FUNCTIONAL ABSTRACTION OF LOGIC GATES FOR SWITCH-LEVEL SIMULATION †

D. T. Blaauw, D. G. Saab, P. Banerjee
Center for Reliable an High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.

J. A. Abraham
Computer Engineering Research Center
University of Texas at Austin
Austin, TX 78712, U.S.A.

ABSTRACT:

Switch-level simulation has become a common means for accurate modeling of MOS circuit behavior. In this paper, we propose a new method for detecting logic gate implementations and accurately modeling their switch-level behavior. The functional abstraction replaces logic gate implementation in the switch-level description with an accurate high-level model which incorporates all relevant switch-level phenomena. The switch-level accuracy of the simulation is, therefore, preserved. However, since the gate implementations are modeled at a higher, more abstract level, the simulation speed is greatly increased. The functional abstraction is automatic and completely transparent to the user. Detection of a gate is determined by expressing the logic function of a transistor network in the sum-of-product notation and is not limited to a specific design style. The proposed algorithms have been implemented and tested on several large circuits, including a complete microprocessor. For this processor, 85% of all transistors were substituted with high-level models. A significant decrease in simulation time and storage requirement occurred for these circuits when gate abstraction was performed.

1. Introduction

Currently, simulation of large circuits is often performed at the switch-level, as introduced by Bryant [1]. Simulation at this level has the advantage that several detailed circuit phenomena, such as charge sharing, bidirectional signal flow, signal strength, and resolution of conflicting signals, are accurately modeled. The disadvantage is that modeling these phenomena requires complicated and time consuming path tracing algorithms. However, for transistor structures implementing static logic gates, charge sharing, bidirectional flow, and conflicting signal phenomena do not occur. A logic gate implementation can be modeled at the switch-level using a simple Boolean function, describing its logic operation, and strength information, describing the equivalent signal strength of the gate. Simulation of these transistor structures is, therefore, performed with much simpler and faster algorithms than traditional path tracing algorithms without compromising the switch-level accuracy. Since a large portion of the circuit consists of logic gates, the simulation speed and storage requirement of the circuit will benefit significantly from abstracting logic gate structures.

This paper presents a new method to detect logic gate implementations in a switch-level transistor description and to model them at a higher level. The gate abstraction substitutes gate implementations with so-called *gate descriptors*. The gate descriptor contains both the logic function and strength information of the gate implementation. During simulation, gate descriptors are evaluated by fast C-functions, while other transistors structures are evaluated with standard switch-level simulation techniques. The gate abstraction operates directly on the switch-level description and is completely transparent to the user. Since the gate abstraction is performed only once for each design, it adds little overhead to the simulation process. Currently, extraction is performed for static CMOS and NMOS gates. However, the presented methods can be extended to include dynamic gates.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 discusses functional abstraction for logic gates. Section 4 presents the algorithms of the gate abstraction. Section 5 discusses the performance results for several large circuits and Section 6 offers concluding remarks.

2. Related Work

Several methods for detecting logic gates have been developed in the area of design verification and generation of gate-level descriptions from transistor descriptions [2]. These methods translate switch-level descriptions into gate-level or logic-level descriptions. During this process, transistor implementations of logic gates are detected and replaced with gates. These methods have the effect of raising the level of the description from the switch-level to the gate-level or higher. However, the switch-level accuracy of the description is lost. Therefore, these methods do not directly apply to switch-level simulation.

In the area of switch-level simulation, work relating to gate abstraction has been performed by [3, 4, 5]. These simulation methods analyze dc-connected components and translate them into Boolean equations. They maintain the switch-level accuracy and express the circuit behavior in terms of logic equations. They differ from gate abstraction in that they operate on all possible transistor constructs and logic gates are not detected as such. A logic gate differs from general transistor structures in that it has dual pull-up and pull-down functions. The internal operation of a gate need not be modeled and the output of the gate is directly derived from the logic state of the input signals. The mentioned switch-level simulation methods, however, cannot assume the pull-up and pull-down functions to be duals. Both the pull-up and the pull-down path of the gate are evaluated and nodes internal to the gate structure are not always eliminated. The potential advantage from specifically detecting logic gates and modeling them at a higher level is, therefore, foregone in these methods.

† This research was supported in part by Semiconductor Research Corporation Contract 90-DP-142, and in part by Motorola, Inc. Austin, TX.

3. Functional Abstraction of Logic Gates

In this paper, we propose a new method for detecting a logic gate implementation and accurately modeling its switch-level behavior. In a switch-level circuit description each circuit element (transistor or circuit node) can display the full range of switch-level phenomena. For instance, all circuit nodes can potentially store a charge, and all transistors are potentially bidirectional. As presented in [6] , we call the possible operation of an element or group of elements its *functional domain*. When a circuit element is considered as part of the overall circuit its actual operation is only a subset of this functional domain which we call its *functional application*. For accurate simulation, only the functional application, rather then the full functional domain of an element must be modeled. Functional abstraction is the process of determining the functional application of a circuit element from its functional domain.

The functional domain of a static gate implementation includes bidirectional signal propagation through each of its transistors, charge sharing effects between its nodes, and charge storage at each node. Most of these phenomena, however, do not occur in the actual operation of the gate, and need not be modeled. A static CMOS gate is characterized by pull-up and pull-down functions that are duals of each others. This means that, for all possible gate inputs, there is always a conducting path from the gate output to either power or ground. Furthermore, for all possible true input values (logic 0 or logic 1), there is either a path to power or to ground, but not simultaneously to both. Because of these characteristics, the functional application of a logic gate is greatly simplified. Below, the switch-level phenomena that occur in the functional domain of a static logic gate, but not in its functional application, are treated.

1 - Charge Sharing and Charge Storage

Charge sharing occurs when two or more nodes, isolated from power and ground, become connected through a conducting path. The resulting signal at these nodes is a function of their logic states and capacitance sizes. Since in static CMOS gates there is always a conducting path from power or ground to the gate output, stored charge on the gate output is overruled. Therefore, neither charge sharing nor charge storage can affect the state or strength of the gate output.

2 - Bidirectional Signal Flow

In a static logic gate, only the new value of the gate output is determined. This gate output is always connected to power or ground through one or more paths. Along these paths, signals propagate only in the direction toward the gate output. Bidirectional signal flow is, therefore, eliminated.

3 - Conflicting Signal Resolution

Signal resolution is necessary when two or more signals with different logic states attempt to drive the same node. When true-values (logic 0 or logic 1) are applied to a logic gate, simultaneous paths to *vdd* and *gnd* do not occur. In this case, signal resolution is, therefore, not needed. In the case that unknowns are applied to the gate, it is possible that simultaneous paths to *vdd* and *gnd* occur. However, if such simultaneous paths occur, each path will contain at least one transistor with an unknown signal controlling its gate. Since both paths contain a transistor in an unknown state, the output always evaluates to unknown. In this case, resolution of conflicting signals is, therefore, also not needed.

Modeling a logic gate is further simplified by the fact that its inputs all drive transistor gates. Only the logic state of the inputs is needed and their strength can be ignored. The logic state of the gate output is thus simply defined as a Boolean function of the logic states of the gate inputs. The only other feature that needs to be modeled for a logic gate is the strength of the pull-up or pull-down path of the gate output. Charge sharing, bidirectional signal flow, and conflicting signal resolution are eliminated from the functional application and need not be modeled. Since the functional application of a logic gate is greatly simplified, its evaluation can be performed very efficiently.

4. Abstraction Algorithms

Gate abstraction for switch-level simulation consists of three parts. The first part, called the logic analysis, obtains the logic pull-up and pull-down functions of a dc-connected component and determines whether it represents a valid logic gate. The second part, called the strength analysis, analyzes the strength of the pull-up and pull-down paths and incorporates this strength information in the gate descriptor substituting the transistor implementation. Finally, the third part generates an evaluation routine for the gate descriptor. Since most gate descriptors share evaluation routines, the total number of generated routines is small. Below, each of the parts of the abstraction process is described in more detail.

4.1. Logic Analysis

The logic analysis of the abstraction process determines the logic pull-up and pull-down function of a dc-connected component. The two functions are then compared to determine if they are functional duals of each other. The algorithm performs this task in three phases as explained below.

Identifying Potential Gate Outputs

The first phase identifies nodes that are potentially gate outputs. A gate output is always connected to a P-type transistor section and a N-type transistor section. Therefore, a node is classified as a *potential gate output* (PGO) if it is connected to the channel of at least one N-type transistor and the channel of at least one P-type transistor. For each PGO node, the dc-connected transistors are then examined to determine if the structure represents a logic gate.

Constructing the Logic Function Tree

In the second phase, transistors that are dc-connected to a PGO are traversed. During this traversal, a directed tree, called the *logic function tree* (LFT), is generated that represents the logic function of the transistor network. A logic function tree has as nodes either OR or AND functions and has circuit signals as leaves, either in complemented or non-complemented form. Each node in the tree has an arbitrary number of children. One logic function tree is generated for the N-type transistor section of the network, called the *N-LFT* and one for the P-type transistor section, called the *P-LFT*. In Figure 1, the P-LFT and N-LFT of an example gate are shown. Any possible logic function can be expressed using the logic function tree representation.

The conversion to the logic tree format serves three purposes: first, the logic pull-up and logic pull-down functions are expressed independently. The algorithm determines the N-section and P-section and traces them separately. Secondly, the connectivity of the network is checked during the tracing. When an illegal construct is encountered, the algorithm either aborts the

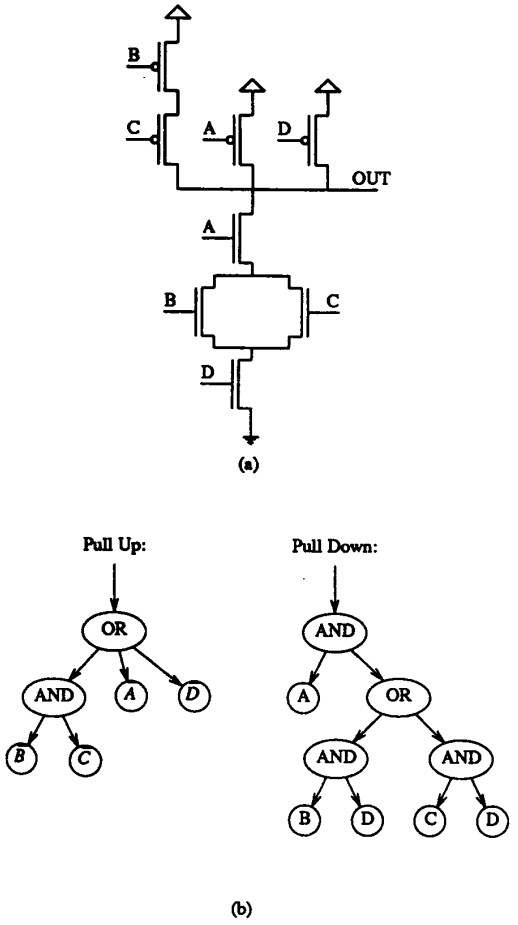

Figure 1. A logic gate with its pull-up and pull-down LFT.

gate detection process or removes the function of the illegal construct from the tree and continues the gate detection. Thirdly, the logic function of cyclic constructs is expressed in a non-cyclic representation. This simplifies further conversions and comparisons of the pull-up and pull-down functions.

The conversation algorithm performs a depth-first trace of all possible paths that start from the PGO node. For each traversed transistor, an *AND* node with a connected *LEAF* node is added to the existing tree. For each circuit node with multiple fan-out, an *OR* node is added to the tree.

Conversion to Sum-of-Product Notation

The LFT generated in phase 2 completely represents the logic pull-up and pull-down function of a transistor network. For a valid gate, the function represented by the P-tree is the dual of that represented by the N-tree. However, the actual form of the tree depends on the used design style, the used circuit extraction program, and the exact layout of the circuit. While the function represented by the N-tree and the P-tree might be duals, the actual trees might not be duals. An example of this situation is shown in Figure 1. Although the structure shown in Figure 1(a) represents a logic gate, Figure 1(b) shows that the generated logic function trees are quite different. By examining the P-LFT and N-LFT directly, it is difficult to determined that they represent dual functions.

In order to detect logic gates independent of the specific design style and layout the third phase of the logic analysis converts the LFT representation into the sum-of-product (SOP) notation. This conversion has the effect of flattening the logic function tree, allowing it to be easily simplified and tested for equivalence. The conversion can generate either the function represented by the tree or its dual. For the P-LFT the SOP notation of the dual of the function is generated, while for the N-LFT the SOP notation of function itself is generated. The SOP notation is generated using a single, depth first traversal of a logic function tree.

The generated SOP notations often contain redundancies. Before the N-SOP and P-SOP notations are tested for equivalence, redundant product terms and variables are removed. In general, this involves the exponential tautology problem presented in ESPRESSO [7]. However, for SOP notations derived during the gate detection process, redundancies in the notation are completely removed in polynomial time. The N-SOP corresponds to the N-section of the gate and involves only N-type transistors. All variables in the N-SOP list are, therefore, non-complemented. Similarly, all variable in the P-SOP are complemented. An SOP list cannot contain both complemented and non-complemented variables, which greatly simplifies the complete removal of the redundancies. Only the two redundancies shown below can occur in the generated SOP notions.

Duplication of variables: $\{a\ b\ c\ a\} \rightarrow \{a\ b\ c\}$

Equivalence of terms: $\{\bar{a}\ \bar{b}\ \bar{c}\ \bar{d} + \bar{a}\ \bar{b}\ \bar{d}\} \rightarrow \{\bar{a}\ \bar{b}\ \bar{d}\}$

Both of these redundancies are easily removed with sequential traversals of the SOP list. The time requirement for this process is $O(n^2)$, where n is the length of the SOP list. After all redundancy in the SOP notations are removed, their equivalence is tested. If the two lists represent a valid logic gate, their SOP lists must be identical except for the ordering of products and variables. This equivalence is established with an $O(n^2)$ search method.

4.2. Strength Analysis

Switch-level simulation uses a simplified conductance model to determine the combined strength of a transistor network [1]. For a chain of transistors connected in series, the combined strength of the chain is the minimum of the transistor strengths in the chain. Conversely, the combined strength of parallel transistor connections is the maximum of the individual transistor strengths. A transistor network implementing a gate can be seen as a set of parallel branches connecting the gate output to *vdd* and *gnd*. Each branch consists of one or more transistors connected in series. The strength of each branch is the minimum strength of the transistors in the branch. The strength of the entire gate is the maximum strength of all conducting branches. We consider the strength of the P-section (to *vdd*) and N-section (to *gnd*) independently. If all branches in a section are of equal strength, it is

called *strength consistent*. Furthermore, if both the N-section and P-section of a gate are strength consistent, the gate is called strength consistent. In this case, a unique pull-up and pull-down strength are identified for the gate, and the gate strength is a function only of the logic state of the gate output. A strength consistent gate is, therefore, fully described by its logic function, its pull-up strength, pull-down strength, and its number of inputs.

If either section of the gate is not strength consistent, the gate output strength depends on which branches in a section are conducting and the gate is called *strength inconsistent*. An example of such a gate is shown in Figure 2. Since the size of transistor $T1$ is greater than that of $T2$, the pull-down strength of the gate is greater when $T1$ is conducting than when only $T2$ is conducting. For strength inconsistent gates, the output strength is not only a function of the logic state of the output, but also of the gate inputs.

The SOP notation derived during the gate abstraction is particularly convenient for strength analysis of gate structures since each product in the SOP notation corresponds to a branch of the gate. During the construction of the logic function tree, each leaf added to the tree is assigned the strength of its associated transistor. When the SOP notation is produced, the strength of a product is then the minimum strength of the leaves in the product. If the strength of all products in the SOP notation are identical for the pull-up function and the pull-down function, the gate is classified as strength consistent. Conversely, if either the pull-up function or the pull-down function contain products with differing strength, the gate is classified as strength inconsistent. It should noted that for a strength consistent gate, the strength of the pull-up function can differ from that for the pull-down function. This is frequently the case, since designers size the pull-up and pull-down transistors to achieve specific rise the fall times.

4.3. Gate Evaluation Methods

After the logic and strength analysis are completed, the gate abstraction program generates a gate descriptor and a gate evaluation routine. The gate descriptor provides the interface between the used simulator and the gate evaluation routine. It is an entry in the circuit description that replaces the original transistor network and points to the subroutine that evaluates the gate. Most gate descriptors share the same evaluation routine.

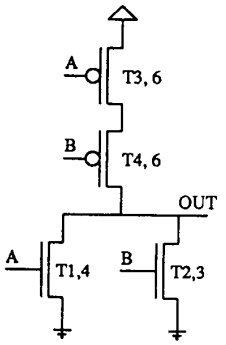

Figure 2. A logic gate with different pull-down strengths. Transistors are labeled as T_n, S, where S is the size of T_n.

The gate descriptor contains a list of parameters, necessary for the evaluation of the gate. When the simulator encounters a gate descriptor, it calls the associated evaluation routine and passes to this routine the list of inputs and outputs, as well as the list of parameters included in the gate descriptor. The method of evaluation depends on the logic function and the strength classification of the gate. Separate approaches are taken for strength consistent and strength inconsistent gates.

Strength Consistent Gates

For strength consistent gates, the gate is fully described by the type of logic function it performs, the number of inputs, and the strength of the pull-up and pull-down functions (respectively called the *up strength* and *down strength*). A gate extracted from a transistor description is either an 'Inverter', a 'NAND', a 'NOR', or an 'AND_OR_Inverter' (AOI) gate. For a AOI gate, not only the total number of inputs is specified in the gate descriptor, but also the grouping of the inputs for each AND in the AOI gate.

Evaluation for consistent NOR, NAND, Inverter, and AOI gates is performed with a simple table lookup or poling routine. For each type of logic gate a separate evaluation routine is used. The evaluation routine for NAND gates is shown in Figure 3. Variables *in* and *out* point, respectively, to a list of input nodes and the output node. The variable *par* points to the vector of parameters contained in the gate descriptor. For clarity, a pseudo-code version of the NAND evaluation routine is shown. Since a gate descriptor contains the number of inputs and the up and down strengths, only one evaluation routine is needed for each type of gate. For strength consistent gates, a maximum of four C-functions is, therefore, needed. The evaluation routine examines the gate inputs sequentially and requires $O(n)$ evaluation time, where n is the number of inputs. It should be noted, however, that each iteration involves only one or two comparison statements. Evaluation for strength consistent gates is, therefore, quite efficient.

Strength Inconsistent Gates

For strength inconsistent gates, the gate output strength is not only a function of the output but also of the gate inputs. Therefore, specifying a single up and down strength is not sufficient and the evaluation method used for consistent gates cannot be used. Instead, the exact output strength is specified for

```
NAND ( in, out, par)
{
  outState = 0;
  for (i = 0; outState != 1 && i < par[Num_Inputs]; i ++) {
    if (state (*in) == 0)
       outState = 1;
    else if (state (*in) == X)
       outState = X;
    in ++;
  }
  setState (out, outState);
  switch(outState) {
  case 0: setStrength (out, par[Down_Str]);
       break;
  case 1: setStrength (out, par[Up_Str]);
       break;
  case X: setStrength (out, max(par[Down_Str], par[Up_Str]));
       break;
  }
}
```

Figure 3. Evaluation code for a NAND gate implementation.

each possible set of input states.

Two evaluation methods were used to model these gates. First, for gates with a small number of inputs, evaluation uses an exhaustive truth table. In this truth table, the state and strength of the gate output is specified for each possible gate input state. The time required for forming the index from the gate inputs is linear with the number of inputs. Each gate input can take on three possible states (logic 0, logic 1, and unknown) and is encoded using two bits. The size of the table, therefore, grows as $O(4^n)$, where n is the number of inputs. Only gates with a maximum of 4 inputs are modeled in this way. For gates with more than 4 inputs, evaluation is performed using the Coded Personality Matrix (CPM) method described in [8]. This method uses a binary encoding of the products in the pull-up and the pull-down functions. The CPM method is modified so that each product is associated with both a logic state and strength. A gate is now evaluated by encoding the logic state of the inputs and traversing the product lists to determine if there is a match. Each time the input states match a product, the gate output is updated with the state and strength information of that product. Since the entire product list is traversed, the evaluation time is linear with the number of products in the product list. The size of this list usually far exceeds the number of gate inputs.

The evaluation method for consistent gates is faster and requires less storage space than those for inconsistent gates. However, inconsistent gates occur infrequent in most circuit designs. For the tested circuits, less then 1.0 % of all detected gates were strength inconsistent. The overall performance of the simulation was, therefore, not seriously affected by the more time intensive modeling of these gates.

5. Performance Results

The proposed algorithms were implemented in the C-language and executed on Sun-4 work stations. The program was tested on several large circuit descriptions, including the circuit description of a large microprocessor. All tested circuits were hierarchically defined and were simulated with the CHAMP [9] simulator. To ensure the correctness of the gate abstraction process, the microprocessor was simulated both with and without gate abstraction for over 20,000 clock cycles. The produced signals of the processor outputs and internal busses were then compared. Both the state and strength of all signals were identical for the two simulations.

Table 1 shows the performance results of the gate abstraction. The column labeled *percentage of transistors replaced* refers to the percentage of transistors substituted with gate descriptors in the flat circuit description. As can be seen, this percentage varies for different types of circuits. The column labeled *size down* refers to the reduction in the hierarchical description size. The simulation speed-up observed when performing gate abstraction varied from 1.81 to 3.16. The simulation speed of the microprocessor increased by a factor of 1.81.

circuit	number of trans.	% trans. replaced	size down	simulation speed-up
decoder	6604	100%	7.47	2.06
alu	1468	86.8%	1.77	1.90
random logic	385	88.3%	1.51	1.94
register file	1542	93.8%	1.29	3.16
microproc	38,479	85.1%	1.57	1.81

Table 1. Performance results of the gate abstraction algorithm.

The execution time of the proposed gate abstractor is roughly equal to the time required for parsing the circuit description. Since the simulation time is several orders of magnitude greater than this and the gate abstraction is only performed once per simulation, the added computational cost is insignificant. Rather, the saving in simulation time greatly outweighs the added computational cost.

6. Conclusions

In conclusion, the proposed algorithm presents an efficient and accurate means of increasing the simulation speed of switch-level simulation. It was shown that for logic gates, most switch-level phenomena cannot occur and modeling these structures is greatly simplified. An algorithm was presented to detect logic gates irrespective of their design style or circuit layout. Logic gates were then classified as strength consistent or strength inconsistent and, for each class, efficient evaluation routines were presented. The algorithms were implemented and tested on several large circuits, including a large microprocessor. For this microprocessor, the gate abstraction algorithm replaced 85% of all transistors with gate descriptors which increased the simulation speed by a factor of 1.81. Since logic gates are conceptually and computationally much simpler than their transistor implementations, significant benefit can be had from the gate abstraction process.

REFERENCES

[1]. R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers* C-33, No.2 pp. 160-177 (Feb. 1984).

[2]. M. Boehner, "LOGEX - An Automatic Logic Extractor from Transistor to Gate Level for CMOS technology," *Proc. IEEE International Design Automation Conference*, pp. 517-522 (1988).

[3]. I.N. Hajj and D.G. Saab, "Symbolic Logic Simulation of MOS Circuits," *Proc. International Symposium on Circuits and Systems*, pp. 246-249 (1983).

[4]. G. Ditlow, W. Donath, and A. Ruehli, "Logic equations for MOSFET circuits," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 752-755 (1983).

[5]. R.E. Bryant, "Boolean Analysis of MOS Circuits," *IEEE Transactions on CAD* CAD-6, No. 4 pp. 634-649 (July 1987).

[6]. D. T. Blaauw, R. B. Mueller-Thuns, D. G. Saab, P. Banerjee, and J. A. Abraham, "SNEL: A Switch-Level Simulator Using Multiple Levels of Functional Abstraction," *IEEE International Conference on Computer Aided Design*, pp. 66-69 (1990).

[7]. R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston (1984).

[8]. D. T. Blaauw, R. B. Mueller-Thuns, D. G. Saab, and J. A. Abraham, "Automatic Generation of Behavioral Models from Switch-Level Descriptions," *Proc. IEEE International Design Automation Conference* , pp. 179-184 (1989).

[9]. D. G. Saab, R. B. Mueller-Thuns, D. T. Blaauw, J. A. Abraham, and J. T. Rahmeh, "CHAMP: Concurrent Hierarchical And Multilevel Program for Simulation of VLSI Circuits," *Proc. IEEE International. Conference on Computer-Aided Design*, (1988).