AN APPROCH TO FAST HIERARCHICAL FAULT SIMULATION

Akira Motohara, Motohide Murakami*, Miki Urano,
Yasuo Masuda, and Masahide Sugano

Semiconductor Research Center
Matsushita Electric Industrial Co., Ltd. Osaka 570 Japan
* Matsushita Soft Research Inc. Osaka 570 Japan

ABSTRACT

We present an approach to hierarchical fault simulation which generates several simulation-models of one circuit and carries out simulation for each. Fault insertion and simulation-model generation is done automatically. Switch-level simulation which utilizes look-up tables is as fast as gate-level simulation. Experimental results show that using behavioral description and switch-level truth tables is effective to improve simulation speed.

1. INTRODUCTION

Fault simulation has been one of the hardest tasks in VLSI development, for the following reasons:
(1) fault simulation requires a large amount of computer resources especially for large VLSI chips,
(2) classical stuck-at fault modeling is easy to deal with, but it is not sufficient for CMOS failure models.

The concurrent fault simulation technique[1, 2] is widely used because of its efficiency and generality. In concurrent simulation, fault free machine (Reference Machine: RM) and faulty machines (Concurrent machine: CM) are simulated at the same time. The differences between RM's behavior and each CM's are maintained in fault state lists. For rather small circuits, all the faults can be handled in one simulation pass. However, for large circuits which contain many faults, several passes are required to avoid memory space problems, such as overflow of fault state lists.

Many efforts to improve concurrent simulation have been reported[3-8]. Rogers et al. introduced a hierarchical approach[5-7], which could reduce the time for fault simulation to o(*n*log*n*) by "wrapping" primitives. However, the following problems are not solved:
(1) size of wrapped table is not negligible especially for the macromodules which have many I/O ports,
(2) overhead of wrapping is not negligible if there are many unique macromodules,
(3) pass separation has to be considered, even if the hierarchical approach is adopted.

We propose an approach to hierarchical fault simulation which uses behavioral models. Our simulator creates several simulation-models of one circuit and carries out simulation for each. Our approach is based on the following ideas:
(1) in practice, it is not possible to simulate all the faults of large circuits at one pass,
(2) in ordinary concurrent simulation, a detailed description of the circuit can be replaced with a high-level description, except for the subcircuits which include faults to be dealt with,
(3) speed-up is possible if we have fewer elements described at a higher level,
(4) more realistic fault modeling is possible if we use a detailed description.

Trade-off between speed and accuracy of simulation has been discussed for many years. We concentrate on stuck-at faults at the gate-level, and stuck-open/on faults of CMOS transistors. The main reason for this compromise is efficiency of simulation. However, the following facts also support our strategy.
(1) Many large VLSIs employ CMOS technology.
(2) A greater part of the physical failures are modeled by open and short at switch-level[15].
(3) Generally, other faults such as coupling, crosstalk or delay faults are not manageable since where they happen, or how long the delay is are difficult to specify.
(4) Many of the faults of NMOS or bipolar circuits can be modeled as stuck-at faults at gate-level.

Since the first work of Bryant[9], many approaches to switch-level simulation have been reported[10-13]. Blocking is a common technique which simplifies switch-level simulation. The circuit under consideration is initially segmented into subcircuits. Within the subcircuit, there are bi-directional signal flows. However, each signal line which connects one subcircuit to another is uni-directional. Shih et al.[14] introduced a technique to model the CMOS complex gate as a function block which is represented using a logic tree. We generalized their technique so as to deal with sequential circuits and circuits which include transmission-gates or domino logic.

2. FUNCTION SIMULATION

Our simulator provides a functional and logic design environment. Designers can write and simulate their specification and realization using behavioral description. In our simulator, the circuit under consideration is represented as an interconnection of macromodules. Each macromodule has to be given its behavioral description and/or structural description. At the lowest level of hierarchy of structural description, all the elements included in the macromodule are logic primitives such as NAND, NOR, D-FF, and etc.

The designer can describe the behavior of a macromodule using a truth table, state transition table, or C-language. Whenever an event occurs at an input pin of a behavioral macromodule, the simulator evaluates its next status. If the behavior is given in the form of truth table or state transition table, the simulator just looks up the table to decide the next status. Behavior described with C-language is compiled and linked before simulation and the macromodule is evaluated just like the usual logic primitives. In this paper we call those macromodules "user-primitives". Figure 1. illustrates how the user's C source file is compiled and linked to the simulator. In figure 1, each box means UNIX file. To perform compilation and linkage without designers aid, "make" of UNIX is used in our simulator. To perform preprocessing of user-primitives, the designer just specifies the module names -- "user" in figure 1.

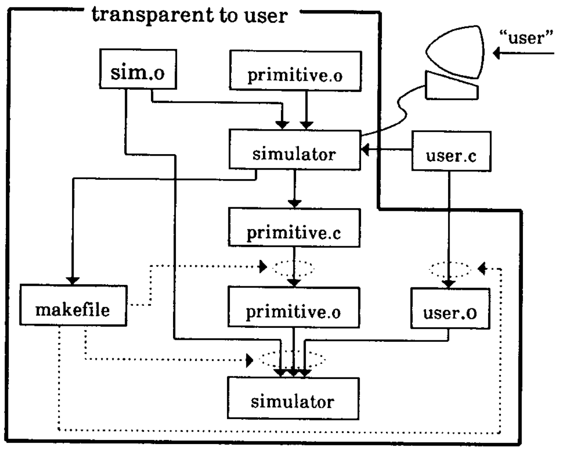


Figure 1. Preprocess of Function Simulation

This name identifies both the primitive name and the file name of the source program. Then the simulator starts the following procedure:

(1) Update the primitive table which holds the primitive-ID and its evaluation routine. This table is defined in "primitive.c" in figure 1. The simulator generates a new "primitive.c" file according to the user's source file "user.c".
(2) Generate a "makefile" which indicates how to make a new executable module of simulator.
(3) Perform "make" in order to create a new simulator.
(4) Perform "exec" to run the new simulator.

3. SWITCH-LEVEL SIMULATION

As illustrated in Figure 2, CMOS complementary logic has pull-up(P) and pull-down(N) logic[14, 16]. Each of these logic cells can be replaced with a function block. In a gate-level simulator, this function block is evaluated as a two-valued logic such as NAND or NOR. However, a gate-level simulator does not have the capability to deal with stuck-open or stuck-on faults correctly. Our simulator transforms a transistor-level circuit into an interconnection of function blocks as part of its preprocessing. Each function block has uni-directional I/O pins and truth tables. In order to handle stuck-open/on faults correctly, two tables are created; one for P-logic, and one for N-logic. Output of the function block is obtained by a rule shown in table 1.

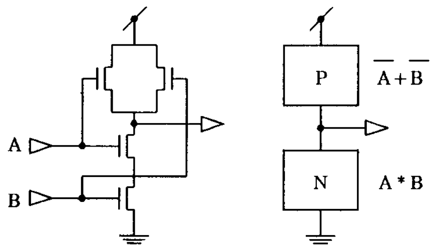


Figure 2. CMOS Cell

Table 1. State of Output

P	N	State
False	False	Z(high impedance)
False	True	LO(low)
True	False	HI(high)
True	True	X(unknown)

The stuck-open/on faults of CMOS transistors may result in Z and X, respectively. Assuming the function block having Z to hold its previous state, we can simulate "two-pattern testing" for the stuck-open faults. In general, the stuck-on faults cause Vdd-Vss short and are not logically tested. Our scheme assumes X for the situation, and the stuck-on faults are tested as potentially detectable faults. Off course, only one table is enough for the fully complementary logic cells where no faults are assumed.

Pass transistor logic can be modeled in the same way. Figure 3 illustrates a generalized logic cell. Logic expression Ti represents the condition that there is a path from the input signal Si to the output of the cell. The output status is obtained by table 1, where P and N have to be replaced with $(P + \Sigma((Si = HI) * Ti))$ and $(N + \Sigma((Si = LO) * Ti))$, respectively. There are some exceptions which can not be evaluated as described above. For example, domino logic shown in figure 4, or pseudo-NMOS logic which uses a "weak" P-transistor as a register rather than a switch, has a state LO if both P-logic and N-logic are true. For these cells, our simulator sets a flag which indicates that this function block has a weak P-transistor. If the result of evaluation is X and the flag is on, it is modified as LO. This is neither general nor a complete solution. However, it is acceptable since it enables us to apply our simple and fast algorithm to a wide variety of circuits.

The preprocess for the switch-level circuits are summarized as follows:

(1) Segment the circuit into subcircuits.
(2) Extract P-logic and N-logic (and T-logic if necessary) for each subcircuit.
(3) Reconstruct the circuit with function blocks which have truth tables.

Logic extraction is done by merging two transistors into one. The merged transistor has a coupled (ANDed if series, ORed if parallel) logic of the two transistors. Logic extraction is not always successful for some circuits such as one which has a bridge structure. If the simulator find no more transistor pairs to be merged before extraction is completed, it starts path searching from output to Vdd / Vss / input signal lines to make P- / N- / T-logic. This merging procedure is effective for the transmission gate illustrated in figure 5. At first circuit segmentation is done by merging the drain and source of one transistor into the same group, and the subcircuit shown in figure 5(a) is obtained. In this case, no transistor pairs are merged. However, the merging procedure can find an inverter (series Vdd~P-Transistor~Node~N-Transistor~Vss) and a transmission gate (parallel P-Transistor and N-transistor). Then the subcircuit is segmented further, as shown in figure 5(b).

The above preprocessing transforms the switch-level circuit into a circuit which includes only uni-directional function blocks. Stuck-open/on faults are modeled as stuck-at faults of the input pins of these function blocks, and we can apply ordinary concurrent fault simulation.

The designer can perform switch-level fault simulation more efficiently at gate-level if the switch-level description of each logic cell is available, because circuit segmentation is not necessary and logic extraction is done only once for each cell. Moreover our simulator can utilize a cell library which contains truth tables for switch-level evaluation, and if the cell library is used, the preprocessor just performs circuit reconstruction.

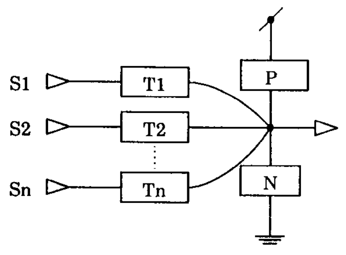


Figure 3. Generalized CMOS Cell

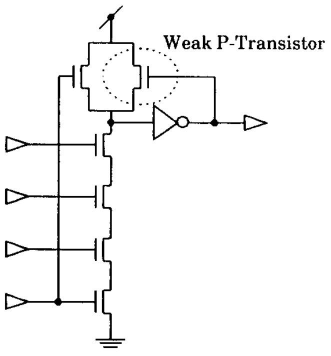
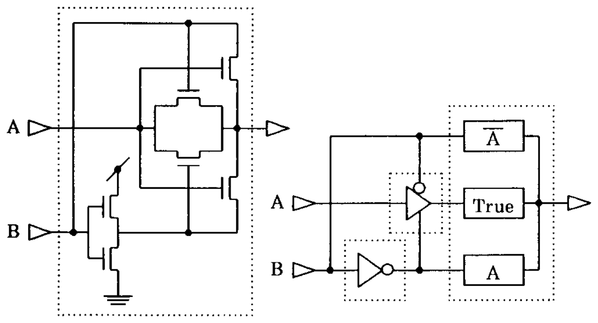


Figure 4. Domino Logic



(a) (b)

Figure 5. Transmission Gate (XOR)

4. HIERARCHICAL FAULT SIMULATION

Indeed there is no problem in performing fault simulation for small circuits. However, some strategies are necessary if the circuit is large compared with the available memory space. Our strategy can be summarized as follows:

(1) Estimate how many elements and faults can be handled at one time, with reasonable computer resource usage.
(2) Decide which nodes in the hierarchy tree should be expanded to the lowest level -- gate-level or switch-level.
(3) Create a simulation model which is the most compact to simulate all the faults under the node selected in (2).
(3) Insert faults at the expanded area and perform fault simulation.

If the circuit is so small that can be simulated in one pass, the root of the hierarchy tree is chosen and the usual "flat" fault simulation will be carried out. However, for the large circuits which require many passes, our simulator creates several simulation models. We discuss each strategy in the following sections.

4. 1. SIZE ESTIMATION

Rogers et al.[7] discussed time complexity of concurrent fault simulation. In this section we discuss the space complexity briefly in order to explain the basis of the size estimation technique adopted in our simulator.

In a good circuit simulator which is based on the event-driven technique, memory is mainly spent to maintain circuit information and event information. In addition, a concurrent fault simulator spends memory to maintain the fault state lists and CM's events. Memory consumed by RM is

$$S_{RM} = \delta_0 n + \varepsilon_0 \alpha_0 n \qquad (1)$$

where δ_0 is the memory size per element, ε_0 is the memory size per event, α_0 is the activity factor of the circuit, and n is the number of elements contained in the circuit. The first term of expression (1) is static and the second is dynamic. We assume that the event lists which have been used are reused and that the second term of (1) keeps near a constant value throughout the simulation. On the other hand, memory consumed by all the CMs is

$$S_{CM} = \delta_1 \beta \gamma n + \varepsilon_1 \alpha_1 \beta \gamma n \qquad (2)$$

where δ_1 is the memory size per fault state list, ε_1 is the memory size per faulty machine's event, α_1 is the average activity factor of faulty machines, β is the average number of faults per element, and γ is the average number of elements in the error propagation paths. In expression (2), $(\beta\gamma n)$ represents the total number of fault state lists in the circuit. Assuming fault drop, total memory size for CMs generally has a peak in the beginning of simulation and soon drops according as the number of faults decreases. Of course our concern is the peak memory size. (1) and (2) lead us to the conclusion that total memory space consumed by a concurrent fault simulator comes to

$$S = (1 + \beta\gamma)(\varepsilon\alpha + \delta)\, n \qquad (3)$$

where we let $\delta_0 \approx \delta_1 \approx \delta$ and $\varepsilon_0 \approx \varepsilon_1 \approx \varepsilon$, to simplify our discussion. We assume

$$\gamma \approx \zeta n. \qquad (4)$$

Indeed it might be rather pessimistic, but it is more reasonable than the assumption

$$\gamma \approx \frac{\sqrt{n}}{2} \qquad (5)$$

which considers the square circuit having no branches, since our simulator can deal with a wide variety of circuits including sequential machines which might have many branches and feedback loops.

In practical application, there is a ceiling for available memory and a number of passes are necessary to simulate all the faults. Relation between the available memory seize M and the number of passes N can be:

$$(1 + \frac{\beta}{N} \zeta n)(\varepsilon\alpha + \delta)\, n \leqq M \qquad (6)$$

$$N = \left\lceil \frac{\beta\zeta n^2}{\frac{M}{\varepsilon\alpha + \delta} - n} \right\rceil . \qquad (7)$$

We use an approximate estimation

$$N = Knf \qquad (8)$$

where K is a factor which depends on the data structure of simulator and the topology of circuit and $f\,(= \beta n)$ is the number of faults.

4. 2. SIMULATION MODEL GENERATION

A set of macromodules which represents one circuit has an implicit hierarchy tree itself. Our simulator uses an explicit hierarchy tree of macromodules in order to perform hierarchical fault simulation efficiently. The first task of our simulator is to make a hierarchy tree from the macromodules. The hierarchy tree contains two kinds of information -- topology of the tree and the attributes of each node of the tree. Each node of the hierarchy tree has the following attributes.

(1) Number of the elements existing in this node and the descendant nodes.
(2) Number of the faults existing in this node and the descendant nodes.

For each node "x", we use notations "x.e" and "x.f" to represent "the number of elements existing in x and the descendant nodes of x", and "the number of faults existing in x and the descendant nodes of x", respectively.

Our simulator first checks the relationship between macromodules, and makes a topology of the hierarchy tree. Attributes of each node are recursively obtained. For example, if the node A has descendant nodes B and C, then the attributes are

$$\mathrm{A.e} = \mathrm{B.e} + \mathrm{C.e} + \mathrm{A.e_0} \qquad (9)$$

$$\mathrm{A.f} = \mathrm{B.f} + \mathrm{C.f} + \mathrm{A.f_0} \qquad (10)$$

where $\mathrm{A.e_0}$ $(\mathrm{A.f_0})$ is the number of elements (faults) in the macromodule corresponding to the node A, excluding B and C.

Once the hierarchy tree is created, the simulator starts a procedure called simulation model generation. The objective of this procedure is to divide the faults into some small groups.

Figure 6 shows the algorithm of simulation model generation written in pseudo-C language. In figure 6, *hiersim(p)* is the routine which performs hierarchical fault simulation and *generate_model(p)* is the routine which performs simulation model generation. K is a constant number ($= K$, in (8)), N_{MAX} is the maximum number of passes for each simulation model which is defined before simulation. Hierarchical simulation is done by calling *hiersim(root)* where *root* is the root of the hierarchy tree. If the routine *hiersim(root)* is called, nodes are recursively checked as to whether they should be expanded. The decision is made according to the expression (8). If a node is chosen to be expanded, *generate_model(p)* is called. The routine *recursive_expand(p)* which is called by *generate_model(p)* recursively expands the descendant nodes in order to replace all the submacromodules with the lowest primitives and insert the faults within the macromodule corresponding to the node *p*. Then the macromodule expanded by *recursive_expand(p)* is linked to other macromodules using *expand(p)*. The routine *expand(p)* creates a circuit for the macromodule corresponding to the node p from its submacromodules.

After the simulation model is generated, fault simulation is carried out. Advantages of our approach are summarized as follows.

(1) We can quickly divide the problem into manageable subproblems by our size estimation technique,
(2) Our approach avoids the unnecessary detailed description before simulation.
(3) Since the top-down design method and mixed-level simulation are not special techniques, our approach can be easily applied with no penalty.

```
hiersim(p)
node    p;
{
    if ( $K * p.e * p.f < N_{MAX}$ || p is a leaf ) {
        generate_model(p);
        fault_simulation( );
    } else {
        for ( each ( c : a child of p ) )
            hiersim(c);
    }
}

generate_model(p)
node    p;
{
    recursive_expand(p);
    while ( p is not a root ) {
        p = father of p;
        expand(p);
    }
}
```

Figure 6. Hierarchical Fault Simulation Algorithm

5. RESULTS

We have implemented the algorithm and made some experiments. What we have to clarify is as follows:

(1) Speed of the switch-level simulation technique.
(2) Cost of the switch-level preprocessing.
(3) Speed of the hierarchical fault simulation.
(4) Cost of the preprocessing for hierarchical fault simulation.
(5) Memory space needed for hierarchical fault simulation.

We tried gate-level and switch-level fault simulation for the same design in order to estimate (1) and (2). And to estimate (3), (4), and (5), we compared the flat and hierarchical fault simulation. The detailed results are shown in the following sections.

5. 1. RESULTS OF SWITCH-LEVEL FAULT SIMULATION

Tables 2 and 3 show the result of switch-level simulation. Computer we used is SUN3-260C(MC68020-25MHz, Main Memory-8MB). Input data of this experiment are:

(1) Flat gate-level description of a circuit, and switch-level description of all the logic cells used in the gate-level circuit,

(2) Flat gate-level description of a circuit, and switch-level cell library which contains the truth-table of each cell.

In table 2, "Circuit Data Compilation" means the time to convert the circuit data into the data structure on memory. "Logic Extraction + Reconstruct", and "Reconstruct" show the time for the preprocessing of (1), and (2), respectively. Results show that preprocessing time is short enough, and that if the cell library is used, we can quickly complete the preprocessing for switch-level simulation.

Table 3 summarizes the result of switch-level fault simulation. Faults we handled are stuck-at faults of I/O pins of function blocks and stuck-open faults of Transistors. In each case, CPU time required for switch-level simulation ranges between 2 and 3 times as long as gate-level simulation. Considering that both the number of faults and the number of lines are 1.5~2 times greater than gate-level, we believe that our method is quite effective for switch-level fault simulation.

Table 2. CPU time(sec) for Switch-Level Preprocessing

Circuit	#Gate	Circuit Data Compilation	Logic Extraction + Reconstruct	Reconstruct
#1	131	0:00	0:00	0:00
#2	1024	0:02	0:05	0:01
#3	1600	0:03	0:10	0:03

Table 3. CPU time(sec) for Switch-Level Simulation

Circuit	#Pattern	Level	#Line	#Fault	Coverage	CPU (sec)	Norm'd
#1	60	Gate	356	536	95.52%	0:15	1.00
		Switch	594	815	93.98%	0:36	2.40
#2	53	Gate	3104	4160	97.86%	3:17	1.00
		Switch	5152	8266	93.77%	8:44	2.66
#3	64	Gate	4840	6480	94.66%	5:50	1.00
		Switch	8040	12880	90.90%	16:34	2.84

5. 2. RESULTS OF HIERARCHICAL FAULT SIMULATION

We used the 8-bit multiplier shown in figure 7 as a test circuit in order to estimate performance of hierarchical fault simulation. The circuit has a partial product generator (PPG) which contains 144 logic gates and 192 faults, and seven adders (ADD) each of which contains 94 logic gates and 355 faults.

Indeed this test circuit is small enough not to need hierarchical approach. However, we set the parameters of simulator so as to perform hierarchical fault simulation. The computer used for this experiment is SUN3-160C(MC68020-16.67MHz, Main Memory-8MB). The result of fault simulation with 50 test-patterns is shown in table 4 and figure 9. Each ADD and PPG is selected, expanded into gate-level, and faults are inserted in it, in turn. The time for hierarchical fault simulation is as same as flat simulation, however, the memory space (measured as the fault effect -- number of CMs) is reduced to about 15% of the space required for flat simulation. If the available memory size is smaller, flat fault simulation requires more CPU time according as the number of passes increases. We tried just small circuit, but it is expected that the hierarchical approach will show more advantages for the larger circuits.

Table 5 shows how much CPU time and memory space are used in each macromodule. It can be pointed out that the memory space required for each macromodule does not depend only on its size (elements / faults), but also its location. For example, the identical macromodules ADD1~ADD7 need different numbers of fault effects -- the space required for the macromodule closest to primary output (ADD7) is much smaller than others. So we should develop a more efficient size estimation method which takes account of expected length of the error propagation paths.

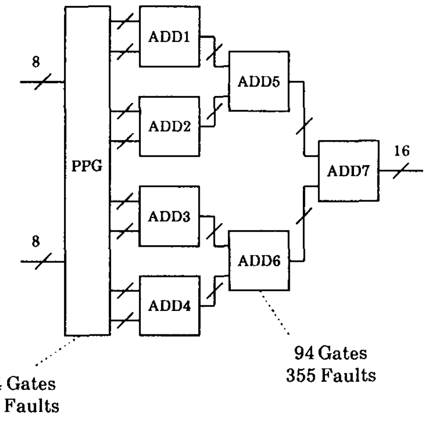


Figure 7. Test Circuit (8-bit multiplier)

Table 4. CPU time(sec) for Hierarchical Fault Simulation

Simulation Mode	#Fault Effect	#Pass	CPU (sec)	Norm'd
Flat	32,770	1	1:37*	1.00
Flat	10,000	4	1:53*	1.16
Flat	5,000	7	2:08*	1.32
Hierarchical	5,020	8	1:38**	1.01

* including model generation time (3sec)
** including model generation time (16sec)

16.3% of CPU time is wasted in simulation model generation in hierarchical fault simulation. We implemented the simulation model generation in a simple way -- read the circuit description files whenever we generate the simulation model, in order to reduce the extra memory space. This overhead is not small, so a more sophisticated method is necessary to enjoy the advantages of the hierarchical approach.

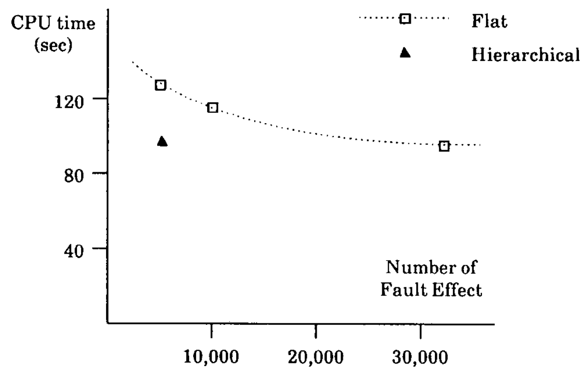


Figure 9. CPU time vs. Number of Fault Effect

Table 5. CPU time for each macromodule

Macro	#Gate	#Fault	#Fault Effect	CPU(sec)	Coverage
PPG	144	192	1,230	10	100.0%
ADD1	94	355	4,391	10	83.1%
ADD2	94	355	4,505	9	85.1%
ADD3	94	355	4,672	13	87.0%
ADD4	94	355	5,020	19	88.5%
ADD5	94	355	3,756	8	84.8%
ADD6	94	355	3,515	10	84.2%
ADD7	94	387	1,978	3	86.8%
Total	814	2,709	29,067	82	86.7%

6. CONCLUSION

We have presented an approach to hierarchical fault simulation. Our approach has two original techniques. One is to use truth tables to evaluate the switch-level circuits, and another is to generate simulation models dynamically. Both are based on general ideas -- using high-level description contributes to the reduction of simulation time and space, and using detailed description is necessary to perform accurate fault analysis. As demonstrated in the experiments, our approach successfully reduces both the time and space of fault simulation.

Presently, we are developing:

(1) more sophisticated algorithms of size estimation, based on the more realistic time and space complexity model,

(2) a more efficient simulation model generation method, and

(3) a parallel implementation of the hierarchical fault simulation.

Since each simulation model can be processed independently from others, the fault distribution technique proposed by Motohara et al.[8] can be applied with generalization.

ACKNOWLEDGMENTS

We would like to thank Kenneth M. Sarocky of Semiconductor Research Center of Matsushita Electric Industrial Co., Ltd. for making helpful suggestions. We would also like to thank Professor Hideo Fujiwara of Meiji University for his valuable comments and suggestions.

REFERENCES

[1] T. G. Ulrich and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," Proc. 10th Design Automation Workshop, pp. 25-27, June 1973.

[2] Y. Levendel and P. R. Menon, "Concurrent Fault Simulation," Fault-Tolerant Computing: Theory and Techniques, Vol. 1, Chapter 3.14, pp. 234-242, 1986.

[3] E. Ulrich, "Concurrent Simulation at the Switch, Gate, and Register Levels," Proc. Int. Test Conf., pp. 703-709, Nov. 1985.

[4] S. Gai, F. Somenzi, and E. Ulrich, "Advanced Techniques for Concurrent Multilevel Simulation," Proc. Int. Conf. on Computer Aided Design, pp. 334-337, Nov. 1986.

[5] W. A. Rogers and J. A. Abraham, "CHIEFS: A Concurrent, Hierarchical and Extensive Fault Simulator," Proc. Int. Test Conf., pp. 710-716, Nov. 1985.

[6] J. F. Guzolek, W. A. Rogers, and J. A. Abraham, "WRAP: An Algorithm for Hierarchical Compression of Fault Simulation Primitives," Proc. Int. Conf. on Computer Aided Design, pp. 338-341, Nov. 1986.

[7] W. A. Rogers and J. A. Abraham, "A Performance Model for Concurrent Hierarchical Fault Simulation," Proc. Int. Conf. on Computer Aided Design, pp. 342-345, Nov. 1986.

[8] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, "A Parallel Scheme for Test-Pattern Generation," Proc. Int. Conf. on Computer Aided Design, pp. 156-159, Nov. 1986.

[9] R. E. Bryant, "A Switch-Level Model and Simulator for MOS digital circuits," IEEE Trans. on Comp., Vol. C-33, No. 2, pp. 160-177, Feb. 1981.

[10]R. E. Bryant and M. D. Schuster, "Performance Evaluation of FMOSSIM, a Concurrent Switch-Level Fault Simulator," Proc. 22nd Design Automation Conf., pp. 715-719, June 1985.

[11]M. Kawai and J. P. Hayes, "An Experimental MOS Fault Simulation Program CSASIM," Proc. 21th Design Automation Conf., pp. 2-9, June 1984.

[12]I. N. Hajj and D. Saab, "Symbolic Logic Simulation of MOS Circuits," Proc. Int. Symp. on Circuits and Systems, pp. 246-249, May. 1983.

[13]I. D. Saab and N. Hajj, "Parallel and Concurrent Fault Simulation of MOS Circuits," Proc. Int. Conf. on Computer Design, pp. 752-756, Oct. 1984.

[14]H. C. Shih and J. A. Abraham, "Transistor-Level Test Generation for Physical Failures in CMOS Circuits," Proc. 23rd Design Automation Conf., pp. 243-249, June 1986.

[15]J. A. Abraham and W. K. Fuchs, "Fault and Error Models for VLSI," Proceedings of the IEEE, Vol. 74, No. 5, pp. 639-654, May 1986.

[16]Y. Levendel and P. R. Menon, "Transition Faults in Combinational Circuits: Input Transition Test Generation and Fault Simulation," Proc. 16th Int. Symp. on Fault-Tolerant Computing, pp. 278-283, June 1986.

[17]N. H. E. Weste and K. Eshraghian, "CMOS Logic Structures," Principles of VLSI Design: A Systems Perspective, Chapter 5. 2, pp. 160-175, Oct. 1985.