearly, $\vec{z}_I \leq \vec{z}$, and $\vec{x}_A \leq \vec{x}$. By the monotonicity of *Next*

$$Next(\vec{z}_I, \vec{x}_A) \leq Next(\vec{z}, \vec{x}),$$

and therefore $Next(\vec{z}, \vec{x})$ must satisfy *Result*.

□

This theorem indicates a straightforward procedure to test that a circuit satisfies an assertion. First, reset all input and state variables to X. Then set each state variable occurring in the formula *Initial* to its specified value. Similarly, set each input variable occurring in the formula *Action* to its specified value. Next, simulate one cycle of system operation. Finally, check that the value of each state variable occurring in *Result* is as specified. Examples of circuit assertions and the resulting simulation patterns will be given in the next section.

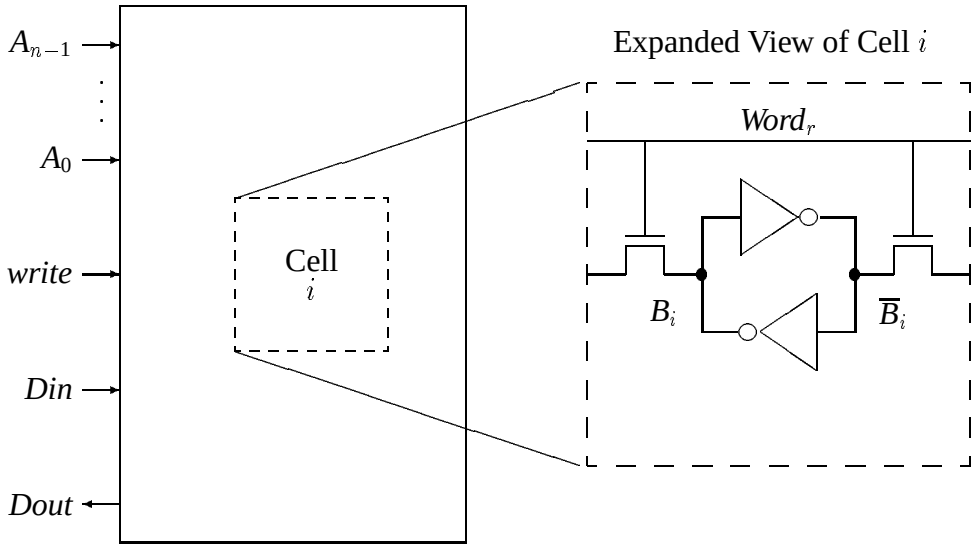

Figure 5: **Static RAM Circuit**

4. RAM Specification

The circuit to be verified is an $N \times 1$ bit static RAM. Memories with larger word sizes can be verified similarly, by verifying each bit of each word individually while setting all other bits to X. Figure 5 illustrates the general plan of the circuit. Assuming $N = 2^n$, the circuit has address inputs $A_{n-1}, \ldots, A_0$, a data input Din, and a control input $write$ that is set to 1 for a write and to 0 for a read operation. The circuit has a single output $Dout$. Each memory cell i contains a feedback path with a pair of inverters connecting nodes B_i and $\overline{B}_i$, along with a pair of access transistors [13]. As a shorthand, the formula $Stored(i, v)$ expresses the fact that value $v \in \{0, 1\}$ is stored in memory cell i:

$$Stored(i, v) \quad \equiv \quad B_i = v \ \wedge \ \overline{B}_i = \neg v.$$

The desired behavior of a memory circuit can be specified quite easily. Each assertion describes the effect of a single memory operation. In the following presentation, we ignore the details of the circuit interface, such as how the clocks are operated, as well as the signalling conventions for the inputs and outputs. Our verification is valid as long as the interface timing is simulated in the same manner as the circuit is operated.

First, a read operation should cause the contents of the addressed memory cell to appear on *Dout* without altering the cell. For all $v \in \{0, 1\}$ and all $0 \leq i < N$:

$$Stored(i, v) \left\{ \ A = i \ \wedge write = 0 \ \right\} \ Dout = v \ \wedge \ Stored(i, v), \tag{1}$$

where the notation $A = i$ is a shorthand indicating that for $0 \leq k < n$, each input line A_k equals i_k, the corresponding bit in the binary representation of i. These assertions can be verified by simulating a total of $2N$ patterns, two for each memory location. Each test involves initializing one memory cell to a value, all other cells to X, and then reading from the cell's address. The test passes if the stored bit appears on *Dout*, and the cell contents remain unchanged. These simulation patterns are called the "read" tests.

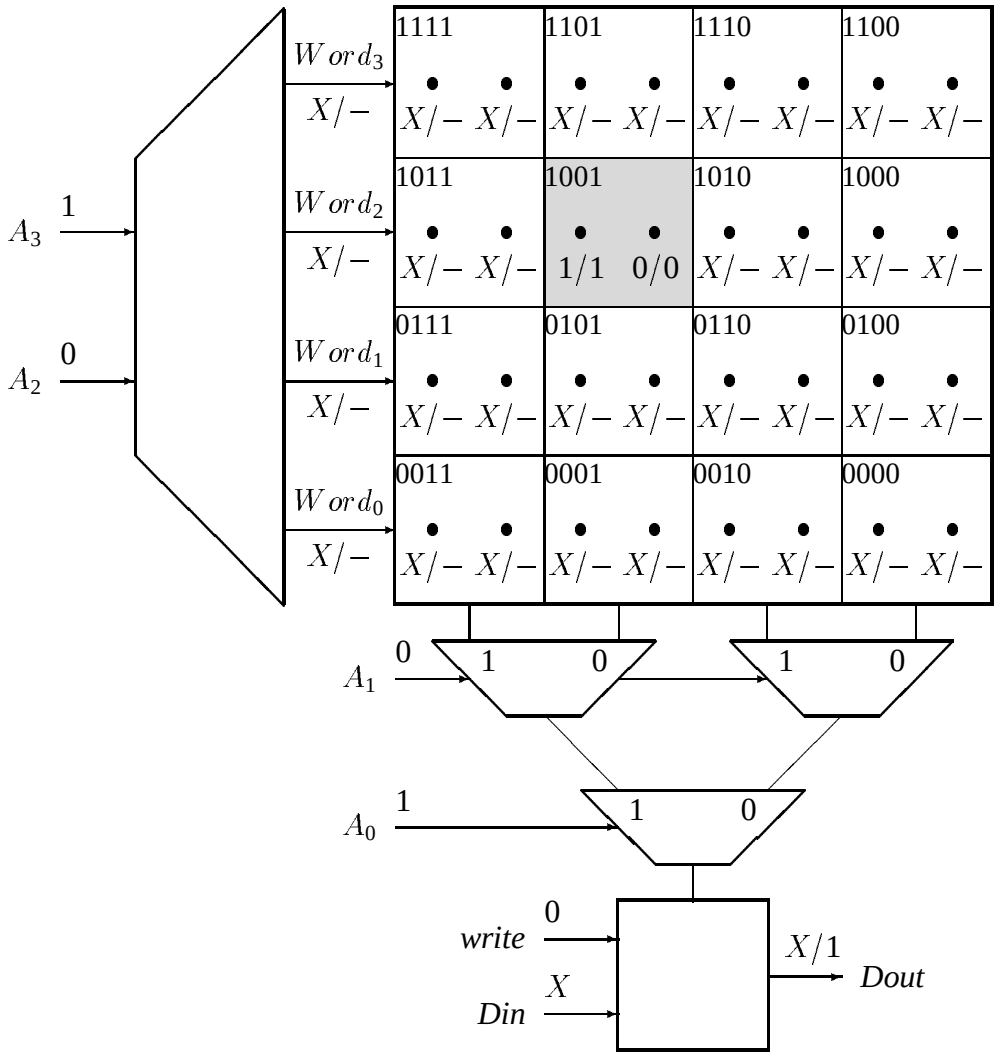

Figure 6: **Read Test Example**. This test proves the correctness of a read operation for address 9 (shaded) and value 1 from a 16-bit RAM. Labels of the form a/b indicate the initial and desired response values of a state variable, where – indicates "don't care".

As an example of a read test, consider the case of reading the value $v = 1$ from location $i = 9$ (1001_2) in a 16-bit RAM. The assertion for this case is given by

$$Stored(9,1) \left\{ A = 9 \wedge write = 0 \right\} Dout = 1 \wedge Stored(9,1).$$

The corresponding test is illustrated in Figure 6. In this figure, each memory cell is labeled by its address, each primary input is labeled by the value assigned during the test, and each state variable is labeled by a pair of states written in the form a/b, where a is the initial value and b is the expected resulting value. When b is –, the response value does not matter. The *Initial* formula $Stored(9,1)$ specifies an assignment $B_9 = 1$ and $\overline{B_9} = 0$. All other state variables are initialized to X. The *Action* formula $A = 9 \wedge write = 0$ specifies an assignment to the primary inputs of $A_3 = 1$, $A_2 = 0$, $A_1 = 0$, $A_0 = 1$, and $write = 0$. Note that Din is set to X, since it does not appear in *Action*. The *Result* formula $Dout = 1 \wedge Stored(9,1)$ specifies an expected response of $Dout = 1$, $B_9 = 1$, and $\overline{B_9} = 0$. Note that only these 3 values need to be checked following the simulation. State variables that do not appear in $Result$ need not be checked.

This example illustrates how three-valued simulation can cover a large number of combinations of input and initial state with a single simulation operation. By setting all memory cells except for cell 9 to X, the test proves that value 1 will be read from location 9 regardless of the state of the rest of the memory. Furthermore, by setting Din to X, the test also proves that this read operation does not depend on the input data.

The second part of the specification states that a write operation should cause the addressed memory cell to be updated. For all $v \in \{0,1\}$ and all $0 \leq i < N$:

$$True \left\{ Din = v \wedge A = i \wedge write = 1 \right\} Stored(i,v), \tag{2}$$

These assertions can be verified by simulating $2N$ patterns, two for each memory location. Starting with all state variables set to X, each test writes a value to a location, and then checks that the value has been stored correctly. These patterns are called the "write" tests.

The final part of the specification states that any memory operation on one cell should not affect the value stored in any other memory cell. For all $v \in \{0,1\}$, and all $0 \leq i, j < N$, such that $i \neq j$:

$$Stored(i,v) \left\{ A = j \right\} Stored(i,v). \tag{3}$$

This set of assertions represents $2N^2$ combinations of address and data values.

To gain more efficiency, we can obtain the same effect with just $2N \log N$ combinations. For an address i with bit representation $\langle i_{n-1}, \ldots, i_0 \rangle$, all addresses j such that $j \neq i$ are covered by the n patterns of the form $\langle X, \ldots, X, \neg i_k, X, \ldots, X \rangle$ for $0 \leq k < n$. For example, the following 4 patterns cover all addresses $j \neq 9$ (1001_2) for $n = 4$:

	A_3	A_2	A_1	A_0
i	1	0	0	1
$k = 3$	0	X	X	X
$k = 2$	X	1	X	X
$k = 1$	X	X	1	X
$k = 0$	X	X	X	0

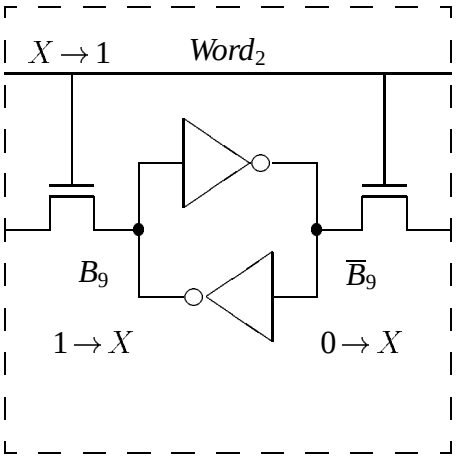

Figure 7: **False Negative Response Example.** The initial value X on the word line causes the memory cell to be corrupted.

Thus, the assertions can be replaced by the following assertions for $v \in \{0, 1\}$, $0 \leq i < N$, and $0 \leq k < n$:

$$Stored(i, v) \left\{ A_k = \neg i_k \right\} Stored(i, v). \tag{4}$$

These assertions can be verified by patterns in which a memory cell is initialized to some value, one of the address inputs is set to the complement of the corresponding bit in the cell's address, and all other input and state variables are set to X. Following the simulation of one cycle, the cell value is compared to its original value. These simulation patterns are termed the "address" tests.

5. Circuit Dependent Refinements

Equations 1, 2, and 4 translate directly into a total of $4N + 2N \log N$ simulation patterns. However, on our example circuit, the simulator gives false negative responses for all of the read and address tests. For example, the read test illustrated in Figure 6 would fail. The cause of this is illustrated in Figure 7. At the start of simulation, all internal state variables, including the control line $Word_2$, are set to X. During the simulation, this line will be set to 1. Before this occurs, however, the simulator models the effect on memory cell 9 of having its access transistors set to X. That is, it sets the internal nodes of the cell to X.

These problems with false negative responses can be overcome by adding one new assertion and by refining the existing ones. The resulting simulation patterns provide an equally rigorous test that the circuit passes. Refining the specification into a set of simulation patterns requires a more detailed consideration of the control sequencing and of the row and column addressing structure. Even with these details, we can ignore many aspects of the design, letting the simulator capture their behavior by its simulation model.

Assume that the circuit is organized as a $\sqrt{N} \times \sqrt{N}$ array of memory cells, where address bits $Arow = A_{n-1}, \ldots, A_{n/2}$ select the row, and address bits $Acol = A_{n/2-1}, \ldots, A_0$ select the column. As an example, Figure 6 shows the addressing structure for a 16-bit RAM. Address inputs A_3 and A_2 are decoded to generate the signals on the 4 word lines. Address inputs A_1 and A_0 control a

tree of bidirectional multiplexors to create a path between the selected column and the data input or output.

5.1. Control Line Initialization

Correct operation of this circuit relies on the fact that when the circuit is quiescent, the access transistors to all memory cells are shut off. That is, at the beginning of every memory cycle, $Word_r = 0$ for $0 \leq r < \sqrt{N}$. Without this property, two cells in a single column could interact in undesirable ways, as occurred in the case illustrated in Figure 7. This fact is formulated as a *system invariant*

$$Inv \quad \equiv \quad \forall(0 \leq r < \sqrt{N})[\mathit{Word}_r = 0].$$

The invariance of this condition is expressed by a single assertion:

$$True \left\{ \; True \; \right\} Inv \tag{5}$$

That is, following any memory operation, the word lines will return to a quiescent condition. Testing this invariant involves simply simulating a single cycle of memory operation with all state and input variables initialized to X and then checking that all word lines are set to 0 at the end.

Once the assertion has been established, the invariant *Inv* can be assumed as a precondition in all other assertions, giving a revised assertion for the read tests for all $v \in \{0, 1\}$, and all $0 \leq i < N$:

$$Inv \; \wedge \; Stored(i, v) \left\{ \; A = i \; \wedge write = 0 \; \right\} Dout = v \; \wedge \; Stored(i, v). \tag{6}$$

That is, we can begin all simulation read cycles with the word lines initialized to 0. In Figure 6, the labels for the word lines would then be $0/-$. With this refi nement, the circuit passes the read tests.

Most circuits require some form of system invariant expressing conditions about the control logic that can be assumed true at the beginning of every clock cycle. Devising the invariant requires a combination of analysis and experimentation. An insuffi cient system invariant will become immediately apparent during subsequent simulations, because output or state variables that should have Boolean values will equal X.

5.2. Row and Column Decoding

Even with the invariant our circuit still passes only half of the the address tests, namely those corresponding to the following equations for $v \in \{0, 1\}$, $0 \leq i < N$, and $n/2 \leq k < n$:

$$Inv \; \wedge \; Stored(i, v) \left\{ \; A_k = \neg i_k \; \right\} Stored(i, v). \tag{7}$$

For these tests, some bit k of the row address is set to $\neg i_k$, a controlling value for the NOR gate of word line decoder for memory cell i. The word line stays at 0 and the bit stored in cell i remains unchanged. These tests are called the "row address" tests. They prove that no memory cell is affected by an operation on a cell in a different row.

As an example, Figure 8 illustrates the addressing patterns for the row address tests for memory location 9 (1001 binary) in the 16-bit RAM of Figure 6. The two address settings: $0XXX$ and

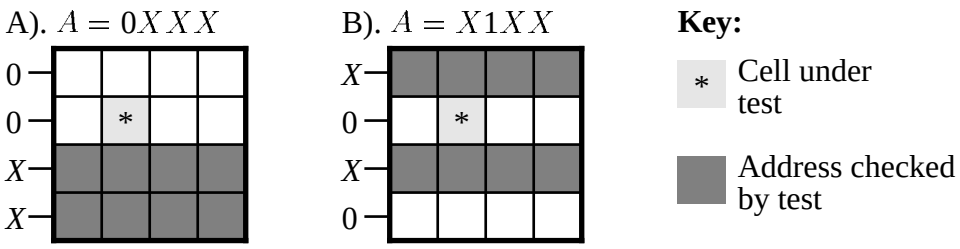

Figure 8: **Row Address Tests for Memory Location 9.** Signals on the left indicate the values on the word lines. The word line controlling cell 9 remains at 0.

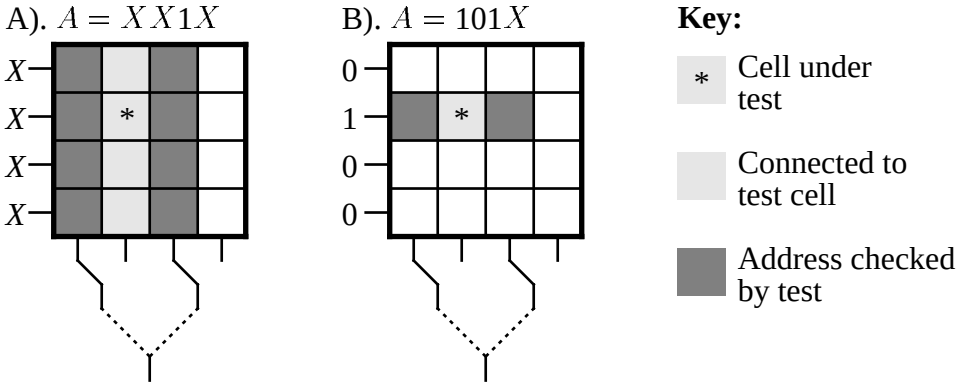

Figure 9: **Example of First (A) and Second (B) Type of Column Address Test for Memory Location 9.** The tree on the bottom indicates the connections formed by the column multiplexors, with dotted lines representing pass transistors with gate value X. In A), the word line value of X causes cell 9 to be corrupted. This is avoided in B).

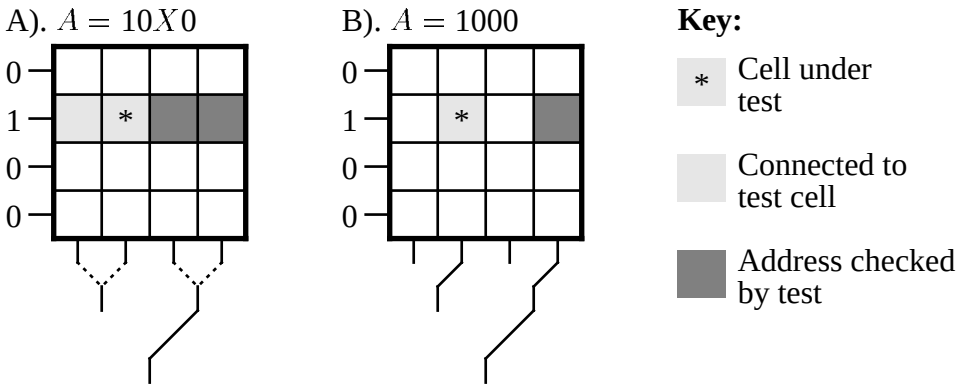

Figure 10: **Example of Second (A) and Third (B) Type of Column Address Test for Memory Location 9.** In A), a sneak path forms through the column multiplexor between cell 9 and an adjacent cell. This is avoided in B).

$X1XX$ cause the word lines to have the values shown on the left. In both cases, cell 9 remains isolated from all others. The dark shaded areas indicate the cell addresses covered by these two tests. The union of these areas includes all addresses in other rows of the memory.

For the cases that fail, the NOR gates of the word line decoders have all X's on their inputs, causing sneak paths to form between the cell under test and other cells in the column. Figure 9A shows an example of such a pattern for memory location 9 in the 16-bit RAM of Figure 6. Although no connection is formed between this cell and the data input or output, (indicated by the tree structure at the bottom), the stored bit is corrupted by the other cells in the column (indicated by the lightly shaded area.)

Fortunately, we can overcome this problem by removing some of the redundancy from the tests. Once a circuit passes the row address tests, we need only show that no memory cell is affected by an operation on a cell in a different column of the same row. This can be expressed by the following equations for $v \in \{0, 1\}$, $0 \leq i, j < \sqrt{N}$, and $0 \leq k < n/2$:

$$Inv \ \wedge \ Stored(j + i\sqrt{N}, v) \left\{ \ Arow = i \ \wedge \ A_k = \neg j_k \ \right\} Stored(j + i\sqrt{N}, v).$$

These assertions define a series of tests in which the memory cell at row i, column j is initialized to a value v, the row address is set to i, and some bit of the column address is set to the complement of the corresponding bit in j. Figure 9B shows an example of such a pattern for memory location 9. The word lines are set so that only cells in a single row are accessed. Furthermore, the column addresses are set so that the column containing cell 9 remains isolated. This test covers the two cell addresses indicated by the darkly shaded area.

Even with this refinement, our circuit encounters a new problem due to the tree structure of the column selector. Under normal operation of the circuit, all cells in the selected row are read, and the pass transistors of the column multiplexors form a path between the selected column and the data input and output. When some of the column address lines equal X, however, the simulator finds false sneak paths throughout the column multiplexor, causing a connection between the cell under test and one in another column. An example of this problem is shown in Figure 10A. Even though only a single row of cells is accessed, a sneak path forms between the column containing cell 9 and an adjacent column (indicated by the lightly shaded area).

Again, this problem can be overcome by removing some of the redundancy from the tests. For a column address j having bit representation $\langle j_{n/2-1}, \ldots, j_0 \rangle$, all column addresses not equal to j are covered by patterns of the form $\langle \neg j_{n/2-1}, X, \ldots, X \rangle$, $\langle j_{n/2-1}, \neg j_{n/2-2}, X, \ldots, X \rangle$, and so on up to $\langle j_{n/2-1}, \ldots, j_1, \neg j_0 \rangle$. Each of these patterns has the property that the simulator will never find a path of potentially conducting transistors (i.e., with gate value 1 or X) between the bit lines of column j, and those of any other column. These tests can be expressed by a revised set of equations for $v \in \{0, 1\}$, $0 \leq i, j < \sqrt{N}$, and $0 \leq k < n/2$:

$$\begin{aligned} & Inv \ \wedge \ Stored(j + i\sqrt{N}, v) \\ & \quad \left\{ \ Arow = i \ \wedge \ A_k = \neg j_k \ \wedge \ \forall (k < t < n/2)[A_t = j_t] \ \right\} \\ & Stored(j + i\sqrt{N}, v). \end{aligned} \tag{8}$$

These tests are called the "column address" tests.

The pattern of Figure 9B shows one of the column address tests for location 9 in the 16-bit RAM of Figure 6. The other is shown in Figure 10B. Observe that in both cases, the column containing cell 9 remains isolated, avoiding any corruption of the value stored there. The darkly shaded areas indicate the cell addresses tested by these patterns. The union of these areas includes all other cells in the row containing cell 9. These, combined with the two row address tests of Figure 8 cover all possible addresses other than location 9. That is, our final selection of address values is:

	A_3	A_2	A_1	A_0
i	1	0	0	1
$k = 3$	0	X	X	X
$k = 2$	X	1	X	X
$k = 1$	1	0	1	X
$k = 0$	1	0	0	0

Equations 2, 5, 6, 7, and 8 together define a total of $1 + 4N + 2N \log N$ simulation patterns that our circuit passes and that prove its correctness.

6. Simulator Performance

The simulation operations called for by our memory verification tests differ markedly from those used in more traditional simulation methodologies. Each involves resetting the simulator to a condition where all input and state variables equal X, setting a small number of inputs and state variables to Boolean values, and then simulating a single cycle. In contrast, most simulators are designed to simulate long sequences of Boolean patterns. The differences between these two styles of simulator usage place differing demands on simulator functionality and performance. In developing the switch-level simulator COSMOS, we attempted to satisfy the needs of both forms of simulation.

Most simulators employ very pessimistic or inefficient algorithms for computing the behavior of a circuit in the presence of X's. With conventional usage, there is no need to do better, because most X's are eliminated at the start of simulation and never arise again. For our verification patterns, however, X's are the rule rather than the exception, and hence the algorithms must be as accurate

N	Verification Patterns	Transistors	Marching Test	Serial Verification	Bit-Parallel Verification	CM Parallel Verification
4	33	113	1.0 s.	2.0 s.	0.6 s.	
16	193	235	8.4 s.	22.6 s.	2.0 s.	
64	1,025	611	117 s.	385 s.	19.3 s.	
256	5,121	1,931	30.8 m.	122 m.	4.4 m.	11 s.
1024	24,577	6,875	7.2 h.	31.9 h.	1.1 h.	46 s.
4096	114,689	25,995	137 h.		23 h.	5.6 m.

Table 2: **COSMOS CPU Times on MicroVax-II and on CM-2**

and efficient as possible. The algorithms used by COSMOS satisfy these goals reasonably well, although, as the static RAM example shows, developing a set of verification patterns requires some understanding of both the circuit design and the simulation algorithm.

In the design of the switch-level simulator COSMOS, we were also able to optimize the efficiency when simulating many short sequences. Most of these optimizations involved simply tuning the performance of code that is normally considered non-critical, such as the code to reset all state variables of a circuit to X.

More significantly, however, we can take advantage of the fact that each of the test sequences can be evaluated independently. By exploiting *data parallelism* [9] the simulator can evaluate many test sequences simultaneously. Running on a conventional 32-bit machine, our program can simulate up to 32 patterns in parallel. This mode exploits the bit-level parallelism available in computer logic operations. The COSMOS preprocessor transforms a transistor network into a set of evaluation procedures that utilize only memory references and logical operations. Hence, bit-level parallelism adds little extra cost. Experiments indicate that it increases simulation performance by a factor of 10–30. Running on a 32K processor Connection Machine, our program can simulate up to 32K patterns in parallel. In this mode, each processor maintains the state of the circuit for a single test case. The controller commands the processors to perform logic operations on their copies of the circuit state.

Although this would appear to be an obvious way to improve performance, most simulators make no use of data parallelism. Many simulation algorithms cannot exploit it. Furthermore, with conventional simulator usage, the simulation patterns are not formulated as a set of independent tests that can be run in parallel. Other work on parallel simulation attempts to exploit *circuit parallelism*, in which many circuit elements are evaluated simultaneously while simulating a single input sequence. Circuit parallel simulation has the advantage that it can be used in the same manner as conventional simulation. To date, however, no one has demonstrated a significant performance advantage over serial simulation.

7. Experimental Results

The verification methodology has been applied to memory sizes ranging from 4 to 4096 bits. The performance of the program is shown in Table 2. Columns 4–6 of this table show simulation CPU times, measured on a Digital Equipment Corporation MicroVax-II. Column 4 shows the

time to simulate a marching test, giving a minimal test that all locations can be written and read, but not proving the circuit's correctness. Column 5 shows the time to simulate the verification patterns without using bit-level parallelism. Column 6 shows the time to simulate the verification patterns using 32 way bit-level parallelism. Column 7 shows the time on a 32K processor Thinking Machines CM-2 with a VAX 8800 as host.

As can be seen, the bit-level parallel verification is faster than a simple marching test! Although a marching test requires simulating only $O(N)$ cycles, the extra $\log N$ factor of the verification patterns is more than compensated for by the speed-up provided by bit-level parallelism. Observe, however, that the overall simulation time in all 3 cases grows roughly quadratically with the memory size. As the memory size grows, both the number of patterns and the time to simulate a single pattern grow at least linearly. This complexity becomes noticeable for larger memory sizes. As can be seen from the table, we are approaching the limit of practicality for conventional machines.

Mapping the simulation onto the CM-2 decreases the time to verify a circuit dramatically. Even our largest circuit requires less than 6 minutes to verify. At present, mapping still larger memory circuits onto the machine is difficult due to the memory limitations of the CM-2 processors. Our mapping requires 4 bits per node plus temporary storage for Boolean evaluation on each processor. The 4K RAM circuit has 8839 nodes and requires nearly 18,000 bits of additional temporary storage. This consumes nearly the entire memory capacity (64K bits/processor) of the current machine. This size limitation will improve as machines with larger memory capacity become available.

8. Observations and Future Directions

This paper has shown that a typical CMOS static RAM design can be formally verified easily and efficiently by three-valued, switch-level logic simulation. Although these patterns were developed specifically for this design, similar techniques can develop patterns for almost all RAM designs. We have extended the methodology to verify a dual-ported RAM from an actual chip design by simulating $O(N \log^2 N)$ patterns [2]. Other classes of memory designs can also be verified by simulating a linear, or nearly-linear number of patterns. Included among these are shift registers, FIFO's, and stacks. On the other hand, content-addressable memories do not seem to fit into this class, since it is not as easy to identify where a particular datum will be stored.

Other classes of circuits cannot be verified by simulating a polynomial number of patterns. Many functions computed by logic circuits, such as addition and parity, depend on a large number of input or state variables. For these circuits, we propose *symbolic* simulation [3] as a feasible and straightforward approach to design verification. A symbolic simulator resembles a conventional logic simulator, except that the user may introduce symbolic Boolean variables to represent input and initial state values. The simulator computes the behavior of the circuit as a function of these Boolean variables. Symbolic simulation can utilize a methodology similar to that shown in this paper, by allowing the formulas in an assertion to be predicates containing universally-quantified Boolean variables. For example, we could view Equations 1, 2, and 3 as each representing a single assertion, and verify the RAM by simulating just three symbolic patterns.

Symbolic simulation has advantages over conventional simulation in both efficiency and ease of use. Our initial experiments indicate that a symbolic simulator using good symbolic Boolean

manipulation algorithms can achieve acceptable performance in many cases where conventional simulation would be impractical. Furthermore, there is less need to optimize simulation efficiency via such circuit-dependent refinements as were used in the RAM verification. Symbolic simulation can more accurately distinguish false from actual sneak paths. For example, suppose input A in Figure 4 were set to Boolean value a. Then the simulator would recognize that no sneak path is formed, since $\overline{a} \cdot a = 0$.

Although we have set a new standard for the size and class of circuit that can be verified formally, it is clear that some other technique is required to verify very large memories. Ideally, a verifier should be able to prove the correctness of an entire family of circuits given a parameterized description of the family [12]. Families of RAM circuits have very concise descriptions and hence seem ideal for this style of verification. However, developing such a verifier that can handle a sufficiently detailed MOS circuit model is no easy task.

References

[1] H. G. Barrow, VERIFY: a program for proving correctness of digital hardware designs. *Artificial Intelligence 24* (1984), 437–491.

[2] D. Beatty, Personal communication, 1988.

[3] D. L. Beatty, R. E. Bryant, and C.-J. H. Seger, "Synchronous circuit verification by symbolic simulation: An illustration," *6th MIT Conference on Advanced Research in VLSI*, April, 1990.

[4] M. E. Breuer, "A note on three-valued logic simulation," *IEEE Transactions on Computers C-21*, 4 (April 1972), 399–402.

[5] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic verification of sequential circuits using temporal logic," *IEEE Transactions on Computers C-35*, 12 (Dec. 1986), 1035–1044.

[6] R. E. Bryant, *A methodology for hardware verification based on logic simulation*, Technical Report CMU-CS-87-128, Carnegie Mellon University, June, 1987.

[7] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: a compiled simulator for MOS circuits," *24th Design Automation Conference*, 1987, 9–16.

[8] R. E. Bryant, "Boolean analysis of MOS circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-6*, 4 (1987), 634–649.

[9] R. E. Bryant, "Data parallel switch-level simulation," *Int. Conf. on Computer-Aided Design*, IEEE, 1988, 354–357.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking," *27th Design Automation Conference*, 1990.

[11] R. W. Floyd, "Assigning meanings to programs," *Proc. Symp. in Applied Mathematics, 19—Mathematical Aspects of Computer Science*, Schwartz, J. T., Ed. AMS, 1967, 19–32.

[12] S. M. German, and Y. Wang, “Formal verification of parameterized hardware designs,” *Int. Conf. on Computer Design*, IEEE, 1985, 549–552.

[13] L. A. Glasser, and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley, Reading, MA, 1985.

[14] M. Gordon, “Why higher-order logic is a good formalism for specifying and verifying hardware,” *Formal Aspects of VLSI Design*, G. Milne and P. A. Subrahmanyam, *eds.*, North-Holland, 1986, 153–177.

[15] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Comm. ACM 12* (1969), 576–580.

[16] W. A. Hunt, “The mechanical verification of a microprocessor design,” *From HDL Descriptions to Guaranteed Correct Designs*, D. Borrione, *ed.*, North-Holland, 1987, 89–129.

[17] J. S. Jephson, R. P. McQuarrie, and R. E. Vogelsberg, “A three-level design verification system,” *IBM Systems Journal 8*, 3 (1969), 178–188.

[18] E. F. Moore, “Gedanken-experiments on sequential machines,” *Automata Studies*, Shannon C. E., and McCarthy, J., Eds. Princeton University Press, Princeton, NJ, 1956, 129–153.

[19] J. P. Roth, *Computer Logic, Testing, and Verification*, Computer Science Press, Rockville, MD, 1980.

[20] M. Rowan-Robinson, *Cosmology*, 2nd edition, Clarendon Press, Oxford, 1981.

[21] R. E. Shostak, “Verification of VLSI designs,” *Proceedings of the Third Caltech Conference on VLSI*, Bryant, R., Ed. Computer Science Press, Rockville, MD, 1983, 185–206.

[22] S. Weingarden, and D. Pannell, “Paragons for memory test,” *International Test Conference*, IEEE, 1981, 44–48.

[23] D. Weise, “Functional verification of MOS circuits,” *24th Design Automation Conference*, ACM and IEEE, 1987, 265–270.