Logic Simulation Using Networks of State Machines*

Peter M. Maurer

Abstract

This paper shows how to simulate a circuit as an interlocked collection of state machines. Separate state-machines are used to represent nets and gates. The technique permits intermixing of logic models, direct simulation of higher-level functions, and optimization techniques for fanout free circuits. These techniques are an extension of techniques that have been used to achieve high-performance event-driven simulations. New, more efficient state-machine implementations are presented, and experimental data is presented that show the efficiency of the new techniques.

1. Introduction

It convenient to view a digital circuit as a network of components with signals flowing from primary inputs to primary outputs. This concept has proven quite useful, and is the basis of many existing simulation algorithms, including unit- and multi-delay simulation[1,2], levelized compiled code simulation [3], and other less conventional algorithms[4]. Regardless of the scheduling algorithms used by a simulator, the ultimate objective is to perform a set of gate simulations and produce a set of observable outputs. For simulators that support a large number of gate types, a substantial body of run-time code must be provided to support the gate simulations.

Although the concept of signal flows has proven useful in many areas of design automation, it does not reflect reality. Electrical components have internal states that change in response to purely local conditions. The state of a gate changes in response to changes in its inputs, while the state of a wire changes in response to changes in its driving gate. One approach to simulation that has adopted this alternative point of view is the Inversion Algorithm[5], which models both gates and wires using a collection of internal states. All simulation operations are performed on local states, with no signal flows between devices.

In the Inversion Algorithm the approach was largely intuitive, and based on certain well-known concepts such as counting dominant inputs. This paper will approach the state-change model from a more theoretical point of view, and will extend the model to a wider class of problems, such as direct simulation of LUT-based FPGA circuits, and direct simulation of complex Boolean equations.

* This research was supported in part by the National Science Foundation under grant MIP-9403414.

Although the approach taken in this paper is quite different from those taken in earlier published works, the inspiration came from the Schuler counting algorithm [8], from work in state-machine based simulation [9] and efficient Boolean function evaluation [10].

2. The State-Machine Approach

As shown in [5], it is possible to compute the state of a multiple-input gate using unary operations. This leads to highly efficient simulation, and permits a number of useful optimizations. The use of unary operations is made possible by maintaining a state-variable for each gate. Any change in an input causes a state-change in the gate which can be computed without referring to other inputs. Figure 1 gives the state-machine structure for a three-input AND gate.

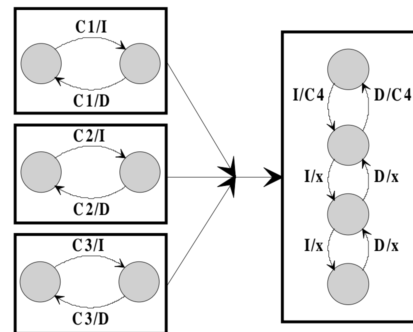


Figure 1. A Sample Network.

In Figure 1, each rectangle represents a single state machine. The symbols on the arcs represent the input and output values. The symbol C1/I indicates that on input C1, the state machine will change state and produce output I. In the Inversion Algorithm, the I and D outputs will cause Increment and Decrement operations to be performed, but the state-machine structure will function correctly regardless of the mechanism used to implement the outputs. The inputs C1, C2, and C3 are modeled as events on the inputs of the gate, while C4 is modeled as an event on the output of the gate. The symbol x is used to indicate *no output*. The state machine on the right is implemented as a simple counter which is incremented and decremented by the I and D inputs.

In some cases it is convenient to associate an additional output with each state. Outputs associated with transitions are called Mealy outputs, while those associated with states are called Moore outputs. State machines interact

via their Mealy outputs. All computations are local and done on single states. Unlike more typical simulations, adding an input does not increase the amount of work that must be done to process existing inputs.

In the three leftmost state machines of Figure 1, it is not specified which state corresponds to zero. This structure represents an AND gate, so the rightmost state of each machine corresponds to zero. If we were to designate the leftmost state as the zero state, the structure would then correspond to an OR gate instead of an AND. Because the association between states and values is not maintained by the simulator, the same structure can be used for either AND or OR gates. Still other functions can be implemented by mixing and matching the associations between right and left.

3. Basic State Machines.

The most fundamental state machines are those used to represent nets. A state machine for a two-valued net is illustrated in Figure 2. This state machine has a single input called "Change." More complex state machines, such as that illustrated in Figure 3, can be used to represent more complex logic models. Since a state machine is a local entity, it is possible to use different logic models for different nets[6]. To simplify the discussion, this paper will assume that all nets are binary.

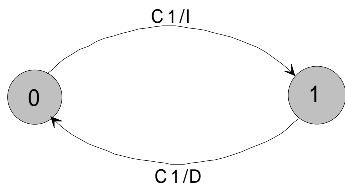


Figure 2. A Binary State Machine.

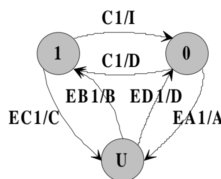


Figure 3. A Trinary State Machine.

The Moore outputs, 0 and 1, illustrated in Figure 2 are virtual and used only during the initialization phase of the simulation. The assignment of ones and zeros can be switched whenever it is convenient to do so. This will alter the function computed by the state machines without changing their structure.

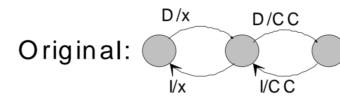
Figure 4 illustrates the functions that can be computed by changing the Moore outputs of the AND gate. The A, B, and C columns indicate the Moore output of the leftmost state. It is possible to implement even more functions by altering the Mealy outputs of the Gate state machine, as illustrated in Figure 5.

The Hyperactive machine can be used to simulate the XOR and XNOR functions, while the Insensitive machine can be used to simulate constant one and constant zero. These functions can also be simulated by less elaborate mechanisms.

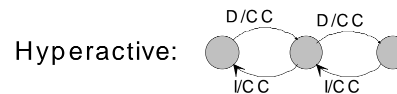
A	B	C	Function
0	0	0	*A* **AND** *B*
0	0	1	*A* **NAND** *B*
0	1	0	*A* **AND Not** *B*
0	1	1	**Not** *A* **OR** *B*
1	0	0	**Not** *A* **AND** *B*
1	0	1	*A* **OR Not** *B*
1	1	0	*A* **NOR** *B*
1	1	1	*A* **OR** *B*

Figure 4. Linear State Machine Functions.

Original:



Hyperactive:



Insensitive:

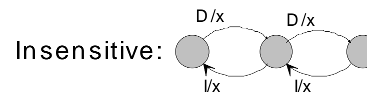


Figure 5. Three Types of Gate State Machine.

Most state machines used by the Inversion Algorithm are linear in shape, because they correspond to a count of dominant inputs. Although limited in structure, these machines can be quite powerful, especially for functions with more than two inputs.

As with 2-input gates, the input and output state machines can be modified, to simulate many different functions. However, because there are more than two transition points, the structure of the Mealy outputs can be correspondingly more complex. Figure 6 gives the variations of the 3-input linear state machine.

Despite the wide variety of functions that can be simulated using linear state machines, for n>2, it is impossible to implement all n-input functions as linear state machines. A simple argument shows that this is true. For n inputs, the linear state machine will have n+1 states, and n links. Altering the input and output state machines will give no more than 2^{n+1} different functions, while altering the Mealy outputs of the gate state machine will yield no more than 2^n different machines, for a total of 2^{2n+1}. However, there 2^{2^n} are different n-input functions. This implies that direct simulation of certain

functions will require a wider variety of state machines. One method of creating more complex machines is to form the cross product of linear state machines, as illustrated in Figure 7.

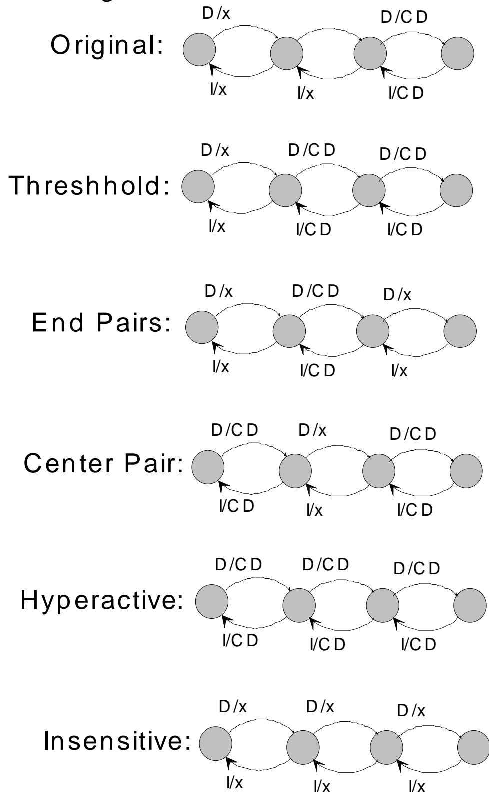


Figure 6. Types of 3-Input Gate State Machines.

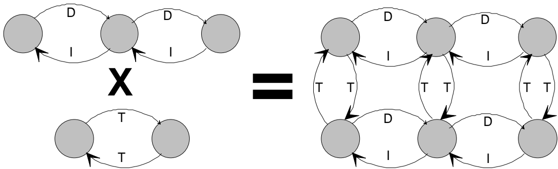


Figure 7. Cross Product of Linear Machines.

4. Systematic Generation of State Machines

A state machine for a complex function can be derived from the basic state machine of Figure 2. The general procedure is to create a cross-product machine, assign Moore outputs based on the function, and then reduce the machine using standard techniques. To illustrate the procedure, we will demonstrate how to create a state-machine for a two-input AND. We start with the fundamental state machines for the inputs, and take the cross-product of these machines as shown in Figure 8.

The Moore outputs of each state are computed by applying the AND function to the Moore outputs of the original machine. The Mealy outputs are computed by identifying the transitions that cause the Moore output of the machine to change. For more complex functions, the computation of the Moore outputs would use the function itself, and the computation of the Mealy outputs would proceed as before by identifying transitions where the Moore output changes.

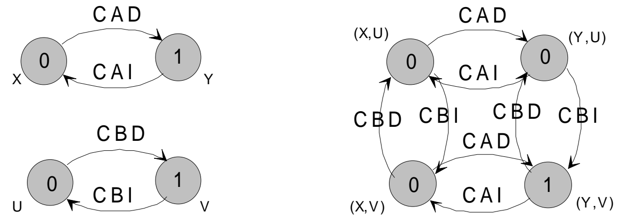


Figure 8. Developing a Gate State Machine.

To simplify the resulting state machine, it is necessary to combine symmetric inputs, leading to some linearization of the machine. In Figure 8, the symmetric inputs A and B can be collapsed by replacing the inputs CAI and CBI with a single input I, and the inputs CAD and CBD with a single input D. This will yield the non-deterministic state machine illustrated in Figure 9. The general procedure for handling symmetric input nets is to combine the associated state machine inputs with a single input representing a change in any of the input nets.

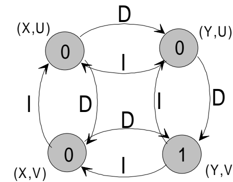


Figure 9. Combined Inputs.

Before the state machine of Figure 9 can be implemented, it is necessary to convert it into a deterministic machine. This will eventually produce a state of the form {(X,V),(Y,V),…}. The equivalent deterministic state machine is given in Figure 10.

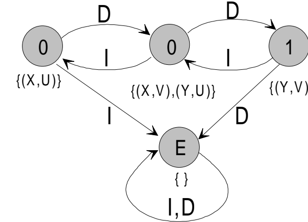


Figure 10. The Equivalent Deterministic Machine.

In the remainder of the paper, we will omit the error state when no transition into that state can occur.

Although a gate state machine for a function can always be created from simple binary machines, it is sometimes easier to treat the function as a collection of simpler functions. This is particularly true when collapsing homogeneous and heterogeneous connections as described in [5]. For illustrative purposes, assume we

wish to find the gate function for a gate consisting of three gates G, H, and K, connected as illustrated in Figure 11.

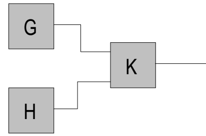


Figure 11. A Collection of Gates.

Suppose the function computed by gate K is z=f(x,y). Taking the cross product of the state machines for G and H creates the state machine for the combined gate. The Moore output of the state (P,Q) is f(p,q), where p is the Moore output of state P and q is the Moore output of state Q. It may be possible to combine symmetric inputs in the result. Figure 12 illustrates this process.

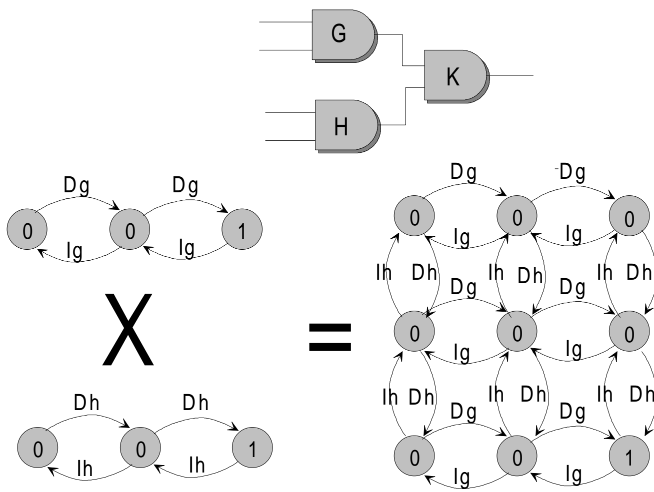


Figure 12. Collapsing Homogeneous Connections.

In Figure 12, it is possible to combine symmetric inputs thus: Dh = Dg = D, and Ih = Ig = I. This will yield the deterministic state machine of Figure 13.

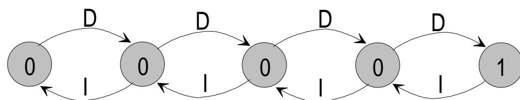


Figure 13. The Deterministic Equivalent.

After constructing the combined state machine, and possibly reducing it, it is necessary to assign the Mealy inputs. Any transition between states with identical Moore outputs is assigned a Mealy output of NULL. All other transitions are assigned Mealy outputs of EvX, where X is the output of the gate. Once the Mealy outputs have been assigned, the Moore outputs can be dropped.

5. Applications

The primary applications of this work are collapsing fanout free networks into a single simulation unit, providing efficient simulations for library functions, and direct simulation of FPGA look-up tables. In this section we concentrate on efficient simulation of library cells. In [5] the binary transitions of nets are modeled by alternating two different event handlers. This model can be extended to handle the more complex state machines described in this paper.

State machine implementations are embedded in the event-handlers for nets, eliminating the need for gate simulations. The most obvious way to handle linear state machines is to maintain a count, and use the I input to increment the count and the D input to decrement the count. Although this procedure has given good for results for simple linear state machines, for more complex state machines, more sophisticated procedures must be used.

Consider a state machine M, which is the cross product of two linear state machines A and B. Assume that the inputs to A are I_a and D_a, and the inputs to B are I_b and D_b. Assume that machine A is implemented using a count, and that k is the maximum value of that count. In the machine M, the inputs I_a and D_a are assigned the value 1, and the inputs I_b and D_b are assigned the value k+1. The machine M will be implemented using a single count. The input I_a will increment the count by 1, the input D_a will decrement the count by 1, the input I_b will increment the count by k+1, and the D_b input will decremented the count by k+1.

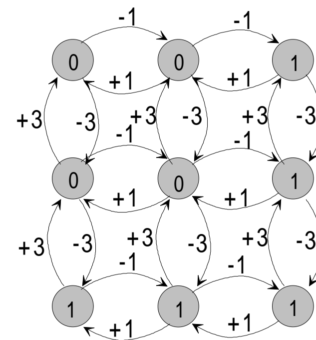


Figure 14. Cross-Product Incr. and Decr.

Figure 14 also shows how to determine when to propagate events. After an increment by 3, an event will propagate if the new count is 4 or 5. After an increment by 1, an event will propagate if the new count is 4 or 7.

Other cross-product machines can be implemented in a similar way. A *general counting machine* is defined to be either a linear state machine, or a cross-product of two other general counting machines. All general counting machines can be implemented using a single count, which is incremented and decremented by different amounts depending on the input. Suppose M is the cross product of two general counting machines A and B. As in the previous example, the inputs of A are assigned the same value that they had in the implementation of the machine

A. If k is the maximum value of the count in the implementation of machine A, then the inputs of B are assigned the value $(k+1)q$ where q is the value of the input in the implementation of B. General counting machines, so general counting machines are powerful enough to simulate any Boolean function.

When testing a new count for event propagation, it may be necessary to compare the count against a large list of values. However, the testing process can be simplified by using multiplication instead of addition as the state-transition function. In this model one uses states of the form 2^0, 2^1, 2^2, 2^3, … . To change state, the current count is multiplied or divided by 2^k. These operations are performed only on positive numbers, so left and right shifts can be used. Each bit position in the state counter represents a different state. Masks can be ANDed with the count to test for several states simultaneously. A non-zero result will generate an output event.

6. Experimental Data

In our first experiment, we constructed one 64x64 array multiplier special functions, and a second using individual gates. XOR functions were used to compute sums, and a custom function was used to compute carries. This function can be computed using the "End Pairs" state machine of Figure 6. To measure the performance of these two circuits, they were simulated on a SUN 300MHz single processor Ultra SPARC-II with 128MB of RAM, using 5000 randomly generated vectors. Figure 15 gives results CPU seconds. As expected, the implementation using functions was significantly faster.

Implementation	CPU Sec.
Gates	**74.0**
Functions	**15.9**

Figure 15. Simulation Times for 64x64 Multipliers.

We conducted a similar experiment using carry-lookahead adders of various sizes. These circuits were constructed from 4-bit carry-lookahead units arranged hierarchically. To support the hierarchical structure, each unit computes three carries, group propagate and group generate functions. Each unit has four propagate and four generate inputs, as well as a carry-in.

Several carry-lookahead adders were created using these units. There were two different implementations, one using AND and OR gates, and another using single-function implementations of for carry and group outputs.. The results are reported in CPU seconds in Figure 16. The performance gains, though substantial, are smaller than those for the array multiplier.

7. Conclusion

The techniques of this paper can significantly increase the power and speed of gate-level simulation. Many different types of functions can be simulated as single gates. Although the Inversion Algorithm was able to collapse heterogeneous (AND-OR) connections, only a portion of the could be eliminated. This paper presents a much simpler method for handling such connections.

Size	Gate Implementation	Function Implementation
4x4	**0.3**	**0.2**
8x8	**0.6**	**0.5**
16x16	**1.2**	**0.9**
32x32	**3.0**	**2.0**
64x64	**7.3**	**5.3**
128x128	**16.1**	**13.6**
256x256	**31.8**	**24.4**
512x512	**85.8**	**57.4**

Figure 16. Carry-Lookahead Adder Performance.

Although the current experimental work focused on the zero delay timing model, it is clear from [7] that these techniques are extensible to more detailed timing models.

8. References

E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events," *J ACM*, Vol. 21, No. 9, Sept., 1978, pp. 777-85.

1. Szygenda, S., D. Rouse, E. Thompson, "A Model and Implementation of a Universal Time-Delay Simulator for Large Digital Nets," *Spring Joint Computer Conference,* 1970, pp. 491-496.
2. Chiang, M., and R. Palkovic, "LCC Simulators Speed Development of Synchronous Hardware," *Computer Design*, Mar. 1, 1986, pp. 87-91.
3. P. Smith, M. R. Mercer, B. Brock, "Demand Driven Simulation: BACKSIM", *Proceedings of the 24th Design Automation Conference*, 1987, pp.181-87.
4. P. M. Maurer, "The Inversion Algorithm for Digital Simulation," Proceedings of ICCAD-94, pp. 259-61.
5. P. M. Maurer, W. J. Schilp, "The Three-Valued Inversion Algorithm," Submitted for Publication. Available from the author (maurer@csee.usf.edu).
6. W. J. Schilp, P. M. Maurer, "Unit Delay Simulation with the Inversion Algorithm," Proceedings of ICCAD-96, pp. 412-7.
7. D. Schuler "Simulation of NAND Logic," Proceedings of COMPCON 72, Sept 1972, pp. 243-5.
8. M. Heydemann, D. Dure, "The Logic Automation Approach to Accurate Gate and Functional Level Simulation," Proceedings of ICCAD-88, pp. 250-253.
9. Sosic, R, J. Gu, R. Johnson, The Unison Algorithm, Fast Evaluation of Boolean Expressions, TODAES Vol 1, No. 4, Oct. 1996, pp. 456-477.