Studies on Logic Simulation and Hardware Description Languages

Nagisa ISHIURA

December 1990

Studies on Logic Simulation and Hardware Description Languages

Nagisa ISHIURA

December 1990

Studies on Logic Simulation and Hardware Description Languages

Nagisa ISHIURA

Abstract

With the recent advances of semiconductor technologies, larger and larger and more and more sophisticated digital systems can be realized as hardware, which in turn makes it difficult to achieve *design verification* of hardware. In this thesis three major topics in logic design verification and hardware description languages are discussed. One is on *acceleration of logic simulation speed.* The increase in computation cost for logic simulation has been and will be the primary problems in design verification. As a solution to this problem, fast simulation methods utilizing vector supercomputers are proposed. Another topic is on *accuracy of logic simulation.* In verification of circuits which depends on subtle timing relations, trade-offs between accuracy and computation cost of simulation becomes an important issue. Discussions are created on this issue both from theoretical and practical point of view. The last topic is on *hardware description languages.* A new model of hardware which can be a base of formal semantics of hardware description languages is presented.

As for acceleration of logic simulation speed, fast logic simulation techniques utilizing *vector supercomputers* are proposed. Vector supercomputers are computers which have special facilities to execute operations on vectors extremely fast. In chapter 3 and chapter 4, new algorithms of logic simulation and fault simulation, respectively, are presented which efficiently bring out the potential of vector super computers.

In chapter 3, three types of simulation algorithms are proposed which are dedicated for 1) zero-delay simulation of combinational circuits, 2) zero-delay simulation of synchronous sequential circuits, and 3) simulation with delay consideration. The first two are based on the compiler-driven method. High vectorization ratio is achieved by processing many input patterns or simulating many gates at a time. Combinational use of the vectorization algorithms and the bit oriented vector logical operations makes it possible to achieve 7.7×10^9 gate-evaluations per second for combinational circuit simulation and 1.4×10^9 gate-evaluations per second for sequential circuit simulation on the supercomputer FACOM VP-200. The acceleration ratio through vectorization is more than 15. The third algorithm for timing simulation is an extension of the conventional event-driven simulation algorithm. Vectorization is achieved by processing all the events together which are scheduled to occur at the same time period. A simulator implemented based on the algorithm marked 230×10^3 events per second on the supercomputer HITAC S-810/20. These performance figures are comparable to those of hardware simulation engines.

In chapter 4, a vector supercomputer oriented fault simulation algorithm, named a *dynamic 2-dimensional parallel fault simulation* is proposed which is dedicated for zero-delay two-valued fault simulation of gate-level combinational circuits with single stuck-at faults. The *bit-parallel simulation technique* which is one of the basic algorithms of fault simulation is extended to *two-dimensional parallel simulation technique.* In this technique many faults for many patterns are processed at a time by vector bitwise logical operations. Although high vectorization ratio is achieved in this method, it does not necessarily lead to efficient fault simulation if we try to combine it with the fault dropping which is an indispensable technique for reducing the computation cost. In order to counter this problem, *dynamic adjustment* of the two parallelism factors is introduced. Experimental results on coverage estimation of random patterns are shown, in which the fault simulator implemented on the FACOM VP-200 supercomputer achieved acceleration ratio of 15 through vectorization and succeeded in simulating 500,000 random patterns on a circuit of 3,000 gates within 30 seconds.

As alternatives to logic simulators on general purpose computers, special purpose hardware for logic simulation and fault simulation have been developed, which achieves very high performance by parallel computation scheme. However, there are trade-offs between simulation speed and flexibility, or affinity for existing CAD systems on general purpose computers. The new approach of developing logic simulators and fault simulators on general purpose vector supercomputer is expected to be the one that fills the gap between software solutions on conventional scalar computers and hardware solutions.

As for accuracy of logic simulation, discussions are created focusing on a delay model under which delay values are not definite and are specified with their minimum and maximum values. At first the difficulty of logic simulation problem under the delay model is theoretically clarified, and then efficient algorithms to solve the problems are proposed.

Chapter 5 is dedicated for the theoretical consideration on modeling of delay and computational difficulty of a *hazard detection problem*. Relation among models of delay and time, accuracy of verification results and computation cost for the verification is discussed taking the hazard detection problem as an example. We also discuss the difference of a discrete time model and a continuous time model. It is shown that the problem of detecting hazards on combinational circuits under uncertain delay assumption is computationally intractable (NP-hard) and that it is hence difficult to solve the problem by a simple extension of the min/max delay simulation technique. It is also shown that there is an essential difference in the verification results obtained based on the discrete time model and the continuous time model. The verification result can be more optimistic in the discrete time model than in the continuous time model. Further discussions are created on the relation between the continuous time model and the discrete time model, in which a lower bound of the width between ticks that make the discrete time model equivalent to the continuous time model.

In chapter 6, a new simulation technique named *time-symbolic simulation* is presented which enables accurate simulation under the uncertainty delay model. The conventional min/max delay simulation techniques have been suffering from pessimistic results brought about by reconvergent fanouts. In time-symbolic simulation the uncertain delay value is expressed by a *variable*, which makes it possible to avoid the pessimism at reconvergent gates. Time-symbolic simulation also enables us to get conditions where the circuit under test behaves as expected, which is of good use for error analysis and for design improvements.

It is difficult to adapt conventional simulation algorithms to time-symbolic simulation. In this chapter, two efficient simulation algorithms for time-symbolic simulation are proposed. One is dedicated for combinational circuits and processes the algebraic formulas representing time by means of the *linear programming.* The other algorithm, which is named *coded time-symbolic simulation (CTSS)*, can handle any kind of gate-level logic circuits. In the CTSS an uncertain delay value is represented using a set of Boolean variables based on binary coding which encodes all the cases of delay values. Simulation is executed by means of *Boolean function manipulation.* Both of the simulators are shown to run within a feasible time for small scale circuits up to 100 gates. In this chapter, various techniques are also proposed for verification of asynchronous circuits based on time-symbolic simulation and for analysis of simulation results.

The importance of the symbolic approach in this chapter lies in that we can analyze the simulation result so as to get useful information on error correction and design improvements, as well as in that we can get accurate simulation results. Symbolic simulation, including time-symbolic and value-symbolic approaches, will be one of the most important techniques for logic design verification along with conventional logic simulation.

The last part of this thesis is dedicated for the discussion on formal semantics of hardware description languages. In the trend of *standardization*, definition of formal semantics of practical hardware description languages is an important issue. In order to define formal semantics of hardware description languages which can support various applications such as logic synthesis and formal verification as well as logic simulation, a model of hardware is indispensable which can express uncertain behavior of hardware in a strict since. In this thesis, a new behavior model of hardware named *NES* (Nondeterministic Event Sequence) is proposed. The NES can express the uncertainty of hardware behavior by means of *nondeterminism.* The behavior of hardware is modeled by nondeterministic abstract machines and is dealt with as a set of all the possible behaviors. In chapter 7, the formal definition of the NES model, and a modeling method of a hardware module and connected hardware modules are presented. Also as an application of the NES model, definition of the semantics of a hardware description language UDL/I is described.

The applications of hardware description languages will be wider and wider. The nondeterministic semantics, which plays an indispensable role in expressing the relations between behavior of circuits in different design levels, is considered to be an essential factor of hardware description languages of the next generation, which can be a basis of variety of applications such as logic synthesis, formal verification, symbolic simulation, and so on.

Contents

Chapter 1

Introduction

1.1 Backgrounds

Recent advances of semiconductor technologies have made it possible to realize large and sophisticated hardware as integrated circuits, which have brought digital systems to wide variety of applications. On the other hand, the size and the complexity of the hardware have made the design processes more and more difficult. It is almost impossible to design hardware of required scale and complexity without the help of *computer-aided design (CAD)* systems.

Among many steps of designing hardware, *design verification* is one of the most laborious ones. The most effective and the most widely used means of verifying correctness of design is *logic simulation*, that is to simulate the behavior of circuit under test on computers.

Computation time required for logic simulation is roughly proportional to the size of a circuit under test and to the length of test pattern sequence. The increase in circuit size, together with the incidental increase in test pattern size, has resulted in rapid growth of the computation time for the simulation. It is reported that about 1800 hours of IBM 370/168 CPU time were required to verify the logic design of 1/4 of a medium-range System 370 CPU [Den83]. It is one of the most important subject in the area of CAD of digital systems to develop high-speed logic simulation techniques. A number of research efforts have been carried out in recent years in order to reduce computation time for logic simulation, which include improvements in modeling of logic circuits, development of efficient simulation algorithms [Bre76, Ulr83, Ish84], improvements in techniques in coding level [Ulr80b, Kro81], and development of special purpose hardware (hardware simulation engines) [Den83, Sas83, Bla84, Nak86, Hir87, Nag86]. Among them the special purpose hardware approach has become a center of attention because of the high performance achieved by parallel computation schemes. However, the performance is obtained at the sacrifice of flexibility and affinity for existing CAD systems on general purpose computers. It is pointed out that it often takes much longer time to compile and to transmit data than to execute simulation on hardware simulation engines. High cost by reason of special purpose hardware is also a demerit of the hardware simulation engines. On the other hand, software simulators on general purpose computers are still attractive for their latest device technologies, economical merits and flexibility. General purpose supercomputers and parallel computers can be new solutions that fill the gap between the two approaches.

Along with the computation time for logic simulation, that of *fault simulation* is another big problem in the field of CAD of digital circuits. Fault simulation is in a way a variation of logic simulation, although it is used for different purposes from logic simulation. While a logic simulator computes behavior of fault-free logic circuits, a fault simulator computes behavior of logic circuits which have faults in them. It is used for analysis of the behavior of faulty circuits, test set generation or quality evaluation of test sets for logic circuits. Fault simulation requires much more computation cost than logic simulation, because simulation must be carried out for each of the faults derived from a certain fault model. Under the single stuck-at fault assumption, the computation time in the worst case is proportional to the square of circuit size [Har87]. Various research projects have been carried out in order to accelerate fault simulation by improving algorithms [Arm72, Ulr80a, Wai85, Nis85, Ant87], or to develop alternative techniques to fault simulation [Abr83, Jai84, Brg85]. In spite of these efforts, there are still pressing requirements for faster fault simulation.

The large computation cost due to the large circuit size has been and will be one of the primary problems in design verification. On the other hand, *accuracy* of simulation is also an important issue. Especially in design verification of asynchronous circuits which operate based on subtle timing relations, much more laborious modeling of delay and time and also much more computation cost are required than in that of synchronous circuits. In the verification concerned with timing there are close relations among models of delay and time, accuracy of verification results and required computation cost. In a simple modeling which require smaller computation cost, design errors may be overlooked or possibilities of design errors may be indicated even for correct designs. One example is the handling of delay whose actual value is unknown and is specified with minimum and maximum values. In logic simulation the min/max delay model is employed to handle such uncertainty. The model allows relatively fast verification but it is well known that the verification results are often too pessimistic due to reconvergent fanouts [Bre76]. It has, therefore, come to be an important research theme to find efficient methods to overcome this problem [Yon89, Cer89]. Although there are many attempts to solve the problems, few discussions have been made on what is the essence of the difficulty and how difficult or how much computation cost is required to solve the problem completely. Another important issue is modeling of time. Many of the existing verification systems are based on a *discrete time model* [Cer89, Hir89, Nak87, Kim88]. There are also few discussions on the point if the discrete time model provides accurate result as compared with a *continuous time model* or if there is a difference in the computation cost of the verification between the two models. In order to develop efficient and yet reliable verification system, it is considered to be important to clarify the theoretical backgrounds on modeling and accuracy of verification.

Modeling and accuracy of verification are also one of the central issues in the field of *hardware description languages (HDL's).* Hardware description languages are kernels of CAD systems for integrated circuits which work as inputs to various CAD tools, design documents and vehicles for design interchange among different CAD systems. Although a lot of research projects have been carried out on hardware description languages, we are now confronted with a big turning point due to two trends; *standardization* and *extension of the applications* of HDL's.

Standardization of a hardware description language (HDL) has an inestimable impact on the development of hardware design, including CAD tool development and design education. There are several activities for standardization in the U. S., Europe, and Japan [Kar89, Pil83, Coe89, Har86]. Since a standard HDL is used by many users, including IC manufactures and tool developers working in various kinds of design culture, we should provide them with a method of sharing a detailed idea on the HDL. It is therefore essential to define rigid syntax and semantics of the language. Although almost all the HDL's are designed on the basis of the formal definition of syntax by a meta language like BNF, there are very few HDL's, especially among the practical ones, which has clear definition of semantics. Although there have been a lot of researches on definition of formal semantics in the area of programming languages [Bjo78]. There have been, however, few studies in the area of HDL's other than [Pil83]. Especially there have been no established models which explain the behavior of the hardware described in HDL's. In view of the trend of standardization, it is considered to be an urgent research theme to develop good behavior models for HDL's and to establish formal methods for defining semantics of HDL's.

Extension of the applications of HDL's is also changing the situation. For many years logic simulation has been the most important application of HDL's. In practical situations semantics of an HDL is defined by means of the simulator for the HDL. However, recent researches in the area of CAD for integrated circuits have brought about outstanding development of techniques for various design support by computers. Especially *logic synthesis* and *formal verification* come to become a practical technique and there are strong demands for HDL's to support these applications. However, the simulation based semantics often causes inconsistencies in handling *don't cares* and *uncertain behavior of hardware.* In logic synthesis and formal verification, we assume all the possibilities for don't cares and uncertain hardware specifications. On the other hand, in logic simulation, they are dealt with using *unknown values.* This is inevitable if we consider efficiency of simulation execution but it often brings about unnatural results. It is considered to be an essential challenge to develop a formal model which can explain the don't cares and uncertain behavior of hardware in order to design hardware description languages of the next generation which are provided with rigid semantics and can be basis of various CAD applications.

1.2 Outline of the Thesis

In this thesis three major topics in logic design verification and hardware description languages are discussed; acceleration of logic simulation speed, accuracy of timing verification, and modeling of hardware behavior for formal semantics of hardware description languages.

(1) Algorithms for high-speed logic simulation

As a new approach to accelerating execution speed of logic simulation, a use of *vector supercomputers* are proposed. Vector supercomputers are the computers that have special facilities to execute operations on vectors extremely fast. As is discussed in 1.1, there are trade-offs between speed performance and flexibility of high-end simulators. Logic and fault simulators on supercomputers are considered to be new a solution that fall between hardware logic simulators and software simulators on general purpose computers. In order to bring out the performance of vector supercomputers, vector processor oriented algorithms for logic and fault simulation are proposed in chapter 3 and chapter 4, respectively.

In chapter 3, three types of simulation algorithms are proposed which are dedicated for 1) zero-delay simulation of combinational circuits, 2) zero-delay simulation of synchronous sequential circuits, and 3) simulation with delay consideration. The first two are based on the compiler-driven method. High vectorization ratio is achieved by simulating many input patterns or processing many gates at a time. Combinational use of the vectorization algorithms and the bit oriented vector logical operations made it possible to achieve 7.7×10^9 gate-evaluations per second for combinational circuit simulation and 1.4×10^9 gate-evaluations per second for sequential circuit simulation on the supercomputer FACOM VP-200, which are faster by a factor of more than 15 as compared with conventional scalar processors. The third algorithm for timing simulation is an extension of the conventional event-driven simulation algorithm. Vectorization is achieved by processing all the events together which are scheduled to occur at the same time period. A simulator implemented based on the algorithm marked 230×10^3 events per second on the supercomputer HITAC S-810/20. These performance figures are comparable to those of hardware simulation engines.

In chapter 4, a vector supercomputer oriented fault simulation algorithm, named *dynamic 2-dimensional parallel fault simulation* is proposed which is dedicated for the zero-delay two-valued fault simulation of gate-level combinational circuits with single stuck-at faults. The *bit-parallel simulation technique* which is one of the basic algorithms of fault simulation is extended to *two-dimensional parallel simulation technique*, in which many faults for many patterns are processed at a time by vector bitwise logical operations. Although high vectorization ratio is achieved in this method, it does not necessarily lead to efficient fault simulation if we try to combine it with the fault dropping which is an indispensable technique for reducing the computation cost. In order to counter this problem, *dynamic adjustment* of the two parallelism factors is combined with the two-dimensional parallel simulation technique. Experimental results on coverage estimation of random patterns are shown, in which the fault simulator implemented on the FACOM VP-200 supercomputer achieved acceleration ratio of 15 through vectorization and succeeded in simulating 500,000 random patterns on a circuit of 3,000 gates within 30 seconds.

(2) Accuracy of logic simulation

Accuracy of logic simulation is discussed both from theoretical and practical point of view, focusing on a delay model in which actual delay values are not definite and are specified with their minimum and maximum values. At first the difficulty of the problems is theoretically clarified, and then efficient algorithms to solve the problems are proposed.

Chapter 5 is dedicated for the theoretical consideration on modeling of delay and computational difficulty of a *hazard detection problem*. Relation among models of delay and time, accuracy of verification results and computation cost for the verification is discussed taking the hazard detection problem as an example. We also discuss the difference of a discrete time model and a continuous time model. It is shown that the problem of detecting hazards on combinational circuits under uncertain delay assumption is computationally intractable (NP-hard) and that it is hence difficult to solve the problem by a simple extension of the min/max delay simulation technique. It is also shown that there is an essential difference in the verification results obtained based on the discrete time model and the continuous time model. The verification result can be more optimistic in the discrete time model than in the continuous time model. Further discussions are created on the relation between the continuous time model and the discrete time model, in which a lower bound of the width between ticks that make the discrete time model equivalent to the continuous time model.

In chapter 6, a new simulation technique named *time-symbolic simulation* is presented which enables accurate simulation under the uncertainty delay model. In time-symbolic simulation the uncertain delay value is expressed by a *variable*, which makes it possible to avoid the pessimism at reconvergent gates. Time-symbolic simulation also enables us to get conditions where the circuit under test behaves as expected which is of good use for error analysis and for design improvements.

It is difficult to adapt conventional simulation algorithms for time-symbolic simulation. In this chapter, two efficient simulation algorithms for time-symbolic simulation are proposed. One is dedicated for combinational circuits and processes the algebraic formulas representing time by means of the *linear programming.* The other algorithm, which is named *coded time-symbolic simulation (CTSS)*, can handle any kind of gate-level logic circuits. In the CTSS a uncertain delay value is represented using a set of Boolean variables based on binary coding which encodes the all the cases of delay values. Simulation is executed by means of *Boolean function manipulation.* Both of the simulators are shown to run within a feasible time for small scale circuits up to 100 gates. In this chapter, various techniques are also proposed for verification of asynchronous circuits based on time-symbolic simulation and for analyzing simulation results and extracting delay conditions where the circuit under test behaves correctly.

(3) Modeling and description of logic circuit for HDL's

In order to define formal semantics of hardware description languages which can support various application such as logic synthesis and formal verification as well as logic simulation, a model of hardware is indispensable which can express uncertain behavior of hardware in a strict sense. In the last part of the thesis, a new behavior model of hardware named *NES* (Nondeterministic Event Sequence) is proposed. The NES can express the uncertainty of hardware behavior by means of *nondeterminism*. The behavior of hardware is modeled by nondeterministic abstract machines and we can deal the behavior of hardware as a set of all the possible behaviors.

In chapter 7, the formal definition of the NES model, and a modeling method of a hardware module and connected hardware modules are presented. Also as an application of the NES model, definition of the semantics of a hardware description language UDL/I is described.

Chapter 2

Logic Simulation

2.1 Modeling of Logic Circuits for Logic Simulation

2.1.1 Modeling of Structure of Logic Circuits

In this thesis we mainly discuss *gate-level* logic circuits. Structure of a logic circuit is modeled by a *directed graph*. The nodes in the graph represent the primary inputs, the primary outputs and the gates of the circuit, and the edges represent connections among them. A primary input is represented by a nodes whose in-degree and out-degree is 0 and 1, respectively, while a primary output by a node whose in-degree and out-degree is 1 and 0, respectively. A node that represents a gate has n incoming edges and 1 outgoing edge. A function and delay are defined for a gate. We do not define them formally here, because it depends on the modeling of signal values, time and delay. We will refer to a circuit corresponding to an *acyclic* graph as a *combinational circuit.*

2.1.2 Modeling of Signal Values

In gate-level modeling of logic circuits, at least two signal values *0* and *1* are necessary so as to represent *logical zero* and *logical one*, respectively. The model of signal values which deals with the two values is called *2-valued logic model* and logic simulation based on the 2-valued logic model

		0	1
AND	0	0	0
	1	0	1
NOT		1	0

(a) 2-valued.

		0	1	X
AND	0	0	0	0
	1	0	1	X
	X	0	X	X
NOT		1	0	X

(b) 3-valued.

		0	1	X	Z
AND	0	0	0	0	0
	1	0	1	X	1
	X	0	X	X	X
	Z	0	1	X	1
NOT		1	0	X	0

(c) 4-valued.

Fig. 2.1 Operation tables of AND and NOT operations.

is called *2-valued logic simulation.* Fig. 2.1 (a) is the operation tables of AND and NOT operations in the 2-valued logic.

Although the 2-valued logic model may be sufficient for dealing with ideal logic circuits, we introduce in the practical logic simulation the following signal values for the purpose of modeling phenomena which are difficult to explain based on the 2-valued logic model or for the purpose of accelerating simulation execution. For the details, refer to [Bre76].

Unknown value: It is introduced in order to represent signal values *which are not definite* and usually denoted as 'X' or 'U'. The unknown values are used in the following situations, for example.

1) Signal values on uninitialized signal lines (especially outputs of uninitialized flip-flops).
2) Output values of gates in response to illegal combinations of input values.
3) Output values of gates in response to *don't care* input values.
4) Signal values which are not logical 0 nor logical 1; in transition from one to the other, or in other states which can not be interpreted as logical 0 nor logical 1.

5) Signal values which can be 0 and 1 depending on delay values of gates.

6) Signal values which can be indefinite because of the design errors.

The model which deals with 0, 1, and unknown value is referred to as a *3-valued logic model.* Fig. 2.1 (b) is the operation tables for the 3-valued logic model. Although the unknown values provide many conveniences, they sometimes cause undesirable results. One of the problems is that the unknown values are used in so many contexts that they lead to unexpected simulation results. There should have been a clear distinction between the signal values which are not logical values and the ones which are indefinite but are logical values. It is pointed out that simulation results tend to be *pessimistic* in the current simulation techniques based on the 3-valued logic model, because it is based on the calculus in which $X + \overline{X}$ comes to X instead of 1. We will discuss the modeling of indefinite signal values in detail in chapter 7.

High-impedance value: It is introduced to express the output of tristate gates and usually denoted as 'Z'. The signal value system consisting of 0, 1, X(unknown) and Z is used in many logic simulators and is called a *4-valued logic model.* The definition of the operation results associated with Z is dependent on the technologies that realize logic gates. Fig. 2.1 (c) is an example of the operation results.

Transient values: They are introduced to distinguish rising and falling signals from erroneous states. They are denoted by 'R' and 'F', respectively, and often used in combination with the min/max delay model mentioned in the next subsection.

In addition, many kinds of signal values which indicate the possibilities of undesirable signals which may caused by various design errors, such as hazards, signal conflict and so on.

On the other hand, as new attempts to extend logic simulation to formal verification, signal values represented by sets [Kim88, Ish90y] or Boolean functions [Car79, Cor81] have been proposed. Logic simulation based on the signal values represented by Boolean functions is called *symbolic simulation.*

2.1.3 Modeling of Time

In gate-level logic simulation, we usually model time by a *totally ordered set* T. If $T = \mathcal{Z}$ (the set of integers) the time model is called *discrete time model* and if $T = \mathcal{R}$ (the set of real numbers) the model is called *continuous time model.* Most of the simulation algorithms and most of the existing logic simulators assume the discrete time model. A time period ticked by an integer is called a *unit time.*

On the other hand, there are approaches in which time is modeled by a *partially ordered set*[Sta85, Tes87]. This model is suitable for representing causality relationship or before-after relationship among events occurring in a circuit and suitable for modeling higher level design.

2.1.4 Modeling of Delay

Most of the existing delay models are based on the discrete time model. Followings are the classification of the delay models which are relevant to the discussions in this thesis.

Zero delay model: It is a model in which all the delay values of the gates are *zero.* Here delay of zero means that the delay time is less than one if measured by the unit time but that there is causality relationship. This model is used when we are not interested in the timing issues on the circuit. It will cause difficulties in simulating logic circuits with feedback loops. Actually delay whose value is zero often causes undesirable situations. We will discuss the issue in chapter 7.

Unit delay model: It is a model in which all the delay values of the gates are *one*.

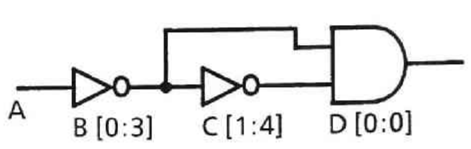
Assignable delay model: It is a model in which arbitrary delay values can be defined for each gate. it is further classified according to what type of delays are assigned.

Nominal delay model: A single delay value is assigned to a gate.

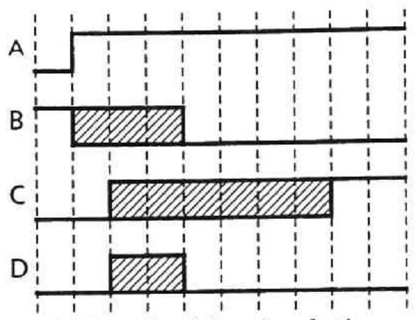
Rise/fall delay model: Two delay values are defined to a gate, which represent delay values for $0 \rightarrow 1$ transitions and for $1 \rightarrow 0$ transitions.

Min/max delay model: In actual logic circuits, delay values of gates vary depending on the difference of process conditions or usage conditions. In order to express this uncertainty the delay value is specified by its minimum and maximum values d^{min} and d^{max}, respectively. When a signal change occurs at time t, the output signal value at time between $t + d^{min}$ and $t + d^{max}$ becomes X (unknown) or transient signal values, as shown in Fig. 2.2. Although this model is based on the very realistic assumption, it has been pointed out that this model has serious shortcomings such that simulation results are often too pessimistic due to reconvergent fanouts [Bre76]. For example, in Fig. 2.2, the unknown states on the output of D indicate the possibility of a static hazard, which never occurs in an actual circuit. We will discuss this issue in detail in chapter 5 and chapter 6.

In actual circuits realized as integrated circuits, delay values of connections among gates are much large than those of gates. In this thesis,



(a) An example circuit.



(b) Result of the simulation.

Fig. 2.2 Min/max delay simulation.

however, we assume delay only on gates because the delay of a connection can be expressed by inserting a buffer gate with the delay.

2.1.5 Logic Simulation

Logic simulation is to compute the signal value sequences on the primary outputs for given descriptions of a logic circuit and signal value sequences on the primary inputs. Formal definition of logic simulation depends on models of signal value, time and delay. We show a definition of the case of the 2-valued logic, discrete time and assignable nominal delay model, as an example.

A sequence of signal values on node n_i, which represents a primary input, a primary output or a gate, is modeled as mapping $v_i : \mathcal{P} \rightarrow \mathcal{B}$, where $\mathcal{P}$ is the set of the non-negative integers and $\mathcal{B} = \{0, 1\}$. Let n_i^1, n_i^2, $\cdots$, $n_i^{m_i}$ be nodes which are the direct predecessor of node n_i and v_i^1, v_i^2, $\cdots$, $v_i^{m_i}$ be the signal value sequences on n_i^1, n_i^2, $\cdots$, $n_i^{m_i}$, respectively. Function f_i and delay value d_i are defined for node n_i, where $f_i : \mathcal{B}_i^m \rightarrow \mathcal{B}$ and $d_i \in \mathcal{P}$. As for nodes representing primary outputs, assume the identity function as f_i. Then the following relation holds for each node n_i.

$$v_i(t - d_i) = f_i(v_i^1(t), v_i^2(t), \cdots, v_i^{m_i}(t)). \tag{2.1}$$

Logic simulation is to compute v_i for each of the primary output n_i for given sequences of signal values on the primary outputs.

2.2 Basic Algorithms for Logic Simulation

2.2.1 Compiler-Driven Simulation and Event-Driven Simulation

We can classify the algorithms of logic simulation into *compiler-driven* methods and *event-driven methods.* The compiler-driven method is a simple algorithm in which logic simulation is executed by computing at each time the formula (2.1) for each node. We refer to the computation of the Boolean function of a gate to an *evaluation* of the gate. We have to pay attention to the order of gate evaluation. In the case of combinational circuits, the order of gate evaluation is usually determined before simulation execution (the details are described in subsection 2.2.4). If a given circuit has feedback loops, it happens that the correct signal value of a node at a certain time is not properly computed by simply evaluate each gate once. In such cases, we usually continue computation until the signal values are stable. If the computation does not stop because of oscillations, simulator detects this and outputs error messages.

Although the compiler-driven simulation algorithm is simple and easy to implement, we are forced to evaluate gates whose input values are the same as the previous values. Signal values do not change so often; the ratio of the signal change is 1~10% or less than that. The event-driven simulation algorithm attempts to reduce the computation cost by evaluating only the gates whose input values are different from those at the previous time period. A change of a signal value is called an *event.* Although extra cost is required to manage events, event-driven simulation is more efficient than compiler-driven simulation especially in handling assignable delay models.

Simulation speed is measured in terms of *gate evaluations per second* in compiler-driven simulation and *events per second* in event-driven simulation.

2.2.2 S-Algorithm and T-Algorithm

We can classify algorithms of logic simulation also from the standpoint if they are based on *space first evaluation* or *time first evaluation* [Ish84, Ish85yy]. The space first evaluation is a strategy in which time is advanced after finishing all the necessary evaluation of gates at a time frame, while time first evaluation is a strategy in which all the possible computation on a gate is performed in time direction for each gate (see Fig. 2.3). We abbreviate an algorithm based on the space first evaluation to *S-algorithm* and on time first evaluation to *T-algorithm.* We can consider four types of algorithms for the combinations of the two strategies in the previous subsection (compiler-driven and event-driven) and the two strategies in this section (S-algorithms and T-algorithms).

(1) Compiler-driven simulation based on the S-algorithm

We assume that the order of gate evaluation is determined before simulation by a method described in 2.2.4. We also assume that time begins with 0 and the final time at which we stop simulation is given.

1) Repeat 2) for each time from 0 to the final time.

2) Repeat 3) until signal values of all nodes become stable.

3) Evaluate gates in the predetermined order.

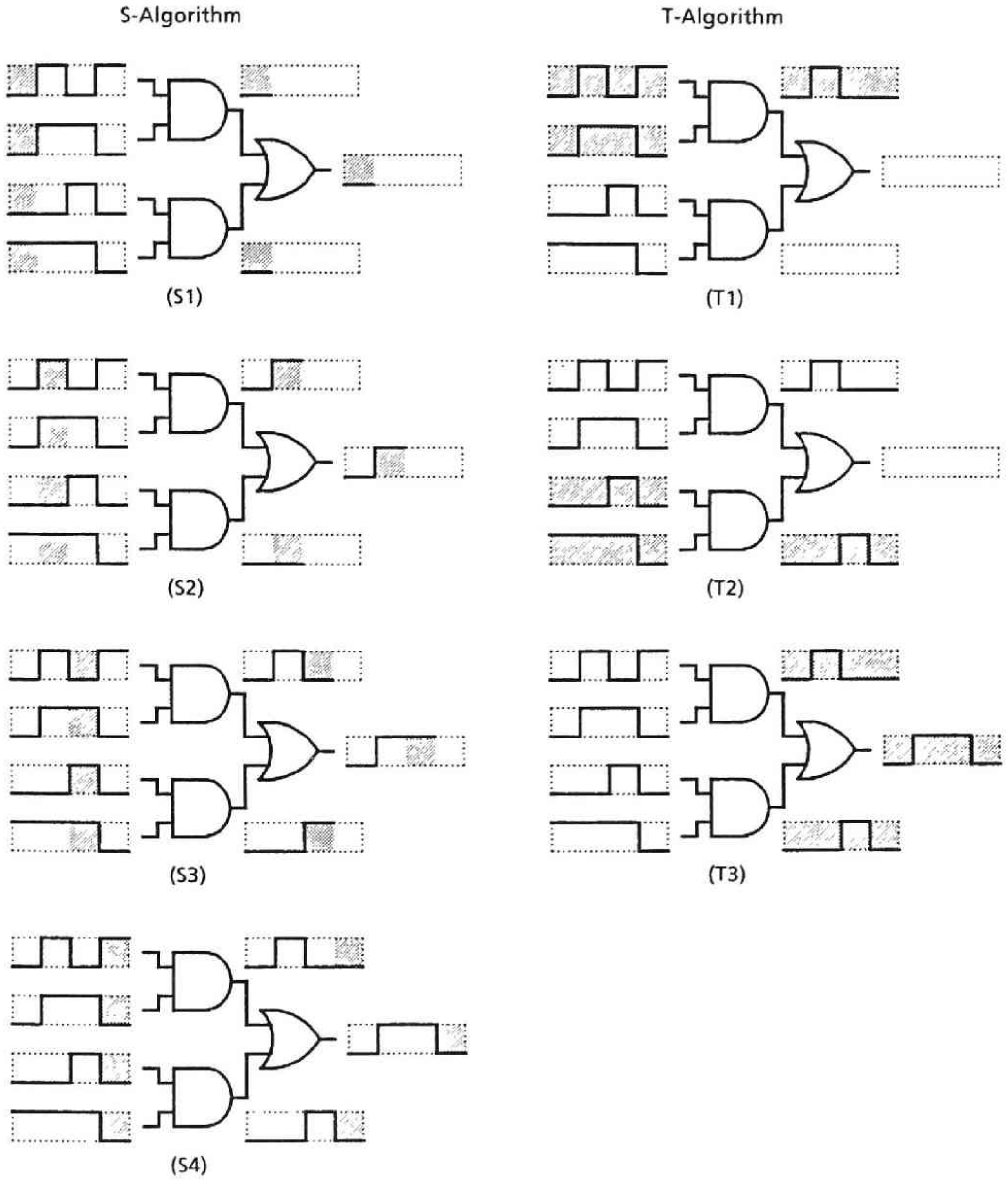


Fig. 2.3 S-algorithm and T-algorithm.

Note that in the case of combinational circuit, the signal values become stable by the first execution of 3) if gates are evaluated in the order described in 2.2.4.

(2) Event-driven simulation based on the S-algorithm

Events whose occurrences are known are maintained in set S.

1) Repeat 2) for each time t from 0 to the final time.

2) Take out events whose occurrence time is t and perform 3) for each event.

3) Evaluate the gates which are affected by the event. If there are signal changes, generate new events and include them in S.

In order to maintain events, we use a data structure called *time wheel*, which consists of linear lists of events and an array of headers to the linear lists. Each list consists of the events which are know to occur at the same time frame. This data structure is suitable for efficient extraction of events in 2) and registration of events in 3).

(3) Compiler-driven simulation based on the T-algorithm

We assume that the order of gate evaluation is predetermined.

1) Repeat 2) until all the signal values become stable.

2) Repeat 3) for each gate in the predetermined order.

3) Perform gate evaluation for all the possible time frames.

Note that in the case of combinational circuit, the signal values become stable by the first execution of 2).

(4) Event-driven simulation based on the T-algorithm

In this algorithm [Ish84, Ish85yy], a sequence of signal values are represented by a linear list of events, where an event is a tuple of time and a signal value.

1) Execute 2) for each gate in the predetermined order.

2) Repeat 3) $\sim$ 4) until there is no processed event at input lines.

3) Among the events which may occur next at input lines, select an event whose occurrence time is the smallest, and compute the effect of the event.

4) If the output value of the gate changes as a result of 4), add an event to the output line whose occurrence time is a sum of occurrence time of the input event and the delay of the gate.

Most of the existing simulators are based on S-algorithms because it is difficult to deal with circuits with feedback loops in T-algorithms. However, T-algorithms are attractive in dealing with combinational circuits because they are simple and much more efficient than S-algorithms in simulating combinational circuits. Espacially, it was discovered independently by [Bar87, Koe86, Ish87] that combination of T-algorithm based compiler-driven simulation and *bit-parallel* gate evaluation technique [Bre76] is a very efficient simulation technique for combinational circuit. It is also shown that event-driven simulator based on the T-algorithm is 7 $\sim$ 8 times faster than that on the S-algorithm [Ish84, Ish85yy].

2.2.3 Code Generation Method and Table-Driven Method

The following two methods are know as ones for implementing logic simulators.

Code-generation method: Generate a code (for computers) to realize gate evaluation and perform simulation by executing the code.

Table-driven method: Generate tables containing information necessary for simulation such as an order of gate evaluation, the kind of each gate and connections among the gates. Perform simulation by referring the table. In the compiler-driven simulation, this method can be regarded as an *interpreter*-driven method.

Generally the code-generation method enables faster simulation execution than table-driven method. The compiler-driven method has good affinity for code generation method. Actually, the compiler-driven simulators in the early times were implemented by this method which is the origin of the name *compiler-driven* method. On the other hand, it is considered to be difficult to implement event-driven simulators based on the code-generation method.

2.2.4 Ordering of Gate Evaluations

In compiler-driven simulation and T-algorithms, an order of gate evaluation is important because we can dispense with futile gate evaluation if we choose a good order. Furthermore it causes a big difference in the storage requirement for simulation as is discussed in chapter 3.

The most popular method of ordering is the one called *level sorting* [Bre76]. In this method gates are ordered according to *level numbers* of gates. The level number $l(n_i)$ of gate n_i is defined as follows based on the notation in the previous section.

$$l(n_i) = \begin{cases} 0 & \text{if } n_i \text{ is a primary input} \\ 1 + \max\{l(n_i^1), l(n_i^2), \cdots, l(n_i^{m_i})\} & \text{otherwise.} \end{cases}$$

For example, in Fig. 2.4, $A - B - C - D - E - F - G$ and $B - C - A - E - D - F - G$ are obtained in this ordering.

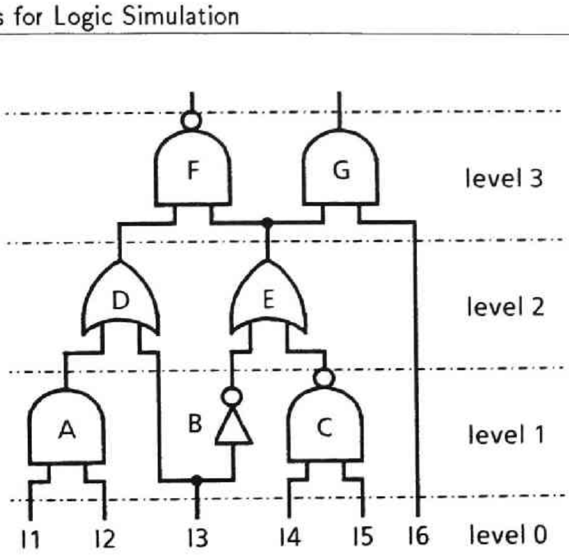


Fig. 2.4 Ordering of gate evaluation.

As a generalization of level sorting, we consider *data flow sorting (DF-sorting)* [Ish87, Ish86]. An *available gate* is a gate whose all input values have already been computed. Let S be the set of available gates. Then the order of gate evaluation is determined by the following procedure.

1) At the beginning, include the gates with its all inputs connected to the primary inputs in S.

2) Repeat 3) and 4) until S becomes empty.

3) Get gate g out of S and evaluate g.

4) After the evaluation of g, if any gates become newly available, add them to S.

By DF-sorting we can get any of the all orders that yield correct simulation result. In Fig. 2.4, $A - D - B - C - E - F - G$ and $B - C - E - G - A - D - F$ are the examples of order obtained by DF-sorting but not by level sorting. The DF-sorting provides us a higher degree of freedom to allow wide choice in determining the gate evaluation order than level sorting. We will use this freedom for reducing the storage requirement for simulation in chapter 3 and chapter 4.

Chapter 3

Fast Logic Simulation Using Vector Super Computers

3.1 Introduction

Computation time required for logic simulation is roughly proportional to the size of a circuit under test and to the length of test patterns. The increase in the circuit size, together with the incidental increase in the test pattern size, has resulted in the rapid growth of the computation time for the simulation. It is one of the most important subjects in the area of Computer-Aided Design (CAD) for Very Large Scale Integration (VLSI) to develop high-speed logic simulation techniques. In this chapter, we propose logic simulation techniques using vector processors, as a new approach to accelerating simulation speed.

In order to reduce simulation time, a number of research efforts have been carried out in recent years. These approaches can be roughly classified into the following kinds.

1) Contrivance on the circuit modeling: The function level modeling and the function level simulation, for example, bring forth the drastic reduction in computation time and in storage requirements (in exchange for a precision, however).

2) Improvements in simulation algorithms: The concurrent simulation technique [Ulr83], clock suppression [Ulr83] and time first evaluation algorithm (T-algorithm) [Ish84, Ish85yy] are such examples.

3) Techniques in coding: Zoom table look-up [Ulr80b] is one of the most popular coding techniques. The vector coding technique for a scalar processor by Krohn [Kro81], though it is in some way analogous to our approach, also falls under this category.

4) Development of special purpose hardware (hardware simulation engines): Several types of hardware simulation engines have been designed and some of them (YSE by IBM [Den83], HAL by NEC [Sas83], and Logic Evaluator by ZYCAD [Bla84], VELVET by Hitachi [Nag86], SP by Fujitsu [Hir87], for example) have actually been implemented and are in practical use.

Using parallel computation schemes, the hardware approach has demonstrated the potential to improve performance by a factor of 1000 over current software solutions. However, software simulators may be as fast as hardware simulators, if the power of the fastest computers, or super computers, can be efficiently harnessed. Furthermore, software is generally more flexible and portable than hardware. From an engineering standpoint, there are many benefits to developing a software simulation method with performance comparable to that of current hardware solutions.

A vector processor is a supercomputer which has the facility to execute operations on vectors extremely fast. Several vector processors have been developed in recent years [Lub85], which are capable of executing several hundred MFLOPS (Million FLoating-point Operations Per Second). Vector processors are developed with a view to accelerating the numerical computation for problems that require enormous computation power. In addition to the powerful facilities for floating-point operations, they have vector operations so versatile that we can use their computation power in many applications.

We must note that all programs cannot enjoy the benefit of vector processors. High vectorization ratio, and yet long vector length are essential for efficient computation. We must tune up our coding schemes or basic algorithms to be suitable for vector processing.

In this chapter, we propose new high-speed logic simulation techniques which efficiently utilize the computation power of vector processors [Ish85ykv, Ish85yki, Ish86, Ish87]. We have developed 3 types of simulation techniques which are dedicated for:

1) zero-delay simulation of combinational circuits,

2) zero-delay simulation of synchronous sequential circuits, and

3) simulation with delay consideration.

The first two are based on the compiler-driven method and the last on the event-driven method.

For the simulation of combinational circuits, we propose vector-parallel simulation technique (VP-technique), which is based on the time first evaluation algorithm (T-algorithm) [Ish84, Ish85yy]. A sequence of states on a signal line is treated as a vector (a pattern vector) and the gate evaluation is performed by vector logical operations on pattern vectors. For the simulation of synchronous sequential circuits, the simulation procedure is vectorized by a gate grouping technique (GG-technique), which is based on the space first evaluation algorithm (S-algorithm). In this case, we increase the vectorization ratio by grouping gates of the same kind and evaluating them together in a vectorized manner.

In order to carry out the timing simulation with sophisticated delay models, we have also developed a vectorization technique for event-driven simulation. The algorithm we adopted is basically the conventional event-driven method with a time mapping technique. The procedures for event fetch, event propagation, gate evaluation and event registration are vectorized by processing all the events together which are scheduled to occur at the same period. Data structures and the operations on events are modified to be suitable for vector processing.

We have implemented logic simulators based on the above simulation techniques on the FACOM VP-100 and VP-200 (266 MFLOPS and 533 MFLOPS, respectively) at Kyoto University and the HITAC S-810/20 (630 MFLOPS) at the University of Tokyo. The maximum performance is about 7.7×10^9 gate-evaluations per second for combinational circuit simulation, 1.4×10^9 gate-evaluations per second for sequential circuit simulation (by the VP-200) and 230×10^3 events per second for event-driven simulation (by the S-810/20). This performance is comparable to that of hardware simulation engines. Moreover, our techniques are so straightforward that we can implement them on most of the recent vector processors without serious modifications.

In the next section, we provide an overview of some important features of vector processors The two sections that follow are devoted to an explanation and consideration of the simulation techniques based on the compiler- driven method. Section 3.3 deals with the simulation of combinational circuits and Section 3.4 with the simulation of synchronous sequential circuits. A vector processor oriented technique for event-driven simulation is stated in Section 3.5. The last section concludes this chapter with some comments.

3.2 Vector Supercomputers

Vector processors are supercomputers which have been developed to meet the requirements for large scale computation in such area as hydrodynamics, numerical weather prediction, and nuclear energy research. Computation speed is increased by executing the uniform operations on array structured data using functional pipeline units. Main storage has a large capacity of several hundred mega bytes and is designed to afford enough access speed to balance the high throughput of the functional pipeline units. The maximum performance of recent vector processors reaches several MFLOPS (Million FLoating-point Operations Per Second) [Lub85], ten to hundred times faster than the largest general purpose computers.

Although the maximum performance of vector processors is very high, this performance is available only when almost all the operations in a program are executed by vector instructions. The execution speed will degenerate in accordance with the decrease of *vectorization ratio*, which is defined as the rate of the operations executed by vector instructions to all the operations. Let us define *vector execution of a program* as executing possible operations by vector instructions, *scalar execution of a program* as executing all the operations by scalar instructions and *acceleration ratio of a program* as the ratio of the execution speed by vector execution to that by scalar execution. When the vectorization ratio is 50 percent, acceleration ratio can not exceed 2.0 no matter how efficient the vector instructions are. Acceleration ratio increases markedly when vectorization ratio goes by 90 percent.

Another important factor to be considered is *vectorlength*. Since the overhead for setting up a vector instruction is considerably large, enough efficiency is not available if the operand vectors are short. (Sometimes vector instruction becomes less efficient than scalar instruction). Moreover, the execution speed is also swayed by such factors as the type of memory accesses, the type of instructions and the number of pipelines which can operate in parallel. With due regard to the above factors, we have not only to tune up the coding schemes but also to choose, modify or newly design basic algorithms so that the programs will be suitable for vector processing.

Table 3.1 summarizes the specification of the FACOM VP-100, VP-

Table 3.1 Specification of vector supercomputers.

	FACOM VP-100	FACOM VP-200	HITAC S-810/20
Instruction	82 vector 195 scalar	82 vector 195 scalar	83 vector 195 scalar
Functional Pipelines	add/logical multiply divide mask load/store × 2	add/logical multiply divide mask load/store × 2	add/logical multiply + add multiply/divide + add mask load × 2 store
Pipeline cycle	7.5ns	7.5ns	15ns
Vector Register	32KB	64KB	64KB
Maximum Capacity of Main Memory	128MB	256MB	256MB
Peak CPU Speed	266MFLOPS	533MFLOPS	630MFLOPS
Vectorizing Facilities	Fortran77/VP Compilers with Interactive Vectorizer	Fortran77/VP Compilers with Interactive Vectorizer	Fortran77 HAP Compilers

200 and the HITAC S-810/20, new vector processors on which we have developed our logic simulators. They have many advanced features making it versatile for a wide range of applications. The functional pipeline units can afford not only floating point operations but also fixed-point operations and bit-wise logical operations. The load/store pipeline, which transfers data between the main storage unit and vector registers, can afford three types of vector accesses:

a) contiguous vector access,

b) constant stridden vector access, and

c) indirectly addressed vector access.

As for the access speed, (a) is the fastest and (c) is the slowest. The indirectly addressed vector access allows an operation coded in the following FORTRAN statements to be vectorized and executed efficiently.

```
      DO 10  I=1,N
   10 A(I)=B(L(I))
```

Where the array L is an array of integers. This facility is particularly appropriate to the vectorization of table look-ups in the logic simulation.

In addition, these vector processors are capable of handling DO loops containing the conditional statements.

```
      DO 20  I=1,N
   20 IF (A(I).GT.0.0) B(I)=B(I)+C(I)
```

Another powerful vector function is a *vector compress function*, which gathers the elements of a vector whose corresponding subscripts satisfy certain conditions. We can use this function by the following coding.

```
      K=0
      DO 30  I=1,N
        IF (A(I).GT.0.0) THEN
        K=K+1
        C(K)=B(I)
      ENDIF
   30 CONTINUE
```

This function plays a very important part in the event-driven simulation to be discussed in Section 5.

As well as the above powerful vector processing facilities, these vector processor have very large main storage. The maximum capacity of the main storage is 256M bytes (VP-200 and S-810/20) and 128M bytes (VP-100).

3.3 Vectorization of Combinational Circuit Simulation

3.3.1 Vector-Parallel Simulation Technique

For the simulation of combinational circuits, we propose *vector-parallel simulation technique (VP-technique)* which vectorizes the simulation procedure. This approach is based on a combination of the compiler-driven method, the T-algorithm and the parallel simulation technique. A sequence of states on a signal line is expressed with a vector of logical type (we call it a *pattern vector*). The evaluation of a gate is performed by a vector logical operation (or combination of operations) corresponding to a logical function of the gate. The order of gate evaluation is just the same as the conventional compiler-driven method. First, the pattern vectors on the primary inputs are given. Then gates are evaluated according to the order determined by level sorting or DF-sorting, and finally the pattern vectors on the primary outputs are computed. If these vector logical operations are bit-wise, we can use the parallel simulation technique in combination. Since one bit is enough to express a state of each signal line in the two-valued logic simulation, we can express w states with a word of w bits. By bit-wise logical operations, a result of gate evaluation for w patterns is computed simultaneously. In our vector-parallel simulation, we express a pattern of length p with a vector of length $\lceil p/w \rceil$, where $\lceil x \rceil$ is the smallest integer not smaller than x. If p is so large that vector length is not allowable because of storage limitations, we divide the pattern into patterns of adequate length and perform simulation by installments.

3.3.2 Reduction of Storage Requirements Based on DF-Sorting

In order to bring out the performance of vector processors, the length of pattern vectors are set to be very long, that is, many patterns are processed at a time. Since the storage space required to store these vectors increases with the length of pattern vectors and with the circuit size, the storage requirements for a large scale simulation will be quite large.

Fortunately, we need not store the pattern vectors for all signal lines in the circuit throughout the entire simulation. At the beginning of the simulation, only the vectors associated with the primary inputs must be stored. As a result of gate evaluation, some new pattern vectors are created and must be stored. On the other hand, there will be some vectors which will no longer be referred to. Since they are not necessary for the simulation after that point, we can use the memory area occupied by the vectors for storing newly generated vectors. By this strategy, the number of pattern vector areas required for simulation is usually expected to be much less than the total number of signal lines in the circuit. We will show some experimental results later.

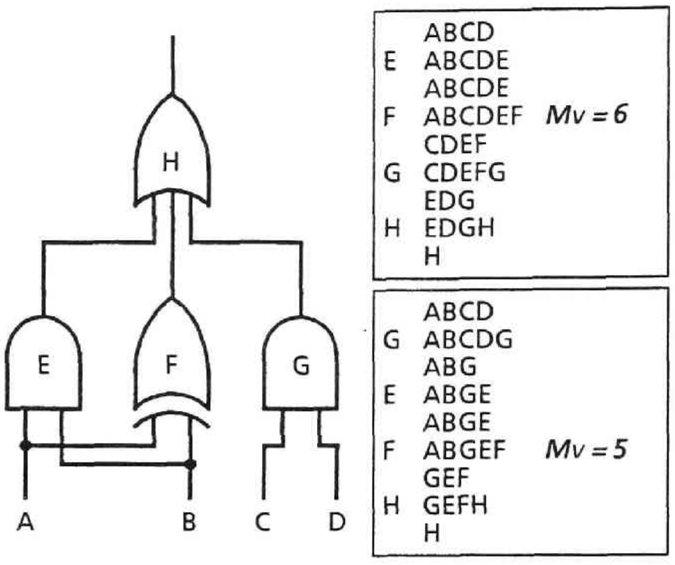


Fig. 3.1 M_v's for different orders of gate evaluation.

Let M_v be the number of pattern vector areas required to simulate a circuit by an order of gate evaluation. M_v varies with the change of the gate evaluation order for a circuit. Fig. 3.1 shows an example, in which M_v is 6 for the order E-F-G-H, while M_v is 5 for the order G-E-F-H. If we employ DF-sorting, we can get more candidates for the order than level sorting and can expect to have better order. Since the DF-sorting can deliver all the orders that allow correct simulation, it would be possible to choose the order that minimizes the storage requirements. But this optimization problem is very hard to solve completely (this problem can be shown to be NP-hard by the transformation from the register sufficiency problem [Gar79]). In order to find a near optimum solution by a brief computation, we introduce the following heuristics into DF-sorting.

We use *NDV* (the Number of Disposable Vectors) as a measure of choice. The *NDV* of an available gate at a certain simulation step is defined as the number of pattern vectors that will no longer be referred

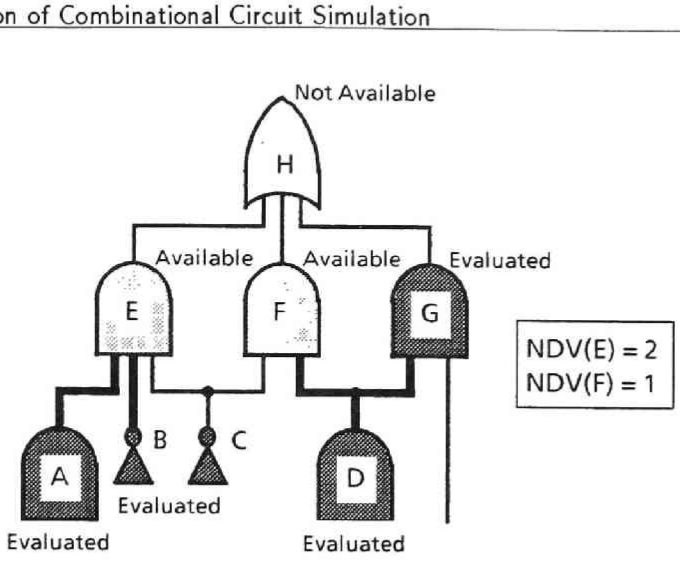


Fig. 3.2 *NDV:* the measure for choosing the next gate.

to after evaluation of the gate. In Fig. 3.2, gates A, B, C, D and G are already evaluated, E and F are available and H is not available. If we evaluate F in this situation, we can release the area for the output of gate D, but we must not release that of gate C because it will be referred to in the evaluation of gate E. Thus the *NDV* of F is 1, and similarly the *NDV* of E is 2. As shown in Fig. 3.3, the number of pattern vectors to be stored is equal to the number of primary inputs at the beginning and becomes 0 at the end. (We assume that result pattern vectors are stored into files as soon as they are computed). It is then clear that the maximum number of pattern vector areas required for simulation may be reduced when the gates with large *NDV* values are processed prior to those with small *NDV* values (see Fig. 3.3).

Table 3.2 shows a typical example of M_v's for the order obtained by our heuristic algorithm based on the DF-sorting and by the conventional level sorting. We can see:

1) M_v is much less than the number of total signal lines (number of gates in a circuit).

2) Compared with the level sorting, our heuristics reduce the M_v by

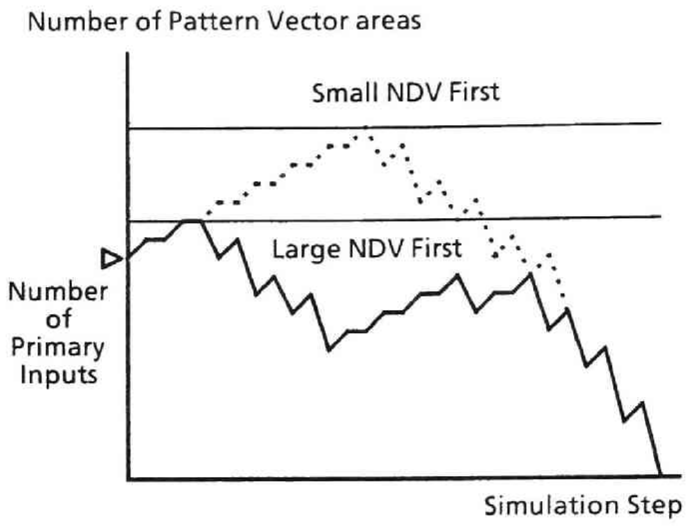


Fig. 3.3 Storage requirement for the strategies of gate ordering.

Table 3.2 Comparison of M_v

Number of Gates (Depth)	Level Sorting	DF-Sorting	Ratio[%]
85 (13)	19	14	73.6
168 (25)	31	22	70.9
270 (22)	38	26	68.4
346 (48)	55	38	69.0
1240 (48)	142	63	44.3
5296 (76)	542	119	22.0

20%~70%.

Since this order is determined statically from the circuit structure in the preprocessing stage, no overhead is brought about in simulation execution.

3.3.3 Multi-Valued Logic Simulation

As is the case with the conventional scalar processors, our vector processors support only two-valued logic operations. In order to handle multi-valued logic [Bre76], we must make up a mechanism to perform multi-valued logic operations. One of the most popular ways to realize multi-valued logic operation is a table look-up technique, such as zoom table look-up [Ulr80b]. Although the procedures for such a table look-up technique may be vectorizable, the simulation speed will be much slower than in two-valued logic simulation by the VP-technique for the following reasons.

1) Since one word is used to represent a logic value, we can not combine the parallel-simulation technique.

2) The array accesses for table look-ups are vectorized by indirectly addressed vector accesses, which are several times slower than vector logic operations.

As an alternative way, we propose the following *vector bit coding technique* which allows a simulation speed near to that of two-valued logic simulation.

1) Encode a vector of multi-valued logic with multiple vectors of two-valued logic.

2) Carry out multi-valued logic operations by combination of vector logic operations for the two-valued vectors.

a	a_1	a_2
0	0	0
1	0	1
Z	1	1
X	1	0

(a)

		b			
	y	0	1	Z	X
a	0	0	0	0	0
	1	0	1	1	X
	Z	0	1	1	X
	X	0	X	X	X

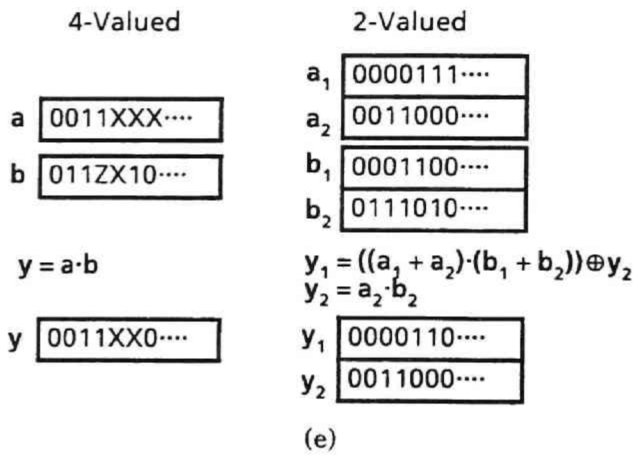
(b)

		b_1b_2			
	y_1y_2	00	01	11	10
a_1a_2	00	00	00	00	00
	01	00	01	01	10
	11	00	01	01	10
	10	00	10	10	10

(c)

$$y_1 = ((a_1 + a_2)\cdot(b_1 + b_2)) \oplus y_2$$
$$y_2 = a_2\cdot b_2$$

(d)



(e)

Fig. 3.4 Four valued logic simulation.
(a) Coding. (b) Table for four-valued AND operation. (c) Decomposition into two-valued logic. (d) Example of realization. (e) Logical operation on vectors.

Fig. 3.4 shows an example for four-valued logic simulation. The values 0 , 1, X and Z are used for representing logical zero, logical one, unknown and high-impedance, respectively. If we use a coding $\{0 = (0,0), 1 = (0,1), X = (1,0), Z = (1,1)\}$ shown in Fig. 3.4 (a), the four-valued sc and operation defined by the table Fig. 3.4 (b) are decomposed into two two-valued logic function expressed with the Karnaugh map in Fig. 3.4 Fig. 3.4 (c), which can be realized by the five two-valued logic operations like those in Fig. 3.4 (d). Here only NOT, AND, OR and EXCLUSIVE-OR are allowed as two-valued logic operations, because only the corresponding vector logic operations are available on our vector processors. As shown in Fig. 3.4 (e), the vector y, the result of four-valued AND between the vector a and b, is computed by encoding a and b with (a_1, a_2) and (b_1, b_2), respectively, and executing the logic operations shown in Fig. 3.4 (d).

As is seen from the above discussion, the computation time required for the gate evaluation is proportional to the number of two-valued logic operations to realize the multi-valued logic operations. In order to reduce the simulation time, it is essential to realize the multi-valued logic operations with the minimum number of two-valued operations. The problems to be considered are:

1) How to find the best coding?

2) How to find the way to realize the objective functions with the minimum number of two-valued operations?

If the coding and the operation table for the multi-valued operation are given, the objective functions will be fixed uniquely. Then the problem 2) is the *minimum logic design problem* [Mur72].

The choice of the coding must be closely related to the kind of the gates making up the circuit to be simulated. For example, in the case where the circuits are assumed to be designed only with ECL gates of NOR-OR function, it is preferable to choose the coding which minimizes

Table 3.3 Number of 2-valued logical operations for 3-valued logic.

Coding				not	and	or	ex-or	n-and	nor	Ave-4
00	01	10	11							
-	0 / 1	1 / 0	X / X	0	2	2	6	2	2	2.5
-	0 / X	X / 0	1 / 1	1	4	5	4	5	5	3.5
-	1 / X	X / 1	0 / 0	1	5	4	4	5	5	3.5
0 / 0	- / 1	1 / -	X / X	2	3	5	3	5	6	3.3
0 / 0	- / X	X / -	1 / 1	2	2	2	6	4	4	3.0
0 / 0	1 / X	X / 1	- / -	2	5	4	4	6	5	3.8
1 / 1	- / 0	0 / -	X / X	2	5	3	4	6	5	3.5
1 / 1	- / X	X / -	0 / 0	2	2	2	7	4	4	3.3
1 / 1	0 / X	X / 0	- / -	2	4	5	4	5	6	3.8
X / X	- / 0	0 / -	1 / 1	1	5	3	3	5	4	3.0
X / X	- / 1	1 / -	0 / 0	1	3	5	4	4	5	3.3
X / X	0 / 1	1 / 0	- / -	0	2	2	6	2	2	2.5

the number of two-valued operations to realize NOT, OR and NOR (of four-valued logic). Moreover we may be able to change the coding for every circuit so as to minimize the total number of two-valued operations required for simulating the circuit.

We have solved the minimum logic design problem mentioned above for the NOT, AND, OR, EXCLUSIVE-OR, NAND and NOR of three-valued and four-valued logic for every possible coding, and have realized the operations with two-valued logic operation and the number of operations required. The branch-and-bound algorithm is employed to solve the minimum logic design problem. The results are shown in Table 3.3

Table 3.4 Number of 2-valued logical operations for 4-valued logic.

Çoding				not	and	or	ex- or	n- and	nor	Ave- 4
00	01	10	11							
0 0	1 X	X 1	Z Z	3	5	4	8	6	5	5.0
0 0	1 Z	Z 1	X X	3	7	6	7	8	6	5.8
0 0	X Z	Z X	1 1	3	4	3	8	6	5	4.5
1 1	0 X	X 0	Z Z	3	7	6	8	8	6	6.0
1 1	0 Z	Z 0	X X	3	7	3	6	7	5	4.8
1 1	X Z	Z X	0 0	3	4	3	9	6	5	4.8
X X	0 1	1 0	Z Z	2	6	4	6	6	4	4.5
X X	0 Z	Z 0	1 1	2	7	3	5	7	4	4.3
X X	1 Z	Z 1	0 0	2	7	6	8	7	5	5.8
Z Z	0 1	1 0	X X	2	6	4	7	6	4	4.8
Z Z	0 X	X 0	1 1	3	8	7	8	8	6	6.5
Z Z	1 X	X 1	0 0	3	6	4	8	6	5	5.3

Table 3.5 Number of 2-valued logical operations.

Ciruit	2-Valued	3-Valued	4-Valued
32-bit Adder $nand \times 384$	768 (1.00)	768 (1.00)	2304 (3.00)
32-bit Adder $nand \times 128$, $and \times 496$ $exor \times 64$	352 (1.00)	704 (2.00)	1344 (3.82)
16-bit Multiplier $nand \times 2880$, $and \times 256$	6016 (1.00)	6272 (1.04)	18304 (3.04)
16-bit Multiplier $nand \times 960$, $and \times 496$ $exor \times 480$	2896 (1.00)	5792 (2.00)	11584 (4.00)

and Table 3.4. From the limitation of the computation time, we have got only approximate solutions for some of them. "Ave-4" in the tables, which gives the average for NOT, AND, OR and EXCLUSIVE-OR, shows that there exist codings that enable three-valued and four-valued logic simulation with a computation time on the average 2.5 times and 4.2 times greater, respectively, than in two-valued logic simulation. If we choose the best coding for each circuit, computation time will be further reduced, as shown in Table 3.5.

3.3.4 Implementation and Experiments

We implemented a gate-level (two-valued) zero-delay simulator based on the above simulation technique. As for the simulation control method, the code generation method is adopted. Fig. 3.5 shows the system configuration. First of all, the circuit description is written in SHDL (Structured Hardware Design Language) [Sak82]. A translator reads this description and generates a FORTRAN program which will realize vector-parallel simulation of the circuit (Fig. 3.6). The order of the gate evaluation is determined in this phase. The gate evaluation is coded using the built-in logic functions of FORTRAN (`IAND`, `IOR` in Fig. 3.6). The FORTRAN program is compiled into an object code for the vector processors

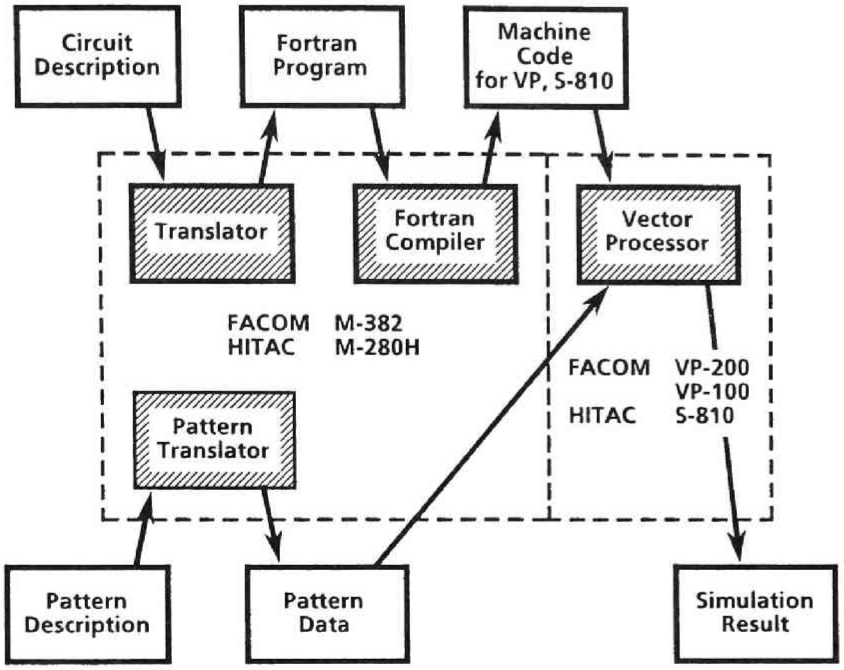


Fig. 3.5 System configuration of the combinational circuit simulator.

```
      DO 10 J = 1,LEN
      BUFF(J,4) = NOT(BUFF(J,1))
      BUFF(J,5) = NOT(BUFF(J,3))
      BUFF(J,6) = NOT(BUFF(J,2))
      BUFF(J,7) = IAND(BUFF(J,1),BUFF(J,2))
      BUFF(J,7) = IAND(BUFF(J,5),BUFF(J,7))
      BUFF(J,8) = IAND(BUFF(J,2),BUFF(J,3))
      BUFF(J,8) = IAND(BUFF(J,4),BUFF(J,8))
      BUFF(J,9) = IAND(BUFF(J,4),BUFF(J,5))
      BUFF(J,9) = IAND(BUFF(J,6),BUFF(J,9))
      BUFF(J,1) = IAND(BUFF(J,1),BUFF(J,3))
      BUFF(J,1) = IAND(BUFF(J,1),BUFF(J,6))
      BUFF(J,7) = IOR(BUFF(J,7),BUFF(J,8))
      BUFF(J,9) = IOR(BUFF(J,9),BUFF(J,1))
      BUFF(J,10) = IOR(BUFF(J,7),BUFF(J,9))
      BUFF(J,10) = NOT(BUFF(J,10))
      BUFF(J,1) = IAND(BUFF(J,4),BUFF(J,5))
      BUFF(J,3) = IAND(BUFF(J,5),BUFF(J,6))
      BUFF(J,7) = IAND(BUFF(J,4),BUFF(J,6))
      BUFF(J,1) = IOR(BUFF(J,1),BUFF(J,3))
      BUFF(J,1) = IOR(BUFF(J,1),BUFF(J,7))
   10 BUFF(J,1) = NOT(BUFF(J,1))
```

Fig. 3.6 A generated Fortran program.

Table 3.6 Summary of the speed performance.
(the combinational circuit simulator)

Computer	Execution Speed $[\times 10^6$ gate / sec$]$		$\frac{\text{Vector}}{\text{Scalar}}$
	Scalar	Vector	
VP-100	-	3865	
VP-200	322	7698	23.9
S-810	93	4204	45.2

by the FORTRAN77/VP compiler (FACOM) or the FORTRAN77HAP compiler (HITAC), and is executed with pattern data on vector processors.

Since the logic functions are compiled into vector logic operations in a straightforward manner, almost all the operations in the program are processed by vector instructions. All the vector accesses are contiguous accesses and the simulation speed is extremely fast. The vector length is $\lceil p/32 \rceil$ (where p is a pattern length to be processed at a time) by the 32-bit parallel simulation technique. Simulation speed then depends on the pattern length but not on the circuit size or circuit structure. Fig. 3.7 shows the relation between simulation speed and vector length. The simulated circuit is a full-adder of 19 gates. The simulation speed grows with the vector length but the nearly maximum speed is available at vector length 128 $\sim$ 512. Table 3.6 shows the maximum speed obtained by the scalar execution and the vector execution on the machines. An execution speed of 7.7×10^9 gate-evaluations per second is obtained on the FACOM VP-200. This speed is 20 $\sim$ 40 times faster than that by the scalar execution.

The capacity of the simulator (the maximum number of gates that can be simulated) is bounded by the storage size. In the VP-technique with

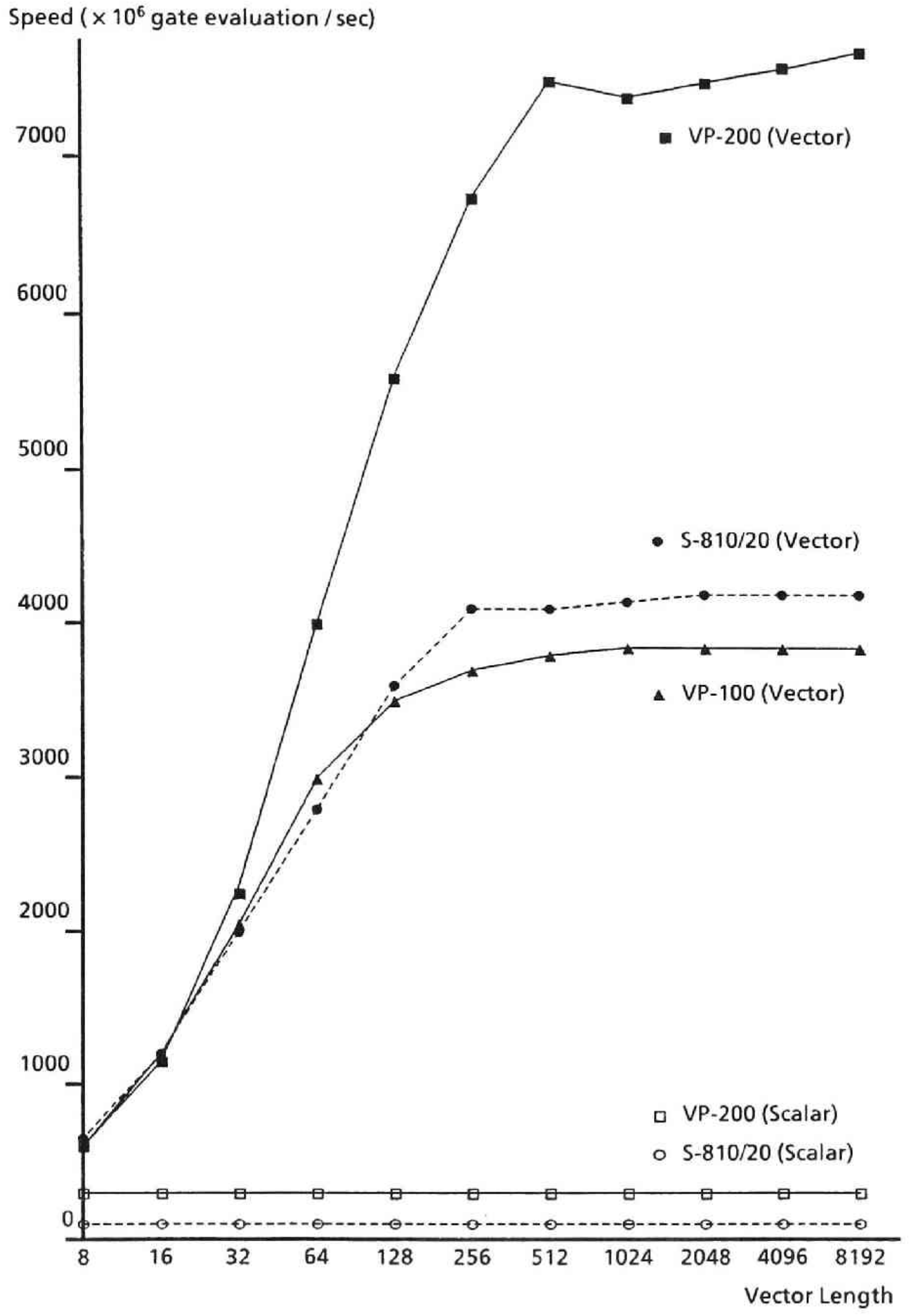


Fig. 3.7 Relation between vector length and simulation speed. (the combinational circuit simulator)

code-generation method, the storage is used for the simulation code and for the pattern vector areas. It is difficult to estimate the precise size of the simulation code, because it depends on the optimization strategies of the FORTRAN compilers. We can say that the code size is roughly proportional to the sum of the number of gates and the number of input/output lines of the gates. In the case of two-input gates, about 24 bytes are required per gate. As for the pattern vector areas, the storage required is proportional to the length of pattern vectors and to the M_v discussed in 3.3. It is difficult to get the M_v of a circuit only from its size because M_v also depends on the structure of the circuit. With reference to the result of Table 3.2, it is estimated that we can simulate more than 6M gates using the main storage of maximum size 256M bytes.

As an effective application of the VP-technique, we consider the Boolean comparison of two given combinational circuits. We prove the logical equivalence of the circuits by applying all possible patterns to the circuits and testing the consistency of the outputs of the two circuits. The consistency is tested by simulating the circuit shown in Fig. 3.8 and checking whether the state on V is always 0.

Let m, n, N_1, N_2 be the number of the primary inputs, the primary outputs, the gates of *circuit-1*, the gates of *circuits-2*, respectively. Then the number of the gate evaluation necessary for proving equivalence is

$$2^m \times (N_1 + N_2 + 2n - 1).$$

In the case where $m = n = 32$ and $N_1 = N_2 = 2000$ (approximately as same as 16-bit multipliers), the computation time required for simulation is about 40 minutes using FACOM VP-200. Although the computation times for pattern generation and result testing must be added (they are also vectorizable), one hour will be enough to prove the equivalence of the circuits.

Recent researches on Boolean comparison using binary decision diagrams [Fuj88, Min90] have made it possible to verify Boolean equivalence

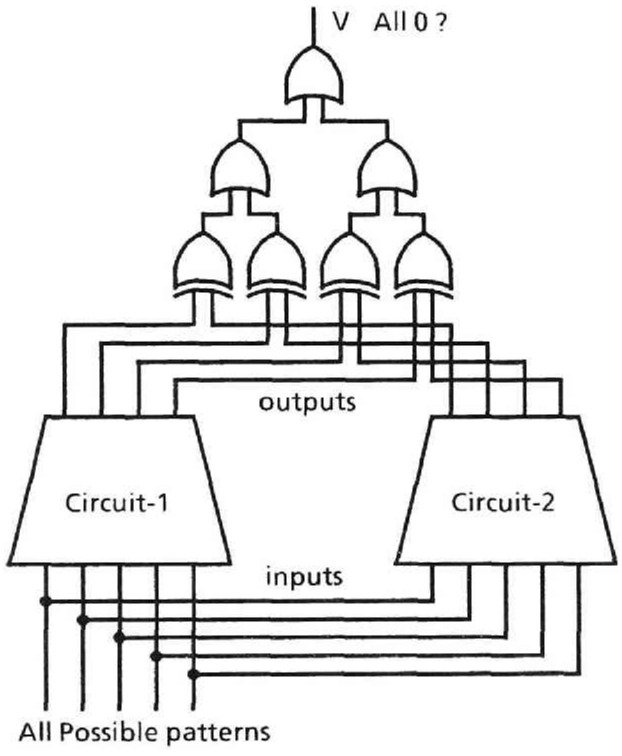


Fig. 3.8 The circuit for Boolean comparison.

within feasible storage and time for many practical functions. However, there are still many functions which are hard to handle by binary decision diagrams [Bry86, Bry90, Ish90y]. Multiplication is one of the typical examples. For such complex functions the use of our logic simulator on vector supercomputers is currently the most effective way of Boolean comparison.

3.3.5 Considerations for Further Acceleration

In the present system, we assign one FORTRAN statement for each gate. Load and store operations will be executed for each gate evaluation but some of them are not necessary if vector resistors are efficiently used. We can reduce this redundant load and store operations by programming in assembler or in machine code, but unfortunately FORTRAN is the only available language for programming on our vector processors. We can expect the similar effect by collecting several neighboring gates and expressing them in a single FORTRAN statement as shown in Fig. 3.9. By simple experiments with a circuit of ten and several gates, about 20 percent improvements in the simulation speed is observed.

3.4 Vectorization of Sequential Circuit Simulation

3.4.1 Gate Grouping Technique

A synchronous sequential circuit is generally composed of a combinational circuit part and registers or memories. For simplicity, we will consider a sequential circuit of the type shown in Fig. 3.10. Since loops in the circuit contain registers, simulation of a synchronous circuit is performed by evaluating the combinational circuit part at every clock period. Unfortunately, time first evaluation is impossible because evaluation of the combinational circuit part at a certain clock period requires the result of

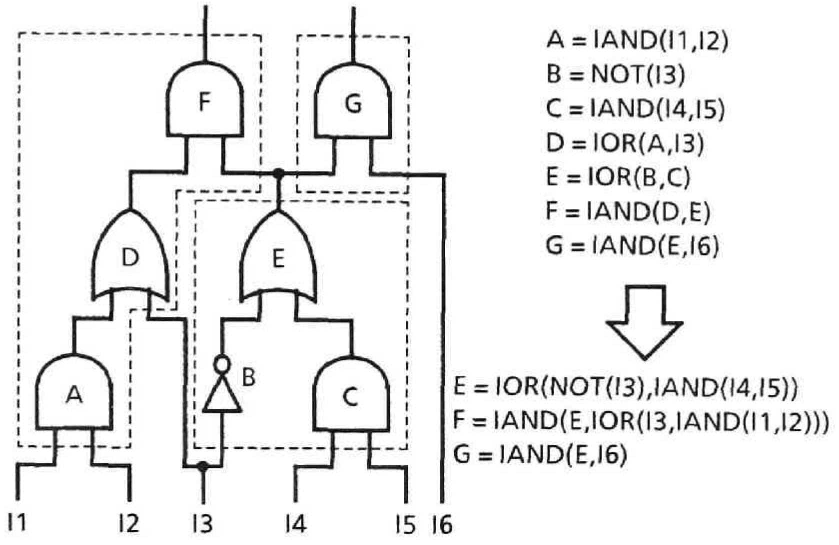


Fig. 3.9 An acceleration technique.

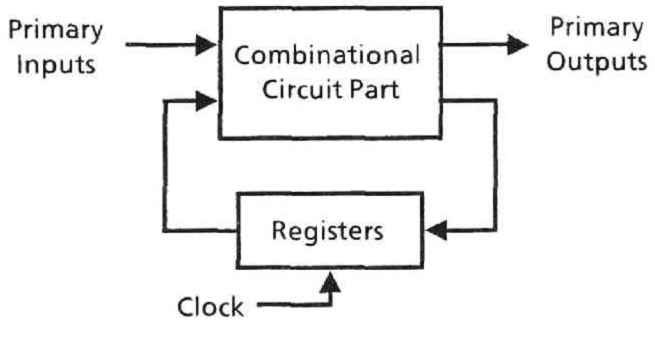


Fig. 3.10 A model of synchronous sequential circuits.

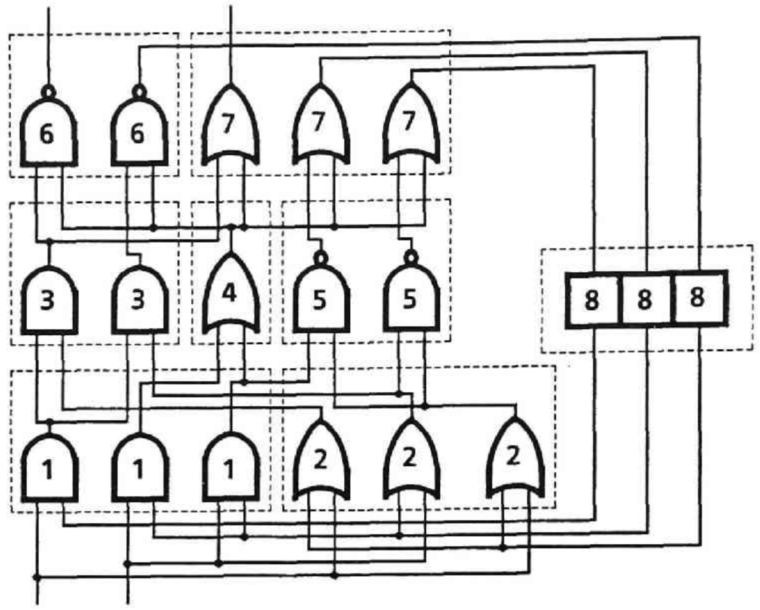


Fig. 3.11 Grouping of gates (1).

the previous clock period. Although it is possible to achieve vectorization by simulating many cases at a time, it will not be feasible for the following reasons.

1) In order to bring out the effect of the vector processing, we must simulate thousands of cases at a time.

2) When a circuit contains a large memory, we have to store the contents of the memory corresponding to all the cases.

We have developed alternative techniques based on the conventional space first evaluation algorithm. We increase the vectorization ratio by the *gate grouping technique (GG-technique)*. Gates of the same type are grouped and are evaluated together in a vectorized manner. The grouping must be done with careful consideration so that the order of the gate evaluation will be correct. Fig. 3.11 shows an example of the grouping. The groups of gates are processed in the order of their numbers

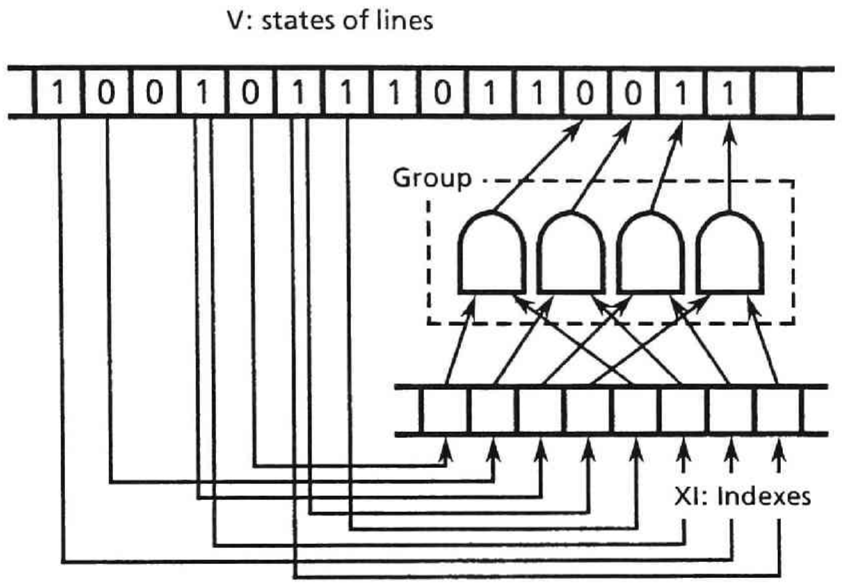


Fig. 3.12 Vectorization of gate evaluation.

in the figure. Registers are treated as simple buffers and are evaluated at the end of the clock period.

Fig. 3.12 illustrates how the gate evaluation procedures are vectorized. The vector V holds the states of the signal lines, and the vector XI gives the correspondence of the input lines of gates and the signal lines. The evaluation of gates in a group is performed as follows.

1) Input states for the gates are fetched from the vector V using the information of XI. This operation is vectorized by indirectly addressed vector accesses.

2) The logical operations for gates are performed. Since all the functions of the gates in a group are the same, this operation can be processed by vector logical operations.

3) The result of the logical operations is stored into the vector V. This is vectorized using contiguous vector access.

In the vector V, one word is assigned for a signal line in order to allow access V by the word. We can use the parallel simulation technique in combination by stuffing b states in a word (where b is the word length). Multi-valued logic simulation is also possible by the vector bit coding technique outlined in Section 4.3.

From the standpoint of efficient vector processing, it is desirable to make the average group size large, or to make the total number of groups small. In our approach, the average group size becomes large when (1) the circuit size is large and (2) the logical depth of the circuit is small. In large scale logic design, there is a tendency to increase the number of registers and reduce the depth of combinational circuits in order to improve testability or throughput [Seg83]. Thus we can conclude that our approach is suitable for the simulation of the latest large scale digital systems.

3.4.2 Grouping Algorithms Based on DF-Sorting

In this section we discuss the problem of gate grouping. The grouping shown in Fig. 3.11 is determined based on level sorting. That is, the gates which have the same functions in the same level are grouped together. They are evaluated in the order of the level number.

There exist other grouping methods which also guarantee correct simulation. Fig. 3.13 shows such a grouping on the same circuit as the previous example which yields larger average group size. This grouping is determined based on the DF-sorting mentioned in Chapter 2. The details of the grouping algorithm are as follows.

S1) At the beginning, gates with all inputs connected to primary inputs or register outputs are included in set S.

S2) Repeat S3) and S4) until S becomes empty.

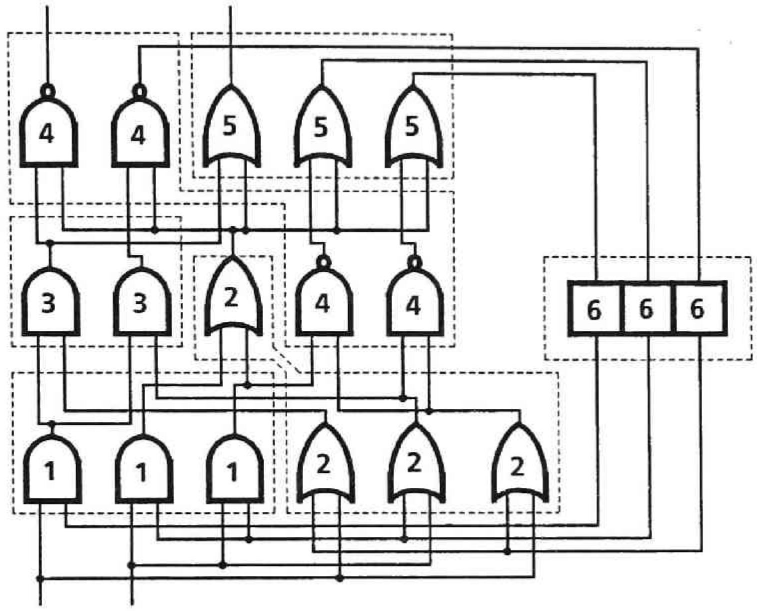


Fig. 3.13 Grouping of gates (2).

S3) Choose one gate type. Get gates of that type out of S, and construct a new group with them.

S4) If available gates are newly produced as a result of evaluation of the group determined in the previous step, include them in S.

S5) Group all the registers.

The average group size depends on the choice in S3). It is a very difficult (NP-hard) problem to make the optimum choice. In order to find a near optimum solution by a brief computation, we have developed the following three heuristic algorithms.

1) *Greedy Strategy*

Count the number of available gates for each gate type. Choose the gate type which has the largest number. This heuristic gives results as good as level sorting.

2) *Level+Greedy Strategy*

Compute the smallest level number of available gates for each gate type. Choose the gate type which gives the minimum level number. If there are multiple candidates, apply the greedy strategy. Although this heuristic seems similar to the grouping based on level sorting there is a difference in that the gates which belong to the different levels can be grouped together only if they are available. Clearly this heuristic guarantees a solution no worse than the level sorting.

3) *Individual-Inverse-Level+Greedy Strategy*

The Individual-Inverse-Level number of a gate g (denoted by $\mathrm{IIL}(g)$) is defined as follows. Here, for consistency, we treat a primary output as a gate with a single input and no output.

for a primary output o:

$\mathrm{IIL}(o) = 1,$

for a gate g:

$$\mathrm{IIL}(g) = \max_h \{ \begin{array}{ll} \mathrm{IIL}(h) + 1 & \text{(if } g \text{ and } h \text{ are of the same function)}, \\ \mathrm{IIL}(h) & \text{(otherwise)} \end{array} \},$$

where gate h is the fan-out destination of the gate g. In this heuristic, the gate type which gives the maximum IIL for available gates is chosen. This heuristic gives a still better solution than the *Level+Greedy* Strategy.

Table 3.7 shows the comparison of the solution obtained by the grouping algorithms stated here. The circuit (arithmetic circuit to compute sum of products) consists of about 7,000 gates of four types and of depth is 125.

Another way to enlarge the group size is to do logic conversion. By converting the original circuit to a circuit which consists of fewer types of gates, the average group size will be larger than in the original circuits. If

Table 3.7 Comparison of the grouping algorithms.

Algorithm	Number of Groups	Average Group Size	CPU Time [sec]
Level Sorting	355	21.1	0.75
Heuristic (1)	366	20.5	0.60
Heuristic (2)	304	24.6	1.13
Heuristic (3)	235	31.9	2.10

we choose a gate which is functionally complete, such as NAND and NOR, it is possible to convert any circuit to the one which consists of gates of a type. Then the average group size is equal to the averaging number of gates in each level. Although the number of operations for gate evaluation may increase, the total simulation time will be reduced if the effect of the improvement in the vector length is large. This conversion technique is considered to be effective when a circuit consists of many types of gates or when the vector length is short. Although the magnitude of the average group size is not typical because the depth is extremely large, we can see that the average group size is enlarged by a factor of about 1.5 by our heuristics. The results in Section 4.4.3 are obtained by the *IIL+Greedy* strategy.

3.4.3 Implementation and Performance Evaluation

We also implemented a sequential circuit simulator on the vector processors. Although it is possible to adopt the code-generation method as well as in the combinational circuit simulator, we took the table-driven method in order to investigate the difference in execution efficiency and preprocessing efficiency brought about by the different simulation control methods. Fig. 3.14 shows the system configuration. From a circuit description, the gate evaluation scheduling is determined and gates are divided into groups. Gates are renumbered so that the gates of each

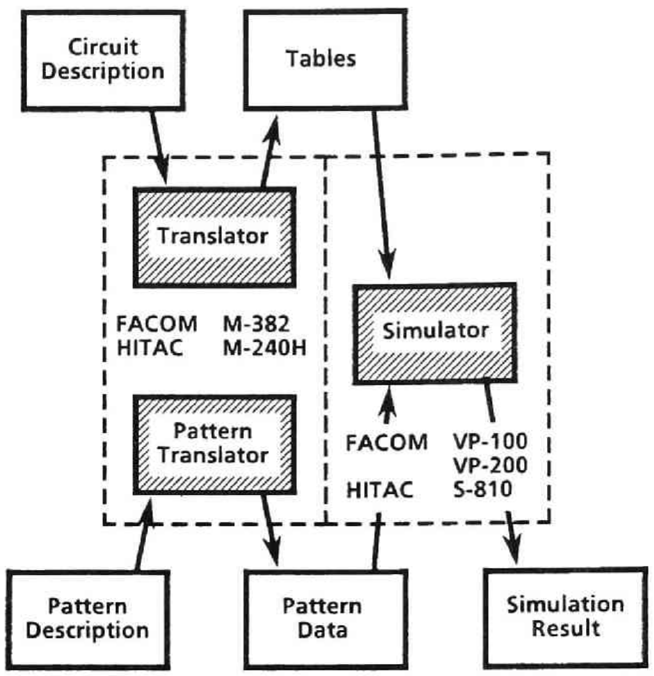


Fig. 3.14 System configuration of the sequential circuit.

group have contiguous numbers.

Tables generated by the translator contain two kinds of information. One is to give the connections and it corresponds to the vector *XI* mentioned in Section 4.4.1. The other table, called a group table, describes the function, number of gates, the index to the *XI* table and the index to the *V* table. Since entries of group table are ordered according to the gate evaluation scheduling, simulation proceeds by interpreting the group table from top to bottom. Gate evaluations are performed by executing a routine corresponding to the function of the group.

The average vector length, which is equal to the average group size, depends on the circuit size and circuit structure as discussed in Section 4.4.1. Fig. 3.15 shows the relation between vector length and execution speed of the simulator. From the figure we can see that the simulation speed saturates at a larger vector length than in the combinational circuit

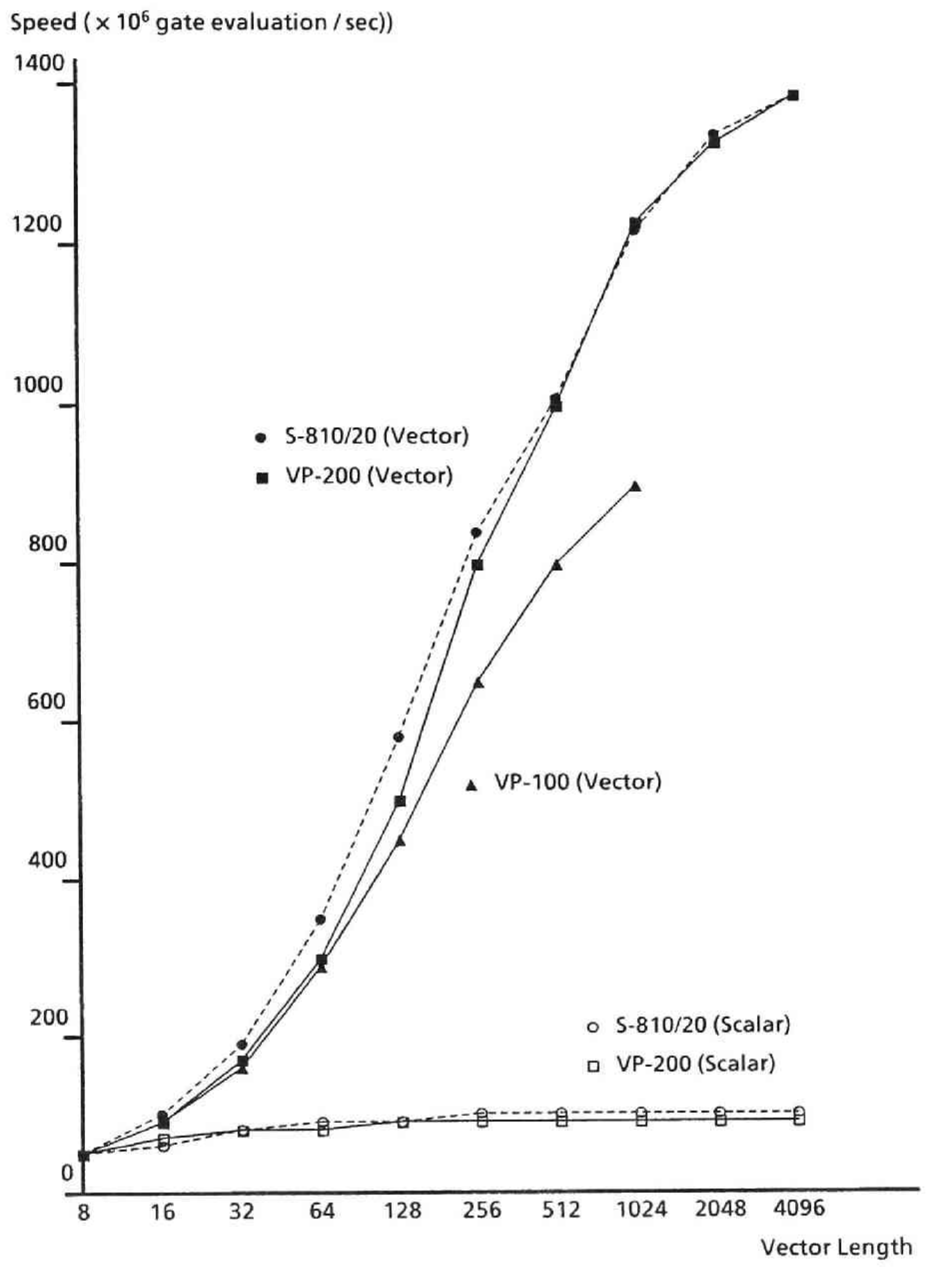


Fig. 3.15 Relation between vector length and simulation speed. (the sequential circuit simulator)

Table 3.8 Summary of the speed performance.
(the sequential circuit simulator)

Computer	Execution Speed [$\times 10^6$ gate / sec]		$\frac{\text{Vector}}{\text{Scalar}}$
	Scalar	Vector	
VP-100	-	902	
VP-200	92	1390	15.1
S-810	97	1386	14.3

simulator. This is because the overhead for table look-ups is dominant when the vector length is small. Table 3.8 summarizes the maximum performance of the simulator. By the vector execution, both the VP-200 and the S-810 yield the simulation speed of 1.4×10^9 gate-evaluations per second, which is over ten times faster than by the scalar execution. Compared with the combinational circuit simulator, it is several times slower because the sequential circuit simulator executes input fetches for the gate evaluation by indirectly addressed vector access, while the combinational circuit simulator does this by contiguous vector access.

In the case of the practical circuit referred to in Table 3.7, simulation speed of 122.6×10^6 gate-evaluation per second is observed on the VP-200 (average vector length is 31.9, see Table 3.7). This result is slower than that shown in Table 3.8. But the efficiency in gate evaluation turns to be higher if we consider that about 1.8 times as many operations are necessary for evaluating a gate, on an average, through the use of gates with three inputs and negative gates. This is because of the similar effects of the load/store optimization technique mentioned in Section 4.3.5.

The storage space is used for storing the group table, the connection information vector XI and the state vector V. Let the total number of input lines be i and the number of signal lines l. Then the size of the XI and V are $4i$ bytes and $4l$ bytes, respectively. As for the group table, 20 bytes are required for a group but it will be negligible when the vector length is large enough. It follows that in the case of 2-input gates the maximum capacity is more than 20M gates using the storage of 256M bytes. This capacity is larger than that of the combinational circuit simulator because we do not need to store many patterns for a signal line.

3.4.4 Code-Generation Method vs. Table-Driven Method

As for the simulation control method, we adopted the code-generation method for the combinational circuit simulator and the table-driven method for the sequential circuit simulator, and we examined the difference in execution efficiency and preprocessing efficiency. The code-generation method has the following advantage over the table-driven method in execution efficiency.

M1) In the code generation step, we can apply the load/store optimization technique mentioned in Section 4.3.5.

M2) There is no overhead for interpreting the table (Compare Fig. 3.7 with Fig. 3.15).

On the other hand, the code generation method has some disadvantages in the preprocessing efficiency:

D1) It takes a lot of computation time to generate FORTRAN programs because of the needs to process character strings.

D2) It also takes a lot of computation time to compile FORTRAN programs because the FORTRAN77/VP compiler and FORTRAN77 HAP compiler perform vectorization and other special optimizations.

In the simulation of small scale circuits, the effect of D1) and D2) are negligible, but the preprocessing overhead will increase significantly in accordance with the increase in circuit size. Moreover, the overhead for interpreting the table will be negligible when a long enough vector length is available. We think that the table driven method is better for large scale simulation.

The above discussion is based on the situation where FORTRAN is the only available language for the programming on our vector processors. If we can program in assembler or machine code, the code-generation method will be more advantageous.

3.4.5 Modeling of Circuits

Although we have discussed only the simple sequential circuit model shown in Fig. 3.10, our simulation techniques can also treat more complicated ones, such as clock distribution logic and registers with reset/preset. This is done by adding proper logic to a buffer representing a register. The buffers, which we have used to represent registers, can also be regarded as unit delays. By adding buffers to every logic gate, unit delay simulation is also possible; thus we can handle asynchronous circuits. In this case we can easily obtain very large vector length because the depth of the combinational circuit part is always just 1. With these capabilities this simulation technique is considered to be flexible and can be put to practical use.

3.5 Vectorization of Event-Driven Simulation

3.5.1 Vectorization of Event Processing

In Section 4.4.3 and 4.4.4, we have discussed the simulation techniques based on the compiler-driven method. We now consider the vectorization of logic simulation based on the event-driven method. The basic algorithm that we adopted is the conventional event-driven method with the time mapping technique [Bre76]. An *event* is defined as a change of the output state of a gate (for simplicity, we assume that a gate has a single output). Events whose occurrence is definite are maintained using the data structure called a *time wheel*. It consists of linear lists chained to a circular list of headers each of which is associated with a time period. Simulation is performed by advancing the time by a certain unit and carrying out the following steps at each time period. Here, the time that we are concerned with is referred to as *current time*.

E1) *Event fetch*: Get *current events*, the events scheduled to occur at the current time, out of the time wheel.

E2) *Event propagation*: Retrieve gates affected by the current events.

E3) *Gate evaluation*: Compute new output states of the gates obtained in E2).

E4) *Event registration*: If there are changes in the output states, register this information into the time wheel as new events.

As a method to avoid the duplication in evaluation and registration for multiple input changes at a gate, the one-pass strategy and the two-pass strategy are known [Ulr69]. Although the one-pass strategy is, in general, slightly more efficient in the gate-level simulation, we adopted two-pass strategy because of restrictions of vector processing.

The procedures for the above four steps are vectorized by processing all the events together which are scheduled in the same time period. No serious modifications are made to the algorithm itself, but the data structures and the operations on events are redesigned to be suitable for vector processing.

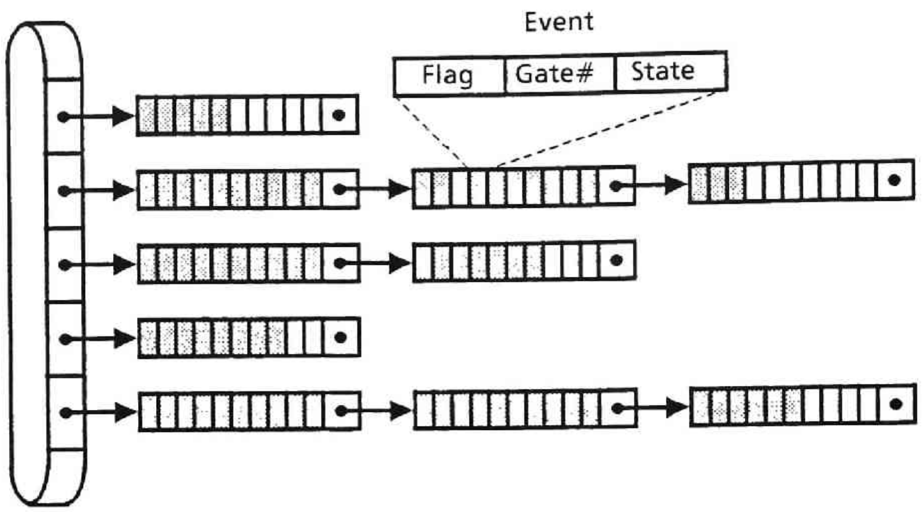


Fig. 3.16 An array structured time wheel.

3.5.2 Event Fetch

Since the original structure of a time wheel does not have good affinity for vector processing, we do not chain events one by one but collect certain numbers of events together (128 events in our current system) and treat them as a list of arrays (see Fig. 3.16). We call this data structure an *array structured time wheel.* In this step, events are first fetched from the time wheel and a *current event vector* is produced so as to vectorize the subsequent event processing (see Fig. 3.17). This operation is a simple duplication of arrays of events owing to the adoption of array structured time wheel, and is easily vectorized. The information of an event consists of a gate index, a new state and a *validity flag.* A validity flag indicates whether the event are canceled or not (in the sophisticated delay mode simulation) and we must exclude the events with 'canceled' flags from the event processing. We make this exclusion in the event propagation step for reasons of efficiency, and here simply copy the events.

Next, external events, the events for the primary inputs or the events to change the state of the gate from outside, are given by an *external*

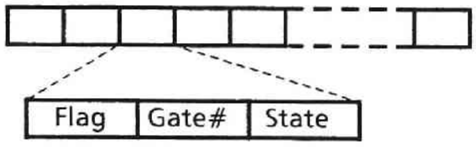


Fig. 3.17 A current event vector.

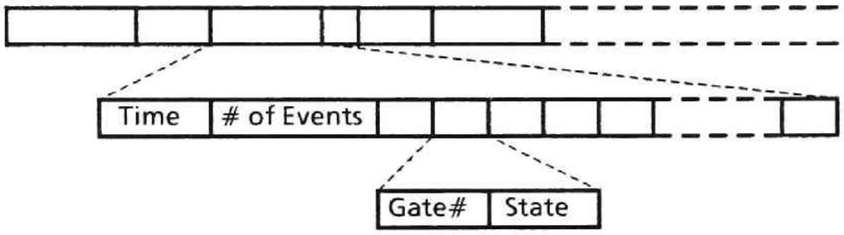


Fig. 3.18 An external event vector.

event vector shown in Fig. 3.18. These events are taken out and are appended to a current event vector at the point where the simulation proceeds to the specified time. In the case where there is an event with the same gate index as fetched from the time wheel, we cancel the one from a time wheel.

We next update the output states of gates according to the information in the current event vector. This is done by storing new output states for the gates in the current event vector. This is vectorized straightforwardly using indirectly addressed vector access. At the same time, we must record the events, as a result of simulation, for the gates that are specified to be traced. This is carried out by the following operations:

1) Test whether the gates, on which current events are occurring, are specified to be traced.

2) Test whether the output states of the gates change. This is necessary because there are some cases where the states are not changed by events on account of the cancellation of events,

3) Gather events which satisfy 1) and 2).

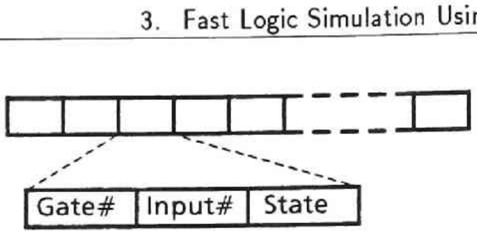


Fig. 3.19 An active gate vector.

Operations 1) and 2) are processed by table look-ups and are vectorizable using indirectly addressed vector access. Operation 3) is vectorized by vector compress function.

3.5.3 Event Propagation

We call a gate an active gate if it is affected by current events. In the event propagation step, we make an active gate vector from the current event vector. As shown in Fig. 3.19, an element of the active gate vector consists of indexes of a gate and input position, which are affected by a current event, and a new state. First of all, the events whose flag indicates 'canceled' are dropped (by vector compress operation). Next fan-out destinations are retrieved for all the gates in the current event vector to get gate indexes for the active gate vector. If an output line of a gate has large number of fan-out destinations (for example, a gate to supply output to a clock line or a reset line), enough vector length is available in searching the fan-out table. But since most signal lines have a few fan-out destinations, maybe 1 to 4, the average vector length will be very short if we search the table in the fan-out direction for each line. We propose to search the table not in the fan-out direction but in the gate index direction. First, we get gate indexes (and input position) of the first fan-out destinations of all the current events. Next we get gate indexes of the second fan-out destinations of current events which have not less than two fan-out destination. Then the third, the fourth and etc. Finally we have a new active gate vector. The operation to gather current events, whose gates have i or more fan-out destinations,

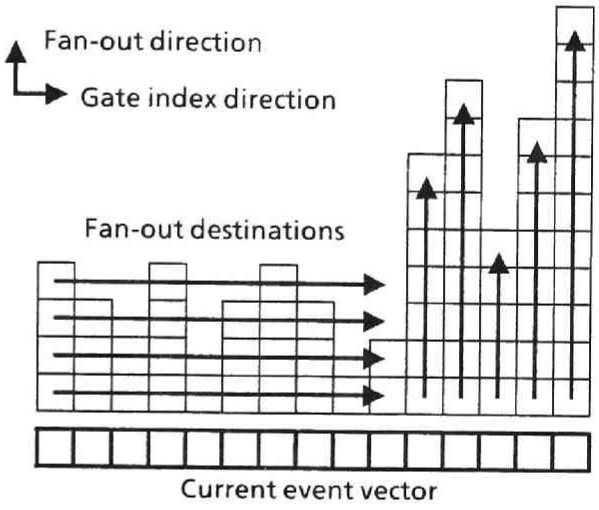


Fig. 3.20 Search of fan-out destinations.

are vectorizable using the vector compress function.

There remains, however, the problem that the vector length will decrease as we come to a large fan-out number. We have the following ideas as countermeasure for this problem:

1) As shown in Fig. 3.20, search the table in the fan-out direction for lines with large fan-out number and in the gate index direction for lines with small fan-out number.

2) If there are gates with many fan-out destinations, divide them into the lots with the adequate size.

3) Combine 1) and 2). Namely search table in the fan-out direction for gates with many fan-out destinations and divide fan-out destinations for gates with the medium number of fan-out destinations.

In the current system only 1) is adopted. It is remained for future research to examine which strategy is effective and at what vector length the way of searching should be changed.

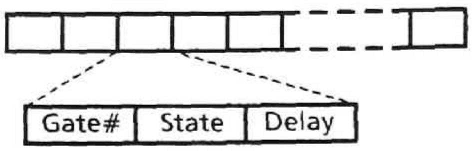


Fig. 3.21 A new event vector.

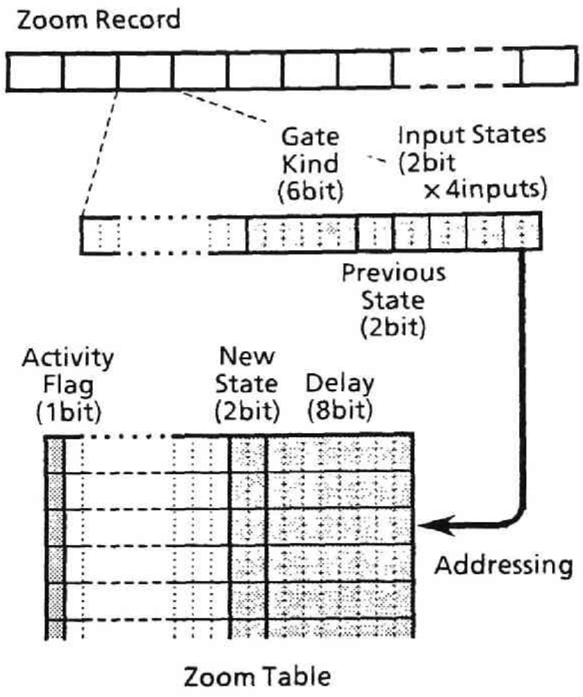


Fig. 3.22 A data structure for gate evaluation.

3.5.4 Gate Evaluation

In this step we perform gate evaluation using the information of an active gate vector and make a new event vector shown in Fig. 3.21, whose element consists of a index of gate, a new state and a propagation delay.

Among gate evaluation techniques devised for efficient logic simulation, we consider zoom table look-up [Ulr80b] is one of the most suitable techniques for vector processing, because the output states of gates are computed by uniform table look-up operations regardless of the functions or the fan-in numbers of the gates. (In this case the bit vector coding technique discussed in 3.3 is not so effective because parallel simulation technique is inherently impossible in the event-driven simulation.) Fig. 3.22 illustrates an example of the data structure for gate evaluation. The information necessary for gate evaluation, such as input states and the types of gates, is described in a zoom record vector. Two bits are used to express an input or output state so as to handle four-valued logic. In this case, the maximum inputs for a gate, which is restricted by the size of the zoom table, is 4. The previous output states are recorded in a zoom record vector as well as (the current) input states in order to get the delay values and presence of output state changes (in case of rise/fall delay model) together with new output states. Using a zoom record vector as an index vector, a zoom table is looked up by indirectly addressed vector access. The information of presence of an output state change, a new output state and a delay value are stuffed into a word to reduce the size of the table and the number of table look-ups. They are separated using vector logical operations and vector shift operations.

Prior to the zoom table look-up, we must update input states in the zoom record vector from the information of the active gate vector. (This operation is also vectorized by logical and shift operations). Here, the handling of multiple changes on gate inputs is somewhat troublesome. As shown in Fig. 3.23, if the both inputs of the gate 7 change, the element of a zoom record vector concerning with gate 7 must be updated from the information of two elements of an active gate vector. In the case of scalar processing no problem occurs because a state of the first and the second input states are updated separately. But in the case of vector processing, (which means that if we vectorize this procedure by compulsion), the result on the zoom record of gate 7 can be erroneous because of the conflict of the store operations. Moreover, the check and the cancellation of such duplicated events are very difficult. In order to avoid these problems, we

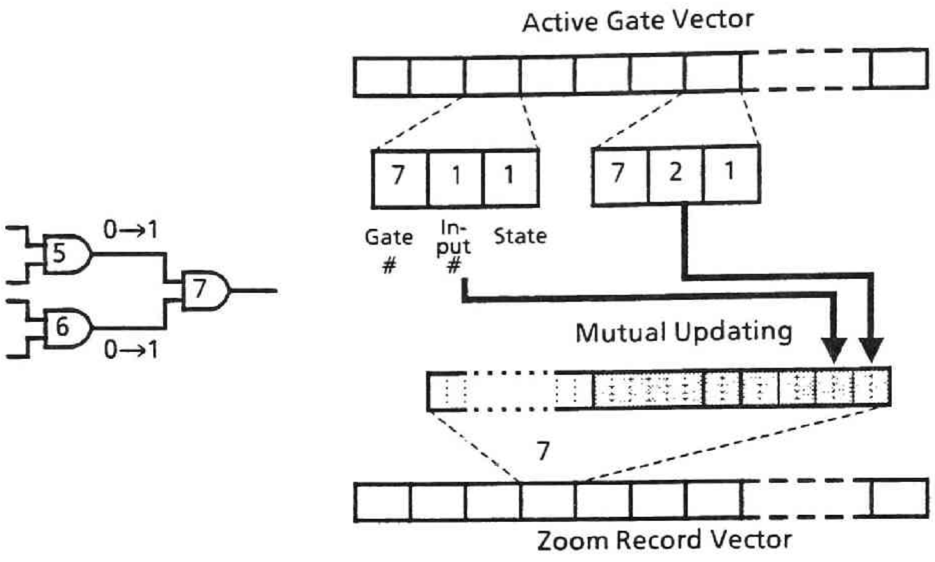
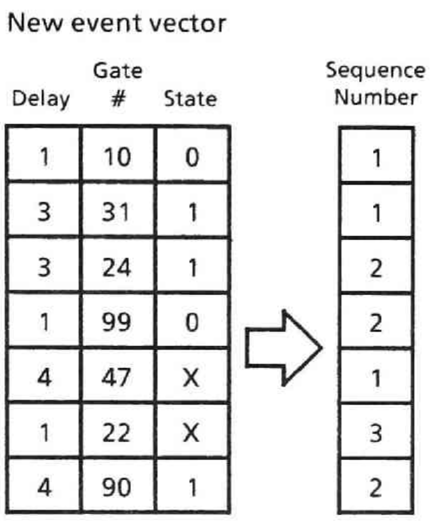


Fig. 3.23 A write conflict for simultaneous input changes.

update the zoom records for each input position. Namely, zoom records are first updated only with the information of active gates whose input position is 1. Subsequently the same operations are carried out for input position 2, 3 and 4. By this dividing strategy, we can avoid the conflict and vectorize the procedure. In addition, we can exclude the duplication of events by checking whether there has been updating for the same gates, at every updating associated with input position 2, 3 and 4. One demerit of this strategy is that vector length becomes shorter than that of an active even vector.

3.5.5 Event Registration

In the event registration step we register events, whose occurrences are newly known in current time, into a time wheel mentioned in 5.2. Gate indexes and new states in the new event vector are written into a list of arrays corresponding to the occurrence time. If the positions for new events to be inserted are determined, they can be written into by in-



Delay	Gate #	State	Sequence Number
1	10	0	1
3	31	1	1
3	24	1	2
1	99	0	2
4	47	X	1
1	22	X	3
4	90	1	2

Fig. 3.24 Counting of the same delay values.

directly addressed vector access. The operation required to determine the position is, in essence, to number the elements of a new event vector for each delay value (see Fig. 3.24). Unfortunately, this operation is unvectorizable in the current architecture of our vector processors and is carried out by scalar execution. But in the case of zero-delay and unit-delay simulation, it can be vectorized in quite a simple manner.

When we use the sophisticated delay models such as rise/fall delay, inertia delay and minimum/maximum delay, the operation of event cancellation is necessary. This operation is to cancel all the events associated with a certain gate which are already registered and maintained in the time wheel. Let d_r and d_f be the rise-delay and fall-delay, respectively, of a gate. Assume $d_f < d_r$ for example. Let e_1 be an input event which occurs at time t_1 and causes a signal rise on the output, and let e_2 be an input event which occurs at time t_2 ($t_1 < t_2$) and causes a signal fall on the output. Then event of the signal rise scheduled at time $t_1 + d_r$ must be canceled at time t_2 if

$$t_2 + d_f < t_1 + d_r.$$

Table 3.9 Summary of the speed performance.
(the event-driven simulator)

Delay Model	Scalar Coding		Vector Coding	
	Scalar Exec. (A)	Vector Exec. (B)	Scalar Exec. (C)	Vector Exec. (D)
Zero	106 89	83 92	44 34	277 335
Unit	87 77	85 90	38 29	296 342
Rise/Fall	65 61	75 78	36 28	206 229

[$\times 10^3$ event/sec]
Upper: VP-200
Lower: S-810/20

Recording the occurrence time of the previous events for every gate and checking the condition denoted as the above expression, we can easily judge whether to cancel the previous event or not. We can also cancel the events by chaining all the events associated with a gate and write 'canceled' into the flag of the events. These operations are vectorizable using indirectly addressed vector access, vector add and vector compare operation.

3.5.6 Implementation and Performance Evaluation

Based on the above consideration, we implemented a gate-level (four-inputs and a single output gates) four-valued simulator with three delay modes (zero-delay, unit-delay and assignable rise/fall delay). In order to evaluate the effect of the vector coding, we also prepared a simulator of conventional scalar coding. Table 3.9 shows the simulation speed obtained by experiments. The circuit simulated is a 16-bit multiplier (combinational circuit) of 1700 gates. The average length of current event vectors was 108, 221 and 206 for zero-delay, unit-delay and rise/fall delay simulation respectively. (A) and (C) in the table indicate the execution speed of the scalar execution, and (B) and (D) of the vector execution. Maximum performance (by the vector execution of our vector coding) of rise/fall delay simulation is about 230×10^3 event per second, or 440×10^3 active gate evaluation per second on the HITAC S-810/20. This is slower than that in zero-delay or unit-delay simulation mode because the procedures for event registration are not vectorized. In the case of our vector coding, simulation speed is accelerated by 8 to 11 times by vector execution (compare (C) with (D)), while very little improvements are observed in the case of conventional scalar coding (compare (A) with (B)). But the overhead for the vectorization is large (compare (A) with (C)), total performance improvements are no more than 4 times (compare (D) with (A)).

3.5.7 Compiler-Driven Method vs. Event-Driven Method

As stated in Chapter 2, the event-driven method has the merits over the compiler-driven method that the gate evaluation count is fewer because only active gates are evaluated. Owing to this merit, event-driven method has a possibility to be advantageous also in the case of zero-delay or unit-delay simulation mode. But our experiments tell us:

1) The overhead of event scheduling and event propagation is comparatively large and the procedures are not easy to be vectorized.

2) On the other hand, the gate evaluation procedures in the compiler-driven simulation have very good affinity for the vector processing.

Since our event-driven simulator is primarily designed for assignable delay simulation, it employs only time mapping technique. If we tune it up for zero-delay simulation and combines level mapping technique, the performance may be improved. But the compiler-driven method is still considered to be advantageous in the zero-delay or unit-delay simulation, so long as the ratio of event occurrences is not so low.

3.6 Remarks and Discussions

High-speed logic simulation techniques suitable for vector processors have been proposed. They are vector parallel simulation technique, gate grouping technique and vectorized event processing technique. As well as the algorithms for simulation, the algorithms for preprocessing are also very important for efficient simulation. In order to reduce the storage requirements or to extend the vector length, we have proposed some heuristic algorithms based on the data flow sorting. We have achieved very high performance through vectorization especially in the compiler-driven simulation of combinational and sequential circuits. The performance of our simulators is comparable to that of hardware simulation engines such as YSE [Den83]. As for the event-driven simulation it was difficult to achieve as much acceleration ratio as the compiler-driven simulation, the final performance of the simulators are significantly high. The reason for that is that the event-driven simulation algorithms is originally not suitable for vector processing. As shown in the experimental results, simple vector coding [Kro81] results in almost no acceleration. There are operations which are essentially unvectorizable or which are vectorizable only at the cost of large increase in computation cost. Addition of new vector instruction for logic simulation, which is employed in VELVET [Nag86], may be a good solution for this problem .

Presently, logic simulators on general purpose scalar computers are still prevalent. In the gate-level simulation, it seems to be difficult to achieve significant acceleration in simulation speed through the improvements of the algorithms. Considering the increase in the size of the circuit (and the size of test patterns) to be simulated, soon it will be indispensable to enlist the aids of hardware simulation engine or parallel or vector computers. In view of the CAD/DA system configuration, logic simulators on a vector processor will be more attractive by the readiness of interconnecting with other CAD/DA tools and the economical efficiency brought about when we share the vector processor with other tools such as a device simulator.

Chapter 4

Fast Fault Simulation Using Vector Super Computers

4.1 Introduction

Fault simulation is to simulate the behavior of a logic circuit which has a *fault* in it. While logic simulation is used for logic design verification, fault simulation is used for analysis of the behavior of faulty circuits, test set generation or quality evaluation of test sets for logic circuits. Fault simulation requires much more computation cost than logic simulation, because simulation must be carried out for each of the faults derived from a certain fault model. For example, the computation cost of fault simulation for a given test vector under single stuck-at fault model, which is the most commonly used one, is $O(n^2)$, where n is the number of the gates constructing the circuit [Har87]. In the practical field of testing, there is a growing interest in extensive use of random patterns and in a built-in self test approach [Wai89] which cover faults that can not be modeled by the single stuck-at fault model and exempt us from the high computation cost for algorithmic test generation [Fuj85]. This leads, in turn, to very high computation costs for fault simulation to evaluate the quality of the random patterns. Various research projects have been carried out in order to accelerate fault simulation by improving the algorithms

[Arm72, Ulr80a, Wai85, Nis85, Ant87, and etc.], or to develop alternative techniques to fault simulation [Abr83, Jai84, Brg85, and etc.]. There have been also researches to develop special purpose hardware which accelerates fault simulation [Cha86] including whole the test generation process [Hir88], or to develop test generation system on parallel computers [Mot86]. In this chapter, we propose a new technique to accelerate fault simulation using *vector supercomputers* [Ish90i].

We discuss the zero-delay two-valued fault simulation of gate-level combinational circuits. *Parallel*, *deductive*, and *concurrent* fault simulation are known as the typical methods of fault simulation [Fuj85]. Although many of the recent fault simulators employ the concurrent simulation technique which is an extension of the event-driven logic simulation technique to fault simulation, we considered it advantageous to base our vector algorithm on the parallel simulation technique because of its suitability for vector processing. We propose a *dynamic two-dimensional parallel fault simulation technique* as a vector processor oriented fault simulation technique.

Parallel simulation utilizes bit-oriented logic operations to perform a lot of gate evaluations simultaneously. We can classify parallel simulation into fault-parallel simulation and pattern-parallel simulation according to the parallelism factors. In each method, by simply extending the unit of gate evaluation from a word to a vector consisting of multiple words, as is in the case of logic simulation, we can execute fault simulation very much efficiently using vector instructions when the vector length obtained is large enough. However we cannot obtain enough vector length or the computation cost increases if we attempt to get enough vector length, because fault dropping is performed when we use fault simulation for test quality evaluation in practice.

In order to meet this problem, we combine a *two-dimensional parallel fault simulation technique* and a technique of *dynamic adjustment of the* *parallelism factors.* In our method,

1) we obtain large vector length by utilizing both fault and pattern parallelism, and

2) efficiently achieve fault dropping by adjusting the two parallelism factors complementarily form pass to pass.

We further reduce the computation time by combining this technique with *selective tracing* under the notion of multiple fault propagation. We implemented a fault simulator based on our new technique on the Fujitsu FACOM VP-200 vector processor and made some experiments. The simulation speed is accelerated by 10∼15 times through vectorization. It is particularly efficient in simulating large circuits with many patterns.

After we show the notion of the dynamic two-dimensional parallel simulation technique in section 4.2, we describe implementation methods of selective tracing under the notion of the multiple fault propagation in section 4.3. In section 4.4, we examine the performance of our simulator under the experimental results. The last section concludes this chapter with some comments.

4.2 Dynamic Two-Dimensional Parallel Simulation Technique

4.2.1 Fault Simulation

A fault simulator computes an output pattern of a logic circuit for each given input pattern under each given fault occurrence. We will use a word *pattern* instead of *vector* in order to avoid confusion between a *test vector* and an *operand vector for a vector processor.* Fault simulators are used for

1) distinguishing faults as detectable (or undetectable) by given input patterns,

2) computing the percentage of fault coverage of given input patterns,

3) generating a test set by selecting effective patterns from given input patterns, and

4) generating a fault dictionary, etc.

In this paper, we focus on gate-level combinational circuits and zero-delay two-valued simulation, and assume a single stuck-at fault model [Bre76]. We are interested in using fault simulation to perform 1)$\sim$3) for given enormous patterns such as random patterns. If we only intend to determine if a fault is detected or not by given patterns, we can delete faults from the undetected fault list as soon as they are detected by some patterns, and we can dispense with simulation for other patterns. This technique which drastically reduces the computation cost is called *fault dropping.*

As for fault simulation techniques, *parallel*, *deductive* and *concurrent* fault simulation are well known [Bre76, Arm72, Ulr80a]. Among them, we considered it advantageous to base our technique on the parallel simulation technique for the following reasons:

1) Parallel simulators have been proved to be as efficient as concurrent simulators by the modifications such as the parallel-pattern single fault propagation (PPSFP) [Wai85]. We considered that there is much more room for further improvement in the computation efficiency of parallel simulation.

2) Simplicity in the data structure and the operations of a parallel simulator is considered to have good affinity for vector processing. Also in logic simulation, it is shown that the acceleration ratio from vectorization in the event-driven technique is larger than that in the compiler-driven techniques [Ish87].

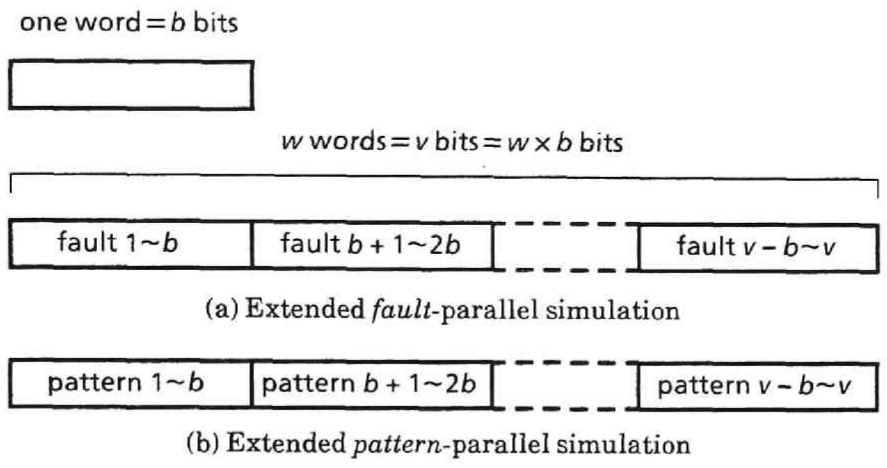


Fig. 4.1 Data structure for extended parallel simulation.

As a vector processor oriented fault simulation technique, we propose a *dynamic two-dimensional parallel fault simulation technique*, which is based on parallel simulation technique.

4.2.2 Two-Dimensional Parallel Simulation

Parallel simulation attempts to reduce the computation time by utilizing bit-oriented logical operations of the computer, and if one word of the computer consists of w bits, w gate evaluations can be performed at a time. Parallel simulation is classified into fault-parallel simulation and pattern-parallel simulation. The former simulates w fault cases for one pattern at a time by assigning each fault case to one bit. The latter, on the other hand, simulates w patterns for one fault at a time assigning each pattern case to one bit.

In either of the two, we can simulate $w \times v$ fault or pattern cases simultaneously by simply extending the simulation unit from a word to a vector consisting of v words. This simulation technique is referred to as the *extended fault-parallel simulation* or the *extended pattern-parallel simulation*. Fig. 4.1 shows the data structure for extended parallel simula-

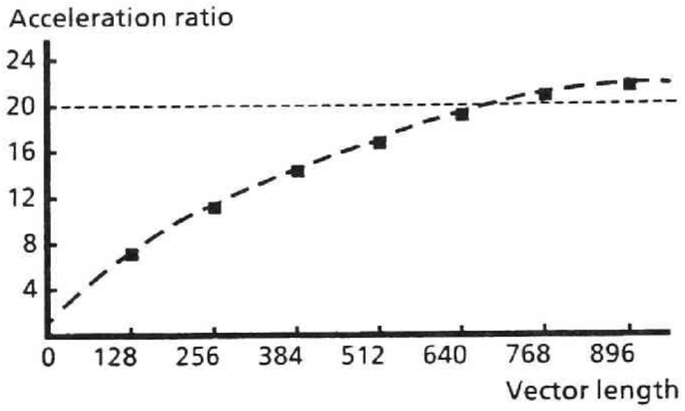


Fig. 4.2 Acceleration ratio in the extended parallel simulation.

tion. Vectorization of the simulation procedure brings about considerable improvement in simulation speed *even on a scalar processor* [Kro81]. On a vector processor, into the bargain, we can achieve much larger acceleration through the use of vector instructions. Fig. 4.2 shows the relation between vector length and acceleration ratio through vectorization on the Fujitsu FACOM VP-200 in extended pattern-parallel simulation. Simulation speed is accelerated by over 20 times provided that a large vector length (more than 700 words) is obtained.

Let w be the number of bits in a word, and f and p, respectively referred to as the fault-parallelism factor and the pattern-parallelism factor, be the number of faults and patterns simulated at a time. Since the vector length in extended pattern-parallel simulation is $\lceil p/w \rceil$ (where $\lceil x \rceil$ is the smallest integer not smaller than x), we can change it arbitrary by changing p. However, we must store p fault-free values for all internal lines and the value p is limited by the storage size if we perform selective tracing which will be described in the next section, . In extended fault-parallel simulation, on the other hand, the vector length $\lceil f/w \rceil$ is restricted by the number of undetected faults. (We assume that the states for the good machine is stored *separately.*) The number of undetected faults decreases as simulation proceeds, which shorten the vector length.

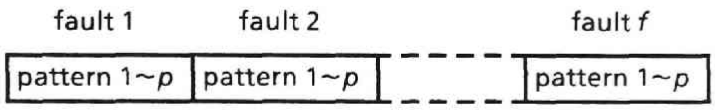


Fig. 4.3 Data Structure for two-dimensional parallel simulation.

Since the desirable vector length is considerably large, neither extended fault- nor pattern-parallel simulation may not be capable of achieving large acceleration. In such a case, we further enlarge the vector length by utilizing both fault- and pattern-parallelism. Namely, we simulate multiple faults for multiple patterns at a time. We call this technique *two-dimensional parallel simulation.* In this technique, the vector length is $\lceil f \times p/w \rceil$, which is much larger than that in simple extended parallel simulation. Fig. 4.3 shows the data structure of two-dimensional parallel simulation. We refer $\lceil p/w \rceil$ words of p patterns as a *packet.* A signal value vector of each line contains f packets corresponding to f faults.

4.2.3 Dynamic Adjustment of the Parallelism Factors

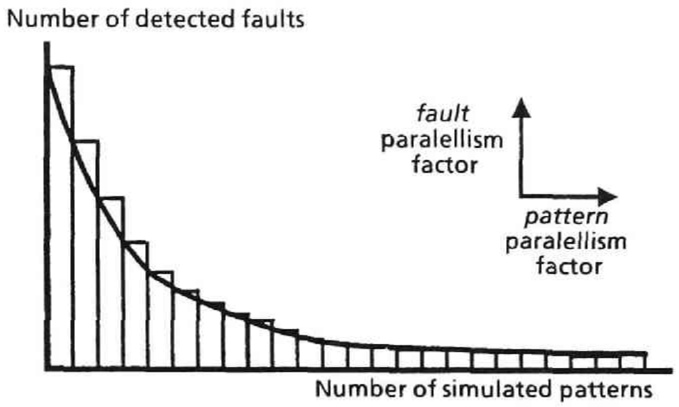
Though fault simulation can be accelerated 20 times faster by extended or two-dimensional parallel simulation, these are the results in the case where a large vector length is obtained. In applying fault simulation to test generation and coverage estimation in practice, we must consider fault dropping. Fault dropping drastically reduces the computation time, but it limits the fault-parallelism factor f and pattern-parallelism factor p (i.e. the vector length) in two-dimensional parallel simulation. Fig. 4.4 shows the relation between the number of undetected faults and the number of simulated patterns. The area of each box represents the vector length, or the computation cost in each pass (where a pass is a process of performing good simulation on p patterns and fault simulation for f faults on the p patterns). As is shown in Fig. 4.4 (a), a lot of undetected faults exist in the early passes, and large vector length is obtainable. However because the number of undetected faults decreases as simulation proceeds, the fault-parallelism factor f becomes extremely small in the later passes, and enough vector length may not be obtained. However, if we attempt to simulate with a large pattern-parallelism factor p in order to get a large vector length in the later passes, as shown in Fig. 4.4 (b), we are obliged to perform wasteful simulation in the early passes for the faults which might have been dropped if simulated with a smaller p.

Therefore, it is very hard to reduce the computation time by simple two-dimensional parallel simulation when we take account of fault dropping, because enough vector length cannot be obtained, or the computation cost increases if we intend to get a large vector length. As a solution for these problems, we propose a dynamic adjustment of the two parallelism factors f and p instead of fixing their values. We change the two parallelism factors complementarily from pass to pass according to the following strategies (as shown in Fig. 4.4 (c)).

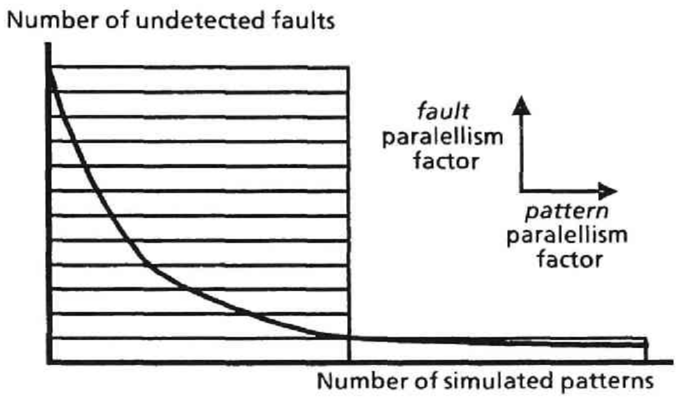
1) We set the pattern-parallelism factor small and the fault-parallelism factor large in the early passes. This is possible because there are many undetected faults in the early passes.

2) We set the pattern-parallelism factor large and the fault-parallelism factor small in the later passes.

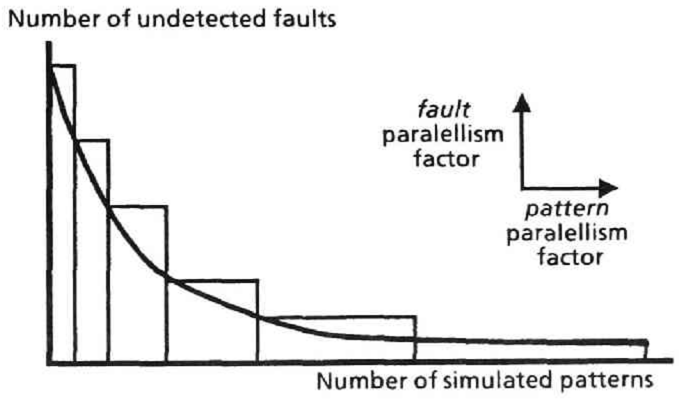
We call this technique the *dynamic two-dimensional parallel fault simulation technique*. In this technique, we can efficiently handle fault dropping and yet keep large vector length.



(a) Extended *fault*-parallel simulation.



(b) Extended *pattern*-parallel simulation.



(c) Dynamic two-dimensional parallel simulation.

Fig. 4.4 Fault dropping and the two parallelism factors.

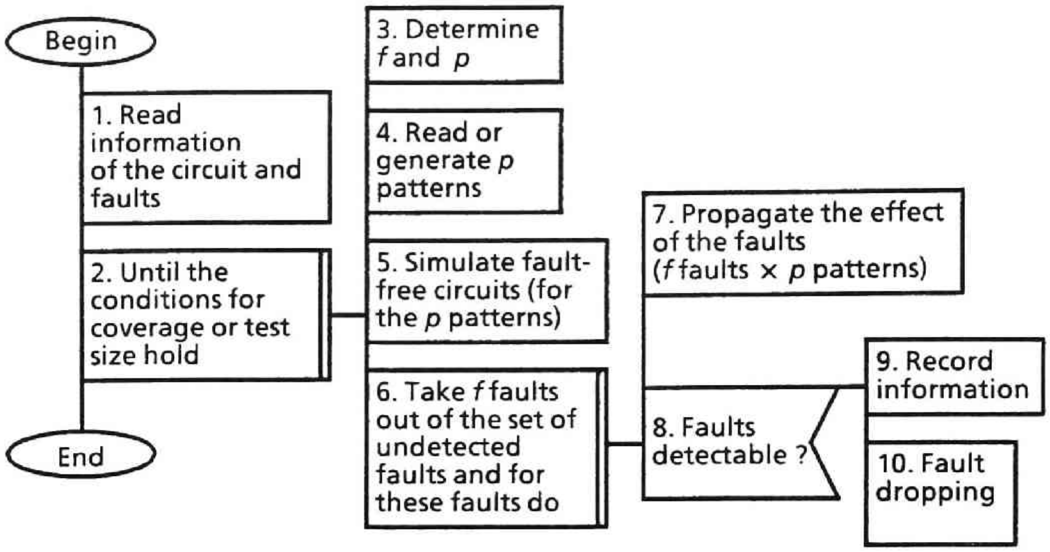

Begin
1. Read information of the circuit and faults
2. Until the conditions for coverage or test size hold
End
3. Determine f and p
4. Read or generate p patterns
5. Simulate fault-free circuits (for the p patterns)
6. Take f faults out of the set of undetected faults and for these faults do
7. Propagate the effect of the faults (f faults × p patterns)
8. Faults detectable ?
9. Record information
10. Fault dropping

Fig. 4.5 Simulation procedure.

4.3 Multiple Fault Propagation

4.3.1 Selective Tracing

Selective tracing is a well-known technique to reduce the computation time of logic and fault simulation [Bre76]. There are several ways of combining parallel fault simulation with selective tracing. Among them we choose the following strategy which is similar to PPSFP [Wai85].

1) Simulate the fault-free circuit (good simulation) and store the values of all the signal lines.

2) Compute the effect of the faults by propagating the faulty values from the fault sources to primary outputs. Since the fault-free values of all the lines have been computed at step 1, we can avoid the waste of simulating the gates whose faulty input values are the same as the fault-free ones.

In PPSFP, faults are processed one by one at step 2 (single fault propagation) by pattern-parallel simulation. In our approach, we process multiple faults at a time (multiple faults doesn't mean *simultaneous* multiple faults but multiple *single* faults). We call this method *multiple fault propagation* by the analogy of single fault propagation. The details of the simulation procedure are shown in Fig. 4.5. At step 5, we perform good simulation for p patterns by the extended pattern-parallel simulation technique, and store the values of all the signal lines. At step 7, the effect of the f faults selected at step 6 for the p patterns are propagated by the dynamic two-dimensional parallel fault simulation technique.

4.3.2 Implementation of Selective Tracing

Our selective tracing consists of the following two concepts:

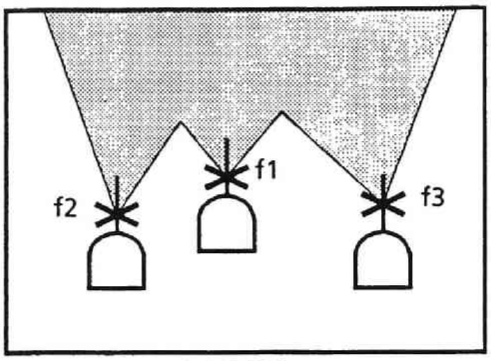
LIM: limit the gates for fault simulation to the gates in the fault effect cones, and

DISC: discontinue fault propagation when the effect of faults disappear.

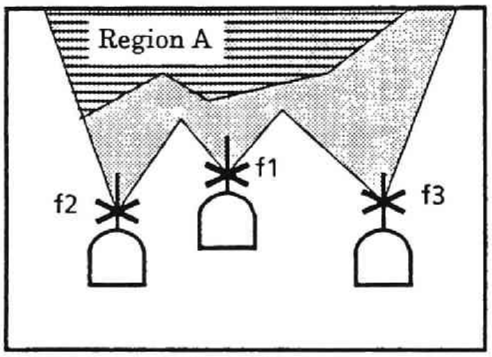
A fault effect cone of a fault is defined as a set of gates on the paths from the faulty line to primary outputs. It is determined by connectivity information only. The shaded region in Fig. 4.6 (a) shows the union of the fault effect cones of some faults. All we have to simulate are only the gates in this region. On the other hand, DISC attempts to reduce the computation cost using the dynamic information of faulty values. The disappearance of the effect of faults can not be found before the simulation. In Fig. 4.6 (b), since faults do not effect on a set of gates in the region A, we avoid wasteful simulation on them.

Implementation of LIM

LIM is easily realized by adopting the event-driven simulation with level mapping technique (we assume the gates in the circuit are levelized in the preprocessing stage). We prepare a *evaluation gate list* for each level



(a) LIM.



(b) DISC.

Fig. 4.6 Selective tracing.

of the circuit, which is initially empty. Fault propagation is performed by the following procedure.

1) Register the gates that have faults on their input or output lines to a evaluation gate list of the corresponding level.

2) Repeat 3)$\sim$5) until all the evaluation gate lists become empty.

3) Take a gate g out of a evaluation gate list in the ascending order of the level number.

4) Evaluate the faulty value of g.

5) Register the successor gates of g to evaluation gate lists of their corresponding levels. If the successor gate is a primary output, we don't register it.

In the multiple fault propagation, we have to take care of the overlap of fault effect cones (the shaded region of Fig. 4.6 (a)). When all fault effect cones of all faults are equivalent, the gate evaluation count is equal to that in the single fault propagation. However we are forced to evaluate more gates wastefully when the intersection of all cones is small. Therefore, it is desirable to select f faults whose effect cones overlap each other at step 6 in Fig. 4.5. In order to avoid overhead during simulation, we simply take f faults successively out of the undetected fault list. Instead, we place the neighboring faults close in the undetected fault list at the preprocessing stage. It is considered to be effective to group faults in a fanout-free region in this preprocessing stage. However, currently, we have not implemented the idea but we simply order faults by traversing signal lines in depth first manner starting from primary inputs.

Implementation of DISC

In the case of single fault propagation, DISC means to discontinue the fault propagation when signal values of a particular line in the assumed

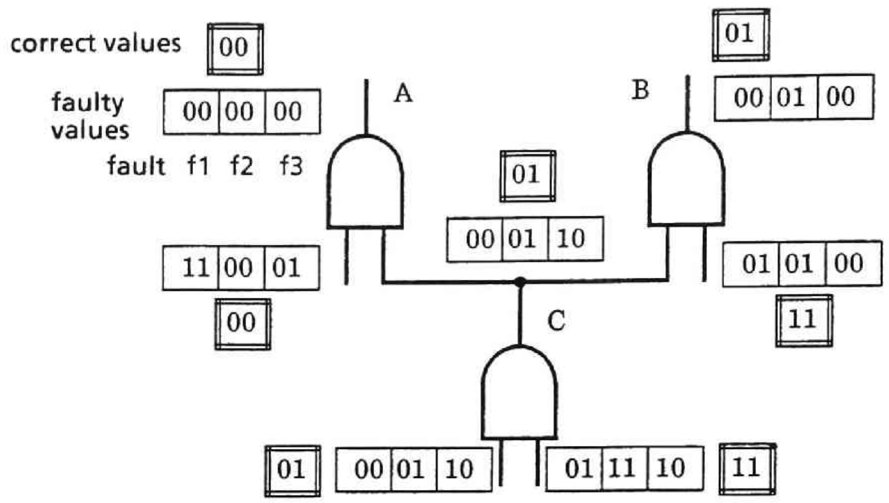


Fig. 4.7 DISC$_{\mathrm{PATH}}$ and DISC$_{\mathrm{FAULT}}$.

fault case are identical to those in the fault-free case for all patterns. In multiple fault propagation, this notion is further divided into the following two.

DISC$_{\mathrm{PATH}}$: We discontinue the fault propagation for a path when signal values of a particular line for all the fault cases are identical to those for the fault-free case for all patterns. This discontinuance is called the discontinuance of propagation for a path.

DISC$_{\mathrm{FAULT}}$: We discontinue the fault propagation for a fault when signal values of all propagation paths for a particular fault case are identical to those for the fault-free case for all patterns. This discontinuance is called the discontinuance of propagation for a fault.

Fig. 4.7 shows an example. Fault-free values are enclosed in doubly lined boxes and faulty values in single lined boxes. Each box represents a packet. The values in the first box of faulty values are the faulty values caused by a fault **f1**, the second **f2**, and the third **f3**. At line A, since all faulty packets are the same as the correct packet, we do not have to simulate the successors of line A. This type of discontinuance is DISC$_{\mathrm{PATH}}$.

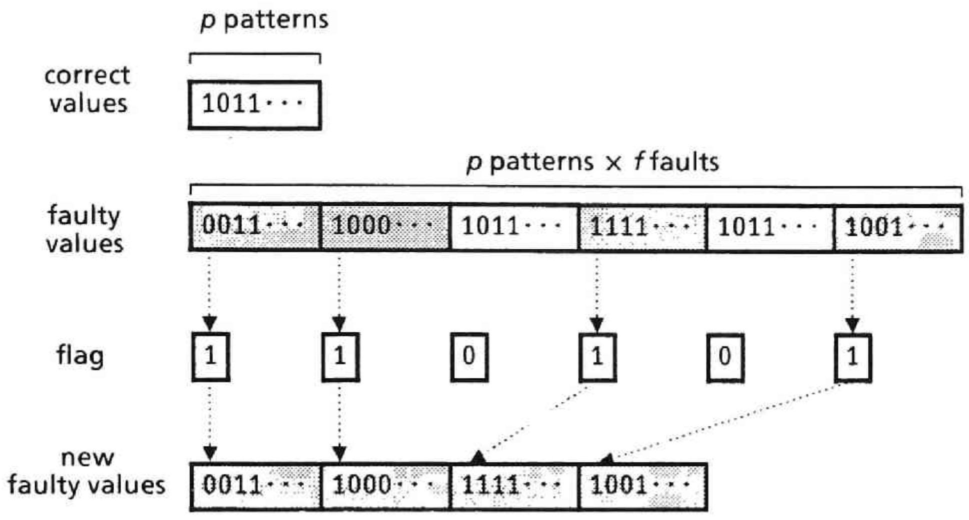
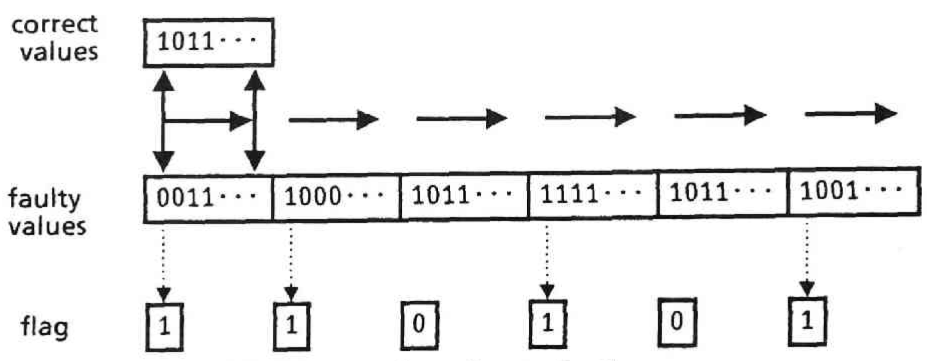


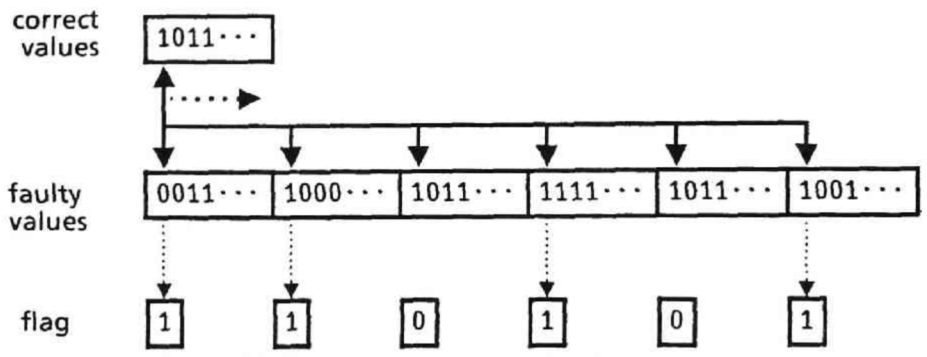
Fig. 4.8 Realization of $\mathrm{DISC}_{\mathrm{FAULT}}$.

On the other hand, since the effects of fault **f2** disappear at both line A and B, if **f2** does not affect the other lines, we can conclude that all the effects of **f2**disappear. Therefore, we do not have to propagate the effect of **f2** any more. This type of discontinuance is $\mathrm{DISC}_{\mathrm{FAULT}}$.

$\mathrm{DISC}_{\mathrm{PATH}}$ can be easily realized by avoiding the registration of the successor gates to the evaluation gate list when the gate output values for all fault cases are identical to those in the fault-free case. Implementation of $\mathrm{DISC}_{\mathrm{FAULT}}$ is somewhat complicated. In order to find the disappearance of all the effects of a fault on all its propagation paths, we store the largest level number on which the fault affects, and compare the number and the current level number. If the number is less than the current level number, we can find the disappearance of the fault effects. When the all effects of a fault disappeared, we eliminate the packet which correspond to the fault from all signal vectors. This elimination is performed as shown in Fig. 4.8. In Fig. 4.8, shaded packets still have the effects of faults and the information is shown by **flag**. We can vectorize the elimination by vector compress operation and can perform it extremely fast. However assume that 100 faults are simulated currently, and the



(a) Pattern-oriented vectorization.



(b) Fault-oriented vectorization.

Fig. 4.9 Complimentary Vectorization.

effects of only one of them have disappeared. Then we are obliged to make copies of 99 packets for each signal vector. Since this manipulation can be an overhead, we eliminate the packets only when the effects of over 30% of the faults under simulation have disappeared.

Complementary Vectorization

In order to realize DISC$_{\text{PATH}}$ and DISC$_{\text{FAULT}}$, we have to compare the fault-free signal values with the faulty signal values at each signal line and have to set the **flag** in Fig. 4.8 if they are different. The process of the comparison can be vectorized in the following two ways according to vectorization parameters.

pattern-oriented vectorization: We compare the packet of fault-free values with each of the f packets of faulty values, one faulty packet at a time, as shown in Fig. 4.9 (a). The comparison is a vector operation with vector length $\lceil p/w \rceil$ and is repeated f times. Since the vector length depends on the number of patterns p, this method is called *pattern-oriented vectorization*.

fault-oriented vectorization: We compare each of the $\lceil p/w \rceil$ words in the packet of fault-free values with the f corresponding words from all the faulty packets, one word at a time, as shown in Fig. 4.9 (b). The comparison is a vector operation with vector length f and is repeated $\lceil p/w \rceil$ times. Since the vector length depends on the number of faults f, this method is called *fault-oriented vectorization*.

Since the maximum vector length $\lceil f \times p/w \rceil$ is limited by the available storage, if f is large, p should be small, and vice versa. Therefore, the two methods described above have a complementary relation, that is, if one is effective, the other is not. We choose the more effective of the two methods according to the values of f and p at that time.

Also in the elimination of packets in DISC$_{\text{FAULT}}$, similar two complementary methods can be considered, and we choose the more effective one according to f and p.

4.3.3 Determination of Parallelism Factors

It is a very important process in the dynamic two-dimensional parallel fault simulation to determine the parallelism factors f and p at each pass (step 3 in Fig. 4.5). We must take the following facts into consideration.

1) Small p makes the effects of fault dropping large.

2) Small p saves the memory area to store p fault-free values of all the signal lines.

3) Small p makes the condition of discontinuance of propagation easy to satisfy.

4) Large p makes the good simulation efficient since the vector length is p.

5) Small f reduces the wasteful simulation when we limit the gates for fault simulation to the gates in the fault effect cones.

Although it is desirable to determine the optimum f and p taking all these conditions into account, it is difficult to measure these conditions completely. We, therefore, measure the decreasing ratio d of the number of undetected faults at the previous path, and determine p as double, same, or a half of p_{prev} according to d, where p_{prev} is the pattern parallelism factor in the previous pass. Namely,

$$d = \frac{\text{the number of undetected faults after the previous pass}}{\text{the number of undetected faults before the previous pass}}$$

and

$$p = \begin{cases} \min(2 \times p_{prev},\, b \times maxp) & \text{if } d_2 < d \\ p_{prev} & \text{if } d_1 \leq d \leq d_2 \\ \max(p_{prev}/2,\, 1) & \text{if } d < d_1 \end{cases}$$

$$f = w \times maxfp/p,$$

where w is the number of bits in a word, $maxp$ and $maxfp$ are constants determined by the available storage size. d_1 and d_2 in the above formulas are

$$d_1 = \left(\frac{r \cdot p_{prev}}{r \cdot p_{prev} + 2w}\right)^2 \quad \text{and} \quad d_2 = \frac{r \cdot p_{prev}}{r \cdot p_{prev} + w}, \quad \text{where } r = 0.1.$$

These bounds are obtained by minimizing $P = C \times T$, where C is the total count of operations for fault simulation at the next pass under the assumption that the number of faults will decrease by d times for every p_{prev} patterns, and T is the computation time per operation on the vector processor, approximated by

$$T = \frac{\alpha(p/w) + \beta}{p},$$

where α and β are constants characterizing the performance of the pipeline.

We briefly show the derivation of the d_2. If we increase the parallelism factor to $2p_{prev}$, the total count of operations C_2, computation time per operation T_2, and the total computation time P_2 at the next pass are computed as follows.

$$\begin{aligned} C_2 &= 2p_{prev} \cdot d, \\ T_2 &= \frac{\alpha(2p_{prev}/w) + \beta}{2p_{prev}}, \\ P_2 &= C_2 \times T_2 = d(2\alpha p_{prev}/w + \beta). \end{aligned}$$

On the other hand, if we do not change the parallelism factor, the total computation time P_1 at the next 2 passes becomes as follows.

$$\begin{aligned} C_1 &= p_{prev}(d + d^2), \\ T_1 &= \frac{\alpha(p_{prev}/w) + \beta}{p_{prev}}, \\ P_1 &= C_1 \times T_1 = d(\alpha p_{prev}/w + \beta). \end{aligned}$$

By solving $P_2 < P_1$, we have

$$\frac{\alpha p_{prev}/w}{\alpha p_{prev}/w + \beta} < d.$$

By replacing α/β by r, we get

$$\frac{r \cdot p_{prev}}{r \cdot p_{prev} + w} < d.$$

The bound d_1 is derived in the same way. As for the value 1.0 of r, we decided it by experiments.

4.4 Implementation and Experiments

4.4.1 Simulation Speed

We have implemented a fault simulator based on the dynamic two-dimensional parallel simulation technique on the Fujitsu FACOM VP-200 vector pro-

Table 4.1 Versions of the selective trancing and execution mode.

	Vector	*Vector*$^+$	*Vector*$^{++}$	*Scalar*$^{++}$
LIM	–	○	○	○
DISC	–	–	○	○
Execution mode	vector execution	vector execution	vector execution	scalar execution

cessor in Fortran77 to create some experiments for its performance evaluation. We simulated the ten benchmark circuits [Brg85f] and measured the CPU time required for simulating 512K (16K words) random patterns. Almost all of the main loops of our simulator were vectorized. Henceforth we refer to this vectorized version as $Vector^{++}$. In order to evaluate the effect of vectorization, we also performed simulation without vector instructions. We call this execution mode scalar execution and denote it by $Scalar^{++}$. Also for the purpose of evaluating the effect of selective tracing, we also performed simulation with no selective tracing and simulation only with LIM. The former is denoted by $Vector$, and the latter is denoted by $Vector^+$. The versions of selective tracing and the execution modes are summarized in Table 4.1.

Table 4.2 shows the results. The maximum vector length $maxfp$ is a parameter which the user gives and was set to 1024 in every mode in our experiments. The maximum pattern-parallelism factor $maxp$ is another parameter and was set to 256 except for $Vector$. Since selective tracing is not performed in $Vector$, we can get a larger $maxp$ with the same storage size, and $maxp$ was set to 1024. Table 4.2 gives the following conclusions.

1) The comparison between $Scalar^{++}$ and $Vector^{++}$ tells us the effect of vectorization. From this comparison, we can see that the simulation speed is accelerated by 10 to 15 times through vectorization. We can

Table 4.2 Results of the simulation on 512K random patterns.

Circuit	Number of gates	Number of faults	Final coverage [%]	Number of undetected faults	Simulation CPU [sec]			
					Vector	Vector^{+}	Vector^{++}	Scalar^{++}
C432	203	524	99.24	4	.176	.280	.275	2.560
C499	275	758	98.94	8	.461	.791	.571	6.236
C880	469	942	100.0	0	.179*	.462	.490	4.651
C1355	619	1574	99.49	8	.930	1.355	.871	10.979
C1908	938	1879	99.52	9	1.930	1.381	1.387	16.026
C2670	1566	2447	91.37	237	57.695	10.318	8.330	99.474
C3540	1741	3428	96.00	137	19.037	15.440	3.935	48.463
C5315	2608	5350	98.89	59	17.864	6.962	3.489	41.818
C6288	2480	7744	99.56	34	15.605	22.388	2.604	38.194
C7552	3827	7550	96.89	235	114.868	19.939	11.518	154.885

***Simulation was stopped as soon as all the faults are detected.**

conclude that the performance of the vector processor is thoroughly brought out in large scale simulation.

2) The comparison between $Vector$ and $Vector^{+}$ tells us the effect of LIM. Also the comparison between $Vector^{+}$ and $Vector^{++}$ tells us the effect of DISC. From these comparisons, we can conclude that the two techniques have a complementary effect, i. e. when one has little effect, the other has large effect, and vice versa.

3) The comparison between $Vector$ and $Vector^{++}$ tells us the effect of selective tracing. Although the speed is slowed down a little in some small circuits, the speed is accelerated about 10 times, in circuits conventionally requiring large computation cost.

Compared with the result in [Wai85], in which PPSFP is implemented on the IBM 3081, our simulator is 7 to 19 times faster. Furthermore, our simulator is portable since it is implemented using only the basic facilities of recent vector processors. The simulator implemented on the VP-200 is executable also on the Hitachi HITAC S-810/20 vector processor without modifying the program. The Hitachi HITAC S-810/20 has about the same potential performance and our simulator achieved almost the same performance as on the VP-200.

4.4.2 Required Storage Size

The storage requirements are mainly for the vectors containing fault-free and faulty values of internal lines. They increase with the vector length at step 5 and step 7 in Fig. 4.5. At step 5, p fault-free values of all gates must be stored in order to perform selective tracing at step 7. Let n be the number of gates in a circuit, then required storage size is $n \times maxp$ [word]. Since we simulate f faults on p patterns at step 7, we need a vector of $maxfp$ [word] per gate. However because we don't have to store vectors of all gates at step 7, we can reduce the required storage by reusing storages for the vectors. The required storage is $M_v \times maxfp$ [word], where M_v is the maximum number of signal vectors which we must store at some time in fault propagation and where M_v is much smaller than n. The above experiment required 10 MB for the largest circuit C7552.

4.5 Remarks and Discussions

As a vector processor oriented fault simulation algorithm, we proposed a *dynamic two-dimensional parallel simulation technique.* We succeeded in obtaining a large vector length without reducing the computation efficiency by introducing a selective tracing method based on *multiple fault propagation.* Experimental results tell us that fault simulation is accelerated by 10$\sim$15 times through vectorization and that our simulator is extremely fast in simulating large circuits on many patterns. Since the critical path tracing for fanout-free region is also vectorizable, we can combine our simulation method with the concept in [Ant87] so as to further enhance the performance. When we use our fault simulator in combination with algorithmic test generation, we can not expect large acceleration because the pattern parallelism is limited. However, we can make the most of our simulator in test generation using random patterns, coverage estimation of a large set of random patterns or a built-in self test design, where large pattern parallelism is available.

Vector processors seem to have great potential for not only numerical computation but also for combinational problems in the area of CAD for digital systems. There will be a lot of earnest researches to develop vector processor oriented algorithms for variety of combinational problems. In converse, it is also important to improve architecture of vector processors suitable to process combinational problems.

Chapter 5

Computational Complexity of Logic Simulation Problems

5.1 Introduction

Design verification is one of the most laborious processes in hardware development. As is discussed in the preceding chapters the computation cost due to the size of the circuit under verification has been and will be one of the primary problems in design verification. On the other hand, *accuracy* of the simulation is also an important issue. Especially in design verification of asynchronous circuits which operate based on subtle timing relations, much more laborious modeling of delay and time and also much more computation cost are required than in that of synchronous circuits.

In the verification concerned with timing there are close relations among models of delay and time, accuracy of verification results and required computation cost. In a simple modeling which require smaller computation cost, design errors may be overlooked or possibilities of design errors may be indicated even for correct designs. One example is the handling of delay whose actual value is unknown and is specified with minimum and maximum values. In logic simulation the min/max delay model is employed to handle such uncertainty. The model allows relatively fast verification but it is well known that the verification results are often too pessimistic [Bre76]. Although there are a lot of attempts to overcome this problems, few discussions have been made on what is the essence of the difficulty and how difficult or how much computation cost is required to solve the problem completely. Another important issue is modeling of time. Many of the existing verification systems are based on a *discrete time model* [Cer89, Hir89, Nak87, Kim88]. There are also few discussions on the point if the discrete time model provides accurate result as compared with a *continuous time model* or if there is a difference in the computation cost for verification between the two models.

In this chapter we take *hazard detection problems* as an example so as to discuss the relation among models of delay and time, accuracy of verification result and computation cost for the verification [Ish88]. Especially we focus on delay model in which the actual delay values are uncertain and are specified with their minimum and maximum values. We also discuss the difference of a discrete time model and a continuous time model. We show that the problem of detecting hazards on combinational circuits under uncertain delay assumption is computationally intractable (NP-hard) and hence that it is difficult to solve the problem by a simple extension of the min/max delay simulation technique [Bre76]. We also show that there is an essential difference in the verification results obtained based on the discrete time model and the continuous time model. The verification result can be more optimistic in the discrete time model than in the continuous time model. However we prove that the discrete time model will provides the same accuracy of the continuous time model with respect to the hazard detection problem by making the time unit small . We clarify to what extent we must make the unit time small. We further discuss the computation cost that we have to pay in order to make the two models equivalent.

In section 5.2 we define models of delay and time and formalize the hazard detection problems. In section 5.3 we discuss computational complexity of the problem of the uncertain delay and discrete time model. We examine the relation between the discrete time model and the continuous time model in section 5.4 and discuss the computational complexity of the continuous time model.

5.2 Hazard Detection Problem and Modeling of Delay and Time

5.2.1 Hazard Detection Problem

In this chapter we discuss hazard detection problems for combinational circuits. In the following discussions, we refer to a combinational circuit simply as a circuit. The number of fan-in's and fan-out's in circuit C is bounded by a constant which is independent of the number of gates in C. Let $\vec{v_x}$ and $\vec{v_y}$ be input assignments to C. A hazard is an occurrence of more than one change of signal values on some of the primary outputs of C for the change of input assignment $\vec{v_x} \rightarrow \vec{v_y}$.

Def 5.1 A Hazard detection problem for a specific input change is defined as follows.

Instance: Circuit C and two input assignments $\vec{v_x}$ and $\vec{v_y}$.

Question: Are there possibilities of hazards for a change of input assignment $\vec{v_x} \rightarrow \vec{v_y}$? □

In discussing the specific input change, we can assume input without loss of generality that a circuit has only one primary. We also assume, for simplicity, that a circuit has only one primary output, which does not affect the results in this chapter. Thus we denote the set of gates which construct circuit C as $G_C = \{g_0, g_1, \cdots, g_n\}$ where g_0 is the primary input [1] and g_n is the gate whose output is connected with the primary output.

[1]We treat the primary input as a gate with no inputs and a single output, for simplicity

5.2.2 Modeling of Delay and Time

Modeling of Uncertainty of Delay

In actual logic circuits, delay values of gates vary depending on the difference of process conditions or usage conditions. In order to express this uncertainty we describe the delay value of g_i by its minimum and maximum values ${d^{min}}_i$ and ${d^{max}}_i$, respectively, where ${d^{min}}_i$ and ${d^{max}}_i$ are non-negative *integers* which satisfies ${d^{min}}_i \leq {d^{max}}_i$. We do not allow real numbers for the delay bounds because it is impossible to describe them within finite length. We define the following three delay models according to the constraints on ${d^{min}}_i$ and ${d^{max}}_i$.

Exact delay model : For each gate g_i, ${d^{min}}_i = {d^{max}}_i$ holds. Namely this is an ideal model where each gate takes the exact delay values as specified.

Uncertain delay model : Each gate takes an arbitrary delay value d_i which satisfies ${d^{min}}_i \leq d_i \leq {d^{max}}_i$. There are no constraints on ${d^{min}}_i$ and ${d^{max}}_i$.

Restricted uncertain delay model : This is an uncertain delay model where ${d^{min}}_i$ and ${d^{max}}_i$ satisfies the following constraints for non-negative constants c^{min} and c^{max}.

$$c^{min} \leq \frac{{d^{max}}_i - {d^{max}}_i}{{d^{max}}_i + {d^{max}}_i} \leq c^{max}.$$

This inequality expresses that the ratio of width of delay uncertainty to the magnitude of the delay value is not extremely large nor small, which is a more realistic assumption than the simple uncertain delay model.

One important assumption in this chapter is that the variation of delay values is *static*. Namely the delay value of each gate may be uncertain but the value is *constant* and do not change with the progress of time. We do not consider *inertia* delay either in this chapter.

(2) Modeling of Time

We discuss the following two models of time.

Discrete time model : We assume that the domain of the time is the set of *integers*. Namely, the delay value of each gate d_i is an integer within the bounds ${d^{min}}_i$ and ${d^{max}}_i$.

Continuous time model : We assume that the domain of the time is the set of *real numbers*. The delay value of each gate d_i can be a real number.

In the exact delay model there is no difference between the two time models because the bounds ${d^{min}}_i$ and ${d^{max}}_i$ (namely the delay value d_i) are specified as integers. On the other hand in the bounded delay models there can be differences because there are infinite time points between ${d^{min}}_i$ and ${d^{max}}_i$ in the continuous time model while there are only finite time points in the discrete time model.

(3) Modeling of Magnitude of Delay Values

We discuss the following two models as for the magnitude of delay values.

Constant delay model : For each gate g_i, ${d^{min}}_i$ and ${d^{max}}_i$ are specified by binary integers of c bits, where c is a constant which is independent of the number of the gates in the given circuit. Namely ${d^{max}}_i = O(1)$.

Exponential delay model : For each gate g_i, ${d^{min}}_i$ and ${d^{max}}_i$ are specified by $p(n)$ bit binary integers, where $p(n)$ is an arbitrary polynomial of n, the number of the gates in the given circuit. Namely ${d^{max}}_i \leq O(2^{p(n)})$.

The delay values of gates in actual integrated circuits are proportional to CR in the first order approximation where capacitance C and resistance R are approximately proportional to the area on the chip. Therefore it is not appropriate to discuss the extremely large delay value as in the exponential delay model. However we introduce the delay model as a mathematical model, which is necessary in discussing the difference between the discrete time model and the continuous time model.

5.2.3 Notation

We can consider hazard detection problems for the twelve models which are the combinations of the two time models, the tree uncertainty models and the two magnitude models. In the following discussion we abbreviate the name of the problems as follows:

$$\mathrm{SHD}\begin{bmatrix}\mathrm{Cnst}\\ \mathrm{Exp}\end{bmatrix}\begin{bmatrix}\mathrm{Exct}\\ \mathrm{Unc}\\ \mathrm{Rst}\end{bmatrix}\begin{bmatrix}\mathrm{Dscr}\\ \mathrm{Cont}\end{bmatrix},$$

where,

Cnst/Exp are the abbreviations of the constant time model and the exponential time model, respectively,

Exct/Unc/Rst are the abbreviations of the exact delay model, the uncertain delay model and the restricted uncertain delay model, respectively, and

Dscr/Cont are the abbreviations of the discrete time model and the continuous time model.

For example SHDCnstUncDscr means the hazard detection problem of the constant and uncertain delay and discrete time model.

5.3 Hazard Detection Problems of the Discrete Time Model

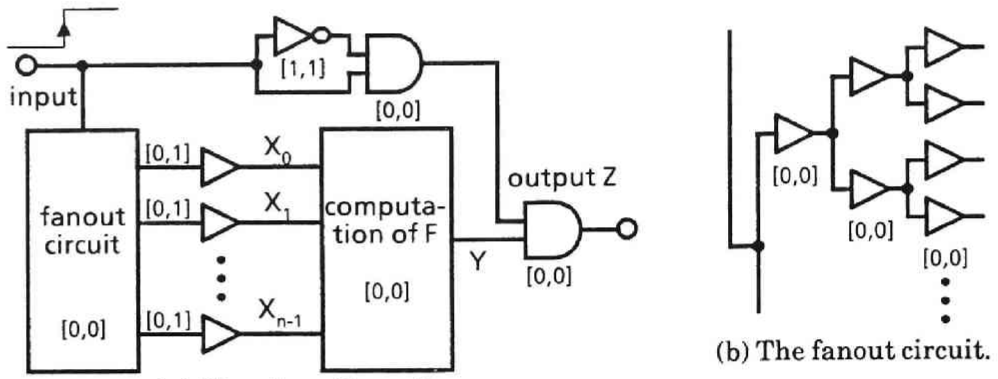
In this section we discuss the computational complexity of the hazard detection problems for the constant delay model and discrete time model. We consider the variation of delay uncertainty under this assumption. It is shown that the problems for the exact delay model (SHDCnst*Exct*Dscr) can be solved within feasible time but the problems for the uncertain delay model and the restricted uncertain delay model (SHDCnst*Unc*Dscr, SHDCnst*Rst*Dscr) are NP-complete.

Th 5.1 SHDCnst*Exct*Dscr belongs to P (deterministic polynomial time).

[Proof] Execute logic simulation and examine the signal changes on the output. Let d^{max} the largest one of ${d^{max}}_1$, ${d^{max}}_2$, $\cdots$, ${d^{max}}_n$. Then the signal value on the primary output will be stable in $n \times d^{max}$ unit times after the input change, where n is the number of gates in the circuit. Since simulation for a unit time is carried out in $O(n)$ time, total computation time is $O(n^2)$. □

Logic simulation using 5-valued logic [Bre76] is used for timing verification taking the delay uncertainty into account. However it is known that accurate result is not obtained by this simulation algorithm as will be discussed in detail in the next chapter. We first show an upper bound of the computation time to obtain accurate result and then show a lower bound. Even if the delay values have uncertainty, we can get an accurate result by simulating all the possible combinations of delay values. In the discrete time model this is possible because we can enumerate all the delay values of gates.

Lem 5.1 SHDCnst*Unc*Dscr, SHDCnst*Rst*Dscr are *in NP* (nondeterministic polynomial time).



(a) Circuit configuration.

(b) The fanout circuit.

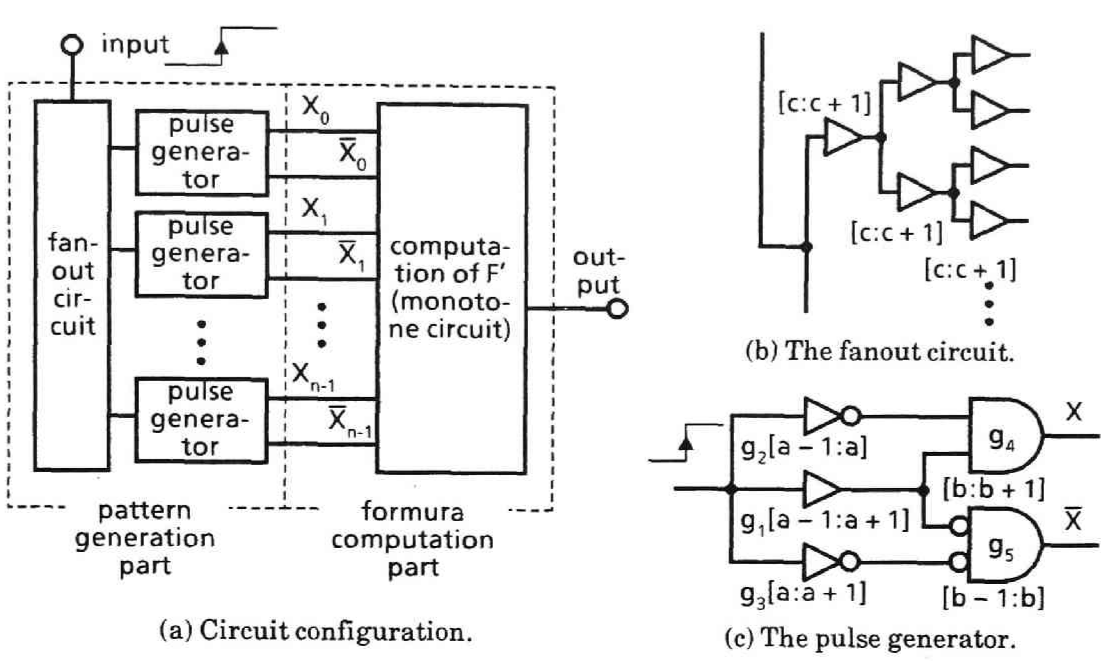
Fig. 5.1 The circuit for SAT of F.
(The uncertain delay model)

[Proof] Guess a combination of delay values which causes a hazard on the primary output and verify the existence of hazards. Since the hazard detection problems for exact delay values are solvable in polynomial time, the problems are solvable in polynomial time by a nondeterministic Turing machine. □

As for a lower bound we have obtained the result that the problems are computationally difficult.

Lem 5.2 SHDCnst*Unc*Dscr and SHDCnst*Rst*Dscr are *NP-hard.*

[Proof] We first show the proof for the uncertain delay model. We show that satisfiability problem of Boolean formulas in conjunctive normal form (CNF-SAT) is reducible into SHDCnst*Unc*Dscr. Namely, for a given CNF formula F, we construct a circuit which has a delay combination to cause hazard if and only if F is satisfiable. The circuit is shown in Fig. 5.1. On line $X_0, X_1, \cdots, X_{n-1}$ all the combinations in $\{0,1\}^n$ can be generated at the time when the input changes from 0 to 1. The output Z stays 0 if F is not satisfiable. On the other hand, there is a possibility of 1-hazard on Z if F is satisfiable. Also note that the circuit is constructed



(a) Circuit configuration.

(b) The fanout circuit.

(c) The pulse generator.

Fig. 5.2 The circuit for SAT of F.
(The restricted uncertain delay model)

from F in polynomial time.

We take the same approach as for the restricted uncertain delay model. We construct the circuit shown in Fig. 5.2. The circuit consists of an input generation part and a formula computation part. The input generation part consists of n pulse generators. The both outputs of the pulse generators are 0 in stable state. When the input changes from 0 to 1, a 1-pulse can be generated on either of the two outputs; a 1-pulse can be generated on X if the delay of g_1 is smaller than a and on $\overline{X}$ if the delay of g_1 is larger than a. The formula computation part is a monotone combinational circuit which computes the $2n$ input logic function f' that satisfies

$$f'(x_0, x_1, \cdots, x_{n-1}, \overline{x_0}, \overline{x_1}, \cdots, \overline{x_{n-1}}) = f(x_0, x_1, \cdots, x_{n-1}),$$

where f is the logic function expressed by Boolean formula F. The combinational circuit for f' is easily obtained by replacing literal $\overline{x_i}$ with the

$(n+i+1)$-th variable of f'. We assume that the combinational circuit is constructed only with AND and OR gates only and that its inside is appropriately synchronized; that is, the pulses given on inputs at the same time reach the same level of the gates in the circuit. In the stable state the output of the circuit is 0 in the stable state. Suppose f is satisfiable. Then a hazard occurs on the output when the pulses corresponding to an assignment that satisfies f are generated. On the other hand, the output of the circuit stays 0 if f is not satisfiable. Thus the satisfiability problem of F is reduced into the hazard detection problem of the circuit. Also note that this circuit can be constructed from F in polynomial time. □

From Lem 5.1 and Lem 5.2 we can lead the following theorem.

Th 5.2 SHDCnstUncDscr and SHDCnstRstDscr are *NP-complete.* □

5.4 Relation between the Continuous Time Model and the Discrete Time Model

In order to clarify the computational complexity of hazard detection problems for the continuous time model, we discuss relation between the continuous time model and discrete time model in this section.

5.4.1 Difference between the Continuous Time Model and the Discrete Time Model

In the exact delay model there is no difference between the continuous and discrete time models because the delay bounds (namely the delay values) are specified by integers. However, in the uncertain delay models there can be differences. As for the lower bounds, we can obtain the same results as in the discrete time model by the same proof as in Lem 5.2.

Lem 5.3 SHDCnstUnc*Cont* and SHDCnstRst*Cont* are NP-hard. □

On the other hand, as for upper bounds, we can not apply the same proof as in Lem 5.1 because there are infinite time points between the bounds in the continuous time model and it is impossible to enumerate the possible of delay values directly. Another problem is that more than $O(2^n)$ signal changes can occur on a single line in the continuous time model while only $d^{max} \times n$ signal changes can occur on a single line in the discrete time model.

The first problem can be solved by enumerating the possible inequality relations among the linear combinations of the delay values, instead of making vain attempt to enumerate the delay values. However, due to the second problem, it takes more than $O(2^n)$ computation time to examine each cases, even if we can enumerate the possible cases. This leads to the following proposition on a upper bound of the computation time of hazard detection problems of the continuous time model, which will be improved in the later section.

Prop 5.4 SHDCnstUnc*Cont* and SHDCnstRst*Cont* are in *nondeterministic exponential time.* □

Then a question is if there is actually difference between the discrete time model and the continuous time model. The answer to the question is *yes*. Fig. 5.3 shows such an example. In this example, a hazard can occur in the continuous time model while a hazard never occurs in the discrete time model.

Note that the hazard can occur in the discrete time model if we set the minimum unit time as half of that in the example. Thus the discrete time model is considered to have the same ability of that of the continuous time model if we make the unit time fine enough. Namely the discrete time model can be regarded as an *approximation* of the continuous time model. From the standpoint of the trade-offs between computation cost and accuracy of timing verification, it is an important issue to clarify

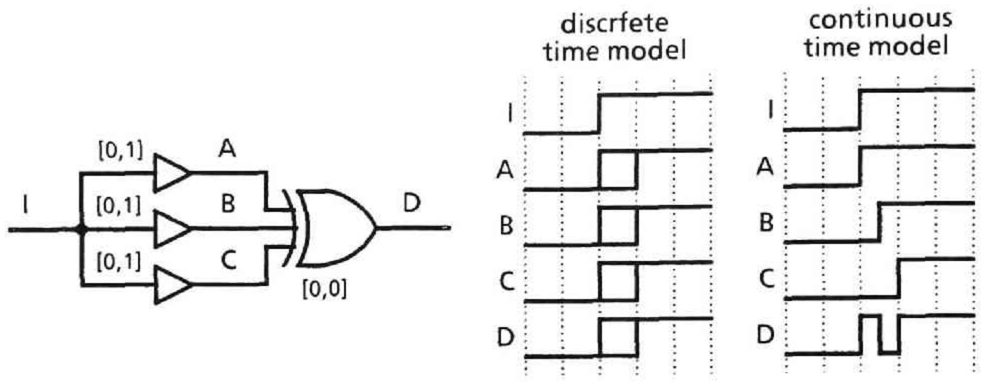


Fig. 5.3 Difference between the discrete and continuous time models.

the computational complexity of the hazard detection problems of the continuous time model. In this chapter we try to clarify relation between the two models by investigating to what extent we must make the unit time fine in order to make the two time models equivalent.

In the example in Fig. 5.3, we can observe the same hazard by making the magnitude of delay values the twice with keeping the unit time unchanged, instead of making the unit time the half. Namely, to make the unit time fine has the same effect as to make the magnitude of delay large with keeping the unit time unchanged. This is the reason why we introduced the exponential delay model. In this section we show the following two are equivalent and thus we can reduce the hazard detection problems of the continuous delay model into those of the discrete time model.

1) A hazard can occur on a circuit in the continuous time model.

2) For some integer m, a hazard can occur on the circuit in the discrete time model whose all delay bounds are multiplied by m, and m is not more than $2n^n$.

For this purpose we first show that the feasibility problem of a certain linear inequality system is equivalent to a hazard detection problem of the uncertain delay models.

5.4.2 The Linear Inequality System Equivalent to a Hazard Detection Problem

Def 5.2 Let x_1, x_2, $\cdots$, x_n be variables which represent the delay value of gates g_1, g_2, $\cdots$, g_n, respectively, and let $\vec{x} = (x_1, x_2, \cdots, x_n)$. Each x_i takes integer value in the case of the discrete time model and a real value in the case of the continuous time model. The variables satisfy the following constraints R.

$$R = \bigwedge_{i=1}^{n} ((d^{min}{}_i \le x_i) \wedge (x_i \le d^{max}{}_i)).$$

□

Def 5.3 We define sets of linear combinations of the n variables, T and T', whose coefficients are in {0,1} and in $\{0, \pm 1\}$, respectively. We also define D as set of constants.

$$\begin{aligned} T &= \{\delta_1 x_1 + \delta_2 x_2 + \cdots + \delta_n x_n \mid \delta_i \in \{0,1\}\}, \\ T' &= \{\eta_1 x_1 + \eta_2 x_2 + \cdots + \eta_n x_n \mid \eta_i \in \{0, \pm 1\}\}, \\ D &= \{\pm d_i^{min}, \pm d_i^{max} \mid i = 1, 2, \cdots, n\}. \end{aligned}$$

□

If we assume the input change occurs at time 0, the time of an event which occur in the circuit is represented by the sum gate delays. Then all the expressions which represent time of events are in set T. Obviously $T \subset T'$. T' is also definable as $T' = \{y_1 - y_2 \mid y_1, y_2 \in T\}$. Now we show that the necessary and sufficient condition for the delay variables to make a hazard on the output. The proof is given in the appendix of this chapter.

Lem 5.5 The necessary and sufficient condition where hazards occur on the output of a circuit is expressed by a linear inequality system of the following form.

$$\begin{array}{l}
R\wedge \; (\\
\qquad (\xi_{1,1} \wedge \xi_{1,2} \wedge \cdots \wedge \xi_{1,p_1}) \vee \\
\qquad (\xi_{2,1} \wedge \xi_{2,2} \wedge \cdots \wedge \xi_{2,p_2}) \vee \\
\qquad \ldots \\
\qquad (\xi_{k,1} \wedge \xi_{k,2} \wedge \cdots \wedge \xi_{k,p_k}) \\
\;)
\end{array}$$

where

$$\xi_{i,j} \in \{t < 0, t \leq 0 \mid t \in T'\} \text{ and } p_1, p_2, \cdots, p_k \leq 3^n.$$

□

The lemma tells that the condition is expressed by a sum of products of linear inequalities whose coefficients are in $\{0,\pm 1\}$ and the constants are in $\{0, d^{min}{}_1, d^{max}{}_1, \cdots, d^{min}{}_n, d^{max}{}_n\}$.

Cor 5.6 The necessary and sufficient condition where the hazard occur on the output of a circuit is expressed by an linear inequality system of the following form.

$$(A_1\vec{x} \lhd \vec{b_1}) \vee (A_2\vec{x} \lhd \vec{b_2}) \vee \cdots \vee (A_k\vec{x} \lhd \vec{b_3}), \tag{5.1}$$

where A_i is the (n, p_i)-matrix whose elements are in $\{0,\pm 1\}$, b_i is a vector in D^n, and $\lhd$ represents that $\leq$ or $<$ relation holds for each row of the vectors. □

Consequently, hazards can occur in the discrete time model if and only if (5.1) has an *integer solution* and hazards can occur in the continuous time model if and only if the inequality system has a *real number solution.*

5.4.3 Reduction of Continuous Time Model into Discrete Time Model

Def 4.3 We define sets of absolute values of determinants of (n, n)-matrix as follows.

$$\begin{aligned} M &= \{|\det(A)| \mid A \text{ is a } (n,n)\text{-matrix whose elements are} \\ &\qquad \text{in } \{0, \pm 1\}\}, \\ M^2 &= \{x \times y \mid x, y \in M\}, \\ 2M^2 &= \{2x \mid x \in M^2\}. \end{aligned}$$

□

Now we prove an important lemma.

Lem 5.7 The following 1) and 2) are equivalent.

1) A Hazard can occur on circuit C in the continuous time model.

2) For a proper m where $m \in 2M^2$, a hazard can occur in the discrete time model on circuit C', which is obtained by multiplying each delay bounds by m.

[Proof] From Cor 5.6 we only have to prove that the following 1') and 2') are equivalent.

1') Linear inequality system (5.1) has a real number solution.

2') Linear inequality system (5.1) obtained by multiplying vector $\vec{b_i}$ by m has an integer solution.

[Proof of 1')→2')] If the linear inequality system in 2') has an integer solution $\vec{v}$, then $(1/m)\vec{v}$ is a solution of (1).
[Proof of 2')→1')] Let $A_i\vec{x} \lhd \vec{b_i}$ be a subsystem of (1) which has a real number solution. The set of solutions of the subsystem conforms the

region which consists of the inside and a part of the surface of an n-dimensional convex polyhedron. Let $\vec{v_1}$ and $\vec{v_2}$ be two of the vertices of the polyhedron. Then the middle point $\vec{v}$ of $\vec{v_1}$ and $\vec{v_2}$ is located inside of the polyhedron, and hence is a solution of the subsystem. Let us consider the magnitude of m which makes $m\vec{v}$ an integer vector.

$\vec{v_1}$ and $\vec{v_2}$ are the solution of $E_1\vec{x} = \vec{q_1}$ and $E_2\vec{x} = \vec{q_2}$, respectively. Note that E_1 and E_2, $\vec{q_1}$ and $\vec{q_2}$ are obtained by choosing n rows from matrix A_i, $\vec{b_i}$, respectively. Therefore all the elements of E_1 and E_2 are in $\{0, \pm 1\}$. Let

$$\Delta_1 = \det(E_1), \Delta_2 = \det(E_2).$$

Then the solution of the equations $\vec{v_1}$ and $\vec{v_2}$ are expressed as follows according to the Cramer's formula.

$$\vec{v_1} = (1/\Delta_1)(v_{1,1}, v_{1,2}, \cdots, v1, n), \vec{v_2} = (1/\Delta_2)(v_{2,1}, v_{2,2}, \cdots, v2, n),$$

where $v_{i,j}$ is an *integer*. The middle point of $\vec{v_1}$ $\vec{v_2}$ is expressed as

$$\begin{aligned} \vec{v} &= (1/2)(\vec{v_1} + \vec{v_2}) \\ &= (1/2\Delta_1\Delta_2)(\Delta_2 v_{1,1} + \Delta_1 v_{2,1}, \Delta_2 v_{1,2} + \Delta_1 v_{2,2}, \cdots, \Delta_2 v_{1,n} + \Delta_1 v_{2,n}) \end{aligned}$$

If we set $m = 2|\Delta_1\Delta_2|$, all the elements of $m\vec{v}$ are integers because Δ_1, Δ_2 and $v_{i,j}$ are integers. It follows that $A_i\vec{x} \lhd m\vec{b_i}$ has an integer solution. Since $|\Delta_1|$, $|\Delta_2| \in M$, $m/2 \in M^2$. □

Cor 5.8 Let μ be the least common multiple of M and $m = 2\mu^2$. Then the following two are equivalent.

1) A hazard can occur on circuit C in the continuous time model.

2) A hazard can occur in the discrete time model on circuit C', which is obtained by multiplying each delay bounds by m. □

Lem 5.9 $m \leq n^n$ holds for $m \in M^2$.

[Proof] The absolute value of the determinant of a matrix is less than or equal to the product of the norms of the column vectors of the matrix. If all the elements in a matrix are in $\{0,\pm 1\}$, the norms of the column vectors are less than or equal to $n^{1/2}$. Then the absolute value of the determinant of the matrix is less than or equal to $(n^{1/2})^n = n^{n/2}$. Hence $m \leq 2(n^{n/2})^2 = 2n^n$. □

Th 5.3 The following two are equivalent.

1) A hazard can occur on circuit C in the continuous time model.

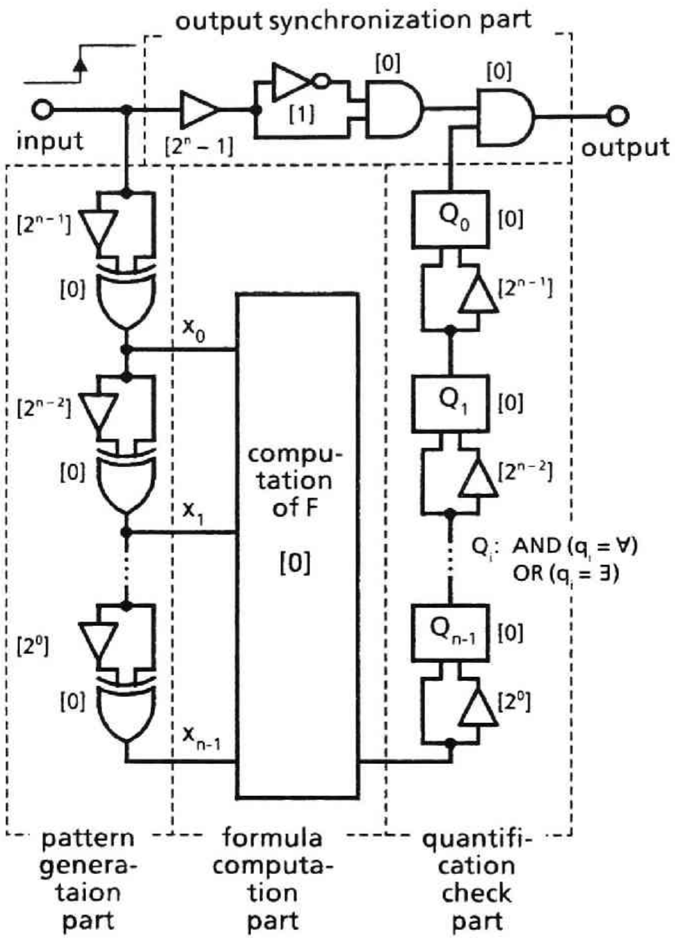
2) A hazard can occur in the discrete time model on circuit C', which is obtained by multiplying each delay bounds by a proper constant m, where $m \leq 2n^n$. □

5.5 Hazard Detection Problems of the Continuous Time Model

We have shown so far that the hazard detection problems of the continuous time model are reducible into those of the discrete time model by multiplying the magnitude of delay bounds by integer m which is bounded by $2n^n$. The reduced problems belong to what we defined as hazard detection problems of the *exponential delay model.* In this chapter we clarify the computational complexity of the problems of the continuous time model by discussing that of the problems of the exponential delay model.

5.5.1 Exponential Delay and Discrete Time Model

We begin with the exact delay model (SHDExp*Exct*Dscr). In the case of the constant delay model, we can solve the problem in deterministic polynomial time by means of logic simulation as is stated in. In the case



output synchronization part
input
output
$[2^n - 1]$
[1]
[0]
[0]
$[2^{n-1}]$
$[2^{n-2}]$
$[2^0]$
x_0
x_1
x_{n-1}
computation of F
[0]
Q_0
Q_1
Q_{n-1}
Q_i: AND ($q_i = \forall$) OR ($q_i = \exists$)
pattern genera-taion part
formula computa-tion part
quantifi-cation check part

Fig. 5.4 The circuit for QBF-SAT.

of the exponential delay model, however, we can not solve the problem in polynomial time by the same method because it may take $O(2^{p(n)})$ unit times for the output signal to be stable. Actually the computational cost increases markedly even in the exact delay model as shown in the following lemma.

Lem 5.10 SHDExp*Exct*Dscr is *PSPACE-hard.*

[Proof] We show the satisfiability problem of quantified Boolean formulas (QBF-SAT) is reducible into SHDExp*Exct*Dscr.

Let a given QBF be

$$(q_0x_0)(q_1x_1)\cdots(q_{n-1}x_{n-1})F(x_0,x_1,\cdots,x_{n-1}),$$

where q_i is a universal quantifier ($\forall$) or a existential quantifier ($\exists$). We construct a circuit shown in Fig. 5.4 for this QBF, whose hazard detection problem is equivalent to the satisfiability problem of the QBF. The circuit consists of a pattern generation part, a formula computation part, a quantification check part and an output synchronization part. The pattern generation part generates all the patterns in $\{0,1\}^n$ on signal lines X_0, X_1, $\cdots$, X_{n-1} using delay gates and EXOR gates. Each pattern has a duration of 1 unit time. They are generated in the descending order of the magnitude when they regarded as binary numbers. The formula computation part computes the value of $F(x_0,x_1,\cdots,x_{n-1})$ for the generated patterns. The circuit to compute F is the direct implementation of F. The output of the formula computation part is a sequence of the result values of F for all the possible input patterns. The quantification check part receives the sequence and checks quantification for each variable beginning with x_{n-1} using a delay gate and an AND or an OR gate. Since the result values of F are aligned in the descending order of the magnitude of input patterns as they regarded as binary numbers, we can compute the satisfiability of universal (existential) quantification for x_i by computing AND (OR, respectively) of pairs of values whose distance is 2^{n-i-1}. In the final step, the value at the time frame which contains the final result of the quantification check is taken out at the output synchronization part. The output of the whole circuit is 0 in the stable state and becomes 1 transiently if and only if the QBF is satisfiable. Thus satisfiability of the QBF is reduce into the hazard detection problem of this circuit. The computation time required for the transformation from a QBF to the circuit is bounded by a polynomial of the formula size. $\square$.

Since the exact delay model is a special case of the uncertain delay model, the same lower bound applies for the uncertain delay model.

Lem 5.11 SHDExp*Unc*Dscr is PSPACE-hard. □

As for the bounded uncertain delay model, we have not obtained the lower bound, which is not important for the discussions in the rest of this chapter.

On the other hand, as for an upper bound, we have obtained the following results that the hazard detection problems of the exponential time and the discrete time model are all solvable using polynomial space.

Lem 5.12 SHDExp*Exct*Dscr is *in PSAPCE.*

[Proof] We use *backward logic simulation*, in which we investigate existence of a hazard by checking the signal value backwards starting from the primary output, instead of the usual logic simulation. We assume that the input pattern changes from v_x to v_y at time 0. We denote the output value of the circuit for input pattern v as $f(v)$. We show an alternating algorithm (an algorithm executable on an alternating Turing machine). We separately consider the two cases where a static hazard can occur and where a dynamic hazard can occur. In the both cases, we use the recursive function *checkVal*(g, t, v) which examines if the output of gate g is v at time t.

Static hazard: In this case $f(v_x) = f(v_y)$ holds. We examine if the output value becomes $\overline{f(v_x)}$ of not due to the input change. Guess a path from the primary input to the primary output. Let d_p be the delay of the path (the sum of the delay values of the gates on the path). Examine if the output value (namely the output of gate g_n) is $\overline{f(v_x)}$ at time d_p by calling *checkVal*$(g_n, d_p, \overline{f(v_x)})$. A static hazard occurs if the answer is *yes*. Otherwise a static hazard does not occur for the guess.

Dynamic hazard: In this case $f(v_x) \neq f(v_y)$ holds. We examine if the output value changes as $f(v_x) \rightarrow \overline{f(v_x)} \rightarrow f(v_x) \rightarrow \overline{f(v_x)}$. Guess two paths from the input to the output. Let d_{p_1} and d_{p_2} be the delay of the paths (assume $d_{p_1} < d_{p_2}$). Examine if the output value of the circuit is $\overline{f(v_x)}$ at time d_{p_1} and $f(v_x)$ at time d_{p_2} by calling *checkVal*$(g_n, d_{p_1}, \overline{f(v_x)})$ and *checkVal*$(g_n, d_{p_2}, f(v_x))$. A dynamic hazard occurs if the both answers are *yes*. Otherwise a dynamic hazard does not occur for the guess.

function *checkVal*(g, t, v)

Trivial case: If $g = g_0$ (the primary input of the circuit) then return the following answer; if $t < 0 \wedge v = v_x$ or $t \geq 0 \wedge v = v_y$ then return *yes*, otherwise return *no*.

Recursive case: If $g \neq g_0$ then guess a Boolean vector $(v_1, v_2, \cdots, v_m)$ for the inputs of gate g which makes the output of the gate v. For each input of g, examine if the value becomes v_i at time $t-d$, where d is the delay of gate g, by calling *checkVal*$(g^i, t-d, v_i)$, where g^i is the gate which feeds the i-th input of gate g. If all the answers are *yes* then return *yes*. Otherwise return *no*.

Since the depth of the circuit is at most n, so is the number of calls of *checkVal*. We can judge existence of a hazard in polynomial time on an alternating Turing machine and the problem belongs to PSPACE. □

Upper bounds for the uncertain delay models are immediately derived from this lemma.

Lem 5.13 SHDExp*Unc*Dscr and SHDExp*Rst*Dscr are *in PSPACE*.

[Proof] We can reduce the problem of the uncertain delay models by guessing the combination of the delay values which causes a hazard. Since the delay combinations can be coded using polynomial space, the problems belong to PSPACE. □

The following theorem concludes this subsection.

Th 5.4 SHDExpExctDscr and SHDExpUnbDscr are both *PSPACE-complete.* □

5.5.2 An Upper Bound of the Computational Complexity of Hazard Detection Problem of the Continuous Time Model

From the discussion so far, we can improve the upper bound of the problem of the constant magnitude, uncertain delay and *continuous* time model in Prop 5.4.

Th 5.5 SHDCnstUnc*Cont* and SHDCnstRst*Cont* are *in PSPACE.*

[Proof] Guess a number from $\{2, 3, \cdots, 2n^n\}$ as m in Th 5.3 and solve the problem obtained by multiplying the delay bounds by m in the *discrete time model.* The problems are of the exponential magnitude and the discrete time model (SHDExpExctDscr and SHDExpUnbDscr) and from Lem 5.13 they are in PSPACE. □

5.6 Remarks and Discussions

We have discussed the computational complexity of the hazard detection problem of the various delay and time models. Table 5.1 summarizes the result.

We have shown that the problem of detecting hazards on combinational circuits under uncertain delay assumption is computationally intractable (NP-hard). This follows that it is difficult to solve the problem by a simple extension of the min/max delay simulation technique and

Table 5.1 Summary of the results.

(a) Results for the constant magnitude delay model.

	Exact (Exct)	Uncertain (Unc)	Restricted (Rst)
Discrete time (Dscr)	in P	NP-complete	NP-complete
Continuous time (Cont)		NP-hard & in PSPACE	NP-hard & in PSPACE

(b) Results for the exponential magnitude delay model.

	Exact (Exct)	Uncertain (Unc)	Restricted (Rst)
Discrete time (Dscr)	PSPACE-complete	PSPACE-complete	NP-hard & in PSPACE

that we must develop different algorithms to achieve accurate timing verification, which will require big computation cost.

We have also discussed the relation between the continuous time model and discrete time model. The verification result can be optimistic in the discrete time model. We showed an concrete example for this. We also showed that the discrete time model will have the same ability of the continuous time model by making the time unit small with respect to the hazard detection problem, and clarify to what extent we must make the unit time small.

In Table 5.1 there still remains a gap between the lower bound and upper bound of the computation cost of the problem of the bounded delay and continuous time model (NP-hard and in PSPACE). It is considered to be an important research theme to clarify the computational complexity of this model, because it is closely related to the essential difference of the ability between the continuous time model and discrete time model.

The high computational complexity of the continuous time model or of the exponential delay model is due to the existence of signal lines which have more than $O(2^n)$ times of signal changes. In the actual circuit, however, we can not observe such a phenomenon because of *inertia* delay. It is also an important research issue to define feasible model for the inertia delay and to discuss the hazard detection problem based on the model.

5.A Appendix: Proof of Lem 5.5

Def 5.4 We call triple (g, t, v) where $g \in G$, $t \in T$ and $v \in \{0, 1, d\}\}$ (signal value d is a *don't care*), a *time-value requirement* for a gate g. We call a set α obtained by the following 1) $\sim$ 3) a *time-value requirement* for a circuit.

1) Initially α consists of time-value requirements of gate g_n, namely the primary output of the circuit.

2) If $(g_i, t, v) \in \alpha$ and $v \neq d$ then include $(g_i{}^1, t - x_i, v^1)$, $(g_i{}^2, t - x_i, v^2)$, $\cdots$, $(g_i{}^m, t - x_i, v^m)$ which satisfy $v = f_i(v^1, v^2, \cdots, v^m)$, where $g_i{}^j$ is the gate which feeds the j-th input of gate g_i.

3) If $(g, t, 0)$, $(g, t, 1)$ and (g, t, d) are in α simultaneously (for the same g and t), α is not a time-value assignment. □

Intuitively (g, t, v) represents that the output of gate g is v at time t. α consists of the consistent tuples.

Def 5.5 Let α be a time-value requirement of a circuit. Let $\alpha_0 = \{(g_0, t, v) \mid (g_0, t, v) \in \alpha\} = \{(g_0, t_1, v_1), (g_0, t_2, v_2), \cdots, (g_0, t_k, v_k)\}$. Let $v_x \to v_y$ be the given input signal change. Then we define the *realization condition* of α, denoted as S_α, as follows.

$$\begin{aligned} S_A &\equiv \sigma_1 \wedge \sigma_2 \wedge \cdots \wedge \sigma_k, \\ \sigma_j &\equiv (t_j < 0) \text{ if } v_j = v_x, \\ & (t_j \geq 0) \text{ if } v_j = v_y. \end{aligned}$$

□

Intuitively, S_α represents the condition where time-value requirement α takes place. We can express the condition where static or dynamic hazards occurs.

Lem 5.14 The necessary and sufficient condition where a static hazard occur (in the case of $f(v_x) = f(v_y)$) is

$$R \wedge \bigvee_{\alpha \in A} S_\alpha,$$

where

$$A = \{\alpha \mid \alpha \text{ is a time-value requirement of circuit } C, (g_n, t, \overline{f(v_x)}) \in \alpha, t \in T'\}.$$

[Proof] $\alpha \in A$ expresses a time-value requirement where the output of the circuit becomes $\overline{f(v_x)}$, namely a static hazard occurs. □

Lem 5.15 The necessary and sufficient condition where a dynamic hazard occur (in the case of $f(v_x) \neq f(v_y)$) is

$$R \wedge \bigvee_{\alpha \in A} (S_\alpha \wedge (t_{\alpha_1} < t_{\alpha_2})),$$

where

$$\begin{aligned} A = \{\alpha \mid\ & \alpha \text{ is a time-value requirement of circuit } C, \\ & (g_n, t_{\alpha_1}, \overline{f(v_x)}) \in \alpha, \\ & (g_n, t_{\alpha_2}, f(v_x)) \in \alpha, \\ & t_{\alpha_1} \in T', t_{\alpha_2} \in T'\}. \end{aligned}$$

[Proof] $\alpha \in A$ expresses a time-value requirement where the output of the circuit changes its value as $f(v_x) \to \overline{f(v_x)} \to f(v_x) \to \overline{f(v_x)}$, namely a dynamic hazard occurs. □

Now we show the proof of the Lem 5.5.

[Proof of Lem 5.5] All the inequalities in S_α have the form of $t < 0$ or $t \leq 0$ where $t \in T'$. $t_{\alpha_1} < t_{\alpha_2}$ in Lem 3.14 can be transformed into the form of $t < 0$ where $t \in T'$ because $t_{\alpha_1} - t_{\alpha_2} \in T'$. All the inequalities in R have the form of $t \leq d$ where $t \in T'$ and $d \in D$. Thus if we expand the condition in Lem 3.10 or Lem 3.11 into the sum-of-product form, we have the condition expressed in the form of the lemma. □

Chapter 6

Time-Symbolic Simulation for Accurate Timing Verification

6.1 Introduction

In design of asynchronous circuits, timing becomes the most important issue. We must examine designs so that they do not fall into erroneous behavior caused by critical races, hazards and oscillations. When the behavior of a circuit depends on subtle timing relations, we must consider the change of delay values which may be caused by variations of process conditions and differences of usage environments. In logic simulation, which is currently one of the most effective methods of dynamic timing analysis, we treat the uncertainty of the delay value by using the min/max delay model [Bre76]. Although the model enables relatively fast simulation, it has been pointed out that simulation results are often too pessimistic if the circuit contains reconvergent fanouts [Bre76]. The simulation results contain so many unknown signal values or report so many possibilities of timing errors that it is very difficult to know if the circuit under test really has design errors. It has, therefore, come to be an important research theme to find effective methods for timing verification taking the delay uncertainty into account and many researches are undertaken on this issue [Yon89, Cer89]. In this chapter we propose a new approach based on *symbolic execution* of logic simulation.

In the conventional symbolic simulation [Car79, Cor81], they introduce variables to represent signal values and execute simulation treating signal values as Boolean expressions. In our approach we execute simulation by representing a gate delay or time of input change by a variable. This idea of *time-symbolic simulation* makes it possible to compute the accurate effect of the uncertainty of actual delays. It also enables us to get useful information for identifying the error location and for finding the way to correct design errors by analyzing the symbolic results. One difficulty in time-symbolic simulation is in the algorithm to carry out simulation. The conventional algorithms are not straightforwardly applicable because the times are not constants any more. In this chapter we show two efficient algorithms for time-symbolic simulation and discuss their application to design verification of asynchronous circuits.

The first algorithm is based on the T-algorithm [Ish84, Ish85yy] and is dedicated for combinational circuits. Symbolic manipulation of time is relatively easy in T-algorithm because the time is advanced independently at each gate in a circuit. Since the simulation result depends on relations among occurrence times of events, we represent the signal history on each signal line using a data structure named an *event tree* instead of a linear list. It is necessary to simplify algebraic expressions and to judge the feasibility of inequalities during the simulation . We solve these problems by reducing them into *linear programming.* Although this time-symbolic simulator can directly deal with combinational circuits only, it is possible to verify the behavior of asynchronous sequential circuits by examining the behavior of their combinational parts. Time-symbolic simulation also enables us to obtain conditions for correct behavior of the circuit, which is of good use for design error correction and design improvements. Since it requires good skill to obtain the conditions from a result of time-symbolic simulation by hand, we also developed a result analysis system for time-symbolic simulation.

In this chapter we propose another algorithm for time-symbolic simulation which can directly deal with circuits containing feedback loops. In this algorithm we assume time to be discrete. A key point of this algorithm is that we can reduce time-symbolic simulation into usual value-symbolic simulation by *encoding* the time variables. Namely, instead of using time variables, we *encode* the cases of possible delay values of an uncertain delay unit and represent the delay variation by *Boolean variables*. Then the time-symbolic simulation is reduced into usual symbolic simulation and we can simulate all kinds of logic circuits based on the conventional S-algorithm. We refer to this new technique as *coded time-symbolic simulation (CTSS)*. In the CTSS all the computational difficulties are condensed into *Boolean function manipulation*. We use an efficient Boolean function manipulator using *shared binary decision diagrams (SBDD's)* [Min90, Min91] as internal representation of Boolean functions. The use of SBDD's in the CTSS drastically reduces storage requirements and enables efficient simulation execution. It is also important to provide aids for analyzing simulation results, for the simulation results of the CTSS are given in the form of Boolean expressions with coded time variables. In this chapter we propose a novel technique of comparing simulation results obtained by the CTSS with desirable behavior based on *symbolic simulation of automata*.

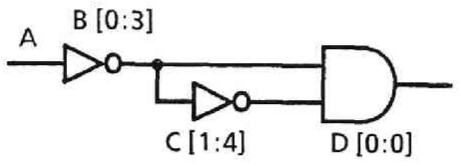
In the following section we discuss the modeling of delay uncertainty using time variables. In section 6.3 we show an algorithm for time-symbolic simulation based on T-algorithm and in section 6.4 discuss applications of time-symbolic simulation to timing verification of logic circuits. In section 6.5 and 6.6, we describe an idea, an efficient algorithm and applications of the coded time-symbolic simulation. Some further discussions will be made in section 6.7.

6.2 Problems of Conventional Min/Max Delay Simulation

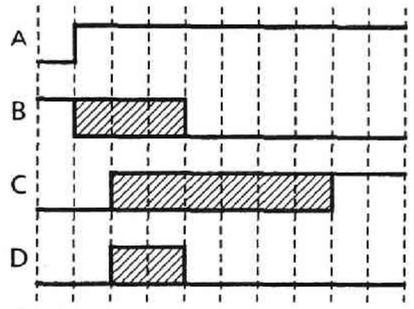
An actual delay value of a logic gate is affected by process conditions or usage conditions. In logic simulation we treat the uncertainty by using the min/max (ambiguity) delay model [Bre76]. It has been pointed out, however, that this model has serious shortcomings such that simulation results are often too pessimistic due to reconvergent fanouts [Bre76]. For example, a timing chart in Fig. 6.1 (b) is the result of the min/max delay simulation for a circuit in Fig. 6.1 (a). The unknown states on the output of D indicate the possibility of a static hazard, which never occurs in an actual circuit because the rising edge on the output of C never precedes the falling edge on the output of$B$. The overpessimism comes from loss of information that the uncertainty in the time of the rising edge on the output of C is partly due to that of the falling edge on the output of B. We are actually simulating the circuit shown in Fig. 6.1 (c) instead of that in Fig. 6.1 (a). Since the unknown states produced in this way propagate around the circuit polluting correct signal states, it becomes impossible to judge if the circuit really has design errors.

There are some simulators which detect reconvergences and avoid the overpessimism to some extent. However, as is shown in chapter 5, the hazard detection problem of a combinational circuit under the uncertain delay model is *NP-hard* and it is therefore considered to be impossible to compute an accurate simulation result by a simple extension of the min/max delay simulation algorithm. Fig. 6.2 shows some difficult examples[1]. Two circuits and expected accurate waveforms for the circuits are illustrated. Although the circuits are very small, the twofold reconvergence make the accurate simulation difficult. Especially it is difficult to predict the absence of hazards at (3). In this chapter, we attempt

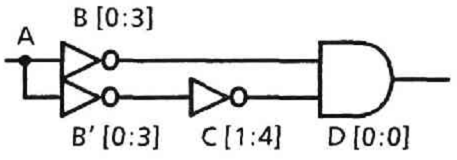
[1]These examples are devised by Mr. Hiroaki KANEHARA at Kyoto University.



(a) A circuit with a reconvergence.

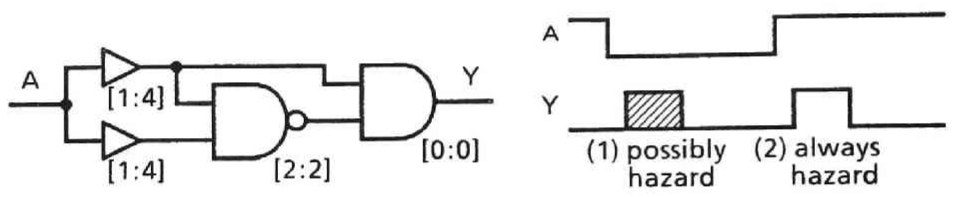


(b) Min/max delay simulation.

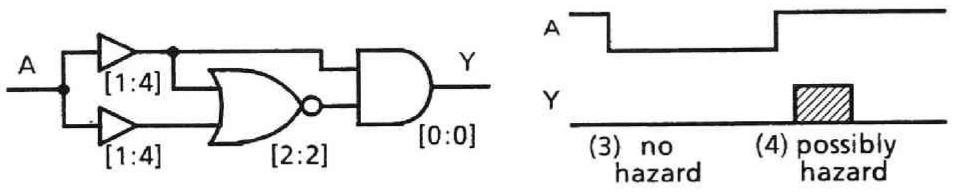


(c) The circuit actually simulated.

Fig. 6.1 Overpessimism in min/max delay simulation.



(a) Circuit *Kanehara-A*.



(b) Circuit *Kanehara-O*.

Fig. 6.2 Difficult examples for the min/max delay simulation.

to solve this problem by expressing delay variation using *time variables* which can take an arbitrary value within the bounds.

6.3 Time-Symbolic Simulation Based on T-Algorithm

6.3.1 Modeling of Uncertain Delay Using Time Variables

The overpessimism of the min/max delay simulation at reconvergent gates is due to the loss of the information that the uncertainty of time of signal changes propagated through different paths has a common source. In our approach we identify the common source of the uncertainty by expressing each uncertain delay value by *a variable* over the real number domain. We refer to the variable as *a time variable*. Minimum and maximum delay values may be specified for an uncertain delay unit. In such a case, we express them by a *set of inequalities* such as $\{3 \leq d, d \leq 5\}$. We call the set of inequalities *variable constraints*. We assume the inequalities in a variable constraints are *linear inequalities* in general forms. It is therefor possible to express the relation between delay values such as $\{d_1 \leq 2d_2\}$. We can model the various kinds of uncertainty of a delay value using the time variables. In this section, we consider the following three models.

Static nominal delay model We assign one time variable to one delay unit. Namely, we assume the delay value is uncertain but *constant.*

Static rise/fall delay model We assign two time variables which represent rise delay and fall delay of the delay unit.

Dynamic delay model We assign a time variable for each signal change. Namely, we assume the delay value can be different at each signal change.

6.3.2 Algorithm Based on S-Algorithm

It is possible to adapt the conventional S-algorithm to the time-symbolic simulation, if efficiency of execution is not critical. The following is the algorithm, in which events scheduled to occur in the future are maintained in set Q.

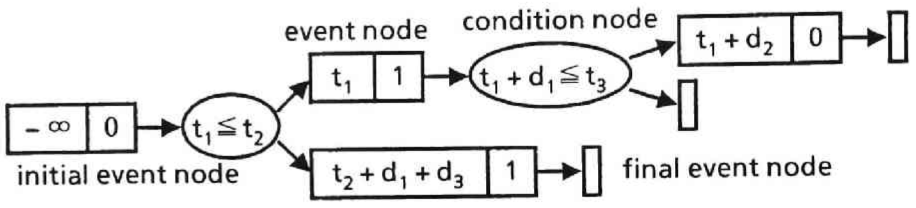
1) Repeat 2)$\sim$4) until Q becomes empty.

2) Get an event e out of Q whose occurrence time is judged to be the smallest. When there are more than one candidate, investigate all the possibilities by branching.

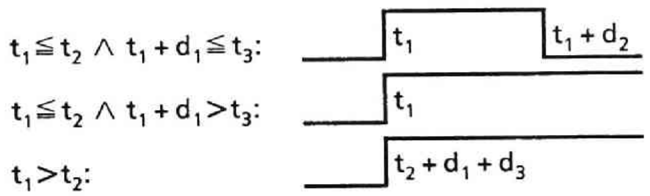
3) Compute the effect of e.

4) If there are new events as the result of 3) put them into Q.

In step 3), we do not have to investigate all the possibilities if the order of occurrences of events does not affect the entire behavior of the circuit. However, we are forced to investigate almost all the possibilities as long as we are not allowed to cancel and recompute the simulation results, because it is almost impossible to know the possibility of inconsistency



(a) An event tree


$t_1 \leqq t_2 \wedge t_1 + d_1 \leqq t_3$:
t_1
$t_1 + d_2$
$t_1 \leqq t_2 \wedge t_1 + d_1 > t_3$:
t_1
$t_1 > t_2$:
$t_2 + d_1 + d_3$

(b) Waveforms represented by the event tree

Fig. 6.3 An event tree.

in advance. As a result, the simulation speed becomes so slow that we can not simulate even a small circuit within feasible time. One of the breakthroughs to this problem is an optimistic strategy such as in [Yon89], which in turn makes the simulation control complicated.

On the other hand, symbolic manipulation of time is relatively easy in the T-algorithm because the time is advanced independently at each gate in a circuit. We propose in this section an efficient algorithm of time-symbolic simulation dedicated for combinational circuits based on the T-algorithm.

6.3.3 Representation of a Signal History by an Event Tree

In the case of usual simulation where times are constants, a signal history of each signal line is represented by a linear list of events. In the case of time-symbolic simulation the simulation results depends on relations among occurrence times of events. We use a data structure named an *event tree* as shown in Fig. 6.3 (a) in order to represent such a signal history.

An event tree is a directed tree. Nodes are classified into *event nodes* and *condition nodes.* An event node is labeled by an event, a tuple of time and a signal value. The time is expressed by a *linear combination of time variables.* A condition node is labeled by a *linear inequality.* We will refer an event node and a condition node simply as an event and a condition, respectively, for simplicity. Direct edges represent the order of event occurrences. The root node of an event tree is an event node whose occurrence time is $-\infty$ and is called the *initial event node.* All the leaf nodes are event nodes and are called *final event nodes* which represent that there are no more events after the events. Each of the event nodes except for final event nodes has just one successor and each of the condition nodes has two successors. The first (the upper, in the figure) edge represents the case where the the linear inequality γ of the node holds, and the second (the lower) edge represents the case where γ does not hold. We refer to γ and $\neg\gamma$ as branching conditions of the first edge and the second edge, respectively. The *path condition* of a node is the product of the branching conditions of the edges constructing the path from the root node to the node. The event tree in Fig. 6.3 (a) represents three event sequences in Fig. 6.3 (b).

6.3.4 Algorithm of Gate Evaluation

The algorithms of gate evaluation in time-symbolic simulation is basically the same as that of the usual logic simulation but for the following exceptions.

1) Since there can be more than one candidate for the next event at each input line of a gate, we must investigate all the possible cases. For example in Fig. 6.4, input A has two candidates ((1) and (2)) and B has also two candidates ((3) and (4)) for the next event. We must investigate the four cases for the combination (1)(3), (1)(4), (2)(3) and (2)(4).

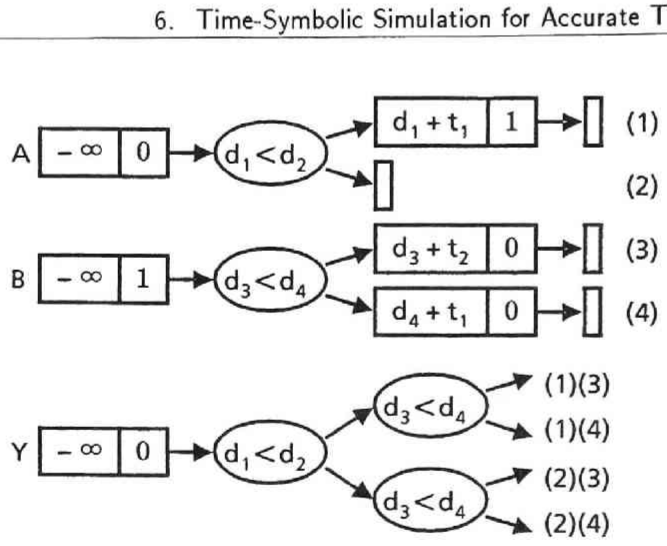


Fig. 6.4 Computation of the output event tree.

2) Even if each input line has a unique candidate for the next event, there are cases where we cannot decide which input has the event of the minimum occurrence time. In such cases we must also investigate all the possibilities. For example, suppose we are investigating the combination (1)(4) in Fig. 6.4. We have to examine the two cases if we can not decide which of d_1 and d_4 is the smaller.

In both 1) and 2), new condition nodes are appended to the output event tree to make branching. Note that there are cases where branching does not occur depending on the path conditions and variable constraints. In the example in 2), we can decide the next event if the variable constraints includes $\{d_1 \leq 3, 5 \leq d_4\}$ because $d_1 + t_1$ is always smaller than $d_4 + t_1$.

Fig. 6.5 shows an algorithm of computing the output event tree of a 2-input gate from the given input event trees, for the static nominal delay model. The followings are the explanation of the algorithm.

1) Function `gateEval` receives root nodes `a` and `b` of the event trees of the inputs and returns the root node of the resulting output event tree.

```
function gateEval(node a, node b) return(node)
begin op(a, b, variable_constraints, unknown, unknown, unknown) end gateEval;

function op(node a, node b, inequality_set s, sig_val vy, sig_val va, sig_val vb)
   return(node)
begin
   if a is a condition node then
   begin
      pc⁺:=pc∪{a.cond};
      pc⁻:=pc∪{¬a.cond};
      if        sat(pc⁺)∧¬sat(pc⁻) then return op(a.upper, b, pc, vy, va, vb)
      else if   ¬sat(pc⁺)∧sat(pc⁻) then return op(a.lower, b, pc, vy, va, vb)
      else if   sat(pc⁺)∧sat(pc⁻) then
         return cnode( a.cond, op(a.upper, b, pc⁺, vy, va, vb),
                               op(a.lower, b, pc⁻, vy, va, vb))
   end
   else if b is a condition node then similar to the case where a is a condition node
   else if both a and b are event nodes then
   begin
      pc⁺:=pc∪{a.time≦b.time};
      pc⁻:=pc∪{a.time>b.time};
      if        sat(pc⁺)∧¬sat(pc⁻) then return evalA(a, b, pc, vy, va, vb)
      else if   ¬sat(pc⁺)∧sat(pc⁻) then return evalB(a, b, pc, vy, va, vb)
      else if   sat(pc⁺)∧sat(pc⁻) then
         return cnode( a.time≦b.time,  evalA(a, b, pc⁺, vy, va, vb),
                                       evalB(a, b, pc⁻, vy, va, vb))
   end
end op;

function evalA(node a, node b, inequality_set s, sig_val vy, sig_val va, sig_val vb)
   return(node)
begin
   vy':=FUNC(a.val, vb);
   next:=op(a.next, b, pc, vy', a.val, vb);
   if (vy'≠vy)  then return enode(a.time+DELAY, vy', next)
                else return next
end evalA;

function evalB is similar to evalA;
```

Fig. 6.5 An algorithm of computing output event trees.

2) Function `op` receives root nodes `a` and `b` of the subtrees of the input event trees, set of inequalities `s` representing variable constraints and path conditions, and the current signal values of the output line and the input lines `vy`, `va`,`vb`. It computes the sub- event tree corresponding to the sub - event trees rooted by `a` and `b`, and returns the root node of the sub- event tree.

3) Function `evalA` and `evalB` receive the same arguments as `op` except that root nodes `a` and `b` are event nodes. They computes output sub- event trees obtained when `a` and `b` occur earlier, respectively.

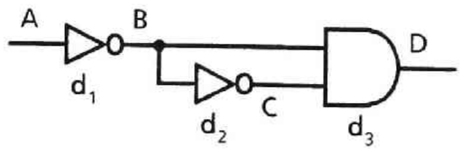
4) The occurrence time, the signal value and the next node of event node `e` are denoted as `e.time`, `e.val` and `e.next`, respectively. The event node whose occurrence time, signal value and next node are `t`, `v` and `n`, respectively, is denoted as `enode(t,v,n)`.

5) The inequality of a condition node `c` is denoted as `c.cond`. The nodes which are pointed by the upper and the lower edges of condition node `c` is denoted as `c.upper` and `c.lower`, respectively. The condition node which satisfies `c.cond=i`, `c.upper=u` and `c.lower=l` is denoted as `cnode(i,u,l)`.

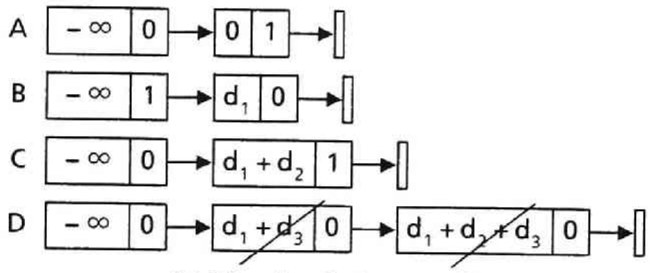
6) `FUNC` is the Boolean function of the gate and `DELAY` is the delay variable of the gate.

7) Function `sat` judges whether a given set of linear inequalities is feasible (whether there is an assignment to the time variables which satisfies all the inequalities in the set) or not. Details of the computation procedure is described in section 6.3.5.

In the case of the static rise/fall delay model, we must change the delay variable according to the new output value. Furthermore we need the *event cancellation* operation as is the case of the conventional logic simulation [Bre76]. Let d_r and d_f be the time-variables representing rise



(a) The circuit in Fig. 6.1.



(b) The simulation result

Fig. 6.6 An example of the time-symbolic simulation (1).

delay and fall delay, respectively, e_r and e_f be the expression representing the time of input events which cause the $0 \rightarrow 1$ event and $1 \rightarrow 0$ event at the output of a gate, respectively. We assume that $e_r < e_f$ absolutely holds. In the case of $e_r + d_r > e_f + d_f$, we have to cancel events at time $e_f + d_f$. We judge the feasibility of the inequalities and make branching if necessary.

Time-symbolic simulation of the dynamic delay model is realized by introducing as many time-variables as output events. We also need event cancellation. In many cases, difference of the delays is not so large. We can express this by the variable constraints. Although we need much more variables than static delay model, we can expect more precise timing analysis.

Fig. 6.6 shows the result of time-symbolic simulation on the same circuit and the same input pattern as in Fig. 6.1. In the computation on gate D, we do not make branching because we can tell that the event on the output of B occurs earlier than that on the output of C by algebraic comparison. As the result, we can conclude that there is no possibility of the hazard on the output of D.

6.3.5 Manipulation of Algebraic Expressions

In order to execute the procedure described in the previous section, it is necessary to treat algebraic expressions which include time variables. The required operations are as follows.

1) Addition of simulation time and gate delays.

2) Simplification of inequalities such as $d_3 + d_2 < d_1 + d_2 \rightarrow d_3 < d_1$.

3) Judgment of feasibility of a set of linear inequalities.

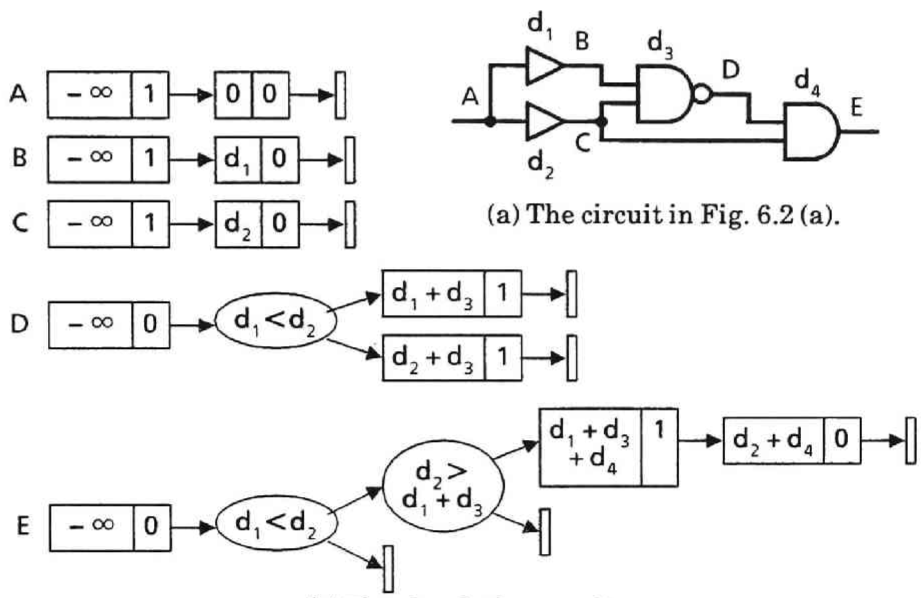
Since all the algebraic expressions appearing in our time-symbolic simulation are linear combinations of time variables, these operations are easily realized. 1) is achieved by addition of each coefficients. 2) is also trivial if we use the normal form representations of inequalities such as $\sum_{i=1}^{n} a_i d_i + c \leq 0$. As for 3), we solve the feasibility problem by *linear programming*.

6.4 Timing Verification by Time-Symbolic Simulation

We have implemented a time-symbolic simulator based on the above algorithm, on the SUN 3/60 workstation in C language. We use the *simplex method* for linear programming.

6.4.1 Hazard Detection

We can tell the possibility of hazards directly from the result obtained by time-symbolic simulation. Fig. 6.7 shows the result of the simulation on the circuit in Fig. 6.2 (a). A falling edge is given to the input. From the result we can tell the possibility of a static hazard on output E.



(a) The circuit in Fig. 6.2 (a).

(b) The simulation result.

Fig. 6.7 An example of the time-symbolic simulation (2).

Furthermore we can tell that the condition where the hazard occurs is $d_1 < d_2 \wedge d_1 + d_3 < d_2$ (namely $d_1 + d_3 < d_2$) and that we can avoid the hazard by increasing the delay value of d_1.

6.4.2 Verification of Asynchronous Sequential Circuits

Although our simulator is applicable only to combinational circuits, we can verify the behavior of asynchronous sequential circuits by examining the behavior of the combinational part [Kim88].

We assume that the followings are given as an instance for the verification.

1) The state transition table of basic mode asynchronous sequential machine M.

2) Gate level implementation C of M.

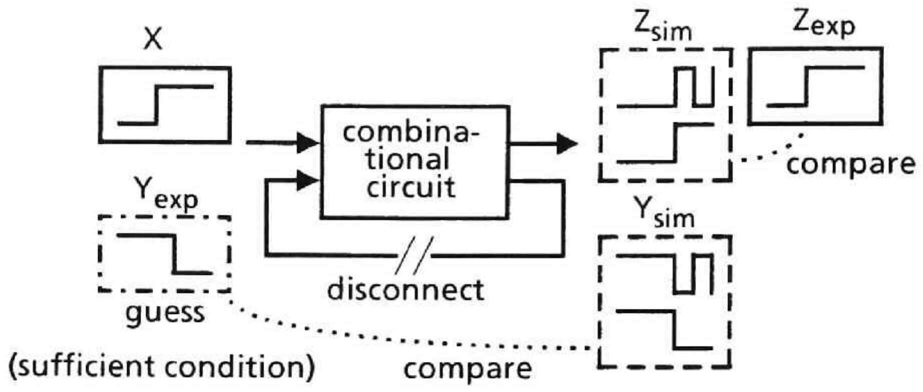


Fig. 6.8 Verification of an asynchronous sequential circuit.

3) The state assignment and the correspondence between state variables and feedback lines in C.

The algorithm of the verification is as follows (see Fig. 6.8):

1) Get a combinational circuit C' by disconnecting the feedback lines corresponding to the state variables.

2) For each state transition, verify the correctness by 3)$\sim$5).

3) Let X, Z_{exp}, and Y_{exp} be event sequences to appear at the primary inputs, the primary outputs and the feedback loops, respectively, when the transition occurs. X and Z_{exp} are derived directly from the state transition table. As for Y_{exp}, we can guess a proper pattern since the signal values of the feedback lines before and after the transition are specified in the state transition table. If the occurrence time of an event is unknown, represent the time by a time variable.

4) Perform time-symbolic simulation on C' giving X and Y_{exp} to the primary inputs and the disconnected lines, respectively, and get the output Z_{sim} and Y_{sim} on the primary outputs and the disconnected lines, respectively.

5) Compare Z_{sim} with Z_{exp}, and Y_{sim} with Y_{exp}. If they are consistent, we can conclude that the correct state transition takes place regardless of the delay variations. Otherwise we can get the condition where the correct transition occurs.

Note that the condition obtained in 5) is a *sufficient condition* because there is a possibility that the correct transition takes place with the event sequences different from Y_{exp} which we chose in 3).

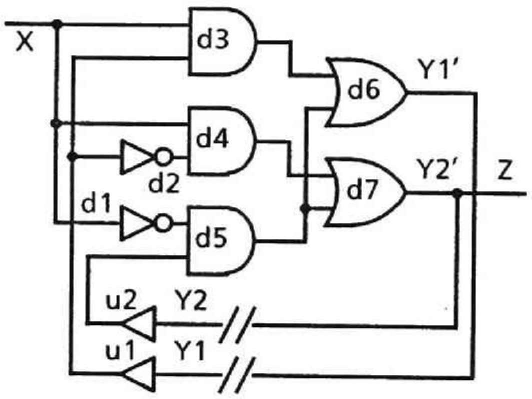
We show an example of verification of a T-flipflop. The state transition table and a gate level implementation are shown in Fig. 6.9 (a) and (b). We get four input and desirable patterns for the four transitions (Fig. 6.9 (c)) from the state transition table. Here we expect that there are no hazards also on the feedback lines. (This assumption is considered to be a feasible one). We introduce time-variables `t1`, `t2`, `t3` and `t4` to represent delay times for the transitions because they are unknown.

By executing time-symbolic simulation and comparing the simulation results, we obtain the condition shown in Fig. 6.9 (d). The four equations express the value of `t1`$\sim$`t4`, namely the time necessary for each transition. We can tell the critical path for each transition from these equations. The two inequalities the condition where the circuit correctly works as a T-flipflop. The inequalities do not contain time variables `u1` and `u2` which represents the delay values of the feedback lines. From this fact, we can conclude that the circuit may fall into an erroneous behavior depending on the variations of the delay values in the combinational part and that it is impossible to avoid the timing error by adjusting the delay values of the feedback lines.

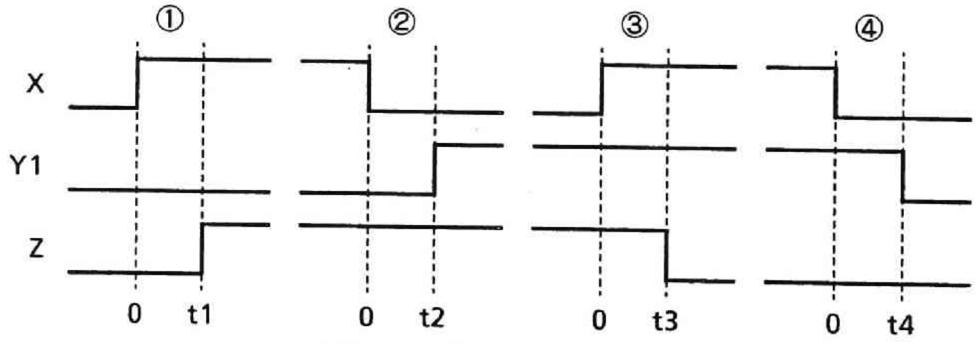
Fig. 6.10 (a) is an alternative design whose combinational part has hazard detection gates. The verification result is shown in Fig. 6.10 (b). In this case, the inequalities contain time variable `t2`. This tells us that there is no danger of combinational hazards but that there is, in turn, the possibility of sequential hazards. We also conclude that we can satisfy

Y1 Y2	Y1′ Y2′ X = 0	Y1′ Y2′ X = 1	Z
0 0	0 0	0 1	0
0 1	1 1	0 1	1
1 1	1 1	1 0	1
1 0	0 0	1 0	0

(a) State transition diagram.



(b) A gate-level implementation.

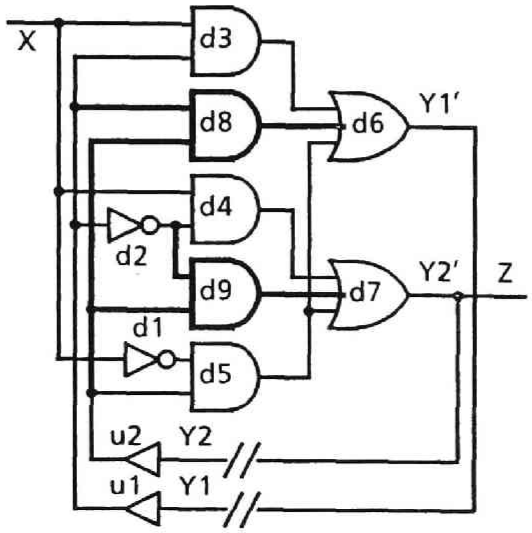


(c) Expected timing charts.

t1 = d4 + d7
t2 = d1 + d5 + d6
t3 = d1 + d5 + d7
t4 = d3 + d6
d1 + d5 < d4
d1 + d5 > d3

(d) Result.

Fig. 6.9 Verification of a T-flipflop.



$t1 = d4 + d7$
$t2 = d1 + d5 + d6$
$t3 = d1 + d5 + d7$
$t4 = d3 + d6$
$d3 < u2 + d8 + d1 + d5 + d7$
$d1 < u2 + d4 + d7$

(a) A gate-level implementation. (b) Result.

Fig. 6.10 Verification of a T-flipflop with hazard detection gates.

the condition by increasing the delay of the feedback line and that the critical paths for the transitions are the same in spite of the addition of the hazard detection gates.

6.4.3 Result-Analysis System

Time-symbolic simulation makes it possible not only to confirm the circuit's correct behavior but to obtain the conditions for correct behavior of the circuit. In order to obtain conditions for correct behavior of circuits, it is necessary to compare event trees with correct event sequences. This comparison process is hard by hand when event trees become large. In order to solve this problem, we have also developed a result analysis system for the time-symbolic simulation.

It compares multiple pairs of an event tree and an expected event sequence and outputs the condition which matches the all pairs. The condition is shown as a sum of products of linear inequalities. It also reduces the duplicated and redundant conditions. For example, the condition

$$d_2 + d_3 + d_6 < d_4 \wedge d_4 + d_5 < d_1 + d_3 \wedge d_2 < d_1$$

is reduced into

$$d_2 + d_3 + d_6 < d_4 \wedge d_4 + d_5 < d_1 + d_3$$

because the third inequality can be derived from the other conditions and is redundant. We judge the redundancy also by means of linear programming.

6.4.4 Performance of the Simulator

The CPU time required for the verification of the T-flipflops is less than 0.1 seconds in total and the simulation speed in this case is about 100 $\sim$ 300 event/second. This speed is considered to be enough for verifying small scale circuits. However, the computation cost of time-symbolic simulation is at least proportional to the exponential of the number of gates in the worst case. Although this is inevitable if we consider the complexity of the problem, it is difficult to apply time-symbolic simulation to large scale circuits. In this case combination use of the time-symbolic simulation and the conventional min/max delay simulation will be effective.

As for the memory requirement, the size of the event trees grow being proportional to the number of gates and will be dominant. However, in the small scale circuit we can not neglect the size of the tableau for the linear programming, which is proportional to the product of the number of time variables and the number of inequalities. Our current simulator can deal with circuits of about 100 gates within the storage of 8MB.

6.5 Coded Time-Symbolic Simulation - CTSS

In the T-algorithm based approach, we can not directly deal with circuits with feedback loops. On the other hand, in the S-algorithm based approach described in section 6.3.2, impractical computation time is required to simulate even a small scale circuits. In this section we propose an alternative algorithm for time-symbolic simulation which can deal with circuits with feedback loops. It is based on *Boolean function manipulation* and can simulate circuits with feedback loops almost in the same speed as circuits without feedback loops.

6.5.1 Modeling of Uncertain Delay by Boolean Variables

The most important assumption in this section is that the time is *discrete* as is in the conventional logic simulation. Namely the simulation time and the delay values take integer values. Under the discrete time assumption we can enumerate the possibilities of actual values of a bounded static uncertain delay. Let us take the circuit in Fig. 6.1 (a) as an example. Each of two inverters B and C, whose delay is specified as [0,3], will take one of four delay values {0,1,2,3}. If we investigate the 16 cases, namely the 4 cases for B multiplied by the 4 cases for C, we can get a completely accurate simulation result. The total number of the cases to be examined will be exponential to the number of uncertain delay components in a circuit. This is inevitable because of the complexity of the problem that we discussed earlier. We focus our attention on how we can make the simulation process efficient.

Again in the example above, 16 possible signal values are associated with a signal line at a time period. If we code the delay of B and C using Boolean variables, such as $delay_B = (b_1, b_0)$ and $delay_C = (c_1, c_0)$, the 16 signal values can be seen as a *Boolean function* of the 4 input variables. Then the simulation with the uncertain delays is reduced into usual *symbolic simulation*. This is a basic idea of our *coded time-symbolic simulation*.

For the convenience of explanation, we assume without loss of generality that a gate in a circuit is either a pure functional gate with delay

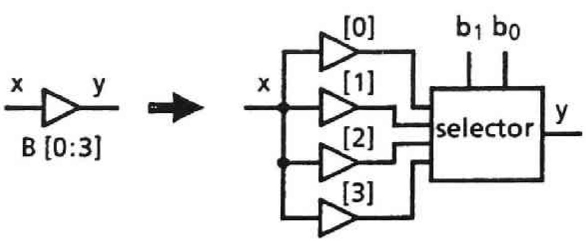


Fig. 6.11 An interpretation of the coding of delay variation.

0 or a pure delay gate with a single input and a single output. Let us denote a signal value on line s at time t as $s[t]$. Then the signal value on output line y of functional gate g is computed by the following equation:

$$y[t] = f_g(x_1[t], x_2[t], \cdots, x_k[t]), \tag{6.1}$$

where f_g is the Boolean function of g, and $x_1, x_2, \cdots, x_k$ are the signal lines which feed g.

As for a delay gate, we can interpret a coding of delay as shown in Fig. 6.11. Time variables b_1 and b_0 are selection inputs to choose one of the four delay possibilities. If we relate (b_1, b_0) with the binary representation of $delay_B$, we can compute the output y of the delay unit according to the following formula:

$$\begin{aligned} y[t] &= \overline{b_1} \cdot \overline{b_0} \cdot x[t] \\ &+ \overline{b_1} \cdot b_0 \cdot x[t-1] \\ &+ b_1 \cdot \overline{b_0} \cdot x[t-2] \\ &+ b_1 \cdot b_0 \cdot x[t-3] \end{aligned}$$

The definition of delay gate g in general is as follows. Let y and x be the output line and the input line of g, min_g and max_g be the minimum and maximum delay value of g, and $g_0, g_1, \cdots, g_l$ (where $l = \lceil \log_2(max_g - min_g + 1) \rceil - 1$) be the time variables coding the possibility of delay values of g. Let us also define $g_i^{<k>}$ and $G^{<k>}$ as follows.

$$g_i^{<k>} = \begin{cases} \overline{g_i} & \text{if } i\text{-th bit of the binary} \\ & \text{representation of } k \text{ is } 0, \\ g_i & \text{otherwise.} \end{cases}$$

$$G^{<k>} = g_l^{<k>} \cdot \cdots \cdot g_1^{<k>} \cdot g_0^{<k>}.$$

The output value of delay gate g is computed according to the equation:

$$\begin{aligned} y[t] &= G^{<0>} \cdot x[t - min_g] \\ &+ G^{<1>} \cdot x[t - min_g - 1] \\ &\ \ \ldots \\ &+ G^{<max_g - min_g>} \cdot x[t - max_g]. \end{aligned} \tag{6.2}$$

Fig. 6.12 is an example of simulation on the circuit in Fig. 6.1 (a). Fig. 6.12 (a) shows a coding of delay variation. Fig. 6.12 (b) shows the simulation result, namely the signal values of each signal line at each time. A rising edge at time 0 is given to A. The Boolean function $\overline{b_1 + b_0 + c_1 + c_0}$ appearing on line C at time 0, for example, indicates that the value is 1 when $b_1 = b_0 = c_1 = c_0 = 0$, namely $delay_B = 0$ and $delay_C = 1$, and otherwise the value is 0. The signal value on D is always 0, which is the accurate result that we expected. In the CTSS, the variables of a delay unit do not appear in the formula to represent the signal value on line s at time t, as long as the delay does not affect s at time t. So we can automatically avoid the useless comparison and branching.

6.5.2 Representation of Boolean Functions by a Shared Binary Decision Diagram

Good representation of Boolean function is a key to efficient symbolic simulation. In our implementation we use a *shared binary decision diagram (SBDD)* [Min90, Min91], which is an improvement of the binary

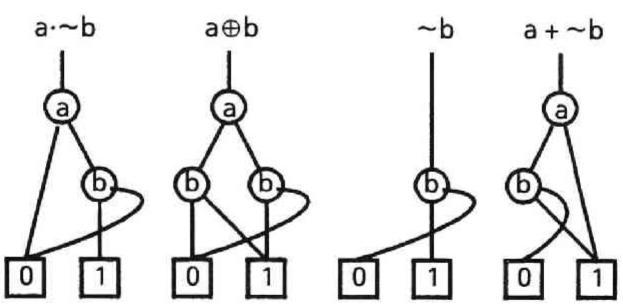
b_1	b_0	$delay_B$	c_1	c_0	$delay_C$
0	0	0	0	0	1
0	1	1	0	1	2
1	0	2	1	0	3
1	1	3	1	1	4

(a) Coding of the delay values.

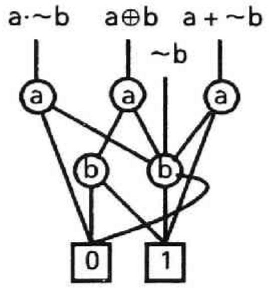
t	$A[t]$	$B[t]$	$C[t]$	$D[t]$
-1	0	1	0	0
0	1	$b_1 + b_0$	0	0
1	1	b_1	$\overline{b_1 + b_0 + c_1 + c_0}$	0
2	1	$b_1 \cdot b_0$	$\overline{b_1 + c_1} \cdot \overline{b_0 \cdot c_0}$	0
3	1	0	$\overline{b_1 + c_1} + \overline{b_0 + c_0} \cdot \overline{b_1 \cdot c_1}$	0
4	1	0	$\overline{b_1 + c_1} + \overline{b_1 + b_0} + \overline{c_1 + c_0}$ $+\overline{b_0 + c_0} \cdot \overline{b_1 \cdot c_1}$	0
5	1	0	$\overline{b_1 + c_1} + \overline{b_1 \cdot c_1}$	0
6	1	0	$\overline{b_1 \cdot b_0 \cdot c_1 \cdot c_0}$	0
7	1	0	1	0

(b) The simulation result.

Fig. 6.12 An example of coded time-symbolic simulation.



(a) Binary decision diagrams.



(b) A shared binary decision diagram.

Fig. 6.13 A shared binary decision diagram (SBDD).

decision diagram [Bry86]. In the SBDD all possible subgraphs are shared among multiple functions, as shown in Fig. 6.13.

The SBDD has the following advantages besides those of the BDD.

1) Many functions can be efficiently expressed *simultaneously*.

2) Many of the operations can be done much faster than those of the BDD. Especially, equivalence of two functions are checked by simply comparing the pointers while, in the BDD isomorphism should be examined.

These two advantages are very much suitable for our purpose. Since in the CTSS we need to represent many Boolean functions to express signal values on signal lines, the property 1) is very favorable. Fig. 6.14 shows the representation of the signal values in the simulation of delay units connected in cascade. We can see how well the subgraphs are shared. The

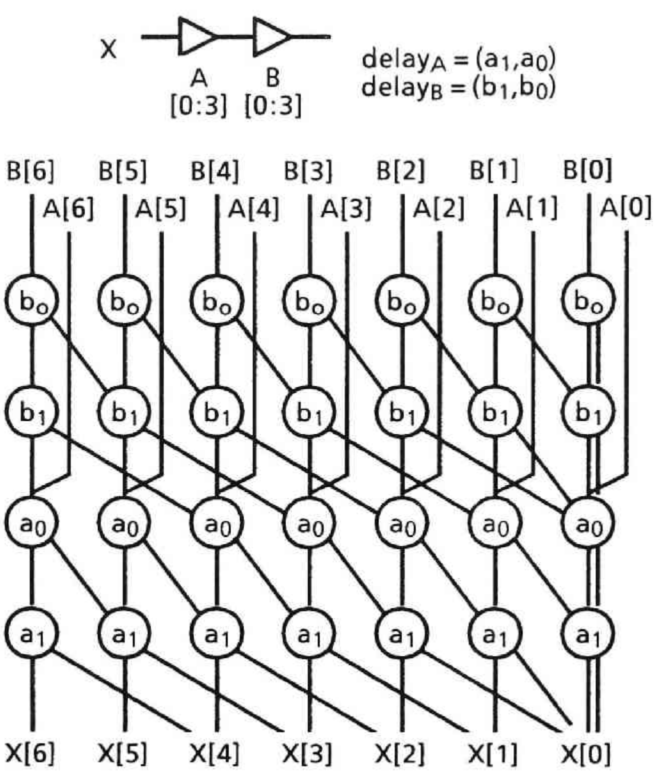

X
A
B
[0:3] [0:3]
delay$_A$ = (a$_1$,a$_0$)
delay$_B$ = (b$_1$,b$_0$)
B[6] B[5] B[4] B[3] B[2] B[1] B[0]
A[6] A[5] A[4] A[3] A[2] A[1] A[0]
b$_0$
b$_1$
a$_0$
a$_1$
X[6] X[5] X[4] X[3] X[2] X[1] X[0]

Fig. 6.14 Representation of signal values by SBDD.

property 2) is also favorable for detecting the changes of signal values in symbolic simulation. The Boolean functions appearing in the CTSS are produced by the logical operations for gate evaluations and the delay operations which correspond to addition and comparison of the integers in binary representation. We can expect efficient simulation because BDD's are known to have good affinity for these operations [Ish90y]

6.6 Timing Verification by Coded Time-Symbolic Simulation

6.6.1 Result Analysis of the CTSS

Although CTSS offers accurate simulation results, they are represented by Boolean functions and it is often difficult to understand the meaning of the Boolean functions and to tell if there exist errors. For example, in Fig. 6.12 the signal values on C satisfies the following relation.

$$0 = C[0] \subset C[1] \subset C[2] \subset \cdots \subset C[6] = 1,$$

where $x \subset y$ is defined as $\overline{x} + y$. From this relation, we can conclude that *there is always a single rising edge on* C regardless of the combination of delay values. However, it seems difficult to derive the fact only by looking at the expressions. It is therefore important to prepare a mechanism to analyze simulation results and tell if the simulation results match desirable behavior of the circuit under test. In this section, we will discuss methods of analyzing results. We propose a novel technique of comparing the simulation results with desirable ones.

6.6.2 Analysis of Simulation Results Based on Symbolic Simulation of Finite Automata

In section 7.4 we compared simulation results with desirable results by expressing the desirable waveforms using the same data structure as simulation results. Although we can also apply this strategy to the CTSS, it is difficult to derive Boolean expressions to represent the specifications in general. In this paper we propose a novel technique of comparing simulation results with desirable waveforms on the basis of symbolic simulation of finite automata. This technique is a generalization of the edge detection technique shown above.

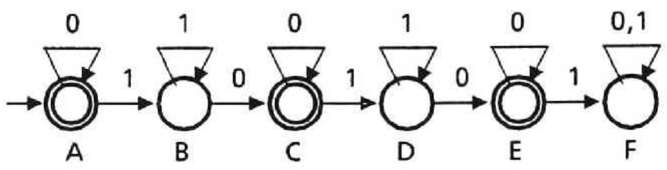


Fig. 6.15 Finite Automaton M_η.

At first we represent desirable behavior of a circuit by regular expression η. We construct deterministic finite automaton A_η which accepts the same set of sequences as η. We design sequential circuit M_η which inputs a sequence and outputs 1 if and only if A_η accepts the sequence. We simulate M_η along with the circuit under test. When the final output of the M_η is 1, we can conclude that the circuit satisfies the specification η regardless of the delay values. When the final output is a Boolean expression containing delay variables, the expression indicates the possible combinations of actual delay values for the correct behavior of the circuit.

For example, when we want to verify that not more than two 1-pulses are allowed on the output line x of the circuit C, the specification is written as:

$$\eta = 00^* + 00^*11^*00^* + 00^*11^*00^*11^*00^*.$$

From this regular expression we can construct deterministic automaton A_η as shown in Fig. 6.15.

By state assignment $A = (1,0,0), B = (0,0,0), C = (1,0,1), D = (0,0,1), E = (1,1,1), F = (0,1,1)$, we get sequential machine M_η expressed by the following equation, where y_1, y_2 and y_3 are state variables and ok is the output of M_η:

$$\begin{aligned} y_1' &= \overline{x} \cdot \overline{y_2} + \overline{x} \cdot y_1 \cdot y_3, \\ y_2' &= y_2 \cdot y_3 + \overline{x} \cdot \overline{y_1} \cdot y_3, \\ y_3' &= y_3 + \overline{x} \cdot \overline{y_1}, \\ ok &= y_1. \end{aligned}$$

By simulating M_η along with the circuit C, we know if C satisfies the specification η.

6.6.3 Extraction of Algebraic Expressions

As a result of the comparison discussed in the previous subsection, we get a Boolean expression indicating delay conditions for correct behavior. We are able to obtain a set of combinations of delay values immediately from this expression. However, it is much more helpful if we obtain algebraic relations between the delay values. Suppose the following expression is obtained, where $delay_A$ and $delay_B$ are coded by (a_2, a_1, a_0) and (b_2, b_1, b_0), respectively.

$$\begin{aligned} ok &= b2 \cdot \overline{a2} + b2 \cdot b1 \cdot \overline{a1} + b2 \cdot b1 \cdot b0 \cdot \overline{a0} \\ &+ b2 \cdot \overline{a1} \cdot b0 \cdot \overline{a0} + \overline{a2} \cdot b1 \cdot \overline{a1} \\ &+ \overline{a2} \cdot b1 \cdot b0 \cdot \overline{a0} + \overline{a2} \cdot \overline{a1} \cdot b0 \cdot \overline{a0}. \end{aligned}$$

It is difficult to realize by what condition the circuit behaves correctly. If we extract the following algebraic expression of $delay_A$ and $delay_B$, we can understand the condition very well.

$$delay_A < delay_B.$$

Extraction of an algebraic expression from a Boolean function represented by a BDD is discussed in [Ohm90]. Currently we have an efficient algorithm to extract a single linear inequality. There is room for a further study to extract delay conditions in general, which are expressed as logical combinations of linear inequalities.

6.6.4 Implementation Issues

In order to enhance the performance of simulators, we usually adopt event driven simulation mechanism. In this implementation, however, we decided to adopt the compiler driven simulation mechanism. It is because we considered that the event driven simulation is not necessarily advantageous for the following reasons:

1) Since an event on the input line of delay gate g at time t affects the output line of g at time $t + min_g, t + min_g + 1, \cdots, t + max_g$, we have to handle much more events than in usual logic simulation.

2) In order to accelerate symbolic operations in a SBDD, we keep recent results of symbolic operations in a hash table [Min90], and we can execute the same symbolic operations as we executed recently by just looking up the table. Since the cost of the table look-ups is much smaller than that of the symbolic operations, we can not expect a drastic reduction of computation time by omitting the same operations according to the event driven simulation strategy.

On the other hand, in the compiler driven simulation, we must pay attention to the order of gate evaluation, because we are required to evaluate a gate for many times until the circuit becomes stable. This requirement brings a considerable drawback to computation time. We classify gates into the following two categories.

1) Delay gates whose minimum delay value is not 0.

2) Functional gates (whose delay value is 0) and delay gates whose minimum delay value is 0.

Since the output value of a gate in Category 1) at time t does not depend on the input value at time t, we can evaluate the gate without waiting for the evaluations of the other gates. However we need to be careful about the order of the evaluations of gates in Category 2). In our implementation, we evaluate all gates in Category 1) first and then evaluate gates in Category 2) in the order of the level number. We exclude a circuit which contains loops consisting of gates in Category 2) in the preprocessing stage.

Table 6.1 Performance of the coded time-symbolic simulator.

Circuit	# of gates	Simulated time units	CPU time [sec]	# of nodes	Speed [event /sec]
adder*1*	6	10	0.2	29	105.0
adder*2*	12	26	0.8	121	97.5
adder*4*	24	34	2.7	653	111.1
adder*8*	48	66	15.0	4,285	78.4
adder*16*	96	130	350.4	31,229	13.3
mult*2*	16	22	1.3	438	126.2
mult*4*	88				
dec*8*	17	18	3.0	6,489	47.3
enc*8*	22	10	4.4	10,424	52.5
tff (1)	7	29	2.4	2,758	84.4
tff (2)	7	29	0.9	75	108.8

We implemented a coded time-symbolic simulator based on the methods described so far. The simulator is written in language C and runs on a Sun3/60 workstation.

6.6.5 Experimental Results

The simulator successfully computed the accurate results of the difficult examples in Fig. 6.2. Table 6.1 shows performance figures on some circuits. The column *circuit* shows the names of circuits simulated. Here, adder$n$ is an n-bit ripple carry adder, mult$n$ an n-bit array multiplier, dec*8* an *8*-bit decoder, enc*8* an *8*-bit priority encoder, and tff a T flip-flop presented in [Ish89]. Bounded uncertain delays of [1,4], minimum 1 and maximum 4, were assigned to all the gates in the circuits, except for tff(2). At first each circuit was initialized with an arbitrary input pattern, and then all the input signal values were inverted all at once at time 0. Simulation was executed until there remain no events except for tff(1).

The T flip-flop contained feedback loops and began oscillation when all the gates had delays of [1,4]. We stopped the simulation at time 29. Row tff(1) shows the result of this simulation. In the simulation of tff(2), the delays were adjusted so that oscillation might not occur (delay of each gate had width of 4). The maximum SBDD size was limited to 100,000 nodes. Simulation was stopped when there was the larger requirement (indicated as '-' in the table).

We counted the number of events occurred during the simulation by a separate program. Simulation speed was computed by dividing the number of events by CPU time for simulation. The simulation speed is about 10 to 100 events per second, which decreased with the growth of circuit size. We conclude that our simulator is much slower than conventional min/max delay simulators, but it is amazingly fast because it simulated 4^{96} cases in about 6 minutes as is shown in the result of adder*16*. The circuit with feedback loops was also simulated at the speed as fast as combinational circuits, though it took a lot of time if the circuit oscillated. Since we have not attempted to order the variables, these figures (the number of nodes and simulation speed) should improve, if the variables are appropriately ordered.

6.7 Remarks and Discussions

New notions of time-symbolic simulation and coded time-symbolic simulation have been proposed as a new approach for accurate timing verification of logic circuits, and its implementation and application have been described. Our simulation techniques make it possible to simulate logic circuits with uncertain delay units precisely. Furthermore they enable us to derive the conditions in which the circuits behave correctly. It is also possible to compute the probability where the circuit under test falls into the erroneous behavior by extending the coding scheme of

CTSS [Deg90]. These by-products are very useful for design correction and design improvements.

Both of the methods are considered to be effective to handle the static variation of the delay value. Although the dynamic variation of a delay value can be also modeled by introducing many variables, we can not expect efficient simulation. There is much room for a study on the simulation methods for dynamic delay variation.

As for simulation speed, we succeeded in simulating small scale circuits within feasible CPU time. In the CTSS the circuit with feedback loops can be simulated in as much time as combinational circuits unless it oscillates. The simulators run fast enough to simulate small scale circuits and are considered to be effective for the verification and redesign of small asynchronous blocks such as flipflops.

It is considered to be difficult to simulate a large scale circuit of more than 10,000 gates by time-symbolic simulation because of the complexity of the problem. As a solution we are now developing an approximated symbolic evaluation technique and the method of combining time-symbolic simulation with the conventional min/max delay simulation.

Chapter 7

NES: A Nondeterministic Behavior Model for Hardware Description Languages

7.1 Introduction

Hardware description languages (HDL's) are kernels of CAD systems for integrated circuits which work as inputs to various CAD tools, design documents and vehicles for design interchange among different CAD systems. Although a lot of research projects have been carried out on hardware description languages, we are now confronted with a big turning point due to two trends; *standardization* and *extension of the applications* of HDL's.

Standardization of a hardware description language (HDL) has an inestimable impact on the development of hardware design, including CAD tool development and design education. There are several activities for standardization in the U. S., Europe, and Japan [Kar89, Pil83, Coe89, Har86]. Since a standard HDL is used by many users, including IC manufactures and tool developers working in various kinds of design culture, we should provide them with a method of sharing a detailed idea on the HDL. It is therefore essential to define rigid syntax and semantics of the language. Although almost all the HDL's are designed on the basis of the formal definition of syntax by a meta language like BNF, there are very few HDL's, especially among the practical ones, which has clear definition of semantics. The task of defining the formal semantics of an HDL can be broken into the following two subtasks.

1) Defining a basic model of explaining hardware behavior.

2) Defining the relation between syntax and semantics based on the behavioral model.

There have been a lot of researches on the method of 2) in the area of programming languages [Bjo78]. There have been, however, few studies in the area of HDL's other than [Pil83]. Especially there have been no established models which explain the behavior of the hardware described in HDL's. In view of the trend of standardization, it is considered to be an urgent research theme to develop good behavior models for HDL's and to establish formal methods for defining semantics of HDL's.

Extension of the applications of HDL's is also changing the situation. For many years logic simulation has been the most important application of HDL's. Actually semantics of HDL's is closely related to efficient simulation algorithms and how to build behavior models which enable efficient simulation has been one of the most important issues. In practical situations semantics of an HDL is defined by means of the simulator for the HDL. However, recent researches in the area of CAD for integrated circuits have brought about outstanding development of techniques for various design support by computers. Especially *logic synthesis* come to become a practical technique and there are strong demands for HDL's to support logic synthesis. However, the simulation based semantics often causes inconsistency. On example is the handling of *don't cares.* In logic synthesis, we assume all the possible values for don't care specifications, but in logic simulation don't care values are dealt with as *unknown values.*

This is inevitable if we consider efficiency of simulation execution but it often brings about unnatural results as is discussed in Chapter 6. The same inconvenience arise also in applying HDL's to *formal verification.* Furthermore the techniques of logic simulation are also changing. Many simulators attempt efforts to avoid unnatural results at the computation cost as small as possible. Now we need a behavior model for HDL's which is disengaged from the conventional simulation techniques and can support various applications such as logic synthesis, formal verification and advanced simulation techniques.

In this chapter we propose a new behavioral model of hardware, named NES (Nondeterministic Event Sequences) model [Ish90y]. The NES model is a generalization of the event-driven simulation mechanism. The most important feature of the NES model is that it models *uncertainty* of hardware behavior by means of *nondeterminism.* Uncertain behavior of hardware is associated with signal values which are not specified or specified as *don't cares*, delays whose values are specified only by their minimum and maximum values, and so on. The uncertainty often makes the semantics of HDL's unclear and ambiguous. We mean by the term "nondeterminism" to take all the possible behavior derived from the description into account. The nondeterministic semantics forms a rigid basis for logic synthesis and formal verification, and also can be a final goal of logic simulation [Yas89, Yas90].

In the following section, we describe basic concepts of the NES model. We show how waveforms and the behavior of a hardware module are modeled and described in section 7.3, and show the modeling of connected modules in section 7.4. We also discuss applications of the NES model in section 7.5.

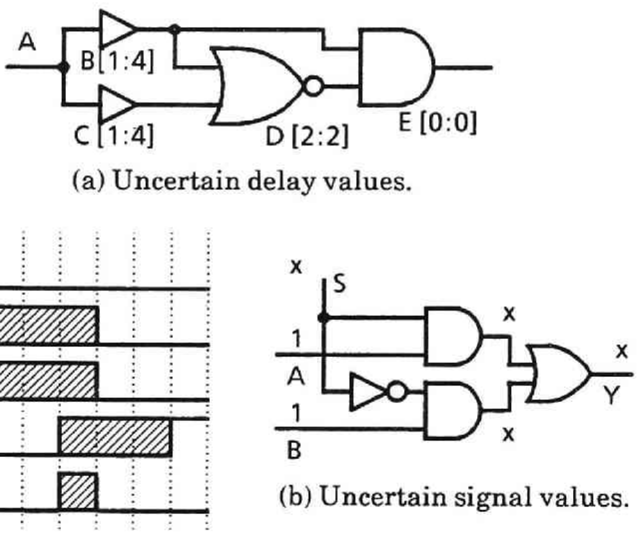


Fig. 7.1 Problems in the deterministic modeling of the behavior.

7.2 Basic Concepts of the NES Model

7.2.1 Modeling of Uncertainty by Nondeterminism

When we attempt to model hardware, we often face with *uncertainty* of hardware behavior, such as a signal value whose value is unknown a delay time whose only minimum and maximum values are known. There are two reasons that explain such uncertainty. One is that we can not specify the exact behavior of hardware in an actual condition. For example, as we mentioned in Chap 6, it is impossible to specify the actual delay time in advance. We refer to this kind of uncertainty as *don't know uncertainty*. Another kind of uncertainty is what we call *don't care uncertainty*. If we do not mind, or do not want to mind, all the details of a design, we do not describe the complete specification but leave a signal value as *don't care* or specify the delay of combinational circuits by a pair of minimum and maximum values.

Traditionally, such kinds of uncertainty has been modeled by introducing a special signal value denoting *unknown*; behavior of hardware has been explained based on a calculus such as $unknown + 1 = 1$ but $unknown + unknown = unknown$. This is considered to be a *simulation oriented* calculus because it enables fast computation. It has been pointed out, however, that this poor calculus often leads to pessimistic and unnatural semantics, as is discussed in Chapter 6. For example, in Fig. 7.1 (a), the unknown value on the line E indicates a possibility of a hazard, which never occurs in an actual circuit. Fig. 7.1 (b) shows another example which explains the pessimism in signal values. The input values on A and B are 1 while the value on S is x which specifies don't care or unknown. According to the deterministic calculus the output value on Y is x. However, in actual circuits, the output value on Y is always 1 regardless of the value on S because this circuit is a gate-level implementation of a selector.

Such a deterministic modeling of uncertainty has been a natural consequence when simulation is by far the most important application of HDL's. The semantics of HDL's has been directly connected with simulation techniques. Nowadays, however, simulation is not the only important application of HDL's. We should take account of synthesis, verification, and other various applications, where simulation technique oriented semantics will cause many inconsistencies. Especially in synthesis and verification we need to deal with the uncertainty of hardware behavior in a strict sense.

In the NES model, we attempt to express the uncertainty by means of *nondeterminism.* We describe the possible behaviors of hardware associated with the uncertainty using a *set.* This concept is similar to that of the coded time-symbolic simulation in Chapter 6. For example, we treat an uncertain delay between 1 and 4 in Figure 1(a) as a set of delays {1,2,3,4}. The behavior of the circuit is explained with all the possible combinations of delay values. Similarly, unknown value on line S of the circuit in Figure 1(b) is treated as a set of values {0,1}, which brings about signal value 1 on line Y. Thus we can define natural and strict semantics for uncertain hardware behavior. This feature is also desirable for creating strict discussions on formal verification and synthesis. We will discuss this issue in section 7.5.

7.2.2 Modeling of a Zero Delay

In designing hardware, timing relations among events are very important. HDL's must have a framework of specification of timing relations. Thus one of the most important issues in developing a behavioral model of hardware is *how to model time.*

Probably one of the simplest way of modeling time is to represent a waveform by a sequence of symbols, each of which is associated with a minimum unit of the discrete time. Then the behavior of a hardware component is represented by a sequential machine over the set of the symbols. Although this modeling realizes simple mathematical handling, it lacks the ability to express the occurrence of multiple events at the same place within a unit time. So it is very difficult to explain the behavior of the circuits that contain loops consisting of *zero delay* components. Here, zero delay means the delay less than the unit time. It is a product of the quantitization of time. If the delay time of a component is less than the minimum unit of the time, it is specified as 0. The zero delay also comes up when we take a clock cycle as a unit time. In such a case, delay of gates is treated as zero delay because it is not measured by the unit time. The zero delay expresses *before-after relationship* or *causality* whose delay time is 0 measured by the unit time,

In order to deal with the zero delay, the time models of bcl (Conlan) [Pil83] and VHDL [Coe89] are designed on the basis of a sub-unit time named a step and a Δ-delay, respectively. The signal value at a unit time is the final result of the infinite repetition of the computation in a step or a Δ-delay. Since these models are invented to compute the final result at

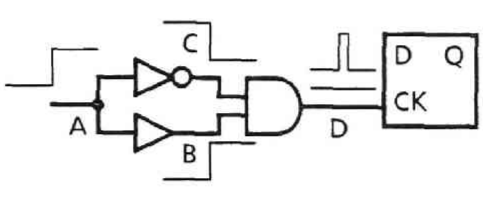


Fig. 7.2 Results dependent on the order of computation.

a unit time, no attention is paid to the order of the computation within a unit time. Actually, a step or a Δ-delay is associated with simulation under the unit delay model. There are cases where the final result depends on the order of the computation. For example, in Fig. 7.2, while the circuit is impractical, there may be a hazard on D which changes the signal value on Q. By using the models of bcl and VHDL, we will never get this result. This is again because these languages are based on the deterministic computation model.

7.3 Modeling and Description of Behavior of a Hardware Module

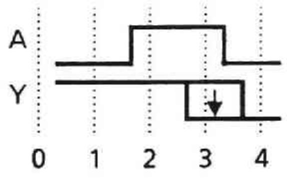
7.3.1 Modeling of Waveforms

In the NES model, we model waveforms on signal lines by a *set of event sequences.* An event is a tuple of a *place* and a *signal value.* A place is associated with a signal line. An intuitive meaning of event (p, v) is that the value of place p becomes v. An event is considered to have no duration. The order of the events in a sequence represents the before-after relationship between events. Namely, the events are totally ordered and there is no concept of *simultaneous occurrence.* Generally, causality between events is modeled by a partial order between events. In our model, as shown in Fig. 7.3, we represent a partial order by a set of all the possible event sequences that are consistent with the partial order.

Time is modeled by special events that indicate the progress of the

$$\begin{tikzcd}
 & b \arrow[dr] & & & \\
a \arrow[ur] \arrow[dr] & & d \arrow[r, Rightarrow] & \{abcd, acbd\} \\
 & c \arrow[ur] & & &
\end{tikzcd}$$

Fig. 7.3 Representation of a partial order by a set of sequences.



(a) Timing chart.

```
{(A,0)(Y,1)(@T,ns)(A,1)(@T,ns)(Y,0)(@T,ns)(A,0)(@T,ns) ….
 (A,0)(Y,1)(@T,ns)(A,1)(@T,ns)(@T,ns)(Y,0)(A,0)(@T,ns) …,
 (A,0)(Y,1)(@T,ns)(A,1)(@T,ns)(@T,ns)(A,0)(Y,0)(@T,ns) …}
```

(b) Representation by a set of event sequences.

Fig. 7.4 Modeling of waveforms.

time. This framework is essential to represent a causality relationship independent of the time. It is also possible to express more than one notion of time in the framework, such as nano-second and clock-cycle, simultaneously. Fig. 7.4 is an example of the representation of waveforms. The waveforms in timing chart (a) is represented by a set of three event sequences. **(@t,ns)** is an event that expresses a progress of the time in nano second. Uncertainty of the time of the falling edge on Y is expressed by the three possible event sequences.

```
X ={ (CK,0)(D,1)(CK,1)(D,0)…
     (D,1)(CK,0)(CK,1)(D,0)…}
```

↓

```
Y ={ (CK,0)(D,1)(CK,1)(Q,1)(D,0)…
     (D,1)(CK,0)(CK,1)(Q,1)(D,0)…}
```

Fig. 7.5 Behavior of a hardware module.

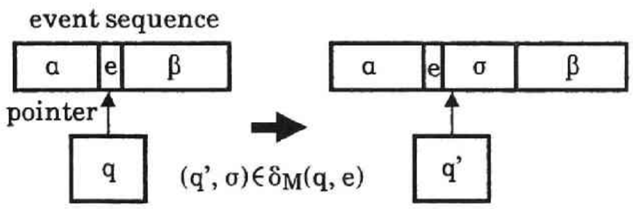


Fig. 7.6 Operation of the abstract machine.

7.3.2 Modeling of Behavior of a Module

The behavior of a hardware module can be regarded as a process which computes, from a set of event sequences on its inputs, the set of event sequences on its inputs and outputs (see Fig. 7.5). We model this process on the basis of the behavior of an abstract machine. Intuitively, the machine scans each of given event sequences with a pointer and inserts output events into proper positions. The abstract machine M over a set of event E is a triple $M = (Q_M, I_M, \delta_M)$, where Q_M is a set of states, $I_M \subset Q_M$ is a set of initial states, and $\delta_M : Q_M \times E \to 2^{Q_M \times E^*}$ is a partial function which defines the state transition of M. $\delta_M(q, e)$ is a *set* of possible *actions* of M in state q and reading event e. An action is specified by a pair of the next state and an output event sequence. The machine *chooses* one of the actions (q', σ) in $\delta_M(q, e)$. An important point is that *this choice is nondeterministic*; that is, all the possible cases are considered. It inserts the output event sequence σ *right after the pointer*, advances the pointer by one and changes its state to state q', as shown in Fig. 7.6.

The formal definition of the behavior of the abstract machine is as follows. Let us define function $\xi_M : Q \times E^* \to 2^{E^*}$ for $M = (Q_M, I_M, \delta_M)$. $\xi(q, x)$ is a set of event sequences which is obtained from x by repeating the possible actions starting from q.

$$\begin{aligned} \xi_M(q, \varepsilon) &= \{\varepsilon\}, \\ \xi_M(q, e \cdot x) &= \{e \cdot y \mid (q', \sigma) \in \delta_M(q, e), \end{aligned}$$

$$y \in \xi_M(q', \sigma \cdot x)\}.$$

The formal semantics of the abstract machine M is defined a function $\Xi_M : 2^{E^*} \to 2^{E^*}$. $\Xi_M(X)$ is a set of event sequences obtained from set of event sequences X given as an input stimulus to M.

$$\Xi_M(X) = \{y \mid y \in \xi_M(s, x), s \in I_M, x \in X\}.$$

7.3.3 Description of Behavior of Modules

We can consider many ways of describing the set of states, the set of initial states and state transition function of the abstract machine. In this paper, we use *prolog* for this purpose. In the following discussion, event (p, v) is described as term `p(v)`. An event sequence and a set of event sequences are described as a list of events and a list of lists of events, respectively. For example, event sequence $(a, 0)(y, 1)$ is described as `[a(0),y(1)]`, and set of event sequences $\{(a, 0)(y, 1), (a, 0)(y, 0)\}$ is described as `[[a(0),y(1)],[a(0),y(0)]]`.

In order to specify $M = (Q_M, I_M, \delta_M)$, we describe I_M and δ_M by the predicates `init` and `delta`, respectively. Q is described implicitly. Predicate `init(` M `,` q `)` declares $q \in I_M$, and `delta(` M `,` q `,` e `,` q' `,` s `)` declares $(q', s) \in \delta_M(q, e)$.

We show a description of the behavior of a D-flipflop (`dff`) as a simple example. `dff` has three places; data input `d`, clock input `ck`, and output `q`. Two state variables are used to describe the states of `dff`; the first one represents the current value of `d` and the second one represents the previous value of `ck`.

```
init(dff,[0,0]).                                    (1)
init(dff,[0,1]).                                    (2)
init(dff,[1,0]).                                    (3)
init(dff,[1,1]).                                    (4)
delta(dff,[M,P],d(D), [D,P],[]).                    (5)
delta(dff,[M,0],ck(1),[M,1],[q(M)]).                          (6)
delta(dff,[M,1],ck(0),[M,0],[]).                              (7)
delta(dff,[M,P],ck(P),[M,P],[]).                              (8)
delta(dff,[M,P],    E ,[M,P],[]) :- E\=d(D), E\=ck(C).        (9)
```

(1) $\sim$ (4) declare that there are four possible initial states, which are the formal interpretation of the unknown initial state of `dff`. (5) is interpreted as "when the current state of `dff` is `[M,P]` and there happens an event `d(D)`, the next state is `[D,P]` and there are no output events". Namely, the new value `D` on the data input `d` is taken into the first state variable. (6) describes the behavior of `dff` at a rising edge of `ck`. The current state `[M,0]` (the previous value on `ck` is 0) and an input event `ck(1)` mean a rising edge on `ck`. On this event, there will be an output event `q(M)`. The event `E` in (9) is an event which occurs neither on `d` nor `ck`. In response to the event, `dff` keeps the current state and outputs no events.

Now, we show how `dff` behaves. Suppose an event sequence `[d(1),ck(0),ck(1),d(0)]` is given to `dff`. The behavior of the abstract machine of `dff` is as follows:

1) Chooses one of the initial states declared by an `init` predicate. Here, as an example, we will trace the case where `[0,0]` is chosen. The pointer is set to point the first event of the sequence.

2) Reads the first event `d(1)`. According to the clause (5) above, changes the state to `[1,0]`. Advances the pointer by 1.

3) Reads the next event `ck(0)`, changes the state to `[1,0]` according to the clause (8), and advances the pointer by 1.

4) Reads `ck(1)`. According to the clause (6), changes the state to `[1,1]` and inserts event `q(1)` after `ck(1)`. As a result, the event

sequence becomes [d(1),ck(0),ck(1),q(1),d(0)]. The pointer is advanced by 1, and it points to q(1).

5) Reads q(1). Stay at state [1,1] and outputs no events according to the clause (9). Advances the pointer by 1.

6) Reads d(0). Changes the state to [1,0] and halts.

As a result, we get the final sequence [d(1),ck(0),ck(1),q(1),d(0)]. Although there are four possibilities for the initial state, all of them lead to the same result. In this example, there are no nondeterministic choices of actions because only one action is specified for each pair of the current state and the input event.

7.3.4 Description of a Zero-Delay Unit

Another example, rather sophisticated but demonstrating the expressiveness of the NES model, is concerned with the description of a zero-delay unit zd. zd transmits only the value on its input q to its output y before an occurrence of event t(ns), which means that time progresses by 1 nano second. When there occur more than one event on the input, events corresponding the input occur on the output in arbitrary timing but the events preserve the order of the occurrence. This machine has in mind a queue that keeps the order of the events, and outputs the events from the queue before the event to advance time comes.

```
init(zd,[]).                                  (1)
delta(zd,[],q(Q),  [Q],[]      ).             (2)
delta(zd,[],q(Q),  [], [y(Q)]).               (3)
delta(zd,[],t(ns),[], []      ).              (4)
delta(zd,[],E,       [], []      ):-
  E\=t(ns),E\=q(Q).                           (5)
delta(zd,[H|T],q(Q),HTQ,[]):-
```

```
  append([H|T],[Q],HTQ).                        (6)
delta(zd,[H|T],q(Q),TQ,[y(H)]):-
  append(T,[Q],TQ).                             (7)
delta(zd,[H|T],E,[H|T],[]):-
  E\=t(ns),E\=q(Q).                             (8)
delta(zd,[H|T],E,T,[y(H)]):-
  E \=t(ns),E\=q(Q).                            (9)
```

In this case, the machine shows nondeterministic behavior. For example, in (2) and (3), two possible actions are specified for current state `[]` and an input event `q(Q)`; to take the value `Q` into the queue (clause (2)), or to output the value `Q` immediately (clause (3)). As a result of the state transitions, more than one sequence can be computed for a given sequence. For example, in response to a sequence

```
[t(ns),q(1),d(0),q(0),t(ns)]
```

the abstract machine computes a set consisting of three event sequences:

```
[t(ns),q(1),d(0),q(0),y(1),y(0),t(ns)]
[t(ns),q(1),d(0),y(1),q(0),y(0),t(ns)]
[t(ns),q(1),y(1),d(0),q(0),y(0),t(ns)]
```

Addition of the next lines will make the `zd` a zero-delay unit with *arbitrary inertia.*

```
delta(zd,[H|T],q(Q),[Q],[      ).           (10)
delta(zd,[H|T],q(Q),[] ,[y(Q)]).            (11)
```

7.3.5 Simulation of the Abstract Machine

The semantics of description of abstract machines can be also described as a prolog program. It is possible, therefore, to simulate the abstract machine described in prolog. The program is shown below. Predicate `xi(M,q,x,y)` means that `M` computes event sequence `y` from event sequence `x` by repeating transitions starting from state `q`. This is a straightforward translation of the definition of $y = \xi_M(q, x)$.

```
xi(M,Q,[],[]).
xi(M,Q,[E|X],[E|Y]) :-
   delta(M,Q,E,NQ,Z), append(Z,X,ZX), xi(M,NQ,ZX,Y).
```

The above program works as a prototype simulator of the NES model. For example, in response to the query

```
xi(zd,Q,[t(ns),q(1),d(0),q(0),t(ns)],Y) :- init(zd,Q).
```

there will be the following solutions:

```
Y = [t(ns),q(1),d(0),q(0),y(1),y(0),t(ns)];
Y = [t(ns),q(1),d(0),y(1),q(0),y(0),t(ns)];
Y = [t(ns),q(1),y(1),d(0),q(0),y(0),t(ns)]
```

7.4 Modeling and Descriptions of Connected Modules

7.4.1 Modeling of Connected Modules

Let M_1 and M_2 be abstract machines. We model the parallel operation of M_1 and M_2 by the behavior of a new abstract machine that is constructed from M_1 and M_2. We denote the new machine as $M_1 || M_2$. The places that have an identical name are to be connected. For example, `dff` || `zd` corresponds to the circuit consisting of `dff` and `zd` where the output `q` of `dff` and the input `q` of `zd` are connected with each other. Let $M_1 = (Q_{M_1}, I_{M_1}, \delta_{M_1})$ and $M_2 = (Q_{M_2}, I_{M_2}, \delta_{M_2})$. The state of $M_1 || M_2$ is defined as a composite of the states of M_1 and M_2. When $M_1 || M_2$ is in a state (q_1, q_2) and reads event e, $\delta_{M_1}(q_1, e)$ and

$\delta_{M_2}(q_2, e)$ are computed. The machine chooses $(q_1', \sigma_1) \in \delta_{M_1}(q_1, e)$ and $(q_1', \sigma_1) \in \delta_{M_1}(q_1, e)$. This choice is also nondeterministic. The next state of the machine is (q_1', q_2'). An output event sequence to be inserted after the pointer is chosen from a set *shuffle*(σ_1, σ_2), which is a set of event sequences obtained by shuffling σ_1 and σ_2. For example, *shuffle*$(abc, xy) = \{abcxy, abxcy, abxyc, axbcy, axbyc, axybc, xabcy, xabyc, xaybc, xyabc\}$.

The formal definition of $M_1||M_2$ is as follows.

$$M_1||M_2 = (Q_{M_1} \times Q_{M_2}, I_{M_1} \times I_{M_2}, \delta_{M_1||M_2}),$$

where

$$\begin{aligned}
&\delta_{M_1||M_2}((q_1, q_2), e) = \\
&\quad \{((q_1', q_2'), \sigma) \mid \\
&\qquad (q_1', \sigma_1) \in \delta_{M_1}(q_1, e), \\
&\qquad (q_2', \sigma_2) \in \delta_{M_2}(q_2, e), \\
&\qquad \sigma \in \mathit{shuffle}(\sigma_1, \sigma_2)\},
\end{aligned}$$

and

$$\begin{aligned}
&\mathit{shuffle}(x, y) = \\
&\quad \{x_1 \cdot y_1 \cdot x_2 \cdot y_2 \cdots x_k \cdot y_k \mid x_i, y_i \in E^*, \\
&\quad\; x = x_1 \cdot x_2 \cdots x_k, y = y_1 \cdot y_2 \cdots y_k\}.
\end{aligned}$$

When feedback loops are constructed by connecting modules, a new abstract machine that represents the behavior of the connected modules must read events which it outputs. That is the reason why the abstract machine inserts an output event sequence *right after the pointer.* There is a possibility of infinite looping of computation. This is the semantics of the description because the described circuit in this case oscillates within a unit time.

7.4.2 Description of Connected Modules

Let us describe the set of initial states and the state transition function of the machine $M1||M2||\cdots||Mn$ as

```
init ([M1,M2,...,Mn], q).
delta([M1,M2,...,Mn], q,e,nq,s).
```

These predicates are automatically derived from the description for $M1$, $M2$, $\cdots$, Mn by the following programs. These programs are also straightforward interpretations of the definition in the previous subsection.

```
init([],[]).;
init([M|MS],[Q|QS]) :- init(M,Q), init(MS,QS).

delta([],[],E,[],[]).
delta([M|MS],[Q|QS],E,[NQ|NQS],SHUFFLE):-
  delta(M, Q, E, NQ, Z),
  delta(MS,QS,E, NQS,ZS),
  shuffle(Z, ZS, SHUFFLE ).

shuffle([],Y,Y).
shuffle(X,[],X).
shuffle([H|T],Y,[H|Z]) :- Y \= [], shuffle(T,Y,Z).
shuffle(X,[H|T],[H|Z]) :- X \= [], shuffle(X,T,Z).
```

The following query enables the simulation of the connection of the `dff` and the `zd`.

```
xi([dff,zd],Q,[d(1),ck(0),ck(1),d(0)],Y) :- init([dff,zd],Q).
```

In this case, the description of `zd` in the previous subsection is not enough. The complete version is as follows.

```
init(zd,[[],0]).
delta(zd,[[],N],q(Q), [[Q],N],[]     ).
delta(zd,[[],0],q(Q), [[],1], [y(Q)]).
delta(zd,[[],N],t(ns),[[],N], []     ).
delta(zd,[[],N],y(Y), [[],NN],[]     ):-
  NN is N-1.
delta(zd,[[],N],E,[[],N],[]):-
  E\=t(ns),E\=q(Q),E\=y(Y).
delta(zd,[[H|T],N],q(Q),[HTQ,N],[]):-
  append([H|T],[Q],HTQ).
delta(zd,[[H|T],0],q(Q),[TQ,1],[y(H)]):-
  append(T,[Q],TQ).
delta(zd,[[H|T],0],q(Q),[[],1],[y(Q)]).
delta(zd,[[H|T],N],E,[[H|T],N],[]):-
  E\=t(ns),E\=q(Q),E\=y(Y).
delta(zd,[[H|T],0],E,[T,1],[y(H)]):-
  E\=t(ns),E\=q(Q),E\=y(Y).
delta(zd,[[H|T],1],y(Y),[T,1], [y(H)]).
```

7.5 Applications of the NES Model

7.5.1 Definition of Semantics of UDL/I

In the standardization project of a hardware design language UDL/I, we attempted to define semantics of the language based on the NES model [Kar89, Yas89]. This is done by means of defining rules of translating a description in UDL/I into a description of an abstract machine of the NES model. Since UDL/I has rich syntax and is based on sophisticated default interpretation, it is not a prospective method to reduce a description in UDL/I directly into a description of the abstract machine. We therefore took the following three steps.

1) Reduce a description in UDL/I into a description in a very small subset of UDL/I, named *core subset.*

2) Resolve conflict of multiple outputs in the description obtained in 1) by inserting conflict resolution logic.

3) Define an abstract machine for a conflict free description in the core subset.

In this method, the core subset works as an interface to the behavioral model. This small interface is also useful in comparing the expressive power of UDL/I with that of other HDL's or in discussing the correctness of the interlanguage translation.

The nondeterministic interpretation takes place, for example, in the following cases.

1) Variety of a delay value: When a delay value is specified only by its minimum and maximum values, the value of the delay is decided by a nondeterministic choice at every event occurrence.

2) Order of event evaluation whose before-after relationship is unknown: All the possible orders are taken into account unless there are no before-after relationships, as discussed in 7.2.2.

3) Unspecified signal values: A logic value is decided by a nondeterministic choice from all the possible logic values when the signal value is unknown for the lack of specification or a *don't care* specification.

4) Unknown value caused by signal conflict: When different signal values are assigned from different sources at the same time, we have to resolve the conflict according to the resolution rules. Nondeterministic choice of conflicting signal values is prepared as one of for conflict resolution rules.

Although the semantics of the final version of UDL/I ver1.0 is determined based on the deterministic semantics, the discussion on the nondeterministic semantics have enabled us to identify the statements to which we must pay attention in defining semantics.

7.5.2 Nondeterministic Semantics and CAD Tools

The most important feature of the NES model is that we can explain the behavior of designs including uncertain factors without ambiguity as mentioned in 2.1. This feature is desirable for creating strict discussions on formal verification and synthesis. In the NES model, the behavior of an abstract machine for a given stimulus is dealt with as a *set* of possible behaviors. We can define the *specification-implementation relationship* between two descriptions of designs in terms of the inclusion relationship between the behaviors of abstract machines for the descriptions. For example, suppose S is a register transfer level description of a circuit which computes function f with delay between 10ns and 20ns, and I is a gate level description of a circuit which also computes f with delay between 14ns and 15ns. Then the behavior of I is included by S and we can say I is an implementation of S. This relationship is mathematically described as follows:

$$\forall X \in E^* : \Xi_{M_I}(X) \subseteq \Xi_{M_S}(X),$$

where M_I and M_S are abstract machines for I and S, respectively. Formal design verification is defined as to prove this relation. Synthesis is also defined as to generate an implementation I for S which satisfies the relation. In this way, we can define what is the implementation of a specification without ambiguity, which will be a guideline for the development of verifiers and synthesizers.

On the other hand, nondeterministic semantics may be undesirable for simulator designers. It is very difficult, or almost impossible, to implement an efficient simulator which guarantees, for all kinds of circuits, results consistent with the nondeterministic semantics. However, deterministic simulators which approximate the nondeterministic semantics can be valid under certain design constraints. An usual zero-delay simulator, which runs quite fast, can compute exactly the same results as the nondeterministic semantics if the circuit under test follows the strict synchronous design methodology, while the simulation result is far from the ideal semantics if the circuit is an asynchronous circuit. We consider that simulation is an approximation of the ideal semantics and many simulators of various accuracy and efficiency levels should be chosen according to design styles.

When we attempt to define deterministic semantics of an HDL of the gate level, we must fix very details of the simulation mechanism for the items 1) $\sim$ 3) in the previous subsection. Simulation efficiency is closely related to 2). Accuracy largely depends on 1) and 3). There have been and will be numerous efforts to develop simulation techniques to enhance the efficiency and accuracy of simulation. Especially, symbolic simulation techniques [Car89, Ish90d] will enable accurate and yet efficient simulation. It is not an appropriate approach to define the deterministic semantics that fixes the accuracy and may stop improvements on simulation techniques. We should rather consider to provide ideal semantics that has a good affinity for formal verification and synthesis and can be a final goal for simulation techniques.

7.6 Remarks and Considerations

We have shown basic concepts, formal definition and a description method of the NES model. We have also described the role of the nondeterministic semantics in the applications of HDL's, especially in logic synthesis, formal verification and advanced logic simulation. We believe that the nondeterministic semantics will be an essential feature for HDL's of the next generation.

One weakness of the NES model is that its mathematical handling is difficult despite that it is formal and mathematically defined. It is therefore difficult to develop tools based on formal methods that directly employ the NES model. This difficulty is due to the ability of the NES model to handle infinite behavior within a unit time. In order to find a new possibility of extending the formal application of HDL's, we are also working on a behavioral model whose mathematical handling is easier and more efficient [Kou90].

Chapter 8

Conclusions

In this thesis, acceleration of logic simulation speed, accuracy of timing verification, and a hardware model for formal semantics of hardware description languages are discussed.

As a new approach to accelerating execution speed of logic simulation and fault simulation, efficient use of *vector supercomputers* were proposed.

In chapter 3, three types of simulation algorithms were proposed which are dedicated for 1) zero-delay simulation of combinational circuits, 2) zero-delay simulation of synchronous sequential circuits, and 3) simulation with delay consideration. As well as the algorithms for simulation, the algorithms for preprocessing are also very important for efficient simulation. In order to reduce the storage requirements or to extend the vector length, we proposed some heuristic algorithms based on the data flow sorting. The simulators implemented based on the simulation techniques were shown to have high performance especially in large scale simulation. The performance of our simulators is comparable to that of hardware simulation engines.

In chapter 4, a *dynamic 2-dimensional parallel fault simulation technique* was proposed as a vector supercomputer oriented fault simulation algorithm which is dedicated for the zero-delay two-valued fault simulation of gate-level combinational circuits with single stuck-at faults. Large vector length is obtained by processing many faults for many patterns at a time and fault dropping is efficiently performed by dynamic adjustment of the two parallelism factors. Computation cost was further reduced by combining the algorithm with selective tracing. Experimental results told us that the fault simulator implemented on the FACOM VP-200 supercomputer achieved acceleration ratio of 15 through vectorization and that the simulator is effective for test generation using random patterns, coverage estimation of a large set of random patterns or a built-in self test design.

Vector processors seem to have great potential for not only numerical computation but also for combinational problems in the area of CAD for digital systems. There will be a lot of earnest researches to develop vector processor oriented algorithms for variety of combinational problems. In converse, it is also considered to be important to improve architecture of vector processors suitable to process combinational problems.

In Chapter 5 accuracy of logic simulation was discussed from theoretical point of view, focusing on a *hazard detection problem* of combinational circuits with uncertain delay units. It was shown that the problem of detecting hazards under uncertain delay assumption is NP-hard both in the discrete time model and the continuous time model and that it is hence difficult to solve the problem by a simple extension of the min/max delay simulation technique. It was also shown that there is an essential difference between the discrete time model and the continuous time model and a lower bound of the width between ticks were shown that make the discrete time model equivalent to the continuous time model.

In chapter 6, a new simulation technique named *time-symbolic simulation* and two efficient algorithms for it were presented. The time-symbolic simulation enables us to get conditions where the circuit under test behaves as expected, as well as accurate simulation result event under existence of uncertainty delay. The two algorithms, which are based on the *linear programming* and *Boolean function manipulation* respectively, were shown to be able to verify small scale circuits up to 100 gates within feasible computation time. With the ability to derive delay conditions for correct behavior and the ability to identify critical paths, time-symbolic simulation is considered to be effective for design error correction and design improvements.

In chapter 7, an *NES model* was proposed as a model of hardware that can express uncertain behavior of hardware by means of *nondeterminism*. Discussions were created on the role of the nondeterministic semantics in various new applications of HDL's such as logic synthesis, formal verification and sophisticated logic simulation.

As alternatives to logic simulators on general purpose computers, special purpose hardware for logic simulation (logic simulation engines) have been developed, which achieves very high performance by parallel computation scheme. However, there are trade-offs between simulation speed and flexibility, or affinity for existing CAD systems on general purpose computers. The new approach of developing logic simulators and fault simulators on general purpose vector supercomputer is expected to be the one that fills the gap between the two approaches.

A part of the growth of the computation time and required storage for logic simulation due to circuit size itself will be compensated by the growth of the performance of computers on which logic simulators run. However, the growth of circuit size and circuit complexity cause incidental increase in test pattern size which leads to additional increase in the computation cost. This is the very part of the growth of simulation cost that we must try to reduce with continual efforts to improve simulation efficiency. It is considered to be difficult to encounter this problem only by enhancing the performance of the simulator. Improvements in the way of simulation or the design for easier verification must be taken into account.

The symbolic approach discussed in chapter 6 may be a good suggestion on how to carry out verification efficiently. Although the symbolic simulation is much slower than the simulation for a specific pattern, it is incredibly faster than simulating the same number of the cases separately. Symbolic simulators based on BDD's can verify the logical equivalence of combinational circuits faster than the logic simulator on a vector processor in many cases [Fuj88, Min90]. These are the examples where we can reduce computation cost for verification by efficiently investigating multiple cases at a time. Formal verification [Hir89] is considered to be an ultimate way of examining all the possible cases efficiently. However, there seems to be still a large distance from logic simulation to formal verification in spite of earnest researches on this area, because complete verification without a guide of design knowledge is still too expensive in large scale design. In near future, verification using symbolic simulation is considered to play an important role as a semi-automatic and semi-exhaustive verification method.

Another way of reducing the complexity of design verification problem is utilizing the information of *design hierarchy*. The existence of a certain design hierarchy implies design constraints on the lower level implementation. The computation cost for verification can be drastically reduced by exploiting the information. Also the combination of verification in higher-level and logic synthesis will be a promising approach. In either case, how to describe hardware in various design levels become an important issue. As is discussed in chapter 7, nondeterministic semantics is an important concept in order to express the relations between two circuits of different design levels. This will be a good base for logic synthesis and formal verification. In order to perform formal verification, however, mathematical handling of the basic model must be easy. This was not achieved in the NES model, but is very important in introducing formal ways into the various applications of HDL's. The author believes that the nondeterministic semantics and a formal model which allows mathematical handling are indispensable factors of HDL's of the next generation.

References

[Ant87] K. J. Antreich and M. H. Schulz: "Accelerated Fault Simulation and Fault Grading in Combinational Circuits", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-6, no. 5, pp. 704–712 (Sep. 1987).

[Abr83] M. Abramovici, P. R. Menon and D. T. Miller: "Critical Path Tracing An Alternative to Fault Fault Simulation", *Proc. ACM/IEEE 20th Design Automation Conference*, pp. 214–220 (Jun. 1983).

[Arm72] D. B. Armstrong: "A Deductive Method for Simulating Faults in Logic Circuits", *IEEE Trans. on Computers*, vol. c-21, no. 5, pp. 464–471 (May 1972).

[Bar87] Z. Barzilai, J. L. Carter, B. K. Rosen and J. D. Rutledge: "HSS — A High-Speed Simulator", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 4, (Jul. 1987).

[Bjo78] D. Bjørner and C. B. Jones: *The Vienna Development Method: The Meta Language*, Lecture Notes in Computer Science, vol. 61, Springer-Verlag (1978).

[Bla84] T. Blank: "A Survey of Hardware Accelerators Used in Computer-Aided Design", *IEEE Design & Test of Computers*, pp. 21–39 (Aug. 1984).

[Bre76] M. A. Breuer and A. D. Friedman: *Diagnosis & Reliable Design of Digital Systems*, Computer Science Press (1976).

[Brg85] F. Brglez: "A Fast Fault Grader: Analysis and Applications", *Proc. International Test Conference 1985*, pp. 785–794 (Aug. 1985).

[Brg85f] F. Brglez and H. Fujiwara: "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN", *International Symposium of Circuits and Systems (ISCAS '85)*, Special Session on ATPG and Fault Simulation (Jun. 1985).

[Bry86] R. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, vol. C-35, no. 8, pp. 677-691 (Aug. 1986).

[Bry90] R. E. Bryant: "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", *private communication* (1990), (to appear in *IEEE Transactions on Computers*).

[Car79] W. C. Carter, W. H. Joyner, Jr. and D. Brand: "Symbolic Simulator for Correct Machine Design", *Proc. ACM/IEEE 16th Design Automation Conference*, pp. 280-286 (Jun. 1979).

[Car89] J. L. Carter, B. K. Rosen, G. L. Smith and V. Pichumani: "Restricted Symbolic Evaluation is Fast and Useful", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-89)*, pp. 38-41 (Nov. 1989).

[Cer89] E. Cerny, P. Rioux and C. Berthet: "Comparison of Specification and Implementation for Asynchronous Circuits with Arbitrary Delays", *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pp. 704–720 (Nov. 1989).

[Cha86] T. Chan and E. Law: "MegaFAULT: A Mixed-Mode Hardware Accelerated Concurrent Fault Simulator", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-86)*, pp. 394–397 (Nov. 1986).

[Coe89] D. R. Coelho: *The VHDL Handbook*, Kluwer Academic Publishers (1989).

[Cor81] W. E. Cory: "Symbolic Simulation for Functional Verification with ADLIB and SDL", *Proc. ACM/IEEE 18th Design Automation Conference*, pp. 82-89 (Jun. 1981).

[Deg90] Y. Deguchi, N. Ishiura and S. Yajima: "Coded Time-Symbolic Simulation: Simulation of Logic Circuits with Nondeterministic Delays", *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 149–156 (Oct. 1990).

[Den83] M. Denneau, E. Kronstadt and G. Pfister: "Design and Implementation of a Software Simulation Engine", *CAD*, vol. 15, no. 3, pp. 123–130 (Mar. 1983).

[Fuj85] H. Fujiwara: *Logic Testing and Design for Testability*, MIT Press Series in Computer Systems, The MIT Press, Cambridge, Massachusetts, London, England (1985).

[Fuj88] M. Fujita, H. Fujisawa and N. Kawato: "Evaluations and Improvements of a Boolean Comparison Method Based on Binary Decision Diagrams", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-88)*, pp. 2–5 (Nov. 1988).

[Gar79] M. R. Garey and D. S. Johnson: *Computers and Intractability – A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company (1979).

[Har86] R. W. Hartenstein, R. Hauck, K. Lemmert and A. Wodtko: *KARL-III Manual*, report AG Hartenstein, Kaiserslautern University (1986).

[Har87] D. Harel: "Is There Hope for Linear Time Fault Simulation?", *Proc. IEEE 17th International Symposium on Fault Tolerant Computing (FTCS-17)*, pp. 28–33 (Jun. 1987).

[Hir87] F. Hirose, M. Ishii, J. Niitsuma, T. Shindo, N. Kawato, H. Hamamura, K. Uchida and H. Yamada: "Simulation Processor "SP" ", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-87)*, pp. 484–487 (Nov. 1987).

[Hir88] F. Hirose, K. Takayama and N. Kawato: "A Method to Generate Tests for Combinational Logic Circuits using an Ultrahigh-speed Logic Simulator", *Proc. IEEE International Test Conference 1988*, pp. 102–107 (Sep. 1988).

[Hir89] H. Hiraishi: "Design Verification of Sequential Machines Based on ϵ-Free Regular Temporal Logic", *Proc. IFIP 9th International Symposium on Computer Hardware Description Languages and their Applications (CHDL 89)*, pp. 249–264 (Jun. 1989).

[Ish84] N. Ishiura, H. Yasuura and S. Yajima: "Time First Evaluation Algorithm for High-Speed Logic Simulation", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-84)*, pp. 197–199 (Nov. 1984).

[Ish85yy] N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation by Time First Evaluation Algorithm"(in Japanese), *Trans. IPS Japan*, vol. 26, no. 3, pp. 459–468 (May 1985).

[Ish85ykv] N. Ishiura, H. Yasuura, T. Kawata and S. Yajima: "High-Speed Logic Simulation Using a Vector Processor", *Proc. IFIP International Workshop on VLSI (VLSI 85)*, pp. 67–76 (Aug. 1985).

[Ish85yki] N. Ishiura, H. Yasuura, T. Kawata and S. Yajima: "High-Speed Logic Simulation on a Vector Processor", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-85)*, pp. 119–121 (Nov. 1985).

[Ish86] N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation Using a Vector Processor"(in Japanese), *Trans. IPS Japan*, vol. 27, no. 5, pp. 510–517 (May 1986).

[Ish87] N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation Using a Vector Processor", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-6, no. 3, pp. 305–321 (May 1987).

[Ish88] N. Ishiura and H. Yasuura: "On a Relation between Time-Models and Computation Time of Hazard Detection Problems"(in Japanese), *Report of Technical Group on Foundation of Computation, IEICE*, COMP88-21 (Jun. 1988).

[Ish89] N. Ishiura, M. Takahashi and S. Yajima: "Time-Symbolic Simulation for Accurate Timing Verification of Asynchronous Behavior of Logic Circuits", *Proc. ACM/IEEE 26th Design Automation Conference*, pp. 497–502 (Jun. 1989).

[Ish90y] N. Ishiura, H. Yasuura and S. Yajima: "NES: The Behavioral Model for the Formal Semantics of a Hardware Design Language UDL/I", *Proc. ACM/IEEE 27th Design Automation Conference*, pp. 8–13, (Jun. 1990).

[Ish90d] N. Ishiura, Y. Deguchi and S. Yajima: "Coded Time-Symbolic Simulation Using Shared Binary Decision Diagram", *Proc. ACM/IEEE 27th Design Automation Conference*, pp. 130–135 (Jun. 1990).

[Ish90i] N. Ishiura, M. Ito, and S. Yajima: "Dynamic Two-Dimensional Parallel Simulation Technique for High-Speed Fault Simulation on a Vector Processor", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-9, no. 8, pp. 868–875 (Aug. 1990).

[Ish90tm] N. Ishiura, M. Takahashi and S. Yajima: "Time-Symbolic Simulation for Accurate Timing Verification of Logic Circuits"(in Japanese), *Trans. IPS Japan*, vol. 31, no. 12 (Dec. 1990, to appear).

[Ish90y] N. Ishiura and S. Yajima: "A Class of Logic Functions Expressible by Polynomial-Size Binary Decision Diagrams", *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 48–54 (Oct. 1990).

[Jai84] S. K. Jain and V. D. Agrawal: "STAFAN: An Alternative to Fault Simulation", *Proc. ACM/IEEE 21st Design Automation Conference*, pp. 18–23 (Jun. 1984).

[Kar89] O. Karatsu: "VLSI Design Language Standardization Effort in Japan", *Proc. ACM/IEEE 26th Design Automation Conference*, pp. 50–55 (Jun. 1989).

[Kim88] S. Kimura, H. Haneda and S. Yajima: "Verification of Asynchronous Sequential Circuits Based on Regular Expression Logic Simulation Method"(in Japanese), *Trans. IEICE*, vol. J71-D, no. 9, pp. 1–10 (Sep. 1988).

[Koe86] S. Koeppe: "Modeling and simulation of delay faults in CMOS logic circuits", *Proc. IEEE International Test Conference 1986*, pp. 530–536, (Sep. 1986).

[Kou90] Y. Koumura: "Formal Semantics of Hardware Description Languages Based on Nondeterministic Sequential Machines", Master thesis, Department of Information Science, Faculty of Engineering, Kyoto University, Japan (Feb. 1990).

[Kro81] H. E. Krohn. "Vector coding techniques for high speed digital simulation". *Proc. ACM/IEEE 18th Design Automation Conference*, pp. 525–528 (Jun. 1981).

[Lub85] O. Lubeck, J. Moore and R. Mendez: "A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S-810/20 and Cray X-MP/2", *Computer*, vol. 18, no. 12, pp. 10–24 (Dec. 1985).

[Min90] S. Minato, N. Ishiura and S. Yajima: "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", *Proc. ACM/IEEE 27th Design Automation Conference*, pp. 52–57 (Jun. 1990).

[Min91] S. Minato, N. Ishiura and S. Yajima: "Share Binary Decision Diagrams for Efficient Boolean Function Manipulation"(in Japanese), *Trans. IPS Japan*, vol. 32, no. 1 (Jan. 1991, to appear).

[Mot86] A. Motohara, K. Nishimura, H. Fujiwara and I. Shirakawa: "A Parallel Scheme for Test-Pattern Generation", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-86)*, pp. 156–159 (Nov. 1986).

[Mur72] S. Muroga and T. Ibaraki: "Design of Optimal Switching Networks by Integer Programming", *IEEE Trans. Computers*, vol. C-21, no. 6 (1972).

[Nag86] S. Nagashima, T. Nakagawa, K. Omota, S. Miyamoto: "Hardware Implementation of VELVET on the Hitachi S-810 Supercomputer", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-86)*, pp. 390–393 (Nov. 1986).

[Nak86] T. Nakata and N. Koike: "Functional Simulation Engine of MAN-YO: a Special Purpose Parallel Machine for Logic Design Automation", *Proc. IEEE 13th Annual International Symposium on Computer Architecture*, pp. 191-197 (Jun. 1986).

[Nak87] H. Nakamura, M. Fujita, S. Kono and H. Tanaka: "Temporal Logic Based Fast Verification System Using Cover Expressions", *Proc. IFIP International Workshop on Very Large Scale Integration (VLSI 87)* (Aug. 1987).

[Nis85] T. Nishida, S. Miyamoto, T. Kozawa and K. Sato: "RFSIM: Reduced Fault Simulator", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-85)*, pp. 13–15 (Nov. 1985).

[Ohm90] M. Ohmura: "Extraction of Logic and Arithmetic Functions from Combinational Circuits", Master thesis, Department of Electronics, Faculty of Engineering, Kyoto University (Feb. 1990).

[Pil83] R. Piloty, M. Barbacci, D. Borrione, D. Dietmeyer, F. Hill and P.'Skelly, *CONLAN Report*, Lecture Notes in Computer Science, vol. 151, Springer-Verlag (1983).

[Sak82] T. Sakai, Y. Tsuchida, H. Yasuura, Y. Ooi, Y. Ono, H. Kano, S. Kimura and S. Yajima : "An Interactive Simulation System for Structured Logic Design – ISS", *Proc. ACM/IEEE 19th Design Automation Conference*, pp. 747–755 (Jun. 1982).

[Sas83] T. Sasaki, N. Koike, K. Ohmori and K. Tomita: "HAL: A Block Level Hardware Logic Simulator", *Proc. ACM/IEEE 20th Design Automation Conference*, pp. 150–156 (Jun. 1983).

[Seg83] M. T. M. Segers: "Testability in a VLSI Environment", *VLSI architecture*, pp. 175–195, Prentice Hall International Inc., NJ. U. S. A (1983).

[Smi86] R. J. Smith: "Fundamentals of Parallel Logic Simulation", *Proc. ACM/IEEE 23rd Design Automation Conference*, pp. 2–12 (Jun. 1986).

[Sta85] J. Staples and V. L. Nguyen: "A fixed point semantics for nondeterministic data flow", *Journal of ACM*, vol. 23, no. 4, pp. 733-742 (Oct. 1976).

[Tes87] S. Teshima, H. Hiraishi and S. Yajima: "Algebraic Specification of Parallel Systems Based on Binary Relations between Events", *Trans. IEICE*, vol. J70-D, no. 1, pp. 19–29 (Jan. 1987).

[Ulr69] E. Ulrich: "Exclusive Simulation of Activity in Digital Networks", *Communications ACM*, vol. 13, pp. 102–110 (Feb. 1969).

[Ulr80a] E. Ulrich et. al: "High-Speed Concurrent Fault Simulation with Vectors and Scalars", *Proc. ACM/IEEE 17th Design Automation Conference*, pp. 374–380 (Jun. 1980).

[Ulr80b] E. Ulrich: "Table Look-Up Techniques for Fast and Flexible Digital Logic Simulation", *Proc. ACM/IEEE 17th Design Automation Conference*, pp. 560–563 (Jun. 1980).

[Ulr83] E. Ulrich: "A Design Verification Methodology Based on Concurrent Simulation and Clock Suppression", *Proc. ACM/IEEE 20th Design Automation Conference*, pp. 709–712 (Jun. 1983).

[Yas89] H. Yasuura and N. Ishiura: "Semantics of a Hardware Design Language for Japanese Standardization", *Proc. ACM/IEEE 26th Design Automation Conference*, pp. 836–839 (Jun. 1989).

[Yas90] H. Yasuura and N. Ishiura: "Formal Semantics of UDL/I and Its Applications to CAD/DA tools", *Proc. IEEE International Conference on Computer Design (ICCD '90)*, pp. 90–94 (Sep. 1990).

[Yon89] T. Yoneda, K. Nakade and Y. Tohma: "A Fast Timing Verification Method Based on the Independence of Units", *Proc. IEEE 19th International Symposium on Fault Tolerant Computing (FTCS-19)*, pp. 134-141 (Jun. 1989).

[Wai85] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom and T. McCarthy. "Fault Simulation for Structured VLSI", *VLSI Systems Design*, pp. 20–32 (Dec. 1985).

[Wai89] J. A. Waicukauski, E. Lindbloom, E. B. Eichelberger and D. O. Forlenza "A Method for Generating Weighted Random Test Patterns", *IBM Journal of Research and Development*, vol. 33, no. 2, pp. 149–161 (Mar. 1989).

Acknowledgment

I would like to express my sincere appreciation to Professor Shuzo Yajima of Kyoto University for his continuous guidance, interesting suggestions, accurate criticisms and encouragements during this research.

I would also like to express my thanks to Associate Professor Hiroto Yasuura of Kyoto University who introduced me to the research field of computer-aided design and has been giving me invaluable suggestions, accurate criticisms and encouragements throughout this research.

I also acknowledge interesting comments that I have received from Associate Prof. Hiromi Hiraishi, Dr. Naofumi Takagi of Kyoto University.

I would like to thank Mr. Tetsuro Kawata, who was with Kyoto University, for his help in carrying out the experiments on the logic simulation in chapter 3.

I would also like to thank Mr. M. Kawai of the NEC Corporation for his valuable comments on parallel fault simulation, Mr. Masaki Ito, Mr. Masaki Kume, Mr. Tatsuya Ohnishi and Mr. Hideo Nakata for their help in carrying out the experiments on the fault simulation in chapter 4.

I would like to thank Dr. Hitoshi Nagamochi, who was with Kyoto University, for his valuable comments on the resolution of the linear programming and Mr. Yasuo Okabe for his comments on the computational complexity theory, with respect to the results in chapter 5.

I would like to thank Mr. Mizuki Takahashi and Mr. Yutaka Deguchi for their help in carrying out experiments on time-symbolic simulation and coded time-symbolic simulation, respectively, in chapter 6. I would like to thank Mr. Shin-ichi Minato who offered me the SBDD manipulation program which is indispensable in the implementation of the coded time-symbolic simulator. I would also like to thank Mr. Hiroaki Kanehara for his discussions on timing simulation.

I would like to thank all the members of the LSI Design Language standardization committee of JEIDA (Japan Electronic Industry Development Association) and Mr. Yasuhito Koumura for their intensive discussion on the NES model and semantics of hardware description languages.

I would like to thank Mr. Noriyuki Takahashi who helped me printing this thesis. Thanks are also due to all the members of the Professor Yajima's Laboratory for their discussions and supports throughout this research.

List of Publications by the Author

Major Pulications

1. H. Yasuura, H. Kano, Y. Ooi, S. Kimura, N. Ishiura and S. Yajima: "ISS: An Interactive Simulation with Input Constraints Monitoring Facility" (in Japanese), *Trans. IPSJ*, vol. 25, no. 2, pp. 285–292 (Mar. 1984).

2. N. Ishiura, H. Yasuura and S. Yajima: "Time First Evaluation Algorithm for High-Speed Logic Simulation", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-84)*, pp. 197–199 (Nov. 1984).

3. N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation by Time First Evaluation Algorithm" (in Japanese), *Trans. IPSJ*, vol. 26, no. 3, pp. 459–468 (May 1985).

4. N. Ishiura, H. Yasuura, T. Kawata and S. Yajima: "High-Speed Logic Simulation Using a Vecter Processer", *Proc. IFIP International Conference on Very Large Scale Integration (VLSI85)*, pp. 73–82 (Aug. 1985).

5. N. Ishiura, H. Yasuura, T. Kawata and S. Yajima: "High-Speed Logic Simulation on a Vector Processor", *Proc. IEEE Internaltional Conference on Computer-Aided Design (ICCAD-85)*, pp. 119–121 (Nov. 1985).

6. N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation Using a Vector Processor" (in Japanese), *Trans. IPSJ*, vol. 27, no. 5, pp. 510–517 (May 1986).

7. N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation on Vector Processors", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-6, no. 3, pp. 305–321 (May 1987).

8. N. Ishiura, M. Ito and S. Yajima: "High-Speed Fault Simulation Using a Vector Processor", *Proc. IEEE International Conference on Computer-Aided Design (ICCAD-87)*, pp. 10–13 (Nov. 1987).

9. N. Ishiura, N. Takagi and S. Yajima: "Sorting on a Vector Processor" (in Japanese), *Trans. IPSJ*, vol. 29, no. 4, pp. 378–385 (Apr. 1988).

10. N. Ishiura, M. Ito and S. Yajima: "Dynamic Two-Dimensional Parallel Simulation Technique for High-Speed Fault Simulation on a Vector Processor" (in Japanese), *Trans. IPSJ*, vol. 29, no. 5, pp. 522–528 (May 1988).

11. N. Ishiura, M. Takahashi and S. Yajima: "Time-Symbolic Simulation for Accurate Timing Verification of Asynchronous Behavior of Logic Circuits", *Proc. ACM/IEEE 26th Design Automation Conference*, pp. 497–502 (Jun. 1989).

12. H. Yasuura and N. Ishiura: "Semantics of a Hardware Design Language for Japanese Standardization", *Proc. ACM/IEEE 26th Design Automation Conference*, pp. 836–839 (Jun. 1989).

13. S. Minato, N. Ishiura and S. Yajima: "Fast Tautology Checking Using Shared Binary Decision Diagram - Benchmark Results -", *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, vol. 2, pp. 580–584 (Nov. 1989).

14. N. Ishiura, H. Yasuura and S. Yajima: "NES: The Behavioral Model for the Formal Semantics of a Hardware Design Language UDL/I", *Proc. ACM/IEEE 27th Design Automation Conference*, pp. 8–13, (Jun. 1990).

15. S. Minato, N. Ishiura and S. Yajima: "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", *Proc. ACM/IEEE 27th Design Automation Conference*, pp. 52–57, (Jun. 1990).

16. N. Ishiura, Y. Deguchi and S. Yajima: "Coded Time-Symbolic Simulation Using Shared Binary Decision Diagram", *Proc. ACM/IEEE 27th Design Automation Conference*, pp. 130–135, (Jun. 1990).

17. N. Ishiura, M. Ito and S. Yajima: "Dynamic Two-Dimensional Parallel Simulation Technique for High-Speed Fault Simulation on a Vector Processor", *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-9, no. 8, pp. 868–875 (Aug. 1990).

18. H. Yasuura and N. Ishiura: "Formal Semantics of UDL/I and Its Applications to CAD/DA tools", *Proc. IEEE International Conference on Computer Design (ICCD '90)*, pp. 90–94 (Sep. 1990).

19. N. Ishiura and S. Yajima: "A Class of Logic Functions Expressible by Polynomial-Size Binary Decision Diagrams", *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 48–54 (Oct. 1990).

20. Y. Deguchi, N. Ishiura and S. Yajima: "Coded Time-Symbolic Simulation: Simulation of Logic Circuits with Nondeterministic Delays", *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 149–156 (Oct. 1990).

21. N. Takahashi, N. Ishiura and S. Yajima: "Fault Simulation for Multiple Faults Using Shared Binary Decision Diagrams", *Proc. Synthesis and Simulation Meeting and International Interchange*, pp. 157–164 (Oct. 1990).

22. N. Ishiura, M. Takahashi and S. Yajima: "Time-Symbolic Simulation for Accurate Timing Verification of Logic Circuits" (in Japanese), *Trans. IPSJ*, vol. 31, no. 12 (Dec. 1990, to appear).

23. S. Minato, N. Ishiura and S. Yajima: "Share Binary Decision Diagrams for Efficient Boolean Function Manipulation" (in Japanese), *Trans. IPSJ*, vol. 32, no. 1 (Jan. 1991, to appear).

Technical Reports

1. H. Kano, Y. Ooi, S. Kimura, N. Ishiura, H. Yasuura and S. Yajima: "ISS: Interactive Logic Design and Verification Support System" (in Japanese), Report of Technical Group on Design Automation, IPS Japan, 15-1 (Dec. 1982).

2. N. Ishiura, H. Yasuura and S. Yajima: "High-Speed Logic Simulation by Time First Evaluation Algorithm" (in Japanese), Report of Technical Group on Electric Computers, IECE, EC84-49 (Dec. 1982).

3. N. Ishiura, H. Yasuura, T. Kawata and S. Yajima: "High-Speed Logic Simulation Using a Vector Processor" (in Japanese), Report of Technical Group on Design Automation, IPS Japan, 25-2 (Feb. 1985).

4. N. Ishiura, M. Kume, H. Yasuura and S. Yajima: "Parallel Fault Simulation Using a Vector Processor" (in Japanese), Report of Technical Group on Fault Tolerant Systems, IECE, FTS86-4 (May 1986).

5. N. Ishiura, H. Yasuura and S. Yajima: "Performance Evaluation of Logic Simulator on Vector Processors" (in Japanese), Report of Technical Group on Circuit and Systems, IECE, CAS86-82 (Sep. 1986).

6. H. Yasuura and N. Ishiura: "On Computational Complexity of Hazard Detection Problems" (in Japanese), Report of Technical Group on Theoretical Foundations of Computing, IEICE, COMP86-64 (Jan. 1987).

7. N. Ishiura, N. Takagi and S. Yajima: "Sorting on Vector Processors" (in Japanese), Report of Technical Group on Theoretical Foundations of Computing, IEICE, COMP86-88 (Mar. 1987).

8. N. Ishiura, M. Ito, H. Yasuura and S. Yajima: "On Dynamic 2-Dimensional Parallel Fault Simulation on a Vector Processor" (in Japanese), Report of Technical Group on VLSI Design Technology, IEICE, VLD87-2 (Apr. 1987).

9. N. Ishiura and S. Yajima: "On Time-Symbolic Simulation" (in Japanese), Report of Technical Group on VLSI Design Technology, IEICE, VLD87-112 (Dec. 1987).

10. N. Ishiura and H. Yasuura: "On a Relation between Time-Models and Computationa Time of Hazard Detection Problems" (in Japanese), Report of Technical Group on Theoretical Foundations of Computing, IEICE, COMP88-21 (Jun. 1988).

11. M. Ito, N. Ishiura and S. Yajima: "Test Generation using a Fast Fault Simulator on a Vector Processor" (in Japanese), Report of Technical Group on VLSI Design Technology, IEICE, VLD88-27 (Jul. 1988).

12. N. Ishiura and H. Yasuura: "On Computational Complexity of Hazard Detection Problems of Combinational Circuits" (in Japanese), RIMS Koukyuroku, vol. 666, pp. 51–60, Research Institute for Mathematical Science, Kyoto University (Jul. 1988).

13. M. Takahashi, N. Ishiura and S. Yajima: "Result-Analysis System for Time-Symbolic Logic Simulation" (in Japanese), Report of Technical Group on Design Automation, IPS Japan, 44-2 (Oct. 1988).

14. N. Ishiura and S. Yajima: "A Nondeterministic Behavior Model for Definition of Formal Semantics of Hardware Description Languages" (in Japanese), Report of Technical Group on VLSI Design Technology, IEICE, VLD89-3 (Apr. 1989).

15. Y. Koumura, N. Ishiura and S. Yajima: "Formal Semantics of Hardware Description Languages Based on Nondeterministic Sequential Machines" (in Japanese), Report of Technical Group on VLSI Design Technology, IEICE, VLD89-75 (Dec. 1989).

16. S. Minato, N. Ishiura and S. Yajima: "Shared Binary Decision Diagram for Efficient Boolean Function Manipulation" (in Japanese), Report of Technical Group on VLSI Design Technology, IEICE, VLD89-80 (Dec. 1989).

17. N. Ishiura, Y. Deguchi and S. Yajima: "Coded Time-Symbolic Simulation Using Shared Binary Decision Diagram", Report of Technical Group on VLSI Design Technology, IEICE, VLD89-81 (Dec. 1989).

18. Y. Deguchi, N. Ishiura and S. Yajima: "Analysis of Timing Error Probability Based on Coded Time-Symbolic Simulation", Report of Technical Group on VLSI Design Technology, IEICE, VLD90-89 (Dec. 1990).

19. N. Takahashi, N. Ishiura and S. Yajima: "Fault Simulation for Multiple Faults Using Shared Binary Decision Diagrams", Report of Technical Group on VLSI Design Technology, IEICE, VLD90-93 (Dec. 1990).