CMU-CS-85-120

Symbolic Verification of MOS Circuits[1]

Randal E. Bryant[2]

Department of Computer Science
Carnegie-Mellon University

Abstract

The program MOSSYM simulates the behavior of a MOS circuit represented as a switch-level network *symbolically*. That is, during simulator operation the user can set an input to either 0, 1, or a Boolean variable. The simulator then computes the behavior of the circuit as a function of the past and present input variables. By using heuristically efficient Boolean function manipulation algorithms, the verification of a circuit by symbolic simulation can proceed much more quickly than by exhaustive logic simulation. In this paper we present our concept of symbolic simulation, derive an algorithm for switch-level symbolic simulation, and present experimental measurements from MOSSYM.

[1]This paper was presented at the 1985 Chapel Hill Conference on VLSI.

[2]This research was funded by the Defense Advanced Research Contracts Agency ARPA Order Number 3597.

1 Introduction

Conventional simulators verify the correct functionality of a logic design only for the particular data simulated. The user is left with the difficult task of choosing data that will detect every design error and deciding when a sufficient set of tests have been applied. In practice, a large fraction of the errors in a design are successfully detected using simulators, either because the user has some intuition about what data patterns are most likely to cause difficulty, because the error causes improper behavior for many different patterns, or because the user simulated large numbers of patterns. In some cases, however, errors remain undetected until well into the design process or even after the circuit has been manufactured, at which time the cost of correcting an error can be extremely high.

To verify the correctness of a digital design in a more rigorous fashion, we propose that they be simulated *symbolically*. Our concept of a symbolic simulator resembles a conventional logic simulator whereby a description of the circuit is loaded into memory, the user specifies a sequence of patterns to be applied to the inputs of the circuit, and the program models how the circuit would behave for these patterns. However, unlike a conventional simulator in which the patterns consist of only the constants 0 and 1 and the program models only how the circuit would behave for these specific data, a symbolic simulator can accept patterns consisting of *Boolean variables* (as well as the constants 0 and 1), and the program computes the *Boolean functions* describing the behavior of the circuit for the set of all possible data represented by these variables. By setting some inputs to constants and some to to variables, a symbolic simulator can perform conventional simulation (all inputs set to constants), symbolic analysis (all inputs set to variables), or a hybrid of the two. Furthermore, a symbolic simulator can represent the behavior of a sequential circuit by computing the sequence of Boolean functions that would appear at each output as a function of the sequences of variables that have been applied to each input. Thus a symbolic simulator has considerably greater power than a conventional simulator.

We believe that a circuit verification program styled as a simulator provides an attractive method for validating a logic design. A simulator provides an interactive environment familiar to most logic designers. The user can try different types of input patterns, and probe points in the network at various points in the simulation to ensure that the circuit functions as expected, or if not, to pinpoint the source of the error. With a simulator, the user can conveniently specify information about a circuit that is not contained in the schematic, such as how the clocks are operated in a sequential circuit. Furthermore, given that a symbolic simulator is able to solve NP-hard problems (e.g. Boolean satisfiability, tautology, and equivalence) we must anticipate that its time complexity will in the worst case be exponential in the number of symbolic variables, barring a major breakthrough in computer science. [12] Hence it may be impractical to verify a circuit with all inputs set to variables, but the user can adopt a hybrid approach in which some inputs are set to constants and some are set to variables. In fact a hybrid approach in which data inputs are assigned symbolic values and control inputs are assigned constant values provides the most natural method to verify many circuits.

For circuits described in terms of logic gate networks, the implementation of a symbolic simulator is conceptually (although not computationally) straightforward, since the function computed by a network is given by the composition of the functions computed by the gates. Our interest, however, is in verifying MOS circuits represented at the switch level as networks of charge storage nodes connected by resistive transistor switches. Several different logic simulators based on the switch-level model have been developed, [6, 23] but until now only limited success has been achieved in deriving the function computed by a switch-level network symbolically. What results have been published suffer from some combination of three shortcomings: a weak logic model, a poor algebraic specification, or an inherently inefficient algorithm. Furthermore, except for Barrow [3], these results are either purely theoretical [9] or are directed at a task less ambitious than ours, [11, 14, 23] namely to extract a functional description of a transistor subcircuit for use in a more conventional simulator. In terms of models, that used by Barrow [13] is extremely weak, able to represent only a limited class of MOS circuits. The models used by Ditlow, *et al* [11] and Terman [23] while better, are somewhat weak in their ability to model ratioed circuits and charge sharing. In terms of algebraic specification, the methods of Barrow, Ditlow, and Terman contain flaws, especially in characterizing the effect of unknown states. The methods of Chen [9] and Hajj [14], on the other hand, derive correct results, but require translations between several different algebras to get them. While these translations are valid mathematically, they are difficult to implement computationally, because it is hard to remove the artifacts of the intermediate algebras from the final result. In particular, these mixed algebras make it difficult to implement *functional composition*, in which the inputs to a logic block are the functions produced as output by other blocks. Such a capability is vital to true symbolic simulation. Finally, the methods implemented by Ditlow, Hajj, and Terman are inherently inefficient, since they analyze a subcircuit by analyzing all possible simple paths formed by the transistors. For many pass transistor networks (e.g. the Tally circuit of Mead and Conway [20]), the number of such paths grows exponentially with the number of transistors. Thus these methods are not even polynomial in the number of algebraic operations.

In this paper we describe an algorithm for a symbolic, switch-level simulator and present experimental results obtained from an implementation by the program MOSSYM. Our logic model is the same as presented in a previous paper [6] and implemented in software by the simulator MOSSIM II [4] and in hardware by the MOSSIM Simulation Engine. [10] This model can represent the most general class of MOS circuits for which a symbolic analysis has been attempted. The behavior computed for a circuit is identical to what would be obtained by exhaustively simulating the circuit with MOSSIM II for all combinations of data represented by the input variables, even when unknown states are present. All equations are specified in a Boolean algebra, having the advantage that the input, intermediate results, and output are all elements of the "natural" algebra for symbolic simulation. We accomplish this by encoding the three logic states 0, 1, and X as pairs of Boolean values, and by accounting for the different signal strengths in a circuit in the structure of the equations to be solved rather than in the elements of the algebra. We then solve the equations by a transitive closure algorithm where the algebraic operations are implemented by a heuristically-efficient set of Boolean function manipulation routines [7]. Our experimental results to date indicate that our approach provides reasonable performance even for circuits of significant complexity.

2 Symbolic Simulation

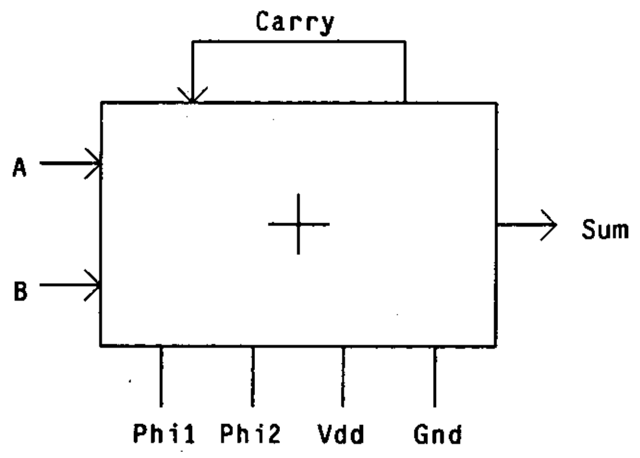


Figure 1. Block Diagram of Serial Adder

Let us further explain our concept of symbolic simulation by means of an example. Figure 1 shows the block diagram for a serial adder with data inputs A and B, clock inputs Phi1 and Phi2, and power and ground connections Vdd and Gnd. Figure 2 shows a stylized symbolic simulation session to verify that this circuit correctly adds two 8-bit numbers $a_7, \ldots, a_0$ and $b_7, \ldots, b_0$ with a carry input *cin*. We say "stylized" because our current simulator cannot print the output functions in the form shown. In fact, this form of printout becomes unreadable for functions of significant complexity. Instead, the user should exploit the symbolic manipulation capabilities of the program by having it test the output functions for equivalence with functions generated from the Boolean equations for addition. As a future enhancement of the simulator, we plan to include an interpreter for a simple Boolean expression language. Given a sufficiently powerful language, the user would never need to print Boolean formulas. In the figure, lines prefixed with ">" indicate commands typed by the user, while lines with prefixes of the form "*n*.4|" are printed by the simulator giving the functions on the specified nodes after the 4th phase of the *n*th clock cycle.

In this session, the power and ground connections are set to constants 1 and 0, respectively, and a series of clock cycles are simulated, where each cycle consists of 4 settings of the clock inputs corresponding to a two-phase, nonoverlapping clock discipline. On the first cycle, both data inputs are set to the carry input *cin* to force this value into the feedback loop of the circuit. The Sum output still equals X, reflecting the initially indeterminate system state, but the user can observe the value on node Carry and confirm that it now equals *cin*. Then the successive bits of the data words are applied to the circuit inputs, and finally both data inputs are set to 0 so that the final carry output will appear on the circuit output.

```
>read serial.ntk
>set vdd:1 gnd:0
>clock phi1:1000 phi2:0010
>watch Sum
>boolean $cin, a_0, b_0, a_1, b_1, \ldots, a_7, b_7$
>set A:$cin$, B:$cin$
>cycle
1.4| Sum:X
>get Carry
1.4| Carry:$cin$
>set A:$a_0$, B:$b_0$
>cycle
2.4| Sum:$a_0 \oplus b_0 \oplus cin$
>set A:$a_1$, B:$b_1$
>cycle
3.4| Sum:$a_1 \oplus b_1 \oplus (a_0 \cdot b_0 + (a_0 + b_0) \cdot cin)$

...

>set A:$a_7$, B:$b_7$
>cycle
9.4| Sum:$a_7 \oplus b_7 \oplus (a_6 \cdot b_6 + (a_6 + b_6) \cdot ( \ldots ))$
>set A:0, B:0
>cycle
10.4| Sum:$a_7 \cdot b_7 + (a_7 + b_7) \cdot (a_6 \cdot b_6 + (a_6 + b_6) \cdot ( \ldots ))$
```

Figure 2. Example Symbolic Simulation Session

This example further demonstrates why we advocate a simulation framework for symbolic circuit verification. First, the user can describe a clocking methodology by simply specifying the values to be applied to the clock inputs during the clock cycle. Such a capability is vital for verifying MOS circuits, given the wide variety of clocking methodologies used and the vital role of the clocks in determining the behavior of circuits containing such structures as precharged logic and dynamic registers. Second, the user can specify how the circuit is initialized by simulating the normal reset sequence, such as in our example by setting both inputs to the carry input value. Most significantly, a symbolic simulator transforms the concept of symbolic verification from a static analysis into a dynamic computation of the circuit's behavior over time.

Our example also illustrates a weakness of circuit verification by symbolic simulation. While this session would verify that the circuit correctly adds two 8-bit numbers, it does not establish the correctness for other word sizes. To do so, we would require a program capable of theorem-proving techniques such as proof by induction. This, of course, would move us from the realm of problems that are merely NP-hard to those that are uncomputable, but it may be useful to implement some form of inductive reasoning.

3 Algorithm

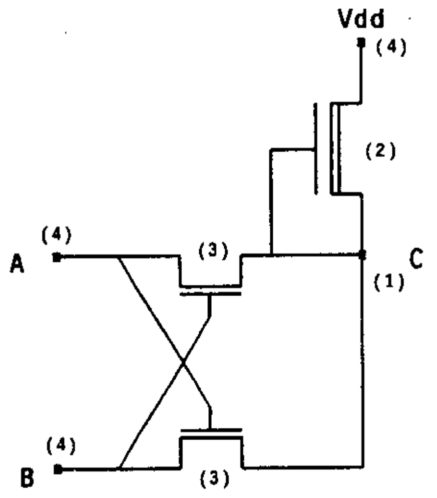
Our algorithm for switch-level simulation can be derived by a three step process. First, we specify the behavior of a switch-level network in terms of Boolean equations. This specification follows directly from our method of describing the function of a switch-level network in terms of the signal paths in a switch graph. Next, we abstract these equations to the Boolean algebra with domain consisting of the set of all Boolean functions over the symbolic variables. Finally, we implement the equations with Boolean transitive closure algorithms and routines for manipulating Boolean functions symbolically. Of these steps, the first requires the greatest effort, but when done properly the remaining two steps follow directly.

We advocate the use of Boolean algebra for symbolic analysis, rather than an algebra combining the notions of logic value and signal strength in the style of Hayes [15], or the mixed integer-logic algebra used by Hajj. Boolean algebra is the natural algebra for verifying a logic circuit. In general, a designer thinks about a circuit in terms of the Boolean values it produces as a function of the Boolean values applied to the inputs. Information about signal strengths is important only when determining the behavior of the circuit, and hence we incorporate this information into the structure of the equations to be solved rather than in the domain of the algebra. Moreover, a Boolean algebra obeys properties that permit powerful techniques for simplification, reduction to canonical form, and testing for equivalence. We can draw from a well-developed body of knowledge on how to represent and manipulate Boolean functions. Finally, with Boolean algebra we can apply *functional abstraction* in which we extend a result from the domain $\{0,1\}$ to the algebra with domain equal to the set of all Boolean functions over a set of Boolean variables. This abstraction technique allows us to transform a conventional switch-level simulation algorithm into a symbolic algorithm.

3.1 The Switch-Level Model

Our approach to switch-level modeling has been described in detail elsewhere. [6] In this paper we will only summarize the key ideas. A switch-level network consists of a set of nodes and a set of transistors. Nodes of type *input* are connected to power sources external to the chip acting much like voltage sources in an electrical network. All other nodes are of type *storage* that, like capacitors in an electrical network, retain their states in the absence of applied inputs and can share charge with other storage nodes. Each storage node is assigned an integer *size* from the set $\{1,2,\ldots,k\}$ indicating (in a highly simplified way), its capacitance relative to nodes with which it may share charge. That is, when a set of storage nodes share charge (due to connections by conducting transistors), the state(s) of the largest node(s) determine the outcome. Input nodes are indicated by size $w > k$. The voltage on node n is represented by its state $state(n) \in \{0,1,X\}$, with 0 and 1 corresponding to low and high voltage levels, respectively, and X corresponding to an indeterminate voltage between low and high indicating an uninitialized network state or an error condition caused by a short circuit or charge sharing.

A transistor is a device with terminals labeled, "gate", "source", and "drain" that acts as a resistive switch connecting the source and drain nodes controlled by the state of the gate node. All transistors are viewed as bidirectional elements with no predetermined direction of information or current flow. A transistor has a *type* indicating the conditions under which it will become conducting and a *strength* indicating (in a highly simplified way) its conductance relative to other transistors in a ratioed circuit. Transistor type **d** denotes a depletion mode device that always conducts. Transistor type **n** denotes an n-channel device that conducts when its gate is in state 1, while transistor type **p** denotes a p-channel device that conducts when its gate is in state 0. When the gate node of an **n** or **p** transistor is in state X, the transistor is modeled as having arbitrary conductance between fully conducting and open circuited. Transistor conductances are modeled by assigning each transistor an integer *strength* from the set $\{k+1, k+2, \ldots, w-1\}$. In a ratioed circuit consisting of paths of conducting transistors from several input nodes to a storage node, the state of the node is determined by the state(s) of the input node(s) connected by maximum strength paths, where the strength of a path equals the minimum transistor strength in the path. Transistor conduction levels are represented by states 0 (nonconducting), 1 (fully conducting), and X (between nonconducting and conducting).



Node sizes and transistor strengths are shown in parentheses.
Figure 3. An nMOS Exclusive-Nor Circuit

As an example, Figure 3 shows the switch-level representation of an nMOS exclusive-nor circuit. This network has input nodes Vdd, A, and B, while the output C is a storage node of size 1. The two pulldown transistors are type **n** and have strength 3, while the pullup transistor is type **d** with strength 2, indicating that a path to A or B through one of the pulldowns will override the path to Vdd through the pullup.

The behavior of a switch-level network is specified by its *steady state response* function. Intuitively, this function can be explained as follows. Given the states of the input nodes, the initial states of the storage nodes, and the states of the transistors, the transistors in the 1 and X states create conducting paths from input nodes to storage nodes and between pairs of storage nodes, causing the storage nodes to attain new voltage levels. The steady state response for a node equals the state (0, 1, or X) this node would attain if the transistors were held fixed long enough for the nodes to stabilize. When transistors or nodes in the X state are present, we define the steady state response on a node as 0 or 1 if and only if it would attain this unique state regardless of the voltages of the nodes and the conductances of the transistors in the X state, and as X otherwise.

For reasons of efficiency and to implement different timing models, a switch-level network can be divided into *dynamic components*, and the steady state response of each component can be computed independently. More formally, for a given set of transistor states, a component consists of a set of storage nodes connected by transistors in the 1 or X state, along with any connected input nodes as well as the connecting transistors. Note that a given input node can be in more than one component, and that the partitioning changes dynamically due to changing transistor states. Let us define the *updating* of a component as the process of setting the component transistors according to the states of their gate nodes, computing the steady state responses of the component nodes, and setting these nodes to their steady state values.

Given a method for computing the steady state response of a component, a switch-level simulator (either conventional or symbolic) can utilize several different schemes for ordering the component computations, effectively giving different models of circuit delay. The choice of an appropriate delay model involves a compromise between the goals of generality (i.e. what class of circuits can be simulated), accuracy with respect to the actual circuit delays, and computational efficiency. In the simple *unit delay* method of MOSSIM II, the effect of a change in input values is simulated by performing repeated iterations each of which involves updating the components containing transistors with gate nodes that changed state in the previous iteration. This approach has the advantage of reasonable generality and efficiency, although it is unsuitable for predicting the actual circuit delays. In the *rank-order* delay model developed by Näher [17] generalizing a method by Lipton, Sedgewick, and Valdes [18], the simulator dynamically orders the components of the circuit such that each component need be updated only once each time the input nodes are changed. This technique applies only to a restricted class of circuits in which for any valid setting of the clocks, the circuit must contain no static feedback paths and must be race-free. By using an efficient set-union algorithm [22] the partitioning and ordering of components can be performed in nearly linear (in the number of transistors) time. This approach gives better efficiency than the unit delay model at a cost of less generality. Finally, in a *computed delay* model, the simulator approximates the actual circuit timing by computing the time required for a node to reach its steady state response using a simplified model of the electrical circuit. [23] This approach provides greater accuracy, but at a cost of greater computational effort.

For symbolic simulation, the factors influencing the selection of a timing model carry different weight than with conventional simulation. First, a computed delay model in which the delays can be data-dependent (e.g. different rising and falling delays) would be impractical given that the values propagated by the simulator represent sets of possible data values. Second, the cost of computing a component's steady state response is significantly higher with symbolic simulation, and hence we wish to minimize these computations. For this reason, we have selected the rank order model in the current implementation of MOSSYM. As future work, we would like to extend this method to circuits containing positive static feedback. To verify the correctness of the timing behavior of a circuit, we advocate the use of ternary simulation [5, 8, 17] to prove freedom from races and hazards in conjunction with static timing analysis [16, 21] to verify adequate performance.

The steady state response for a component in a switch-level network can be expressed mathematically in terms of the the paths between nodes formed by the conducting transistors. This approach provides a uniform representation for the variety of different ways logic states are formed in MOS circuits, including stored charge, charge sharing, and both ratioed and complementary logic. For a given set of transistor states, a *switch graph* is constructed by associating a vertex with each node and an undirected edge with each transistor in the 1 or X state between the vertices corresponding to the source and drain nodes. A *rooted path p* is a directed path in a switch graph originating at vertex $Root(p)$, terminating at vertex $Destination(p)$ and consisting of a (possibly empty) set of edges $Edges(p)$. The *strength* of path p, denoted $|p|$ is defined as

$$|p| \;=\; \min[Size(Root(p)),\ \min\{Strength(e) \mid e \in Edges(p)\}]$$

where for vertex v, $Size(v)$ equals the size of the corresponding node and for edge e, $Strength(e)$ equals the strength of the corresponding transistor. A rooted path represents a source of charge from its root to its destination with driving power indicated by its strength. Rooted paths can be classified into three types according to their strength. A path with strength $1 \le |p| \le k$ represents a source of stored charge from a storage node with an approximate capacitance determined by the size of this node. Note that for every storage node there is a path representing the stored charge already on the node having root and destination equal to the corresponding vertex and containing no edges. A path with strength $k < |p| < w$ represents a source of current from an input node with an approximate conductance determined by the strength of the weakest transistor in the path. Finally a path with strength $|p| = w$ must have root and destination corresponding to an input node and contain no edges. Such a path represents the external current supplied to the input node.

In general, the steady state response of a node depends on only a subset of paths to the corresponding vertex in the switch graph, namely those which are not blocked. A *definite* path is defined as a rooted path p such that no edge in $Edges(p)$ corresponds to a transistor in the X state. A path p is said to be *blocked* if for some initial segment p' of p (i.e. $Root(p') = Root(p)$ and $Edges(p') \subseteq Edges(p)$) and for some *definite* path q, $Destination(p') = Destination(q)$ and $|p'| < |q|$. Intuitively, a path is blocked if the source of charge it represents would be overridden at some intermediate node in the path. Define the path relation $\mathcal{P}$ as $m \mathcal{P} n$ if there is an unblocked path p in the switch graph with $Root(p)$ corresponding to node m and with $Destination(p)$ corresponding to node n. Then the steady state response on node n is given by the equation

$$steady(n) \;=\; \text{l.u.b.}\{state(m) \mid m \mathcal{P} n\},$$

where l.u.b. represents the least upper bound over the ordering $0<X$ and $1<X$. In other words, if all unblocked sources of charge to a node drive it to 0 (or to 1), then the steady state response equals 0 (or 1). Otherwise, if the node is driven by conflicting sources or by sources of unknown value, the steady state response equals X. It can be shown that this characterization of the steady state response provides an accurate modeling of the effects of unknown states as well as several important mathematical properties.

3.2 Equation Specification

To specify the steady state response for a circuit in terms of operations in a Boolean algebra, we must devise a means of dealing with the three possible state values and the effects of different signal strengths. We do this by encoding the three state values as pairs of Boolean values, and by incorporating the strength effects into the structure of the equations to be solved. In the ensuing discussion, we use the symbol $+$ to indicate Boolean Or, $\cdot$ to indicate Boolean And, and $\neg$ to indicate Complement. Furthermore, we will indicate the Or of a set of values A as $\sum_{a \in A} a$, where if the set is empty, the sum equals 0.

First, a state value $y \in \{0,1,X\}$, is represented by two Boolean values $y1, y0 \in \{0,1\}$ by the encoding shown below.

y	$y1$	$y0$
1	1	0
0	0	1
X	1	1

With this encoding, for a set V consisting of elements $v \in \{0,1,X\}$, if $y = \text{l.u.b.}(V)$, then

$$y1 \;=\; \sum_{v \in V} v1$$

$$y0 \;=\; \sum_{v \in V} v0$$

Thus we will specify the state of a node n by Boolean values $state1(n)$ and $state0(n)$ and develop methods to compute the steady state response values $steady1(n)$ and $steady0(n)$. For a transistor t we can define Boolean functions $def(t)$ and $pot(t)$ indicating the conditions when the transistor is definitely conducting (in state 1) or potentially conducting (in state 1 or X) depending on the state of the gate node n and the transistor type as follows:

type	$def(t)$	$pot(t)$
n	$\neg state0(n)$	$state1(n)$
p	$\neg state1(n)$	$state0(n)$
d	1	1

Suppose we wish to compute the steady state values for a component containing nodes $\{n_1, \ldots, n_q\}$ and transistors collected into sets of the form $T_s(i, j)$ consisting of those transistors of strength greater than or equal to s having source and drain nodes n_i and n_j. To account for the different strength levels, we solve 3 Boolean equations at each node n for each strength s: $steady1_s(n)$ (respectively $steady0_s(n)$) indicating the conditions under which the node is driven to 1 (respectively 0) by a path of strength greater than or equal to s with no blocking path of strength greater than s, and $block_s(n)$ indicating the conditions under which the node will be the destination of a definite path of strength greater than or equal to s. From these values we can compute the steady state values as

$$steady1(n) = \sum_{s=1}^{w} steady1_s(n)$$

$$steady0(n) = \sum_{s=1}^{w} steady0_s(n)$$

These equations can be expressed in in terms of Boolean matrix operations. To compute the values of $block_s(n)$, construct a $q \times q$ matrix D with entries d_{ij} equal to

$$d_{ij} = \sum_{t \in T_s(i,j)} def(t),$$

and compute

$$B = I + D + D \cdot D + \cdots + D^{q-1},$$

i.e. the reflexive transitive closure of D. From this we obtain

$$block_s(n_i) = \sum_{1 \leq j \leq q,\, Size(n_j) \geq s} b_{ij}$$

To compute the values of $steady1_s(n)$ and $steady0_s(n)$, construct a $q \times q$ matrix E with entries

$$e_{ij} = \neg block_{s+1}(n_i) \cdot \sum_{t \in T_s(i,j)} pot(t)$$

for $i \neq j$, and with diagonal entries

$$e_{ii} = \neg block_{s+1}(n_i),$$

where by definition, $block_{w+1}(n) = 0$. That is, e_{ij} specifies the conditions under which the switch graph could have an edge of strength greater than or equal to s from node n_j to node n_i but no blocking path to node n_i of strength greater than s, while e_{ii} specifies the conditions under which a node could have a path to itself that is not blocked by a path of strength greater than s. Now let

$$P = E + E \cdot E + \cdots + E^{q-1},$$

i.e. the irreflexive transitive closure of E. From this we can obtain:

$$steady1_s(n_i) = \sum_{1 \leq j \leq q,\, Size(n_j) \geq s} p_{ij} \cdot state1(n_j)$$

$$steady0_s(n_i) = \sum_{1 \leq j \leq q,\, Size(n_j) \geq s} p_{ij} \cdot state0(n_j)$$

Thus, by solving a total of $3w$ equations in a Boolean algebra (some of which can be simplified or eliminated), we achieve the same effect as the simulator MOSSIM II does by solving 3 equations in an algebra of path strengths.

3.3 Abstraction

To implement a symbolic simulator, we simply extend the results of the previous section by solving the equations in the Boolean algebra having as domain the set of all Boolean functions over the symbolic variables. The simplicity of this step masks the importance of the result, a tribute to the power of Boolean algebra.

Consider a set of v Boolean variables. These variables are introduced by the user during the course of a symbolic simulation, such as we showed in the example simulation session by the command

```
>boolean cin,a0,b0,a1,b1, . . . ,a7,b7
```

Let D_v denote the set of all Boolean functions over the symbolic variables, i.e. $D_v = \{f: \{0,1\}^v \rightarrow \{0,1\}\}$. The algebra $B_v = \langle D_v, +, \cdot, \neg, \mathbf{0}, \mathbf{1} \rangle$, consisting of these functions, the operations of Or, And, and Complement on these functions, and the constant functions $\mathbf{0}$ and $\mathbf{1}$ that always yield the values 0 and 1, respectively satisfies the properties of a Boolean algebra.

Suppose we let the "Boolean values" of the previous section range over the members of D_v including the node states, matrix entries, and so on, where for each instance of 0 and 1 in these equations, we substitute the functions $\mathbf{0}$ and $\mathbf{1}$. Furthermore, we extend the definition of a dynamic component as a set of nodes connected by transistors t for which $pot(t) \neq \mathbf{0}$. To set an input node n to a Boolean variable a, we assign a to $state1(n)$ and $\neg a$ to $state0(n)$. To set an input node to the constant 1, we assign $\mathbf{1}$ to $state1(n)$ and $\mathbf{0}$ to $state0(n)$, and similarly to set the node to the constants 0 or X. By solving the equations of the previous section in the algebra B_v we simulate the circuit symbolically, with the node states being functions of the symbolic variables. If for some node n, the simulator computes functions $state1(n) = f$, and $state0(n) = \neg f$, then the behavior of the circuit at this node is to compute the function f. On the other hand, if the two state functions are not complementary, then for some sets of input data the circuit will produce an X on this node, indicating either an uninitialized state variable or an error due to a short circuit or charge sharing.

In fact, our technique is more general than we have described, because we can simulate a circuit with the two state functions of each input node set to arbitrary elements of D_v. For example, we could apply some function f to node n by assigning f to $state1(n)$ and $\neg f$ to $state0(n)$. This can be useful when evaluating part of a system that receives inputs from another part rather than from the system inputs. As another example, if we wish to evaluate the *ternary* behavior of a circuit, i.e. its behavior when some inputs equal X, we can introduce two Boolean variables for each node n, $a1$ and $a0$, and assign these variables to $state1(n)$ and $state0(n)$. As a final example, by interpreting the X state as indicating a transition from 0 to 1 or from 1 to 0, the method of ternary simulation [5, 8, 17] can be used detect potential circuit timing errors. With ternary simulation, the effect of an input node n changing from state y to z is simulated by first simulating the circuit with n set to l.u.b.(y, z) and then with n set to z. We can apply this technique with symbolic simulation with y and z equal to Boolean functions by first simulating the circuit with $state1(n)$ set to $y + z$ and $state0(n)$ set to $\neg y + \neg z$, and then with $state1(n)$ set to z and $state0(n)$ set to $\neg z$. Ternary symbolic simulation provides a very strong form of verification: that the circuit will function correctly under all possible input and timing conditions. Thus, by applying the technique of functional abstraction, we obtain a simulator with a rich set of capabilities.

To illustrate our technique with an example, consider the switch-level representation of the exclusive-nor circuit shown in Figure 3. To analyze the circuit for Vdd set to 1, A and B set to Boolean variables a and b, respectively, and C initially in state X we would compute for node C:

$$\begin{array}{lll} steady1_4 = 0 & steady0_4 = 0 & block_4 = 0 \\ steady1_3 = a \cdot b & steady0_3 = a \cdot \neg b + b \cdot \neg a & block_3 = a + b \\ steady1_2 = \neg(a+b) & steady0_2 = 0 & block_2 = 1 \\ steady1_1 = 0 & steady0_1 = 0 & block_1 = 1 \end{array}$$

$$steady1 = \neg(a+b) + a \cdot b = \neg(a \oplus b)$$

$$steady0 = a \cdot \neg b + b \cdot \neg a = a \oplus b$$

Hence, the circuit behaves as advertised.

3.4 Implementation

To implement a symbolic simulator, we require a method of solving the equations for the steady state response, as well as a method of representing and manipulating Boolean functions symbolically.

Suppose a component contains n nodes, t transistors, and is being evaluated symbolically in the Boolean algebra of v variables. We will describe possible methods to compute the steady state response and measure their time complexity in terms of the total number of algebraic operations. We must solve matrix equations of the general form $y = A^* b$, where A is an $n \times n$ Boolean matrix with no more than t nonzero entries. We could use Warshall's algorithm [1] to compute A^*. This method can require $O(n^3)$ operations, even if the component is relatively sparse (i.e. $t \ll n^2$). For sparse networks we obtain better performance by the Jacobi relaxation method, performing iterations of the form:

$$y^0 = b$$

$$y^i = y^{i-1} + A y^{i-1}$$

until convergence (a maximum of n iterations). Better yet, by applying Gauss-Seidel relaxation, and by exploiting the sparseness of the matrix, we can solve the equation with $O(t \cdot \min(n, 2^v))$ operations. Assuming the number of strengths w is constant, the total complexity of computing the steady state response will therefore be $O(t \cdot \min(n, 2^v))$ algebraic operations. For the case where $v = 0$, i.e. we are performing conventional switch-level simulation, each algebraic operation requires constant time and hence the time complexity is $O(t)$. In practice, however, a method that solves for the steady state response in an algebra of path strengths [6] achieves somewhat better performance.

Finally, we require a method of representing Boolean functions, to perform the operations of And, Or, and Complement, and to test for equivalence. A variety of different methods have been described for this task, all of which have time complexity $\Omega(2^{v})$ for functions of v variables. In fact, many researchers believe that no polynomial-time algorithm for this task exists. In practice, however, reasonable success has been achieved for representing and manipulating the Boolean functions arising in applications related to logic design. For our implementation we chose an approach in which Boolean functions are represented by directed acyclic graphs [7] in a manner similar to the notation presented by Akers [2]. This representation has the advantage that many common functions, such as the functions corresponding to the sum and carry bits of addition, parity functions, and majority functions require graphs that grow linearly or quadratically in v. Two functions represented by graphs can be tested for equivalence in time proportional to the sum of the two graph sizes and can be And'ed or Or'ed in at most time proportional to the product. Furthermore, a function can be complemented in time proportional to its graph size. As a result, a symbolic simulator can operate much faster than one might expect.

4 Experimental Results

To evaluate the feasibility of symbolic simulation, we used MOSSYM to verify different sizes of the ALU circuit shown in Mead and Conway [20] with the control lines set to perform addition, and with both input data words as well as the carry input set to symbolic values. We then compared the functions computed for the sum and carry outputs with functions generated from the Boolean equations for addition. This circuit utilizes a precharged Manchester carry chain and pass transistor multiplexors. The fact that it works relies on rather subtle aspects of MOS circuit design, and hence a switch-level verification is quite useful. The time required for this verification is given in Table . For comparison we show the time required to perform the same analysis with MOSSIM II by exhaustively simulating the circuit for all possible input combinations. For each set of inputs, we reset all node states to X, simulated one clock cycle, and compared the outputs to the binary representation of the sum of the two data words and the carry input. Both programs are written in Mainsail [19] and executed on a Digital Equipment Corp. VAX 11/780. Thus our comparison is fairly realistic, except that far more time and effort has been spent optimizing the performance of MOSSIM II than with MOSSYM.

Word Size	Transistors	Symbolic (CPU time)	Exhaustive (CPU time)
1	43	20 s.	3.6 s.
2	86	40 s.	19 s.
4	172	82 s.	454 s.
8	344	196 s.	49 hrs.*
16	688	566 s.	648 yrs.*

* - estimated

For this class of circuits, the time required for symbolic verification grows quadratically with the size, although a linear term dominates for small word sizes. This growth can be justified on the grounds that the sizes of the function representations (and consequently the time required for the algebraic operations) grows linearly with the word size, as does the total number of algebraic operations. The performance is far better than one would predict by a worst case analysis. Consequently the exponential time required for exhaustive simulation eclipses the time required for symbolic verification for sufficiently large word sizes. The cross over occurs at somewhere between a 2 and 3 (e?) bit word size. Considering that the exhaustive verification of the 3-bit ALU requires only 128 clock cycles, it is somewhat surprising that the more complex method of MOSSYM can outperform conventional simulation for this case.

The performance of MOSSYM depends primarily on the data structure sizes for the Boolean functions being manipulated and secondarily on how the circuit implements these functions. For a large class of circuits, we believe that symbolic simulation will prove practical.

5 Future Research

Our program MOSSYM represents only a prototype of a true symbolic simulator. A variety of improvements could be made in its performance and its capabilities. Perhaps the most obvious method to improve performance would be to *compile* switch-level subcircuits into Boolean expressions, so that we could compute the steady state response of a component by symbolically evaluating logic expressions rather than by solving the steady state response equations for the component each time. As an extreme example, MOSSYM requires 200 times longer to compute the output of an nMOS or CMOS inverter for a function applied to the input than it does to simply complement the function, in part because our manipulation routines can complement a function very quickly. Our program already has the basic capabilities required to compile switch-level subcircuits, but we have not yet implemented this feature.

The availability of a symbolic simulator raises a difficult set of issues regarding how one should go about verifying a complex digital system. The desired behavior of a system must be specified in a more rigorous fashion than most circuit designers currently use. For arithmetic circuits it is relatively easy to describe the behavior in terms of algebraic expressions, and the verification program can easily translate these into Boolean functions. For a complex sequential system, however, it is less clear how to devise a verification procedure that will rigorously establish correctness. For example, consider a random access memory. One might attempt to verify it by simply writing a symbolic value into a symbolic address and then reading the contents at a second symbolic address. However, this procedure would not detect an error in the address decoding logic that causes two addresses to map into the same location. For a circuit such as a microprocessor, the problems of specifying the desired behavior and devising a verification procedure become even more difficult. Finding solutions to these problems could contribute to the more general goal of designing digital systems in a more reliable manner.

I would like to acknowledge the contributions of Michael Schuster during the early stages of this research.

References

[1] A. Aho, J. Hopcroft and J. Ullman.
The Design and Analysis of Computer Algorithms.
Addison Wesley, 1974.

[2] S.B. Akers.
Binary Decision Diagrams.
IEEE Transactions on Computers C-27(6):509-516, June, 1978.

[3] H.G. Barrow.
Proving the Correctness of Digital Hardware Designs.
VLSI Design V(7):64-77, July, 1984.

[4] R.E. Bryant, M. Schuster, and D. Whiting.
MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual.
Technical Report 5033:TR:82, Calif. Inst. of Tech., January, 1983.

[5] R.E. Bryant.
Race Detection in MOS Circuits by Ternary Simulation.
In F. Anceau and E.J. Aas (editor), *VLSI 83*, pages 85-95. August, 1983.

[6] R.E. Bryant.
A Switch-Level Model and Simulator for MOS Digital Systems.
IEEE Transactions on Computers C-33(2):160-177, February, 1984.

[7] R.E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
October, 1984.

[8] J. Brzozowski and M. Yoeli.
On a Ternary Model of Gate Networks.
IEEE Transactions on Computers :178-183, March, 1979.

[9] M.C. Chen.
Space-Time Algorithms: Semantics and Methodology.
PhD thesis, Calif. Inst. of Tech., May, 83.

[10] W.J. Dally and R.E. Bryant.
A Special Purpose Processor for Switch-Level Simulation.
In *International Conference on Computer Aided Design* . IEEE, October, 1984.

[11] G. Ditlow, W. Donath, and A. Ruehli.
Logic Equations for MOSFET Circuits.
In *International Symposium on Circuits and Systems*, pages 752-755. IEEE, 1983.

[12] M. Garey and D. Johnson.
Computers and Intractibility: a Guide to the Theory of NP-Completeness.
Freeman, 1979.

[13] M. Gordon.
A Very Simple Model of Sequential Behaviour of nMOS.
In J. Gray (editor), *VLSI 81*, pages 85-94. Academic Press, August, 1981.

[14] I.N. Hajj and D. Saab.
Symbolic Logic Simulation of MOS Circuits.
In *International Symposium on Circuits and Systems.* IEEE, 1983.

[15] J. Hayes.
A Unified Switching Theory with Applications to VLSI Design.
Proceedings of the IEEE 70(10):1140-1151, October, 1982.

[16] N.P. Jouppi.
TV: an nMOS Timing Analyzer.
In R.E. Bryant (editor), *Third Caltech Conference on VLSI*, pages 71-86. March, 1983.

[17] T. Lengauer, S. Näher.
Delay-Independent Switch-Level Simulation of Digital MOS Circuits.
Technical Report TR 03/84, SFB 124-B2, Univ. d. Saarlandes, 1984.

[18] R.J. Lipton, R. Sedgewick, and J. Valdes.
Programming Aspects of VLSI.
In *Principles of Programming Languages.* ACM, 1982.

[19] *Mainsail Language Manual*
Xidak, Inc., Menlo Park, CA, 1982.

[20] C. Mead and L. Conway.
Introduction to VLSI Systems.
Addison Wesley, 1980.

[21] J.K. Ousterhout.
Crystal: A Timing Analyzer for nMOS Circuits.
In R.E. Bryant (editor), *Third Caltech Conference on VLSI*, pages 57-70. March, 1983.

[22] R.E. Tarjan.
Efficiency of a Good but Nonlinear Set Union Algorithm.
J. ACM 22(2):215-225, April, 1975.

[23] C.J. Terman.
Simulation Tools for Digital LSI Design.
PhD thesis, MIT, October, 1983.