MIT/LCS/TR-304

SIMULATION TOOLS FOR DIGITAL LSI DESIGN

Christopher Jay Terman

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Nos. N00014-75-C-0661 & N00014-83-K-0125.

Simulation Tools for Digital LSI Design

by

Christopher Jay Terman

September 1983

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661, N00014-83-K-0125

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachusetts 02139**

Simulation Tools for Digital LSI Design

by

Christopher Jay Terman

Submitted to the Department of Electrical Engineering and Computer Science

on August 30, 1983 in partial fulfillment of the requirements for

the degree of Doctor of Philosophy

Abstract

This thesis proposes a timing simulator (RSIM) based on a uniquely simple transistor model. RSIM allows a designer to determine both the functional and approximate timing characteristics of a MOS network with more accuracy than gate-level simulation, and using larger circuits than are accommodated by circuit analysis programs. In RSIM, transistors are modeled as resistors; the logic states of a transistor's terminal nodes determine its effective resistance. Using this model, a MOS network is simulated as a network of resistors where each node's value is determined by the resistance of its connections to various inputs. Transition times are determined from the RC time constant calculated for the node by examining the surrounding network; (R from the transistors, C from the interconnect and gate capacitance). The network's behavior as inputs are given values is calculated by an efficient event-driven algorithm.

Two changes to the underlying model are also investigated:

(1) further simplifying the transistor model to an on/off switch (which can be thought of as a degenerate resistor). Several approaches to switch-level simulation are developed, one particularly well-suited for implementation using parallel hardware.

(2) modeling the behavior of a network of switches by a system of logic equations. Various compilation strategies are evaluated for producing code that implements the system of equations.

Name and Title of Thesis Supervisor:

Stephen A. Ward,
Associate Professor of Computer Science and Engineering

Key Words and Phrases:

circuit simulation, logic simulation, timing analysis, CAD tools

ACKNOWLEDGMENTS

Thanks everybody:

Steve Ward Ron Rivest Gerry Sussman

Bert Halstead Clark Baker Mark Johnson
and the rest of RTS, past and present

Dave Gross Doug Williams Bob Yodlowski
Jeff Fox

Debbie Cohn

The good advice, kind words, insight, and support provided over the years by these fine folks, and others, have made this thesis possible.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research (Contract Nos. N00014-75-C-0661 and N00014-83-K-0125).

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

Simulation plays an important role in the design of integrated circuits. Using simulation, a designer can determine both the functionality and the performance of a design before the expensive and time-consuming step of manufacture. The ability to discover errors early in the design cycle is especially important for MOS circuits, where recent advances in manufacturing technology permit the designer to build a single circuit that is an order of magnitude larger than ever before possible. This thesis presents three new algorithms designed specifically for the simulation of large digital MOS circuits.

Today's MOS circuits offer special challenges to a simulation program, challenges that are not met very well by current simulators. New integrated circuits can incorporate hundreds of thousands of transistors; the sheer number of transistors dictates that a simulation algorithm use simple, computationally efficient transistor models. In addition, designers take advantage of the symmetry of the MOS transistor to build circuit configurations with behavior beyond the ken of traditional logic simulators. The new simulators introduced here are designed to meet these challenges.

1.1. Overview of the thesis

To use a simulator, the designer enters a design into the computer, typically in the form of a list of circuit components where each component connects to one or more *nodes*. A node serves as a wire, transmitting the output of one circuit component to other components connected to the same node. The designer then specifies the voltages or logic levels of particular nodes, and calls upon the simulator to predict the voltages or logic levels of other nodes in the circuit. The simulator bases its predictions on models describing the operation of the components; a simulator is characterized by the types of component models it employs. Two of the more popular approaches are:

- component models based on the actual physics of the component; for example, a transistor model that relates current flow through the transistor to the terminal voltages, device topology, and manufacturing parameters of the actual device.
- component models based on a description of the logic operation performed by the component, *e.g.*, NAND and NOR gates.

The first type of model is found in circuit analysis programs such as ASTAP [Weeks73] or SPICE [Nagel75] which try to predict the actual behavior of each component with a high degree of accuracy. Current circuit analysis programs do the job well, perhaps too well; at no small cost, they provide a wealth of detail, at sub-nanosecond resolution, about the voltage of each node and the amount of current through each device. (For example, a properly calibrated circuit analysis program is able to predict, within a few per cent, the amount of current that flows through an actual transistor.) This level of detail would swamp the designer if collected for the entire circuit while simulating, say, a microprocessor. Fortunately, the designer is spared this fate, since the computational cost of circuit analysis restricts its applicability to circuits with no more than a few hundred devices.

One solution to the problem of simulator performance is to adopt a simpler component model, such as the gate-level model introduced above. This approach works well when dealing with implementation technologies that adhere to gate-level semantics (*e.g.*, bipolar gate arrays). However, MOS circuits contain bidirectional switching elements that cannot be modeled by the simple composition of Boolean gates. Since many of the circuit techniques that make MOS attractive for LSI and VLSI applications take advantage of this non-gate-like behavior, it is important to model such circuits accurately.

This thesis explores the possibility of providing the essential information (functionality and comparative timing) for large digital circuits by using models that bridge the gap between the gate-level and detailed models discussed above. The goals to be met by these new models are summarized in the following list:

(i) The underlying model must be computationally tractable for large circuits. The empirical nature of the verification provided by simulation suggests that it must be applied extensively if the results are to be useful; timely simulation encourages this.

(ii) Transistor-level simulation is necessary to accurately model the circuit structures found in MOS designs. This allows the designer to simulate what was designed — an advantage, since requiring separate specification of a design for simulation purposes only introduces another opportunity for error.

(iii) The results must be correct, or at least conservative; a misleading simulation that results in unfounded confidence in a design is probably worse than no simulation at all. Here, we must trade off the conflicting desires of accuracy and efficiency.

Two models are examined in detail by the thesis:

- a *linear model* in which a transistor is modeled by a resistance in series with a voltage-controlled switch. The state of the switch is controlled by the voltage of transistor's gate node.
- a *switch model*, similar to the linear model, except that resistance values are limited to one of two quantities: 0 for for n- and p-channel devices, and 1 for depletion devices.

MOS circuits are easily transformed to use either model, as illustrated by the following figure.

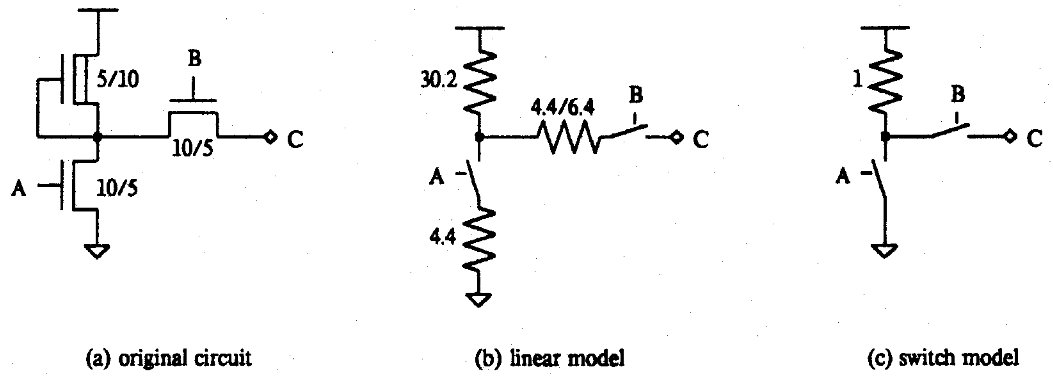


Figure 1.1. *Two approaches to modeling a simple MOS circuit*

The linear model forms the basis for the RSIM simulator. In RSIM, networks of transistors and electrical nodes form an R-C network (R for the transistors, C for the interconnect and gate capacitance); the network's behavior under different inputs is calculated by a selective-trace (event-driven) algorithm. The comparatively fast "pseudo circuit analysis" that is possible with the linear model allows the designer to determine both the functional and approximate timing characteristics of a network. RSIM goes a long way towards meeting the three goals outlined above. The algorithm employed to estimate the behavior of a linear network is much faster than a typical circuit analysis program. Resistors are inherently bidirectional; the network analysis makes no *a priori* assumptions about the direction of current flow through each resistor. Finally, the results are at least qualitatively correct and, in general, conservative — in some cases more conservative than designers themselves might like. With the appropriate choice of model parameters, the results can even be quantitatively useful.

The switch model is a simplification of the linear model that is useful when only a circuit's functionality is of interest (*i.e.*, no information on performance is wanted). Like a traditional gate-level simulator, a switch-level simulator bases its predictions on an abstraction of the actual circuit, but the switch model is able to handle the bidirectional nature of MOS transistors much more successfully than a gate-level model. The switch model is incorporated by ESIM, a simulator that has seen extensive use in the last few years.

Certainly a major goal of RSIM and ESIM is to provide a fast, useful simulation of MOS circuits, but the story does not end there. Another motivation for new simulation algorithms is the changing nature of the design community. In order to cope with the increasing complexity of integrated circuit design, new design methodologies have developed (*e.g.*, [Mead80]) that impose constraints on the way circuits are constructed. One can no longer afford to hand-craft each transistor, so rules of thumb are created to aid in the choice of transistor sizes. Clever circuit configurations are avoided in favor of circuits composed under the guidance of composition rules (*e.g.*, [Bell81]) that rule out arbitrary circuits and the obscure electrical behavior they imply.†

These new design methodologies have opened up the field of LSI design to a new breed of "Mead and Conway" designer, *i.e.*, a designer who is a sophisticated architect, but who is not a specialist in LSI technology. An important aspect of the simulators described in this thesis is that their underlying models are easily understood by this new breed of designer. The abstractions embodied by the simulators are faithful enought to the actual electrical behavior of a circuit that the achievement of a successful simulation run indicates freedom from a large class of potential failure modes. If a simulation does point out an error, it does so in a manner that leads even the novice designer to a good understanding of the circuit as actually designed and the ways in which it might differ from the intended design.

However, the simulators are based on *models* of actual behavior. As with any model,

†State-of-the-art designs intentionally exploit the "obscure" behavior of certain circuits (*e.g.*, sense amplifiers), often to considerable commercial advantage. RSIM and its relatives are not as useful for this type of design as conventional circuit analysis programs. But the professionals engaged in such well-focused designs are not the audience addressed by Mead and Conway (and RSIM).

discrepancies are likely to exist between the model predictions and the actual behavior of a circuit. The tools described here attempt to be conservative, *i.e.*, to give pessimistic predictions, but this cannot be guaranteed. Thus, it is important that the designer become acquainted with the inner workings of the models and their shortcomings. The tools perform a calculation one could do by hand (only faster and with greater accuracy and consistency) — they should *not* be treated as black boxes. The models presented here are simple enough to enable any designer to gain the necessary understanding.

A final motivation for new simulation technology is the desire to improve simulator performance. It seems that digital computers ought to be well suited for the simulation of digital logic. Unfortunately, current simulation schemes involve several layers of interpretation (*e.g.*, command interpretation, access to the network data base, model evaluation), and their performance suffers as a result. Happily, much of this overhead can be eliminated through the application of traditional compilation techniques. This is the theme of the final section of the thesis, and the motivation for the development of CSIM, a combination compiler/simulator. CSIM compiles a network into a simulation subroutine; the subroutine contains code to compute the new value of each node from its old value and the values of other nodes in the network. The compilation is particularly easy when the node is the output of a logic gate, and the work presented here extends the compilation technique to any node in a MOS circuit. Simulating the network entails executing the subroutine repeatedly until no nodes change value. If the circuit is very active, *i.e.*, if many nodes change value each time the network is simulated, the simulation subroutine computes new node values many times faster than the corresponding event-driven simulation. There has been much interest recently in special purpose hardware for simulation [Pfister82, Zycad83]. It may be that such developments are premature, and that substantially better simulation performance can still be obtained from general-purpose computers.

The relationship among RSIM, ESIM, and CSIM is illustrated in the table below.

	RSIM	ESIM	CSIM
node values	logic-level (from voltages)	logic-level	logic-level
model level	transistor	transistor	node equations
components	resistors & capacitors	switches & capacitors	equations (from switches)
scheduling	event-driven	event-driven	compile-time
relative speed	1	.5 – 3	.1 – 100

No one simulator has a speed advantage, for reasons explained in subsequent chapters. It is not unusual to use all three simulators during the course of a design, since each brings out a different aspect of a circuit's behavior. ESIM is often used during the early stages of a design when the designer is fleshing out the logic. RSIM is used to determine which portions of the design are in need of a careful performance analysis; usually the performance of most of the circuit can be debugged with the level of detail provided by RSIM. Finally, CSIM is useful for long simulation runs intended to verify the functionality of the design through extensive diagnostics.

This thesis presents the new models and their accompanying simulators in detail, exploring the ramifications of each model and discussing the accuracy and usefulness of their predictions. The next section gives a brief outline of the remaining chapters.

1.2. Outline of the remaining chapters

The thesis has three main parts. The first part focuses on the linear model and the RSIM simulator.

Chapter 2	description of the switch/resistor transistor model incorporated by RSIM; outline of the method for calculating a node's value using the linear transistor model; propagation of changes through the network; choosing model parameters; analysis of sample circuits using linear model.
Chapter 3	justification of the linear model by analysis of true behavior of MOS logic gates; comparison of actual voltages and propagation delays with RSIM's predictions; proposal for modifications to the model based on insight gained during analysis; analysis of sample circuits using updated model.
Chapter 4	details of converting the linear model into a workable simulation algorithm; optimizations for improving simulator performance; mechanisms for controlling the voltage and transition time predictions for specific nodes; review of the successes and failures of the linear model.

The second part (Chapter 5) presents the switch-level model. The chapter begins with a discussion of the representation of node values and explains why many extant simulators adopt a representation that leads to unnecessary difficulties. Next, two switch-level algorithms are presented. The first is a straightforward adaptation of the RSIM algorithm, replacing its resistance computations with simpler ones that reflect the resistance value constraints of the switch model. The second algorithm is based on an entirely different approach; each computation handles a single transistor and uses only local information (the type of the transistor and the states of its terminal nodes). The computation is easy to understand and appeals to our intuition about the way transistors really operate. The simulation proceeds by repeatedly computing new node values for the source and drain nodes of individual transistors, choosing the transistors in any convenient order. The simulation is complete when no further changes in the network state are possible. The termination of this relaxation algorithm is proved, and the final network state is shown to be independent of the order in which the individual computations are performed. The second algorithm is well suited for implementation on the new parallel architectures just now becoming available; the approach discussed here is a first cut at designing simulation algorithms tailored for use on parallel engines.

The third part (Chapter 6) investigates the possibility of using various compilation schemes to improve the performance of the switch-level simulator. A technique is proposed for constructing a set of equations for each node in the network. These equations relate the new value of a node to its current value and the values of other nodes in the network. The network can be simulated by evaluating each node's equations in turn; several ways of ordering the nodes for evaluation are discussed. The section concludes with several examples of simulation routines that were compiled directly from the network data base. When executed, these routines result in a simulation several orders of magnitude faster than otherwise possible.

The thesis concludes with a discussion of other work in the area of simulation and its relationship to the ideas presented here.

CHAPTER TWO

A Linear Network Model for MOS Simulation

The electrical model described in this chapter can be used as the basis for a logic-level simulation of a network of MOS transistors. Other models are of course possible, ranging in accuracy and detail from circuit analysis to high-level functional simulation. While the chosen model does not encompass many of the operational details of real MOS networks (most notably, detailed transistor modeling) it is adequate to efficiently determine the basic functionality and the approximate timing characteristics of a network. Short circuits, charge sharing, nodes with multiple drivers, bidirectional "pass" transistors, and so on are modeled correctly.

The first section describes the switch/resistor transistor model incorporated by RSIM. Using this model, a MOS network is simulated as a resistor network where each node's value is determined by the resistance of its connections to various inputs. The second section outlines the method for calculating the value of each node. This is followed by an explanation of the use of component models to predict the propagation of new input values through a network. The fourth section discusses techniques for choosing model parameters and compares RSIM's predictions with those of a circuit analysis program. The chapter concludes with a summary of the model's ingredients.

2.1. RSIM's transistor model

The transistor model in RSIM can be quite simple since it is only used to predict the final logic state of a node and the length of time each state transition takes. As an example of how the model works, consider a simple inverter: one can think of the effective resistance of its component devices at any moment as

$$R_{eff:pullup} = \frac{i_{ds:pullup}}{v_{ds:pullup}} \qquad R_{eff:pulldown} = \frac{i_{ds:pulldown}}{v_{ds:pulldown}} \tag{2.1}$$

The following figure shows the actual effective resistance of an inverter's pullup and pulldown as a function of the inverter's output voltage (assuming no load current).

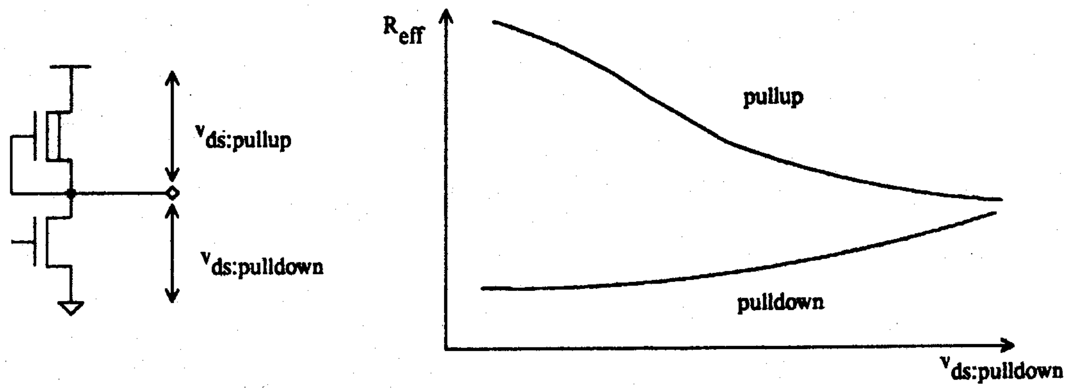


Figure 2.1. *Effective device resistances in an inverter*

Although the effective resistances of the transistors change as their terminal voltages vary, it might be possible to use "average channel resistances" to characterize the transistors' behavior.

The other salient feature of a transistor's operation is its switch-like behavior. With certain voltages on a transistor's terminal nodes, it makes no connection at all between its source and drain terminals — the transistor is "off". As the relative terminal voltages change, the transistor turns "on", conducting current between its source and drain terminals. As illustrated in the previous figure, the transistor is more "on" at some times than others, but the distinction among the different "on" states can be ignored for simplicity.

There are three basic types of transistor switches found in MOS circuits:

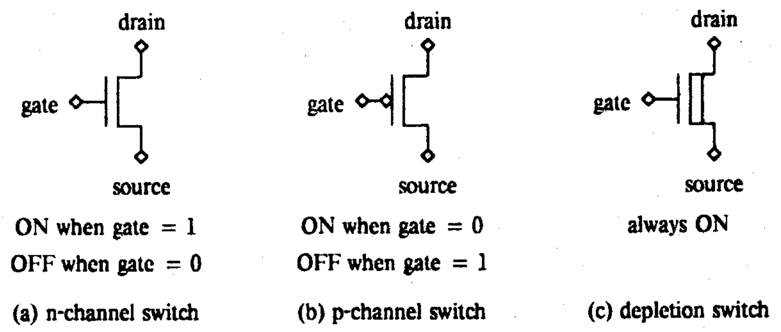


Figure 2.2. *Three types of MOS transistor switches*

The difference between n-channel and p-channel switches is the logic level which turns on the switch. The depletion switch is always on; it is usually connected to VDD in a way that provides a source of current to keep its output node charged high. More precise distinctions between the switch types, and the need for a depletion device (and why an ordinary switch does not suffice) are discussed in Chapter 3.

One can build on the observations made above to construct a linear transistor model for RSIM:

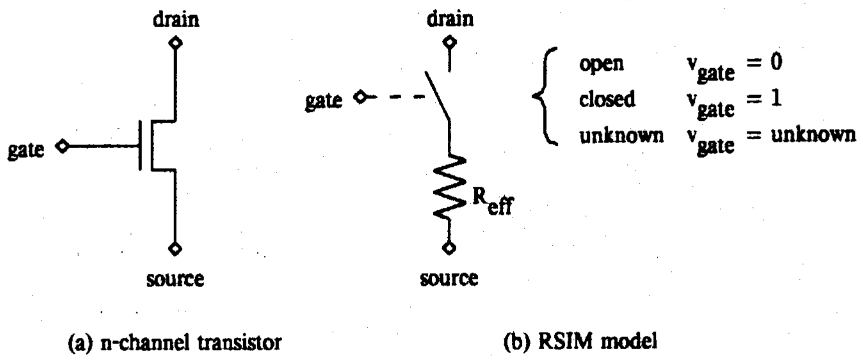
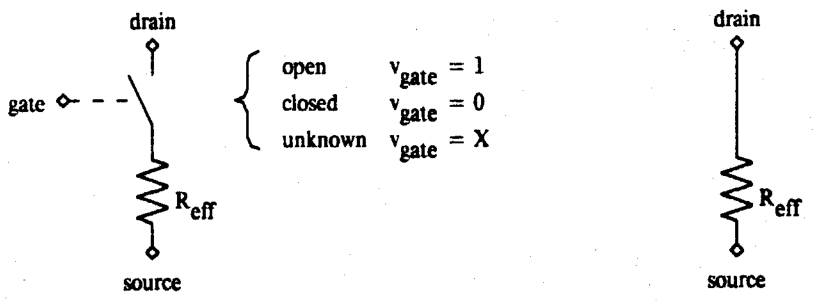


Figure 2.3. *RSIM model for n-channel transistor*

It is easy to tabulate the sort of connection that exists between the source and drain terminals as a function of the gate voltage:

$$R_{ds} = \begin{cases} R_{eff} & \textit{switch closed} \quad (v_{gate}=1) \\ \infty & \textit{switch open} \quad (v_{gate}=0) \\ [R_{eff},\infty] & \textit{switch unknown} \quad (v_{gate}=X) \end{cases} \tag{2.2}$$

Note that uncertainty about the state of the switch leads naturally to an interval describing the resistance of the source-drain connection. In fact, all the network calculations use interval arithmetic, and the bounds of the resulting intervals are used when converting voltages to logic states, etc.; no other mechanisms are needed to deal successfully with X states in the network. Models for other types of transistors differ in the way the position of the switch is determined from v_{gate}:



(a) p-channel transistor model (b) depletion transistor model

Figure 2.4. *RSIM models for p-channel and depletion transistors*

The effective resistance R_{eff} is determined separately for each transistor and depends on

width, length	dimensions of the active transistor area. Various non-linear effects make R_{eff} a more complicated function of the transistor geometry than just length/width.
type	Most MOS circuits contain more than one type of transistor. The different types are distinguished by different values for their threshold voltage. Since the current conducted by a transistor is a function of its threshold voltage and hence its type, the modeling resistance also depends on the transistor type.
context	Accuracy in choosing the effective resistance can be improved by distinguishing several contexts in which a transistor may appear: for example, an enhancement transistor can be used as a pulldown or source-follower in addition to its default pass gate configuration. Surprisingly few contexts need to be recognized to encompass a large portion of digital MOS designs.

The determination of R_{eff} is made once for each transistor and does not depend on any dynamic properties of the circuit to be simulated. During simulation the only device information RSIM uses about a transistor is its effective resistance.

Actually RSIM uses not one, but three effective resistances for each transistor. To understand why, recall that RSIM tries to predict the transition time and final voltage for a node, as shown in the following figure.

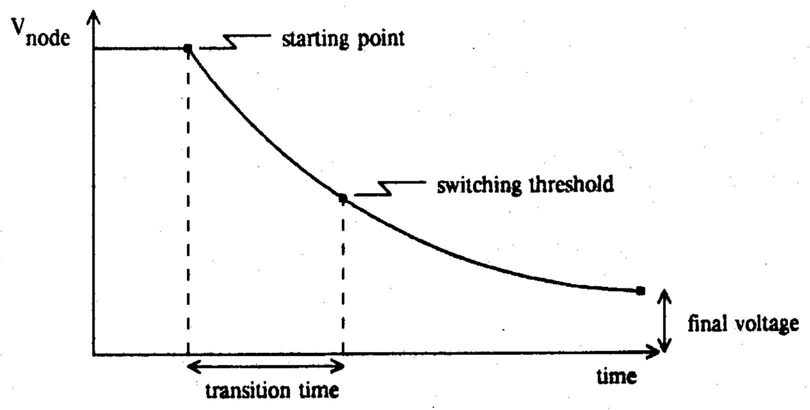


Figure 2.5. *R_{eff} is used to predict transition time and final voltage*

One would like to calibrate the model to give accurate predictions for both quantities, but that is impossible with a single set of resistances. To solve this problem, RSIM uses three resistances for each transistor:

R_{static}	when calculating the final voltage.
R_{dynlow}	when calculating the transition time for high-to-low transitions.
$R_{dynhigh}$	when calculating the transition time for low-to-high transitions.

Two "dynamic" resistances are used so that the asymmetric behavior of pass devices can be accurately predicted. Computations involving R_{eff} are triplicated, one for each of the three actual resistances, so subsequent calculations can use the appropriate value.

2.2. RSIM's node model

Voltages in this model are quantized into one of three values; this corresponds to our intuition for digital logic and greatly simplifies the simulation calculations. If all node voltages are normalized to fall in the range [0, 1], then the possible quantized values are

0	logic low — voltages in the range $[0, v_{low}]$;
1	logic high — voltages in the range $[v_{high}, 1]$;
X	intermediate voltages, $[v_{low}, v_{high}]$, or unknown voltages, [0, 1] — to be conservative X is always interpreted as representing the larger interval;

where v_{low} and v_{high} are the predetermined logic thresholds.

How is the value of a node determined? Using the transistor model described in the previous section, the original network is transformed into a network of resistors (formerly transistors) and capacitors (formerly nodes). If a node is not connected to any input, it is said to be *charged* with a logic state determined by the state of the last driven node it was connected to. If two or more charged nodes in different logic states are connected then *charge sharing* occurs. In this case, all the connected nodes reach the same logic state; this state is determined by the relative capacitances and initial logic states of the nodes in the stage. For example, if a large (high capacitance) node such as a data bus were connected by a pass transistor to a small node such as the input to a register cell, then the small node would "share" the charge of the large node as its final value regardless of the charge it had initially. Even nodes that ultimately have a connection to an input participate in charge sharing; the extent of their participation is governed by the relative sizes of the charge-sharing time constant and the time constant associated with the input connection.

Electrically connected nodes form natural groupings, called *stages*, bordered by input nodes (usually VDD and GND). If nodes in a stage are allowed to share charge, all will reach the same voltage, V_{share}, given by

$$V_{share:min} = \frac{\sum_{1\ nodes} c_i}{\sum_{all\ nodes} c_i} \qquad V_{share:max} = \frac{\sum_{1\ nodes} c_i + \sum_{X\ nodes} c_i}{\sum_{all\ nodes} c_i} \tag{2.3}$$

where the sums are over nodes in the current stage. Since nodes at logic state X contribute an undetermined amount of charge to the result, V_{share} is an interval whose bounds represent the worst case assumptions about the actual values of X nodes. These bounds are compared with the logic thresholds when calculating the charge-sharing value:

$$\textit{Charge-sharing value} = \begin{cases} 0 & V_{share:max} \leq v_{low} \\ 1 & V_{share:min} \geq v_{high} \\ X & \textit{otherwise} \end{cases} \tag{2.4}$$

This calculation is not strictly accurate when the stage contains transistors with gates of X. Such transistors might not make any connection at all; invalidating the various sums in equation 2.3. An alternative charge-sharing calculation that addresses this problem is discussed in Section 4.1.1.

When one accounts for the resistance between nodes, it is difficult to calculate transition times for any nodes that change value because of charge sharing. RSIM simply schedules any charge-sharing transitions so they happen immediately. A more reasonable time constant might be $(\sum_i R_i)C_{eff}$ where the first term is the sum of all the resistances in the stage and

$$C_{eff} = \begin{cases} \sum\limits_{0 \text{ and } X \text{ nodes}} c_i & \textit{Charge-sharing value} = 1 \\ \sum\limits_{1 \text{ and } X \text{ nodes}} c_i & \textit{Charge-sharing value} = 0 \\ 0 & \textit{otherwise} \end{cases} \qquad (2.5)$$

is the amount of capacitance in the stage that needs to be charged/discharged to reach the charge-sharing value. This time constant is surely an upper bound on the time of any transition in the stage. Note that transitions to X still happen immediately, a conservative assumption.

If a stage is connected to one or more inputs, the inputs determine the final voltage of each node in the stage. The effect of inputs on a particular node is characterized by the Thevenin equivalent for the stage (including the inputs at the boundary), regarding the given node as the output:

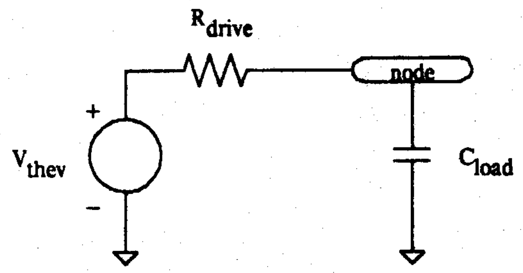


Figure 2.6. *Equivalent circuit for a network node*

V_{thev} a voltage interval $[V_-, V_+]$ in the range [0, 1] specifying the possible voltages the output node may have. This value is calculated using each transistor's R_{static} resistance.

R_{drive} a resistance interval $[R_-, R_+]$ in the range $[0, \infty]$. Three versions of this value are calculated; $R_{drive:low}$, using R_{dynlow} for each transistor; $R_{drive:high}$, using $R_{dynhigh}$; and $R_{drive:x}$ (see section 4.1.2). The appropriate version is chosen depending on the final voltage predicted by V_{thev}.

V_{thev} and R_{drive} are generally intervals, since the effective transistor resistances from which they are derived might themselves lie in an interval. Chapter 4 describes how V_{thev}, C_{load}, and R_{drive} are estimated for nodes in actual networks.

It is sometimes useful to categorize a node according to its equivalent R_{drive}, *i.e.*, how it affects neighboring nodes to which it becomes connected by conducting transistors:

input ($R_{drive} = 0$). Node is a designated input node (*e.g.*, VDD or GND). The value of input nodes can only be changed by explicit simulator commands; the assumption is that inputs supply enough current to be unaffected by connections (possibly shorts to other inputs) made by transistors.

driven ($R_{drive} < \infty$). Node is part of a voltage divider between two inputs, *i.e.*, it is connected by transistors to other driven or input nodes. Driven nodes can affect the value of charged nodes without being affected themselves, but may be forced to an X state if shorted to a driven or input node that has a different logic level.

charged ($R_{drive} = \infty$). Node is connected, if at all, only to other charged nodes. Until reconnected to some other part of the network, charged nodes maintain their current logic state indefinitely (charge storage with no decay).

If R_{drive} is infinite, equation 2.4 predicts the correct final value for the node and no further work is needed. If $R_{drive} < \infty$, and the node is not an input, the final state of a driven node is calculated from the V_{thev} interval $[V_-, V_+]$:

$$\textit{Final value} = \begin{cases} 0 & V_+ \leq v_{low} \\ 1 & V_- \geq v_{high} \\ X & \textit{otherwise} \end{cases} \tag{2.6}$$

As an example, consider several different states of a NOR gate:

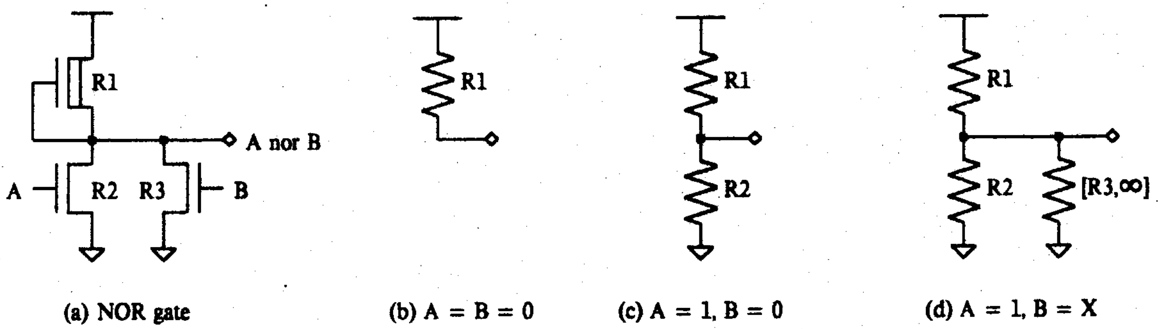


Figure 2.7. *Equivalent circuits for a NOR gate with different inputs*

$$V_{thev} = \begin{cases} 1 & \textit{figure}\,2.7(b) \\ \dfrac{R_2}{R_1 + R_2} & \textit{figure}\,2.7(c) \\ \left[\dfrac{R_2 \,||\, R_3}{R_1 + (R_2 \,||\, R_3)}, \dfrac{R_2}{R_1 + R_2}\right] & \textit{figure}\,2.7(d) \end{cases} \tag{2.7}$$

If the final value of a node differs from its charge-sharing value, then the appropriate event is scheduled $R_{eff} C_{eff}$ seconds in the future, where

$$R_{eff} = \begin{cases} R_{drive:high} & \textit{final value} = 1 \\ R_{drive:low} & \textit{final value} = 0 \\ R_{drive:x} & \textit{final value} = X \end{cases} \tag{2.8}$$

$$C_{eff} = \begin{cases} \sum\limits_{0 \text{ and } X \text{ nodes}} c_i & \textit{final value} = 1 \\ \sum\limits_{1 \text{ and } X \text{ nodes}} c_i & \textit{final value} = 0 \\ \sum\limits_{0 \text{ and } 1 \text{ nodes}} c_i & \textit{final value} = X \end{cases} \tag{2.9}$$

where the sums are computed for nodes in the current stage. Note that transitions to X are not immediate, but have a time constant related to the fastest transition the node can make. This means that a momentary short-circuit, such as that shown in the following figure, does not necessarily cause a node to become X; what happens depends on the relative sizes of the various time constants.

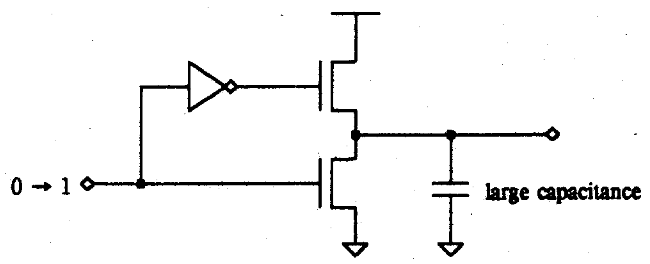


Figure 2.8. *A momentary short-circuit does not necessarily cause an X value*

If the delay through the inverter is small compared to the time constant of the output node, no X transition will be processed for the output node (one is scheduled, but is aborted when the pullup turns off).

To better understand the interaction between the charge-sharing and final-value calculations, consider the following example:

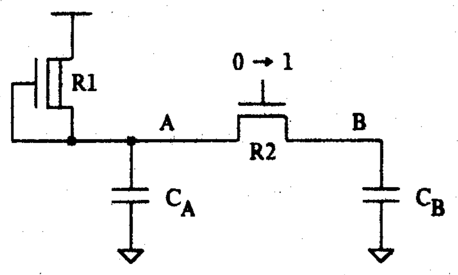


Figure 2.9. *Sample circuit for charge-sharing and final-value calculation*

Assuming that C_B is initially charged low and that charge sharing happens immediately (an assumption RSIM makes), there are several different scenarios:

$C_A \ll C_B$ node A goes low immediately because of charge sharing with B. Then, both nodes are driven high by the pullup — node A at time $R_1(C_A + C_B)$, and node B at time $(R_1 + R_2)(C_A + C_B)$.

$C_A >> C_B$	node B goes high immediately because of charge sharing with A; the pullup has nothing to contribute.
$C_A \approx C_B$	both A and B go to X immediately and are then pulled up with the same time constants as for $C_A << C_B$.

If R_2 is reasonably smaller than R_1, then the assumption that charge sharing happens quickly is valid, and these scenarios are satisfactory. As R_2 approaches R_1 in value, the time constants associated with charge sharing approach those of the pullup, and the assumption of immediate charge sharing is a relatively poor one.† Augmenting the charge sharing calculation as described in equation 2.5 would improve the prediction in this case.

In summary, calculating a node's value involves two separate computations, each of which can generate a new event:

(1) a charge-sharing event describing an immediate change in the node's state caused by the redistribution of charge among the capacitors for nodes in the current stage. This type of event is generated when two stages are merged (*i.e.*, a transistor turned on).

(2) a final-value event describing what the final, driven state of the node will be. This type of event is generated when $R_{drive} < \infty$.

Chapter 4 describes the way these two events are reconciled with each other and with pending events to produce a final set of transitions for a node.

2.3. RSIM's network model

The networks‡ simulated by RSIM are made up of two basic components:

(i) electrical nodes which serve as wires. Each node has a capacitance that is the sum of two contributions: (1) capacitance between other layers and the conducting layers that make up the node; and (2) capacitance from the gate junctions formed by the node.

(ii) three-terminal transistors (mosfets) which act as switches. Each transistor conditionally connects two nodes (called the source and drain of the transistor) depending on the voltage of the third node (called the gate of the transistor).

Some nodes (*e.g.*, VDD and GND) are designated as inputs that supply the current needed to change the

†This illustrates the asymmetry between the timing of transitions due to charge sharing and those due to the final value calculation, *i.e.*, R2 affects only the final value transition. This anomaly could be exploited to produce rather bizarre predictions, *e.g.*, a node changes faster if it is connected to a capacitor than if it is connected to an input! As a practical matter, circuit performance seldom depends on the timing of charge-sharing transitions, and these anomalies are not significant.

‡Networks can be entered as schematics [Terman82] or extracted from layout information [Baker80]. The latter approach provides fairly accurate estimates of the capacitance of each node.

voltage of a node by charging/discharging the node's capacitor. As the voltage of a node changes, switches controlled by the node open or close, making connections that cause the voltages of other nodes to change. It is RSIM's job to predict the dynamic behavior of a network of nodes and switches, estimating the voltage of each node, the state of each switch, and the charge/discharge rate when a node changes value. From the designer's point of view, this translates into knowledge about the logic level of each node and the transition time associated with each change of logic level.

It is easy to build switch configurations that compute simple logic functions of node values. For example:

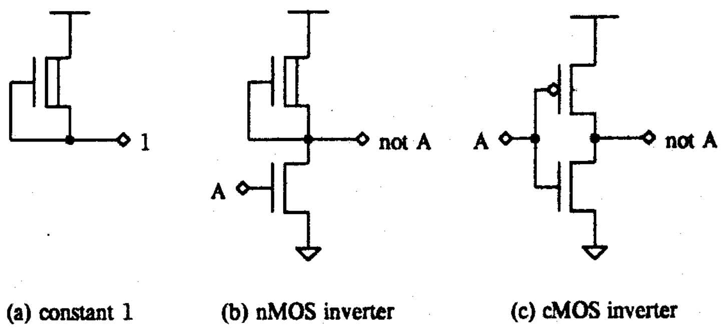


Figure 2.10. *Examples of switch configurations that perform logic operations*

The output node in figure 2.10(a) is connected to a depletion switch configured as a current source; its value is always a logic high. Such circuits are called pullups because their output nodes are always "pulled-up" to logic high. In figure 2.10(b) a "pulldown" switch has been added, controlled by node A. The pulldown is sized so that, when it is on, it conducts more current than the pullup supplies. When A is 1, the output node is "pulled-down" to 0. Of course, when A is 0, the pulldown is off and the pullup ensures that the output is 1; the net result is an inverter circuit. Figure 2.10(c) is an inverter constructed from one p-channel and one n-channel device. Typically, the manufacturing process can provide either p-channel devices or depletion devices, but not both, in the same circuit. More complicated logic circuits are constructed using series and parallel switch configurations.

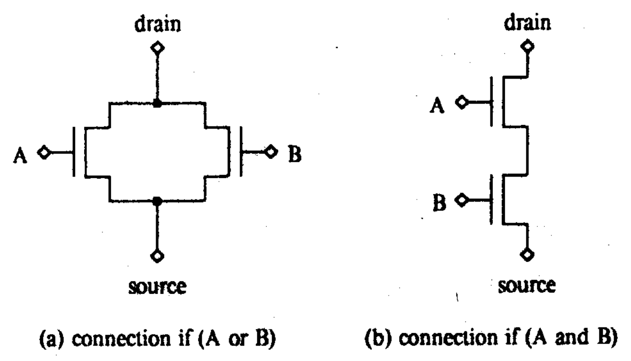


Figure 2.11. *Logic functions associated with series and parallel configurations*

If the two-switch circuits shown above replace the pulldown in figure 2.2(b), the result is a two-input NOR or NAND gate.

In all the circuits presented so far, the inputs are electrically isolated from the outputs, *i.e.*, if the output signal is corrupted somehow — by a short circuit, for example — the input signals are unaffected. The isolation provided by the gate connection leads to a natural decomposition of the network into *stages* made up of nodes and transistors. Nodes belong to different stages only if they are guaranteed to be electrically isolated. For example, in the following circuit, nodes A, B, C, and D are all isolated from one another. Node E is not isolated from D, so it is in the same stage as D.

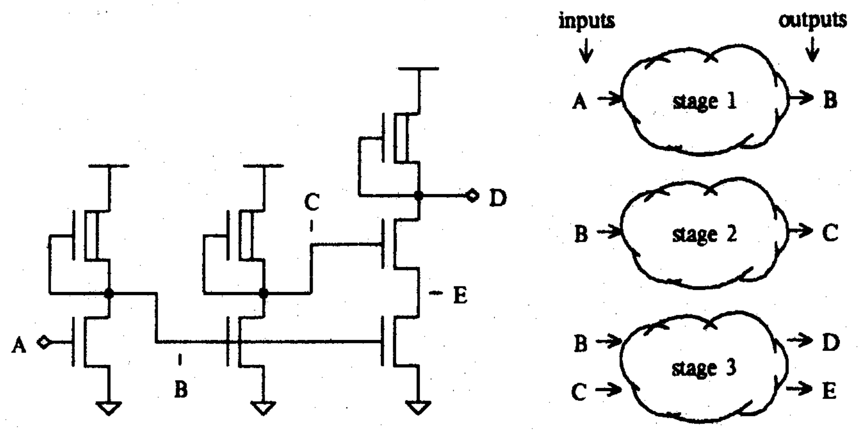


Figure 2.12. *Simple circuit that has three stages*

Note that VDD and GND (and, in fact, any input) are not treated as nodes in the ordinary sense when checking to see if two nodes belong to the same stage. For example, node B is not considered to connect to node C by a path involving GND and two of the pulldown transistors. Given a particular node, a tree walk of the network is performed to find all other nodes in the stage. The tree walk first locates all "on" switches which have a source/drain connection to the original node. Nodes connected to the drain/source of those switches are part of the same stage as the original node. The tree walk continues until it locates all nodes that can be reached from the original node by a path of "on" switches; this set of connected nodes and the "on" transistors that form the connections make up a single stage. Note that the decomposition of the network into stages is a dynamic process, *i.e.*, one that depends on the node values of the network.† For example, the following circuit can be decomposed into 2, 3 or 4 stages depending on the value of nodes A and B.

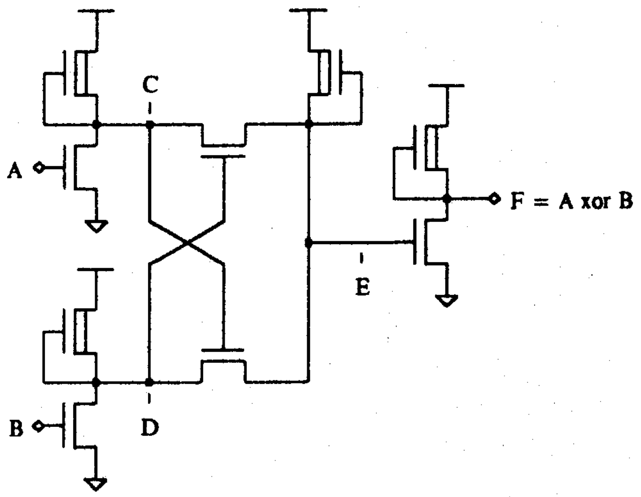


Figure 2.13. *Circuit with multiple decompositions*

Node F is always in a separate stage. If A=0 and B=0, then C, D, and E all form a single stage; if A=1 and B=0, then D is isolated from C and E; and so on.

When RSIM simulates a network, it does its analysis stage by stage. Since the values of nodes in a stage are closely related (the nodes are shorted together), it makes sense to calculate all the values at the same time. By the same reasoning, all the transistors and nodes that influence the value of a particular node are in the same stage as that node. Stages are the analogs of gates in a gate-level simulator. In a gate network, each node's value is determined by a single gate, and the output of a gate is electrically isolated from the inputs; the gate is the ideal unit of analysis. In MOS networks with bidirectional devices, the traditional gate model is not adequate; hence the motivation for stages.

†This differs from the notion of "transistor group" introduced by [Bryant81]. A transistor group contains all nodes that *might* become connected, *i.e.*, a stage with all switches considered to be conducting. Transistor groups can be quite large — for example, in circuits with barrel shifters that potentially short together all bits in a data path — whereas stages are usually quite small.

A simulation step starts when the designer changes the value of an input. (By definition, any node given a value by the designer is treated as an input node by the simulator.) The value of the input influences other pieces of the network in two ways:

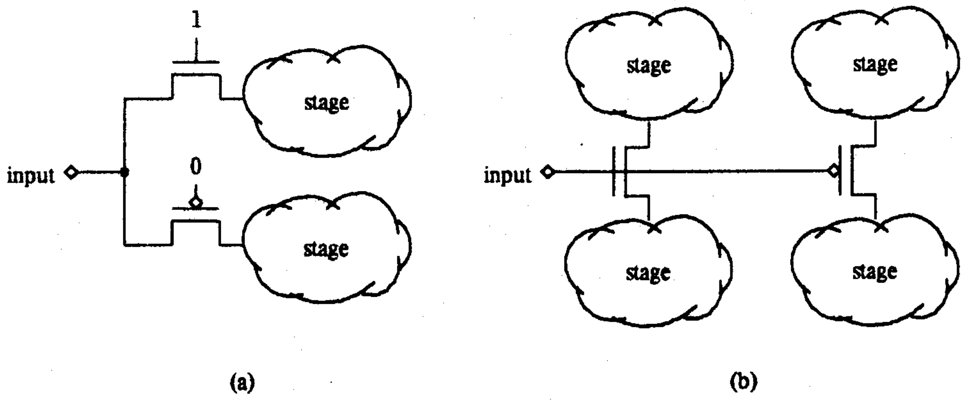


Figure 2.14. *Two ways in which an input affects a network*

The simulator first recalculates the values of nodes in stages connected to the input by the source/drain connections of conducting switches (figure 2.14(a)). Then, for each switch controlled by the input, stages on each side of the switch are analyzed (figure 2.14(b)). If the switch becomes conducting because of the new input value, the pieces of the network on either side form one large stage. If the switch just turned off, it partitions what was previously one large stage into two smaller stages.

If a node changes value as a result of analyzing a stage, RSIM calculates the transition time by estimating the length of time required to charge/discharge the node's capacitance. The name of the node, its new value, and the estimated time when the transition to the new value occurs are all remembered as an *event*. The simulator maintains a list of pending events, keeping the list sorted by time, with the earliest event first.

When processing new input values causes a node to change value, a new event is generated and saved on the event list. After all inputs have been processed, the simulator processes events, starting with the first element of the event list. For each event, the specified node is assigned its new value. Then, any stages affected by this change (as shown in figure 2.14(b)) are analyzed, possibly generating new events, which are then added to the event list. The simulator continues processing events until the event list is empty. The network is said to have "settled" at this point, and the new input values have been completely propagated through the network.

Note that if no nodes change value when a stage is analyzed, no new events are generated. Portions of the network that remain quiescent are not analyzed, since the simulator only analyzes stages affected by inputs or by nodes on the event list. By limiting simulation effort to the changing portions of the network, the event list mechanism enables the simulator to handle large circuits. The amount of computation required for a simulation step is proportional to the amount of circuit activity, not the size of the circuit.

To get a better feeling for the way a change propagates through a network, consider the following simulation of the XOR circuit presented in figure 2.13. Nodes A and B are inputs; values for the other nodes are determined by the simulator.

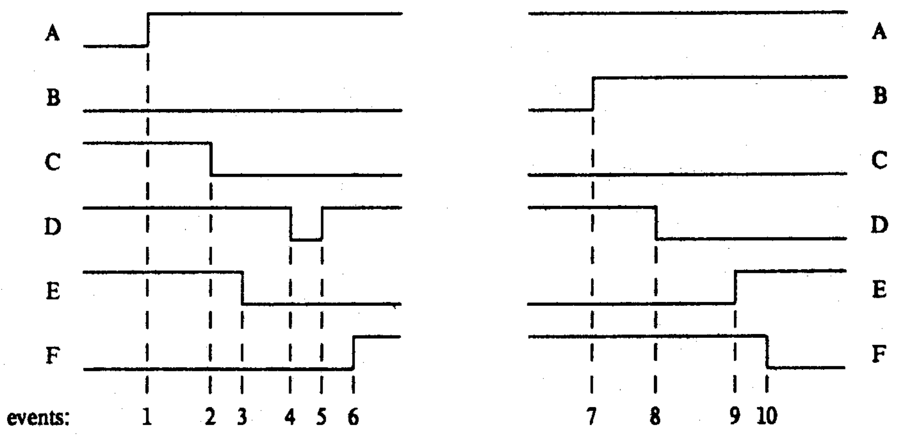


Figure 2.15. *Waveforms for simulation example*

Event #1. Node A is set to 1 by the user. The simulator recalculates all stages affected by A, in this case, the stage containing nodes C, D, and E (which form one stage because C and D are 1).

All three nodes are pulled down by the switch controlled by A, so events #2, #3, and #4 are scheduled to set C, D, and E to 0. Note that the simulator calculates a different transition time for each node. C changes most quickly since it is connected directly to the pulldown. D is the slowest since it discharges through the two pass devices connecting it to the pulldown.

Event #2. C changes from 1 to 0, causing the stages containing D and E to be analyzed.

At the time event #2 is processed, nodes D and E are still 1, although they both have events pending for transitions to 0. When node C goes low, it partitions what was once one large stage into two stages — one containing only D, the other containing both C and E. Analysis of the stage containing D shows that D is no longer pulled down, invalidating the upcoming transition. The simulator has several choices:

(1) Notice that D is currently 1, so just remove the pending event for D. This results in D never changing value. This is not a bad prediction if D is scheduled to change substantially after C.

(2) Schedule another event (#5) for node D, which changes its value back to 1; set the event time so that #5 happens after #4. This choice is best if C and D are both scheduled to become 0 in close succession.

(3) Remove D's pending event as in (1), but report a glitch (an aborted transition) to the user [Thompson74]; a sort of compromise between (1) and (2). Some simulators only report glitches if the aborted event has been pending "long enough" [Nahm80].

(4) Schedule another event as in (2) that changes D's value back to 1; also change the pending event to be a transition to X, or, alternatively, remove the pending event and schedule an immediate transition to X.

As one can see, scheduling a new event is a thorny issue when it involves a node that already has events pending. Since D's value does not really matter (it does not control any switches itself), the first alternative seems the most reasonable. Given the simplicity of the RSIM model, it probably does not pay to overly complicate the scheduling of events. The transition-time estimates are not accurate enough to allow subtle distinctions to be made based on the relative transition times of nodes; RSIM avoids choices (2), (3), and (4) since they involve such distinctions. Note that a similar problem arises for node E. It has an event pending for a transition to the correct value (E is still going low), but the event could be rescheduled to reflect a faster transition time since the pullup on node D no longer impedes the transition. Chapter 4 details the exact choices made by RSIM under various circumstances.

Returning to the example:

Event #3. Node E is changed to 0, causing the stage containing node F to be analyzed. F is calculated to change value, so event #6 is scheduled.

Events #4,5. Discussed in the preceding paragraph.

Event #6. F is set to 1. F does not affect any other stages, so no events are added to the event list.

At this point, the event list is empty, and the network has settled. If the user now changes node B to 1, a somewhat simpler sequence of events ensues:

Event #7. Node B is set to 1 by the user, causing the simulator to analyze the stage containing D. D is predicted to go low, resulting in the scheduling of event #8.

Event #8. D is set to 0, separating C and E into different stages which are then analyzed. C shows no change, but E is scheduled to go high (event #9) now that it is disconnected from C's pulldown.

Event #9. E changes to 1, and as a consequence F is predicted to change to 0 (event #10). Note that the low-to-high transition time can be very different than the high-to-low transition time; RSIM takes into account the relative sizes of the pullup and pulldown.

Event #10. Finally, F is set to 0.

Once again the event list is empty, and the network has settled.

2.4. Calibrating and using the RSIM model

From a practical viewpoint, the success of RSIM depends to a large degree on the choice of the modeling resistance for each transistor. The principal goal of the calibration process is to choose resistances that lead to accurate predictions. Actually, there are two separate sets of resistances to be chosen: static and dynamic. Static resistances, used to estimate node voltages, are comparatively easy to choose. When a circuit does not depend on device ratios for correct operation — *e.g.*, a pulled-up node or a CMOS gate — the values chosen for static resistances do not affect the voltage computation, since the nodes connect to only one polarity of input. When a circuit makes a connection to inputs of different polarities — *e.g.*, a nMOS gate with a logic-low output — the intervening nodes become part of a voltage divider, and the transistor resistances must be chosen to predict the divider's output voltage. Since only the ratio of the pullup and pulldown devices is constrained, there is considerable freedom in choosing the actual resistance values. Of course, inauspiciously chosen values can run afoul of range and round-off problems in the computation, but such problems are easily avoided.

A more interesting problem is the choice of appropriate dynamic resistance values. One approach involves performing a series of experiments designed to measure the resistance of each type of transistor in various circuit contexts:

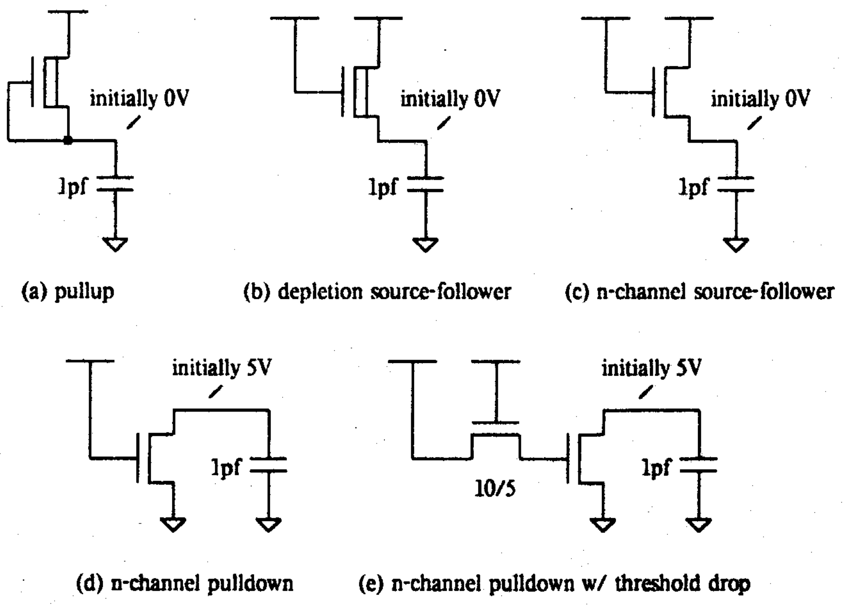


Figure 2.16. *Simple experiments for measuring channel resistances*

Ideally, the experiments should be performed using actual circuits; when this is impractical, a well-calibrated circuit analysis program can be used to gather the needed measurements. Each of the experiments entails measuring the length of time required for the output to rise or fall from its starting voltage to the switching threshold. (Section 3.4.1 describes the reason for using single threshold, and the method for choosing it.) If the load capacitance is known, an appropriate channel resistance can be calculated, essentially inverting the computation performed by RSIM. Appendix 2 presents the transistor resistances derived in this manner for a typical 5μ nMOS process.

Unfortunately, while the experiments outlined above lead to usable predictions of circuit performance, the predictions are not as accurate as one might like. The problem with the experiments is that the resistance measurements are made in a rather artificial context. Factors important in determining the behavior of a transistor in a particular circuit (*e.g.*, shape of the input waveform, Miller capacitances, etc.) are not measured by the proposed experiments. Since the simple RSIM model does not account for these factors, they are missing completely from the calculations, leading to inaccurate predictions. There are two alternatives:

(1) Modify the RSIM model to include effects deemed important when making performance predictions. It is difficult to start down this road and still keep the model simple; carried to its logical conclusion, this course of action leads to a circuit analysis program — the very thing RSIM tries to avoid. There are, however, alternatives that fall short of abandoning the simple model; these are discussed at the end of Chapter 3.

(2) Conduct more sophisticated experiments using circuit configurations found in actual designs.

An example of the second approach is the following experiment:

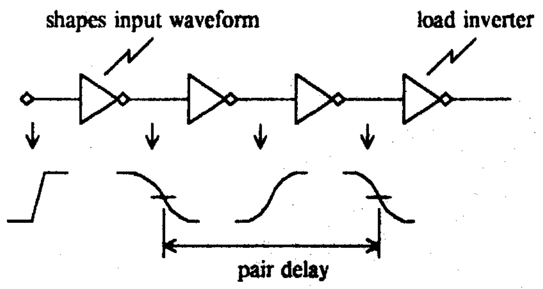


Figure 2.17. *Deriving resistances by measuring inverter pair delay*

The delay through a pair of inverters involves both a rising transition (measuring the pullup resistance) and a falling transition (measuring the pulldown resistance). The initial inverter provides an appropriately shaped input waveform; the last inverter provides a realistic output load. The measured pair delay is arbitrarily split into a rising delay and a falling delay (say, ¾ and ¼ respectively), so that the pullup and pulldown resistances can be calculated. This leads to good predictions for the chains of inverting logic so common in MOS designs. Similar experiments can be designed to measure other resistances. The danger in this approach is that, because of the *ad hoc* nature of the experiments, the resistances might be inappropriate for new circuit configurations. However, with a prudent choice of circuits during calibration and design, this danger can be minimized.

The following examples are analyzed using the simple calibration given in Appendix 2. The results give a feel for the performance of the "pure" resistance model, and also set the stage for the model improvements suggested in Chapter 3. The calculation of node voltages is straightforward and is not mentioned in the discussion below, which focuses on the calculation of transition times. The first example is a path through a PLA:

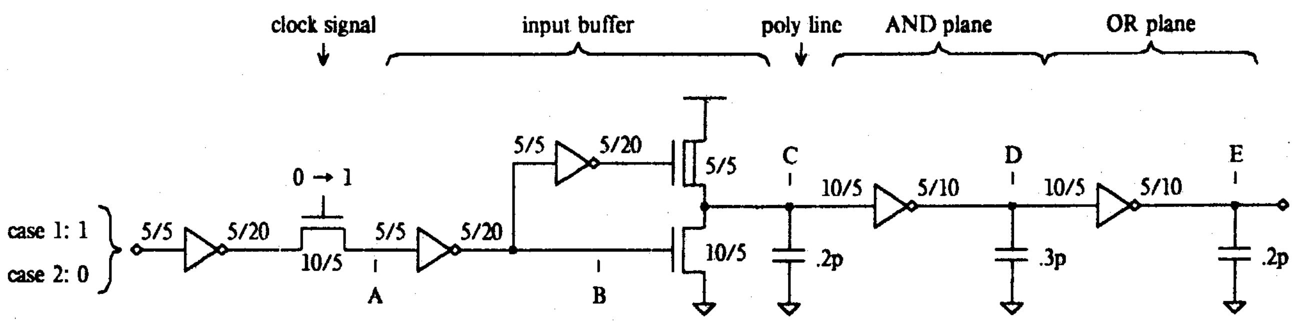


Figure 2.18. *Sample circuit showing path through PLA*

Transistor sizes are given in microns as width/length. When the clock signal goes high, the input signal (buffered by the inverter on the left) propagates through the input buffer and the two PLA planes. The following figure shows the equivalent resistor/capacitor network; resistances are given in KΩ and capacitances in pf.

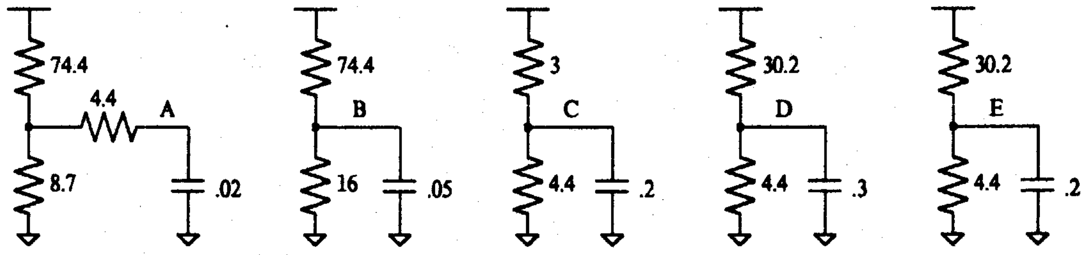


Figure 2.19. *Equivalent RC network for PLA circuit (shows dynamic resistances)*

Note that the pullup for node C is recognized as a depletion source-follower without considering the actual voltage on its gate. Since depletion devices are always on, the inverter which leads from node B to the gate of the pullup is ignored by RSIM, and the timing for node C is always controlled by node B. Also note that the resistance chosen for the pulldown for node B reflects the threshold drop of node A.

When calculating R_{dynlow}, RSIM simply calculates the net resistance to ground, ignoring the effects of any pullups. For example, a falling transition for node B takes (16)(.05) = 0.8ns. This approach is not only simpler, but is conservative. (Adding the pullup resistance actually decreases the fall time from the Thevenin point of view). Using this approach, the table shows the results of propagating two different data values through the PLA. The time of each node's transition is shown in nanoseconds, as predicted by RSIM and SPICE.

		A	B	C	D	E
Case 1	transition	↓	↑	↓	↑	↓
	RSIM	0.3	4.0	4.9	14.0	14.9
	SPICE	0.8	3.5	6.8	15.5	20.7
Case 2	transition	↑	↓	↑	↓	↑
	RSIM	1.6	2.4	3.0	4.3	10.3
	SPICE	0.6	1.9	3.3	6.4	12.1

The discrepancies between the RSIM and SPICE predictions (-28% in case 1, -14% in case 2) can be traced to the fact that the current RSIM model does not account for the shape of the input waveform when analyzing a stage.† This is particularly noticeable in case 1 for the transition of node E. The long rise time of node D slows the falling transition of E to a considerable extent; a fact blithely ignored by RSIM.

The second example is a section of the OM2 data path [Mead80] consisting of the logic to drive a register select line, a register cell, and a bus line. The path to be analyzed starts with the clock going high, driving the select line high, finally causing the register cell to discharge the pre-charged bus line.

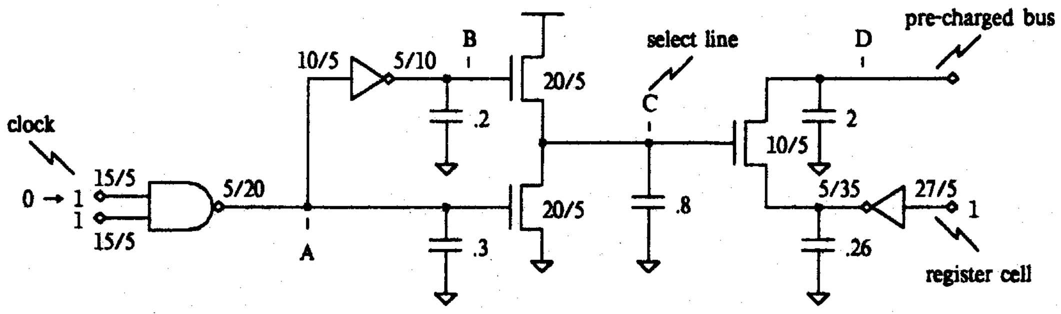


Figure 2.20. *Register select and bus drive circuitry from OM2 data path*

†Examining the times in this example, one might be tempted to multiply the effective resistances by a constant factor in an effort to improve the accuracy of the predictions. But not all predictions underestimate the true transition time, and, as will be seen in Chapter 3, there are other improvements that can be made that address the root of the problem.

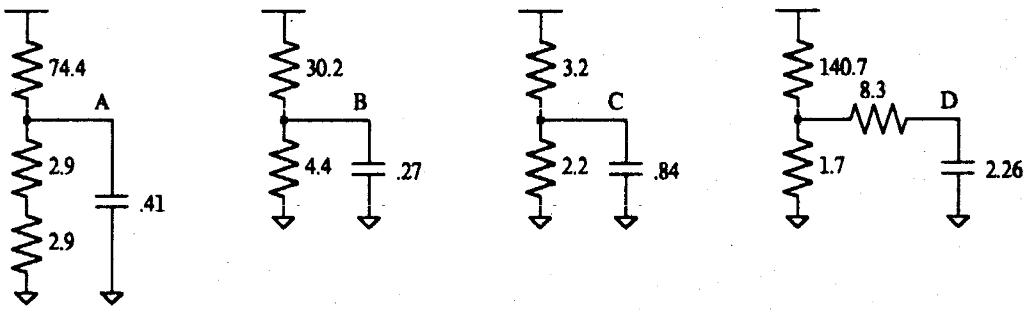


Figure 2.21. *Equivalent RC network for OM2 data path example*

The comparative analysis is given below; RSIM comes to within 9% of the SPICE prediction.

	A	B	C	D
transition	↓	↑	↑	↓
RSIM	2.4	10.6	13.3	35.9
SPICE	2.6	9.1	19.6	39.6

2.5. Summary

The RSIM model can be summarized as follows:

- Transistors are modeled as switches with series resistors. Three resistances are chosen for each transistor and used to predict node voltages and transition times. Resistance values are determined by experiments, either with actual circuits or using a circuit analysis program.
- Using the transistor model, a network of transistors and nodes is simulated as a network of resistors (from transistors) and capacitors (from nodes). A node's value is determined by voltages calculated in two ways: (1) from charge sharing with electrical neighbors, and (2) from the Thevenin equivalent circuit for pieces of network connecting the node to the inputs. When a node changes value, the timing for the transition is given by an RC time constant calculated using the resistances and capacitances of the surrounding network.
- The network is viewed as an assemblage of small stages, each simple enough that its operation can be predicted in a straightforward manner. Information propagates through the network as a series of events (changes in a node's value); each event leads to an analysis of affected stages using the models described above. The isolation between stages of digital circuits allows each stage to be analyzed separately; the relative independence of one stage from another is one reason why the very rough approximations of RSIM are so serviceable.

Several factors important for making accurate performance predictions are missing from both the RSIM model and the simple calibration experiments proposed in section 2.4. Chapter 3 suggests some modifications to the model that correct the more important oversights. Many implementation details unspecified in this chapter are discussed in Chapter 4. Chapter 4 also catalogs the successes and failures of the RSIM model, as finally implemented.

CHAPTER THREE

Justification of the Linear Network Model

This chapter undertakes a performance analysis of logic gates and other digital circuits with the goal of establishing a physical justification for the RSIM model. By comparing the resulting equations with those proposed by RSIM, one can judge the accuracy with which the RSIM model predicts circuit behavior. As an added benefit, insight into actual circuit operation helps to motivate model modifications that improve the accuracy of the predictions.

The first section lays the groundwork for the analysis, presenting the first-order equations that describe the operation of MOS transistors. The second section describes the node voltages found in common digital logic circuits and compares the results to RSIM's predictions. The next two sections analyze the propagation delay of logic gates and other network components. Finally, several modifications to the RSIM model are proposed, and the resulting predictions are compared to those of the original model.

3.1. Electrical models for mosfets and gates

The active component in a MOS circuit is the *mosfet*, a type of transistor. The mosfet has three terminals: the source and drain (two symmetric connections), and the gate. By convention, the source and drain are chosen such that v_{ds}, the voltage of the drain with respect to the source, positive. v_{gs}, the voltage of the gate with respect to the source, can be either positive or negative. Depending on the relative voltages of the three terminals, the mosfet conducts varying amounts of current between the source and drain terminals. The amount of current conducted depends on the region in which the mosfet operates. There are three possible regions:

$$i_{ds} = \begin{cases} 0 & v_{gs} - v_{th} < 0 \quad (\mathit{off}) \\ \frac{\kappa}{2}(v_{gs} - v_{th})^2 & 0 \leq v_{gs} - v_{th} \leq v_{ds} \quad (\mathit{saturated}) \\ \kappa(v_{gs} - v_{th} - \frac{v_{ds}}{2})v_{ds} & v_{gs} - v_{th} > v_{ds} \quad (\mathit{linear}) \end{cases} \tag{3.1}$$

where v_{th} is the threshold voltage of the mosfet and

$$\kappa = \frac{w}{l}\,\mu\, C_o \approx \frac{w}{l}\,(25\frac{\text{microamps}}{\text{volt}^2}) \tag{3.2}$$

is a constant that depends on the width w and length l of the particular mosfet under consideration. The numeric estimate is for a typical nMOS process. These equations ignore second order effects on i_{ds}.

In an nMOS process, there are two types of mosfets, distinguished by the setting of their thresholds:

type of device	*threshold* (VDD $\equiv$ 1)
n-channel	$v_{te} \simeq 0.14$
depletion	$v_{td} \simeq -0.6$

As we saw in Chapter 2, the simplest form of logic gate that uses these devices consists of:

a single depletion pullup with its gate and source attached to the output node and its drain attached to VDD, and

one or more pulldown paths connecting the output node to ground, each path containing one or more n-channel devices.

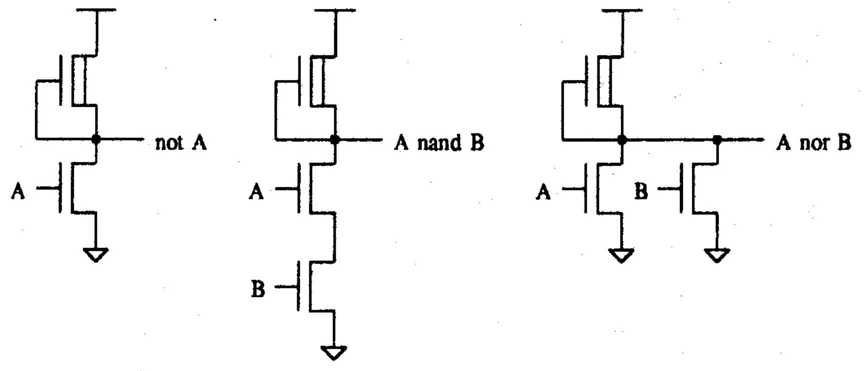


Figure 3.1. *nMOS logic gates*

The depletion pullup is configured so that $v_{gs:pu} = 0$; since the threshold of a depletion device is negative, $v_{gs:pu} - v_{td} > 0$, and the pullup is never off. Each n-channel pulldown is configured to be on when its gate voltage exceeds v_{te} and off otherwise. If all the n-channel devices in a particular pulldown chain are conducting, the output load capacitance is discharged through the pulldown path and the output voltage is lowered ($v_{out} = v_{ol} = \textit{logic low}$); otherwise the pullup pulls the output high ($v_{out} = v_{oh} = \textit{logic high}$).

Equation 3.1 can be specialized for a depletion pullup, using the fact that $v_{gs:pu}$ is always zero:

$$i_{pu} = \begin{cases} \frac{\kappa_{pu}}{2} |v_{td}|^2 & |v_{td}| \le (1-v_{out}) \\ \kappa_{pu}(|v_{td}| - \frac{(1-v_{out})}{2})(1-v_{out}) & |v_{td}| > (1-v_{out}) \end{cases} \tag{3.3}$$

where v_{out} is the voltage of the gate/source node of the pullup. Since the drain of the pullup is connected to VDD, $v_{ds:pu} = 1 - v_{out}$. To avoid confusion, the equations will be written in terms of $|v_{td}|$ since v_{td} is negative. The current conducted by the n-channel pulldown in an inverter is given by:

$$i_{pd} = \begin{cases} 0 & v_{in} - v_{te} < 0 \\ \frac{\kappa_{pd}}{2}(v_{in} - v_{te})^2 & 0 \le v_{in} - v_{te} \le v_{out} \\ \kappa_{pd}(v_{in} - v_{te} - \frac{v_{out}}{2})v_{out} & v_{in} - v_{te} > v_{out} \end{cases} \tag{3.4}$$

where v_{in} is the voltage of the gate node of the pulldown. Note that the source of the pulldown is connected to ground ($v_{in} = v_{gs:pd}$) and the drain is connected to the inverter's output ($v_{out} = v_{ds:pd}$).

For proper operation of the inverter, the sizes of the pullup and pulldown are chosen so that $i_{pd} > i_{pu}$ when the pulldown is on.

To understand the behavior of an inverter in more detail, it is useful to plot i_{ds} of the component devices as a function of the inverter's output voltage:

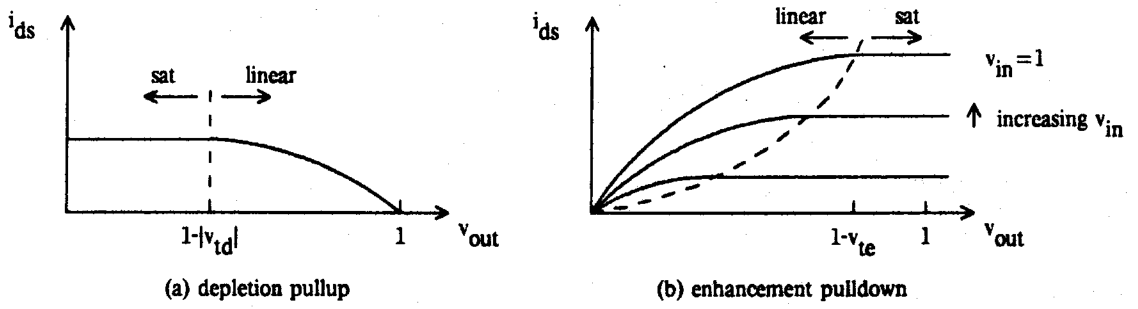


Figure 3.2. *mosfet I-V characteristics*

The i_{ds} of a depletion pullup depends only on v_{out} and thus a single curve suffices to show their relationship. For the n-channel pulldown, there is a family of curves for i_{ds} corresponding to different values of v_{in}.

The intersection of the i_{ds} curves for the pullup and pulldown shows the inverter's output voltage, given a particular input voltage:

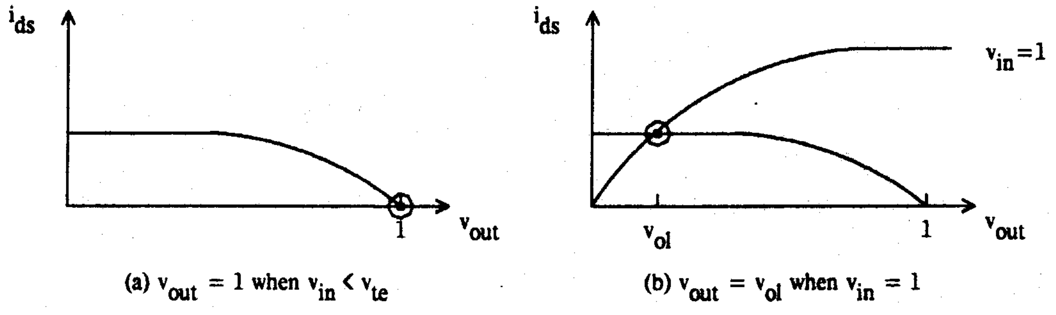
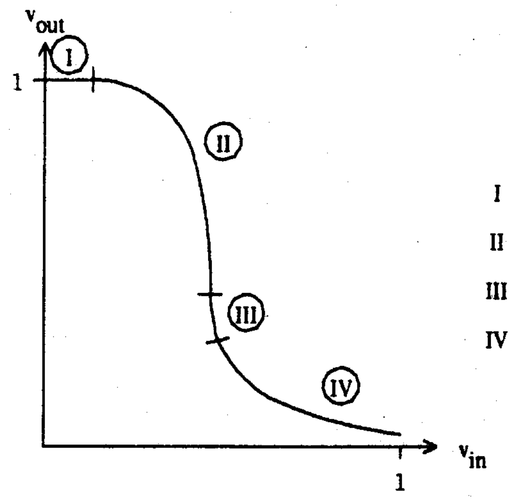


Figure 3.3. *v_{out} is determined by i_{pu} and i_{pd}*

In fact, one can plot the DC voltage transfer curve for an inverter, which shows the inverter's output voltage as a function of its input voltage.



I	pulldown = off	pullup = linear
II	pulldown = sat	pullup = linear
III	pulldown = sat	pullup = sat
IV	pulldown = linear	pullup = sat

Figure 3.4. *Voltage transfer curve for an inverter*

The four regions (I−IV) of the curve correspond to various combinations of the pullup's and pulldown's operating regions. Note that the relationship between v_{in} and v_{out} shown in figures 3.3 and 3.4 applies when the voltages are allowed to stabilize; in a circuit with changing voltages, the relationship between the v_{in} and v_{out} is considerably more complicated, as will be seen in section 3.4.

The next few sections use the equations presented here to develop equations for the quantities predicted by RSIM — node voltages and transition times — so that the RSIM model can be evaluated and perhaps improved.

3.2. Node voltages

When $v_{in} < v_{te}$, the n-channel pulldown conducts no current; the depletion load continues to conduct as long as $v_{out} < 1$. Therefore, the logic high output voltage of an inverter is given by the equation:

$$v_{oh} = 1 \tag{3.5}$$

When $v_{in} > v_{te}$, the n-channel pulldown is on and the output node reaches an equilibrium voltage v_{ol}, which is determined by (1) the relative sizes of the pullup and pulldown and (2) the gate voltage on the pulldown. v_{ol} is that voltage where the current of the pulldown (at this point in its linear region) is balanced by the current of the pullup (in saturation):

$$\kappa_{pd}(v_{in} - v_{te} - \frac{v_{ol}}{2})v_{ol} = \frac{\kappa_{pu}}{2} \, | \, v_{td} \, |^2 \tag{3.6}$$

If one assumes that $v_{in} = 1$ (as is the case when v_{oh} of the previous stage is 1) and that $v_{ol} \ll 1 - v_{te}$, then

$$v_{ol} \approx \frac{1}{2R} \frac{|v_{td}|^2}{(1-v_{te})} \approx \frac{0.21}{R} \tag{3.7}$$

where $R = \frac{\kappa_{pd}}{\kappa_{pu}} = \frac{l_{pu}}{w_{pu}} \frac{w_{pd}}{l_{pd}}$ is the ratio of the sizes of the pullup and pulldown. R is chosen so as to guarantee that the low output of a gate turns off the pulldowns of gates connected to the output, *i.e.*, so that v_{ol} is less than v_{te} by a comfortable margin; typically R is chosen to be about 4 if $v_{in} = 1$.

Now consider the RSIM model for an inverter:

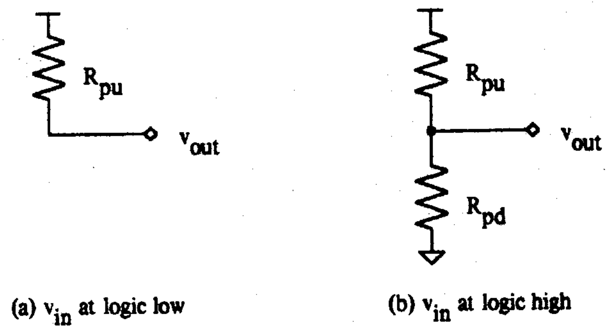


Figure 3.5. *RSIM inverter model*

When v_{in} is low, the pulldown is off and the inverter is modeled with a single resistor. In this configuration, RSIM predicts

$$v_{oh:RSIM} = 1 \tag{3.8}$$

agreeing with equation 3.5, independent of the value chosen for R_{pu}. When v_{in} is high, the inverter is modeled by a voltage divider. RSIM predicts

$$v_{ol:RSIM} = \frac{R_{pd}}{R_{pd} + R_{pu}} \tag{3.9}$$

One should choose R_{pu} and R_{pd} so that $v_{ol:RSIM}$ is the same as v_{ol}, as given by equation 3.7. Thus the RSIM model can accurately predict the output voltages of logic gates; in fact, there are two unknowns and only one equation to satisfy, so there is some freedom in choosing the static resistance values.

There are circuits for which RSIM does not properly predict node voltages. For example, in the following circuit, the voltage of node B only reaches $1 - v_{te}$:

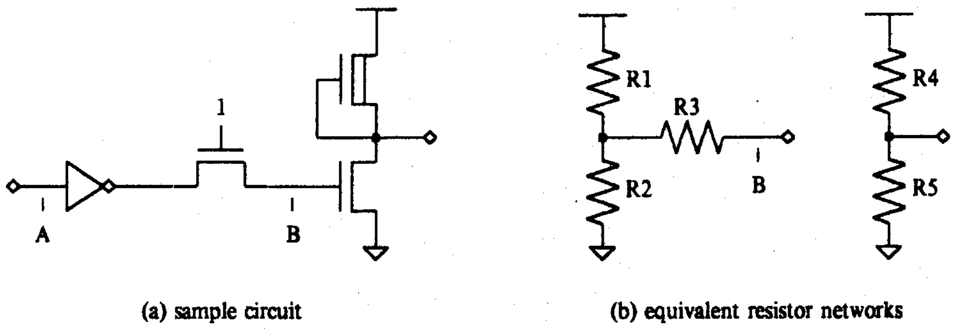


Figure 3.6. *Sample circuit illustrating voltage drop across pass transistor*

N-channel devices configured the same way as the horizontal transistor in figure 3.6(a) are called "pass" transistors, and are used to implement dynamic latches, various types of steering logic, and so on. Figure 3.6(b) shows the equivalent resistor networks for the circuit. According to this model, the voltage for node B should reach VDD when node A is low. In the actual circuit, however, the pass transistor cuts off when B reaches $1 - v_{te}$ since, at that point, $v_{gs:pass} = v_{te}$. In general, the source voltage of a pass transistor never rises above a threshold-drop below its gate voltage. Thus the RSIM model incorrectly predicts the voltage of node B.

In fact, the network analysis performed by RSIM does recognize that node B never reaches VDD. As shown by several examples in Chapter 2, the resistance for a pulldown with a gate that has a threshold voltage drop is not chosen in the same way as the resistance for a normal pulldown. In other words, the value of R5 in figure 3.6(b) reflects the knowledge that node B has a threshold drop. This knowledge could also be used to adjust the prediction of B's voltage, but this is not currently part of the calculation.

There are many other circuit configurations that are beyond the ability of RSIM to analyze, although most such circuits could not, in all fairness, be called digital. One important exception, which RSIM does not handle, but which occurs in performance-critical digital circuits, is called *bootstrapping.*

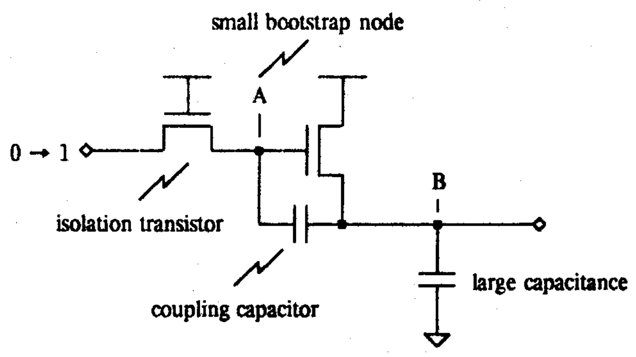


Figure 3.7. *Bootstrap circuits lead to voltages greater than VDD*

Node A is small compared to node B, to which it is capacitively coupled. The coupling capacitor need not be explicit; often enough coupling is provided by the gate/source overlap capacitance of the transistor controlled by A. Node A is driven high through a pass transistor, and in turn enables the n-channel pullup that is controlled by A and connected to node B. Since the capacitance of A is small compared to that of B, A reaches a significant voltage before the voltage of node B begins to change; the difference is usually around 3 volts in common bootstrap configurations. As the voltage of node B increases, the coupling capacitor maintains this initial voltage difference between nodes A and B, and so the voltage of A increases correspondingly.† It is not unusual for node A to reach 8 volts or more. This, of course, increases the voltage on the gate of the pullup, which in turn increases the current flowing into node B. The net result is that node B reaches its final value much more quickly than one might expect. Just as important, the voltage of B rises all the way to VDD instead of stopping two threshold drops below, as a simple analysis might predict.

Both the faster transition time and higher-than-expected voltage for node B are completely missed by RSIM. Since such circuits are often used in time-critical portions of the network, it would be nice for RSIM to make correct predictions in this case. Unfortunately, there is no simple change to the simple RSIM model that achieves the desired result. However, by systematically replacing bootstrap circuits with more conventional circuits sized to give the same performance, RSIM can produce the correct results. This technique is discussed in the section on escape mechanisms in Chapter 4.

In summary, RSIM

†The pass device through which node A is driven isolates A from the driving circuitry. After the voltage of node A reaches $1 - v_{te}$, the pass device cuts off, and stays off no matter large the voltage on node A becomes. This is because $v_{g:pass} - v_{te}$ will be less than the voltage on either the source or the drain.

(i) predicts the output voltage of logic gates with acceptable accuracy.

(ii) does not predict threshold drops introduced by pass transistors, but does perform a static analysis of the network to recognize transistors whose gates are subject to a threshold drop, and adjust the modeling resistance accordingly.

(iii) does not handle bootstrap and other more exotic circuits. However, a pattern matching/replacement technique is available for substituting equivalent circuits that simulate correctly.

3.3. Propagation delay: overview

When choosing a single number to characterize the timing behavior of a circuit, one often settles for determining the *propagation* delay: a measure of the length of time required for a change in an input value to be reflected in the output value. In digital circuitry, a significant change is one where the signal changes from logic low to logic high or vice versa. For a particular transition it is common to define "change" in relation to a threshold; the signal is said to change when it crosses the threshold. Consider the following single input, single output circuit:

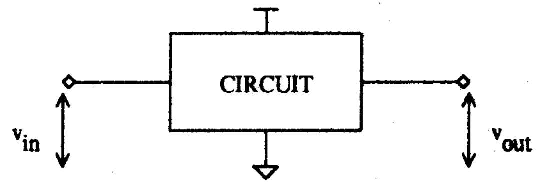


Figure 3.8. *Test setup for measuring propagation delay*

The propagation delay is defined as

$$t_p = t_{output} - t_{input} \tag{3.10}$$

where

t_{output}	is the time when the output voltage crosses the output threshold voltage;
t_{input}	is the time when the input voltage crosses the input threshold voltage.

This definition works well for a transition between 0 and 1; however, delays associated with a transition to the X state are still not well defined since it is unclear whether the signals in question cross the threshold or not. Aside from this technical difficulty, the notion of propagation delay involving X's is rather muddy since X is not a "real" logic value, but more of an error state. The simulation algorithm must assign some delay to such a transition, and RSIM conservatively chooses the fastest possible transition of which the node is capable (see equations 2.8 and 2.9).

The next step is to choose the input and output thresholds, a choice that depends on the particular circuit to be analyzed. There are two important criteria for choosing thresholds:

(1) The delay should never be negative. The thresholds should be chosen so that the input always crosses its threshold before the output does. The simulation algorithm quite naturally processes events in the scheduled order; allowing a negative delay might require backing-up a previously processed event.

(2) The output threshold for a circuit should be chosen without regard to its use, allowing a single threshold to be chosen for all inputs and outputs. In that case, only one delay computation is needed for each signal transition.

Though these criteria are not compatible in general, they can both be met for the digital circuits of interest here.

To simplify the analysis below, will restrict the class of input waveforms considered. In his work on waveform bounding, Wyatt [Wyatt83] observes that the transfer functions characterizing digital MOS circuitry meet certain criteria which guarantee that

if two monotonic trial waveforms are chosen that bound the actual input waveform (which also must be monotonic), then the response of the circuit to the trial waveforms will bound the actual output waveform.

Thus one can choose computationally convenient input waveforms, *e.g.*, simple voltage ramps, and determine the bounds on the propagation delay by analyzing ramps that bound the true input waveform.

3.4. Propagation delay: logic gates

In order to explore the timing behavior of MOS logic gates, this section analyzes the behavior of an nMOS inverter with a simple voltage ramp on its input. The analysis is based on the first-order equations for the component devices, presented in the previous section. The derivation is easily extended to more complex gates by adjusting the parameters of the inverter's pulldown to model the net pulldown-path resistance of the currently active pulldowns in the complex gate (see section 3.4.4). The derivation also applies to cMOS logic gates; the analysis of the low-to-high transition caused by a p-channel pullup is very similar to the high-to-low transition caused by an n-channel pulldown. For simplicity, only nMOS gates are considered below.

For the purposes of the analysis, the inverter output is connected to a fixed capacitance that models the load driven by the inverter.

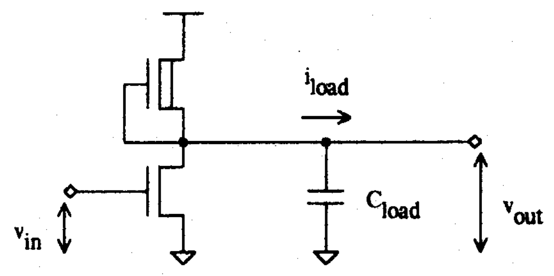


Figure 3.9. *Inverter circuit to be analyzed*

At each moment, the output voltage and the current charging/discharging the load capacitance are related by

$$i_{load} = C_{load} \frac{dv_{out}}{dt} \tag{3.11}$$

Unfortunately, this differential equation is hard to use as it stands because i_{load} is a function of both v_{out} and t. However, if one can find a suitable approximation for i_{load} that removes the dependency on v_{out}, then the change in output voltage over a given time period can be determined by integrating:

$$C_{load}(\Delta v_{out}) = \int_0^t i_{load}(t)\, dt \tag{3.12}$$

The time needed for v_{out} to change a specified amount is calculated by first performing the integration and then solving the resulting equation for t. This suggests the following plan of attack:

(i) Find suitable approximations for i_{load} to remove the dependencies on v_{out}.

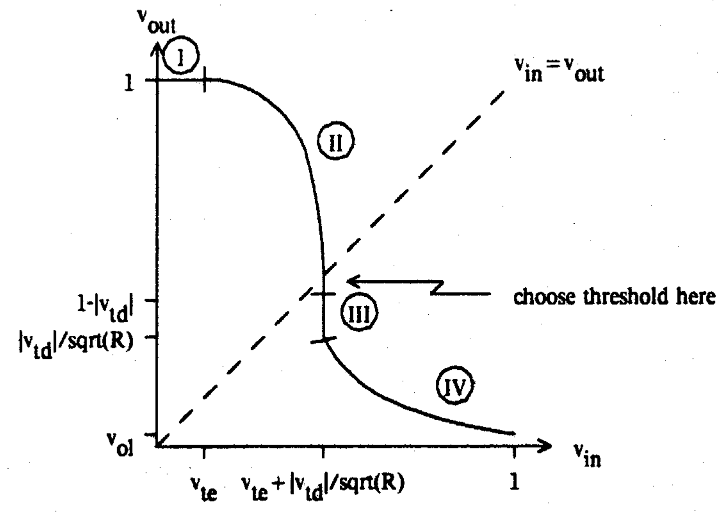
(ii) Compute the output transition time using equation 3.12.

(iii) Subtract from (ii) the input transition time, giving the actual delay from input to output. Rearrange the answer into an RC term (what RSIM predicts) and an error term.

This discussion starts with a small digression on choosing the appropriate threshold voltage.

3.4.1. Choosing the input/output threshold

To see if one can choose a single logic threshold and still guarantee that the predicted delay is never negative, it is useful to consult the voltage transfer curve for an inverter:



I	pulldown = off	pullup = linear
II	pulldown = sat	pullup = linear
III	pulldown = sat	pullup = sat
IV	pulldown = linear	pullup = sat

Figure 3.10. *Voltage transfer curve for inverter*

The transfer curve shows the static behavior of the inverter; for any given input voltage, it tells what the output voltage must be for the pullup and pulldown currents to balance. If the input changes rapidly enough, the output voltage may lag behind. If the input is going from low to high, then the transfer curve shows the *minimum* output voltage for a given input voltage; for a high-to-low input transition, the transfer curve shows the *maximum* output voltage for a given input voltage.

Since it is desirable for the input and output thresholds to be the same, the input/output threshold voltage v_{thresh} is chosen to be the point on the transfer curve where $v_{in} = v_{out}$.† This means that during a low-to-high input transition, if $v_{in} < v_{thresh}$, then $v_{out} > v_{thresh}$, no matter how fast or slow the transition. In other words, the propagation delay is never negative. A similar argument applies for the other transition. To estimate v_{thresh}, first notice that at the region II—region III boundary,

$$v_{in} = v_{te} + \frac{|v_{td}|}{\sqrt{R}} \quad \text{and} \quad v_{out} = 1 - |v_{td}| \tag{3.13}$$

If $R = 4$, then $v_{in} = .44$ and $v_{out} = .4$, and so v_{thresh} is in region II (just barely). In this region the pulldown is in saturation and the pullup is in the linear region:

$$\frac{\kappa_{pd}}{2}(v_{in} - v_{te})^2 = \kappa_{pu}(|v_{td}|(1-v_{out}) - \frac{(1-v_{out})^2}{2}) \tag{3.14}$$

†The same choice of threshold has been made in several other simulators [Koppel78, Nahm80].

Setting $v_{in} = v_{out} = v_{thresh}$, and solving for v_{thresh} yields $v_{thresh} = .439$ — close enough to the II−III boundary that the distinction is not important.

3.4.2. Low-to-high output transition time, t_{plh}.

To calculate t_{plh}, an approximation for i_{load} is needed. i_{load} is just the difference between the pullup current (i_{pu}) and the pulldown current (i_{pd}), so one strategy is to approximate the current through each component individually. Recall that v_{thresh} is near the region II−region III boundary of the inverter's voltage transfer curve, and notice that the part of the transition involved in the prediction (v_{out} rising from 0 to v_{thresh}) takes place almost entirely with the inverter operating in regions III and IV. This means that the pullup is in saturation, *i.e.*,

$$i_{pu} = \frac{\kappa_{pu}}{2} \, | v_{td} |^2 \equiv i_{\max} \tag{3.15}$$

Choosing a specific approximation for i_{pd} is not as straightforward. However, a good starting point is an approximation of the form shown in the following figure.

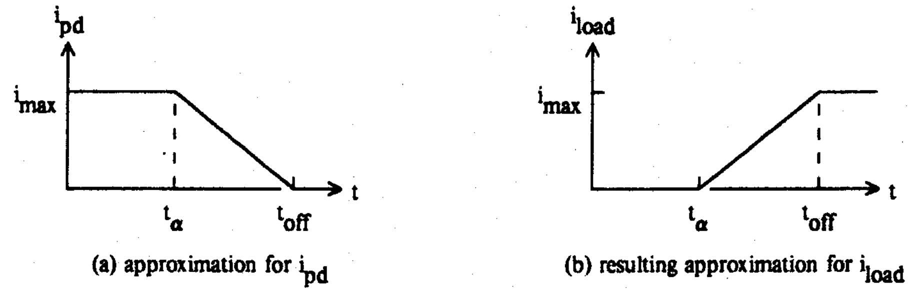


Figure 3.11. *Approximation of i_{pd} for t_{plh} calculation*

t_{off} is the time at which $v_{in} = v_{te}$. At this point in the development, there is not much one can say about t_α, the time at which the pulldown current first starts to decrease. Certainly $t_\alpha = t_{off}$ is an upper bound (resulting in a step function for i_{pd}). Similarly, $t_\alpha = 0$ is a lower bound since that is the time when the input voltage first changes. The choice of a specific value for t_α will be discussed later.

With this approximation, the output transition time, t_h, is given by

$$C_{load}(v_{thresh}) = \int_0^{t_h} i_{load}(t)\, dt \tag{3.16}$$

where

$$i_{load}(t) = \begin{cases} 0 & t \le t_\alpha \\ i_{\max}\left(\dfrac{t - t_\alpha}{t_{off} - t_\alpha}\right) & t_\alpha \le t \le t_{off} \\ i_{\max} & t_{off} \le t \end{cases} \tag{3.17}$$

Solving equation 3.16 for t_h yields

$$t_h = \begin{cases} R_{pu} C_{load} + \frac{1}{2}(t_{off} + t_\alpha) & t_h \ge t_{off} \\ [2R_{pu} C_{load}(t_{off} - t_\alpha)]^{\frac{1}{2}} + t_\alpha & t_h < t_{off} \end{cases} \tag{3.18}$$

where $R_{pu} = \dfrac{v_{thresh}}{i_{\max}}$. Recalling that $t_{plh} = t_h - t_{input}$,

$$t_{plh} = \begin{cases} R_{pu} C_{load} + \frac{1}{2}(t_{off} + t_\alpha) - t_{input} & t_{plh} \ge t_{off} - t_{input} \\ [2R_{pu} C_{load}(t_{off} - t_\alpha)]^{\frac{1}{2}} + t_\alpha - t_{input} & t_{plh} < t_{off} - t_{input} \end{cases} \tag{3.19}$$

The following figure plots t_{plh} as a function of t_{input}. Note that there is a relationship among the values of t_{input}, t_{off}, and t_α. For this plot, a linear relationship is assumed for the values. Their exact relationship is determined by the shape of the input waveform, a topic pursued below.

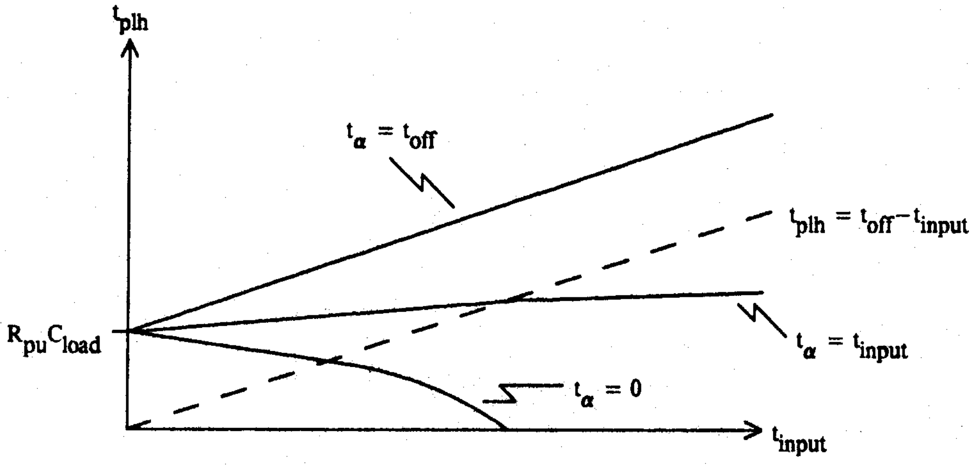


Figure 3.12. *t_{plh} as a function of t_{input}*

Several interesting observations can be made. When the input is a voltage step, t_{off}, t_α, and t_{input} are all zero, so $t_{plh:step} = R_{pu} C_{load}$, *i.e.*, a simple RC time constant — precisely the prediction made by the RSIM model.

To see what happens when the input is not a step, notice that

$$t_{plh} \leq R_{pu} C_{load} + \frac{1}{2}(t_{off} + t_\alpha) - t_{input} \tag{3.20}$$

since

$$[2R_{pu} C_{load}(t_{off} - t_\alpha)]^{\frac{1}{2}} + t_\alpha - t_{input} \leq R_{pu} C_{load} + \frac{1}{2}(t_{off} + t_\alpha) - t_{input} \tag{3.21}$$

when $t_{plh} > t_{off} - t_{input}$. (This can be verified by comparing the derivatives of the two sides of the inequality or by simply extending the linear portion of the t_{plh} curves — those portions above the dotted line — in the plot above.) Equation 3.20 looks like the response for a step input delayed by an amount that depends solely on parameters of the input waveform.

Figure 3.12 provides some insight into the choice of an appropriate value for t_α. From the plot, one can see that t_{plh} eventually goes to zero for some choices of t_α, but increases indefinitely for other choices. By determining whether t_{plh} goes to zero in an actual circuit, it is possible to narrow the range of choices for t_α. If the input changes slowly enough, one expects the output voltage to follow the voltage transfer curve very closely. (This is essentially the definition of the voltage transfer curve.) Thus, when $v_{in} = v_{thresh}$, it follows that $v_{out} = v_{thresh}$ since v_{thresh} is the balance point of the inverter. This implies $t_{plh} = 0$ for sufficiently slow input transitions.

Examining the bottom term of equation 3.19, one can see that t_{plh} is zero for slow input transitions only if $t_\alpha < t_{input}$.† In other words, if $t_\alpha \geq t_{input}$, the predicted propagation delay can never be zero; the prediction will be longer than the true propagation delay. Thus, it is possible to rewrite equation 3.20 using $t_\alpha = t_{input}$ and still preserve the inequality.

$$t_{plh} \leq R_{pu} C_{load} + \frac{1}{2}(t_{off} - t_{input}) \tag{3.22}$$

This equation can be simplified still further with some assumptions about the input waveform.

†The bottom term has the form $[f(t)]^{1/2} + g(t)$ which reaches zero for large t only if g(t) is negative.

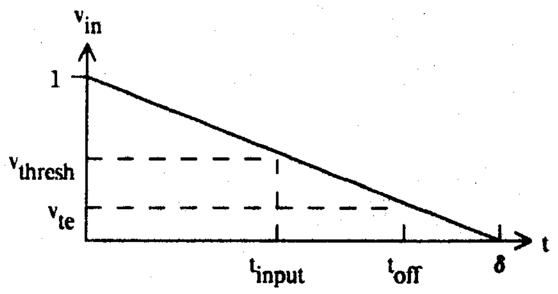


Figure 3.13. *Assumed input waveform for low-to-high output transition*

If the input is a falling voltage ramp which starts at $t = 0$ and reaches zero at $t = \delta$, then $t_{input} = (1 - v_{thresh})\delta$ and $t_{off} = (1 - v_{te})\delta$. Substitution into equation 3.22 yields

$$t_{plh} \leq R_{pu} C_{load} + \frac{\delta}{2}(v_{thresh} - v_{te}) = R_{pu} C_{load} + (0.15)\delta \tag{3.23}$$

where the numerical estimate is computed for a typical 5μ nMOS process. Thus RSIM potentially underestimates t_{plh} for a logic gate with a slow input transition (a large δ). As δ decreases (a faster input transition), the accuracy of RSIM's predictions increases. Note that R_{pu} is exactly the resistance measured by the experiment proposed in figure 2.16(a).

3.4.3. High-to-low output transition time, t_{phl}.

In the previous section, the equation for t_{plh} was developed by overestimating the current through the pulldown, leading to an upper bound for the low-to-high propagation delay. The same technique can be used to estimate t_{phl}, the high-to-low transition time. In this case, however, one wants to underestimate the pulldown current (and overestimate the pullup current) to find an upper bound for t_{phl}.†

For the portion of the high-to-low output transition which is of interest (v_{out} falling from 1 to v_{thresh}), the pullup is in its linear region. As before, i_{pu} can be approximated by the pullup's saturation current; an overestimate, but one consistent with the goals of this section. Also as before, estimating the pulldown current is difficult. Consider the following diagram of various load lines for

†Most MOS circuits use multiple-phase clocking, with simple logic circuits between latches controlled by different phase clocks. This means that circuit performance is determined by the *maximum* propagation delay through the simple logic; this is the only quantity estimated by RSIM. Other technologies (TTL, ECL) support single-clock, synchronous designs in which minimum propagation delays can be very important for correct circuit operation. This is rare in MOS circuits, and such designs are not supported by RSIM.

the pulldown. The trajectory of a load line shows i_{pd} as a function of time:

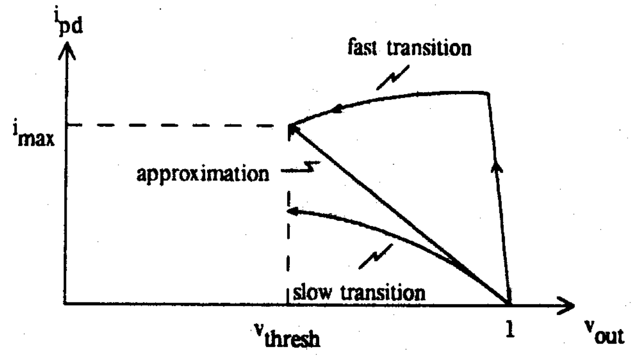


Figure 3.14. *Load lines for the pulldown for various input transitions*

When the input transition is fast in comparison to the output transition, the pulldown turns on to its maximum current capacity (the upper load line in figure 3.14). As v_{out} drops, the current in the pulldown also decreases, and the trajectory follows the maximum current curve until it reaches v_{thresh}. When the input transition is slow, the output voltage falls fast enough to keep the pulldown and pullup currents balanced (the bottom load line in figure 3.14), so the trajectory for i_{pd} follows the i_{pu} curve.

In the proposed approximation, i_{pd} rises linearly to a maximum current equal to the actual current through the pulldown when $v_{in} = 1$ and $v_{out} = v_{thresh}$. This certainly underestimates the actual pulldown current for a fast transition, and is roughly equal to the pulldown current for a slow transition, except for the last part of the transition. Fortunately, in this portion of the transition (near the threshold), a small change in the input voltage causes a large change in the output voltage, so only a small amount of time is actually spent in the overestimated part of the transition. This approximation leads to the following estimate for i_{load}:

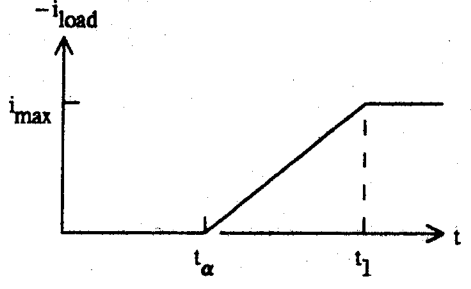


Figure 3.15. *Estimate of i_{load} for t_{phl} calculation*

where t_1 is the time at which $v_{in} = 1$ and $i_{\max}$ is the maximum pulldown current minus the pullup current.

$$i_{\max} = \kappa_{pd}(1 - v_{te} - \frac{v_{thresh}}{2})v_{thresh} - \frac{\kappa_{pu}}{2} | v_{td} |^2 \tag{3.24}$$

As before, t_α will be chosen to ensure that the estimate is an upper bound to the actual propagation delay.

The derivation of a formula for t_{plh} and the choice of t_α is very similar to that of the previous section, so only the conclusion is presented here:

$$t_{phl} \le R_{pd} C_{load} + \frac{1}{2}(t_1 - t_{input}) \tag{3.25}$$

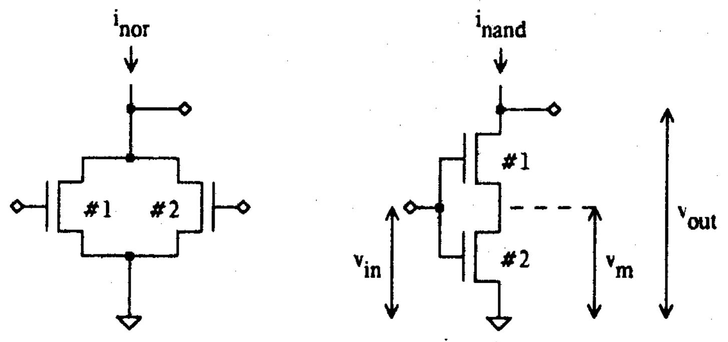
where $R_{pd} = \dfrac{1 - v_{thresh}}{i_{\max}}$. If the input is a rising voltage ramp that starts at $t = 0$ and reaches 1 at $t = \delta$, then

$$t_{phl} \le R_{pd} C_{load} + \frac{\delta}{2}(1 - v_{thresh}) = R_{pd} C_{load} + (0.28)\delta \tag{3.26}$$

As before, RSIM potentially underestimates t_{phl} for a logic gate with a slow input transition (a large δ). As δ decreases (a faster input transition), the accuracy of RSIM's predictions increases. Note that the experiment proposed in figure 2.16(d) does *not* measure R_{pd}. Instead, the experiment measures the average resistance associated with the fast input transition shown in figure 3.14, omitting the contribution of the pullup. This resistance is less than R_{pd}, although is it not clear by how much. This net result is a tendency to underestimate t_{phl} by the original RSIM model, calibrated as in Appendix 2.

3.4.4. Why analyzing inverters is sufficient

The results of sections 3.4.2 and 3.4.3 were developed for the nMOS inverter. This section extends the results to NAND and NOR gates as well. Equations are developed for the amount of current flowing through the NOR and NAND pulldown configurations and then the results are compared with the equations for a simple inverter.



(a) NOR pulldown configuration (b) NAND pulldown configuration

Figure 3.16. *Currents through NOR and NAND transistor configurations*

The propagation delay of a NOR gate with a single active pulldown is exactly that of an inverter. If both pulldowns are active simultaneously, $i_{nor} = i_1 + i_2$, since the current through each pulldown can be computed independently. Thus, when both pulldowns are on, and their gates are at the same voltage (*i.e.*, logic high), the total current through the pulldowns is

$$i_{nor} = \begin{cases} (\kappa_1 + \kappa_2)(v_{in} - v_{te} - \frac{v_{out}}{2})v_{out} & \textit{(linear)} \\ \frac{\kappa_1 + \kappa_2}{2}(v_{in} - v_{te})^2 & \textit{(saturated)} \end{cases} \tag{3.27}$$

which is equivalent to the current through a single pulldown sized so that

$$\kappa_{single\ pulldown} = \kappa_1 + \kappa_2 \tag{3.28}$$

As one might expect, this is the formula for combining two conductances in parallel.

The analysis of a NAND gate is more complicated because the currents through the two pulldowns are not independent. The currents through the pulldowns are given by

$$i_1 = \begin{cases} \kappa_1(v_{in} - v_m - v_{te} - \frac{v_{out} - v_m}{2})(v_{out} - v_m) & \textit{(linear)} \\ \frac{\kappa_1}{2}(v_{in} - v_m - v_{te})^2 & \textit{(saturated)} \end{cases} \tag{3.29}$$

$$i_2 = \kappa_2(v_{in} - v_{te} - \frac{v_m}{2})v_m \quad \textit{(linear)} \tag{3.30}$$

where v_m is the voltage of the node that is common to the two pulldowns. Two equations are needed for the top pulldown, because the pulldown may be in either its saturated or linear region, depending on the relative values of v_{in} and v_{out}. Only one equation is needed for the bottom pulldown, because it is assumed that v_m is never large enough for the bottom pulldown to become saturated. In the steady state i_1 must equal i_2. This gives a set of equations to solve for v_m; substituting the solution into equation 3.29 yields the net current through the pulldown. The result is

$$i_{nand} = \begin{cases} \dfrac{\kappa_1\kappa_2}{\kappa_1 + \kappa_2}(v_{in} - v_{te} - \dfrac{v_{out}}{2})v_{out} & \textit{(linear)} \\ \dfrac{\kappa_1\kappa_2}{2(\kappa_1 + \kappa_2)}(v_{in} - v_{te})^2 & \textit{(saturated)} \end{cases} \tag{3.31}$$

This is the same amount of current as that for a single pulldown sized such that

$$\kappa_{single\ pulldown} = \frac{\kappa_1\kappa_2}{\kappa_1 + \kappa_2} \tag{3.32}$$

Again, as one might expect, this is the formula for combining two conductances in series.

The conclusion to be drawn from equations 3.28 and 3.32 is that the current flowing through a parallel or a series configuration of pulldowns can be modeled as the current flowing through a single pulldown of the appropriate size. This means that the formulas for the propagation delay through an inverter are directly applicable to more complex logic gates.

3.5. Propagation delay: source-followers and pass transistors

The analysis which follows is not very rigorous; its purpose is to show that the RSIM models for logic gates overestimate the propagation delay through a circuit containing pass transistors and source-followers. Although better estimates would be desirable, the existing models are sufficient given the relatively constrained use of these components in actual circuits.

A source-follower (so called because the voltage of the source node "follows" the voltage of the gate node) is an n-channel device with its drain connected to VDD.

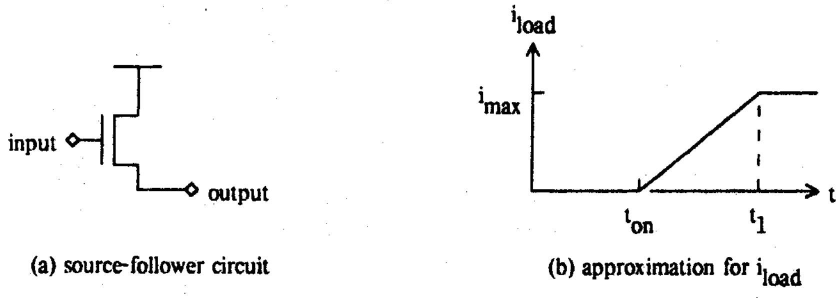


Figure 3.17. *Source-follower circuit configuration*

In the circuit shown in figure 3.17(a), the output voltage of the source-follower cannot rise higher than a threshold drop below the voltage of its input. Thus, the maximum voltage for the output of a source-follower is $1 - v_{te}$; this is why a depletion pullup (which can drive its output to VDD) is preferred in an ordinary logic gate.

Since a source-follower can only pull a node up, only the propagation delay associated with the low-to-high output transition needs to be analyzed. (A rising output transition corresponds to a rising input transition; unlike most logic circuits, a source-follower does not invert the sense of its input). During a very slow input transition, the output voltage tracks the input voltage, and the propagation delay is equal to the time needed for the input to rise from v_{thresh} to $v_{thresh} + v_{te}$. For a ramp input, this implies $t_{plh} = (v_{te})\delta = (0.14)\delta$ where δ is the time needed for the input to rise from 0 to VDD.

For a fast input transition — one where the input reaches 1 before the output reaches v_{thresh} — the current through the source-follower can be approximated as shown in figure 3.17(b). t_{on} is the time at which $v_{in} = v_{te}$, and t_1 is the time at which $v_{in} = 1$. i_{max} is estimated by the average current flowing through the source-follower during the transition:

$$i_{max} = \frac{\kappa_{sf}}{2}(1 - v_{te} - \frac{v_{thresh}}{2})v_{thresh} \tag{3.33}$$

One can calculate t_{plh} using an approach similar to that of section 3.4.2; the result is

$$t_{plh} = R_{sf}C_{load} + \frac{1}{2}(t_1 + t_{on}) - t_{input} \tag{3.34}$$

where $R_{sf} = \frac{v_{thresh}}{i_{max}}$. If the input is assumed to be a voltage ramp with transit time δ, the final equation for t_{plh} is

$$t_{plh} = \begin{cases} R_{sf}C_{load} + (0.35)\delta & (small\,\delta) \\ (0.14)\delta & (large\,\delta) \end{cases} \tag{3.35}$$

A source-follower is usually used to drive a large output load, so when δ is small, the RC term dominates. This suggests that the two pieces of the equation can be reconciled as

$$t_{plh} = R_{sf}C_{load} + (0.14)\delta \tag{3.36}$$

This equation is very similar to 3.23, which describes t_{plh} for an ordinary logic gate, so no special handling is needed for a source-follower.

In the analysis of section 3.4 and the first part of this section, each examined device had essentially two terminals, since one terminal of each device connected to VDD or GND. Moreover, input signals were applied to the gate node of the device. The analysis now turns to circuits that contain three-terminal components, *i.e.*, pass transistors. A pass transistor is any transistor not configured as a pulldown, pullup, or source-follower; some examples of circuits containing pass transistors are presented in section 3.2.

There are two basic configurations for a pass transistor: one with the gate node as input, and the source and drain as outputs; the other with the source/drain as input, and the drain/source as output (assuming that the gate is at logic high†). As the following table shows, when the gate of a pass transistor is the input, the pass transistor behaves like one of the components analyzed earlier.

input (gate)	*source or drain*	*pass device acts as*	*analyzed in*
falls	rises	pulldown turning off	section 3.4.2
falls	falls	enhancement pullup turning off	—
rises	falls	pulldown turning on	section 3.4.3
rises	rises	source-follower	beginning of this section

The second pass transistor configuration presents a new analysis problem. Assume that the drain connection is the input (which remains constant) and that the source node undergoes a transition. If the drain undergoes a step transition from high to low at time 0, and the source follows, [Horowitz83] suggests the best estimate for the voltage of the source is

$$v_{source}(t) = 1 - \tanh\left(\frac{t}{R_{pass}C_{load}}\right) \tag{3.37}$$

†Although the analysis focuses on n-channel pass transistors, it can be extended to p-channel pass transistors in a straightforward manner.

This equation can rearranged to give the propagation delay:

$$t_{phl} = R_{pass} C_{load} \tanh^{-1}(1 - v_{thresh}) = (0.63) R_{pass} C_{load} \tag{3.38}$$

Similarly, Horowitz suggests the best estimate for the voltage at the source, given a rising step at the drain, is

$$v_{source}(t) = 1 - \frac{1}{\dfrac{t}{R_{pass} C_{load}} + 1} \tag{3.39}$$

which gives

$$t_{plh} = R_{pass} C_{load} \frac{v_{thresh}}{1 - v_{thresh}} = (0.79) R_{pass} C_{load} \tag{3.40}$$

In both cases, the RC time constant of the RSIM model overestimates the propagation delay of a step input. For a slow input transition, the source voltage tracks the drain voltage, resulting in essentially zero propagation delay. (In this respect, the delay through a pass transistor is similar to the delay through a logic gate.) Although no direct evidence is provided here, the circumstantial evidence indicates that the predictions for propagation delay through a logic gate are upper bounds for the propagation delay through a pass transistor, regardless of the speed of the input transition.

Pass transistors are often used in series within a switching-logic implementation of multiplexors, etc.

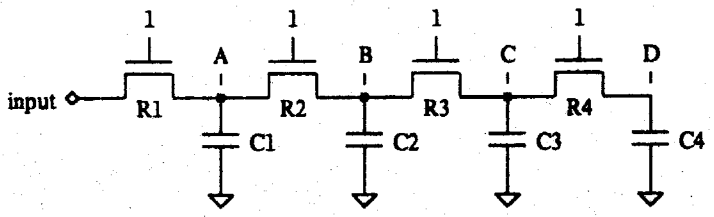


Figure 3.18. *Pass transistors connected in series*

Horowitz extends his estimates for the voltage of a particular node e to a chain of pass transistors by replacing the RC terms in equations 3.38 and 3.40 with

$$\tau_{De} = \sum_k R_{ke} C_k \tag{3.41}$$

where R_{ke} is the resistance of the path common to node e and node k. Thus, his estimate for the delay associated with a falling transition on node D of figure 3.18 is

$$t_{phl} = (0.63)[\, R_1C_1 + (R_1+R_2)C_2 + (R_1+R_2+R_3)C_3 + (R_1+R_2+R_3+R_4)C_4 \,] \tag{3.42}$$

If all the resistances are equal, and all the capacitances are equal, $t_{phl} = 6.3RC$. The RSIM estimate for the same transition is

$$t_{phl} = (\sum_k R_k)(\sum_k C_k) = 16RC \tag{3.43}$$

which overestimates the delay by a considerable margin. For a long chain of pass transistors, the RSIM estimate is very pessimistic; fortunately, performance constraints limit designers to chains of length four or less. Nevertheless, performance prediction for a circuit containing pass transistors is clearly an area where RSIM can be improved.†

3.6. Implications for the RSIM model

The analysis of the propagation delay of logic gates indicated that an RC time constant is a very good estimate for the delay of a gate when the input waveform is a voltage step. The analysis concludes that a simple RC time constant underestimates the actual propagation delay if the input waveform is assumed to be a voltage ramp with a rise/fall time of δ. More accurate estimates for the propagation delays are

$$\begin{aligned} t_{plh} &\le R_{pu}C_{load} + \Delta_{in:fall} \\ t_{phl} &\le R_{pd}C_{load} + \Delta_{in:rise} \end{aligned} \tag{3.44}$$

where

$$\begin{aligned} \Delta_{in:fall} &= \frac{1}{2}(v_{thresh} - v_{te})\delta \approx (0.15)\delta \\ \Delta_{in:rise} &= \frac{1}{2}(1 - v_{thresh})\delta \approx (0.28)\delta \end{aligned} \tag{3.45}$$

are offsets that depend only on parameters of the input waveform. Section 3.5 shows that these equations are satisfactory upper bounds on the propagation delay through other (non-gate) circuit configurations.

†It is straightforward modification of RSIM to make it use equations 3.38 and 3.40 instead of the lumped RC formula. However, these equations only apply to circuits containing a single driver; until the theory is extended to include multiple-driver configurations, it seems safest to use the conservative lumped RC approximation.

The computation of the propagation delay would be easier if it involved only the τ of the output node. A rearrangement of the time accounting accomplishes this:

(1) Report the time of the output transition as happening at τ time units after the input transition.

(2) Schedule the event associated with the output transition for $\tau + \Delta$ time units after the input transition where $\Delta = (0.28)(\textit{total rise time})$ for rising transitions, and $\Delta = (0.15)(\textit{total fall time})$ for falling transitions.

In other words, the effects of the input rise/fall time are factored in when the input transition is scheduled, so the Δ terms in equation 3.44 can be omitted when computing subsequent τ's. This rearrangement is illustrated in the following figure.

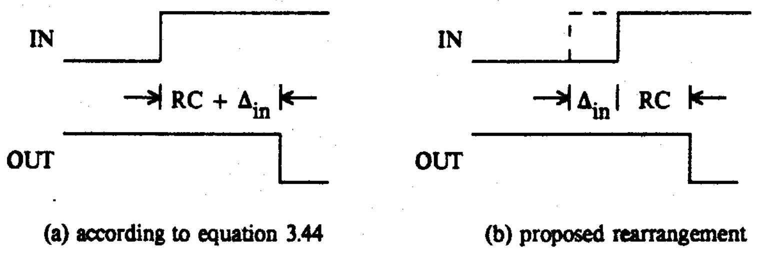


Figure 3.19. *Rearrangement of time accounting for transitions*

The total rise and fall times of a transition are related to the RC time constant of the transition. When the input is modeled as a ramp, the total rise/fall time is $(2.3)\tau$ since τ is measured using $v_{thresh} = 0.44$. As a result, the transitions of a given node can be handled in the following way:

(1) Compute the RC time constant (τ) for the node.

(2) Report the time of the transition as τ time units in the future.

(3) Schedule the associated event at
$(1.6)\tau$ time units in the future for a rising transition, or
$(1.3)\tau$ time units in the future for a falling transition.

Note that $1.6 = 1 + (2.3)(0.28)$ and $1.3 = 1 + (2.3)(0.15)$. This scenario assumes that all consequences of a rising transition involve a falling transition, and *vice versa*. This is not always the case for a source-follower or a pass transistor, but the error involved (the difference between 1.6 and 1.3) is not large enough to be significant. The old scheme (accounting for the input transition time during each delay computation) can be used if desired.

Now that the model incorporates some information about the input waveform, it is interesting to review the examples presented in section 2.4. First the PLA calculations:

	node	transition	τ	RSIM predicts transition	SPICE predicts transition	RSIM schedules event
	A	↓	0.3	0.3	0.8	0.4
	B	↑	3.7	4.1	3.5	6.3
Case 1	C	↓	0.9	7.2	6.8	7.5
	D	↑	9.1	16.6	15.5	22.1
	E	↓	0.9	23.0	20.7	—
	A	↑	1.6	1.6	0.6	2.6
	B	↓	0.8	3.4	1.9	3.6
Case 2	C	↑	0.6	4.2	3.3	4.6
	D	↓	1.3	5.9	6.4	6.3
	E	↑	6.0	12.3	12.1	—

As one can see, RSIM's estimates are now better, and they overestimate transition times with reasonable consistency. (One expects overestimates because of the inequality in equation 3.44). The estimate for Case 1 is 11% greater than the SPICE prediction; for Case 2, 2% greater. The story is similar for the OM2 data path example:

node	transition	τ	RSIM predicts transition	SPICE predicts transition	RSIM schedules event
A	↓	2.4	2.4	2.6	3.1
B	↑	8.2	11.3	9.1	16.2
C	↑	2.7	18.9	19.6	23.2
D	↓	22.6	45.8	39.6	—

RSIM's prediction is 15% greater than that of SPICE. Note that the event for node B is scheduled using the rule for a rising transition — formulated assuming that any consequent transitions will be falling — even though node C is also undergoing a rising transition. This accounts for much of the overestimate by RSIM.

In conclusion, this chapter shows justification for the linear transistor model, especially if all waveforms can be modeled as steps. Of course, transitions are not steps in actual circuit operation; this fact motivated changes to the linear model, still allowing it to provide acceptable predictions of circuit behavior.

CHAPTER FOUR

Simulation Using a Linear Network Model

This chapter focuses on various RSIM implementation issues. The first section presents a detailed description of the simulation algorithm, with step-by-step accounts of the charge-sharing and final-value computations. Several techniques for speeding up the computations are described in the second section. The third section outlines some mechanisms available to the user for forcing the value and timing predictions for given nodes. The chapter concludes with an evaluation of the strengths and weaknesses of RSIM.

4.1. The RSIM simulation algorithm

RSIM uses the following simple recipe for simulating a circuit:

(i) Accept new input values from the user. Perform the new-value computation (figure 4.2) for each new input value; this propagates the new value to nodes connected to the input by the source/drain connection of a transistor switch (see figure 2.14(a)). In addition, schedule the appropriate event so that any transistors affected by the new input value will be processed.

(ii) Process events from the event list, stopping (1) when the event list is empty, (2) when a node the user is tracing changes value, or (3) when the specified amount of simulated time has elapsed.

(iii) Loop back to (i) to accept new inputs.

The main loop of the simulator (step (ii) above) is described in the following figure. The node associated with each event is assigned its new value, and all stages affected by the new value are located and processed. (An affected stage is one that contains a source/drain node — called a *seed node* — of a transistor which has the event node as their gate.) The processing of a stage has two steps: first a charge-sharing computation for the stage, then a calculation of the final value of each node in the stage. Before each of the two steps, the COMPUTE flag of each seed node is set to indicate that the stage containing the seed node needs processing. A stage is processed only if its seed node has the COMPUTE flag set; as part of the processing, COMPUTE flags for nodes in the current stage are reset. This mechanism ensures that a stage is processed only once, even if it contains more than one seed node.

```
while event list not empty {
    n := node associated with first event on event list
    remove first event from event list
    set n's value to the value specified by the event

    /* do charge-sharing computation for each affected stage [see section 4.1.1] */
    for each transistor with n as gate node, set COMPUTE flag for source
    for each transistor t with n as gate node
        if t has just turned on and COMPUTE still set for source node
            do charge-sharing computation for source

    /* do new-value computation for each affected stage [see figure 4.2] */
    for each transistor with n as gate node, set COMPUTE flag for source and drain
    for each transistor with n as gate node {
        if COMPUTE still set for source, do new-value computation for stage containing source
        if COMPUTE still set for drain, do new-value computation for stage containing drain
    }
}
```

Figure 4.1. *Main loop of RSIM algorithm*

Note that the charge-sharing computation deals only with the source stage of each transistor, but the final-value computation deals with both the source and drain stages. This is because the charge-sharing calculation only deals with transistors known to be on; therefore, the source and drain belong to the same stage, and a stage computation involving the source automatically involves the drain.

The procedure for calculating the final value for each node in a stage is outlined in the following figure.

```
initialize connection list to have starting node as only element
set pointer to beginning of connection list
if starting node is an input, input_found := true, else input_found := false

/* find all nodes in current stage */
while pointer not at end of connection list {
    n := node currently pointed at
    for each "on" transistor with source connected to n {
        if drain is an input, input_found := true
        else if drain not on connection list, add drain to end of list
    }
    advance pointer to next list element
}

/* compute new final value for each node in stage */
if no inputs found, all done (charge-sharing has computed the correct value)
else for each node on connection list {
    if node is an input, do nothing (its value is set by user)
    compute final value for node [section 4.1.2]
    reset VISITED flag (set by final-value computation) for each node on connection list
    reset node's COMPUTE flag
}
```

Figure 4.2. *Subroutine to compute new final value for every node in stage*

The details of the charge-sharing and final-value computations are presented in the next two subsections, followed by a description of event management in RSIM.

4.1.1. Charge-sharing computation

When a transistor turns on, its source and drain nodes become part of the same stage. As explained in section 2.2, if the voltages of all the nodes in a stage are not already identical, they become so through charge sharing. In order to calculate the charge-sharing value for each node, RSIM computes three summary capacitances from the capacitances of each node in the stage:

C_{high}	total capacitance of nodes with current state of logic high.
C_{low}	total capacitance of nodes with current state of logic low.
C_x	total capacitance of nodes with current state of X.

The summary capacitances are used to compute the charge-sharing value for the stage, as specified by equations 2.3 and 2.4:

$$\textit{charge-sharing value} = \begin{cases} 0 & \dfrac{C_{high} + C_x}{C_{low} + C_{high} + C_x} < v_{low} \\ 1 & \dfrac{C_{high}}{C_{low} + C_{high} + C_x} > v_{high} \\ X & \textit{otherwise} \end{cases} \qquad (4.1)$$

An event is scheduled for each node, specifying an immediate transition to the charge-sharing value. (See section 4.1.3 to find out what happens to new events.)

The charge-sharing computation is outlined in the following figure. The procedure performs a tree walk of a stage starting with a node passed as an argument from the new-value procedure. Since the nodes in the stage do not require processing in a particular order, the procedure is implemented without recursion.

```
initialize list to have starting node as only element
set pointer to beginning of list
reset capacitance accumulators

/* visit all nodes in stage, compute summary capacitances */
while pointer not at end of list {
    n := node currently pointed at
    add capacitance of n to appropriate accumulator
    for each "on" transistor t with source connected to n {
        if drain is an input or static(t) > maxres, do nothing
        else if drain not on list, add drain to end of list
    }
    advance pointer to next list element
}

/* make each node in stage have charge-sharing value */
compute charge-sharing value using equation 4.1
for each node on list {
    reset node's COMPUTE flag
    schedule immediate transition to charge sharing value
}
```

Figure 4.3. *Non-recursive routine for charge-sharing computation*

If the resistance of a transistor is large enough, its source and drain nodes might not share charge — at least not very quickly. The user can specify a maximum resistance parameter (*maxres*) that controls the scope of the charge-sharing calculation; the traversal of nodes in a stage stops at transistors with a resistance greater than *maxres*. The COMPUTE flag indicates to the main RSIM loop which stages have been processed by the charge-sharing calculation; the main loop uses the flag to ensure that the charge-sharing calculation is performed only once for each stage.

Equation 4.1 leads to incorrect results when the surrounding network contains X transistors (transistors with gates of X). A portion of the network that can be reached only through X transistors might not be connected to the original node at all, and so should not make an active contribution to the node's charge-sharing value. An alternative (suggested by Dave Gross) is the use of capacitance intervals to accumulate the contribution of X connections. In this scheme, the capacitance accumulators have interval values, *e.g.*, $C_{high} = [C_{high}.min, C_{high}.max]$. The minimum value is the total capacitance of nodes guaranteed to be connected to the current node; the maximum value also includes the capacitance of nodes only reachable by X transistors. A separate charge-sharing computation occurs for each node in the stage, as outlined in the following figure.

```
if node is input, $C_{high} = C_{low} = C_x = [0,0]$
else {
    local_$C_{high}$ := local_$C_{low}$ := local_$C_x$ := [0,0]
    add node's capacitance to max and min of accumulator for node's value
    set VISITED flag for current node
    for each "on" transistor, t, with source connected to current node {
        if drain does not have VISITED flag set {
            recursively determine parameters for drain node
            if value of gate node for t is not X {
                local_$C_{high}$.min := local_$C_{high}$.min + $C_{high}$.min
                local_$C_{low}$.min := local_$C_{low}$.min + $C_{low}$.min
                local_$C_x$.min := local_$C_x$.min + $C_x$.min
            }
            local_$C_{high}$.max := local_$C_{high}$.max + $C_{high}$.max
            local_$C_{low}$.max := local_$C_{low}$.max + $C_{low}$.max
            local_$C_x$.max := local_$C_x$.max + $C_x$.max
        }
    }
    set $C_{high}$ = local_$C_{high}$, and so on
}
```

Figure 4.4. *Subroutine to compute capacitance intervals*

The results determine the maximum and minimum node voltage, which determine the charge-sharing value for the node:

$$\textit{charge-sharing value} = \begin{cases} 0 & \dfrac{C_{high}.max + C_x.max}{C_{low}.min + C_{high}.min + C_x.min} < v_{low} \\ 1 & \dfrac{C_{high}.min}{C_{low}.max + C_{high}.max + C_x.max} > v_{high} \\ X & \textit{otherwise} \end{cases} \tag{4.2}$$

Capacitances for nodes connected by X transistors contribute to the final value only in a negative sense, *i.e.*, they may cause a node to go to X, but never contribute to a value of 0 or 1. Leaving the VISITED flag set as each new node is discovered ensures that each node is visited only once. After completing the charge-sharing computation for a node, its COMPUTE flag is reset; the VISITED flags for all nodes in the stage are also reset, in preparation for the next node's computation.

One disadvantage of the interval approach is that a separate calculation is performed for each node in the stage, whereas the original scheme required only one calculation per stage. In addition, the interval calculation must be performed by a recursive tree walk to ensure the correct handling of X transistors. Fortunately, this computation can be merged with the tree walk described in the following section, so the incremental cost is fairly small.

4.1.2. Final-value computation

The final, driven value of a node is determined by the resistance of paths from the node to various inputs. As we saw in chapter 2, a convenient way to characterize these paths is to calculate the Thevenin equivalent for the portion of the network that can be reached from the node of interest. Equation 2.6 relates the final value of a node to V_{thev}, the Thevenin equivalent voltage. The time constant for a transition in the value of a node is also determined by the surrounding network; the necessary parameters can be computed during the Thevenin calculation.

For computational convenience, RSIM actually computes RH and RL, the resistances of a resistor divider that represents the effect of the surrounding network.

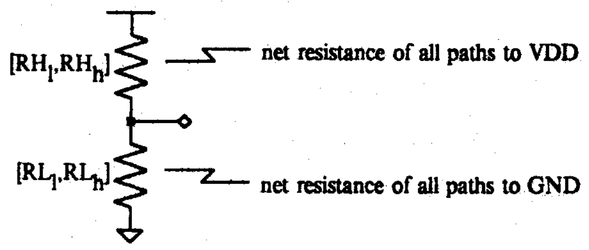

[RH$_l$,RH$_h$]
net resistance of all paths to VDD
[RL$_l$,RL$_h$]
net resistance of all paths to GND

Figure 4.5. *characteristic resistor divider for a node*

RH and RL might be resistance intervals ($RH = [RH_l, RH_h]$ and $RL = [RL_l, RL_h]$) if there are X values in the surrounding network. The Thevenin equivalent voltage is easily calculated from the characteristic divider:

$$V_{thev} = [\ V_l, V_h\ ] = [\ \frac{RL_l}{RL_l + RH_h}, \frac{RL_h}{RL_h + RH_l}\ ] \tag{4.3}$$

For example, the lowest possible voltage is calculated using the least resistance to GND (specified by RL_l) and the greatest resistance to VDD (specified by RH_h). Couching the computation in terms of the characteristic resistance is advantageous for several reasons. Resistances to VDD and GND represent, in a natural way, the connections made by MOS logic, as shown in chapter 3. With the aid of some simple rules, it is easy to incrementally analyze any MOS network in terms of its component resistances. Because resistances are directly related to the implementation, they can represent certain circuit configurations — *e.g.*, short circuits ($RH = RL = 0$) — that cannot be simply characterized using the Thevenin equivalent. The remainder of the section describes a tree walk algorithm to compute the parameters needed for determining a node's value and for scheduling the appropriate transition.

The computation of RH and RL proceeds by tracing paths to the inputs that are reachable from the node of interest, and then calculating the resistance of each path, starting at the input and working back toward the original node. Two rules are helpful for calculating path resistance. The first rule specifies the apparent path resistances when a divider exists on the other side of a resistor:

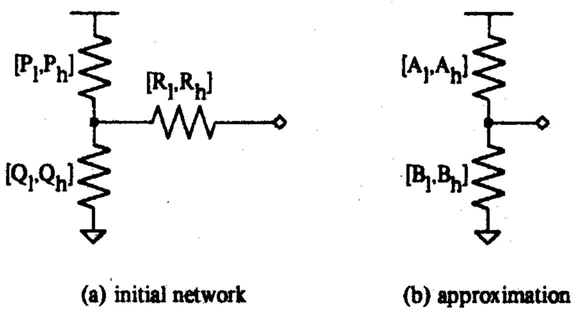


Figure 4.6. *Reduction rule for resistor divider with series resistor*

The parameters for the apparent resistances (A and B in figure 4.6(b)) cannot be determined exactly, an approximation is therefore necessary. Appendix 3 explains why this is so, and derives the following formulas for the approximation:

$$\begin{aligned} A_l &= P_l + R_l + R_l\frac{P_l}{Q_l} & A_h &= P_h + R_l\frac{P_h}{P_l} + R_l\frac{P_h}{Q_l} \\ B_l &= Q_l + R_l + R_l\frac{Q_l}{P_l} & B_h &= Q_h + R_l\frac{Q_h}{Q_l} + R_l\frac{Q_h}{P_l} \end{aligned} \tag{4.4}$$

The second rule is much simpler; it indicates how to merge the resistances of two separate paths to obtain the net resistance for both paths:

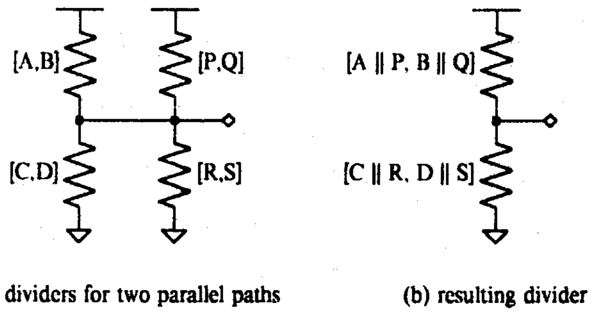


Figure 4.7. *Reduction rule for combining two parallel paths*

To compute the Thevenin equivalent for a particular node, one starts by locating all conducting transistors connected to that node and then recursively analyzing the network on the other side of each of the transistors. Each node is marked as its analysis begins; recursive calls ignore portions of the network involving marked nodes. This keeps the analysis expanding outward, eventually terminating at a dead-end (no paths leading to unmarked nodes) or an input. These particular circuits are easy to analyze, as shown in the following figure.

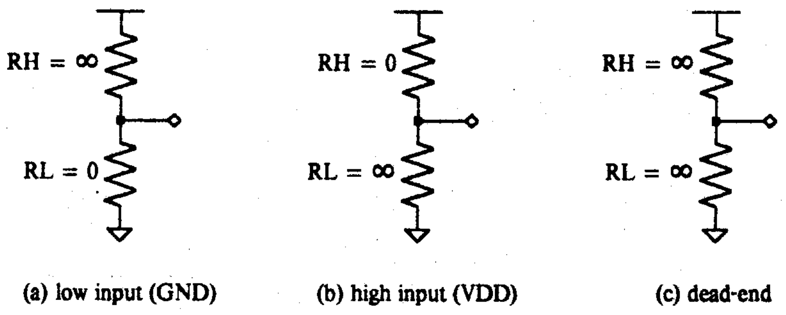
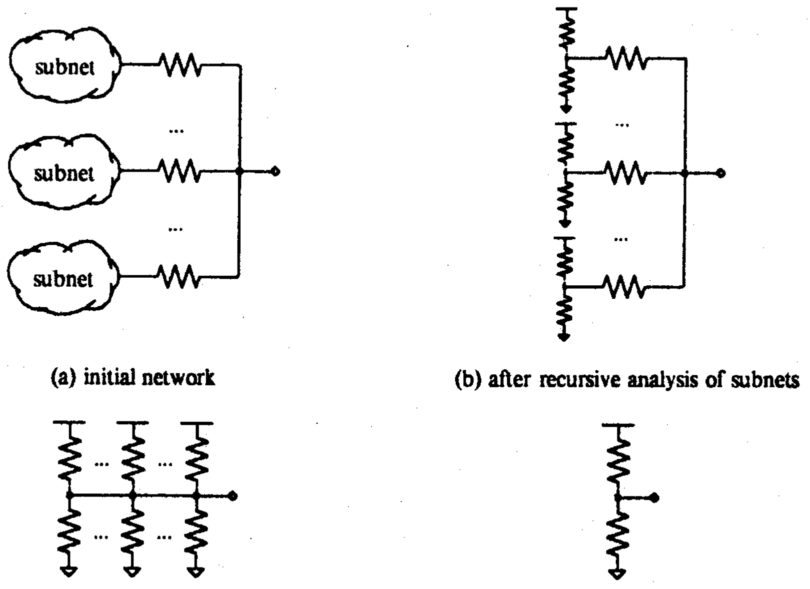


Figure 4.8. *Characteristic dividers for input nodes and dead-ends*

The resistance of paths leading from a particular node are combined using the two reduction rules above. Using the first rule, the results of a recursive call (shown as P and Q in figure 4.6) are combined with the resistance of the conducting transistor leading to that piece of the network (shown as R), to yield the net resistance of the path. This resistance is combined with the resistances from other recursive calls using the second reduction rule. When all paths have been accounted for, the analysis for the node is complete. The resulting divider is the desired answer, or, is used as part of the analysis of some other node if the analysis was performed because of a recursive call. The process is diagramed in the following figure.



(a) initial network (b) after recursive analysis of subnets

(c) after applying first reduction rule (d) after applying second reduction rule

Figure 4.9. *Network analysis by repeated rule application*

The complete analysis procedure is outlined in the next figure. The results are stored in eight global variables:

RH	resistance interval for net resistance of all paths to VDD. Path resistance computed using static resistance of each transistor.
RL	resistance interval for net resistance of all paths to GND. Path resistance computed using static resistance of each transistor.
R_{vdd}	net resistance to VDD, computed using the dynamic-high resistance of each transistor. Simple series/parallel calculation; paths containing X transistors are ignored.
R_{gnd}	net resistance to GND, computed using the dynamic-low resistance of each transistor. Simple series/parallel calculation; paths containing X transistors are ignored.
R_X	net resistance to all inputs, computed using the dynamic-high resistance to high inputs, and dynamic-low resistance to low inputs. Simple series/parallel calculation; includes paths containing X transistors.
C_{high}	total capacitance of nodes with current state of logic high.
C_{low}	total capacitance of nodes with current state of logic low.
C_X	total capacitance of nodes with current state of X.

If the interval charge-sharing calculation is merged with this calculation, the upper limit of the capacitance intervals in the charge-sharing calculation can be used in place of the three capacitance accumulators just defined. The procedure also uses four stack-allocated local variables to accumulate the first four quantities listed above, during the calculation for each node.

```
if node is logic low input {
    return with RH = $R_{vdd}$ = $\infty$ and RL = $R_{gnd}$ = $R_x$ = 0
} else if node is logic high input {
    return with RH = $R_{vdd}$ = $R_x$ = 0 and RL = $R_{gnd}$ = $\infty$
} else {
    local_$R_{vdd}$ := local_$R_{gnd}$ := local_$R_x$ := local_RH := local_RL := $\infty$
    add node capacitance to appropriate accumulator
    set VISITED flag for current node
    for each "on" transistor, t, with source connected to current node {
        if drain does not have VISITED flag set {
            recursively determine parameters for drain node
            combine static(t) with RH and RL using first reduction rule
            combine result with local_RH and local_RL using second reduction rule
            if value of gate node for t != X {
                local_$R_{vdd}$ := local_$R_{vdd}$ || (dynhigh(t) + $R_{vdd}$)
                local_$R_{gnd}$ := local_$R_{gnd}$ || (dynlow(t) + $R_{gnd}$)
            }
            local_$R_x$ := local_$R_x$ || (min(dynhigh(t),dynlow(t)) + $R_x$)
        }
    }
    set $R_{vdd}$ = local_$R_{vdd}$, RH = local_RH, and so on
}
```

Figure 4.10. *Subroutine to compute parameters of resistor divider*

Marking each node as it is visited (by setting its VISITED flag) avoids cycles and keeps the tree walk expanding outward from the starting node. If the network does contain cycles, the subroutine only approximates the true resistance to VDD and GND. For example, consider the following logic gate where the output (the pulled-up node) is the node of interest:

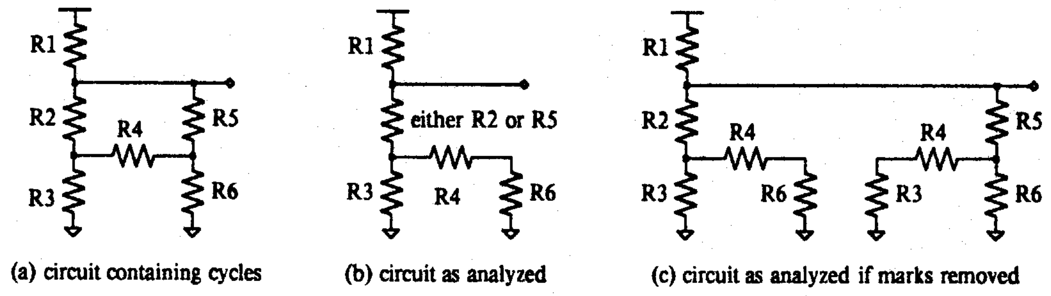

R1
R2
R3
R4
R5
R6
(a) circuit containing cycles
either R2 or R5
(b) circuit as analyzed
(c) circuit as analyzed if marks removed

Figure 4.11. *Analysis of circuit containing cycles*

Since the marks are not removed when the analysis of a path is completed, RSIM treats the cycle as if the circuit were configured as shown in the circuit in figure 4.11(b). This approximation results in an overestimate of the actual resistances. If a node's mark were removed as the procedure exited, all paths through the network would be explored (as shown in figure 4.11(c)); in this case, the resistance would be underestimated, leading to optimistic performance predictions.

Cycles are relatively rare in nMOS designs; when they occur, the extra path is often redundant, *i.e.*, the circuit is designed to work correctly if any path in the cycle became the sole connection. This means the approximation used by RSIM is usually not out of line with the designer's intentions. This statement holds for cMOS as well, with one notable exception — the cMOS pass gate:

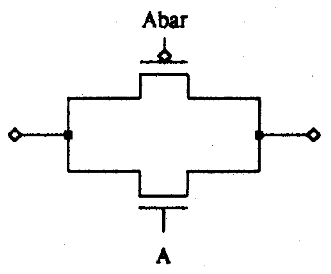


Figure 4.12. *A cMOS pass gate*

In this circuit configuration, one device is sized to carry most of the load, and the other exists simply to ensure no threshold drop across the gate. In analyzing such a circuit, RSIM arbitrarily chooses the transistor that makes the connection; the other transistor's contribution is ignored. This is satisfactory if the transistor with the smaller resistance is chosen, but such is not always the case. To correct the problem, the transistor list for each node can be arranged in order of increasing resistance; this ensures paths of least resistance are examined and marked first. Note that this solution only works when the paths in a cycle have a length of one transistor (as in the pass gate above). If the paths are longer, there is no guarantee that the path of least total resistance will happen to start with the transistor that has the least resistance.

After the various parameters are calculated, the final value of a node can be calculated using equations 2.6 and 4.1:

$$\textit{final value} = \begin{cases} 0 & V_h < v_{low} \textit{ or } (\textit{old value}=0 \text{ and } RH_l=\infty) \\ 1 & V_l > v_{high} \textit{ or } (\textit{old value}=1 \text{ and } RL_l=\infty) \\ X & \textit{otherwise} \end{cases} \tag{4.5}$$

The extra clause for "0" and "1" values prevents a node from being unnecessarily forced to X when it has no connection to inputs of the opposite logic state. The appropriate event is scheduled $R_{eff}C_{eff}$ seconds in the future, where

$$R_{eff} = \begin{cases} R_{gnd} & \mathit{final\ value} = 0 \\ R_{vdd} & \mathit{final\ value} = 1 \\ R_x & \mathit{final\ value} = X \end{cases} \tag{4.6}$$

$$C_{eff} = \begin{cases} C_{high} + C_x & \mathit{final\ value} = 0 \\ C_{low} + C_x & \mathit{final\ value} = 1 \\ C_{low} + C_{high} & \mathit{final\ value} = X \end{cases} \tag{4.7}$$

The disposition of this event depends on the nature of any pending events and the node's current value; see section 4.1.3 for the details of event management.

The user has some control over the final-value computation. The time constant for event scheduling can be forced to 1, implementing a unit-delay simulation. This is useful when a node value is to be calculated using transistor resistances, but transition timing is not important. Another option is flagging those events corresponding to transitions to X, where the X value is specifically caused by a ratio error (rather than other X's in the network). Such transitions are characterized by $RH_h < \infty$ and $RL_h < \infty$; if an X exists in the surrounding network, one or both of these parameters is infinite. When a flagged event is processed, the transition is reported to the user as a ratio error. Because the error report is delayed until the flagged event is processed, short-lived ratio errors (those caused by small differences in propagation delays) are ignored, and the error reports reflect only significant ratio errors. Of course, in some designs, even long-lived ratio errors might not affect correct circuit operation, so the reporting is optional.

When $RH_l = RL_l = \infty$, the node is not connected to any inputs, and the charge-sharing computation described in the previous section correctly computes the node's final value. Ordinarily, the final-value calculation does not schedule any events in this case, but the user can optionally request the scheduling of a charge-decay event. A charge-decay event sets the node value to X after a specified interval which the user can set. At first glance, it might seem odd to schedule all decay events using the same interval; a more suitable estimate might be based on factors such as the node's capacitance, the number of transistors connected to the node, and so on. However, precise predictions are not necessarily the most useful here. The actual decay time for MOS circuits is in the millisecond range. Since it is unlikely that a simulation spans that long a period of simulated time, a precise accounting of the decay time never results in a decay! A more useful approach is based on the observation that a designer usually intends for all dynamic nodes to be refreshed every few clock cycles. When the decay time is set to an interval slightly larger than the intended refresh rate, the unrefreshed nodes decay quickly, and the user receives a suitable error report. Thus, even a short simulation run catches a decay problem. This type of debugging experiment can be much more effective than a precise estimate in pinpointing a problem.

4.1.3. Event Management

Up to two events can be pending for a node:

(1) a charge-sharing (CS) event. CS events are always immediate events, *i.e.*, they are scheduled for the current simulated time.

(2) a final-value (FV) event, scheduled for sometime in the future.

Thus, up to two transitions are possible for a given node. Each event corresponds to a real transition, *i.e.*, the new value of a CS event always differs from the current value of the node, and the new value of a FV event differs from that of the CS event (or the current node value if there is no pending CS event). Since only two transitions can be pending at any moment, newly calculated events must be merged with the pending events. Section 2.3 hinted at the issues involved; in general, RSIM makes its choices based on the principle that the most recently *calculated* event best reflects the current network configuration. Since no information is available that explains why any pending events were created, there is little (if any) reason to save a previously-calculated event in preference to the newer one.

The following figure describes the simple merging rules used by RSIM:

```
if merging new CS event {
    abort pending CS and FV events
    if new charge-sharing value is different from current node value
        schedule new CS event
}
if merging new FV event {
    if new value differs from CS value (or, if no CS event pending, current node value)
        schedule new FV event
}
```

Figure 4.13. *Merging a new event with pending events*

A new CS event aborts a pending FV event because a new final-value computation always occurs after the charge-sharing computations are complete. Although this approach is simple, it occasionally leads to pessimistic predictions. For example, if one input of a two-input NOR gate turns on substantially before the other, the propagation delay is actually determined by the time of the first input's transition. With the merging scheme outlined above, the two events scheduled at the time the second input turns on cause other events to be aborted — those scheduled because of the first input's transition. This occurs even if one of the aborted events is scheduled for an earlier time than the second event. In other words, with the merging scheme above, the propagation delay of a NOR gate might be incorrectly measured from the later input. There is no simple fix to the merging rules above that solves this problem. The correct solution requires knowledge of both the new CS event and the new FV event, so that pending events can be saved if they are compatible with both newer events. If the charge-sharing and final-value calculations are merged, as suggested at the end of section 4.1.1, it should be straightforward to implement the correct merging scheme.

There are several alternatives for dealing with aborted events. The simplest approach is to handle the event as if it were never scheduled, *i.e.*, do nothing. This is the approach RSIM adopts. Another approach is motivated by the physical significance of an aborted event. Since the signal changes between the transition start time (the time when the charge-sharing or final-value computation was performed) and the transition end time (the scheduled time of the event), the action of aborting the event corresponds to a stop in mid-transition. Aborted transitions are termed glitches [Thompson74]; these malformed signals sometimes have significant impact on the operation of a circuit and should be reported to the user. This report can be in the form of a forced transition to X, or just a simple error message. Interestingly, a user who has the option to receive glitch reports almost always disables that feature [Ulrich73]. The reason given is that the duration of an aborted transition is usually short enough so that the actual signal does not change significantly; hence no glitch actually occurs.†

Scheduling an event entails inserting it into the event list, placed according to its scheduled time. An event list implemented as a simple list would impose a noticeable scheduling overhead. RSIM adopts several techniques for reducing this overhead. It quantizes simulated time, and rounds off each event time to the nearest time quanta; in the current implementation, the time quanta is 0.1 nanosecond. The event list is implemented in two pieces:

(1) an *event array*. Each array element is a doubly-linked list of events for a particular time quanta.

(2) an *overflow list*, a doubly-linked list of events, sorted by event time.

This organization is similar to that found in many conventional gate-level simulators [Vaucher75,

†Some researchers propose showing transitions between logic states as 0-X-1 or 1-X-0, where the initial transition to X happens immediately. Thus aborted events leave the node value at X until some subsequent event re-establishes a legitimate logic state. This suggestion doubles the number of events in a simulation; a cost which might outweigh the advantages.

Ulrich76]. The event lists are doubly-linked to allow quick removal of an aborted event from the list. The data structures are diagramed in the following figure.

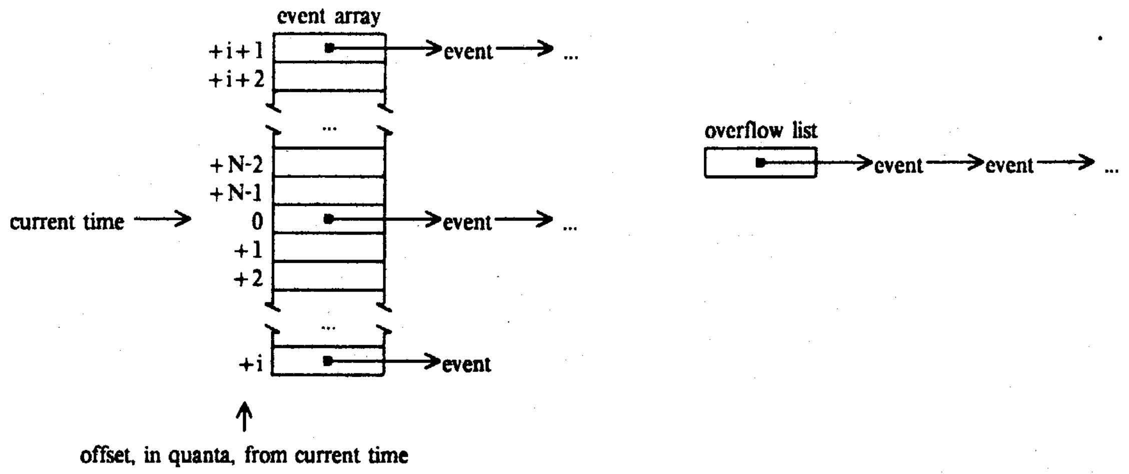


Figure 4.14. *The event list is implemented with an event array and overflow list*

The event array is managed as a circular buffer in which the N array elements hold events for the next N time quanta. An array index indicates which array element corresponds to the current simulated time. If a new event is scheduled for a time M quanta in the future, where $M < N$, the event is added to the end of the event list stored in array element $(index + M) \bmod N$; no sorting or searching is required. If $M > N$, the event is inserted into the overflow list according to its scheduled time. The array size is chosen so that most events are scheduled directly into the array. With a time quanta of 0.1 nanoseconds, a 128- or 256-element array captures most events in modern MOS designs. Note that events are added to the end of an event list. This ensures that events are processed in first-in, first-out order, *i.e.*, in the order created. Thus, cause-and-effect relationships are preserved.

To find the next event to process, the event array is searched starting at the current index, until an event is found. Each increment of the index corresponds to advancing simulated time by one time quanta. If the array is empty, simulated time is advanced to equal the scheduled time of the first event on the overflow list; this event becomes the next one to processed. When an event is located for processing, the overflow list is examined to find events whose scheduled times are less than N time quanta away from the new simulated time. Such events are moved from the overflow list to the appropriate list in the event array. This preserves the first-in, first-out event ordering mentioned above.

4.2. Speeding up the simulation

No simulator is fast enough. Increased simulator performance is always in demand, either to achieve faster turnaround during the design process, or to allow more complete testing during verification. This section discusses several techniques for improving the performance of the algorithms presented in the previous section.

It is not surprising to learn that, during event processing, most of the time is spent in the final-value calculation.† To compute the final value for a given node, the final-value computation must visit all the nodes in the current stage. Thus, if there are n nodes in the stage, processing the entire stage takes $O(n^2)$ time. Since the remainder of the processing is proportional to the size of the stage, the real bottleneck is the final-value computation. Performance can be improved by

(1) introducing a cache for final-value computations, with the intent of eliminating the recalculation of parameters for subnetworks.

(2) reducing the number of nodes in the stage.

(3) reducing the cost of each calculation, for example, by substituting integer arithmetic for floating-point. This alternative will not be discussed further, except to note that a 32-bit integer has over 9 orders of magnitude of dynamic range, sufficient for representing MOS resistances.

Clearly, the first improvement is most significant when n is large. The third improvement is important when n is small and the dominant cost is the actual arithmetic. The second improvement works on making (3) more important than (1). The improvements are discussed in turn below.

As it is currently formulated, the final-value procedure performs many redundant computations. Consider the circuit diagram for a 5-node stage shown in (a) below, and one of its subcircuits, shown in (b) below.

†The discussion in this section is limited to that portion of the simulator which propagates new values through the network. RSIM has an interpreted LISP-like command language which the designer uses to prepare new input values and process the results of a simulation step. Depending on the sophistication of the simulation environment built by the user, a substantial portion of the total time can be spent in the command language interpreter. Of course, there is room for improvement here too, but that is outside the scope of this thesis.

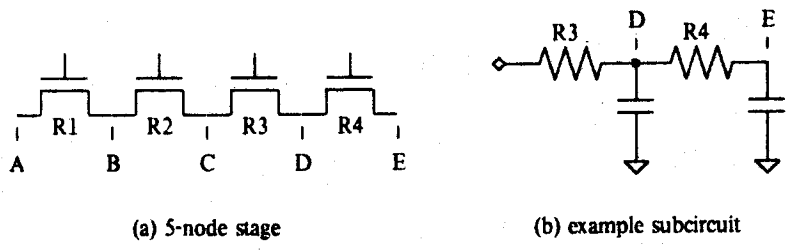


Figure 4.15. *Stage containing 5 nodes and 4 transistors*

When one traces the computations performed by the final-value procedure (see figure 4.10), it becomes apparent that the parameters for a specific subcircuit are calculated several times. The computations for nodes A, B, and C all need the same information about the subcircuit in figure 4.15(b); there is no reason to compute the information more than once.

The amount of redundant computation can be reduced by caching the result from each call to the final-value procedure.† Before each call, the cache is searched to see if the subcircuit was analyzed previously; if so, the results are taken from the cache and not recomputed. If the cache has constant access time, the cost of the final-value analysis for a stage is reduced to $O(n)$, a significant saving when n is large. In RSIM, the cache does not need to accommodate arbitrary amounts of information; associating two cache entries with each transistor (one for the source, one for the drain) is sufficient. The source cache retains the network parameters for the subnetwork connected to the drain node (including the transistor), and the drain cache is similar. When the analysis of a subnetwork is completed, the result is placed in the appropriate cache.

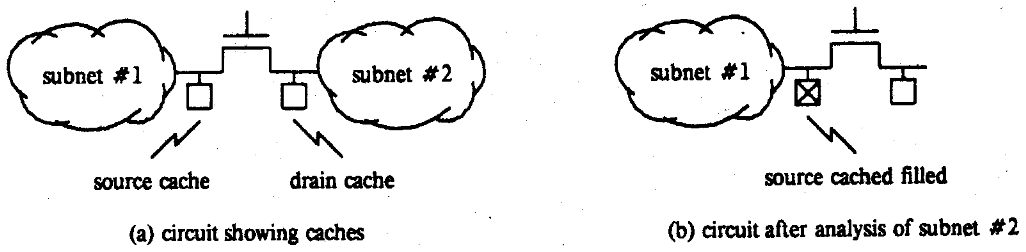


Figure 4.16. *Transistor cache scheme*

In the figure above, once subnet #2 has been analyzed and the result saved in the source cache, subsequent analyses involving the same transistor and subnet use the cached result. The following

†This caching technique is known in the LISP community as memoization.

figure shows the cache status after calculation of the final value for node D of figure 4.15(a).

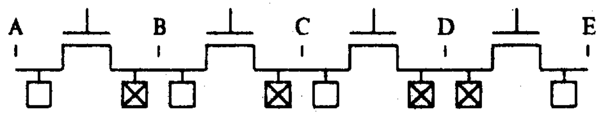


Figure 4.17. *Cache status after final-value calculation for node D*

Subsequent analysis of node C, for example, requires only a single recursive call (rather than four as before).

There are several reasons why the transistor cache might not be the ideal solution. The amount of information in each cache entry — 8 parameters — is quite large compared to the transistor data base. This suggests that cache entries should be dynamically allocated when needed, and returned when the computation is complete. The combined costs of storage management and cache access might exceed the cost savings realized on stages of modest size. These objections can be addressed by associating cache entries with nodes instead, or using the cache only when the stage exceeds a specified size.

However the cache is organized, its introduction has a substantial impact on the amount of computation required for the final-value analysis of a stage. Another improvement mentioned at the beginning of the section is reducing the number of nodes in a stage. The key element of this is the notion of useless nodes, *i.e.*, nodes that do not connect to any transistor gates and hence whose values are irrelevant. Such nodes commonly occur in a pulldown path containing more than one transistor, such as the node marked by an asterisk in figure 4.18(a).

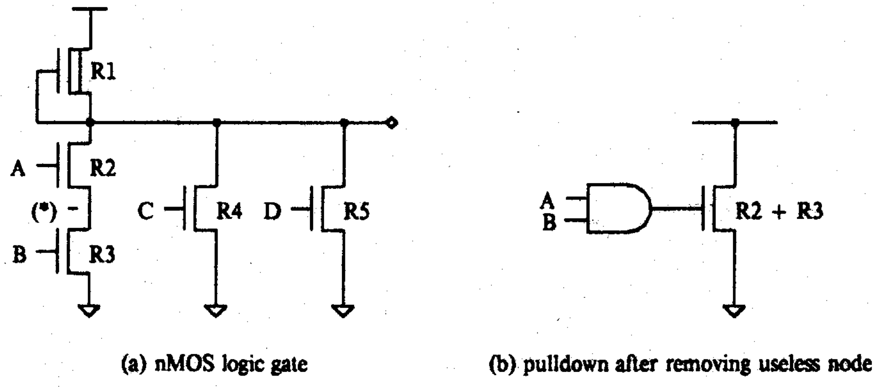


Figure 4.18. *Removing useless nodes from a stage*

Section 3.4.4 mentions that a pulldown with more than one transistor is electrically equivalent to a single-transistor pulldown of the appropriate size. This suggests that such a pulldown can be replaced by a circuit like the one shown in figure 4.18(b). All the nodes in the pulldown except the output and GND are eliminated, and all the pulldown transistors are replaced by a single transistor. The gate value of the single transistor is the logical conjunction of the values of the gates of the original pulldown chain. In fact, RSIM uses a compact representation for the generalized MOS gate:

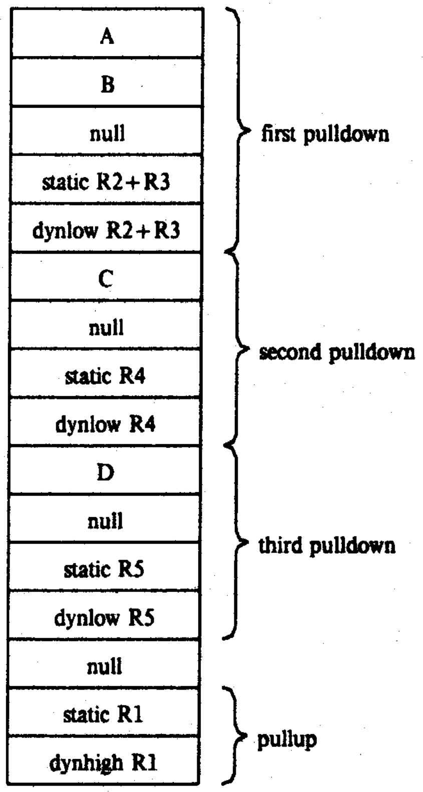


Figure 4.19. *Efficient internal representation of an nMOS logic gate*

All transistors and nodes that make up the gate are eliminated, and the resulting gate structure is associated with the output node. The output can still connect to other transistors that are not recognized as part of a logic gate; only those transistors that implement a MOS logic gate are compressed. The resistance parameters of a gate structure are computed very efficiently by RSIM — many times more quickly than the analysis of the equivalent network.

The compression of gate circuits into the compact internal representation also results in a considerable space saving. Somewhere between 40% and 80% of the transistors in most circuits are eliminated when the gate structures are built. This resulting simulation runs roughly twice as fast as the uncompressed network. This optimization is probably the single largest contributor to the ability of RSIM to deal with very large MOS circuits.

4.3. Escape mechanisms

Previous sections of this chapter introduced mechanisms that allow the user to adjust the operation of the simulator as a whole. There are occasions, however, when a large-scale adjustment is inappropriate, and only the predictions for a single node need correction. This section discusses several "escape" mechanisms provided by RSIM for adjusting the predictions for small groups of nodes and transistors.

The modifications discussed here are *ad hoc* in nature; their motivation arises from purely practical considerations. The mechanisms are not intended to allow wholesale changes in the simulation computation, but are provided so the designer can correct particularly egregious or far-reaching errors in the simulation of specific circuits. Since the mechanisms treat the symptoms and not the disease, their effectiveness is limited to local improvements.

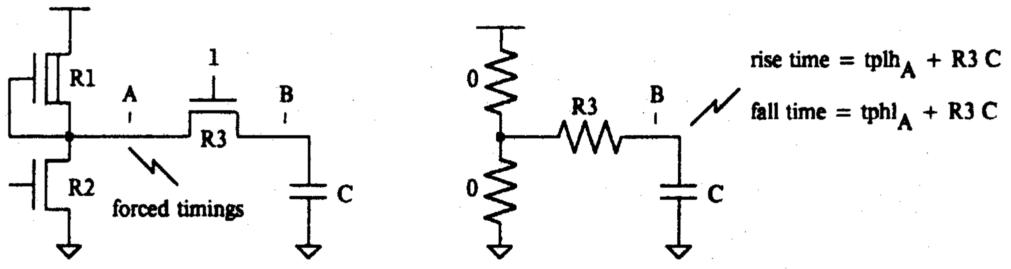
The are four user-adjustable parameters for each node:

VLOW	the logic low threshold for the node (specified in normalized voltage units).
VHIGH	the logic high threshold for the node (specified in normalized voltage units).
TPLH	the low-to-high transition time for the node (specified in time quanta).
TPHL	the high-to-low transition time for the node (specified in time quanta).

By adjusting the logic thresholds with VLOW and VHIGH, the user can prevent predictions of X values for circuits with non-standard pullup/pulldown ratios. This can be useful in a circuit where a node's voltage swing is reduced for performance or other reasons (for example, in input buffers or bit-lines of dynamic memory circuits).

The transition time parameters force the timing of all the node's transitions. These parameters allow adjustment of the timing of critical nodes to agree with predictions of circuit analysis programs. Clocks, for example, often are generated by special circuitry designed to drive the a capacitive load. Intricate timing chains involving bootstrapping, etc. increase the speed of clock distribution circuitry to acceptable levels. Most of these circuit techniques are beyond RSIM's ability to predict accurately; incorrect predictions for critical signals can throw off the whole simulation. Using the transition time parameters, the designer can force the rise and fall times of critical signals to their proper values, improving the quality of the remainder of the simulation.

It is obvious how transition time parameters affect the scheduling of events, but what about the timing of a node connected directly to a forced node by a source/drain connection? A workable scenario treats a node with forced timings as an input, setting its dynamic resistance (R_{vdd}, R_{gnd}, and R_X) and capacitance parameters to zero. (Note that the value calculation, which uses static resistances, is unaffected.) The transition time for a node connected to a forced node is the sum of the given transition time for the forced node and the RC time constant of the path from the forced node.



(a) original circuit with forced node (b) equivalent network for node B

Figure 4.20. *How forced timings affect neighboring nodes*

If a node is connected to more than one forced node, the smallest forced time constant is used. Neighbors of forced nodes always change value after the forced node — a reasonable prediction.

A much more powerful mechanism for forcing the desired prediction is modification of the circuit itself, replacing troublesome configurations with others that simulate correctly. Piecemeal modification of a large circuit can quickly lead to a loss of confidence in the simulation results, especially if the replacements are performed in a haphazard manner. On the other hand, the systematic identification and replacement of specific subcircuits, drawing from a library of approved replacements, offers the opportunity to improve simulation accuracy for common subcircuits.

The pattern matching/replacement program MATCH, written by John Iler [Iler83], provides an efficient way to systematically modify pieces of large circuits. The circuit to be modified is identified by a pattern specifying a prototype subcircuit. Each node in the prototype is given a type which controls what nodes it matches in the actual circuit:

(1) matched only by a circuit node with exactly the same connections specified in the pattern.

(2) matched by a circuit node with at least the connections specified in the pattern, but the circuit node may also have other connections.

(3) matched by a node with the same name.

The pattern indicates which prototype nodes attach to each transistor in the prototype, and can further constrain the match by giving an explicit size or resistance for each prototype transistor. The replacement can modify parameters of existing circuit components, and add or delete components. For example, the following figure shows a pattern and replacement for the bootstrap circuit discussed in section 3.2.

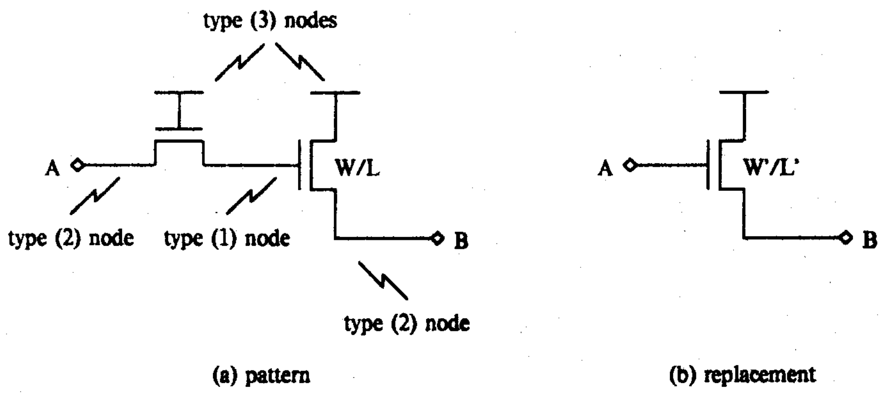


Figure 4.21. *Pattern/replacement for bootstrap circuit*

MATCH is regularly used in at least one industrial environment to improve the predictions of RSIM. Iler suggests other uses for the program: gathering of circuit statistics, identifying common circuit errors, and implementing circuit changes (ECO's) without requiring the regeneration of the entire netlist. MATCH has proved to be a handy tool.

4.4. An evaluation of RSIM

RSIM has simulated a large number of designs, both in university and industrial environments. Industrial designers are attracted to RSIM because of its ability to correctly predict the functionality of most MOS circuits without designer intervention — a unique capability in a logic simulator efficient enough to accommodate large designs. RSIM's timing estimates are helpful in locating gross timing errors in industrial designs, but the conservative nature of the estimates make them unsatisfactory for fine tuning critical circuitry. In short, RSIM allows the verification of large industrial designs, at a level of detail not obtainable with other simulators.

Timing estimates appear to be more important for academic users who, more often than not, have not paid as much attention to the performance of each individual circuit component. RSIM makes a good breadboard for locating performance bottlenecks and experimenting with potential solutions.

Since transition timings automatically reflect output loadings and device sizes, the naive user's attention is focused on critical portions of the design. RSIM is a good companion for the novice designer because of its ability to qualitatively model much of the behavior of MOS circuitry.

RSIM advances the state of the art of simulation in several ways. The linear model embodied by RSIM is a systematization of a common rule-of-thumb for estimating circuit performance. The simulator was originally developed simply to automate the calculation of RC time constants, and to reap the benefits of applying the same timing criteria uniformly to the entire circuit. The analysis of propagation delay in Chapter 3 justifies the use of the linear model as a simple approximation and extends the rule-of-thumb to include the affects of the input waveform timings on gate propagation delay. RSIM breaks new ground by combining logic-level simulation with the ability to automatically estimate transition times directly from the electrical properties of the circuit components. While the results are less accurate than circuit analysis, the designer is compensated by an increase in computation speed by several orders of magnitude. RSIM represents a first cut at a stylized form of circuit analysis which attempts to model the significant effects at far less cost than traditional analysis techniques. The proven utility of RSIM augurs well for further developments in the area between logic simulation and circuit analysis.

The introduction of intervals to characterize the operation of circuit components controlled by X-valued signals is a novel technique for merging electrical analysis with the logical concept of unknown signal values. The use of intervals allows one to easily compute the electrical consequences of unknown node values, resulting in predictions more satisfactory than those obtainable from conventional logic simulators or circuit analysis programs.

There is, of course, plenty of room for improvement in RSIM! For example, interconnect is not modeled at all. As a circuit's physical size decreases, the transmission delay introduced by the interconnect is as large as the propagation delay of the gates. Certain layout techniques, such as a long run of polysilicon, are inherently slow and might become the fatal flaw in an otherwise carefully tuned design. [Penfield81] offers some computationally reasonable models for predicting transmission delays; these models are well-suited for incorporation into RSIM. His analysis, along with that of [Horowitz83], offers some insight into the correct modeling of pass gates and distributed capacitances. (The lumped approximation used by RSIM can be very pessimistic.) Along the same lines, the development of better time constants for charge-sharing events would improve the modeling of circuits containing both large and small capacitances.

Another class of problems is introduced by the one-pass nature of the computations. In order to limit the amount of computation needed for each prediction, the algorithms are constrained to make only one pass over the surrounding network. While most MOS circuits are trees, and hence amenable to a one-pass analysis, circuits that contain cycles are not handled correctly. The proposed solution — choosing a single path through the cycle to represent the cycle's resistance — is definitely *ad hoc*; performing the correct series/parallel analysis would be preferable.

There is also a need to consider the effects of deviations in device performance from that predicted by first-order theory. Some effects (channel length modulation, body effect, short channel effects) might best be handled during the calibration process. Other effects (Miller capacitance) may lead to further modifications in the model or calculation of device parameters in order to ensure conservative predictions. Finally, there is the possibility that work on waveform bounding [Wyatt83], which seeks to obtain closed-form equations for the waveform of each node of a circuit, can provide a replacement for the linear model presented here.

CHAPTER FIVE

Simulation Using a Switch Network Model

If a designer is only interested in the logical properties of a circuit, *i.e.*, those properties independent of performance issues, it is possible to simplify the linear model of the previous chapter even further by modeling each transistor as an on/off switch whose state is determined by the type of transistor and the state of its gate node. This chapter discusses the switch model from two points of view: first, as a special case of the linear model, and then as a self-contained model. But first, a small digression on the representation of node values is in order.

5.1. Representing node values

The success or failure of a logic-level simulator often hinges on the choice of the set of possible node values. If the set is too small, the actual node value may not be precisely described by any one of the available values and the simulator must choose an approximation. Usually the approximation involves some variant of the X (unknown) value which may carry logical implications beyond what the network itself imposes — such a choice is termed either "conservative" or "pessimistic" depending on one's point of view. If the set is large, it becomes difficult to establish whether the simulator's calculations are correct in all cases. Relying on the accumulated evidence of many simulation runs when arguing correctness lacks the rigor that leads to total confidence in the algorithm. This section develops criteria for evaluating a set of node values.

There are three major influences on the choice of the node-value set:

(1) the need to report node values to the user;

(2) the need to determine the state of each network component from the values of its terminal nodes; and

(3) the need to represent intermediate values during an incremental simulation calculation.

If only the first two influences are considered, a three-value set — 0, 1, and X† — will suffice for logic-level simulation. Users and component models cannot reasonably expect more information than provided by this set, since most logic-level algorithms cannot support more detailed deductions from arbitrary MOS networks with any degree of accuracy. It is the third influence that leads to all the complication.

Almost all logic simulators analyze a network piece by piece, modifying their estimates for node values as the effect of each piece of the network is determined. Until the new-value computation is completed, the intermediate node values serve as accumulators that store all the information the simulator has about the effects of network pieces already examined. Thus, distinct values are needed for all qualitatively different intermediate states; *e.g.*, a node currently at logic high might have that value because examination of the network to date revealed that it was (i) storing charge, (ii) connected to a depletion pullup, or (iii) being precharged by an enhancement device. The simulator must distinguish among these possibilities, since the final value of node may be different in each case if, for example, further network processing discovers a pulldown for the node. The exact number of values needed depends on the details of the simulation computation; most simulators fall into one of the two categories discussed below. As will be seen, the two categories are distinguished by their approach to X values.

†It might be useful to distinguish X', an unknown, but legitimate logic value (*e.g.*, the output of a pair of cross-coupled inverters) from other types of X values. X' values are well behaved in logic operations, for example, $B + \neg B = 1$ if the value of B is X', but equals X if the value of B is X. Such distinctions might be important during initialization. [Stevens83] describes a simulator that uses this distinction to improve its predictions for certain simple logic circuits.

5.1.1. Cross-product value sets

One intuitively appealing approach to choosing a set of node values is to think of each value as having several distinct attributes chosen from independent categories. Thus, for example, one might characterize a node's logic state and the "strength" of the value separately. The logic state is usually one of 0, 1, or X; sometimes a high-impedance state, Z, is included to represent the output of tri-state logic gates [Flake80, Holt81]. The strength indicates what sort of network connection exists between the source of the value and the current node:

> *input.* Node is a designated input (*e.g.*, VDD or GND). The value of an input node can only be changed by explicit simulator commands — the assumption is that inputs supply enough current to be unaffected by connections (possibly shorts to other inputs) made by transistor switches.
>
> *driven.* Node is connected by closed switches to inputs or other driven nodes. Driven nodes can affect the value of weak or charged nodes without being affected themselves, but may be forced to an X state if shorted to an input or driven node that has a different logic level.
>
> *weak.* Node is connected to an input node by a depletion-mode transistor. Weak nodes can affect charged nodes without being affected themselves, but are forced to a driven state when connected to another driven or input node. A weak node returns to the appropriate weak state when completely disconnected from driven or input nodes (*i.e.*, a weak node can never enter the charged state).
>
> *charged.* Node is connected, if at all, only to other charged nodes. Until reconnected to some other part of the network, charged nodes maintain their current logic state indefinitely (charge storage with no decay). This is the default state of all non-weak nodes.

Other strengths can be included to model the effects of differently sized transistors, node capacitors, etc.

The plethora of 9-, 12-, and 16-state logic simulators (see [Newton80]) use values chosen from the set formed by the cross product of the various value attributes. For example, a 9-state simulator might use

		logic state		
		0	1	X
	driven	DL	DH	DX
strength	weak	WL	WH	WX
	charged	CL	CH	CX

Note that in this formulation, X is treated as sort of a third logic value on a par with 0 and 1; presumably X's are generated by the simulator to model invalid combinations of 0's and 1's. The implication is that one can determine if a value should be X without any consideration of strengths. (Remember that the main motivation of forming the cross product is that the various attributes are independent). This can lead to pessimistic predictions, as is shown in an example below.

It is useful to order the possible signal values according to their relative strengths. Intuitively, value A is stronger than value B, written A > B, if value A predominates when both signals are shorted together. Of course there are situations where neither value emerges unscathed — for example, when two signals of the same strength but opposite logic states are shorted — in which case neither signal is said to be stronger than the other. The notion of strength can be formalized using a lattice of node values, for example:

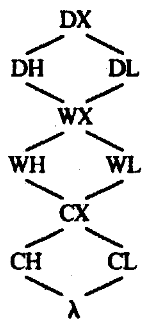


Figure 5.1. *Lattice of node values for a 9-state simulator*

The node value λ is used to represent the null signal, *i.e.*, no signal at all.

Referring to the lattice, given two values A and B, A > B if A is not equal to B and there is an upward path through the lattice that starts at B and reaches A. For example

DX is greater than all other signals,

DH is greater than WL, but

WL is not greater than WH.

The least upper bound (l.u.b.) of two values A and B, written A ∪ B, is defined to be the value C such that

(i) $C \geq A$

(ii) $C \geq B$

(iii) for every value D, if $D \geq A$ and $D \geq B$, then $D \geq C$.

Examining the lattice above, it is easy to see that the l.u.b. always exists for any two node values. Note that if A > B, A ∪ B = A; the l.u.b. captures our intuition about what should happen when two signals of different strengths are shorted together. With the appropriate placement of X values in the lattice, the l.u.b. can be used to predict the outcome when *any* two signals are shorted.

The interpretation of X values captured by the lattice above is quite appropriate for describing the logic state of nodes involved in a short circuit:

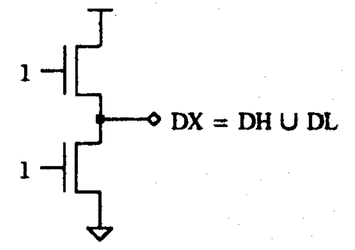


Figure 5.2. *A short circuit leading to an X value*

Assuming the two transistors are the same size, the middle node's value is the result of merging two equal strength signal values. According to our lattice, this merger yields an X value. Short circuits are the mechanism by which X's are introduced into a network previously containing only 0's and 1's.

However, the situation is not as straightforward when one considers connections formed by transistors with a gate signal of X. The resulting values cannot be computed directly using the ∪ operation on the source and drain signals, and once that hurdle has been surmounted, there is some difficulty in choosing which value to use from the cross-product value set. Consider the following analysis of a node with stored charge and connection to two transistors.

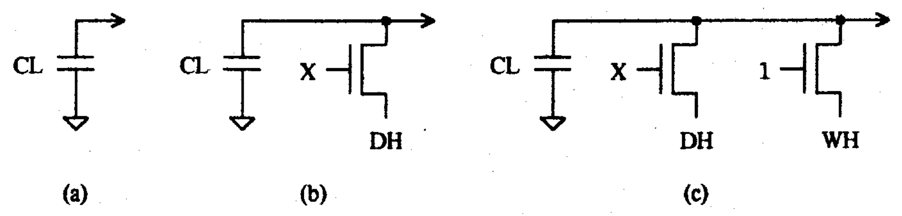


Figure 5.3. *Incremental analysis of a simple network*

Before any connections to the node have been discovered (figure 5.3(a)), the node maintains the charge of its last driven value, say, logic low; the simulator would assign the node a value of CL. After the first transistor is discovered (figure 5.3(b)), the facts change:

(i) Because of the X on the gate of the transistor, one cannot be certain what type of connection exists between the node in question and the DH on the other side of the transistor. Thus, the new logic state of the node should be X.

(ii) The strength of the new value is uncertain, but clearly "weak" or "charged" would be inappropriate since they understate the strength in the case where the unknown gate value was actually a 1.

Since a weak or charged value could be overridden by an enhancement pulldown discovered later on, mistakenly leading to DL value, the simulator has no choice but to select a driven value. The conclusion: DX is the only state available that handles all eventualities in a conservative fashion. Of course, with knowledge of what the rest of the network contains, the simulator could make a more intelligent choice, but this is beyond the ken of an incremental algorithm.

By the time a connection to a depletion pullup is discovered (figure 5.3(c)), the die has been cast: the previously chosen DX value overrides any contribution by the pullup (DX $\cup$ anything = DX). While this answer is not wrong, it is more conservative than required; at this point the logic state of the node should be 1. The pullup guarantees a logic 1 with the unknown connection to DH, only leaving doubts about the strength of the value (somewhere between weak and driven).

Proponents of cross-product value sets might point out that the analysis would have generated a different answer if the transistors had been discovered in a different order. The somewhat embarrassing ability to produce two different answers for the same network, both correct, is caused by the fact that the merge operation is not associative when connections are made through transistors with X gates. In fact, most incremental simulators that use cross-product value sets perform the incremental analysis in an order that yields a reasonable answer on the example above. Unfortunately, it is usually possible to confound them with more complex circuits containing X's; while such circuits are not commonplace, they often crop up during network initialization when all nodes start off at X.†

In conclusion, it is possible to build effective simulators using cross-product value sets; however, they can make conservative predictions on circuits that contain X's. In practice, this leads to difficulty in initializing some circuits and to occasional over-propagation of X values.

†[Bryant81] suggests using an incremental calculation only for subnetworks of nodes connected by non-X transistors. Once these values have been computed, a separate computation merges subnets connected by X transistors. Since this computation has global knowledge of the network, it can avoid the problems mentioned here.

5.1.2. Interval value sets

The difficulties with the cross-product value set arise because of its separation of the notion of strength and logic state. Once a node value is set to an X value at some strength, it cannot return to a normal logic state unless overpowered by a stronger signal; if a node is set to the strongest X value, it stays at that value for the rest of the computation. As in the example above, this leads to conservative predictions when the strongest X value is chosen because of the lack of suitable alternatives. Specifically the difficulty came about because the simulator had to pick the highest strength to be on the safe side; there was no value available that would indicate that the logic low signal which contributed to the intermediate X value was of very low strength and hence might be overridden by later network components.

This suggests a different approach to constructing the set of possible nodes values, one based on intervals. First one starts with a set of node values with a range of strengths and 0/1 logic states, for example, the six non-X states used above: {DH, DL, WH, WL, CH, CL}. Then additional values are introduced by forming intervals from two of the basic values; if there are six basic values, then there are $\binom{6}{2}$ = 15 such intervals, leading to a total of 21 node values altogether.

Intervals represent a range of possible values for a node. The size of the range is related to the strength of its end points. If we arrange the six basic values in a spectrum ranging from the strongest 1 (DH) to the strongest 0 (DL), the possible node values can be shown graphically:

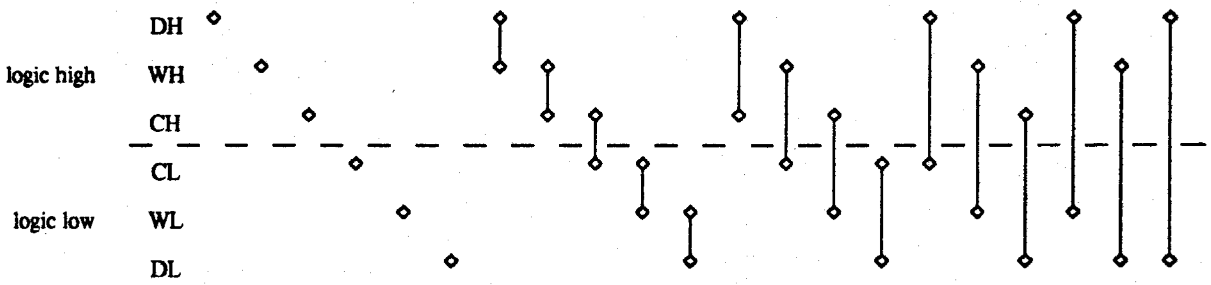


Figure 5.4. *The 21 node values of the interval value set*

Intervals that do not cross the center line correspond to a valid logic state: intervals above the line represent logic high values, and those below the line, logic low. Intervals that cross the center line represent X values. (The X values of the previous section correspond to intervals with equal strength end points: DX = [DL,DH], WX = [WL,WH], and CX = [CL,CH].) Thus, X values result from ambiguity about which of the base values best represents the true node value. As will be seen below, this is more satisfactory than thinking of X as a third, independent logic state.

When the simulator merges two node values, it chooses the smallest interval that covers all the possible node states. However, unlike the cross-product value set, the interval set can represent X values without loosing track of the strengths of the signals that lead to the X values. Consider the problems raised by figure 5.3(b). Using an interval value set, the resulting node value is naturally represented by [CL,DH], an interval that corresponds to an X logic state. When the pullup is discovered (figure 5.3(c)), the simulator can narrow this interval to [WH,DH] since the pullup overpowers the weaker CL value. This corresponds to a logic high signal — a sensible answer.

An algebra for calculating the result of merging two interval node values is developed in [Flake83]; a different approach is adopted in section 5.4.1 where a detailed description of the merge operation can be found. With an interval value set, the merge operation is commutative and associative, and the network can be processed in any order without affecting the final node values. The extra 12 values introduced by the interval value set are needed to carry sufficient information about how the current value was determined, to ensure that the final answer is independent of the processing order.

The examples above suggest the following conjecture about the correct size of a node value set. Assuming that one has s different signal strengths and two logic levels (0 and 1), then $2s + \binom{2s}{2}$ values are needed to ensure that the signal algebra is well-formed. In simulators with too few states, some states take on multiple meanings; for example, the DX value in the cross-product value set is used to describe nodes that fall into 5 separate values in the interval value set:

[DL,DH] [WL,DH] [CL,DH] [WH,DL] [CH,DL]

This lack of expressive power on the part of cross-product value sets is what leads to pessimistic predictions for node values in certain networks.

5.2. Developing the switch model

Switch models of MOS circuits are of interest since a switch is the simplest component that meets the criteria outlined in Chapter 1: switches are inherently bidirectional and the logic operations they implement can be computed with acceptable efficiency in large networks.

Randy Bryant [Bryant79], one of the first to apply switch-level simulation to MOS transistor networks, viewed the network as divided into equivalence classes. Two nodes are equivalent if they are connected by a path of closed switches. Nodes in the same equivalence class as VDD are assigned a logic high state; those equivalent to GND, a logic low state. A pullup (a depletion-mode transistor which is always on in the switch model) gives the node to which it is attached a special property: if an equivalence class of nodes does not contain either VDD or GND, but does contain a pulled-up node, all the nodes in the class are assigned a logic high state. Finally, if an equivalence class contains neither an input nor a pulled-up node, it is "storing charge" and maintains whatever logic state it had last.

The simulator based on this switch model iteratively calculates the equivalence classes for all the nodes in the network until two successive calculations return the same result (*i.e.*, no nodes change state). Unfortunately this pure switch model has some deficiencies:

(i) Switches in indeterminate states (those with "gate" nodes of X) make the equivalence calculation somewhat more difficult. The desired computation is inefficient since it involves a combinatorial search; all combinations of on/off assignments to switches in the X state need to be investigated to determine whether a switch's state makes a difference. If the network is unaffected by a switch's state, the switch can be ignored; otherwise all affected nodes are assigned the X state.

(ii) The equivalence calculation is much more time consuming than necessary since it deals with the whole circuit rather than focusing only on the parts which change.

(iii) In certain circuits transistor "size" is important, and the notion of size cannot be expressed in the pure switch model. A pullup is a trivial example: viewed as a switch it was always on, but more "weakly" than the "strong" switches in the pulldown. The size of transistors also determines the "strength" of various driver circuits; for example, it is common for the write amplifier of a static memory to force a value into a memory cell by simply overpowering the weaker gate in the cell itself.

The remainder of this chapter investigates different approaches to solving the first two problems outlined above. The third problem is addressed with some success by RSIM which uses size information not only to calculate node values but to provide timing information as well.†

The following sections present two different formulations of the switch model:

- a model where each node value is computed via a "global" examination of the network. If the network has no explicit feedback, each node value is computed exactly once, but this calculation is more expensive than the one below.

- a model based on "local" interactions where the simulator examines the source and drain nodes of each transistor and updates the state of one or both nodes. The examination/update process continues until there are no further updates to be made, *i.e.*, the network has "relaxed" into its final state. Under this scheme each calculation is trivial but a node value might be computed more than once even

†Bryant [Bryant81] proposes extending the switch model to include a hierarchy of switch sizes, a generalization of the *ad hoc* solution for pullups. His thesis develops an algebra, in the spirit of Boolean algebra, for dealing formally with such networks.

when there is no explicit feedback in the circuit.

ESIM (the author's switch-level simulator) is a hybrid of these two formulations. ESIM implements a global node-value calculation using a node-value representation close to the one used by the local simulator. This results in a calculation very similar to that implemented by RSIM, except that abstract "logical" resistances ($R_{eff} = 0, 1,$ and ∞) are substituted for the "real" resistances used in RSIM. Since this type of simulation algorithm is discussed at length in Chapter 4, it will not be pursued here. Instead, the remainder of this chapter focuses on the new formulations introduced above.

The local formulation is attractive because it appeals to our intuition about how transistors really work. The high degree of potential parallelism in the update calculation makes it a very attractive algorithm for many of the new parallel architectures now under development; the combination of parallel hardware and intrinsically parallel algorithms may be the key to overcoming the capacity limitations of current simulation techniques.

5.3. The global switch model

The global simulator calculates a node's value by computing the effect of each input on the node of interest. The simulation is global in that each node value is based directly on the values of the inputs to which it is connected. Thus, the values of non-input nodes do not enter into the computation. This means that 0, 1, and X will suffice as final node values; a node state need only capture the logic state of the node and no strength information is necessary.

5.3.1. Node values in the global switch model

Each transistor switch in the network is assigned a state determined from the transistor's type and the current value of its gate node. This state models the switch-like qualities of the source-drain connection without trying to capture any more detailed information about the connection — a simplification of the linear model presented in earlier chapters.

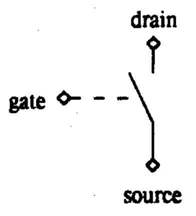
The state of a transistor switch summarizes the type of connection that exists between its source and drain nodes. For MOS circuits, the possible switch states are:

open	no connection, the state of a non-conducting n-channel (gate = 0) or p-channel (gate = 1) transistor.
closed	source and drain shorted, the state of a conducting n-channel (gate = 1) or p-channel (gate = 0) transistor.
unknown	uncertain connection between source and drain, the state of an n- or p-

channel transistor whose gate is X.

weak — the state of a depletion transistor. Depletion devices are always assigned this state, regardless of the state of their gate nodes.

The relationship between a switch's state, its types, and its gate value is summarized in the following figure.



logic(gate)	n-channel	p-channel	depletion
1	closed	open	weak
0	open	closed	weak
X	unknown	unknown	weak

Figure 5.5. *Switch state as a function of transistor type and gate voltage*

In the global simulator, the value of a node is determined by the inputs to which it is connected and the states of the intervening switches. During the calculation of a node's value, the simulator uses the interval node-value set presented in figure 5.4. When the calculation is complete, the resulting interval is used to determine the final logic state of the node, using the following table.

final logic state = 0	*final logic state = 1*	*final logic state = X*
CL	DH	[DH,CL]
[CL,WL]	[DH,WH]	[DH,WL]
[CL,DL]	[DH,CH]	[DH,DL]
WL	WH	[WH,CL]
[WL,DL]	[WH,CH]	[WH,WL]
DL	CH	[WH,DL]
		[CH,CL]
		[CH,WL]
		[CH,DL]

Table 5.1. *Relationship between final logic state and computed interval value*

The calculation of a node's value begins by discovering all the inputs which can be reached from the node by paths of closed, weak, and unknown switches. If no inputs can be reached, the final logic state of the node is determined by a charge sharing calculation described in the next section. If one or more inputs can be reached, their contribution to the node's value is determined by an incremental calculation which starts at the inputs and works its way back toward the node.

The value of a logic low input is DL; the value of a logic high input is DH. As the calculation works back toward the node of interest, it computes an effective value that indicates the effects of intervening switches on the original input value. The effect of a switch on a value it transmits is specified by the *switch* function:

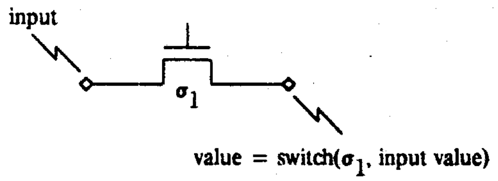


Figure 5.6. *Effective value of an input after passing through a switch*

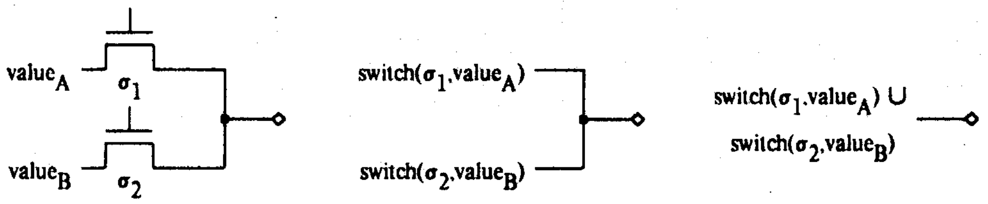
The effect of a switch on a value is a function of the value and the switch's state:

value	*switch state*			
	open	*closed*	*weak*	*unknown*
DH	λ	DH	WH	[DH,λ]
[DH,WH]	λ	[DH,WH]	WH	[DH,λ]
[DH,CH]	λ	[DH,CH]	[WH,CH]	[DH,λ]
[DH,CL]	λ	[DH,CL]	[WH,CL]	[DH,CL]
[DH,WL]	λ	[DH,WL]	[WH,WL]	[DH,WL]
[DH,DL]	λ	[DH,DL]	[WH,WL]	[DH,DL]
WH	λ	WH	WH	[WH,λ]
[WH,CH]	λ	[WH,CH]	[WH,CH]	[WH,λ]
[WH,CL]	λ	[WH,CL]	[WH,CL]	[WH,CL]
[WH,WL]	λ	[WH,WL]	[WH,WL]	[WH,WL]
[WH,DL]	λ	[WH,DL]	[WH,WL]	[WH,DL]
CH	λ	CH	CH	[CH,λ]
[CH,CL]	λ	[CH,CL]	[CH,CL]	[CH,CL]
[CH,WL]	λ	[CH,WL]	[CH,WL]	[CH,WL]
[CH,DL]	λ	[CH,DL]	[CH,WL]	[CH,DL]
CL	λ	CL	CL	[λ,CL]
[CL,WL]	λ	[CL,WL]	[CL,WL]	[λ,WL]
[CL,DL]	λ	[CL,DL]	[CL,WL]	[λ,DL]
WL	λ	WL	WL	[λ,WL]
[WL,DL]	λ	[WL,DL]	WL	[λ,DL]
DL	λ	DL	WL	[λ,DL]

Table 5.2. *switch(σ, value) as a function of σ and value*

A new value, λ, is introduced to describe the value transmitted by an open (non-conducting) switch, *i.e.*, no value at all. The value λ is weaker than CH or CL, and corresponds to a logic state of X.

When two paths merge, their effective value is determined using the ∪ operation introduced in section 5.1.1.


value$_A$
σ_1
value$_B$
σ_2
switch(σ_1,value$_A$)
switch(σ_2,value$_B$)
switch(σ_1,value$_A$) ∪
switch(σ_2,value$_B$)

(a) two values to merge (b) values including effect of switches (c) merged value

Figure 5.7. *Merging the values for two paths which join*

The ∪ operation is defined using the lattice shown in the following figure.

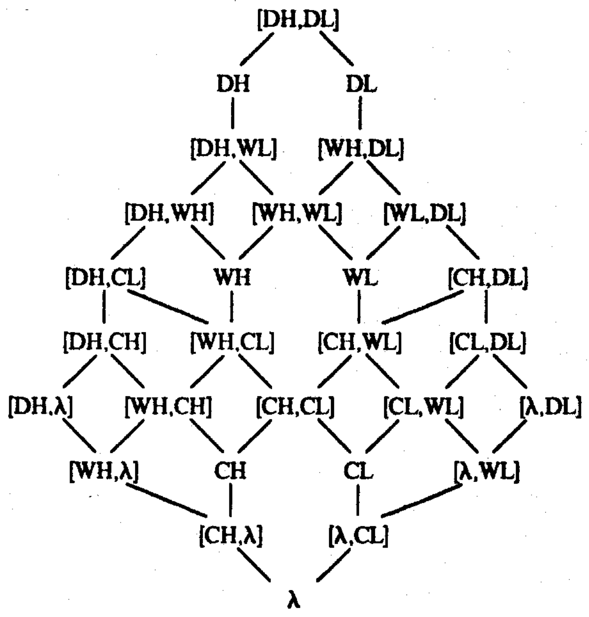


Figure 5.8. *Lattice for interval-node value set*

Following the procedure outlined in figure 5.7, the contributions of all inputs connected to the node of interest can be reduced to a single interval. This interval is merged (using ∪) with the contribution from the node's current logic state

$$\textit{contribution of current logic state} = \begin{cases} CL & \textit{if current logic state} = 0 \\ CH & \textit{if current logic state} = 1 \\ [CH,CL] & \textit{if current logic state} = X \end{cases} \qquad (5.1)$$

to give the final interval characterizing the node's new logic state.

As an example of how the new-value calculation works, consider the following circuit:

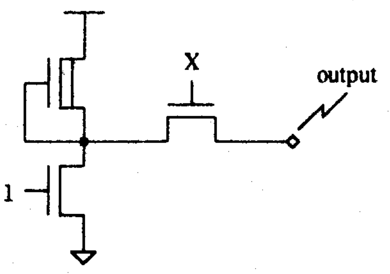


Figure 5.9. *Example circuit*

Assume that the current logic state of the output is 0. The new-value calculation for this circuit is shown in the following figure.

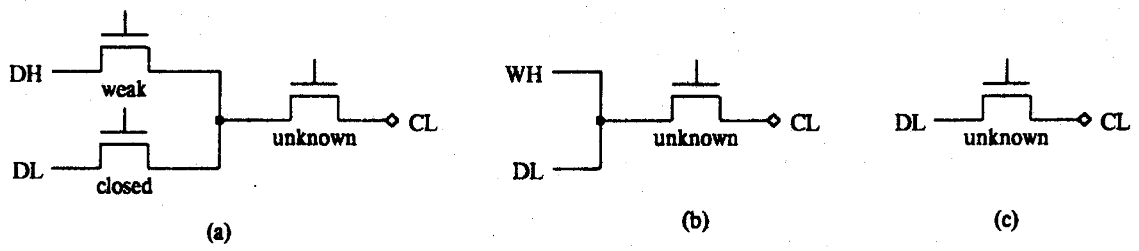


Figure 5.10. *New-value calculation for circuit in figure 5.9*

The final interval for the output node is $CL \cup [\lambda, DL] = [CL, DL]$ which corresponds to a logic low state. This makes sense; the previous state of the output node was logic low, so the uncertain connection to the inverter does not affect its logic state, just the strength with which its driven. Note that it is important to merge the values of paths that join before continuing with the calculation since

$$switch(\sigma, \alpha \cup \beta) \neq switch(\sigma, \alpha) \cup switch(\sigma, \beta) \qquad (5.2)$$

when using this particular value set and *switch* function. For example, if the WH and DL values had been merged *after* transmission by the switch in the unknown state, the final interval for the output node would have been [DH,WL], which corresponds to an X logic state. The calculation described here performs all possible merges *before* transmitting the result through the appropriate switch.

5.3.2. The global simulation algorithm

This section outlines the basic steps for propagating new information about the inputs to the rest of the network, recalculating node values (where necessary) using the global value calculation in the previous section.

When a node changes value, it can affect the network in one of two ways:

(i) directly, through source/drain connections of conducting transistors.

(ii) indirectly, by affecting the state of transistor switches controlled by the changing node. This is turn can cause the source and drain nodes of those switches to change value.

The global simulator accounts for these two effects using to different mechanisms. Directly affected nodes are handled implicitly by the new-value computation which recomputes new values for all directly affected nodes whenever a node changes value. This is a reasonable organization: if A directly affects B, then B directly affects A; it makes sense to compute both values at the same time since they are closely related. Direct effects are not handled implicitly, however, when the user changes the value of an input node. In this case, the simulator invokes the new-value computation on the input, not to recompute the input's value (which is set by the user), but to recompute the values of all directly affected nodes.

The indirect effects of a value change are managed by an *event list* that identifies all transistor switches that have changed state. Actually, the event list keeps track of the nodes that have changed, but this is equivalent since the network data base maintains a list of transistors controlled by each node. The simulator operates by removing the first node from the event list, and then performing a new-value computation for the sources and drains of all transistors controlled by that node. The new-value computation accounts for all the direct effects of the new transistor state and adds events to the event list if indirect effects are present. This process continues until the event list is empty, at which point the network has "settled" and the simulator waits for further input.

```
while event list not empty {
    n := node associated with first event on event list
    remove first event from event list
    for each transistor with n as gate node {
        set COMPUTE flag for source and drain
    }
    for each transistor with n as gate node {
        if COMPUTE still set for source, compute new value for source [fig. 5.14]
        if COMPUTE still set for drain, compute new value for drain
    }
}
```

Figure 5.11. *Main loop of global simulation algorithm*

Finding nodes affected by an event is straightforward; recomputation of values is needed for the sources and drains of all transistors with the changing node as gate. For example, if the node marked (*) in the following figure changes, nodes B and C need recomputation.

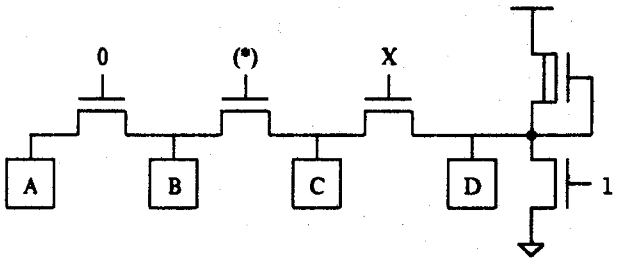


Figure 5.12. *Event for node (*) involves nodes B and C*

Of course, node D also needs to be recomputed, as will be discovered during the processing of B and C (see below).

To recompute the value of a given node, the simulator first makes a *connection list* containing all nodes connected to the first node by a path of conducting transistors. The idea is to start with a node known to be affected by an event, and then find that node's electrical neighbors, and so on, halting whenever an input is reached. In the example above, if the (*) node's value is 1, the connection list for node B contains nodes B, C, and D. If the (*) node's value is 0, the connection list for node B contains only node B. Node A is not included in the list in either case because it is not connected to node B by a path of conducting transistors. In the code below, which computes the connection list for a given node, the terms "source" and "drain" are used to distinguish one terminal node of a transistor from the other, and do not imply anything about the terminals' relative potential.

```
initialize list to have starting node as only element
set pointer to beginning of list
INPUT_FOUND := false
reset capacitance accumulators
while pointer not at end of list {
    n := node currently pointed at
    add capacitance of n to appropriate accumulator
    for each "on" transistor with source connected to n {
        if drain is an input, INPUT_FOUND := true
        else if drain not on list, add drain to end of list
    }
    advance pointer to next list element
}
```

Figure 5.13. *Non-recursive routine to build connection list*

In addition to the connection list, the routine sets INPUT_FOUND to true if the tree walk discovered at least one input, and maintains three capacitance accumulators, one for each logic state. The connection list drives the new-value computation:

```
make connection list starting with given node [fig. 5.13]
if no inputs found, do charge sharing
else for each node on connection list {
    compute interval value for node [fig 5.15]
    determine new logic state using Table 5.1
    if different from old logic state {
        update logic state to new value
        enqueue new event
    }
}
reset COMPUTE flag for each node on connection list
```

Figure 5.14. *Subroutine to compute new value for node*

If no inputs are found while building the connection list (INPUT_FOUND is false), the group of nodes is completely isolated from any inputs and a charge sharing computation determines the nodes' new values. Assuming that all the node capacitors are shorted together, the resulting voltage is

$$\textit{voltage of shorted capacitors} = \frac{\sum \textit{capacitors at logic high}}{\sum \textit{all capacitors}} \tag{5.3}$$

Capacitors with a logic state of X are assumed to be charged high when computing the maximum possible voltage, and charged low when computing the minimum voltage:

$$\textit{charge sharing value} = \begin{cases} 0 & \dfrac{C_{high} + C_X}{C_{total}} < 0.2 \\ 1 & \dfrac{C_{high}}{C_{total}} > 0.8 \\ X & \textit{otherwise} \end{cases} \tag{5.4}$$

where C_{total} is the sum of the capacitance accumulators, C_{high} is the accumulator corresponding to logic high, and C_X is the accumulator corresponding to logic X.

If one or more inputs are found (INPUT_FOUND is true), the value of each node is determined in accordance with the procedure described in the previous section. The interval value is calculated for each node in turn and the node's new logic state is computed using Table 5.1. New events are added to the end of the event list whenever a node changes value. If a changing node is already on the event list, nothing happens (the node is not moved to the end of the list).

For efficiency, each affected node's value is only computed once while processing a given event. The connection list ensures that all affected nodes are recomputed; the COMPUTE flag ensures that once a node has appeared on some connection list, it will not be resubmitted for processing during the current event.

The computation of a node's value is easily described by a recursive procedure which analyzes the surrounding network:

```
if node is logic low input {
    return DL
} else if node is logic high input {
    return DH
} else {
    LOCAL_IV := value specified by equation 5.1
    set VISITED flag for current node
    for each "on" transistor, t, with source connected to current node {
        if drain does not have VISITED flag set {
            recursively determine interval value for drain node
            LOCAL_IV := LOCAL_IV ∪ switch(σ_t, drain's interval value)
        }
    }
    reset VISITED flag for current node
    return LOCAL_IV
}
```

Figure 5.15. *Subroutine to compute interval value for node*

The variable LOCAL_IV is a stack-allocated local variable of the subroutine. Returning to the example in figure 5.12, assuming that the (*) node's value is 1, and that the old values for B, C, and D are $B=1$, $C=0$, and $D=0$, the following calls are made when computing the new value for node C:

```
compute_params(C)
    LOCAL_IV = CL
    compute_params(D)
        LOCAL_IV = CL
        compute_params(VDD)
            return DH
        LOCAL_IV = CL ∪ WH = WH
        compute_params(GND)
            return DL
        LOCAL_IV = WH ∪ DL = DL
        return DL
    LOCAL_IV = CL ∪ [λ,DL] = [CL,DL]
    compute_params(B)
        LOCAL_IV = CH
        return CH
    LOCAL_IV = [CL,DL] ∪ CH = [CH,DL]
    return [CH,DL]
```

Figure 5.16. *Trace of interval value computation for example in figure 5.12*

Marking each visited node (by setting its VISITED flag) avoids cycles; this keeps the tree walk expanding outward from the starting node. The VISITED flags are reset as the routine backs out of the tree walk, so all possible paths through the network are eventually analyzed.

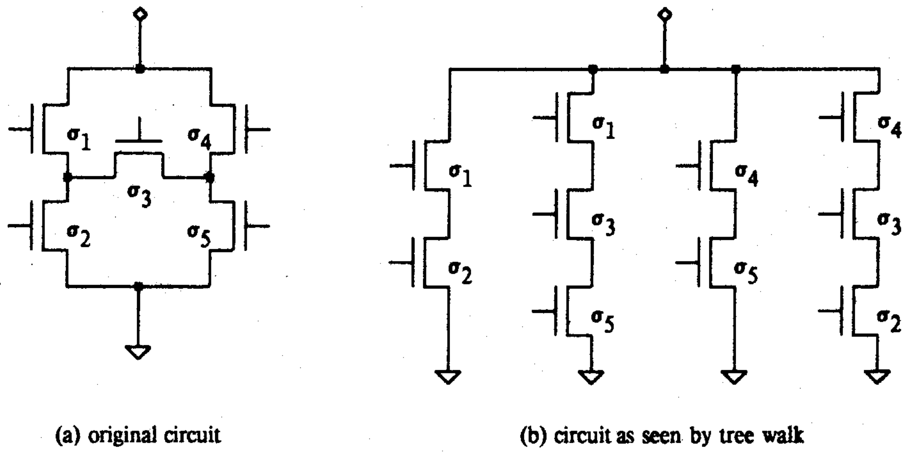


Figure 5.17. *The tree walk traces out all possible paths*

If the network contains cycles, the tree walk might lead to more computation than a series/parallel analysis; this is a problem for circuits containing many potential cycles (such as barrel shifters), especially during initialization when many of the paths are conducting because control nodes are X. To speed up the calculation, a node's VISITED flag can be left set, restricting the search to a single path through a cyclic network. This technique produces correct results only if paths leading away from a node are explored in order of increasing resistance, *i.e.*, one must ensure that the first time a node is reached, it is by the path of least resistance. Of course, the flags must be reset once the entire computation is complete; fortunately, the connection list provides a handy way of finding all the nodes that are visited without resorting to yet another tree walk. Another alternative for speeding up the calculation is the caching technique described in section 4.2.

5.3.3. Interesting properties of the global algorithm

The event list serves to focus the attention of the global simulator; new values are computed only for nodes which appear on the event list or which are electrically connected to event-list nodes. Portions of the network that are quiescent are not examined by the simulator. Algorithms that have this property are said to be *selective-trace* or *event-driven* algorithms and generally run much faster than algorithms which are not event driven [Szygenda75].†

An interesting implication of selective trace is that special care must be taken to ensure that "constant" nodes, such as the output of an inverter with its input tied to GND, are processed at least once (otherwise they will have the wrong values). One technique is to treat VDD and GND as ordinary inputs when first starting a simulation run — sort of a power-up sequence as VDD and GND change from X to 1 and 0 respectively. Computing both the direct and indirect consequences of changes in VDD and GND might involve a tremendous amount of computation since the whole circuit is affected; often only computing the indirect consequences is a sufficient and less costly alternative.

Although there is no explicit mention of time in the global simulator, the first-in, first-out (FIFO) processing of events imposes some ordering on the changes of node values. This ordering is similar to, but not the same as, the unit-delay ordering used by many gate-level simulators. In an event-driven unit-delay algorithm, the output of each gate that had an input change is recomputed using the current values of the input nodes. The new output values are saved and imposed on the network only after processing all gates. The net effect is that each computation cycle (representing a unit of time) propagates information through one level of gate, *i.e.*, each gate has unit delay. Because changes in node values are imposed all at once, values change simultaneously, which can lead to problems in

†Exceptions to this rule are some hardware-based simulation algorithms, such as programs run on the Yorktown Simulation Engine [Pfister82]. The builders of the YSE point out that simulations might well run slower because the extra communication and branching needed to implement selective trace would compromise the parallelism and pipelining used to great advantage in the YSE. However, if sufficiently large portions of the circuits could be ignored, the overhead of selective trace could be worth the investment (see Chapter 6).

circuits containing feedback paths.

The global simulator implements a pseudo unit-delay algorithm. New events are added to the end of the event list, so the oldest changes are processed before any consequences of those changes are processed. Thus, FIFO event management leads to the same sequence of gate evaluations as a unit-delay algorithm. However, because the global algorithm changes values in the network incrementally rather than all at once, it is possible to find circuits that behave differently under the two simulators:

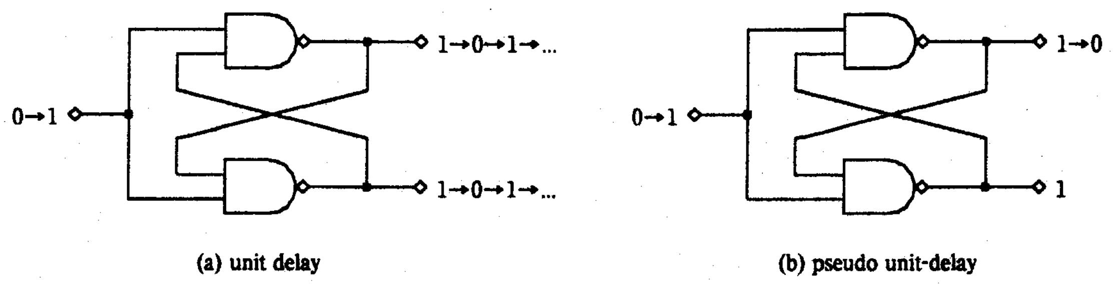


Figure 5.18. *Circuit that distinguishes unit-delay from pseudo unit-delay*

A 0-1 transition on the input causes a unit-delay algorithm to loop forever. The global algorithm predicts only one transition — the output of whichever gate it processes first. Neither answer is completely correct; the actual circuit enters a meta-stable state on a 0-1 input transition, eventually settling to a particular configuration determined by subtle differences in the gains of the two gates. It will not remain in the meta-stable state forever, so an infinite oscillation is a poor prediction. On the other hand, the final configuration chosen by the global simulator depends on the order of some list in the network data base. The predicted outcome is the same each time, not necessarily the best prediction.† The global simulator does not offer a general solution to the oscillation problem; both simulators will oscillate on the following circuit.

†[Bryant81] suggests that the oscillation can be detected and the offending node values replaced by X, but the technique for determining the number of oscillations to allow yields answers so large for circuits of any substantial size that this is not a very practical alternative.

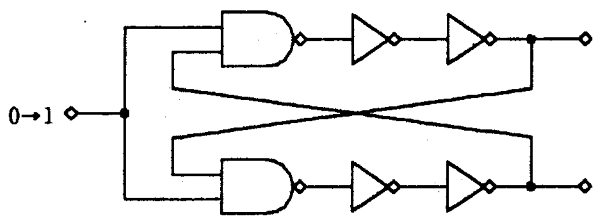


Figure 5.19. *Circuit which causes both simulators to oscillate*

Along the same lines, the global simulator predicts that the output of the circuit below will oscillate when the input changes from 1 to 0.

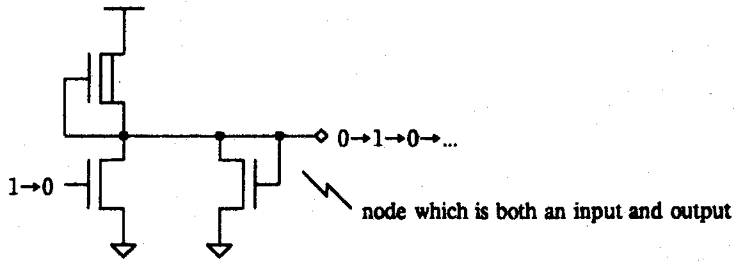


Figure 5.20. *Circuit with a node that is both an input and output*

The actual output quickly rises to the balance point of the pullup/pulldown combination. In a logic-level simulation, this corresponds to finding a solution to the equation $\alpha = \neg\alpha$ which has the solution $\alpha = X$ (a reasonable logic-level representation for the balance point). This example is drawn from a larger class of circuits where a node is both an input and output of the circuit. Since the new-value computation uses current transistor states (determined by current node values) to predict the new values, it is impossible to predict the value of a node that depends on its own value. This limitation has not proven to be a problem in practical circuits.

5.4. The local switch model

It is interesting to speculate about replacing the tree walk performed by the global simulator with a strictly local computation. After all, the models of transistor behavior presented in Chapter 3 show that a transistor is controlled by the voltages of its three terminal nodes, *i.e.*, each transistor operates independently, basing its behavior on only local information available at its terminals. The simulation model described in this section works in much the same way. The basic operation involves updating the terminal node values of a transistor switch using only information about their previous values and the state of the switch.

Relaxation-based algorithms leave one a little nervous. Will the relaxation terminate? Does the final answer depend on the order in which the individual computations are performed? These questions are answered below, after a description of the algorithm itself.

5.4.1. Node values in the local switch model

The set of node values and the computation developed for the global simulator must be adapted for use by the local simulator. The necessity for an adaptation is explained at the end of section 5.4.2. (The discussion is postponed until after the local simulation algorithm has been presented, when it will be easier to explain why the global simulator's techniques do not work in the local simulator's context.)

In the local simulator, a node value is a pair

$$\langle \text{high,low} \rangle$$

that separately lists what type of connection exists to each of the two possible input signals. The high component summarizes what is known about paths to VDD, and the low component describes paths to GND. Ignoring for the moment switches with gates of X, four types of connections can be distinguished for each component:

∞	no paths to inputs, no charge storage.
S	charge storage.
1	there is a path to the appropriate input, but it passes through one or more depletion switches.
0	there is a path of conducting n-channel (gate = 1) and p-channel (gate = 0) switches to the given input.

A switch with a gate of X may or may not make a connection; the resulting path is characterized by an interval describing the range of alternatives. $\binom{4}{2} = 6$ intervals are needed to describe all possible combinations of paths.

The value of VDD is $\langle 0,\infty \rangle$ and of GND is $\langle \infty,0 \rangle$; some other examples are shown in the following figure.

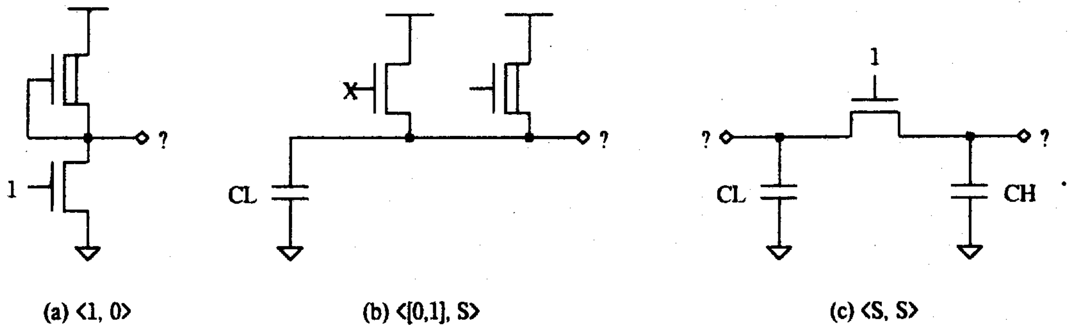


Figure 5.21. *Examples of node values in the local simulator*

This organization provides for many more values than actually needed by the simulator; many of the values make distinctions that are not important in determining a node's logic state. For example, <1,0> and <S,0> both represent values corresponding to pulled-down nodes — it does not matter what the high component contributes if it is weaker than the low component. The advantage of this notation is the ease of computing what a given signal looks like from the other side of a transistor switch:

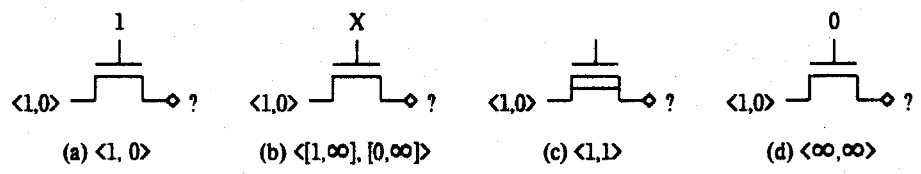


Figure 5.22. *<1,0> value as seen across various transistor switches*

This will prove very useful in describing the update operation below.

Using the technology developed in section 5.1.1, a lattice can be constructed that indicates the relative ordering of the various component values:

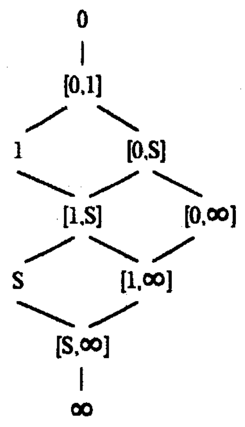


Figure 5.23. *Lattice for the ten possible component values*

The ∪ operation can be used to calculate the result of considering two paths in parallel:

$$<h_1, l_1> \cup <h_2, l_2> \equiv <h_1 \cup h_2, l_1 \cup l_2> \tag{5.5}$$

Each component is merged separately according to the lattice given above. Similarly, two values can be ordered by comparing their components:

$$<h_1, l_1> \leq <h_2, l_2> \;\; \textit{iff} \;\; h_1 \leq h_2 \text{ and } l_1 \leq l_2 \tag{5.6}$$

A logic state can be associated with a value <h,l> using the following table:

						h					
		0	[0,1]	[0,S]	[0,∞]	1	[1,S]	[1,∞]	S	[S,∞]	∞
	0	X	X	X	X	0	0	0	0	0	0
	[0,1]	X	X	X	X	X	X	X	0	0	0
	[0,S]	X	X	X	X	X	X	X	X	X	0
	[0,∞]	X	X	X	X	X	X	X	X	X	X
	1	1	X	X	X	X	X	X	0	0	0
l	[1,S]	1	X	X	X	X	X	X	X	X	0
	[1,∞]	1	X	X	X	X	X	X	X	X	X
	S	1	1	X	X	1	X	X	X	X	0
	[S,∞]	1	1	X	X	1	X	X	X	X	X
	∞	1	1	1	X	1	1	X	1	X	X

Table 5.3. *Logic state associated with <h,l>*

5.4.2. The local simulation algorithm

The local simulator implements a relaxation-based calculation for propagating input values through the network. The calculation has three major steps:

Step 1. Determine the state of each transistor switch from its type and the logic state of its gate node. If no switches are found that changed state since the last examination, the network is said to have "settled" and the simulator waits for more input.

Step 2. Reset each non-input node value to its charged value, a value that corresponds to the node's last logic state but does not have sufficient strength to force the value of any neighboring nodes.

Step 3. Repeatedly pick a transistor and update the values of its source and drain nodes according to the formula given below, continuing until the relaxation is complete (no node changes value as the result of an update). Upon completion, return to Step 1.

Each of these steps is described in more detail below.

Figure 5.5 shows how a switch's state is determined from its type and the logic state of its gate node. Once determined, the switch state remains stable through Steps 2 and 3 even if the gate changes value. This arrangement is necessary for the correct operation of the simulator since a node's value might temporarily be incorrect during the relaxation computation while information continues to propagate towards the node from various inputs. For example, the output of a NAND gate may momentarily appear to be pulled-up, because the near-by pullup affects the node's value before information can propagate from GND up the pulldown chain. Since there are no guarantees about the ordering of updates, a node's value is known to be correct only when the relaxation process terminates.

Step 2 makes sure that the relaxation starts off with a clean slate; when this step is complete, only input nodes have values that can cause the values of neighboring nodes to change. This ensures that values for non-input nodes are determined exclusively by the values of the input nodes.

$$\textit{charged value} = \begin{cases} <\infty, S> & \textit{current logic state} = 0 \\ <S, \infty> & \textit{current logic state} = 1 \\ <S, S> & \textit{current logic state} = X \end{cases} \tag{5.7}$$

If a node is not connected to any input, the charged value is an accurate representation of its final value. The update calculation performs a rudimentary charge sharing computation; a charged node can become connected to another charged node with the same logic state, and still maintain its value. Connection to a charged node with a different logic state results in both node values becoming <S,S>.

Note that precharge/discharge circuits are simulated correctly.

An update operation involves the source and drain nodes of a single transistor switch. The new values of the source and drain are calculated from their old values and the state (σ) of the switch:

$$\begin{aligned} v'_{source} &= v_{source} \cup switch(\sigma,\ v_{drain}) \\ v'_{drain} &= v_{drain} \cup switch(\sigma,\ v_{source}) \end{aligned} \tag{5.8}$$

The function *switch(σ, value)* formalizes our intuition about the effect on a value as it passes through a switch in a given state (see figure 5.22). The new value of a terminal node is the result of merging its old value with the old value of the other terminal node after it has passed through the switch.

$$switch(\sigma,\ <h,l>) = \begin{cases} \infty & \sigma = open \\ <h,l> & \sigma = closed \\ <h + [0,\infty],\ l + [0,\infty]> & \sigma = unknown \\ <h + [1,1],\ l + [1,1]> & \sigma = weak \end{cases} \tag{5.9}$$

where "+" is the series operation described in the following table:

+	[0,0]	[0,1]	[0,S]	[0,∞]	[1,1]	[1,S]	[1,∞]	[S,S]	[S,∞]	[∞,∞]
[0,0]	[0,0]									
[0,1]	[0,1]	[0,1]								
[0,S]	[0,S]	[0,S]	[0,S]							
[0,∞]	[0,∞]	[0,∞]	[0,∞]	[0,∞]						
[1,1]	[1,1]	[1,1]	[1,S]	[1,∞]	[1,1]					
[1,S]	[1,S]	[1,S]	[1,S]	[1,∞]	[1,S]	[1,S]				
[1,∞]	[1,∞]	[1,∞]	[1,∞]	[1,∞]	[1,∞]	[1,∞]	[1,∞]			
[S,S]	[S,S]	[S,S]	[S,S]	[S,∞]	[S,S]	[S,S]	[S,∞]	[S,S]		
[S,∞]	[S,∞]	[S,∞]	[S,∞]	[S,∞]	[S,∞]	[S,∞]	[S,∞]	[S,∞]	[S,∞]	
[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]	[∞,∞]

Table 5.4. *Series operation for local simulator*

In general, the local algorithm's predictions are more pessimistic than those of the global simulator. The following figure illustrates the analysis performed by the local simulator for the circuit shown in figure 5.9. (The global simulator's analysis is shown in figure 5.10)

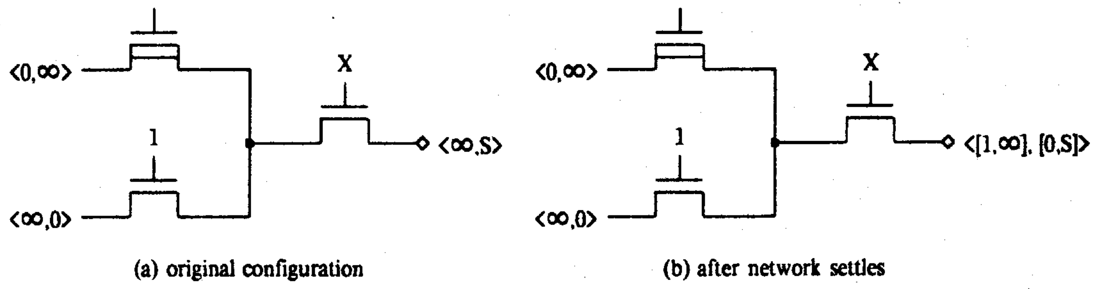
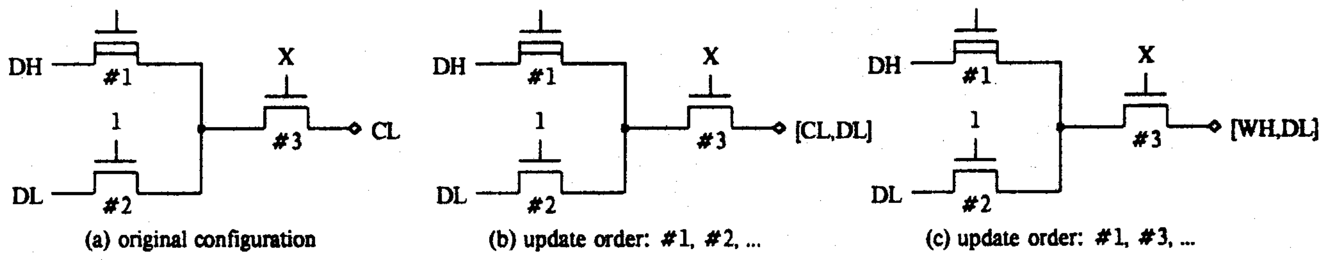


Figure 5.24. *Local simulator analysis for circuit in figure 5.9*

As shown in figure 5.24(b), the local simulator predicts the logic state of the output node to be X — a pessimistic answer. (The global simulator predicts a logic state of 0.) On the other hand, the local simulator cannot simply adopt the value set and computation of the global simulator. The reason why is illustrated by the following figure.



DH
#1
X
1
#3
CL
DL
#2
(a) original configuration
DH
#1
X
1
#3
[CL,DL]
DL
#2
(b) update order: #1, #2, ...
DH
#1
X
1
#3
[WH,DL]
DL
#2
(c) update order: #1, #3, ...

Figure 5.25. *Global simulator's computation using update operations*

The figure shows the final node values (*i.e.*, the values after the network has settled, and further updates make no change to the network), assuming that the first few updates were performed in different orders. Figure 5.25(b) shows the final node values if switch #1 is updated first, followed by switch #2. Figure 5.25(c) shows the final node values if switch #1 is updated first, followed by switch #3. As one can see, the value of the output node differs in the two examples.

If the local simulator's predictions of the final node values are to be independent of update order, it must be the case that

$$switch(\sigma, \alpha \cup \beta) = switch(\sigma, \alpha) \cup switch(\sigma, \beta) \tag{5.10}$$

In other words, it cannot matter if early estimates of a node's value (α) are transmitted to neighboring nodes before additional information (β) arrives. Unfortunately, equation 5.10 is in direct conflict with equation 5.2 which indicates that order makes a difference in the analysis of certain circuits (such as the one in figure 5.9) when using the global simulator's value set. Thus, the local simulator cannot simply adopt the global simulator's value set.

5.4.3. Interesting properties of the local algorithm

In order to answer the questions raised when first introducing the local algorithm, some definitions will be useful. Let S be the set of switch-state vectors $\sigma_1\sigma_2 \cdots \sigma_t$ where t is the number of transistor switches in the network. Similarly, let V be the set of node-value vectors $v_1v_2 \cdots v_n$ where n is the number of nodes in the network. Then S$\times$V is the set of possible network states.

> **Definition.** Let X and Y be network states. $X \geq Y$ if $S_X = S_Y$ and $V_X \geq V_Y$ where comparison between vectors is done component by component.

The update operation changes one network state to another; one writes $X \rightarrow Y$ if a sequence of zero or more updates changes the network state X into the network state Y. $X \rightarrow_m Y$ means that m or fewer updates will change X into Y.

The update operation can potentially change two elements of the node-value vector; the switch-state vector is never affected by an update. Not every update causes the network state to change. For example, if the update chooses an open switch, the resulting network state will be the same as the original state. In the presentation below, it is useful to distinguish those updates that result in a change in the network state from those that do not:

> **Definition.** Let X and Y be network states. $X \Rightarrow Y$ if $X \rightarrow_1 Y$ and $X \neq Y$.

In fact, $X \Rightarrow Y$ implies $Y > X$, a simple consequence of equation 5.9 and the definition of $\cup$. A *stable* network state is one which does not change as the result of any update:

> **Definition.** Let X be a network state. X is stable if, for any network state Y, $X \rightarrow Y$ implies $X = Y$.

It follows directly from this definition that a state is stable if and only if no $\Rightarrow$ operations are possible on the state. Once a stable state is reached, the relaxation process can safely be terminated since further updates will not change the network state. This suggests the following metric for measuring how far the relaxation process has to go:

> **Definition.** Let X be a network state. *order*(X) is defined to be the largest integer m such that there exist states $Y_1, ..., Y_m$ where $X \Rightarrow Y_1 \Rightarrow \cdots \Rightarrow Y_m$.

The termination of the relaxation process is assured by the following theorem:

> **Theorem 5.1.** For any network state X, $order(X)$ is finite.

The proof is based on the observation that there are only finitely many network nodes and possible node values. This means for any given network state X, there are finitely many states Y such that $Y > X$. Since each $\Rightarrow$ operation produces a state strictly greater than its predecessor, one can perform the $\Rightarrow$ operation only finitely many times before all the possible states are exhausted. ∎

For a given starting network state, Theorem 5.1 tells us that a stable state can be reached with only a finite number of $\Rightarrow$ operations. In fact, one can prove that there exists a unique stable state for any network state, but first we must lay a little more groundwork.

> **Lemma 5.2.** Let W and X be network states. If $order(W) = m$ and $W \Rightarrow X$, then $order(X) < m$.

Suppose that $order(X) \geq m$, then there exists a sequence of $\Rightarrow$ operations $W \Rightarrow X \Rightarrow Y_1 \Rightarrow \cdots \Rightarrow Y_{order(X)}$. This implies $order(W) \geq m+1$, a contradiction. ∎

> **Lemma 5.3.** (Church-Rosser property) Let W, X, and Y be network states. If $W \rightarrow_1 X$ and $W \rightarrow_1 Y$, then there exists a network state Z such that $X \rightarrow Z$ and $Y \rightarrow Z$.

Appendix 1 presents a proof based on a case by case analysis of the possible choices for X and Y, demonstrating for each case a sequence of updates that lead to a common state Z.

This sets the stage for proving the uniqueness of the stable state. For readers acquainted with the lambda calculus, the following theorem has a familiar ring. There are many similarities between the update operation and λ-conversion; the discussion of normal forms and the Church-Rosser theorem found in [Curry74] inspired the concept of stable states and the existence and uniqueness theorems presented here.

> **Theorem 5.4.** Let W, X, and Y be network states. If $W \rightarrow X$ and $W \rightarrow Y$, then there exists a network state Z such that $X \rightarrow Z$ and $Y \rightarrow Z$.

The proof proceeds by induction on the order of W. If $order(W) = 0$, then W is stable and so $W = X = Y = Z$. Without loss of generality, if $order(W) > 0$, one can assume $X > W$ and $Y > W$ since if this were not the case, the result follows trivially. If $order(W) = 1$, the result follows as a direct consequence of Lemma 5.3. To show for $order(W) = n+1$, first note that there exist states A and B such that $W \Rightarrow A \rightarrow X$ and $W \Rightarrow B \rightarrow Y$. Then, by Lemma 5.3, there also exists a state C such that $A \rightarrow C$ and $B \rightarrow C$.

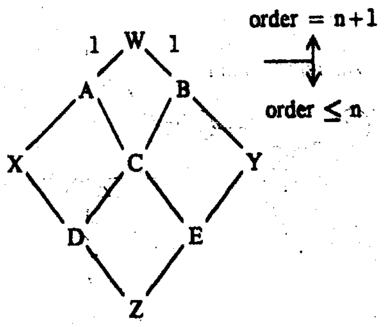


Figure 5.26. *Relationship between states in proof for Theorem 5.4*

Using Lemma 5.2, note that the orders of A, B, and C are all less than $n+1$. Thus, by the induction hypothesis, there exists a state D such that $X \rightarrow D$ and $C \rightarrow D$. Similarly, there exists a state E such that $Y \rightarrow E$ and $C \rightarrow E$, also by the induction hypothesis. Finally, by a third appeal to the induction hypothesis, there exists a state Z such that $D \rightarrow Z$ and $E \rightarrow Z$. ▮

Taken together, Theorems 5.1 and 5.4 imply the following corollary:

Corollary 5.5. Let X be a network state. There exists a unique network state Y such that Y is stable and $X \Rightarrow \cdots \Rightarrow Y$.

Thus, the relaxation process terminates for any starting network configuration, yielding the same stable state regardless of the order chosen for performing the updates.

One of the attractions of the local algorithm is the opportunity it affords for parallel processing, especially during the relaxation process. Allowing parallel updates introduces the problem of merging conflicting node values at the end of the updates. The simplest solution is to allow updates to happen simultaneously only if they operate on separate portions of the network state. With this restriction, each node is involved in at most one update operation, and the potential for conflict is avoided. If the number of available processors is a lot smaller than the number of nodes in the network, there is only a small probability of a processor lying idle, because there are an insufficient number of allowable updates.

Parallel implementations that avoid conflicting updates are covered by the existence and uniqueness results obtained above, since it is easy to convert the set of updates performed at any time step into an equivalent sequence of sequential updates. This approach has sufficient parallelism to keep many current parallel architectures quite busy. However, there are architectures on the drawing boards with very large numbers of processors; it is interesting to speculate about algorithms that can usefully employ as many processors as, say, there are transistors in the network.

To explore the possibilities, imagine a multi-processor constructed of the following elements:

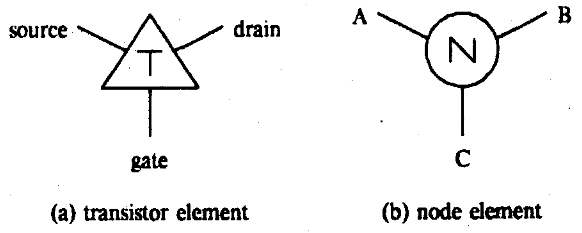


Figure 5.27. *Simulator processing elements*

Both types of elements synchronize their operation to a four-phase global clock:

Phase 1. The transistor element samples the values of its source and drain connections and calculates new values using internal information about its type and current state.

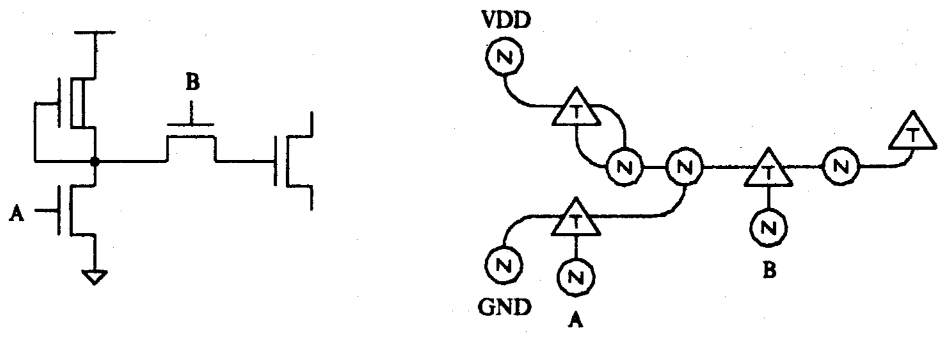
Phase 2. The newly updated values are driven on to the source and drain connections by the transistor elements.

Phase 3. Each node element samples one of its three connections and computes the least upper bound of the sampled value and its stored state. The connections can be sampled in any convenient order; the only requirement is that a connection not be ignored indefinitely.

Phase 4. The node elements drive their connections with the value computed during Phase 3.

Note that the node element is particularly capricious; it ignores two of its three connections in any given cycle. This complicates the notion of an update since there is no guarantee that the two node elements attached to the source and drain connections of a transistor element will be listening when the results of an update are made available. It becomes especially confusing when one of the elements is listening and one is not, which results in "half" an update. Of course, one can conceive of less bizarre node elements, but if it is possible to prove correct operations under the proposed conditions, a much wider class of parallel architectures will be appropriate for the local algorithm.

The elements are wired together in a way that mirrors the topology of the network to be simulated; multiple node elements are used to model network nodes with a large number of connections.



(a) circuit schematic (b) element interconnect

Figure 5.28. *Example wiring diagram for simulator elements*

By providing one processor per transistor and node, this implementation exhibits all the parallelism one could reasonably expect. Steps 1 and 2 of the local algorithm are accomplished in a single clock cycle. During Step 3, an update calculation for each transistor is performed every clock cycle. A wired-or'ed signal visiting all the node elements can detect when the relaxation process is complete; a similar signal connected to all transistor elements can indicate when the network has settled.

This scheme is not as fanciful as it seems — the Connection Machine project [Hillis81] now underway at the M.I.T. Artificial Intelligence Laboratory has an architecture well suited to an implementation similar to the one described above. Fully configured, its one million elements would be able to simulate sizeable circuits at very high speeds. However, the real purpose in proposing this architecture is to provide a vehicle for analyzing the operation of the local algorithm in a parallel environment.

A key insight into the design of a parallel engine is that the value stored by each node element must be non-decreasing with time, *i.e.*, if v_{i_1}, ..., v_{i_t} are the values of node element i at successive clock cycles, then $v_{i_1} \leq \cdots \leq v_{i_t}$. The "ratcheting" of node values up the lattice, which was crucial in showing termination of the relaxation in a sequential implementation, must be preserved in the parallel implementation. With this in mind, consider adding a communications link between two node elements:

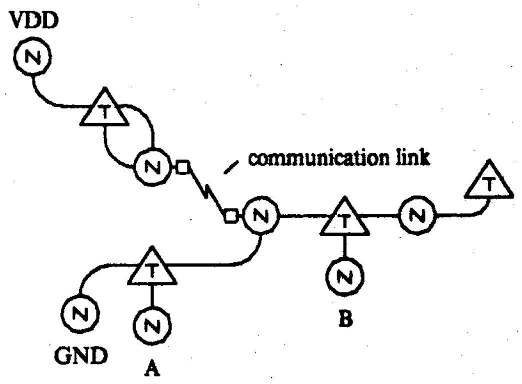


Figure 5.29. *Simulation engine incorporating communication link*

Since the system must already accommodate the unpredictable behavior of node elements, the demands on the link are minimal; messages cannot be garbled and the network cannot become partitioned indefinitely. However, messages can be dropped or delivered in any order since these failures do not affect the monotonicity of a node's value.

Two important questions remain to be answered about parallel implementations that allow conflicting updates:

(1) Is there an analog for Lemma 5.3?

(2) Does this parallel implementation give the same answer as the sequential implementation?

The author's speculation is that both questions can be answered affirmatively. This belief is based on the observations that no information is lost that cannot be recalculated, and the operation of the switches and merging of results remains unchanged. Given that the order in which the propagation happens was shown to be irrelevant by Theorem 5.4, it seems unlikely that the slightly more baroque propagation mechanism of a parallel implementation would seriously change the picture.

CHAPTER SIX

Simulation Using a Pre-compiled Network Model

The simulation algorithms presented in previous chapters rely on examination of the surrounding network to determine the value of a given node. The surrounding network is re-examined every time the node's value needs recalculation. This chapter investigates breaking this process into two steps: a single complete network analysis which builds a set of four logic equations for each node, indicating the types of connections between the node and VDD or GND; and simulation, where the value of each node is determined by evaluating its equations built during the first step. Not only is the overhead of a tree walk avoided each time a node value is calculated, but evaluating logic equations is also a very fast operation for most computers.

Each step is discussed in a separate section. The first section describes the derivation of logic equations for each network node — even those which are not directly outputs of MOS logic gates. The second section presents several approaches for building a logic simulator based on the evaluation of the node equations.

6.1. Reducing switch paths to logic equations

The switch-level algorithm in Chapter 5 determines the value of a node from information about the node's current connections to VDD and GND. The information is regathered each time a new value is calculated for the node. In most cases, only a small number of potential paths exist from a node to VDD and GND. This suggests that it might be economical to determine ahead of time the conditions for which a path exists to, say, GND. For example, the output of a NOR gate with inputs A and B is pulled down if either A or B is non-0. The existence of a pulldown path can be determined by evaluating the expression "A OR B"; a search of the network is not required to discover which pulldowns are currently conducting.

This section describes the derivation of a set of four Boolean equations for each node:

DH_A	An expression indicating under what conditions a path of conducting n-channel and/or p-channel devices exists from node A to VDD.
DL_A	An expression indicating under what conditions a path of conducting n-channel and/or p-channel devices exists from node A to GND.
WH_A	same as DH_A, except the path contains at least one depletion device.
WL_A	same as DL_A, except the path contains at least one depletion device.

If an expression evaluates to true (1), the corresponding path exists; if the expression evaluates to false (0), no path exists. Since nodes can have X values, expressions involving node values can evaluate to X; in this case, the corresponding path may or may not exist. The equations involve the ordinary Boolean operators AND ("·"), OR ("+"), and NOT ("¬"). These operations are easily extended to accommodate X values:

AND	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

NOT	
0	1
1	0
X	X

The algorithm for constructing logic equations is similar to that for computing the Thevenin equivalent for a node (see section 4.1.2). The algorithm begins with an expanding tree walk, stopping when an input or dead-end is reached. During the tree walk, all switches are assumed to be on, since the tree walk is performed before any node values are calculated. (During simulation, the actual state of the switch is represented symbolically in the equation.) The algorithm continues by retracing the steps of the tree walk back toward the original node; during this process, the equations are built. The equations for the terminal nodes are trivial; the following table is the analogue of figure 4.8:

terminal node	*DH*	*DL*	*WH*	*WL*
VDD	1	0	0	0
GND	0	1	0	0
dead-end	0	0	0	0

Merging the equations for two (or more) paths which join at a given node occurs in several steps.

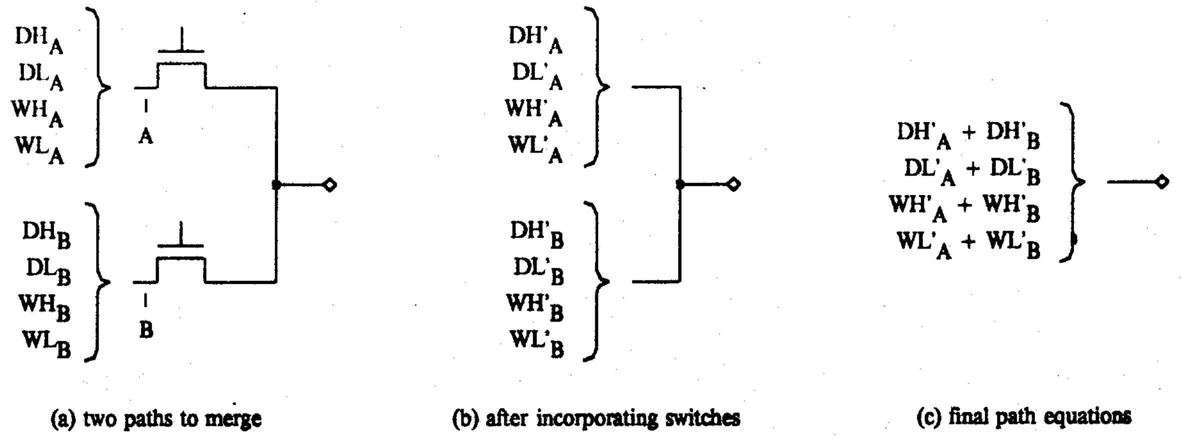


Figure 6.1. *Merging the equations for two paths which join*

The process begins by modifying the equations for each path to reflect the contribution of the switch in series with the path (figure 6.1(b)). The necessary formulas appear below. For example, DH' is the new equation derived by combining DH with *gate*, the value of the switch's gate node.

$$DH' = \left\{ \begin{array}{ll} DH \cdot gate & \textit{n-channel switch} \\ DH \cdot \neg gate & \textit{p-channel switch} \\ 0 & \textit{depletion switch} \end{array} \right. \tag{6.1}$$

$$DL' = \left\{ \begin{array}{ll} DL \cdot gate & \textit{n-channel switch} \\ DL \cdot \neg gate & \textit{p-channel switch} \\ 0 & \textit{depletion switch} \end{array} \right. \tag{6.2}$$

The equations for the "strong" paths (above) are straightforward; when the connection is made by regular switch, the path equation and the the switch's gate value are combined using AND. If the connection is made with a depletion device, the strong path is terminated. Equations for "weak" paths (below) are slightly more complicated since a depletion switch changes a strong path into a weak one. These formulas also reflect the fact that a strong path overpowers a weak path, *i.e.*, equations for weak paths are forced to 0 if a strong path is present. The reason for this extra complication will be clear in an example below.

$$WH' = \begin{cases} gate \cdot WH \cdot \neg DL & \textit{n-channel switch} \\ \neg gate \cdot WH \cdot \neg DL & \textit{p-channel switch} \\ DH + (WH \cdot \neg DL) & \textit{depletion switch} \end{cases} \tag{6.3}$$

$$WL' = \begin{cases} gate \cdot WL \cdot \neg DH & \textit{n-channel switch} \\ \neg gate \cdot WL \cdot \neg DH & \textit{p-channel switch} \\ DL + (WL \cdot \neg DH) & \textit{depletion switch} \end{cases} \tag{6.4}$$

After the equations for each path are modified to include the series switches, they are combined (using OR) to derive the final equations for the node, as shown in figure 6.1(c). When the analysis for a node is complete, the four equations characterize all paths from the node to VDD and GND.

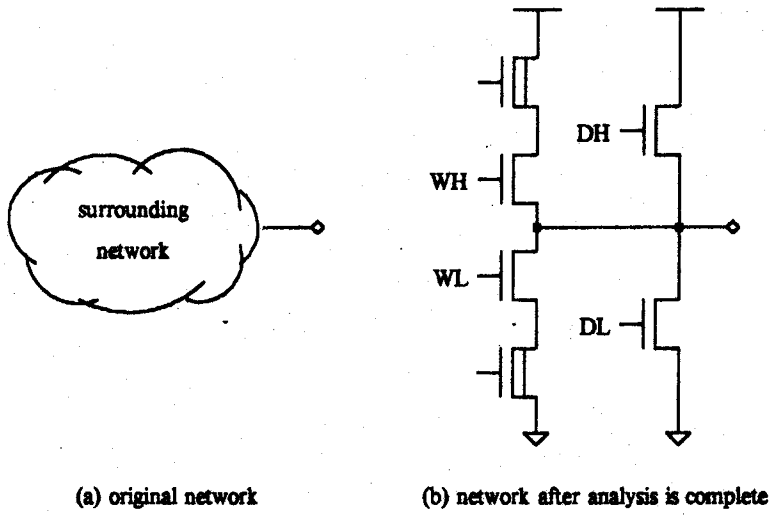


Figure 6.2. *The four equations characterize all paths from node*

In other words, for each node, the surrounding network (figure 6.2(a)) has been reduced to an equivalent, but much simple network (figure 6.2(b)). All the information about paths in the original network is now stored in the node equations, where it can be efficiently utilized. For example, to determine if a node is pulled-down, all one has to do is evaluate the DL equation — no examination of the network is necessary.

The value of node can be determined from the values of the four equations and the node's previous value, by table lookup:

		DH/WH								
		00	01	0X	10	11	1X	X0	X1	XX
	00	prev	1	prev+X	1	1	1	prev+X	1	prev+X
	01	0	X	X	1	1	1	X	X	X
	0X	prev · X	X	X	1	1	1	X	X	X
	10	0	0	0	X	X	X	X	X	X
DL/WL	11	0	0	0	X	X	X	X	X	X
	1X	0	0	0	X	X	X	X	X	X
	X0	prev · X	X	X	X	X	X	X	X	X
	X1	0	X	X	X	X	X	X	X	X
	XX	prev · X	X	X	X	X	X	X	X	X

Table 6.1. *Node value table for equation-based simulation*

There are a few special cases which can be summarized more concisely.† For most nodes in nMOS circuits, $DH = WL = 0$, *i.e.*, connections to VDD are made only through depletion pullups, and depletion devices are not used elsewhere in the circuit. In this case, the value of a node is given by a single equation:

$$\textit{node value} = (WH + \textit{previous value}) \cdot \neg DL \qquad (\textit{when } DH = WL = 0) \tag{6.5}$$

Equation 6.5 can be simplified further for a node that is directly pulled up ($WH = 1$), *i.e.*, a node which is the output of a logic gate:

$$\textit{node value} = \neg DL \qquad (\textit{when } DH = WL = 0 \text{ and } WH = 1) \tag{6.6}$$

In most cases, therefore, calculating the value of a node requires evaluating only a single equation.

Some examples will help illustrate the analysis. First, consider an inverter with a pass gate connected to its output.

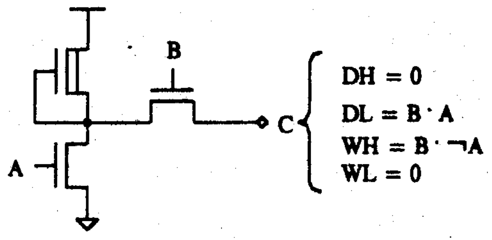


Figure 6.3. *Logic equations for output of inverter with series pass gate*

†Current hardware simulation engines [Pfister82, Zycad83] implement all functions through table lookup, so they can implement the function tabled above as efficiently as, say, Boolean operations. This is not true of most general-purpose machines; hence the motivation for finding simpler representations where possible.

Using equation 6.5, the value of C is given by $C' = (B \cdot \neg A + C) \cdot \neg(B \cdot A)$. The value of this equation is tabled below for the various values of A and B.

	C'	B 0	1	X
	0	C	1	C+X
A	1	C	0	C·X
	X	C	X	X

When B is 0, the pass gate is turned off, and C retains its old value. When B is 1, the pass gate is on, and C is the complement of A. Finally, when B is X, C is also X, except when the output of the inverter is the same as the previous value of C. In this case, the output retains its old value, which makes sense since there is nothing forcing it to change. This last statement is true only because $WH_C = B \cdot \neg A$; the $\neg A$ term forces the pullup equation to 0 when the pulldown of the inverter is active. If the WH equation did not reflect the contribution of the pulldown, *i.e.*, if $WH_C = B$, the value C would be unnecessarily forced to X when the value of B was X.

The next example is the XOR gate presented in Chapter 2.

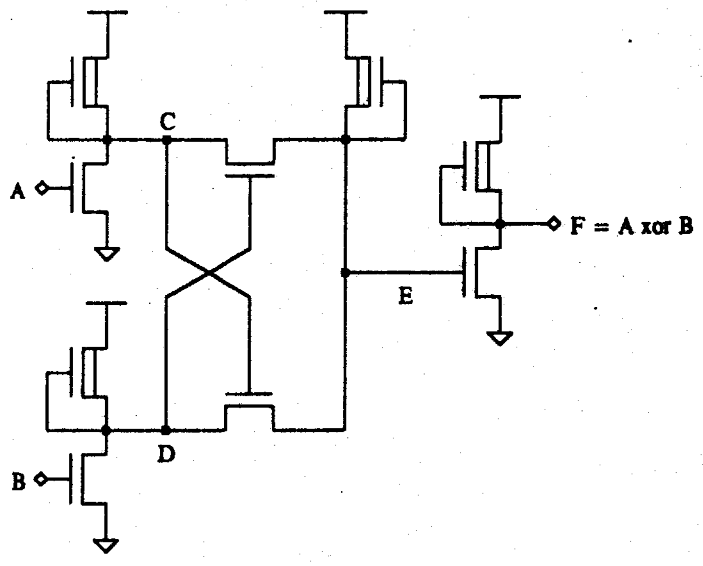


Figure 6.4. ***XOR logic gate***

The equations for each node appear in the following table.

node	*DH*	*DL*	*WH*	*WL*
C	0	A+D·C·B	1	0
D	0	B+C·D·A	1	0
E	0	C·B+D·A	1	0
F	0	E	1	0

These equations might seem incorrect at first — it is not at all obvious that $F = A\ \mathrm{XOR}\ B$. However simplifying the the equations for C and D shows:

$$C = \neg(A + D \cdot C \cdot B) = \neg(A + \neg(B + C \cdot A) \cdot C \cdot B) = \neg A \tag{6.7}$$

and similarly, $D = \neg B$. These results can be used to rewrite the equation for F in terms of A and B:

$$F = \neg E = C \cdot B + D \cdot A = \neg A \cdot B + \neg B \cdot A = A\ \mathrm{XOR}\ B \tag{6.8}$$

In actual use, the equations are not simplified. The above substitutions do verify, however, that the equations compute the correct value for *F*.

Some circuit configurations have very simple connection paths during actual operation of the circuit, but the circuits can appear very complicated when no information is known about the values of various control lines. This is especially true of a circuit containing nMOS switching logic, such as a barrel shifter or tally circuit. If no information is available about the values of the control lines in a barrel shifter, it appears to short together all the incoming and outgoing data bits. The logic equations for a node in such a circuit can become very large — in some cases, large enough to be impractical. The analysis procedure monitors the size of the equations under construction. If they grow too large, the procedure is aborted and the node is flagged. At simulation time, the value of a flagged node is determined using the normal switch-level simulation algorithm.† Flagging a small number of nodes eases the analysis of the remainder of the circuit. (The number of flagged nodes has been less than 1% of the total number of nodes in all the designs processed to date.) Using this technique, the speed-up in simulation afforded by the use of logic equations can be enjoyed by circuits even where 100% conversion to equations is not possible.

Keeping track of gate expressions for transistors crossed during the initial, expanding phase of the tree walk allows the equation-building algorithm to eliminate duplicate AND terms in the results.

†Reversion to ordinary switch-level simulation for especially complicated circuits is easily accomplished by general-purpose computers, but can be next to impossible for special-purpose hardware.

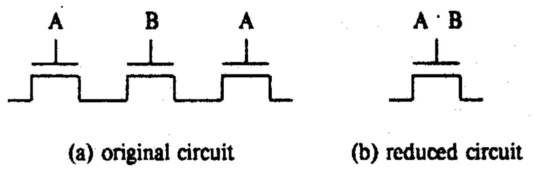


Figure 6.5. *On-the-fly elimination of duplicate AND terms*

This minor optimization can reduce equation size substantially in some circuits. Consider, for example, a tally circuit from [Mead80].

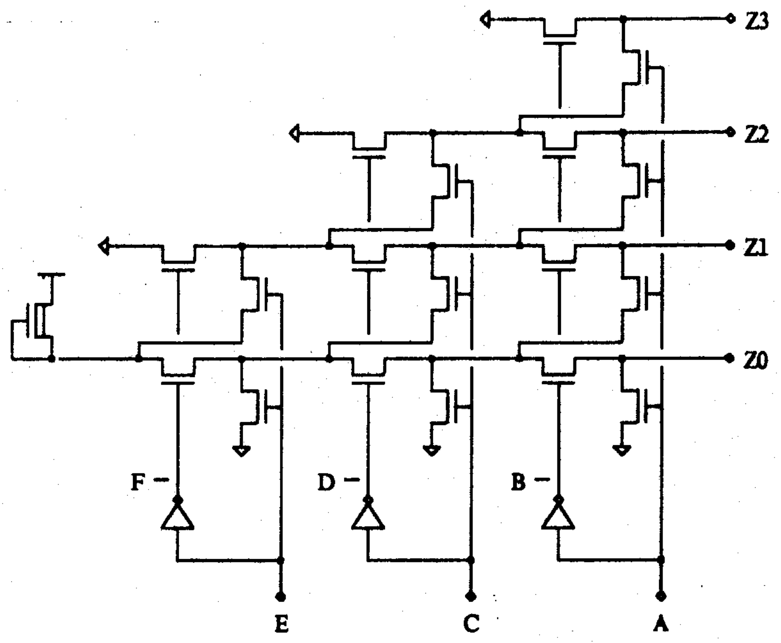


Figure 6.6. *Tally circuit*

This tally circuit has three inputs: A, C, and E. A tally circuit counts the number of 1-inputs; $Z0 = 1$ when no inputs are high, $Z1 = 1$ when exactly one input is high, and so on. The equations produced for the outputs appear somewhat complicated, for example:

$$DL_{Z1} = B{\cdot}(A+D{\cdot}(C+F+E{\cdot}F)+C{\cdot}(D+E+F{\cdot}E)) + A{\cdot}(B+C+D{\cdot}(C+E+F{\cdot}E)) \quad (6.9)$$

$$WH_{Z1} = B{\cdot}(D{\cdot}E+C{\cdot}F+A{\cdot}C{\cdot}E) + A{\cdot}D{\cdot}(F+C{\cdot}(E+B{\cdot}E)) \quad (6.10)$$

These equations are hard to verify as they are, but they can be simplified by removing B, D, and F. (Again, the simulator does not simplify the equations, but this is the easiest method for us to use to verify the operation of the algorithm.) Using the identities $B = \neg A$, $D = \neg C$, and $F = \neg E$, the equations reduce to:

$$DL_{Z1} = \neg A \cdot \neg C \cdot \neg E \; + \; \neg A \cdot C \cdot E \; + \; A \cdot C \; + \; A \cdot \neg C \cdot E \tag{6.11}$$

$$WH_{Z1} = \neg A \cdot \neg C \cdot E \; + \; \neg A \cdot C \cdot \neg E \; + \; A \cdot \neg C \cdot \neg E \tag{6.12}$$

Substituting these formulas into equation 6.5 gives

$$Z1' = A \cdot \neg C \cdot \neg E \; + \; \neg A \cdot C \cdot \neg E \; + \; \neg A \cdot \neg C \cdot E \tag{6.13}$$

As expected, *Z1* is true if exactly one input is high. Of course, evaluating this last equation would be much faster than using the original equations, 6.9 and 6.10. Unfortunately, equation simplification is a very time consuming operation; the computational investment required to process all the equations for a large circuit would probably not be recovered by decreased simulation time. In addition, the equations for most nodes are simple, and simplification beyond that suggested by equation 6.6 (a simplification which is easily recognized) does not result in much improvement.

6.2. Compiling logic equations for simulation

It is easy to build a simulator that uses the node equations developed in the previous section. The simplest approach [Denneau82] is to allocate two node-value arrays; one to hold the current values of each node, and the other to collect new node values as they are computed. Each node is assigned an index which can be used to access its current value in the first array, or to store its new value in the second array. A simulation subroutine for the network is built by generating code that calculates the value of each node, where the code for one node is followed by the code for the next. (Since new node values are kept separate from the current node values, the order in which nodes are processed by the compiler does not matter.) A single simulation step, which propagates new input values to other nodes in the network, is implemented as follows:

(1) For each input node, set its current-value array entry to the designated input value.

(2) Execute the simulation subroutine. This fills the new-value array.

(3) Compare the current-value and new-value arrays. If their contents are identical, the network has settled and the simulation step is over. Otherwise copy the new-value array to the current-value array, and return to step (1).

This simulation algorithm has several interesting properties. Each execution of the simulation subroutine corresponds to one step of a unit-delay simulator. Node values are updated all at once in step (3); hence, the simulator implements a true unit-delay algorithm as described in section 5.3.3. Note that no special handling of input nodes is required when generating code — the new values calculated for input nodes in step (2) are overridden by user-specified values in step (1). Note also that the calculations of the simulation subroutine are not event driven; the implications are discussed below.

The value of a node is computed from its four node equations, using the code generated by one of the following alternatives:

(1) If $DH = WL = 0$ and $WH = 1$, emit code that calculates the node value using equation 6.6.

(2) If $DL = WL = 0$ and $WH \neq 1$, emit code that calculates the node value using equation 6.5.

(3) Otherwise, emit code which evaluates each of the four node equations, and then concatenates the resulting values with the previous value of the node to create an index into Table 6.1. As an optimization, the code generator can check for other special cases (constant values for *WH* and *WL*) and generate accesses to smaller tables if appropriate.

Code is generated for each equation using standard compilation techniques. The logic instructions of the target machine are used for expression evaluation. (Some provision must be made to incorporate X values in a way that still permits use of the native logic instructions; see the example at the end of this section.) Access to a node's current value requires only an indexed reference into the current-value array; storing generated values requires an indexed reference to the new-value array.

There are some inefficiencies inherent in this approach. An extra execution of the simulation subroutine is performed during each simulation step — "extra" in the sense that the last execution produces the same result as the one before (that is how the simulator identifies it as the last execution). In addition, the value of each node is calculated during each call to the simulation subroutine, even if the inputs to the node's equations have not changed.

This last objection can be addressed by making a more intelligent choice about the order in which node values are calculated, by identifying the nodes that affect node *A*'s value (*i.e.*, nodes that appear in the equations for *A*) and then generating code to compute the values of these nodes before generating code to compute the value of *A* [Case78, Denneau82]. In addition, references to a node's current value are directed to the new-value array if a new value for the node was computed earlier in the subroutine. For example, the circuit in the following figure has several cascaded logic gates.

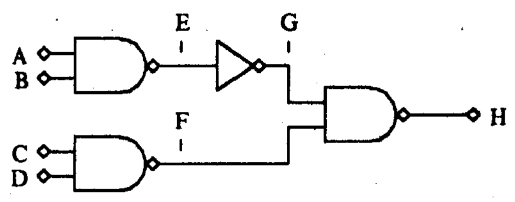


Figure 6.7. *Cascaded logic gates*

Under the new organization, the compiler generates code for nodes A and B before generating code for node E, and so on. The resulting code propagates a new input value from A to H in a single execution. (The earlier scheme would have required three calls to the simulation subroutine to achieve the same effect.)

To implement this scheme, the compiler assigns a numeric *level* to each node. The level of input nodes is defined to be 0; the level of a non-input node α is

$$level(\alpha) = 1 + \max(\ \textit{level of nodes affecting}\ \alpha\ ) \tag{6.14}$$

Referring to the example in figure 6.7, if nodes A through D are inputs, $level(E) = 1$ and $level(H) = 3$. Code is first generated for level 1 nodes, then level 2 nodes, and so on. When compiling an equation, if a node value is needed, the node's level determines where that value comes from. The value of a level 0 node is taken from the current-value array, and the value of a node with a level greater than 0 is taken from the new-value array. (New values are stored in the new-value array, as always.)

The definition of a node's level in equation 6.14 runs into some difficulty if the circuit has feedback. Consider, for example, the following circuit:

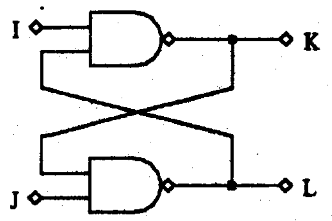


Figure 6.8. *Circuit with feedback*

In attempting to assign a level to node K, one discovers that the definition is circular, *i.e.*, the level of node K is defined in terms of itself. The compiler solves this problem by arbitrarily splitting a node that is in the feedback loop into two nodes. One copy is treated as an input, and the other as a normal network node. Both are assigned the same index so that the input value is updated each time the new-value array is copied to the current-value array. Thus, the circuit in figure 6.8 is compiled as if it had the following configuration:

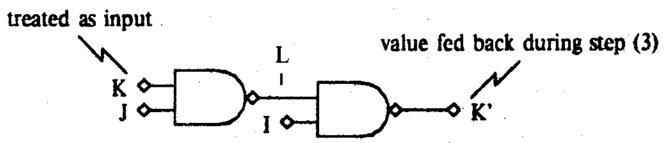


Figure 6.9. *Feedback circuit as it appears to the compiler*

For the purposes of compilation, the feedback loop is broken; the value is actually fed back during step (3) above when the new-value array is copied to the current-value array. This means that a circuit containing feedback might require more than a single execution of the simulation subroutine before the network settles. As it turns out, most MOS circuits contain feedback loops since charge decay requires that storage nodes be refreshed. A clocked feedback loop offers special compilation opportunities, which are discussed below.

Compiling nodes by level ensures that only a single execution of the simulation subroutine is needed to settle the network, assuming the network contains no feedback. The new organization introduces other differences from the original compilation strategy. Node values are not updated all at once in this scheme; the simulation subroutine implements a pseudo unit-delay simulation. Input nodes must be assigned a level of 0, which means nodes must be declared as inputs before the compilation process begins. This eliminates the possibility of interactive debugging, where one wants the capability to consider any node as an input. Typically, the designer uses the original compilation strategy when initially checking out the circuit, and then uses compilation-by-level when performing long verification runs.

Most node-value references are satisfied using the new-value array in the compilation-by-level scheme. This suggests that is might be worthwhile to eliminate the storage overhead and copying time involved for managing two arrays by merging them into a single array. This is straightforward, provided a new technique is developed for detecting when the simulation step is complete. If the circuit has no feedback, only a single execution of the code is needed. When there is feedback, a single execution also suffices, if the current and new value of split nodes (*e.g.*, K and K' in figure 6.9) agree. Only when the old and new values are different is another execution required. This can be arranged by comparing the two values before the new value is stored into the array. If the comparison shows them to be unequal, a flag is set to indicate that another execution is needed. Note that the whole simulation subroutine is re-executed; this is simpler than trying to untangle interlocking feedback loops to determine the subset of the code that must be re-executed.

With this improvement, the compilation-by-level scheme produces a simulation subroutine that:

(i) uses a single node-value array.

(ii) evaluates nodes in a reasonable order: the values of a node's inputs are calculated before the value of the node itself is calculated.

(iii) deals with feedback by splitting some node in the feedback loop into an input node (assigned level 0) and a regular node. Both nodes are assigned the same index, so when the value of the regular node is recomputed it updates the value of the input node also. Before storing the value of a split node into the node-value array, it is compared with the current value; if the values are different a flag is set.

(iv) uses the flag described in step (iii) to indicate when another iteration is needed. If the flag is set during an execution of the code, another iteration is performed; otherwise, the subroutine is finished.

The following is an extended example which illustrates the result of a compile-by-level for a single bit in a nMOS counter. The circuit diagram for the counter bit is shown in the following figure.

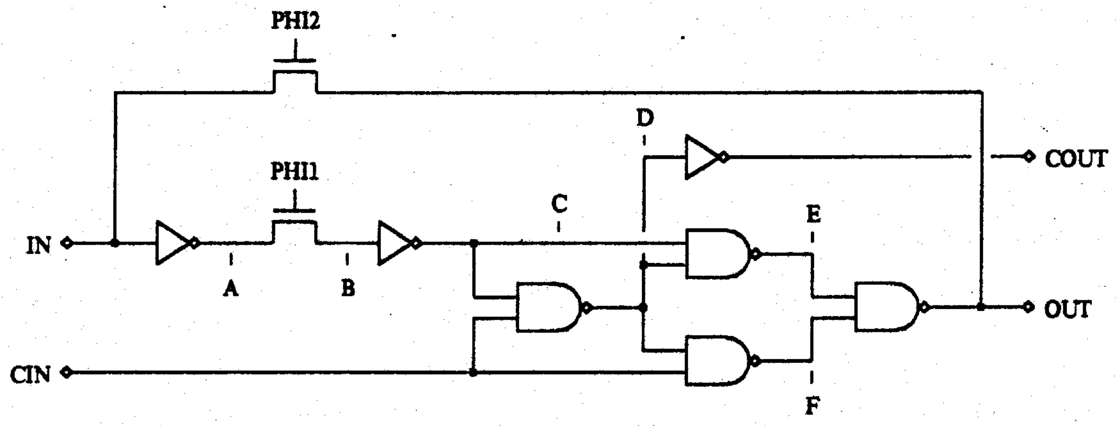


Figure 6.10. *Circuit diagram for a one-bit counter*

The target machine for this example is the DEC VAX-11. A node value is 2-bit quantity (logic low = 0, logic high = 3, X = 1) stored in a byte location; the node-value array is implemented as an array of bytes. Logical AND† and OR instructions produce the desired answers with this value encoding. However, using this encoding, the complement instruction does not correctly implement the NOT

†The VAX does not, in fact, have an AND instruction. Instead, a "bit clear" (BIC in VAX parlance) is provided, which implements an AND-COMPLEMENT operation. This introduces a few circumlocutions in the generated code.

operation, so NOT is performed by table lookup. The index of each node is indicated symbolically in the code below (the index of node *A* is written "_A").

```
; r10 = pointer to value array
; ntbl = table giving NOT of value
; xtbl = table giving bit complement of value
; xntbl = table giving bit complement of NOT of value

step:
    clrl    r0                          ; so regs can be used index registers
    clrl    r1
    movb    #1,iterate_flag             ; nonzero indicates no iteration needed

1:  movb    _PHI2(r10),r0
    bisb3   ntbl(r0),_OUT(r10),r0       ; r0 = !phi2 + out = !(phi2 * !out)
    movb    _OUT(r10),r1
    bicb3   xtbl(r1),_PHI2(r10),r1
    bisb2   _IN(r10),r1                 ; r1 = (phi2 * out) + in
    bisb3   xtbl(r1),r0,_IN(r10)        ; in = r0 * r1

    movb    _IN(r10),r0
    movb    ntbl(r0),_A(r10)            ; a = !in

    movb    _PHI1(r10),r0
    bisb3   ntbl(r0),_A(r10),r0         ; r0 = !phi1 + a = !(phi1 * !a)
    movb    _A(r10),r1
    bicb3   xntbl(r1),_PHI1(r10),r1
    bisb2   _B(r10),r1                  ; r1 = (phi1 * !a) + b
    bicb3   xtbl(r1),r0,_B(r10)         ; b = r0 * r1

    movb    _B(r10),r0
    movb    ntbl(r0),_C(r10)            ; c = !b

    movb    _C(r10),r0
    bicb3   xtbl(r0),_CIN(r10),r0
    movb    ntbl(r0),_D(r10)            ; d = !(c * cin)

    movb    _C(r10),r0
    bicb3   xtbl(r0),_D(r10),r0
    movb    ntbl(r0),_E(r10)            ; e = !(c * d)

    movb    _D(r10),r0
    bicb3   xtbl(r0),_CIN(r10),r0
    movb    ntbl(r0),_f(r10)            ; f = !(d * cin)

    movb    _D(r10),r0
    movb    ntbl(r0),_COUT(r10)         ; cout = !d

    movb    _E(r10),r0
    bicb3   xtbl(r0),_F(r10),r0
    cmpb    ntbl(r0),_OUT(r10)          ; check !(e * f) against old value
    beql    2f
    movb    ntbl(r0),_OUT(r10)          ; if different, save new value
    clrb    iterate_flag                ; and set iterate flag so we do it again
2:

    bbcs    #1,iterate_flag,1b          ; check flag, iterate if set
    rsb
```

The code is a relatively straightforward implementation of the equations for each node. Nodes *PHI1*, *PHI2*, and *CIN* are designated as input nodes. Note that the feedback loop is broken by splitting node *OUT*, an arbitrary choice. The resulting simulation is several orders of magnitude more efficient than a standard switch-level simulation. For example, the value of *B* is calculated in six instructions; the value of *C* in only two. The code is also relatively compact compared to the usual network data base.

Although compiling by level greatly reduces the amount of wasted computation, there are still occasions when the values of nodes are unnecessarily calculated. Some input transitions have little effect on node values; *e.g.*, when *PHI1* or *PHI2* in the one-bit counter above change from 1 to 0. This suggests that the performance of the simulator can be improved by generating multiple simulation routines, where each routine corresponds to a fixed value for one or more inputs. This is particularly advantageous when the inputs selected for special processing have a major impact on the circuit to be simulated. For example, in a circuit using two clocks, three separate simulation routines can be generated: one generated assuming both clocks are low (called, say, CLOCK00), and the other two generated assuming one of the clocks was high (CLOCK10 and CLOCK01). A four-phase clock cycle is simulated by executing the simulation subroutines in the correct order:

```
jsb     clock10                     ; PHI1 high
jsb     clock00                     ; both clocks low
jsb     clock01                     ; PHI2 high
jsb     clock00                     ; both clocks low
```

To generate a input-specific simulation routine, the user specifies which nodes are inputs, and for each input

(1) gives the input's logic value, and

(2) indicates whether the input is stable or has just changed to the specified value.

The compiler applies several optimizations during code generation†: constant folding based on knowledge of input node values, and compile-time selective trace that ignores nodes whose values remain unchanged. (The stable/changing specification is used by the selective trace optimization.) The selective trace is especially effective in reducing the amount of generated code.

In the examples below, *PHI1* and *PHI2* are specified as changing inputs, and *CIN* an unchanging input. The first example — the code generated for the one-bit counter with both clocks low — illustrates just how effective the optimizations can be:

†The optimizations are inspired by those found in traditional optimizing compilers [Harrison77, Wulf75]. Because of the branch-free nature of the code and the pervasive influence of clock signals, many of the optimizations are much more effective in this domain than in traditional compilation problems.

```
clock00:                                ; code for phi1 = 0, phi2 = 0, cin = 1
    clrb    _PHI1(r10)                  ; phi1 = 0
    clrb    _PHI2(r10)                  ; phi2 = 0
    rsb
```

The values of *PHI1* and *PHI2* are set by the code since they are specified as changing inputs. (The value an unchanging input is assumed to be set by the user, or by code executed earlier.) Node *B* is determined to be unaffected by the change in *PHI1*, as are nodes *IN* and *PHI2*. In fact, the compile-time selective trace does not find any nodes that change value, except for the changing inputs.

The next code sequence, corresponding to *PHI1* going high, is somewhat longer, since that is the transition when the circuit performs most of its work.

```
clock10:                                ; code for phi1 = 1, phi2 = 0, cin = 1
    clrl    r0                          ; so reg can be used as index register
    movb    #3,_PHI1(r10)               ; phi1 = 1
    clrb    _PHI2(r10)                  ; phi2 = 0
    movb    _A(r10),_B(r10)             ; b = a
    movb    _B(r10),r0
    movb    ntbl(r0),_C(r10)            ; c = !b
    movb    _C(r10),r0
    movb    ntbl(r0),_D(r10)            ; d = !(c * cin) = !c
    movb    _D(r10),r0
    movb    ntbl(r0),_COUT(r10)         ; cout = !d
    movb    _C(r10),r0
    bicb3   xtbl(r0),_D(r10),r0
    movb    ntbl(r0),_E(r10)            ; e = !(c * d)
    movb    _D(r10),r0
    movb    ntbl(r0),_F(r10)            ; f = !(d * cin) = !d
    movb    _E(r10),r0
    bicb3   xtbl(r0),_F(r10),r0
    movb    ntbl(r0),_OUT(r10)          ; out = !(e * f)
    rsb
```

A node that connects to the rest of the network through a single pass transistor (*e.g.*, node *B* in the counter) is treated specially by the compiler, because such nodes are so common in MOS networks. When the pass transistor is turned on by fixed-value input, the generated code is particularly efficient (a single move in the example above).

The last code sequence, corresponding to *PHI2* going high, is relatively short; the compile-time selective trace finds only a few nodes whose values needed to be computed.

```
clock01:                                ; code for phi1 = 0, phi2 = 1, cin = 1
    clrl    r0                          ; so reg can be used as index register
    clrb    _PHI1(r10)                  ; phi1 = 0
    movb    #3,_PHI2(r10)               ; phi2 = 1
    movb    _OUT(r10),_IN(r10)          ; in = out
    movb    _IN(r10),r0
    movb    ntbl(r0),_A(r10)            ; a = !in
    rsb
```

Simulation of a four-phase clock cycle using these three routines requires executing only 36 VAX instructions. The earlier compiled code sequence requires 39 instructions for a single simulation step, for a total of more than 150 executed instructions when simulating a full clock cycle. Input-specific subroutines result in a considerable improvement.

Although the impact of compile-time selective trace makes it a worthwhile optimization, only so many input-specific routines can be generated. Assuming that all combinations of inputs are possible, the number of routines needed grows exponentially with the number of fixed inputs. Thus, while computations caused by the changing of a few inputs can be reduced to the bare minimum, many unnecessary computations are still performed. For example, in a 10-bit counter, the nodes comprising the higher data bits are recomputed during each clock cycle, even though those nodes actually change value far less frequently. Presumably, the appropriate checks could be inserted into the code, resulting in branches around sections of code that do not need to be executed. In the counter example, when the carry-in of a data bit is zero, the code for its level and all higher levels does not need to be executed. However, a very sophisticated compiler would be needed to handle this situation. It is unclear what further gains will be possible in the search to reduce unnecessary computation.

In summary, the compilation techniques discussed in this chapter are well-suited for producing code that implements a fast switch-level simulation of a stable design. The potential increase in simulation speed allows more exhaustive checkout than is possible with interactive (and slower) simulators. Compilation-based simulation is most appropriate for a circuit with a high degree of circuit activity; if each circuit component is active during each simulation step, there is very little unnecessary computation by the simulation subroutine. On the other hand, for a large circuit with little activity, an event-driven interactive simulator might actually outperform a compiled simulation. Fortunately, not many designers strive for designs in this latter category.

CHAPTER SEVEN

CONCLUSIONS

The models and simulators presented in this thesis were developed to fill the need for simulation tools suitable for large MOS designs. At the outset of the project, there were surprisingly few alternatives; even today, much of the work in the area of simulation tools concentrates on refurbishing traditional gate-level simulators and circuit analysis programs. (The current state of these efforts is outlined at the end of the chapter.) The work reported here takes a different approach, seeking to develop new algorithms, guided by the following goals:

(1) The algorithms must be suitable for the logic-level simulation of large digital MOS circuits; "large" meaning circuits containing 10,000 to 50,000 transistors.

(2) Important aspects of MOS behavior (bidirectionality, charge sharing/storage, pullup/pulldown ratios, etc.) should be modeled in a useful way.

(3) Performance estimates should be calculated directly from the actual parameters of the circuit components. Ideally, the calculations are based on the same rules of thumb used by designers when estimating circuit performance.

The RSIM simulator meets all three goals, while maintaining a reasonable balance between simulator performance and accuracy of predictions. Rather than performing a detailed simulation of each transistor's operation, RSIM uses the linear model to directly predict the logic state of each node and to estimate transition times when nodes change state. The net effect is a trade of some prediction accuracy for an increase in simulation speed. When the linear model is conservatively calibrated, its predictions can be used to identify problem circuits in need of more accurate analysis. Usually, a large percentage of a circuit passes the scrutiny of RSIM, and so the expense associated with detailed simulation of the whole circuit is avoided. In addition to serving as the basis for simulation, the linear model can be used in timing analysis and might serve to quickly generate initial waveforms for a relaxation-based circuit analysis program.

RSIM has been in use in both university and industrial environments since the spring of 1982. During that time it has simulated several hundred designs, ranging in size from very small to approximately 40,000 transistors. Because RSIM is fast enough to simulate a whole circuit, it often uncovers circuit flaws that have fallen between the cracks during the simulation of smaller pieces of the design. The trend shows that RSIM is viewed as a companion to circuit analysis, using it for all logic-level verification and preliminary timing analysis, and resorting to circuit analysis for those paths identified as critical by RSIM.

The simulation algorithm is embedded in a LISP-like command language [Terman82] that has been used to write quite elaborate programs to drive the simulation and process the results. Since programs to prepare simulation input are much less tedious to construct than the input itself, designers have been able to conduct more tests than they might otherwise do. For example, it is a simple matter to use a set of test vectors that drive a register-transfer-level simulation as input to an RSIM run, and compare the predictions of the two simulations, all under program control.

With careful calibration, RSIM's predictions for combinational logic are within 30% of those of SPICE. For circuits relying on analog behavior (sense amplifiers, bootstrapped nodes, etc.) or chains of pass devices, the predictions are less accurate. To compensate, several "escape" mechanisms exist which allow the designer to specify the logic thresholds and transition times for individual nodes so that the results of more detailed simulations can be incorporated into RSIM. Usually this mechanism need be invoked for only a few critical nodes (*e.g.*, clock driver outputs). Another alternative is to identify subcircuits and replace them with logically equivalent circuits that can be simulated easily; a network preprocessor [Iler83] that performs subcircuit matching and replacement is available and has been used to good effect. With these enhancements, RSIM has proved to be a fairly reliable filter for detecting circuits in need of more careful analysis.

For those stages of the design process that do not require performance information, a switch model might be more appropriate than a linear model. A switch-level simulation is particularly useful in the early stages of a design when one is experimenting with the organization of the logic, and sizing each device would be distracting. The switch models presented in this thesis are straightforward, especially in the treatment of X values and their effect on the network. The switch model as embodied in ESIM (which uses the global algorithm outlined in Chapter 5) is quite compatible with the linear model used in RSIM. In fact, in the current implementation both models exist side by side and one can choose either model when propagating a set of changes through the network. This flexibility is useful during initialization of a network, when performance information is not a major concern.

Simulator performance is always an important issue, one that has been addressed throughout the thesis. Chapter 4 describes several techniques for speeding up the RSIM algorithm; using a compressed representation of logic gates and caching subnetwork calculations decreases the execution time of RSIM by a factor of two or more. The local switch algorithm presented in Chapter 5 is ideal for implementation on parallel architectures. Like many relaxation algorithms, it can effectively utilize many processors, and so holds the promise of large performance improvements in simulation when parallel processors move out of the experimental stage. A different approach for improving the performance of switch-level simulation is described in Chapter 6, which proposes performing the network analysis once, before simulation, and using the results to compile a set of logic equations for each node. When evaluated in the proper order by a conventional computer, the resulting switch-level simulation is many times faster than simulation using traditional techniques. The node equations can also be used to develop instruction sequences for special-purpose simulation hardware — *e.g.*, the Yorktown Simulation Engine, or the Zycad multi-processor — extending the benefits of high-speed gate evaluation to arbitrary MOS networks [Barzilai83].

The remainder of this chapter discusses other work in the area of simulation related to the topics of concern in this thesis. These topics include:

- algorithms for fast circuit analysis; circuit analysis using simplified models
- mixed-mode simulation
- logic-level simulation using pre-determined transition delays
- models for estimating circuit performance
- other switch-level simulation algorithms

Each of these areas is discussed below.

The most detailed and accurate network simulation is provided by circuit analysis programs, such as ASTAP [Weeks73] or SPICE [Nagel75]. The capacity limitation of circuit analysis is a prime motivation for the development of simpler simulation models; recent improvements in circuit analysis algorithms are making inroads into the traditional performance problems of circuit analysis. Device models are the heart of a circuit analysis program. The models are usually analytic; they contain formulas that predict device performance from information about voltage histories, physical properties of materials, etc. In a circuit, the behavior of a particular device might be determined by several electrical nodes which, in turn, are affected by other devices; *i.e.*, a system of circuit equations is needed to describe the behavior of the circuit as a whole. To make finding a solution computationally feasible, most circuit analysis programs proceed in two steps:

(1) The circuit is partitioned so that, at a particular time step, the change in voltage on each node is approximated as a linear function of the node voltages (and their derivatives). It is during this step that device models must be evaluated.

(2) Solving the resulting set of equations numerically (see [SV80]).

These two steps can be quite time consuming, although for large circuits the second step becomes the dominant factor [Newton80]. RSIM reduces both costs by using a very simple device model whose effects can be predicted without the need for expensive numerical techniques.

The cost of model evaluation can be reduced by replacing the analytic device models with tables relating device current to terminal voltages [Chawla75, Fan77]. These tables can be derived from a one-time evaluation of the original analytic models, or filled directly from device measurements. In these simulators, the current charging/discharging of each node capacitor is determined from the present node voltages; thus, the change in node voltage for each time step can be calculated directly and the cost of solving a set of simultaneous equations is avoided. Another approach to reducing the cost associated with dealing with large matrices of equation coefficients uses a relaxation technique [Lelarasmee81, Newton83] to successively approximate the voltage waveform for each node in the circuit. The solution for each node is computed separately, using the estimates of other node voltages computed during earlier iterations. Again, this avoids the cost of solving a large set of simultaneous equations. It is also possible to skip the recalculation of a node's waveform during a particular iteration if it can be determined that the estimates for the surrounding network have not changed substantially since the last iteration (*i.e.*, selective-trace comes to circuit analysis). These techniques can speed up circuit analysis by an order of magnitude or more, but the programs are still limited to circuits with a few thousand components.

Recent work on simulators has tried to combine the computational advantages of gate-level simulation with the precision afforded by circuit analysis; this has lead to a new family of mixed-mode simulators: [Chen78, Gardner79, Hill79, Agrawal80, Newton80]. The designer can specify gate-level or functional simulation for simple or previously-verified pieces of the circuit, reserving the expense of circuit analysis for critical sections of the design. There are two problems that remain to be solved in mixed-mode simulators: conversion between the different representations of node values used by the different levels, and the related problem of choosing which type of simulation should be used for each subcircuit. The designer can introduce errors into the simulation by an unfortunate choice of level at a critical point in the circuit; special care must be exercised to avoid discontinuities and other pitfalls of the numerical solution techniques. Like circuit analysis, mixed-mode analysis still requires the touch of an expert lest it produce misleading results.

Clearly, it is only a matter of time before mixed-mode simulation becomes true hierarchical simulation in which the results of detailed low-level simulation are automatically summarized for use in subsequent high-level simulations. A hierarchical system would also decide what level of simulation is appropriate for each subcircuit. Viewed in this light, RSIM can be thought of as the first step toward automatic identification of critical subcircuits. With a foot in both worlds, RSIM provides an easy path for descending into circuit analysis or for abstracting toward higher-level logic functions.

Another approach to timing simulation that retains the speed advantages of gate-level simulation is determining the transition delays for each node before simulation begins. Some gate-level simulators [Szygenda72, Case78] allow the user to assign node delays. This type of simulator can be extended to handle MOS networks, after a fashion [Sherwood81, McDermott82]. The result is a system that can quickly calculate estimates for signal delays in a network. Unfortunately, the delays are not calculated automatically (and hence are prone to error or wishful thinking on the part of the designer), and are approximate at best for pass transistor circuits so common in MOS circuits. A more effective technique for pre-computing delays is the use of the results of actual measurements or circuit analysis runs [Pilling73, Nahm80]. The delays are measured/calculated for "standard" gate configurations, and the results used to estimate the performance for the actual configuration of each node in the network. [Nahm80] mentions several shortcomings of this approach. Circuits with multiple inputs are difficult to analyze since a particular input transition is chosen when performing the analysis; also, the effect of overlapping input transitions, the slope of the input waveform, and dynamic changes in the output load are not considered. (Interestingly, all these problems are solved in a straightforward way by RSIM, at no great loss in execution speed, as evidenced by the performance figures quoted by Nahm.) [Okasaki83] suggests overcoming these problems by expanding the set of "standard" configurations to include most of those commonly found in MOS circuits (complex and/or gates, pass gates, etc.). The price for the increase in accuracy is a corresponding increase in the complexity of the model for each gate; his simulator spends a fair amount of time determining which pre-computed delay should be used, given the current configuration of the network. In summary, the performance variations introduced by non-standard circuit configurations, and changes in the network due to changing node values seem to offset any advantages offered by pre-determined transition delays.

Not much has been published about models that are suitable for quickly determining the transition times for particular network configurations. A switched linear Thevenin model is described in [Glasser80]; a simulator based in part on this model is described in [Tamura83]. Multiple resistances are used to describe each transistor; conceptually, the appropriate resistance is selected by a rotary switch controlled by the transistor's gate voltage. Each resistance is chosen to model the actual channel resistance in a particular region of device operation. The linear model presented in this thesis can be viewed as a simplification of Glasser's model, with only two possible switch positions selecting between resistances of R_{eff} and ∞. A simple version of the linear model also appears in [Ousterhout83] and [Jouppi83]; both indicate that the model improvements suggested in Chapter 3 are needed in order to improve prediction accuracy. [Horowitz83] presents a simple model that describes the performance of a network of pass gates; his model is discussed in section 3.5.

One simulator with many of the same aspirations as the switch-level simulators described in Chapter 5 is MOSSIM, written by Randy Bryant [Bryant81]. MOSSIM uses a switch transistor model similar to that presented here, but its calculations are organized differently since (1) node values are represented using a cross-product value set and (2) the analysis is based on a static decomposition of the network. A major difference in the simulation calculation comes in the handling of X values and their effect on the surrounding network. Bryant handles such values in a separate stage of the computation, using global knowledge of the network configuration to resolve values of subnetworks connected by X transistors. (Other differences between the two approaches are discussed in Chapters 2 and 5.) The extra complexity of his algorithms results in some degradation in simulator performance over that achieved by the simulators described here.

APPENDIX ONE

Proof of Lemma 5.3

Lemma 5.3. Let W, X, and Y be network states. If $W \rightarrow_1 X$ and $W \rightarrow_1 Y$, then there exists a network state Z such that $X \rightarrow Z$ and $Y \rightarrow Z$.

Recalling how the update operation works, it is not hard to believe that the Lemma is true. The value of a node indicates the resistance of paths from the node to VDD and GND. An update exchanges path information across a switch, and the $\cup$ operation ensures that information is never lost (the indicated resistance to an input can never increase). Intuitively, an update only adds information about possible paths to the network state, so no matter what switch is chosen for an update, one can also go back to other switches latter on.

The proof is straightforward, demonstrating how a state Z can be constructed for each possible X and Y. The proof depends on some simple properties of the $\cup$ operation and the *switch* function:

$$\begin{aligned} A \cup A &= A \\ A \cup B &= B \cup A \\ \alpha \cup switch(\sigma, \alpha) &= \alpha \\ switch(\sigma, switch(\sigma, \alpha)) &= switch(\sigma, \alpha) \\ switch(\sigma, \alpha \cup \beta) &= switch(\sigma, \alpha) \cup switch(\sigma, \beta) \end{aligned} \tag{A1.1}$$

which can be verified directly from the definition of $\cup$ and equation 5.9.

If the two updates leading to states X and Y involve only one switch, $X = Y$ and the Lemma is trivially true. If two separate switches are involved, there are three cases to consider which differ in the number of nodes affected.

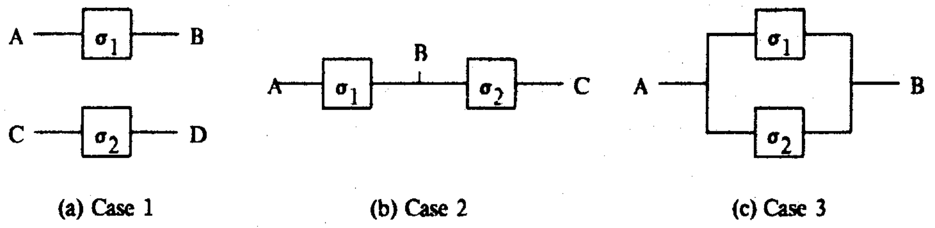


Figure A1.1. *Three cases in proof of Lemma 5.3*

For notational convenience, define the functions f and g to describe the effects of switch 1 and 2 respectively:

$$\begin{aligned} f(\alpha) &\equiv switch(\sigma_1, \alpha) \\ g(\alpha) &\equiv switch(\sigma_2, \alpha) \end{aligned} \tag{A1.2}$$

Each of the two updates is labeled by the switch it operates on; for example, S_1 refers to an update involving switch 1. A sequence of updates is written as S_iS_j, which is taken to mean update S_i, followed by update S_j.

Case 1: no nodes in common. As the following diagram indicates, when the updates have no nodes in common, they result in the same state when applied in either order.

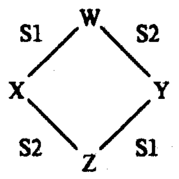


Figure A1.2. *State diagram when no nodes in common*

This is shown by considering the values for nodes A, B, C, and D after each update:

sequence	A	B	C	D
S_1	A ∪ f(B)	B ∪ f(A)	C	D
S_1S_2	A ∪ f(B)	B ∪ f(A)	C ∪ g(D)	D ∪ g(C)
S_2	A	B	C ∪ g(D)	D ∪ g(C)
S_2S_1	A ∪ f(B)	B ∪ f(A)	C ∪ g(D)	D ∪ g(C)

The final states of the two sequences are the same, demonstrating the desired network state, Z.

Case 2: one node in common. As the following diagram indicates, when the updates have one node in common, $S_1S_2S_1$ is equivalent to $S_2S_1S_2$.

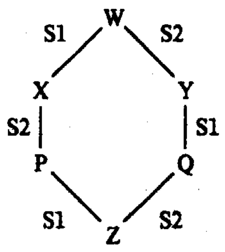


Figure A1.3. *State diagram when one node in common*

This is shown by considering the values for nodes A, B, and C after each update:

sequence	A	B	C
S_1	A ∪ f(B)	B ∪ f(A)	C
S_1S_2	A ∪ f(B)	B ∪ f(A) ∪ g(C)	C ∪ g(B ∪ f(A))
$S_1S_2S_1$	A ∪ f(B) ∪ f(B ∪ f(A) ∪ g(C))	B ∪ f(A) ∪ g(C) ∪ f(A ∪ f(B))	C ∪ g(B ∪ f(A))
S_2	A	B ∪ g(C)	C ∪ g(B)
S_2S_1	A ∪ f(B ∪ g(C))	B ∪ g(C) ∪ f(A)	C ∪ g(B)
$S_2S_1S_2$	A ∪ f(B ∪ g(C))	B ∪ g(C) ∪ f(A) ∪ g(C ∪ g(B))	C ∪ g(B) ∪ g(B ∪ g(C) ∪ f(A))

Using the identities in equation A1.1, the final values of A, B, and C for each sequence can be simplified to

$$\begin{aligned} A_{final} &= A \cup f(B) \cup f(g(C)) \\ B_{final} &= B \cup g(C) \cup f(A) \\ C_{final} &= C \cup g(B) \cup g(f(A)) \end{aligned} \tag{A1.3}$$

The final states of the two sequences are the same, demonstrating the desired network state, Z.

Case 3: two nodes in common. As in Case 1, when the updates have no nodes in common, they result in the same state when applied in either order. This is shown by considering the values for nodes A and B after each update:

sequence	A	B
S_1	$A \cup f(B)$	$B \cup f(A)$
S_1S_2	$A \cup f(B) \cup g(B \cup f(A))$	$B \cup f(A) \cup g(A \cup f(B))$
S_2	$A \cup g(B)$	$B \cup g(A)$
S_1S_2	$A \cup f(B \cup g(A))$	$B \cup g(A) \cup f(A \cup g(B)$

Again, using the identities in equation A1.1, the final values of A and B for each sequence can be simplified to

$$\begin{aligned} A_{final} &= A \cup f(B) \cup g(B) \\ B_{final} &= B \cup g(A) \cup f(A) \end{aligned} \tag{A1.4}$$

The final states of the two sequences are the same, demonstrating the desired network state, Z. ∎

APPENDIX TWO

RSIM Calibration Tables for a 5μ nMOS Process

RSIM's transistor model relies in part on three modeling resistances for each transistor in the network:

R_{static}	for calculating V_{thev},
R_{dynlow}	for calculating the transition time for high-to-low transitions, and
$R_{dynhigh}$	for calculating the transition time for low-to-high transitions.

These resistances are chosen for each transistor on the basis of its geometry, type, and usage in the circuit. The static resistance is chosen to obtain a good prediction for the 0-output voltage of a logic gate. Actually this constrains only the ratio of the n-channel and pullup static resistances, so there is considerable freedom in choosing these values.

The dynamic resistances for each transistor type are specified in the following diagram. Because of their special nature, depletion devices configured as pullups are treated separately from other depletion devices.

transistor type	R_{dynlow}	$R_{dynhigh}$
n-channel	Tables A2.1 & A2.2	Table A2.3
depletion	(see text)	Table A2.4
pullup	—	Table A2.5

The tables appear at the end of this appendix. R_{dynlow} is not needed for a pullup, but might be needed for other configurations of depletion devices (*e.g.*, if one appeared in a pulldown path). If desired, a very high R_{dynlow} can be specified for depletion devices to flag the use of a depletion device in a pulldown path.

The tables below were prepared by analyzing the simple SPICE experiments proposed in section 2.4. As mentioned in that section, more sophisticated experiments might be more appropriate for designers who wish to push RSIM to its limits. These tables are used by examples in the thesis; for actual simulation, some of the values should be derated (increasing the resistance) to ensure conservative estimates.

The experiments were run using version 2G.5 of SPICE with the following device models (a typical 5μ nMOS process):

```
MODEL ENH NMOS (LEVEL=2 VTO=1.0 PHI=0.55 GAMMA=0.4 CGSO=4.5E-10 PB=0.85
JS=1E-18 CJ=7.2E-5 CJSW=3.6E-10 TOX=1E-7 NSUB=1.0E15 XJ=1E-6 LD=0.7E-6
UO=690 UCRIT=1E5 UEXP=0.12 MJ=0.5 MJSW=0.27)

MODEL DEP NMOS (LEVEL=2 VTO=-3.3 PHI=0.55 GAMMA=0.47 CGSO=4.5E-10 PB=0.85
JS=1E-18 CJ=7.2E-5 CJSW=3.6E-10 TOX=1E-7 NSUB=1.0E15 XJ=1E-6 LD=0.7E-6
UO=690 UCRIT=1E5 UEXP=0.12 MJ=0.5 MJSW=0.27)
```

Rise time is measured as the length of time needed for an output to rise from 0 volts to 2.134 volts — the balance point of a 4:1 inverter built using this process. (Section 3.3.1 explains why the balance point is chosen as the threshold.) Fall time is the length of time needed for an output to fall from 5 volts to the threshold.

Note that widths and lengths are shown in microns, and the table values are in units of KΩ per square of channel; one must multiply the appropriate table entry by the number of squares of channel (length÷width) to get a transistor's resistance. For table entries marked "*", no value is available because of a SPICE bug.

R_{enh}		Length						
		5	10	20	30	40	50	100
	5	8.7	13.6	16.2	17.1	17.5	17.8	18.4
	10	8.8	13.7	16.2	17.1	17.6	17.8	18.5
	20	8.8	13.8	16.3	17.3	17.8	18.0	18.9
Width	30	9.0	13.8	16.5	17.4	17.9	18.2	19.2
	40	9.6	14.0	16.6	17.6	18.1	18.5	19.6
	50	10.0	14.0	16.8	17.7	18.3	18.7	20.0
	100	10.0	15.0	17.0	18.7	19.3	19.8	21.9

Table A2.1. *Channel resistance ($K\Omega/\square$) for n-channel pulldowns*

$R_{enh\text{-}thresh}$		Length						
		5	10	20	30	40	50	100
	5	16.0	26.3	31.5	33.3	34.1	34.6	35.6
	10	16.6	26.9	32.1	33.7	34.6	35.0	35.9
	20	17.6	28.0	32.9	34.4	35.1	35.5	35.5
Width	30	18.6	28.8	33.5	34.8	35.4	35.8	36.4
	40	19.2	29.6	33.8	35.1	35.7	36.0	36.6
	50	20.0	30.0	34.3	35.3	35.9	36.2	36.8
	100	22.0	31.0	35.5	36.3	36.8	37.0	37.6

Table A2.2. *Channel resistance ($K\Omega/\square$) for n-channel pulldowns with threshold drops*

$R_{enh\text{-}sf}$		Length						
		5	10	20	30	40	50	100
	5	12.6	22.8	28.8	31.2	32.5	33.5	36.7
	10	12.8	23.1	29.5	32.2	34.0	35.4	40.5
	20	12.8	23.6	30.8	34.3	26.9	39.0	48.1
Width	30	13.2	24.3	32.1	36.5	39.8	42.7	55.7
	40	13.6	24.8	33.6	38.5	42.7	46.4	63.3
	50	14.0	25.5	35.0	40.7	45.6	50.1	70.9
	100	14.0	28.0	41.5	51.3	60.3	68.6	-

Table A2.3. *Channel resistance ($K\Omega/\square$) for n-channel source-followers*

$R_{dep\text{-}sf}$		Length						
		5	10	20	30	40	50	100
Width	5	3.0	4.3	5.0	5.3	5.4	5.6	6.0
	10	3.0	4.3	5.1	5.4	5.7	5.9	6.6
	20	*	4.4	5.3	5.8	6.2	6.4	7.8
	30	*	4.5	5.6	6.2	6.6	7.0	8.9
	40	*	4.8	5.8	6.5	7.1	7.6	10.1
	50	*	5.0	6.0	6.8	7.5	8.2	11.3
	100	*	*	7.5	8.7	10.0	11.2	17.2

Table A2.4. *Channel resistance (KΩ/□) for depletion source-followers*

R_{dep}		Length						
		5	10	20	30	40	50	100
Width	5	8.8	15.1	18.6	19.9	20.4	20.8	21.6
	10	8.8	15.2	18.7	19.9	20.5	20.8	21.6
	20	*	15.2	18.8	20.0	20.6	21.0	21.7
	30	*	15.3	18.9	20.1	20.7	21.0	21.8
	40	*	15.5	19.0	20.1	20.8	21.1	21.9
	50	*	15.5	19.0	20.3	20.9	21.3	22.0
	100	*	*	19.5	20.7	21.5	21.8	22.5

Table A2.5. *Channel resistance (KΩ/□) for depletion pullups*

APPENDIX THREE

Approximation for Resistor Divider and Series Resistor

As part of the incremental computation for the Thevenin equivalent of a network, it is necessary to approximate a resistor divider and series resistance (figure A3.1(a)) by a simple resistor divider (figure A3.1(b)).

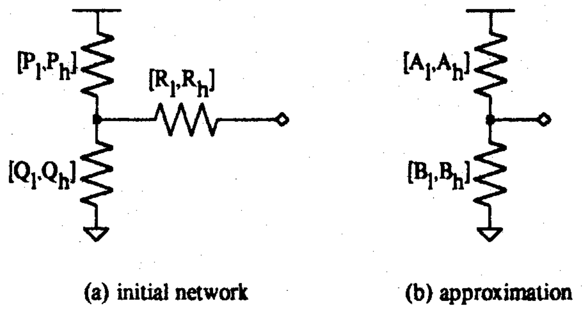


Figure A3.1. *Initial resistor network and desired approximation*

As usual, each resistance is potentially a resistance interval. An exact choice for the modeling resistance is impossible (as will be shown below) so the goal of this appendix is the choice a suitable approximation.

Consider a resistor divider with pullup resistance P and pulldown resistance Q.

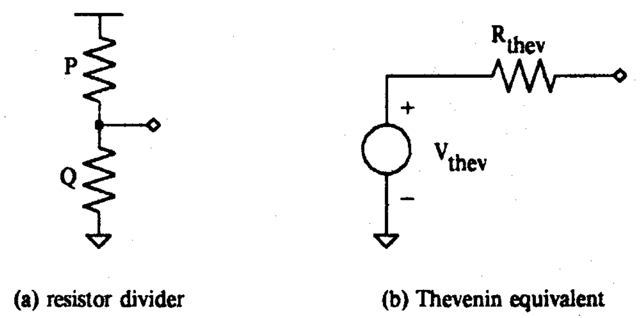


Figure A3.2. *Resistor divider and Thevenin equivalent*

The parameters of the Thevenin equivalent are

$$V_{thev} = \frac{Q}{P+Q} \quad \text{and} \quad R_{thev} = P \,||\, Q \tag{A3.1}$$

which can be rearranged as linear equations relating R_{thev} and V_{thev}:

$$R_{thev} = P\ V_{thev} \quad \text{and} \quad R_{thev} = Q\ (1 - V_{thev}) \tag{A3.2}$$

If P and Q are intervals — $P = [P_l, P_h]$ and $Q = [Q_l, Q_h]$ — then the Thevenin parameters also are intervals:

$$V_{thev} = [\ \frac{Q_l}{Q_l + P_h}, \frac{Q_h}{Q_h + P_l}\ ] \quad \text{and} \quad R_{thev} = [\ P_l \,||\, Q_l, P_h \,||\, Q_h\ ] \tag{A3.3}$$

If one plots the Thevenin parameter values (R_{thev} *vs.* V_{thev}), as P and Q are varied independently over their respective intervals, equation A3.3 suggests the resulting area would be rectangular, but this is not the case, as is illustrated by the following figures.

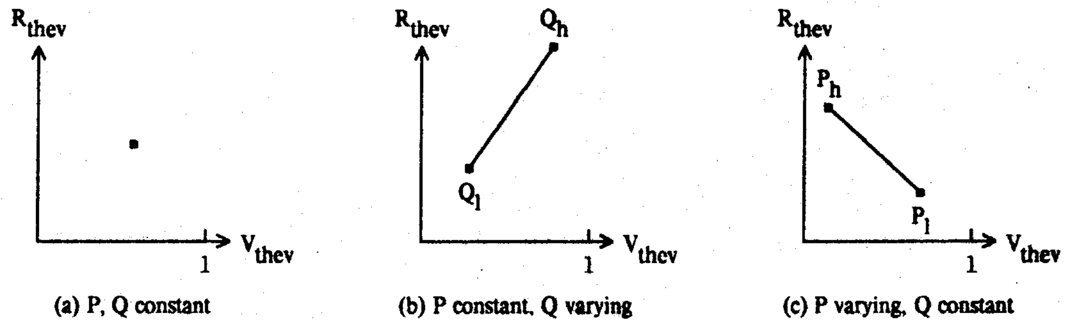


Figure A3.3. *Thevenin plots as P and Q are varied one at a time*

Equation A3.2 tells us that if, say, Q is held constant and P is varied, the plot is a straight line of slope

Q, which, if extended, would intersect the R_{thev} axis at $V_{thev} = 1$ (see figure A3.3(c)). When both *P* and *Q* are varied (see figure A3.4), the plot produces a diamond-shaped quadrilateral, and not a rectangle.

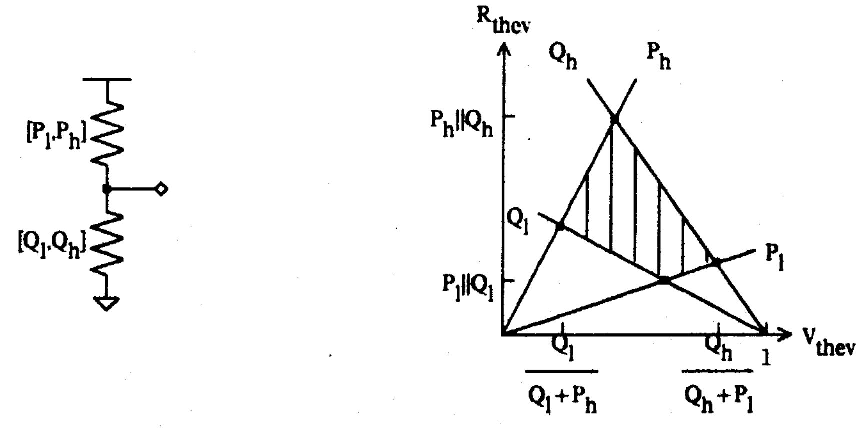


Figure A3.4. *Thevenin plot as P and Q are varied simultaneously*

Although the limits of R_{thev} and V_{thev} are the ones shown in equation A3.3, certain combinations of Thevenin parameters permitted by the equation are clearly ruled out by the diagram above.

If a series resistance *R* is now added, the resulting Thevenin plot is shown in the following figure.

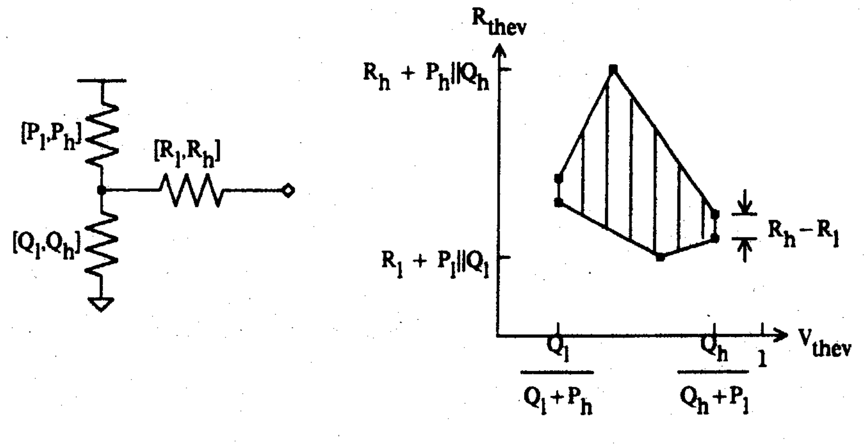


Figure A3.5. *Thevenin plot when series resistance is added*

The result is not a plot of a resistor divider at all. In order to approximate the circuit by a divider, a decision is needed concerning which information to preserve with the approximation.

Since the approximation under development is used to calculate V_{thev}, it is important to preserve information about the maximum and minimum of the circuit's voltage. This constraint fixes the right and left vertices of the diamond. The top and bottom vertices are constrained by the choice of resistance information to preserve; since it is better to overestimate than to underestimate resistances, the minimum value of R_{thev} is preserved. The resulting divider is shown graphically in the following figure. The voltage constraints are shown as dashed vertical lines; the resistance constraint as the circled vertex.

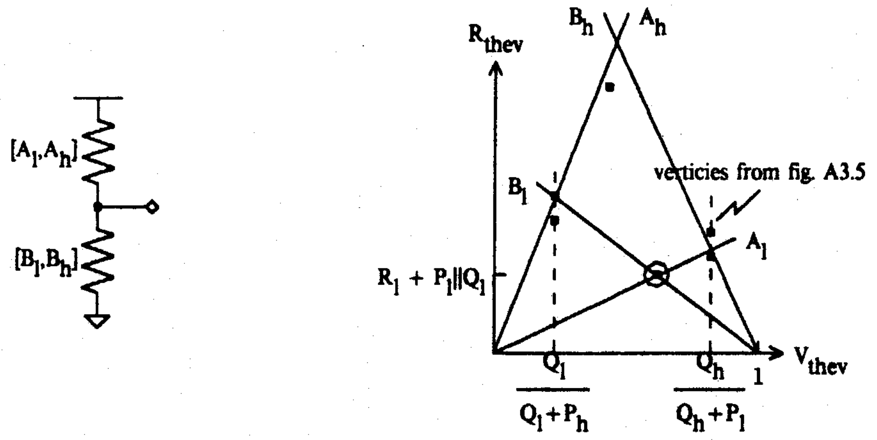


Figure A3.6. *Thevenin plot showing approximating divider*

The values for A_l and B_l are determined by the second constraint and equation A3.2:

$$R_l + (Q_l \,||\, P_l) = A_l \frac{Q_l}{P_l + Q_l} \quad \text{and} \quad R_l + (Q_l \,||\, P_l) = B_l(1 - \frac{Q_l}{P_l + Q_l}) \qquad \text{(A3.4)}$$

This fixes the two lines that form the bottom half of the diamond. Next, the values of A_h and B_h are chosen so that the left and right vertices of the diamond have the same V_{thev} coordinates as in figure A3.5:

$$\frac{B_l}{A_h + B_l} = \frac{Q_l}{P_h + Q_l} \quad \text{and} \quad \frac{B_h}{A_l + B_h} = \frac{Q_h}{P_l + Q_h} \qquad \text{(A3.5)}$$

Solving equations A3.4 and A3.5 for the parameters of the approximating divider yields:

$$\begin{aligned} A_l &= P_l + R_l + R_l\frac{P_l}{Q_l} & A_h &= P_h + R_l\frac{P_h}{P_l} + R_l\frac{P_h}{Q_l} \\ B_l &= Q_l + R_l + R_l\frac{Q_l}{P_l} & B_h &= Q_h + R_l\frac{Q_h}{Q_l} + R_l\frac{Q_h}{P_l} \end{aligned} \qquad \text{(A3.6)}$$

Note that all resistances are greater than the minimum resistance of the series resistor (R_l). A different choice of what resistance information to preserve (as was made in early versions of RSIM), might cause A_l and B_l to be less than R_l, leading to pessimistic voltage predictions for some nMOS circuits.

REFERENCES

[Agrawal80]	V. Agrawal, *et al*, "A Mixed-mode Simulator," *Proceedings of 17th Design Automation Conference*, June 1980.
[Baker80]	C. Baker, *Artwork Analysis Tools for VLSI Circuits*, M.I.T. Laboratory for Computer Science TR-239, May 1980.
[Barzilai83]	Z. Barzilai, *et al*, "Simulating Pass Transistor Circuits using Logic Simulation Machines," *Proceedings of 20th Design Automation Conference*, June 1983.
[Bell81]	A. Bell, M. Stefik, and L. Conway, *The Deliberate Engineering of Methodologies for Integrated System Design*, Knowledge-Based VLSI Design Group, Xerox PARC, Memo KB-VLSI-81-3 (working paper), April 1981.
[Bryant79]	R. Bryant, PhD thesis proposal, M.I.T. Department of Electrical Engineering and Computer Science, December 1979.
[Bryant81]	R. Bryant, *Logic Simulation of MOS LSI*, M.I.T. Laboratory for Computer Science TR-259, 1981.
[Case78]	G. Case, "SALOGS-IV — A Program to Perform Logic Simulation and Fault Diagnosis," *Proceedings of 15th Design Automation Conference*, June 1978.
[Chawla75]	B. Chawla, H. Gummel, and P. Kozak, "MOTIS — An MOS Timing Simulator", *IEEE Transactions on Circuits & Systems*, Vol. CAS-22, No. 13, December 1975.
[Chen78]	R. Chen and J. Coffman, "Multi-Sim, A Dynamic Multi-Level Simulator," *Proceedings of 15th Design Automation Conference*, June 1978.
[Curry74]	H. Curry and R. Feys, *Combinatory Logic*, North-Holland Publishing Company, Amsterdam, 1974.
[Denneau82]	M. Denneau, "The Yorktown Simulation Engine," *Proceedings of 19th Design Automation Conference*, June 1982.
[Fan77]	S. Fan, M. Y. Hseuh, A. Newton, and D. Pederson, "MOTIS-C: A New Circuit Simulator for MOS LSI Circuits," *Proceedings IEEE International Symposium on Circuits and Systems*, April 1977.
[Flake80]	P. Flake, P. Moorby, and G. Musgrave, "Logic Simulation of Bi-directional Tri-state Gates," *Proceedings of IEEE International Conference on Circuits and Computers*, October 1980.
[Flake83]	P. Flake, P. Moorby, and G. Musgrave, "An Algebra for Logic Strength Manipulation," *Proceedings of 20th Design Automation Conference*, June 1983.
[Gardner79]	R. Gardner and P. Weil, "Hierarchical Modeling and Simulation in VISTA," *Proceedings of 16th Design Automation Conference*, June 1979.

[Glasser80] L. Glasser, *The Analog Behavior of Digital Integrated Circuits*, M.I.T. VLSI Memo No. 80-36, December 1980.

[Harrison77] W. Harrison, "A New Strategy for Code Generation — the General Purpose Optimizing Compiler," *Proceedings of Fourth ACM Symposium on the Principles of Programming Languages*, 1977.

[Hill79] D. Hill and W. vanCleemput, "SABLE: A Tool for Generating Structural, Multi-level Simulations," *Proceedings of 16th Design Automation Conference*, June 1979.

[Hillis81] W. D. Hillis, *The Connection Machine*, M.I.T. Artificial Intelligence Laboratory Memo No. 646, September 1981.

[Holt81] D. Holt and D. Hutchings, "A MOS/LSI Oriented Logic Simulator," *Proceedings of 18th Design Automation Conference*, June 1981.

[Horowitz83] M. Horowitz, "Timing Models for MOS Pass Networks," *Proceedings of the IEEE International Symposium on Circuits and Systems*, 1983.

[Iler83] J. Iler, *A VLSI Circuit Recognizer for Enhancing Simulator Accuracy*, MS Thesis, M.I.T. Department of Electrical Engineering and Computer Science, January 1983.

[Jouppi83] N. Jouppi, "TV: An nMOS Timing Analyzer," *Proceedings of the Third Caltech VLSI Conference*, 1983.

[Koppel78] A. Koppel, S. Shah, and P. Puri, "A High Performance Delay Calculation Software System for MOSFET Digital Logic Chips," *Proceedings of 15th Design Automation Conference*, June 1978.

[Lelarasmee81] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli, *The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrate Circuits*, Memorandum No. UCB/ERL M81/75, Electronics Research Laboratory, University of California, Berkeley, June 1981.

[McDermott82] R. McDermott, "Transmission Gate Modeling in an Existing Three-value Simulator," *Proceedings of 19th Design Automation Conference*, June 1982.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Massachusetts, 1980.

[Nagel75] L. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, ERL Memo No. ERL-M520, University of California, Berkeley, May 1975.

[Nahm80] H. Nham and A. Bose, "A Multiple Delay Simulator for MOS LSI Circuits", *Proceedings of 17th Design Automation Conference*, June 1980.

[Newton80] A. Newton, *Timing, Logic and Mixed-mode Simulation for Large MOS Integrated Circuits*, NATO Advanced Study Institute on Computer Design Aids for VLSI Circuits, Sogesta-Urbino, Italy, July/August 1980.

[Newton83] A. Newton and A. Sangiovanni-Vincentelli, *Relaxation-based Electrical Simulation*, University of California, Berkeley, 1983.

[Okasaki83] K. Okasaki, T. Moriya, and T. Yahara, "A Multiple Media Delay Simulator for MOS LSI Circuits," *Proceedings of 20th Design Automation Conference*, June 1983.

[Ousterhout83] J. Ousterhout, "Crystal: A Timing Analyzer for nMOS VLSI Circuits," *Proceedings of the Third Caltech VLSI Conference*, 1983.

[Penfield81] P. Penfield and J. Rubinstein, *Signal Delay in RC Tree Networks*, M.I.T. VLSI Memo No. 81-40, January 1981.

[Pfister82] G. Pfister, "The Yorktown Simulation Engine: Introduction," *Proceedings of 19th Design Automation Conference*, June 1982.

[Pilling73] D. Pilling and H. Sun, "Computer-Aided Prediction of Delays in LSI Logic Systems," *Proceedings of 10th Design Automation Workshop*, June 1973.

[SV80] A. Sangiovanni-Vincentelli, *Circuit Simulation*, NATO Advanced Study Institute on Computer Design Aids for VLSI Circuits, Sogesta-Urbino, Italy, July/August 1980.

[Sherwood81] W. Sherwood, "A MOS Modelling Technique for 4-State True-Value Hierarchical Logic Simulation," *Proceedings of 18th Design Automation Conference*, June 1981.

[Stevens83] P. Stevens and G. Arnout, "BIMOS, an MOS oriented multi-level logic simulator," *Proceedings of 20th Design Automation Conference*, June 1983.

[Szygenda72] S. Szygenda, "TEGAS2 — Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," *Proceedings of 9th ACM Design Automation Workshop*, June 1972.

[Szygenda75] S. Szygenda and E. Thompson, "Digital Logic Simulation in a Time-Based, Table-Driven Environment," *IEEE Computer*, Vol. 8, March 1975.

[Tamura83] E. Tamura, K. Ogawa, and T. Nakano, "Path Delay Analysis for Hierarchical Building Block Layout System," *Proceedings of 20th Design Automation Conference*, June 1983.

[Terman82] C. Terman, *User's Guide to NET, PRESIM, and RNL*, M.I.T. Laboratory for Computer Science, September 1982.

[Thompson74] E. Thompson, *et al*, "Timing Analysis for Digital Fault Simulation Using Assignable Delays," *Proceedings of 11th Design Automation Conference*, June 1974.

[Ulrich73] E. Ulrich and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," *Proceedings of 10th Design Automation Workshop*, June 1973.

[Ulrich76] E. Ulrich, "Non-integral Event Timing for Digital Logic Simulation," *Proceedings of 13th Design Automation Conference*, June 1976.

[Vaucher75] J. Vaucher and P. Duval, "A Comparison of Simulation Event List Algorithms," *Communications of the ACM*, April 1975.

[Weeks73] W. Weeks, *et al*, "Algorithms for ASTAP — A Network Analysis Program," *IEEE Transaction on Circuit Theory*, Vol. CT-20, November 1973.

[Wulf75] W. Wulf, *et al*, *The Design of an Optimizing Compiler*, American Elsevier, New York, 1975.

[Wyatt83] J. Wyatt, *et al*, "Waveform Bounding for VLSI Timing," *Proceedings IEEE International Conference on Computer Design*, October 1983.

[Zycad83] *LE-1000 Series Logic Evaluator Intermediate Form Specification,*, Release 1.0, Zycad Corporation, Roseville, MN, 1983.