

# Converting BDDs, Boolean Expressions and Truth Tables to Simulation State-Machines

**Peter M. Maurer**  
**Department of Computer Science and Engineering**  
**University of South Florida**  
**Tampa, Florida 33620**

## 1. Abstract

State machine-based simulation is an effective way to perform efficient logic-level simulations for design verification. The main drawback is that in many cases the logic model of the circuit is not available until the later stages of the design. The techniques presented in this paper allow efficient state-machine base simulations to be used much earlier in the design cycle. The techniques presented here can be used to create efficient state-machine-based simulations for circuits specified using Binary Decision Diagrams (BDDs), Boolean equations, and truth tables. Input symmetries can be used to simplify state machines. These techniques can be combined with gate-level techniques to create a simulator that is usable at many different stages of the design cycle.

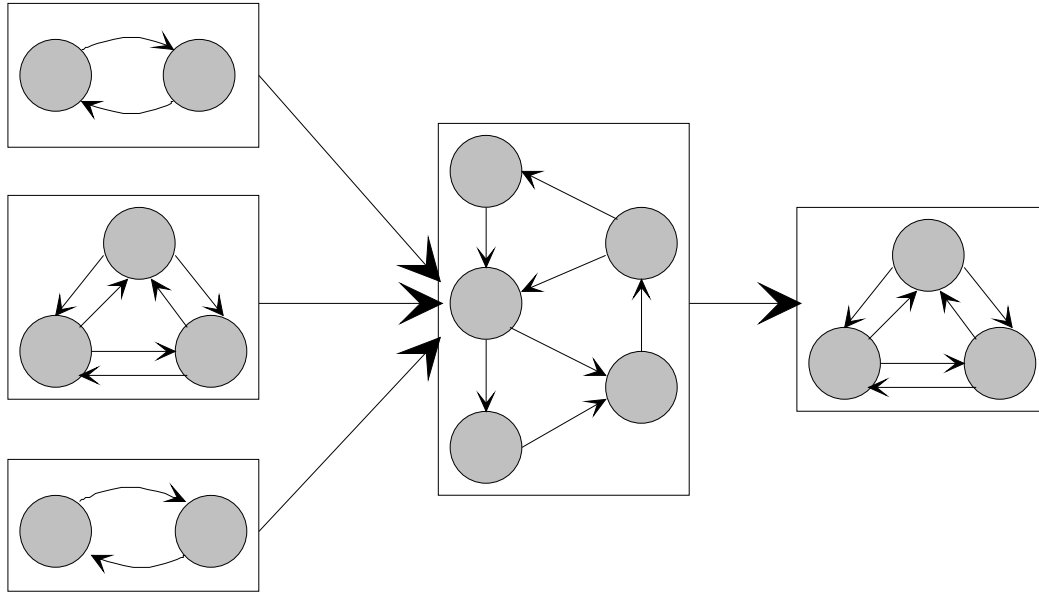
## 2. Introduction

Logic-level simulation[1-6] is an effective method of design verification. State machine-based simulation[7-11] is a high-performance event-driven technique for performing logic-level simulation. Existing state-machine-based simulation techniques assume that circuits are specified as networks of gates. However, many of these techniques are also designed to be used for design verification, which is typically done at the earliest stages of circuit development. At this stage of development, a gate-level description of the circuit may not be available for the entire circuit. Some portions of the circuit may exist only as a set of Boolean equations. At the earliest stages of development the circuit may exist only as a set of Binary Decision Diagrams (BDDs)[12,13] or even truth tables. Existing state-machine simulation techniques cannot be used for these types of specifications. On the other hand, this is the point in the design where verification is most important. Any bugs introduced at the high level will be propagated through the lower levels of the design. The more automated the design process becomes, the more true this is. This paper introduces a new method of creating state-machine simulations that can be used on many different types of specifications, including BDDs, Boolean expressions, and truth tables. These techniques can be combined with those of [7,10,11] to produce an efficient simulator that can be used at many different stages of circuit design.

## 3. State Machine Simulation

Our approach uses the basic state machine technique used in the Inversion Algorithm, the EVCF technique, and the general state-machine approach[7,11,14]. In this approach there are three types of state machines, input state machines, gate state machines, and queueing state machines. These state machines are interlocked in the sense that the output

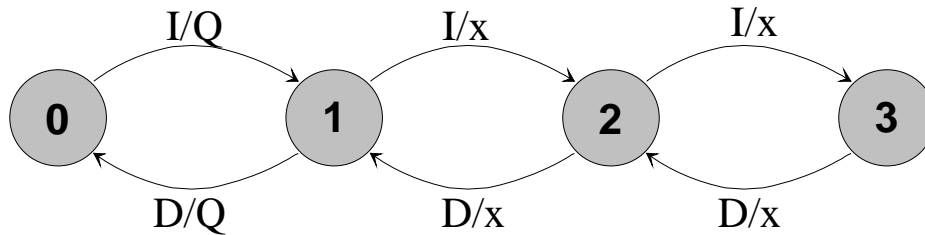
of one machine becomes the input of another. Figure 1 illustrates the basic structure of these state machines.



**Figure 1. A Sample Network.**

Our state machines have two different types of outputs, Mealy outputs which are used for communication between machines, and Moore outputs that are used during the design phase of the machines and then discarded. During simulation, only the Mealy outputs exist. Input state machines are normally two-state machines that are used to represent binary nets. More complex state machines can be used to implement multi-valued logic models. Different types of input state machines can be freely mixed in a single simulation, and it is possible to dynamically change the logic model used for a particular input. Most queueing state machines are also two-state machines. Again, more complex queueing machines are required to support multi-valued logic models.

Gate simulation is done using gate state machines, such as that pictured in Figure 2. The advantage of using such state machines is that inputs can be handled one at a time, without the need for evaluating multi-input functions.



**Figure 2. A Gate State Machine.**

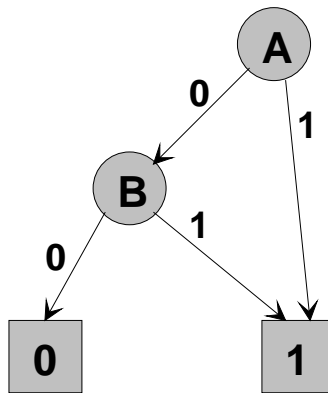
In Figure 2, the inputs I and D represent Increment and Decrement operations that are performed on the dominant-count of a gate. Not all gate state machines have the linear structure pictured in Figure 2. More complex gates such as AOI and OAI gates have multi-dimensional state machines. The most detailed type of gate state machine is an n

dimensional hypercube which is used to simulate an n-input gate. In most cases it is possible to use the symmetries in the gate input to reduce the complexity of such machines. (We note that no gate actually requires an n-dimensional hypercube machine.) Because the state machine technique allows inverted inputs and outputs to be treated the same as non-inverted inputs and outputs, the opportunities for finding input symmetries are greatly enhanced.

In the remainder of this paper we will describe techniques for generating gate state machines from BDDs, Boolean equations, and truth tables. It is likely that these techniques can be used with other representations as well.

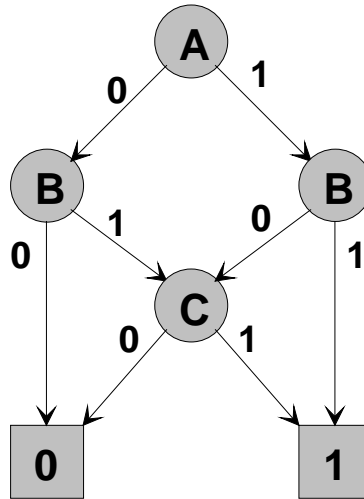
#### 4. Generating State Machines from BDD Representations

Binary Decision Diagrams (BDDs) are a popular mechanism for specifying and manipulating the design of a circuit. Unlike Boolean equations and gate representations, Reduced Ordered BDDs (ROBDDs) are known to be canonical for the function they represent. Figure 3 shows a BDD. The circles represent an input variable test with two possible outcomes, while the squares represent the final outcome of one or more tests. As with gate state machines, the BDD of Figure 3 can be used to represent several different functions by relabeling the arcs and outcome nodes.



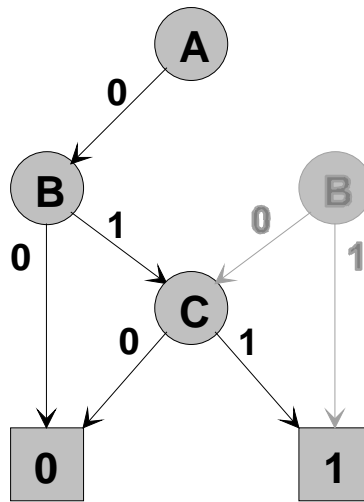
**Figure 3. A BDD for a Two-Input OR Gate.**

Any BDD representation of a function can be transformed into a state machine suitable for simulation. If no state reductions are used, an n-dimensional hypercube machine will be produced, but the use of input symmetries can greatly simplify the generated machine. To illustrate the BDD technique, consider the carry output of a full adder. This is a fully symmetric three-input function whose BDD representation appears in Figure 4.

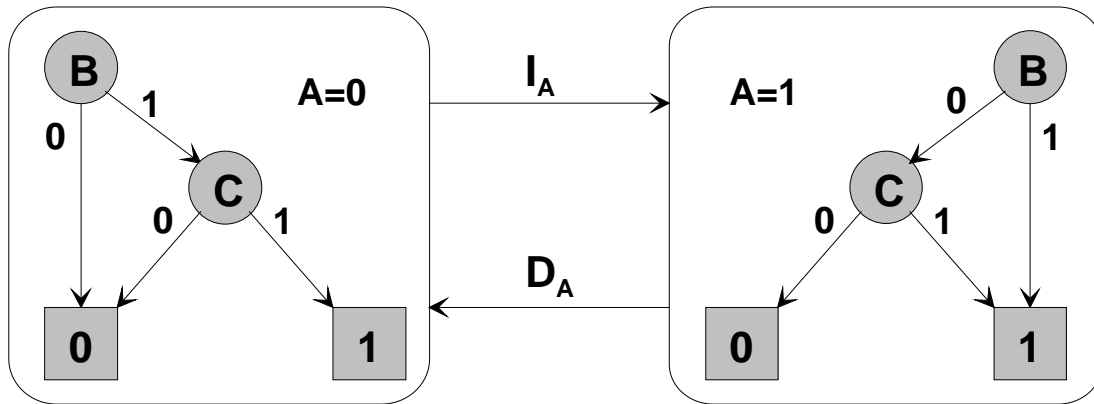


**Figure 4. A Three-Input Carry Function.**

To produce a state machine from this BDD, we eliminate one variable at a time, and replace the variable with a pair of states. For example, we eliminate the variable A by setting A equal to zero and computing the BDD for the resultant two-input function. We then repeat this procedure by setting A equal to 1. The resultant BDDs will be used to label the two states of the new state machine. When setting A equal to zero, we eliminate the edge labeled 1, and remove all unreachable vertices, as illustrated in Figure 5. We then eliminate the node for A, and shortcut any incoming edges through to the target vertex of the remaining edge. (Since we have eliminated the topmost node, shortcutting is unnecessary.) The resultant state machine is given in Figure 6.

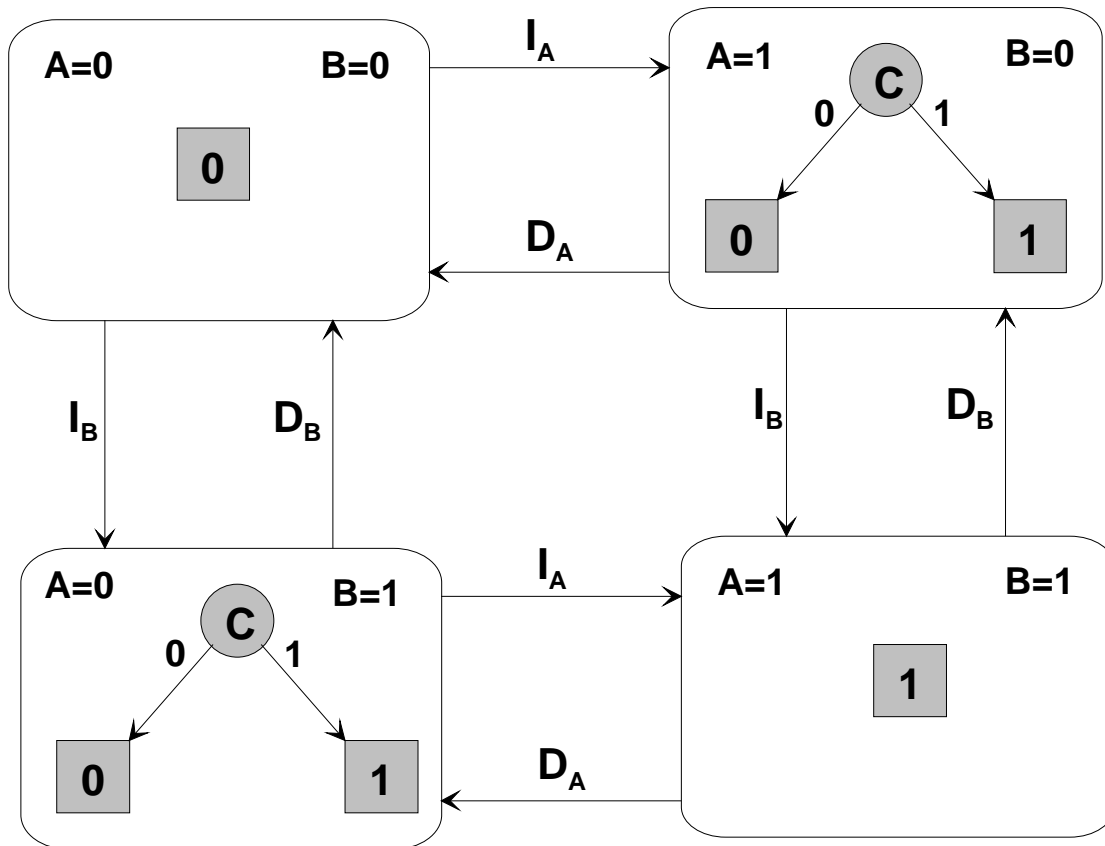


**Figure 5. Eliminating a Variable.**



**Figure 6. The First BDD Reduction.**

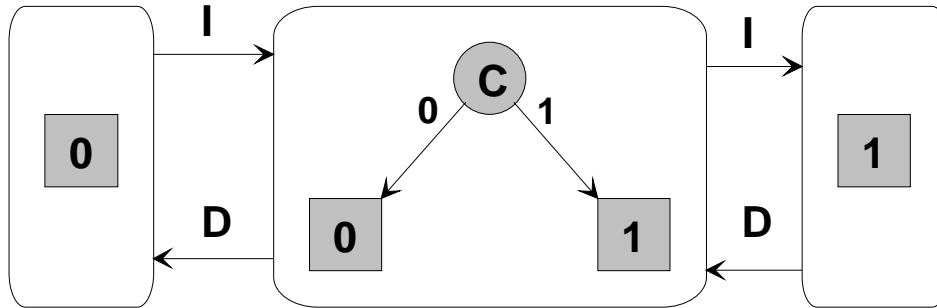
Eliminating the second variable, B, produces the state machine pictured in Figure 7. The symmetric inputs A and B have not yet been identified with one another. This step can be left for the end, but it is more efficient to perform it immediately. Identifying the inputs and performing a state reduction will produce the state machine of Figure 8.



**Figure 7. The Second BDD Reduction.**

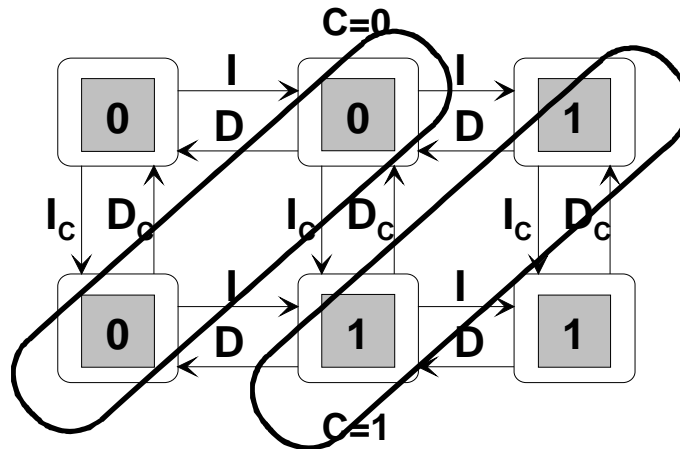
When reducing the state machine of Figure 7 to that of Figure 8, the BDD label of a state is treated as its Moore output. Thus in Figure 8, there are three different Moore outputs, the constant 0, the constant 1 and the BDD that tests the state of C. As with

conventional state reduction, states can be combined only if their Moore outputs are the same.



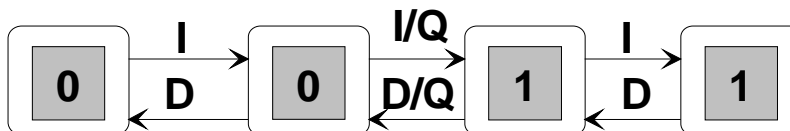
**Figure 8. The Reduced Intermediate Machine.**

Figure 9 shows the result of the elimination of the final variable C. Again, it is possible to identify inputs and perform a state reduction.



**Figure 9. The Third BDD Reduction Step.**

The state machine of Figure 10 shows the result of the final state machine reduction. Mealy outputs are added to the transitions that represent a change in the Moore output.

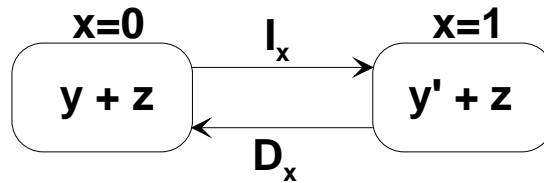


**Figure 10. The Final State Machine.**

## 5. Boolean Expressions

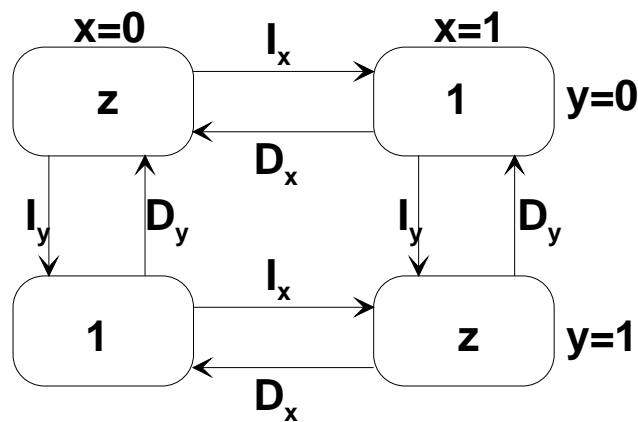
The variable elimination technique can also be used with Boolean expressions. Given a function  $f(x_1, \dots, x_n)$ , the techniques for creating  $f_1(x_2, \dots, x_n)$  and  $f_0(x_2, \dots, x_n)$  are already well known. We can use these techniques to eliminate variables. As with BDDs, the resultant functions  $f_1$  and  $f_0$  will be used to label the states of a state machine. Consider for example the Boolean expression  $x'y + xy' + z$ . This expression is partially symmetric in the variables  $x$  and  $y$ . This fact can be used to simplify the resultant state

machine. Figure 11 shows the state machine that would result from eliminating the variable  $x$ .

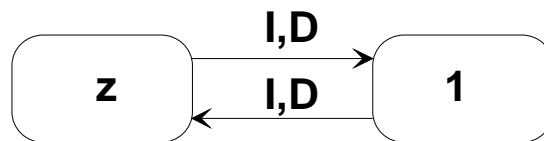


**Figure 11. An Expression State Machine.**

Next we eliminate the variable  $y$ , and reduce the state machine using the partial symmetry of inputs  $x$  and  $y$  to reduce the state machine. Figure 12 and Figure 13 show this process.



**Figure 12. Eliminating Two Variables From an Expression.**



**Figure 13. The Reduced Expression State Machine.**

Finally, we eliminate the variable  $z$ , producing the state machine of Figure 14. This state machine still needs to be enhanced with mealy outputs on the transitions between the 0 and 1 states.

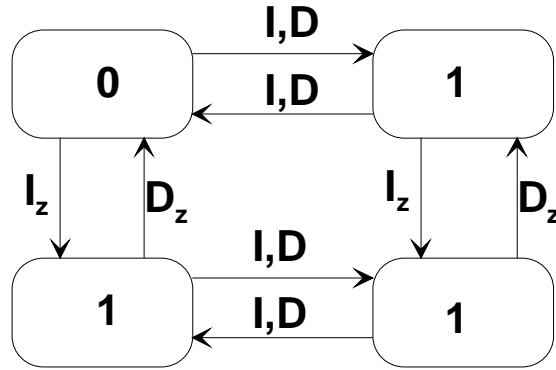


Figure 14. The Final Expression State Machine.

## 6. Truth Tables

Consider this. Assume that the variable names are  $x$ ,  $y$ ,  $z$ , and  $w$ , and that the truth table of Figure 15 expresses the value of these variables in the order specified. Although it may not be apparent from the table, this function is partially symmetric in  $x$  and  $y$ , and is also partially symmetric in  $z$  and  $w$ .

0000	0	0100	0	1000	0	1100	0
0001	0	0101	1	1001	1	1101	1
0010	0	0110	1	1001	1	1110	1
0011	0	0111	1	1011	1	1111	0

Figure 15. A Truth Table.

We can create a state machine from this truth table. First we must divide the entries into two sets, one that represents the zeros of the function, and one that represents the ones. This gives us the following pair of sets. For convenience, we will express the input combinations in decimal.  $\{0,1,2,3,4,8,12,15\}\{5,6,7,9,10,13,14\}$ .

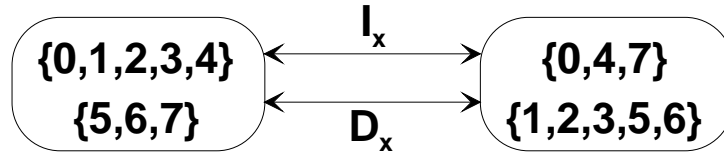
To eliminate  $x$ , we need to create two pairs of sets, one that corresponds to  $x=0$ , and one that corresponds to  $x=1$ . These sets are easy to create, because the entries in the above pair that correspond to  $x=0$  are those less than 8, while those that correspond to  $x=1$  are those that are greater than or equal to 8. When processing the entries for  $x=1$ , it is necessary to subtract the value 8 from each entry to reflect the fact that  $x$  has been eliminated as an input to the function.

Applying this procedure to the above pair of sets gives the following pairs of sets.

$$x=0, \{0,1,2,3,4\} \{5,6,7\}$$

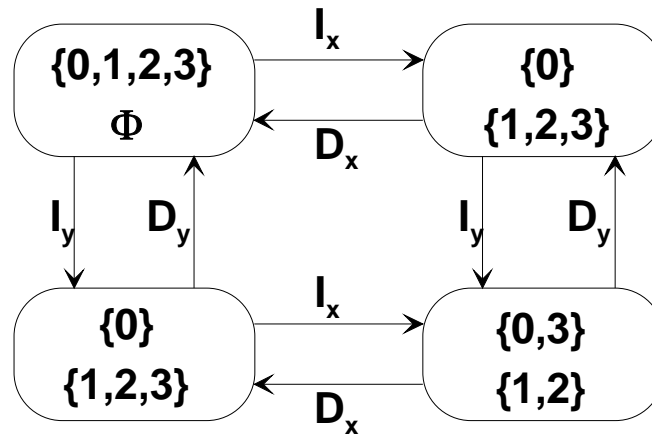
$$x=1, \{0,4,7\} \{1,2,3,5,6\}$$

As with BDDs and expressions, we use these pairs of sets to label the states of a state machine as illustrated in Figure 16.



**Figure 16. A Truth-Table Machine.**

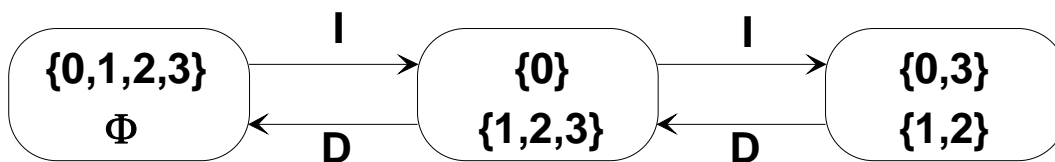
We apply the same procedure to each of these pairs of sets. In this case, the values corresponding to  $y=0$  are those that are less than 4, and when we process the values for  $y=1$  we must subtract 4 from each element. After creating these new sets and using them to label the states of a state machine we obtain the state machine of Figure 17.



**Figure 17. Eliminating Two Variables from a Truth Table.**

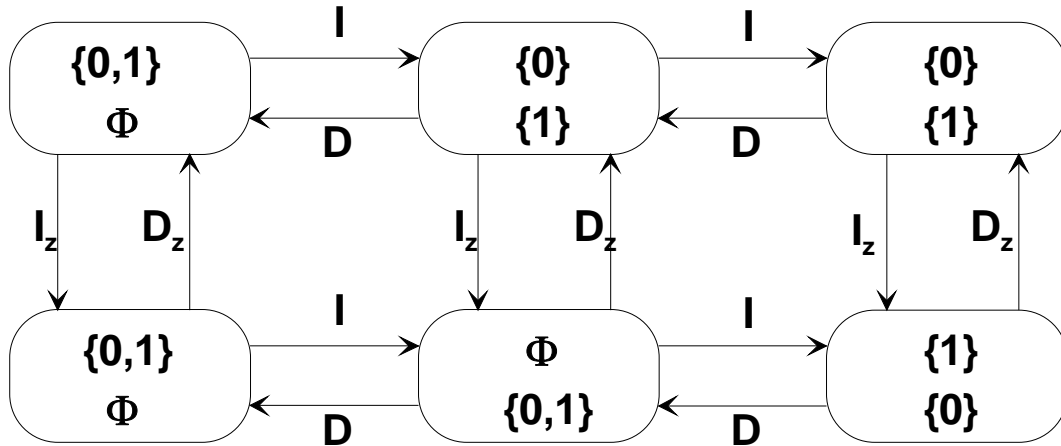
When one of the sets becomes empty, this indicates that a constant value has been achieved for the state. The variable-elimination procedure must continue until each state achieves a constant value.

As noted above, this function is symmetric in  $x$  and  $y$ . Note also that the set pair  $\{0\}\{1,2,3\}$  is used as the label for two different states. Because of this, it is possible to perform a state reduction on this state machine, giving the state machine of Figure 18.



**Figure 18. Reducing the Truth Table Machine.**

Next, we eliminate  $z$  giving the machine of Figure 19. In this case the values less than two correspond to  $z=0$ , and we must subtract 2 from each element when processing  $z=1$ .



**Figure 19. Eliminating Three Variables from a Truth Table.**

Finally, we eliminate the last variable,  $w$ . The resultant state machine is given in Figure 20. We have simplified the diagram by creating two copies of the preceding state machine and placing them into a pair of “super states.” When a transition is made between super states, the actual transition occurs between the two sub-states in corresponding positions.

At this point each state contains exactly one occurrence of the empty set. The position of the non-empty set in the pair determines the Moore output of the state. If the first set of the pair is non-empty, then the value is zero, otherwise it is one. As mentioned above, the inputs  $z$  and  $w$  are symmetric. A careful examination of Figure 20 will also reveal that the bottom row of states in the first super state is identical to the top row of states in the bottom super state. This will enable us to combine the inputs  $z$  and  $w$  and perform a state reduction. We will replace the set pairs with their Moore outputs at the same time, giving the state machine of Figure 21.

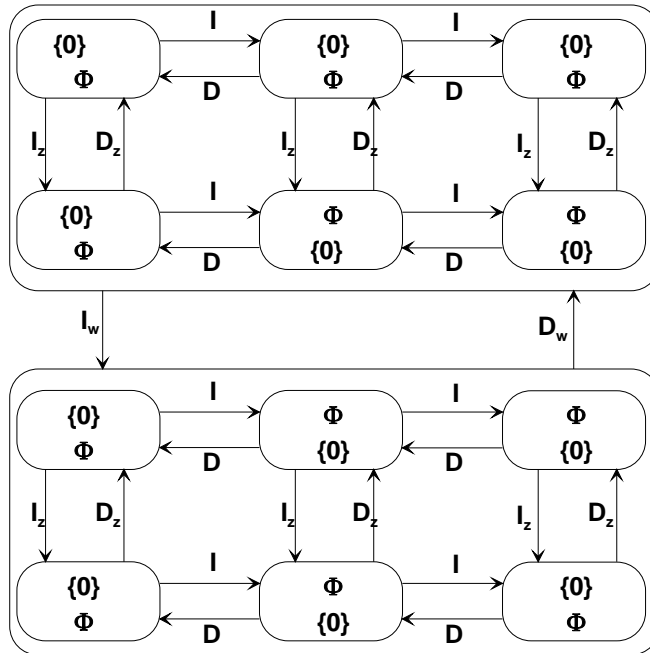


Figure 20. A State Machine with Super States.

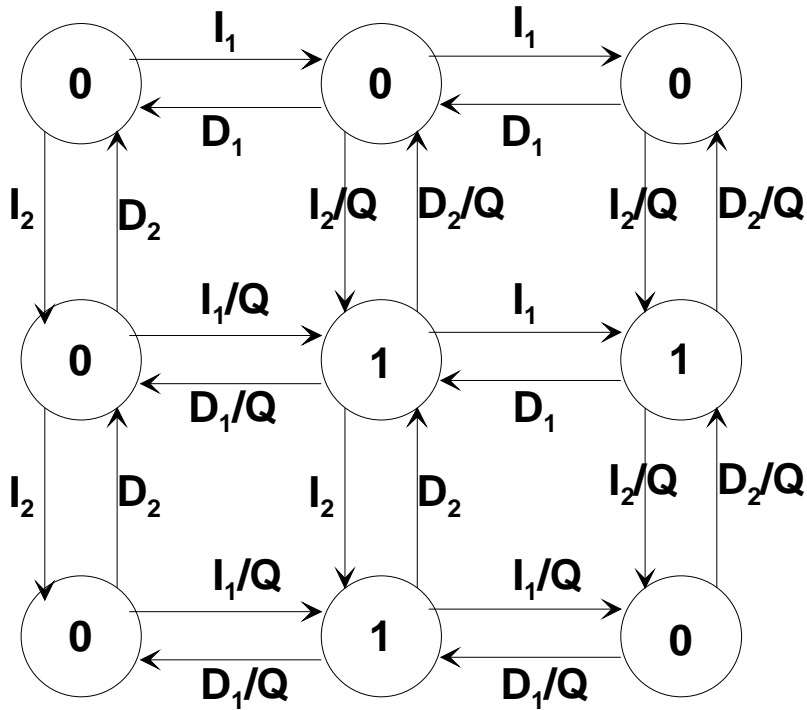


Figure 21. The Final Truth-Table Machine.

There are many different ways to specify truth table values. The techniques used in this section can easily be extended to these other representations.

## 7. Conclusion

The state-machine analysis presented in this paper can significantly increase the simulation power and speed of gate-level and functional simulations. The techniques presented here are widely applicable, and can be applied to designs specified using gates, BDDs, Boolean equations, and truth tables. The variable-elimination technique is the most effective but it requires a knowledge of input symmetries for optimum efficiency. The gate-based techniques can use the symmetries of the gates themselves for state machine reduction, but do not produce optimum results with replicated inputs.

The techniques described here clearly identify the three types of state machines used in simulation, input machines, gate machines, and queueing machines. Separating the implementations of these machines permits a mixing and matching of various types of simulation models, without imposing severe restrictions. The techniques presented here are easily adaptable to multi-delay and nominal-delay timing simulations. Existing scheduling techniques can be used without the need for complex optimization algorithms [23]. This is an important innovation, because it allows state machine techniques to be integrated with virtually all past and current simulation research.

Perhaps the most intriguing aspect of state-machine based simulation is its potential for stimulating future research and development in simulation. Although we have experimented extensively with implementations of these techniques, we have barely scratched the surface of the techniques presented in this paper. The techniques presented here can be combined in so many different ways, that a complete exploration of the various possibilities could take many years. One potential area for future investigation is the area of meta-elements in the simulation. Meta-elements are simulation structures that superficially resemble gates and nets, but are used to dynamically tune the simulation rather than to produce simulation results. This concept was explored briefly in [17] for the purpose of converting three-valued nets into two-valued nets, and in [25] for the purpose of disabling portions of the simulation. Despite this future potential, the techniques already described here are sufficient in themselves to provide significant improvements in the speed of existing simulations.

## 8. References

1. E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events," *JACM*, V.21, N.9, Sep. 1978, pp. 777-85.
2. Lewis, D. M. "A Hierarchical Compiled Code Event-Driven Logic Simulator," *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.
3. R. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proceedings of the 24th Design Automation Conference*, 1987, pp. 9-16.
4. Barzilai, Z., J. L. Carter, B. K. Rosen J. D. Rutledge, "HSS-A High-Speed Simulator," *IEEE Transactions on Computer Aided Design*, Vol. CAD-6, No. 4, , pp. 601-617, July, 1987.
5. 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.

6. Z. Wang, P. Maurer "LECSIM: A Levelized Event Driven Compiled Logic Simulator," *Proceedings of the 27th Annual Design Automation Conference*, 1990, pp. 491-496.
7. P. Maurer, "The Inversion Algorithm for Digital Simulation" *IEEE Transactions on Computer Aided Design*, Vol. 16, No. 7, July 1997, pp. 762-769.
8. D. Schuler "Simulation of NAND Logic," *Proceedings of COMPCON 72*, Sept 1972, pp. 243-5.
9. M. Heydemann, D. Dure, "The Logic Automation Approach to Accurate Gate and Functional Level Simulation," *Proceedings of ICCAD-88*, pp. 250-253.
10. P. M. Maurer, W. J. Schilp, "The Three-Valued Inversion Algorithm," *Proceedings of the 2000 Conference on VLSI*.
11. P. Maurer, "Event Driven Simulation Without Loops or Conditionals," *Proceedings of ICCAD-2000*, 2000, pp. 23-26.
12. Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, vol. C-35, Aug. 1986, pp. 677-691.
13. P.C. McGeer, K.L. McMillan, A. Saldanha, A.L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast Discrete Function Evaluation using Decision Diagrams", *Proceedings ICCAD-95*, 1995, pp. 402-407.
14. P. Maurer, "Logic simulation using networks of state machines," *Proceedings DATE-2000*, pp. 674-678, Mar. 2000.