Verifying a Static RAM Design by Logic Simulation[1]

Randal E. Bryant

Computer Science Department
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

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 can be verified by simulating just $O(N \log N)$ patterns. The technique has been applied to a CMOS static RAM design using the COSMOS switch-level simulator. 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.

1. Introduction

Although logic simulators are widely used to test circuit designs informally, they have not been recognized as tools for formally proving the correctness of circuits. Conventional wisdom holds that verifying a circuit by simulation is at best impractical and at worst impossible. 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 [9], a sequential system cannot be fully characterized by observing its response to a sequence of stimuli. This would seem to indicate that, unless supplemented by detailed knowledge of the circuit structure, no amount of simulation can prove the correctness of a sequential system.

Most researchers have turned to automated theorem provers [1, 6] to

[1]To appear in the proceedings of the Fifth MIT Conference on Advanced Research in VLSI, March 1988.

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 [10] 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 [2] 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 especially well for small, asynchronous controller circuits but is impractical for circuits, such as memories, having large numbers of possible states.

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.

For performance reasons, most simulators err on the side of pessimism in modeling the effects of X values. That is, they will produce an X at some point even though it can be shown that the circuit would produce a 0 or 1 in all cases covered by the X values on the input and initial state variables. This can cause a verifier based on three-valued simulation to give a *false negative* response, labeling a correct design as defective. Fortunately, these erroneous responses always have the form

of producing an X at some point where a 0 or 1 was expected.

On the other hand, a verifier based on three-valued simulation can never produce a *false positive* response, labeling a defective design as correct. That is, if a circuit passes our verification tests, then no other simulation sequence will uncover additional errors. Although this claim is only demonstrated informally in this paper, it is backed up by a more formal theory and proof [5]. Of course, this style of verification proves the correctness of the actual circuit only if the simulator faithfully models the circuit behavior. Any approach to formal verification must assume that its abstract model of circuit behavior is valid.

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.

This paper develops these ideas in more detail, using as a case study the verification of a CMOS static RAM circuit design by the switch-level simulator COSMOS [4]. 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 [7].

This circuit 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.

A high-level specification of the desired circuit behavior can be expressed quite easily. A straightforward translation of the specification into a set of simulation patterns, however, yields false negative responses. Overcoming these problems requires taking into account the details of the row and column addressing structure. Although this places additional burden on the user, it does not compromise the rigor of the verification in any way, and the amount of detail is reasonably small.

The resulting verification requires simulating only $O(N \log N)$ patterns. Even a minimal test of a memory design 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. Furthermore, we have been able to tune the performance of our simulator to match the characteristics of the simulation patterns arising from formal verification. This tuning makes formal verification require no more simulation time than a minimal design test.

2. Verification Methodology

Specifications are expressed in a notation similar to Floyd-Hoare assertions [8]. Each assertion is an equation of the form

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

where *Initial* specifies a precondition on the initial circuit state, *Action* specifies a circuit operation, and *Result* specifies a postcondition on the circuit output and 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 $var = 0$, for some circuit input, output, or state variable var. For this paper, an *Action* will specify a condition on the inputs for a single cycle of circuit operation. An assertion states that for any initial circuit state satisfying *Initial*, and any circuit operation satisfying *Action*, the resulting circuit state and output should satisfy *Result*.

With a three-valued simulator, a circuit can be shown to satisfy an assertion by simulating a single pattern. Starting with every input and state variable set to X, the input and state variables appearing in the formulas *Initial* and *Action* are set to their specified values. Then the circuit is simulated for one cycle, and the values on the output and state variables appearing in *Result* are compared to their specified values. The monotonicity requirement imposed on the simulator guarantees that a circuit satisfies the assertion if it passes this test.

3. System 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 1 illustrates

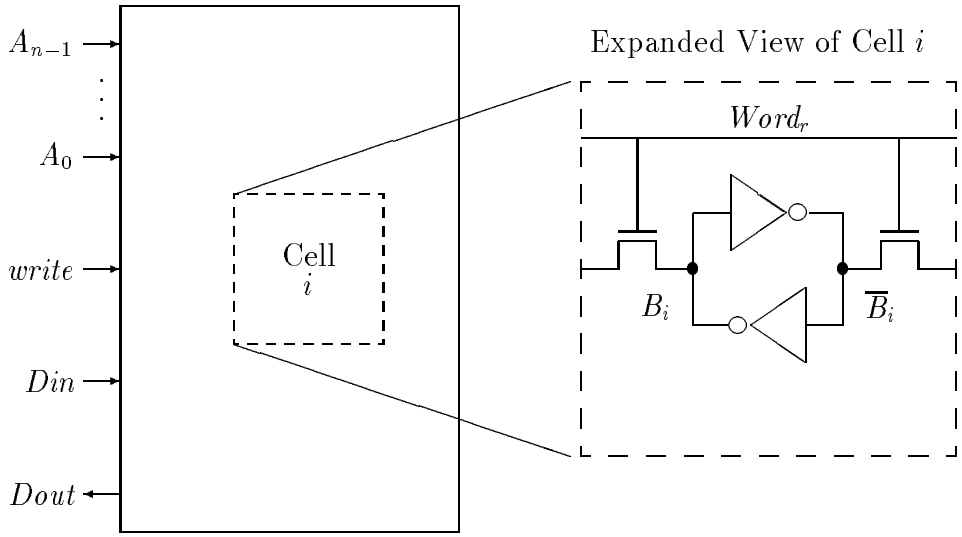

Figure 1: *Static RAM Circuit*

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 [7]. As a shorthand, the formula $Store(i, v)$ expresses the fact that value $v \in \{0, 1\}$ is stored in memory cell i:

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

Unlike many other sequential systems, the desired behavior of a memory circuit can be specified quite easily. First, a write operation should cause the addressed memory cell to be updated. For all $v \in \{0, 1\}$ and all $0 \leq i < N$:

$$True \left\{ \ Din = v \ \wedge A = i \ \wedge write = 1 \ \right\} Store(i, v), \qquad (1)$$

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 $2N$ 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.

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

$$Store(i,v) \left\{ A = i \ \wedge write = 0 \right\} Dout = v \ \wedge \ Store(i,v). \quad (2)$$

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 locations 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.

Finally, 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 $i \neq j$:

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

This set of assertions represents $2N^2$ combinations of address and data values. However, 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 $0 \leq k < n$. Thus, the assertions can be replaced by the following assertions for $v \in \{0,1\}$, $0 \leq i < N$, and $0 \leq k < n$:

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

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.

4. 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. By adding one new assertion and refining the existing ones, we can devise an equally rigorous test that the circuit passes.

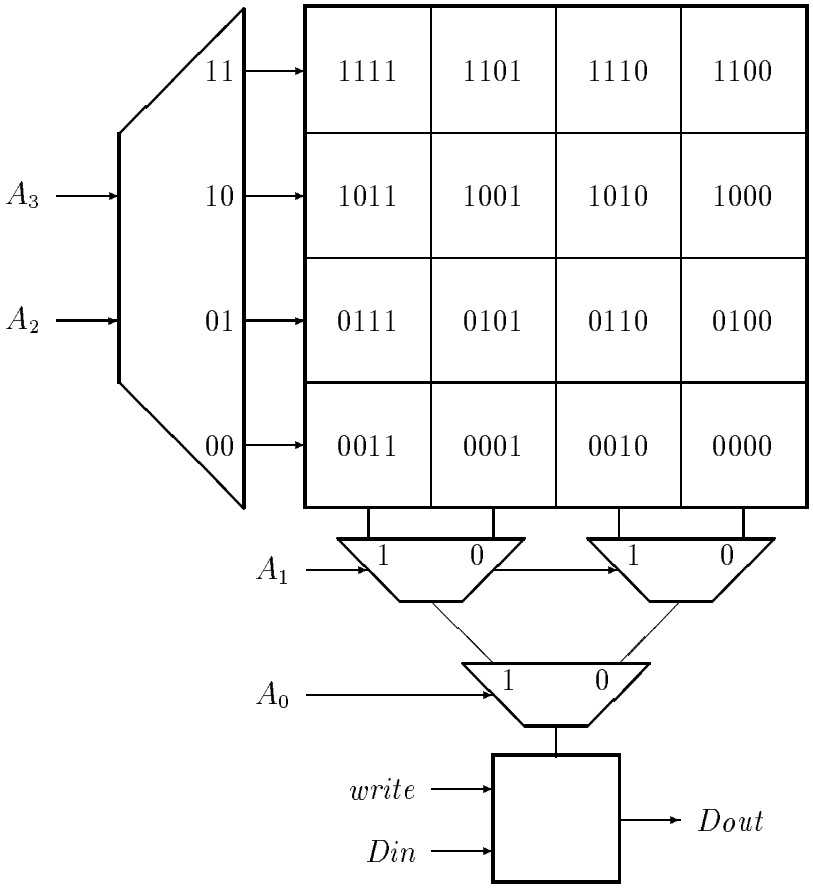

Figure 2: *Detailed Addressing Structure of a 16-Bit RAM.* Each cell is labeled with the binary representation of its address.

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 2 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.

4.1. 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. This fact is formulated as a *system invariant*

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

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 $0 \leq i < N$:

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

That is, we can begin all simulation read cycles with the word lines initialized to 0. With this refinement, 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

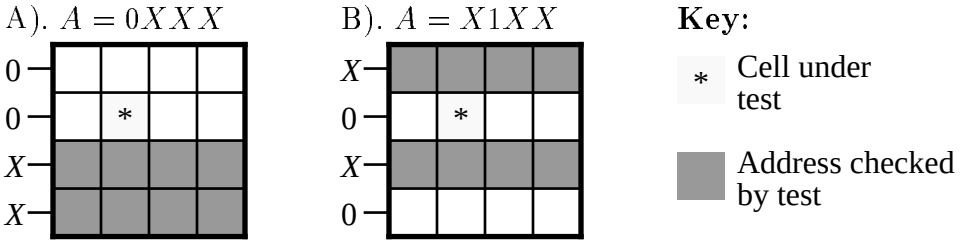

Figure 3: *Row Address Tests for Memory Location 9.* Signals on the left indicate the values on the word lines. The word line controlling cell 9 remains at 0.

of every input cycle. Devising the invariant requires a combination of analysis and experimentation. An insufficient system invariant will become immediately apparent during subsequent simulations, because output or state variables that should have Boolean values will equal X.

4.2. 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 $n/2 \leq k < n$:

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

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 3 illustrates the addressing patterns for the row address tests for memory location 9 (1001 binary) in the 16-bit RAM of Figure 2. The two address settings: $0XXX$ and $X1XX$ 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 4A shows an example of such a pattern for memory location 9 in the 16-bit RAM of Figure

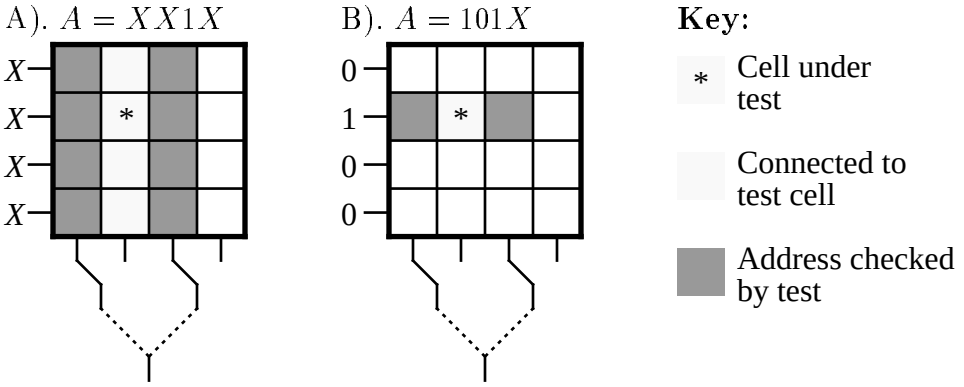

Figure 4: *Example of Initial (A) and Refined (B) Column Address Test for Memory Location 9.* The tree on the bottom indicates the connections formed by the column multiplexors, with dotted lines representing pass transistors with gate value X. In A), the word line value of X causes cell 9 to be corrupted. This is avoided in B).

2. 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 $0 \leq k < n/2$:

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

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 4B 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

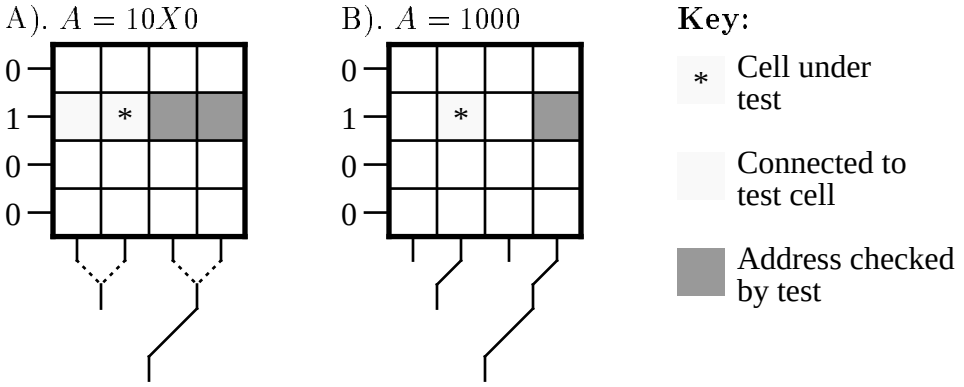

Figure 5: *Example of Refined (A) and More Refined (B) Column Address Test for Memory Location 9.* In A), a sneak path forms through the column multiplexor between cell 9 and an adjacent cell. This is avoided in B).

and the data input and output. When some of the column address lines equal X, however, the simulator finds false sneak paths throught 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 5A. 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 $0 \leq k < n/2$:

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

These tests are called the "column address" tests.

The pattern of Figure 4B shows one of the column address tests for location 9 in the 16-bit RAM of Figure 2. The other is shown in Figure

5B. 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 3 cover all possible addresses other than location 9.

Equations 1, 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.

5. 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 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 were able to exploit the bit-level parallelism available with computer logic operations to simulate up to 32 sequences in parallel on a machine with a 32 bit word size. The COSMOS preprocessor transforms a transistor

N	Transistors	Marching Test	Serial Verification	Parallel Verification
4	113	1.0s.	2.0s.	0.6s.
16	235	8.4s.	22.6s.	2.0s.
64	611	117s.	385s.	19.3s.
256	1931	30.8m.	122m.	4.4m.
1024	6875	10.4h.	47.9h.	1.5h.

Table 1: COSMOS CPU Times on DEC MicroVax-II

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. Although this would appear to be an obvious source of speed-up, most simulators make no use of bit-level 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.

6. Experimental Results

The verification methodology has been applied to memory sizes ranging from 4 to 1024 bits. The performance of the program is shown in Table 6. The last 3 columns of this table show simulation CPU times, measured on a Digital Equipment Corporation MicroVax-II. The first of these columns 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. The second shows the time to simulate the verification patterns without using bit-level parallelism. The final column shows the time to simulate the verification patterns using 32 way bit-level parallelism.

As can be seen, the 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. We estimate that the verification of a 4096-bit memory will re-

quire between 1 and 2 days of CPU time even in parallel mode. Clearly, this is approaching the limit of practicality.

7. Observations and Conclusions

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. 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, and 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 3 symbolic patterns. Although the additional overhead required by symbolic simulation would probably cause it to require longer for the RAM verification, there would be less need for circuit-dependent refinements. In addition, symbolic simulation would yield polynomial-time performance for a much wider range of circuits.

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 [6]. 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.

Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency, ARPA Order Number 4976.

References

References

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

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

[3] Bryant, R. E. Symbolic verification of MOS circuits. *1985 Chapel Hill Conference on VLSI*, Fuchs, H., Ed. Computer Science Press, Rockville, MD, 1985, 419–438.

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

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

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

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

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

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

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