Verification of Synchronous Circuits by Symbolic Logic Simulation

Randal E. Bryant
Carnegie Mellon University

Abstract. *A logic simulator can prove the correctness of a digital circuit when it can be shown that only circuits implementing the system specification will produce a particular response to a sequence of simulation commands. By simulating a circuit* symbolically*, verification can avoid the combinatorial explosion that would normally occur when evaluating circuit operation over many combinations of input and initial state. In this paper, we describe our methodology for verifying synchronous circuits using the stack circuit of Mead and Conway as an illustrative example.*

1. Verification by Simulation

Logic simulators have long been used to test for errors in digital circuit designs. Typically, however, the user only simulates a limited set of test cases and assumes that the circuit is correct if the simulator yields the expected results for all cases. Unfortunately, this form of simulation provides no guarantee 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. Conventional wisdom holds that simulators are incapable of more rigorous verification. They are viewed in the same class as program debuggers–useful tools for informal testing, but nothing more.

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.

Using a conventional simulator, a surprisingly large class of circuits can be verified in polynomial time (as measured in the circuit size). For example, an N-bit random access memory (RAM) can be verified by simulating just $O(N \log N)$ patterns with

a switch-level simulator [6]. By this means, we have successfully verified a 4K static CMOS RAM. In this verification, we exploit the property that 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.

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 [2] 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. Earlier attempts at symbolic simulation [7] had only limited success, because they did little more than trace the possible control sequences of the system. By endowing a symbolic simulator with a powerful Boolean manipulation capability [4], we derive canonical representations of the Boolean functions describing the value of every state variable and primary output in the circuit. These functions can then be compared with ones derived from the system specification.

Our first symbolic simulator MOSSYM [2] demonstrated the feasibility of symbolic switch-level simulation, verifying ALU circuits in minutes of CPU time that would have required centuries to verify by exhaustive simulation. Since that time, others have improved the efficiency of symbolic simulation significantly [9]. Most recently, we have added a symbolic capability to our switch-level simulator COSMOS [5]. On benchmarks for which MOSSYM required 10 minutes of CPU time, COSMOS requires only 11 seconds.

2. Verification Methodology

Verifying a combinational circuit by symbolic simulation is conceptually straightforward. Each circuit input is set to a different Boolean variable, and the circuit is simulated to derive representations of the Boolean functions for each primary output. These output functions are then compared to ones derived from the system specification. This technique even applies to some forms of clocked circuits, such as those using domino logic or precharged carry chains, where the outputs should only depend on the most recent inputs. Such a circuit is verified by simulating it over one complete clock cycle, with the clock inputs set to constants according to the clocking pattern, and with the remaining inputs set to Boolean variables. With the COSMOS symbolic simulator, we have verified a variety of such circuits including ALU's and an 80-bit priority encoder.

Sequential circuits, however, require a totally different verification methodology. We have shown [3] that, with only a limited set of exceptions, a sequential system can-

not be verified by simply observing its output response to a set of simulation patterns. This result holds even when using three-valued simulation. Instead, verification requires specifying how the system state is represented in the circuit. Verification then involves proving that the state transition behavior of the circuit matches that of the system specification.

We have devised a methodology for verifying synchronous circuits that involves specifying the circuit at 4 levels of abstraction:

1. The desired functionality, expressed as a state machine operating on an abstracted system state.
2. The circuit interface, expressed in terms of the clocking patterns, as well as the times within a clock cycle at which the inputs are applied and the outputs sensed.
3. The mapping between the abstract system state and the internal circuit state.
4. A transistor-level description of the circuit.

The first two parts comprise the *external* specification of the system. That is, they define the desired input/output behavior. We find it convenient to divide the external specification into these two components. The first part defines the high level system behavior in a manner that is independent of any implementation. The second part defines the details of the input/output interface including the signalling and clocking conventions. The third part of the specification provides the details we require about the internal circuit state in order to make verification by simulation possible. This is far less information than most other approaches to formal verification require. From these first three parts, we can derive a set of symbolic patterns to be simulated.

The fourth part of the system specification is used by the COSMOS preprocessor to produce an executable symbolic simulation program. In effect, the preprocessor automatically converts the detailed representation of the circuit structure into an executable representation of the circuit behavior. If the simulator produces the desired response to the symbolic patterns, then the circuit is proved correct.

3. Specification Example

We illustrate our verification methodology by verifying the nMOS stack circuit presented in Mead and Conway [8]. This circuit is an interesting test case for formal verification for several reasons:

- It uses circuitry that cannot be represented at the gate level. Hence, switch-level modeling is imperative.

- Two stack control signals are multiplexed onto a single input. Consequently, the circuit timing is tricky.
- The circuit employs a small amount of pipelining. The stack command signals for clock cycle t are supplied during the last part of cycle $t-1$ and the first half of cycle t.

It is important that any approach to formal verification be able to handle subtleties such as these, because they are frequent sources of design errors. We cannot expect designers to simplify their circuits and interfaces simply to facilitate formal verification.

3.1. High Level Specification

Specifying the desired behavior of a stack is straightforward. We assume that the stack is 1 bit wide and k bits deep. When d elements are stored in the stack, the stack locations are numbered 1 (top) to d (bottom). We will define the desired behavior of the stack in terms of a set of predicates. The definitions of these predicates will later be formalized according to the detailed circuit interface and state representation. For now, we describe the predicates informally:

$push(t)$: The stack executes a Push operation on clock cycle t.

$pop(t)$: The stack executes a Pop operation on clock cycle t.

$hold(t)$: The stack executes a Hold operation on clock cycle t.

$Depth(d,t)$: There are d ($0 \leq d \leq k$) items on the stack at end of cycle t.

$Stored(i,t,v)$: Value $v \in \{0,1\}$ is stored at stack location i ($1 \leq i \leq k$) at end of cycle t.

$Input(t,v)$: Value $v \in \{0,1\}$ is supplied to the stack input during cycle t.

$Output(t,v)$: Value $v \in \{0,1\}$ appears on the stack output during cycle t.

With these predicates, we can define the effect of a push operation on a nonfull stack on cycle t in terms of the circuit state on cycle $t-1$ and the circuit input on cycle t. That is, a push operation should cause the input to be stored in stack location 1, and the value at each stack location i to move to location $i+1$. Thus, for any d such that $0 \leq d < k$:

$$\begin{array}{l} Depth(t-1,d) \ \wedge \ \forall[1 \leq i \leq d] Stored(i,t-1,v_{i+1}) \\ \quad \wedge \ Input(t,v_1) \ \wedge \ push(t) \\ \quad \Rightarrow \\ Depth(t,d+1) \ \wedge \ \forall[1 \leq i \leq d+1] Stored(i,t,v_i) \end{array} \tag{1}$$

In this equation, the variables v_i for $1 \leq i \leq k$ are considered to be universally quantified over $\{0,1\}$.

Similarly, the effect of a pop operation on a nonempty stack should be to transfer the value at each stack location $i+1$ to location i, and for the output to equal the value popped off the top. Thus, for any d such that $1 \leq d \leq k$:

$$\begin{array}{l} Depth(t-1,d) \ \wedge \ \forall[1 \leq i \leq d] Stored(i,t-1,v_i) \ \wedge \ pop(t) \\ \Rightarrow \\ Depth(t,d-1) \ \wedge \ \forall[1 \leq i < d] Stored(i,t,v_{i+1}) \ \wedge \ Output(t,v_1) \end{array} \tag{2}$$

Finally, during a hold operation, the stack contents should not change. Thus, for any d such that $0 \leq d \leq k$:

$$\begin{array}{l} Depth(t-1,d) \ \wedge \ \forall[1 \leq i \leq d] Stored(i,t-1,v_i) \ \wedge \ hold(t) \\ \Rightarrow \\ Depth(t,d) \ \wedge \ \forall[1 \leq i \leq d] Stored(i,t,v_i) \end{array} \tag{3}$$

The above specification should hold for any implementation of a stack. We have been careful not to overspecify its behavior. We have not specified the effect of a push operation on a full stack or a pop operation on an empty stack. We also have not specified the stack output for stack operations other than pop. Details such as these vary from one stack implementation to another. If a particular application requires a specific way of handling such boundary conditions, these could also be verified.

3.2. Interface Specification

Figure 1 gives a timing diagram of the stack circuit interface. The circuit operates with a two-phase nonoverlapping clock. The stack command is specified by a pair of signals, `OP1` and `OP2`, according to the following table:

Operation	`OP1`	`OP2`
Push	1	0
Pop	0	1
Hold	0	0

As is shown in Figure 1, these two signals are multiplexed onto circuit input `OP`. That is, the stack command for clock cycle t is specified by supplying `OP1` on the `Phi2` phase of cycle $t-1$, and `OP2` on the `Phi1` phase of cycle t. The input data must be supplied during the `Phi1` phase, and the output is valid during the `Phi2` phase.

3.3. State Specification

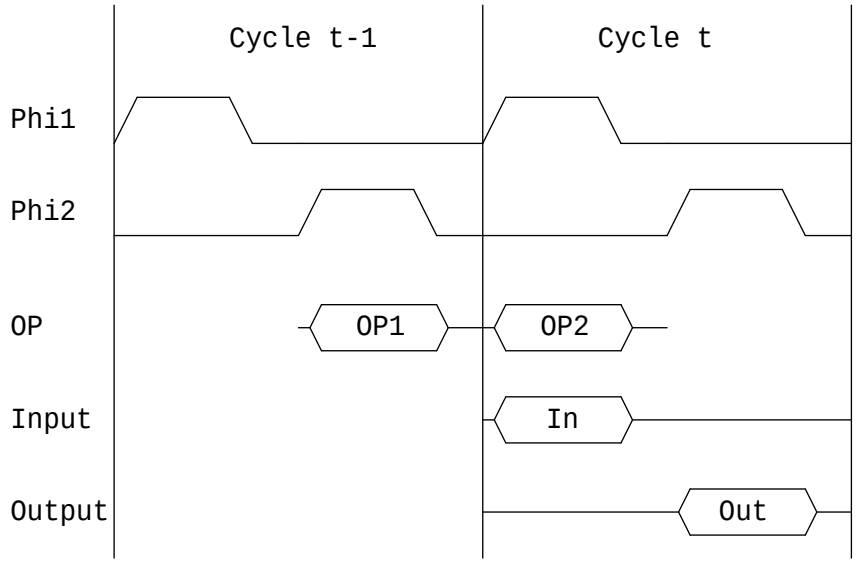

Figure 1: **Stack Circuit Timing Interface.** Control signal `OP` is pipelined 1/2 cycle ahead of the data signals `In` and `Out`.

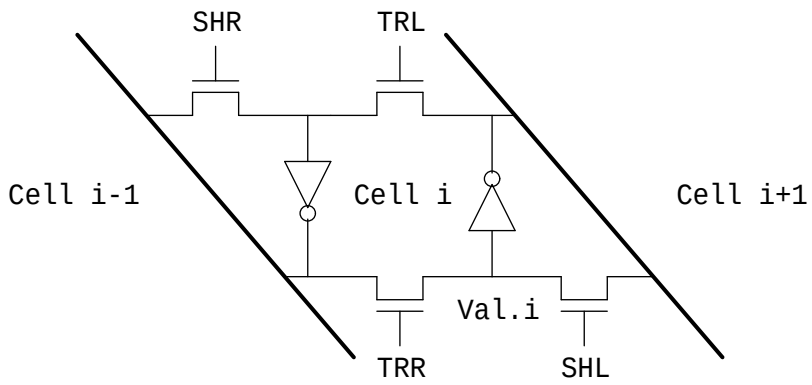

Figure 2: **Cell i of Stack.** Stack element i is stored on node `Val.`i.

Figure 2 shows a single cell of the stack circuit. The stack is created by composing these cells horizontally, numbered 1 to k from left to right. The control signals `SHR`, `TRL`, `TRR`, and `SHL` are derived from the clocks and the `OP` input by the control circuitry [8, p. 73]. The charge on the node labeled `Val`.i is considered to represent the contents of stack location i in complemented form. That is, we will consider the predicate $Stored(i, t, v)$ as specifying that at the end of cycle t, node `Val`.i is charged to logic value $\neg v$.

This particular stack implementation does not keep track of the stack depth. Instead, on a push or pop operation, it shifts whatever data happen to be stored in the stack cells. Thus, we can handle the details how the stack operates for different depths by verifying a stronger set of conditions than is required to satisfy Equations 1, 2, and 3. The modified equations omit all explicit references to the stack depth. Instead, they place constraints on a range of stack locations that covers all possible depths. The effect of a push operations becomes:

$$\begin{array}{l} Input(t, v_1) \ \wedge \ \forall[1 \leq i \leq k{-}1] Stored(i, t{-}1, v_{i+1}) \ \wedge \ push(t) \\ \quad \Rightarrow \\ \forall[1 \leq i \leq k] Stored(i, t, v_i) \end{array} \tag{4}$$

The effect of a pop operation becomes:

$$\begin{array}{l} \forall[1 \leq i \leq k] Stored(i, t{-}1, v_i) \ \wedge \ pop(t) \\ \quad \Rightarrow \\ \forall[1 \leq i < k] Stored(i, t, v_{i+1}) \ \wedge \ Output(t, v_1) \end{array} \tag{5}$$

Finally, the effect of a hold operation becomes:

$$\begin{array}{l} \forall[1 \leq i \leq k] Stored(i, t{-}1, v_i) \ \wedge \ hold(t) \\ \Rightarrow \\ \forall[1 \leq i \leq k] Stored(i, t, v_i) \end{array} \tag{6}$$

Any circuit which satisfies these modified equations must also satisfy the original equations, since the modified equations place more stringent conditions on the circuit operation.

Given the modified verification conditions, a definition of the predicate *Stored* is the only information we need about the internal structure of the circuit to carry out the verification. We do not require any details of the control circuitry, or even as much detail of the cell design as is illustrated in Figure 2.

4. Simulation Patterns

From the three parts of the system specification given in the previous section, we can derive a set of patterns for the symbolic simulator. The patterns consist of three sections, one for each stack command. Each section starts by resetting the circuit so

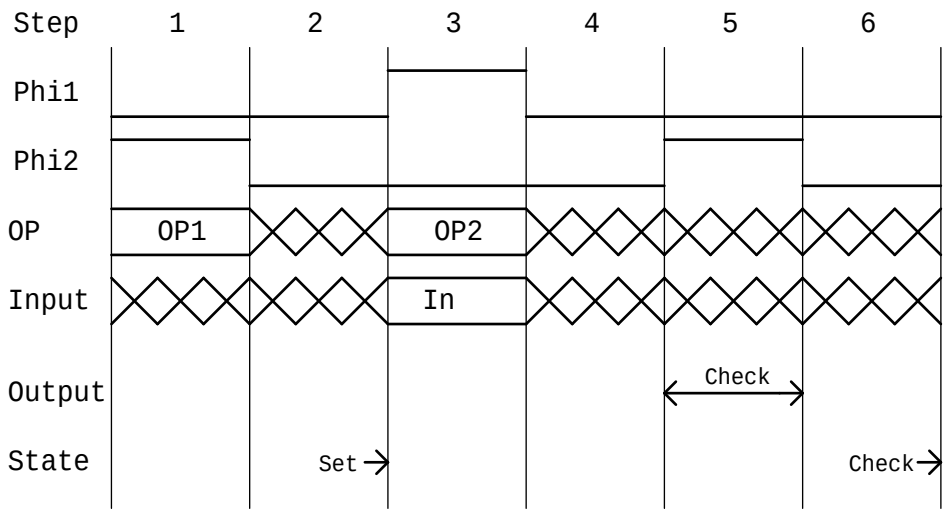

Figure 3: **General Form of Simulation Patterns**. Verifying 1 cycle of system operation requires simulating 1 1/2 cycles (6 steps) of circuit operation due to the pipelining of the control signals.

that every node has state X. It then simulates 1 1/2 clock cycles: the last half of cycle $t-1$ and all of cycle t.

Each section has the general form illustrated in Figure 3. The patterns involve simulating 6 steps: 2 for cycle $t-1$ and 4 for cycle t. During each step, an input is set either to 0 (low horizontal line), 1 (high horizontal line), X (cross hatches), or to a case-specific value (annotated double line). Thus, the clocks are operated according to a two-phase, nonoverlapping discipline. During steps 1 and 3, input `OP` is set to the appropriate values for the operation being verified. On all other steps, `OP` is set to X, indicating that the value on this input should not matter during these times. During step 3, data input `In` is set to either a variable (push) or to X. It is set to X on all other steps. When verifying the pop operation, the circuit output is monitored during step 5. The states of the stack cells are set after step 2 and are checked after step 6.

Observe how these simulation patterns use the power of three-valued modeling to express the full generality of the circuit operation. By initializing all internal nodes to X, we cover all possible operations that could have occurred prior to the last half of cycle $t-1$. By setting `OP` to X on all steps other than 1 and 3, we cover all possible operations in which this signal either makes a transition or is set to some value during these other steps. In actual use, `OP` would be set either to 0 or to 1 during step 5 to specify the `OP1` value for cycle $t+1$. By simulating the circuit with this value set to X, we cover both possibilities. Similarly, by setting the data input to X on all steps other than 3, we cover all possible times at which this signal may make a transition. Observe also that by setting and checking the internal state exactly one clock cycle apart, we maintain a consistent view of how data are stored.

The values of `OP1` and `OP2` for the three stack operations have already been specified. The following table shows the other values used for the 3 different sets of patterns:

Operation	Settings			Checks		
	`In`	`val.`i	`val.`k	`Out`	`val.`i	`val.`k
		$1 \leq i < k$			$1 \leq i < k$	
Push	v_1	$\neg v_{i+1}$	X	–	$\neg v_i$	$\neg v_k$
Pop	X	$\neg v_i$	$\neg v_k$	v_1	$\neg v_{i+1}$	–
Hold	X	$\neg v_i$	$\neg v_k$	–	$\neg v_i$	$\neg v_k$

The symbolic patterns involve k Boolean variables $v_1, \ldots, v_k$. Under the "Checks" section, '–' indicates that this value need not be checked. These patterns follow directly from our modified system specification (Equations 4, 5, and 6), the interface specification, and the state specification.

5. Experimental Results

We have successfully verified stacks of both $k = 4$ and $k = 16$. Running on a Digital Equipment Corporation Microvax-II, the symbolic simulations require only 1 and 2.3 seconds, respectively. In fact, the time complexity of the verification grows only linearly with the circuit size.

6. Observations

This example illustrates the power of symbolic simulation as a tool for formal verification. Although the verification uses a detailed, switch-level model, virtually all details of the internal circuit structure were handled automatically. We only needed to specify a mapping from the abstract system state to the state representation within the circuit. Furthermore, the simulation interface provides a natural means to specify operational details such as the clocking scheme, the input and output timing, etc. Using the power of three-valued modeling, we could verify circuit operation for cycle t in such a way that it would be compatible with any operations on cycles $t-1$ and $t+1$.

Verifying other implementations of stack circuits requires creating different simulation patterns according to their interface and state specifications. For example, in a stack implemented as a RAM plus a pointer to the top of stack, stack location i in the system specification would correspond to RAM address $d - i$, when the stack depth equals d. We could introduce Boolean variables $d_0, \ldots, d_k$, where $d_i = 1$ indicates that the stack depth equals i. The symbolic simulation patterns would then involve Boolean formulas over $v_1, \ldots, v_k$ and $d_0, \ldots, d_k$.

Whereas our approach involves defining a mapping from specification states to the circuit states, Bose and Fisher have developed a methodology where the mapping is made from the circuit state to the specification state [1]. They have verified circuits containing very subtle forms of pipelining. More experience is required to see how best to incorporate the detailed state representation.

In our example, we derived the symbolic simulation patterns manually. We hope to make this process more automated. Doing so requires a more formal notation to express the interface specification.

We feel we have just begun to tap the power of symbolic simulation in terms of both expressive power and performance. We hope to create a tool that can be used to verify a wide range of complex VLSI systems in a manner that circuit designers find intuitive.

7. Acknowledgements

Derek Beatty and Carl Seger have provided valuable assistance in this work. This research was supported by the Defense Advanced Research Projects Agency, ARPA Order Number 4976.

References

[1] S. Bose, and A. L. Fisher, "Verifying pipelined hardware using symbolic logic simulation," *International Conference on Computer Design*, IEEE, 1989.

[2] R. E. Bryant, "Symbolic verification of MOS circuits," *1985 Chapel Hill Conference on VLSI*, 1985, 419–438.

[3] R. E. Bryant, "Can a simulator verify a circuit?", in *Formal Aspects of VLSI Design*, G. J. Milne, and P. A. Subrahmanyam, *eds.*, North-Holland, 1986, 125–126.

[4] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677–691.

[5] 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.

[6] R. E. Bryant, "Verifying a static RAM design by logic simulation," *Fifth MIT Conference on Advanced Research in VLSI*, 1988, 335–349.

[7] J. A. Darringer, "The application of program verification techniques to hardware verification," *16th Design Automation Conference*, 1979, 375–381.

[8] C. A. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[9] D. S. Reeves, and M. J. Irwin, "Fast methods for switch-level verification of MOS circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. CAD-6, No. 5 (Sept., 1987), 766–779.