1992

Event-driven logic simulator

Hue-Hua Ann Chen
Lehigh University

AUTHOR:

Chen, Hue-Hua Ann

TITLE:

Event-Driven Logic Simulator.

DATE: January 17, 1993

EVENT-DRIVEN LOGIC SIMULATOR

by

Hue-Hua Ann Chen

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1992

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Thesis Advisor

12/7/92
Date

~~CSEE~~ Department Chairperson
EECS

12/7/92
Date

Acknowledgements

I would like to thank Professor Edwin Kay for helping in the development of this thesis. He has spent much time reviewing and commenting on the style and content of this paper.

I would also like to thank my management in AT&T Bell Laboratories for their support, especially Jane Wachinski, my immediate supervisor.

Table of Contents

ABSTRACT

As the complexity of VLSI circuits has increased, simulators have been built to verify their design. Various types and levels of simulators have evolved to tackle different phases of the design process. Improving the speed of verification at the gate-level will have a large impact on the overall design turn-around time due to the enormous number of gates included in a VLSI circuit. This thesis seeks to build a gate-level simulator model which is itself efficient and also allows empirical studies of different evaluation techniques to improve the efficiency. A highly efficient basic model is achieved mainly due to the modularity of the model and a well designed data structure. These same factors enable the model to be easily adapted to undertake various empirical studies. The empirical studies show that one evaluation technique is clearly better than the others in speed. However, there are other factors that have to be considered, too.

I. INTRODUCTION

Ever since the early days of the electronic age, design verification has been an important part of the design process of digital circuits. The reason is simple. It is much more cost effective to verify accuracy of a design before manufacturing than to repair or rebuild thousands of erroneous circuits.

Not too long ago, verification was carried out by constructing an actual prototype of the circuit from discrete components interconnected by external wires. The prototype was then used to evaluate the logical correctness and the timing characteristics of a design. This method was rendered infeasible by the explosive growth of the size of the digital devices. The number of components in a very large-scale integrated (VLSI) circuit can reach hundreds of thousands (Levendel et al. 1989). The complexity of circuitry has also increased at the same time. It has become too costly and too time-consuming to build prototypes for VLSI circuits. These factors along with the rapid improvements in speed and size of computers and the rapid decrease in the cost of computing have ushered in the computer aided design (CAD) tools.

A CAD tool which has become a viable replacement for physical prototyping as a design verification tool is the simulator. A simulator allows a designer to simulate how a circuit under design would behave in reality, thus verifying design against the customer specifications. It allows the detection and measurement of events that may be very

difficult or impossible to detect in the actual system. A simulator also enables a circuit designer to play "what if" during the design process to test different ideas and optimize the design.

The complexity of electronic devices has reached such a level that even in the field of simulation, no single simulator can handle all aspects of simulation for a complex circuit. As a result, different types of simulators have emerged to tackle different areas of simulation. One popular way to categorize the simulators is based on their level of abstraction of the digital system. Even in this area of categorization, many different models of leveling have been presented (Abramovici et al. 1990, Johnson 1979, Miczo 1986). Johnson, for example, suggested five levels in the hierarchy of circuit models (Fig. 1) (Johnson, 1979), and simulation aids are built throughout the hierarchy:

1) The behavioral simulator is at the highest level. At this level, a system is simulated in terms of the algorithms that it performs and the focus is the overall soundness of the system.

2) At the next level is functional simulator. Also called register transfer level simulator, it is used to simulate the flow of data and control signals within and between functional blocks such as registers, encoders, decoders, arithmetic logic units (ALU), etc.

3) Next in the hierarchy is the logic simulator, also called gate-level simulator,

which simulates the interconnection of switching elements or logic gates in a system. The focus here is on verifying the logical correctness of designs intended to implement functional building blocks. Hence, this type of simulator is also called design verification simulator.

4) A transistor/electrical level simulator expands individual gate to transistors to verify its behavior.

5) At the lowest level is the geometric level simulator which simulates a circuit in terms of physical shapes.

Simulation at a high level of abstraction requires less detailed processing, hence simulation speed is greater. However, the loss of information may obscure details essential to proper understanding of the circuit's behavior. The lower the level, the higher the event intensity and computational intensity, and thus more costly and more time-consuming, but more accuracy can be achieved.

One may also classify simulators according to the type of internal model they process. With this classification, simulators can be divided into two major types; compiled simulators and event-driven simulators:

1) A compiled simulator executes a compiled-code model. The compiled code is generated by converting a network description into a series of machine language instructions to reflect the functions and interconnections of the individual

components in the circuit. And simulation is accomplished by executing the resulting program a given number of (clock) times (Ulrich 1965).

2) An event-driven logic simulator operates on a framework which is a close imitation of the structure and operation of a logical network. This type of simulator recognizes that the amount of activity within a circuit at a given time is often so minimal that simulating the entire circuit is unnecessary (Ulrich, 1969). Rather simulating only those components that are involved in signal changes would serve the same purpose. This type of simulator is called event-driven because the entire simulation is driven by the events; an event is considered to have occurred when a signal change takes place. When an event occurs on a net, then all components driven by that net are simulated. Some of the activated elements may in turn change their output values, thus generating new events. To propagate events along the interconnections among elements, an event-driven simulator needs a structural model of a circuit to work with.

Compiled simulation is mainly oriented toward functional verification and is not concerned with the timing of the circuit. This makes it applicable mostly to synchronous circuits, for which timing can be separately verified. In contrast, the passage of time is central to event-driven simulation, which can work with accurate timing models. Thus event-driven simulation is more general in scope, being also applicable to asynchronous circuits (Abramovici et al. 1990).

Event-driven simulators can be further divided into some sub-categories based on the delay model each adopts. Delay is imposed by a gate to the signals propagating through it. It is the interval between an output change and the input change(s) that caused it. However, it is completely ignored in zero delay simulators which simply simulates the logic function performed by the elements. In unit delay simulators, each logic element is assigned the same delay. The rise and fall delay simulators recognize that for some devices the time required for the output signal to rise and to fall could be quite different and associate different rise and fall delays with every gate. But if the gate delays are independent of the output change, a transition-independent delay model can be used. In this kind of simulators, one delay is assigned to each gate.

Some logic simulators feature the traditional binary values, i.e. 0 and 1 to represent the states of logic gates. Some recognize that when a circuit is powered up, the initial state of some elements may not be uniquely determined. To process the unknown initial state, these simulators use a separate logic value, denoted by u, to indicate an unknown logic value. The rules of logic for a three-valued logic system are summarized in Table 1. Then it was discovered that in some circumstances, the use of the unknown value can lead to inconsistent results. To solve the problems, some simulators use several distinct unknown values. Unfortunately, this technique becomes quite cumbersome for large circuits, since now the value of some lines would be represented by large Boolean expressions (Breuer 1972). That is probably why the three-valued logic systems are still quite prevalent (VLSI 1987).

INPUT		OUTPUT		
X1	X2	AND	OR	INV(x1)
0	0	0	0	1
0	1	0	1	1
0	u	0	u	1
1	0	0	1	0
1	1	1	1	0
1	u	u	1	0
u	0	0	u	u
u	1	u	1	u
u	u	u	u	u

Table 1 Truth Table for Three-Valued Logic System
(u stands for the unknown logic value)

Another important factor in an event-driven simulator is the element evaluation technique it uses. In the simulation loop, each activated element is evaluated. Those that change value will generate new events. Various evaluation techniques have been suggested, for example input scanning, truth table, zoom table, etc (Abramovici et al. 1990). Each has its own strengths and weaknesses. Since evaluation is a key procedure executed repeatedly

during simulation, its efficiency can have a great impact on the overall system performance as follows:

1) The input scanning technique takes advantage of a characteristic in the logic gates, that often the value of a gate can be determined by just one controlling input value. Looking at Table 1 closely reveals that whenever one of the input values for an AND gate is 0, the value of the gate is 0. Similarly, for the OR gate, the controlling input value is 1. These input values, 0 and 1, are said to be controlling because they determine the value of their respective gate's output regardless of the values of the other inputs. The implication of this characteristic on evaluation is during the evaluation process, not all input values have to be scanned. The value of a gate can be determined as soon as its controlling value is scanned. The scanning has to continue to the end only when no controlling value is encountered.

2) Logic gates can also be evaluated using a truth table. This approach picks up the input values as a group and use them to index directly into a table containing the output response corresponding to that input combination. This point can be illustrated by Table 2 for the AND gate. Notice that the input value can be combined to form a binary word. And the integer value of this word can be used as an index to look up the output value in the Z column. If the output is stored in an array, say V, then the integer value of the word can be used as an index to retrieve the output value in the V array. Truth tables can also be generalized for

multi-valued logic.

X1	X2	Binary Word	Integer Value	Z
0	0	00	0	0
0	1	01	1	0
1	0	10	2	0
1	1	11	3	1

Table 2 Truth Table for an AND Gate Using Two-Valued Logic, illustrating input values can be combined into a binary word to serve as index for retrieving output value.

3) The zoom table takes the truth table one step further. Rather than examining the type of the evaluated element to determine the truth tables to access, truth tables for all the gate types are placed in contiguous memory to form a much larger truth table or zoom table. To evaluate an element, the gate type is packed in the same word with its input values so the value of this word can be used as an index into the zoom table.

The input scanning technique achieves run-time economy by not scanning all input values

when feasible. Evaluation techniques based on truth tables should be fast due to the reduction of operations required. The zoom table technique should be even more efficient than the truth table because there are fewer decisions to be made; one simple access to the zoom table produces the output regardless of the gate type.

Due to the increasing complexity of VLSI circuits, logic simulation is still a time-consuming process. In recent years, another approach has emerged, namely to build special-purpose hardware, called a simulation engine or a simulator hardware accelerator, to speed up simulation by using parallel and/or distributed processing architectures.

The main objective of the VLSI circuit designer is to obtain accurate designs with as low a turn-around time as possible. As mentioned earlier, a large portion of the time spent by simulation tools at different stages during a design cycle occurs at the lower levels, like the gate level due to the large number of gates that can be contained in a VLSI circuit. Improvement in the overall turn-around time can be achieved greatly by shortening the simulation time at the gate level.

Over the years, numerous gate-level simulators have been developed. Some were developed by vendors for commercial purposes like CADAT, HILO, and SILOS. Some were developed by institutions or corporations for research or internal applications. The intricate design details of commercial simulators are seldom revealed. That is to be expected due to the vital value of the information. The information available about these simulators are usually limited to their published features, like the number of logic levels,

the number of primitives supported, etc. Generally speaking, more information are available for simulators from the other two sources. For example, SIMAD, developed by American Microsystems Inc., is an event-driven six-valued logic simulator (Holt et al. 1981). It uses zoom table technique for gate evaluation. MOSSIM, developed by MIT, is a three-valued logic simulator using a unit delay timing model (Bryant 1980).

The objective of this thesis is to write a gate-level logic simulator model for design verification. The focus is to develop a simulator model that is efficient, flexible, and modular so it can be used to conduct empirical studies of the impact on performance of the simulator by applying different evaluation techniques. The simulator model will use functions to implement the input scanning technique. Then another mechanism, macro, is used to implement the input scanning technique. This is followed by an implementation of the truth table technique. Due to its close similarity to the truth table technique, the zoom table technique is not implemented here. This model will be a general purpose simulator, i.e. applicable to both synchronous and asynchronous circuits, therefore it will be an event-driven simulator. To focus on improving the speed while maintaining the accuracy, the model will use the transition-independent delay model and will adopt the three-valued logic.

The body of the thesis will discuss ***SIM***, the simulator model built for the project, its system organization, the design considerations put in to achieve the objective, how well ***SIM*** achieves the objective, the evaluation techniques used for the empirical studies and the result of the empirical studies.

II. SIM: a gate-level event-driven logic SIMulator

A. System Organization

SIM is a system of programs designed to be used for gate-level logic design verification. It is implemented in the UNIX[1] environment on a SUN workstation SPARC 2. It is written in ***C*** and adheres to ANSI ***C*** requirements for maximum portability. To achieve modularity, each major function is handled by a separate program and each may in turn be composed of several sub-programs. The system is composed of five basic parts:

1. A circuit-description-language compiler.
2. A stimuli-description-language compiler.
3. A command system.
4. A simulation engine.
5. An output system.

A block diagram showing the functional relationship of the various parts of the *SIM* system is presented in Fig. 2. A logic circuit can be described to the *SIM* system through the circuit-description language. The circuit description is then translated by the language compiler into a structural model composed of simulation tables. The stimuli (signal changes) to be applied to the input elements are supplied through a stimuli-description language. The stimuli description is then translated by the language compiler into event

[1] UNIX is a trademark of AT&T Bell Laboratories.

lists. The command system directs all the actions of simulation. The simulation engine performs the actual simulation. And the output system produces the output from the simulation.

1. Circuit description language compiler

A logic circuit is described to the ***SIM*** system through a circuit description language I designed. This language permits the entry of all information concerning the particular circuit at the gate level. At the gate level, circuits are described in terms of logic elements such as ANDs and ORs. Logic elements supported in ***SIM*** are AND, OR, INV (inverse), XOR (exclusive or), NAND (not and), NOR (not or), and XNOR (not exclusive or).

The circuit description language consists of description statements. Statements are of the following kinds:

1) the NAME statement, which introduces the circuit description and declares the name of the circuit.

2) the INPUT/OUTPUT statements, which specify the primary INPUT and OUTPUT components in the circuit.

3) the interconnection declarations, which describe the type of a gate, its fan-in elements, the name of the gate, and its delay value.

These three kinds of statements have to be entered consecutively. Syntax for the statements consists of a keyword followed by a white space, then a keyvalue or a list of keyvalues. A keyword is composed of all capital letters and has to be one of those

described below. The syntax of the keyvalues varies depending on the kind of statement:

1) The NAME statement has the following syntax

NAME circuit_name

Here NAME is the keyword. The keyvalue circuit_name declares the name of the circuit and has to start with an alphabetic letter, followed by alphanumeric characters or '_'. For example, demo, test, demo_1.

2) Syntax for the INPUT/OUTPUT statements consists of

component_type component_name{, component_name}

where {} stands for repeating 0 or more times. Here the keyword is the component_type, which can be either INPUT or OUTPUT. Each component_name represents a primary INPUT or OUTPUT component. Each component has to start with an alphabetic letter, followed by alphanumeric characters or '_'. The circuit description language compiler distinguishes upper case letters from lower case letters. For example, the component i_a is interpreted as different from I_a or I_A. The components are separated from each other by a ','. Multiple INPUT/OUTPUT statements are acceptable as long as each starts with the proper keyword.

3) Syntax for the interconnection statements consists of

gate_type fan_in_list, gate_name, delay

Here the gate_type is the keyword and has to be one of those gate types

supported, i.e. AND, OR, etc. The fan_in_list is enclosed in a pair of parenthesis, with the components separated from each other by ','[2]. Syntax for all components is the same, i.e syntax for the fan_in components and the gate component follows that of the INPUT/OUTPUT components. Delay value has to be a positive integer. If not specified, the delay value will be set to 1 for this gate. The interconnection declarations can be listed in any order, thus facilitating changes to the description.

In these statements, extra blanks or tabs before, between, or after tokens are allowed. A line containing only blanks is ignored.

As an example of the circuit description, a circuit[3], ***demo***, as shown in Fig. 3 can be encoded as:

```
NAME demo
INPUT i_a, i_b, i_c, i_d, i_e, i_f
OUTPUT o_c, o_e, o_f
AND     (i_a, i_b, i_c),    g_c,    2
XOR     (g_c, g_d),         o_c,    3
OR      (i_c, i_d),         g_d,    5
INV     (i_e),              o_e,    4
NOR     (i_e, i_f),         o_f,    2
(gate type) (fan-in list)   (gate name)  (delay)
```

Once the circuit description is entered, usually through a source list often referred to as netlist, the language processor compiles the description into data tables to be used by the

---

[2] Note that the fan-in list for an INV gate consists of only one element.

[3] This seemingly obscure circuit is created mainly to illustrate the intricacies of the data structures to be discussed later.

simulator. The language processor has a substantial number of checks built into it to detect and intercept most errors before they can get into the system. These checks include syntax checks (for missing parameters, illegal characters, etc.) and circuit connectivity and consistency checks such as a primary INPUT component has to fan out to some gate(s), a non-INPUT component has to have fan-in elements, etc. These features enable the users to check the coding of a circuit efficiently in terms of cost and time. At the same time, they ensure that the model to be simulated is constructed accurately.

When errors are detected in a netlist, the compiler displays the error messages describing the nature of the errors followed by the offending lines. Whether the execution of the program continues depends on the severity of the errors. If the compiler determines that the errors are minor and simulation can still be carried out, ***SIM*** will prompt for confirmation to continue. Otherwise, ***SIM*** will request that the errors be corrected.

2. *Stimuli description language compiler*

The stimuli to be applied to the primary INPUT components are described to the ***SIM*** system through the stimuli description language I designed. The information, often contained in a source file usually referred to as logic waveform, include the time indicators of when signal changes take place and the signal values.

The stimuli description language consists of homogeneous description statements whose syntax consists of a keyword and a list of keyvalues in the following format

component_name signal_change{, signal_change} [/]

where [] denotes optional. The keyword here is one of the primary INPUT components. Each signal_change is composed of a pair of time and value, separated by ',', and enclosed in a pair of parenthesis. Similar to the circuit description language, extra blanks or tabs before, between, or after tokens are allowed. And a line containing only blanks is ignored. The information for a component can span several lines as long as the line continuation mark '/' is put at the end of each line.

As an example of the stimuli description, the waveform for ***demo*** as shown in Fig. 4 is described as:

```
i_a (1, 0), (3, 1), (7, 0), (10, 1)
i_b (3, 1), (4, 0), (6, 1)
i_c (2, 1), (3, 0), (8, 1)
i_d (0, 1), (3, 0), (6, 1)
i_e (1, 1), (2, X[4]), (7, 0)
i_f (2, 0), (4, 1), (7, 0)
(INPUT name)  (time, value) (time, value) ....
```

Once the waveform is entered, the stimuli language processor compiles the information into event lists to be used by the simulator. Again the language processor has a substantial number of checks built into it to detect and intercept most errors. These checks include syntax checks and logical checks such as all primary INPUT components have to have a waveform, time indicators have to be greater and equal to 0 and in proper sequence, signal values have to be the supported ones, etc. These checks enable the users to check

[4] X stands for u, the unknown logic value in three-valued logic as mentioned earlier.

the accuracy of the waveforms before the actual simulation starts and assures that accurate event lists will be constructed. Error handling is similar to that in the circuit description language.

3. *Command System*

The control of ***SIM*** system actions is accomplished by means of a command system. The command system controls the interfaces with the user such as prompting for netlist, waveform, specification for simulated waveforms to be stored, etc. It directs the execution of the circuit description compiler and the stimuli compiler to work on the netlist and waveform to build the data tables and event lists. It starts the simulation engine upon receiving a cue from the user and runs the engine until the user specified time or when no event is left. In essence, the overall orchestration of the system execution is conducted by the command system.

4. *Simulation engine*

In the heart of the ***SIM*** system is the simulation engine. Here the actual simulation of the circuit under test is performed, based on the structural model composed for the circuit by the circuit description language compiler and the event lists composed by the stimuli description language compiler. The result from the simulation indicates how the circuit will behave over a certain period of time. The simulation engine is driven by the simulation algorithm which manipulates the events, the core units in simulation, to occur

in a correct temporal order. ***SIM*** implements the event-driven simulation algorithm described in Abramovici et al. (1990). The main conceptual flow in the algorithm can be depicted as shown in Fig. 5.

Fig. 5 shows that at the beginning of each loop during simulation, the simulation time is advanced to the next unit, which becomes the current simulation time. Next, the simulator retrieves from the event list the events scheduled to occur at the current time and propagates each event's signal value to all of its fan-out elements, thus activates the fan-out elements. Evaluation of the activated elements then takes place and may result in new events. The new events are scheduled to occur in the future according to the delays associated with the operation of the elements. The simulator inserts the newly generated events in the event list. The simulator continues as long as there is logic activity in the circuit; that is until the event list becomes empty.

5. *Output System*

The result from the simulation consists of events scheduled for all the components in the circuit simulated. Depending on the situation, the designer may not want to watch the result of all components. Since the result will be stored in a file specified by the user and storing events takes up time and space, it makes sense to store only the events generated for those components that the designer is interested. The output system in ***SIM*** lets the user specify the components to be stored. The user has three options: to list the components one by one, enter 'out' for all OUTPUT components, or 'all' for every single

component in the circuit. If nothing is entered, 'all' is assumed.

For the circuit ***demo*** used in the example, if the user has specified 'g_c g_d out' as components whose waveforms are to be stored, the output file would contain the following data:

```
at 3  g_c: 0
at 3  o_f: 0
at 4  o_f: X
at 5  o_e: 0
at 5  g_d: 1
at 6  o_f: 0
at 6  o_e: X
at 8  o_c: 1
at 8  g_d: 0
at 9  o_f: 1
at 11  o_c: 0
at 11  o_e: 1
at 11  g_d: 1
at 12  g_c: 1
at 14  o_c: 1
at 15  o_c: 0
```

The output, if plotted using a graphics package, would look like Fig. 6.

To provide some insight into the use of the ***SIM*** system, a sample session for using ***SIM*** to simulate ***demo*** is presented in the APPENDIX.

B. Design considerations

Obviously for the development of such a complicated system, many things have to be considered during the design and implementation of the system. It would be too

voluminous to cover all of them in the thesis. Here only the major considerations will be discussed.

1. *Single-pass vs. multi-pass in the two compilers*

In the design of the two language compilers, I had to decide whether to adopt a single-pass approach or multi-pass. Multi-pass was considered as one possible approach because it solves a problem which can be illustrated by looking at how the circuit description language compiler translates the information in the netlist into the data tables for simulation. As the compiler goes down the netlist line by line to analyze the information and create records in the simulation tables, some essential information for a component may not be available until a few line below. The multi-pass approach solves the problem by reading the netlist several times until it gets all the information to construct the data tables for all the components. In contrast, the single-pass approach attempts to accomplish all of that in one pass. If possible, the single-pass approach would be more efficient because it saves time by reading the netlist only once. The two compilers in ***SIM*** were designed with a single-pass approach. This experience proves that single-pass approach is possible with a carefully designed data structure.

2. *Data structure*

In the design of a program, the data structure is as important as the overall algorithm. The data structure holds the information that is operated on during the execution of a program.

The right data structure for an operation can make it simple and efficient while the wrong one can make it cumbersome and inefficient. Data structure is important because the way data are represented significantly affects the clarity, conciseness, speed of execution, and storage requirements of the program.

The impact of data structure on the program efficiency of a simulator is even greater due to the complexity and magnitude of data to be processed by a simulator. The simulation algorithm operates on two sets of data: a structural model of a circuit to propagate events and a list of events to simulate signal changes. Its efficiency depends heavily on how well these two are designed to perform the tasks and how well they are connected to each other. The significance of the data structures cannot be overstated because they also dictate the simulator's accuracy, generality, and extensibility. A poor choice of structures could completely undermine the entire development efforts.

To explain the data structures designed for the ***SIM*** system, general block diagrams of the structural model and the event list are presented first (Fig. 7 and 8), followed by a closer view of the data structures in an actual application of simulating ***demo*** (Fig. 9, 10, and 11).

The structural model, as illustrated in Fig. 7, is composed of four tables: the component table (COMP_tab), the fan-out element table (FAN_OUT_tab), the fan-in element table (FAN_IN_tab), and a hash table (HASH_tab). Each stores a specific type of data and serves a specific purpose. Connected and combined, they form the structural model of the

SIM system. The model is constructed by the circuit description language compiler during translation of the netlist and is accessed by the simulation engine for information during simulation.

COMP_tab: This is the basis for the operation of the simulator. It contains information for all components in the circuit, including pointers to all information used in evaluation of elements during simulation. Specifically each record contains the following information for a component: name, pointer to its first fan-in component's value in the FAN_IN_tab (fan-in pointer), number of fan-in components (nfanin), signal value, pointer to the FAN_OUT_tab, logic gate type, delay value, output flag, pointer to next component in the COMP_tab, and pointer to next component with same hash value.

FAN_OUT_tab: This is used to store the interconnections between components. A component which fans out to a gate will point to an entry in this table. The entry pointed in turn points to the record for the fan-out gate in the COMP_tab, thus connects a component with its fan-out elements. The entry in the FAN_OUT_tab also contains a pointer to the FAN_IN_tab. The entry in the FAN_IN_tab will be used to store the value of the component. This value, along with the values of its other fan-in elements, will be used in the evaluation of this particular fan-out element during simulation.

FAN_IN_tab: As mentioned above, this is used to store signal values passed in from its fan-in components for a logic gate during evaluations.

HASH_tab: This is used to store index of components in the COMP_tab.

Fig. 8 shows the data structure for storing events in ***SIM***, so far referred to as event list in the thesis. The event list is actually composed of two connected parts. One is the TIMEWHEEL array and the other is the EVENT_tab. An element in the TIMEWHEEL array could point to an entry in the EVENT_tab. Each event contains the address of the event's component in the COMP_tab and its signal value at that time unit.

The event list is constructed for the primary input components by the stimuli language compiler during translation of the waveform and is used by the simulation engine to schedule simulation. This list is updated by the simulation engine during simulation as new events are created. Due to the significant role it plays in the event-driven simulation, a separate section is devoted to discuss the TIMEWHEEL later in the thesis.

In the structural model and the event list, only HASH_tab and TIMEWHEEL are fixed size arrays. The rest, loosely referred to as tables, are actually linked lists, not fixed length tables. There is a good reason for not using fixed amount of storage because whatever the size it has, it might not be large enough in some cases while wastefully large in others. Using linked lists allows the "tables" to grow dynamically.

To help the understanding of the structural model and the event management, a representation of the internal image of the tables in the simulation of ***demo*** are depicted in Fig. 9, 10 and 11.

Fig. 9 shows how the components in the COMP_tab are connected to the HASH_tab. Since each component has a different hash value, each is connected to a different slot in the HASH_tab. The only exception is g_d. Component g_d shares the same hash value as i_a, thus will attempt to connect to the same slot in the HASH_tab. In other words, the two components collide with each other. ***SIM*** solves the collision problem by linking all elements sharing the same hash value in a linked list. So, in this example, g_d is linked to i_a for the HASH_tab via a special pointer. This arrangement allows access to components with the same hash value. Since there are times when all of the components in the COMP_tab have to be accessed, another pointer is set up to link the components together for the COMP_tab.

Fig. 10 shows how gates g_c, g_d, and o_c are connected to their respective fan-in components through the connections between the COMP_tab, the FAN_OUT_tab, and the FAN_IN_tab. Here each record in the COMP_tab and the FAN_OUT_tab is presented separately to make the interconnections more visible. A closer look at gate g_c illustrates how a gate with multiple fan-in elements is handled in this structure. Since g_c has three fan-in components: i_a, i_b, and i_c, ***SIM*** has the fan-out pointer of each of the three point to g_c. The pointers to the FAN_IN_tab for these three point to a contiguous area in that table with the first one in that area pointed to by the fan-in pointer of g_c. And g_c's nfanin (number of fan-in elements) shows 3. How the connection is made for an element fanning out to multiple components is exemplified by component i_c. Since i_c fans out to g_c and g_d, its fan-out pointer points to a list of two entries in the

FAN_OUT_tab, one to g_c and one to g_d. The two entries in the FAN_OUT_tab point to two entries in the FAN_IN_tab for i_c, one for evaluation of g_c and the other for evaluation of g_d.

Fig. 11 shows the events for the primary inputs included in the waveform file. For example, it shows at time 1, two events occur: i_e changes value to 1 and i_a changes to 0. In the figure, the * in front of the component name is used to denote its address in the COMP_tab.

Several points are worth mentioning here in the design of the data structures for ***SIM***.

Why use the HASH_tab?

Several techniques were considered as a means to build and access the COMP_tab, such as linear list, binary tree, and hashing. For the management of the COMP_tab, ***SIM*** needs a method that offers efficient insertion and efficient searching. First, before a component is inserted in the COMP_tab, searching is done to make sure that it is not there already. Then insertion is done if it is not in the COMP_tab. And once a component is inserted, it could be looked up many times in building the rest of the COMP_tab and in scheduling events. Therefore, efficient insertion and searching are crucial to the performance of ***SIM***.

The linear list is the simplest to implement, but its performance is poor when the number of entries, n, in the list gets large as its processing time increases linearly with n. When

n is large, the binary search takes much less time, in the worst case, than the linear search because it make lg n rather than n comparisons going through its loop body lg n rather than n times (Korsh et al. 1988). However, it requires the construction of the binary tree, an overhead to guarantee the elements to be in sorted order which is not needed in ***SIM*** because where one component is stored in relation to the others has no bearing at all on the program execution. The hashing technique is chosen because it offers excellent average search and insertion times. Its only drawback is that it does not support ordered traversals. But as mentioned above, order is not important here.

Why create a separate table for fan-out information?

The reason for using a separate table to store fan-out information may not be clear when one looks at Fig. 7. It seems that the fan-out pointer could point directly to the fan-out gate in the COMP_tab to indicate the connection. That is true when all components fan out to one element. But that is rare in real circuits. In real circuits, components fan out to various number of elements. Assigning a fixed number of pointers in the record for a component has the same drawbacks as mentioned earlier in the discussion for fixed amount of storage; i.e. the number of pointers might not be enough for complicated circuits, and wasteful for simple circuits. With a separate table, which is actually a linked list, to store the information, maximum flexibility can be achieved.

Why create a separate table for fan-in information?

For similar reason, the FAN_IN_tab is used to store values for various number of fan-in elements for a logic gate. One point worth mentioning here is that a gate component's fan-in pointer points to its first fan-in component's entry in the FAN_IN_tab and the entries for the fan-in components for a gate occupy contiguous slots in the FAN_IN_tab. This strategy may appear redundant because a gate which fans out to several elements would have its value stored in the value area of every one of its fan-out elements. There are advantages for doing this. First, this allows instantaneous access to all of its fan-in values during evaluation of a gate. Secondly, the modularity of this design facilitates the empirical studies of the evaluation techniques to be conducted.

Why store component's address in the EVENT_tab instead of its name?

The reason again is for efficiency. By storing the address of a component, it does not have to be looked up again during simulation.

A look at how an event is simulated will demonstrate how well the data structures facilitate the simulation. Fig. 11 indicates that at time 0, component i_d becomes 1. So, when the simulation starts, this will be the first event to be simulated. First, in the propagation stage, armed with i_d's address, ***SIM*** quickly locates its record in the COMP_tab. From there, ***SIM*** immediately finds its entry in the FAN_IN_tab (Fig. 10), through its entry in the FAN_OUT_tab. This allows ***SIM*** to quickly transport i_d's value from the TIMEWHEEL to the FAN_IN_tab. Similarly, in the evaluation stage, i_d's fan-out pointer points to g_d, and g_d's fan-in pointer in turn leads ***SIM*** to its first fan-in

value in the FAN_IN_tab. Since g_d's nfanin (number of fan-ins) indicates it has two fan-in elements, two fan-in values starting from the one pointed to by g_d's fan-in pointer will be used in g_d's evaluation, in this case one is for i_c and the other for i_d. Component i_c's value is X because before simulation starts all fan-in values are initialized to unknown. Component i_d's value is 1 from the propagation operation. ***SIM*** now has all the information it needs for evaluating g_d. Since g_d is an OR gate, the result of evaluation would be 1, different from original value of X[5]. So, we get a new event and the event will take place at time 5 because g_d's delay is 5, 0 + 5. This event will be inserted by ***SIM*** at time 5 in the TIMEWHEEL.

3. Timewheel

Event list management is the mechanism used by the simulation engine to schedule events. Two prevailing data structures for event list are linear linked list and array of linked list. The advantage of using a linear linked list is the advance of time can be accomplished by jumping from event to event rather than scanning for events within a table. However, to insert an event in this linked list requires searching an average of half the elements in the linked list and then modifying two pointers. As the number of events grows, due to increased circuit size or increased activity, the average search time grows. To reduce this time, a scheduling mechanism, called a delta-t loop or "timing wheel", was introduced by Ulrich (1965, 1969). In essence, the timing wheel is an array of linked lists.

---

[5] Before simulation starts, all components' value is initialized to unknown, too.

This is more efficient than the linear linked list because the search is avoided by using the time t to provide an index into the array.

The array in the timing wheel represents time, one cell for each time unit. In scheduling, the simulation engine advances from cell to cell, doing nothing at cells with no linked list attached and processing the events at cells with list of events attached.

Obviously, to run a long simulation, over many units of simulated time, would require an array of prohibitive length. To circumvent that, the timing wheel was designed with a cyclic strategy and should be viewed as a circular table. In other words, using the timing wheel, the simulation is conducted in cycles, with the engine scanning from the beginning of the wheel to the end in each cycle, till no events are left. With the circular nature of this structure, the time slots into which newly scheduled events must be inserted can be determined by using a modulo M addition where M is the size of the array. In this mechanism, events beyond the range of the wheel are stored in an overflow "remote" event list and should be brought into the wheel as their time falls within the cycle of simulation.

Due to the efficiency of the timing wheel mechanism, it has become the predominant scheduling method used in event-driven simulators (Breuer et al. 1981). It is adopted by ***SIM*** for the same reason and has proven to serve its purpose well.

4. Memory Management

As mentioned earlier, linked lists are used extensively in ***SIM***. The very reason for using linked lists dictate memory to be allocated as it is needed. This dynamic allocation of memory can be easily carried out in ***C*** by calling the malloc function each time when a new item is to be created. But malloc is not a speedy process and if it has to be invoked thousands of times, as it would be in simulating modern day circuits, the performance of the program would be affected unfavorably. To improve the program efficiency, ***SIM*** uses a different approach. In ***SIM***, a chunk of memory is allocated a time and a portion of it is handed out each time when a new item is to be created. When one chunk of memory is exhausted, another chunk is allocated. This saves time in allocating memory while still maintaining the dynamic nature of data structure. The size of the chunk can be set customarily. If sufficient memory exists in the system where the program will be deployed, the size can be set to a large amount to reduce the times that memory has to be allocated and thus improve efficiency. However, if the memory availability is a concern, the size can be set smaller, and thus set back the efficiency a little bit. In essence, a trade off between space and efficiency has to be made depending on the situation.

III. Results

As mentioned earlier, the objective of creating ***SIM*** is twofold; 1) to develop a gate-level logic simulator model that is efficient, and 2) to create a model that is flexible and modular so it can be used to conduct empirical studies of the impact on performance by applying different evaluation techniques.

Before a discussion on the efficiency of ***SIM*** can start, how efficiency is measured has to be determined first. One of the most commonly used indicators of program efficiency is running time: how long the program takes to perform its tasks. In the case of an event-driven simulator, since its core task is scheduling events, its efficiency is usually measured by how long it takes to schedule a certain number of events. To put it in another way, efficiency is measured by how many events it can schedule in a certain time period.

To measure ***SIM***'s execution time, the command, *time*, provided by the UNIX operating system on SUN is used. It times the execution of a given command. When the command is completed, *time* displays the elapsed time during the command, the CPU time spent in the system, and the CPU time spent in the execution of the command. The combined CPU time is used to measure the performance of ***SIM***.

A test circuit is needed to serve as the subject for simulation by applying different evaluation techniques. Preferably this circuit generates a large amount of network

activities so meaningful empirical studies can be conducted. The circuit ***test***, as presented in Fig. 12, is picked for exactly that reason because given the stimuli as presented in Fig. 13, it will generate events continuously until the specified time limit. For the empirical studies of each evaluation technique, the circuit ***test*** was simulated ten times and the average from these simulations was then used to represent the performance of that evaluation technique.

A. Assessing *SIM*'s Performance

The result from using ***SIM*** to simulate the circuit ***test*** for 300,000 time units showed that it took ***SIM*** 59.217 CPU second to run. This included the compilation of the netlist and the waveform, and the simulation of 2,999,962 events. From another angle, this means that ***SIM*** can simulate 50,660 events per CPU second. This shows that ***SIM*** is very efficient compared to the other software gate-level logic simulators.[6]

B. Empirical Studies and *SIM*'s adaptability for them

1. Input Scanning by functions

As mentioned earlier, ***SIM*** uses functions to implement the input scanning evaluation

---

[6] There is no clear-cut benchmark of efficiency for gate-level simulators. But a rule of thumb of 10,000 events per CPU second can be deduced from Zycad's claim that their hardware accelerator can simulate 1,000,000 events per CPU second and that their hardware accelerator's performance is 100 times faster than most software simulators. (Zycad is a major vendor of hardware accelerators).

technique. The determination of the result for each gate type is carried out by a separate function. The implementation utilized the function pointer feature in ***C***. In ***C***, once a function is defined, the function name also serves as a pointer to the function, namely, the address where the function's definition begins in memory. Since a function pointer is just like any other pointer, it can be assigned as a value to pointer variables of the right type and also passed as an argument in a function call. Using function pointers, the evaluation of the activated elements is carried out in one single statement in ***SIM***. That exact same statement will call the proper function depending on the gate type of the element.

As demonstrated by the result presented earlier, using functions to implement the input scanning technique serves the program well. This methodology is flexible to implement. Functions can be added for other gate types without changing that evaluation statement.

2. *Input Scanning by Macros*

The input scanning evaluation technique can also be implemented by using macros. Macro is one of the facilities provided by ***C***. It uses the #define directive and has the form

#define name replacement text

The replacement text can be as simple as a constant or as complex as a function. Like regular functions, the function-like macros can be defined with arguments. The macro substitution by its replacement text is performed by the ***C*** preprocessor, which is conceptually a separate first step in compilation.

The principal benefit of macros and macro calls is run-time efficiency. A macro call saves the function-call overhead by producing in-line code at compile time. Macros, on the other hand, increase code size and have some drawbacks. For example, pointers may address functions as mentioned above but not macros. Furthermore, not all functions will convert to macros. And even for those that do, some extra work might be needed to make the conversion work.

The adaptation of ***SIM*** to use macros involved two modifications. First, due to the lack of pointers for macros, the single evaluation statement mentioned above has to be replaced by a multi-statement conditional construct which invokes different macro calls according to the gate type. Second, the evaluation functions have to be converted to macros. Due to the modularity design of ***SIM***, these modifications are localized to one area and can be carried out with ease.

The empirical studies showed that ***SIM*** with macro took 54.858 CPU second to simulate 2,999,962 events. In other words, it simulates 54,686 events per CPU second. That is a 7.9% improvement over the implementation with function. If we focus the comparison on the two different implementations by suppressing the output operation, macro's improvement is actually higher. Without output, function simulates 74,113 events per CPU second while macro simulates 82,333 events per CPU second, an 11.1 % improvement.

3. Truth Table

The implementation of the truth table technique uses the bit-field in *C*. A bit-field is a set of adjacent bits within a single storage unit, for example a binary word, and allows defining and accessing the bits directly.

As mentioned earlier, truth table implementations should be quite efficient due to reduced operations. However, there are several drawbacks associated with this technique. First, this requires the building of the truth tables. Even though this has to be done only once, when the number of inputs changes, the tables have to be rebuilt. On the same note, the number of inputs is restricted by the truth tables constructed. Second, the size of the tables increases exponentially when the number of inputs or logic values increases[7]. As a result, this technique is usually used when the elements depend on a small number of inputs. Third, software developed on a hardware platform may not be portable to the other platforms due to different methods used by different computers to store binary words.

The adaptation of ***SIM*** to implement the truth table technique was a little more involved than macro. First, a program was written to build the arrays to store output values for six-input three-valued truth tables. These tables are loaded into ***SIM*** during compilation. Second, the FAN_IN_tab was restructured to assign a binary word to each gate which has fan-in elements. The tie between the FAN_IN_tab and the FAN_OUT_tab still exists to connect a gate to its fan_in elements. Third, a small number of programs have to be

---

[7] Generally speaking, for k-valued operation, to figure out how many bits to code the k values is to find the q, where q is the smallest integer such that $k <= 2^q$. Since there are possibly 2^q representations for each input variable, if there are n variables, the array size has to be 2^{qn}.

modified to accommodate the restructuring of the FAN_IN_tab. Last, the conditional construct used in the macro section for evaluation was modified to retrieve logic values from the proper arrays. Again, due to the modularity of ***SIM*** and its data structure, the modifications are localized and manageable.

The empirical studies showed that ***SIM***, using the truth table technique, took 47.078 CPU seconds to simulate 2,999,962 events. In other words, it simulates 63,723 events per CPU second, better than both implementations of the input scanning technique, with a 16.5% improvement over the macro implementation and 25.8% improvement over the function implementation. Without output, it simulates 100,370 events per CPU second, a 21.9% improvement over macro and 35.4% improvement over function.

The results are summarized in Table 3.

	Input Scanning		Truth Table
	Function	Macro	
With Output	50,660	54,686	63,723
Without Output	74,113	82,333	100,370

Table 3 Empirical Studies Results from running ***test*** for 300000 time units, showing truth table technique is more efficient than the two implementations of input scanning.

IV. Conclusion

Mainly due to its modularity and a well designed data structure, ***SIM*** has proven to be a highly efficient gate-level logic simulator that can simulate hundreds of thousands of events accurately. These same factors have enabled ***SIM*** to be easily adapted to study the impact on performance by applying different evaluation techniques.

The empirical studies prove that the truth table evaluation techniques yields a better performance than the two different implementation of the input scanning evaluation technique. But the truth table technique is less flexible, occupies much more space which could increase exponentially, and has potential portability problem. These factors have to be weighed against the improved performance when a decision has to be made on which evaluation technique to adopt in building an event-driven logic simulator.

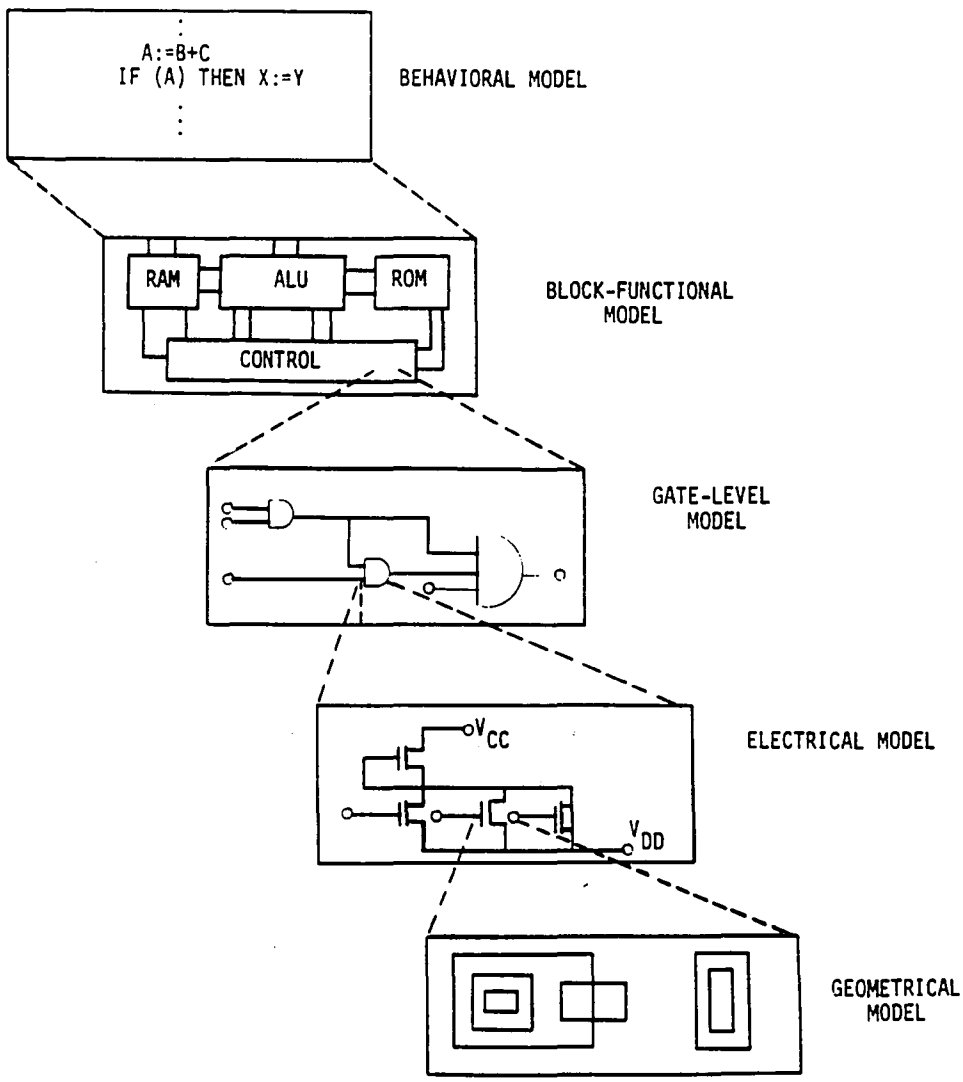

Fig. 1 Hierarchy of Digital Circuit Models

In a top down design, the process starts with the system specification at the behavioral level and goes through a series of decomposition until reaching the physical layout at the geometrical level. Simulators are built throughout this process of decomposition.

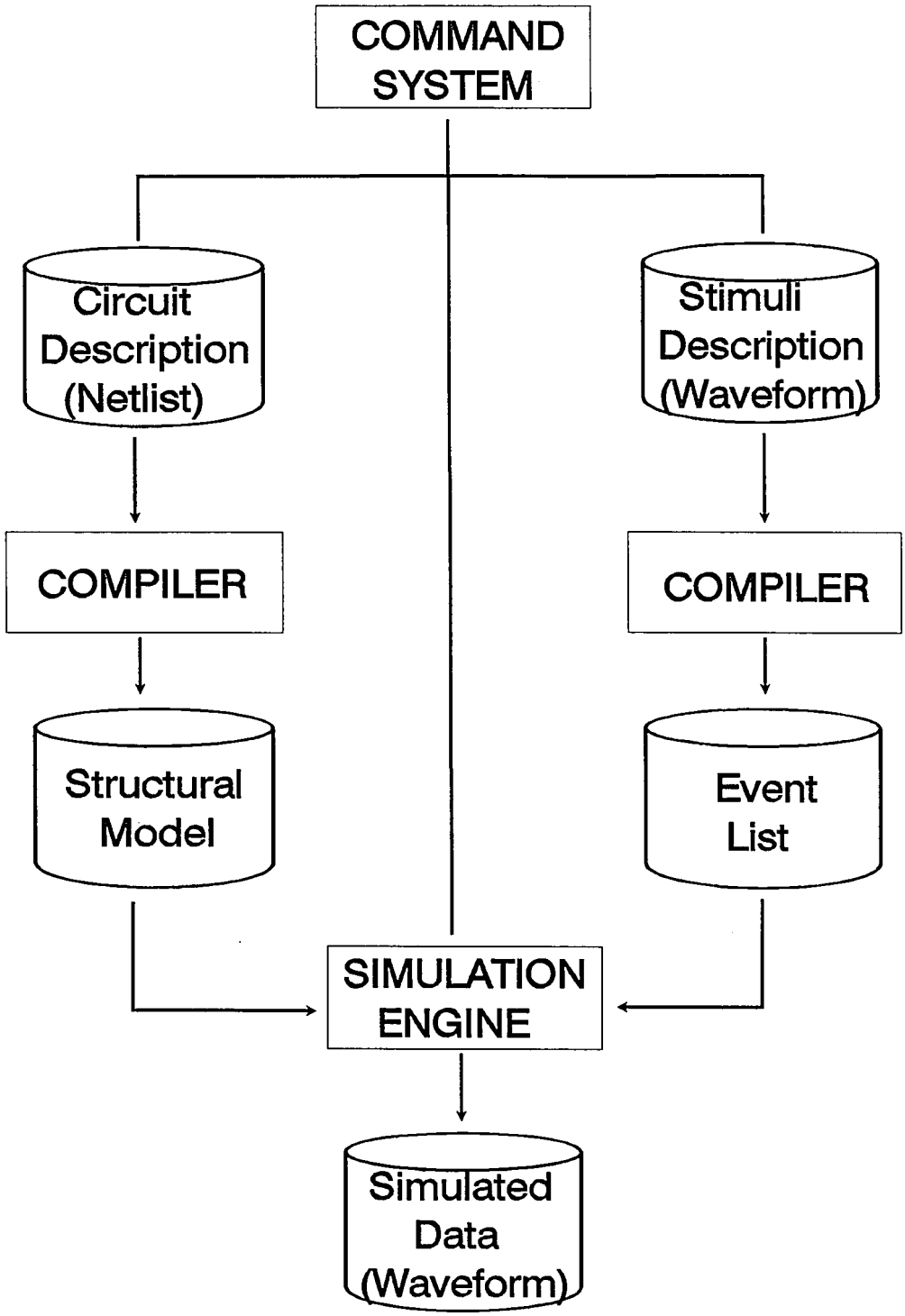

Fig. 2 Block Diagram of SIM System, an Event-Driven Logic Simulator
The command system directs all actions. The two compilers translate circuit description and stimuli descriptionn into data structures for the simulation engine which does the actual simulation.

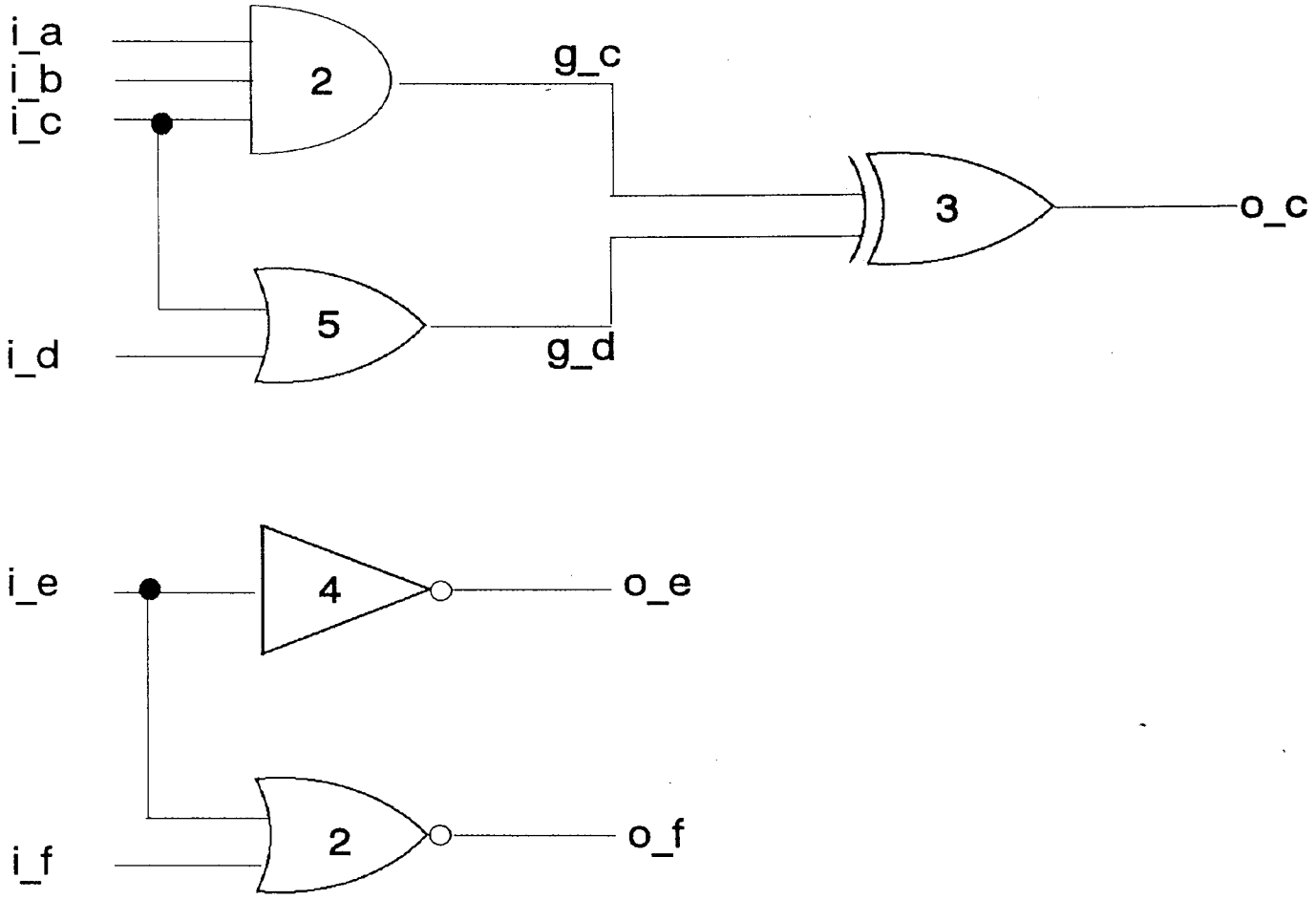

Fig. 3 Circuit demo
Created to illustrate the data structures of SIM (number inside each gate denotes the delay value)

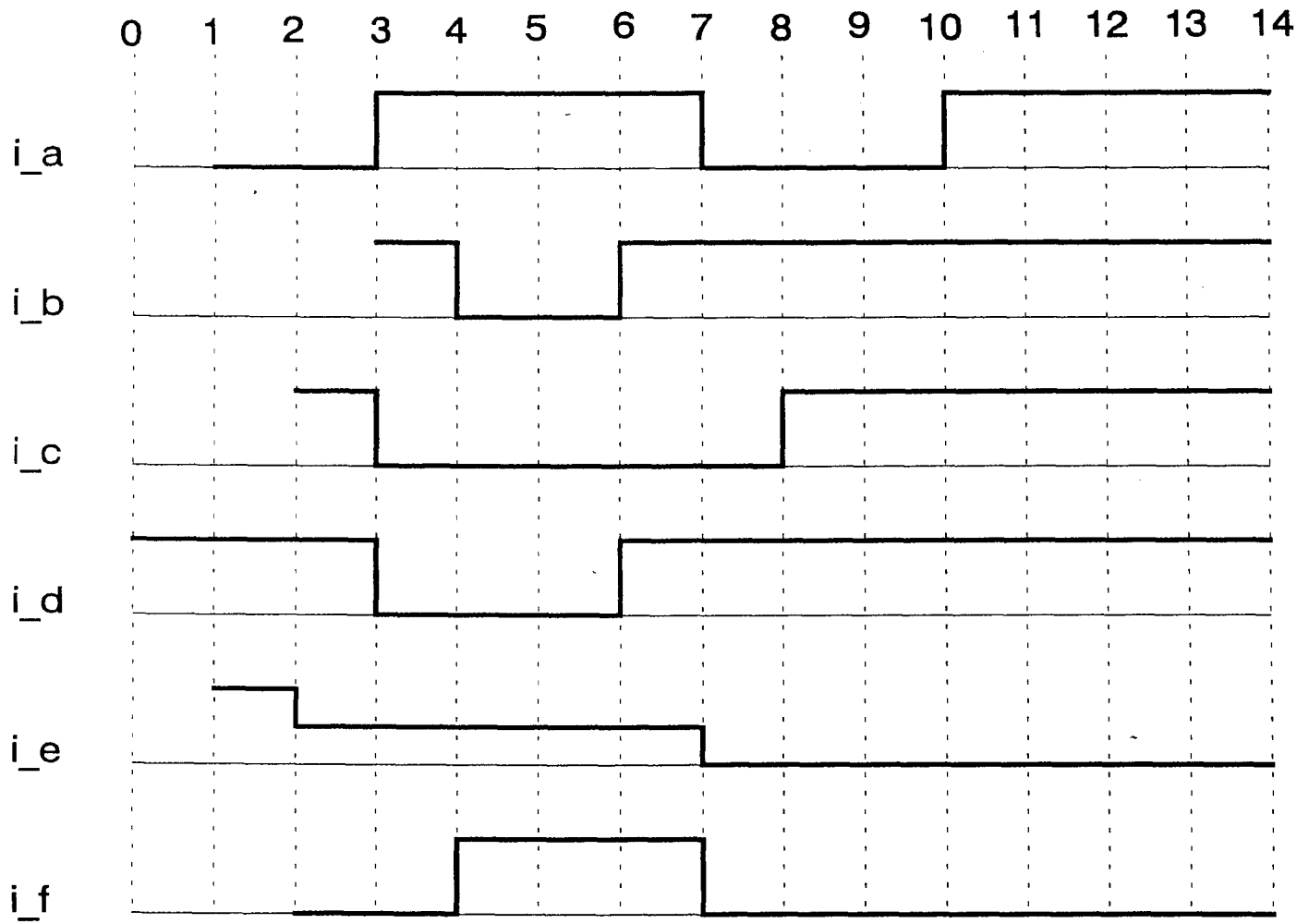

Fig. 4 Waveform for inputs to circuit demo of Fig. 3

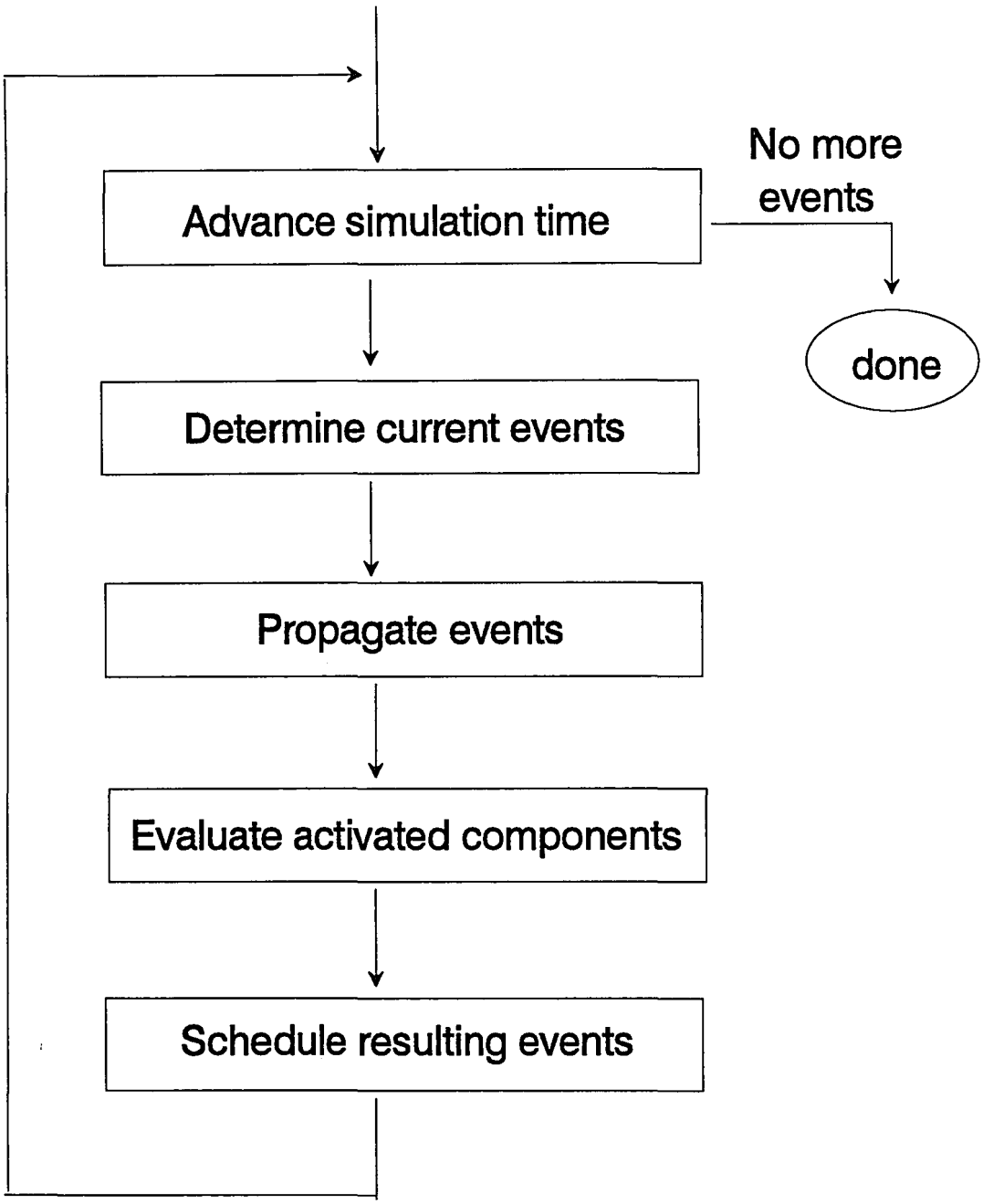

Fig. 5 Main Flow of Event-Driven Simulation
During simulation, these tasks are carried out in each cycle until no events are lefft.

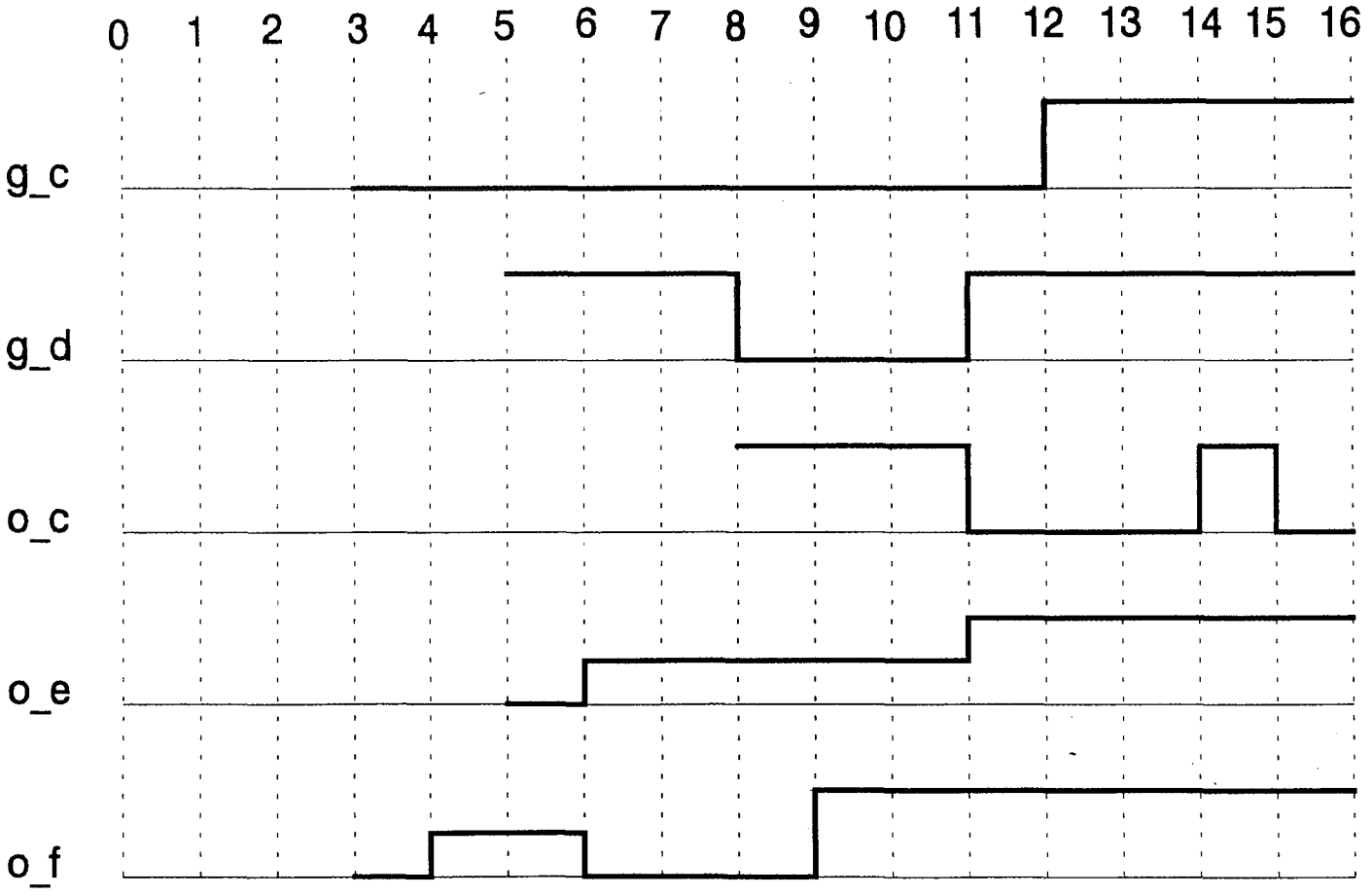

Fig. 6 Resulted Waveform from Simulating demo of Fig. 3

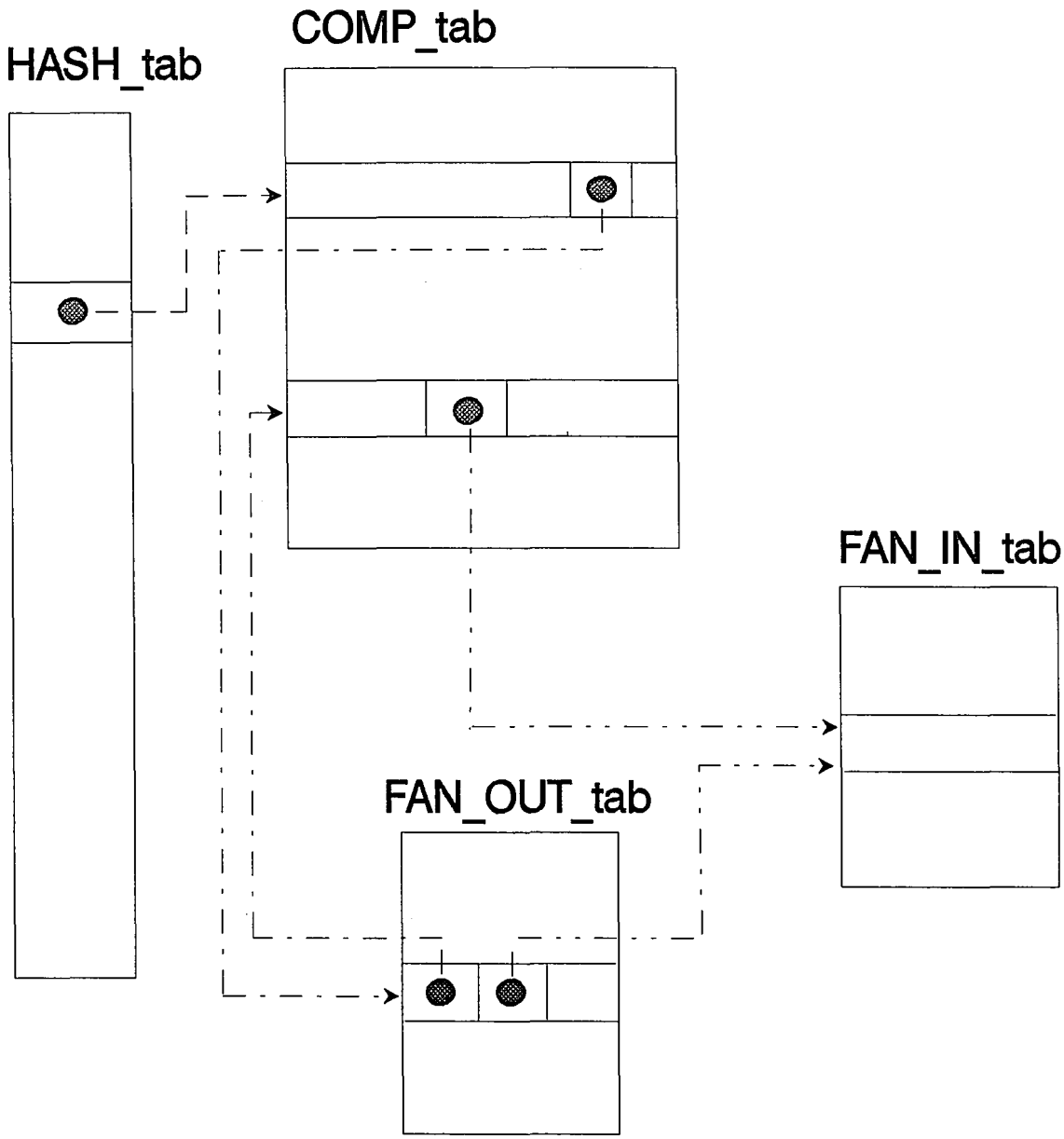

Fig. 7 Structural Model of SIM, data basis for simulation.
COMP_tab contains information for all components in a circuit.
FAN_OUT_tab stores the interconnections between components.
FAN_IN_tab contains signal values for fan-in components.
HASH_tab stores index of components in the COMP_tab.
Each dot with a dashed line is a pointer connecting tables together.

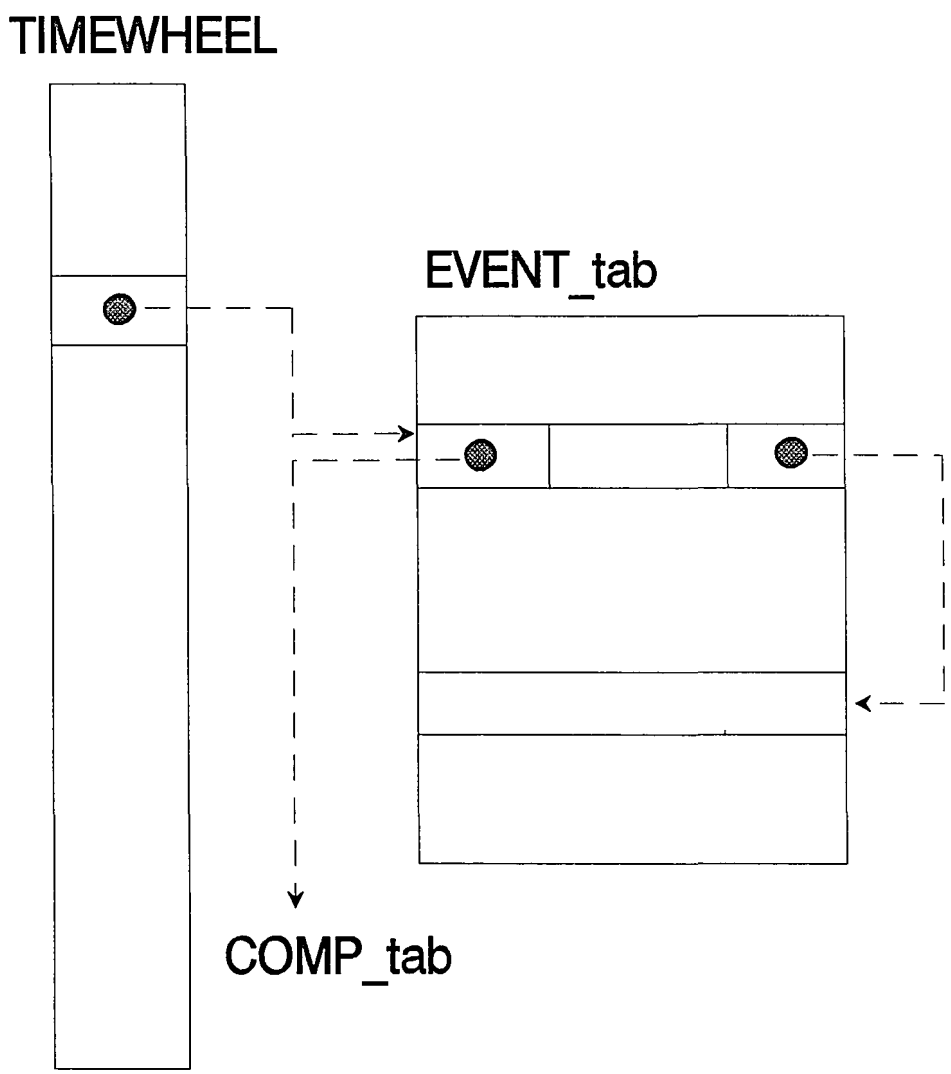

Fig. 8 Structure for Event List, mechanism for simulation. Timewheel is viewed as a circular table and controls the scheduling of events which are stored in the Event_tab. Each dot with a dashed line is a pointer connecting tables together.

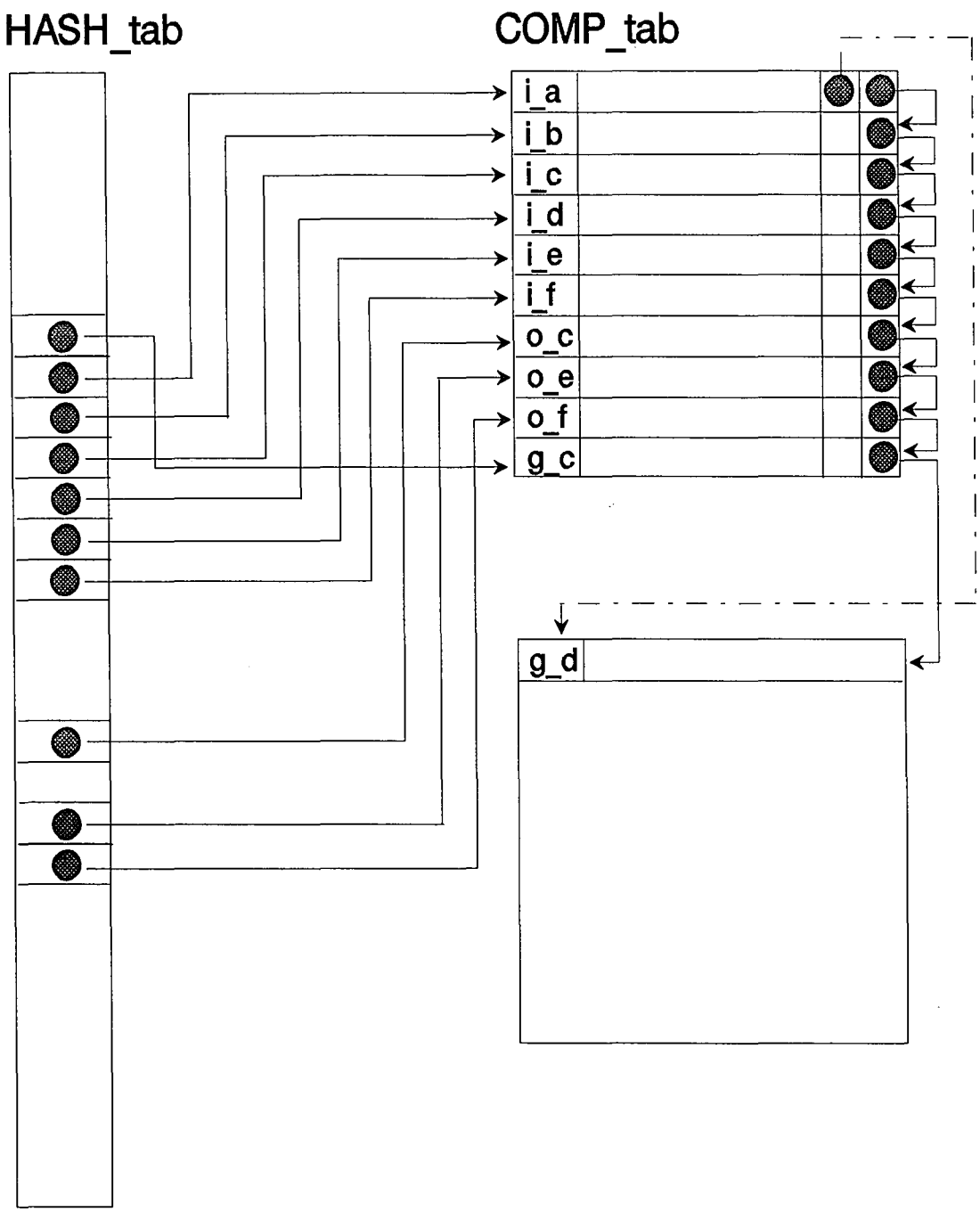

Fig. 9 Internal Image of HASH_tab and COMP_tab during simulation of demo of Fig. 3
Each dot in the HASH_tab is a pointer to an entry in the COMP_tab.
In the COMP_tab,
each dot with a solid line is a pointer connecting the entries for the COMP_tab.
each dot with a dashed line is a pointer connecting the entries for the HASH_tab.

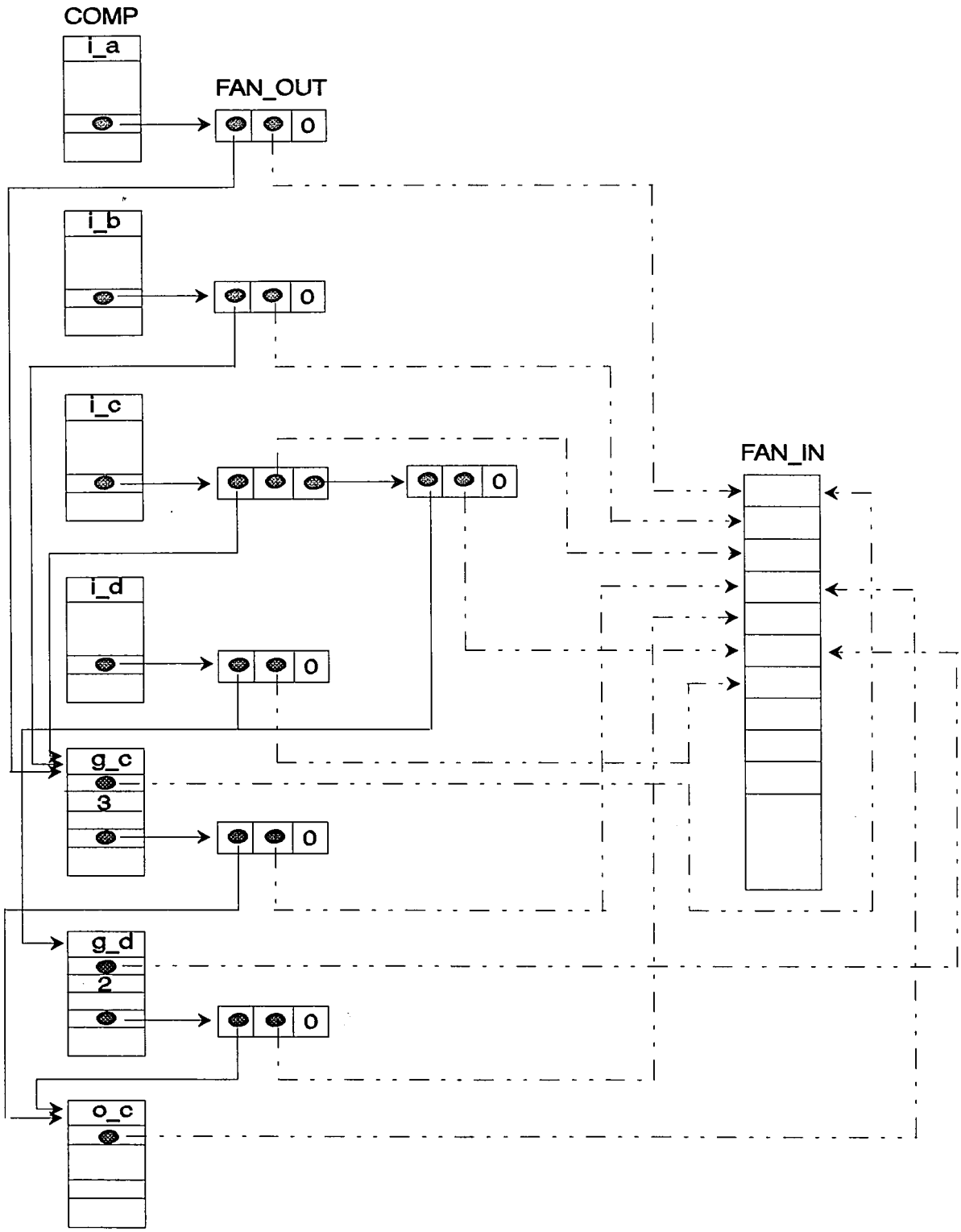

Fig. 10 Internal Image of COMP_tab, FAN_OUT_tab, & FAN_IN_Tab during simulation of demo of Fig. 3
In the COMP_tab,
the first entry, e.g. i_a, denotes the name of a component.
the dot with a solid line is a pointer to an entry in the FAN_OUT _tab.
the dot with a dashed line is a pointer to its first fan-in value in the FAN_IN_tab.
In the FAN_OUT_Tab,
the dot with a solid line is a pointer to an entry in the COMP_tab.
the dot with a dashed line is a pointer to an entry in the FAN_IN_tab.

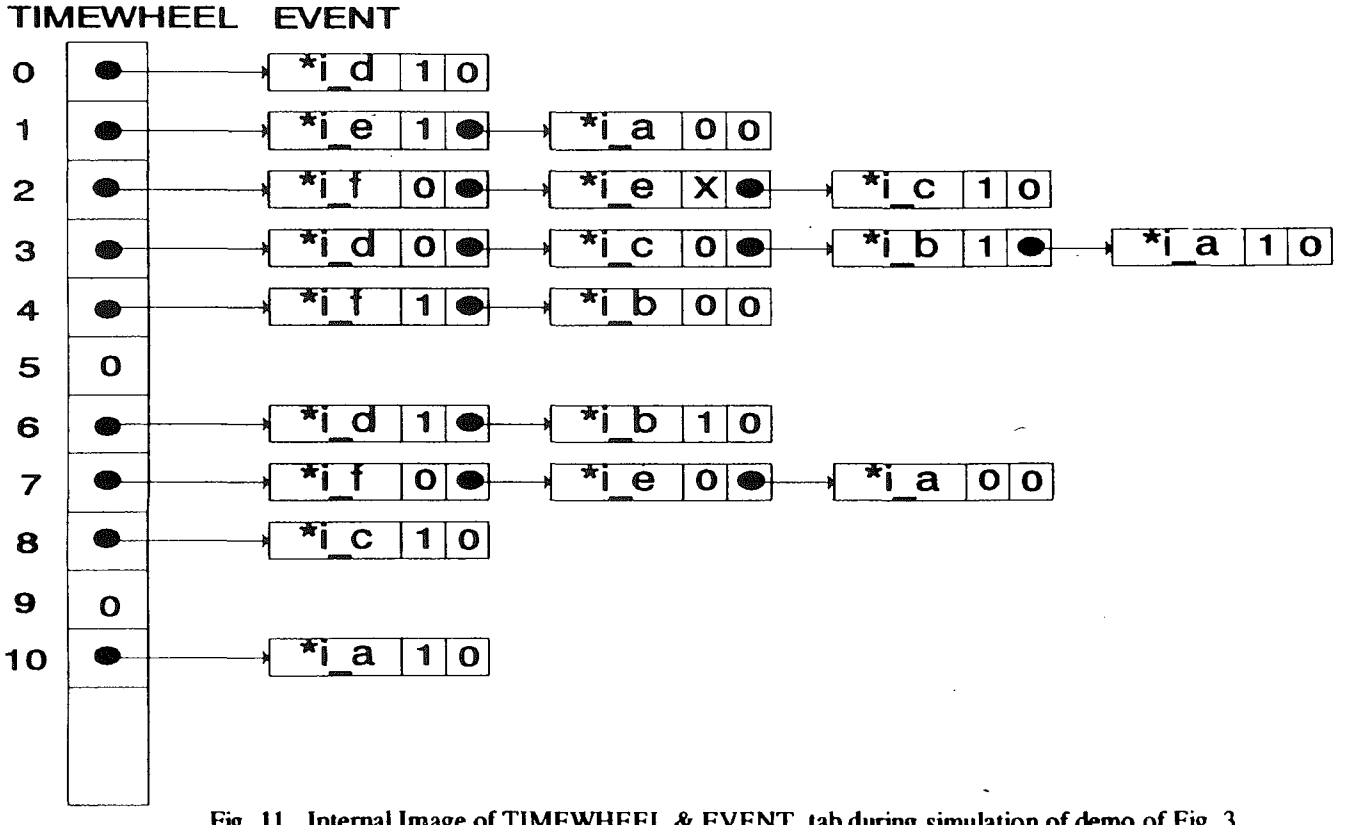

Fig. 11 Internal Image of TIMEWHEEL & EVENT_tab during simulation of demo of Fig. 3
Each dot in the TIMEWHEEL is a pointer to an entry in the EVENT_tab.
In the EVENT_tab,
the first entry, e.g. *i_d, denotes the component's address in the COMP_tab.
the second entry, e.g. 1, indicates the signal value.
the dot is a pointer to the next event in the same time slot.

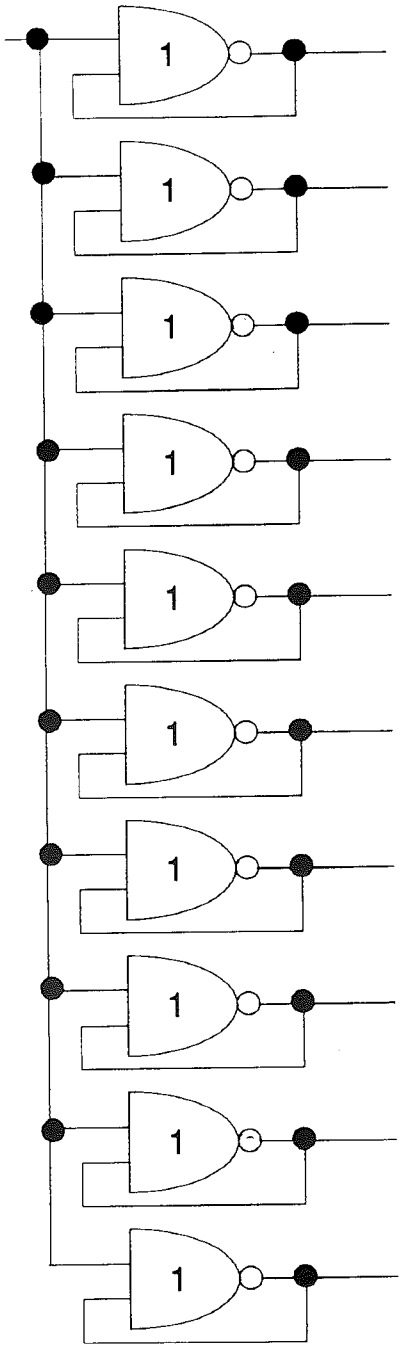

Fig. 12 Circuit test
Used as subject of simulation in benchmarking different evaluation techniques.

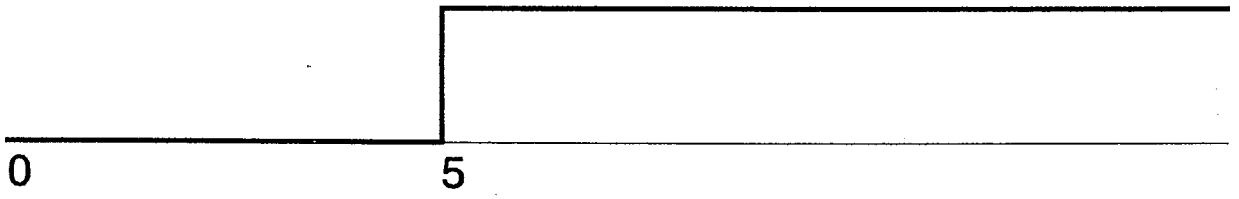

Fig. 13 Waveform for Input to Benchmark Circuit test of Fig. 12

V. References

Abramovici, M., Breuer, M.A., and Friedman, A.D.
Digital Systems Testing and Testable Design.
New York: W.H. Freeman & Co., 1990.

d'Abrea, M. "Gate-Level Simulation" IEEE Design and Test, Dec.1985, pp. 63-71.

Breuer, M.A. "A Note on Three-Valued Logic Simulation" IEEE Trans. on Computers, Vol. C-21, No. 4, April 1972, pp.399-402.

Breuer, M.A., and Parker A.C. "Digital System Simulation: Current Status and Future Trends" Proceedings 18th Design Automation Conference, 1981, pp. 269-275.

Brown, A. VLSI Circuits and Systems in Silicon
New York: McGraw-Hill, 1991.

Bryant, R.E. "MOSSIM: a Logic-Level Simulator for MOS LSI User's Manual" MIT VLSI Memo No. 80-21, July 1980.

Hitchcock, R.B. "Timing Verification and the Timing Analysis Program" Proceedings 19th Design Automation Conference, 1982, pp. 594-604.

Holt, D., and Hutchins, D. "A MOS/LSI Oriented Logic Simulator" Proceedings 18th Design Automation Conference, 1981, pp. 280-187.

Korsh, J.F., and Garrett, L.J. Data Structures, Algorithms, and Program Style Using C.
Boston: PWS-Kent Publishing Co., 1988.

Johnson, W.A. "Behavioral-level Test Development" Proceedings 16th Design Automation Conference, 1979, pp. 171-179.

Levendel, Y.H., Menon, P.R., and Patel, S.H. "Special-Purpose Computer for Logic Simulation Using Distributed Processing" Bell System Technical Journal, Vol. 61, No. 10, Dec. 1982, pp. 2873-2909.

Maczo, A. Digital Logic Testing and Simulation.
New York: John Wiley & Sons, 1986.

Ruehli, A.E. "Survey of Analysis, Simulation and Modeling for Large Scale Logic Circuits" Proceedings 18th Design Automation Conference, 1981, pp. 124-129.

Schuler, D.M. "Simulation of NAND Logic" Proceedings COMPCON 1972, Sep. 1972, pp. 243-245.

Szygenda, S.A. "TEGAS2 -- Anatomy of a General Purpose Test Generation and Simulation System for Digital Circuit" 25 Years of Electronic Design Automation, 1988, pp. 306-317.

Ulrich, E.G. "Time-Sequenced Logical Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths" Proceedings 20th ACM National Conference, 1965, pp. 437-448.

Ulrich, E.G. "Exclusive Simulation of Activity in Digital Networks" Communications of the ACM, Vol. 12, Feb. 1969, pp. 102-110.

VLSI editors "1987 Survey of Logic Simulators" VLSI Systems Design, Vol. 8, No. 2, Feb. 1987, pp. 71-86.

VI. Appendix

A sample session using *SIM*

```
$ sim
Please enter the netlist file: demo.net

Please enter the waveform file: demo.sti

Please list component(s) whose waveforms are to be stored. 3 options:
   list one by one,
     separated by space. List can span more than one line,
     but each component cannot. Terminate with a '.'.
   enter 'out': all OUTPUT components
   enter 'all': all components (INPUT, OUTPUT, and every component
                in between) (DEFAULT)
Enter: g_c g_d out

Please enter the file to store the waveforms: demo.17

If ready to simulate, enter 'run' followed by the number of
time units. If number not specified, the simulation will be run
till 19999 or when no events are left whichever comes first: run
event count: 35
$
```

VII. Biography

Ann Chen has worked for AT&T Bell Laboratories since 1979. Ann, currently a member of technical staff there, is working in software engineering for mask manufacturing. Ann received a Bachelor degree in Foreign Languages and Literature from National Taiwan University in 1973 and a Master degree in Business Administration from Indiana University in 1977. Ann was born in Taiwan in 1951.

END

OF

TITLE