AUTOMATIC GENERATION OF BEHAVIORAL MODELS FROM SWITCH-LEVEL DESCRIPTIONS

David T. Blaauw, Daniel G. Saab*, Robert B. Mueller-Thuns*, Jacob A. Abraham** and Joseph T. Rahmeh***

* Computer Systems Group, University of Illinois, Urbana, IL 61801
** Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712

ABSTRACT: This paper discusses the automatic generation of high-level software models from switch-level circuit descriptions. The proposed algorithms operate directly on the hierarchical description, and incorporate information about the design such as the structure, regularity, functionality, and control signals in the generation process. New algorithms are proposed and have been implemented for combinational modules and bus structures. A significant speedup has been obtained for these modules of a commercially available chip.

1. Introduction

Advances in Very Large Scale Integration (VLSI) technology have made possible the implementation of large and increasingly complex systems on a single integrated circuit chip. This has led to the need for accurate and efficient simulation tools capable of handling complex designs.

Simulation of MOS circuits is currently performed at the switch-level. Recently, hierarchical simulators has been proposed to accelerate the simulation of circuits at this level [1]. Using hierarchy, the structure of commonly used subcircuits needs to be stored only once and can be referenced when needed. Furthermore, modules from the hierarchy can be substituted by high-level software models. When a large percentage of the circuit modules is replaced by accurate, efficient and compact software models, the simulation speed is greatly increased.

Because the module size and complexity has increased with the advance of VLSI technology, generation of such software models has become a difficult task. Manual coding of these models is both time consuming and error-prone. Therefore, they must be generated automatically.

Several algorithms to abstract behavioral models from a circuit description have been proposed. Path finding algorithms are presented in [2, 3, 4, 5, 6]. For each channel connected component they construct logic equations describing the steady state of each node in a partition. The equations associated with a component can then be compiled into executable code. A limiting factor of path finding algorithms is that the model size can grow exponentially with the circuit size. In [7] work is presented on translating a flat, transistor circuit description into a register transfer level model. However, since a register transfer level description does not support the accuracy required for switch-level simulation, most of the presented algorithms cannot be used for switch-level model generation.

An additional drawback of these algorithms is that they operate on a flat transistor description. Abstracting smart models from these descriptions has not yet been accomplished. Especially for large and complex circuits, the generated model becomes large and inefficient compared to hand generated models. To improve model generation, advantage must be taken of design information, such as the structure, regularity, functionality, and control signals of the circuit.

The automatic model generator proposed in this paper effectively obtains and incorporates this information in the generation process. The key to our approach is that this design information is preserved in a hierarchical circuit description. By operating directly on the hierarchical description, the generator can identify the design information for many of the circuit modules. The obtained information is then utilized in the generation process, which produces more efficient software models. Information obtained from a hierarchical description of a design improves the software model generation in the following ways:

(1) The structure of a module can be exploited. For instance, different approaches can be taken for modules constructed of gates versus those constructed of transistors.

(2) The regularity of structure is more visible and easily obtained, and can be used to increase efficiency of the software model.

(3) The hierarchical description preserves the functional partition of the design which can be used to guide the generation of software models. Uncommon input patterns can be treated in a less efficient manner so that more efficient treatment is given to frequently occurring input patterns. Furthermore, it is possible to make reasonable simplifications in the circuit model based on the behavioral information.

(4) Control signals and their function can be identified more easily if the function of circuit is known.

Using a simulation system that combines event-driven hierarchical simulation and high-level software models allows the generator to judiciously choose which modules are translated into a high-level model and which are simulated at the lower primitive level in an event-driven manner. Some circuit modules will have no recognizable structure or behavior and do not lend themselves to smart model generation. Such modules are more efficiently simulated with an event-driven simulator. The conjunction of event-driven simulation and high-level model generation allows, as appropriate, either of these methods to be used for a module.

Not only logic but also fault simulation can benefit from high-level software models. Here the circuit description of the module and the software model are used side by side. If the

module contains no faults, the software description is used for evaluation; however, if a fault is present in the module, the circuit description of the module must be used. In practice, only a small subset of faults is simulated at any one time. Since these can easily be located in one or a few modules, most modules can still be evaluated with software description. In this way, simulation time can be decreased significantly for fault simulation using high speed software models.

2. Issues In Generation of High-Level Models

Since the hierarchical description emerges during the design process, the modules in the hierarchy contain logically distinct substructures. Each substructure has unique structure, regularity, functional behavior and control signals. In practice, only a small number of distinct types of substructures occur and a library of these types can be constructed. Once a module is classified as a particular type, its characteristics can be incorporated into the software model.

Modules are first classified according to their functional behavior. Modules that have no internal state, such as PLAs and combinational gate circuits, are called **combinational modules.** Model generation for combinational modules involves analyzing the circuit to obtain the Boolean function equivalent of the circuit. Modules that have an internal state are called **sequential modules.** These substructures are diverse enough to justify individual model generation algorithms for each type of substructure.

We have focused on model generation for two types of modules commonly used in complex systems: combinational modules and sequential modules containing busses. In the case of busses, structural information is used to produce efficient code. Therefore, path finding algorithms were not used indiscriminately for the entire substructure and, rather than partitioning the circuit into channel connected components, the circuit was partitioned according to functionality. Figure 1 shows how a simple circuit is partitioned into a two bit multiplexer and two input circuits. The following paragraphs present some of the general principles used to generate fast and efficient code for sequential modules:

(1) **Using Simple Code:** Our object is to keep the generated code as simple as possible. No function calls or pointer manipulations were incorporated in the software models. The most complex structures in currently generated models are **if** statements and **for** loops.

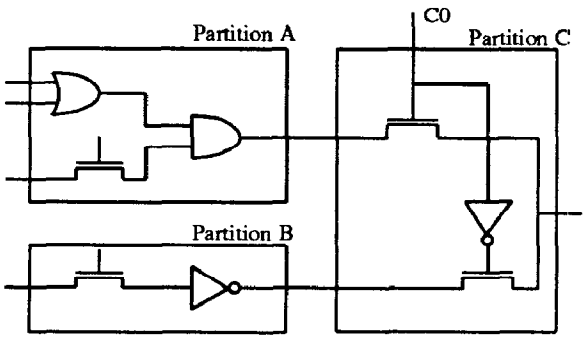

Figure 1. Circuit partitioned according to functionality.

(2) **Improve Code Ordering Using Control Signals:** Control signals determine the flow of signals through the circuit. The value of a control signal determines which partitions of the circuit will effect the output. If, in Figure 1, control signal **C0** is logic 1, partition A must be evaluated, while if **C0** is logic 0, partition B must be evaluated. If **C0** is unknown, both partition A and partition B must be evaluated. To evaluate the circuit efficiently, control signal **C0** is tested; then code corresponding to partition A, partition B, or both is executed followed by code corresponding to partition C. In this way unnecessary execution of code is avoided using knowledge of the control signal.

(3) **Gathering of Control Signals:** Control signals are used repeatedly in a module. Those parts of the circuit that use the same control signal may be gathered into one partition. Partitioning in this fashion is called **control gathering** [7]. In this way, the control signal is inspected only once in order to determine how to proceed with the evaluation. For instance, in the ALU circuit of Figure 2, control signal **C0** determines whether the output of the ALU is connected to the output of gate **G1** or if it retains its previous value. Assume that multiple bit slices of this structure are used. Control gathering may now be performed using control signal **C0**, so that control signal **C0** is inspected only once. If the signal has a state of logic 0, all outputs retain their current value. However, if the signal has a state of logic one, the gathered logic is evaluated.

(4) **Special Control Signals:** Some control signals perform special functions, such as precharging, predischarging, reset, and clock signals. Knowledge of the function of these control signals may be used in the model generation process. Reset signals, for instance, often override other control signals and bring the circuit to a predetermined state. This state can be assigned immediately if the reset signal is active.

(5) **Directional Analysis:** Although signals can flow in both directions between source and drain in a conducting transistor, in practice one finds that the majority of transistors in a circuit are unidirectional. Circuit analysis is performed to classify each transistor as unidirectional or bidirectional. The evaluation of unidirectional transistors is much simpler than that of bidirectional transistors. Using directional knowledge in the model generation process can thus increase the efficiency of the software model.

3. Implementation and Scope of the Model Generator

The model generator was implemented using the C-language. Currently, models are generated for combinational

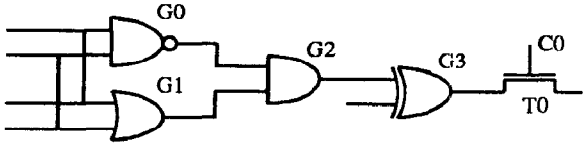

Figure 2. Control gathering with an ALU circuit.

modules, including PLAs and combinational gate modules, as explained in Section 4, and for sequential modules containing bus structures, as explained in Section 5.

The model generator consists of three phases:

(1) **Parsing and Preliminary Analysis:** The circuit is checked and stored into an internal data structure. Any analysis needed by the generation algorithms, such as directional analysis, is performed.

(2) **Module Classification:** The modules are classified into types of substructures such as bus type, combinational gate type, ALU type, etc., based on the modules structure and components. This classification is not yet performed automatically. The user has to inspect the circuit description and inform the generator of its type for each module.

(3) **Code Generation:** The model generation algorithms are applied to the modules. Each type of module has a separate algorithm associated with it.

4. Model Generation for Combinational Modules

In this section a new model generation algorithm for combinational modules is introduced. Traditionally, combinatorial circuits have been modeled using an exhaustive truth table. For each combination of input values, this table contains the corresponding output values. In three valued logic (0, 1, X), the total size of the table is $3^i o$ bytes, where i is the number of inputs and o is the number of outputs.

When used appropriately for small circuits, this table method provides fast evaluation. Since the table size grows exponentially with the number of inputs, many combinational gate modules in a circuit will be too large. Since we stress the need to generate models for large modules, different algorithms were found to handle these modules. One method, presented in [8], stores the personality matrix of the circuit in the software model. The size of the personality matrix as well as the number of operations required for evaluation is $O((i+o)p)$, where i is the number of inputs, o is the number of outputs, and p is the number of product lines.

In order to produce more compact and efficient code, a new algorithm, called the coded personality matrix method (CPM), was developed. This algorithm uses a binary coding to store the personality matrix and Boolean comparisons in its evaluation algorithm. The encoding drastically reduces the size of the stored table and the Boolean comparisons provide for fast evaluation. The software model is automatically generated from a switch-level description. Figure 3 shows the code for the software model corresponding to a circuit with the personality matrix shown in Figure 4. The function of the AND plane is encoded using two integer lists: the **maskList** and the **moldList**. The function of the OR plane is encoded and stored in the integer matrix **outList**.

The AND plane of the personality matrix is fully described by a set of products of the input variables called **product terms**. Each product line has a product term associated with it that describes for which input pattern it is activated. We use a two bit encoding to record the state of each input variable in the product. One bit, called the **mask bit**, indicates whether the variable is present in the product term. The other bit, called the **mold bit**, indicates whether the variable appears in complemented or uncomplemented form. A product term is thus represented by a string of mold and mask bits. Each string is stored in one or more machine words. The lists of mold and

```
1 procedure CPM(inputP, outputP)
2         char                *inputP, *outputP;
3 {
4         int                 i, j, index;
5         static int  moldList [4] = {0x00, 0x08, 0x02, 0x06 };
6         static int  maskList [4] = {0x0C, 0x0E, 0x0A, 0x06 };
7         static int  outList  [4][3] = {
8                     { 0, -1, -1 }, { 1, -1, -1 },
9                     { 2, -1, -1 }, { 0, 1, -1 } };
10        int                 iMask;
11        int                 iMold;
12
13  /* create iMold and iMask */
14  iMask = iMold = 0;
15  for (i = 0; i < number_of_inputs; i++) {
16    iMask = iMask << 1;
17    iMask = iMask | ((inputP[i] & unknown_mask)>>1);
18    iMold = iMold << 1;
19    iMold = iMold | (inputP[i] & logic_state_mask);
20    }
21  iMask = ~ iMask;
22
23  for (i = 0; i < number_of_outputs; i++)
24    outputP[i] = driving_zero; /* preset the outputs to D,0 */
25  for (i = 0; i < number_of_product_lines; i ++)
26    if (!((iMold ^ moldList[i]) & iMask & maskList[i]))
27        /* hit has been found */
28        if ((~iMask) & maskList[i]) /* prop of X  */
29          for (j = 0; (index = outList[i][j]) != -1 ; j ++)
30            if (outputP[index] & logic_zero_mask)
                  onode[index] = driving_unknown;
31        else  /* no prop of X */
32          for (j = 0; (index = outList[i][j])  != -1; j ++)
33            outputP[index] = driving_one;
34 }
```

Figure 3. Model code of coded personality matrix method.

product	input variables				output variables		
line	A	B	C	D	O0	O1	O2
P0	0	0	n	n	1	n	n
P1	1	0	0	n	n	1	n
P2	1	n	0	n	n	n	1
P3	n	0	n	0	1	1	n

n: not connected

Figure 4. Personality matrix of a small PLA.

mask words are stored in array **moldList** and **maskList** respectively (see Figure 3). The AND plane is encoded with $S_{AND}=2wp$ words, where p is the number of product lines, and w is the number of words needed to store the mold and mask bit strings.

The OR plane of a PLA is encoded and stored in a two dimensional array of integers (array **outList** in Figure 3). The integers in a row are the indices of the outputs that are connected to a product line. The size of the output matrix is $S_{OR}=(m+1)p$, where m is the maximum number of outputs to which a product line is connected. The total storage size for the

CPM method is $S_{TOT}=(m+1+2w)p$.

The evaluation algorithm consists of the following steps:

(1) The input pattern is encoded.

(2) The outputs are preset to driving zero.

(3) Each product term is compared to the input to determine whether they match.

(4) Each time the input matches a product term (called a **hit**), further comparisons are made to see if any unknowns in the input propagate to the output and the outputs are updated accordingly.

The input pattern to a circuit is described by a product and is encoded using two words, **iMask** and **iMold**. A hit on a product line is now determined with only 4 Boolean operations (see Figure 3, line 33). When a hit is discovered, a second comparison is performed to determine whether there is an unknown in the input pattern that propagates to the product line. A product line becomes unknown if the input contains an unknown and the product term at that position does not contain a 'don't care' condition. Two Boolean operations are required to determine this, as shown in Figure 3, line 35. When a hit on a product line is detected, the algorithm traces through the output list and sets all the corresponding outputs to driving one (Figure 3, line 42) or driving unknown (Figure 3, line 38). The example in Figure 5(a) shows the comparison between input $ABCX$ and product term $ABCX$. The input generates a hit; however, since the unknown in input corresponds with a 'don't care' condition in the product term, the unknown does not propagate. Figure 5(b) shows the same comparison process when input $ABXD$ is applied. In this case the unknown in the input propagates to the product line.

The advantage of the coded personality matrix method is that a hit can be detected with only 4 Boolean operations. If the input does not have many unknowns, only a few product lines generate a hit. The evaluation cost in the best case is thus $O(p)$. If, in the worst case, each product line generates a hit, the total evaluation cost is $O(pm)$, where p is the number of product lines and m is the length of the output lists. In either case the computation speed compares favorably to the cost of the method presented in [8] which is $O((i+o)p)$.

input variable = $AB\bar{C}X$			input variable = $AB\underline{X}D$		
product term = $ABCX$			product term = $ABCX$		
finding a hit:			finding a hit:		
iMold:	1101		iMold:	1101	
moldList[j]:	1100		moldList[j]:	1100	
	——	xor		——	xor
	0001			0001	
iMask:	1110		iMask:	1101	
	——	and		——	and
	0000			0001	
maskList[j]:	1110		maskList[j]:	1110	
	——	and		——	and
	0000			0000	
propagation of unknown:			propagation of unknown:		
iMask:	1110		iMask:	1101	
	——	not		——	not
	0001			0010	
maskList[j]:	1110		maskList[j]:	1110	
	——	and		——	and
	0000			0010	
	(a)			(b)	

Figure 5 Illustration of CPM comparison process.

5. Model Generation for Bus Type Modules

A new algorithm has been developed to generate software models for bus type substructures. A bus structure may consist of one or more busses, which can be connected together with transistors, called **link transistors**. Each bus may contain transistors that write to the bus (**input transistors**), read from the bus (**output transistors**), or precharge the bus (**precharge transistors**). A wide range of busses can be accepted by the model generator. An input/output transistor can be modeled with a combination of an input and an output transistor.

Software model evaluation may be performed efficiently by utilizing the structure of the bus. Two modes of operation are defined for a bus structure: the **normal mode** and the **exception mode**. A bus is in normal mode operation if it operates according to its intended design. Normal mode operation consists of two clock phases: the **precharge phase** and the **evaluation phase**. During the precharge phase, the bus wires are precharged or predischarged. During the evaluation phase, the precharge transistor is inactive, and the bus is controlled solely by the input and link transistors. Only one input transistor can be active during this phase. Bus wires may also be connected together by link transistors, allowing signals to flow between busses.

In exception mode, the bus operates in a manner deviating from the normal mode operation due to design errors or faults studied in fault simulation. Specifically, exception mode operation will occur when two or more input transistors are simultaneously active or when an unknown is placed on the gate of a transistor.

An understanding of the normal and exception mode operations aids in the design of a high speed model. The behavior of the bus in normal mode is simple, so that evaluation may be performed very fast. Since the bus is in normal mode for most of its operation time, evaluation of this mode is given priority. Operation in the exception mode is detected and handled accurately, but it infringes on the fast evaluation of the normal mode as little as possible. Evaluation of the exception mode may suffer a slight loss in speed but the overall performance of the model will benefit.

The evaluation consists of three steps:

(1) The input transistors are evaluated.

(2) Inter-bus signal flow is resolved.

(3) The output transistor signal flow is evaluated.

The three steps are each explained below in more detail.

5.1. Input Transistor Evaluation

In this step of the evaluation, each bus is evaluated individually. The first step in the evaluation is to determine whether the bus is in a precharge phase or an evaluation phase. This can be done by inspecting the state of the precharge transistor's gate.

In normal mode operation, the precharge phase can be performed by a single assignment statement. Exception mode operation occurs when one or more input ports are active in

addition to the precharge transistor. If they try to drive the bus to a value different from the precharge value, a conflict occurs and the final value of the bus depends on the relative strength of the input and precharge transistors. Since precharge transistors are able to supply a large charge quickly, they are stronger than input transistors and will override the input transistors. The active input ports can slow down the precharging, but in switch-level simulation this delay can be ignored.

In the evaluation phase, the precharge transistor is inactive. The bus is therefore controlled only by the input transistors. The fact that input transistors are unidirectional allows them to be evaluated easily with a lookup table. The size of the lookup table is determined by the number of strengths and states used to represent a signal. In our implementation, 4 strengths and 3 states are used. The table length is 1 Kbyte, which does not significantly increase the memory requirement since the list is stored only once for all generated models.

The unidirectional lookup method can be used for a bus in both normal mode operation and exception mode operation. There is still an advantage to the distinction of these modes of operation. Since in normal mode operation only one of the input transistors is active, the input transistors will be off during most of their operation time. By checking the gates of the transistors first, the speed of the evaluation is increased. If the gate of a transistor is active, the lookup operation is used for its evaluation. If the gate is inactive, the lookup operation does not need to be performed and the model can immediately proceed to the next transistor.

Additional speed increase may be obtained by gathering the gate signals. Usually the bus structure of a circuit consists of many identical bit slices. The input transistors can be collected into groups with common gate signals. A gate signal must then be checked only once for all transistors in a group. If the gate signal is inactive, none of the input transistors needs to be evaluated, while if the control signal is active, lookup operations are performed for each individual transistor. When the number of bit slices is large, such as 16, 32, or 64 bits, a significant speedup is obtained.

5.2. Evaluation of Inter-Bus Signal Flow

In step 1 of the evaluation, the busses are evaluated independently, thereby ignoring that two or more busses can be connected using bidirectional link transistors. In step 2 of the evaluation this bidirectional signal flow is resolved.

Figure 6 shows an example of a simple circuit with three busses. It is assumed that each bus has already been evaluated individually using evaluation step 1. To ensure that a bus in the network reaches a steady state, information from all busses must reach that bus. This means that information must traverse a path between every possible pair of busses [9].

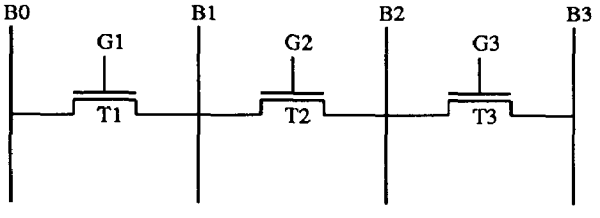

Figure 6. Link transistor network in a bus structure.

The link transistor network is evaluated with the path finding algorithm used in the compiled simulator SLS [6]. Advantage was taken of the fact that cyclic link transistor structures are not permitted in bus structures. A steady state can therefore be obtained by executing a fixed sequence of unidirectional transistor evaluations as explained in [9]. Since the sequence of evaluations is independent of the state of the transistors, the graph corresponding to the transistor network need not be stored in the software model. A fixed sequence of lookup operation can therefore be executed directly at run time.

5.3. Evaluation of Output Transistors

After steps 1 and 2, the bus lines reach their final value. In the last step of the evaluation algorithm, the output ports are evaluated. The output transistors support unidirectional signal flow, feeding signals from the bus to the output. The output port transistors are evaluated similarly to the input ports in step 1. To increase the speed of the evaluation, control gathering is performed and the state of the gate is checked before performing the lookup function.

6. Performance Evaluation

A number of software models for bus and combinational modules were generated and executed to test the accuracy and efficiency of the code. Tested bus modules varied in size from 56 to 657 transistors, and combinational modules had personality matrix sizes ranging from 35 to 28K entries. All circuits were taken from the circuit description of a commercially available chip.

Input vectors for the simulation of the modules were obtained by simulating the whole circuit and extracting the vectors applied to the modules during this simulation. The input patterns used in the speed measurements were, therefore, representative of the patterns applied to the module during actual operation of the chip. For each measurement approximately 1000 input vectors were simulated. The simulation speeds shown in Tables 1 and 2 represent the true evaluation time. It was obtained by eliminating from the total simulation time the overhead due to file access, etc.

The characteristics of the combinational gate circuits used in the performance evaluation are shown in Table 1(a). The transistor count refers to the number of elements in the transistor implementation. The circuit depth refers to the median number of levels in the tree structure of the circuit. It indicates the number of gates visited by a signal flowing from an input to an output.

Table 1(a) shows the simulation time in CPU seconds for the transistor and CPM implementations when run on a SUN 3/50 work station. Table 1(a) shows that the CPM model becomes more efficient when the circuit modules are large and have a large depth. Table 1(b) shows the storage requirements in Kbytes. The storage requirement for the CPM software model is between 7.5 and 9.0 times less than that required by the transistor implementation.

In Table 2 the simulation speed of the transistor and software model implementation of various bus structures is shown. Table 2(a) shows the simulation times for bus modules of varying sizes when the number of bit slices is two. Table 2(b) shows the simulation results when the number of bit slices of the bus structure is varied. Table 2(a) indicates that that a larger

circuit	trans. count	circuit depth	simulation time trans.	simulation time CPM	speedup
ckt1	3330	5	723.90	7.78	93.0
ckt2	2370	5	516.72	5.88	87.9
ckt3	640	1	180.92	11.96	15.1
ckt4	1350	4	220.68	4.86	45.4
ckt5	376	1	89.46	5.50	16.3
ckt6	447	4	140.34	2.28	61.6
ckt7	46	3	23.66	0.74	32.0

(a)

circuit	simulation size transistor	simulation size gate	simulation size CPM	simulation size-down gate	simulation size-down CPM
ckt1	63	64	7	0.98	9.0
ckt2	54	52	6	1.04	9.0
ckt3	32	32	4	1.00	8.0
ckt4	15	24	2	0.63	7.5
ckt5	30	27	3	0.90	10.0
ckt6	15	29	2	0.52	7.5

(b)

Table 1. Performance of combinational gate modules.

circuit trans. count	circuit input count	circuit output count	speed (CPU sec) trans.	speed (CPU sec) model	speedup
110	152	6	24.44	1.94	12.6
92	125	6	20.94	1.76	11.9
74	98	6	17.46	1.40	12.5
56	71	6	14.22	1.12	12.6

(a)

circuit # bit slices	circuit trans. count	circuit input count	circuit output count	speed (CPU sec) trans.	speed (CPU sec) model	speedup
12	657	639	36	311.52	18.46	16.9
8	437	443	24	178.40	8.94	19.9
4	220	250	12	89.36	4.86	18.4
1	55	103	3	24.18	1.66	14.6

(b)

Table 2. Performance of bus modules.

speedup is attained for models with a large number of bit slices.

7. Conclusions

This paper describes work performed as part of a larger logic and fault simulation project. With the advent of more complex and widely used VLSI circuits, fast switch-level simulation is a pressing need. The methods to generate high-level software model presented in this paper attempt to bridge this speed gap.

A software model generator was developed with the following goals in mind:

(1) The model algorithms must take advantage of the structural and behavioral information preserved in a hierarchical circuit description.

(2) The generator must be capable of generating efficient models for large modules.

The presented algorithms were implemented and successfully used for modules from a commercially available chip. The event-driven simulator CHAMP [1] performed one to two orders of magnitude faster when a module was replaced with a software model. It was demonstrated that classifying the modules according to structure and components allows a specialized algorithm to be applied to each type of module and produces more efficient code.

Ongoing work includes the development of additional algorithms for module types not yet included in the generator. In the long run, the generator is intended to develop into a comprehensive system that automatically performs classification and code generation for large circuits.

ACKNOWLEDGMENTS

This work was supported in part by Motorola, Inc. Austin, TX, and in part by the Semiconductor Research Corporation Contracts 87-DP-109 at the University of Illinois at Urbana and 88-DJ-142 at the University of Texas at Austin.

REFERENCES

[1] 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," in *Proc. IEEE Int. Conference on Computer-Aided Design*, Santa Clara, CA, 1988.

[2] R.E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Scheffler, "COSMOS: A Compiled Simulator for MOS Circuits," *Proc. ACM IEEE 24th Design Automation Conference*, pp. 9-16, 1987.

[3] G. Ditlow, W. Donath, and A. Ruehli, "Logic equations for MOSFET circuits," in *Proc. of the IEEE International Symposium on Circuits and Systems*, Newport Beach, CA, pp. 752-755, May 1983.

[4] R. M. Apte, N-S Chang , and J. Abraham, "REDUCE-Logic extraction for NMOS circuits," in *IEEE Int. Conf. on Circuits and Computers*, New York , Oct. 1982.

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

[6] Z. Barzilai, D. K. Beece, L. M. Huisman, V. S. Iyengar, and G. M. Silberman, "SLS - A Fast Switch-Level Simulator," *IEEE Transactions. on CAD*, vol. CAD-7, No. 8, pp. 838-849, Aug. 1988.

[7] A. Brish, R. Keinan, and Y. Ravid, "A Smart System that Compiles RTL Models from Schematics," *VLSI System Design*, pp. 32-35, Feb. 1988.

[8] H. P. Chang and J. A. Abraham, "Use of High Level Descriptions for Speedup of Fault Simulation," in *Proc. International Test Conference*, Washington D.C., pp. 1-7, Sept. 1987.

[9] I. Spillinger and G. M. Silberman, "Improving the Performance of a Switch-Level Simulator Targeted for a Logic Simulation Machine," *IEEE Transactions. on CAD*, vol. CAD-5, No. 3, pp. 396-404, July 1986.