3. LOGIC SIMULATION

About This Chapter

First we review different aspects of using logic simulation as a tool for design verification testing, examining both its usefulness and its limitations. Then we introduce the main types of simulators, compiled and event-driven. We analyze several problems affecting the accuracy and the performance of simulators, such as treatment of unknown logic values, delay modeling, hazard detection, oscillation control, and values needed for tristate logic and MOS circuits. We discuss in detail techniques for element evaluation and gate-level event-driven simulation algorithms. The last section describes special-purpose hardware for simulation.

3.1 Applications

Logic simulation is a form of *design verification testing* that uses a model of the designed system. Figure 3.1 shows a schematic view of the simulation process. The simulation program processes a representation of the input stimuli and determines the evolution in time of the signals in the model.

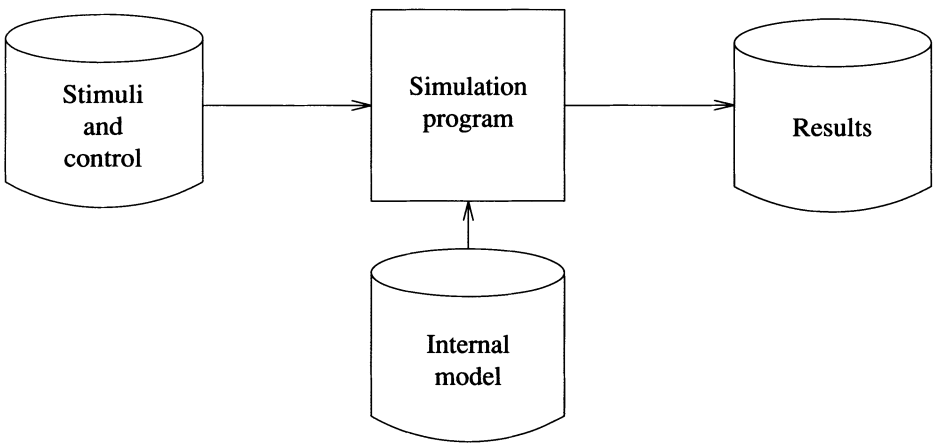


Figure 3.1 Simulation process

The verification of a logic design attempts to ascertain that the design performs its specified behavior, which includes both function and timing. The verification is done by comparing the results obtained by simulation with the expected results provided by the specification. In addition, logic simulation may be used to verify that the operation of the system is

- correct independent of the initial (power-on) state;
- not sensitive to some variations in the delays of the components;
- free of critical races, oscillations, "illegal" input conditions, and "hang-up" states.

Other applications of simulation in the design process are

- *evaluation of design alternatives* ("what-if" analysis), to improve performance/cost trade-offs;
- *evaluation of proposed changes* of an existing design, to verify that the intended modifications will produce the desired change without undesired side effects;
- *documentation* (generation of *timing diagrams*, etc.).

Traditionally, designers have used a prototype for the verification of a new design. The main advantage of a prototype is that it can run at operating speed, but building a prototype is costly and time-consuming. Prototypes built with discrete components lack accuracy as models of complex ICs. Simulation *replaces the prototype with a software model*, which is easily analyzed and modified. Simulation-based design verification benefits from additional features not available to a prototype-based process, such as

- checking error conditions (e.g., bus conflicts);
- ability to change delays in the model to check worst-case timing conditions;
- checking user-specified expected values during simulation;
- starting the simulated circuit in any desired state;
- precise control of the timing of asynchronous events (e.g., interrupts);
- the ability to provide an automated testing environment for the simulated circuit, by coupling it with an RTL model that drives and/or observes the circuit during simulation.

Interestingly, simulation has also been used during the *debugging of the prototype* of a new system. As reported in [Butler *et al.* 1974], designers found that tracing some problems on the simulated model was easier than tracing them on the prototype hardware. Although simulation runs much slower than the hardware, debugging using simulation provides the user with the ability to suspend the simulation on the occurrence of user-specified conditions and to display the value of any desired signal, including lines not directly observable on the hardware. Such a feature becomes more valuable in LSI-based designs, where the number of signals not directly observable is large.

Debugging software or microcode to run on hardware still under development can be started using a simulation model before a prototype is available.

Another use of logic simulation is to *prepare the data base for guided-probe testing*, consisting of the expected values of the lines accessible for probing.

3.2 Problems in Simulation-Based Design Verification

Three interrelated problems in simulation-based design verification testing (and in any type of testing experiment as well) are

- How does one generate the input stimuli? (test generation)
- How does one know the results are correct?
- How "good" are the applied input stimuli, i.e., how "complete" is the testing experiment? (test evaluation)

The input stimuli are usually organized as a sequence of *test cases*, where a test case is intended to verify a certain aspect of the behavior of the model. The results are considered correct when they match the expected results, which are provided by the specification of the design. Initially, the specification consists of an informal (mental or written) model of the intended behavior of the system. During a top-down design process, the highest-level formal model (usually an RTL model) is checked against the initial informal model. After that, any higher-level model defines the specification for its implementation at the next lower level. Checking that the implementation satisfies the specification is reduced to applying the same test cases to the two models and using the results obtained from the higher level as the expected results for the lower level. An example of such a hierarchical approach is the design verification process described in [Sasaki *et al.* 1981].

It is important to understand the difference between test generation for design verification, where the objective is to find design errors, and test generation for detecting physical faults in a manufactured system. Most types of physical faults can be represented by logical faults, whose effects on the behavior of the system are well defined. Based on the difference between the behavior in the presence of a fault and the fault-free behavior, one can derive a test for that fault. Many logical fault models allow the possible faults to be enumerated. The existence of a set of enumerable faults allows one to determine the quality of a test by computing the *fault coverage* as the ratio between the number of faults detected by that test and the total number of faults in the model. In contrast, the space of design errors is not well defined and the set of design errors is not enumerable. Consequently, it is impossible to develop test generation algorithms or rigorous quality measures for design verification tests.

Although the set of design errors is not enumerable, experience shows that most design errors are related to the sequencing of the data transfers and transformations rather than to the data operations themselves. The reason is that data (i.e., arithmetic and logic) operations are more regular and "local," while their control and timing in a complex system may depend on other operations occurring concurrently. Therefore, the usual strategy in design verification is to emphasize exercising the control. For example, minimal test cases for an instruction set processor consist of executing every instruction in the repertoire. In addition, the test designer must consider sequences of instructions that are relevant to the operation of the processor, interactions between these sequences and interrupts, and so on.

Design verification via simulation suffers from several limitations. As there are no formal procedures to generate tests, producing the stimuli is generally a heuristic process that relies heavily on the designer's intuition and understanding of the system

under test. A system that passes the test is shown to be correct only with respect to the applied test cases, and hence only a partial correctness can be proved. Moreover, the completeness of the tests cannot be rigorously determined. (Some heuristic measures used to determine the quality of a test with respect to the control flow of the system will be presented in Chapter 8.)

In spite of these limitations, simulation is an effective technique for design verification, and experience (for example, [Monachino 1982]) has shown that it helps discover most design errors early in the design process. For LSI/VLSI circuits, where design errors are costly and prototypes are impractical, logic simulation is an invaluable aid.

3.3 Types of Simulation

Simulators can be classified according to the type of internal model they process. A simulator that executes a compiled-code model is referred to as a *compiler-driven simulator*, or a *compiled simulator*. The compiled code is generated from an RTL model, from a functional model written in a conventional programming language, or from a structural model. A simulator that interprets a model based on data structures is said to be *table-driven*. The data structures are produced from an RTL model or a structural model. The interpretation of the model is controlled by the applied stimuli, and results in a series of calls to the routines implementing primitive operators (for an RTL model) or primitive components (for a structural model).

Let us consider a circuit in operation and look at the signals changing value at some arbitrary time. These are called *active* signals. The ratio between the number of active signals and the total number of signals in the circuit is referred to as *activity*. In general, the activity of a circuit is between 1 and 5 percent. This fact forms the basis of *activity-directed simulation* [Ulrich 1965, 1969], which simulates only the active part of the circuit.

An **event** represents a change in the value of a signal line. When such an event on line i occurs, the elements having i as input are said to be *activated*. The process of determining the output values of an element is called **evaluation**. Activity-directed simulation evaluates only the activated elements. Some of the activated elements may in turn change their output values, thus generating new events. As activity is caused by events, activity-directed simulation is also referred to as *event-driven simulation*. To propagate events along the interconnections among elements, an event-driven simulator needs a structural model of a circuit. Hence event-driven simulation is usually table driven.

Compiled simulation is mainly oriented toward functional verification and is not concerned with the timing of the circuit. This makes it applicable mostly to synchronous circuits, for which timing can be separately verified [Hitchcock 1982]. In contrast, the passage of time is central to event-driven simulation, which can work with accurate timing models. Thus event-driven simulation is more general in scope, being also applicable to asynchronous circuits.

Event-driven simulation can process *real-time inputs*, that is, inputs whose times of change are independent of the activity in the simulated circuit. This is an important feature for design verification testing, as it allows accurate simulation of nonsynchronized events, such as interrupts or competing requests for use of a bus.

Compiled simulation allows inputs to be changed only when the circuit is stable. This is adequate when the input stimuli are *vectors* applied at a fixed rate. Note that real-time inputs include the fixed-rate vectors as a particular case.

Often, the two simulation types are combined, such that an event-driven algorithm propagates events among components, and the activated components are evaluated by compiled-code models.

The *level of simulation* corresponds to the level of modeling employed to represent the simulated system. Thus, we can have

- *register-level simulation*, for systems modeled entirely in RTL or as an interconnection of components modeled in RTL;
- *functional-level simulation*, for systems modeled as an interconnection of primitive functional blocks (sometimes this term is also used when the components are modeled in RTL);
- *gate-level simulation*;
- *transistor-level simulation* (we consider only logic-level and not circuit-level analog simulation);
- *mixed-level simulation.*

3.4 The Unknown Logic Value

In general, the response of a sequential circuit to an input sequence depends on its initial state. However, when a circuit is powered-up, the initial state of memory elements such as F/Fs and RAMs is usually unpredictable. This is why, before the normal operation begins, an initialization sequence is applied with the purpose of bringing the circuit into a known "reset" state. To process the unknown initial state, simulation algorithms use a separate logic value, denoted by u, to indicate an **unknown logic value**. The u logic value is processed together with the binary logic values during simulation. The extension of Boolean operators to 3-valued logic is based on the following reasoning. The value u represents one value in the set $\{0,1\}$. Similarly, we can treat the values 0 and 1 as the sets $\{0\}$ and $\{1\}$, respectively. A Boolean operation B between p and q, where $p,q \in \{0,1,u\}$, is considered an operation between the sets of values representing p and q, and is defined as the union set of the results of all possible B operations between the components of the two sets. For example,

$$AND(0,u)=AND(\{0\},\{0,1\})=\{AND(0,0),AND(0,1)\}=\{0,0\}=\{0\}=0$$

$$OR(0,u)=OR(\{0\},\{0,1\})=\{OR(0,0),OR(0,1)\}=\{0,1\}=u$$

Similarly, the result of NOT(q), where $q \in \{0,1,u\}$, is defined as the union set of the results of NOT operations applied to every component of the set corresponding to q. Hence

$$NOT(u)=NOT(\{0,1\})=\{NOT(0),NOT(1)\}=\{1,0\}=\{0,1\}=u$$

Figure 3.2 shows the truth tables for AND, OR, and NOT for 3-valued logic. A general procedure to determine the value of a combinational function $f(x_1, x_2, ...,x_n)$ for a given input combination $(v_1\ v_2\ \cdots\ v_n)$ of 0, 1, and u values, works as follows:

1. Form the cube $(v_1\ v_2\ \cdots\ v_n \mid x)$.
2. Using the modified intersection operator given in Figure 3.3, intersect this cube with the primitive cubes of f. If a consistent intersection is found, then the value of f is obtained in the right-most position; otherwise set f=u.

AND	0	1	u
0	0	0	0
1	0	1	u
u	0	u	u

OR	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

NOT	0	1	u
	1	0	u

Figure 3.2 Truth tables for 3-valued logic

$\cap$	0	1	x	u
0	0	$\varnothing$	0	$\varnothing$
1	$\varnothing$	1	1	$\varnothing$
x	0	1	x	u
u	$\varnothing$	$\varnothing$	u	u

Figure 3.3 Modified intersection operator

To understand this procedure, recall that an x in a primitive cube denotes a "don't care" value; hence an unknown value is consistent with an x in a primitive cube. However, a binary input value specified in a primitive cube is a required one, so a u on that input cannot generate the corresponding output value. For example, for an AND gate with two inputs, the cube $u\,0 \mid x$ matches a primitive cube, but $u\,1 \mid x$ does not.

There is a loss of information associated with the use of 3-valued logic [Breuer 1972]. This can be seen from the NOT truth table: in the case where both the input and the output have value u, we lose the complementary relation between them. The same effect occurs between complementary outputs of a F/F whose state is u. In the presence of reconvergent fanout with unequal inversion parities, this loss of information may lead to pessimistic results. This is illustrated in Figure 3.4, where the output of NAND gate is actually 1, but is computed as u according to the rules of 3-valued logic.

It may appear that the use of complementary unknown values u and $\bar{u}$, along with the rules $u.\bar{u}$=0 and $u+\bar{u}$=1, would solve the above problems. This is true, however, only

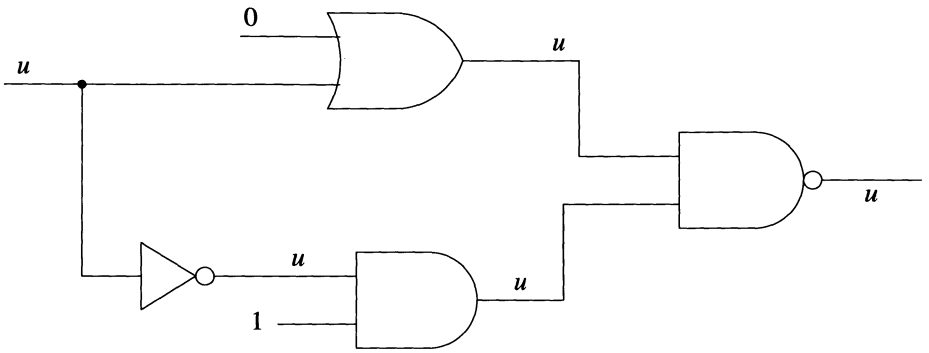


Figure 3.4 Pessimistic result in 3-valued simulation

when we have only one state variable set to u. Figure 3.5 illustrates how the use of u and $\overline{u}$ may lead to incorrect results. Since it is better to be pessimistic than incorrect, using u and $\overline{u}$ is not a satisfactory solution. A correct solution would be to use several distinct unknown signals u_1, u_2, ..., u_k (one for every state variable) and the rules $u_i.\overline{u}_i=0$ and $u_i+\overline{u}_i=1$. Unfortunately, this technique becomes cumbersome for large circuits, since the values of some lines would be represented by large Boolean expressions of u_i variables.

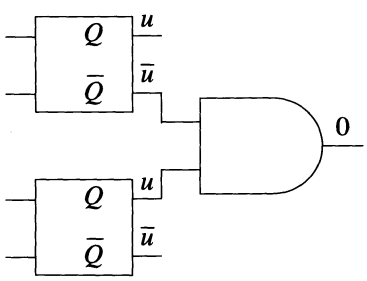


Figure 3.5 Incorrect result from using u and $\overline{u}$

Often the operation of a functional element is determined by decoding the values of a group of control lines. A problem arises in simulation when a functional element needs to be evaluated and some of its control lines have u values. In general, if k control lines have u values, the element may execute one of 2^k possible operations. An accurate (but potentially costly) solution is to perform all 2^k operations and to take as result the union set of their individual results. Thus if a variable is set to 0 in some operations and to 1 in others, its resulting value will be $\{0,1\} = u$. Of course, this

solution is practical if 2^k is a small number. For example, assume that two bits in the address of a ROM have u values. This leads to four evaluations, each accessing a different word. The resulting output will have a binary value b in those bit positions where every accessed word has value b, and u values wherever the accessed words do not match.

In an asynchronous circuit, the presence of u values may indicate an oscillation, as shown in Figure 3.6. Signals involved in a high-frequency oscillation often assume a voltage between the levels corresponding to logic 0 and 1. Thus, in addition to a static unknown value, u may also represent a dynamic unknown value or an indeterminate logic value.

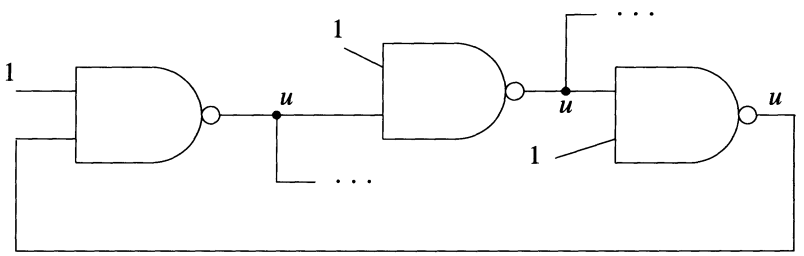


Figure 3.6 Oscillation indicated by u values

3.5 Compiled Simulation

In compiled simulation, the compiled-code model becomes part of the simulator. In the extreme case, the simulator is nothing but the compiled-code model. Then this code also reads the input vectors and outputs the results. In general, the compiled-code model is linked with the simulator's core, whose tasks include reading the input vectors, executing the model for every vector, and displaying the results.

We will illustrate the operation of a compiled simulator using the circuit of Figure 3.7. It is a synchronous circuit controlled by the periodic clock signal *CLK*. We assume that, after a new input vector is applied, there is enough time for the data input of the F/F to become stable at least t_{setup} before the F/F is clocked, where t_{setup} is the setup time of the F/F. This assumption can be independently verified by a timing verification program [Hitchcock 1982]. If this assumption is satisfied, the simulation can ignore the individual gate delays, as the exact times when signals in the combinational logic change are not important. Then, for every vector, the simulation needs only to compute the static value of F and to transfer this value to Q.

The code model is generated such that the computation of values proceeds level by level. This assures that whenever the code evaluates a gate, the gates feeding it have already been evaluated. The values of the primary inputs A and B (which have level 0) are read from a stimulus file. The only other signal with level 0 is the state variable Q; first we assume that its initial value is known. Then the simulator must process

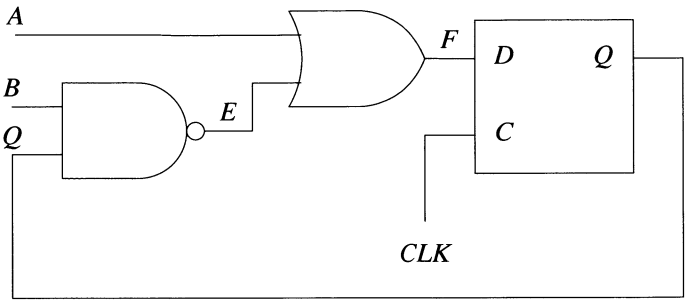


Figure 3.7 Synchronous circuit

only binary values, which are stored in the variables *A,B,Q,E,* and *D*. The following is the assembly code model of the circuit:

```
LDA   B
AND   Q
INV
STA   E
OR    A
STA   F
STA   Q
```

Note that a compiled simulator evaluates all the elements in the circuit for every input vector.

If the initial value of Q is unknown, the simulator must process the values 0, 1, and u. These values are coded with 2-bit vectors as follows:

$$0 - 00$$
$$1 - 11$$
$$u - 01$$

We can easily observe that an AND (OR) operation between 2-bit vectors is correctly done by ANDing (ORing) the individual bits. However a NOT operation cannot be done only by complementing the bits, because the complement of u would result in the illegal code 10. The solution for NOT is to swap the two bits after complementation.

Let us consider the evaluation of an AND gate with inputs A and B and output C by an operation $C = A.B$, where "." represents the AND instruction of the host computer. If we restrict ourselves to 2-valued logic, we need only one bit to represent the value of a signal. In *parallel-pattern evaluation* [Barzilai *et al.* 1987], we use a W-bit memory location of the host computer to store the values of the same signal in W different vectors. If the "." instruction works on W-bit operands, then $C = A.B$ simultaneously computes the values of C in W vectors. Of course, this is valid only in combinational circuits, where the order in which vectors are applied is not relevant.

This method speeds up 2-valued compiled simulation by a factor of W (typically, $W = 32$). For three logic values, only $W/2$ vectors can be simulated concurrently.

Compiled simulation can also be used for asynchronous circuits, using the model shown in Figure 3.8, which assumes that delays are present only on the feedback lines. In response to an input vector x, the circuit may go through a series of state transitions, represented by changes of the state variables y. We assume that an input vector is applied only when the circuit is stable, i.e., $y=Y$. Feedback lines, which have level 0, must be identified before the code model for the combinational circuit C is generated. Figure 3.9 outlines the general procedure for simulating an asynchronous circuit. The execution of the model computes the values of z and Y based on x and y.

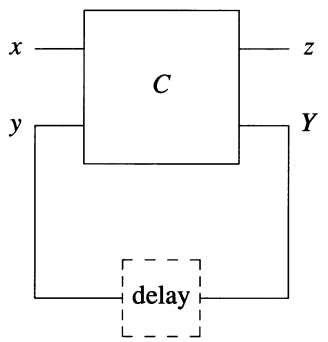


Figure 3.8 Asynchronous circuit model

This type of simulation is not accurate for asynchronous circuits whose operation is based on certain delay values. For example, the circuit of Figure 3.10(a) could be used as a pulse generator: when A has a 0→1 transition, the delay of the inverter B creates an interval during which both inputs of C have value 1, thus causing a 0→1→0 pulse on C. Without careful modeling, this pulse cannot be predicted by a compiled simulator, which deals only with the static behavior of the circuit (statically, C is always 0). To take into account the delay of B, which is essential for the intended operation of the circuit, B should be treated as a feedback line, as shown in Figure 3.10(b). With this model, the circuit can be correctly simulated. In general, however, deriving such a "correct" model cannot be done automatically and requires input from the user.

Even when the structure of the circuit allows the identification of the feedback loops, different models can be derived from the same circuit (see Figure 3.11). Because the different models have different assumptions about the location of the delays, they may respond differently to the same stimuli. For example, consider the simulation of the latch given in Figure 3.11(a) for the vector 00 followed by 11. Using the model with Q as the feedback line, we determine that QN changes from 1 to 0, while Q=1 in both vectors. However, using the model with QN as the feedback line, we would compute that Q changes from 1 to 0, while QN=1. The reason for the different results is that

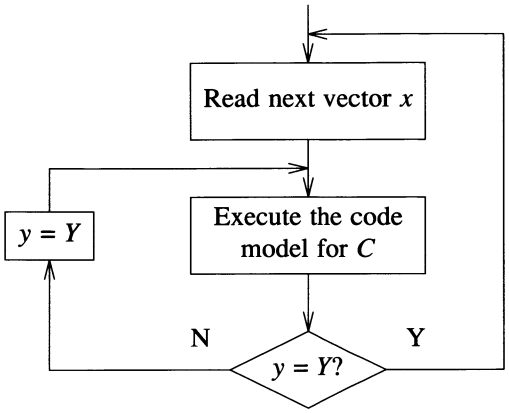


Figure 3.9 Asynchronous circuit simulation with compiled-code model

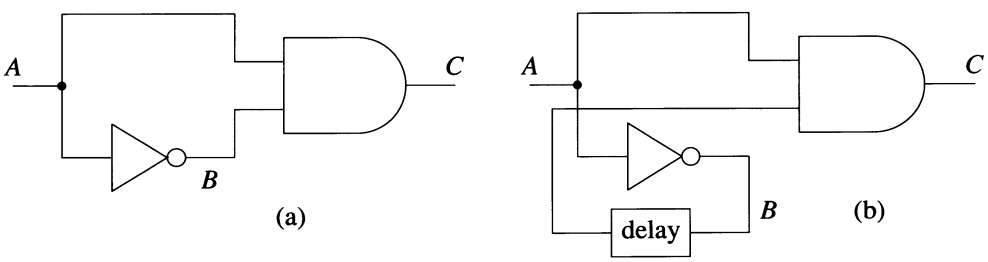


Figure 3.10 (a) Circuit used as pulse generator (b) Correct model for compiled simulation

the two vectors cause a critical race, whose result depends on the actual delays in the circuit. This example shows that a compiled simulator following the procedure outlined in Figure 3.9, cannot deal with races and hazards which often affect the operation of an asynchronous circuit. Techniques for detecting hazards will be discussed in Section 3.9.

3.6 Event-Driven Simulation

An event-driven simulator uses a structural model of a circuit to propagate events. The changes in the values of the primary inputs are defined in the stimulus file. Events on the other lines are produced by the evaluations of the activated elements.

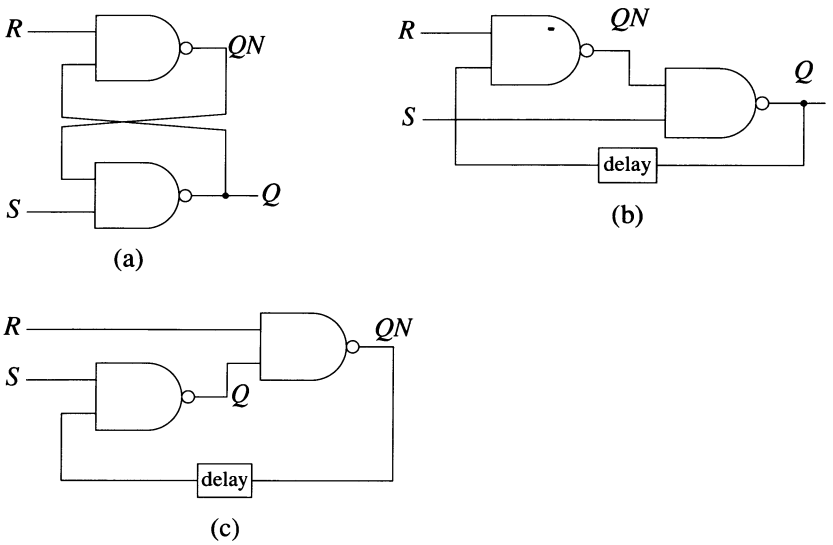


Figure 3.11 (a) Latch (b)&(c) Possible models for compiled simulation

An event occurs at a certain (simulated) time. The simulation *time-flow mechanism* manipulates the events such that they will occur in a correct temporal order. The applied stimuli are represented by sequences of events whose times are predefined. The events scheduled to occur in the future (relative to the current simulated time) are said to be *pending* and are maintained in a data structure called an *event list.*

Figure 3.12 shows the main (conceptual) flow of event-directed simulation. The simulation time is advanced to the next time for which events are pending; this becomes the current simulation time. Next, the simulator retrieves from the event list the events scheduled to occur at the current time and updates the values of the active signals. The fanout list of the active signals is then followed to determine the activated elements; this process parallels the propagation of changes in the real circuit. The evaluation of the activated elements may result in new events. These are scheduled to occur in the future according to the delays associated with the operation of the elements. The simulator inserts the newly generated events in the event list. The simulation continues as long as there is logic activity in the circuit; that is, until the event list becomes empty.

The evaluation of an element M modeled by a nonprocedural RTL may generate a *state event* that denotes a change in the value of an internal state variable of M. (In procedural RTL, changes of state variables occur immediately, without processing via the event list.) When such a state event occurs, it activates only the element M that has generated it; that is, it causes M to be reevaluated.

For simplicity, in the above description we have assumed that all the events defining the applied stimuli were inserted in the event list before simulation. In fact, the

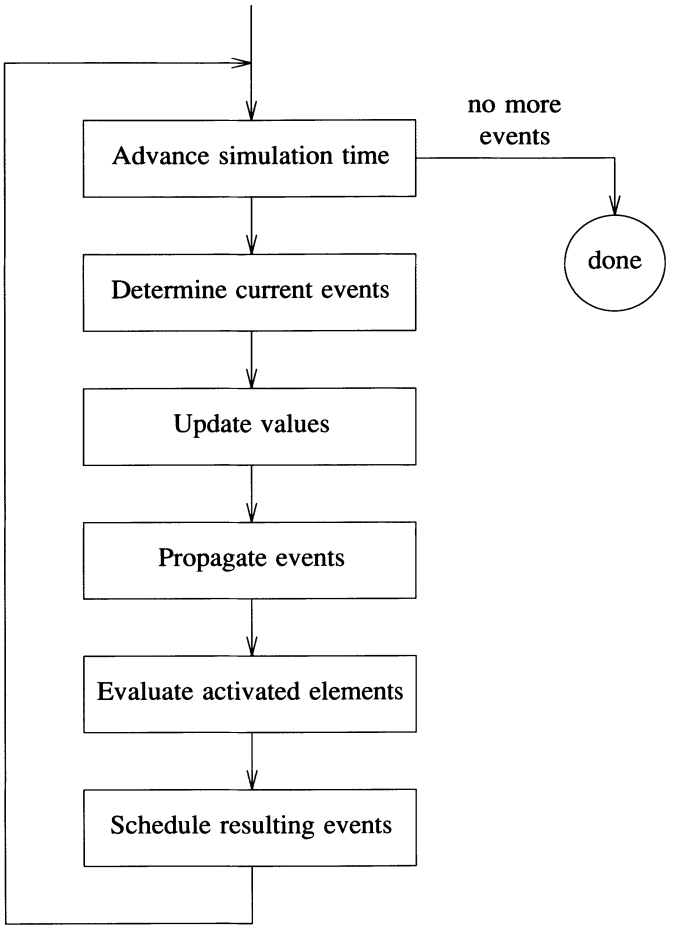


Figure 3.12 Main flow of event-driven simulation

simulator has periodically to read the stimulus file and to merge the events on the primary inputs with the events internally generated.

In addition to the events that convey updates in signal values, an event-driven simulator may also process *control events*, which provide a convenient means to initiate different activities at certain times. The following are some typical actions requested by control events:

- Display values of certain signals.
- Check expected values of certain signals (and, possibly, stop the simulation if a mismatch is detected).

- Stop the simulation.

3.7 Delay Models

Many different variants of the general flow shown in Figure 3.12 exist. The differences among them arise mainly from different *delay models* associated with the behavior of the components in the model. Delay modeling is a key element controlling the trade-off between the accuracy and the complexity of the simulation algorithm.

3.7.1 Delay Modeling for Gates

Every gate introduces a delay to the signals propagating through it. In modeling the behavior of a gate, we separate its function and its timing as indicated in Figure 3.13. Thus in simulation an activated element is first evaluated, then the delay computation is performed.

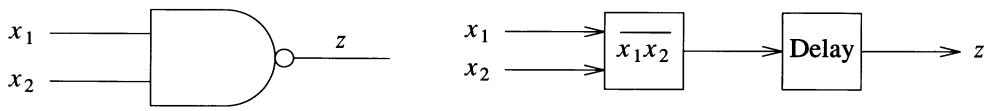


Figure 3.13 Separation between function and delay in modeling a gate

Transport Delays

The basic delay model is that of a *transport delay,* which specifies the interval d separating an output change from the input change(s) which caused it.

To simplify the simulation algorithm, delay values used in simulation are usually integers. Typically they are multiples of some common unit. For example, if we are dealing with gate delays of 15, 20, and 30 ns, for simulation we can scale them respectively to 3, 4, and 6 units, where a unit of delay represents the greatest common divisor (5 ns) of the individual delays. (Then the times of the changes at the primary inputs should be similarly scaled.) If all transport delays in a circuit are considered equal, then we can scale them to 1 unit; this model is called a *unit-delay model.*

The answers to the following two questions determine the nature of the delay computation.

- Does the delay depend on the direction of the resulting output transition?
- Are delays precisely known?

For some devices the times required for the output signal to rise (0 to 1 transition*) and to fall (1 to 0) are greatly different. For some MOS devices these delays may

* We assume a *positive logic* convention, where the logic level 1 represents the highest voltage level.

differ by a ratio of 3 to 1. To reflect this phenomenon in simulation, we associate different *rise and fall delays*, d_r and d_f, with every gate. The delay computation selects the appropriate value based on the event generated for the gate output. If the gate delays are not a function of the direction of the output change, then we can use a *transition-independent delay model.* Figures 3.14(a) and (b) illustrate the differences between these two models. Note that the result of having different rise and fall delays is to change the width of the pulse propagating through the gate.

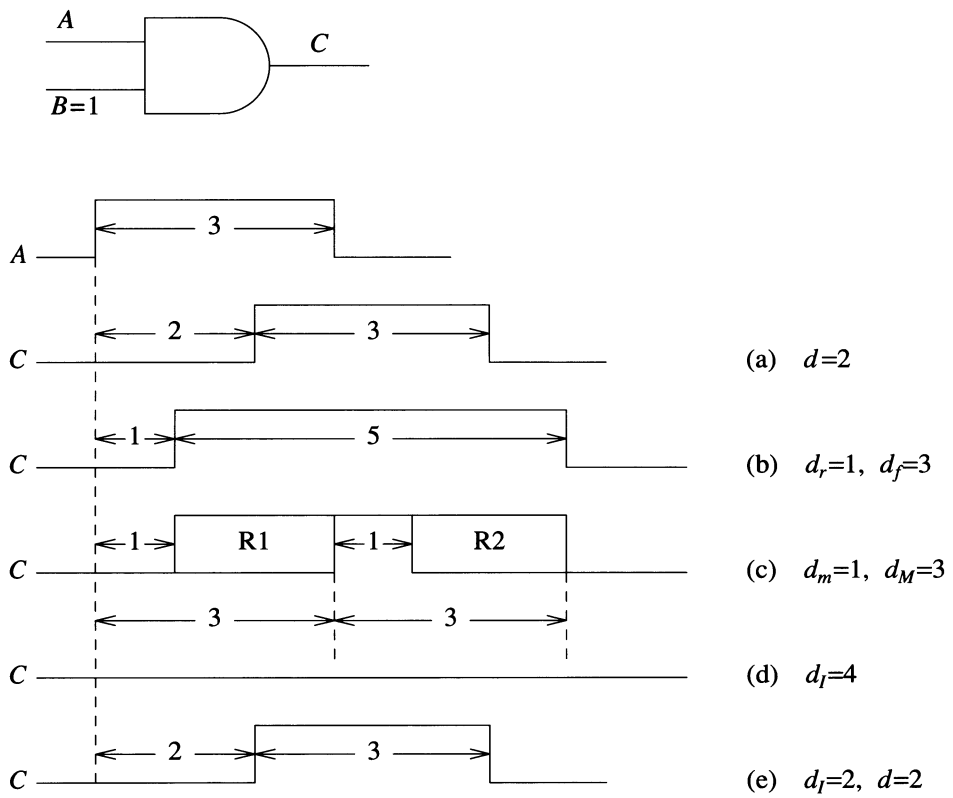


Figure 3.14 Delay models (a) Nominal transition-independent transport delay (b) Rise and fall delays (c) Ambiguous delay (d) Inertial delay (pulse suppression) (e) Inertial delay

Often the exact transport delay of a gate is not known. For example, the delay of a certain type of NAND gate may be specified by its manufacturer as varying from 5 ns to 10 ns. To reflect this uncertainty in simulation, we associate an *ambiguity interval*, defined by the minimum (d_m) and maximum (d_M) delays, with every gate. This model, referred to as an *ambiguous delay model*, results in intervals (R1 and R2 in Figure 3.14(c)) during which the value of a signal is not precisely known. Under the assumption that the gate delays are known, we have a *nominal delay model.* The rise

and fall delay model and the ambiguous delay model can be combined such that we have different ambiguity intervals for the rise (d_{rm}, d_{rM}) and the fall (d_{fm}, d_{fM}) delays [Chappel and Yau 1971].

Inertial Delays

All circuits require energy to switch states. The energy in a signal is a function of its amplitude and duration. If its duration is too short, the signal will not force the device to switch. The minimum duration of an input change necessary for the gate output to switch states is called the *input inertial delay* of the gate, denoted by d_I. An input pulse whose duration is less than d_I is said to be a *spike,* which is *filtered* (or *suppressed*) by the gate (Figure 3.14(d)). If the pulse width is at least d_I, then its propagation through the gate is determined by the transport delay(s) of the gate. If the gate has a transition-independent nominal transport delay d, then the two delays must satisfy the relation $d_I \leq d$ (Problem 3.6). Figure 3.14(e) shows a case with $d_I = d$.

A slightly different way of modeling inertial delays is to associate them with the gate outputs. This *output inertial delay* model specifies that the gate output cannot generate a pulse whose duration is less than d_I. An output pulse may be caused by an input pulse (and hence we get the same results as for the input inertial delay model), but it may also be caused by "close" input transitions, as shown in Figure 3.15(a). Another case in which the two inertial delay models would differ is illustrated in Figure 3.15(b). With the input inertial delay model, the input pulses are considered separately and they cannot switch the gate output. However, under the output inertial delay model, the combined effect of the input pulses forces the output to switch.

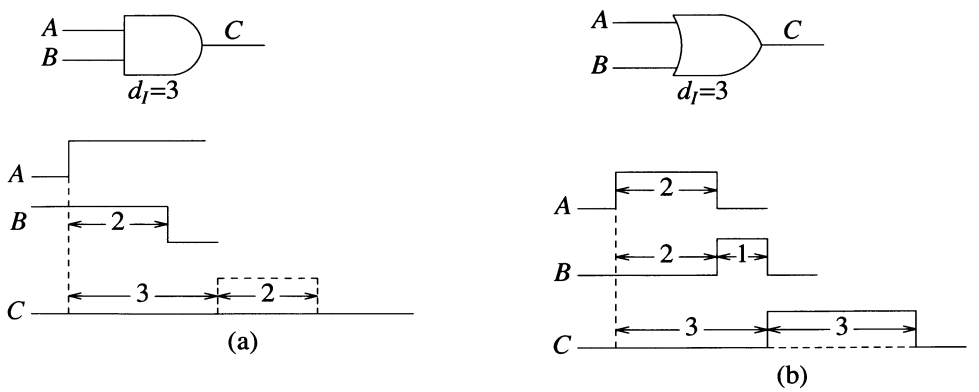
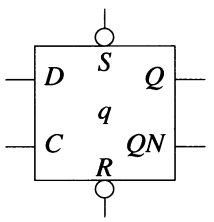


Figure 3.15 Output inertial delay

3.7.2 Delay Modeling for Functional Elements

Both the logic function and the timing characteristics of functional elements are more complex than those of the gates. Consider, for example, the behavior of an edge-triggered *D* F/F with asynchronous set *(S)* and reset *(R)*, described by the table given in Figure 3.16. The symbol ↑ denotes a 0-to-1 transition. The notation $d_{I/O}$

represents the delay in the response of the output O to the change of the input I. The superscript (r or f) distinguishes between rise and fall delays, where necessary. For example, the third row says that if the F/F is in initial state $q = 1$ and S and R are inactive (1), a 0-to-1 transition of the clock C causes the output Q to change to the value of D (0) after a delay $d^f_{C/Q} = 8$. Similarly, the output QN changes to 1 after a delay $d^r_{C/QN} = 6$. The last row specifies that the illegal input condition $SR = 00$ causes both outputs to be set to u.



q	S	R	C	D	Q	QN	*Delays*
0	0	1	x	x	1	0	$d_{S/Q} = 4$ $d_{S/QN} = 3$
1	1	0	x	x	0	1	$d_{R/Q} = 3$ $d_{R/QN} = 4$
1	1	1	↑	0	0	1	$d^f_{C/Q} = 8$ $d^r_{C/QN} = 6$
0	1	1	↑	1	1	0	$d^r_{C/Q} = 6$ $d^f_{C/QN} = 8$
x	0	0	x	x	u	u	

Figure 3.16 I/O delays for a D F/F

Similar to the input inertial delay model for gates, the specifications of the F/F may include the *minimum pulse widths* for C, S, and R required to change the state of the F/F.

Additional timing specifications deal with requirements for avoiding race conditions between C and D. The *setup time* and the *hold time* are the minimum intervals preceding and following an active C transition during which D should remain stable in order to have a correct (i.e., predictable) F/F operation. Some simulation systems can detect improper operation with respect to setup and hold conditions [Evans 1978, Tokoro *et al.* 1978]. Rather than having these checks done by the simulator, a better approach is to include them in the functional models developed for F/Fs.

3.7.3 Delay Modeling in RTLs

Different ways of specifying delays in RTL models have been discussed in Chapter 2. In general, RTLs offer a more abstract view of the system, and hence the delay modeling is less detailed. Many RTLs use a cycle timing model. If delays are assigned to individual operations, they have the meaning of nominal transport delays.

3.7.4 Other Aspects of Delay Modeling

In high-speed circuits, the delays introduced by the propagation of signals along wires become as significant as the delays of components. Since these delays depend on wire lengths, they are known only after the routing of the circuit is performed (at least tentatively). Many design automation systems are able to extract information automatically from the layout data and to update the simulation data base. The delay introduced by a signal line i with only one fanout can be assimilated in the delay of

the gate that generates i. However, if i has several fanouts, each one of its fanout branches may involve a different propagation delay. These are usually modeled by inserting *delay elements* in the appropriate places (see Figure 3.17). A delay element realizes an identity logic function, and its purpose is only to delay the signal propagation.

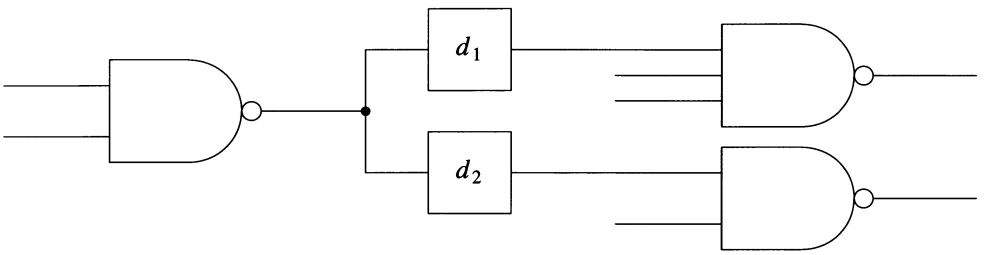


Figure 3.17 Wire delays modeled by delay elements

Another factor that affects delay modeling is the *loading* of a signal, where the apparent delay of a gate grows with the fanout count of its output signal.

3.8 Element Evaluation

The evaluation of a combinational element is the process of computing its output values given its current input values. The evaluation of a sequential element is also based on its current state and computes its next state as well. Evaluation techniques depend on many interrelated factors, such as the system of logic values used in simulation, the way values are stored, the type of the elements, and the way they are modeled.

As the evaluation of a combinational element G must analyze the input values of G, a first question is how are these values made available to the evaluation routine. First let us assume that signal values are stored in a table parallel to the signal tables (see Figure 2.21). Then finding the input values of G is an indirect process, which first determines the inputs by following the fanin list of G and then accesses their values. A second way is to maintain the input values of G in a contiguous area of memory associated with G (see Figure 3.18). Although this scheme may appear wasteful, since it replicates the value of a signal with k fanouts in the value area of every one of its fanout elements, it presents several advantages:

- The evaluation routines are faster, because they can directly access the needed values.
- Values can be "packed" together, which allows more efficient evaluation techniques.

- Since the evaluation routines no longer need to determine the inputs of the evaluated elements, storage can be saved by not loading the fanin data in memory during simulation.
- The separation between the value of a signal and the values of its fanout branches is useful in fault simulation, as these values can be different in the presence of faults (in logic simulation they are always the same).

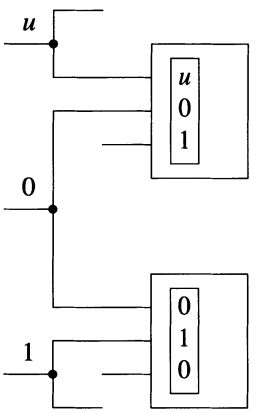


Figure 3.18 Input values maintained per element

In the following, we assume that input values are maintained per element (as illustrated in Figure 3.18) and that values of the state variables of sequential elements are similarly stored.

Truth Tables

Let n be the number of inputs and state variables of an element. Assuming only binary values, a truth table of the element has 2^n entries, which are stored as an array V, whose elements are vectors of output and state variable values. For evaluation, the n values of the inputs and state variables are packed in the same word, and the evaluation uses the value of this word — say, i — as an index to retrieve the corresponding output and next state stored in $V[i]$.

Truth tables can also be generalized for multivalued logic. Let k be the number of logic values, and let q be the number of bits needed to code the k values. That is, q is the smallest integer such that $k \leq 2^q$. Then the size of the array needed to store a truth table of a function of n k-valued variables is 2^{qn}. For example, a truth table for an element that depends on five binary variables requires $2^5 = 32$ entries. For 3-valued logic, the truth table has $3^5 = 243$ entries, and the size of the array needed to store it is $2^{10} = 1024$.

Evaluation techniques based on truth tables are fast, but because of the exponential increase in the amount of memory they require, they are limited to elements that depend on a small number of variables.

A trade-off between speed and storage can be achieved by using a one-bit flag in the value area of an element to indicate whether any variable has a nonbinary value. If all values are binary, then the evaluation is done by accessing the truth table; otherwise a special routine is used. This technique requires less memory, since truth tables are now defined only for binary values. The loss in speed for nonbinary values is not significant, because, in general, most of the evaluations done during a simulation run involve only binary values.

Zoom Tables

An evaluation technique based on truth tables must first use the type of the evaluated element to determine the truth table to access. Thus checking the type and accessing the truth table are separate steps. These two consecutive steps can be combined into a single step as follows. Let t be the number of types and let S be the size of the largest truth table. We build a *zoom table* of size tS, in which we store the t individual truth tables, starting at locations 0, S, ..., $(t-1)S$. To evaluate an element, we pack its type code (in the range 0 to $t-1$) in the same word with its values, such that we can use the value of this word as an index into the zoom table (see Figure 3.19).

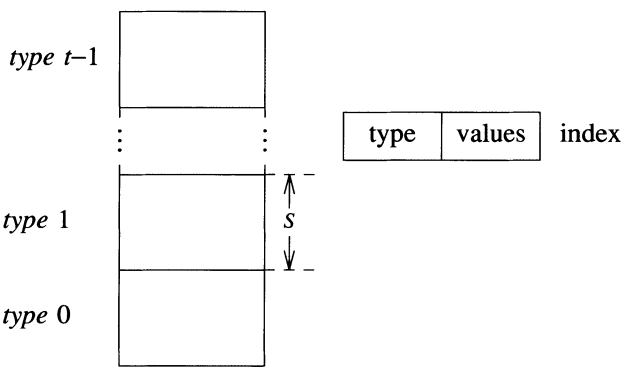


Figure 3.19 Zoom table structure

This type of zoom table is an instance of a general speed-up technique that reduces a sequence of k decisions to a single step. Suppose that every decision step i is based on a variable x_i which can take one of a possible set of m_i values. If all x_i's are known beforehand, we can combine them into one cross-product variable $x_1 \times x_2 \times ... \times x_k$ which can take one of the possible $m_1 m_2 ... m_k$ values. In this way the k variables are examined simultaneously and the decision sequence is reduced to one step. More complex zoom tables used for evaluation are described in [Ulrich *et al.* 1972].

Input Scanning

The set of primitive elements used in most simulation systems includes the basic gates — AND, OR, NAND, and NOR. These gates can be characterized by two parameters, the *controlling value* c and the *inversion* i. The value of an input is said to be controlling if it determines the value of the gate output regardless of the values of the other inputs; then the output value is $c \oplus i$. Figure 3.20 shows the general form of the primitive cubes of any gate with three inputs.

$c\ x\ x$	$c \oplus i$
$x\ c\ x$	$c \oplus i$
$x\ x\ c$	$c \oplus i$
$\bar{c}\ \bar{c}\ \bar{c}$	$\bar{c} \oplus i$

	c	i
AND	0	0
OR	1	0
NAND	0	1
NOR	1	1

Figure 3.20 Primitive cubes for a gate with controlling value c and inversion i

Figure 3.21 outlines a typical gate evaluation routine for 3-valued logic, based on scanning the input values. Note that the scanning is terminated when a controlling value is encountered.

```
evaluate (G, c, i)
begin
    u_values = FALSE
    for every input value v of G
        begin
            if v = c then return c⊕i
            if v = u then u_values = TRUE
        end
    if u_values return u
    return c̄⊕i
end
```

Figure 3.21 Gate evaluation by scanning input values

Input Counting

Examining Figure 3.21, we can observe that to evaluate a gate using 3-valued logic, it is sufficient to know whether the gate has any input with c value, and, if not, whether it has any input with u value. This suggests that, instead of storing the input values for every gate, we can maintain a compressed representation of the values, in the form of two counters — *c_count* and *u_count* — which store, respectively, the number of inputs with c and u values [Schuler 1972]. The step of updating the input values (done

before evaluation) is now replaced by updating of the two counters. For example, a 1→0 change at an input of an AND gate causes the *c_count* to be incremented, while a 0→*u* change results in decrementing the *c_count* and incrementing the *u_count*. The evaluation of a gate involves a simple check of the counters (Figure 3.22). This technique is faster than the input scanning method and is independent of the number of inputs.

```
evaluate (G, c, i)
begin
    if c_count > 0 then return c⊕i
    if u_count > 0 then return u
    return c̄⊕i
end
```

Figure 3.22 Gate evaluation based on input counting

3.9 Hazard Detection

Static Hazards

In the circuit of Figure 3.23, assume that $Q = 1$ and A changes from 0 to 1, while B changes from 1 to 0. If these two changes are such that there exists a short interval during which $A=B=1$, then Z may have a spurious 1→0→1 pulse, which may reset the latch. The possible occurrence of a transient pulse on a signal line whose static value does not change is called a *static hazard.*

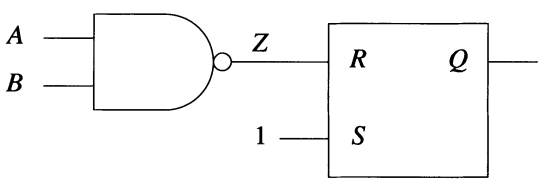


Figure 3.23

To detect hazards, a simulator must analyze the transient behavior of signals. Let $S(t)$ and $S(t+1)$ be the values of a signal S at two consecutive time units. If these values are different, the exact time when S changes in the real circuit is uncertain. To reflect this uncertainty in simulation, we will introduce a "pseudo time unit" t' between t and $t+1$ during which the value of S is unknown, i.e., $S(t') = u$ [Yoeli and Rinon 1964, Eichelberger 1965]. This is consistent with the meaning of u as one of the values in the set {0,1}, because during the transition period the value of S can be independently

observed by each of its fanouts as either 0 or 1. Then the sequence $S(t)\ S(t')\ S(t+1) = 0u1$ represents one of the sequences in the set {001,011}. These two sequences can be interpreted, respectively, as a "slow" and a "fast" 0→1 transition. Returning now to the example of Figure 3.23, the corresponding sequences are $A = 0u1$ and $B = 1u0$. The resulting sequence for Z, computed by bitwise NAND operations, is $1u1$. This result shows that a possible output sequence is 101, and thus it detects the unintended pulse.

The general procedure for detecting static hazards in a combinational circuit C works as follows. Assume that C has been simulated for time t and now it is simulated for time $t+1$. Let E be the set of inputs changing between t and $t+1$.

Procedure 3.1

1. Set every input in E to value u and simulate C. (The other inputs keep their values.) Let $Z(t')$ denote the value obtained for a signal Z.
2. Set every input in E to its value at $t+1$ and simulate C. □

Theorem 3.1: In a combinational circuit, a static hazard exists on line Z between the times t and $t+1$ if and only if the sequence of values $Z(t)\ Z(t')\ Z(t+1)$ (computed by Procedure 3.1) is 1u1 or 0u0.

Proof: Clearly, a sequence $0u0$ or $1u1$ is a sufficient condition for the existence of a static hazard. To prove that it also a necessary one, we rely on the following two facts, easily derived from the 3-valued truth tables of the basic gates.

1. If one or more gate inputs change from binary values to u, then the gate output either remains unchanged or changes from a binary value to u.
2. If one or more gate inputs change from u to binary values, then the gate output either remains unchanged or changes from u to a binary value.

Hence any gate whose value in t' is not u has the same binary value in t, t', and $t+1$, and therefore cannot have a static hazard. □

Because Procedure 3.1 ignores the delays in the circuit, it performs a worst-case analysis whose results are independent of the delay model. Hence, we say that it uses an *arbitrary delay model.*

The next example illustrates how different delay models (0-delay, unit-delay, and arbitrary delays) affect hazard detection.

Example 3.1: Consider again the circuit of Figure 3.10(a) and the input sequence A=010. For the 0-delay model we obtain B=101 and C=000. Thus no hazard is predicted. This is because a 0-delay model deals only with the static behavior of a circuit and ignores its dynamic behavior.

For the unit-delay model, the signal sequences are B=1101 and C=0010. Thus a pulse is predicted in response to the 0→1 transition of A.

For the arbitrary delay model, the signal sequences (obtained with Procedure 3.1) are A=$0u1u0$, B=$1u0u1$, and C=$0u0u0$. Thus a hazard is predicted for both the rise and fall input transitions. This is an overly pessimistic result, since in general either the path through the inverter or the direct path from A to C has the most delay, and then

only one of the input transitions should cause a hazard. However, under an arbitrary delay model it is not known which path has the most delay, and hazards are predicted for both transitions. □

The analysis based on sequences of consecutive values underlies the hazard detection mechanism of most simulators [Hayes 1986]. Many simulators use multivalued logic systems that represent (implicitly or explicitly) different sets of sequences. Figure 3.24 shows such a set of values and their corresponding sequences. The result of a logic operation between these values (Figure 3.25) can be obtained by performing the same operation bitwise between the corresponding sequences.

Value	Sequence(s)	Meaning
0	000	Static 0
1	111	Static 1
0/1, *R*	{001,011} = 0*u*1	Rise (0 to 1) transition
1/0, *F*	{110,100} = 1*u*0	Fall (1 to 0) transition
0*	{000,010} = 0*u*0	Static 0-hazard
1*	{111,101} = 1*u*1	Static 1-hazard

Figure 3.24 6-valued logic for static hazard analysis

AND	0	1	*R*	*F*	0*	1*
0	0	0	0	0	0	0
1	0	1	*R*	*F*	0*	1*
R	0	*R*	*R*	0*	0*	*R*
F	0	*F*	0*	*F*	0*	*F*
0*	0	0*	0*	0*	0*	0*
1*	0	1*	*R*	*F*	0*	1*

Figure 3.25 AND truth table for 6-valued logic

Some simulators combine the values 0* and 1* into a single value that denotes a hazard. Sometimes this value is also combined with the unknown (*u*) value [Lewis 1972].

Dynamic Hazards

A *dynamic hazard* is the possible occurrence of a transient pulse during a 0→1 or 1→0 signal transition. The analysis for detecting dynamic hazards requires 4-bit sequences. For example, the sequence 0101 describes a 1-pulse during a 0→1 transition. The "clean" 0→1 transition corresponds to the set {0001,0011,0111}. Figure 3.26 shows an 8-valued logic system used for static and dynamic hazard

analysis [Breuer and Harrison 1974]. We can observe that it includes the six values of Figure 3.24, to which it adds the values R^* and F^* to represent dynamic hazards.

Value	Sequence(s)	Meaning
0	0000	Static 0
1	1111	Static 1
0/1, R	{0001,0011,0111}	Rise transition
1/0, F	{1110,1100,1000}	Fall transition
0*	{0000,0100,0010,0110}	Static 0-hazard
1*	{1111,1011,1101,1001}	Static 1-hazard
R^*	{0001,0011,0111,0101}	Dynamic 1-hazard
F^*	{1110,1100,1000,1010}	Dynamic 0-hazard

Figure 3.26 8-valued logic for static and dynamic hazard analysis

A Boolean operation B between p and q, where p and q are among the eight values shown in Figure 3.26, is considered an operation between the sets of sequences corresponding to p and q and is defined as the union set of the results of all possible B operations between the sequences in the two sets. For example:

$$\text{AND}(R,1^*) = \text{AND}(\{0001,0011,0111\},\{1111,1011,1101,1001\}) =$$
$$= \{0001,0011,0111,0101\} = R^*$$

Hazard Detection in Asynchronous Circuits

We will analyze hazards in an asynchronous circuit of the form shown in Figure 3.8 using an arbitrary delay model. Assume that all values at time t are known (and stable); now we apply a new vector x at time t+1. Let E be the set of primary inputs changing between t and t+1.

Procedure 3.2

1. Set every input in E to value u and simulate C. For every feedback line Y_i that changes to u, set the corresponding state variable y_i to u and resimulate C. Repeat until no more Y_i changes to u.
2. Set every input in E to its value at t+1 and simulate C. For every Y_i that changes to a binary value b_i, set the corresponding y_i to b_i and resimulate. Repeat until no more Y_i changes to a binary value. □

Theorem 3.2: If the final value of Y_i computed by Procedure 3.2 is binary, then the feedback line Y_i stabilizes in this state (under the given input transition), regardless of the delays in the circuit.

Proof: Exercise. □

The following is an important consequence of Theorem 3.2.

Corollary 3.1: If the final value of Y_i computed by Procedure 3.2 is u, then the given input transition may cause a critical race or an oscillation. □

Procedure 3.2 does not require the feedback lines to be identified. It can work with an event-driven simulation mechanism by propagating first the changes that occur between t and t' until values stabilize, then the changes occurring between t' and t+1. In each simulation pass the values are guaranteed to stabilize (see Problem 3.17). Because of the arbitrary delay model, the order in which elements are evaluated is not important. Note that while executing step 1, any element whose value is currently u need not be reevaluated. Also, in step 2, any element whose current value is binary need not be reevaluated.

Example 3.2: Consider again the latch given in Figure 3.11(a). Assume that at time t, R=S=0 and Q=QN=1, and that at time t+1 both R and S change to 1. Following Procedure 3.2, in step 1 we set R=S=u, and as a result we obtain Q=QN=u. In step 2 we set R=S=1, and Q and QN remain stable at u. This shows that under an arbitrary delay model the operation of the circuit is unpredictable. Depending on the actual delays in the circuit, the final state may be Q=0, QN=1 or Q=1, QN=0 or the circuit may oscillate. □

An important assumption of Procedure 3.2 is that the circuit is operated in *fundamental mode*, that is, stimuli are applied only when the values in the circuit are stable. This assumption precludes the simulation of real-time inputs (see Problem 3.18).

3.10 Gate-Level Event-Driven Simulation

3.10.1 Transition-Independent Nominal Transport Delays

Now we will particularize the general simulation algorithm given in Figure 3.12 for gate-level simulation using a transition-independent nominal transport delay model.

First we consider that the event list is organized as shown in Figure 3.27, where events scheduled to occur at the same time in the future are stored in the same list. The time order is maintained by chaining the list headers in appropriate order, i.e., $t_p < t_q < t_r$. An entry (i,v_i') in the list associated with t_q indicates that at time t_q the value of line i is scheduled to be set to v_i'.

We assume that values and delays are kept in tables similar to those depicted in Figure 2.21; *v(i)* denotes the current value of gate i and *d(i)* denotes the nominal delay of i.

Figure 3.28 shows the general structure of the event-driven simulation algorithms discussed in this section. Algorithm 3.1, given in Figure 3.29, provides a first implementation of the line "process entries for time t" of Figure 3.28. Algorithm 3.1 employs a *two-pass strategy*. In the first pass it retrieves the entries from the event list associated with the current time t and determines the activated gates. In the second pass it evaluates the activated gates and schedules their computed values. This strategy assures that gates activated by more than one event are evaluated only once.

Note that in Algorithm 3.1, an entry (i, v_i') in the event list does not always represent a change in the value of i. The problem of determining when a new output value of an activated gate is indeed an event is illustrated in Figure 3.30. In response to the

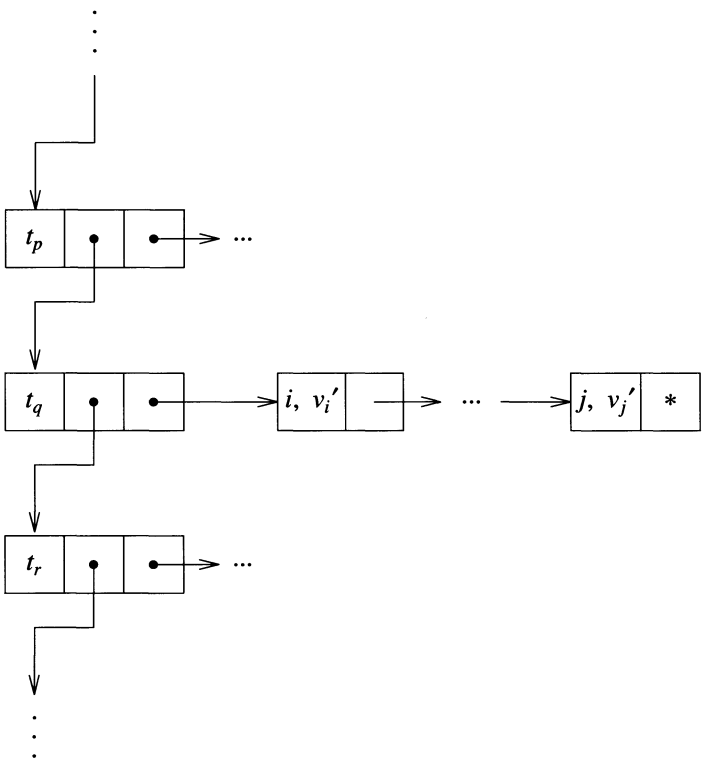


Figure 3.27 Event list implemented as a linked list structure

```
while (event list not empty)
   begin
      t = next time in list
      process entries for time t
   end
```

Figure 3.28 General structure for event-driven simulation

event (a,1) at time 0, (z,1) is scheduled for time 8. Since now the gate is in a transition state (i.e., the scheduled output event has not yet occurred), there is a temporary logic inconsistency between the current output value (0) and the input values (both 1). Thus when the next input event (b,0) occurs at time 2, Algorithm 3.1 cannot determine whether setting $z = 0$ at time 10 will represent a change. The strategy employed is to enter in the event list all the new values of the activated gates

```
Activated = ∅   /* set of activated gates */
for every entry (i,v_i') pending at the current time t
   if v_i' ≠ v(i) then
      begin   /* it is indeed an event */
         v(i) = v_i'   /* update value */
         for every j on the fanout list of i
            begin
               update input values of j
               add j to Activated
            end
      end
for every j ∈ Activated
   begin
      v_j' = evaluate (j)
      schedule (j,v_j') for time t+d(j)
   end
```

Figure 3.29 Algorithm 3.1

and to do the check for activity when the entries are retrieved. For the example in Figure 3.30, (z,0) will be scheduled for time 12 as a result of (a,0) at time 4, but z will already be set to 0 at time 10.

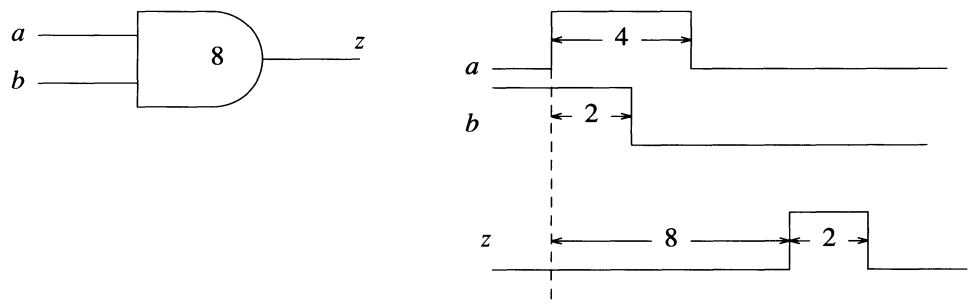


Figure 3.30 Example of activation of an already scheduled gate

Algorithm 3.2, given in Figure 3.31, is guaranteed to schedule only true events. This is done by comparing the new value v_j', determined by evaluation, with the *last scheduled value* of j, denoted by *lsv(j)*. Algorithm 3.2 does fewer schedule operations than Algorithm 3.1, but requires more memory and more bookkeeping to maintain the last scheduled values. To determine which one is more efficient, we have to estimate

how many unnecessary entries in the event list are made by Algorithm 3.1. Our analysis focuses on the flow of events in and out of the event list.

```
Activated = ∅
for every event (i,v_i') pending at the current time t
   begin
      v(i) = v_i'
      for every j on the fanout list of i
         begin
            update input values of j
            add j to Activated
         end
   end
for every j ∈ Activated
   begin
      v_j' = evaluate (j)
      if v_j' ≠ lsv(j) then
         begin
            schedule (j,v_j') for time t+d(j)
            lsv(j) = v_j'
         end
   end
```

Figure 3.31 Algorithm 3.2 — guaranteed to schedule only events

Let N be the total number of events occurring during a simulation run. If we denote by f the average fanout count of a signal, the number of gate activations is Nf. A gate that has $k > 1$ simultaneously active inputs is activated k times, but it is evaluated only once. (The entries in the set *Activated* are unique.) Hence the number of gate evaluations is bounded by Nf. Because most gates are evaluated as a result of only one input change, we can approximate the number of evaluations by Nf. Let q be the fraction of gate evaluations generating output events. Then

$$N_1 = qNf \tag{3.1}$$

represents the total number of events scheduled during simulation. To these N_1 events, we should add N_2 events corresponding to the changes of the primary inputs entered in the event list before simulation. The equation

$$N_1 + N_2 = N \tag{3.2}$$

states that all the events entering the event list are eventually retrieved. Since usually $N_2 << N_1$, we obtain

$$qNf \cong N \tag{3.3}$$

Hence, $qf \cong 1$. This result shows that on the average, only one out of f evaluated gates generates an output event. Thus Algorithm 3.1 schedules and retrieves f times more

items than Algorithm 3.2. Typically the average fanout count f is in the range 1.5 to 3. As the event list is a dynamic data structure, these operations involve some form of free-space management. Also scheduling requires finding the proper time slot where an item should be entered. We can conclude that the cost of the unnecessary event list operations done by Algorithm 3.1 is greater than that involved in maintaining the *lsv* values by Algorithm 3.2.

One-Pass Versus Two-Pass Strategy

The reason the *two-pass strategy* performs the evaluations only after all the concurrent events have been retrieved is to avoid repeated evaluations of gates that have multiple input changes. Experience shows, however, that most gates are evaluated as a result of only one input change. Hence a *one-pass strategy* [Ulrich 1969], which evaluates a gate as soon as it is activated, would be more efficient, since it avoids the overhead of constructing the *Activated* set. Algorithm 3.3, shown in Figure 3.32, implements the one-pass strategy.

```
for every event (i,v_i') pending at the current time t
   begin
      v(i) = v_i'
      for every j on the fanout list of i
         begin
            update input values of j
            v_j' = evaluate (j)
            if v_j' ≠ lsv(j) then
               begin
                  schedule (j,v_j') for time t+d(j)
                  lsv(j) = v_j'
               end
         end
   end
```

Figure 3.32 Algorithm 3.3 — one-pass strategy

Figure 3.33 illustrates the problem associated with Algorithm 3.3. Inputs a and b are scheduled to change at the same time. If the events are retrieved in the sequence (b,0), (a,1), then z is never scheduled. But if (a,1) is processed first, this results in (z,1) scheduled for time 4. Next, (b,0) causes (z,0) also to be scheduled for time 4. Hence at time 4, z will undergo both a 0-to-1 and a 1-to-0 transition, which will create a zero-width "spike." Although the propagation of this spike may not pose a problem, it is unacceptable to have the results depending on the order of processing of the concurrent events. Algorithm 3.4, shown in Figure 3.34, overcomes this problem by detecting the case in which the gate output is repeatedly scheduled for the same time and by canceling the previously scheduled event. Identifying this situation is helped by maintaining the *last scheduled time*, *lst(j)*, for every gate j.

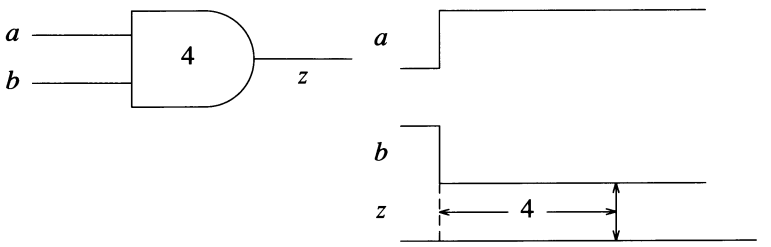


Figure 3.33 Processing of multiple input changes by Algorithm 3.3

```
for every event (i,v_i') pending at the current time t
  begin
    v(i) = v_i'
    for every j on the fanout list of i
      begin
        update input values of j
        v_j' = evaluate (j)
        if v_j' ≠ lsv(j) then
          begin
            t' = t+d(j)
            if t' = lst(j)
              then cancel event (j,lsv(j)) at time t'
            schedule (j,v_j') for time t'
            lsv(j) = v_j'
            lst(j) = t'
          end
      end
  end
```

Figure 3.34 Algorithm 3.4 — one-pass strategy with suppression of zero-width spikes

Let us check whether with this extra bookkeeping, the one-pass strategy is still more efficient than the two-pass strategy. The number of insertions in the *Activated* set performed during a simulation run by Algorithm 3.2 is about Nf; these are eliminated in Algorithm 3.4. Algorithm 3.4 performs the check for zero-width spikes only for gates with output events, hence only N times. Canceling a previously scheduled event is an expensive operation, because it involves a search for the event. However, in most instances $t + d(j) > lst(j)$, thus event canceling will occur infrequently, and the

associated cost can be ignored. Therefore we can conclude that the one-pass strategy is more efficient.

The Timing Wheel

With the event list implemented by the data structure shown in Figure 3.27, scheduling an event for time t requires first a search of the ordered list of headers to find the position for time t. The search time is proportional to the length of the header list, which, in general, increases with the size of the circuit and its activity. The header list is usually dense, that is, the differences between consecutive times for which events are scheduled are small. This suggests that employing an array of headers (see Figure 3.35), rather than a linked list, would be more efficient, since the search is avoided by using the time t to provide an index into the array. The disadvantage is that the process of advancing the simulation time has now to scan all headers (which follow the current time) until the next one with activity is encountered. But if the headers are dense, this represents only a minor overhead.

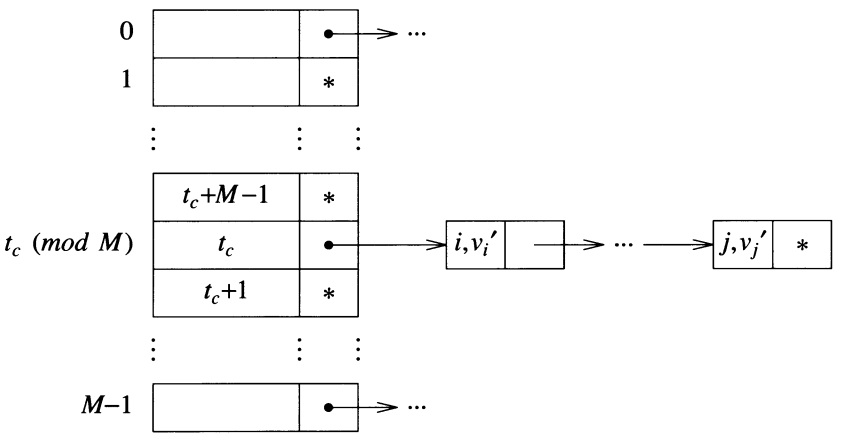


Figure 3.35 Timing wheel

The array of headers and their associated lists store events scheduled to occur between the current simulation time t_c and t_c+M−1, where M is the size of the array. The header for the events scheduled for a time t in this range appears in the array in position t modulo M. Because of the circular structure induced by the modulo-M operation, the array is referred to as a *timing wheel* [Ulrich 1969]. Any event scheduled for time t_c+d, where d<M, can be inserted in the event list without search. Events beyond the range of the wheel ($d \geq M$) are stored in an overflow "remote" event list of the type shown in Figure 3.27. Insertions in the remote list require searching, but in general, most events will be directly processed via the timing wheel. To keep the number of headers in the remote list at a minimum, events from that list should be brought into the wheel as soon as their time becomes included in the wheel range.

3.10.2 Other Logic Values

3.10.2.1 Tristate Logic

Tristate logic allows several devices to time-share a common wire, called a *bus*. A bus connects the outputs of several *bus drivers* (see Figure 3.36). Each bus driver is controlled by an "enable" signal E. When E=1, the driver is enabled and its output O takes the value of the input I; when E=0, the driver is disabled and O assumes a high-impedance state, denoted by an additional logic value Z. O=Z means that the output is in effect disconnected from the bus, thus allowing other bus drivers to control the bus. In normal operation, at most one driver is enabled at any time. The situation when two bus drivers drive the bus to opposite binary values is called a *conflict* or a *clash* and may result in physical damage to the devices involved. A pull-up (pull-down) function, realized by connecting the bus to power (ground) via a resistor, provides a "default" 1 (0) logic value on the bus when all the bus drivers are disabled.

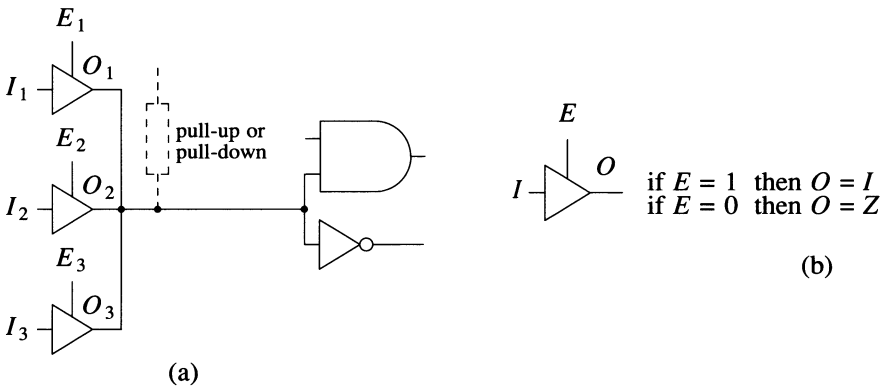


Figure 3.36 (a) Bus (b) bus driver

A bus represents another form of wired logic (see Section 2.4.4) and can be similarly modeled by a "dummy" component, whose function is given in Figure 3.37. Note that while the value observed on the bus is the forced value computed by the bus function, for every bus driver the simulator should also maintain its own driven value, obtained by evaluating the driver. One task of the bus evaluation routine is to report conflicts or potential conflicts to the user, because they usually indicate design errors. Sometimes, it may also be useful to report situations when multiple drivers are enabled, even if their values do not clash. When all the drivers are disabled, the resulting Z value is converted to a binary value if a pull-up or a pull-down is present; otherwise the devices driven by the bus will interpret the Z value as u (assuming TTL technology).

Recall that the unknown logic value u represents a value in the set $\{0,1\}$. In simulation, both the input I and the enable signal E of a bus driver can have any of the values $\{0,1,u\}$. Then what should be the output value of a bus driver with I=1 and E=u? A case-by-case analysis shows that one needs a new "uncertain" logic value to

	0	1	Z	u
0	0^1	u^2	0	u^3
1	u^2	1^1	1	u^3
Z	0	1	Z^4	u
u	u^3	u^3	u	u^3

Figure 3.37 Logic function of a bus with two inputs (1 — report multiple drivers enabled; 2 — report conflict; 3 — report potential conflict; 4 — transform to 1(0) if pull-up (pull-down) present)

represent a value in the set $\{1,Z\}$. Figure 3.38 gives a complete truth table of a bus driver; the results for E=u are given as sets of values.

		E		
		0	1	u
I	0	Z	0	$\{0,Z\}$
	1	Z	1	$\{1,Z\}$
	u	Z	u	$\{u,Z\}$

Figure 3.38 Truth table for a bus driver

3.10.2.2 MOS Logic

In this section we describe several techniques that allow traditional simulation methods to be extended to MOS components. The basic component in MOS logic is the *transmission gate* (Figure 3.39(a)), which works as a switch controlled by the gate signal G. When G=1, the transistor is on and the switch is closed, thus connecting its source (S) and drain (D) terminals; when G=0 the transistor is off and the switch is open. Although a transmission gate is intrinsically bidirectional, it can be used as a unidirectional or as a bidirectional element. As a unidirectional element (Figure 3.39(b)), its function is similar to that of a bus driver, except that when the transmission gate is open, the wire connected to its output retains its previous value. This is caused by the charge stored on the stray capacitance C associated with the output wire. (After a long *decay time*, C will be discharged, but usually circuits are operated sufficiently fast so that decay times can be considered infinite.) If the output of the transmission gate is connected to a bus without a pull-up or pull-down (Figure 3.39(c)), the stored value is the last forced value on the bus. This stored value, however, is overridden by the value driven by any gate (connected to the same bus) that is turned on and thus provides an escape path for the stored charge.

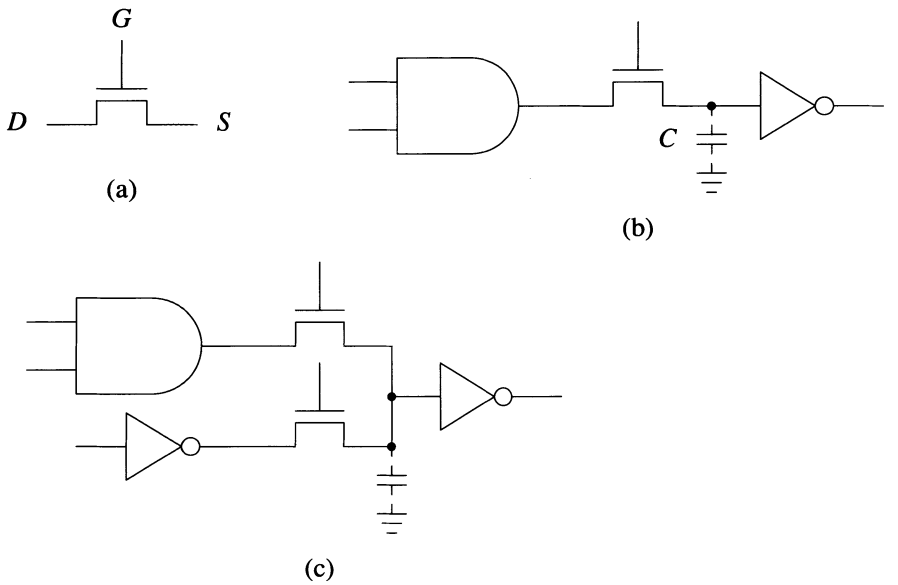


Figure 3.39 (a) Transmission gate (b) Transmission gate used as a unidirectional component (c) Transmission gates connected to a bus

This behavior illustrates the concept of *logical strength*, which is a digital measure of the relative current drive capability of signal lines. The values stored on wires — also called *floating* or *weak values* and denoted by Z_0 and Z_1 — correspond to low-current drive capability. They can be overridden by *strong values* (0 and 1) supplied by outputs of turned-on gates (or power and ground leads, or primary input leads), which have high-current drive capability.

The diagram given in Figure 3.40 shows the strength relations of a typical set of values used in MOS simulation. The level of a value in the diagram corresponds to its strength. The unknown value u is the strongest, and the high-impedance Z is the weakest. The weak unknown value Z_u denotes an unknown stored charge associated with a wire. The function B of a bus driven with the values v_1, v_2, ..., v_n can be defined as producing a unique value v_c=B (v_1, v_2, ...v_n), where v_c is the weakest value whose strength is greater than or equal to the strength of every v_i ($1 \le i \le n$). Thus $B(0,1) = u$, $B(Z_0,1) = 1$, $B(Z_0,Z_1) = Z_u$, $B(1,Z) = 1$.

The concept of strength allows a pull-up load connected to power to be treated as a logic component, called an *attenuator* [Hayes 1982], whose function is to transform the strong 1 provided by the power line into a Z_1 value supplied to the bus. This value is overridden by a strong value driven by any gate, and it overrides the Z value provided by the turned-off gates. Similarly, a pull-down load connected to ground transforms a 0 value into a Z_0.

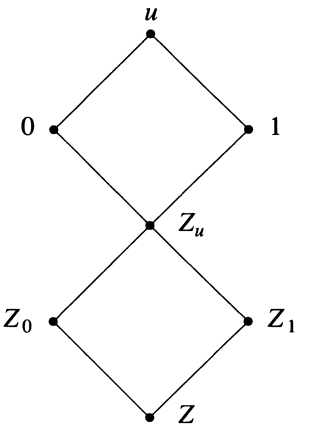


Figure 3.40 Typical values used in MOS simulation and their relative strengths

Figure 3.41(a) illustrates a transmission gate used as a bidirectional switch. When G=0, the switch is open, and C and F are independent. But when G=1, C and F become connected by a bus. One way of modeling the bidirectionality of the gate controlled by G is with two unidirectional gates connected back to back, as shown in Figure 3.41(b) [Sherwood 1981].

3.10.3 Other Delay Models

3.10.3.1 Rise and Fall Delays

Conceptually, the rise/fall delay model is a simple extension of the transition-independent delay model. Instead of using one delay value for scheduling events, one selects the rise or the fall delay of an element, depending on the result of its evaluation. Since rise and fall delays are associated (respectively) with 0→1 and 1→0 transitions, a first question is what delays should be used for transitions involving u values. For example, a 0→u transition is either a 0→1 transition or not an event at all. Thus if an event occurs, its associated delay is the rise delay. Similarly u→1 will use the rise delay, and 1→u and u→0 will use the fall delay.

The following example illustrates some of the complications arising from the rise/fall delay model.

Example 3.3: Consider an inverter with $d_r = 12$ and $d_f = 7$. Assume that a 1→0→1 pulse of width 4 arrives at its input. The first input transition (occurring at time 0) results in scheduling the output to go to 1 at time $0 + t_r = 12$. The second input transition at time 4 would cause the output to be scheduled to go to 0 at time $4 + t_f = 11$. This "impossible" sequence of events appears because the effect of the second input transition propagates to the output faster than the effect of the first one. Here the correct result should be that the output does not change at all; thus the first

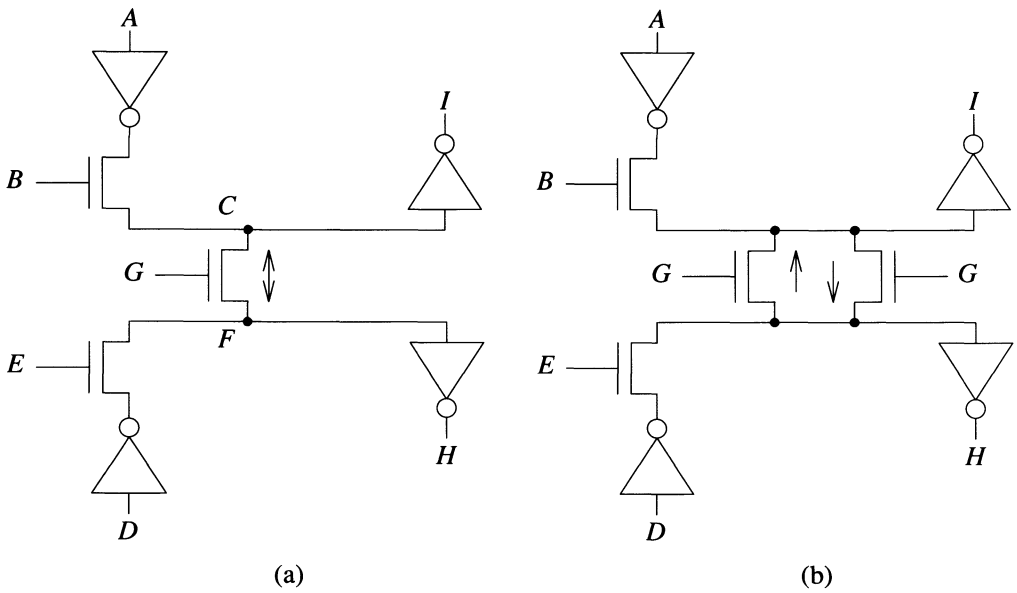


Figure 3.41 (a) Transmission gate used as bidirectional component (b) Model by unidirectional gates

event should be canceled and the second one not scheduled. Note that the input pulse is suppressed, even without an inertial delay model. □

As this example shows, the main difficulty associated with the rise/fall delay model arises when previously scheduled events have to be canceled. Let v_j' be the value computed by the evaluation of an activated gate j at the current time t. The procedure *Process_result*, outlined in Figure 3.42, processes the result of the evaluation and determines whether event canceling is needed. For this analysis, we assume (as in Algorithm 3.4) that for every signal j, one maintains its last scheduled value — *lsv(j)* — and its last scheduled time — *lst(j)*. In the "normal" case, the new event of j occurs after the last scheduled event of j. When the order of these two events is reversed, the value v_j' reaches the output before *lsv(j)*, and the last scheduled event is canceled. After that, *lst(j)* and *lsv(j)* are updated to correspond to the new last scheduled event of j and the entire process is repeated. If after canceling its last scheduled event, j has more pending events, finding the last one requires a search of the event list. Otherwise, *lsv(j)* can be set to the current value of j, and *lst(t)* can be set to the current simulation time. (This analysis is helped by maintaining the count of pending events of every signal.)

Let us apply this procedure to handle the result of the second evaluation of the inverter of Example 3.3. Let j be the output of the inverter. At time t=4, the last scheduled event is a transition to *lsv(j)*=1 at time *lst(j)*=12. The result of the second evaluation is v_j'=0. The delay d for the transition *lsv(j)*$\rightarrow v_j'$ is d_f=7, so this event would occur at time t'=11. Because $t' <$ *lst(j)*, the event scheduled for time 12 is canceled. As there

```
Process_result (j, v_j')
begin
    while v_j'≠lsv(j)
        begin
            d = delay for the transition lsv(j)→v_j'
            t' = t+d
            if t' > lst(j) then
                begin
                    schedule (j, v_j') for time t'
                    lst(j) = t'
                    lsv(j) = v_j'
                    return
                end
            cancel event (j, lsv(j)) at time lst(j)
            update lst(j) and lsv(j)
        end
end
```

Figure 3.42 Processing an evaluation result with the rise/fall delay model

are no more pending events for j, $lsv(j)$ is set to its current value (0), and $lst(j)$ is set to t=4. Now v_j'=$lsv(j)$ and no event is scheduled.

3.10.3.2 Inertial Delays

The inertial delay model leads to problems similar to those of the rise/fall delay model. Here event canceling is required to implement spike filtering. For example, consider a gate j with an output inertial delay d_I. Assume that j has been scheduled to go to $lsv(j)$ at time $lst(j)$, and that a second evaluation of j has computed a new value $v_j' \neq lsv(j)$, which would occur at time t'. If $t'-lst(j) < t_I$, then the output pulse is suppressed by canceling the last scheduled event of j.

3.10.3.3 Ambiguous Delays

In the ambiguous delay model, the transport delay of a gate is within a range $[d_m, d_M]$. (We assume transition-independent delays). Thus when the gate responds to a transition occurring at an input at time t, its output will change at some time during the interval $[t+d_m, t+d_M]$. (See Figure 3.14(c).) To reflect this uncertainty in simulation, we will use the 6-valued logic of Figure 3.24, where 0/1 (1/0) is the value of a signal during the interval in which it changes from 0 to 1 (1 to 0). Hence a gate output changing from 0 to 1 goes first from 0 to 0/1, holds this value for an interval d_M-d_m, then goes from 0/1 to 1. For uniformity, we will treat a 0→1 transition on a primary input as a 0→0/1 event, followed (at the same time) by a 0/1→1 event. When scheduling a gate output, its minimum delay d_m will be used for changes to the 0/1 and 1/0 values, and its maximum delay d_M for changes to binary values.

Example 3.4: In the circuit shown in Figure 3.43 the delay of C is between 2 and 4, and that of D between 2 and 5. Assume that at time 0, A changes from 0 to 1. C is evaluated twice, first with A=0/1, then with A=1. The first evaluation causes C to change to 0/1 after a delay $d_m(C)$=2; as a result of the second evaluation, C is scheduled to take value 1 after a delay $d_M(C)$=4. At time t=2, D is evaluated with C=0/1, and its new 1/0 value is scheduled for time $t+d_m(D)$=4. The next evaluation of D (at time 4) produces a binary value (0), which is scheduled using the delay $d_M(D)$=5. □

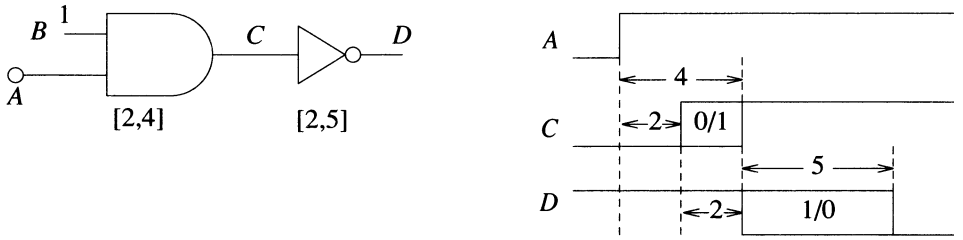


Figure 3.43 Simulation with ambiguous delay model

A simulator using the ambiguous delay model computes the earliest and the latest times signal changes may occur. In the presence of reconvergent fanout, this analysis may lead to overly pessimistic results. For example, in the circuit shown in Figure 3.44, assume that A changes from 1 to 0 during the interval [20, 30]. If the delay of B is between 6 and 10, the simulator predicts that B changes from 0 to 1 during the interval [26, 40]. Thus it appears that in the interval [26, 30], A=1/0 and B=0/1, which creates a static hazard at C. This result, however, is incorrect because the transitions of B and A are not independent; the transition of B occurs between 6 and 10 time units after the transition of A, so that no pulse can appear at C. Additional processing is required to remove the *common ambiguity* among the inputs of the same gate [Bowden 1982].

A less pessimistic way of treating ambiguous delays is to perform several simulation runs, each one preceded by a random selection of the delay value of every element from its specified range.

When a tester applies input vectors to a circuit, the inputs will not change at the same time because of skews introduced by the tester. These skews can be taken into account in simulation by adding a gate with ambiguous delays to every primary input of the model.

3.10.4 Oscillation Control

Although Procedure 3.2 can detect potential oscillations, the arbitrary delay model it uses often leads to pessimistic results. That is, the problems reported will not occur under a more realistic delay model. Another drawback of Procedure 3.2 is that it is not applicable for real-time inputs. In this section we discuss other methods for detecting oscillations in event-driven simulation.

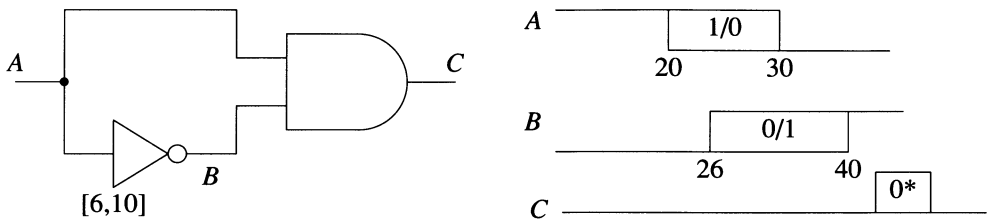


Figure 3.44 Pessimistic result in ambiguous delay simulation

Why is this problem important? An accurate simulation of a circuit that oscillates will result in repeated scheduling and processing of the same sequence of events, with the simulation program apparently caught in an "endless loop." This may consume a lot of CPU time. Figure 3.45 illustrates a simple example of oscillation. Assume that every gate has a delay of 3 nsec, and that initially S=R=1, y=0, and $\bar{y}$=1. If S goes to 0 for 4 nsec, y and $\bar{y}$ will oscillate as a result of a pulse continuously traveling around the loop composed of the two gates.

The process of detecting oscillations during simulation and taking appropriate corrective action is called *oscillation control.* The corrective action is to set the oscillating signals to u. In addition, it is desirable to inform the user about the oscillation and to provide the option of aborting the simulation run, which may have become useless. Oscillation can be controlled at two levels, referred to as local control and global control.

Local oscillation control is based on identifying conditions causing oscillations in specific subcircuits, such as latches or flip-flops. For the example in Figure 3.45, y=$\bar{y}$=0 is an oscillating condition. Note that this implies R=S=1, which is the condition enabling propagation along the loop. Also, the states y=0, $\bar{y}$=u and y=u, $\bar{y}$=0 can cause oscillation. The appropriate corrective action is to set y=$\bar{y}$=u.

Local oscillation control can be implemented in several ways. One way is to have the simulator monitor user-specified conditions on certain gates [Chappel *et al.* 1974]; these are typically cross-coupled gates used in latches. An easier way to implement local oscillation control is via modeling techniques. If latches are modeled as primitive or user-defined functional elements, checking for conditions causing oscillations (and for other "illegal" states as well) can be part of the model. Gate-level models can also be used to detect and correct oscillations [Ulrich *et al.* 1972]. Figure 3.45(c) shows such a model. In the normal operation of the latch (y=0 and $\bar{y}$=1, or y=1 and $\bar{y}$=0, or y=$\bar{y}$=1), G=1 and this value does affect the rest of the circuit. However, any of the "illegal" states (y=$\bar{y}$=0, or y=0 and $\bar{y}$=u, or $\bar{y}$=0 and y=u) will cause G=u, which in turn (with R=S=1) will stop the oscillation by setting y=$\bar{y}$=u.

Because local oscillation control applies only for specific subcircuits, it may fail to detect oscillations involving feedback loops that are not contained in those subcircuits. Correct identification of this type of global oscillation is not computationally feasible,

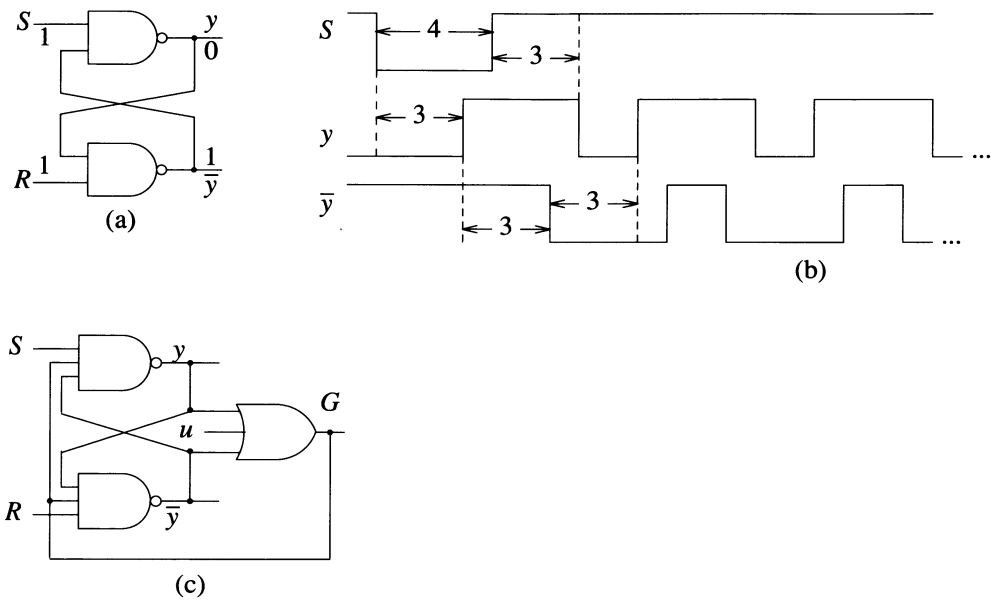


Figure 3.45 (a) Latch (b) Oscillation waveforms (c) Model for oscillation control

since it would require detecting cyclic sequences of values for any signal in the circuit. Most simulators implement *global oscillation control* by identifying unusually high activity during simulation. A typical procedure is to count the number of events occurring after any primary input change. When this number exceeds a predefined limit, the simulator assumes that an oscillation has been detected and sets all currently active signals to u to terminate the activity caused by oscillations. Of course, this procedure may erroneously label legitimate high activity as an oscillation, but the user can correct this mislabeling by adjusting the activity limit.

3.11 Simulation Engines

Logic simulation of complex VLSI circuits and systems is a time-consuming process. This problem is made more acute by the extensive use of simulation in the design cycle to validate the numerous design changes a project usually undergoes. One solution to this problem is provided by hardware specially designed to speed up simulation by using parallel and/or distributed processing architectures. Such special-purpose hardware, called a *simulation engine* or a simulator hardware accelerator, is usually attached to a general-purpose host computer. Typically, the host prepares and downloads the model and the applied stimuli into the simulation engine, which performs the simulation and returns the results to the host.

Simulation engines achieve their speed-up based on two different strategies, referred to as model partitioning and algorithm partitioning. *Model partitioning* consists of

dividing the model of the simulated circuit into disjoint subcircuits, concurrently simulated by identical processing units (PUs), interconnected by a communication medium (see Figure 3.46). *Algorithm partitioning* consists of dividing the simulation algorithm into tasks distributed among different PUs working concurrently as stages of a pipeline. Many simulation engines combine both model partitioning and algorithm partitioning.

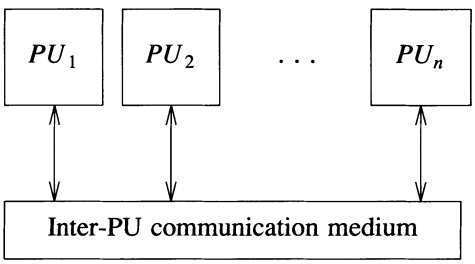


Figure 3.46 Simulation engine based on model partitioning (connections with host not shown)

The Yorktown Simulation Engine (YSE) [Denneau 1982] is based on partitioning the model among up to 256 PUs, interconnected by a crossbar switch. Each PU simulates a subcircuit consisting of up to 4K gates, with some specialized PUs dedicated for simulating RAMs and ROMs. The YSE implements a compiled simulation algorithm, in which every gate is evaluated for every input vector. Internally, every PU employs algorithm partitioning, with pipeline stages corresponding to the following structure of a compiled simulator:

```
for every gate G
    begin
        determine inputs of G
        get the values of the inputs
        evaluate G
        store value of G
    end
```

Figure 3.47 shows a block diagram of a PU. The "program counter" provides the index of the next gate to be evaluated (recall that in compiled simulation a gate is evaluated only after all its input values are known). This index is used to access the "instruction memory," which provides the inputs of the gate, its type and other information needed for evaluation, and the address where the computed output value should be stored. Signal values are stored in the "data memory" block. The YSE uses four logic values (0, 1, *u*, and *Z*), which are coded with two bits. Evaluations are done by the "function unit," whose structure is shown in Figure 3.48. A gate in the YSE model can have up to four inputs, whose values are provided by the data memory. These values are first passed through "generalized DeMorgan" (GDM) memory blocks, which contain truth tables for 16 functions of one 4-valued variable. The function of a

GDM is selected by a 4-bit GDM code provided by the instruction memory. Typical GDM functions are identity (in which the input value is passed unchanged), inversion, and constant functions. (Constant functions are primarily used to supply noncontrolling values to unused gate inputs.) Evaluations are done with a zoom table technique. The tables are stored in the "function memory," whose address is obtained by concatenating the gate type with the input values transformed by the GDM codes. The output value is similarly passed through another GDM.

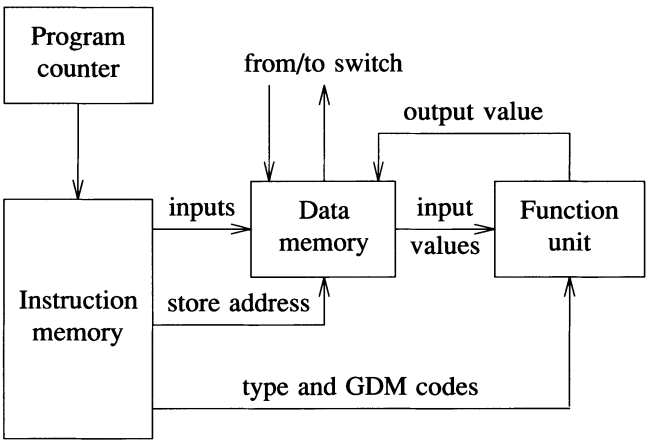


Figure 3.47 YSE processing unit

Model partitioning is done by the host as a preprocessing step. Because of partitioning, inputs of a subcircuit assigned to one PU may be outputs of subcircuits assigned to different PUs. The values of such signals crossing the partition boundaries must be transmitted across the inter-PU switch. All PUs are synchronized by a common clock, and each PU can evaluate a gate during every clock cycle. The newly computed output values are sent out to the switch, which can route them further to the other PUs that use them. The scheduling of the inter-PU communications is done together with the model partitioning [Kronstadt and Pfister 1982]. The evaluation of a gate by one PU may be delayed until all its input values computed by other PUs arrive through the switch. The goal of the partitioning is to minimize the total waiting time during simulation.

Figure 3.49 depicts the architecture of an event-driven simulation engine, similar to that described in [Abramovici *et al.* 1983]. The tasks of the algorithm are partitioned among several PUs, configured as stages of a circular pipeline. The PUs may be hardwired (i.e., their tasks are directly implemented in hardware) or programmable. Programmable PUs offer more flexibility at a cost of some loss in performance. Another advantage of programmable PUs is that they can be identically built and their different tasks can be realized by different code. (Microprogrammable PUs, whose tasks are directly implemented in microcode, offer a good trade-off between

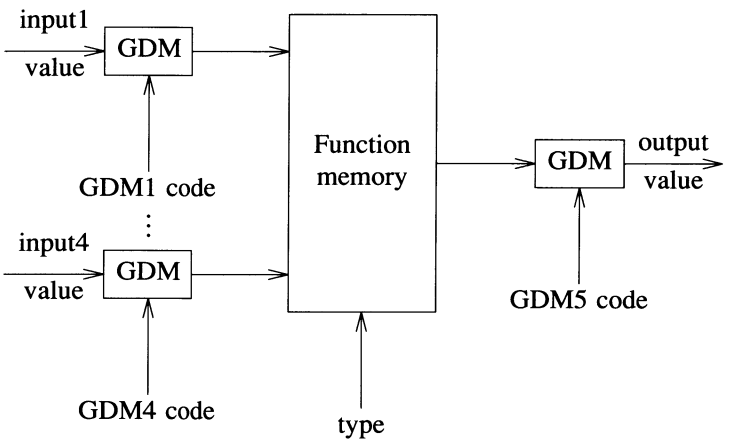


Figure 3.48 YSE function unit

performance and flexibility.) Every PU has a local memory that can be loaded by the host (connections with the host — usually a shared bus — are not shown in Figure 3.49).

A brief outline of the main tasks of the PUs of Figure 3.49 follows. The Event List Manager maintains the event list in its local memory. It advances the simulation time to the next time for which events are scheduled, retrieves these concurrent events, and sends them to the Current Event PU. This PU updates the signal values (which are maintained in its local memory) and sends the incoming events to the Event Router. In addition, the Current Event PU performs auxiliary tasks such as global oscillation control and sending results to the host. The Event Router stores the fanout list in its local memory. It determines the activated gates and sends them (together with their input events) to the Evaluation Unit. The Evaluation Unit maintains the input values of every gate in its local memory. It updates the input values of the activated gates and evaluates them. The resulting output events are sent to the Scheduler, whose local memory stores delay data. The Scheduler determines the appropriate delay to be used for each event and sends the events and their times of occurrence to the Event List Manager, which inserts them in the event list. The Event List Manager also processes the events scheduled for the primary inputs, which are received from the host.

The efficiency of a pipeline depends on the distribution of the workload among its stages. If the average workload is not uniformly distributed, the stage that is most heavily loaded becomes the bottleneck of the pipeline. Such an imbalance can be corrected by replacing the bottleneck stage with several parallel processors. Even a pipeline that is balanced on the average may have temporary disparities between the workloads of different stages. These variations can be smoothed out by buffering the data flow between PUs using first-in first-out (FIFO) buffers.

A simulation engine implementing event-driven simulation can also be based on model partitioning among parallel PUs, as shown in Figure 3.46. The communication

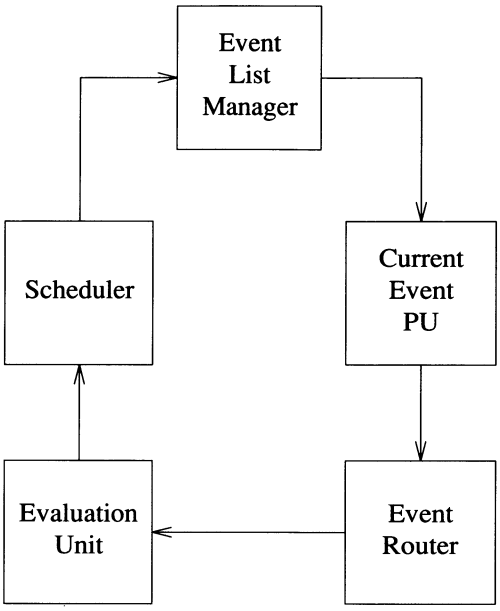


Figure 3.49 Simulation engine based on algorithm partitioning for event-driven simulation

medium is primarily used to transmit events that propagate between subcircuits assigned to different PUs. The architecture described in [VanBrunt 1983] combines model partitioning and algorithm partitioning, by connecting up to 16 PUs of the type illustrated in Figure 3.49 via a shared bus.

In an architecture based on model partitioning, one PU, called the master, assumes the task of coordinating the work of the other PUs, referred to as slaves [Levendel *et al.* 1982]. Each slave has its own event list, but the simulation time is maintained by the master. In every simulation cycle, the master advances the simulation time by one time unit and initiates all the slaves. Then each slave processes the events pending for that time (if any) in its local event list. Events generated for signals internal to a subcircuit are inserted into the local event list, while events crossing the partition boundaries are sent to the appropriate PUs via the communication medium. Every PU reports to the master when it is done. The master synchronizes the slaves by waiting for all of them to complete processing before advancing the time and initiating the new cycle.

The partitioning of the simulated circuit into subcircuits assigned to different PUs is guided by two objectives: (1) maximize the concurrency achieved by the ensemble of PUs, by trying to uniformly distribute the simulation activity among PUs, and (2) minimize the inter-PU communication. Partitioning algorithms are discussed in [Levendel *et al.* 1982, Agrawal 1986].

REFERENCES

[Abramovici *et al.* 1983] M. Abramovici, Y. H. Levendel, and P. R. Menon, "A Logic Simulation Machine," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-2, No. 2, pp. 82-94, April, 1983.

[Agrawal *et al.* 1980] V. D. Agrawal, A. K. Bose, P. Kozak, H. N. Nham, and E. Pacas-Skewes, "A Mixed-Mode Simulator," *Proc. 17th Design Automation Conf.*, pp. 618-625, June, 1980.

[Agrawal 1986] P. Agrawal, "Concurrency and Communication in Hardware Simulators," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-5, No. 4, pp. 617-623, October, 1986.

[Agrawal *et al.* 1987] P. Agrawal, W. T. Dally, W. C. Fisher, H. V. Jagadish, A. S. Krishnakumar, and R. Tutundjian, "MARS: A Multiprocessor-Based Programmable Accelerator," *IEEE Design & Test of Computers*, Vol. 4, No. 5, pp. 28-36, October, 1987.

[Barzilai *et al.* 1987] Z. Barzilai, J. L. Carter, B. K. Rosen, and J. D. Rutledge, "HSS — A High-Speed Simulator," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 4, pp. 601-617, July, 1987.

[Blank 1984] T. Blank, "A Survey of Hardware Accelerators Used in Computer-Aided Design," *IEEE Design & Test of Computers*, Vol. 1, No. 3, pp. 21-39, August, 1984.

[Bowden 1982] K. R. Bowden, "Design Goals and Implementation Techniques for Time-Based Digital Simulation and Hazard Detection," *Digest of Papers 1982 Intn'l. Test Conf.*, pp. 147-152, November, 1982.

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

[Breuer and Harrison 1974] M. A. Breuer and L. Harrison," Procedures for Eliminating Static and Dynamic Hazards in Test Generation," *IEEE Trans. on Computers*, Vol. C-23, No. 10, pp. 1069-1078, October, 1974.

[Butler *et al.* 1974] T. T. Butler, T. G. Hallin, K. W. Johnson, and J. J. Kulzer, "LAMP: Application to Switching System Development," *Bell System Technical Journal*, Vol. 53, No. 8, pp. 1535-1555, October, 1974.

[Chappell and Yau 1971] S. G. Chappell and S. S. Yau, "Simulation of Large Asynchronous Logic Circuits Using an Ambiguous Gate Model," *Proc. Fall Joint Computer Conf.*, pp. 651-661, 1971.

[Chappell *et al.* 1974] S. G. Chappell, C. H. Elmendorf, and L. D. Schmidt, "LAMP: Logic-Circuit Simulators," *Bell System Technical Journal*, Vol. 53, No. 8, pp. 1451-1476, October, 1974.

[Chen *et al.* 1984] C. F. Chen, C-Y. Lo, H. N. Nham, and P. Subramaniam, "The Second Generation MOTIS Mixed-Mode Simulator," *Proc. 21st Design Automation Conf.*, pp. 10-17, June, 1984.

[Denneau 1982] M. M. Denneau, "The Yorktown Simulation Engine," *Proc. 19th Design Automation Conf.*, pp. 55-59, June, 1982.

[Eichelberger 1965] E. B. Eichelberger, "Hazard Detection in Combinational and Sequential Circuits," *IBM Journal of Research and Development*, Vol. 9, pp. 90-99, March, 1965.

[Evans 1978] D. J. Evans, "Accurate Simulation of Flip-Flop Timing Characteristics," *Proc. 15th Design Automation Conf.*, pp. 398-404, June, 1978.

[Hayes 1982] J. P. Hayes, "A Unified Switching Theory with Applications to VLSI Design," *Proc. of the IEEE*, Vol. 70, No. 10, pp. 1140-1151, October, 1982.

[Hayes 1986] J. P. Hayes, "Uncertainty, Energy, and Multiple-Valued Logics," *IEEE Trans. on Computers*, Vol. C-35, No. 2, pp. 107-114, February, 1986.

[Hitchcock 1982] R. B. Hitchcock, "Timing Verification and Timing Analysis Program," *Proc. 19th Design Automation Conf.*, pp. 594-604, June, 1982.

[Ishiura *et al.* 1987] N. Ishiura, H. Yasuura, and S. Yajma, "High-Speed Logic Simulation on Vector Processors," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 3, pp. 305-321, May, 1987.

[Kronstadt and Pfister 1982] E. Kronstadt and G. Pfister, "Software Support for the Yorktown Simulation Engine," *Proc. 19th Design Automation Conf.*, pp. 60-64, June, 1982.

[Levendel *et al.* 1982] Y. H. Levendel, P. R. Menon, and S. H. Patel, "Special-Purpose Logic Simulator Using Distributed Processing," *Bell System Technical Journal*, Vol. 61, No. 10, pp. 2873-2909, December, 1982.

[Lewis 1972] D. W. Lewis, "Hazard Detection by a Quinary Simulation of Logic Devices with Bounded Propagation Delays," *Proc. 9th Design Automation Workshop*, pp. 157-164, June, 1972.

[Monachino 1982] M. Monachino, "Design Verification System for Large-Scale LSI Designs," *Proc. 19th Design Automation Conf.*, pp. 83-90, June, 1982.

[Saab *et al.* 1988] D. G. Saab, R. B. Mueller-Thuns, D. Blaauw, J. A. Abraham, and J. T. Rahmeh, "CHAMP: Concurrent Hierarchical and Multilevel Program for Simulation of VLSI Circuits," *Proc. Intn'l. Conf. on Computer-Aided Design*, pp. 246-249, November, 1988.

[Sasaki *et al.* 1981] T. Sasaki, A. Yamada, and T. Aoyama, "Hierarchical Design Verification for Large Digital Systems," *Proc. 18th Design Automation Conf.*, pp. 105-112, June, 1981.

[Schuler 1972] D. M. Schuler, "Simulation of NAND Logic," *Proc. COMPCON 72*, pp. 243-245, September, 1972.

[Sherwood 1981] W. Sherwood, "A MOS Modeling Technique for 4-State True-Value Hierarchical Logic Simulation," *Proc. 18th Design Automation Conf.*, pp. 775-785, June, 1981.

[Szygenda *et al.* 1970] S. A. Szygenda, D. W. Rouse, and E. W. Thompson, "A Model and Implementation of a Universal Time Delay Simulator for Digital Nets," *Proc. AFIPS Spring Joint Computer Conf.*, pp. 207-216, 1970.

[Szygenda and Lekkos 1973] S. A. Szygenda and A. A. Lekkos, "Integrated Techniques for Functional and Gate-Level Digital Logic Simulation," *Proc. 10th Design Automation Conf.*, pp. 159-172, June, 1973.

[Tokoro *et al.* 1978] M. Tokoro, M. Sato, M. Ishigami, E. Tamura, T. Ishimitsu, and H. Ohara, "A Module Level Simulation Technique for Systems Composed of LSIs and MSIs," *Proc. 15th Design Automation Conf.*, pp. 418-427, June, 1978.

[Ulrich 1965] E. G. Ulrich, "Time-Sequenced Logic Simulation Based on Circuit Delay and Selective Tracing of Active Network Paths," *Proc. 20th ACM Natl. Conf.*, pp. 437-448, 1965.

[Ulrich 1969] E. G. Ulrich, "Exclusive Simulation of Activity in Digital Networks," *Communications of the ACM*, Vol. 13, pp. 102-110, February, 1969.

[Ulrich *et al.* 1972] E. G. Ulrich, T. Baker, and L. R. Williams, "Fault-Test Analysis Techniques Based on Logic Simulation," *Proc. 9th Design Automation Workshop*, pp. 111-115, June, 1972.

[VanBrunt 1983] N. VanBrunt, "The Zycad Logic Evaluator and Its Application to Modern System Design," *Proc. Intn'l. Conf. on Computer Design*, pp. 232-233, October, 1983.

[Yoeli and Rinon 1964] M. Yoeli and S. Rinon, "Applications of Ternary Algebra to the Study of Static Hazards," *Journal of the ACM*, Vol. 11, No. 1, pp. 84-97, January, 1964.

PROBLEMS

3.1 Consider a function $f(x_1,x_2,x_3)$ defined by the following primitive cubes: $x10 \mid 0$, $11x \mid 0$, $x0x \mid 1$, and $011 \mid 1$. Determine the value of f for the following input vectors: $10u$, $01u$, $u1u$, and $u01$.

3.2 The D F/F used in the circuit shown in Figure 3.50 is activated by a $0 \rightarrow 1$ transition on its C input. Assume that a sequence of three $0 \rightarrow 1 \rightarrow 0$ pulses is applied to CLOCK.

a. Show that simulation with 3-valued logic fails to initialize this circuit.

b. Show that simulation with multiple unknown values does initialize this circuit (denote the initial values of the F/Fs by u_1 and u_2).

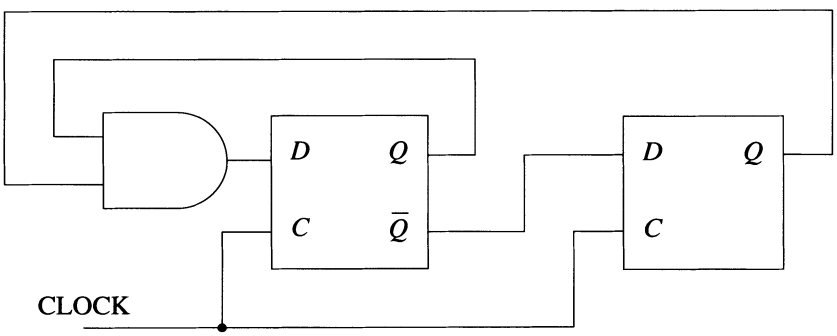


Figure 3.50

3.3 Use 3-valued compiled simulation to simulate the circuit of Figure 3.7 starting with $Q = u$ for the input vectors 01 and 11.

3.4 For 3-valued compiled simulation, someone may suggest using both 01 and 10 to represent the value u. The benefit of this dual encoding would be that the NOT(a) operation could be done just by complementing the 2-bit vector a (without bit swapping). Show that this encoding may lead to incorrect results.

3.5 Using the compiled simulation procedure outlined in Figure 3.9, simulate the latch given in Figure 3.11(a) for the vectors 00 and 11. Assume a model in which both Q and QN are treated as feedback lines (with equal delays).

3.6 Analyze the propagation of the pulse at A shown in Figure 3.14 for the following rise and fall delays of gate C: (1) $d_r = 3$, $d_f = 1$; (2) $d_r = 4$, $d_f = 1$; (3) $d_r = 5$, $d_f = 1$.

3.7 Explain why the (input or output) inertial delay of a gate cannot be greater than the smallest transport delay associated with the gate, i.e., $d_I \leq \min \{d_r, d_f\}$.

3.8 Determine the response of the F/F of Figure 3.16 to the two sets of stimuli given in Figure 3.51. Assume $q = 0$, $D = 1$, $S = 1$.

3.9 How can a truth table of an OR gate with four inputs be used to evaluate an OR gate with a different number of inputs?

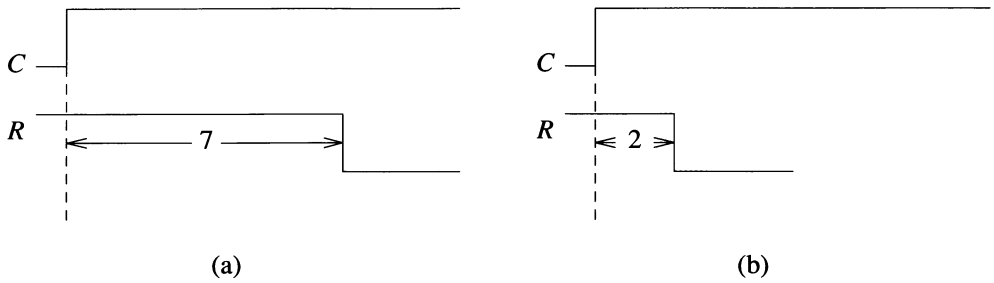


Figure 3.51

3.10 Determine a formula to compute the *utilization factor* of an array used to store the truth table of a function of n k-valued variables, where the value of a variable is coded with q bits (that is, the proportion of the array actually used).

3.11 Construct a zoom table for evaluating AND, OR, NAND, and NOR gates with two inputs with binary values.

3.12 Let us model an OR gate with two inputs as a sequential machine M whose state is given by the two counters, *c_count* and *u_count*, used to evaluate the gate via the input counting method. The input of M represents an input event of the OR gate, that is, a variable whose values are in the set $E = \{0\rightarrow1,\ 1\rightarrow0,\ u\rightarrow0,\ u\rightarrow1,\ 0\rightarrow u,\ 1\rightarrow u\}$. The output of M represents the result of evaluating the OR gate by a variable whose values are in the set $E \cup \Phi$, where Φ denotes no output event. Construct a state diagram for M, showing all possible state transitions during simulation. What is the initial state of M?

3.13 Outline evaluation routines for an XOR gate, based on: (1) input scanning, and (2) (a technique similar to) input counting.

3.14 Construct a truth table for a 2-input NAND gate for the 8-valued logic system of Figure 3.26.

3.15 Show that the existence of a static hazard is a necessary condition for the creation of a dynamic hazard.

3.16 Consider the circuit shown in Figure 3.52 and the input sequence shown. Calculate the output sequence at J and K using the 6-valued logic of Figure 3.24 and the 8-valued logic of Figure 3.26. Verify that the 8-valued logic system correctly identifies a dynamic hazard at J, while the 6-valued logic system produces the same result for lines J and K.

3.17 Show that for an asynchronous circuit with n feedback lines, Procedure 3.2 will simulate the circuit C at most $2n$ times.

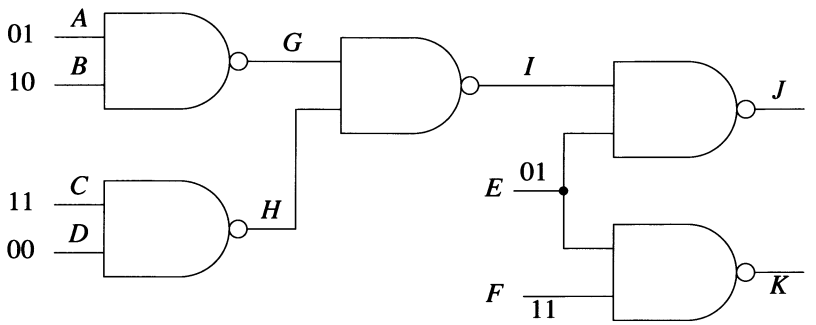


Figure 3.52

3.18 Consider the latch of Figure 3.53 and assume that, starting with the shown values, A changes to 1, then goes back to 0.

a. Simulate the circuit assuming an arbitrary delay model (apply Procedure 3.2).

b. Simulate the circuit assuming that all gates have 8 nsec delay and that A is 1 for 12 nsec.

c. If the results for a. and b. are different, explain why.

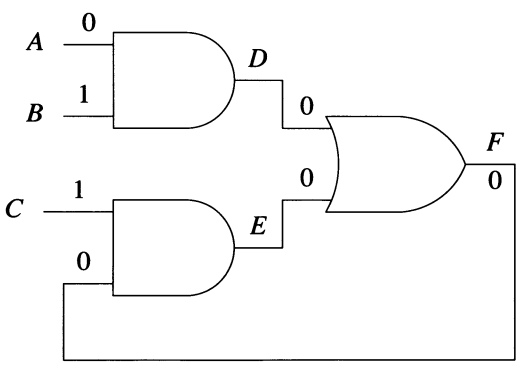


Figure 3.53

3.19 Regarding Algorithm 3.2 (Figure 3.31), it may appear that if $v(i) = lsv(i)$, i.e., the current and the last scheduled values of i are the same, then the gate i is "stable." Give a counterexample showing how i can have pending events when $v(i) = lsv(i)$. Will Algorithm 3.2 correctly handle such a case?

3.20 Assume that the largest possible gate delay is D. Discuss the advantages and disadvantages of using a timing wheel of size $M = D + 1$.

3.21 Extend the truth table of a bus with two inputs (given in Figure 3.37) to process all the output values of a bus driver shown in Figure 3.38.

3.22 Simulate the circuit of Figure 3.36(a), with $I_1 = u$, $E_1 = 0$, $I_2 = 0$, $E_2 = u$, $I_3 = 0$, $E_3 = 1$.

3.23 Simulate the circuit of Figure 3.41(a) for the following input vectors.

A	B	D	E	G
1	1	0	1	0
1	0	0	1	1
0	1	1	0	1
1	0	1	0	0

3.24 Simulate the latch shown in Figure 3.11(a) for the input sequence shown in Figure 3.54, assuming the following delays for the two gates:

a. $d = 2$

b. $d_l = 2$

c. $d_l = 1$

d. $d_r = 3$, $d_f = 1$

e. $d_r = 1$, $d_f = 3$

f. $d_m = 1$, $d_M = 2$

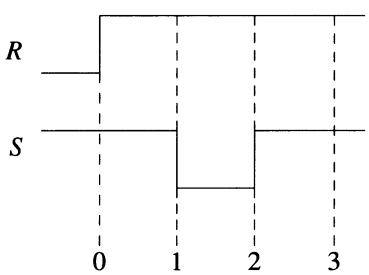


Figure 3.54

3.25 Determine the function of the circuit shown in Figure 3.55.

3.26 For the circuit of Figure 3.45(a), determine the relation between the width of a 1→0→1 pulse at S and the transport delays of the gates in the circuit, necessary for the oscillation to occur.

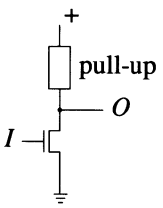


Figure 3.55