Gate-Level Simulation of Digital Circuits Using Multi-Valued Boolean Algebras

Scott Woods Giorgio Casinovi

School of Electrical & Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250

Abstract

This paper describes an algorithm for the simulation of gate-level logic. Multiple logic levels are used to describe the state of each node. Each state corresponds to a different voltage level, and the number of levels to be used for a simulation is user-defined. This feature simplifies considerably the interface between a digital and an analog simulator. A DC solver is incorporated to find the initial operating point of a circuit before a transient analysis begins. This solver has the capability of finding the operating point of gates located in feedback loops. For transient analysis, a gate delay model that takes into account the slope of the input waveforms is used. The performance of the algorithm is demonstrated by simulations of a number of benchmark circuits.

1 Introduction

As the percentage of integrated circuits that contain both analog and digital components keeps growing, so does the importance of mixed-mode simulation as a verification tool. Extensive literature exists on this topic: for a detailed bibliography the interested reader is referred to [1, 2] and to the references listed therein. A feature common to all published mixed-mode simulation algorithms is that they must trade-off speed versus accuracy, in particular when it comes to simulating the digital portion of a circuit. In terms of speed, gate-level simulation would have to be the preferred choice, because it can run almost three orders of magnitude faster than a circuit level simulation. Even higher levels of simulation such as timing simulation can run up to 100 times slower than a gate-level simulation [2]. On the other hand, it is also desirable to keep as much information as possible about signal levels: denoting the output of a gate as "unknown," as is done in most logic simulators, provides no information about the actual voltage present at the gate's output. This causes problems at the digital-analog interface of mixed-mode simulators.

An attempt to reconcile these two conflicting requirements was made with the development of so called electrical-logic analysis, or ELOGIC [3]. This is a switch-level timing analysis technique in which signals can take any of a number of user-specified voltage levels between a logic zero and a logic one. The ELOGIC simulation algorithm then computes the time necessary for a node voltage to change its value from its current level to an adjacent one. In this way it is possible to control the signal resolution, and in particular to trade off speed for accuracy. However, the simulation is still performed at the device level using ordinary device models, so that the computational effort required is still substantially greater than gate-level logic simulation [2]. A different approach is given in [1], where a functional-level mixed-mode simulator is described. As in ELOGIC simulation, an arbitrary number of voltage levels between a logic zero and a logic one is available to represent signal values. Ordinary AND and OR Boolean operations are replaced by *min* and *max* operations on the signal levels. In this way logic functions can be simulated at the gate level, and a certain number of high-level analog operations can be handled as well.

The simulation algorithm described here also tries to retain the speed of traditional gate-level simulation, while at the same time maintaining enough information on the waveforms generated by digital logic gates to compute accurate delays and provide lower level simulators with detailed data. In particular, multiple logic states (levels) are used to describe the state of each node. Each state corresponds to a voltage level, and the the number of levels to be used for a simulation is user-defined. These logic levels provide more information about a waveform than the traditional states (0, 1, X, Z), so slopes of waveforms can be determined and accurate times can be calculated for threshold crossings. There are however a number of differences between the algorithm described here and the one proposed in [1]: one is that our implementation requires only ordinary Boolean algebra operations (except for computing delays). As a consequence of this fact, it is possible to determine the initial state of the circuit by solving a set of Boolean algebraic equations, so that a valid operating point for the circuit can be determined before a transient analysis begins,

This work was supported in part by the National Science Foundation under Grant MIP-9211163.

thus eliminating the need for an unknown state. The DC solver has the ability to handle gates in feedback loops, so it is not limited to combinational circuits.

2 Multiple-level logic

An obvious way to try to combine the speed of gate-level logic simulation with the finer resolution of multiple signal levels is to use Boolean algebras containing more than two elements. A complete mathematical treatment of such algebras is beyond the scope of this paper, and can be found, for instance, in [4, 5]. In brief, a Boolean algebra is a set of elements on which three operations, called AND ($\wedge$), OR ($\vee$) and NOT ($\overline{\ }$), have been defined. It can be shown that, as a consequence of the laws that Boolean operations must satisfy, a finite Boolean algebra must contain exactly 2^n elements.

An order relation ($\leq$) can be introduced in a Boolean algebra by the following definition:

$$x \leq y \Leftrightarrow x \wedge y = x$$

or equivalently:

$$x \leq y \Leftrightarrow x \vee y = y.$$

Unfortunately this order relation is only *partial*, not *total* (except in the case of a two-element algebra): this means that there exist pairs of elements x, y such that neither $x \leq y$ nor $y \leq x$ is true. It can be shown that if $z = x \wedge y$, then *z* is the largest element such that $z \leq x$ and $z \leq y$: therefore, the AND operation can can be regarded as a sort of *min* operation on the the Boolean algebra. Similarly, if $z = x \vee y$, then *z* is the smallest element such that $x \leq z$ and $y \leq z$.

At a first look it would appear that the easiest way to perform logic simulation with more than two signal levels would be to use a Boolean algebra with more than two elements, with each element in the algebra corresponding to a different signal level. However, because the various Boolean operations are supposed to model the behavior of physical gates, additional constraints must be taken into account: for instance, it seems reasonable to require that the output of an AND gate be equal to the lowest input level (i.e. the AND Boolean operation must behave like the *min* operator on the input signals [1]). Similarly, the OR operation should yield the *max* of the inputs. Unfortunately those requirements are incompatible with the lattice structure of Boolean algebras, which are only partially ordered sets, while the set of all possible signal values is obviously totally ordered. This type of problem, which presents itself whenever Boolean algebras with more than two elements are involved, was already pointed out in [6].

This difficulty can be overcome by representing the value of a signal with a *pair* of Boolean algebra elements in the following way. Let $A = \{a_0 \leq a_1 \leq \ldots \leq a_n\}$ and $B = \{b_0 \leq b_1 \leq \ldots \leq b_n\}$ be two chains in a Boolean algebra such that $a_0 = b_0 = 0$ and $a_n = b_n = 1$, and such that $a \in A \Rightarrow \overline{a} \in B$ and $b \in B \Rightarrow \overline{b} \in A$. A signal level can now be represented by a pair (x^I, x^F), with $x^I \in A$ and $x^F \in B$: the pairs $(0,0)$ and $(1,1)$ represent the signal levels corresponding to logic 0 and logic 1, respectively, while other pairs correspond to intermediate levels. It is now possible to define Boolean operations on the signals in the following way:

$$\begin{aligned}
(x_1^I, x_1^F) \wedge (x_2^I, x_2^F) &= (x_1^I \wedge x_2^I, x_1^F \wedge x_2^F) && (1)\\
(x_1^I, x_1^F) \vee (x_2^I, x_2^F) &= (x_1^I \vee x_2^I, x_1^F \vee x_2^F) && (2)\\
\overline{(x_1^I, x_1^F)} &= (\overline{x}_1^F, \overline{x}_1^I). && (3)
\end{aligned}$$

It is easy to verify that, with those restrictions placed on the pair (x^I, x^F), the logic operations thus defined behave on signals as desired: the output of an AND is the minimum among the inputs, the output of an OR the maximum, and a NOT generates a signal level which is as close to a logic zero as the input was to a logic one.

The signal representation described above can be implemented by describing each logic level by a binary string of length $2n$, divided in a lower and an upper half: the lower half represents x^I and the upper half represents x^F in the pair (x^I, x^F). The number of ones in either half of this binary string determines the logic level: if the string contains all zeros, it is a logic zero, and if the string contains all ones, it is a logic one. Representations of intermediate levels are obtained by shifting the same number of ones in both halves of the string. In the lower half of the string the ones must be shifted in from the right, so that all the least significant bits of the string are ones and the the most significant bits are zeros. The opposite operation is performed on the upper half of the binary string: the ones must be shifted in from the left, so that all the ones are contained in the most significant bits and all the zeros are in the least significant bits. Since both halves always have the same number of ones and zeros, the two halves will always be symmetrical about the middle. As an example, a representation with five signal levels is shown Table I, with 0V corresponding to a logic zero, and 5V corresponding to a logic one. In this example it is assumed the intermediate levels are equally spaced, but this does not necessarily have to be the case.

The signal operations defined in eqns. (1–3) require only simple bitwise boolean operations. This eliminates the need for table lookups and the overhead of generating tables. As long as the entire binary string can be stored in a single computer word, the AND or OR of any two signals require only a single bitwise operation, regardless of the number of levels involved. The complementation operation (NOT) is slightly more complicated, because it requires complementing the entire binary string bitwise, and then swapping

TABLE I
A five-level signal representation

level	voltage	binary string
1	0.00	0000 0000
2	1.25	1000 0001
3	2.50	1100 0011
4	3.75	1110 0111
5	5.00	1111 1111

the upper half with the lower half. A few examples of the AND, OR, and NOT operations performed on signals with five logic levels are shown below.

$$\begin{aligned}\mathbf{AND}:\ (1111\,1111)\wedge(1100\,0011) &= (1100\,0011)\\ (1000\,0001)\wedge(1110\,0111) &= (1000\,0001)\end{aligned}$$

$$\begin{aligned}\mathbf{OR}:\ (1111\,1111)\vee(1100\,0011) &= (1111\,1111)\\ (1000\,0001)\vee(1110\,0111) &= (1110\,0111)\end{aligned}$$

$$\begin{aligned}\mathbf{NOT}:\ \overline{(1110\,0111)} &= (1000\,0001)\\ \overline{(1100\,0011)} &= (1100\,0011)\end{aligned}$$

Note that if the number of logic states is odd, there is a middle state which is the complement of itself (such as in the last example of the NOT function). This state allows for a valid operating point for feedback loops where an odd number of inversions exists.

3 Initialization

Most switch-level or gate-level simulation algorithms have no mechanism for computing the initial operating point of a digital circuit containing feedback loops. The common solution is to initialize all the nodes to an undefined state *X*, and then to continue the simulation with the usual rules of three-element logic algebra. It will be shown next that, using the signal representation introduced in the previous section, it is possible to compute the initial state of any digital circuit, if one exists, or to determine that one does not exist, or that more than one exists, and, in the last case, what nodes may have more than one solution and must therefore be truly considered to have an undefined state. All this is possible because, as pointed out earlier, all operations on signals can be expressed in terms of ordinary Boolean algebra operations.

In traditional circuit simulation, the DC operating point of a circuit is found by computing a solution of a set of real algebraic equations. Similarly, the initial state of a digital circuit can be computed by solving a set of boolean

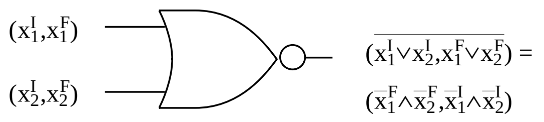
(x_1^I, x_1^F)
(x_2^I, x_2^F)
$\overline{(x_1^I \vee x_2^I, x_1^F \vee x_2^F)} =$
$(\bar{x}_1^F \wedge \bar{x}_2^F, \bar{x}_1^I \wedge \bar{x}_2^I)$

Figure 1. Boolean equations for a NOR gate.

algebraic equations: for instance, in [7] Gaussian elimination is used to solve a set of linear boolean equations. In the case considered here, the equations will generally be nonlinear, so a different algorithm is needed. A suitable one is a straightforward extension of Gaussian elimination, namely the method of *successive eliminations*, described below; a broader and more thorough discussion of this algorithm can be found in [8]. At the basis of the algorithm are the following theorems.

Theorem 1 ([4, p. 8]) *Let a, b be two elements in a Boolean algebra. Then $b \le \bar{a} \Leftrightarrow a \wedge b = 0$.*

Theorem 2 ([4, p. 58]) *Let a, b, x be elements in a Boolean algebra. The following statements are equivalent:*

$$\begin{aligned}(a \wedge x) \vee (b \wedge \bar{x}) &= 0\\ b \le x \le \bar{a}.\end{aligned}$$

Consequently, the equation $(a \wedge x) \vee (b \wedge \bar{x}) = 0$ has solutions if and only if $a \wedge b = 0$.

It should be stressed again that the theorems above are valid in any Boolean algebra, not just in the ordinary binary algebra (where they are trivial).

If the state of each node in a digital circuit is described by a pair of Boolean variables (as explained in the previous section), each gate in the circuit generates two Boolean equations. For example, the NOR gate shown in Fig. 1 generates the equations:

$$\begin{aligned}x_3^I &= \bar{x}_1^F \wedge \bar{x}_2^F\\ x_3^F &= \bar{x}_1^I \wedge \bar{x}_2^I.\end{aligned}$$

The above equations have the form:

$$x_i = f_i(x_1, \ldots, x_n)$$

which, by the laws of Boolean algebra, is equivalent to [4]:

$$(x_i \wedge \bar{f_i}) \vee (\bar{x}_i \wedge f_i) = 0.$$

Therefore there is no loss of generality in assuming that the set of equations to be solved has the form:

$$f_i(x_1, \ldots, x_n) = 0 \qquad i = 1, \ldots, m. \tag{4}$$

This system is equivalent to the single equation:

$$F^1(x_1, \ldots, x_n) = \bigvee_{i=1}^{m} f_i(x_1, \ldots, x_n) = 0. \tag{5}$$

Algorithms to solve Boolean equations in more than one unknown use the result stated in Theorem 2 and the Shannon decomposition of Boolean functions [9]. The standard method [4] to solve eqn. (5) consists of eliminating one unknown at a time using the Shannon decomposition: if $F^1 = (x_1 \wedge F^1_{x_1}) \vee (\bar{x}_1 \wedge F^1_{\bar{x}_1})$, then eqn. (5) can be rewritten as:

$$(x_1 \wedge F^1_{x_1}(x_2, \ldots, x_n)) \vee (\bar{x}_1 \wedge F^1_{\bar{x}_1}(x_2, \ldots, x_n)) = 0.$$

By Theorem 2, x_1 must then satisfy the inequalities:

$$F^1_{\bar{x}_1}(x_2, \ldots, x_n) \le x_1 \le \bar{F}^1_{x_1}(x_2, \ldots, x_n) \qquad (6)$$

which can be satisfied if and only if the following equation is satisfied:

$$F^2(x_2, \ldots, x_n) = F^1_{\bar{x}_1}(x_2, \ldots, x_n) \wedge F^1_{x_1}(x_2, \ldots, x_n) = 0.$$

Thus the number of unknowns has been reduced by one from the original equation, because the function F^2 does not depend on x_1. Recursive application of this technique reduces the original system of equations to one equation in one unknown, whose solutions (if they exist) can be determined through the use of Theorem 2. By back substitution, values for the other unknowns can be computed through inequalities of the type shown in eqn. (6).

However, if this algorithm were implemented exactly as described above, the computational effort required to solve even a system of moderate size would quickly exceed practical limits. To get around this problem, a slight modified version of this algorithm is described next. This implementation relies on the sparsity of the system of equations being solved: in this case, this means that each equation in the system depends explicitly only on a small subset of the total number of unknowns. It is well-known that the equations describing an electrical network are almost always sparse [10]. To take advantage of this fact the equations will be split into smaller groups, according to the unknowns that affect them. Let:

$$S_i = \{j : f_j \text{ depends on } x_i \text{ but not on } x_1, \ldots, x_{i-1}\},$$

and define:

$$\begin{aligned} G^i(x_i, \ldots, x_n) &= \bigvee_{j \in S_i} f_j(x_i, \ldots, x_n) \\ F^1(x_1, \ldots, x_n) &= G^1(x_1, \ldots, x_n) \\ F^{i+1} &= (F^i_{x_i} \wedge F^i_{\bar{x}_i}) \vee G^{i+1}. \end{aligned}$$

It is immediate to verify that F^i depends only on $x_i, \ldots, x_n$ (the claim is obvious for $i = 1$; by induction, F^i depends only on $x_i, \ldots, x_n$, so $F^i_{x_i}$ and $F^i_{\bar{x}_i}$ depend only on $x_{i+1}, \ldots, x_n$, as does G^{i+1}, and hence F^{i+1}). The solutions of the original system of equations can then be computed in the following way:

```
$(F_x, F_{\bar{x}}) \leftarrow (0, 0)$;
GateList $\leftarrow$ A list of all gates in circuit;
for each node in circuit {
    Let x be the variable at this node;
    $(F_x, F_{\bar{x}}) \leftarrow$ Shannon_Decomp$(x, F_x \wedge F_{\bar{x}})$;
    for each gate in GateList connected to this node {
        Let f be the function describing this gate;
        $(F_x, F_{\bar{x}}) \leftarrow (F_x, F_{\bar{x}}) \vee$ Shannon_Decomp$(x, f)$;
        Remove gate from GateList;
    }
}
```

Figure 2. An algorithm to solve Boolean equations.

Theorem 3 *Any n-tuple $x_1, \ldots, x_n$ that satisfies the following set of inequalities:*

$$F^i_{\bar{x}_i}(x_{i+1}, \ldots, x_n) \;\le\; x_i \;\le\; \bar{F}^i_{x_i}(x_{i+1}, \ldots, x_n), \qquad i = n, \ldots, 1$$

is a solution of the system of equations (4).

Proof: By Theorem 2, the inequalities above imply that:

$$(x_i \wedge F^i_{x_i}) \vee (\bar{x}_i \wedge F^i_{\bar{x}_i}) = F^i = 0, \qquad i = 1, \ldots, n,$$

which in turn implies that $G^i = 0, i = 1, \ldots, n$. By definition of the functions G^i, this means that $f_j(x_1, \ldots, x_n) = 0, j = 1, \ldots, n$. □

The advantage of this algorithm is that it limits the number of functions that must be handled at the same time, as well as the number of variables on which each function depends, thus reducing the computational effort required to obtain a solution. For an efficient implementation, the functions involved can be represented using BDD's [11, 12]. A simplified pseudo-code description of the algorithm is shown in Fig. 2. The function *Shannon_Decomp()* returns a pair of functions corresponding to the Shannon decomposition of a function with respect to the specified variable: if $f = (x \wedge f_x) \vee (\bar{x} \wedge f_{\bar{x}})$, then *Shannon_Decomp*$(x, f) = (f_x, f_{\bar{x}})$. Functions describing gates are understood to have been put in the form $f = 0$, as explained earlier in this section. Operations on pairs of functions are performed elementwise, e.g. $(f_1, f_2) \vee (g_1, g_2) = (f_1 \vee g_1, f_2 \vee g_2)$. Upon termination of the procedure, $(F_x, F_{\bar{x}})$ contains a pair of constants, which determine the range of values for the variable at the last node. The values of the variables at other nodes can be determined by saving the factors of the intermediate Shannon decompositions, as explained in Theorem 3.

To further limit the overall computational effort, the circuit is partitioned into strongly connected components before the DC solution is computed. This is done by treating the circuit as a directed graph, and performing two depth-first searches on the graph. The time to partition the circuit is bound by $O(M)$, where M is the number of edges in the graph. The DC solution is computed by using a graph traversal algorithm to propagate the primary inputs throughout the circuit: when a strongly connected component is met, the algorithm described above is used to compute the voltages at the nodes contained in that component.

4 Transient analysis

A standard event-driven selective-trace algorithm [13] is used for transient analysis. The delay model implemented in this algorithm is taken from [1]: it calculates both a delay time for inputs to propagate to the output and a slope value for the output. To take into account the gain of a gate, the output slope s_o is expressed as a gate-specific function $f_o()$ of the input slope s_i and the capacitance at the gate's output, C_o:

$$s_o = f_o(s_i, C_o).$$

Similarly, rise and fall delay values t_r and t_f are calculated as functions of s_i and C_o:

$$\begin{aligned} t_r &= f_r(s_i, C_o) \\ t_f &= f_f(s_i, C_o). \end{aligned}$$

The functions $f_o(), f_r()$ and $f_f()$ can be be given in the form of two-dimensional tables. Alternatively, experimental data shows that the gate delay as a function of the reciprocal of the input slope resembles closely a linear segment [1]. Therefore the gate delays can also be computed approximately as:

$$\begin{aligned} t_r &= a_0/s_i + a_1 \\ t_f &= b_0/s_i + b_1 \end{aligned}$$

where a_0, a_1, b_0 and b_1 are gate-specific coefficients describing the gate's delay characteristics. In the same way, instead of using a table the output slope can be computed by first multiplying the input slope (of the input which caused the transition) by a gain factor k specific to the gate, up to a maximum slew rate s_{max} determined by the gate:

$$s_r = \min(ks_i, s_{max}).$$

The value thus obtained, s_r, does not take the capacitive load of the gate into account. The actual value of the output slope is computed by scaling s_r according to the load [1]:

$$s_o = s_r \frac{C_g}{C_g + \sum_i C_i}.$$

In this formula, C_g is the output capacitance of the gate itself, while the C_i's represent the fanout capacitances.

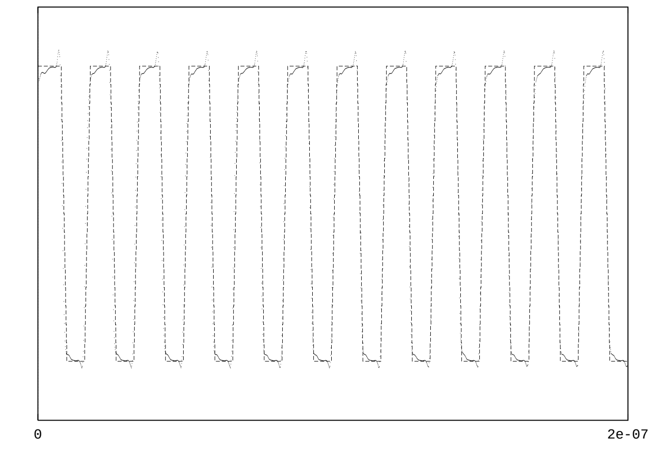

Figure 3. Output waveforms of the ring oscillator.

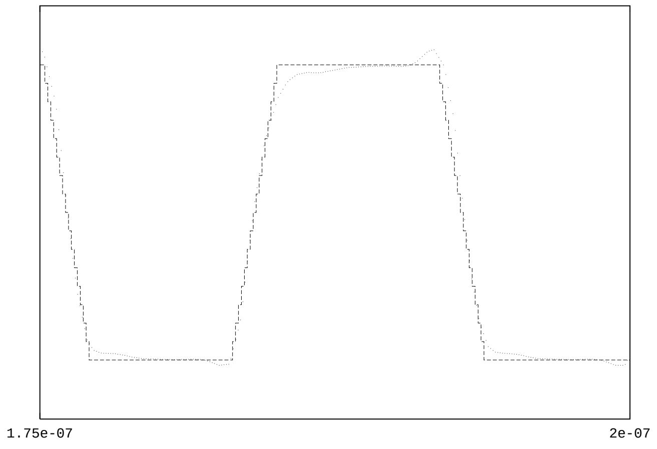

Figure 4. One period of the oscillator's output.

5 Numerical results

The algorithms described in the previous sections were implemented in a multi-level logic simulator, whose performance was tested on a number of circuits.

The first test circuit was a ring oscillator consisting of six inverters and a nand gate. A comparison between the waveforms obtained by our simulator and by SPICE are shown in Fig. 3: the dashed line shows the results using the simulation algorithm described here (17 logic states were used), and the dotted line shows the results of a SPICE simulation. For 200 nanoseconds of simulation time, the gate-level simulation ran at over 600 times faster than the SPICE simulation. A better comparison between the two waveforms can be made in Fig. 4, which shows a blow-up of one period of the oscillator's output.

The performance of the algorithm to compute the initial

TABLE II
DC solution times (in seconds) for ISCAS89 benchmarks

Ckt. Name	No. of Gates	Longest Fbk. Loop	CPU Time
s208.1	112	5	0.083
s298	133	18	0.27
s344	175	24	0.35
s349	176	25	0.40
s382	179	27	0.23
s386	165	59	0.53
s400	185	28	0.52
s420	234	5	0.13
s444	202	37	0.40
s510	217	161	34.0
s526n	215	19	0.38
s641	398	161	1.90
s713	412	93	1.20
s820	294	144	14.0
s832	292	147	24.0
s953	424	125	34.0
s1196	547	0	0.033
s1238	526	0	0.033
s1423	731	12	1.20
s1494	653	0	0.10
s1488	659	0	0.18
s5378	2958	252	27.0
s35932	17793	204	560
s38417	23815	0	33.0
s38584	16310	17	230

state of a digital circuit, described in Section 3, was tested on a number of circuits taken from the 1989 ISCAS benchmarks. A summary of the results is shown in Table II: the numbers in the "Longest Fbk. Loop" column indicate the number of gates in the longest feedback loop present in the circuit, while the CPU times refer to a Sparcstation 10/40 running SunOS 4.1.3. It can be seen that the required computational effort is always very reasonable, even in the case of circuits of respectable size containing deep feedback loops.

Finally, simulations were run on a number of circuits taken from the 1985 and 1989 ISCAS benchmarks. The benchmark circuits were simulated by applying one hundred random input vectors every 100 ns. with a central clock toggling every 20 ns. The results of the simulations are shown in Tables III and IV, which, for comparison pusposes, show also the times necessary to simulate the same circuits with VERILOG XL. All CPU times refer to a Sparcstation 10/40 running SunOS 4.1.3.

TABLE III
Simulation times (in seconds) for ISCAS85 benchmarks

Ckt. Name	No. of Gates	CPU Time	VERILOG XL		
			Comp.	Link	Sim.
c432	160	0.8	0.8	0.4	0.3
c499	202	1.0	0.7	0.2	0.4
c880	383	1.5	0.7	0.3	0.4
c1355	546	5.6	1.0	0.5	0.5
c1908	880	4.5	1.1	0.5	0.5
c2670	1193	5.0	1.2	0.9	0.9
c3540	1669	9.0	1.7	0.9	0.8
c5315	2307	15.0	2.1	1.3	1.7
c6288	2416	79.0	4.0	1.0	4.0
c7552	3512	24.0	3.0	1.4	2.4

TABLE IV
Simulation times (in seconds) for ISCAS89 benchmarks

Ckt. Name	No. of Gates	CPU Time	VERILOG XL		
			Comp.	Link	Sim.
s208.1	112	0.2	0.5	0.2	0.5
s298	133	0.3	0.5	0.3	0.4
s344	175	0.6	0.6	0.2	0.6
s349	176	0.5	0.5	0.3	0.6
s382	179	0.5	0.5	0.3	0.6
s386	165	0.4	0.5	0.2	0.5
s400	185	0.4	0.5	0.3	0.6
s420	234	0.4	0.5	0.3	0.6
s444	202	0.4	0.5	0.3	0.6
s510	217	0.2	0.5	0.3	0.5
s526n	215	0.4	0.5	0.3	0.5
s641	398	0.8	0.7	0.4	0.6
s713	412	1.1	0.6	0.4	0.7
s820	294	0.7	0.6	0.3	0.5
s832	292	0.7	0.7	0.3	0.5
s838.1	478	0.9	0.8	0.4	0.8
s953	424	0.9	0.7	0.4	0.8
s1196	547	1.6	0.7	0.3	0.7
s1238	526	1.7	0.7	0.4	0.6
s1423	731	1.6	0.8	0.6	1.0
s1494	653	1.4	0.8	0.4	0.5
s1488	659	1.3	0.7	0.4	0.5
s5378	2958	5.6	2.7	1.5	2.4
s35932	17793	93.0	31.0	10.2	23.2
s38417	23815	110.0	86.3	11.1	23.9
s38584	16310	69.0	40.1	10.0	20.0

6 Conclusion

We have described an algorithm that extends logic simulation to multiple-level signals. This is made possible by representing the value of a signal with a pair of elements of an *n*-dimensional Boolean algebra. This representation makes it possible to manipulate signals with an arbitrary number of intermediate levels using only Boolean algebra operations and in a way that mimics the operation of physical gates. There are several advantages to this approach: because operations in an *n*-dimensional Boolean algebra can be implemented as ordinary bitwise Boolean operations on a binary string, the speed of logic simulation is retained independently of the number of levels used (as long as the number of levels does not exceed the computer word length). Moreover, the problem of finding the initial state of a digital circuit ("DC solution") can be cast as the problem of finding the solution of a set of Boolean equations. An algorithm for that purpose has been described, and it has been shown how it can be modified to take advantage of the sparsity of the system of equations. On the other hand, the availability of an arbitrary number of intermediate signal levels between a logic zero and a logic one eliminates the need for an "unknown" state, and makes it easier the interfacing of digital and analog simulators. As a disadvantage, it should be mentioned that certain analog elements, such as adders and dividers, cannot be handled by our algorithm (while they can by the algorithm described in [1]).

A simulator implementing the techniques described here was developed, and its performance was tested on a number of benchmark circuits. The examples given show that it achieves a satisfactory trade-off between speed and accuracy. Future research plans include merging this simulator with the multi-level analog simulator described in [14].

References

[1] Genhong Ruan, Jiri Vlach, James A. Barby, and Ajoy Opal, “Analog Functional Simulator for Multilevel Systems”, *IEEE Transactions on Computer-Aided Design*, vol. CAD-10, no. 5, pp. 565–576, May 1991.

[2] Eduardo L. Acuna, James P. Dervenis, Andrew J. Pagones, Fred L. Yang, and Resve A. Saleh, “Simulation Techniques for Mixed Analog/Digital Circuits”, *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, pp. 353–362, April 1990.

[3] Young H. Kim, J. E. Kleckner, R. A. Saleh, and A. R. Newton, “Electrical-Logic Simulation”, in *Proceedings of the 1984 International Conference on Computer-Aided Design*. IEEE, November 1984, pp. 7–9.

[4] Sergiu Rudeanu, *Boolean Functions and Equations*, North-Holland Publishing Co., Amsterdam, 1974.

[5] Paul R. Halmos, *Lectures on Boolean Algebras*, Van Nostrand Co., Princeton, NJ, 1963.

[6] Melvin A. Breuer, “A Note on Three-Valued Logic Simulation”, *IEEE Transactions on Computers*, vol. C-21, no. 4, pp. 399–402, April 1972.

[7] Randal E. Bryant, “Algorithmic Aspects of Symbolic Switch Network Analysis”, *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 4, pp. 618–633, July 1987.

[8] Scott F. Woods and Giorgio Casinovi, “A Mixed Digital/Analog Gate-Level Simulation Algorithm”, Submitted for publication in *IEEE Transactions on Computer-Aided Design*.

[9] C.E. Shannon, “A Symbolic Analysis of Relay and Switching Circuits”, *Transactions of the American Institute of Electrical Engineers*, pp. 713–723, 1938.

[10] William J. McCalla, *Fundamentals of Computer-Aided Circuit Simulation*, Kluwer Academic Publishers, Norwell, MA, 1988.

[11] Randal E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation”, *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[12] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant, “Efficient Implementation of a BDD Package”, in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, Orlando, FL, June 1990, pp. 40–45.

[13] Melvin A. Breuer and Arthur D. Friedman, *Diagnosis & Reliable Design of Digital Systems*, Computer Science Press, Inc., Rockville, MD, 1976.

[14] Giorgio Casinovi and Jeen-Mo Yang, “Multi-Level Simulation of Large Analog Systems Containing Behavioral Models”, *IEEE Transactions on Computer-Aided Design*, vol. 13, no. 11, pp. 1391–1399, November 1994.