An Algorithm for MOS Logic Simulation

Randal E. Bryant, Massachusetts Institute of Technology

With the introduction of simplified MOS design techniques by Mead and Conway (1980) we have seen a proliferation of custom LSI systems designed by non-specialists. For most people, however, VLSI design is still the ultimate "batch job"—a very complex system must be designed, hand debugged and fabricated before it can be tried out, by which time no corrections can be made. If we are to realize the full potential of custom VLSI, we cannot spend the time, patience, and personnel to hand-verify designs and iterate through several prototypes, as currently done in industry. Fortunately, well-designed simulation tools can eliminate much of this tedium, and greatly increase the chances of first-time success.

Traditionally, MOS designers have been forced to use either logic-gate simulators or circuit simulators such as SPICE (Nagel 1975). Neither choice works well. Logic-gate simulators are based on the Boolean gate model in which a network consists of a set of unidirectional logic gates connected by one-way, memoryless wires. This does not reflect the nature of MOS designs, which contain bidirectional logic elements (field-effect transistors) and bidirectional wires with sufficient capacitance (including the attached gate capacitances) to store signals dynamically. Numerous *ad hoc* extensions, such as high impedance states and unidirectional models of pass transistors, have been added to logic gate simulators, but they fail to correct the fundamental mismatch between the logic model and the logic technology. Alternatively, circuit simulators can accurately model any logic design, but do so with such great detail that the amount of computation required and the human effort needed to set up the simulation and interpret the results are overwhelming.

As an alternative to conventional logic and circuit simulators, the simulator MOSSIM (Bryant 1980) was developed specifically for the logical simulation of MOS LSI. MOSSIM models a logic network as a set of nodes connected by FET "switches". This network model has the following advantages:

1. It is based on the actual structure of the design rather than on its intended function.
2. It is universal, because MOSFETs are a common denominator for all MOS LSI designs.

Unlike circuit simulators, however, MOSSIM models the network in a highly idealized way. It utilizes only 3 logic levels: 0,1, and X (for undefined or unknown). The fourth or "high-impedance" state seen in other simulators is not needed, because MOSSIM automatically simulates "tri-state" devices as a consequence of its network model. Transistors are modeled as perfect switches, and no attempt is made to model the timing in great detail. These idealizations allow the simulator to operate at speeds approaching those of conventional logic simulators. Entire VLSI designs can be simulated in an interactive environment. MOSSIM has shown great expressive power in modeling many varieties of MOS designs directly.

We have also had great success in extracting the switch-level network directly from the mask layout files described in CIF (Mead and Conway 1980), using a program written by Clark Baker (Baker 1980) (see article in this issue of LAMBDA). Because the simulator needs only a simple description of the network structure, the extraction program can be relatively straightforward. This combination of automatic network extraction, followed by simulation, uncovers many errors in both the logic design and the layout. It has the advantage of testing the design which will actually be implemented, rather than the one the designer thinks will be implemented. With the wide variety of human and partially-debugged computer systems currently producing layouts, this double-check has proved invaluable. As computerized tools become more sophisticated and layouts are produced "untouched by human hands", layout extraction may become less crucial. However, a switch-level simulator could still be integrated into the design system to aid the logic design checking.

In the following sections, an algorithm for a switch-level simulator will be described in sufficient detail to enable the reader to implement it. Programs will be presented in the style used in Aho, Hopcroft and Ullman (Aho, Hopcroft and Ullman 1974). The algorithm described here corresponds to the most recent version of MOSSIM. This program can model both ratioed nMOS and CMOS designs which operate either with a conservative clocking scheme or with speed-independent logic. It runs reasonably fast, because it preprocesses the network extensively, and because it simulates parts of the network with functional models. Some possible extensions of the basic program will be discussed at the end.

Simulator Features

Network Model

In MOSSIM a logic network consists of a set of nodes connected by a set of transistors. There are three types of nodes:

1. *Input* provides a strong signal to the circuit, and comes from off-chip (e.g., Vdd, Gnd, Phi1, Phi2, etc.)
2. *Pullup* attached via a pullup resistor to Vdd; will pro-

vide a 1 signal unless grounded (e.g., the output of an nMOS inverter.)

3. *Normal* provides no signal but is capable of storing a signal dynamically.

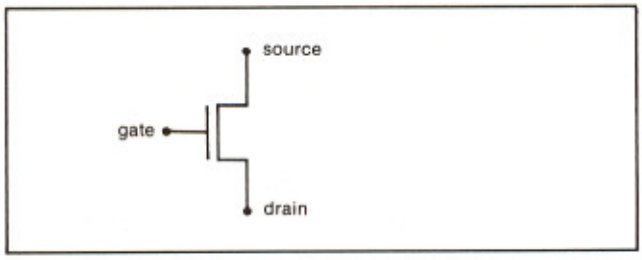


FIGURE 1. A transistor is a three-node device.

A transistor is a three-node device as shown in Figure 1. No distinction is made between the source and drain connections of a transistor. Transistors can be either n-type, p-type, or d-type. All act as voltage-controlled switches as shown in Table 1.

n-type		p-type		d-type	
gate signal	effect	gate signal	effect	gate signal	effect
0	open	0	closed	0	closed
1	closed	1	open	1	closed
X	unknown	X	unknown	X	closed

TABLE 1. Transistor types and behavior.

D-type (for "depletion") transistors are used to model the poly-diffusion crossovers seen in some MOS designs. They are not used to model depletion loads.

User Interface

A simulator requires two kinds of information: the network to be simulated, and the operations to be performed on it. For the simulator to aid the design of complex VLSI systems, it must provide the user with power and flexibility, while avoiding unnecessary tedium or awkward conventions. After several generations, MOSSIM has evolved into a program which allows the user to test a design in a reasonably pleasant, interactive environment, as documented in (Bryant 1980).

Network Input

MOSSIM can accept a network description in several different formats. A preprocessing program takes one of these descriptions, performs a topological analysis, and creates a file for the simulation control program. By keeping the front end of this program modular, it can easily be adapted to new network formats.

For describing networks prior to mask layout, the macro language NDL (Network Description Language) was developed. This language provides the following features:

- symbolic names for nodes
- pre-defined system elements such as transistors, various logic gates, etc.
- macro expansion of user-defined subnetworks.

NDL allows the user to specify large systems as a hierarchy of subnetworks. It has proved somewhat awkward to use, however, because it lacks many features of a first-class programming language, such as integer arithmetic, iteration, conditionals, etc. Rather than extending the language further, it seems wise to embed a set of network description primitives into an existing programming language.

For simulating systems based on a layout, the layout extraction program described previously can be used. In the future, the network description should be incorporated into an integrated VLSI design system.

Simulator Control

Testing a VLSI system has many of the characteristics of testing a large computer program. Because of its size, the design must be viewed modularly and at several different levels of detail. Errors may occur deep within one part of the design and propagate in entirely unexpected ways. Thus, an interactive software debugger forms a useful model for a simulation control program.

With MOSSIM, the user can exercise a network through a sequence of inputs and set or probe nodes using symbolic names. User-defined clocking schemes can be applied, and collections of nodes can be combined into vectors. A section of the network can be tested in isolation by "forcing" signals onto selected nodes, temporarily turning them into input nodes.

Possible future extensions include breakpoints and more sophisticated levels of control, such as user-defined procedures. Once again, these features would best be provided by embedding the simulation control within an existing programming language.

Design Philosophy

MOSSIM is designed to test the functionality of a particular class of VLSI systems. It does not try to model detailed electrical and timing behavior. It assumes that in clocked systems, the clock signals are spaced far enough apart to allow the circuit to settle between each signal change, but close enough together to prevent the loss of stored charge due to leakage. Furthermore, the circuit must contain no critical races. That is, the circuit should settle to a unique state regardless of the individual signal delays. Systems built with two-phase, non-overlapping clocks generally fulfill these requirements, as do most self-timed systems. For the purposes of this discussion, clocked systems will be assumed. Likewise, the simulator will not catch electrical errors such as improper inverter ratios, or multiple-threshold voltage drops.

The behavior of MOSSIM is based on that of an abstract model of MOS circuits. Any implementation should not depend on the details of the actual program code. For example, no anomalous behavior should arise from a chance ordering of events or from artifacts of the data structures used in the implementation. Furthermore, any questions about the correctness of the simulator should be resolved by studying the abstract model. With this in mind, very few design decisions must be made arbitrarily. The abstract model for MOSSIM is described in my thesis (Bryant, in preparation).

Detailed Description of the Simulator

Simulator Operation

Topological Analysis

As an aid to both the speed and the understanding of the simulation algorithms, it helps to structure the network first through a topological analysis.

Input Node Splitting

Input nodes provide strong signals to the network. These signals cannot be modified by the internal operations of the network. Thus we can split each logical input node into a number of physical input nodes, one for each transistor to which it is connected. For example, instead of viewing a number of logic gates as sharing a single node Gnd, each pulldown transistor can be viewed as having its own source of a 0 input, as shown in Figure 2(b).

Input node splitting serves two purposes. First, it prevents anomalous behavior caused by short circuits (when two input nodes with different states are connected) from spreading throughout the network. With input node splitting, nodes along the shorting path(s) of a short circuit are set to X, while others are unaffected. Second, it helps to fragment the network into smaller entities called "transistor groups."

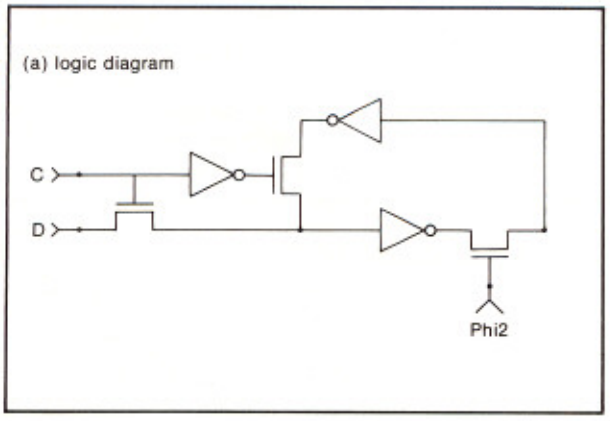
In the implementation, a map must be maintained from the logical input nodes seen by the user to the physical input nodes seen by the simulator. Every time the user sets the state of an input node, the program must update the corresponding internal nodes.

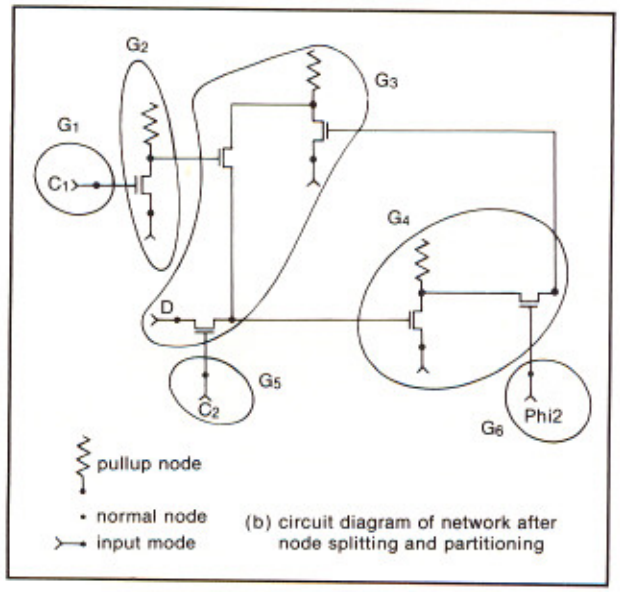
Transistor Groups

Our network model is unique in that the logic element is a bilateral device where neither the source nor the drain can be characterized as input or output. This property, while important for modeling MOS designs, causes difficulty in devising simulation algorithms and in applying performance improvements such as selective node updating.

The gate node of a transistor, however, is a pure input to the logic element: its state determines the state of the switch, but the gate node is not directly affected by transistor operations. We can exploit the unidirectional connection of the gate nodes by partitioning the network into a set of transistor groups. Nodes within a transistor group are interconnected through the bilateral nodes of transistors. Between transistor groups, only unilateral interactions occur from a node in one group to the gate of a transistor in another group.

FIGURE 2. Topological analysis of network.





Transistor groups can be defined in the following way: create an undirected graph with a vertex for every network node and an edge between a pair of vertices, if the corresponding nodes form the source and drain for some transistor. This graph can be partitioned into its connected components using the algorithm described in Appendix I. The set of nodes and transistors corresponding to the vertices and edges in a component form a transistor group. An example of a network partitioned into transistor groups is shown in Figure 2(b). Note that a split input node may lie in several groups.

Network Representation

Let us assume that we have a logic network in which all input nodes have been split, and the nodes and transistors have been partitioned into a total of g transistor groups. We can represent the structure of a network, Ω, in the following way:

$$\Omega = (N, T)$$

where

$$N = \{ n_{ij} \mid 1 \le i \le g \}$$

denotes the set of nodes, and

$$T = \{ t_{ik} \mid 1 \le i \le g \}$$

denotes the set of transistors. In both cases, the first subscript indicates the group number, and the second indicates the element within the group.

Information about the nodes and transistors is encoded in the following functions:

NTYPE:	$N \rightarrow \{$ *input, pullup, normal* $\}$,	the node type
TTYPE:	$T \rightarrow \{ n, p, d \}$,	the transistor type
GATE:	$T \rightarrow N$,	the gate node
SOURCE:	$T \rightarrow N$,	the source node
DRAIN:	$T \rightarrow N$,	the drain node

The state of the network is defined as:

$$M = (S, X)$$

where

$$S = \{ s_{ij} \mid 1 \le i \le g \}$$

denotes the node states, s_{ij} is an element of { 0, 1, X }, and

$$X = \{ x_{ik} \mid 1 \le i \le g \}$$

denotes the transistor states, x_{ik} is an element of {open, closed, undefined}.

Timing Model

With MOSSIM the flow of time is viewed at three levels of granularity:

cycle: A complete sequencing of the system clocks.
phase: One set of clock and data input values.
step: The atomic time unit of the simulator. Within a phase, unit steps are taken until the circuit settles.

A clock in MOSSIM is defined as a set of sequences to be applied cyclically to a set of nodes. For example, a two-phase, non-overlapping clock can be defined as:

Phi1: 0 1 0 0 Phi2: 0 0 0 1,

where both Phi1 and Phi2 are input nodes. A phase corresponds to a single setting of the input nodes (both data and clock). The above example has four (!) phases within each clock cycle. Implementing the phase simulation requires some effort. Once this is done, implementing the cycle level and other aspects of the user interface requires a fair amount of effort but no theoretical insights.

Phase Simulation

Program I shows how a phase simulation is performed. The procedure is given a list A containing pairs of values $< n_{ij}, y >$, y is an element of { 0, 1, X }, indicating a node and its new state. At the very start of the simulation:

$$A = \{ < n_{ij}, X > \mid n_{ij} \in N \}$$

i.e., all nodes are to be initialized to X. The program works by maintaining an event list E containing those group numbers in which a node or transistor state has been changed, but the group has not been resimulated. First, the requested node changes are applied, and the transistors with these nodes as gate nodes are updated. The procedure SET __ TRANS sets the transistor state according to Table 1. This causes new groups to be added to E. Next, a series of unit steps is taken until the event list E is emptied, indicating the network has settled. Once the network settles, a new phase can begin.

Program I. Phase Simulation

Input: A network Ω, a network state M, and a list A containing node state pairs $<n_{ij}, y>$.
Result: M is updated by simulating the effects of the new values in A.

```
procedure PHASE(Ω, M, A):
begin
    E ← ∅;
    for each <n_ij, y> in A do
      begin
          s_ij ← y;
          E ← E∪ {i};
          for each t_k1 such that GATE(t_k1) = n_ij do
            begin
              x_k1 ← SET __ TRANS(TTYPE(t_k1), y);
              E ← E∪ {k};
            end
      end;
    while E ≠ ∅ do
      E ← UNIT __ STEP(Ω, M, E);
end
```

As an implementation consideration, the maximum number of unit steps for a clocked system can be limited to | N |, i.e., the total number of nodes in the network. This will prevent non-terminating simulations caused by oscillatory behavior. This limit arises because the number of unit steps required is related to the depth of the combinational logic.

Unit Step Simulation

The unit step simulation is shown in Program II. During a unit step, the transistor states are held fixed, while the flow of signals through the transistors is simulated with the procedure GROUP __ STEP. This procedure is applied only to groups on the event list. Each updating results in some number (possibly 0) of nodes changing state. These nodes are accumulated in the Set D. Following the charge flow simulation, the transistors which have gate nodes in D are updated, and the groups in which these transistors lie are added to a new event list E for use in the next unit step.

Program II. Unit Step Simulation

Input: A network Ω, a network state M, and a list of groups E.
Result: M is updated by simulating the groups in E.
Output: A new list of groups requiring simulation.

```
procedure UNIT __ STEP(Ω, M, E):
begin
      D ← ∅;
      for each i in E do
          D ← D ∪GROUP __ STEP(Ω, M, i);
      E' ← ∅;
      for each n_ij ∈ D do
          for each t_k1 such that GATE(t_k1) = n_ij do
              begin
                x_k1 ← SET __ TRANS(TTYPE(t_k1), s_ij);
                E' ← E' ∪ {k};
              end;
      return (E')
end
```

By first changing the node states while holding the transistors fixed and then changing the transistor states with the nodes fixed, the transistors in effect switch 1 time unit after their gate nodes change. Hence the term "unit delay". This technique also ensures that the simulation behavior is independent of the ordering of groups in E.

Signal Flow Simulation

Thus far, we have remained external to the transistor groups, and the simulator appears much like an event-driven, unit-delay logic gate simulator. Within a transistor group, however, things operate much differently.

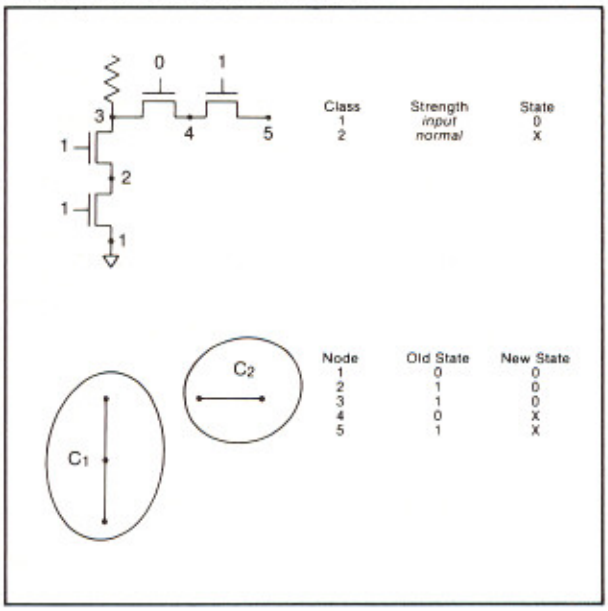
Program III shows the signal flow simulation. At the start we are given a transistor group containing a set of nodes and a set of transistors. Our task is to update the node states according to the flow of signals through the transistors. First, all pullup nodes are initialized to 1. Then, we consider the undirected graph containing a vertex for each node in the group and an edge corresponding to each transistor in the "closed" state. The connected components of the graph partition the set of nodes into a set of equivalence classes called CLASSES. An algorithm for performing this partition is given in Appendix I. An initial computation of the state for each class is performed by the procedure STATE, shown in Program IV. In this procedure, the "strongest" node(s) in the class are inspected, where node strengths are ordered as *input* > *pullup* > *normal*. The strength of the class is defined as the strength of its strongest node(s). If the states of the strongest nodes are equal, the class state becomes this state; otherwise it becomes X. The latter case represents either a short-circuit (for input) or charge-sharing (for normal). If the class strength is *pullup*, the class state always becomes 1.

A simple (and contrived) example of signal flow simulation is shown in Figure 3, showing a group consisting of a NAND gate with a chain of two pass transistors on its output. With the transistor gates set as shown, two equivalence classes are formed. The first class contains the ground node. Hence the class strength is *input* and the class state is 0. The second class contains two normal nodes with different states. Hence the class strength is *normal* and the class state is X.

If the group contains x-transistors (n or p type transistors having X on their gate nodes), then we must look at potential interactions between the classes. The state of an x-transistor is unknown. Hence, it is assumed to have an unpredictable behavior: it may be open, closed, or somewhere in between. To deal with them consistently, the following philosophy is adopted: if a node has a unique state regardless of the behavior of the x-transistors in the group, then the node will be set to this state; otherwise it will be set to X.

Thus far, we have computed the state of each class assuming all x-tran-

FIGURE 3. Simulation example I.



sistors are set "open". The second part of the signal flow simulation analyzes the classes and sets the state of a class to X if some combination of x-transistors could be set to "closed" and result in a different class state. First, a supergraph is formed containing a vertex for each equivalence class and an edge between a pair of vertices if an x-transistor connects two network nodes in the corresponding classes. The connected components of the supergraph partition the classes into a set of superclasses, in which each superclass contains a set of classes linked by x-transistors.

If a superclass contains only one element, no further analysis is required. Otherwise the strength of the superclass is computed as the strength of its strongest class(es). The state of the superclass is computed as the state of the strongest class(es) if they are all equal, and as X if they are not. The procedure SUPERSTATE closely resembles the procedure shown in Program IV. Any class with a state different from the superclass state must be set to X, because its state would be changed if all x-transistors were set to "closed". We will call these classes "poisoned". Furthermore, a poisoned class can poison a neighbor, unless the neighbor is stronger, even if the neighbor's original state equals the superclass state. That is, the state of the neighbor would be changed if the x-transistor connecting it to the poisoned class were set "closed" while all others were set "open". This poisoning can spread through the classes, stopped only by classes with greater strength than the original poisoned classes. The procedure POISON, shown in Program V, expands the initial set of poisoned classes to include those to which the poisoning spreads. This procedure works by computing a value, "pstrength," for each class indicating either its original strength (if not poisoned) or the strength with which it is poisoned. All poisoned classes must be set to X.

Program III. Signal Flow Simulation

Input: A network Ω, a network state M, and a group number i.
Result: The node states in group i are updated.
Output: A list of those nodes which have changed state.

```
procedure GROUP_STEP(Ω, M, i):
begin
    for each n_ij such that NTYPE(n_ij) = pullup do s_ij ← 1;
    G ← form an undirected graph with V = {n_ij}
          and E = {<k,1> | for some t_pq:
                SOURCE(t_pq) = n_ik, DRAIN(t_pq) = n_i1, and x_pq
                = "closed"};
    CLASSES ← PARTITION(G);
    for each C_j ∈ CLASSES do
        strength|j|, state |j| ← STATE (Ω, M, C_j);
    SG ← form an undirected graph with V = CLASSES
          and E = {<k,1> | for some t_pq:
                SOURCE(t_pq) ∈ C_k, DRAIN(t_pq) ∈ C_1, and x_pq =
                "unknown"};
    SUPERCLASSES ← PARTITION(SG);
    for each SC_j ∈ SUPERCLASSES do
        if | SC_j | > 1
        then
            begin
              y ← SUPERSTATE(strength, state, SC_j);
              P ← {C_k ∈ SC_j | state|k| ≠ y };
              POISON(P, SG, strength);
              for each C_k ∈ P do state|k| ← X
            end;
    D ← ∅;
    for each C_j ∈ CLASSES do
        for each n_ik ∈ C_j do
            if NTYPE(n_ik) ≠ input & s_ik ≠ state|j|
              then
                  begin
                    s_ik ← state|j|;
                    D ← D ∪ {n_ik};
                  end
    return(D)
end
```

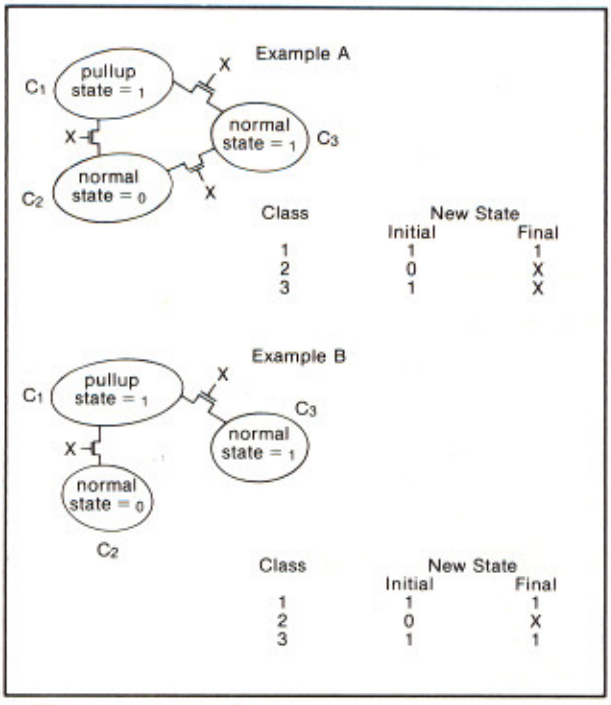
Program IV. Class State Computation

Input: A network Ω, a network state M, and a set of nodes C.
Output: The combined strength and state of the nodes.

```
procedure STATE(Ω, M, C):
begin
    k ← null;
    y ← X;
    for each n_ij ∈ C do
        comment node types are ordered input>pullup>normal>null
        if NTYPE(n_ij) > k
          then
              begin
                  k ← NTYPE(n_ij);
                  y ← s_ij
              end
          else
              if NTYPE(n_ij) = k & s_ij ≠ y
                  then y ← X;
    return (k,y)
end
```

The spreading of X through the classes depends on the exact topology of the supergraph as shown in Figure 4. In example A, the poisoned class C_2 can poison C_3, because they are of equal strength. In example B, C_3 remains in the 1 state because the poisoning from C_2 cannot spread through C_1.

FIGURE 4. Spreading of X through classes.



Program V. Spreading of X through a Superclass

Input: An initial set of poisoned classes P, a supergraph of classes SG, and strength: an array indicating the strength of each class.
Result: P is expanded to include other poisoned classes.

```
procedure POISON(P, SG, strength):
begin
    pstrength ← strength;
    B ← P;
    while B ≠ ∅ do
      begin
        Cj ← an element of B with maximal strength;
        B ← B - { Cj };
        for each Ck adjacent to Cj in SG do
            if pstrength[k] < pstrength[j]
            or (pstrength[k] = pstrength[j] & Ck ∉ P)
            then
              begin
                P ← P ∪ {Ck };
                B ← B ∪ { Ck};
                pstrength[k] ← pstrength[j]
              end
      end
end
```

Once the class states have been computed, the state of each non-input node is set to the state of its class.

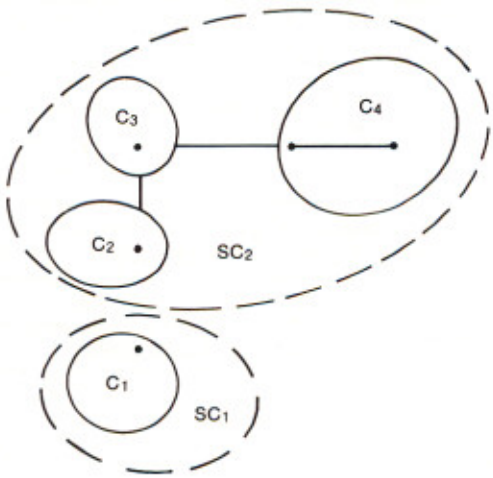
In implementing the procedure POISON, one should note that there are only a small number of possible class strengths (i.e., 3). Thus, we can implement the set B with a set of "buckets", one for each strength. This permits both insertion and ordered removal in constant time.

Figure 5 shows the signal flow simulation for the same transistor group as before, but with several x-transistors.

Time Complexity

If the simulation is coded carefully, especially the graph partitioning, then the procedure GROUP _ STEP requires a time proportional to the number of nodes and transistors in the group. For all but a few pathological networks, each group will be resimulated only a constant number of times within each phase. Thus, each phase will have a time complexity of $\mathcal{O}(\,|\,N\,|\,+\,|\,T\,|\,)$.

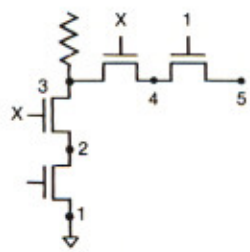
FIGURE 5. Simulation example II.



Node	Old State	New State	
		Initial	Final
1	0	0	0
2	0	0	X
3	0	1	1
4	1	1	1
5	1	1	1

Class	Strength	State	
		Initial	Final
1	*input*	0	0
2	*normal*	0	X
3	*pullup*	1	1
4	*normal*	1	1

Superclass	Strength	State
1	*input*	0
2	*pullup*	1



Extensions

Once the basic simulator has been developed, it can be extended in several ways to provide greater speed, generality, or functionality. Unlike many simulators which try to increase their generality through a number of *ad hoc* extensions, MOSSIM permits additional features to be built upon a solid foundation.

Mixed-Mode Simulation

After performing the topological analysis of a network, one often finds that a large number of transistor groups contain some simple combination of circuit elements. For example, a group may contain a single inverter such as G_2 in Figure 2(b), a NAND or NOR gate, or a gate with a single pass transistor on its output such as G_4 in Figure 2(b). For such common and easily-understood configurations, it seems wasteful to model them at a transistor level. Instead, following the topology analysis, we can try to match each group against a small set of canonical configurations. For those groups which can be matched, the list of transistors can be replaced wth information which will allow the simulator to apply a functional simulation analogous to a logic gate simulator. Since the simulator is event-driven at the group level, these two modes (transistor-based and function-based) can be merged quite easily. This optimization affects only the simulation speed, not its functionality.

One could apply mixed-mode simulation techniques to designs in which the user wants to describe some portions of the network procedurally and other parts in terms of in-

dividual transistors. This would allow incomplete designs to be simulated, as well as systems which must interface with other modules. As long as a subsystem can be characterized in terms of some input-output behavior, the program for the unit step simulation can treat it as if it were a transistor group.

Charge Transfer Logic

Some MOS circuit designs depend on the relative sizes of several capacitances for their logical behavior. This is seen frequently with pre-charged buses, in which a charge is first placed on a bus, and then this bus drives its signal through a pass transistor onto a smaller capacitance node, such as the storage node of a dynamic RAM cell. In the program just described, the nodes will be set to X (unless they were previously in the same state), because the simulator does not know the relative capacitance values.

This shortcoming can be remedied by allowing different kinds of normal nodes, e.g., "big" nodes and "small" nodes. The previously described algorithms can be extended by ranking the node types as:

input > pullup > big > small.

All other aspects of the simulation remain unchanged.

Node Forcing

In debugging a logic design, it often helps to isolate a portion of the design from the rest of the network and test its input-output behavior. To do this, we must be able to force signals onto a set of nodes, and these signals must override the normal actions of the network. If we define a new node strength, "forced," where

forced > input > pullup > ...

then we find that the algorithm that has already been developed implements this feature.

Implementation

MOSSIM has been implemented on a DEC-20 system as a series of CLU programs. Executable versions of the program are available from the author, as is user documentation.

A program called CONVERT takes network descriptions either written in a macro language or from the output of a layout analyzer and creates a description suitable for simulation. In this new description, the network is partitioned into transistor groups. The following configurations are replaced by their functional equivalents: NOT, NAND, NOR, NOTp, NANDp, and NORp, where the "p" indicates a single pass transistor on the output of the logic gate. These improvements save memory space as well as CPU time.

The program RUNSIM allows the user to read in a network produced by CONVERT and exercise the design interactively. This program implements the algorithm just presented, although table lookup methods are applied whenever possible to reduce the number of conditional statements. For example, the state and strength of a node, class, or superclass are encoded as a single integer, and this value is used as an index into tables for computing class states, superclass states, etc.

Run time figures vary greatly depending on the size of the network and the amount of activity to be simulated. For small and medium-size networks (up to 1000 transistors), less than 1 second of CPU time is required for each clock cycle. For a LISP microprocessor chip containing 10,000 transistors, between 5 and 12 seconds of CPU time are required per cycle. These speeds have been found adequate for interactively exercising a variety of systems and could be reduced with further optimizations.

References

Aho, A. V.; Hopcroft, J.E.; and Ullman, J.D. 1974. *The Design and Analysis of Computer Algorithms*, Reading: Addison-Wesley.

Baker, C. M. June 1980. *Artwork Analysis Tools for VLSI Circuits*, M.S. thesis, M.I.T. Department of EECS.

Bryant, R. E. July 1980. *MOSSIM: A Logic-Level Simulator for MOS LSI, User's Manual*, Integrated Circuit Memo 80-21, M.I.T. Department of EECS.

Bryant, R. E. in preparation. *Logic Simulation of MOS LSI*, PhD thesis, MIT Deparment of EECS.

Mead, C.A. and Conway, L.A. 1980. *Introduction to VLSI Systems*, Reading: Addison Wesley.

Nagel, L. May 1975. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Technical Report UCB ERL-M250, Electronics Research Laboratory, University of California, Berkeley.

λ

Appendix I—Partitioning of an Undirected Graph

An algorithm to partition an undirected graph into its connected components is required at several different points in MOSSIM. Run time statistics show that as much as 50% of the total execution time may be spent computing these partitions. Thus, a very careful and efficient coding is required.

A variation of the depth-first search algorithm (5.2) of Aho, Hopcroft, and Ullman (1974) is shown in Program VI. This algorithm takes $\mathcal{O}(|V| + |E|)$ time if the marking of a vertex as "new" or "old" is encoded as a Boolean array and a pointer is maintained to the first "new" vertex.

Program VI. Partitioning of an Undirected Graph

Input: A graph G = (V, E), where V is a set of vertices, and E is a set of edges.

Output: A set of classes $C_1,...,C_q$ such that $C_1 \cup ... \cup C_q = V$; $C_i \cap C_j = \emptyset$, $i \neq j$; and $v \in C_i$, w adjacent to v in G $\Rightarrow$ $w \in C_i$.

```
procedure PARTITION(G):
begin
    i ← 1;
    for all v in V do mark v "new";
    while there exists a vertex v in V marked "new" do
      begin
        Ci ← {v};
        SEARCH(Ci, v);
        i ← i+1
      end;
    return({ C1,...,Ci-1})
end

procedure SEARCH(C, v):
begin
    mark v "old";
    for each vertex w adjacent to v in G do
      if w is marked "new"
        then
          begin
            C ← C ∪ { w };
            SEARCH(C, w)
          end
end
```

About the Author

Randal E. Bryant received his BS in Applied Math from the University of Michigan in 1973, his MS in Electrical Engineering from MIT in 1977, and is currently completing his PhD at MIT. His research interests include logic simulators for VLSI systems, and data-flow computer architecture.