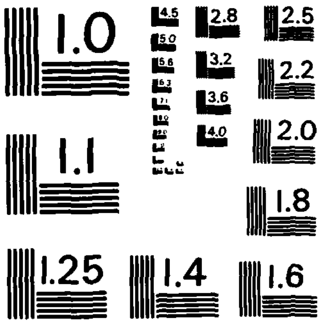
AD-A161 371 SWITCH-LEVEL TIMING SIMULATION OF MOS VLSI (METAL-OXIDE-SEMICONDUCTOR VER..(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB V B RAO JAN 85 R-1032 1/3

UNCLASSIFIED N00014-84-C-0149 F/G 9/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

REPORT R-1032) MARCH 1985 UILU-ENG 85-2207 (12)

COORDINATED SCIENCE LABORATORY

SWITCH-LEVEL TIMING SIMULATION OF MOS VLSI CIRCUITS

VASANT BANGALORE RAO

APPROVED FOR PUBLIC RELEASE. DISTRIBUTION UNLIMITED.

ERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release, distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) R-1032 UILU-ENG 85-2207	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A

6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Laboratory University of Illinois	6b. OFFICE SYMBOL *(If applicable)* N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS *(City, State and ZIP Code)* 1101 W. Springfield Avenue Urbana, Illinois 61801		7b. ADDRESS *(City, State and ZIP Code)* 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Joint Services Electronics Program and IBM	8b. OFFICE SYMBOL *(If applicable)* N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-84-C-0149 and IBM Tech

8c. ADDRESS *(City, State and ZIP Code)*
JSEP - 800 N. Quincy St., Arlington, VA 22217
IBM - General Technology Div., Burlington, VT 05401

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
N/A	N/A	N/A	N/A

11. TITLE *(Include Security Classification)* SWITCH-LEVEL TIMING SIMULATION OF MOS VLSI CIRCUITS

12. PERSONAL AUTHOR(S) Rao, Vasant Bangalore

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Yr., Mo., Day)*	15. PAGE COUNT
Interim Technical, final	FROM Aug. '80 TO Dec. '84	January 1985	245

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES		
FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Switch-level simulation, timing simulation, NMOS circuits, delay operation, graph algorithm

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report deals with the development of a fast and accurate simulation tool for very-large-scale integrated (VLSI) circuits consisting of metal-oxide-semiconductor (MOS) transistors. Such tools are called switch-level timing simulators and they provide adequate information on the performance of the circuits with a reasonable expenditure of computation time even for very large circuits. The algorithms presented in this thesis can handle only n-channel MOS(NMOS) circuits, but are easily extendible to handle complementary MOS(CMOS) circuits as well.

The algorithms presented in this report have been implemented in a computer program called MOSTIM. In all the circuits simulated thus far, MOSTIM provides timing information with an accuracy of within 10% of that provided by SPICE2, at approximately two orders of magnitude faster in simulation speed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASS[illegible]D/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER *(Include Area Code)*	22c. OFFICE SYMBOL None

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

SWITCH-LEVEL TIMING SIMULATION OF MOS VLSI CIRCUITS

BY

VASANT BANGALORE RAO

B.Tech., Indian Institute of Technology, 1980
M.S., University of Illinois, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

SWITCH LEVEL TIMING SIMULATION OF MOS VLSI CIRCUITS

Vasant Bangalore Rao, Ph.D.
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1985

This dissertation deals with the development of a fast and accurate simulation tool for very-large-scale integrated (VLSI) circuits consisting of metal-oxide-semiconductor (MOS) transistors. Such tools are called switch-level timing simulators and they provide adequate information on the performance of the circuits with a reasonable expenditure of computation time even for very large circuits. The algorithms presented in this thesis can handle only n-channel MOS (NMOS) circuits, but are easily extendible to handle complementary MOS (CMOS) circuits as well.

An NMOS circuit is modeled as a set of nodes connected by transistor switches. Three strengths and three states are used to represent the signals at the nodes in the circuit. The strengths in decreasing order are **input**, **pullup**, and **normal**. The three states used are **0**, **u**, and **1**, with **0** and **1** representing the conventional *low* and *high* signal values respectively while the **u** state is used to represent *intermediate* signal values and sometimes to represent situations of *conflict*. Each switch is either *open*, *closed*, or in an *intermediate* state.

The enhancement transistors in the NMOS network are first partitioned into driver and pass transistors. The NMOS network itself is then partitioned into multifunctional blocks (MFB), pass transistor blocks (PTB), and input sources (SRC). The partitioning is an automatic process that is completely transparent to the user and can be performed in linear time. The partitioned blocks are then ordered for processing so that, whenever possible, a block is scheduled for processing only after all its inputs have been previously processed. Since this is not possible for blocks forming feedback loops, a novel dynamic windowing scheme is used to schedule such blocks.

The blocks in the partitioned network are then simulated at the switch level using graph algorithms, producing so-called *zero-delay* ternary signal waveforms. The zero-delay signal transitions are then delayed by using delay and filtering operators. The characteristics of the delay operator are computed in a presimulation phase by simulating five different circuit primitives using an accurate circuit simulator such as SPICE2. These characteristics are stored in a table. During the simulation a circuit block is mapped onto one of the five primitives and appropriate delay values are obtained by fast table lookup techniques. Several factors such as block configuration, loading, device geometries, and input slew rates are taken into account while computing the delay values.

The algorithms presented in this thesis have been implemented in a computer program called MOSTIM. In all the circuits simulated thus far, MOSTIM provides timing information with an accuracy of within 10% of that provided by SPICE2, at approximately two orders of magnitude faster in simulation speed.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation and gratitude to Professor Timothy N. Trick, my dissertation advisor, for his invaluable guidance and continuing encouragement during the course of my graduate studies. I would also like to thank Professors Ibrahim Hajj and Vijaya Ramachandran for being members of my dissertation committee and for their support. I wish to thank all the members of the Circuits and Systems Group at the Coordinated Science Laboratory, Urbana, for many interesting discussions and helpful suggestions. I am very grateful to Mita Desai for her continued support and understanding, and also for helping me with the manuscript. I would also like to thank Beth Piver, Eric Peterson, and Rosemary Wegeng for their help in preparing the manuscript.

Finally, I wish to thank my parents, Indira and Sathyanarayana, for their everlasting love, *encouragement, patience, and support*. They have always been a great source of inspiration to me. This thesis is dedicated to both of them.

This research was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under contract number N00014-79-C-0424, and in part by the IBM Corporation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The design of an electronic circuit, traditionally, started with the designer who, with a mental picture, translated his or her ideas into the form of a circuit schematic. This step relied heavily on the human designer's intuition, past experience, and knowledge to make reasonable approximations. This was followed by the "breadboarding" phase in which an actual prototype of the circuit was constructed from discrete components interconnected by external wires and was tested. The performance of the circuit, if not found satisfactory, was then improved by adjusting the circuit element values in a somewhat trial-and-error fashion.

The advent of integrated circuits, however, has greatly changed the picture. There are several steps involved in the design of a *very large-scale integrated* (VLSI) circuit, which may consist of several hundreds of thousands of components, mainly transistors. The circuit designer first obtains a very high-level functional description of the circuit based on specifications provided by the user. The synthesis, often called the top-down process, translates this high-level description into various levels including the register level, the transistor level (or electrical level) etc. and terminates at the physical mask-level, i.e. , the actual layout of the patterns of metal, semiconductor, and insulating material by which the components and the interconnections are achieved. This is followed by the design verification, or the bottom-up process, wherein a software tool called an extractor is first used to obtain a circuit level (or transistor level) description from the physical layout. The breadboarding phase is replaced by using a simulation tool to predict the performance of the circuit which is then compared with the user's specifications, thus completing the so-called design loop. If the performance is not satisfactory, certain changes are made and the whole process is then repeated. The total time spent in the design loop is usually referred to as the *turn-around time*.

The main objective of the VLSI circuit designer is to obtain designs with as low a turn-around time as possible. Computer-aided design tools have become virtually indispensable at various steps in the design process to perform tasks which would otherwise take a very long time if they were done by human beings. Using silicon compilers can speed up the top-down synthesis process considerably since they produce the mask level description, straight away, from the functional description without any human intervention. Certain software tools known as *design rule checkers* (DRC) and *electrical rule checkers* (ERC) are also used. These perform the rather mundane tasks of checking to see if the layout satisfies all the design rules of the technology and whether there are any topological faults from the electrical point of view such as a floating node, and a short between power and ground. There is, however, a bottleneck in speeding up the bottom-up design verification process which is in the simulation of the electrical behavior of the circuit. This bottleneck is due to the unavailability of a simulation tool that is capable of accurately predicting the performance of an entire VLSI circuit at a reasonable cost. The accuracy of a simulator is important, since otherwise the integrated circuit which is fabricated and tested might turn out to perform rather unsatisfactorily. For large circuits (typically of the kind in today's VLSI technology), the speed of simulation is equally important so that the entire circuit can be simulated in a reasonably small amount of computation time. However, as we shall see in Chapter 2 of this thesis, speed and accuracy of a simulator are often conflicting requirements among existing simulation tools.

In this dissertation we will be primarily concerned with providing a fast and accurate simulation tool to a VLSI circuit designer which gives adequate information on the performance of the circuit with a reasonable expenditure of computation time even for very large circuits. In Chapter 2 of this thesis we will review some of the existing simulators for integrated circuits and classify them into two distinct categories, namely, *analog simulators* and *digital simulators*. Analog simulators treat an electronic circuit as a continuous dynamical system with electrical signals such as voltages and currents

Digital simulators, on the other hand, view the circuit as a digital network with signals occupying discrete states such as low (0) and high (1). For small circuit blocks where analog voltage levels are critical in evaluating circuit performance, or where strong coupling exists, analog circuit simulators such as SPICE2 [1] and ASTAP [2] can be used to predict the performance of the circuit very accurately. As the size of the circuit (number of components) increases, however, using these simulators is no longer cost-effective. Several decomposition techniques have been used to speed up their performance and have resulted in a new generation of analog simulators [3-15] which are, however, cost-effective for circuits limited to at most ten thousand devices.

The existing digital simulators [13-27] can be further divided into Boolean gate-level [13-18] and switch-level [19-27] simulators. In the Boolean gate model a circuit consists of a set of logic gates connected by unidirectional memoryless wires. The logic gates compute Boolean functions of their input signals and transmit these values along the wires to the inputs of other gates. Each gate input has a unique signal source. Information is only stored in the feedback paths of sequential circuits. The Boolean gate model, however, cannot describe some of the newer technologies currently used in VLSI circuit design, especially circuits with Metal-Oxide-Semiconductor (MOS) transistors. The MOS transistor can be treated as a voltage-controlled switch with three terminals : drain, gate, and source. The signal at the gate terminal controls the connection between drain and source terminals. Therefore, some MOS pass transistor networks can implement combinational logic in ways that resemble relay contact networks more closely than conventional logic gate networks. Dynamic memories using MOS devices can store information without feedback paths by exploiting the capacitance of the wires (interconnect region) and the gates of the transistors attached to them. A variety of bus structures can provide multidirectional, multipoint communication. Thus, MOS circuits consist of bidirectional switching elements connected by bidirectional wires with memory due to the interconnect and device capacitances and hence cannot be modeled accurately by Boolean gate-level simulators.

A new class of digital simulators has recently emerged specifically for simulating MOS VLSI circuits. These switch-level [19-27] simulators model an MOS circuit as a set of nodes connected by transistor switches. Each node occupies a discrete number of states **0**, **1**, or **X** for the intermediate or unknown state and each switch is either *open*, *closed*, or in an *intermediate* or *unknown* state. These simulators can handle a variety of MOS configurations such as logic gates, pass transistors, busses, static and dynamic memory. Digital simulators, in general, operate at sufficient speeds to test entire VLSI systems, since the circuit behavior is modeled at a logical rather than a detailed electrical level. However, these simulators do not model the dynamics of the circuits properly and are often useful only in predicting steady-state responses of the signals. Analog simulators, on the other hand, predict both steady-state and transient responses fairly accurately, if the device models used are accurate, but are cost-effective only for circuits with less than a few thousand components, which are considered small in the present day VLSI technology.

The algorithms presented in this thesis have led to the development of a switch-level timing simulator for MOS VLSI circuits. This simulator, MOSTIM, is an attempt to bridge the gap between analog and digital simulators. It performs simulations at a switch level and hence runs at speeds close to that of digital simulators. Further, it uses a delay operator to delay signal transitions accurately and hence provides the timing accuracy comparable to that of analog simulators.

MOSTIM uses 3 strengths and 3 states to represent node signal values. The strengths in decreasing order are **input**, **pullup**, and **normal**. The three states used are **0**, **u**, and **1**, with **0** and **1** representing the conventional *low* and *high* signal values respectively while the **u** state is used to represent *intermediate* signal values and sometimes to represent situations of *conflict*. The input to MOSTIM is a transistor-level circuit description in a SPICE2 input format. The program begins by partitioning the entire MOS network into several functional blocks. The partitioning is an automatic process that is completely transparent to the user. The partitioned blocks are then ordered for processing so that, whenever possible, a block is scheduled for processing only after all its inputs have been previously processed. Since this is not possible for blocks forming feedback loops, a novel dynamic windowing scheme is used to schedule such blocks. The blocks are then processed at a switch level producing so-called *zero-delay* ternary signal waveforms. These zero-delay waveforms are first ***delayed*** suitably by the delay operator and then *filtered* to produce realistic waveforms. MOSTIM, at present, handles only n-channel MOS (NMOS) circuits, but the algorithms presented in this dissertation can be easily extended to complementary MOS (CMOS) circuits as well.

In Chapter 3, the algorithms for partitioning the input network into various blocks and the ordering of these blocks for processing are discussed. The input network to MOSTIM is assumed to consist of voltage sources, NMOS transistors - both *depletion* and *enhancement* types- and a fixed capacitance from each circuit node to ground. The key to the partitioning strategy is to divide the set of enhancement transistors into *driver transistors* and *pass transistors*. A graph-theoretic algorithm achieves this in computation time linear with the number of enhancement devices. The driver transistors are then grouped together to form *multifunctional blocks* (MFB) and the pass transistors are grouped together to form *pass transistor blocks* (PTB). A third type of block called *input source* (SRC) is created from the voltage sources, clocks etc. A directed graph **G** is then constructed with vertices corresponding to the various circuit blocks, namely, MFB's, PTB's, and SRC's, and directed arcs describing the interconnections between them. A modified version of a depth first search known as Tarjan's algorithm [31] is used to detect *strongly connected components* (SCC) in **G**. The vertices within an SCC correspond to blocks forming *feedback loops* in the original circuit and are collapsed into single vertices thus creating an acyclic reduced graph $\tilde{\mathbf{G}}$. The vertices of $\tilde{\mathbf{G}}$ are then placed in topological order for processing.

The algorithms for the switch-level simulation of multifunctional blocks and pass transistor blocks are presented in Chapter 4. An MFB is a single output, multiple input, unidirectional block whose steady-state output is a Boolean function of its inputs. A graphical technique using *internal-node eliminations* is used to evaluate the state of the signal at the output, given the input signal states. No attempt is made to evaluate signals at the internal nodes of the MFB. In the switch level simulation of a PTB, however, the signal at every node within the PTB is evaluated. The transistors in a PTB are modeled as bidirectional switches whose conduction states (i.e., open, closed, or intermediate) are controlled by the signal at the corresponding gate terminals. A strong node forces its state on a weaker node connected to it via a path of conducting transistors at any given time instant. The algorithm is quite similar to the one used in conventional switch level simulators such as MOSSIM [19], except for the interpretation of the **u** state (or **X** state as used in MOSSIM).

The switch-level simulation algorithms described in Chapter 4 generate zero-delay ternary waveforms for each pull-up node in an MFB and each normal node in a PTB. A delay operator, described in Chapter 5, is used to delay pairs of complete transitions (i.e., **0**→**u** followed by **u**→**1**, or **1**→**u** followed by **u**→**0**) in the zero-delay waveforms. The delay operator computes appropriate delay values by taking several parameters into account, such as block configuration, loading, device geometries, and input slew rates. For NMOS technology, knowing the delay characteristics of five different circuit primitives is sufficient, within reasonable limits of accuracy, to compute delays through any general MFB or PTB. These five primitives are simulated using an accurate circuit simulator such as SPICE2 [1], or SLATE [3], for various device and circuit parameters, and the delay values are extracted and stored in a delay table. This can be done in a presimulation phase. During simulation, MOSTIM then maps an MFB or a PTB into one of the five primitives and obtains the appropriate delay value through fast table lookup methods, and interpolation when necessary. Clearly, the delay values are functions of various circuit and device parameters. However, using *time scaling* techniques, it will be shown that, only one parameter, namely, the input slew rate, is sufficient for determining delays in three of the five primitives. The effect of the rest of the parameters can be accounted for by using certain *scale factors*. For the remaining two primitives, however, there are three parameters necessary to obtain delay values. Thus, time scaling helps reduce the size of the delay tables considerably.

In Chapter 6 we discuss techniques used to process blocks within an SCC. In order to perform a switch level simulation of a block (MFB or PTB), the waveforms at the input nodes to the blocks must necessarily be known. Since this is not possible for blocks within a SCC, these have to be handled separately. A *waveform relaxation* technique could be used, wherein the blocks are processed iteratively in a predetermined order with unknown input waveforms initially relaxed and output waveforms constantly updated. Several drawbacks of this technique will be discussed. A new dynamic windowing method which overcomes most of these drawbacks will be presented. In principle, this new scheme is quite similar to the classical *event-driven time-wheel* approach used in conventional logic simulators [13,19], except that events take place during *intervals of time* instead of occurring *instantaneously*. The entire time interval of analysis is automatically partitioned into variable size *windows* such that the signal at each node in each block within the SCC occupies a steady state (i.e., 0 or 1) at the window boundaries. Associated with each window is a set of blocks scheduled for processing during that window. This new scheme does not require an *a priori* ordering of blocks within the SCC, and is also seen to take less computation time and less storage.

A number of NMOS circuits have been simulated using MOSTIM. The performance is discussed in Chapter 7. In all the circuits simulated thus far, MOSTIM provides timing information with an accuracy of within 10% of that provided by SPICE2 [1], at approximately two orders of magnitude faster in simulation speed. The performance is also compared with some of the recent attempts made in switch level timing simulation such as RSIM [26]. Finally, in Chapter 8, we provide some conclusions along with some suggestions for future research.

CHAPTER 2

OVERVIEW OF SIMULATION TECHNIQUES

Simulation plays a major role in the process of designing an integrated electronic circuit. By using a simulator, the circuit designer can evaluate the performance of the design before going into the expensive and time-consuming manufacturing process. There are two basic approaches to simulating an integrated electronic circuit. The first, and more traditional approach is to treat the circuit as a continuous dynamical system and obtain a set of nonlinear algebraic-differential equations with electrical variables such as voltage, current, and charge to describe its behavior. The objective of an *analog simulator* is to solve this set of equations, numerically, and obtain the detailed waveforms at various nodes in the circuit. An alternate approach is to view the circuit as a digital system in which the signals occupy discrete states. Since the majority of VLSI circuits are primarily digital in nature, *digital simulators* are often successful in predicting steady-state responses in these circuits. Analog simulators are generally quite accurate in evaluating the performance of circuits, but are not fast enough to handle entire VLSI circuits. Digital simulators, on the other hand, are able to simulate very large circuits, but, unfortunately, are not accurate in modeling the dynamics in these circuits.

2.1 Analog Simulation

For small circuit blocks where analog voltage levels are critical to determine circuit performance, or where strong coupling exists, circuit simulators such as SPICE2 [1] and ASTAP [2] can be used to provide accurate information on the behavior of the circuit. These simulators will be referred to as *standard circuit simulators*. These are general purpose simulators in that they can handle almost any type of circuit element such as resistors, capacitors, inductors (both self and mutual), voltage and current sources (independent and controlled), nonlinear devices (transistors, diodes, etc.), and transmission lines. They can also perform many types of analyses such as dc analysis, ac (or small-signal) analysis, noise analysis, and transient or time-domain analysis. In present day IC design, however, standard circuit simulators are primarily used for time-domain transient analysis, which happens to be the most complicated and expensive type of analysis.

The transient analysis of a circuit involves the solution of a system of nonlinear algebraic-differential equations describing the analog behavior of the circuit. Standard circuit simulation involves, essentially, three basic numerical methods in solving the circuit equations:

1. An implicit integration method which approximates the time-derivative operator in the system of differential equations with a divided difference operator. The circuit equations are thus transformed into a sequence of nonlinear algebraic difference equations.
2. The Newton-Raphson algorithm for solving the sequence of nonlinear equations, iteratively, by generating a set of linear algebraic equations.
3. The Gaussian elimination method for finding the solution of a system of linear algebraic equations.

The circuit simulator SPICE2 uses the Modified Nodal Method (MNA) [32] to formulate the circuit equations, whereas ASTAP uses the Sparse Tableau [33] approach. In either case, the time $\mathbf{T_f}$ spent by the simulator to formulate the circuit equations grows almost linearly with the size of the circuit. However, the time $\mathbf{T_s}$ required to solve these equations increases at a faster rate and rapidly becomes the dominant cost of analysis. Moreover, most of $\mathbf{T_s}$ is spent in the Gaussian elimination process which involves the solution of a matrix equation of the form $\mathbf{Ax=b}$, where $\mathbf{A}$ is the circuit Jacobian matrix, $\mathbf{x}$ is a vector of unknown circuit variables and $\mathbf{b}$ is a known source vector. In a typical large scale circuit, the matrix $\mathbf{A}$ is usually very sparse (i.e., it has very few nonzero elements). Hence, the Gaussian elimination in standard circuit simulators is usually implemented by using sparse matrix methods [34]. It is important to exploit the sparsity of the matrix $\mathbf{A}$, since the computational time required to perform

Gaussian elimination of a full $n \times n$ matrix, using Crout's algorithm [34], is proportional to n^3 (theoretically, better algorithms exist with smaller exponents [67]). In digital circuits, however, using sparse matrix techniques [1], the Gaussian elimination has been empirically shown to take computational time that is, on an average, proportional to n^α, where $\alpha \in [1.2, 1.5]$.

SPICE2 and ASTAP have proven to be reliable and effective when the size of the circuit, measured by the number of components, is small. As the size of the circuit increases, the computer time and storage space used up by these simulators increase rapidly despite the use of sparse matrix techniques. In particular, the time T_s required to solve the circuit equations exhibits a nonlinear increase with circuit size. In SPICE2, T_s is less than 10% of the total computation time for a circuit with less than 30 nodes but reaches almost half the total time for a circuit with a thousand nodes [13]. The problem is further aggravated by the fact that for larger circuits, more information is generally needed to verify the circuit performance, and hence, longer simulation times are required. It has been estimated that the simulation of a circuit with around 10,000 MOS transistors from $t=0$ to $t=1000$ns, using SPICE2 on an IBM 370/168 Computer, would take at least 30 hours of CPU-time [55]. Since 30 hours is clearly prohibitive, the cost-effective use of standard circuit simulators is limited to circuits, with less than a few hundred components, which are considered small in the present day VLSI technology.

2.2 Decomposition Techniques for Analog Simulation

Several attempts have been made to speed up the performance of standard circuit simulators. This resulted in the development of a variety of analog simulators such as SLATE [3], MACRO [4], MOTIS [5], MOTIS-C [6], PREMOS [7], RELAX [10], SPLICE [13], DIANA [14], and SAMSON [15]. These nonstandard analog simulators can be meaningfully classified according to the *decomposition* techniques employed by them, in order to achieve the improvement in speed. Decomposition refers to any technique that subdivides the original problem into several subproblems. Each subproblem corresponds to solving only a subset of the original system equations for a subset of system variables.

Decomposition can be applied at any of the three levels of the standard circuit simulation approach, namely, the differential equation level (or sometimes called the time level), the nonlinear algebraic equation level, or the linear algebraic equation level. The original system of equations is viewed by a decomposition technique, no matter at what level it is applied, as a composition of several subsystems with interactions among them. Each subsystem is usually solved in a manner similar to the conventional techniques used in standard circuit simulators. Hence, the main feature of a decomposition technique is the handling of the interactions between the various subsystems.

The majority of large integrated circuits are digital in nature, and hence, several properties of such circuits can be exploited during the simulation process. Digital circuits tend to be structurally regular and repetitive. A typical large digital circuit is usually composed of a number of small subcircuits, normally referred to as logic gates. Several of these logic gates are functionally and topologically the same, and thus analyzing one is very similar to analyzing the others. Furthermore, only a small fraction of the circuit variables is actively changing state at any time instant in a large digital circuit. For circuits containing over 1000 transistors, typically more than 80% of the circuit variables are steady (not changing) at any given time instant. As the size of the circuit increases, the fraction of active (changing) circuit variables tends to fall even further. This inactivity, or *latency*, in a large digital network can be exploited by an analog simulator in a number of ways. The main advantages in using decomposition techniques are

1. The structural regularity and repetitivity of the subsystems can be exploited.
2. Incorporating bypassing schemes at several levels to exploit the latency of a subsystem can result in additional savings in computing time.
3. Decomposition techniques are suitable for computers with parallel or pipeline architectures since two or more subsystems can be solved concurrently.

There are two different approaches to achieving system decomposition, namely, *tearing* and *relaxation* [36]. These two approaches are characterized by different ways of updating the interactions between subsystems and by different numerical properties. The tearing approach aims to retain the same numerical convergence and stability properties as of the standard circuit simulation approach, while the relaxation methods (also called *temporal* or *indirect* methods) have completely different numerical properties.

2.2.1 Tearing Decomposition

Solving a network by tearing decomposition is an approach in which a part of the network is torn away, so that the remaining subnetworks are disconnected and thus can be analyzed independently. The solutions of the individual subnetworks are then combined with those of the torn-away part of the network in order to obtain the solution of the entire network. There are basically two types of tearing, namely, *node-tearing* and *branch-tearing* depending upon whether circuit nodes or branches are removed to tear down the network. The program SLATE [3] utilizes the node-tearing approach at the linear equation level. The LU-factorization of the original Jacobian matrix during the standard Gaussian elimination process is performed by cleverly exploiting the block structure of the matrix reordered in a special form, thus achieving savings in computation time. Another approach is to decompose the system at the nonlinear equation level by introducing additional iteration loops in the standard Newton's method. This multilevel Newton method is used in MACRO [4]. Tearing methods, in general, are well-suited for parallel processing and retain the numerical convergence and stability properties of the standard approach.

2.2.1.1 Tearing of Linear Systems

At the linear equation level, tearing is used to solve a set of linear algebraic equations of the form

$$\mathbf{A}\mathbf{x}=\mathbf{b} \tag{2.1}$$

where $\mathbf{A}$ is an $\mathbf{n}\times\mathbf{n}$ matrix, $\mathbf{x}$ is an unknown vector and $\mathbf{b}$ is a known vector in $\mathbf{R}^{\mathbf{n}}$.

The standard Gaussian elimination process involves the LU-factorization of **A** such that **A=LU**, where **L** and **U** are *lower triangular* and *upper triangular* matrices respectively. In general, we have **PA=LU**, where **P** is a permutation matrix. This is followed by a forward substitution step wherein a temporary vector **y** is first computed from

$$\mathbf{Ly}=\mathbf{b} \tag{2.2}$$

after which **x** is computed in the backward substitution step from

$$\mathbf{Ux}=\mathbf{y}. \tag{2.3}$$

In case the permutation matrix **P** is not the identity, then we can replace the known vector **b** by the vector **Pb** in Equation (2.2). It must also be noted that Equations (2.2) and (2.3) can be solved without explicitly computing matrix inverses since the corresponding matrices are triangular. However, as the size of the matrix, **n**, becomes large, even Gaussian elimination turns out to be prohibitively expensive.

Algebraically, tearing can be considered as reordering the network variables such that Equation (2.1) has a *bordered block diagonal* (BBD) form

$$\begin{bmatrix} \mathbf{D} & \mathbf{P} \\ \mathbf{Q}^T & \mathbf{T} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \mathbf{w} \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \mathbf{s} \end{bmatrix} \tag{2.4}$$

where $\mathbf{w} \in \mathbf{R}^k$ is the vector of tearing variables and $\mathbf{v} \in \mathbf{R}^m$ is the vector of the remaining unknown variables. **T** is a $\mathbf{k}\times\mathbf{k}$ tearing matrix corresponding to the variables in **w**. Removal of the variables in **w** tears the network into μ independent subnetworks. **D** is an $\mathbf{m}\times\mathbf{m}$ *block diagonal* matrix corresponding to these subnetworks. Assuming that the i^th^ subnetwork has $\mathbf{m}_i$ variables and the $\mathbf{m}_i\times\mathbf{m}_i$ matrix corresponding to this is $\mathbf{D}_i$, we then get the following partition :

$$\mathbf{D} = \begin{bmatrix} \mathbf{D}_1 & & & \\ & \mathbf{D}_2 & & \\ & & \cdot & \\ & & & \cdot \\ & & & & \mathbf{D}_\mu \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \cdot \\ \cdot \\ \mathbf{v}_\mu \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \cdot \\ \cdot \\ \mathbf{y}_\mu \end{bmatrix}.$$

Further $\mathbf{P}^T = [\,\mathbf{P}_1^T\,\mathbf{P}_2^T\,..\,\mathbf{P}_\mu^T\,]$ and $\mathbf{Q}^T = [\,\mathbf{Q}_1^T\,\mathbf{Q}_2^T\,..\,\mathbf{Q}_\mu^T\,]$ where $\mathbf{P}_i$ and $\mathbf{Q}_i$ are $\mathbf{m}_i\times\mathbf{k}$ matrices constituting the border.

The solution strategy is to first eliminate the variables $\mathbf{v}$ from the system resulting in the following reduced subsystem :

$$(\mathbf{T} - \mathbf{Q}^T\mathbf{D}^{-1}\mathbf{P})\mathbf{w} = \mathbf{s} - \mathbf{Q}^T\mathbf{D}^{-1}\mathbf{y}. \tag{2.5}$$

Solving (2.5) gives the tearing variables $\mathbf{w}$, after which the ith subnetwork can be solved to yield $\mathbf{v}_i$ as

$$\mathbf{D}_i\mathbf{v}_i = \mathbf{y}_i - \mathbf{P}_i\mathbf{w} \tag{2.6}$$

for each $i = 1,2,\dots,\mu$.

It must be noted that both Equations (2.5) and (2.6) represent subproblems much smaller than the original problem since, typically, $\mathbf{k} \ll \mathbf{n}$ and $\mathbf{m}_i \ll \mathbf{n}$. Further, these equations can be solved without actually inverting any of the matrices involved. The details are given in [35] and will not be discussed here. Parallel processors could be employed to solve Equation (2.6) for different subnetworks. Thus tearing aids in saving computation time over Gaussian elimination of a rather large system of linear equations.

2.2.1.2 Tearing of Nonlinear Systems

At the nonlinear equation level, tearing is applied in the multilevel Newton iteration procedure used in MACRO [4]. In this approach the circuit is assumed to be described in a hierarchical fashion. In a two-level hierarchy, a circuit is composed of certain functional units, called blocks. Each block is a small subnetwork consisting of basic circuit elements such as transistors, resistors, and capacitors. The circuit variables in a block are divided into two categories, namely, *endogenous* - those that interact only with variables inside the block, and *exogenous* - those that also interact with variables outside the block. Let $\mathbf{u} \in \mathbf{R}^k$ denote the exogenous variables for a subcircuit. The endogenous variables are, in turn, partitioned into two sets. The first set, called the *output* variables, and denoted by $\mathbf{y} \in \mathbf{R}^k$, are in

1-1 correspondence with the exogenous variables. For example, if the exogenous variables are chosen to be node voltages, then the set of output variables will be branch currents entering the subcircuit from these nodes. The second set, denoted by $\mathbf{x} \in \mathbf{R}^m$, is the set of *internal* variables.

The static behavior of each subcircuit can be determined by solving a system of equations of the form

$$\mathbf{H}(\mathbf{u},\mathbf{x},\mathbf{y}) = \mathbf{0}. \tag{2.7}$$

Given $\mathbf{u}$, the interaction of the subcircuit with the rest of the circuit is completely described by $\mathbf{y}$. Thus Equation (2.7) can be solved to yield an *exact macromodel* for the subcircuit, which is a mapping from $\mathbf{u}$ to $\mathbf{y}$. Therefore the original circuit can be treated as composed of black boxes whose input-output behavior is modeled by macromodels, leading to the network equations of the form

$$\mathbf{F}(\mathbf{u},\mathbf{y},\mathbf{w}) = \mathbf{0} \tag{2.8}$$

where $\mathbf{w} \in \mathbf{R}^p$ is a vector of network variables not interacting with any of the subcircuits.

The two-level Newton-Raphson algorithm can then be described as follows. Each subcircuit having equations of the form of Equation (2.7) is first solved using a Newton-Raphson iterative technique yielding $\mathbf{y}$ as a function of $\mathbf{u}$ denoted by $\mathbf{y} = \mathbf{G}(\mathbf{u})$. The next level of Newton-Raphson iterations is applied to Equation (2.8) with $\mathbf{y} = \mathbf{G}(\mathbf{u})$ to yield the complete solution to the network.

The two-level technique can easily be extended to many levels of hierarchy in the circuit and is extremely useful if circuits are described in a multilevel hierarchical fashion. The main advantage in using this approach is that, at each level, the Newton-Raphson algorithm is applied only to a relatively small number of equations, thus gaining computational speed. Like other tearing methods, this scheme permits individual subcircuits to be processed in parallel while still retaining the essential properties of the corresponding standard technique, which in this case is the *quadratic convergence* of the Newton-Raphson method.

2.2.2 Relaxation Decomposition

Relaxation or temporal decomposition techniques are used by several nonstandard analog simulators such as MOTIS [5], SPLICE [13], RELAX [10], and SAMSON [15], to achieve higher computational speeds. Relaxation can also be applied at any of the three levels of the standard circuit simulation approach, namely, the linear equation level, the nonlinear equation level, and the differential equation level. These methods are characterized, however, by completely different numerical convergence and stability properties.

2.2.2.1 Relaxation of Linear Systems

As in Section 2.2.1.1, suppose, once again, that the linear system of equations to be solved is of the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{x}, \mathbf{b} \in \mathbf{R}^n$, and $\mathbf{A}$ is an $\mathbf{n}\times\mathbf{n}$ matrix. There are two well-known relaxation techniques that could be used to solve the above system iteratively. These are the Gauss-Jacobi method and the Gauss-Seidel method. Both these methods are iterative in nature, as are relaxation methods in general, and generate a sequence of vectors $\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \cdots, \mathbf{x}^i, \mathbf{x}^{i+1}, \cdots$ where $\mathbf{x}^0$ is some initial guess. This sequence *converges* to a solution $\mathbf{x}^*$ for any initial guess, provided some conditions involving the matrix $\mathbf{A}$ are met. In this case the iterations stop when the error $\delta^{i+1} = \| \mathbf{x}^{i+1} - \mathbf{x}^i \| < \epsilon$ where $\epsilon > 0$ is preassigned.

The relaxation begins by partitioning $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{L}+\mathbf{D}+\mathbf{U} \tag{2.9}$$

where $\mathbf{L}$ and $\mathbf{U}$ are *strictly lower* and *strictly upper* triangular matrices and $\mathbf{D}$ is a *purely diagonal* matrix. Thus the original system of equations can written as

$$\mathbf{Dx} = \mathbf{b} - \mathbf{Lx} - \mathbf{Ux}. \tag{2.10}$$

The Gauss-Jacobi method then computes $\mathbf{x}^{i+1}$ from $\mathbf{x}^i$ as

$$x^{i+1} = D^{-1}(b - (L+U)x^i) \tag{2.11}$$

while the Gauss-Seidel computes

$$x^{i+1} = D^{-1}(b - Lx^{i+1} - Ux^i). \tag{2.12}$$

More precisely, Gauss-Seidel computes the j^{th} component of x^{i+1} as j is incremented from 1 to n as follows :

$$x_j^{i+1} = D_{jj}^{-1}(b_j - \sum_{k=1}^{j-1} L_{jk}x_k^{i+1} - \sum_{k=j+1}^{n} U_{jk}x_k^i) \tag{2.12a}$$

since $L_{jk} = 0$ for $k \geqslant j$ and $U_{jk} = 0$ for $k \leqslant j$ by definition.

From Equation (2.11) one gets

$$(x^{i+1} - x^i) = -D^{-1}(L+U)(x^i - x^{i-1})$$

and hence, for the Gauss-Jacobi method

$$\delta^{i+1} \leqslant \|D^{-1}(L+U)\|\,\delta^i \tag{2.13}$$

by definition of the *induced norm* of a matrix [38]. Similarly, for the Gauss-Seidel method one gets

$$\delta^{i+1} \leqslant \|(L+D)^{-1}U\|\,\delta^i \tag{2.14}$$

In either case, we have $\delta^{i+1} \leqslant \|M\|\,\delta^i$ where M denotes, generically, the matrices involved in Equations (2.13) and (2.14). From the above equations, it can be shown that these relaxation methods have the following properties :

a) The iterations converge (i.e. $\delta^i \rightarrow 0$ as $i \rightarrow \infty$) for any initial guess x^0 if and only if $|\lambda(M)| < 1$ for each eigenvalue λ of M.

b) The iteration converges in one step if the rows and columns of A are permuted such that U is identically zero.

c) Speed of convergence, in most cases, is improved if A is permuted into *nearly lower* triangular form.

d) In general, convergence depends on the numerical properties of L, D, and U. Convergence is typically rapid for the first few iterations, and then gets progressively slower. The *asymptotic* rate of convergence is linear.

e) The speed of convergence of the Gauss-Seidel method is generally faster than that of the Gauss-Jacobi method.

The advantage of the Gauss-Seidel method is that at each iteration only a triangular system of equations has to be solved. Moreover, considerable improvement in speed of convergence can usually be achieved if **A** can be permuted into a form which is nearly triangular. The disadvantage of this method is its weak convergence. In some cases, if convergence is achieved, it is only linear. Thus if **M** has an eigenvalue of modulus close to 1, it may take many iterations to reduce the error by an order of magnitude. If **A** is *diagonally dominant*, which implies that all eigenvalues of **M** have modulus strictly less than 1, then convergence is guaranteed.

2.2.2.2 Relaxation of Nonlinear Systems

Relaxation methods to solve nonlinear difference equations are used in a class of analog simulators, known as *timing simulators* [5-8]. The algorithms used in these simulators depart radically from the methods used in standard circuit simulators in a number of ways; some of which are

1) The types of networks are restricted to circuits containing only MOS transistors and lumped capacitors from each node to ground.

2) The nonlinear device characteristics, in most cases, are stored in tables, and are not evaluated analytically during simulation.

3) Both sparse Gaussian Elimination and conventional Newton-Raphson techniques are discarded as solution methods and some accuracy may be sacrificed in the quest for speed.

The first timing simulator to be implemented was MOTIS [5], which, in fact, is still considered a landmark in the Computer-Aided Design (CAD) area. The original MOTIS, as implemented, had some problems with accuracy, convergence, and *coupling* such as floating capacitors (i.e., a capacitor across two nodes). Several simulators such as MOTIS-C [6], SPLICE [13], and MOTIS-II [7], were implemented subsequently to overcome some of these problems. To elucidate some of the ideas used in these simulators, assume that the nodal equations of an MOS network are of the form

$$\mathbf{C}\dot{\mathbf{v}} + \mathbf{J}(\mathbf{v}) = 0 \tag{2.15}$$

where $\mathbf{v} \in \mathbf{R}^m$ is the vector of node voltages as a function of time, $\dot{\mathbf{v}}$ is its time derivative, $\mathbf{C}$ is the capacitance matrix, and $\mathbf{J}(\mathbf{v})$ is the vector of currents feeding the capacitors. Using the Backward Euler method to discretize the time derivative operator, we get

$$\dot{\mathbf{v}}^{n+1} = (\mathbf{v}^{n+1} - \mathbf{v}^{n})/\mathbf{h}_n \tag{2.16}$$

where $\mathbf{v}^k$ is the value of $\mathbf{v}$ computed at time $\mathbf{t}_k$, and $\mathbf{h}_k = \mathbf{t}_{k+1} - \mathbf{t}_k$. Assuming that the values of $\mathbf{v}$ have been computed at time points $\mathbf{t}_0, \mathbf{t}_1, \cdots, \mathbf{t}_n$, we now develop the procedure to evaluate $\mathbf{v}^{n+1}$. Substituting Equation (2.16) into Equation (2.15) and denoting the unknown variable $\mathbf{v}^{n+1}$ by $\mathbf{y}$, we get

$$\mathbf{C}\mathbf{y} + \mathbf{h}_n\mathbf{J}(\mathbf{y}) - \mathbf{C}\mathbf{v}_n = 0, \tag{2.17}$$

which, in general, can be rewritten as a system of nonlinear equations of the form

$$\begin{gathered} g_1(y_1,y_2,\cdots,y_m) = 0 \\ g_2(y_1,y_2,\cdots,y_m) = 0 \\ \vdots \\ g_m(y_1,y_2,\cdots,y_m) = 0 \end{gathered} \tag{2.18}$$

The relaxation techniques used to solve the above equations are often termed as *point-wise relaxation* methods as opposed to *waveform relaxation* methods [9], wherein the relaxation is applied at the differential equations level itself. The point-wise relaxation techniques solve equations in (2.18) by sweeping one equation at a time and solving for one variable at a time while *relaxing* the remaining variables to their previous values. The process is repeated until the unknown variables converge or the iteration count exceeds a preset value. In MOTIS a Gauss-Jacobi-like scheme is used to solve equations in (2.18) *approximately* by obtaining $\mathbf{y}_i$ from the following scalar equation :

$$g_i(v_1^n, v_2^n, \cdots, v_{i-1}^n, y_i, v_{i+1}^n, \cdots, v_m^n) = 0 \tag{2.19}$$

It must be pointed out that the above nonlinear scalar equation could be solved using a Newton-Raphson iterative procedure. In MOTIS, however, the solution is taken to be the value obtained after the first iteration itself. Furthermore, the values of $\mathbf{y}_i$ obtained after the first sweep of the equations in (2.18) are taken to be the values of $\mathbf{v}_i^{n+1}$ and, once again, the iterations are not carried out until convergence. Thus the algorithms in MOTIS compute a vector $\mathbf{y}$ which solves the equations in (2.18) approximately, and sets $\mathbf{v}^{n+1} = \mathbf{y}$. These approximations are justified when sufficiently small time steps are taken to discretize the equations in (2.17).

The MOTIS-C program [6] modifies the procedure used in MOTIS by using a Gauss-Seidel-like approach, which computes $\mathbf{y}_i$ from the following equation :

$$g_i(v_1^{n+1}, v_2^{n+1}, \cdots, v_{i-1}^{n+1}, y_i, v_{i+1}^n, \cdots, v_m^n) = 0. \tag{2.20}$$

Once again, the above nonlinear scalar equation is solved only approximately by stopping after a single Newton-Raphson step. Furthermore, only a single relaxation sweep is taken through the equations in (2.18). In SPLICE [13], this approach is modified by repeatedly sweeping through the equations in (2.18) until convergence is achieved or until the number of iterations equations exceeds an *a priori* bound, in which case, the time step $\mathbf{h}_n$ is reduced and the process is repeated. The advantage of using a Gauss-Seidel-like approach over a Gauss-Jacobi-type approach used in MOTIS, is that, usually, the Gauss-Seidel iterations converge more rapidly.

The program PREMOS [8] uses a modified Gauss-Seidel predictor algorithm for the solution of equations in (2.18). In this approach, while solving the ith equation for the variable $\mathbf{y}_i$, the previous variables are updated, i.e., $y_j = v_j^{n+1}$ for $j<i$, while the variables with $j>i$ are predicted by $y_j = v_j^n + (v_j^n - v_j^{n-1})h_{n-1}/h_{n-2}$. Among all the various time-point relaxation methods discussed above, the Gauss-Seidel, with prediction, is seen to perform the best, provided sufficiently small time-steps are taken. Also, experience with SPLICE [13] and MOTIS [5] has shown that repeated iteration sweeps are required in order to achieve accuracy. The convergence and stability properties of these methods are studied in some detail in [36].

2.2.2.3 Relaxation of Differential Equations

In this section we discuss a technique in which relaxation is applied directly to the system of nonlinear algebraic-differential equations describing the circuit. As a result, the system is decomposed into several decoupled subsystems of nonlinear algebraic-differential equations, each of which can then be solved using standard techniques, namely, stiffly stable, implicit numerical integration methods, Newton-Raphson iterations, and sparse Gaussian elimination. Furthermore, this type of decomposition allows the latency of the subsystems to be exploited in the most natural way. This relaxation technique is called the Waveform Relaxation Method (WRM) [9] and is used in the simulator RELAX [10].

In order to describe the WRM process, consider the nonlinear algebraic-differential equations describing the behavior of any general circuit to be of the form

$$f(\dot{x}(t),x(t),u(t)) = 0 \tag{2.21a}$$

$$E(x(0) - x_0) = 0 \tag{2.21b}$$

where $t \in [0,T]$ is the independent time variable, $x(t) \in R^p$ is the vector of unknown variables at time t, $\dot{x}(t)$ is the time derivative of x at time t, $u(t) \in R^r$ is the vector of input variables at time t, $x_0 \in R^p$ is the given initial value of x, $f : R^p \times R^p \times R^r \rightarrow R^p$ is a continuous function, and $E \in R^{n \times p}$ is a matrix of rank $n \leq p$, such that $Ey(t)$ is the *state* of the circuit at time t. Alternatively, the vector function $x(t)$, $t \in [0,T]$ can be treated as an element x in the vector space of bounded functions $L_\infty^p[0,T]$, with the norm defined as

$$\|x\| = \max_{t\in[0,T]} \|x(t)\|_p \tag{2.22}$$

where for any $z\in\mathbf{R}^p$ we define

$$\|z\|_p = \max_{j=1,2,\cdots p} |z_j|$$

where $z_1,z_2,\cdots,z_p$ are the *scalar components* of z.

There are two major processes involved in the WRM algorithm for solving the equations in (2.21) over a given time interval $[0,T]$, namely, the *assignment-partition* process and the *relaxation* process. In the assignment-partition process, each unknown variable is assigned to an equation in which it is involved. Then the system of equations in (2.21a) is partitioned into m disjoint subsystems of equations of the following form in which the dependence on time is not explicitly shown :

$$\begin{bmatrix} f_1(\dot{x}_1,x_1,d_1,u) \\ f_2(\dot{x}_2,x_2,d_2,u) \\ \cdot \\ \cdot \\ f_m(\dot{x}_m,x_m,d_m,u) \end{bmatrix} = 0 \tag{2.23a}$$

$$E(x(0)-x_0) = 0 \tag{2.23b}$$

where, for each $i=1,2,\cdots,m$, $x_i\in\mathbf{R}^{p_i}$ is the subvector of unknown variables assigned to the ith partitioned subsystem, $f_i:\mathbf{R}^{p_i}\times\mathbf{R}^{p_i}\times\mathbf{R}^{2p-2p_i}\times\mathbf{R}^r\rightarrow\mathbf{R}^{p_i}$ is a continuous function, and

$$d_i = (x_1,\cdots,x_{i-1},x_{i+1},\cdots,x_m,\dot{x}_1,\cdots,\dot{x}_{i-1},\dot{x}_{i+1},\cdots,\dot{x}_m)^T.$$

For the ith subsystem, x_i is the vector of *endogenous* variables, while x_j, with $j\neq i$, are the vectors of *exogenous* variables. If, for each $i = 1,2,\cdots,m$, the vector d_i is treated as an input to the ith subsystem, then clearly, the solutions of the equations in (2.23a) can be obtained by solving the m subsystems independently. Therefore, the vector d_i is called the *decoupling* vector for the ith subsystem.

The relaxation process starts with an initial guess of the waveforms for each unknown variable and solves the equations in (2.23) iteratively. During each iteration, each subsystem is solved for its endogenous variables for the entire time interval $[0,T]$ by using approximated waveforms for its decoupling vectors. If we use the superscript $\mathbf{k}$ to denote vectors obtained during the $\mathbf{k}^{th}$ iteration, then the WRM algorithm can be described as starting with an initial guess of waveforms $\mathbf{x}^0(t) : t \in [0,T]$ such that $\mathbf{x}^0(0) = \mathbf{x}_0$ and sweeping through the equations in (2.23a) one by one such that during the $\mathbf{k}^{th}$ iteration, the waveforms $\mathbf{x}_i^k(t) : t \in [0,T]$ are obtained by solving

$$\mathbf{f}_i(\dot{\mathbf{x}}_i^k, \mathbf{x}_i^k, \mathbf{d}_i^k, \mathbf{u}) = 0 \tag{2.24a}$$

$$\mathbf{E}_i(\mathbf{x}_i^k(0) - \mathbf{x}_{i0}) = 0 \tag{2.24b}$$

where, if Gauss-Seidel relaxation is used, then the decoupling vectors are taken as

$$\mathbf{d}_i^k = (\mathbf{x}_1^k, \cdots, \mathbf{x}_{i-1}^k, \mathbf{x}_{i+1}^{k-1}, \cdots, \mathbf{x}_m^{k-1}, \dot{\mathbf{x}}_1^k, \cdots, \dot{\mathbf{x}}_{i-1}^k, \dot{\mathbf{x}}_{i+1}^{k-1}, \cdots, \dot{\mathbf{x}}_m^{k-1})^T$$

or, if Gauss-Jacobi relaxation is used, then

$$\mathbf{d}_i^k = (\mathbf{x}_1^{k-1}, \cdots, \mathbf{x}_{i-1}^{k-1}, \mathbf{x}_{i+1}^{k-1}, \cdots, \mathbf{x}_m^{k-1}, \dot{\mathbf{x}}_1^{k-1}, \cdots, \dot{\mathbf{x}}_{i-1}^{k-1}, \dot{\mathbf{x}}_{i+1}^{k-1}, \cdots, \dot{\mathbf{x}}_m^{k-1})^T.$$

The iterations stop when the error $\delta^k = \|\mathbf{x}^k - \mathbf{x}^{k-1}\|$ becomes sufficiently small, where the norm of the vector of waveforms is defined in Equation (2.22) above.

In contrast to the conditions for convergence of point wise relaxation methods discussed in the previous sections, it has been shown by Lelarasmee [9] that the conditions for convergence of the waveform relaxation method are fairly mild. First, the circuit Equations (2.21a) and (2.21b) are transformed into a *canonical* form so that the error after the $\mathbf{k}^{th}$ iteration can be expressed as a function of the error after the previous iteration in the form of a *contraction mapping*. If the initial waveform guesses and the inputs are all *piecewise continuous*, and the canonical functions are globally *Lipschitz continuous* and contractive, then it is shown in [9] that *uniform convergence* is guaranteed for the WRM algorithm under the norm defined in Equation (2.22). The convergence, however, is linear as in other relaxation methods.

In spite of the surprisingly mild conditions for convergence which are easily satisfied by most practical electronic circuits, the WRM procedure implemented in simulators such as RELAX [10] and

RELAX2 [11] suffers from certain drawbacks. The main drawback is that if fairly *strong coupling* exists between the various partitioned subsystems, as in circuits with logic feedback loops such as finite state machines, asynchronous sequential circuits, and ring oscillators, the number of iterations required for convergence may be prohibitively large and also proportional to the length of the interval of analysis. Some of the drawbacks have been overcome in RELAX2.1 [12], wherein the time interval [0,T] is partitioned into certain slots or *windows* and the subsystems are analyzed only for the duration of a present window before moving on to the next window, and so on. However, it has been shown in [37] that, in the case of stiff systems where the coupling among the subsystems causes the stiffness, the sizes of the windows have to be reduced considerably in order to keep the iteration count during a window within a prescribed bound. This would then require an extremely large number of windows to span the entire time interval of analysis.

2.3 Digital Simulation

Digital simulators [13-26], or logic simulators as they are often called, form an important class of computerized tools for designing very large integrated circuits. These simulators provide a discrete "on/off" type analysis of the circuit under test. Signal values are described by a fairly small number of discrete levels rather than in a continuous range as is the case in an analog simulator. Through the use of very simple models for the devices and Boolean arithmetic to perform operations on the discrete signal values, digital simulators are often capable of economically analyzing circuits containing the equivalent of over 100k active devices. The dynamics of the circuit are, however, modeled by simply *delaying* the various signal transitions between the discrete levels. In most cases a simple, user-defined rise and fall delay between the input and output of a logic-gate or transistor-group is used. Thus digital simulators, at best, provide a fairly crude, first-order timing analysis of the circuit under consideration.

Digital simulators are useful and popular since most integrated circuits are primarily digital in nature. The usefulness of a digital simulator, however, depends greatly on the consistency and accuracy with which it can model the logic behavior of a full range of design techniques available to the designers of integrated circuits. Of course, no digital simulator can model all designs with complete accuracy, because it does not simulate the detailed analog behavior of the circuit. It should, nonetheless, provide as close a model as possible within a set of well-defined limitations. As a further requirement, a digital simulator for VLSI circuits must be efficient enough to simulate entire systems with reasonable speed. A digital simulator has, as its basis, an abstract model of how digital systems function. This *logical model* describes both the structure and the behavior of a system in terms of a set of primitive elements, a set of interconnections, and a set of rules for operation. For a simulator to accurately and reliably simulate a system, the logical model must reflect its actual structure and operation. Digital simulators can be divided into two categories, namely, Boolean gate-level simulators [13-18] and switch-level simulators [19-27].

2.3.1 Gate-level Simulation

The Boolean logic gate model has formed the theoretical basis for logic design ever since the advent of electronic logic. In this model a circuit is composed of several logic gates connected by unidirectional, memoryless wires. The logic gates themselves are collections of transistors and/or other circuit elements which perform a logic function. A logic gate may be a simple inverter, NAND gate, or NOR gate, or a more complex functional unit such as flip-flops and registers. The logic gates compute Boolean functions of their input signals and transmit these values along wires to the inputs of other gates to which it might be connected. Each gate input has a unique signal source. Information is stored only in feedback paths of sequential circuits. The Boolean gate model directly implements the well-known two-valued Boolean algebra and hence has a well-defined specification which can guide the simulator implementation.

The unilateral nature of logic gates is fundamental to the operation of gate-level simulators. For each binary vector at the input nodes of a logic gate, the binary value (i.e., **0**, or **1**) at the output is computed and propagated on to the inputs of other gates that might be connected to it. Due to the inertial elements such as node capacitances present in the circuit, however, a change in the state of the input to a gate would propagate to the output only after a certain time delay. Simulators which do not account for this delay can analyze only combinational circuits. Thus, simulators which handle sequential circuits must estimate the propagational delay through a logic gate and they do so in several ways. Some simulators operate in the so-called *unit delay* mode, where all logic gates are assumed to have the same delay. Unit-delay simulators, however, can verify only the steady-state behavior or the logic functionality of the digital circuit. In order to provide some kind of timing information, some simulators allow *assignable delays* where the user can assign specific delays through any of the logic gates used in the simulation. Even in assignable delay simulators, the delay values may only be integer multiples of a fundamental time quantum, usually referred to as the *minimum resolvable time* (MRT). For example, the MRT in a certain simulator may be 0.1 ns, in which case a gate delay of 10 units represents an effective delay of 1.0 ns.

The difference in propagational delays through different signal paths in a network of logic gates may sometimes cause undesirable situations, such as *static hazards* and *dynamic hazards*. Hazards [28,29,39,40,64] are situations where it is possible for spurious *glitches* or spikes to appear in an otherwise smooth analog waveform at the output of a logic gate. In a sequential circuit, the occurrence of a glitch could cause the circuit to malfunction. Therefore, the detection of hazards and race conditions [23,62,65] are very important, and hence, most digital simulators caution the user when they occur. The detection of hazards is possible by introducing a third state, usually denoted by **X**, to represent signal transitions [28,29,39,40,64]. In this dissertation, we do not consider race conditions since we assume that timing is known, and hence any potential race condition will be resolved according to the timing.

The Boolean gate model cannot represent many of the design techniques currently used in VLSI design. This is especially true in the case of MOS VLSI circuits. The MOS *pass transistor* is often used to implement combinatorial logic in ways which resemble relay contact switches more closely than logic gates. These bidirectional elements are difficult to handle using the gate model and are often approximated by unidirectional gates. Dynamic memory can store information without feedback paths by exploiting the capacitances of the wires and the gate terminals of the transistors attached to them. A variety of bus structures is often used to provide multidirectional, multipoint communication. Hence, most existing digital simulators extend the Boolean gate model in various ways to handle MOS circuits.

Many simulators extend the two-valued logic of Boolean algebra with a third value to represent an *unknown* or *undefined* logic level. This **X** state could indicate an uninitialized signal, a signal held between two logic thresholds, or a signal in a **0**→**1** or **1**→**0** transition. The **X** state is handled algebraically by extending the binary Boolean algebra to a ternary or three-valued DeMorgan's algebra [18,39]. Thus, even with this extension, many of the desirable mathematical properties of the Boolean gate model are preserved. The **X** state implemented this way is also useful in the detection of hazards and race conditions [23,28,39,40,62,64]. Alternatively, some simulators implement the **X** state by an enumeration technique in which the simulation is repeated with the nodes in the **X** state set to all possible combinations of **0**'s and **1**'s [41]. Nodes that remain in a unique binary state for all combinations are set to this state, while all others are set to **X**. To simulate tristate gates and logic busses, some simulators use a fourth state, called the *high impedance* state, and often denoted by **H** [16]. This **H** state is also used sometimes to model dynamic memory by allowing a node to retain its previous logic state if the outputs of all logic gates connected to the node are at the **H** level.

As far as simulation is concerned, most gate-level simulators belong to one of two general types. The first is based on the Huffman logic model [42], as shown in Figure 2.1. In this model, all the feedback paths in the network are initially broken resulting in a purely combinatorial network, which is

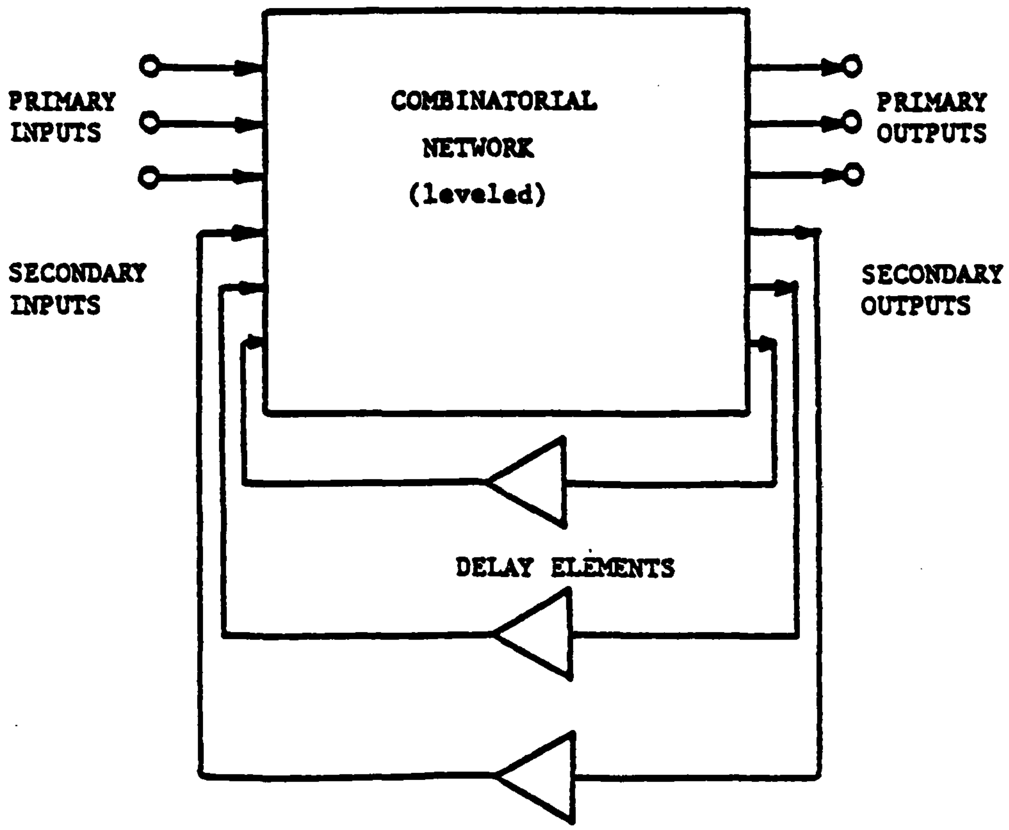


Figure 2.1 : The Huffman logic model for logic analysis

then levelized in terms of signal dependence. The feedback is restored by inserting delay elements between the secondary outputs and secondary inputs of the combinatorial part of the network. The analysis begins by applying the input excitations and following paths where the signal states change through the network to the outputs. The delays are applied to any secondary output change and the analysis of the combinatorial block begins once again. The process is repeated until the requested input sequence has been completed. This approach is used in SALOGS [16], and is quite efficient for circuits where relatively few delays are significant or, in other words, for *nearly combinational* circuits.

The second and more common approach is based on the use of a time queue (TQ) [43] as shown in Figure 2.2. Each entry in the queue represents a discrete point in simulation time. Time moves ahead in fixed increments which correspond to consecutive entries in the TQ. Each entry in the queue contains a pointer to a list of *events* which are to occur at that instant of time. An event is usually defined as a change in the logical state of an output node of an *element*. The element, in this case, may be a voltage source or a logic gate. The new state may or may not be the same as the state already held by the output line. If the new state is different from the old one, then all elements whose input lines are connected to this output line, called *fanout* elements, must be processed to see if this change affects their outputs. If an element gets processed at say, time t_i, and the input event is found to cause an output event, then the output event is assumed to occur at time t_{i+k} where $k>0$ represents a *positive delay* through the logic gate. The fanouts of the output node then get scheduled for processing at time t_{i+k}. If the state of an output node remains unchanged, then the fanouts are not added to the time queue. This approach is often referred to as a *selective trace* technique, or an *event-driven* scheme, or sometimes even as *dynamic leveling*. In the case of logic simulation, no penalty in accuracy or stability of analysis is incurred with the use of the selective trace method. One of the advantages of this scheme is that it allows different gates to have different delays and, moreover, the delay value through a gate is also allowed to change as the simulation proceeds. This is especially good for MOS logic gates which have different delays for *rising* transitions and *falling* transitions at the output respectively. Furthermore, the presence of feedback among the logic gates does not complicate the simulation, since the delay

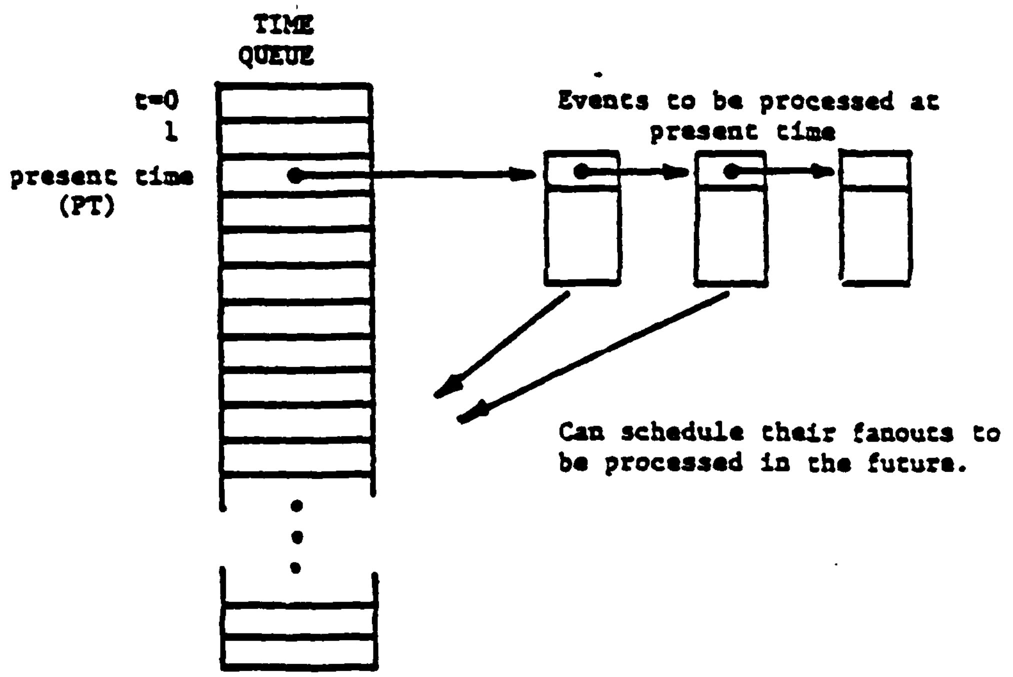

TIME
QUEUE
t=0
1
present time
(PT)
Events to be processed at
present time
Can schedule their fanouts to
be processed in the future.

Figure 2.2 : The principle of the time queue simulator

through a feedback loop would schedule a gate for processing only in the future and never at the present time in the queue. Several logic simulators such as TEGAS [17], and SPLICE [13], use the TQ approach successfully.

Gate-level simulators, however, are not entirely suitable for the digital or logic simulation of MOS circuits. This is due to the fundamental mismatch between the Boolean gate model and the behavior of MOS logic circuits. MOS circuits consist of bidirectional switching elements connected by bidirectional wires with memory (considering the capacitance of the interconnect and of the transistor gates as contributing to the wire's memory). Hence, the need for a different approach to the digital modeling and simulation of MOS circuits is apparent and is discussed in the following section.

2.3.2 Switch-level Simulation

A new class of digital simulators known as switch-level simulators has emerged fairly recently as an alternative to the more conventional gate-level simulators, specifically for the simulation of MOS VLSI circuits. The Boolean gate concept is discarded altogether in these simulators, and is replaced by a bidirectional *switch model* which closely matches the structure and behavior of MOS circuits.

One of the first switch-level simulators to be implemented is MOSSIM [19], in which an MOS logic network is modeled as a set of nodes interconnected by a set of transistor switches. MOSSIM uses three logic levels, **0**, **1**, and **X**, to describe signal values at the various nodes. The level **X** is the *undefined* or sometimes *unknown* level used to represent a signal level that cannot be uniquely determined due to an ambiguity in the network condition. Each node is also assigned a *strength* which indicates the extent to which the node can force its value on other nodes connected to it via a path of conducting switches. *Input nodes* are the strongest and provide externally generated signals such as, power lines, ground, clock drivers, and data inputs. A node connected to a voltage source through a pullup resistor is called a *pullup node*. The pullup resistor is normally realized using a depletion transistor with gate and source terminals shorted. A pullup node is at level **1** unless there is a path of conducting transistors to an input node, in which case the pullup node takes on the value of the stronger node. The rest of the nodes in the circuit are *normal nodes*. These nodes are the weakest and are capable of only storing charge dynamically. Thus we have three types of nodes with strengths ordered as **input > pullup > normal**.

An MOS transistor is modeled as a three node device which acts as a bidirectional switch between its *drain* and *source* nodes with the signal at the *gate* node controlling the state of the switch. There are two types of transistors allowed in MOSSIM : n-type and p-type. When the gate signal is a **0**, the n-type (p-type) switch is open (closed), and when the gate signal is a **1**, the n-type (p-type) switch is closed (open). The status of either switch is *unknown*, i.e., it may be open, closed, or somewhere between, when the gate signal is in the **X** state. When a switch is closed, it is treated as a bidirectional switch, and no distinction is made between drain and source nodes of the device.

The network can be described to MOSSIM in terms of transistors, logic gates, and user-defined macros, but these are all translated into a transistor level representation for simulation. The program begins by *splitting* each input node (including the ground node) into a number of physical input nodes, one for each transistor to which it is connected. This is possible since the input nodes provide strong signals to the network which cannot be modified by the internal operations of the network. The gate node of a transistor is treated as a pure input to the switch and its state determines the conduction state of the switch. This helps partition the set of transistors into *groups*, which can be defined as follows : Consider an undirected graph with a vertex for every node in the circuit and an edge between drain and source nodes for each transistor. The graph will then have several connected components. The set of nodes and transistors corresponding to a component forms a group. Thus all bilateral interactions between nodes take place within a group.

A clock in MOSSIM is defined as a set of binary sequences to be applied cyclically to a set of input nodes. A *phase* is one set of clock and input data values. The basic quantum of time is a *unit step*. Within a phase, the circuit is assumed to settle down after a certain number of unit steps. The simulation begins by first initializing all nodes to the **X** state. At the beginning of each phase all input nodes are assigned their new values and the groups whose input nodes are changed are placed in an *event list* that has initialized previously. Then a series of unit steps are taken until the event list is emptied, indicating that the network has settled. Once the network settles, the simulation of the next phase can begin. During a unit step simulation, the states of the transistors within a group are held fixed and the values of the pullup and normal nodes are updated. This is done for each group in the event list. Each updating results in a certain number (possibly zero) of nodes changing states which are accumulated in a set of *active nodes*. After all the groups in the event list are simulated, the transistors whose gate nodes are active are updated and the groups in which these transistors lie are added to a new event list for use in the next unit step. By first changing the node states while holding the transistor states fixed and then changing the transistor states with the nodes fixed, the transistors, in effect, switch one unit of time after their gate nodes change. Thus if the transistor groups are treated as conventional logic gates, the simulation appears very much like an event-driven, unit-delay gate-level simulation. The procedure for updating the node states within a group, however, is very different.

We now describe the algorithms in MOSSIM used to update the states of the drain and source nodes of transistors within a group based on the concept of node strengths. Initially, all pullup nodes are set to logical **1**. Next, an undirected graph is constructed with a vertex corresponding to each node in the group and an edge between the drain and source nodes of each transistor in the *closed* state. The connected components of this graph partition the set of nodes into equivalence classes. Within each class, the strongest nodes are determined based on the ordering **input > pullup > normal**. The strength of the class is then the strength of the strongest nodes. If the states of the strongest nodes are equal, then class state is set to this state; otherwise the class state is set to **X**. If the class strength is *pullup*, the class state is always a **1**.

If the group contains *x-transistors*, which are transistors having an **X** state on their gate nodes, then the unknown switching behavior of these transistors could alter the class states. To deal with them consistently, MOSSIM adopts the following philosophy : if a node has a unique state regardless of the conduction state of the x-transistors, then the node will be set to this state; otherwise it will be set to the **X** state. Thus the state of each class computed as described above is based on the assumption that all x-transistors are *open*.

The second part of simulating the group begins by forming a *supergraph* containing a vertex for each class and an edge between two vertices if an x-transistor connects two network nodes in the two corresponding classes. The connected components of the supergraph partition the classes into a set of *superclasses*, in which each superclass is a set of classes linked by x-transistors. If a superclass contains only one class then no further analysis is needed. Otherwise, the strength of a superclass is computed as the strength of its strongest classes. The state of a superclass is set to the state of the strongest classes if they are all equal, and **X** if they are not. A class is said to be *poisoned* if its state is different from the superclass state. Furthermore, a poisoned class could poison a neighboring class which is not stronger than itself even if the state of the neighbor is the same as the superclass state. Thus poisoning can spread through classes and be stopped only by classes with greater strength than the original poisoned class. The state of each poisoned class is then reset to **X**. Once the states of all the classes have been computed, the state of each node in a class is set to its class state.

Several modifications and extensions of the basic MOSSIM philosophy have been considered by a number of authors [20-25,63,65]. In [20], Bryant provides an abstract model for the switch-level simulation of MOS logic networks which is more general and formal than the one in MOSSIM. Unlike MOSSIM, only two types of nodes, namely, input nodes and normal nodes are allowed. A third type of transistor, called d-type (for depletion), is introduced which is *closed* regardless of its gate signal. To model *ratioed logic*, transistors may have different strengths (or conductances) when in the closed state. Thus, a stronger transistor (such as an inverter pulldown) is able to override a weaker one (such as a pullup load transistor). In MOSSIM, each normal node is modeled as having a capacitance of unknown value which can store charge but cannot drive its signal onto another node in a different state. Unfortunately, this model cannot describe the behavior of many bus designs in which a relatively high capacitance bus node is connected to a node of lower capacitance (such as the storage node in a three-transistor dynamic RAM cell) resulting in both nodes having the same logic state that was originally on the bus. In the new model each normal node is assigned a *size*, which is indicative of the value of the node-capacitance.

The time and the electrical behavior of the logic network are described in a formal way in [20] by introducing the notion of a *target function*. Given a particular set of input node, transistor, and initial normal node states, the target function provides the final states of the normal nodes. For circuits free of critical races, the logical behavior of the network can be modeled by repeated application of the target function. The passage of time is modeled just as in MOSSIM, i.e., every application of the target function is like advancing a unit step in time. The electrical behavior of the network is modeled by defining the target state function in terms of a set of steady-state voltages in an *order-of-magnitude* electrical network. This class of networks models the conducting transistors by linear resistors, where the resistances (or conductances) of different strength transistors differ by orders of magnitude. As a result, any path to an input node containing only transistors of large strength is modeled as overriding any path containing a transistor with lesser strength. Similarly, the normal nodes are modeled by capacitors where the capacitances for different size nodes differ by orders of magnitude. Thus, the target states formed on a set of nodes through charge sharing depend only on the state of the largest size node(s) in the set. Furthermore, no attempt is made to accurately compute the node voltages. Instead, they are classified into three logic levels, **0**, **X**, and **1**. Although the target state is defined in terms of an electrical model, it can be computed logically, without evaluating any electrical network. By introducing an abstraction called *logic signals*, an iterative method which uses only operations on a simple, discrete algebra is used for computing the target state function. A logic signal provides a composite description of a switch-level network at some node for a particular set of node and transistor states, much in the same way as a Thevenin equivalent network for an electrical network. Finding the target state then reduces to finding a minimum solution of a set of equations involving logic signals.

In [21], Byrd et al. have independently developed a consistent, complete, circuit theoretic based interpretation of switch-level simulation and modeling. They formally relate the true behavior of real conductance networks and the switch-level model. As in Bryant's model [20], transistor switches are modeled as linear conductors whose conductances belong to an arbitrarily deep hierarchy of conductance classes, $G^1, G^2, \cdots, G^p$, where any $g^i \in G^i$ and $g^j \in G^j$ satisfies $g^i >> g^j$, if $i > j$. Some drawbacks of Bryant's solution of the conductance network using a minimum principle with a discrete algebra are pointed out and a more general circuit theoretic based procedure which expresses a signal at a normal node as a convex combination of the input signals is presented. PARCHEMIN is a switch-level simulator using these algorithms.

In [22], the notion of a well-designed circuit is introduced and an improved switch-level simulator that runs extremely fast on such circuits is presented. This simulator also detects race conditions and handles the **X** state in a clean and efficient manner. A linear-time algorithm that detects race conditions in any nonoscillating circuit (i.e., a circuit that is acyclic within a clock phase) has been developed by Ramachandran [23]. In certain cases this algorithm is overly cautious and might indicate a presence of a race condition, when in reality, the circuit has no race condition. In [65], the authors introduce a new model, known as the NC-model, for switch-level simulation, and show that the simulation of any circuit (including oscillating circuits) can be performed in quadratic time under this new model.

An alternative approach to switch-level simulation is based on generation and evaluation of symbolic logic expressions [24]. A special discrete algebra is used, and logic expressions for a node are generated hierarchically, where each level of hierarchy represents the influence of node signals of a particular strength on that node. In evaluating the logic expressions, the undefined **X** state does not present any special problem due to the versatility of the new algebra. Furthermore, simulating the basic faults in MOS circuits is easily incorporated, thereby making this a fairly attractive scheme. These ideas are used in EXPRESS-II [25], a fast and efficient switch-level fault simulator for MOS designs.

2.4 Mixed-mode or Hybrid Simulation

An ideal simulator for VLSI circuits would be one which has the speed and efficiency of digital or logic simulators while providing the accuracy and detail of an analog simulator. An attempt to achieve this is through mixed-mode or hybrid simulation. In many of the VLSI circuits the detail and accuracy provided by the analog simulators are not required for the entire circuit under investigation, but only for some *critical* areas of the circuit. This is particularly true of large digital circuits, where often a simple digital simulation (gate-level or switch-level) provides sufficient information about the performance of much of the circuit, while some parts, such as sense amplifiers in memory circuits or tightly coupled analog blocks, might require more detailed modeling and analysis.

By providing a range of models, from highly accurate and complex analog device models to much less accurate but greatly simplified gate-level or switch-level models, the circuit designer can reduce the simulation time significantly by choosing the computationally less expensive models whenever it is appropriate and possible. Another property of large circuits which may be exploited is their relative inactivity or latency. In a typical VLSI circuit, usually only less than 20% of the signals change values significantly at any one time instant.

Hybrid analysis programs allow the designer to use a combination of analysis techniques and models, ranging from circuit and timing simulation to much cheaper digital simulation, in the same program. These simulators, such as SPLICE [13], DIANA [14], and SAMSON [15], have been observed to realize a one or two order of magnitude reduction in simulation time and substantially lower memory than standard circuit simulators, while still providing a detailed circuit-level analysis where necessary.

Mixed-mode or hybrid simulators, however, work well as long as only small, isolated sections of the circuit need to be simulated as analog circuits. Unfortunately, the partitioning of the circuit into sections which require analog simulation and those which do not is not fully automatic; some amount of human intervention is still required. Furthermore, trying to combine analog and digital models in a single program requires rather unsatisfactory approximations at the interfaces. For example, if the output of a section of logic gates is to be interfaced to an input of a section modeled as an analog circuit, a logic-to-voltage waveform conversion is required. This, of course, cannot be done with any accuracy, since much of the necessary information is lacking. The resultant outputs of the analog section must then be viewed somewhat skeptically. Similarly, certain states used in logic simulators, such as the unknown state **X**, or the high-impedance state **H**, do not represent a single voltage and therefore cannot be interfaced with an analog simulator. Therefore, unless great care is exercised, a hybrid simulator could end up providing the accuracy of a logic simulator at the speed of an analog simulator, rather than *vice versa*.

2.5 Switch-level Timing Simulation

The problem of *switch-level timing simulation* of a digital circuit can be defined as follows:

Consider the analog waveform $V_n(t)$, $t \in [t_0, t_f]$ at a certain node n in a digital circuit and choose $p-1$ threshold values, ordered as $v_1 < v_2 < \cdots < v_{p-1}$. Define the *p-state digital equivalent* of V_n to be

$$X_n(t) = x_i \text{ if } v_i < V_n(t) \leqslant v_{i+1} \tag{2.25a}$$

where $x_0, x_1, \cdots, x_{p-1}$ are the p digital states and v_0 and v_p are the minimum and maximum values of the analog waveforms respectively. We also define

$$T_n = \{t_k : V_n(t_k) \in \{v_1, v_2, \cdots, v_{p-1}\}\}. \tag{2.25b}$$

Thus T_n is the set of threshold crossing times of the analog waveform at node n in the circuit, or alternatively, the set of state transition times of its p-state digital equivalent. The aim of a switch-level timing simulator is to obtain the p-state digital equivalents X_n for each $n \in \Pi$, with special emphasis on computing (or estimating) the elements of $T = \bigcup_{n \in \Pi} T_n$, where Π denotes the set of nodes of interest to the user. For brevity in notation, we shall use SLT to stand for *switch-level timing*, and so the elements of the set T of threshold crossing times will be referred to as *SLT estimates*.

Since most VLSI circuits are primarily digital in nature, the circuit designer is very often satisfied in performing an SLT simulation in the design-verification process since this enables him to estimate the propagation delays, speeds of computation, optimal clocking rates, etc. The usefulness of an SLT simulator can be measured by considering two factors, namely, the *simulation cost* which is primarily an increasing function of the CPU time and memory used and, secondly, the accuracy of the SLT estimates. There are two major approaches that could be used to perform an SLT simulation on a large digital circuit :

(1) Use an analog simulator and convert the resulting analog waveform into their p-state digital equivalents directly, by choosing an appropriate set of p-1 threshold voltages.

(2) Use a digital simulator with delay estimation that computes the p-state digital waveform at each circuit node and generates the SLT estimates.

Since it is impossible to obtain the exact waveforms analytically, in a typical VLSI circuit, the SLT estimates produced by standard circuit simulators are considered accurate enough and are often taken as references to compare the accuracies of the SLT estimates produced with other simulators. Simulators using the first approach include the so-called *timing simulators* such as MOTIS [5], MOTIS-C [6], and PREMOS [8], which are analog simulators using relaxation techniques to speed up the simulation process as described in Section 2.2.2.2 .

In spite of the several attempts made to speed up the performance of standard circuit simulators as discussed in Section 2.2, analog simulators are still very expensive to use to analyze circuits with more than 10k devices. Digital simulators, on the other hand, have a distinct advantage in speed over analog simulators. Several large circuits with over 100k transistors have been successfully handled by these simulators. However, they provide rather inaccurate SLT information due to the poor modeling of the dynamics of the circuits. Most digital simulators produce two-state digital waveforms and account for the circuit dynamics by *delaying* the transition between states. In all cases the delays are taken to be *single-threshold* delays. Furthermore, these simulators do not take into account the dependence of propagation delays on circuit parameters, such as load capacitance, strengths of devices, input slew-rates, and other factors.

Based on the above facts, one can conclude that the circuit designer who wishes to use one of the existing analog or digital simulation tools to generate SLT estimates in VLSI circuits is placed in a difficult situation. Analog simulators provide fairly accurate SLT estimates at prohibitive simulation costs, while digital simulators can handle entire VLSI circuits but provide very poor SLT estimates, or sometimes, none at all.

It is therefore clearly necessary to provide the circuit designer with a simulation tool capable of providing accurate SLT estimates for VLSI circuits at reasonable simulation costs, thus having the best features of both analog and digital simulators. To this end, one is more likely to succeed in trying to incorporate better timing models in digital simulators since efforts to speed up analog simulators seem to be approaching a limit which is far below the speeds of the digital ones. Restricting oneself to the MOS technology seems to make the problem a little easier. An attempt has been made recently to model the MOS transistor as a linear resistor resulting in an RC-delay model for the circuit dynamics which is used in RSIM [26]. This is a logic-level timing simulator which predicts the logic state of a node and uses an RC time constant to estimate the transition times if the node changes state. The transistor model in RSIM is a gate-voltage dependent resistance R_{ds} between drain and source terminals. When the switch is *closed,* we have $R_{ds} = R_{eff}$, when *open* $R_{ds} = \infty$, and when in the *unknown* state (which means $v_{gate} = X$) the drain-source connection is described by a *resistance interval,* i.e., $R_{ds} = [R_{eff}, \infty]$. The effective resistance R_{eff} is determined separately for each transistor as a function of the device width and length, the transistor type, and other device parameters. The determination of the effective resistance is made once for each transistor and is about the only device information used by RSIM. Voltages in the RSIM model are quantized into one of three values, 0, 1, or X, and decided by choosing two threshold voltages, v_{low} and v_{high}.

The effect of the resistive network on a particular node is modeled by a Thevenin equivalent circuit. The values of V_{thev} and R_{thev} are computed, in some cases approximately, based on a series-parallel-type approach which is illustrated in [27]. The value of V_{thev} (which may be a voltage interval in some cases) decides the new state of the node. If the new value at a node is different from the previous one, then a transition is scheduled $R_{thev}C_{load}$ time units later, where C_{load} is the net capacitance at the node. Actually, RSIM uses three values of the effective resistance for a transistor, namely, a static value used to determine V_{thev}, and two others to be used in determining rise and fall delays. All these values are determined in a presimulation phase using an accurate circuit simulator such as SPICE2 [1]. Charge sharing effects are also taken into account. A nice feature of this type of simulation is that the X state does not impose any particular difficulty as far as the simulation is concerned. The simulator is event-driven and is fast enough to simulate circuits of up to 50k transistors. The SLT estimates are, however, computed only by single threshold RC delays and are sometimes found to be even more than 30% off when compared with those of SPICE2, especially in the case of MOS circuits with large pass-transistor chains.

This dissertation deals primarily with the development of a switch-level timing simulator with an empirically observed accuracy of the SLT estimates generated to be within 10% of those of SPICE2. The high accuracy of the SLT estimates without the use of an analog simulator can be attributed to the use of a *delay-operator* which will be discussed in detail in Chapter 5. This operator uses a notion of *two-threshold* delays, and is thus able to account for, among several other factors, the effect of the slope of the analog input waveforms on the timing at the output of a logic gate or a functional block.

CHAPTER 3

NETWORK PARTITIONING AND ORDERING

In this chapter an MOS network model that is used to provide accurate switch-level timing (SLT) estimates will be presented. The network is then partitioned into several subnetworks, or blocks. The set of blocks is further partitioned into its strongly connected components (SCC). The SCC's in the network are then ordered for simulation. Throughout this dissertation, the algorithms will be outlined and discussed for n-channel MOS (NMOS) circuits with *depletion loads* only. Several extensions to handle circuits with other technologies, such as complementary MOS (CMOS), will, however, be mentioned briefly in Chapter 8.

3.1 NMOS Network Model

An NMOS digital network $\mathbf{\Omega}$ consists of a set of *nodes* N interconnected by a set of n-channel MOS transistors M. The network description can be extracted directly from the layout using circuit extractors [49,61], or has to be given by the user. In any case, the network description is assumed to contain a netlist of all the NMOS transistors along with several geometrical and process parameters such as length (L) and width (W) of each device, zero-bias device threshold voltage (VTO), transconductance parameter (KP), the analog waveform at the input sources and a fixed lumped capacitance from each node to ground. Specifying a grounded capacitance from each node might seem to be a restriction, but most circuit extractors could be asked to compute equivalent device capacitances along with the capacitance due to the interconnect regions. In this chapter, the only device parameter used will be VTO. This parameter will separate the set of transistors into enhancement and depletion types. The rest of the parameters will be used in Chapters 4 and 5 to generate accurate SLT estimates.

There are three types of nodes : *input* nodes, *pullup* nodes, and *normal* nodes. Input nodes, which are modeled as voltage sources, provide the strongest signals to the network from the outside. Examples of input nodes include the power supply (V_{DD}), the ground node, as well as all the input clock signals. Pullup nodes are attached to the power supply V_{DD} via a pullup resistor. These include the output nodes of NMOS inverters, NAND gates, NOR gates, etc. A pullup node retains the value of the supply unless forced to ground through a path of conducting devices. The remaining nodes in the circuit are classified as normal nodes. These are the weakest nodes as they cannot force their signals on a stronger node but are capable of storing a signal dynamically.

In the context of switch-level timing simulation, as defined in Section 2.5 of this thesis, the user is only interested in obtaining p-state digital equivalents of the analog waveforms at various nodes in the circuit over a certain time interval $[t_0 , t_f]$. Clearly, the larger the number of states, the better is the level of detail provided, and thus, the more useful is the information to the user. It is also clear that using an analog simulator to obtain the analog waveforms and then converting them to p-state digital equivalents is highly cost-ineffective for large integrated circuits. Hence, it is desirable to generate the required digital equivalents directly via p-state digital simulation. However, the complexity of digital simulation dramatically increases with the number of states p, particularly in the context of generating accurate timing estimates. The choice of $p=2$ must be rejected outright, since in this case only binary (i.e., 0 or 1) waveforms are produced. Binary waveforms contain no information whatsoever, on the slopes of the corresponding analog waveforms, the presence of glitches, or other information which is often useful to a designer when evaluating the performance of a circuit. In our model therefore, we use three states (i.e., $p=3$) to describe the values of digital signals, which seems to be a fair compromise between the level of detail and the generation of accurate SLT estimates. Thus at any time $t \in [t_0 , t_f]$, the three-state (or ternary) digital signal $X_n(t)$ at node $n \in N$ is related to its analog counterpart $V_n(t)$ as follows :

$$X_n(t) = \begin{cases} 0 <==> 0.0 \leqslant V_n(t) \leqslant V_L \\ u <==> V_L < V_n(t) < V_H \\ 1 <==> V_H \leqslant V_n(t) \leqslant V_{DD} \end{cases} \quad (3.1)$$

where V_L and V_H are two thresholds chosen such that $0.0 < V_L < V_H < V_{DD}$. Here, **u** is an *intermediate* state between the steady low and high states **0** and **1** used to represent signals in transition, model slopes of changing analog waveforms, detect spurious glitches and hazards, etc. In our model, the intermediate state is not used as an *unknown* or *undefined* state as the **X** state in MOSSIM [19], but rather as an analog voltage between the two thresholds V_L and V_H and hence can never be considered as a steady state **0** or **1**. It is this interpretation of the third logic level that helps simplify the procedure for switch-level simulation as will be seen later in Chapter 4. The ternary state $X_n(t)$ of a node $n \in N$ at some time $t \in [t_0, t_f]$ will be denoted simply by X_n whenever there is no ambiguity in time.

The ternary algebra used to manipulate the discrete signals is an extension of the binary Boolean algebra. The ternary algebra is an algebra defined on the set $L = \{0, u, 1\}$, with three basic operations of AND ($\wedge$), OR ($\vee$), and INVERSE ($\neg$). For any $x, y \in L$, the operations of AND and OR are defined as follows:

x	y	x∨y	x∧y
0	0	0	0
0	u	u	0
0	1	1	0
u	0	u	0
u	u	u	u
u	1	1	u
1	0	1	0
1	u	1	u
1	1	1	1

and for any $x \in L$ its INVERSE $\neg x$ is defined as follows:

x	¬x
0	1
u	u
1	0

Clearly, **L** is closed under all three operations and the system $(\mathbf{L},\wedge,\vee,\neg)$ forms a *distributive* lattice [39] with zero element **0** and universal element **1**. Most of the properties of Boolean Algebra are preserved in the ternary algebra, except for the Law of Excluded Middle since $\mathbf{u}\vee\neg\mathbf{u} = \mathbf{u} \neq \mathbf{1}$ and $\mathbf{u}\wedge\neg\mathbf{u} = \mathbf{u} \neq \mathbf{0}$.

An NMOS transistor is modeled as a three-terminal device with a switch between the drain and source terminals and the signal at the gate controlling the status of the switch. In this dissertation we will only consider transistors whose drain and source nodes are different. In some technologies the drain and source regions of a transistor may correspond to the same net in the layout as a means of implementing a variable resistance. We shall, however, exclude such networks from our model. Associated with each device is a resistance which is a primarily a function of the ratio of the physical length to width (L/W) of the device when laid out. There are two types of NMOS transistors, namely, the *enhancement* type and the *depletion* type. Enhancement devices are characterized by positive device threshold voltages (i.e., VTO > 0) and behave as voltage-controlled switches. Depletion devices, on the other hand, have a negative VTO and are mainly used to implement pullup resistors. The gate and source nodes of a depletion device are usually shorted resulting in a two-terminal resistor. In the case of an enhancement NMOS device, the switch between drain and source nodes is *open*, *closed*, or in an *intermediate* state depending on whether the signal at the gate node is a **0**, **1**, or **u**, respectively. In the case of a depletion device, the switch is always *closed* irrespective of the signal at the gate node. Algebraically, each transistor $\mathbf{m}\in\mathbf{M}$ has a state $\mathbf{Z_m}\in\{\mathbf{0},\mathbf{u},\mathbf{1}\}$, where **0** indicates open, **u** indicates intermediate, and **1** indicates closed. Although the transistor states and the node states are different physical phenomena, the same mathematical objects will be used to represent both.

Mathematically, the NMOS network $\Omega(N,M)$ can be specified by giving a listing of nodes in N and transistors in M and the following functions:

NODTYP :	$N \rightarrow \{input, pullup, normal\}$	the node type
TRNTYP :	$M \rightarrow \{enhancement, depletion\}$	the transistor type
GATE :	$M \rightarrow N$	the gate node
SOURCE :	$M \rightarrow N$	the source node
DRAIN :	$M \rightarrow N$	the drain node
CAP :	$N \rightarrow [C_{min}, C_{max}]$	the node capacitance
RES :	$M \rightarrow [R_{min}, R_{max}]$	the transistor resistance

At any instant in time the *state* of the network is represented by $\Omega(X,Z)$ where $X=\{X_n : n \in N\}$ and $Z=\{Z_m : m \in M\}$ with $X_n, Z_m \in \{0,u,1\}$ representing the ternary states of node n and transistor m at that time instant. Under stable or steady-state conditions, the transistor states Z are functions of node states X. For example, consider a transistor m with gate node n, i.e., $GATE(m) = n$. If $TRNTYP(m) =$ *enhancement*, then $Z_m = X_n$ in the steady-state, otherwise if $TRNTYP(m) =$ *depletion*, then $Z_m = 1$ always.

3.2 Network Partitioning

In this section we describe the strategy and algorithms to partition the NMOS network $\Omega(N,M)$ into several transistor-disjoint subnetworks $\Omega_1, \Omega_2, \cdots, \Omega_s$, where each subnetwork or block Ω_i has a certain special configuration that would aid the simulation process. The partitioning strategy is basically to divide the set of enhancement transistors into two types, namely, driver transistors and pass transistors. The transistors of a particular type are then grouped together to constitute a subnetwork or a block if they have a common DC-path between their source and drain nodes (a notion that will be made precise in Section 3.2.2). The key to deciding whether an enhancement transistor is a driver transistor or a pass transistor is in the notion of an *external node* which will also be defined in Section 3.2.2. It is much easier to formally present our ideas and concepts if the NMOS network is viewed as an undirected graph; therefore we begin by reviewing some basic fundamentals from graph theory for the sake of completeness and also for the benefit of readers who are not familiar with the subject. An excellent reference on the fundamentals of graph theory is a book by Bondy and Murty [50].

3.2.1 Review of Graph Theory

An *undirected graph* H is an ordered triple $(V(H), E(H), \psi_H)$, consisting of a nonempty set $V(H)$ of *vertices*, a set $E(H)$ of *edges*, that is disjoint from $V(H)$, and an *incidence function* ψ_H which associates with each edge of H an unordered pair of (not necessarily distinct) vertices in H. If e is an edge and v and w are vertices such that $\psi_H(e) = <v,w>$, then e is said to *join* v and w, the vertices v and w are called the *ends* of e, and moreover, v and w are said to be *adjacent* in H. In this case we will usually refer to the edge e as simply $<v,w>$. The set of all vertices in H that are adjacent to the vertex v is denoted by $Adj_H(v)$. The two ends of an edge are incident with the edge and *vice versa*. If the two ends of an edge are the same, then the edge is called a *loop*, otherwise it is a *link*. The symbols $\nu(H)$ and $\epsilon(H)$ are used to denote the number of vertices and edges in graph H respectively, i.e., $\nu(H) = |V(H)|$ and $\epsilon(H) = |E(H)|$. When only one graph is under discussion it will be denoted by H, and we will use V, E, ν, and ϵ instead of $V(H)$, $E(H)$, $\nu(H)$, and $\epsilon(H)$. An undirected graph is usually represented pictorially on a plane by associating one point (or a dot) for each vertex and joining two points by a line (not necessarily straight) if the corresponding vertices are joined by an edge. As an example, consider a graph H with

$$V(H) = \{v_1, v_2, v_3, v_4, v_5\}$$
$$E(H) = \{e_1, e_2, e_3, e_4, e_5, e_6\}$$

and the incidence function defined by

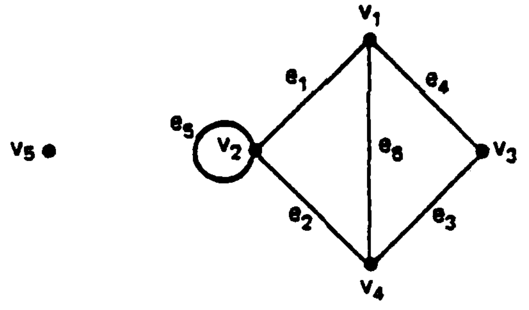
$$\psi_H(e_1) = <v_1,v_2>, \psi_H(e_2) = <v_2,v_4>, \psi_H(e_3) = <v_4,v_3>$$

$$\psi_H(e_4) = <v_3,v_1>, \psi_H(e_5) = <v_2,v_2>, \psi_H(e_6) = <v_1,v_4>$$

The pictorial representation of this graph is shown in Figure 3.1. The point representing the vertex v_5 is isolated in the picture since there are no edges incident on this vertex in this case. Hence vertices with no edges incident on them are called *isolated* vertices. Henceforth, we shall refer to a graph by its pictorial representation.

A graph F is a *subgraph* of H, written as $F \subseteq H$, if $V(F) \subseteq V(H)$, $E(F) \subseteq E(H)$, and ψ_F is a *restriction* of ψ_H to $E(F)$. If V' is a subset of V, then the subgraph of H whose vertex set is V' and whose edge set is the set of all edges of H that have both ends in V' is called the *induced subgraph* of H by V' and is denoted by $H[V']$. The induced subgraph $H[V \setminus V']$, denoted by $H - V'$, is the subgraph obtained from H by deleting the vertices from V' along with all their incident edges. If E' is a nonempty subset of E, then the subgraph of H *induced* by E' is the one with vertex set as the set of the ends of edges in E' and edge set E', and is denoted by $H[E']$. The subgraph obtained from H by deleting the edges in E' is denoted as $H - E'$. It must be pointed out that deleting vertices from a graph involves deleting incident edges also; however, deleting edges involves only the removal of edges while leaving the set of vertices intact, i.e., $V(H - E') = V(H)$. Similarly, $H + E'$ is a graph obtained from H by inserting a new set of edges E' which are disjoint from the old set of edges $E(H)$. Again, in this case, the ends of the edges in E' must necessarily be in V(H) since no new vertices are added. If F and H are two undirected graphs then their *union* is a graph, denoted by $F \cup H$, whose vertex set is $V(F) \cup V(H)$ and whose edge set is $E(F) \cup E(H)$. If F and H are *disjoint* graphs, then their union is usually denoted by $F + H$. The *degree* $d_H(v)$ of a vertex v in H is the number of edges incident on v, with each loop counting as two edges.

A *walk* in an undirected graph H is a finite, nonempty sequence $W = v_0 e_1 v_1 e_2 v_2 \cdots e_k v_k$ whose terms are alternately vertices and edges in H such that for each $1 \leq i \leq k$ the ends of e_i are v_{i-1} and v_i. In this case W is said to be a walk from v_0 to v_k, or a (v_0, v_k)-path in H, and the integer k is



FP–8526

Figure 3.1 : An undirected graph H

called the *length* of the walk. The vertices v_0 and v_k are called the *origin* and *terminus* of the walk respectively, while the vertices $v_1, v_2, \ldots, v_{k-1}$ are its *internal* vertices. If all the vertices in a walk are distinct then it is said to be a *path*. Usually, the subgraph of H whose vertices and edges are terms of a path is also referred to as a path. A walk is closed if it has positive length (i.e., $k > 0$) and its origin and terminus are the same. A closed walk whose origin and internal vertices are distinct is a *cycle;* just as with paths we sometimes use the term "cycle" to denote the graph corresponding to the cycle. Two vertices v and w of H are said to be *connected* if there exists a (v, w)-path in H. A subgraph F is a *component* of H if it is a maximal induced subgraph such that any two of its vertices are connected. If H has only one component then H is *connected*, otherwise, it is *disconnected.* The number of components of H is denoted by $\omega(H)$.

3.2.2 Driver and Pass Transistors

We begin this section by intuitively explaining the difference between driver and pass transistors through some examples. We then formally present our strategy to decide whether an enhancement device in the network is a driver transistor or a pass transistor and present an algorithm to achieve this in linear time. Finally, we show how the nodes and transistors in a network can be partitioned into various subnetworks or blocks, where each block could be one of three types, namely, input sources (SRC), a collection of driver transistors along with a depletion device (MFB), or a collection of pass transistors (PTB).

Before going into the formal definitions, we would like to provide the reader with some intuition on deciding between driver and pass transistors in a network. We define *external* nodes to be the set of nodes of "input" strength apart from the ground node together with those nodes of "normal" strength that are either gate nodes of enhancement transistors or are user-requested output nodes. Now consider a graph on the nodes of an NMOS network with an edge between the drain and source nodes of each enhancement transistor. Let us focus our attention on a pullup node, say $\mathbf{n_p}$ in the graph. For each such pullup node we consider the subnetwork composed of the depletion device connected to the pullup node and the transistors corresponding to all the paths between $\mathbf{n_p}$ and the ground node. If all the nodes corresponding to the internal vertices in each of these paths are of "normal" strength and if none of these nodes is an external node, we can then define the above subnetwork to be a multi-functional block (MFB) and all the enhancement transistors in it as driver transistors. Furthermore, each MFB must contain a unique pullup node. Consider an example of an NMOS network shown in Figure 3.2(a) and the corresponding graph in Figure 3.2(b). From the above definition, clearly $\mathbf{m_3}$ is a driver transistor. The transistors $\mathbf{m_1}$ and $\mathbf{m_2}$ are also drivers since the internal node, $\mathbf{n_4}$, is of "normal" strength and is not an external node. The node $\mathbf{n_3}$ is an external node by definition and hence $\mathbf{m_4}$ and $\mathbf{m_5}$ are not driver transistors. In fact $\mathbf{m_4}$, $\mathbf{m_5}$, and $\mathbf{m_6}$ are pass transistors. The MFB corresponding to the pullup node, $\mathbf{n_2}$, in this example, is the subnetwork consisting of the depletion transistor $\mathbf{m_8}$ along with the driver transistors $\mathbf{m_1}$, $\mathbf{m_2}$, and $\mathbf{m_3}$. The subnetwork composed of the pass transistors $\mathbf{m_4}$, $\mathbf{m_5}$, and $\mathbf{m_6}$ is called a pass transistor block (PTB). As far as switch-level simulation is concerned, an MFB can be treated as a switching network of driver transistors between the pullup node and the ground node. Note, by definition, the only node that is stronger than the pullup node in such a switching network is the ground node. Furthermore, one need not compute the waveforms at any of the internal nodes of the switching network. Therefore the signal at the pullup node of an MFB is computed using a simple technique using internal node eliminations, which will be discussed in Section 4.2.2 in Chapter 4. In fact, as we shall see in Chapter 4, the steady-state signal at the pullup node of an MFB is simply a Boolean function of the signals at the gate nodes of its driver transistors. For example, in the circuit of Figure 3.2(a) the signal at the node $\mathbf{n_2}$ is $\neg((\mathbf{x_1} \wedge \mathbf{x_2}) \vee \mathbf{x_3})$, where $\mathbf{x_1}$, $\mathbf{x_2}$, and $\mathbf{x_3}$ are the signals at the gate nodes of transistors $\mathbf{m_1}$, $\mathbf{m_2}$, and $\mathbf{m_3}$, respectively. In other words, an MFB can be considered to be a single output, multiple input logic gate. The switch-level simulation of a PTB, however, is a more difficult task since one needs to compute the signals at each node within the PTB. Therefore, the algorithms used to simulate a PTB are much more complex than the ones used to simulate an MFB, and these will be discussed in Section 4.2.3 in Chapter 4. Also, the techniques we will use to delay the

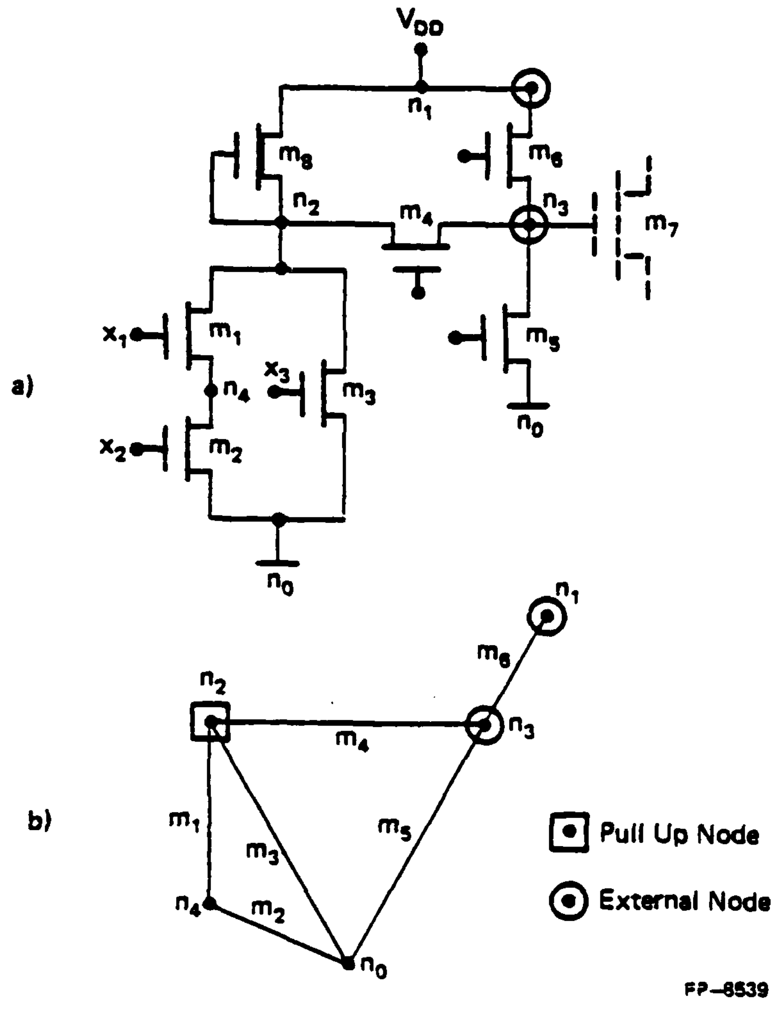


Figure 3.2(a): An NMOS circuit with external nodes
(b): The graph representing the circuit in part (a)

signal transitions at the pullup node of an MFB are different from those we will use for the nodes of a PTB. Hence we choose to differentiate between driver and pass transistors.

The above definition for a driver transistor is, in fact, only a sufficient condition satisfied by driver transistors as the following example demonstrates. Consider the NMOS network shown in Figure 3.3(a), and the corresponding graph in Figure 3.3(b). In this example $\mathbf{n}_4$ is an external node by definition. Let us suppose $\mathbf{n}_3$ is simply a node of "normal" strength and is not an external node. In this case the path consisting of $\mathbf{m}_4$ and $\mathbf{m}_5$ would satisfy the above definition of driver transistors and hence these transistors would be included in the MFB with pullup node $\mathbf{n}_2$. However, one needs to compute the signal at $\mathbf{n}_4$ since this determines the switching state of transistor $\mathbf{m}_7$, and in order to do this, we need to compute the signal at node $\mathbf{n}_3$ which, by the above definition, is an internal node of an MFB. We therefore have to modify our definition of a driver transistor. To this end, we introduce the concept of a pseudo-external node. A node of "normal" strength is said to be a *pseudo-external* node if it can be connected to an external node by a path that does not contain a pullup node or the ground node. Clearly, the signals at the pseudo-external nodes have to be computed in order to compute the signals at the external nodes of "normal" strength. Hence such a node cannot be an internal node of an MFB. We therefore modify our definition of driver transistors to be the transistors in those paths between a pullup node and ground that do not contain an external or pseudo-external node. Thus transistors $\mathbf{m}_4$ and $\mathbf{m}_5$ in the example in Figure 3.3(a) are not driver transistors. The above modification is, however, still inadequate to be a necessary condition to be satisfied by driver transistors as it does not agree with our intuition in the following example. Consider the NMOS network shown in Figure 3.4(a) and the corresponding graph in Figure 3.4(b). In this case we have two pullup nodes, namely, $\mathbf{n}_2$ and $\mathbf{n}_4$ and no external or pseudo-external nodes in the network. However, node $\mathbf{n}_3$ cannot be considered an internal node in either of the two MFB's since its signal can be influenced by either of the two pullup nodes. Hence the transistors $\mathbf{m}_4$, $\mathbf{m}_5$, and $\mathbf{m}_6$ must be treated as pass transistors in this example. To include this case in our definition we would have to treat the other pullup nodes in the network as external nodes while we are trying to determine the driver transistors between a particular

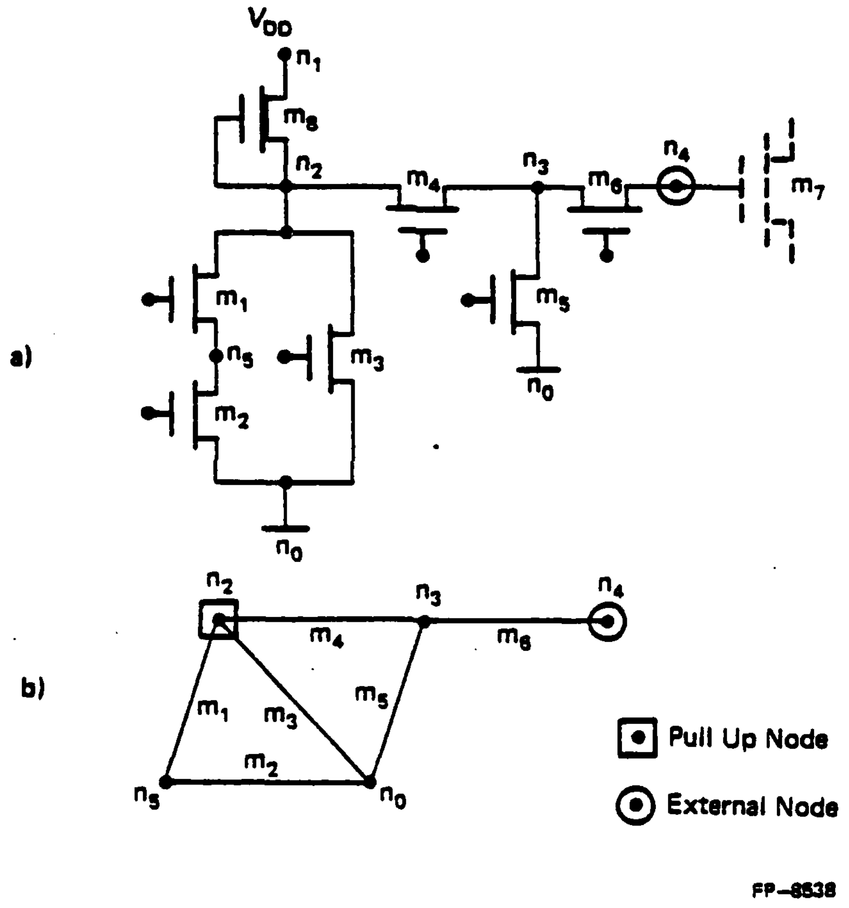


Figure 3.3(a): An NMOS circuit with pseudo-external nodes
(b): The graph representing the circuit in part (a)

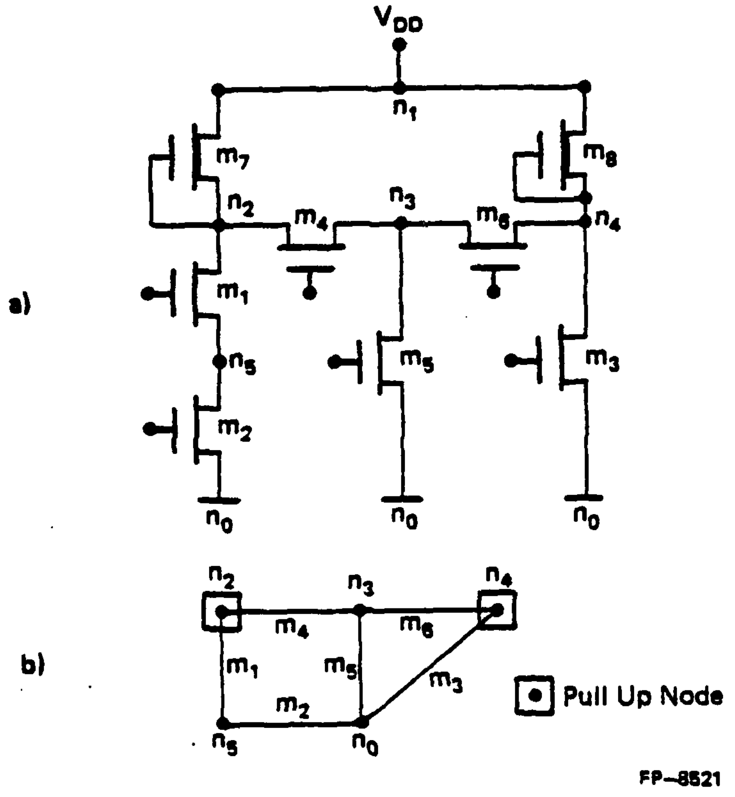


Figure 3.4(a): An NMOS circuit with no external or pseudo-external nodes
(b): The graph representing the circuit in part (a)

pullup node and ground. Thus, if we treat $\mathbf{n}_4$ as an external node, then $\mathbf{n}_3$ becomes pseudo-external and hence we get $\mathbf{m}_1$ and $\mathbf{m}_2$ as the only driver transistors in the MFB corresponding to $\mathbf{n}_2$. Similarly, if we treat $\mathbf{n}_2$ as an external node we get $\mathbf{m}_3$ as the only driver transistor in the MFB corresponding to $\mathbf{n}_4$.

The purpose of the above discussions was mainly to help the reader form some kind of an intuitive idea on the difference between a driver and a pass transistor. The above definitions were by no means precise and were not meant to be formal defi tions. We now develop a completely precise and formal definition of driver and pass transistors by introducing the notion of splitting a vertex in a graph. Consider an undirected graph $H(V, E, \psi_H)$ and vertex v in the graph of degree $k \geqslant 1$, i.e., $d_H(v) = k \geqslant 1$. The vertex v is said to be *loop-free* if there are no loops incident on v. The entire graph is loop-free if all its vertices are loop-free, i.e., it has no loops as edges. A graph is said to be *isolated* if all its vertices are isolated, i.e., it has no edges.

Definition 3.1 : Let v be a loop-free vertex of degree $k \geqslant 1$ in an undirected graph H. The *v-split graph* or the graph obtained on splitting v in H, is a graph obtained by splitting the vertex v into k new vertices $y_1, y_2, \cdots, y_k$, with each edge formerly joining the vertex v to w_i now joining y_i to w_i. We denote the v-split graph as $H \bullet v$. More formally we can define the v-split graph of H as

$$H \bullet v = ((H - v) \cup Y) + E_{yw} \tag{3.2}$$

where Y denotes an isolated graph on the k new vertices $\{y_1, y_2, \cdots, y_k\}$ and $E_{yw} = \{<w_i, y_i> : i = 1,2, \cdots, k\}$. Thus *splitting* a vertex creates a new graph with $k-1$ more vertices but with the same set of edges. This is in contrast to the notion of *adding* new edges to a graph in which case the vertex set is unaltered while new edges are added to the graph. It can easily be seen that if $k=1$, then splitting the vertex v does not alter the graph, i.e., $H \bullet v = H$ if $d_H(v)=1$. Similarly, the notion of vertex splitting can be extended to include the case $k=0$ by defining $H \bullet v = H$ if $d_H(v)=0$. If $V' = \{v_1, v_2, \cdots, v_q\}$ is a subset of loop-free vertices in H then the V'-split graph of H can be defined as follows :

$$H \bullet V' = (\cdots((H \bullet v_1)\bullet v_2)\cdots)\bullet v_q . \tag{3.3}$$

$H \bullet V'$ is well-defined since the order in which the vertices of V' are split does not matter. The end result is always the same. As an example consider the graph shown in Figure 3.5(a). The graph obtained by splitting the vertices v_1 and v_2 is shown in Figure 3.5(b).

An undirected graph H represents a network $\mathbf{\Omega}$ if there is a vertex in H corresponding to each node in the network and an edge between two vertices if the corresponding nodes are the source and drain nodes of some enhancement transistor. Let $\mathbf{M_E}$ and $\mathbf{M_D}$ denote the sets of enhancement and depletion transistors in the network respectively. We can then formally define a graph *representing* a network as follows :

Definition 3.2 : An undirected graph $H(V,E,\psi_H)$ is said to *represent* an NMOS network $\mathbf{\Omega(N,M)}$ if there exist bijections $\theta{:}V \to \mathbf{N}$ and $\phi{:}E \to \mathbf{M}_E$ such that $\psi_H(e) = <v,w>$ if and only if $\{\theta(v),\theta(w)\} = \{\mathbf{DRAIN}(\phi(e)),\mathbf{SOURCE}(\phi(e))\}$.

Theorem 3.1 : If H represents an NMOS network Ω, then H is a loop-free graph.

Proof : If e is a loop in H, then it follows from the above definition that $\theta(v) = \mathbf{DRAIN}(\phi(e)) = \mathbf{SOURCE}(\phi(e))$ where v is the vertex incident with the loop. But this is impossible since this means that the source and drain nodes of some transistor are tied together and we do not consider such networks in our model as explained in Section 3.1. Hence H has no loops. □

Let $\mathbf{N_I}$, $\mathbf{N_P}$ and $\mathbf{N_N}$ denote the sets of input, pullup, and normal nodes in the network respectively. It must be noted that, by definition, the ground node (GND) is treated as an input node. Also, by definition, $\mathbf{N_P} = \{\mathbf{n} \in \mathbf{N} : \mathbf{n}{=}\mathbf{SOURCE}(\mathbf{m})$ for some $\mathbf{m} \in \mathbf{M_D}\}$, i.e., every pullup node is a source node for a depletion device. The fact that there is a *unique* depletion device for each pullup node follows from the practices of conventional NMOS circuit designers. Let $\mathbf{N_O} \subseteq \mathbf{N_N}$ be the subset of normal nodes at which the user wishes to observe the output waveforms. Also, let $\mathbf{N_G} = \{\mathbf{n} \in \mathbf{N_N} : \mathbf{n}{=}\mathbf{GATE}(\mathbf{m})$ for some $\mathbf{m} \in \mathbf{M_E}\}$ denote the set of normal nodes that are gate nodes of enhancement transistors in the network. The nodes in $\mathbf{N_G}$ are also called *controlling* nodes [22,23], since these nodes control the state of

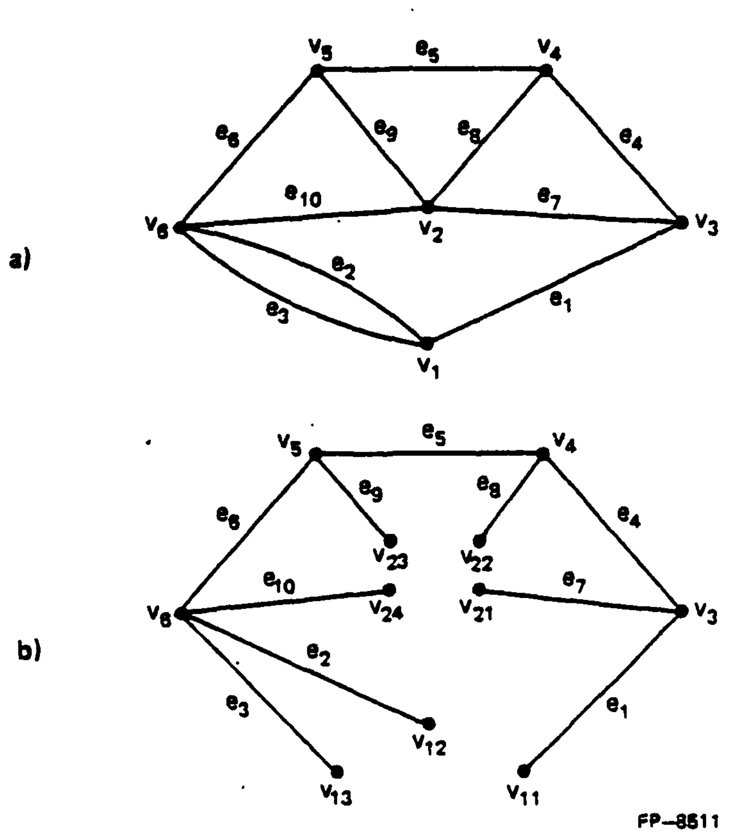


Figure 3.5(a): A loop-free graph H
(b): The graph obtained by splitting v_1 and v_2 in H

the transistor switches in the network.

Definition 3.3 : The set of *external* nodes is defined as

$$N_E = N_G \cup N_O \cup (N_I \backslash \{GND\}) \tag{3.4}$$

the union of three sets, namely, the set of normal nodes which are gate nodes of enhancement transistors, the set of user-requested normal output nodes, and the set of input nodes without the ground node.

Let V_I ,V_P ,V_E denote the sets of input, pullup, and external vertices in H corresponding to the input, pullup, and external nodes in the network. Let $H_I = H \bullet V_I$ be the graph obtained by splitting the input vertices in H. In other switch-level simulators [19,25,26], the transistors in the network are partitioned into several groups where each transistor group is simply a component of H_I. We would, however, like to further partition the transistors into driver and pass transistors. For this purpose we consider $H_{IP} = H_I \bullet V_P$ which is the graph obtained by splitting the pullup vertices in addition to the input vertices from H. The *strength* of a vertex v in H is the strength of the *corresponding node* $\theta(v)$ in the network Ω. Splitting a vertex retains the strength, i.e., the strength of the new vertices is the same as that of the original vertex before splitting. Also, splitting a vertex in a graph does not change the set of edges. Let C^H denote the subgraph of H induced by the edges in $E(C)$ for any component C of H_{IP}. Note that $E(H_{IP})=E(H)$ and hence C^H is well-defined. Consider a component C of H_{IP}. Then, clearly, C^H satisfies one and only one of the following conditions :

1. C^H contains at least one external vertex.

2(a). C^H contains no external vertices and no pullup vertices.

2(b). C^H contains no external vertices and exactly one pullup vertex.

2(c). C^H contains no external vertices and at least two pullup vertices.

Definition 3.4 : A component C of H_{IP} is said to be a *driver component* if C^H satisfies condition 2(b) given above.

Definition 3.5 : A component C of H_{IP} is said to be a *pass component* if C^H satisfies either condition 1, or 2(a), or 2(c) given above.

A component satisfying condition 2(a), i.e., having no external and no pullup vertices, is very rare since this represents a subnetwork containing only normal nodes, with the possibility of the ground node being included, while none of the normal nodes being gate nodes of enhancement devices or user-requested output nodes. Thus, this type of subnetwork neither interacts with other subnetworks nor is of any interest to the user. For the sake of completeness, however, we include this possibility also and label the component as a pass component.

The edges in a pass component are called *pass edges* while those in a driver component are called *driver edges*. It must be mentioned, once again, that splitting vertices in graphs does not alter the edge set of the original graph and so we have a partition of the edges of H into two sets, namely, the set of pass edges E_P and the set of driver edges E_D. We are now ready to define driver transistors and pass transistors in the NMOS network.

Definition 3.6 : An enhancement transistor $\mathbf{m}$ in the NMOS network $\mathbf{\Omega}$ is a *driver transistor* if $\phi^{-1}(\mathbf{m}) \in E_D$ and is a *pass transistor* if $\phi^{-1}(\mathbf{m}) \in E_P$, where $\phi^{-1}(\mathbf{m}) = e \iff \mathbf{m} = \phi(e)$.

We now form subgraphs with pass edges and driver edges and use these to define partitions of the NMOS network into special subnetworks. Let $H^1 = H_I - E_P$ be the graph obtained by removing all the pass edges from the V_I-split graph of H and let $H^2 = H_I - E_D$ be the graph obtained by removing all the driver edges from H_I. Hence H^1 contains only driver edges and H^2 contains only pass edges. The subgraph induced by the driver edges in a component of H^1 is called a *D-block* of H and the subgraph induced by the pass edges in a component of H^2 is called a *P-block* of H. Once again, we make no distinction between edges in H and the graphs obtained by splitting its vertices since all these graphs have the same set of edges. We thus have partitioned the graph H into several edge-disjoint subgraphs H_i ; $i = 1,2, \cdots, s$ where each H_i is either a D-block or a P-block. If H_i is a D-block then it must have a unique pullup vertex and no external vertices as a consequence of its definition. This fact and that in conventional NMOS designs a pullup node is connected to a unique depletion device allows us to make the following definition. The notion of an *induced subnetwork* is similar to that of induced subgraphs in a graph.

Definition 3.7 : A *multifunctional block* (MFB) is a subnetwork of Ω induced by the transistors corresponding to the edges of a D-block in H together with the depletion device connected to its pullup vertex (node). An MFB is a *proper MFB* if it also contains the ground node (which incidentally is **not** an external node and hence does not violate the above definition). In an improper MFB the pullup node is always stuck at **1** (i.e., maintains the value of V_{DD}) and hence we shall only consider proper MFB's which we will refer to simply as MFB. The pullup node is the *output node* of the MFB while the gate nodes of the driver transistors are its *input nodes*. The rest of the nodes, namely the drain and source nodes of the driver transistors, apart from the pullup node and the ground node, are the *internal nodes* of the MFB.

Definition 3.8 : A *pass transistor block* (PTB) is a subnetwork of Ω induced by the transistors corresponding to the edges of a P-block in H. Once again, the gate nodes of all the pass transistors are *input nodes* to the PTB. The rest of nodes, namely, the drain and source nodes of the pass transistors, could either be input nodes, or output nodes, or both (sometimes called *ioputs* for both input and output [7]), or none of the above depending upon the interaction of the PTB with the other blocks in the network. If a drain or source node of a pass transistor is of input strength it is an input node to the PTB, if it is of pullup strength it is an ioput (i.e., both input and output) node, and if it is a normal external node it is strictly an output node of the PTB.

The above definitions of driver and pass transistors completely agree with the author's intuition in all cases considered. For example, consider, once again, the circuit in Figure 3.4(a). The graph H_{IP} in this case, shown in Figure 3.6, has three components. The subgraph C_i^H in this example is the same as the component C_i itself, for each $i = 1,2,3$. The components, C_1 and C_3, clearly contain no external

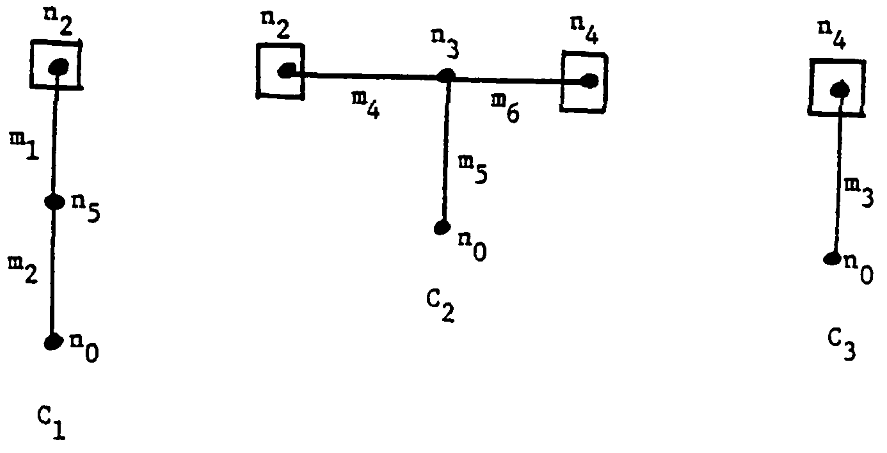


Figure 3.6 : The graph $H_{;P}$ for H in Figure 3.4(b)

vertices and exactly one pullup vertex and hence both are driver components according to Definition 3.4. The component C_2, however, contains no external vertices but has two pullup vertices and is hence a pass component according to Definition 3.5. A more detailed example is given in Section 3.4.

3.2.3 Partitioning Algorithm and Its Complexity

In this section we will discuss the algorithm to partition the NMOS network into MFB's and PTB's. Instead of dealing with the network $\Omega(N,M)$ we will be concerned with the graph $H(V,E)$ that represents the network. Obtaining the graph that represents the network merely involves altering the data structure that represents the networks to the one that represents a graph. Once we have identified the D-blocks and P-blocks in H then, clearly, identifying the MFB's and PTB's is trivial. Hence we shall mainly concentrate on the procedure PARTITION given below that partitions the graph H into several edge-disjoint subgraphs and labels each subgraph as either a D-block or a P-block.

Algorithm 3.1

Input : An undirected graph $H(V,E)$ with
V_I : the subset of input vertices and
V_P : the subset of pullup vertices.
V_E : the subset of external vertices.
Output: A set of edge-disjoint subgraphs $\Sigma = \{H_1, H_2, \cdots, H_s\}$ of H
and a function $BLK : \Sigma \rightarrow \{"D-block", "P-block"\}$.

```
procedure PARTITION (H)
begin
        $E_D \leftarrow \emptyset$
        $E_P \leftarrow \emptyset$
        $F^1 \leftarrow$ SPLIT($H, V_I$)
        $F^2 \leftarrow$ SPLIT($F^1, V_P$)
        $\Phi \leftarrow$ COMPONENT($F^2$)
        for each $C_j \in \Phi$ do
        begin
                $n_E \leftarrow |V_E \cap V(C_j^H)|$
                $n_P \leftarrow |V_P \cap V(C_j^H)|$
                if ($n_P = 1$ & $n_E = 0$) then
                        $E_D \leftarrow E_D \cup E(C_j)$
                else
                        $E_P \leftarrow E_P \cup E(C_j)$
                end if
        end
        $\Sigma_1 \leftarrow$ COMPONENT($F^1 - E_P$)
        for each $H_i \in \Sigma_1$ do
                $BLK(H_i) \leftarrow$ "$D$-block"
        $\Sigma_2 \leftarrow$ COMPONENT$(F^1 - E_D)$
        for each $H_i \in \Sigma_2$ do
                $BLK(H_i) \leftarrow$ "$P$-block"
        $\Sigma \leftarrow \Sigma_1 \cup \Sigma_2$
        return $(\Sigma, BLK)$
end
```

In the above algorithm we must ensure that any vertex that is split in a graph is, in fact, loop-free. This is indeed the case since from Theorem 3.1 we have that the entire graph H is loop-free. The *time complexity* of an algorithm to solve a problem is said to be $O(f(n))$ if the maximum amount of computation time (or number of computation steps) taken by the algorithm is at most $cf(n)$ over all inputs of size n, where c is some constant. The space complexity is similarly defined as an upper bound on the amount of space required by an algorithm to solve a problem. Two excellent references on the subject of time and space complexity of algorithms are Aho, Hopcroft, and Ullman [51] and Garey and Johnson [52]. In most graph algorithms the input size n is taken to be $|V|+|E|$, where $|V|$ and $|E|$ are the number of vertices and edges in the graph respectively. The time (or space) complexity is said to be *linear* if $f(n) = n$. The following theorem demonstrates that Algorithm 3.1, described above, is of linear time complexity.

Theorem 3.2 : The Algorithm 3.1, described above, correctly partitions the edges of H into driver edges and pass edges and its time complexity is $O(|V|+|E|)$ where V is the set of vertices and E is the set of edges in graph H.

Proof : The correctness of algorithm can easily be verified since it partitions the edges of H directly according to Definitions 3.4 and 3.5.

In order to discuss the time complexity, we will use the adjacency list [51] representation for graphs. This consists of a list of vertices and a linked list of edges. Each element of the vertex list contains the name (or label) of a vertex, say v, followed by a pointer to the location in the edge list of the first edge incident on it. Each element of the edge list contains the name of the vertex adjacent to v, an edge label, followed by the location of the next edge incident on v, and so on. A null-pointer (0) indicates that there are no more edges incident on v. This is repeated for each vertex in the graph. In case of undirected graphs each edge $<v,w>$ appears twice, once in the adjacency list of v and once in that of w. In this case there is a link established between the two locations. The total storage space required by this representation is $O(|V|+|E|)$.

The procedures SPLIT and COMPONENT are used several times in the above algorithm. If we can show that the time complexity of each of these two procedures is $O(|V|+|E|)$, then we are done with the proof since the rest of the computations in PARTITION can easily be verified to be of linear time complexity. Consider the operation of splitting a vertex v of degree k from a graph F. This merely involves altering the data structure to represent the new graph and can be easily shown to have a time complexity of $O(k)$. Thus SPLIT (F,V') is of time complexity $O(q)$ where $V' \subseteq V(F)$, and $q = \sum_{v \in V'} d_F(v)$. Since $q \leq |E(F)|$ we have that SPLIT (F,V') is of complexity $O(|E(F)|)$. We have therefore established that both SPLIT (H,V_I) and SPLIT (F^1,V_P) require $O(|E|)$ computation steps, where $E = E(H) = E(F^1)$, since the splitting of a vertex from a graph does not alter the edge sets.

The procedure COMPONENT (F) returns the various components in the graph F. A Boolean array of the vertices is maintained to mark a vertex as **new** or **old**, such that every time this array is altered, a pointer exists to indicate the location of the first vertex marked **new**. Initially all vertices of F are marked **new**. The procedure begins by starting from the first vertex marked **new** and using a depth-first search (DFS) algorithm [51] to determine all the vertices connected to the starting vertex via a path in F. These vertices induce a component and are all marked **old**. The whole process is repeated by starting from the first vertex that is now still marked **new** until all vertices are marked **old**. Each application of the DFS algorithm returns the list of vertices in a component of F in computation time linearly proportional to the number of edges in that component [51]. Thus if one does not have to scan the array to look for a starting vertex marked **new**, which is possible by maintaining the required pointer, the time-complexity of the entire procedure COMPONENT (F) is $O(|E(F)|)$. Since this procedure is used thrice in PARTITION (H) and each time on a graph with at most $|E(H)|$ edges we can conclude that the time-complexity of PARTITION (H) is $O(|V|+|E|)$. □

To model the voltage-source elements connected to the input nodes of the network we introduce a third type of block called input sources (SRC) consisting of only a node of input strength (and no transistors). This node is said to be the *output node* of the SRC. Thus, in this section, we have shown why and how we partition an NMOS network $\mathbf{\Omega(N,M)}$ into several subnetworks where each subnetwork is one of three types, namely, MFB, PTB, or SRC. We have also demonstrated an algorithm by which this partitioning can be achieved in computation time that is at most linearly proportional to the number of nodes and transistors in the network. We will use the same symbol $\mathbf{\Sigma}$ to denote the set of partitioned blocks in the network and henceforth we shall refer to the partitioned NMOS network as $\mathbf{\Omega(N,M,\Sigma)}$ along with a function $\mathbf{BLK : \Sigma \rightarrow \{"MFB","PTB","SRC"\}}$ indicating the type of block. Furthermore, $\mathbf{INP(\Omega_i)}$ and $\mathbf{OUT(\Omega_i)}$ will be used to denote the sets of input and output nodes of subnetwork $\mathbf{\Omega_i \in \Sigma}$.

3.3 Ordering of Partitioned Blocks for Processing

Let $\mathbf{\Omega(N,M,\Sigma)}$ be the NMOS network that has been partitioned into MFB's, PTB's, and SRC's. We will say that the above network has been *processed* if the ternary digital waveforms at each external node in the network are obtained. The network will be processed by processing each of its blocks in a certain order. A block is said to be processed, if given the ternary waveforms at the input nodes to the block, the waveforms at its output nodes are obtained. Thus, in order to process a block, the ternary waveforms at its input nodes must be known. Hence, we must process the blocks in a certain order so that this condition is always satisfied (whenever possible). In this section we will show when such an ordering exists, and if so, how one obtains it.

Definition 3.9 : For each node $\mathbf{n_i \in N}$ in the network, let $\mathbf{FOUT(n_i)}$ denote the *fanout list* for the node which is the set of blocks in Σ having n_i as an input node, and let $FIN(n_i)$ denote its *fanin list* which is the set of blocks with n_i as an output node. Thus,

$$FOUT(n_i) = \{\Omega_j : n_i \in INP(\Omega_j)\}$$

and

$$FIN(n_i) = \{\Omega_j : n_i \in OUT(\Omega_j)\}.$$

It must be noted that if n_i is an ioput node of a PTB then the PTB would appear both in its fanin and fanout lists. Furthermore, either list could be empty for certain nodes; for example, both lists would be empty for internal nodes of an MFB. Let (Ω_j,Ω_k,n_i) denote an ordered triple $\Sigma\times\Sigma\times N$. The ordered triple (Ω_j,Ω_k,n_i) is said to be an *I/O-triple* if $\Omega_j \in FIN(n_i)$ and $\Omega_k \in FOUT(n_i)$. If a node n_i is of pullup strength, i.e. $NODTYP(n_i)$ = **pullup**, and if it is an ioput node of a PTB, Ω_j, then the I/O-triple (Ω_j,Ω_j,n_i) is said to be a *nonadjacent* I/O-triple. An I/O-triple that is not a nonadjacent I/O-triple is said to be an *adjacent* I/O-triple. It must be emphasized that in the case n_i is a pullup node that is an ioput of a PTB, Ω_j, then the only nonadjacent I/O-triple in $FIN(n_i)\times FOUT(n_i)\times\{n_i\}$ is (Ω_j,Ω_j,n_i); the remaining I/O-triples are adjacent. In this case, if there is another node n_q that is not of pullup strength, i.e., $NODTYP(n_q) \neq$ **pullup**, such that Ω_j appears both in its fanin and fanout lists, then the I/O-triple (Ω_j,Ω_j,n_q) is indeed an adjacent I/O-triple. The fact that a pullup node can be an ioput of only one PTB follows from the definition of the PTB. Thus we have partitioned the set of I/O-triples into two disjoint categories, namely, the adjacent ones and the nonadjacent ones. Using the adjacent I/O-triples in the network, we will now introduce the notion of a good ordering in which the blocks of a network could be processed.

Definition 3.10 : A *sequential ordering* R on the blocks of a partitioned network $\Omega(N,M,\Sigma)$ is a 1-1 function $R : \Sigma \rightarrow \{1,2,\cdots,s\}$ where $s = |\Sigma|$. The sequential ordering R is said to be a *good ordering* for the network if $R(\Omega_j) < R(\Omega_k)$ for every **adjacent** I/O-triple (Ω_j,Ω_k,n_i) in the network. We exclude nonadjacent I/O-triples from our definition since in this case the equality will be forced (and so the inequality will never be satisfied) for any sequential ordering.

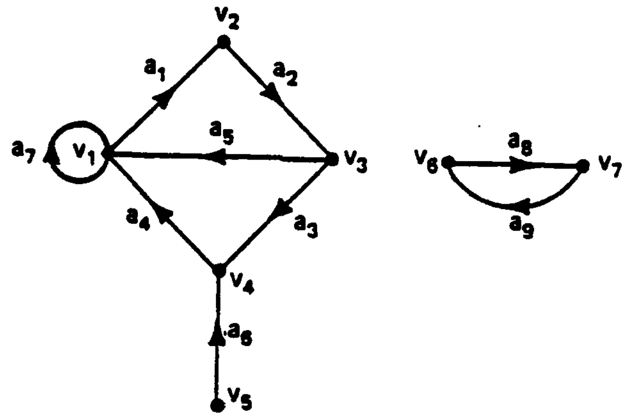
A good ordering, as defined above, is clearly a desirable ordering for processing the blocks in a network, since in this case, whenever a block is scheduled for processing, all the blocks in the fanin lists of each of its input nodes have been previously processed, thus, providing input signals to the this block. A good ordering, however, may not exist for some networks. As an example, consider an MFB $\mathbf{\Omega}_k$ in a network having its output node $\mathbf{n}_p$ connected back to one of its inputs. In this case, the network is said to have feedback, and the definition of a good ordering would be violated by the adjacent I/O-triple $(\mathbf{\Omega}_k,\mathbf{\Omega}_k,\mathbf{n}_p)$ for any sequential ordering. Hence, there is no good ordering for such a network. In the remaining part of this chapter we will show that a good ordering exists only for networks not having any kind of feedback, and proceed to handle the case of a network with feedback. The latter is important since most of the networks designed in present day NMOS technology do have feedback in some form or another, for example, flip-flops, ring oscillators, and most clocked sequential circuits in general. To this end, we will use the notion of a directed graph derived from a partitioned network. But first we review some basic concepts on directed graphs from Bondy and Murty [50], for the sake of readers not very familiar with the subject.

3.3.1 Directed Graphs

A *directed graph* G, often abbreviated as a *digraph*, is formally defined as an ordered triple $(V(G),A(G),\psi_G)$ consisting of a nonempty set $V(G)$ of *vertices*, a set, $A(G)$, of *arcs* that is disjoint from $V(G)$, and an *incidence function* ψ_G that associates with each arc of G an ordered pair of (not necessarily distinct) vertices of G. If a is an arc and v and w are vertices such that $\psi_G(a) = (v,w)$, then a is said to *join* v to w; v is the *tail* of a, and w is its *head* and the arc is usually referred to as simply (v,w). A digraph G' is a *subdigraph* of G if $V(G')\subseteq V(G)$,$A(G')\subseteq A(G)$ and the incidence function $\psi_{G'}$ is the restriction of ψ_G to $A(G')$. With each digraph G we can associate an undirected graph H on the same vertex set; corresponding to each arc of G there is an edge of H with the same ends. The graph H is said to be the *underlying graph* of G. The terminology and notation for subdigraphs are similar to those used for subgraphs. Just as graphs, digraphs also have a simple pictorial representation. A digraph is represented by a diagram of its underlying graph together with arrows on its edges, with each arrow pointing towards the head of the corresponding arc. Figure 3.7(a) shows a digraph G and its underlying graph H is shown in Figure 3.7(b).

A *directed walk* in G is a finite nonempty sequence $W = (v_0, a_1, v_1, \cdots, a_k, v_k)$, whose terms alternate between vertices and arcs, such that, for each $i = 1, 2, \cdots k$ the arc a_i has head v_{i-1} and tail v_i. Directed *paths* and *cycles* are similarly defined. The vertex v_0 is called the *origin* of the directed path while v_k is its *terminus*, and the rest of the vertices are called internal vertices. The integer k denotes the length of the directed path. Once again, the integer k denotes the length of the directed cycle. A directed cycle of length k is referred to as a *k-cycle*. If there exists an arc a in G such that $\psi_G(a) = (v, v)$, then a is a *loop* in G, and v, a, v is an example of a one-cycle in G. As with paths and cycles in undirected graphs, we will also refer to the subdigraphs induced by the arcs in a directed path or cycle as a directed path or cycle. Further, for convenience, we will drop the term "directed" and refer to directed paths and directed cycles simply as paths and cycles.

A path in G with origin v and terminus w is called a (v, w)-path. If there is a (v, w)-path in G then the vertex w is said to be *reachable* from v in G. This, however, does not imply that v is also reachable from w. Two vertices v and w are said to be *strongly connected* in G, denoted by $v \sim w$, if each is reachable from the other. Clearly, $\sim$ is an equivalence relation on $V(G)$ and it partitions $V(G)$ into nonempty subsets $V_1, V_2, \ldots, V_\mu$, such that if $v \in V_i$ and w is strongly connected to v in G, then w must also be $\in V_i$. The subdigraphs $G[V_1], G[V_2], \ldots, G[V_\mu]$ induced by the partition are called the *strongly connected components* of G. Note, by definition, a vertex v in G is always strongly connected to itself, i.e., $v \sim v$ since one can always choose a directed path of length 0 and reach v from itself and *vice versa*. Thus, $G[V_i]$ is a *trivial* strongly connected component if it contains only one vertex, i.e., $|V_i| = 1$. It can be easily shown that if $G[V_i]$ is a nontrivial strongly connected com-



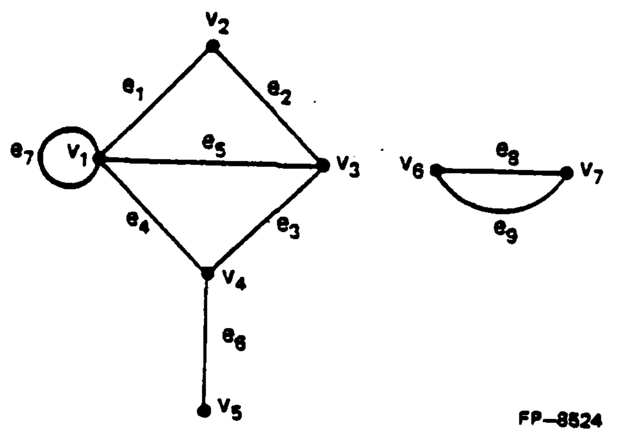


Figure 3.7(a): A digraph $G(V, A)$
(b): The underlying graph $H(V, E)$

ponent of G, i.e., $|V_i| \geqslant 2$, then it must necessarily contain a k-cycle with $k \geqslant 2$. Thus, presence of nontrivial strongly connected components in a digraph implies the presence of directed cycles. We use $\mu(G)$ to denote the number of strongly connected components in G. We say that G itself is strongly connected if $\mu(G) = 1$. Figure 3.8(a) shows a digraph which has three strongly connected components as shown in Figure 3.8(b). Hence the digraph is not strongly connected, while its underlying undirected graph is connected, since it has only one component. This clearly illustrates the difference between strongly connectedness in digraphs and connectedness in undirected graphs.

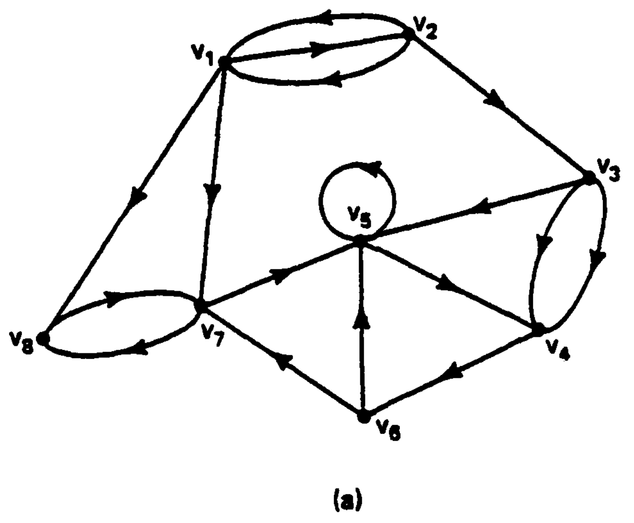
The *in-degree* $d_G^-(v)$ of a vertex v in G is the number of arcs having v as their head vertex. Similarly, the *out-degree* $d_G^+(v)$ of a vertex v is the number of arcs having v as their tail vertex. Just as with undirected graphs, we shall use the symbols $\nu(G)$ and $\epsilon(G)$ to denote the number of vertices and arcs in G. We shall also drop the letter G from most of the notations whenever possible.

3.3.2 Presence of Feedback and its Detection

Let $\mathbf{\Omega(N,M,\Sigma)}$ be a partitioned network. Let $\mathbf{\Upsilon}$ denote the set of I/O-triples of the network, i.e., $\mathbf{\Upsilon} = \bigcup_{\mathbf{n_i \in N}} \mathbf{FIN(n_i) \times FOUT(n_i) \times \{n_i\}}$, and let $\mathbf{\Upsilon_a}$ denote the set of adjacent I/O-triples in $\mathbf{\Upsilon}$.

Definition 3.11 : A directed graph $G(V, A, \psi_G)$ is said to be *derived* from an NMOS partitioned network $\mathbf{\Omega(N,M,\Sigma)}$ if there exist bijections $\theta: \Sigma \to V$ and $\phi: \Upsilon_a \to A$ such that the triple $v = (\Omega_j, \Omega_k, n_i) \in \Upsilon_a$ is an adjacent I/O-triple in the network if and only if $\psi_G(\phi(v)) = (\theta(\Omega_j), \theta(\Omega_k))$. Thus for every adjacent I/O-triple $v = (\Omega_j, \Omega_k, n_i)$ of the partitioned network, there is an arc $a = \phi(v)$ in the derived digraph G with tail vertex $\theta(\Omega_j)$ and head vertex $\theta(\Omega_k)$. The digraph G is said to be *acyclic* if it has no directed cycles. Just as with blocks in a network, we have sequential orderings on vertices of a digraph.

Definition 3.12 : A sequential ordering R on the vertices of a digraph G is said to be a *topological*



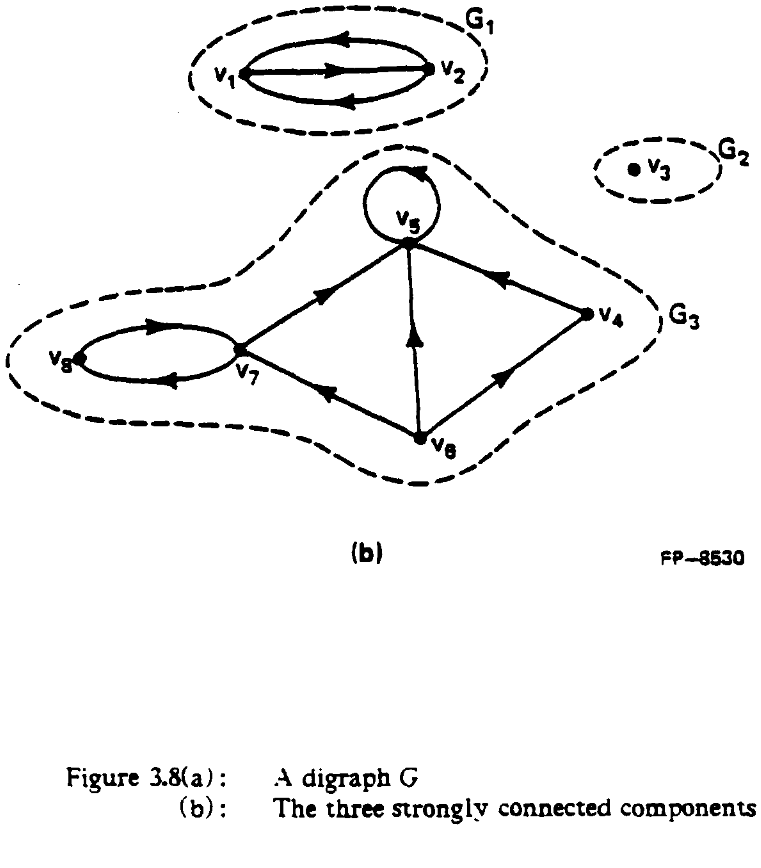


Figure 3.8(a): A digraph G
(b): The three strongly connected components of G

ordering if for every arc a with tail v and head w the strict inequality $R(v) < R(w)$ is satisfied.

Theorem 3.3 : If $\mathbf{\Omega}$ is a partitioned NMOS network and G is its derived digraph, then the following three conditions are equivalent :

(1) there is a good ordering on the blocks of $\mathbf{\Omega}$,

(2) G is acyclic, and

(3) there exists a topological ordering on the vertices of G.

Proof :

We shall first show that (1) => (2). Suppose $\mathbf{R}$ is a good ordering on the set of blocks $\mathbf{\Sigma}$ of the network. We will show that the derived digraph cannot contain a directed cycle. Suppose G has a directed k-cycle. If $k=1$, then there is a loop a with both ends at some vertex v. By Definition 3.11, there exists an adjacent I/O-triple $\phi^{-1}(a) = (\mathbf{\Omega}_j, \mathbf{\Omega}_j, \mathbf{n}_i)$, where $\mathbf{\Omega}_j = \theta^{-1}(v)$ in the network. This adjacent I/O-triple would clearly violate Definition 3.9 for $\mathbf{R}$. If $k > 1$ then let v denote the vertex in the k-cycle C whose corresponding block $\mathbf{\Omega}_k = \theta^{-1}(v)$ is ordered first by $\mathbf{R}$ among blocks corresponding to the other vertices in the cycle, i.e., $\mathbf{R}(\theta^{-1}(v)) \leqslant \mathbf{R}(\theta^{-1}(w))$ for all $w \in C$. Since $k > 1$ there is an arc a from w to v in the cycle (and hence in G) with $w \neq v$. But this would mean that there is an adjacent I/O-triple $(\Omega_j, \Omega_k, n_i)$ in the network where $\Omega_j = \theta^{-1}(w)$, thus leading to $\mathbf{R}(\Omega_j) < \mathbf{R}(\Omega_k)$ which contradicts the above choice of the vertex v. Hence the proof by contradiction.

The fact that (2) => (3) is a well-known result on digraphs and can be found in most standard textbooks on graph theory, such as [50]. Hence we will only outline this part of the proof. Suppose G is an acyclic digraph. Then there must be a vertex of in-degree 0 in G, since, if not, consider the longest directed path in G. If the first vertex of this path does not have in-degree 0, then either G has a cycle or a longer path. Hence pick a vertex, say v, whose in-degree is 0. The rest of the proof that G has a topological ordering is by induction on the number of vertices of G. The basis for induction is clearly satisfied for all digraphs containing only one vertex. Now suppose that all acyclic digraphs on less than ν vertices have a topological ordering. Let G have ν vertices. Then $G-\nu$ has no cycles and has $\nu-1$ vertices, and so must have a toplogical ordering, say R'. Now let R be an ordering of G such that $R(v)=1$ and $R(w)=R'(w)+1$ for all other vertices $w \neq v$ in G. Then clearly, R is a topological ordering for G.

The fact that (3) => (1) follows trivially from the definitions of good orderings of Σ, topological orderings of vertices in G and the fact that G is derived from $\mathbf{\Omega}$. □

We now introduce the concept of feedback in a partitioned network.

Definition 3.13 : A partitioned NMOS network $\mathbf{\Omega}$ is said to have *feedback* among its blocks if its derived digraph G has directed cycles. Thus $\mathbf{\Omega}$ is *feedback-free* if G is acyclic and is *internal feedback-free* if G has no directed loops. A block $\mathbf{\Omega}_j \in \Sigma$ is said to have *internal feedback* if the corresponding vertex $\theta(\mathbf{\Omega}_j)$ in G is incident with a directed loop.

It is clear that this definition of feedback in the networks conforms to the standard notion of feedback in circuits. It should also be clear now why we only considered adjacent I/O-triples while constructing the derived digraph. Had we chosen all I/O-triples to create arcs in G we would have directed loops corresponding to every nonadjacent I/O-triple. This would then amount to declaring that a network has internal feedback simply because it has a pullup node that is an ioput of a PTB, which does not conform to our usual conception of feedback in circuits. We are now ready to say that a network has a good ordering if and only if it is feedback-free. We state this result without proof below, since it easily follows from Theorem 3.3 and the definition of feedback-free networks.

Theorem 3.4 : A partitioned network $\mathbf{\Omega(N,M,\Sigma)}$ has a good ordering on its partitioned blocks if and only if it is feedback-free.

A good ordering of the blocks in a feedback-free network can easily be obtained by first placing the vertices of the derived digraph (which in this case will be acyclic, by definition) in a topological order and then placing the corresponding blocks of the network in the same order. If, however, the network has feedback (which is the more common case in the present-day NMOS designs), the derived digraph contains directed cycles and hence no topological (good) ordering exists on its vertices (blocks). In this case, therefore, one must detect the blocks in the network that are within feedback loops, treat these as special blocks and and place the rest of the blocks in a "good" ordering. We formalize these ideas below.

Definition 3.14 : If V_i is a set of vertices in a strongly connected component of G, then the corresponding set $\Sigma_i = \{\theta^{-1}(v) : v \in V_i\}$ of blocks in Σ is defined to be a *strongly connected component* (SCC) of the network. Thus we have a partition $\Sigma_1, \Sigma_2, \ldots, \Sigma_\mu$ of the blocks in Σ.

Let $V_1, V_2, \ldots, V_\mu$ denote the partition of the vertex set of the digraph G into strongly connected components. We define the *condensation* of G to be a digraph $\tilde{G}$ consisting of vertices $w_1, w_2, \ldots, w_\mu$ with an arc having head w_i and tail w_j if and only if $i \neq j$ and there is an arc in G with head $x \in V_i$ and tail $y \in V_j$. Consider the digraph G shown in Figure 3.8(a). Its condensation $\tilde{G}$, shown in Figure 3.9, is clearly acyclic. We will show that, for any digraph G, its condensation $\tilde{G}$ is acyclic and hence, from Theorem 3.3, it has a topological ordering, which corresponds to an ordering of the SCC's of Σ. To this end we need the following intermediate result.

Lemma : If C denotes a directed cycle in the digraph G then all its vertices must be within a strongly connected component of G.

Proof : (See [50]). Consider any two vertices, say, x and y in $V(C)$. Since C is a cycle, there is a directed path from x to y and also a return path from y to x in C. But C is a subdigraph of G and hence x is reachable from y and y is reachable from x in G. Therefore, by definition x and y must be in the same strongly connected component. □

Theorem 3.5 : The condensation $\tilde{G}$ of any digraph G must be acyclic.

Proof : (See [50,53]). By definition, $\tilde{G}$ has no directed loops, and so has no one-cycle. If $\tilde{C}$ is a k-cycle in $\tilde{G}$ with $k > 1$, then the vertices of G in the set $\bigcup_{w_j \in \tilde{C}} V_j$ must belong to a directed cycle and hence,

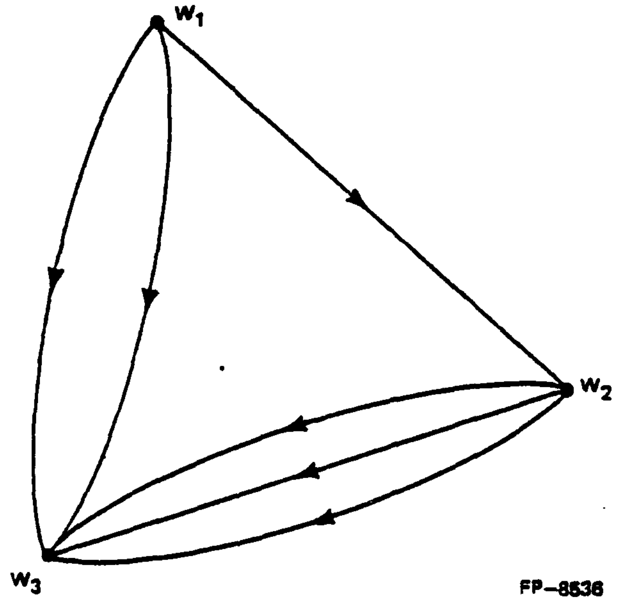


Figure 3.9 : The condensation of the digraph G in Figure 3.8(a)

from the above lemma, must all be in one strongly connected component, which is a contradiction. □

Our strategy to schedule the blocks of Σ for processing is to start by detecting the strongly connected components in the derived digraph G. We then obtain the condensation of G and proceed to find a topological ordering in $\tilde{G}$. This then corresponds to some ordering on the SCC's of Σ. The processing of the network Ω then begins by processing the SCC ordered first, followed by the one ordered second and so on. An SCC is said to be *simple* if it contains only one block of Σ and that block has no internal feedback. A simple SCC is processed by algorithms described in Chapter 4. If an SCC is not simple then the blocks within it are processed using special techniques described in Chapter 6. The algorithm presented below, well-known as Tarjan's algorithm [31], partitions the vertex set of any digraph into its strongly connected components. A vertex w is an *in-neighbor* of the vertex v in G if (w,v) is an arc of G and is an *out-neighbor* of v if (v,w) is an arc of G. We use $Adj_G^-(v)$ and $Adj_G^+(v)$ to denote the sets of in-neighbors and out-neighbors of the vertex v in G. In Tarjan's algorithm two integers $k[v]$ and $L[v]$ are computed for each vertex v in the digraph G, known as depth-first number and lowpoint [53] respectively. A digraph G is said to be a *rooted digraph* if it contains a vertex, say *root*, such that all vertices in G are reachable from *root*. In the case of the derived digraph G we try and make it a rooted digraph by inserting a new vertex called *root* and directing arcs from this new vertex to every vertex of in-degree 0 in the original G. In the original derived digraph G every vertex corresponding to an SRC block in the circuit must indeed have in-degree 0 and so the above notion is well-defined. Further, if there is a vertex that is not reachable from the new vertex *root* then it is also not reachable from any of the vertices corresponding to the SRC blocks in the network. This means that the input signals would never propagate to such blocks in the network and so they need not be simulated. Hence we are only interested in simulating those blocks in the circuit that correspond to vertices that are reachable from the vertex *root* in the above new digraph. We will still refer to the modified derived digraph as G itself and will assume that it is a rooted digraph.

Algorithm 3.2

Input : A rooted-digraph $G(V, A)$, with a special vertex *root*.
Output: A partition of $V - root$ into strongly-connected components
$V_1, V_2, \ldots, V_\mu$.

```
procedure SCC_DETECT (G)
begin
        $i \leftarrow 1$;
        for each $v \in V$ do
                MARK [v] ← "new";
        $\mu \leftarrow 0$;
        initialize STACK to empty;
        $v \leftarrow root$;
        DFS (v);
end

procedure DFS (v)
begin
        MARK [v] ← "old";
        $k[v] \leftarrow i$;
        $i \leftarrow i + 1$;
        $L[v] \leftarrow k[v]$;
        push v on STACK;
        for each vertex $w \in Adj_G^+(v)$ do
                begin
                        if MARK [w] = "new" then
                                DFS (w);
                                $L[v] \leftarrow \mathrm{MIN}(L[v], L[w])$;
                        else if $k[w] < k[v]$ and $w \in STACK$ then
                                $L[v] \leftarrow \mathrm{MIN}(k[w], L[v])$;
                        end if
                end
        if $L[v] = k[v]$ and $v \neq root$ then
                $\mu \leftarrow \mu + 1$;
                $V_\mu \leftarrow \emptyset$;
                repeat
                        pop x from STACK;
                        $V_\mu \leftarrow V_\mu \cup \{x\}$;
                until $x = v$;
        end if
end
```

The above algorithm terminates for finite digraphs and does so with linear time complexity and, furthermore, correctly partitions the vertices of the digraph into strongly connected components. This fact follows from the theorem below which we state without proof. Its proof can be found in several books on graph algorithms such as [31], [51], and [53].

Theorem 3.6 : The procedure SCC_DETECT (G) partitions the vertices of V into its strongly connected components correctly with time complexity of $O(\max(|V|,|A|))$.

We now describe an algorithm that creates a new digraph $\tilde{G}$ which is the condensation of the digraph G. We will use two procedures CREATE (x) and ADD_ARC (x,y) to create vertices and add arcs in the data structure that represents $\tilde{G}$. The data structure is the same as that for undirected graphs explained in Section 3.2.3, consisting of a list of vertices, and for each vertex an adjacency list, implemented as a linked list, of the out-neighbors of the vertex.

Algorithm 3.3

```
Input : A digraph $G(V,A)$ with a partition $V_1, V_2, \ldots, V_\mu$
        of its vertex set into strongly-connected components.
        A function SCCOMP : $V \to \{1,2,\ldots,\mu\}$ such that for any
        vertex $v \in V$, if i = SCCOMP($v$) then $v \in V_i$.
Output: The condensation digraph $\tilde{G}$ of $G$.

procedure CONDENSE (G)
begin
    $V(\tilde{G}) \leftarrow \emptyset$;
    $A(\tilde{G}) \leftarrow \emptyset$;
    for $i \leftarrow 1$ until $\mu$ do
        CREATE $(w_i)$;
    for each arc $(x,y) \in A(G)$ do
        begin
            $i \leftarrow$ SCCOMP($x$);
            $j \leftarrow$ SCCOMP($y$);
            if $i \neq j$ then
                ADD_ARC $(w_i, w_j)$;
            end if
        end
    return $\tilde{G}$;
end
```

The above algorithm clearly is of time complexity $O(\nu+\epsilon)$, where $\nu = |V(G)|$ and $\epsilon = |A(G)|$. We finally present an algorithm to produce a topological ordering on the vertices of the digraph $\tilde{G}$ which is known to be acyclic from Theorem 3.5, and hence, from Theorem 3.3 must have such an ordering. This algorithm uses a QUEUE to store some vertices. One could also use a STACK instead which would result in a different ordering. We use $d^-(w)$ and $Adj^+(w)$ to denote the in-degree and out-neighbors of vertex $w \in V(\tilde{G})$, i.e., we drop the subscript $\tilde{G}$ from the usual notations for convenience.

Algorithm 3.4 [51]

```
Input : An acyclic digraph $\tilde{G}(\tilde{V},\tilde{A})$, with $\mu=|\tilde{V}|$.
Output: A 1-1 function $R : \tilde{V} \rightarrow \{1,2,\ldots,\mu\}$ such that
        for every arc $(w_i,w_j)$ in $\tilde{A}$, $R(w_i)<R(w_j)$.

procedure TOP_ORDER ($\tilde{G}$)
begin
        $k \leftarrow 1$;
        for each vertex $w_i \in \tilde{V}$ do
                begin
                        $I[w_i] \leftarrow d^-(w_i)$;
                        if $d^-(w_i) = 0$ then
                                push $w_i$ into QUEUE;
                        end if
                end
        while QUEUE is not empty do
                begin
                        pop vertex $w_j$ from QUEUE;
                        $R[w_j] \leftarrow k$;
                        $k \leftarrow k+1$;
                        for each vertex $w_k \in Adj^+(w_j)$ do
                                begin
                                        $I[w_k] \leftarrow I[w_k]-1$;
                                        if $I[w_k] = 0$ then
                                                push $w_k$ into QUEUE;
                                        end if
                                end
                end
        return $R$;
end
```

The topological ordering R on $\tilde{G}$ provides us with an ordering **ORD** on the set of SCC's $\{\Sigma_1,\Sigma_2,\ldots,\Sigma_\mu\}$ such that $\mathbf{ORD}(\Sigma_i) = R(w_i)$, where w_i is the vertex of $\tilde{G}$ corresponding to the SCC Σ_i.

3.4 An Example to Illustrate Partitioning and Ordering

In this section we will consider the NMOS network shown in Figure 3.10 as an example to illustrate the partitioning and ordering algorithms described in the earlier sections of this chapter. This network consists of 17 nodes $\mathrm{N}=\{\mathbf{n}_0,\mathbf{n}_1,\ldots,\mathbf{n}_{16}\}$ and 20 transistors $\mathrm{M}=\{\mathbf{m}_1,\mathbf{m}_2,\ldots,\mathbf{m}_{20}\}$. The set $\mathrm{M_E}=\{\mathbf{m}_1,\mathbf{m}_2,\ldots,\mathbf{m}_{15}\}$ is the set of enhancement devices and $\mathrm{M_D}=\{\mathbf{m}_{16},\ldots,\mathbf{m}_{20}\}$ is the set of depletion devices. The set of nodes can be partitioned into three classes according to their strengths, namely, the nodes of "input" strength

$$\mathbf{N_I=\{n_0,n_1,n_2,n_3,n_4,n_5\}},$$

the nodes of "pullup" strength

$$\mathbf{N_P=\{n_6,n_7,n_8,n_9,n_{10}\}},$$

and the nodes of "normal" strength

$$\mathbf{N_N=\{n_{11},n_{12},n_{13},n_{14},n_{15},n_{16}\}}.$$

The node $\mathbf{n_0}$ is the ground node and $\mathbf{n_1}$ is the supply node to the network. The set of external nodes in this case is

$$\mathbf{N_E=\{n_2,n_3,n_4,n_5,n_{11},n_{12}\}}.$$

The graph H representing this network is shown in Figure 3.11. We only show the nonisolated vertices in the graph. Also, we refer to the vertices and edges of the graph as nodes and transistors in the network, respectively, for the sake of convenience, i.e., in this case the bijections θ and ϕ used in Definition 3.2 are both identity mappings. The graph H_I obtained by splitting the nodes of input strength from H is shown in Figure 3.12 and the graph H_{IP} is shown in Figure 3.13(a). The graph H_{IP} has seven components. The subgraphs of H induced by the edges in each of these components are shown in Figure 3.13(b). Among these subgraphs, the subgraph C_2^H has two external nodes while C_5^H has two pullup nodes, and so the corresponding components C_2 and C_5 are declared as pass components. The rest of the components can easily be verified to be driver components. Thus, the set of driver transistors is

$$\mathbf{M_D=\{m_1,m_2,m_3,m_4,m_5,m_6,m_7,m_8,m_9,m_{10}\}}$$

and the set of pass transistors in the network is

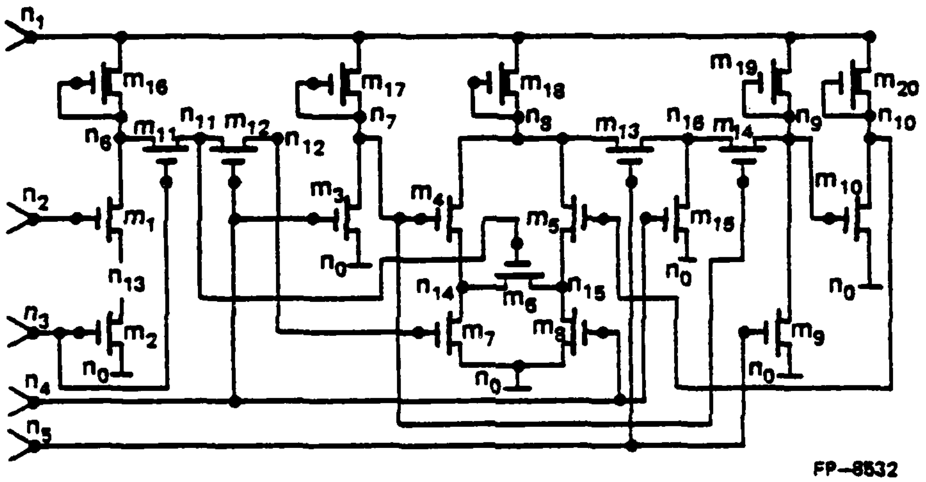


Figure 3.10 : An example of an NMOS network

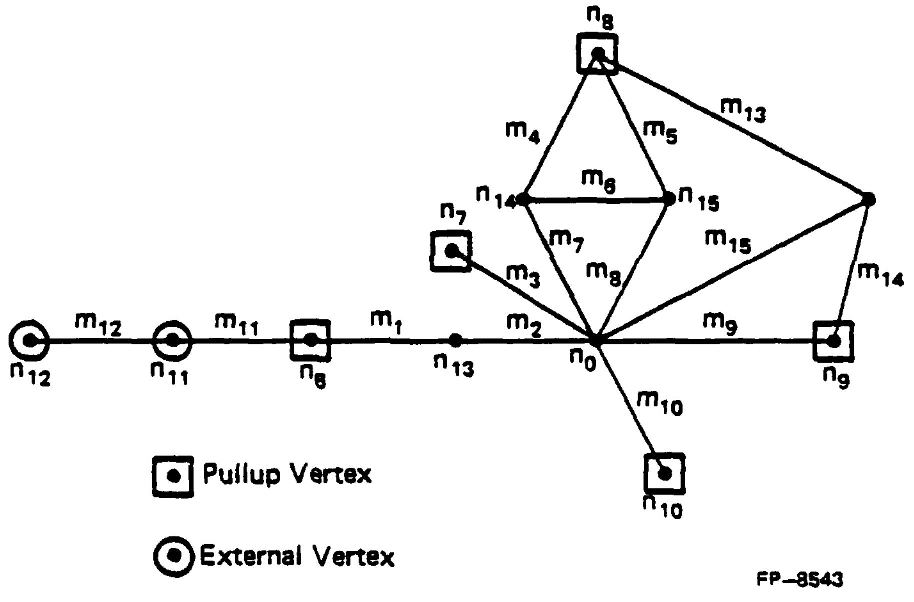


Figure 3.11 : The graph H representing the network in Figure 3.10

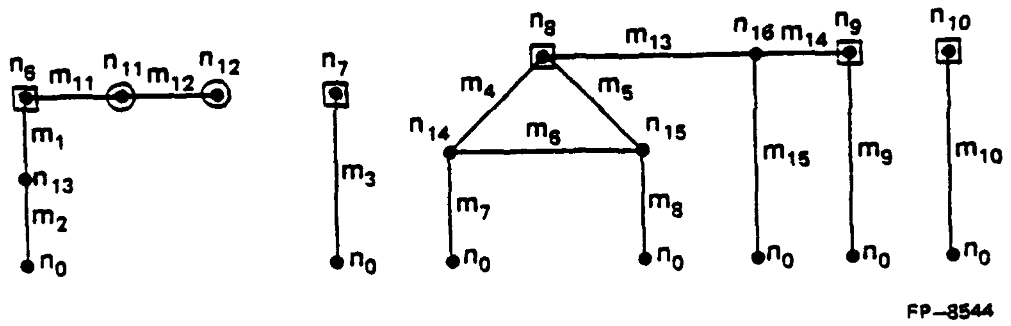


Figure 3.12 : The graph H_I for H in Figure 3.11

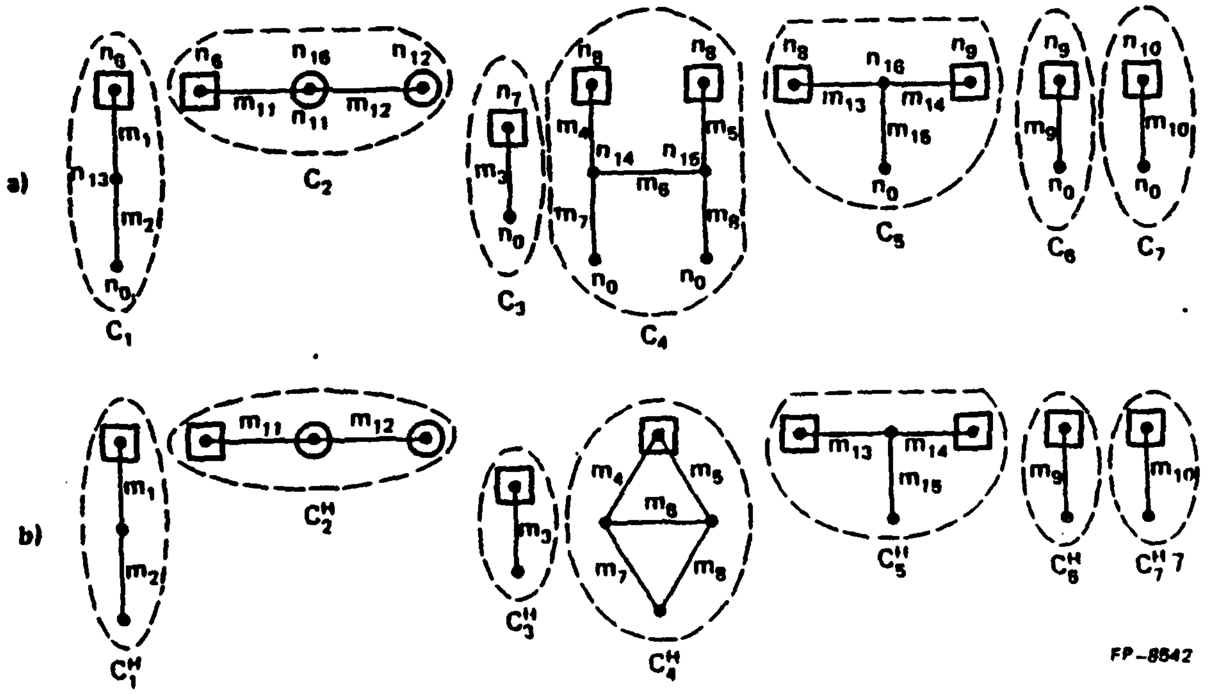


Figure 3.13(a): The graph H_I for H in Figure 3.11
(b): The corresponding edge-induced subgraphs of H

$M_P=\{m_{11},m_{12},m_{13},m_{14},m_{15}\}$.

It can easily be verified that the subgraphs C_2^H and C_5^H are the P-blocks of H and the rest of the subgraphs in Figure 3.13(b) are the D-blocks of H. Thus the transistors in the network can be now partitioned into seven blocks, two of which are PTB's and the remaining five are MFB's. We provide below a listing of the transistors in each block along with the set of its input and output nodes. In the case of an MFB the first transistor in its list is a depletion load device.

Block	Transistors	Input Nodes	Output Nodes
MFB_1	m_{16},m_1,m_2	n_2,n_3	n_6
MFB_2	m_{17},m_3	n_4	n_7
MFB_3	$m_{18},m_4,m_5,m_6,m_7,m_8$	$n_4,n_7,n_{10},n_{11},n_{12}$	n_8
MFB_4	m_{19},m_9	n_5	n_9
MFB_5	m_{20},m_{10}	n_9	n_{10}
PTB_1	m_{11},m_{12}	n_3,n_4,n_6	n_6,n_{11},n_{12}
PTB_2	m_{13},m_{14},m_{15}	n_4,n_5,n_7,n_8,n_9	n_8,n_9

In addition to the seven blocks given above, the network also has five SRC's which we list below along with the node of input strength in each of them.

Block	Output Node
SRC_1	n_1
SRC_2	n_2
SRC_3	n_3
SRC_4	n_4
SRC_5	n_5

We have thus partitioned the network into five SRC's, five MFB's and two PTB's. We are now ready to form the fanin and fanout lists for each node in the network. The table below gives these lists for each node which has both its fanin and fanout lists nonempty.

Node	Fanin List	Fanout List
n_2	SRC_2	MFB_1
n_3	SRC_3	MFB_1,PTB_1
n_4	SRC_4	MFB_2,MFB_3,PTB_1,PTB_2
n_5	SRC_5	MFB_4,PTB_2
n_6	MFB_1,PTB_1	PTB_1
n_7	MFB_2	MFB_3,PTB_2
n_8	MFB_3,PTB_2	PTB_2
n_9	MFB_4,PTB_2	PTB_2,MFB_5
n_{10}	MFB_5	MFB_3
n_{11}	PTB_1	MFB_3
n_{12}	PTB_1	MFB_3

From the above table, we see that, node n_6 is an ioput of PTB_1 and nodes n_8 and n_9 are ioputs of PTB_2. Hence, out of the 22 I/O-triples we get three nonadjacent triples, namely, (PTB_1,PTB_1,n_6), (PTB_2,PTB_2,n_7), and (PTB_2,PTB_2,n_8). The remaining 19 triples are adjacent I/O-triples. Given the adjacent I/O-triples, we can construct the derived digraph G as shown in Figure 3.14. This digraph contains ten vertices and 19 arcs. We also include a vertex "root" and join it to the five SRC vertices as shown in the same figure. Using Algorithm 3.2 on this digraph gives us ten strongly connected components which we list below.

SCC	Blocks
Σ_1	SRC_1
Σ_2	SRC_2
Σ_3	SRC_3
Σ_4	SRC_4
Σ_5	SRC_5
Σ_6	MFB_1
Σ_7	MFB_2
Σ_8	MFB_4
Σ_9	PTB_1
Σ_{10}	MFB_3,MFB_5,PTB_2

Thus, Σ_{10} is the only SCC that is not simple. Since G has no self loops, the network has no internal feedback. Note that had we considered the adjacent I/O-triples in constructing the derived digraph, we would get self loops. The network, however, has an SCC that is not simple, and hence, has feedback among MFB_3, PTB_2, and MFB_5. The condensation digraph $\tilde{G}$ is shown in Figure 3.15. From Algo-

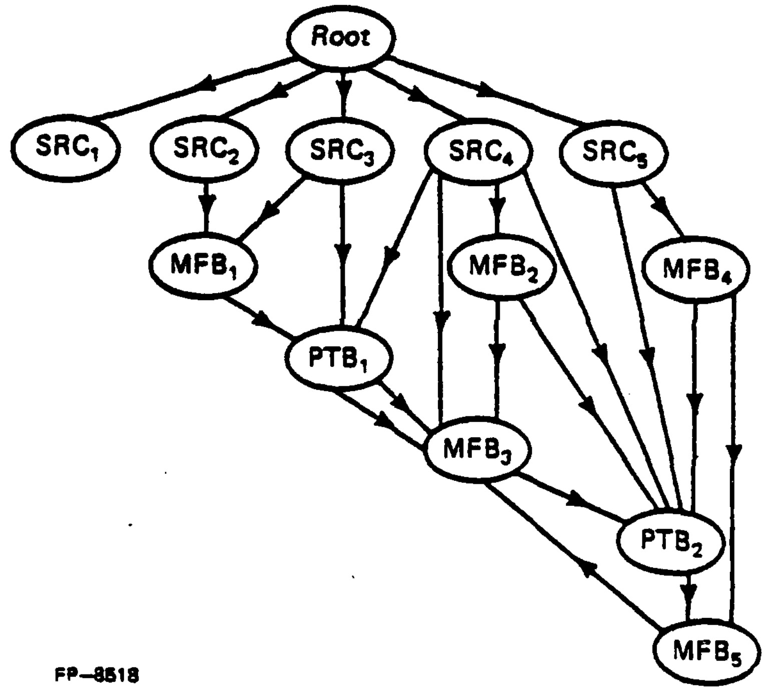
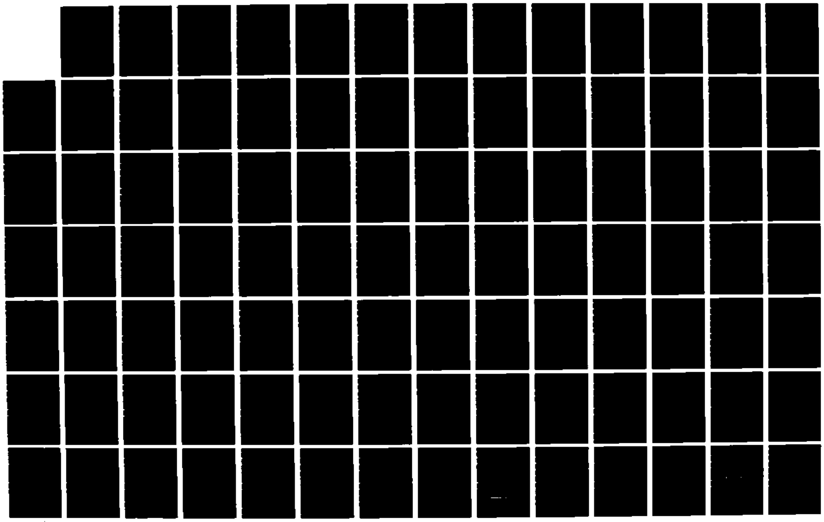
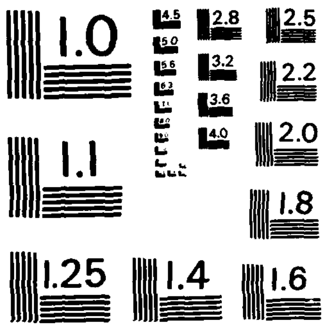


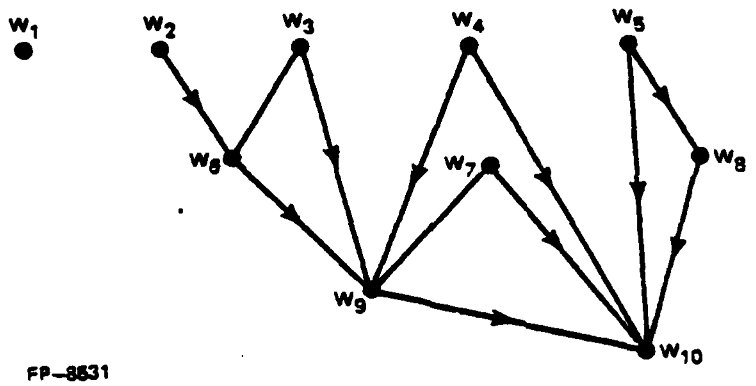
Figure 3.14 : The derived graph G

AD-A161 371 SWITCH-LEVEL TIMING SIMULATION OF MOS VLSI (METAL-OXIDE-SEMICONDUCTOR VER..(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB V B RAO JAN 85 R-1032 2/3

UNCLASSIFIED N00014-84-C-0149 F/G 9/5 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6



FP-8631

Figure 3.15 : The condensation graph $\tilde{G}$

rithm 3.4 on $\tilde{G}$ we get a topological ordering R such that $R(w_i)=i$; $i=1,2,\ldots,10$. This induces an ordering **ORD** on the SCC's of the network such that **ORD(Σ_i) = i ; i=1,2, . . . , 10**, in this case.

3.5 Conclusions

In this chapter we began by representing an NMOS network **Ω(N,M)** as a set of nodes **N** interconnected by a set of NMOS devices **M**. We then partitioned the set of enhancement transistors into driver transistors and pass transistors. Following this, the driver transistors are grouped together to form MFB's while pass transistors are grouped together to form PTB's. Another type of block, called SRC, is introduced to model the input voltage sources connected to the input nodes of the network. The partitioned network is represented as **Ω(N,M,Σ)**, where **Σ** is the set of partitioned blocks which could be MFB's, PTB's, or SRC's. We then introduced the concept of feedback among blocks in the network and showed that a good ordering for processing the various blocks is possible only for feedback-free networks. In case the network has feedback, the set of blocks is partitioned into its strongly connected components (SCC's). Finally, we came up with an ordering of the SCC's for processing. The partitioning and the ordering of the blocks have both been shown to take computation time that is linear in the number of circuit nodes and number of devices.

CHAPTER 4

SWITCH-LEVEL SIMULATION

Let $\Omega(N,M,\Sigma)$ be a partitioned NMOS network in which the set of blocks Σ has been further partitioned into its strongly connected components (SCC's) $\Sigma_1,\Sigma_2,\ldots,\Sigma_\mu$. Let **ORD** denote the ordering in which the SCC's have been scheduled for processing. If an SCC is simple, i.e., it consists of exactly one block (an MFB, PTB, or SRC) with no internal feedback, then it is simulated at the switch level by algorithms described in this chapter. By simulating or processing a block, we mean obtaining the ternary digital waveforms at the output(s) of the block given those at the inputs to the block over the entire time interval of interest. In case the SCC is not simple, a special event-driven windowing technique, to be described in Chapter 6, is used to simulate the various blocks within the SCC. This special technique partitions the entire time interval into several windows and uses the algorithms described in this chapter to simulate only the active blocks within each window.

4.1 Ternary Signals and Sequences of Transitions

Let $(L,\vee,\wedge,\neg)$ denote the ternary algebra on the set $L = \{0,u,1\}$ with binary operations OR ($\vee$), and AND ($\wedge$), and a unary operation INVERSE ($\neg$), as defined in Section 3.1. Let $[t_0,t_f]$ denote the time interval in which the network is to be simulated. At each time instant, the signal at a node in the network is assumed to occupy a ternary value from L, i.e., a **0**, **u**, or **1**, while this value might change with time. Such a signal is called a *ternary signal*. A node $n_i \in N$ is associated with a *ternary digital waveform*, denoted by X_i, which is a mapping $X_i : [t_0,t_f] \rightarrow L$, such that $X_i(t)$ is the ternary value of the signal at node n_i at time $t \in [t_0,t_f]$. A *transition* in a ternary signal is defined as a change in the ternary value of the signal taking place at a certain time instant. Thus, to completely specify a transition, we need to specify both the type of transition and the time at which it occurs. A *transition type* is an ordered pair (x,y) where $x,y \in L$ and $x \neq y$. There are six possible transition types, namely, $(0,u),(u,1),(1,u),(u,0),(0,1),(1,0)$. In accordance with the fact that a ternary digital waveform has a corresponding analog waveform, given by the inverse of the transformation in Equation 3.1, only the first four out of the six types of possible transition types are allowed. These *allowable transition types* are $(0,u),(u,1),(1,u)$ and $(u,0)$. We will consider only allowable transition types and, henceforth, drop the qualifier "allowable" whenever possible. For the sake of convenience in implementation, the entire simulation time interval $[t_0,t_f]$ is discretized by choosing a *minimum resolvable time* (MRT), denoted by h_{min}, so that a time point t can be represented by an integer k if $t \in [t_0+k^*h_{min}, t_0+(k+1)^*h_{min})$. Thus two different time points within this interval are considered indistinguishable and are represented by the same integer k and *vice versa*. If $K = (t_f-t_0)/h_{min}$, then the time at which a transition takes place within $[t_0,t_f]$ can be denoted by an integer $k \in [K]=\{0,1,2,\ldots,K\}$. The value of h_{min} is usually chosen to be very small, typically one or two orders of magnitude smaller than the rise or fall times of the analog signals. We can now represent a transition α as an ordered triple $(x,y,k) \in L \times L \times [K]$ where $(x,y) \in L \times L$ is the transition type and k denotes the time of its occurrence. Furthermore, x is the *initial* value of α and y is its *final* value.

Let $S = \alpha_1,\alpha_2,\ldots,\alpha_p$ be a *sequence* of transitions where each $\alpha_j=(x_j,y_j,k_j)$. The sequence S is said to be *chronological* if $k_1 < k_2 < \cdots k_p$. A chronological sequence is said to be *compatible*, in addition, if (x_j,y_j) is an allowable transition type for each $1 \leqslant j \leqslant p$ and $y_j=x_{j+1}$ for each $1 \leqslant j \leqslant p-1$. In a compatible sequence therefore, the final value of every term in the sequence is equal to the initial value of the succeeding term.

Let $t_k = t_0+k \times h_{min}$ and let $X(t):t \in [t_0,t_f]$ be a ternary signal waveform such that no more than one transition occurs in a time interval $[t_k,t_{k+1})$ for any integer k. We will call such a waveform a *proper* waveform. Clearly any ternary waveform will be a proper waveform if h_{min} is chosen as suggested above. Henceforth, we will assume that such an h_{min} has been chosen and that all ternary waveforms are indeed proper. In a proper waveform, therefore, if a transition occurs at a real time $t \in [t_k, t_{k+1})$ then any other transition must occur in some other interval disjoint from this. We use the notations t^- and t^+ to denote time points just before and just after the time t. We represent a proper waveform **X** by a sequence **S** of transitions as follows :

1. Initially, $S \leftarrow \emptyset$, and $k \leftarrow 0$.

2. If there is a $t \in [t_k, t_{k+1})$ such that $X(t^-) \neq X(t^+)$, then set $x \leftarrow X(t^-)$, $y \leftarrow X(^+)$, and append the transition $\alpha = (x,y,k)$ to **S**.

3. Set $k \leftarrow k+1$ and repeat step 2, until $k=K$.

If, however, a ternary signal is constant throughout the time interval $[t_0,t_f]$ then it does not undergo any transitions. We represent such a signal by a sequence consisting of a single transition of a suitable type taking place before t_0. Thus a waveform that is always **0** is represented by $(u,0,-1)$, and a constant **1** signal by $(u,1,-1)$, where the integer -1 represents all time points $t < t_0$. A constant **u** signal, though seldom occurring in practice, can also be represented either by $(0,u,-1)$ or $(1,u,-1)$. We will adopt the convention that -1 will be used to denote transition times in the case of constant signals only.

Let $S_a = \alpha_1, \alpha_2, \ldots, \alpha_p$ and $S_b = \beta_1, \beta_2, \ldots, \beta_q$ be two sequences of transitions, and let X_a and X_b denote their corresponding ternary digital waveforms respectively. The waveform X_c such that $X_c(t) = X_a(t) \vee X_b(t)$ for each $t \in [t_0,t_f]$ is called the "OR" of X_a, X_b. Similarly a waveform X_d is the "AND" of X_a, X_b is $X_d(t) = X_a(t) \wedge X_b(t)$ for each $t \in [t_0,t_f]$. The sequence S_c of transitions that represents X_c is denoted by $S_a \vee S_b$ and S_d that represents X_d is denoted by $S_a \wedge S_b$. Also, we can define the "INVERSE" of a sequence S_a representing the waveform X_a to be the sequence of transition, denoted by $\neg S_a$, representing a waveform X_e, where $X_e(t) = \neg X_a(t)$ for each $t \in [t_0,t_f]$. We therefore have two binary operations $\vee$ and $\wedge$ and one unary operation $\neg$ on sequences of transitions. As an illustration consider two (compatible) sequences of transitions:

S_a = (0,u,10), (u,1,70), (1,u,100), (u,1,110), (1,u,500), (u,0,600)

S_b = (1,u,200), (u,0,300), (0,u,700), (u,1,800).

The corresponding waveforms X_a and X_b are shown in Figure 4.1. The sequences obtained by performing the "OR" and "AND" operations on these two sequences are

$S_a \vee S_b$ = (1,u,500), (u,0,600), (0,u,700), (u,1,800)

and

$S_a \wedge S_b$ = (0,u,10), (u,1,70), (1,u,100), (u,1,110), (1,u,200), (u,0,300)

respectively, and their corresponding waveforms X_c and X_d are also shown in Figure 4.1. The sequence obtained by performing the "INVERSE" operation on S_a is

$\neg S_a$ = (1,u,10), (u,0,70), (0,u,100), (u,0,110), (0,u,500), (u,1,600)

which is obtained by simply inverting each ternary value in every term of the sequence.

Two sequences $S_a = \{\alpha_i\}_{i=1}^{p}$ and $S_b = \{\beta_i\}_{i=1}^{p}$ with the same number of terms, and where $\alpha_i=(x_i,y_i,k_i)$ and $\beta_i=(x'_i,y'_i,k'_i)$, are *type-equal* if $x_i=x'_i$ and $y_i=y'_i$ for each $1 \leqslant i \leqslant p$ and *time-equal* if $k_i=k'_i$ for each $1 \leqslant i \leqslant p$. The two sequences are *equal* if they are both type-equal and time-equal. It must be noted that two sequences can be compared for equality if and only if they have the same number of terms. For example, the two sequences

(0,u,100), (u,1,200), (1,u,300), (u,0,400)

and

(0,u,110), (u,1,190), (1,u,250), (u,0,360)

are type-equal but not time-equal, whereas

(0,u,100), (u,1,200), (1,u,300), (u,0,400)

and

(0,u,100), (u,1,200), (1,u,300), (u,1,400)

are time-equal but not type-equal.

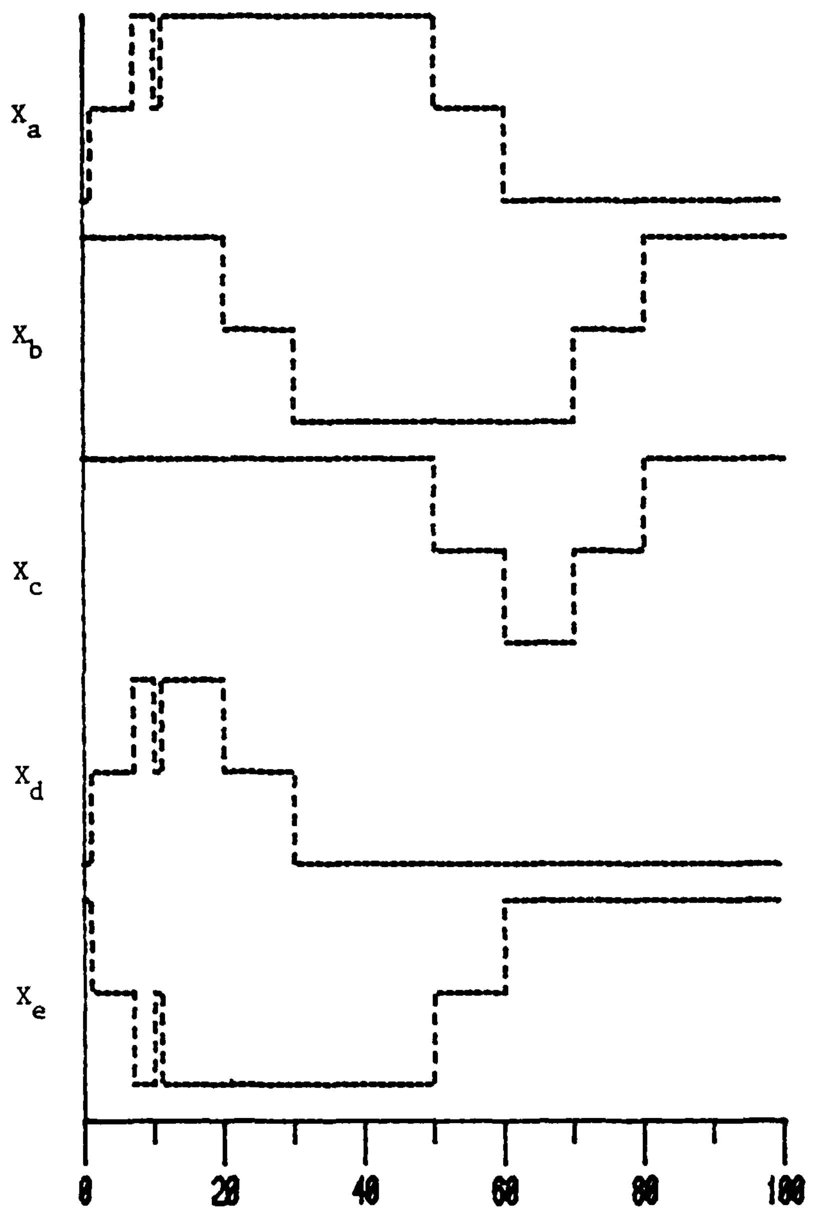


Figure 4.1 : Ternary digital waveforms

We now introduce notions of complete and partial pairs of transitions in a compatible sequence $S_a = \alpha_1, \alpha_2, \ldots, \alpha_p$, where $\alpha_i = (x_i, y_i, k_i)$.

Definition 4.1 : Two successive terms α_i and α_{i+1} in a compatible sequence with $x_i \in \{0,1\}$ are said to form a *complete pair* of transitions if $y_{i+1} = \neg x_i$ and a *partial pair* if $y_{i+1} = x_i$. It must be noted that since the sequence is compatible and the transition types are allowable, the above choice of x_i forces $y_i = x_{i+1} = u$.

For example, the pair (0,u,100), (u,1,200) is a complete pair while (0,u,100), (u,0,200) is a partial pair. A complete pair of transitions corresponds to an analog waveform crossing both the threshold limits, thereby completing the transition, whereas a partial pair represents a potential glitch or a hazard [29].

Definition 4.2 : A compatible sequence of transitions is said to be a *complete sequence* if it has no partial pairs. The *completion* of a compatible sequence is the **maximal** compatible subsequence consisting of only complete pairs of transitions. For example, the completion of the sequence

(0,u,100), (u,1,200), (1,u,250), (u,1,260), (1,u,300), (u,0,350), (0,u,400), (u,0,420)

is the sequence

(0,u,100), (u,1,200), (1,u,300), (u,0,350).

Given two compatible and complete sequences $S_a = \{(x_i, y_i, k_i)\}_{i=1}^{p}$ and $S_b = \{(x_i, y_i, k'_i)\}_{i=1}^{p}$ that are type-equal but not necessarily time-equal, we define a measure on the difference in transition times between the two sequences to be

$$\rho(S_a, S_b) = \max_{1 \leq i \leq p} \left| \frac{k_i - k'_i}{k_i} \right|. \tag{4.1}$$

It must be noted that the above measure is defined only for complete sequences which are type-equal. Two compatible sequences (not necessarily complete) are said to be *time-comparable* if their completions are type-equal. If S_a and S_b are two time-comparable sequences, i.e., their respective completions S'_a and

S'_b are type-equal, we then define an extended measure $\hat{\rho}$ to be

$$\hat{\rho}(S_a,S_b) = \rho(S'_a,S'_b). \tag{4.2}$$

As an example, consider two compatible sequences

$$S_a=(0,u,40),(u,0,50),(0,u,100),(u,1,200),(1,u,300),(u,0,400)$$

and

$$S_b=(0,u,110),(u,1,195),(1,u,260),(u,1,280),(1,u,330),(u,0,450)$$

whose completions are

$$S'_a=(0,u,100),(u,1,200),(1,u,300),(u,0,400)$$

and

$$S'_b=(0,u,110),(u,1,195),(1,u,330),(u,0,450)$$

respectively. Since S'_a and S'_b are type-equal we have S_a and S_b are time-comparable and, in this case, $\hat{\rho}(S_a,S_b)=\rho(S'_a,S'_b)=12.5\%$.

4.2 Switch-level Simulation of a Block

Let S_i denote the sequence of transitions, computed by switch-level simulation, and let V_i be the actual *analog* waveform at a node $n_i \in N$ in the network. We can obtain the three-state digital equivalent of V_i using the transformation in Equation 3.1. Let $\tilde{S}_i$ denote the sequence of transitions corresponding to this ternary digital equivalent. We define the aim of our switch-level timing simulator to compute S_i that is time-comparable to $\tilde{S}_i$, such that, $\hat{\rho}(\tilde{S}_i,S_i)<\epsilon$ where ϵ is a measure of the *accuracy* of the timing in the simulation. It must be noted that we are only interested in guaranteeing the timing in case of complete pairs of transitions and not for partial pairs. However, partial pairs will be included in the sequence to warn the user of a possible glitch or hazard at a node in the network. In this section we will discuss algorithms that will compute a so-called *zero-delay* sequence of transitions at the output nodes of a block in a simple SCC of the network. The complete pairs of transitions are then delayed by a delay operator to be discussed in the next chapter, followed by a filtering operation to produce sequences that represent realistic waveforms and improve the accuracy of the timing in case of partial pairs of transitions.

4.2.1 Simulation of an SRC

Let $\mathbf{\Omega}_c$ be an SRC with output node $\mathbf{n}_o$ in a partitioned NMOS network. Since an SRC, by definition, does not have any inputs, its corresponding vertex cannot be in any directed cycle in the derived digraph. Let V_o denote the analog waveform at node $\mathbf{n}_o$ during the time interval $[t_0,t_f]$. Since **NODTYP**($\mathbf{n}_o$) = **input** a description of V_o would be available in the input description of the network. Thus, simulating an SRC would simply amount to computing the sequence of transitions S_o directly from the analog waveform V_o as described below.

Algorithm 4.1

```
Input  : An SRC $\Omega_o$ with output node $n_o$,
         an analog waveform $V_o(t)$ for $t \in [t_0,t_f]$
         and two threshold voltages $V_L$ and $V_H$.
Output : A sequence of transitions $S_o$ representing the ternary
         equivalent of $V_o$.

procedure SRC_SIM ( $\Omega_c$ )
begin
        $S_o \leftarrow \emptyset$;
        $k \leftarrow 0$;
        $t_b \leftarrow t_0$;
        ind ← "constant";
        repeat
                $t_a \leftarrow t_b$;
                $t_b \leftarrow t_a + h_{min}$;
                $v_a \leftarrow V_o(t_a)$;
                $v_b \leftarrow V_o(t_b)$;
                $v_1 \leftarrow v_a - V_L$;
                $v_2 \leftarrow v_b - V_L$;
                $v_3 \leftarrow v_a - V_H$;
                $v_4 \leftarrow v_b - V_H$;
                if ($v_1 \leqslant 0$ & $v_2 > 0$) then
                        append (0,u,k) to $S_o$;
                        ind ← "variation";
                else if ($v_3 \leqslant 0$ & $v_4 > 0$) then
                        append (u,1,k) to $S_o$;
```

```
                    ind ← "variation";
            else if ($v_3 \geq 0$ & $v_4 < 0$) then
                    append (1,u,k) to $S_o$;
                    ind ← "variation";
            else if ($v_1 \geq 0$ & $v_2 < 0$) then
                    append (u,0,k) to $S_o$;
                    ind ← "variation";
            end if
            k ← k+1;
            $t_b \leftarrow t_b + h_{min}$;
        until $t_b > t_f$;
        if ind="constant" then
            if ($v_1 < 0$) then
                    append (u,0,−1) to $S_o$;
            else if ($v_1 \geq 0$ & $v_3 \leq 0$) then
                    append (0,u,−1) to $S_o$;
            else if ($v_3 > 0$) then
                    append (u,1,−1) to $S_o$;
            end if
        end if
end
```

In the above algorithm, the indicator **ind** is used to decide whether the analog waveform crossed any of the threshold limits. In case it does not, then the sequence is set to the appropriate transition occurring at integer time $k=-1$, i.e., at real time $t < t_0$. We now state the following theorem, the proof of which is fairly obvious, but it is an important result to be used in the later sections.

Theorem 4.1 : The sequence S_o computed by Algorithm 4.1 is a compatible sequence and represents the ternary equivalent of the analog waveform V_o.

4.2.2 Simulation of an MFB

Let Ω_f be an MFB that is to be simulated with n_o as its (unique) output node and INP(Ω_f) as its input nodes. For each input node n_i let S_i denote the sequence of transitions at that node, and let $z_i \in L$ denote the ternary value of the node signal at some time instant. Also, let S_o be the sequence of transitions to be computed and z_o denote an instantaneous value of the signal at node n_o. Let INTERN denote the set of internal nodes within the MFB. As mentioned earlier, an MFB can be viewed as a network of switches between the drain and source nodes of its driver transistors whose conduction states are controlled by the ternary signals at the gate terminals. Since an MFB has no external nodes, by definition, the sequences at its internal nodes need not be computed. The fundamental idea in conventional switch-level simulation is that the signal at a node can only be changed by a signal at a stronger node and can change the signals only at weaker nodes. In a proper MFB the only node stronger than the output node (which is a pullup) is the ground node whose signal is always at **0**. Hence, to compute z_o one only has to compute the state of conduction of the switches connecting the output node to the ground node. Thus, we can think of z_o to be a special kind of a ternary function of the input signals $\{z_i : n_i \in INP(\Omega_f)\}$. If the MFB is not proper, its output node signal is always at **1** irrespective of its input signals since no internal node can influence the value of this signal. This specialized structure of an MFB enables us to use a much simpler and more efficient algorithm for its simulation rather than using the more complex conventional switch-level algorithms such as the ones used in MOSSIM [19] or EXPRESS-II [25].

Before we describe the actual algorithms to simulate an MFB we digress briefly to study the properties of some ternary functions. Let p be a positive integer and let L^p denote the pth Cartesian power of L, i.e., L^p is the set of all ternary vectors $(z_1,z_2,\ldots,z_p)$ where $z_i \in L$ for each $i=1,2,\ldots,p$. A *p-variable ternary function* $f(z_1,z_2,\ldots,z_p)$ is a mapping $f:L^p \rightarrow L$.

Definition 4.3 : A *p-variable B-ternary function* is a p-variable ternary function which is either constantly **0** or **1**, or obtained from its arguments $z_1,z_2,\ldots,z_p$ by successive application of the algebraic operations of $\vee$, $\wedge$, or $\neg$. An example of a five-variable B-ternary function is

$$f(z_1,z_2,z_3,z_4,z_5) = \neg((z_1\wedge z_3)\vee(z_2\wedge z_4)\vee(z_1\wedge z_5\wedge z_4)\vee(z_2\wedge z_5\wedge z_3)). \qquad (4.3)$$

Associated with each variable z_i are two *literals*, namely, z_i and $\neg z_i$. Thus a p-variable B-ternary function can have at most **2p** literals. We will use the symbol w_j to denote a literal. The literal is said to be in its *normal form* if $w_j=z_i$ and in its *inverted form* if $w_j=\neg z_i$. A *product term* is a B-ternary function that is obtained by successively performing the $\wedge$ operation on its literals. For example, if

$w_{i1}, w_{i2}, \ldots, w_{ir}$ are r literals, then the corresponding product term is $w_{i1} \wedge w_{i2} \wedge \cdots \wedge w_{ir}$. Since the $\wedge$ operation on **L** is both associative and commutative, the order of the literals does not matter and hence the product term is well-defined. Thus any p-variable B-ternary function f consisting of q literals $w_1, w_2, \ldots, w_q$ where $q \leqslant 2p$ can be expressed as

$$f = g_1 \vee g_2 \vee \cdots \vee g_\nu \tag{4.4}$$

where g_j is a product term of a subset of the q literals, for each $j = 1, 2, \ldots, \nu$. This result follows directly from the corresponding well-known result that any switching function can be expressed in a *sum of products* form and can be found in any standard text book on switching theory, such as [54], since the relevant laws of conventional two-valued Boolean algebra used in its proof are easily extended to the ternary case. We will use the term *sum* as analogous to the $\vee$ operation and *product* as analogous to the $\wedge$ operation. Thus the result in the above Equation (4.4) can be simply stated as any B-ternary function can be expressed as a sum of products of its literals. Similarly it can also be shown that any B-ternary function f can also be expressed as a product of sums of its literals [54], i.e.,

$$f = h_1 \wedge h_2 \wedge \cdots \wedge h_\mu \tag{4.5}$$

where each h_j is a sum of a subset of literals.

We now introduce the notion of *zero-delay* through a block in a network. By this we mean that there are no delay elements present in the block and that at any instant of time the ternary value of the output signal can be determined from those at its input signals at the same instant of time. We say that an MFB with p-inputs $z_1, \ldots, z_p$ *realizes* a p-variable ternary function f if the ternary output signal z_o can be expressed as $z_o = f(z_1, z_2, \ldots, z_p)$ while the MFB is assumed to operate in the zero-delay mode.

The zero-delay value of z_o of an MFB of p driver transistors $m_1, m_2, \ldots, m_p$ can be computed as follows. Let z_i be the value of the signal at the gate node of m_i. Let H_f be the D-block in the graph representing the network corresponding to the MFB. Each edge of H_f has a *conduction state* associated with it which is equal to the ternary value of the gate signal of the corresponding driver transistor.

The *state* of a path **P** in the graph is defined as the product term of the states of the edges in the path. If there is a path between the output vertex and the ground vertex with state **1**, then, clearly the signal at the output node will be *forced* to have the value of the ground signal (which is stronger) which is a **0**, i.e., $z_o = 0$ in this case. If all paths between the output and the ground vertices have state **0** then $z_o = 1$. If there are no paths with state **1** and at least one path with state **u** then, in this case, $z_o = u$. Let $P_1, P_2, \ldots, P_s$ denote all the paths between the output vertex and ground vertex in the MFB and let g_i denote the state of path P_i for each $i=1,2,\ldots,s$. Clearly, each g_i is a product term of the ternary signals at the gate nodes of the transistors corresponding to the edges in path P_i. From the above simple arguments it is clear that the ternary value of the output signal can be obtained by summing all the g_i's and inverting the resulting sum, i.e.,

$$z_o = \neg(g_1 \vee g_2 \vee \cdots \vee g_s). \tag{4.6}$$

Thus z_o is a p-variable B-ternary function of its arguments $z_1, z_2, \ldots, z_p$, which are the signals at the input nodes to the MFB. It must be noted that in each product term g_i above, no literal appears in its inverted form, i.e., all literals appear in their normal form. Such a product term will be referred to as a *normal* product term. We now present some interesting results in the synthesis of networks composed of MFB's to realize any combinatorial switching function.

Theorem 4.2 : Any p-variable B-ternary function $f(z_1, z_2, \ldots, z_p)$ that can be expressed as the inversion of a sum of normal product terms, as in Equation (4.6), can be realized by a single MFB with **p** input nodes.

Proof : We begin constructing an MFB with a supply node (connected to a power supply V_{DD}), a ground node, and **p** input nodes $n_1, n_2, \ldots, n_p$ such that the ternary signal at n_i is z_i for each $i=1,2,\ldots,p$. We then include a depletion transistor with drain node connected to the supply and source and gate nodes tied together at a node n_o which we will call the output node of the MFB. We now introduce the notion of a series chain of transistors which will be made use of in the construction of the driver block of the MFB.

A set of δ transistors is said to form a δ *series chain* if the subgraph induced by the edges corresponding to these transistors in the graph representing the network is a path of length δ. The nodes corresponding to the end vertices of the path will be called the *end nodes* of the chain. An example of a 4 series chain is shown in Figure 4.2.

Let the B-ternary function be expressed in the required form as

$$f = \neg(g_1 \vee g_2 \vee \cdots \vee g_s)$$

where g_j is a product term of δ_j normal literals for each $j=1,2,\ldots,s$. Corresponding to each g_j we insert a δ_j series chain of enhancement transistors with one end node as n_o and the other as the ground node. Each transistor in a series chain is associated with a normal literal appearing in the corresponding product term. The gate node of a transistor corresponding to a literal z_i is connected to the input node n_i. We thus have s series chains of enhancement transistors connected in parallel across the output node n_o and the ground node. It can easily be verified that such a configuration would correspond to a D-block in a graph representing the network and hence the subnetwork we have constructed constitutes an MFB. Furthermore this MFB would realize the required B-ternary function. □

The simplest proper MFB is an *inverter* consisting of exactly one driver transistor as shown in Figure 4.3(a). Even simpler than this is an MFB with no driver transistors, in which case, the ternary signal at the output is always at **1** (which incidentally is a B-ternary function by definition). Figures 4.3(b) and (c) show two-input NAND and NOR gates respectively. As an illustration of the technique used in the proof of the above theorem we consider the five-variable B-ternary function given in Equation (4.3). An MFB with ten driver transistors realizing this function is shown in Figure 4.4. This consists of four series-chains connected in parallel across the output of the MFB and the ground node consisting of two, two, three, and three driver transistors respectively. One measure of the complexity of an MFB could be chosen as the number of driver transistors in the MFB. It must be noted that Theorem 4.2 does not say anything about the uniqueness of the MFB realization. In fact there could be several MFB's realizing the same B-ternary function. Figure 4.5 shows another MFB realizing the same

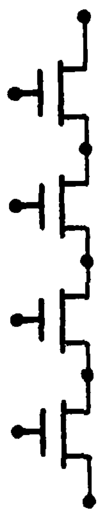


Figure 4.2 : A 4 series chain of transistors

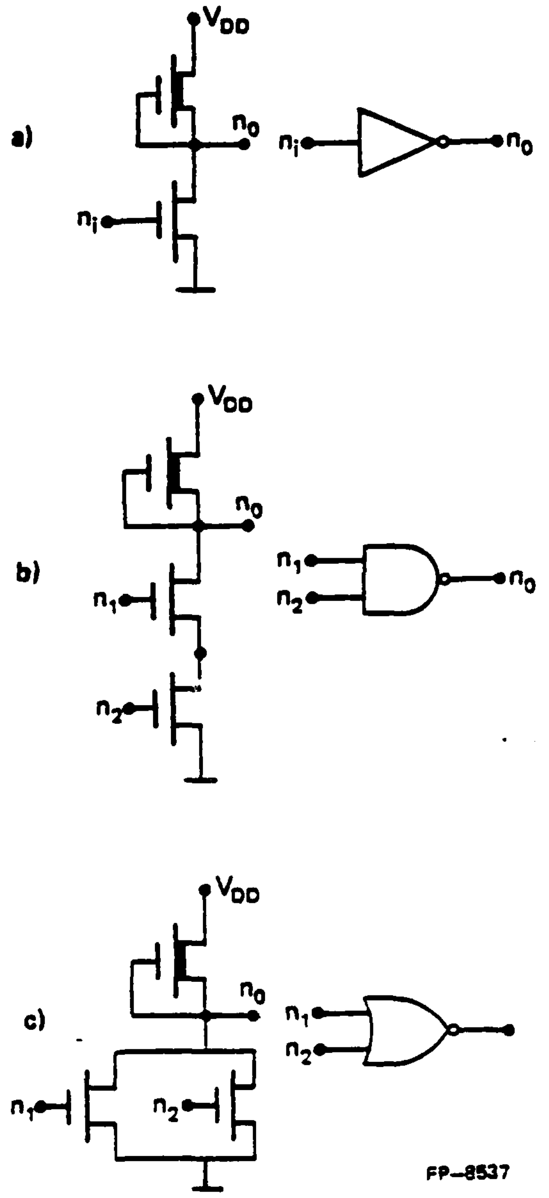


Figure 4.3(a) : A simple inverter
(b) : A two-input NAND gate
(c) : A two-input NOR gate

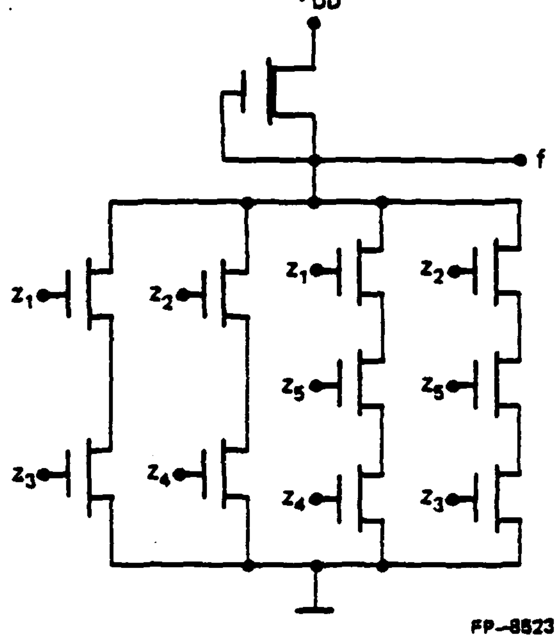


Figure 4.4 : An MFB realization of f in equation (4.3)

B-ternary function as in Equation (4.3). This MFB, in fact, has only five driver transistors and is an example of using *bridged configurations* to reduce the number of transistors in a series-parallel realization.

The B-ternary functions considered by Theorem 4.2 are of a rather restricted nature. If we relax the requirement that only a single MFB be used in the realization, we can consider a subnetwork of MFB's realizing any general B-ternary function. The *number of levels* in a subnetwork composed of blocks can be defined as the length of the longest directed path in the corresponding subdigraph within the digraph derived from the partitioned network. The following result shows that any B-ternary function can be realized by a two-level subnetwork of MFB's.

Theorem 4.3 : Let $f(z_1,z_2,\ldots,z_p)$ be any p-variable B-ternary function. Then f can be realized by a at most two-level subnetwork consisting of $\beta+1$ MFB's with β of these MFB's being simple inverters, where $\beta \leqslant p$.

Proof : Let $f = h_1 \wedge h_2 \wedge \cdots \wedge h_s$ be the product of sums expression of the B-ternary function f. Since $\neg(\neg(f)) = f$ we can rewrite the function as

$$f = \neg(g_1 \vee g_2 \vee \cdots \vee g_s)$$

where $g_j = \neg(h_j)$ can be easily shown to be a product term for each $j=1,2,\ldots,s$, through simple ternary algebraic manipulations. The rest of the proof is very similar to that of Theorem 4.2 in that an MFB is constructed with a series chain for each product term and the number of transistors in a series chain equal to the number of literals (both normal and inverted) in the corresponding product term. If all literals appearing in the product terms are in their normal form, then from Theorem 4.2, a one-level realization can be obtained. If a literal $\neg z_i$ appears in a product term g_j in its inverted form, the gate node of the corresponding transistor in the series chain is connected to the output of an inverter whose input is connected to node n_i. Clearly, the number of inverters needed is equal to the number of literals appearing in their inverted form in a product term which is at most p. It can be easily verified that the output of the MFB apart from the inverters in the subnetwork is the required B-ternary

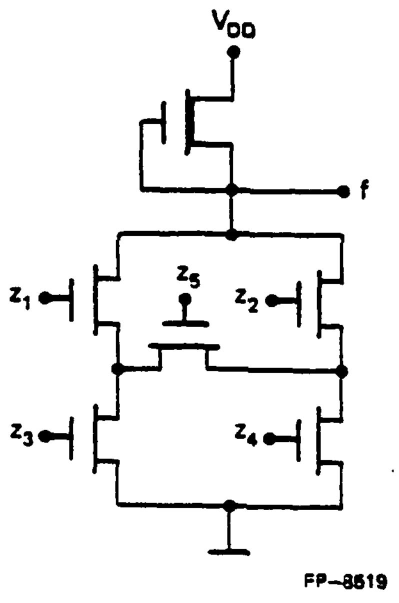


Figure 4.5 : Another MFB realization of f in Equation (4.3)

function of the signals at the input nodes of the subnetwork. Furthermore, this is a two-level subnetwork. □

As an illustration consider the following three-variable B-ternary function

$$f_1 = (z_1 \wedge z_2) \vee ((\neg z_1 \vee z_2) \wedge z_3)$$

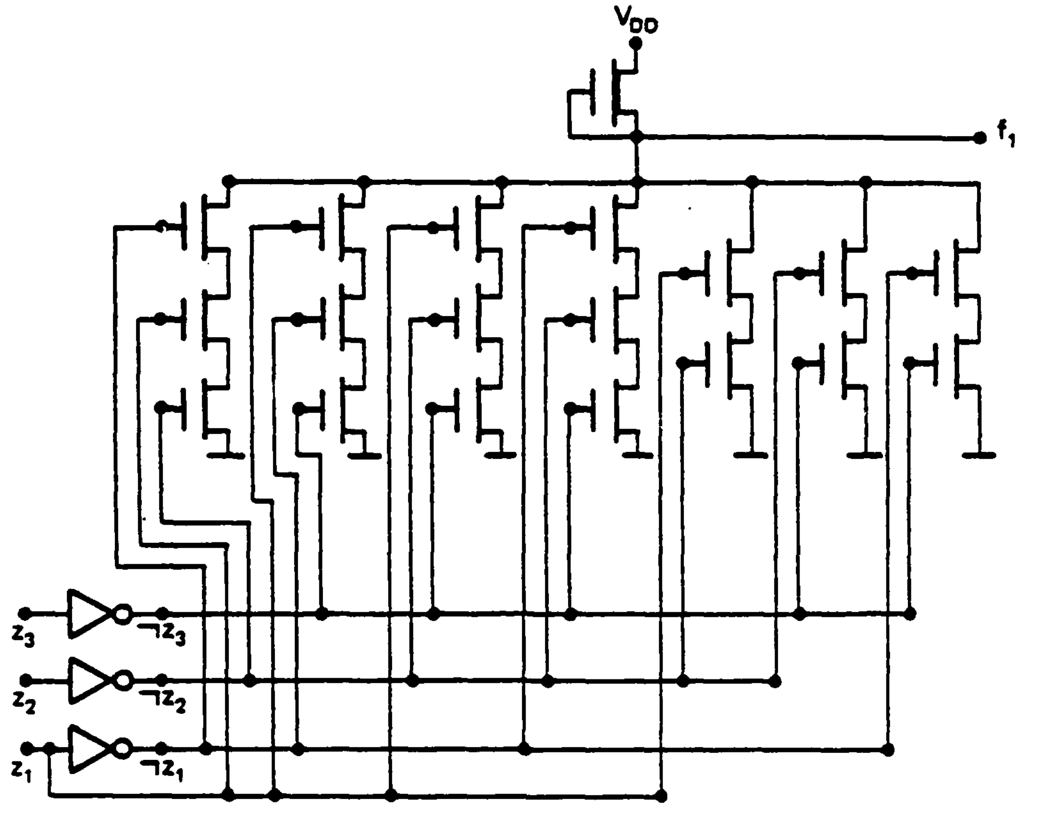
which can be expressed in its products of sum form as

$$f_1 = (z_1 \vee \neg z_1 \vee z_2) \wedge (z_1 \vee \neg z_1 \vee z_3) \wedge (\neg z_1 \vee z_2) \wedge (z_2 \vee z_3) \wedge (z_1 \vee z_3) \wedge (\neg z_1 \vee z_2 \vee z_3) \wedge (z_1 \vee z_2 \vee z_3).$$

Using simple algebraic manipulations this reduces to

$$\neg((\neg z_1 \wedge z_1 \wedge \neg z_2) \vee (\neg z_1 \wedge z_1 \wedge \neg z_3) \vee (z_1 \wedge \neg z_2) \vee (\neg z_2 \wedge \neg z_3) \vee (\neg z_1 \wedge \neg z_3) \vee (z_1 \wedge \neg z_2 \wedge \neg z_3) \vee (\neg z_1 \wedge \neg z_2 \wedge \neg z_3)),$$

which is in the required form as an inverse of a sum of product terms. We then have a series chain for each product term above. In the first series chain, the gate of the first transistor is connected to the output of an inverter whose input is connected to node $\mathbf{n}_1$, the gate of the second transistor is connected directly to node $\mathbf{n}_1$, while the gate of the third is connected to the output of another inverter with input node $\mathbf{n}_2$. This is repeated for each of the remaining series chains. The complete realization involving an MFB with 18 driver transistors and three other inverters is shown in Figure 4.6(a). A much simpler realization with an MFB containing only six driver transistors and three inverters is shown in Figure 4.6(b). Thus, Theorem 4.3 only guarantees the existence of an MFB that realizes a B-ternary function, and its proof describes a technique to construct one such realization using series chains of transistors connected in parallel across the output node and ground. However, it may be possible to construct another MFB to realize the same B-ternary function using a different design philosophy and may turn out to be even simpler than the first realization. Therefore, MFB's play a very important role in NMOS designs since any combinatorial switching function, which is a restriction of a B-ternary function to the two-valued Boolean algebra, can be realized by a at most two-level subnetwork composed of only MFB's according to Theorem 4.3 . In practical designs, however, the designer may want to realize several combinatorial switching functions in the same subnetwork which might require more



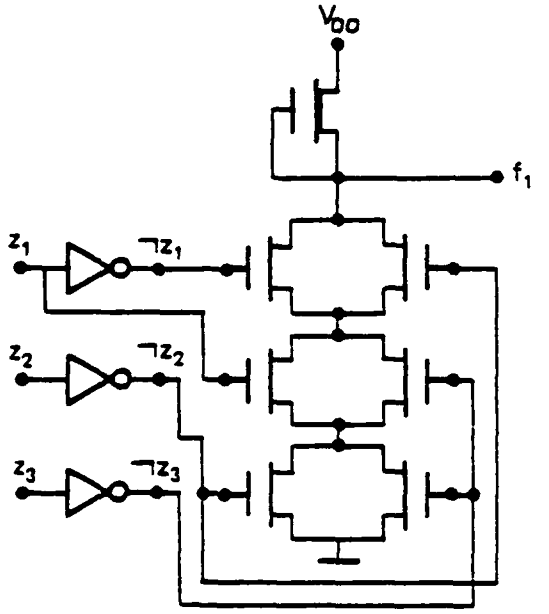


Figure 4.6(a) : A two-level MFB realization of f_1
(b) : A single MFB realization of f_1

levels. Furthermore, the use of pass transistors in realizing combinatorial logic [56] sometimes yields NMOS designs with better performance.

We will now describe the algorithm to simulate an MFB with no internal feedback. The algorithm begins by first assumming that the MFB is in a zero-delay mode and computes a sequence of transitions called the *zero-delay sequence* at its output node. Each transition in the zero-delay sequence is then delayed by a delay operator followed by a filtering process that produces a chronological and compatible delayed sequence. In this section we will focus our attention only on obtaining the zero-delay sequence at the output node of the MFB given the sequences of transitions at its input nodes. The delay and filtering operations will be discussed in Chapter 5.

Consider an MFB with a set of driver transistors $\mathbf{M_f=\{m_1,m_2,\ldots,m_p\}}$, a set of input nodes $\mathbf{INP_f=\{n_1,n_2,\ldots,n_p\}}$ and an output node $\mathbf{n_o}$, where $\mathbf{n_i=GATE(m_i)}$ for each $\mathbf{i=1,2,\ldots,p}$. Let $\mathbf{S_i}$ be the sequence of transitions at node $\mathbf{n_i}$ with transition times between integers $\mathbf{K_1}$ and $\mathbf{K_2}$. In the case of MFB's in simple SCC's we can assume $\mathbf{K_1=0}$ and $\mathbf{K_2=K}$, i.e., the input sequences are known for the entire time interval. In other situations the values of $\mathbf{K_1}$ and $\mathbf{K_2}$ would be decided by an algorithm to process blocks in a general SCC to be discussed in Chapter 6. Let $\mathbf{H_f}$ be the D-block corresponding to the MFB in the graph representing the network. Each edge $\mathbf{e_i}$ corresponding to transistor $\mathbf{m_i}$ is associated with an *edge sequence* $\mathbf{S(e_i)}$ which is initially set to $\mathbf{S_i}$. Two edges in a graph are said to be *parallel* if they have the same end vertices. A *simple graph* is a graph with no self loops and no parallel edges. The *simplification of a graph* is a graph obtained by collapsing all parallel edges into a single edge whose edge sequence is the sum ($\vee$) of the sequences of the parallel edges. We define the *elimination of a vertex* $\mathbf{v}$ from a simple graph as a procedure involving the following two steps:

(1) For every pair of vertices $\mathbf{a}$ and $\mathbf{b}$ adjacent to $\mathbf{v}$ in the graph, add an edge between $\mathbf{a}$ and $\mathbf{b}$ with the edge sequence of this new edge being the product ($\wedge$) of the sequences corresponding to the edges $\mathbf{<v,a>}$ and $\mathbf{<v,b>}$, respectively.

(2) Delete the vertex **v** (and all edges incident on it) from the new graph obtained in step (1).

It must be noted that eliminating a vertex from a simple graph could create parallel edges in the new graph. If we treat the graph H_f as a two-terminal network of switches between the output vertex and the ground vertex, we can define a *transmission function* **T** that denotes the state of "conduction" between the output and ground vertices as follows :

a) Each edge of the graph represents a switch whose state at any instant of time could be **open**, **intermediate**, or **closed**, denoted by symbols **0**, **u**, or **1**, respectively. Thus the edge sequence represents the variation of the state of the switch with time and can be defined to be the transmission function through the edge.

b) The transmission function through a path is defined to be the product ($\wedge$) of the transmission functions through the edges in the path.

c) The transmission function **T** between the output vertex and ground is the sum ($\vee$) over all possible paths between the two vertices of the transmission function through each path.

Clearly, **T** is a sequence of transitions and $S_o = \neg T$.

Theorem 4.4 : The operations of simplification of a graph and internal vertex elimination in a simple graph do not alter the transmission function between the output vertex and the ground vertex in the graph.

Proof : Let us consider a set of parallel edges $\hat{E} = \{e_1, e_2, \ldots, e_q\}$ between vertices **a** and **b** in a graph **H**. Let us partition the set of all paths Π between the output vertex and the ground vertex in **H** into two sets, namely, $\hat{\Pi}$ and Π', where $\hat{\Pi}$ is the set of all paths containing an edge $e_i \in \hat{E}$ and Π' is the set not containing any $e_i \in \hat{E}$. Let H_1 be the graph obtained from **H** by replacing the set $\hat{E}$ by a single edge e^* between **a** and **b**, with $S(e^*) = S(e_1) \vee S(e_2) \vee \cdots \vee S(e_q)$. If $\hat{\Pi}_1$ denotes all paths between the output vertex and the ground vertex in H_1 containing e^*, then, clearly the set of all paths Π_1 between output and ground vertices in H_1 is $\Pi_1 = \hat{\Pi}_1 \cup \Pi'$. Let $T(P)$ denote the transmission function through a path **P** and let $T(\Pi)$ denote the sum of transmission functions through each path in the set Π. The transmission function between output vertex and ground vertex in $\mathbf{H}$ is clearly

$$T=T(\hat{\Pi})\vee T(\Pi')$$

while that in H_1 is $T(\hat{\Pi}_1)\vee T(\Pi')$. Let P be some path in $\hat{\Pi}_1$ and $F=P-e^*$. Clearly, F is either a path or a union of two disjoint paths. In either case let $T(F)$ denote the product of the transmission functions through the edges in F. It is also easy to see that $P_i=F+e_i$ is a path in $\hat{\Pi}$ for each $i=1,2,\ldots,q$ and

$$T(P)=T(F)\wedge S(e^*)=T(P_1)\vee T(P_2)\vee\cdots\vee T(P_q).$$

Therefore, $T(\hat{\Pi})=T(\hat{\Pi}_1)$, and so the transmission function between the output vertex and ground vertex in H is the same as that in H_1. We can repeat the same argument for a set of parallel edges in H_1 and so on until we end up with a simple graph. Hence, the transmission function between two vertices in a graph does not change on simplification of the graph.

Now let us consider a simple graph H and an internal vertex v in the graph. Let Π_v be the set of paths from the output vertex to the ground vertex containing the vertex v and let Π' be the ones without v. If Π denotes the set of all paths between the output and ground vertices in H, then clearly the transmission function $T=T(\Pi)=T(\Pi_v)\vee T(\Pi')$. Suppose the degree of v in H is q and let $Adj_H(v)=\{w_1,w_2,\ldots,w_q\}$. Since H is simple, all vertices adjacent to v must be distinct. Let e_i denote the edge joining v and w_i in H. Let H_1 denote the graph obtained from H by eliminating v. Let the new edge that joins w_i and w_j in H_1 be denoted by e_{ij}. By definition $S(e_{ij})=S(e_i)\wedge S(e_j)$. Let $E_q=\{e_{ij} : i,j=1,2,\ldots,q, i\neq j\}$. Let Π_1 denote the set of all paths between the output vertex and ground in H_1. If $\hat{\Pi}$ denotes the set of paths between the output vertex and ground vertex in H_1 that contains edges from E_q, then clearly $\Pi_1=\hat{\Pi}\bigcup\Pi'$. We can divide the set $\hat{\Pi}$ into two disjoint subsets, $\hat{\Pi}_1$ containing only one edge from E_q and $\hat{\Pi}_2$ containing more than one edge from E_q. It can be easily verified that given any path $P_2\in\hat{\Pi}_2$ there exists a path $P_1\in\hat{\Pi}_1$ such that the terms in $T(P_2)$ are *subsumed* by the terms of $T(P_1)$, i.e., $T(P_1)\vee T(P_2)=T(P_1)$. Therefore, $T(\hat{\Pi})=T(\hat{\Pi}_1)$. Given a path $P\in\Pi_v$ such that w_i and w_j are the vertices adjacent to v on this path, we can construct a path P_1 such that $P_1=P-v+e_{ij}$. Clearly, $P_1\in\Pi_1$ and $T(P)=T(P_1)$. Thus there is a 1-1 correspondence between paths in

Π_v and $\hat{\Pi}_1$ and $T(\Pi_v)=T(\hat{\Pi}_1)$. Therefore,

$$T(\Pi_1)=T(\hat{\Pi})\vee T(\Pi')=T(\hat{\Pi}_1)\vee T(\Pi')=T(\Pi_v)\vee T(\Pi')=T(\Pi)=T,$$

and hence the theorem is proved. □

The algorithm to obtain the zero-delay sequence of transitions at the output node of a MFB begins with the simplification of the D-block corresponding to the MFB. It then picks an internal vertex in this simple graph and eliminates it and then simplifies the resultant graph. This process of elimination followed by simplification is repeated for each internal vertex. The end result would be a simple graph on two vertices, namely, the output vertex and the ground vertex. If the MFB is proper, then its D-block is a connected graph containing the ground node, and so the graph resulting from the elimination of all internal vertices followed by successive simplification would have an edge between the output and ground vertices. From Theorem 4.4, the transmission function between the output and ground vertices is the sequence associated with this single edge, and S_o would be the inverse of this sequence. Once the zero-delay sequence is obtained the transition times are delayed by a delay operator and the whole sequence is filtered using techniques to be discussed in Chapter 5.

Algorithm 4.2

Input : An MFB Ω_f, a set M_f of driver transistors,
a sequence of transitions S_i at the gate node of each
$m_i \in M_f$, the D-block of the MFB $H_f(V_f,E_f)$
with all vertices in V_f apart from the output vertex and the
ground vertex marked as "internal".
K_1 and K_2 are the end points of an interval during
which simulation is to be performed.
Output : A sequence S_o of transitions at the output node n_o of the MFB.

```
procedure MFB_SIM ( $\Omega_f$,$K_1$,$K_2$)
begin
        $S_o \leftarrow \emptyset$;
        for each edge $e_i \in E_f$ do
                $S(e_i) \leftarrow$ WINDOW ($S_i$,$K_1$,$K_2$);
        $H_0 \leftarrow$ SIMPLIFY ($H_f$);
        $j \leftarrow 0$;
        while there exists a vertex v in $H_j$ marked "internal" do
                begin
                H'←ELIMINATE $(v,H_j)$;
                $H_{j+1}$←SIMPLIFY(H');
                j←j+1;
            end
        if there exists an edge in $H_j$ then
            $e_o$← edge in $H_j$;
            $S_o$←¬S($e_o$);
            DELAY_FILTER $(S_o, \Omega_f)$;
        else
            append (u,1,−1) to $S_o$;
        end if
end

procedure ELIMINATE (v,H)
begin
        $\hat{H}$←H
        for each pair of vertices $w_i, w_j \in Adj_H(v)$ do
            begin
                $e_i$← <v,$w_i$>;
                $e_j$← <v,$w_j$>;
                add a new edge $e_{ij}$ in $\hat{H}$ joining $w_i$ and $w_j$;
                $S(e_{ij}) \leftarrow S(e_i) \wedge S(e_j)$;
            end
        return $\hat{H}$−v;
end
```

In the above algorithm, the choice of **v** as an internal vertex picked for elimination from H_j is important from the complexity point of view. If the degree of **v** in H_j is q, then the total number of edges added as a result of eliminating **v** from H_j is $q(q-1)/2-q$, which is equal to $q(q-3)/2$. Note that $q \geq 2$ if **v** is to be on a path in H_j. If $q=2$ then the new graph has one edge less than the number in H_j while the number of edges is unchanged if $q=3$. Hence a vertex of lowest degree in H_j is picked as the best candidate for elimination. The procedure WINDOW returns a sequence of those transitions occurring between K_1 and K_2 in its input sequence.

At this stage, we would like to point out that the procedures used in Algorithm 4.2 can be used to compute the transmission function between any two nodes in a two-terminal switching network provided the states at the internal nodes in such a network are not required for simulating other blocks in the network. In the case of an MFB, by definition, such a switching network exists, naturally, between the pullup node of the MFB and the ground node. Now let us consider a PTB which is viewed as a network of switches between the drain and source nodes of its pass transistors. A general PTB would clearly result in a multiport switching network. Once again, in general, one would be required to compute the states at several nodes within such a network since these could be external nodes according to our definitions in Chapter 3. Furthermore, the delay characteristics of PTB's are different from those of MFB's as will be seen in Chapter 5. Hence we choose to differentiate between MFB's and PTB's and we simulate them using different techniques.

4.2.3 Simulation of a PTB

Let $\mathbf{\Omega}_t$ be a PTB with a set of pass transistors $\mathbf{M}_t$. Let $\mathbf{NDS}_t=\{\mathbf{DRAIN}(\mathbf{m}_i),\mathbf{SOURCE}(\mathbf{m}_i):\mathbf{m}_i \in \mathbf{M}_t\}$ be the set of drain and source nodes of the pass transistors in the PTB and let $\mathbf{NG}_t=\{\mathbf{GATE}(\mathbf{m}_i):\mathbf{m}_i \in \mathbf{M}_t\}$ be the set of gate nodes. Consider the set $\mathbf{\Theta}$ of transition times of the signals at the gate nodes arranged in an ascending order. These time points divide the time interval of simulation into several *phases* such that during each phase $\boldsymbol{\phi}_j=(\mathbf{k}_j,\mathbf{k}_{j+1})$ the signal at each gate node in $\mathbf{NG}_t$ is at a fixed ternary value, i.e., a **0**, **u**, or **1**. The time $\mathbf{k}_j$ is the *initial time* and the time $\mathbf{k}_{j+1}$ is the *final time* of phase $\boldsymbol{\phi}_j$. Let $\mathbf{s}_{i,j}$ denote the fixed ternary state of the signal at gate node $\mathbf{n}_i \in \mathbf{NG}_t$ during phase $\boldsymbol{\phi}_j$. We partition the set $\mathbf{NDS}_t$ of drain and source nodes of pass transistors in the PTB into three subsets:

1. $\mathbf{N}_i=\{\mathbf{n}_\lambda \in \mathbf{NDS}_t : \mathbf{NODTYP}(\mathbf{n}_\lambda)=\text{"input"}\}$, the set of nodes of input strength,
2. $\mathbf{N}_p=\{\mathbf{n}_\lambda \in \mathbf{NDS}_t : \mathbf{NODTYP}(\mathbf{n}_\lambda)=\text{"pullup"}\}$, the set of nodes of pullup strength, and
3. $\mathbf{N}_n=\{\mathbf{n}_\lambda \in \mathbf{NDS}_t : \mathbf{NODTYP}(\mathbf{n}_\lambda)=\text{"normal"}\}$, the set of nodes of normal strength.

We are given the sequences of transitions at each node in $\mathbf{N}_i$ and $\mathbf{N}_p$ in the PTB. Our task is to compute the sequences of transitions at the nodes in $\mathbf{N}_n$. We do this in phases. Initially all the node sequences for $\mathbf{N}_n$ are set to the null sequence. We then simulate the PTB in the first phase $\boldsymbol{\phi}_1$ followed by the next phase and so on, updating the node sequences for the normal nodes in each phase. The simulation of a phase ϕ_j begins by constructing an undirected graph $\mathbf{H}_t$ with vertex set $\mathbf{V}_t$=$\mathbf{NDS}_t$ corresponding to the drain and source nodes of the pass transistors and the edge set $\mathbf{E}_t$ initially empty. For each pass transistor $\mathbf{m}_i \in \mathbf{M}_t$, an edge is inserted between **DRAIN**($\mathbf{m}_i$) and **SOURCE**($\mathbf{m}_i$) if $s_{i,j}=1$, i.e., if the signal at the gate node of the transistor is at a **1** during ϕ_j. Each connected component of the graph represents a switching network with nodes connected by two terminal switches that are in the closed state. Consider a component $\mathbf{C}_r$ of the graph. Let $\mathbf{STG}_r$ denote the subset of the strongest nodes (vertices) in $\mathbf{C}_r$, where the node strengths are ordered as **input > pullup > normal**. The *strength of the component* $\mathbf{C}_r$ is then defined to be the strength of its strongest node(s). If $|\mathbf{STG}_r| > 1$ and the strength of $\mathbf{C}_r$ is either **input** or **pullup**, then a *conflict* is declared at each normal node in the component. In case a node is experiencing a conflict in the present phase ϕ_j, there could be two possibilities, namely, the node was in a conflict in the previous phase ϕ_{j-1}, or it was not. In the former case the duration of the present phase is added to the existing value of the duration of the conflict. In the latter case the conflict is said to have started in the present phase and its duration is set to the duration of the phase.

If the strength of $\mathbf{C}_r$ is **normal**, then *charge sharing* is said to take place among the normal nodes in the component. Given any sequence of transitions, one can define the *initial value* of the signal to be the ternary value before the occurrence of the first transition and the *final value* to be the one after the last transition. For each node $\mathbf{n}_\nu \in \mathbf{C}_r$ let $\mathbf{S}(\mathbf{n}_\nu)$ denote the existing sequence of transitions at the node and $\mathbf{s}_\nu$ denote the final value of this sequence. We define an *equivalent voltage* $\mathbf{v}_\nu$ corresponding to the ternary signal $\mathbf{s}_\nu$ as $\mathbf{v}_\nu$=**0.0**, α*V_{DD}, or V_{DD} depending on whether $\mathbf{s}_\nu$=**0**, **u**, or **1**, respectively, where $0<\alpha<1$ is an empirical parameter. The default value for α is **0.5**. The *charge* on a node $\mathbf{n}_\nu$ is defined to be the product $\mathbf{v}_\nu$***CAP**($\mathbf{n}_\nu$), where **CAP**($\mathbf{n}_\nu$) is a lumped capacitance from node $\mathbf{n}_\nu$ to ground. In the case of charge sharing among the nodes of a component of normal strength, the total charge in the component is computed by summing up the charges on each node in the component and this quantity is divided by the total capacitance to yield a final voltage

$$v_f = \frac{\sum_{n_\nu \in C_r} v_\nu * CAP(n_\nu)}{\sum_{n_\nu \in C_r} CAP(n_\nu)}.$$

The final ternary value s_f reached by all the nodes in the component after charge sharing is then computed from v_f as $s_f=0$, **u**, or **1** depending on whether $v_f \leqslant V_L$, $V_L < v_f < V_H$, or $V_H \leqslant v_f$, respectively, where V_L and V_H are the low and high thresholds as defined in Chapter 3. For each node n_ν, if $s_\nu = s_f$ then no further analysis is required. Otherwise, if either s_ν or s_f is a **u**, then the transition (s_ν, s_f, k_{j+1}) is appended to the sequence $S(n_\nu)$. If $s_\nu \in \{0,1\}$ and $s_f = \neg s_\nu$, then the pair of transitions (s_ν, u, k_j), (u, s_f, k_{j+1}) is appended to the node sequence $S(n_\nu)$. The transition times are then suitably delayed and the sequence is filtered appropriately.

If $|STG_r|=1$ and the strength of the component is either **input** or **pullup** then the component is simulated as follows. Let n_s be the unique strongest node in the component. Let S_s be the sequence of transitions at the strongest node occurring within the phase, i.e., taking place between k_j and k_{j+1}. Consider a normal node n_ν in this component. If the node was experiencing a conflict in the previous phase then the conflict is declared as *resolved* in the present phase. Suppose a conflict that existed between times k_i and k_j for some $i<j$ at n_ν has now been resolved in the present phase. If the duration of the conflict $k_j - k_i$ is more than a preselected parameter ϵ_c, known as a *conflict parameter*, then the conflict at n_ν is declared as a *major conflict*, otherwise, it is a minor conflict. In case of a major conflict, a transition from the state of the node n_ν just before k_i to the **u** state is created at time k_i followed by a transition from **u** to the initial value of S_s at time k_j. Thus, in a major conflict, a node is forced to occupy the **u** state for the entire duration of the conflict. Minor conflicts are totally ignored. Once all conflicts (if any) are resolved, we again consider each normal node n_ν in the component. If the initial value of S_s is different from the final value of the existing sequence $S(n_\nu)$, then the appropriate transitions to the initial value of S_s are appended to the node sequence $S(n_\nu)$ followed by appending the sequence S_s itself. Each of the transitions appended is then suitably delayed and filtered.

Thus far, we have only considered transistors which are in the **closed** state during a phase ϕ_f. A pass transistor is said to have a state $\mathbf{u}^*$ if its gate node is at the **u** state in the present phase but occupies a **1** in the next phase. A transistor in the $\mathbf{u}^*$ state in the present phase is in an **intermediate** conducting state but would occupy a **closed** state during the next phase. This interpretation is radically quite different from the interpretation of the presence of the **X** state at the gate node of a transistor in conventional switch-level simulators such as MOSSIM [19]. The second part of the simulation of the PTB within a phase begins by constructing a supergraph with a vertex for each component C_r of H_t and an edge between two vertices C_r and C_s if a transistor in the $\mathbf{u}^*$ state has its drain node in C_r and source node in C_s or *vice versa*. The transistors whose gate signals are in the **0** state or in a **u** but not in a $\mathbf{u}^*$ state are ignored during the present phase. The connected components of the supergraph partition the components of H_t into *supercomponents*, such that each supercomponent consists of a set of components linked by pass transistors in the $\mathbf{u}^*$ state.

If a supercomponent consists of only one component, then no further analysis is required for this phase. Otherwise, the strength of the supercomponent is computed as the strength of the strongest component. If the strength of a supercomponent is **input** or **pullup** and it contains more than one strongest component, then this would lead to a conflict in the next phase and the simulation is postponed until the next phase. If the strength of a supercomponent is **normal** then this would clearly lead to charge-sharing in the next phase and, once again, the simulation is postponed until the next phase. The only situation left to consider is when the strength of a supercomponent is **input** or **pullup** and it has only one strongest component. Suppose the strongest component has only one strongest node whose final value in the present phase is s_f. Then for each node in each normal component, the transitions from the final value of its node sequence to s_f are appended to the node sequence. The transitions are delayed only if the node is not in a conflict during the present phase. If the strongest component has more than one strongest component then, once again, the simulation is postponed until the next phase.

The algorithm, described above, for the simulation of a PTB is somewhat heuristic, and instead of presenting a formal description, we will illustrate several of its features through an example. Consider a PTB shown in Figure 4.7, consisting of six pass transistors. We would like to simulate the PTB between 0.0 and 80.0 ns with a minimum resolvable time $h_{min} = 0.01$ ns. Thus transition times will be represented by integer multiples of 0.01 ns. Let us suppose that we will ignore any conflict lasting less than 0.1 ns, i.e., we choose the conflict parameter $\epsilon_c = 10$. For purposes of illustration we use an arrow head at a node to indicate **input** strength and a triangle to indicate **pullup** strength. Thus nodes n_0, n_5, and n_6 are of **input** strength while nodes n_1, n_2, n_3, and n_4 are of **pullup** strength. The nodes n_7 and n_8 are normal nodes in the circuit. The set of gate nodes for the pass transistors is $NG = \{n_1, n_2, n_3\}$. Let us assume the sequences of transitions at these nodes, which have already been computed, to be

$$S_1 = (0,u,4025), (u,0,4060), (0,u,6025), (u,1,6070)$$

$$S_2 = (1,u,2015), (u,0,2025)$$

$$S_3 = (0,u,2025), (u,1,2060), (1,u,6075), (u,0,6100)$$

respectively. The signal at the ground node n_0 is at **0** for all time and that at the supply node n_6 is at a **1** always. Node n_5 is driven by a pulsed voltage source with a sequence of transitions

$$\begin{aligned} S_5 = \; & (0,u,1001), (u,1,1005), (1,u,2014), (u,0,2018), (0,u,3002), (u,1,3006), (1,u,4013), (u,0,4017), \\ & (0,u,5002), (u,1,5006), (1,u,6014), (u,0,6018), (0,u,7002), (u,1,7006) \end{aligned}$$

and the node n_4, which is the output of an inverter with n_3 as input, has a zero-delay sequence $S_4 = \neg S_5$. The transition times of the gate sequences S_1, S_2, and S_3, arranged in order gives us the set

$$\Theta = \{2015, 2025, 2060, 4025, 4060, 6025, 6070, 6075, 6100\}$$

which has nine elements and hence results in ten phases. The first phase is $\phi_1 = (0,2015)$, the second is $\phi_2 = (2015,2025)$, and so on, until the last phase which is ϕ_{10}. We recall that $s_{i,j}$ is the fixed ternary state occupied by the gate node n_i during phase ϕ_j. We will represent these in a 3×10 matrix

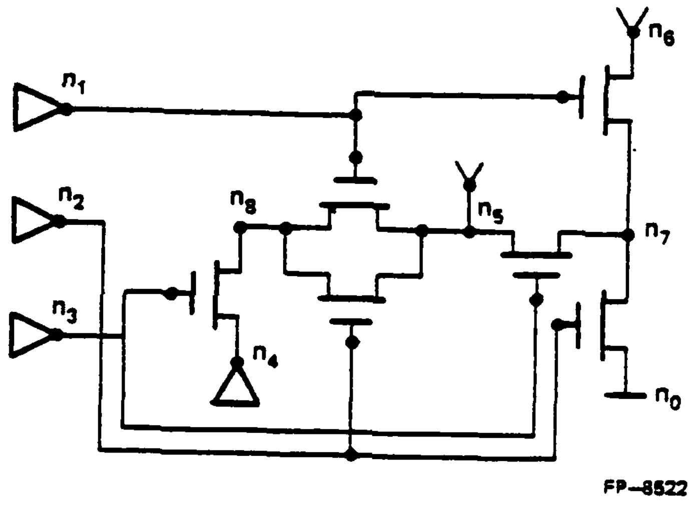


Figure 4.7 : An example of a PTB

$$A = \begin{vmatrix} 0 & 0 & 0 & 0 & u & 0 & u^* & 1 & 1 & 1 \\ 1 & u & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & u^* & 1 & 1 & 1 & 1 & 1 & u & 0 \end{vmatrix}. \qquad (4.7)$$

For example, from the third column of the above matrix we see that nodes $\mathbf{n}_1$ and $\mathbf{n}_2$ are in the **0** state during the third phase and node $\mathbf{n}_3$ is in the $\mathbf{u}^*$ state. The second row of the matrix says that node $\mathbf{n}_2$ is in the **1** state during ϕ_1, in the **u** state during ϕ_2 and **0** from then on until the end. The simulation in each phase will consist of two parts. In the first part we will contruct a graph on six vertices, namely, $\mathbf{n}_0$, $\mathbf{n}_4$, $\mathbf{n}_5$, $\mathbf{n}_6$, $\mathbf{n}_7$, and $\mathbf{n}_8$, with edges corresponding to transistors whose gate signals are in the **1** state. The second part will deal with a supergraph whose vertices are components of the first graph and edges corresponding to transistors with gate signals in the $\mathbf{u}^*$ state.

Phase 1, (0,2015)

From the first column of the matrix **A** in Equation (4.7) we see that in this phase only node $\mathbf{n}_2$ is in the **1** state. The graph is shown in Figure 4.8(a) and has four components. Components C_1 and C_4 have only one node each and therefore no analysis is necessary. The strength of C_2 is **input** and it contains only one input node $\mathbf{n}_0$. The node $\mathbf{n}_0$ is always in the **0** state, i.e., its corresponding sequence is **(u,0,−1)**. The the normal node $\mathbf{n}_7$ in this component will have this transition appended to its existing sequence, which is the null sequence initially. The strength of the component C_2 is also **input** and it also contains only one strongest node, namely, $\mathbf{n}_5$. The sequence of transitions in S_5 occurring within ϕ_1 is **(0,u,1001), (u,1,1005)**. This will be the zero-delay sequence to be appended to the sequence at the normal node $\mathbf{n}_8$ in this component. Thus on delaying and filtering the sequences at the normal nodes we get

S_7 = **(u,0,−1)**

S_8 = **(0,u,1022), (u,1,1188).**

Since there are no transistors in the $\mathbf{u}^*$ state in this phase, the second part of the phase simulation can be bypassed.

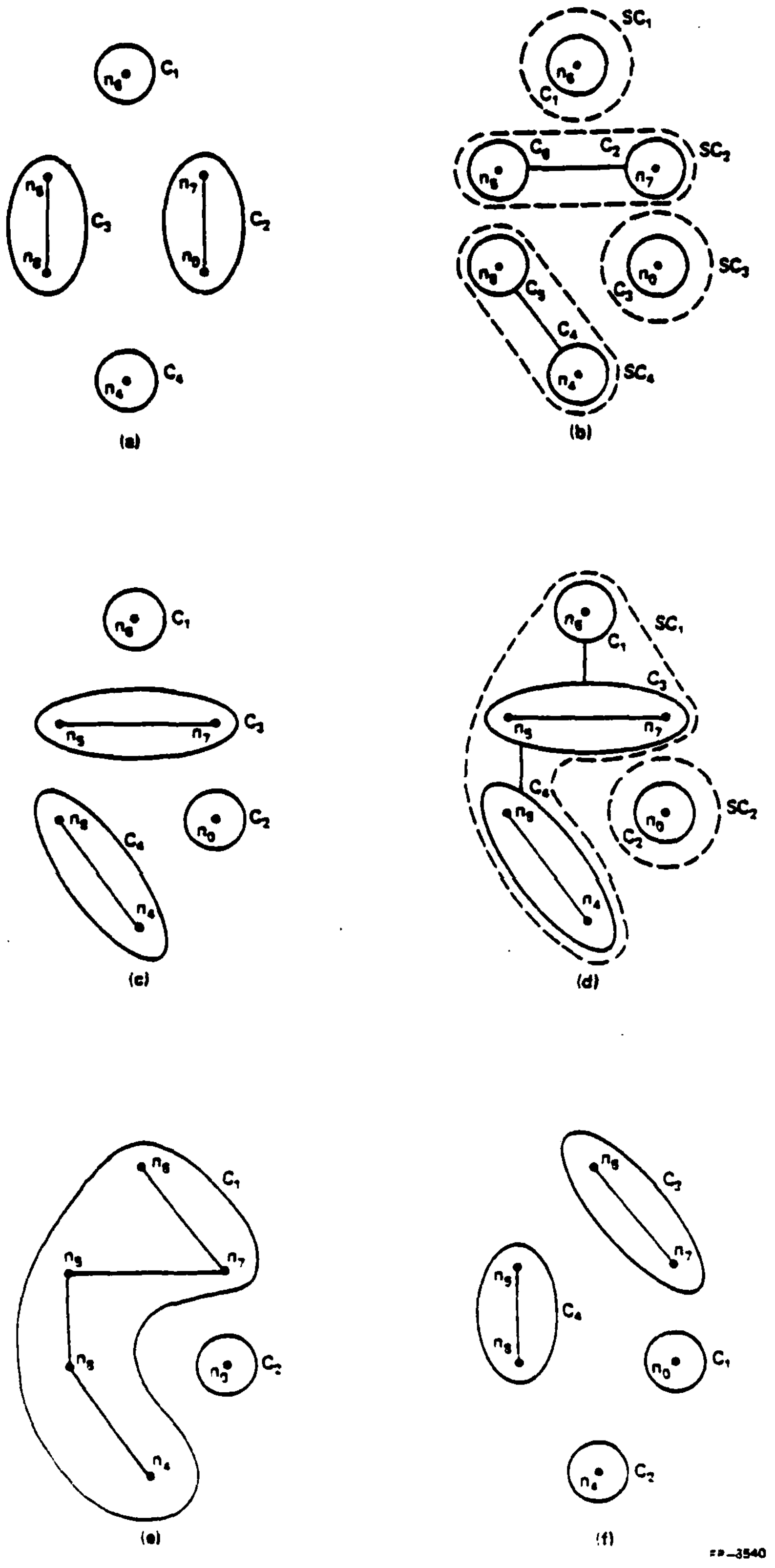


Figure 4.8 : Graphs and supergraphs for different phases in the PTB example

Phase 2, (2015,2025)

There are no transistors in this phase with gate signals either in the **1** or in the **u*** state, and hence the entire phase simulation can be bypassed. There is no change in the sequences S_7 and S_8 given above.

Phase 3, (2025,2060)

There are no transistors with gate signals in the **1** state in this phase. The graph therefore will have no edges and hence the first part of the simulation in this phase can be bypassed. The signal at node n_3, however, is in the **u*** state, thus resulting in a supergraph with six vertices and two edges as shown in Figure 4.8(b). Two of the supercomponents SC_1 and SC_3 contain only one component each and therefore need not be analyzed any further. The supercomponent SC_2 has C_6 as its only strongest component, and the final value of node n_5, which is the only node in C_6, in this phase is **0**. Since this agrees with the final value of the existing sequence at node n_7, which also happens to be the only normal node in C_2, we conclude that there is no change in sequence S_7 in this phase. The supercomponent SC_4 is of strength **pullup** consisting of a pullup component C_4 and a normal component C_5. The component C_4 consists of only one pullup node n_4, the final value of whose sequence in this phase is a **1**. Once again this agrees with the final value of the existing sequence at node n_8, which is the only normal node in C_5, and hence there in no change to either S_7 or S_8 in this phase.

Phases 4, 5, and 6, (2060,6025)

From time 2060 up to 6025, node n_3 is fixed at the **1** state and node n_2 is fixed at **0**. Node n_1, however, is at **0** during ϕ_4=(2060,4025), occupies the **u** state temporarily during ϕ_5=(4025,4060), and comes back to the **0** state during ϕ_6=(4060,6025). The graph during these three phases is shown in Figure 4.8(c). It consists of four components. Two of these components, C_1 and C_2, contain only one vertex each and need not be analyzed any further. In C_3, node n_7 is connected to n_5. Let $\hat{S}$ denote the sequence of transitions in S_5 occurring between 2060 and 6025, i.e.,

$$\hat{S} = (0,u,3002), (u,1,3006), (1,u,4013), (u,0,4017), (0,u,5002), (u,1,5006), (1,u,6014), (u,0,6018).$$

Since the initial value of $\hat{S}$ agrees with the final value of the existing S_7, we simply append $\hat{S}$ to S_7. In C_4, node n_8 is connected to the node n_4. The sequence of transitions in S_4 occurring during these phases is clearly $\neg\hat{S}$. Once again, since the initial value of $\neg\hat{S}$, which is 1, agrees with the final value of the existing S_8, we simply append $\neg\hat{S}$ to S_8. On delaying the transitions that were just appended we get

$$S_7 = (0,u,3023), (u,1,3189), (1,u,4032), (u,0,4076), (0,u,5023), (u,1,5189), (1,u,6032), (u,0,6076)$$

$$S_8 = (0,u,1022), (u,1,1188), (1,u,3065), (u,0,3237), (0,u,4174), (u,1,4657), (1,u,5065), (u,0,5237), (0,u,6175), (u,1,6658).$$

It must be noted that the last pair of transitions in S_8 takes place well after ϕ_6 and could be deleted by the filtering operator during simulation in ϕ_7. Furthermore, the second part of the simulation can be bypassed.

Phase 7, (6025,6070)

The graph during this phase is the same as the one in Figure 4.8(c) and there is no change in either S_7 or S_8 after the first part of simulation in this phase. The supergraph constructed in the second part of the simulation is shown in Figure 4.8(d). The supercomponent SC_2 consists of only one component C_2 and hence need not be analyzed any further. The supercomponent SC_1, however, consists of three components, namely, C_1, C_3, and C_4. C_1 and C_3 are of **input** strength while C_4 is of **pullup** strength. Since the transistors linking the components in this phase would be **closed** in the next phase, a possibility of the three components merging into one during the next phase exists. The new component would then have two strongest nodes, thereby leading to a conflict. Hence, we do not make any changes in either S_7 or S_8 even after the second part in this phase.

Phase 8, (6070,6075)

In this phase both n_1 and n_3 are in the 1 state, thus resulting in the graph shown in Figure 4.8(e). The component C_1 has two strongest nodes, namely, n_6 and n_5. Therefore a conflict is declared at the normal nodes n_7 and n_8. The duration of the conflict at both these nodes is **6075−6070=5**, or 0.05 ns in real time. Note that this situation was anticipated in the second part of the simulation of ϕ_7. The component C_2 has a single node and hence need not be analyzed any further. Since there are no gate nodes in the u^* state during this phase, the second part can be bypassed.

Phase 9, (6075,6100)

The graph constructed in the first part of the simulation in this phase is shown in Figure 4.8(f). It consists of four components, two of which, namely, C_1 and C_2, have only one node each. The component C_3 consists of a normal node n_7 connected to an input node n_6. The component C_4 has normal node n_8 connected to the input node n_5. Since both n_7 and n_8 were involved in a conflict situation in the previous phase, this conflict is now resolved. The total duration of the conflict in either node was **5** in integer time, which is less than $\epsilon_c=10$, and hence the conflict is declared as a minor conflict and is ignored. The final value of S_7 is a **0** while the state of the node n_6 is a **1** since it is the supply node. Hence we append the pair **(0,u,6075), (u,1,6076)** to S_7, which on delaying would result in

S_7 = **(0,u,3023), (u,1,3189), (1,u,4032), (u,0,4076), (0,u,5023), (u,1,5189), (1,u,6032), (u,0,6075), (0,u,6106), (u,1,6273).**

The initial state of the node n_5 in this phase is a **0**. The final value of S_8 can be seen to be a **1**. However, the last pair of transitions **(0,u,6175), (u,1,6658)** takes place well after the present phase. Hence this pair is deleted from S_8 and now the final value of S_8 is a **0** which agrees with the initial state of the strongest node, n_5, in its component. This is an example of the filtering operation to be discussed in Chapter 5. Since S_5 has no transitions occurring in this phase, we are done with the first part of the simulation in this phase. The second part is bypassed. Thus the sequence at node n_8 after this phase turns out to be

S_8 = **(0,u,1022), (u,1,1188), (1,u,3065), (u,0,3237), (0,u,4174), (u,1,4657), (1,u,5065), (u,0,5237).**

Note that we have deleted the last pair of transitions from the previous sequence S_8.

Phase 10, (6100,8000)

In this phase the graph remains the same as in the previous phase. The sequence S_7 does not change since n_7 is still connected to the supply node n_6. The pair of transitions **(0,u,7002), (u,1,7006)** from S_5 occurring within this phase get delayed and appended to S_8.

Thus the final result is that the sequences at nodes $\mathbf{n}_7$ and $\mathbf{n}_8$ are

S_7 = **(0,u,3023) , (u,1,3189) , (1,u,4032) , (u,0,4076) , (0,u,5023) , (u,1,5189) , (1,u,6032) , (u,0,6076) , (0,u,6106) , (u,1,6273)**

and

S_8 = **(0,u,1022) , (u,1,1188) , (1,u,3065) , (u,0,3237) , (0,u,4174) , (u,1,4657) , (1,u,5065) , (u,0,5237) , (0,u,7023) , (u,1,7189).**

4.3 Conclusions

We began this chapter by defining transitions between ternary states and showed how sequences of transitions can be used to represent ternary digital waveforms of signals. We also presented algorithms that perform a switch-level simulation of SRC's, MFB's, and PTB's. In the case of an SRC the sequence of transitions at the output node is constructed directly from the input description of its analog waveform. In the case of an MFB we showed that the zero-delay state of its output node at any instant of time is a B-ternary function of the states of its input nodes at the same time instant. Furthermore, the output node of an MFB is of **pullup** strength and the only stronger node in the D-block of the MFB is the ground node. On exploiting all these properties of an MFB, we came up with a fairly simple graph algorithm based on simplification of graphs and eliminating internal vertices in simple graphs to compute the sequence of transitions at the output node of an MFB directly from those at the input nodes of the MFB. For a PTB, we presented a more complex, and somewhat heuristic, approach utilizing the full power of conventional switch-level simulation. This approach is similar to that of MOSSIM [19], except for the interpretation of the intermediate **u** state (or the **X** state as used in MOSSIM). We illustrated the approach with the help of a simple example.

If a block of a partitioned network appears in a simple SCC, and if the SCC's have been processed according to the ordering presented at the end of Chapter 3, then the sequence of transitions at each input node to the block will be known for the entire time interval of interest. In this case the block can be simulated for the entire period of time by algorithms described in this chapter. Otherwise, the blocks are simulated only over certain windows in time. The end points of these windows are specified by a special algorithm to be described in Chapter 6.

CHAPTER 5

DELAY AND FILTERING OPERATIONS

The algorithms described in the previous chapter compute zero-delay sequences of transitions at the output nodes of an MFB and normal nodes of a PTB. By zero delay, we mean a transition at the gate node of a transistor causes a transition in the switching state of the transistor immediately, and this change affects the state of other nodes without any delay in time. In this chapter we will consider altering the transition times so that the resulting sequence would then correspond to a ternary waveform that is fairly close to the ternary equivalent of the analog waveform if computed by an accurate circuit simulator. The task of the delay operator is to alter the transition times only for a complete pair of transitions. Each application of the *delay operator is followed by a filtering operation* which accounts for the effect of delaying a complete pair of transitions on the future transitions in the sequence. The filtering operator also transforms a partial pair of transitions into a form that can be handled by the delay operator.

The delay operator is characterized by delay functions which are computed for a set of standard circuit primitives and stored in tables. This step involves the use of an accurate circuit simulator to simulate each primitive and could consume large amounts of computation time. The circuit primitives, however, do not change as long as the technology remains fixed and hence the computations of the delay functions need be performed only once for each technology. This step, therefore, can be considered as a preprocessing phase since the same delay tables could be used to simulate many different networks designed in a fixed technology. The delay operator then computes new values for transition times in a complete pair of transitions at a certain node in a general block in two steps. First, a mapping technique is used to transform the block into a configuration that resembles one of the primitives for which the delay functions have been computed. Time scaling is then used to transform the new configuration into a standard primitive after which the delay values can be obtained through a table lookup.

5.1 Computation of Delay Functions for Standard Primitives

In the case of conventional NMOS depletion load technology, we consider five basic configurations, called *primitives*.

Primitive 1 : A simple inverter driving a lumped grounded capacitance C_1. An input signal V_{in} is applied at the gate node of the driver transistor m_D and the output, V_o, is observed at the source node of the load transistor m_L as shown in Figure 5.1. We consider two types of input waveforms, namely,

Type "0" : V_{in} rising from 0 V to 5 V

and

Type "1" : V_{in} falling from 5 V to 0 V.

Primitive 2 : A pass transistor m_P whose drain is connected to a constant DC voltage source V_{DC} and the gate driven by a pulse V_{in} rising from 0 V to 5 V. The source node of m_P is connected to a grounded capacitance C_2 as shown in Figure 5.2. The output waveform V_o in this case is observed at the source node of m_P. We consider two types of V_{DC}, namely,

Type "0" : $V_{DC} = 0$ V

and

Type "1" : $V_{DC} = 5$ V.

Primitive 3 : A pass transistor m_P whose gate is held fixed at 5 V and drain driven by an input pulse V_{in}. The source node, which is also the output node, has a waveform V_o and is connected to a grounded capacitance C_2 as shown in Figure 5.3. We consider two types of input waveforms, namely,

Type "0" : V_{in} rising from 0 V to 5 V

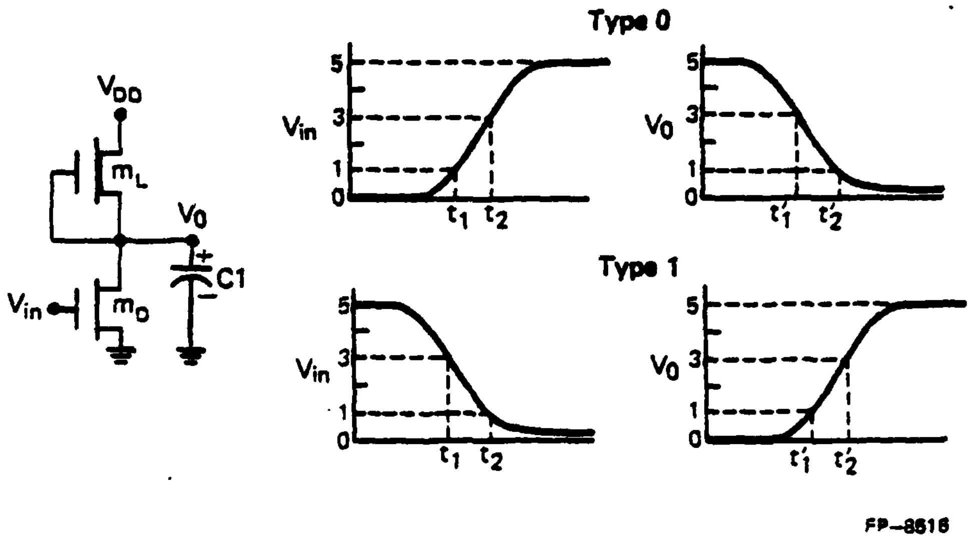


Figure 5.1 : Primitive 1 of the delay operator

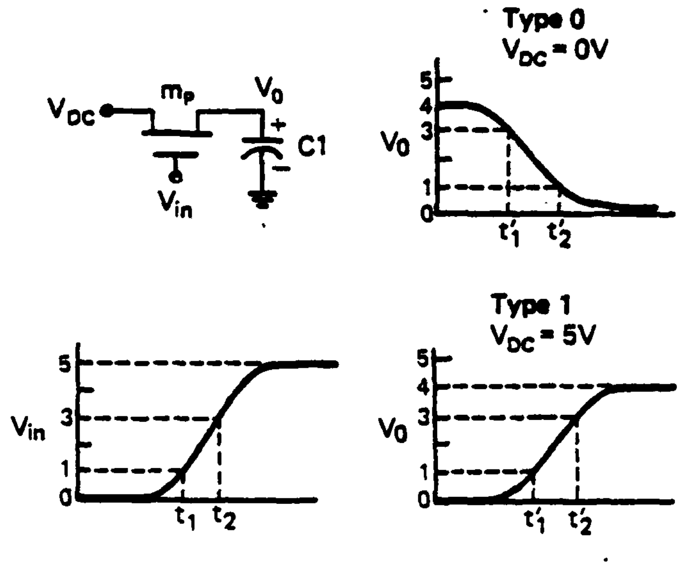


Figure 5.2 : Primitive 2 of the delay operator

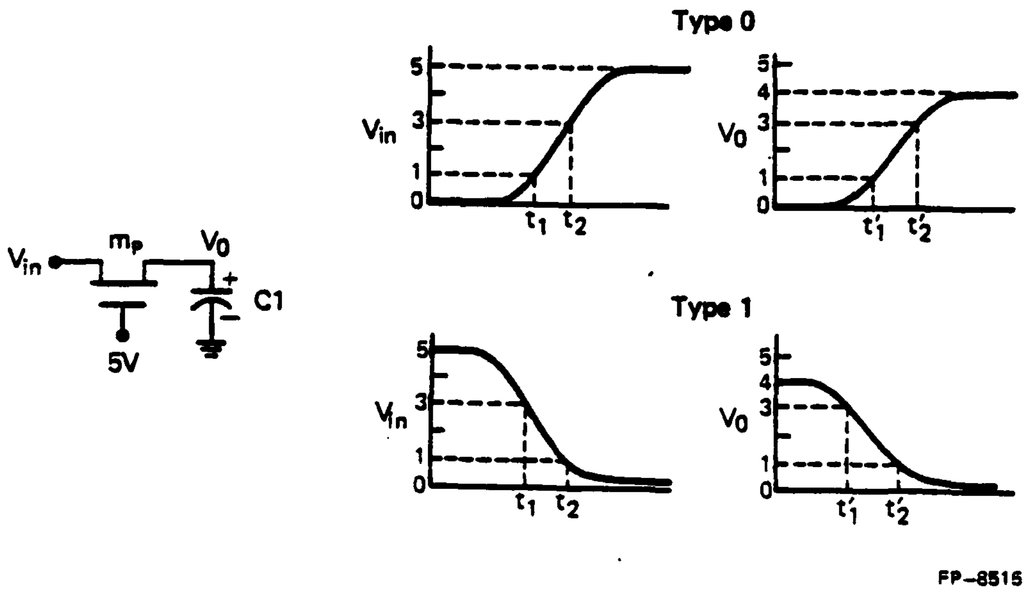


Figure 5.3 : Primitive 3 of the delay operator

and

Type "1" : V_{in} falling from 5 V to 0 V.

Primitive 4 : A simple inverter with driver transistor $\mathbf{m_D}$ and load $\mathbf{m_L}$ driving a pass transistor $\mathbf{m_P}$. Grounded capacitors $\mathbf{C_1}$ and $\mathbf{C_2}$ are connected to the pullp node of the inverter and to the source node of the pass transistor, respectively. A pulse V_{in} rising from 0 V to 5 V is applied at the gate of the pass transistor $\mathbf{m_P}$ while the gate of the driver transistor $\mathbf{m_D}$ is connected to a fixed DC voltage source V_{DC} as shown in Figure 5.4. There are two types of V_{DC}, namely,

Type "0" : $V_{DC} = 0$ V

and

Type "1" : $V_{DC} = 5$ V.

Primitive 5 : Same configuration as primitive 4 except that the gate of the pass transistor $\mathbf{m_P}$ is held fixed at 5 V while a pulse V_{in} is applied at the gate of the driver transistor $\mathbf{m_D}$ as shown in Figure 5.5. Here, we consider two types of input pulses, namely,

Type "0" : V_{in} rising from 0 V to 5 V

and

Type "1" : V_{in} falling from 5 V to 0 V.

In each of the above primitives we have an input waveform V_{in} which varies between V_{DD}=5 V and 0 V and produces an output waveform V_o. For a fixed input waveform, the shape of the output V_o could depend upon several circuit, device, and process parameters. The parameters we would consider are the following: zero-bias device threshold (VTO), both for enhancement and depletion devices, a resistance for each device which is a function of the ratio of its channel length (L) to its width (W), the transconductance parameter, $KP=\mu_n\epsilon_{ox}/t_{ox}$, which in turn is a function of the carrier mobility μ_n, the permittivity of the oxide material ϵ_{ox} and the thickness of the oxide t_{ox}, and finally, the capacitance at each node. Among these parameters we assume that all enhancement transistors have the same zero-bias threshold, VTO_E, all depletion transistors have the same VTO_D, and that these values remain

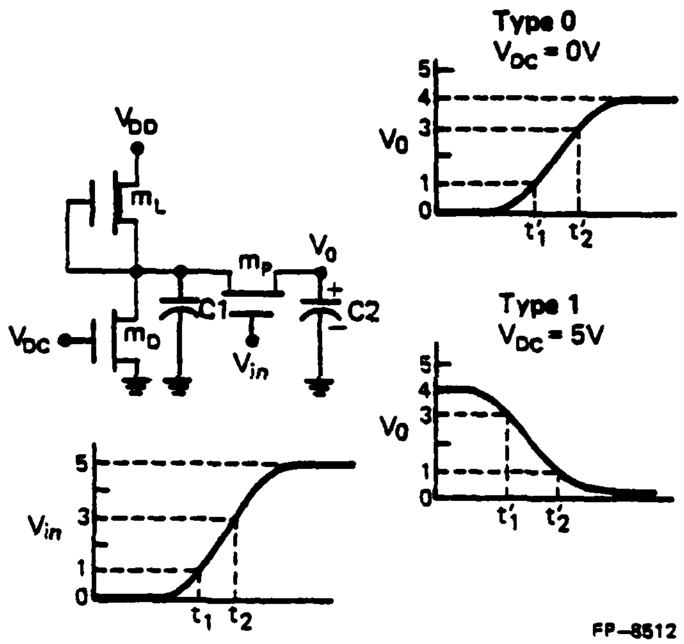


Figure 5.4 : Primitive 4 of the delay operator

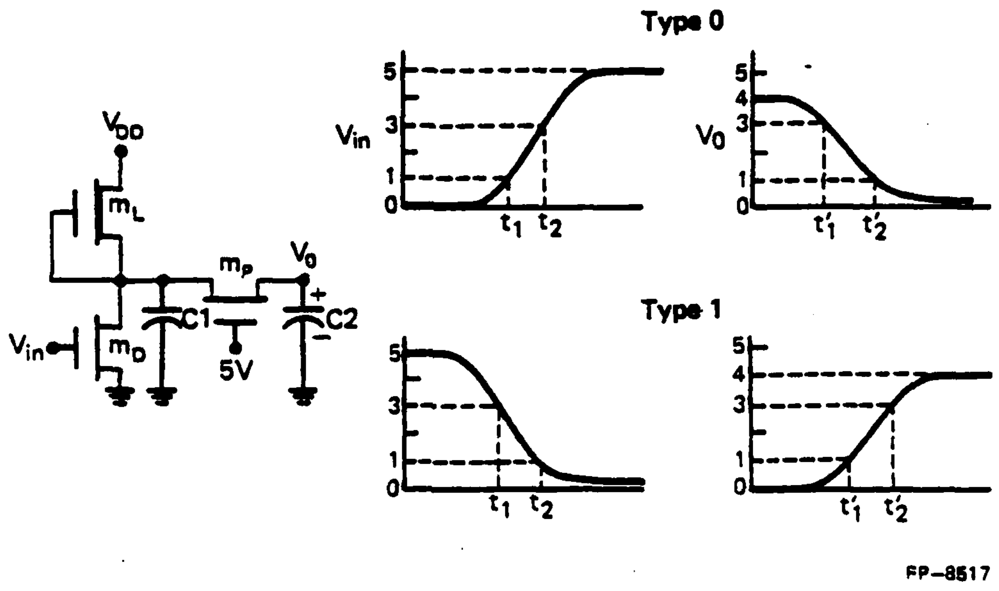


Figure 5.5 : Primitive 5 of the delay operator

fixed for a given technology. Typical values are $\mathbf{VTO_E=+1.0V}$ and $\mathbf{VTO_D=-3.0V}$. The rest of the parameters are allowed to vary between the different devices and nodes in the network. In the five primitives described above, we let $\mathbf{R_D=RES(m_D)}$, $\mathbf{R_L=RES(m_L)}$, and $\mathbf{R_P=RES(m_P)}$ denote the device resistances of the driver, load, and pass transistors, respectively. We will choose a standard driver, a standard load, and a standard pass transistor, and let $\mathbf{R_{DS}}$, $\mathbf{R_{LS}}$, and $\mathbf{R_{PS}}$ denote the resistances of these *standard devices*, respectively. A typical set of standard devices is

Load : **W=5 μ , L=10 μ,**

Driver : **W=10 μ , L=5 μ,**

Pass : **W=10 μ , L=10 μ.**

For the above choice of standard load and driver devices, we notice that $\mathbf{R_{LS}/R_{DS}=4}$. We will refer to this ratio as the *standard inverter ratio* and denote it by $\boldsymbol{\delta_S}$.

Let $\mathbf{C_{iS}}$ denote the *standard capacitance* in the case of the $\mathbf{i^{th}}$ primitive. Typically, $\mathbf{C_{iS}=0.01\ pF}$ for $\mathbf{i=1,2,3}$ and $\mathbf{C_{iS}=0.1\ pF}$ for $\mathbf{i=4,5}$. A primitive is a *standard primitive* if $\mathbf{R_D=R_{DS}}$, $\mathbf{R_L=R_{LS}}$, $\mathbf{C_1=C_{1S}}$ in primitive 1, $\mathbf{R_P=R_{PS}}$, $\mathbf{C_2=C_{2S}(C_{3S})}$ in primitives 2 and 3, and $\mathbf{R_P=R_{PS}}$, $\mathbf{R_L/R_D=\delta}$, $\mathbf{C_2=C_{4S}(C_{5S})}$ in primitives 4 and 5. In primitives 4 and 5 let us define two dimensionless quantities $\boldsymbol{\beta=R_D/R_P}$ and $\boldsymbol{\gamma=C_1/C_2}$. We use these two additional parameters to completely specify the standard primitive. We allow $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ to be variable over ranges $[\beta_{min},\beta_{max}]$ and $[\gamma_{min},\gamma_{max}]$ respectively.

Consider one of the above primitives. We treat $\mathbf{V_{in}}$ to be an analog ramp waveform with a full swing of $\mathbf{V_{DD}}$. This waveform will then cross both the threshold voltages $\mathbf{V_L}$ and $\mathbf{V_H}$. Let $\mathbf{t_1}$ and $\mathbf{t_2}$ denote the two threshold crossing times. Clearly, this change in the input waveform would cause the output waveform $\mathbf{V_o}$ to cross both the thresholds also. Let $\mathbf{t'_1}$ and $\mathbf{t'_2}$ be the output threshold crossing times. We define $\mathbf{\Delta_{in}=t_2-t_1}$ as a measure of the slew rate of the input signal and two delay quantities, $\mathbf{\Delta t_1=t'_1-t_1}$, known as the *inertial delay*, and $\mathbf{\Delta t_2=t'_2-t'_1}$, known as the *rise/fall delay*. Thus, given $\mathbf{t_1}$ and the two delay quantities, we can easily compute $\mathbf{t'_1=t_1+\Delta t_1}$ and $\mathbf{t'_2=t'_1+\Delta t_2}$. We will use the symbol $\mathbf{\Delta t_o}$ to refer to both the delays collectively.

We will now consider computing Δt_o for standard primitives. We first consider the standard primitive 1 with rising inputs, i.e., type "0". In this case the device sizes and node capacitances are fixed at their standard values. We consider an input ramp V_{in} with a certain rise time, resulting in some value of Δ_{in}. We then simulate this circuit using an accurate circuit simulator, such as SPICE2 [1], which gives us a falling waveform for V_o. From both the input and the output waveforms we can compute the threshold crossing times t_1, t_2, t'_1, and t'_2, and hence both the delays Δt_1 and Δt_2. We then repeat this for a falling input ramp, i.e., type "1", with the same slew rate as before and compute two more delay values. This experiment is then repeated with input ramps of different slew rates, each time producing four more delay values (two in each type), which are stored in a table as functions of Δ_{in}. The entire procedure is repeated to generate the delay tables in the case of standard primitives 2 and 3. The tables in all three cases are one-dimensional since their entries are functions of only Δ_{in}. Each table entry contains four values, namely, Δt_1 and Δt_2 for type "0" and the same for type "1".

In the case of standard primitives 4 and 5, we need to specify the values of C_1, R_D and R_P in order to completely specify the circuit. We do this with the help of the parameters β, γ, and δ. For fixed values of these parameters, we get $C_1=\gamma C_2$, $R_D=\beta R_P$, and $R_L=\delta R_D$, where R_P and C_2 take on the standard values. For the present we consider the inverter ratio δ to be a fixed parameter. We will remove this restriction in the later sections. We start with some initial values for β and γ, simulate the circuit using SPICE2 [1], and obtain a set of delay values for each value of Δ_{in}. We repeat this procedure for different values of β and γ and generate three-dimensional delay tables. Each entry in the table contains four delay values as before; however, these values are now functions of three parameters, namely, the slew rate of the input Δ_{in}, a ratio of driver to pass transistor resistance β, and a ratio of capacitances γ.

We have therefore described the generation of delay tables for a fixed technology. In case of a change in technology, the procedure has to be repeated to generate a new set of tables. It must be noted that we consider a change in the values of the zero-bias device thresholds VTO_D and VTO_E as a change in the process technology. However, if there is only a change in the transconductance parameter (KP) or any of the parameters that affect its value, we can use the same set of delay tables as will be shown in Section 5.2. The delay values are plotted as functions of input slew rate Δ_{in} for primitives 1, 2, and 3 and as functions of Δ_{in}, β and γ for primitives 4 and 5 in Appendix I for a particular technology.

5.2 Delay Functions for Nonstandard Primitives

In this section we will show how we can compute the delay values for nonstandard primitives from the delay tables for standard primitives computed in the previous section. By nonstandard primitives, we mean, primitives that have nonstandard devices and nonstandard node capacitances.

For the analysis below we choose a simple DC analog model for an NMOS transistor by ignoring body effect, channel length modulation, short channel effects, and other higher-order effects. Then for any primitive i, where i=1,2,3, we can write the first-order differential equation for the output waveform in the following simplified form :

$$\frac{dV_o(t)}{dt} = \frac{1}{\sigma_i} f_i(V_o(t), V_{in}(t)) \tag{5.1}$$

where

$$\sigma_1 = \frac{R_D C_1}{\eta KP},$$

$$\sigma_2 = \sigma_3 = \frac{R_P C_2}{\eta KP},$$

η is a fixed constant for a given technology, and f_1, f_2 and f_3 are some nonlinear functions of their arguments. It must be noted that in case of a nonstandard primitive 1, the Equation (5.1) is obtained by assuming that the inverter ratio $\delta=\delta_S$, where δ_S denotes the standard inverter ratio. We justify this assumption with the following arguments. In the case of falling output waveforms, i.e., a type "0" situation, the current I_D through the driver transistor is primarily responsible for discharging the output capacitance C_1 and hence there is no significant change if, in this case, the load transistor is replaced by a depletion device with $R_L=\delta_S R_D$. Similarly, for rising output waveforms, i.e., a type "1" situation, the current I_L through the load transistor is primarily responsible for charging C_1 and hence there is no significant change if, in this case, the driver is replaced by one with $R_D=R_L/\delta_S$. It is, therefore, reasonable to assume that even in the case of a nonstandard primitive 1, the inverter ratio is fixed at δ_S, and so δ need not be included as the third argument for the function f_1.

In the case of a nonstandard primitive i, where i=4,5, we can describe the analog behavior of the two unknown waveforms $V_1(t)$, the voltage across the capacitance C_1, and $V_o(t)$, the output voltage, with the help of the following two first-order differential equations :

$$\frac{dV_1(t)}{dt} = \frac{1}{\sigma_i} f_{i1}(V_o(t),V_1(t),V_{in}(t),\beta,\gamma) \tag{5.2a}$$

$$\frac{dV_o(t)}{dt} = \frac{1}{\sigma_i} f_{i2}(V_o(t),V_1(t),V_{in}(t),\beta,\gamma) \tag{5.2b}$$

where

$$\sigma_4 = \sigma_5 = \frac{R_P C_2}{\eta KP},$$

f_{41}, f_{42}, f_{51}, and f_{52} are some nonlinear functions of their respective arguments. Once again, we have not included the parameter δ as one of the arguments in the above functions since it is reasonable to assume that $\delta=\delta_S$ using the same arguments as in the case of primitive 1.

From Equation (5.1), it is clear, that in a fixed technology, if we fix the input waveform V_{in} and the value of the parameter σ_i, then we will get the same output waveform V_o in primitives 1, 2, and 3. If, in addition, we also fix the type, namely "0" or "1", in a primitive, then fixing V_{in} is equivalent to fixing the value of Δ_{in}, which is the measure of the input slew rate. Hence, in the case of a nonstandard primitive i, where i=1,2,3, the delays (both inertial delay and rise/fall delay) at the output, collectively denoted by Δt_o, are only functions of two parameters, namely, Δ_{in} and σ_i. In the case of primitives 4 and 5, from Equations (5.2a) and (5.2b), it is clear that if we fix V_{in}, β, γ, and σ_i, we will get the same waveforms for both V_1 and V_o. Hence, in the case of a nonstandard primitive i, where i=4,5, the delays Δt_o are functions of four parameters, namely, Δ_{in}, β, γ, and σ_i. In the previous section we have computed the delay functions for the case $\sigma_i = \sigma_{is}$, where σ_{is} denotes the value of the parameter σ_i computed for a standard primitive i= 1, 2, 3, 4, or 5. Using the same set of delay tables, we will now demonstrate a technique, known as *time scaling*, to compute the delay functions for nonstandard primitives, i.e., primitives with $\sigma_i \neq \sigma_{is}$.

Suppose we introduce a new time variable $\tau = \alpha t$, where α is a *scale factor*. we can then rewrite the Equations (5.1), (5.2a), and (5.2b) in terms of τ as :

$$\frac{dV_o(\tau)}{d\tau} = \frac{\alpha}{\sigma_i} f_i(V_o(\tau),V_{in}(\tau)) \tag{5.3}$$

and

$$\frac{dV_1(\tau)}{d\tau} = \frac{\alpha}{\sigma_i} f_{i1}(V_o(\tau),V_1(\tau),V_{in}(\tau),\beta,\gamma) \tag{5.4a}$$

$$\frac{dV_o(\tau)}{d\tau} = \frac{\alpha}{\sigma_i} f_{i2}(V_o(\tau),V_1(\tau),V_{in}(\tau),\beta,\gamma). \tag{5.4b}$$

If we now set $\alpha=\sigma_i/\sigma_{is}$ in each of the above equations, we get :

$$\frac{dV_o(\tau)}{d\tau} = \frac{1}{\sigma_{is}} f_i(V_o(\tau),V_{in}(\tau)) \tag{5.5}$$

and

$$\frac{dV_1(\tau)}{d\tau} = \frac{1}{\sigma_{is}} f_{i1}(V_o(\tau),V_1(\tau),V_{in}(\tau),\beta,\gamma) \tag{5.6a}$$

$$\frac{dV_o(\tau)}{d\tau} = \frac{1}{\sigma_{is}} f_{i2}(V_o(\tau),V_1(\tau),V_{in}(\tau),\beta,\gamma) \tag{5.6b}$$

which are the same as Equations (5.1), (5.2a), and (5.2b), respectively, with t and σ_i replaced by τ and

σ_{iS}. Thus, the Equations (5.5), (5.6a), and (5.6b) represent the behavior of the **standard** primitives in a new time domain with τ as the time variable. The slew rate of the input in this new time domain is Δ_{in}/α. If $\Delta\tau_o$ denotes the delays (both inertial and rise/fall) at the output in the new time domain, then clearly $\Delta t_o = \alpha\Delta\tau_o$. But $\Delta\tau_o$ can be obtained from the delay tables compiled in the previous section for standard primitives for input slew rate Δ_{in}/α in primitives 1, 2, and 3, and for resistance ratio β and capacitance ratio γ, as additional parameters, in primitives 4 and 5. Let $g_i(\Delta)$ denote the delay functions tabulated as a function of input slew rate Δ, for standard primitive i, where i= 1, 2, or 3, and let $g_i(\Delta,\beta,\gamma)$ denote those tabulated as a function of input slew rate, resistance ratio, and capacitance ratio for standard primitive i= 4 or 5. We can then outline the scheme for computing the delay values of nonstandard primitives from those computed for standard primitives as follows :

a) Let i be the primitive number and let Δ_{in} be the input slew rate. Compute σ_i.

b) Compute $\alpha \leftarrow \sigma_i/\sigma_{iS}$.

c) If i= 1, 2, or 3, then obtain $\Delta t_o \leftarrow \alpha g_i(\Delta_{in}/\alpha)$.

d) If i= 4 or 5, then obtain $\Delta t_o \leftarrow \alpha g_i(\Delta_{in}/\alpha,\beta,\gamma)$.

It must be pointed out that the delay functions for nonstandard primitives could be computed just as in the standard case by introducing an additional parameter σ_i in each of the tables for the i^{th} primitive. This would then mean storing two-dimensional tables for primitives 1, 2, and 3, and four-dimensional tables for primitives 4 and 5. By using the scaling technique outlined above, we have managed to obtain the delay values with only one-dimensional and three-dimensional tables, respectively. Thus we have considerably reduced both the CPU-storage space and the preprocessing time for generating the delay tables However, we have used a very simple device model for the NMOS transistors to derive this technique, and this could cause some errors in the delay predictions if more complex device models are used. This is one of the factors responsible for timing errors of the delay operator.

5.3 Delay Operator for MFB's and PTB's

In this section we describe a delay operator which alters the transition times in a complete pair of zero-delay transitions at the output node of an MFB and at normal and pullup nodes of a PTB.

We first consider an NMOS network in which each MFB is an inverter and each PTB consists of a single pass transistor. Let $\mathbf{n}_o$ be the output node of an inverter and let $(\mathbf{x},\mathbf{u},\mathbf{k}_j)$, $(\mathbf{u},\neg\mathbf{x},\mathbf{k}_{j+1})$ be a pair of complete transitions of the zero-delay sequence $\mathbf{S}_o$ computed by the switch-level simulation algorithms given in Chapter 4. Also, suppose that $\mathbf{n}_o$ is not an ioput node of a PTB. Let $C_o=CAP(\mathbf{n}_o)$ denote the lumped capacitance from the output node to ground. Let $\mathbf{R}_D$ and $\mathbf{R}_L$ be the device resistances of the driver and load transistors of the inverter, respectively. Let us first consider the case $\mathbf{x}=1$. In this case the pair $(0,\mathbf{u},\mathbf{k}_j)$, $(\mathbf{u},1,\mathbf{k}_{j+1})$ must have been in the sequence at the input node of the inverter. We model this as a type "0" situation in a primitive 1 with $\Delta_{in}=(\mathbf{k}_{j+1}-\mathbf{k}_j)\times\mathbf{h}_{min}$. We then compute $\sigma_1=(R_DC_o)/(\eta KP)$ and the scale factor $\alpha=\sigma_1/\sigma_{1S}$. Let $\Delta\tau_1$ and $\Delta\tau_2$ be the inertial and fall delay values obtained from the delay tables for the type "0" case in a standard primitive 1 corresponding to the input slew rate of Δ_{in}/α. We then compute $\mathbf{k}'_j=\mathbf{k}_j+\alpha\Delta\tau_1/\mathbf{h}_{min}$ and $\mathbf{k}'_{j+1}=\mathbf{k}'_j+\alpha\Delta\tau_2/\mathbf{h}_{min}$ and replace the transition times $\mathbf{k}_j$ and $\mathbf{k}_{j+1}$ by the new times $\mathbf{k}'_j$ and $\mathbf{k}'_{j+1}$, respectively, in the sequence $\mathbf{S}_o$. In the case $\mathbf{x}=0$ we compute the new transition times in the same manner as above, except that we model it as a type "1" situation in a primitive 1 and compute σ_1 with $\mathbf{R}_D=\mathbf{R}_L/\delta_S$.

We now consider a PTB consisting of a single pass transistor. The only situation in which we will use the delay operator is when one node among the drain and source nodes is a normal node and the other is either a pullup node or a node of input strength. Without loss of generality we assume that the source node is the normal node with a capacitance C_2. Consider a certain phase in the simulation of the PTB and let the complete pair of transitions $(\mathbf{x},\mathbf{u},\mathbf{k}_j)$, $(\mathbf{u},\neg\mathbf{x},\mathbf{k}_{j+1})$ be discovered at the source node during this phase. Let us first consider the state of the gate node to be fixed at 1 during this phase. Then clearly the same pair of transitions must have occurred at the drain node during this phase. If the drain node is of input strength, then this is modeled as a primitive 3 with type "0" if $\mathbf{x}=0$ and type "1" if $\mathbf{x=1}$. In either case the delay values for this nonstandard primitive are computed with $\Delta_{in}=(\mathbf{k}_{j+1}-\mathbf{k}_j)\mathbf{h}_{min}$ and $\sigma_3=(R_PC_2)/(\eta KP)$ where R_P is the resistance of the pass transistor. If the drain node is of pullup strength then let R_D and R_L denote the resistances of the driver and load transistors in the corresponding inverter and let C_1 be the capacitance at the drain node. If $\mathbf{x=1}$, we model this as a type "0" situation in primitive 5 and compute Δ_{in} and σ_5 as in the case of primitive 3, shown above. In addition, we compute $\beta=R_D/R_P$ and $\gamma=C_1/C_2$. If $\mathbf{x=0}$, we model this as a type "1" situation in primitive 5 and compute the same parameters as before, except that, $\beta=R_L/(\delta_S R_P)$. In either case we can alter the transition times $\mathbf{k}_j$ and $\mathbf{k}_{j+1}$ by computing the delay values for the appropriate nonstandard primitives. We now consider the case when the gate node of the pass transistor is in the $\mathbf{u}^*$ state in the phase. By definition, there must be a transition $(\mathbf{u,1,k}_j)$ at the gate node. Let the transition time of the previous transition at the gate node be $\mathbf{k}_i$, where $\mathbf{k}_i<\mathbf{k}_j$. If the drain node is of input strength we model this as a primitive 2 with type "0" if $\mathbf{x=1}$ and type "1" if $\mathbf{x=0}$. If the drain node is of pullup strength, we model this as a primitive 4 with type "0" if $\mathbf{x=0}$ and type "1" if $\mathbf{x=1}$. In all these situations we compute $\Delta_{in}=(\mathbf{k}_j-\mathbf{k}_i)\mathbf{h}_{min}$ and the other parameters as in the previous case and compute the delay values for the appropriate nonstandard primitive.

We have thus defined the delay operator for inverters and PTB's consisting of single pass transistors. In the case of a general MFB, we describe a mapping technique that maps the MFB into an equivalent inverter and use the delay operator on the inverter. In the case of a general PTB, we describe a mapping technique based on the use of the Elmore time constant [46], which maps a component (or a supercomponent) occurring in a phase during the simulation of the PTB into an equivalent single pass transistor driving some equivalent load capacitance. We can then use the delay operator, defined above, on this single equivalent pass transistor.

Consider an MFB with output node $\mathbf{n}_o$, and a load transistor of resistance R_L. Suppose $C_1=CAP(\mathbf{n}_o)$ is the capacitance at the output node of the MFB. Now, let us consider the case when a complete pair of zero-delay transitions $(\mathbf{0,u,k}_j)$, $(\mathbf{u,1,k}_{j+1})$ occurs at the output node. We then map the

MFB into an equivalent inverter driving the capacitance C_1, with load transistor having a resistance R_L and a driver transistor with resistance R_L/δ_S, where δ_S is the standard inverter ratio. If $(1,u,k_j)$, $(u,0,k_{j+1})$ is the sequence of transitions at the input node of the equivalent inverter, then $(0,u,k_j)$, $(u,1,k_{j+1})$ would be the zero-delay sequence at the output node of such an inverter. Thus, the two configurations are zero-delay equivalent. We assume that these two are also delay-equivalent and obtain new transitions k'_j and k'_{j+1} by using the delay operator on the inverter and treat these as the new transition times at the output node n_o of the MFB. Let us then consider the other case when the zero-delay transitions $(1,u,k_j)$, $(u,0,k_{j+1})$ occur at the output node of the MFB. In this case we first construct a network of resistances with a resistance of value $=$RES(m_i) between the drain and source nodes of a driver transistor m_i if its gate node is at the 1 state in the interval $(k_{j+1},k_{j+1}+1)$. Let R_{eq} denote the equivalent resistance between n_o and ground in such a network. Let C_{eq} denote the sum of all capacitances at the internal nodes of the above network and $C_L=C_1+C_{eq}$ denote the total capacitance obtained by lumping all the internal node capacitances on the output node. We then map the MFB into an equivalent inverter driving a net capacitance C_L with a driver transistor of resistance $R_D=R_{eq}$ and load transistor of resistance $R_L=\delta_S R_{eq}$. The sequence at the input node of the equivalent inverter would then be $(0,u,k_j)$, $(u,1,k_{j+1})$. We have two zero-delay equivalent configurations once again and we define the delay operator on the MFB to be the delay operator on the equivalent inverter. We illustrate the mapping technique with an example shown in Figure 5.6(a). In this case, the zero-delay sequence at the output node n_o is $(1,u,100)$, $(u,0,200)$. In the time interval $(200,201)$, we see that the signals at the gates of transistors m_1,m_3,m_4, and m_5 are each in the 1 state. Hence, we obtain the resistive network as shown in Figure 5.6(b), with $R_i=$RES(m_i), and compute the equivalent impedance R_{eq}. The equivalent inverter, shown in Figure 5.6(c), consists of a driver with resistance R_{eq}, a load with resistance $\delta_S R_{eq}$. The signal at the gate terminal of the driver is $(0,u,100)$, $(u,1,200)$ and the effective load capacitance at the output node of the inverter is the sum of the node capacitances at nodes n_o, n_1, and n_2 in the original MFB as shown in Figure 5.6(c).

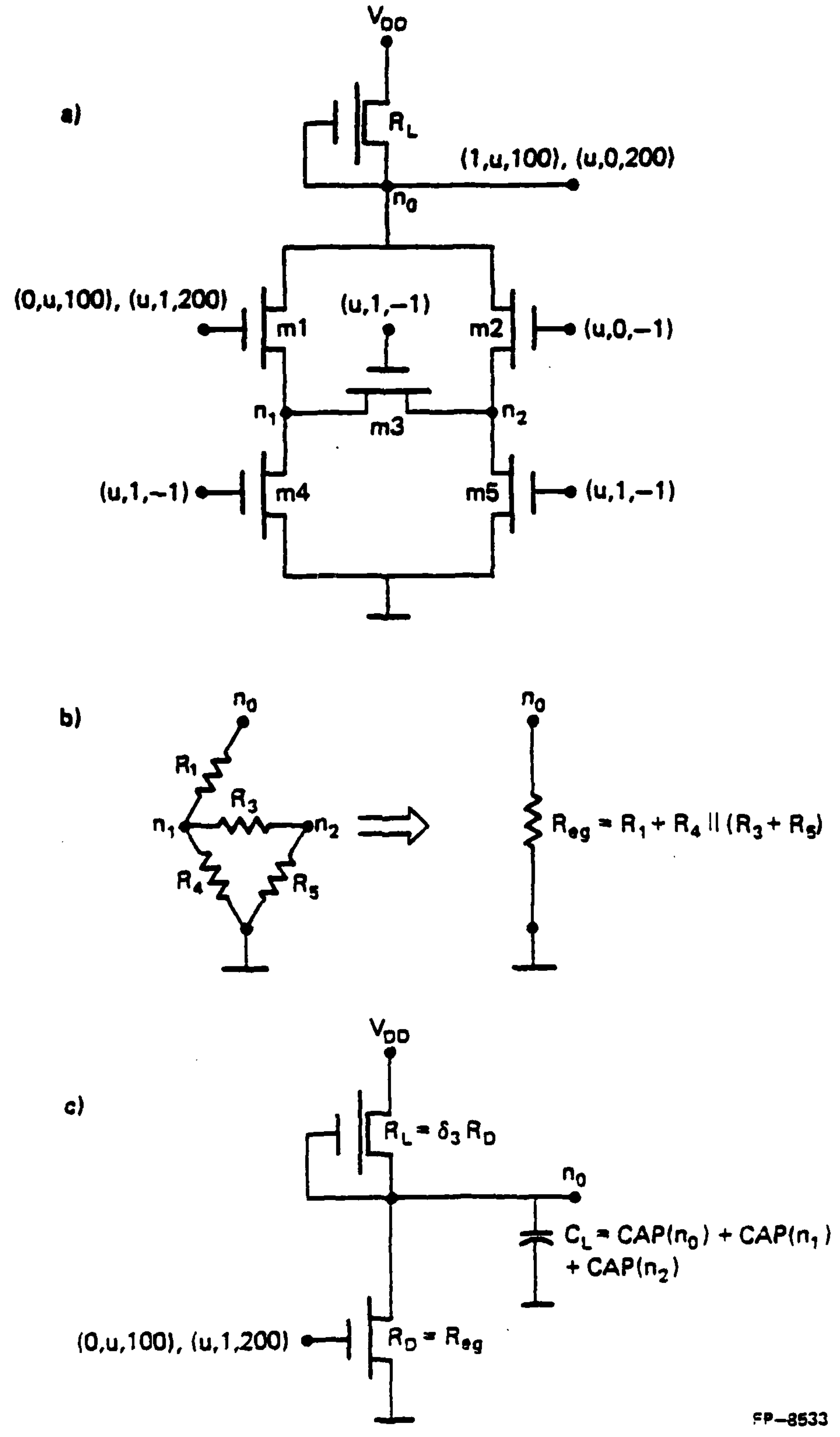


Figure 5.6(a): An example of an MFB
(b): The corresponding resistive network
(c): The equivalent inverter

We now digress a little to discuss the implementation of an algorithm to find the equivalent conductance between two terminals **a** and **b** in a network of conductances (or resistances). We can treat such networks as weighted graphs with each edge having a weight equal to the corresponding resistance and hence can use the terminology we developed for graphs for networks as well. Any node other than **a** or **b** in the network is an *internal node*. Clearly, any set of parallel conductances can be replaced by a single conductance equal to the sum of the parallel conductance. We define this process as the *simplification* of the network. Now consider an internal node of degree 2 in the network. We can eliminate this node from the network by replacing the conductances G_1 and G_2, connected to it by a conductance of value $G_1G_2/(G_1+G_2)$ between the two nodes adjacent to it. It must be noted that eliminating such a node does not change the equivalent conductance between the nodes **a** and **b** in the network. We now extend the notion of eliminating an internal node to nodes of degree $k \geqslant 2$. Let n_0 be an internal node of degree $k \geqslant 2$ in a simplified network and let $n_1, n_2, \ldots, n_k$ be its adjacent nodes. Let G_i be the conductance between n_0 and n_i for each $i=1,2,\ldots,k$. We then define the elimination of n_0 from the network to be a new network without n_0 with a conductance G_{ij} between each pair of nodes n_i and n_j originally adjacent to n_0, such that $G_{ij}=G_iG_j/G_{tot}$, where $G_{tot}=\sum_{i=1}^{k} G_i$ is the sum of all the conductances connected to n_0 in the old network.

Theorem 5.1 : The elimination of an internal node from a simple network does not change the equivalent conductance between the nodes **a** and **b** in the network.

Proof : Let n_0 be an internal node of degree $k \geqslant 2$ and let $n_1, n_2, \ldots, n_k$ be its adjacent nodes. Let I_i denote the current flowing through G_i from n_i to n_0 in the network for each $i=1,2,\ldots,k$, as shown in Figure 5.7(a). If for each $i=1,2,\ldots,k$ we can show that the sum of the currents flowing away from n_i through the all the conductances G_{ij}, $j=1,2,\ldots,k$ $j \neq i$ in the new network is equal to I_i, then we are clearly done with the proof. To this end, suppose v_i denotes the voltage at node n_i for each $i=0,1,\ldots,k$. Then $I_i=G_i(v_i-v_0)$ and $\sum_{i=1}^{k} I_i = 0$. Therefore,

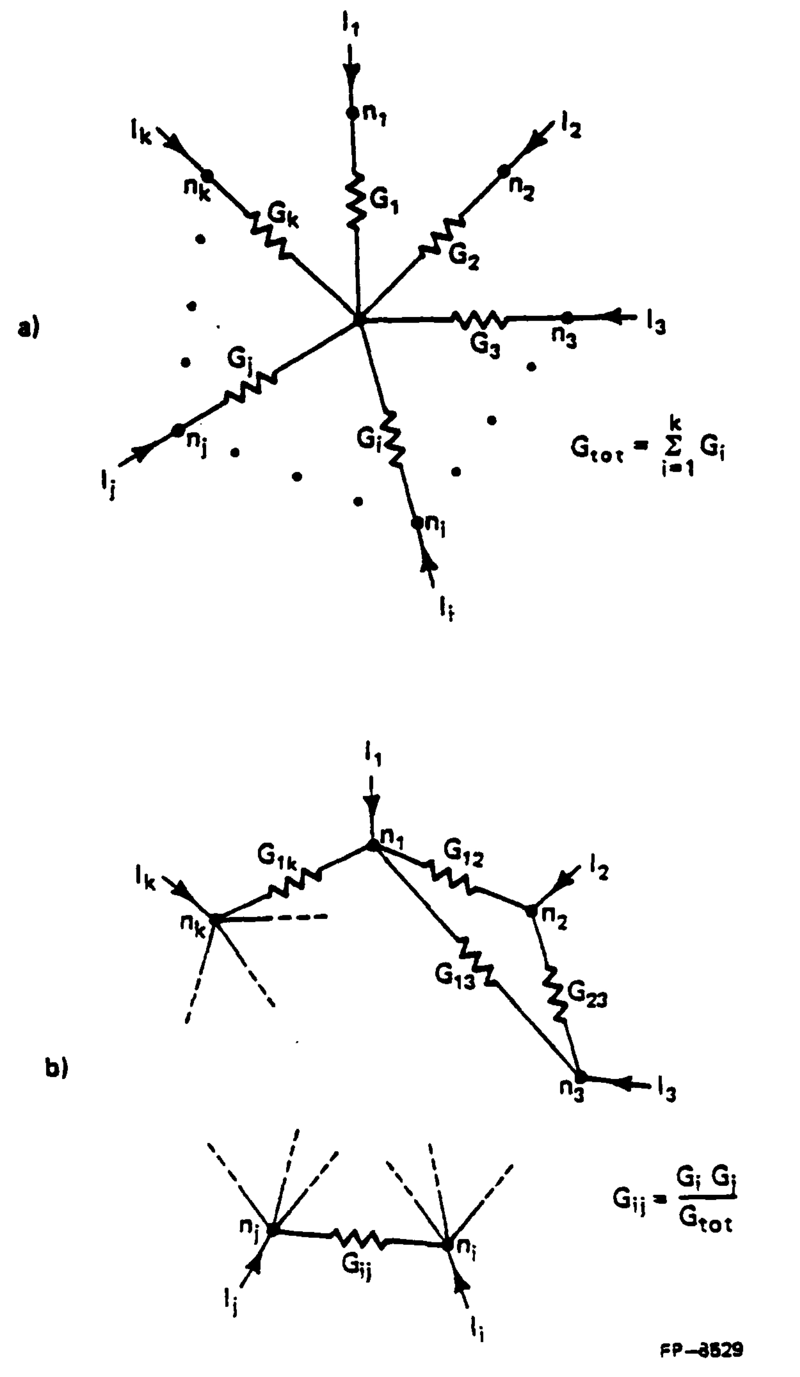


Figure 5.7(a): An internal node, n_0, in a conductance network
(b): The network obtained after eliminating n_0

$$v_0 = \frac{\sum_{i=1}^{k} G_i v_i}{G_{tot}}$$

where $G_{tot} = \sum_{i=1}^{k} G_i$. On substituting this value for v_0 in the previous equation, we get for each $i=1,2,\ldots,k$,

$$G_i v_i - G_i \frac{\sum_{j=1}^{k} G_j v_j}{G_{tot}} = I_i$$

which on simplification gives

$$\sum_{j \neq i} G_{ij}(v_i - v_j) = I_i.$$

Now the above equation is valid for each $i=1,2,\ldots,k$ and, furthermore, its left-hand side is precisely the total current leaving n_i through the conductances G_{ij}, $j=1,2,\ldots,k$ $j \neq i$. The network obtained after eliminating n_0 is shown in Figure 5.7(b). Hence the proof is completed. □

Our algorithm to compute the equivalent conductance G_{ab} between two terminals a and b in a network of resistances can now be described as follows:

1) Simplify the network, i.e., replace all conductances in parallel by a single conductance equal to the sum of the parallel conductances.

2) Pick an internal node of smallest degree in the existing simple network and eliminate it from the network.

3) Simplify the resulting network.

4) If there is an internal node in the existing network, then go to step 2. Otherwise, set G_{ab} to be the conductance between a and b in the final network and STOP.

Notice the similarity between this algorithm and Algorithm 4.2 used to compute the zero-delay sequences at the output nodes of an MFB. In fact, both these algorithms can be run in parallel on the same data base used for representing graphs. It must also be noted that the above algorithm would still work if we had picked any internal node as the next candidate for elimination However, we, pick the node with the smallest degree for the same reasons as explained in Algorithm 4.2. This completes our discussion on the implementation of the algorithm to compute the equivalent conductance between two terminals in a network of resistances.

We now describe the delay operator for a general PTB. We begin by introducing the notion of the Elmore time constant [46] in an RC-tree. A graph $\mathbf{T}$ is a *tree* if it is connected and has no cycles. In each tree, we can focus our attention on a special vertex called the *root* of the tree. If a vertex $\mathbf{a}$ is a root of a tree $\mathbf{T}$, then $\mathbf{T}$ is said to be rooted at $\mathbf{a}$, denoted by $\mathbf{T}_{\mathbf{a}}$. In any tree, there is a unique path from the root to any other vertex in the tree (in fact, there is a unique path between any two vertices in a tree). We say that a network composed of resistances and capacitances forms an *RC-tree* if the subnetwork of resistances, when viewed as a weighted graph, forms a tree and there is a capacitance from each node of the network to ground. Note that all capacitors in such a network are grounded, i.e., there are no floating capacitors. Consider an RC-tree rooted at node $\mathbf{n}_0$ and let $\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_p$ be the rest of the nodes. Let $\mathbf{C}_i$ denote the capacitance from node $\mathbf{n}_i$ to ground, for each $\mathbf{i}=1,2,\ldots,\mathbf{p}$. Let $\mathbf{P}_i$ denote the unique path from the root $\mathbf{n}_0$ to the node $\mathbf{n}_i$ and let $\mathbf{P}_{ij}=\mathbf{P}_i \cap \mathbf{P}_j$ denote the portion of the path between the root and $\mathbf{n}_i$ that is common to that between the root and $\mathbf{n}_j$. Let $\mathbf{R}_{ij}$ denote the sum of all the resistances in $\mathbf{P}_{ij}$. If $\mathbf{P}_{ij}=\emptyset$, then $\mathbf{R}_{ij}=0$. We can now associate a time constant τ_i, known as *Elmore time constant* for each node $\mathbf{n}_i$ in the RC-tree, defined as

$$\tau_i = \sum_{j=1}^{p} \mathbf{R}_{ij}\mathbf{C}_j.$$

Without loss of generality, we need only consider rooted trees in which the root vertex has degree 1, since if the root vertex has degree $\mathbf{k} > 1$, then we can split this vertex and obtain $\mathbf{k}$ subtrees, each rooted at a vertex of degree 1. As far as computing Elmore time constants is concerned, we need only consider the subtree containing the node for which the time constant is to be computed since the node capacitances in the other subtrees have no effect on its computation. Let us, therefore, consider an RC-tree rooted at node $\mathbf{n}_0$ and let $\mathbf{R}_1$ be the (unique) resistance connected to $\mathbf{n}_0$. An example of such a network is shown in Figure 5.8. Then for each node $\mathbf{n}_i$ we define an *Elmore equivalent capacitance* $C_{eq,i}$ to be the ratio of the Elmore time constant τ_i to the resistance $\mathbf{R}_1$, i.e., $C_{eq,i}=\tau_i/\mathbf{R}_1$. For the node $\mathbf{n}_1$ in the network in Figure 5.8, the values for the Elmore time constant and equivalent capacitance are

$$\tau_1=\mathbf{R}_1(C_1+C_2+C_3+C_4+C_5+C_6+C_7)$$

$$C_{eq,1}=C_1+C_2+C_3+C_4+C_5+C_6+C_7$$

while for node $\mathbf{n}_7$ they are

$$\tau_7=\mathbf{R}_1(C_1+C_2+C_3+C_4+C_5+C_6+C_7) + \mathbf{R}_3(C_3+C_4+C_6+C_7) + \mathbf{R}_4(C_4+C_7) + \mathbf{R}_7C_7$$

$$C_{eq,7}=C_1+C_2+(1+\frac{\mathbf{R}_3}{\mathbf{R}_1})C_3+(1+\frac{\mathbf{R}_3+\mathbf{R}_4}{\mathbf{R}_1})C_4+C_5+(1+\frac{\mathbf{R}_3}{\mathbf{R}_1})C_6+(1+\frac{\mathbf{R}_3+\mathbf{R}_4+\mathbf{R}_7}{\mathbf{R}_1})C_7.$$

Let us now consider a phase in the simulation of a general PTB. Let **O** be a component of the graph that is constructed in the first part of the simulation in this phase. The only kinds of components on which we will be using the delay operator are those containing exactly one strongest node, and that node being of **input** or **pullup** strength. The other kinds of components would lead to conflicts or charge sharing. Therefore, let **O** be a component with the strongest node $\mathbf{n}_0$ and let $\mathbf{n}_1,\mathbf{n}_2,\ldots,\mathbf{n}_p$ be the rest of the nodes in the component. We then construct an RC-network from **O** by replacing each edge by a resistance equal to the resistance of the corresponding pass transistor and a capacitance C_i=CAP($\mathbf{n}_i$) from each node $\mathbf{n}_i$ to ground. We first simplify the network and then obtain a *spanning* RC-tree, **T**, from the network. By a spanning tree of a graph, we mean a subgraph which is a tree and includes all the nodes of the original graph. The fact that every connected graph has a spanning tree is a standard result in graph theory, the proof of which can be found in almost any textbook on the subject, such as [50]. For each node $\mathbf{n}_i$, $\mathbf{i}=1,2,\ldots,\mathbf{p}$ we compute the delays for a complete transition in its node sequence as follows. Let $\mathbf{R}_1$ be the unique resistance connected to the root $\mathbf{n}_0$ in the tree **T**. In case the degree of $\mathbf{n}_0$ is $\mathbf{k}>1$, we then split the node $\mathbf{n}_0$ and consider the rooted subtree containing $\mathbf{n}_i$. We begin by computing the Elmore equivalent capacitance $C_{eq,i}$ at this node, which involves the computation of the Elmore time constant. We then construct an equivalent circuit with a single pass

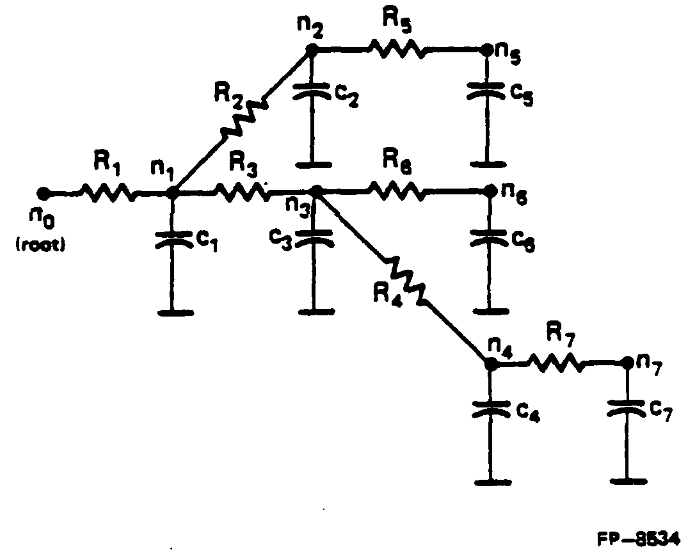


Figure 5.8 : An RC-Tree rooted at n_0

transistor of resistance $\mathbf{R}_1$ with drain node $\mathbf{n}_0$ and source node $\mathbf{n}_i$. The capacitance at node $\mathbf{n}_0$ is $\mathbf{CAP}(\mathbf{n}_0)$ itself, while the capacitance at the source node of this equivalent pass transistor is $C_{eq,i}$. If $\mathbf{n}_0$ is of **input** strength, then this is a nonstandard primitive 3. If $\mathbf{n}_0$ is of **pullup** strength, then we replace the corresponding MFB by its equivalent inverter and treat the whole configuration as a non-standard primitive 5. We then obtain the new transition times for node $\mathbf{n}_i$ by applying the delay operator on the equivalent single pass transistor configuration. This process is repeated for each node in the component. In case the node $\mathbf{n}_0$ is of **pullup** strength, we delay the transitions in its sequence by lumping all the capacitances in the RC-network at $\mathbf{n}_0$ and reduce the resulting configuration to a non-standard primitive 1, using the mapping technique that maps an MFB into an equivalent inverter.

Let us now consider a supercomponent **SC** in the second part of the phase simulation of a PTB. We will only consider the situation when **SC** has only one strongest component and that such a component has only one strongest node. The other situations lead to conflicts or charge sharing and hence are not handled by the delay operator. We will, first, restrict ourselves to the case when **SC** has only one edge, say $\hat{e}$. Let $\mathbf{O}_0$ and $\mathbf{O}_1$ be the two components joined by $\hat{e}$ and let $\mathbf{R}_P$ denote the resistance of the pass transistor corresponding to this edge. We define *contraction* of a component to be collapsing all the vertices of the component into a single node with capacitance equal to the sum of all the node capacitances in the component. The strength of this node is the strength of the component. Without loss of generality let us assume that $\mathbf{O}_0$ is the stronger component. Hence, we will be interested only in obtaining delay values for transitions at nodes in component $\mathbf{O}_1$. Let $\mathbf{n}_1$ be the node (drain or source) of the pass transistor corresponding to $\hat{e}$ in the component $\mathbf{O}_1$. We begin by obtaining a spanning tree $\mathbf{T}_1$ of $\mathbf{O}_1$ that is rooted at $\mathbf{n}_1$. Let $\mathbf{n}_0$ be the strongest node in $\mathbf{O}_0$. We contract the component $\mathbf{O}_0$ into a single node, which we will still call $\mathbf{n}_0$. We then modify the tree $\mathbf{T}_1$ by including the node $\mathbf{n}_0$ and joining it to $\mathbf{n}_1$ by an edge $\hat{e}$. We then declare the root of the new tree $\mathbf{T}$ to be the node $\mathbf{n}_0$. We then construct an RC-tree rooted at $\mathbf{n}_0$ by replacing each edge of $\mathbf{T}$ by the resistance of its corresponding pass transistor and a capacitance from each node to ground. We can now compute the Elmore equivalent capacitance for each node in this RC-tree. Then for each node in $\mathbf{O}_1$ we consider a single pass transistor with drain node $\mathbf{n}_0$ and its associated capacitance and source node driving the Elmore equivalent capacitance of the node under consideration. This then corresponds to a nonstandard primitive 2 or 4 depending upon whether the node $\mathbf{n}_0$ is of **input** or **pullup** strength, respectively.

The case of a supercomponent **SC** having more than one edge seldom occurs in practice. We shall, however, discuss this situation too for the sake of completeness. We begin by constructing a spanning tree on the components of the supercomponent with the root being the strongest component, say $\mathbf{O}_0$. Let $\mathbf{n}_0$ be the strongest node in $\mathbf{O}_0$. Consider an edge of this tree $\mathbf{e}_k$ joining components $\mathbf{O}_i$ and $\mathbf{O}_j$. Without loss of generality, assume that $\mathbf{O}_i$ is closer to the root than $\mathbf{O}_j$. In this case $\mathbf{O}_i$ is said to be the *father component* and $\mathbf{O}_j$ is the *son component* of $\mathbf{e}_k$, respectively. For each edge $\mathbf{e}_k$, then, we apply the delay operator on the nodes of its son component by contracting all the components present in the path connecting its father component to the root into a single node $\mathbf{n}_0$ and treating $\mathbf{e}_k$ as joining $\mathbf{n}_0$ and the son component. This corresponds to the situation of **SC** having only one edge $\mathbf{e}_k$ and so we can now use the RC-tree technique described in the previous paragraph.

We have, therefore, described the delay operator which could be used to alter the transition times of a complete pair of zero-delay transitions at the output node of any MFB and at normal and pullup nodes of any PTB. There are mainly two steps involved. The first step is to map the MFB or PTB into a nonstandard primitive and the second step is to use time scaling to compute the delay values in nonstandard primitives from those computed for standard primitives. Both these steps could cause timing errors. However, as we shall see in Chapter 7, the switch-level timing estimates generated by this approach are fairly accurate in a variety of NMOS circuits considered.

5.4 Filtering Operation

In this chapter, thus far, we have described a delay operator which alters the transition times in a pair of complete transitions. Thus, if a sequence consists of only a pair of complete transitions, then we can use the delay operator directly on this sequence. In this sequence we will consider the effect of delaying a pair of complete transitions on the subsequent terms of the sequence. As an example, consider an inverter, with the following zero-delay sequence computed at its output node.

$$S_o=(0,u,k_1),(u,1,k_2),(1,u,k_3),(u,0,k_4).$$

This sequence is the result of a compatible and chronological input sequence, and is, therefore, also compatible and chronological. Let us first apply the delay operator to the first pair of transitions and compute the new transition times k'_1 and k'_2. By definition, $k'_1<k'_2$. If $k'_2<k_3$ then we simply apply the delay operator to the second pair also and compute the resulting delayed sequence to be :

$$S_o=(0,u,k'_1),(u,1,k'_2),(1,u,k'_3),(u,0,k'_4).$$

This delayed sequence is compatible and chronological. If, however, $k'_1<k_3<k'_2$, this means that at the time the driver transistor of the inverter starts to turn ON, the output node is still in the **u** state and so the **(u,1)**-type transition cannot occur at the output. Hence we simply compute the delayed output sequence in this case to be :

$$S_o=(0,u,k'_1),(u,0,k_4)$$

which is a partial pair of transitions that would represent a glitch at the output node. Furthermore, if $k_3<k'_1$, then there cannot be any transitions taking place at the output and so the output remains in the **0** state for all time which is represented by the sequence :

$$S_o=(u,0,-1).$$

What we have described above is the example of the filtering operator, which takes the zero-delay sequence as its input sequence and using the delay operator computes an output sequence that provides a better representation of the ternary equivalent of the analog waveform at the node under consideration.

We now describe the filtering operation in general. Consider any sequence **S** of transitions. We mark a term of **S** as "delayed" if the delay operator has been used previously on this term, otherwise, we mark it "undelayed." The subsequence of **S** consisting of all its terms marked "delayed" is called the *delayed* part of **S**. The rest of the sequence is the *undelayed* part. Thus, we can consider any sequence of transitions to be the catenation of its delayed part and its undelayed part. Let us consider **S** as an input sequence to the filtering operator. The output of the filtering operation will then be a sequence $\hat{S}$ which is computed as described below. First, the filtering operator replaces any partial pair $(\mathbf{x,u,k}_i), (\mathbf{u,x,k}_{i+1})$ of transitions in the undelayed part of **S** by two complete pairs $(\mathbf{x,u,k}_i), (\mathbf{u,\neg x,k}_i+1), (\neg\mathbf{x,u,k}_{i+1}-1), (\mathbf{u,x,k}_{i+1})$. This is done by procedure COMPLETE (S) used below. We will also make use of the procedure WINDOW ($S,\mathbf{k}_a,\mathbf{k}_b$) that returns those transitions in **S** occurring between $\mathbf{k}_a$ and $\mathbf{k}_b$. The algorithm that performs the filtering is given below.

Algorithm 5.1

```
procedure FILTER (S)
begin
    $\hat{S} \leftarrow \emptyset$;
    S←COMPLETE (S);
    while there is a transition in S marked "undelayed" do
        begin
            $(x,u,k_i) \leftarrow$ first transition marked "undelayed" in S;
            $(u,\neg x,k_{i+1}) \leftarrow$ next transition marked "undelayed" in S;
            $k'_i,k'_{i+1} \leftarrow$ DELAY $(k_i,k_{i+1})$;
            mark $(x,u,k_i)$ as "delayed" in S;
            mark $(u,\neg x,k_{i+1})$ as "delayed" in S;
            $\hat{S} \leftarrow$ WINDOW $(\hat{S},0,k_i)$;
            y← final value of $\hat{S}$;
            if (y=x) then
                append $(x,u,k'_i), (u,\neg x,k'_{i+1})$ $\hat{S}$;
            else if (y=u) then
                append $(u,\neg x,k_{i+1})$ $\hat{S}$;
            end if
        end
    return $\hat{S}$;
end
```

The sequence of transitions $\hat{S}$ obtained after filtering can easily be verified to be compatible and chronological.

CHAPTER 6

SIMULATING STRONGLY CONNECTED COMPONENTS

In this chapter we discuss the use of a special windowing technique to simulate the MFB's and PTB' within a strongly connected component (SCC). The algorithm presented splits the entire time interval of interest $[0,K]$ into various time slots or windows such that all pairs of signal transitions (both partial and complete) take place entirely within one of these windows. This is achieved by maintaining a sequential list of intervals of transitions which is updated dynamically as the algorithm progresses. The algorithm is, in a sense, event-driven, since only those circuit blocks that are *active* within a window are processed and the *fanouts* of the output nodes of these blocks are scheduled for processing in the future. We begin by reviewing two well-known and classical techniques, namely, the waveform relaxation method and the time-point relaxation method, that could be used to simulate the blocks in the network. We will show that neither of these schemes are entirely suitable in our type of simulation and hence there is a need for the event-driven windowing technique that we will present.

6.1 Waveform Relaxation Versus Time-point Relaxation

Let $\boldsymbol{\Omega}(N,M,\Sigma)$ be a partitioned NMOS network in which the set of blocks Σ is further partitioned into its strongly connected components $\Sigma_1,\Sigma_2,\ldots,\Sigma_\mu$. Let $[0,K]$ denote the time interval of simulation. Suppose the SCC Σ_i is currently scheduled for processing. If Σ_i is a simple SCC then the single block contained in it could be simulated during $[0,K]$ by the algorithms discussed in the previous chapters. Hence, suppose that $\Sigma_i=\{\Omega_1,\Omega_2,\ldots,\Omega_p\}$, where $p\geqslant 2$ and each Ω_j is either an MFB or a PTB.

The blocks within Σ_i could then be simulated using a waveform relaxation iterative scheme WR_SIM described below. Let $\mathbf{R}_i$ be an ordering on the blocks of Σ_i. Without loss of generality we can assume that the blocks of Σ_i are placed according to $\mathbf{R}_i$, i.e., $\mathbf{R}_i(\Omega_j)=\mathbf{j}$ for each $\mathbf{j}=1,2,\ldots,\mathbf{p}$. For any node $\mathbf{n}_k \in \mathbf{N}$ in the network let $\mathbf{S}_k$ denote the most recently computed sequence of transitions or the *present sequence* at the node and let $\hat{\mathbf{S}}_k$ denote the previously computed sequence or the *past sequence* at the node. Also, let $s_k \in \{0,1\}$ denote the *initial state* at node $\mathbf{n}_k$, which is either provided by the user, or is arbitrarily set to $\mathbf{0}$. Let N_i denote the list of all the circuit nodes contained in the blocks within Σ_i. The algorithm begins by setting the present sequence of transitions at any node that has not been previously computed to a constant sequence corresponding to the initial condition at that node for all time $[\mathbf{0},\mathbf{K}]$. The iterative procedure begins by setting the past sequence equal to the present sequence for each circuit node in the SCC. The individual blocks within the SCC are then simulated according to the ordering $\mathbf{R}_i$ over the entire time interval $[\mathbf{0},\mathbf{K}]$ by algorithms described in the previous chapters. In each case the present sequences at the input nodes of a block are taken as the input sequences for simulation and the present sequences at the output nodes of the block are updated after the simulation. The procedure EQUAL then checks for equality between the present sequence and the past sequence during the time interval $[\mathbf{0},\mathbf{K}]$ at each node and returns the value $\mathbf{0}$ if they are found equal and $\mathbf{1}$ if not. Here, two sequences are considered equal if they have the same number of terms and are both type-equal as well as time-equal as defined in Section 4.1 in Chapter 4. The iterations are carried out until both present and past sequences are found equal for each node in the SCC.

Algorithm 6.1

Input : A strongly-connected component Σ_i and an ordering $\mathbf{R}_i$,

such that the blocks within Σ_i are arranged according to $\mathbf{R}_i$.

Output : Sequences of transitions at output nodes of each block within Σ_i.

```
procedure WR_SIM ($\Sigma_i$,$R_i$,0,K)
begin
```

```
    for each node $n_k \in N_i$ do
        begin
            if ($S_k = \emptyset$) then
                $S_k \leftarrow (u, s_k, -1)$;
            end if
        end
    repeat
        for each node $n_k \in N_i$ do
            $\hat{S}_k \leftarrow S_k$;
        for $j \leftarrow 1$ until p do
            if $\Omega_j$ is an MFB then
                MFB_SIM ($\Omega_j$,0,K)
            else if $\Omega_j$ is a PTB then
                PTB_SIM ($\Omega_j$,0,K)
            end if
        ind←0;
        for each node $n_k \in N_i$ do
            begin
                ind←EQUAL ($S_k$,$\hat{S}_k$,0,K);
            end
    until ind=0
end
```

We now discuss several features of the above algorithm. We first consider obtaining an *a priori* ordering $\mathbf{R}_i$ on the blocks of the SCC. Given any such ordering, we define a node to be *initially relaxed* if it is an input node of a block within the SCC and its present sequence has not yet been updated in the current iteration at the time of simulating the block. In the above algorithm, the present sequence of a node gets updated only after simulating the block to which it is an output node. Hence, in the case of an initially relaxed node the blocks in its fanin list are ordered after the blocks in its fanout list. Given an ordering on the vertices of a digraph, we say that an arc is a *forward arc* if its tail vertex appears before its head vertex in the ordering ; otherwise, the arc is said to be a *feedback arc*. If we consider the vertices of the derived digraph, as defined in Chapter 3, corresponding to the blocks within the SCC Σ_i, then any ordering $\mathbf{R}_i$ would result in a set of feedback arcs. Furthermore, the number of feedback arcs produced by $\mathbf{R}_i$ is an upper bound on the number of initially relaxed nodes due to $\mathbf{R}_i$. Clearly, the best choice for $\mathbf{R}_i$ is one which results in the least number of initially relaxed nodes since this would speed up the convergence of the above algorithm. However, this corresponds to finding an ordering that results in the minimum number of feedback arcs, which is an NP-Complete problem

[52,53,57]. Therefore, the choice of the *a priori* ordering $\mathbf{R}_i$ affects the speed of convergence of the above algorithm and finding the best ordering, in this respect, turns out to be a difficult problem from the computational complexity point of view. This is one of the drawbacks of the waveform relaxation scheme.

Another aspect that needs to be considered is that the number of iterations turns out to be proportional to the number of transitions at the various circuit nodes in certain circuits such as the ring oscillator. This is also one of the major drawbacks in the waveform relaxation method WRM [9]. Finally, this scheme requires storing two sequences of transitions for the entire time interval $[\mathbf{0,K}]$ at each node which could be a considerable amount of computer storage for large SCC's. In spite of all these drawbacks, this scheme could still be used in our type of switch-level simulation since it is easy to implement and is compatible with the delay and filtering operations. In Appendix II, we will discuss the problem of finding an optimum ordering that results in the minimum number of feedback arcs in a digraph. We also discuss an algorithm, proposed by Younger [60], that finds such an ordering in case of a general digraph. This would then be the *a priori* ordering $\mathbf{R}_i$ used in Algorithm 6.1.

An alternative approach is to use the time-point relaxation method for the simulation of the entire partitioned network $\mathbf{\Omega(N,M,\Sigma)}$. In this approach there is no need to handle blocks within an SCC in a special way since the scheme is *event-driven*, as discussed in Section 2.3.1, and is used in several digital simulators [13,17,19,25,26]. In order to use this approach in our type of simulation, we could define an *event* as a transition $(\mathbf{x,y,k}_i)$ occurring at time $\mathbf{k}_i$. A time queue (TQ) is used to maintain a list of events occurring at different instants of time. If an event $(\mathbf{x,y,k}_i)$ occurs at some node $\mathbf{n}_j$ in the network, then all the blocks in the fanout list of $\mathbf{n}_j$ are processed at time $\mathbf{k}_i$. If on processing a block at $\mathbf{k}_i$, a transition is observed at an output node of the block, then this is defined as a new event, and is scheduled to occur at time $\mathbf{k'}_i > \mathbf{k}_i$. Thus, $\mathbf{k'}_i - \mathbf{k}_i > \mathbf{0}$ is a positive delay in propagating an event occurring at an input node of a block to an output node of the block. It is this feature that makes the use of time-point relaxation particularly attractive for processing blocks within feedback loops.

In our type of switch-level simulation, the emphasis is on generating accurate timing estimates which is possible by using the delay and filtering operations described in Chapter 5. However, the delay operator can only operate on a pair of complete transitions and therefore, events can be propagated through a block only in pairs. Consider an example of an inverter with a sequence $(\mathbf{0,u,k_1})$, $(\mathbf{u,1,k_2})$ at its input node causing a sequence $(\mathbf{1,u,k'_1})$, $(\mathbf{u,0,k'_2})$ at its output node. In order to use the time-point relaxation scheme, we would have to be able to compute the value of $\mathbf{k'_1}$ only with the knowledge of the input event $(\mathbf{0,u,k_1})$. This is however impossible, since the delay operator needs to know the values of both $\mathbf{k_1}$ and $\mathbf{k_2}$ before it can compute $\mathbf{k'_1}$ and $\mathbf{k'_2}$. Furthermore, it is possible to have $\mathbf{k'_2 < k_2}$, which means that the input event $(\mathbf{u,1,k_2})$ causes the output event $(\mathbf{u,0,k'_2})$ at an earlier time, thus violating the basic assumption that one only advances in time in the TQ and never has to backtrack. Therefore, the time-point relaxation method, as such, is not suitable for our type of simulation.

6.2 Event-driven Dynamic Windowing Algorithm

In the previous section we discussed two relaxation methods to simulate the blocks in a network. The first method, namely, the waveform relaxation method, could be used in our type of simulation since it is compatible with the delay and filtering operations, but suffers from several drawbacks in the case of blocks within a strongly connected components. The second method, namely, the time-point relaxation method, is used in several digital simulators, mainly because blocks within strongly connected components do not pose any special problems, but it is found to be incompatible with the delay and filtering operations, and hence, cannot be used, as such, in our type of simulation. In this section we describe a new scheme to handle blocks within a SCC which overcomes most of the above drawbacks in the waveform relaxation method by incorporating some of the ideas of the time-point relaxation method. The main idea is to use the so-called *windowing* technique in the waveform relaxation procedure, as suggested in [11,12], wherein it is shown that the number of iterations is exponentially proportional to the size of the time interval of analysis. This suggests dividing the entire time interval of interest into many time slots or windows so that waveform relaxation can be performed within each window. These waveforms generate initial conditions for the next window and so on. If all the windows have the same size, then there exist an optimum number of windows which minimize the total number of iterations (and hence the total CPU time for analysis) as shown in [11].

The choice of windows, however, is very crucial in our type of switch-level simulation since the initial states at each node for each window must be the steady states **0** or **1** in order to obtain good timing through the delay operator, and to perform the filtering operation successfully. This appears to be a no-win situation since deciding on the placement of windows seems to require a prior knowledge of the digital waveform (or sequences of transitions) at each circuit node within the SCC. Here we describe a successful solution to this problem by using a sequential list of time intervals which is dynamically updated as the algorithm progresses. In addition, the new scheme is event-driven, and therefore requires no *a priori* ordering of blocks within a SCC. Before going into the description of the algorithm, a few definitions and notations are needed.

Consider an SCC $\mathbf{\Sigma}_i$ consisting of a set of blocks $\mathbf{\Omega}_1,\mathbf{\Omega}_2,\ldots,\mathbf{\Omega}_p$. Let $\mathbf{EXT}_i$ denote those circuit nodes in the blocks within the SCC for which the node sequences have already been computed. For each circuit node $\mathbf{n}_k$ in the SCC, let $\mathbf{FO}(\mathbf{n}_k)=\mathbf{FOUT}(\mathbf{n}_k)\cap\mathbf{\Sigma}_i$ denote the set of blocks within $\mathbf{\Sigma}_i$ for which $\mathbf{n}_k$ is an input node.

Definition : A *transition interval* for a node is the time interval during which the node is in the intermediate state **u**. Associated with each transition interval **I** for a node $\mathbf{n}_k$ is a fanout list of blocks, denoted by $\mathbf{F(I)}$, which is initially set to $\mathbf{FO}(\mathbf{n}_k)$. Let $\mathbf{a(I)}$ and $\mathbf{b(I)}$ denote the initial and final times of the transition interval **I**.

Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be any two transition intervals. We say that $\mathbf{I}_1<\mathbf{I}_2$ if and only if $\mathbf{b}(\mathbf{I}_1)<\mathbf{a}(\mathbf{I}_2)$. If $\mathbf{I}_1\cap\mathbf{I}_2\neq\emptyset$, then we say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are *incomparable*. We thus have introduced the notion of a partial order "<" on a set of intervals. Let $\mathbf{L}=\{\mathbf{I}_1,\mathbf{I}_2,\ldots,\mathbf{I}_q\}$ be a sequential list of intervals. We say that

L is an *ordered list* if $I_1 < I_2 < \cdots < I_q$. We say that an interval I is *contained* in L if $I \subseteq I_j$ for some $I_j \in L$. Given any interval I and an ordered list of intervals L the following procedure returns an updated ordered list $\hat{L}$ containing the interval I.

```
Input  : An ordered list $L=\{I_1,I_2,\ldots,I_q\}$ of intervals,
         and a new interval I.
Output : A new ordered list $\hat{L}$ containing I.

procedure INCLUDE(I,L)
begin
        $\hat{L} \leftarrow \emptyset$;
        $\eta \leftarrow 0$;
        for j ← 1 until q do
                begin
                        if $I_j < I$ then
                                $\eta \leftarrow 1$;
                                $\hat{L} \leftarrow \hat{L} \bigcup I_j$;
                        else if $I \cap I_j \neq \emptyset$ then
                                $\eta \leftarrow 1$;
                                $I \leftarrow I \bigcup I_j$;
                                $F(I) \leftarrow F(I) \bigcup F(I_j)$;
                        else if $I < I_j$ then
                                if $\eta = 1$ then
                                        $\hat{L} \leftarrow \hat{L} \bigcup I$;
                                end if
                                $\eta \leftarrow 0$;
                                $\hat{L} \leftarrow \hat{L} \bigcup I_j$;
                        end if
                end
        return $\hat{L}$;
end
```

The algorithm for the new dynamic windowing technique can now be described as follows. The ordered set L is initialized to the empty set. Every transition interval at each node in EXT_i is included in L. The set L is altered dynamically as the algorithm progresses. At any stage, we have a partition of the entire time interval $[0,K]$ into windows by taking the final times of the disjoint intervals in L as the boundaries of the windows. The set L plays the role of the time queue (TQ) used in the time-point relaxation method. Here events take place over transition intervals rather than occurring instantaneously. If a transition interval at an input node of a block causes a transition interval at an output node of the block, then the end points of the new interval can be computed by our delay operator. Thus, this new scheme is compatible with our delay and filtering operations.

Algorithm 6.2

```
procedure WIN_SIM ($\Sigma_i$)
begin
    $L \leftarrow \emptyset$;
    for each circuit node $n_k \in EXT_i$ do
        begin
            for each transition interval $I_j$ of $n_k$ do
                begin
                    $L \leftarrow$ INCLUDE ($I_j$,L)
                end
        end
    $K_2 \leftarrow 0$;
    while L is not empty do
        begin
            $I \leftarrow$ first interval in L;
            $K_1 \leftarrow K_2$;
            while F(I) is not empty do
                begin
                    $K_2 \leftarrow b(I)$;
                    $\Omega_r \leftarrow$ first block in F(I);
                    for each output node $n_k$ of $\Omega_r$ do
                        $\hat{S}_k \leftarrow$ WINDOW($S_k$,$K_1$,$K_2$);
                    if $\Omega_r$ is an MFB then
                        MFB_SIM ($\Omega_r$,$K_1$,$K_2$)
                    else if $\Omega_r$ is a PTB then
                        PTB_SIM ($\Omega_r$,$K_1$,$K_2$)
                    end if
                    for each output node $n_k$ of $\Omega_r$ do
                        begin
                            $S'_k \leftarrow$ WINDOW($S_k$,$K_1$,$K_2$);
                            for each transition interval $I_m$ of $n_k$ do
                            begin
                            if $I < I_m$ then
                            $L \leftarrow$ INCLUDE($I_m$,L);
                            else if $S'_k \neq \hat{S}_k$ then
                            $L \leftarrow$ INCLUDE($I_m$,L);
                            end if
                            end
                        end
                    delete the first block from F(I);
                end
            delete the first interval from L;
        end
end
```

The above algorithm to process the blocks within an SCC begins by forming **L** by including each transition interval of each circuit node in $\mathbf{EXT}_i$. The first interval in **L** is chosen as the window of interest. The blocks in its fanout list, which are MFB's and PTB's, are then only for the duration of the present window until the list is empty. Each time a block gets processed, a transition interval in an output node is included in **L** if and only if one of the following two conditions are satisfied:

a) All transitions in the output node occur after the present window, I.

b) The transitions at the output node, occurring during the present window after processing the block, are different from those before processing the block.

After the fanout list for the present window is empty the interval is deleted from **L** and the whole process is repeated until **L** is empty.

Consider the execution of the above algorithm on an SCC Σ_i. After the initialization of **L** by including the transition intervals of the nodes in $\mathbf{EXT}_i$, it could get updated by the inclusion of the transition intervals at the output nodes of the block that has been just simulated. This could alter either the endpoint, K_2, of the present window, or could append a set of blocks to the existing fanout list **F(I)** of the present window. If the latter situation continues, it is possible that a block could reappear in the fanout list of the present window, after it has been deleted before, and is hence resimulated during the present window. We say that an SCC is ***well-behaved*** if, during the execution of Algorithm 6.2, none of its blocks is ever resimulated during the same window.

Thus, in a well-behaved SCC the delay characteristics of the various blocks are such that one does not have to perform any iterations at all. If, however, the SCC is not well-behaved, then the algorithm extends the fanout list of the present window and resimulates the active blocks until convergence is achieved for the duration of the present window. This is equivalent to performing waveform relaxation iterations within the present window. It is possible to conjure up an SCC for which the initial window gets continually extended until it becomes the entire time interval. In this case using the above Algorithm 6.2 becomes equivalent to Algorithm 6.1. However, such a situation is of theoretical interest only, and probably never occurs in practical circuits. Thus, in the worst case, the new dynamic windowing technique performs at least as well as the waveform relaxation method. In fact, the SCC's in several practical circuits considered were all well-behaved, in which case Algorithm 6.2 performs much better than Algorithm 6.1 in all respects. To begin with, there is no need to place the blocks of the SCC in any particular order, since the procedure in Algorithm 6.2 is *event-driven*, i.e., only those blocks that are *active* in a window are processed during that window. Secondly, no iterations are performed in case of a well-behaved SCC, thereby saving considerable amounts of computation time. Finally, the active blocks are processed only during a window (and not for the entire time interval), thus causing a reduction in both computation time and memory space required to store the sequences of transitions.

CHAPTER 7

MOSTIM : IMPLEMENTATION AND PERFORMANCE

The algorithms described in Chapters 3 to 6 have been implemented in a computer program called MOSTIM, a switch-level timing simulator for NMOS circuits. MOSTIM is written in FORTRAN and runs on a VAX 11/780 computer with the UNIX operating system. It has about 9600 lines of FORTRAN code which includes about 5800 lines from the front end of SPICE2G.1. The main flow chart for MOSTIM is shown in Figure 7.1. The NMOS network is described to MOSTIM in the same input description language as SPICE2 [1]. The three overlays MAIN, READIN, and ERRCHK, borrowed from SPICE2G.1, read in the input file describing the network and establish the data base to store the necessary information about the circuit elements, their model parameters, and interconnection, etc. A dynamic memory manager is used to allocate space for each element. The input description language allows the use of a multilevel hierarchy of subcircuits, which is flattened out in the ERRCHK overlay. This overlay also checks for topological errors, such as a node connected to less than two circuit elements and a loop of voltage sources as well as errors in the specifications of the model parameters for the circuit elements. The subroutine PARTITION then partitions the NMOS network into MFB's, PTB's, and SRC's, using algorithms described in Chapter 3 of this thesis. The set of blocks in the partitioned network is then further partitioned into strongly connected components (SCC's) and these are ordered by subroutine ORDER. The subroutine SIMULATION processes the SCC's in the above ordering. If an SCC is simple, then the appropriate subroutine SRC_SIM, MFB_SIM, or PTB_SIM, described in Chapter 4, is used to simulate the block for the entire time interval of interest. If an SCC contains more than one block, then it is simulated by subroutine WIN_SIM, which, in turn, uses subroutines MFB_SIM and PTB_SIM to simulate the individual MFB's and PTB's over windows in time, as described in Chapter 6. The subroutines MFB_SIM and PTB_SIM interact dynamically with

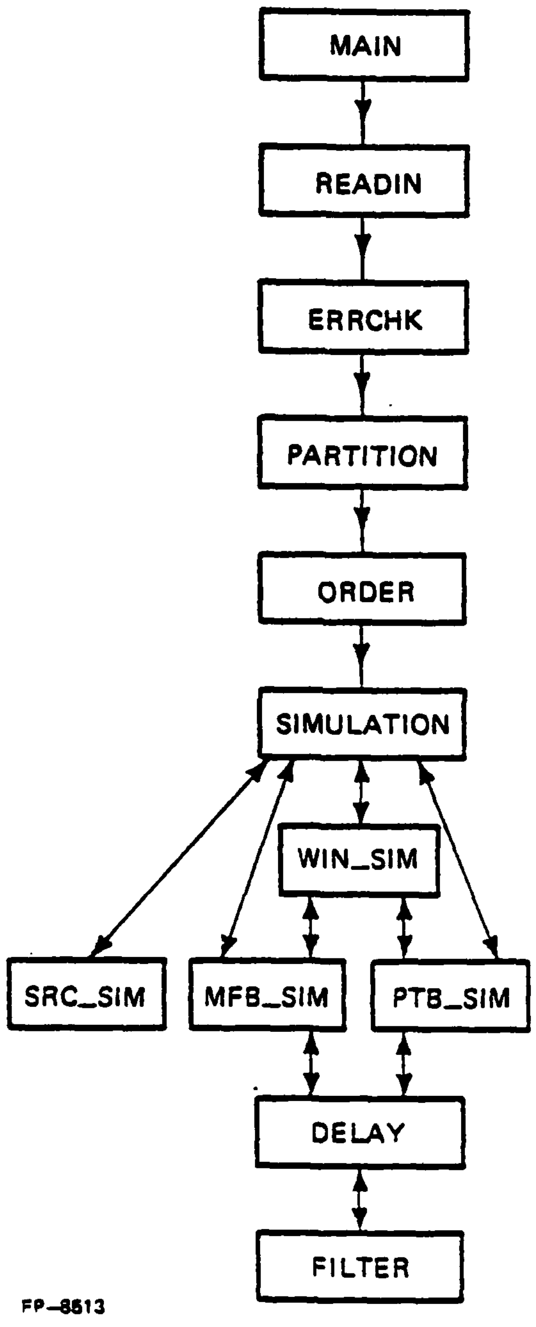


Figure 7.1 : Flow chart for MOSTIM

subroutines DELAY and FILTER, described in Chapter 5, to alter the transition times of the zero-delay sequences produced and filter the resulting delayed sequences. Extensive use of *linked lists* is made throughout the program. These linked lists are implemented in FORTRAN with the help of one-dimensional arrays.

We now evaluate the performance of MOSTIM based on its computational speed (complexity) and the accuracy of its switch-level timing (SLT) estimates. We first evaluate the computational speed by considering several examples. The first example is a combinatorial NAND gate implementation of a one-bit full-adder circuit, shown in Figure 7.2, which was cascaded to produce full-adders from one to four bits. Table 7.1 shows the rate of growth of CPU-time versus the number of transistors. The total CPU-time taken by MOSTIM includes the time taken for partitioning and ordering, and also the time for the switch-level simulation, the delay and filtering operations. The total job times taken by SLATE [3] and SPICE2G.1 [1] are also provided for comparison.

Table 7.1 : The growth-rate of CPU-time of MOSTIM, SLATE, and SPICE2G.1

Adder Bits	Number of Transistors	CPU - Seconds		
		MOSTIM	SLATE	SPICE2G.1
1	33	1.40	61.1	184.0
2	66	2.03	133.2	371.1
3	99	2.55	195.8	556.3
4	132	3.45	252.9	767.0

This table shows that the total time taken by MOSTIM is fairly linear with circuit size and is about 120-200 times faster than SPICE2G.1 and about 40-60 times faster than SLATE. A second example is a chain of identical inverters. Figure 7.3(a) shows a chain of five inverters. Throughout this chapter we

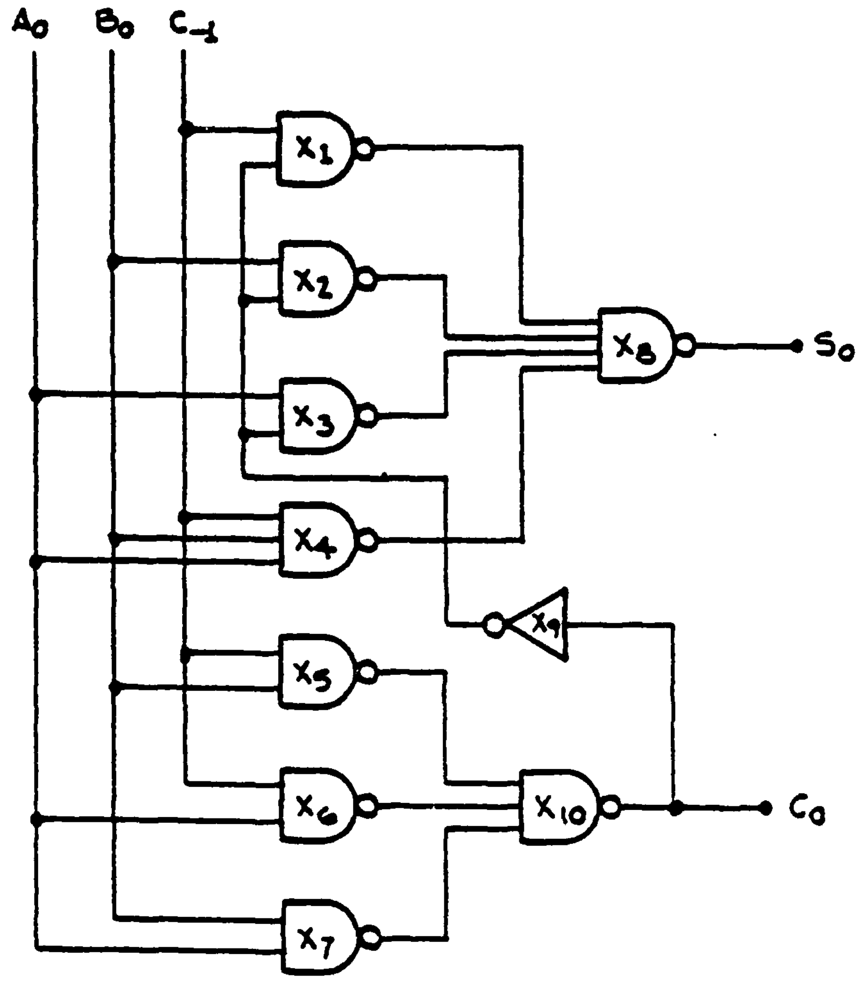
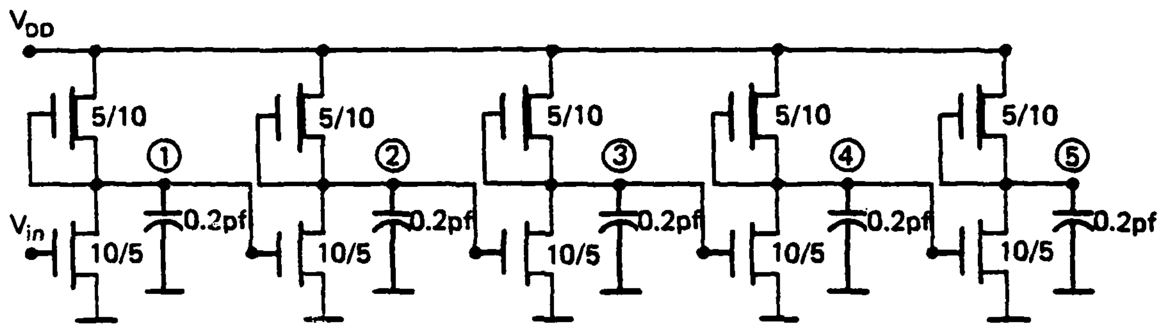


Figure 7.2 : A one-bit combinational full-adder



FP–8520

Figure 7.3(a) : A chain of 5 inverters

will represent analog waveforms produced by SPICE2G.1 with solid lines and the ternary digital waveforms produced by MOSTIM with dotted lines. The waveforms at the output of every fifth inverter in a 50-inverter chain produced by both MOSTIM and SPICE2G.1 are shown in Figure 7.3(b). Table 7.2, below, gives the CPU-times taken by both MOSTIM and SPICE2G.1 for a chain of identical inverters. These values are plotted against the number of inverters in the chain in Figures 7.3(c) and 7.3(d).

Table 7.2 : CPU-times taken by MOSTIM and SPICE2G.1 on a chain of inverters

Number of Inverters	CPU - Seconds	
	MOSTIM	SPICE2G.1
5	0.62	21.63
10	0.87	43.10
15	1.18	70.35
20	1.48	121.83
30	2.05	235.98
50	3.19	645.28

From both of the examples considered above, it can be concluded that the CPU-time taken by MOSTIM grows linearly with circuit size and is around two orders of magnitude faster than SPICE2G.1.

We now consider several examples of NMOS circuits simulated using MOSTIM. A one-bit full-adder circuit with pass transistors used to realize part of the logic is shown in Figure 7.4(a) and a cascaded two-bit adder in Figure 7.5(a). The input and output waveforms in both these circuits are shown in Figures 7.4(b) and 7.5(b), respectively. The presence of a partial pair of transitions in a ternary digital waveform indicates the presence of a *glitch* in the corresponding analog waveform. We classify a glitch as a *major glitch* or a *minor glitch* according to whether or not the glitch crosses a threshold

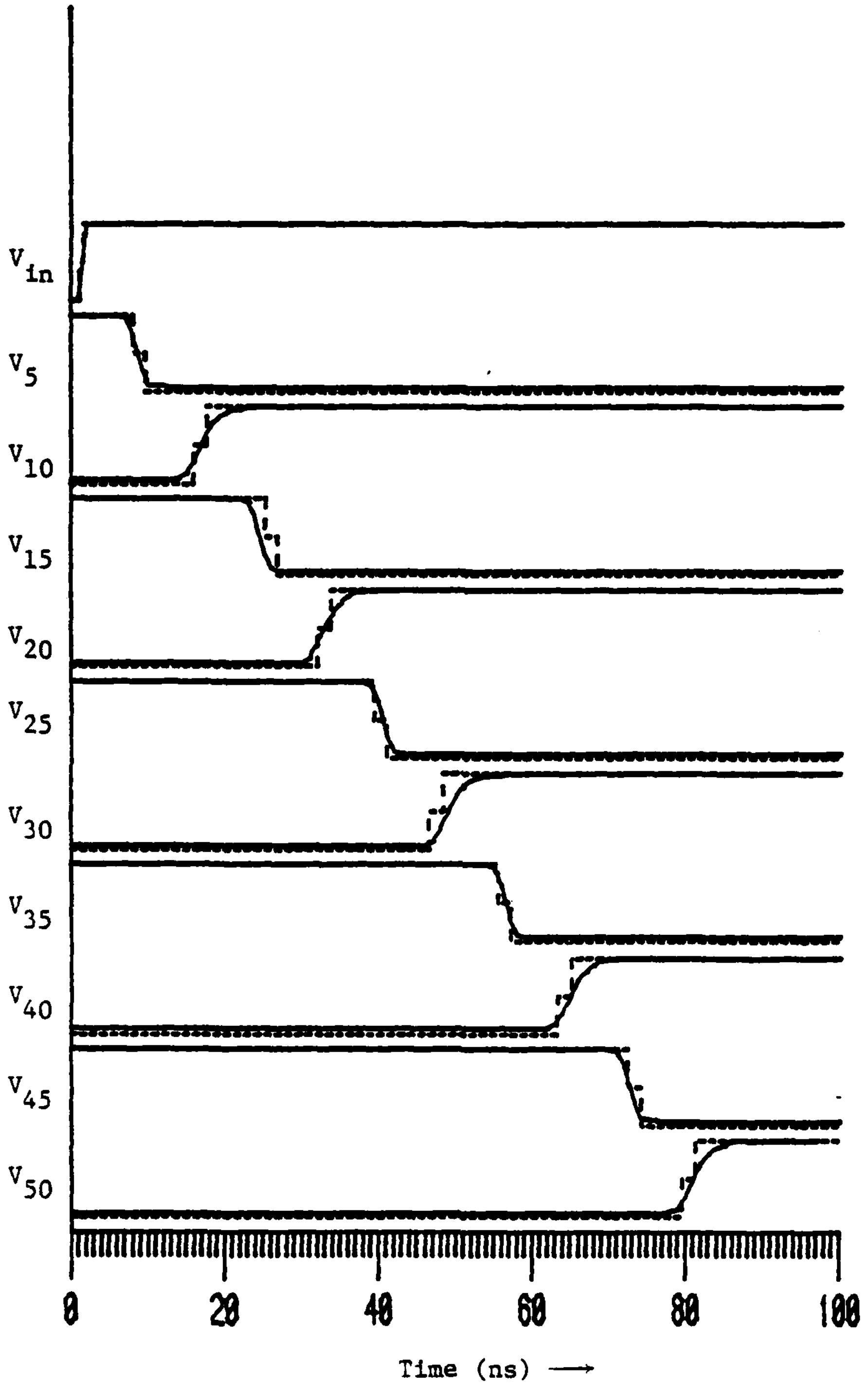


Figure 7.3(b) : Waveforms for a 50-inverter-chain circuit

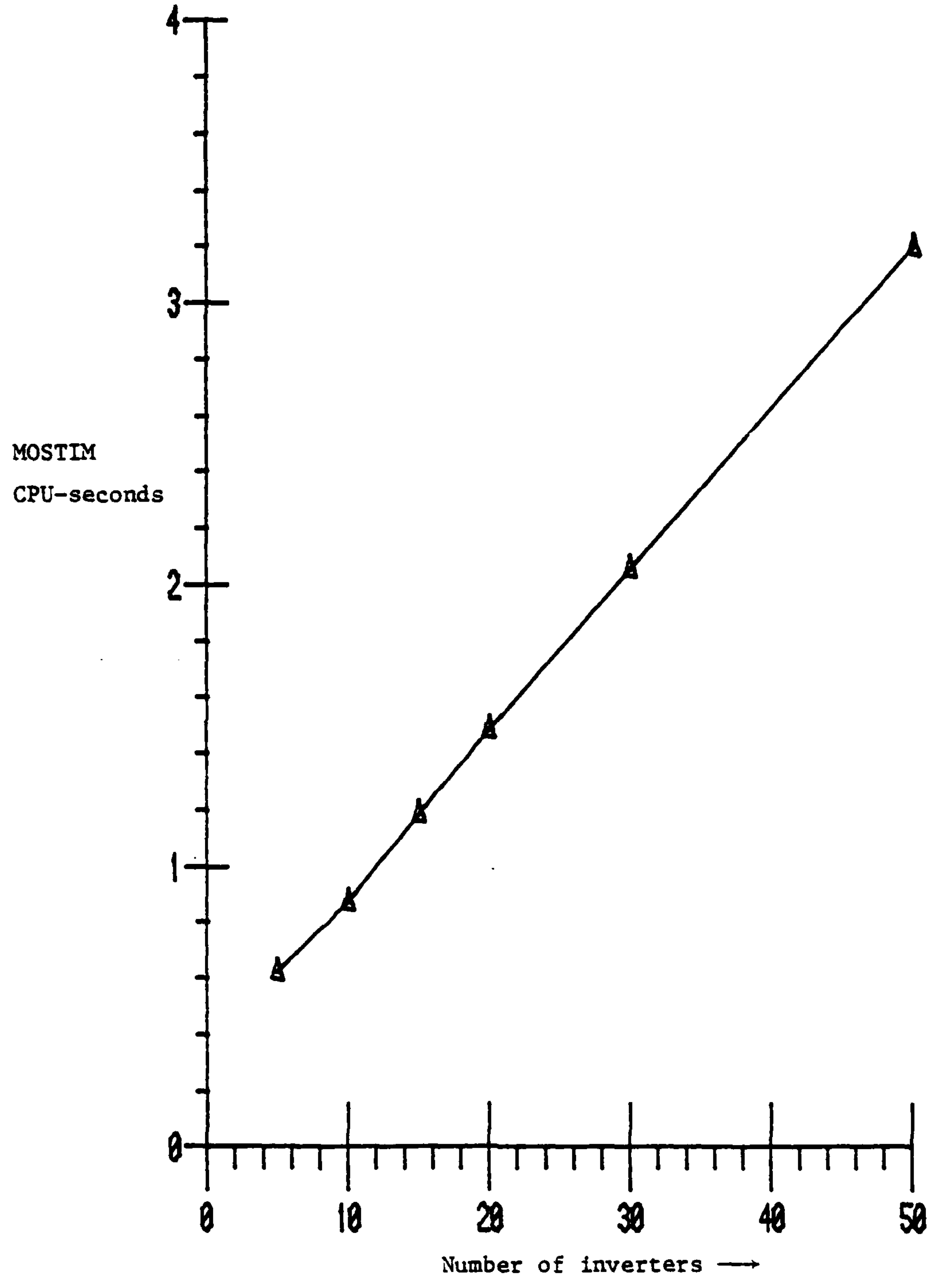


Figure 7.3(c) : CPU-time taken by MOSTIM on a chain of inverters

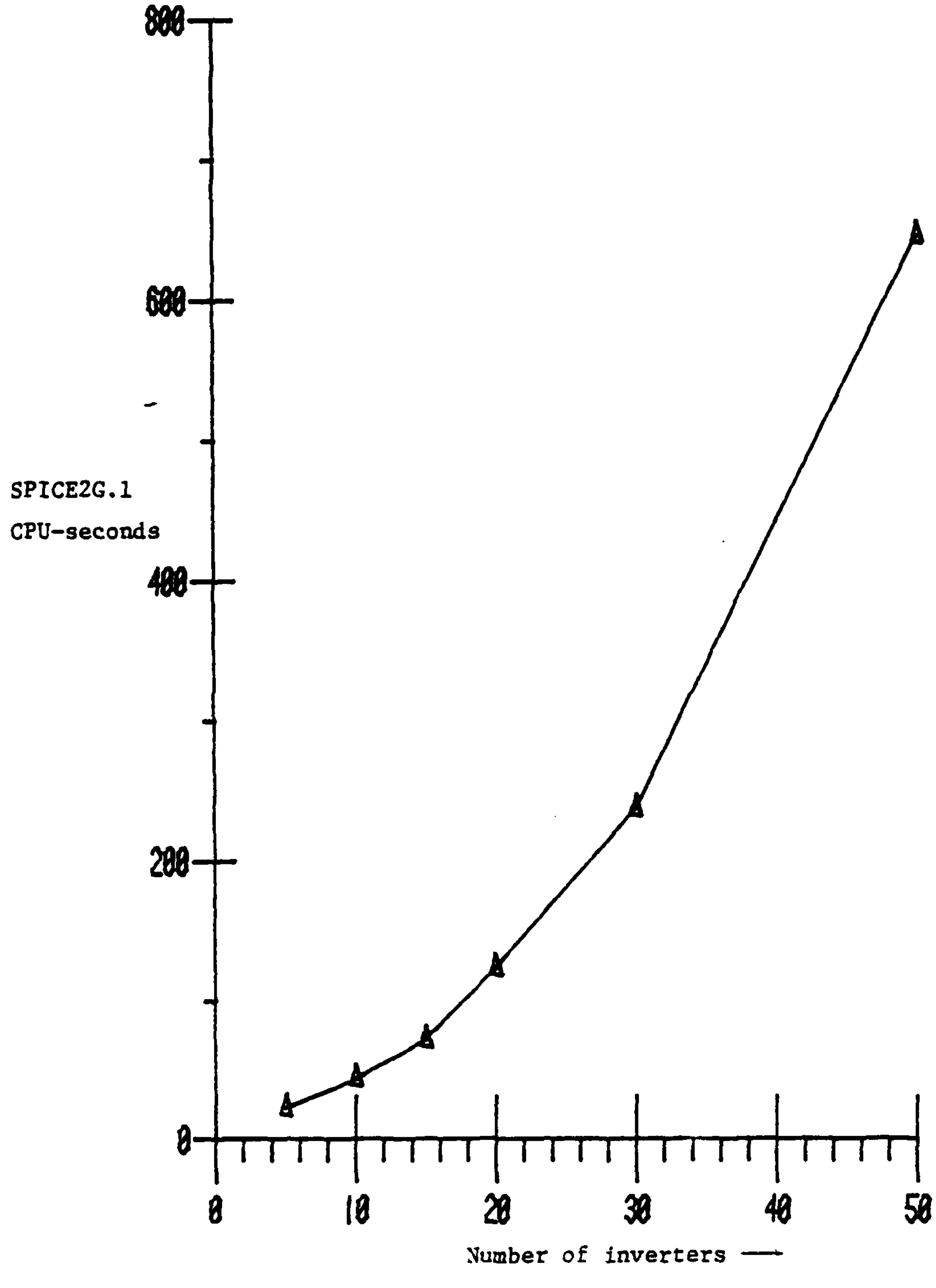


Figure 7.3(d) : CPU-time taken by SPICE2G.1 on a chain of inverters

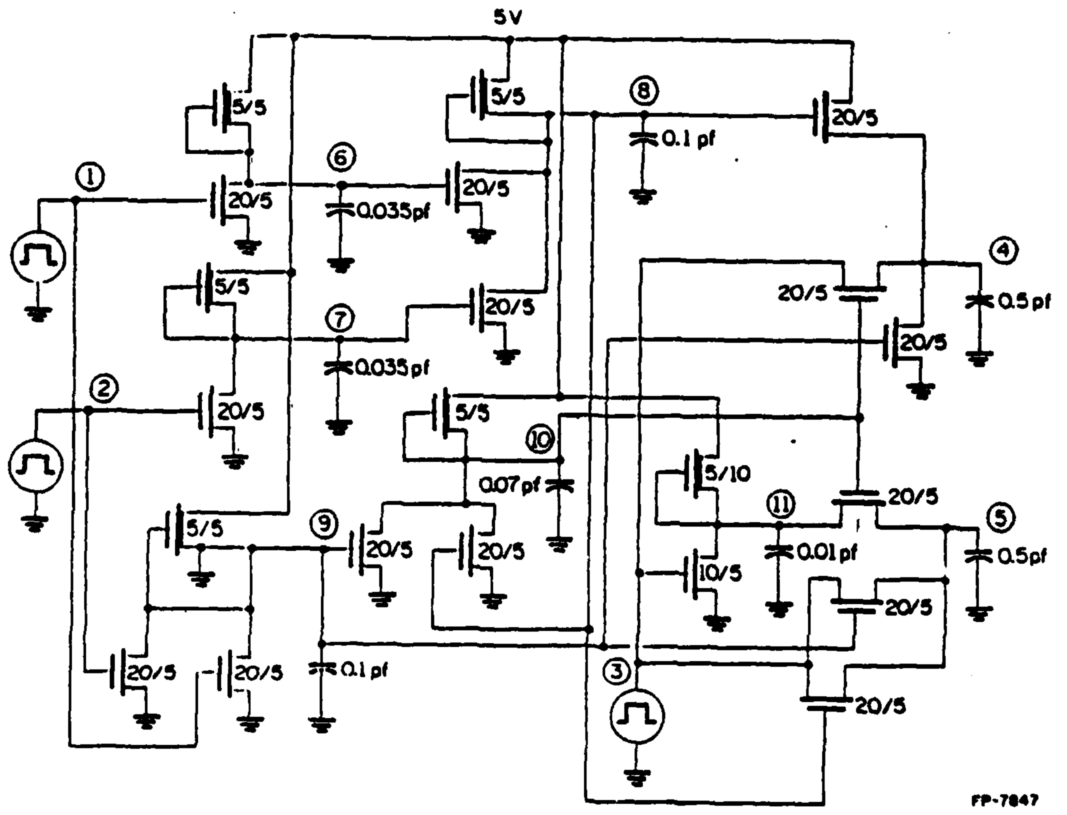


Figure 7.4(a) : A one-bit full-adder with pass transistors

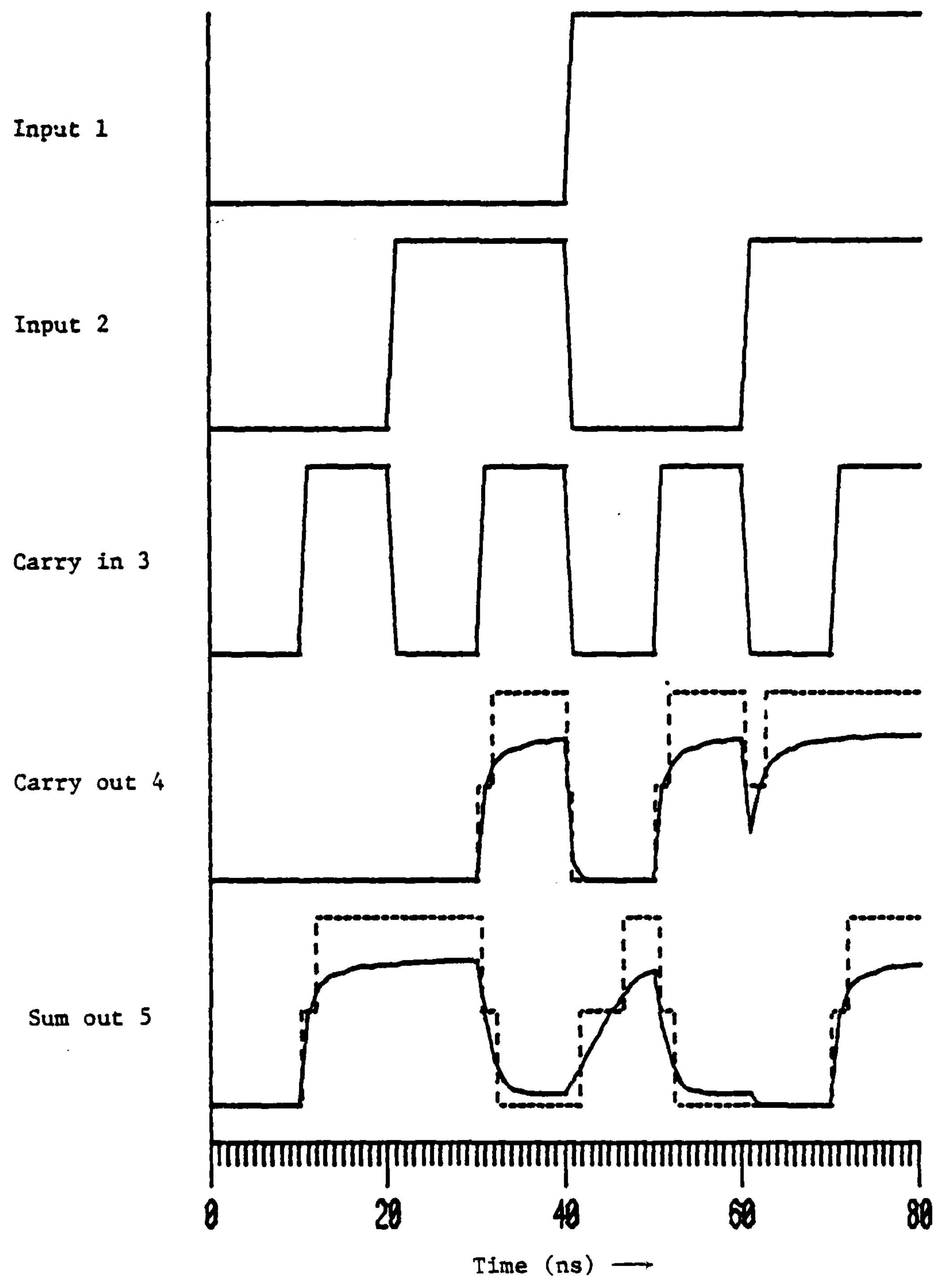


Figure 7.4(b) : Waveforms for a one-bit full-adder with pass transistors

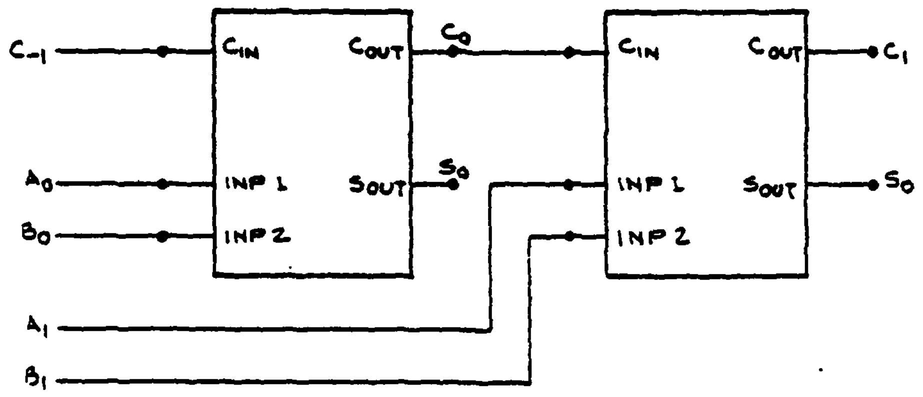


Figure 7.5(a) : A two-bit full-adder with pass transistors

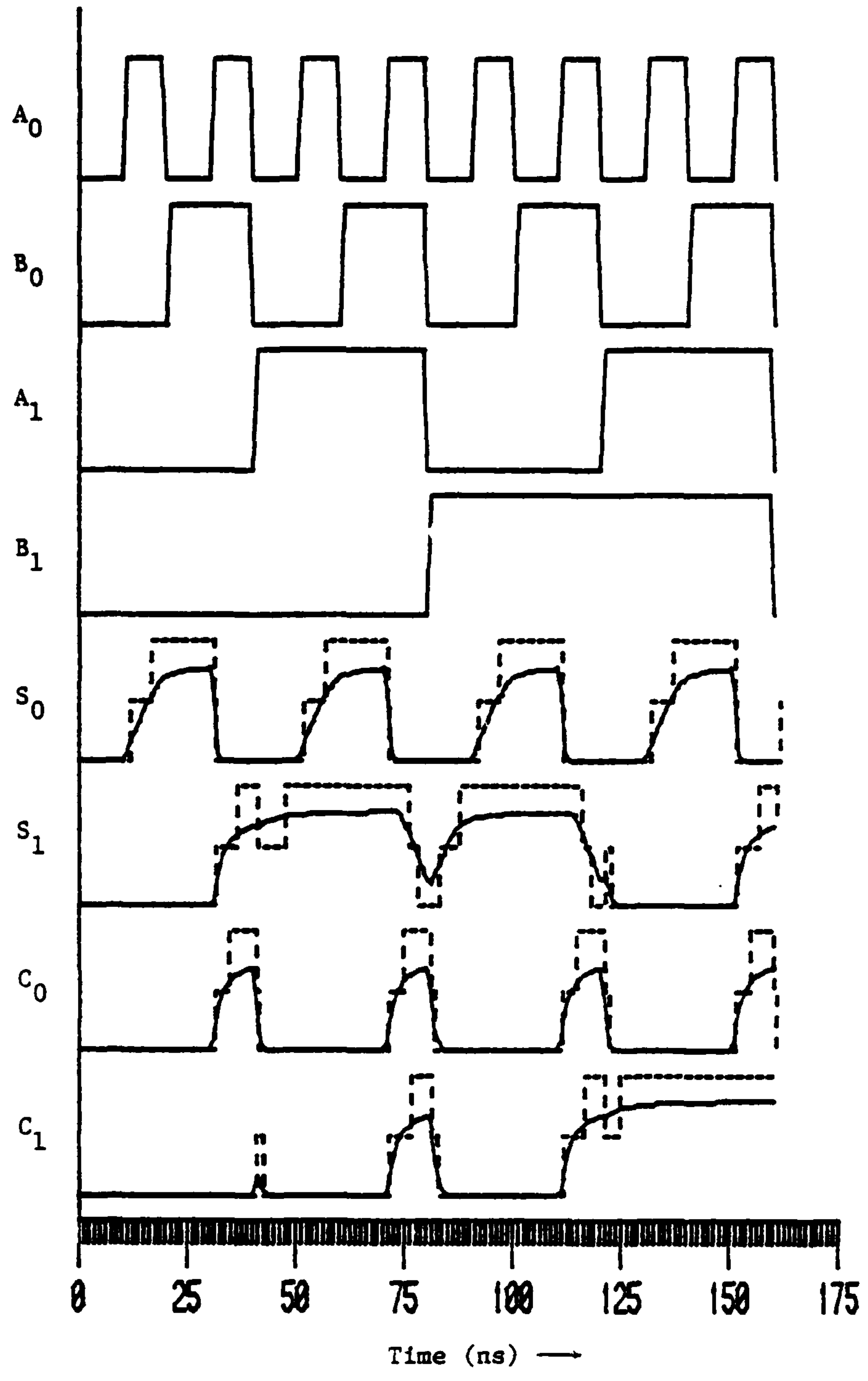


Figure 7.5(b) : Waveforms for a two-bit full-adder with pass transistors

limit. MOSTIM indicates only major glitches in the plots of its waveforms. However, every glitch, major or minor, is flagged and printed out in a separate diagnostic file for each circuit if it is required by the user. An SR-flip-flop circuit is shown in Figure 7.6(a) and its waveforms in Figure 7.6(b). A three-stage ring oscillator is shown in Figure 7.7(a). The final partition of the interval **[0.0ns,40.0ns]** into windows along with the list of blocks to be simulated in each window are given in Table 7.3. Here $\mathbf{MFB}_1$ is the two-input NOR gate, and $\mathbf{MFB}_2$ and $\mathbf{MFB}_3$ are the two inverters, respectively. The waveforms for this circuit are shown in Figure 7.7(b).

A one-bit register is shown in Figure 7.8(a). It is used to realize a three-bit shift register shown in Figure 7.8(b) which can shift both left (down) or right (up). Pass transistors are made use of in several places in the circuit, first, to load the input data onto a bus (node 1), then to transfer data between the bus and registers and also to precharge the bus. The input waveforms applied and the output waveforms produced are shown in Figure 7.8(c). A tally circuit composed of only pass transistors [56] is shown in Figure 7.9(a). In this circuit, all the pass transistors constitute a single PTB. The waveforms for this circuit are shown in Figure 7.9(b). The simulations of the three-bit shift register circuit and the tally circuit test the performance of the mapping technique of the delay operator using Elmore-equivalent capacitances as described in Chapter 5. Finally, we consider a PLA with 149 transistors as shown in Figure 7.10(a). This network is partitioned into 42 MFB's and 12 PTB's. The only nontrivial SCC in the partitioned network consists of 17 MFB's and 4 PTB's. The waveforms for this circuit are shown in Figure 7.10(b).

Among all the networks described above, let us first consider those networks with feedback. Table 7.4 compares the performance of the waveform relaxation method (Algorithm 6.1) and the new event-driven dynamic windowing scheme (Algorithm 6.2) used to simulate the blocks within the SCC's. This table demonstrates that the new windowing technique performs considerably better and is more efficient than the waveform relaxation method. In Table 7.5 we provide a list of all the circuits that have been simulated using MOSTIM thus far, along with the number of transistors (indicated in

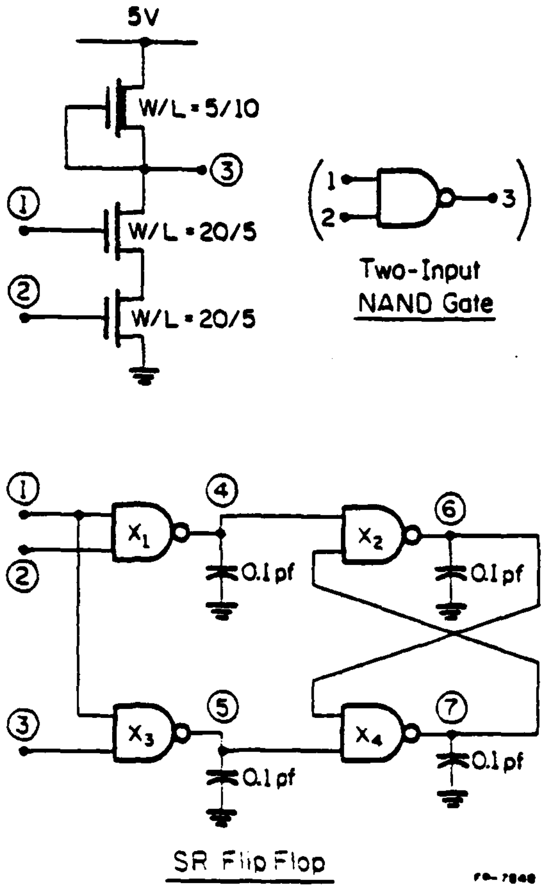


Figure 7.6(a) : An SR-flip-flop

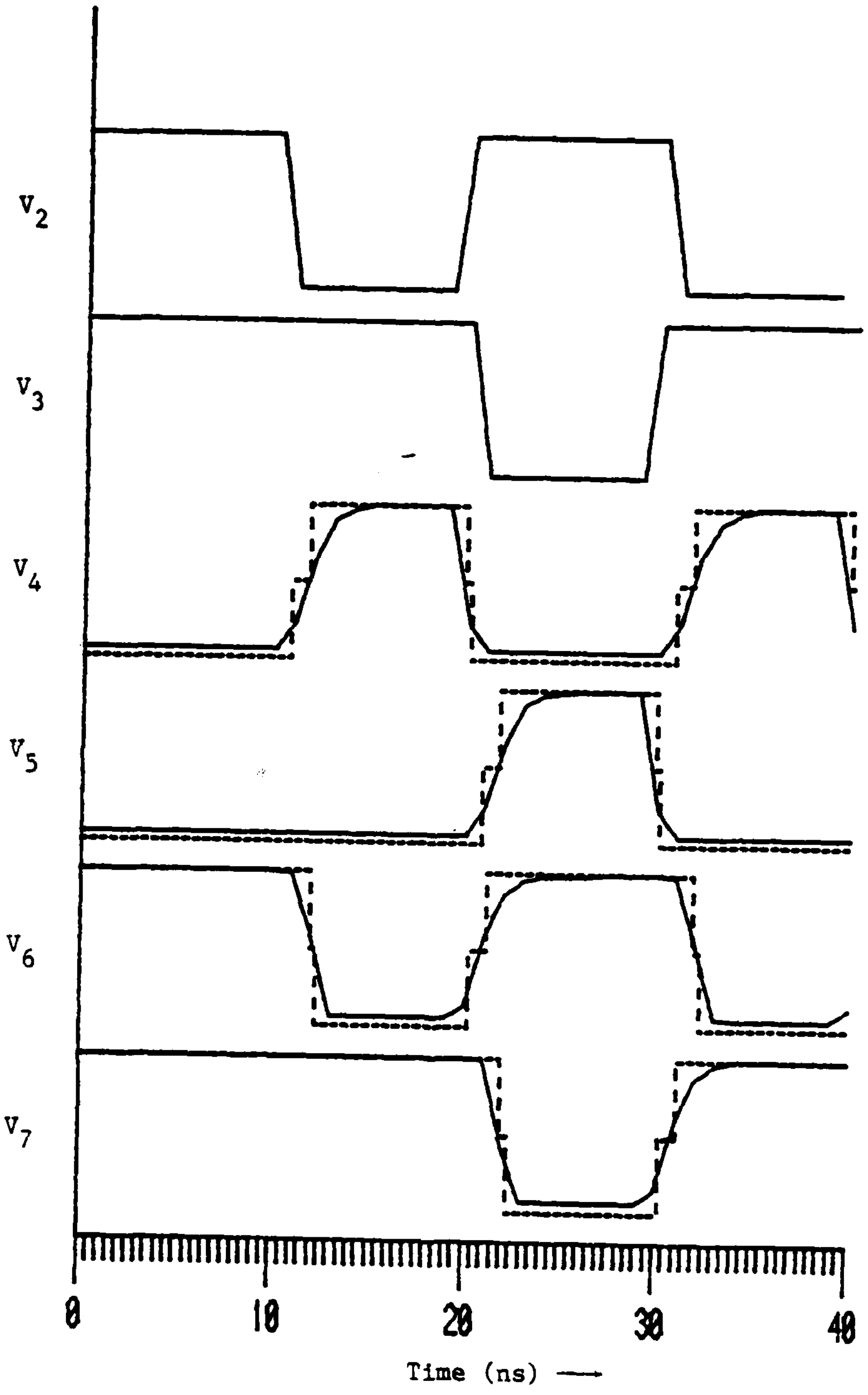


Figure 7.6(b) : Waveforms for an SR-flip-flop

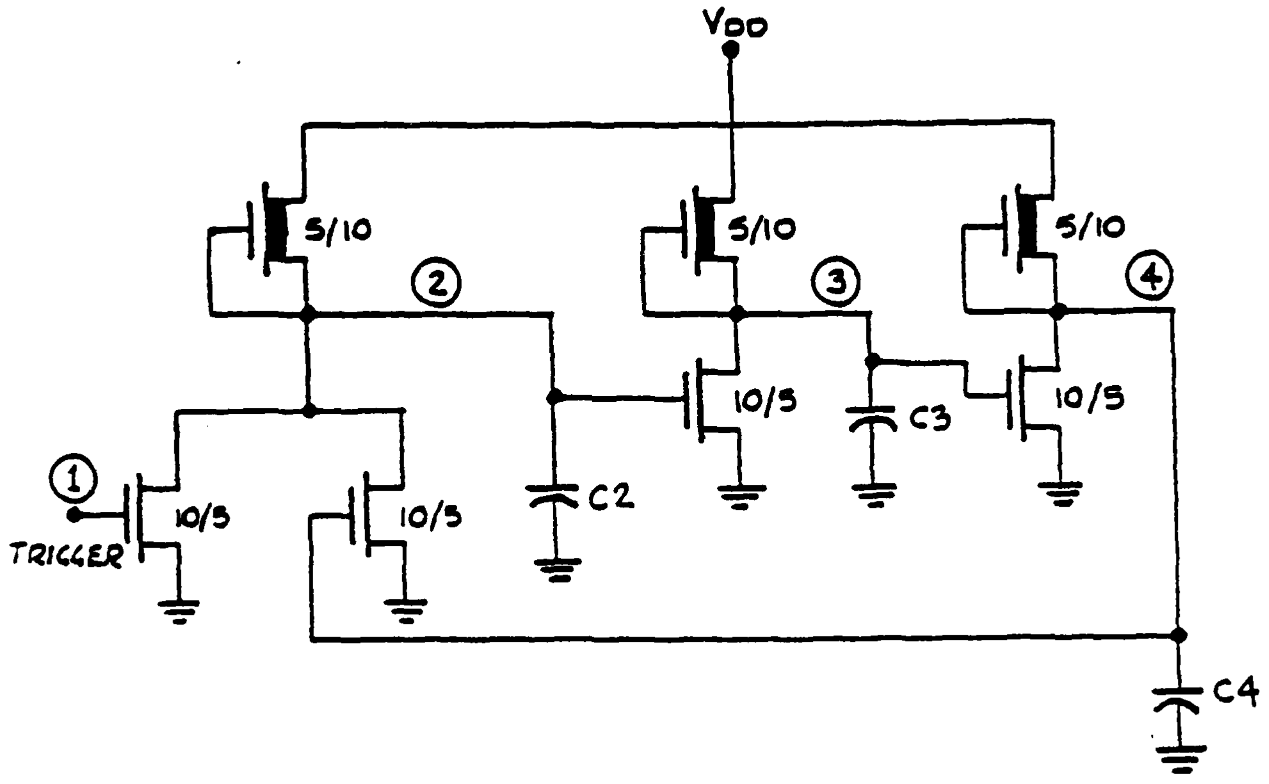


Figure 7.7(a) : A three-stage ring oscillator

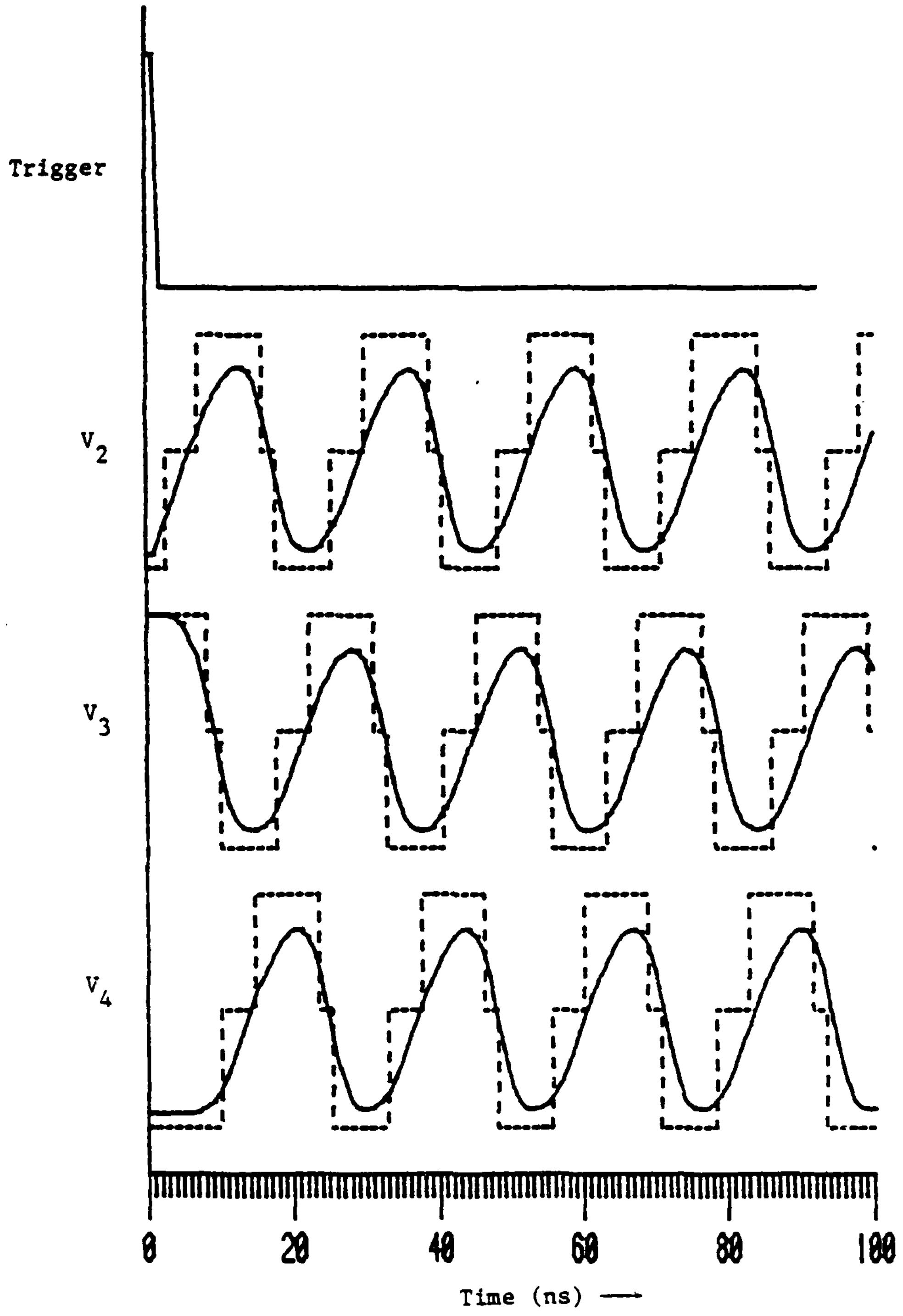


Figure 7.7(b): Waveforms for a three-stage ring oscillator

AD-A161 371 SWITCH-LEVEL TIMING SIMULATION OF MOS VLSI (METAL-OXIDE-SEMICONDUCTOR VER..(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB V B RAO JAN 85 R-1032 3/3

UNCLASSIFIED N00014-84-C-0149 F/G 9/5 NL

END
FILMED
DTIC

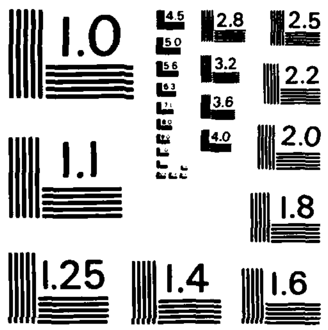

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

Table 7.3 : Final list of windows for a three-stage ring oscillator

WINDOW		BLOCKS
0.00	1.08	MFB_1
1.08	3.46	MFB_2
3.46	4.65	MFB_3
4.65	6.49	MFB_1
6.49	7.68	MFB_2
7.68	9.52	MFB_3
9.52	10.73	MFB_1
10.73	12.57	MFB_2
12.57	13.78	MFB_3
13.78	15.62	MFB_1
15.62	16.83	MFB_2
16.83	18.67	MFB_3
18.67	19.88	MFB_1
19.88	21.72	MFB_2
21.72	22.93	MFB_3
22.93	24.77	MFB_1
24.77	25.98	MFB_2
25.98	27.82	MFB_3
27.82	29.03	MFB_1
29.03	30.87	MFB_2
30.87	32.08	MFB_3
32.08	33.92	MFB_1
33.92	35.13	MFB_2
35.13	36.97	MFB_3
36.97	38.18	MFB_1
38.18	40.02	MFB_2

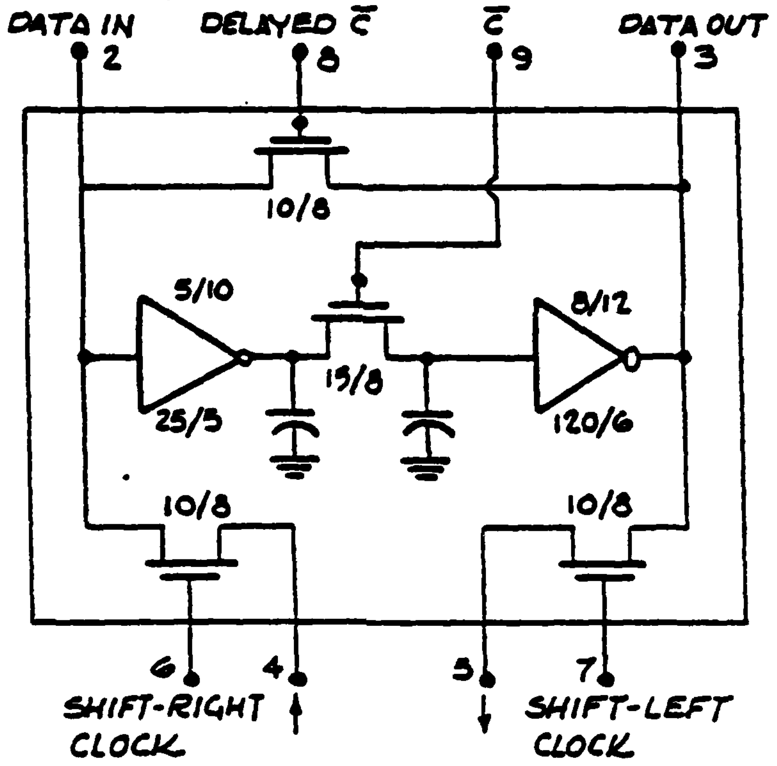


Figure 7.8(a) : A one-bit register

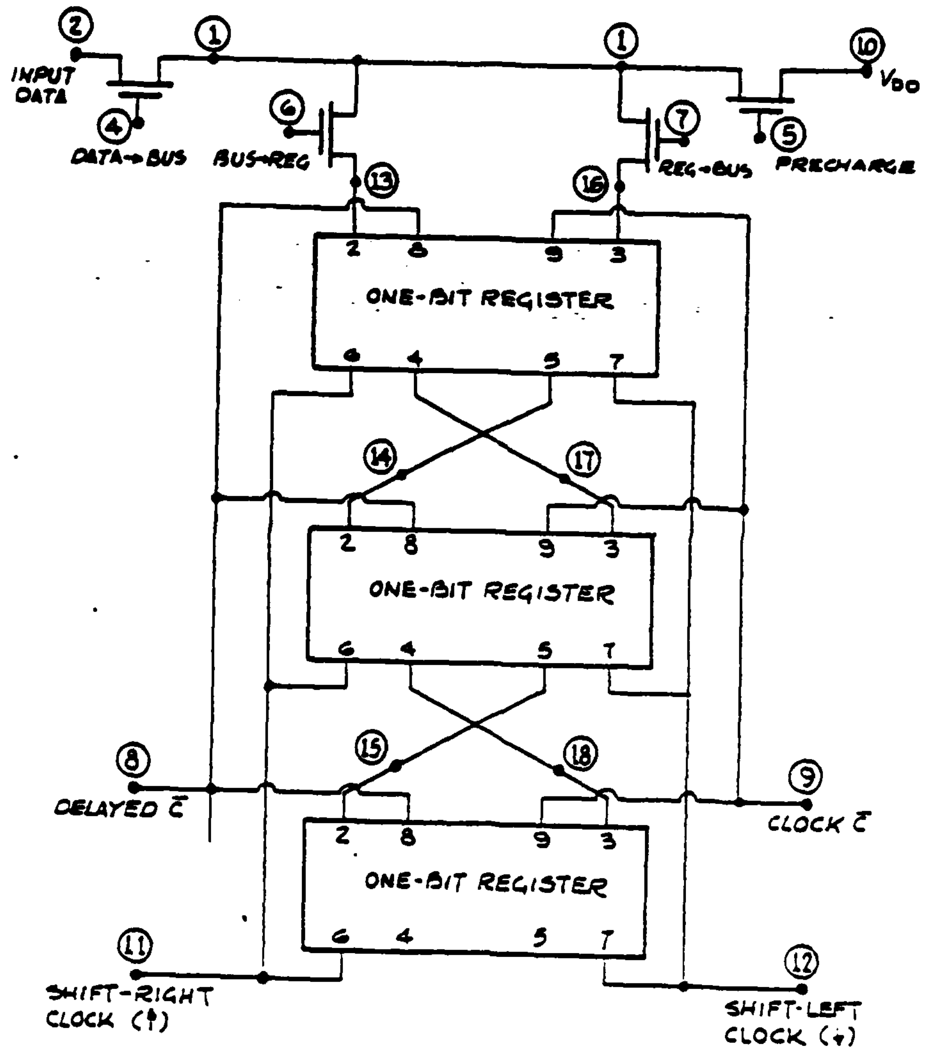


Figure 7.8(b) : A three-bit shift register

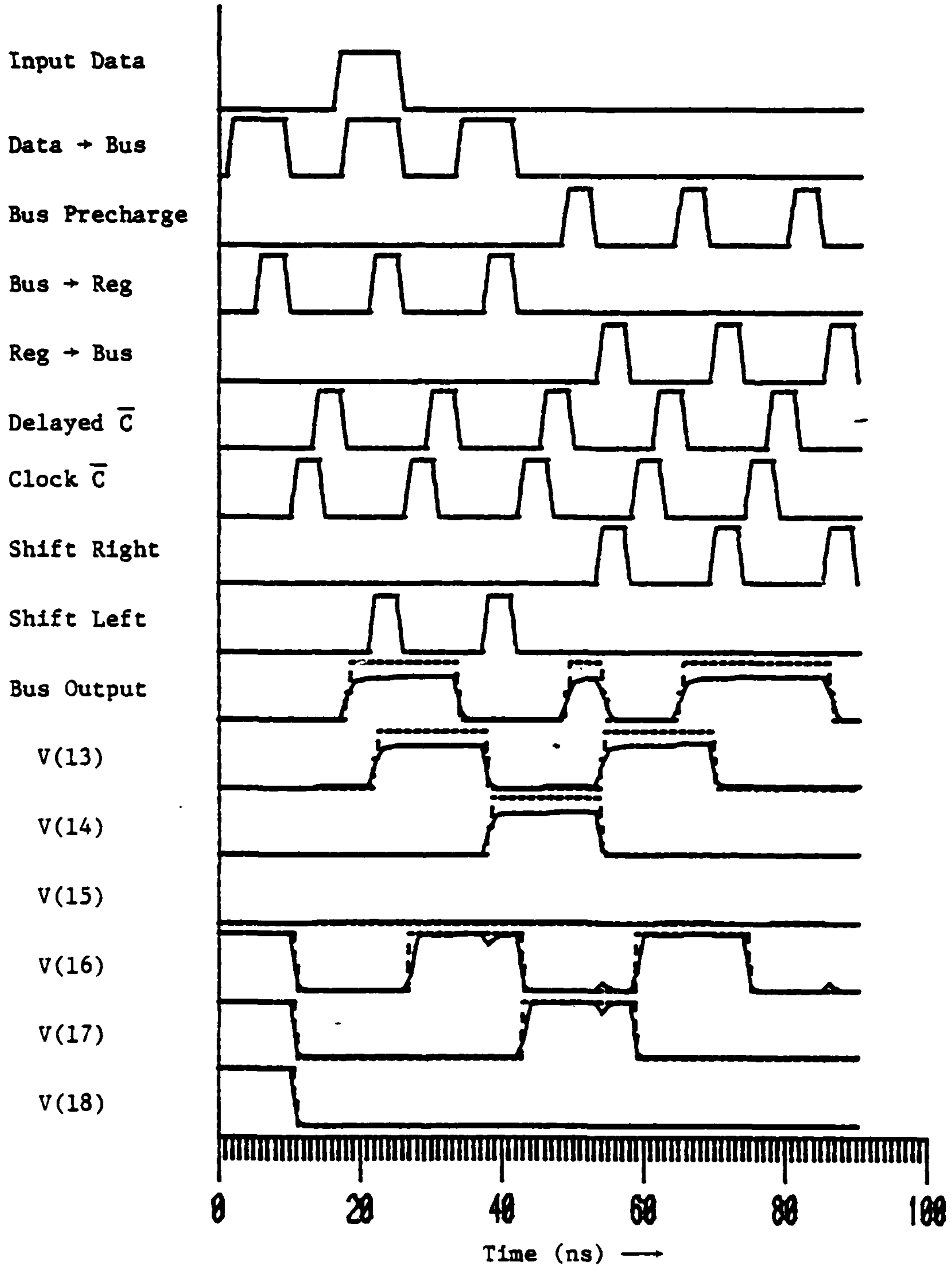


Figure 7.8(c) : Waveforms for a three-bit shift register

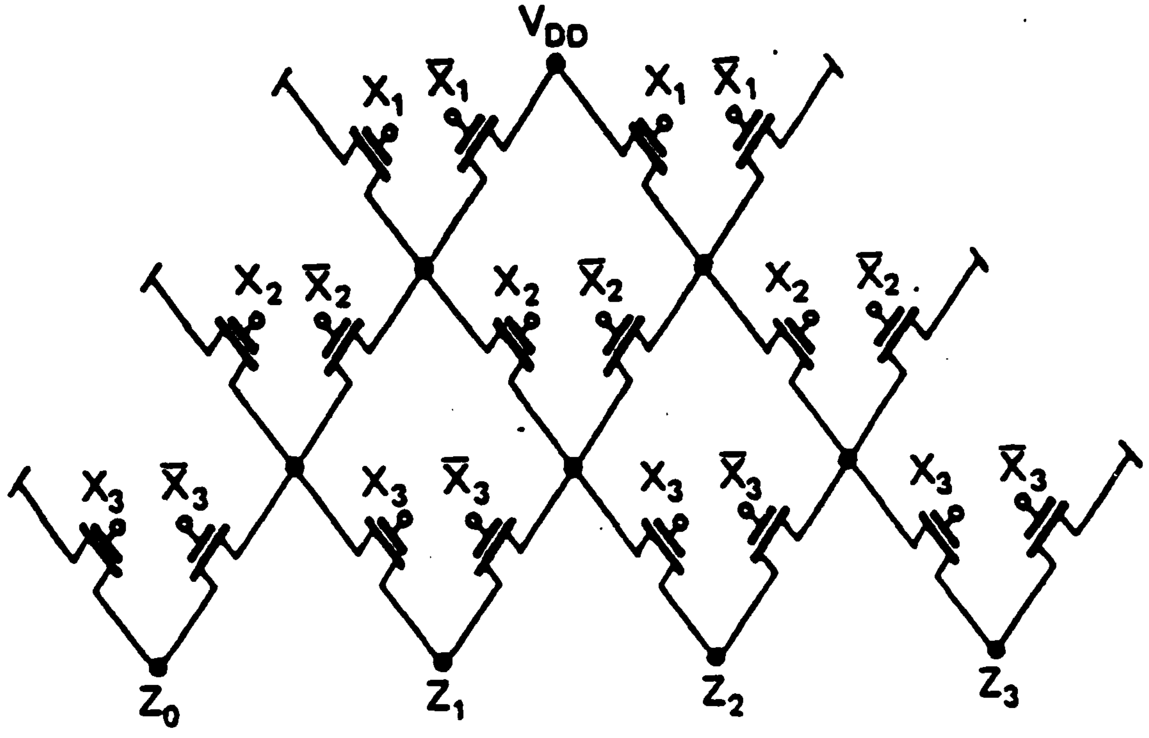


Figure 7.9(a) : A tally circuit composed of only pass transistors

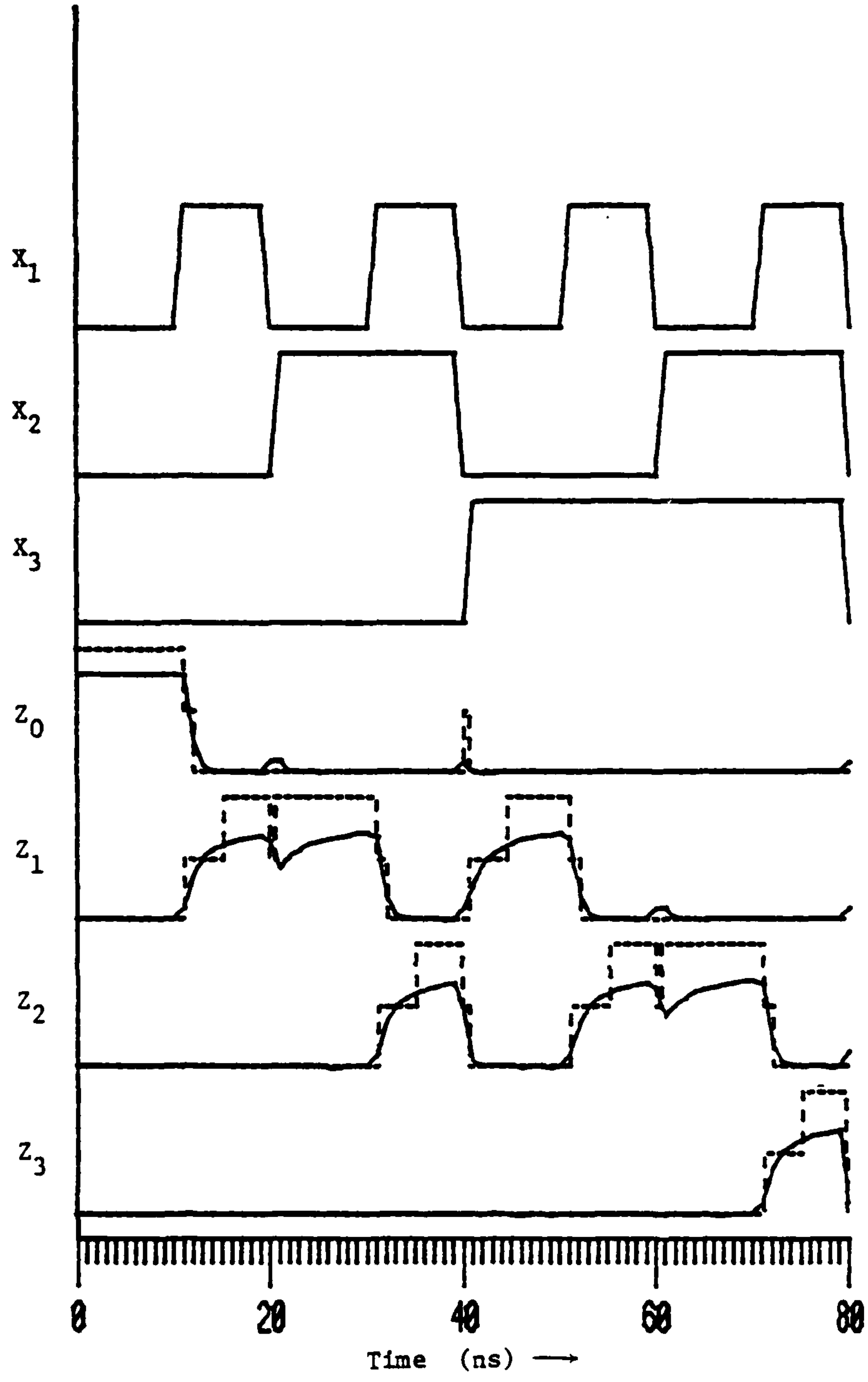


Figure 7.9(b) : Waveforms for a tally circuit

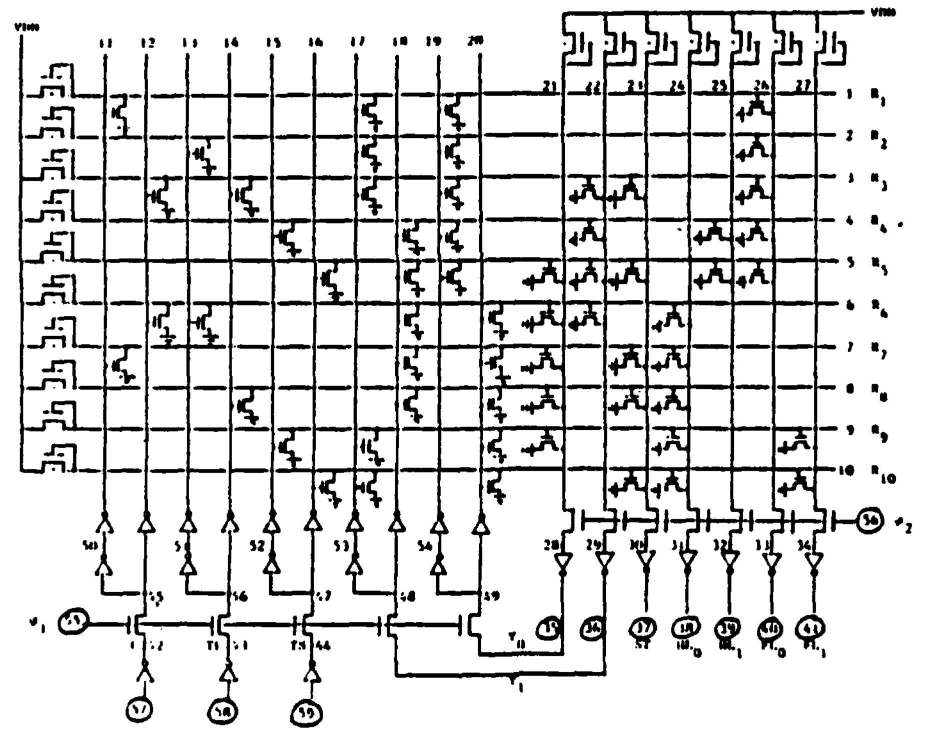


Figure 7.10(a) : A PLA circuit

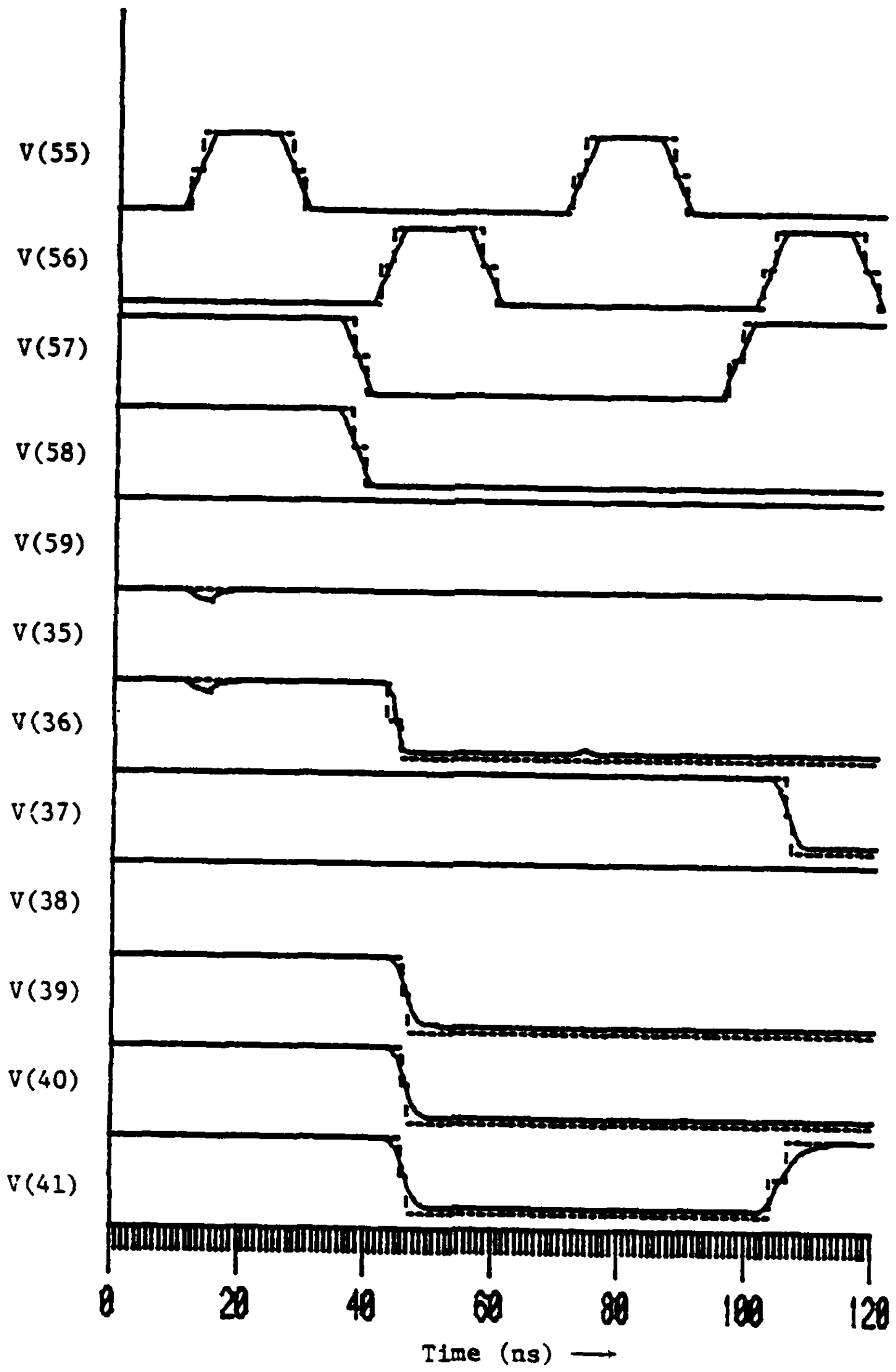


Figure 7.10(b) : Waveforms for a PLA circuit

Table 7.4 : CPU-seconds taken by Algorithms 6.1 and 6.2 to simulate networks with feedback

CIRCUIT		MOSTIM		SPICE2G.1
		Algorithm 6.1	Algorithm 6.2	
3-stage Ring Oscillator	(7)	5.23	1.05	104.60
SR-flip-flop	(12)	1.33	0.86	90.37
3-bit Shift Register	(29)	19.23	6.55	363.30
15-stage Ring Oscillator	(31)	-	1.36	139.85
2-bit Full Adder	(42)	7.82	5.27	794.25
PLA	(149)	13.56	5.85	827.43

parenthesis), and the CPU-time taken by MOSTIM. The CPU-time taken by SPICE2G.1 is also given for comparison.

From each of the waveforms in the circuits described above, one can easily verify that the SLT estimates generated by MOSTIM for pairs of complete transitions are fairly accurate. More precisely, consider the sequence of transitions S at some node in a circuit that is produced by MOSTIM and let $\tilde{S}$ be the ternary equivalent of the analog waveform produced by SPICE2G.1 at the same node. We then consider the extended measure $\tilde{\rho}(\tilde{S},S)$, defined in Equation 4.2 of Chapter 4, to be the measure of the accuracy of the SLT estimates generated by MOSTIM. Figure 7.11 is a scatter plot of the transition times of complete pairs of transitions as computed by MOSTIM against the corresponding threshold crossing times of the analog waveform as computed by SPICE2G.1 for each node in each of the circuits

Table 7.5 : A list of circuits simulated by MOSTIM

CIRCUIT		MOSTIM	SPICE2G.1
3-stage Ring Oscillator	(7)	1.05	104.6
SR-flip-flop	(12)	0.86	90.37
Tally circuit	(18)	3.59	132.37
1-bit Full Adder	(21)	1.28	119.32
3-bit Shift Register	(29)	6.55	363.30
15-stage Ring Oscillator	(31)	1.36	139.85
2-bit Full Adder	(42)	5.27	794.25
50-inverter chain	(100)	3.19	645.28
4-bit Combinatorial Full Adder	(132)	3.45	767.00
PLA	(149)	5.85	827.43

listed in Table 7.5. The maximum percentage error in the timing estimates produced by MOSTIM in all these circuits is 8.75%. For purely combinational logic circuits with no pass transistors, such as the chain of inverters shown in Figure 7.3(a), the error is less than 3%.

In the case of RSIM [26], which is also a switch-level timing simulator for MOS circuits, some of the timing predictions even for purely combinational circuits have been reported to be around 30% of those of SPICE2. For circuits with chains of pass transistors, the predictions are even less accurate. In

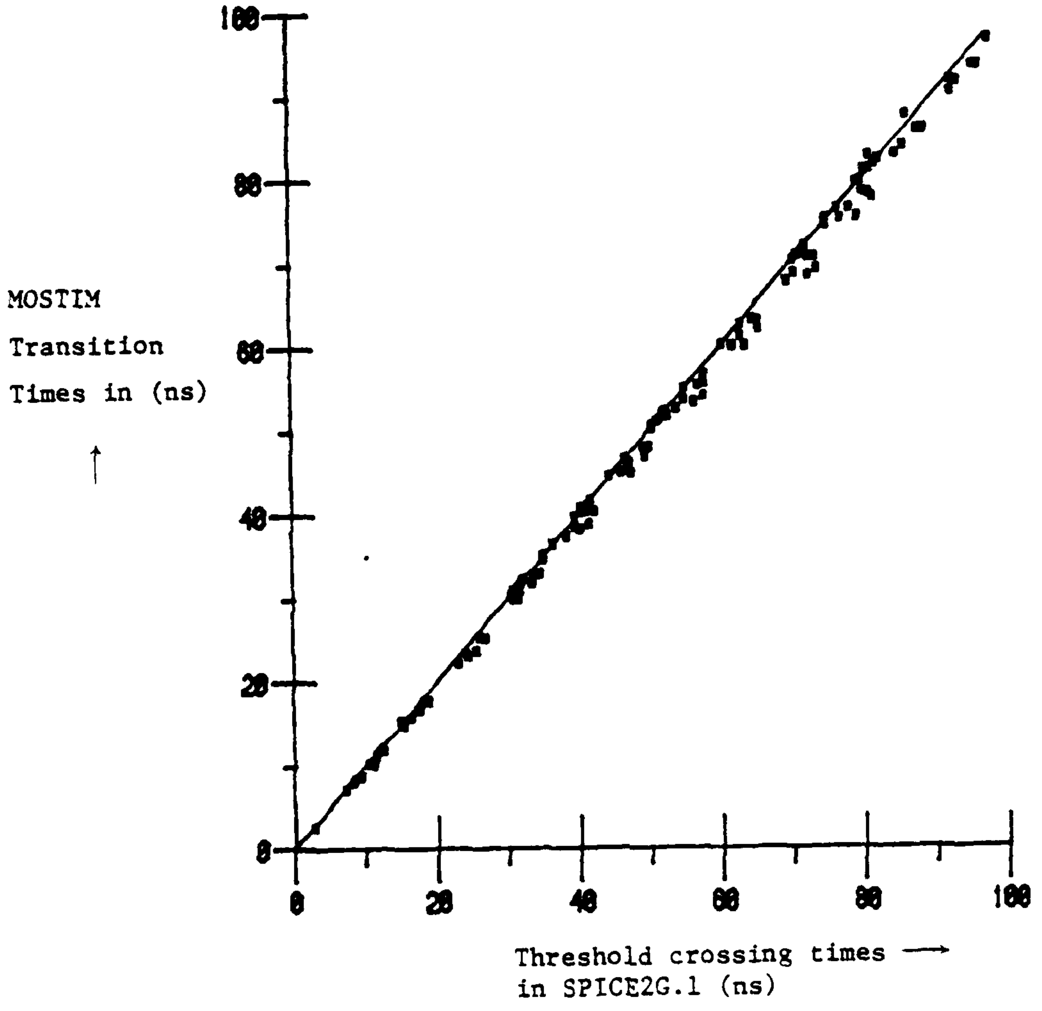


Figure 7.11 : A scatter plot illustrating the timing accuracy of MOSTIM

comparison, the results presented in this chapter indicate that MOSTIM is capable of generating timing estimates within 10% of those of SPICE2G.1 at speeds of around two orders of magnitude higher, which is around the same speed improvement as obtained with RSIM.

CHAPTER 8

CONCLUSIONS

The aim of switch-level timing simulation of VLSI circuits is to provide the circuit designer with digital waveforms at various nodes in the circuits with special emphasis on the accuracy of the times at which the signals change state. In this dissertation we have described a switch-level timing simulator for NMOS circuits which is a fast and accurate simulation tool that gives adequate information on the performance of the circuit with a reasonable expenditure of computation time even for very large circuits. In Chapter 2 of this thesis we reviewed some of the existing simulators for integrated circuits and classified them into two distinct categories, namely, analog simulators and digital simulators. We found that digital simulators in general operate at sufficient speeds to test entire VLSI systems, since the circuit behavior is modeled at a logical rather than a detailed electrical level. However, these simulators do not model the dynamics of the circuits properly and are often useful only in predicting steady-state responses of the signals. Analog simulators, on the other hand, predict both steady-state and transient responses fairly accurately, but are cost-effective only for circuits with less than a few thousand components, which are considered small in the present day VLSI technology.

The algorithms presented in this thesis have lead to the development of a switch-level timing simulator for NMOS VLSI circuits called MOSTIM, an attempt to bridge the gap between analog and digital simulators. MOSTIM performs simulations at a switch level and, hence, runs at speeds close to those of digital simulators. Furthermore, it uses a delay operator to delay signal transitions accurately and, hence, provides the timing accuracy comparable to those of analog simulators.

In Chapter 3, we discussed the algorithms for partitioning the input network into various blocks and the ordering of these blocks for processing. The key to the partitioning strategy is to divide the set of enhancement transistors into driver transistors and pass transistors. We presented a graph-theoretic algorithm that achieves this in computation time which is linear with the number of enhancement devices. The driver transistors were then grouped together to form multifunctional blocks (MFB) and the pass transistors were grouped together to form pass transistor blocks (PTB). We created a third type of block called input source (SRC) to model voltage sources, clocks, etc. We then constructed a directed graph **G** with vertices corresponding to the various circuit blocks, namely, MFB's, PTB's, and SRC's, and directed arcs describing the interconnections between them. A modified version of a depth first search known as Tarjan's algorithm [31] is used to detect strongly connected components (SCC) in **G**. The vertices within an SCC correspond to blocks forming feedback loops in the original circuit and are collapsed into single vertices, thus creating an acyclic reduced graph $\tilde{\mathbf{G}}$. The vertices of $\tilde{\mathbf{G}}$ are then placed in topological order for processing.

The algorithms for the switch-level simulation of multifunctional blocks and pass transistor blocks are presented in Chapter 4. An MFB is a single output, multiple input, unidirectional block, whose steady-state output is a Boolean function of its inputs. A graphical technique using internal node eliminations is used to evaluate the state of the signal at the output, given the input signal states. No attempt is made to evaluate signals at the internal nodes of the MFB. In the switch-level simulation of a PTB, however, the signal at every node within the PTB is evaluated. The transistors in a PTB are modeled as bidirectional switches whose conduction states (i.e., **open**, **closed**, or **intermediate**) are controlled by the signal at the corresponding gate terminals. A strong node forces its state on a weaker node connected to it via a path of conducting transistors at any given time instant. The algorithm is quite similar to the one used in conventional switch-level simulators such as MOSSIM [19], except for the interpretation of the **u** state (or **X** state as used in MOSSIM). This algorithm also handles situations of conflict between two strong signals, charge sharing, etc.

The switch-level simulation algorithms described in Chapter 4 generate zero-delay ternary waveforms for each pullup node in an MFB and each normal node in a PTB. A delay operator described in Chapter 5 is used to delay pairs of complete transitions (i.e., **0→u** followed by **u→1**, or **1→u** followed by **u→0**) in the zero-delay waveforms. The delay operator computes appropriate delay values by taking several parameters into account, such as block configuration, loading, device geometries, and input slew rates. For NMOS technology, knowing the delay characteristics of five different circuit primitives is sufficient, within reasonable limits of accuracy, to compute delays through any general MFB or PTB. These five primitives are simulated using an accurate circuit simulator such as SPICE2 [1], or SLATE [3], for various device and circuit parameters, and the delay values are extracted and stored in a delay table. This is done in a presimulation phase. During simulation, MOS-TIM then maps an MFB or a PTB into one of the five primitives and obtains the appropriate delay value through fast table lookup methods, and interpolation when necessary.

In Chapter 6 we discussed techniques used to process blocks within an SCC. In order to perform a switch-level simulation of a block (MFB or PTB), the waveforms at the input nodes to the blocks must necessarily be known. Since this is not possible for blocks within an SCC, these have to be handled separately. A waveform relaxation technique could be used, wherein the blocks are processed iteratively in a predetermined order with unknown input waveforms initially relaxed and output waveforms constantly updated. Several drawbacks of this technique were discussed. A new dynamic windowing method that overcomes most of these drawbacks was presented. In principle, this new scheme is quite similar to the classical event-driven time-wheel approach used in conventional logic simulators [13,19], except that events take place during intervals of time instead of occurring instantaneously. The entire time interval of analysis is automatically partitioned into variable size windows such that the signal at each node in each block within the SCC occupies a steady state (i.e., **0** or **1**) at the window boundaries. Associated with each window is a set of blocks scheduled for processing during that window. This new scheme does not require an *a priori* ordering of blocks within the SCC, and is also seen to take less computation time and less storage than the waveform relaxation method.

A number of NMOS circuits have been simulated using MOSTIM. The performance is discussed in Chapter 7. In all the circuits simulated thus far, MOSTIM provides timing information with an accuracy of within 10% of the timing provided by SPICE2 [1], at approximately two orders of magnitude faster in simulation speed. MOSTIM also provides much better timing estimates than RSIM [26] at approximately the same speed of simulation.

We now consider several extensions that could be used to improve the performance of MOSTIM. At present, MOSTIM is capable of only handling NMOS circuits. A few modifications are needed to include CMOS technologies as well. In the partitioning scheme, the graph used to represent the network would now consist of two types of edges, namely, n-type and p-type edges, corresponding to n-channel and p-channel transistors, respectively. In conventional CMOS circuits, pass transistors are usually implemented using n-channel and p-channel devices having common drain and source nodes. The edges corresponding to these transistors can be easily detected and removed from the graph. Once this is done, a pullup node can be identified as a node adjacent to both n-type and p-type edges in the resulting graph. One can then use the scheme described in Chapter 3 of this thesis to complete the partitioning. An MFB in CMOS would consist of a network of n-channel devices between the pullup node and ground and a dual network of p-channel devices between the pullup node and V_{DD}. A PTB would also consist of both n-channel and p-channel pass transistors. The algorithms to perform the zero-delay switch-level simulation remain primarily the same, except that a p-channel device is modeled as a switch that is **closed** when its gate signal is at **0** and **open** when its gate is at **1**. The delay primitives have to be redefined by using CMOS inverters and pass transistors and the delay functions have to be recomputed for these new primitives. The mapping techniques used by the delay operator must now also account for the resistances of the p-channel devices in addition to those of the n-channel devices. With the above modifications MOSTIM can be extended to handle CMOS circuits as well.

The use of ratioed logic, as suggested in [20,21], would result in a better scheme to handle conflicts in a PTB. The delay operator has to be extended to provide better timing in these situations. Providing better timing estimates in case of charge sharing also needs to be investigated. Most conventional network extractors create an RC-network to model the interconnect regions in an integrated circuit. The resistance of the metal lines can be neglected, but the resistances of the polysilicon and the diffusion lines have a considerable effect on the propagational delays in the circuit. Using reduced-order modelling techniques, such as the Elmore time-constant approach, to generate equivalent lumped capacitances at each node in the circuit is another topic that needs to be investigated. Further research is also needed to use a MOSTIM-like approach for other technologies, such as bipolar, ECL, and I^2L.

Thus far, we have only considered the deterministic simulation of integrated circuits. It is well-known, however, that random fluctuations inherent in the IC manufacturing process affect the performance of VLSI circuits significantly. This is further aggravated by the scaling down of device sizes and the interconnect regions. The circuit designer is, therefore, often interested in obtaining some statistical information about the timing in the circuit. A Monte-Carlo simulation of the entire VLSI circuit can prove to be prohibitive in terms of CPU-time. As an alternative, one could compute the statistical behavior of the delays through standard primitives using the conventional Monte-Carlo methods and store the necessary information in tables. One could then map a general block in the network onto one of the standard primitives and obtain the statistical timing information through a look-up table. This approach is very similar to the operation of the delay operator in MOSTIM, and it needs further investigation.

APPENDIX I

PLOTS OF DELAY FUNCTIONS

In this appendix, we will show plots of the inertial delay, Δt_1, and the rise/fall delay, Δt_2, in both types, "0" and "1", as functions of the input slew-rate Δ_{in} for standard primitives 1, 2, and 3, and as functions of β and γ, in addition, for standard primitives 4 and 5 for the following technology.

VTO_E=+1.0 V

VTO_D=−3.0 V

V_{DD}=+5.0 V

KP=100 $\mu A/V^2$

Standard devices :

Load : W=5 μ , L=10 μ

Driver : W=10 μ , L=5 μ

Pass : W=10 μ , L=10 μ

Standard capacitances :

C_{1S}=0.01 pF

C_{2S}=0.01 pF

C_{3S}=0.01 pF

C_{4S}=0.10 pF

C_{5S}=0.10 pF

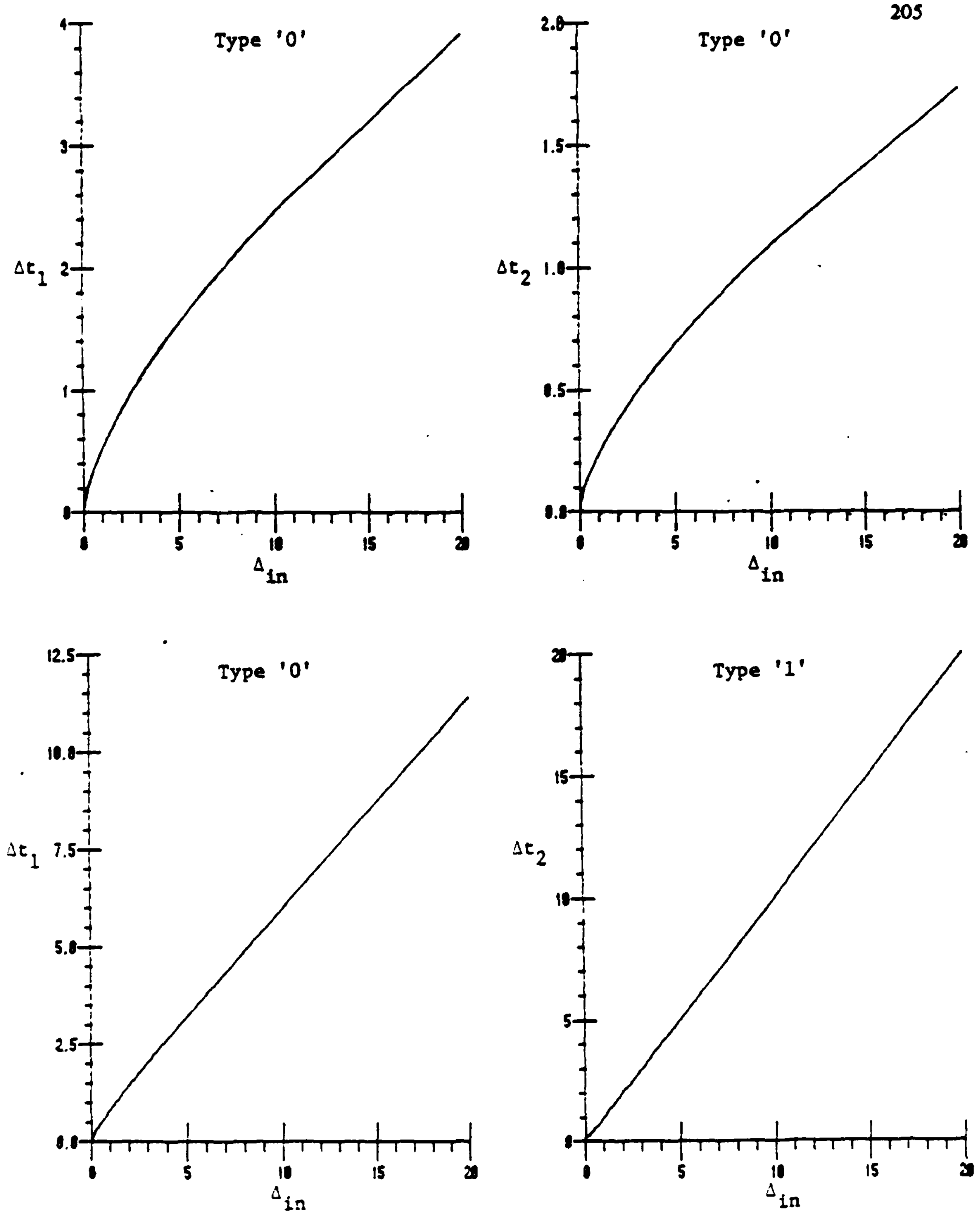


Figure A1.2 : Delay functions for standard primitive 2

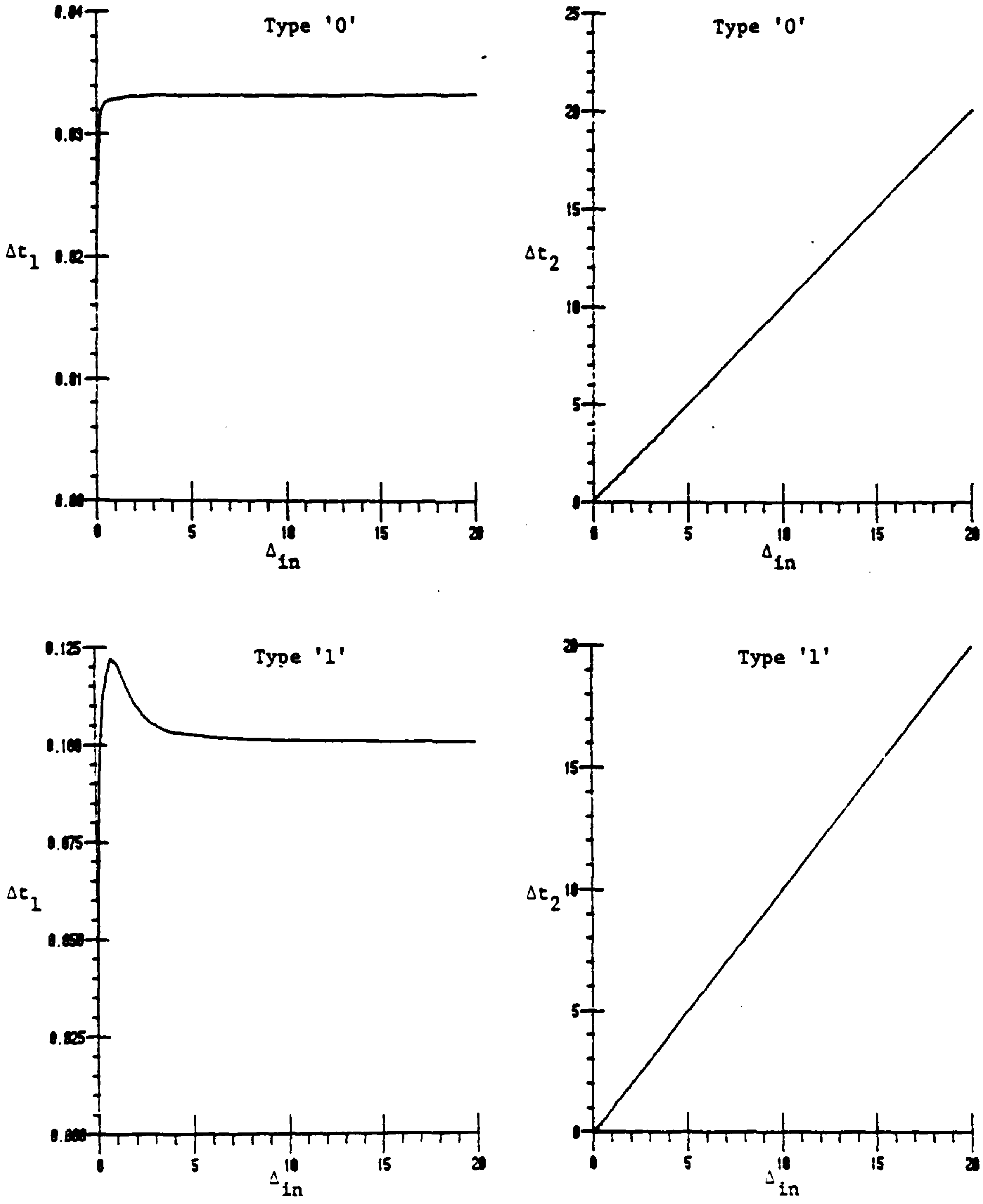


Figure A1.3 : Delay functions for standard primitive 3

Dimensionless parameters :

$\beta \in \{0.1, 1.0, 5.0\}$

$\gamma \in \{0.1, 1.0, 10.0\}$

$\delta = 4.0$

The plots are shown in Figures A1.1 to A1.11. The delay values in all these plots are in nano seconds.

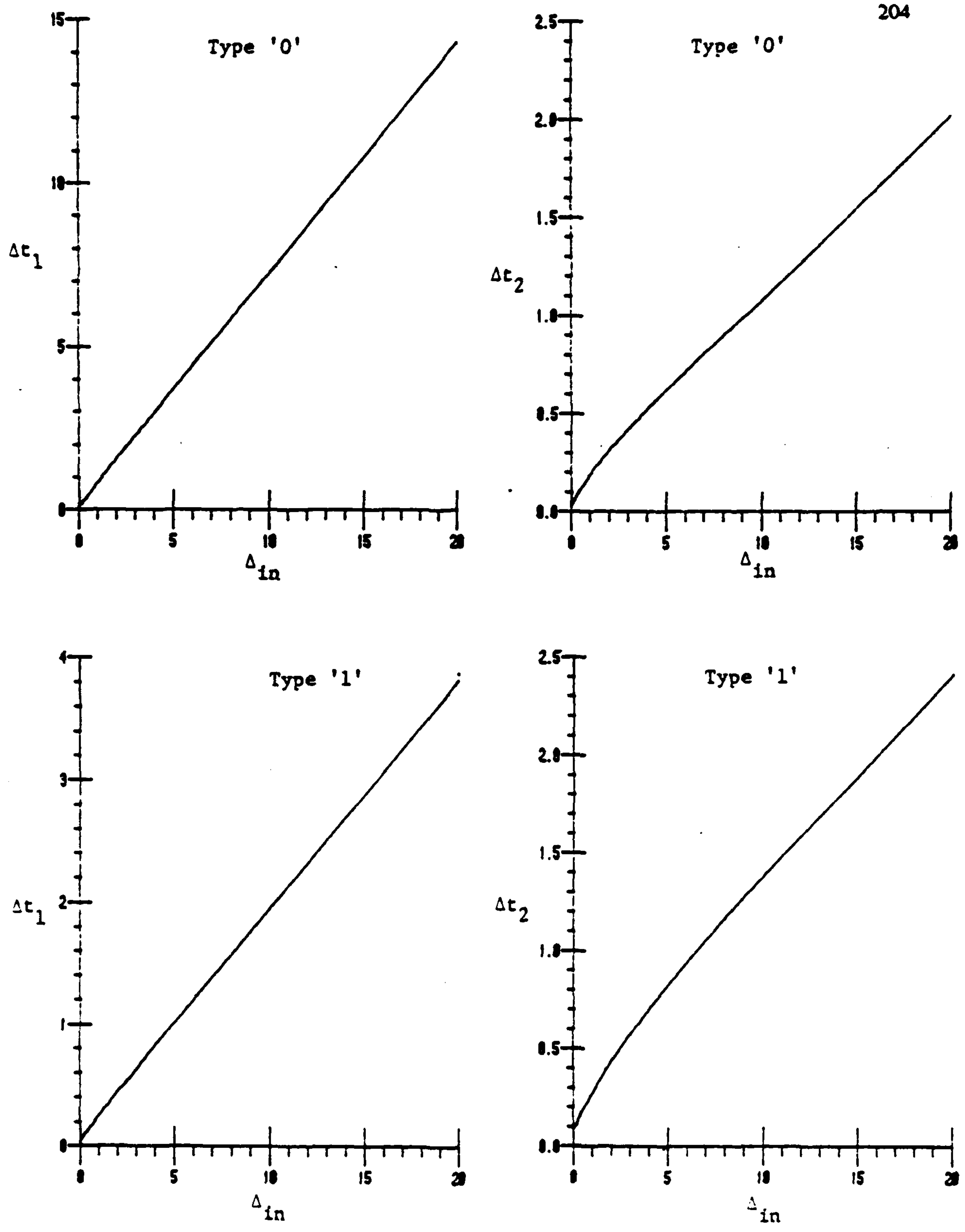


Figure A1.1 : Delay functions for standard primitive 1

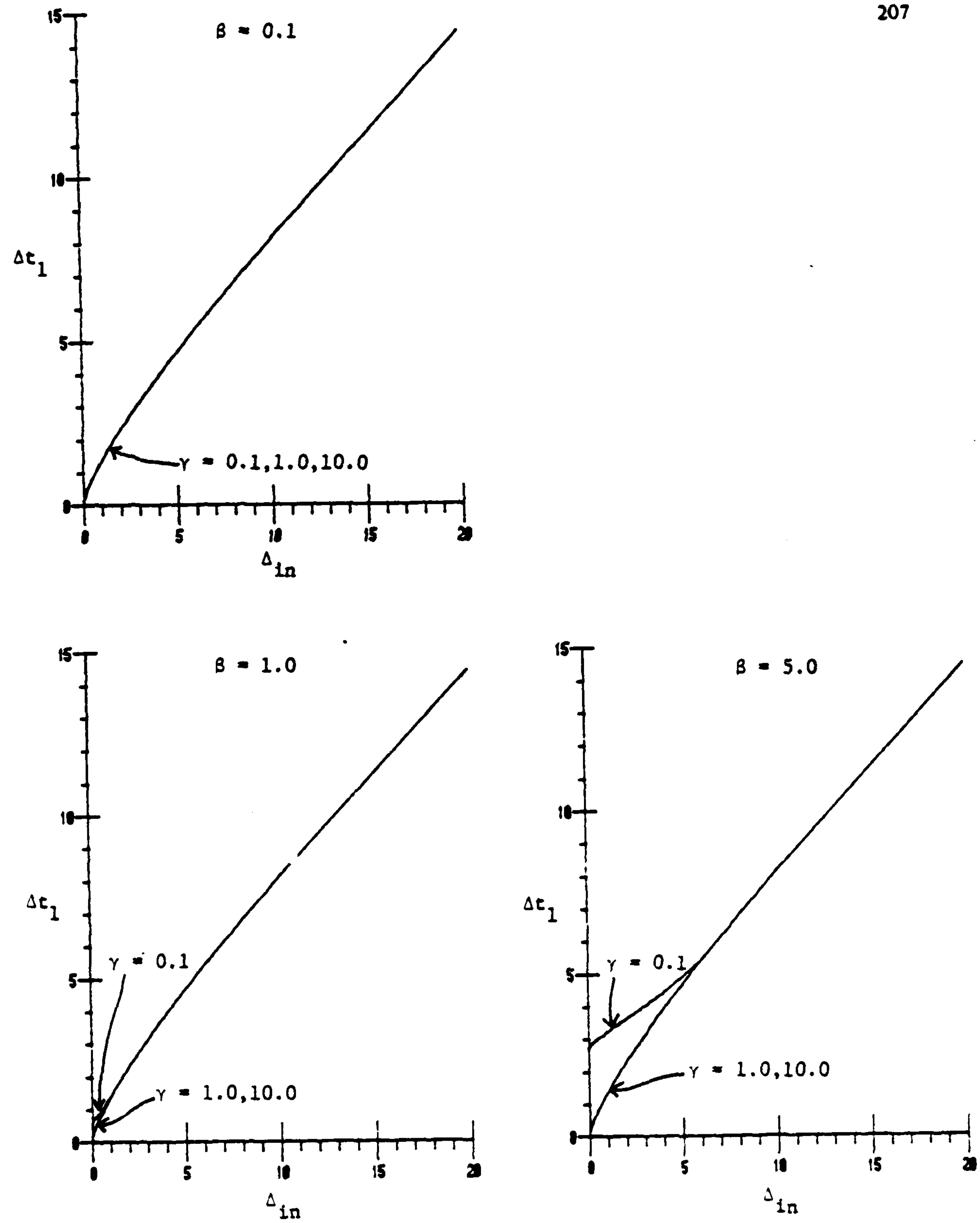
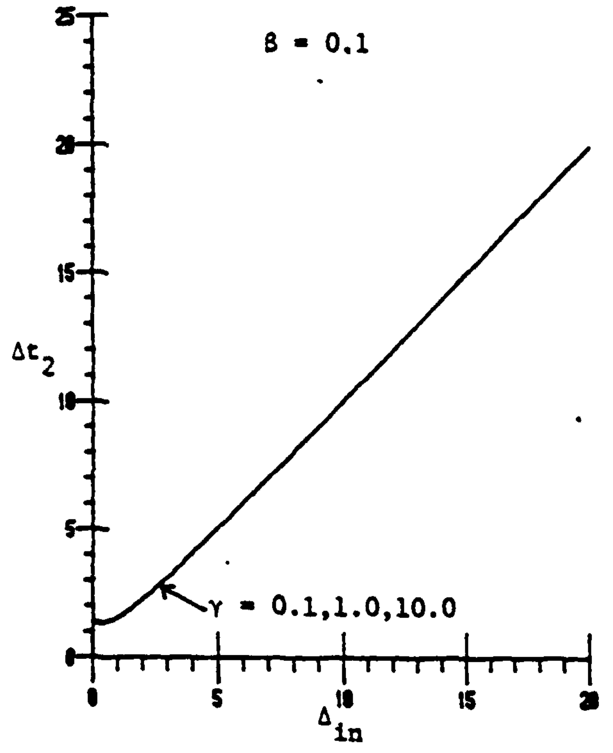
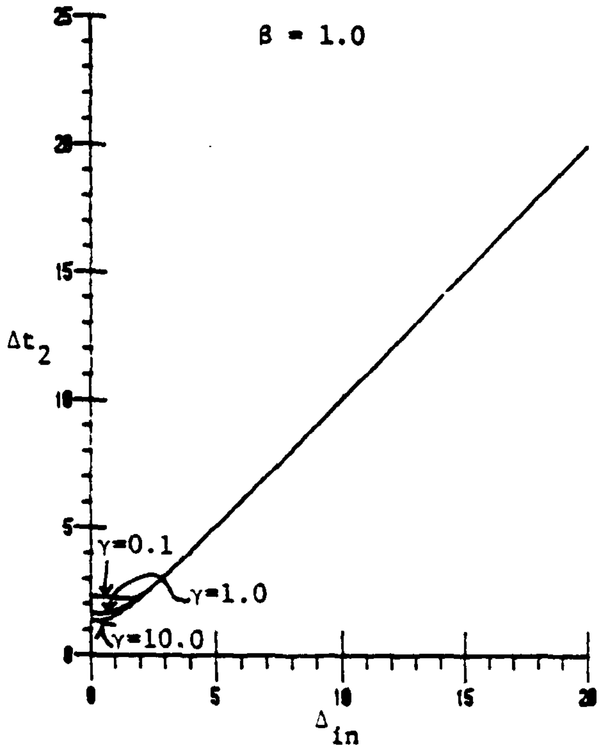


Figure A1.4 : Inertial delay for standard primitive 4, type "0"





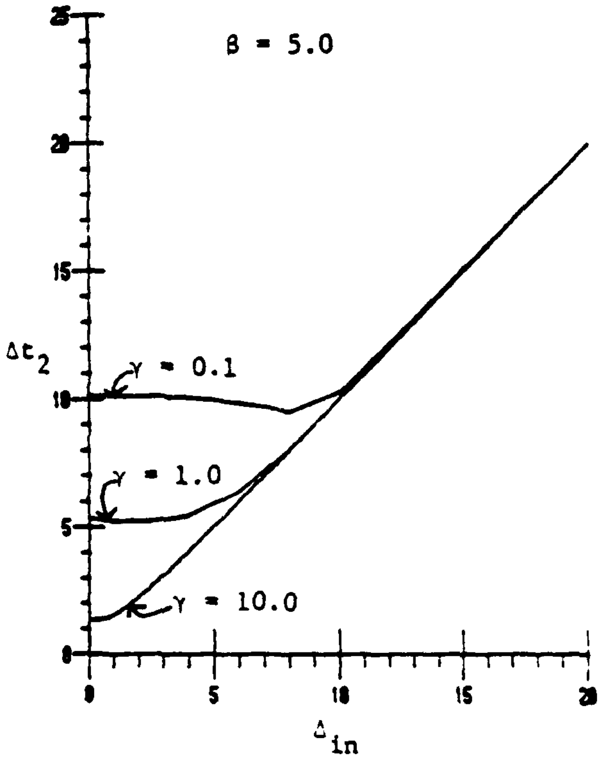
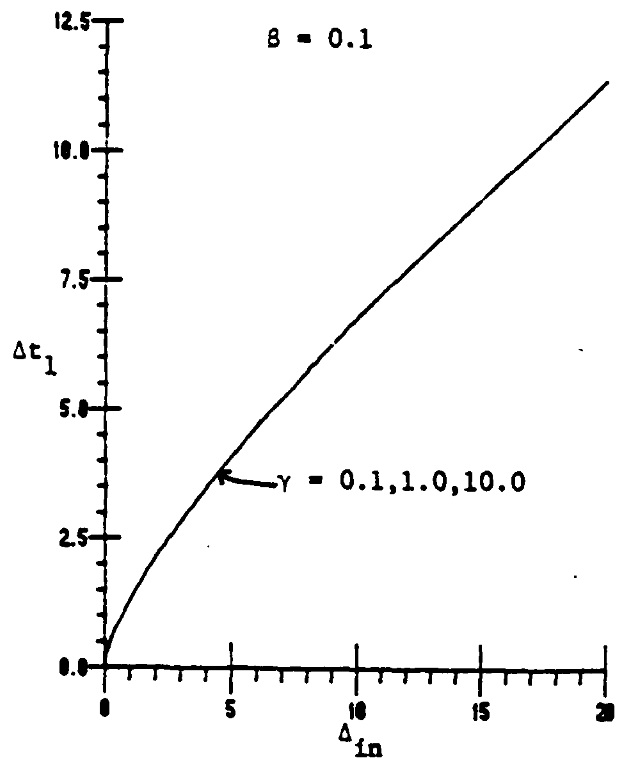
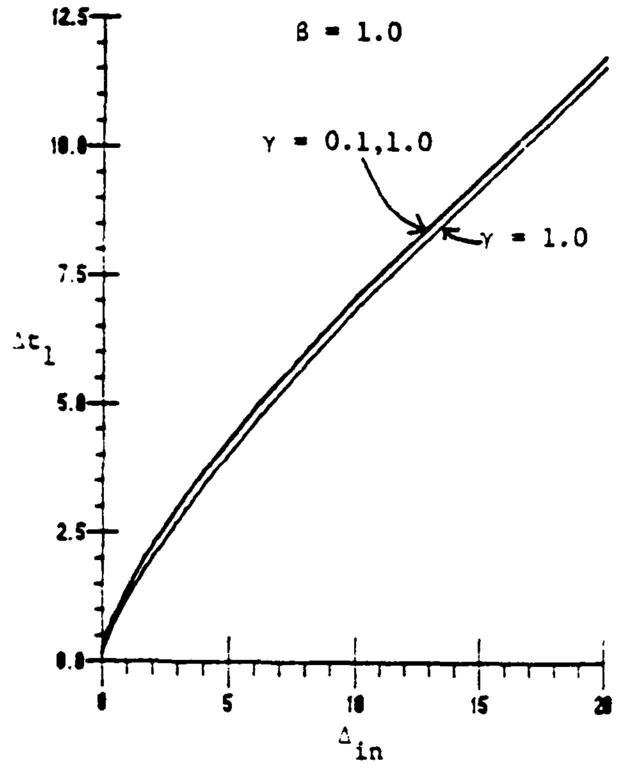


Figure A1.5 : Rise delay for standard primitive 4, type "0"





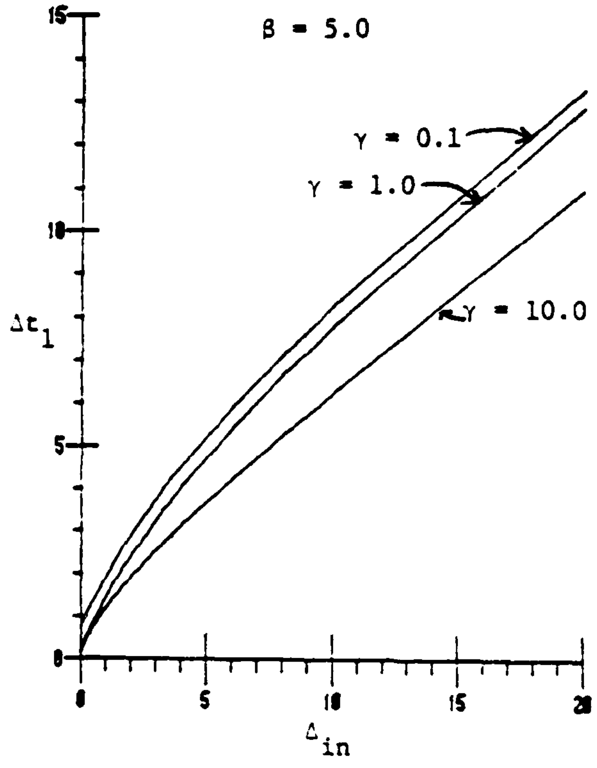
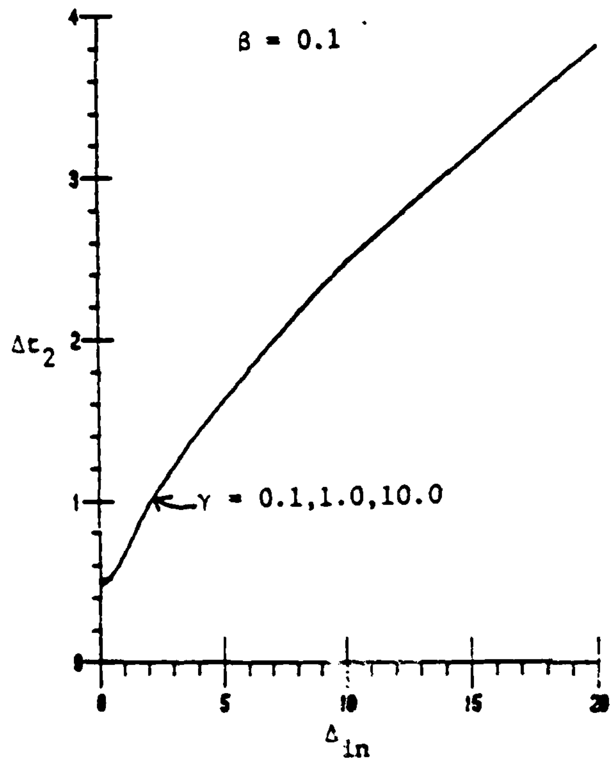
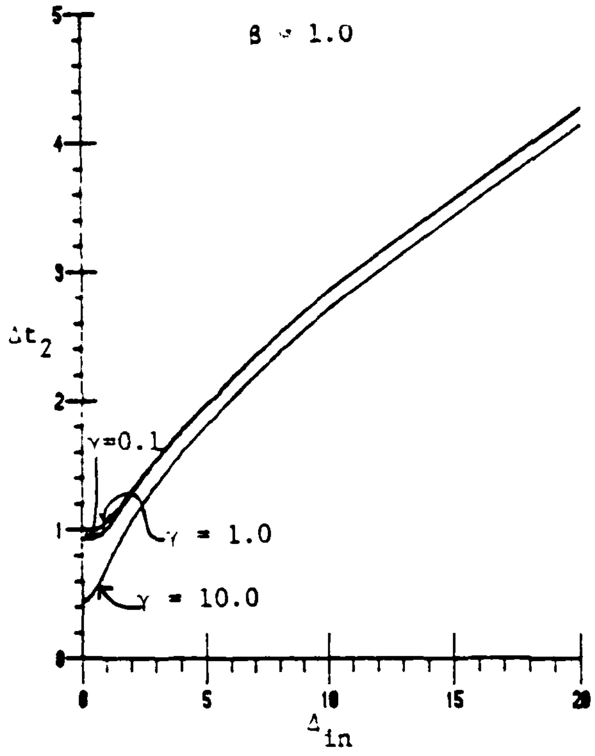


Figure A1.6 : Inertial delay for standard primitive 4, type "1"





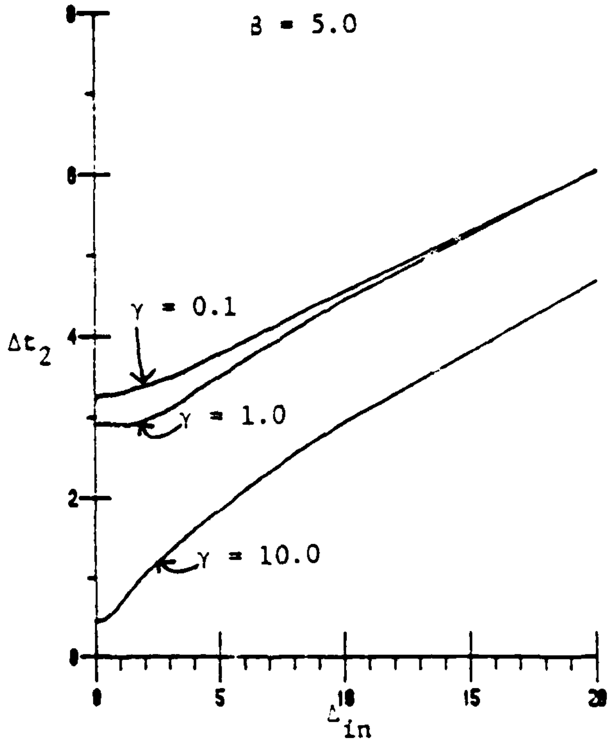


Figure A1.7 : Fall delay for standard primitive 4, type "1"

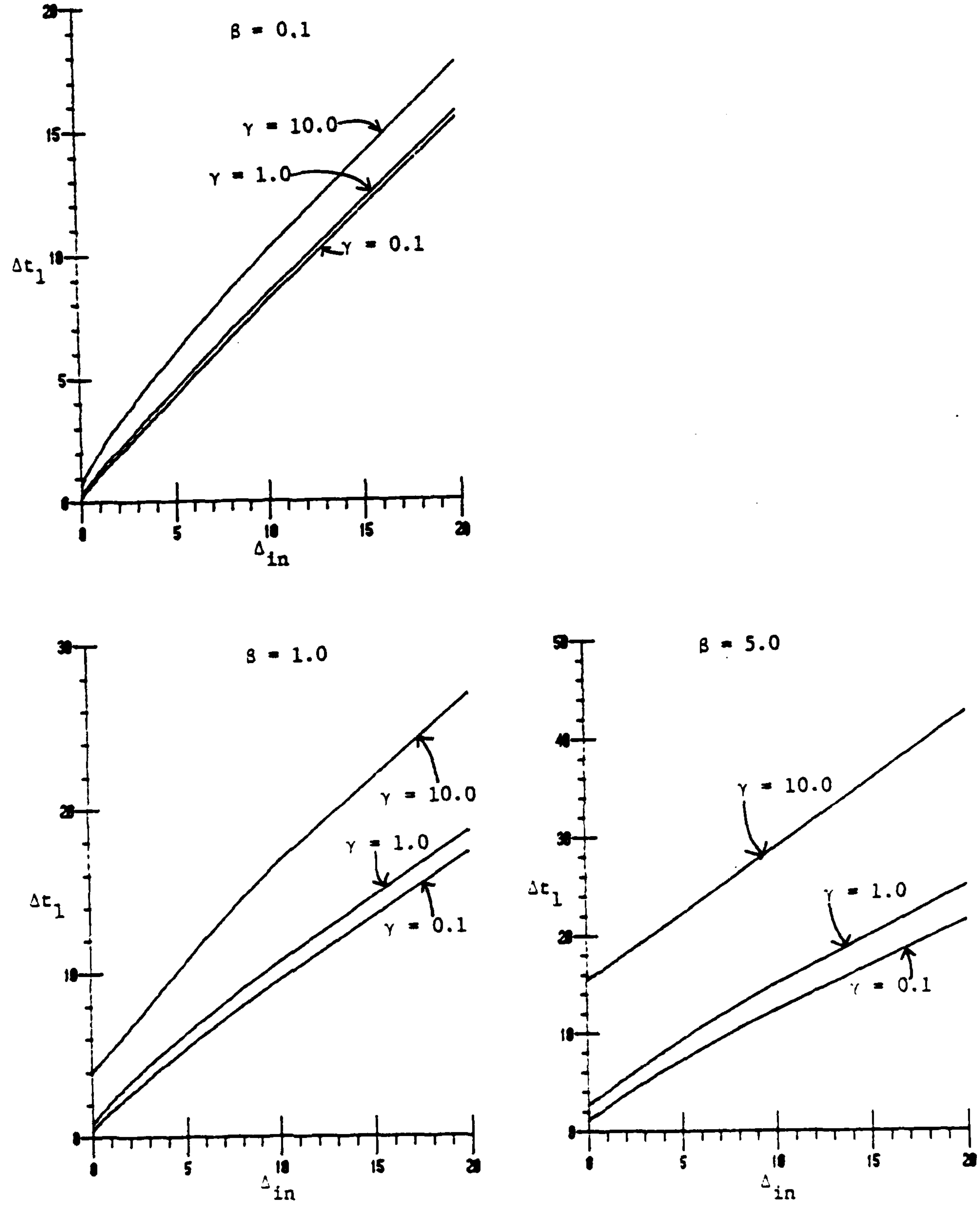
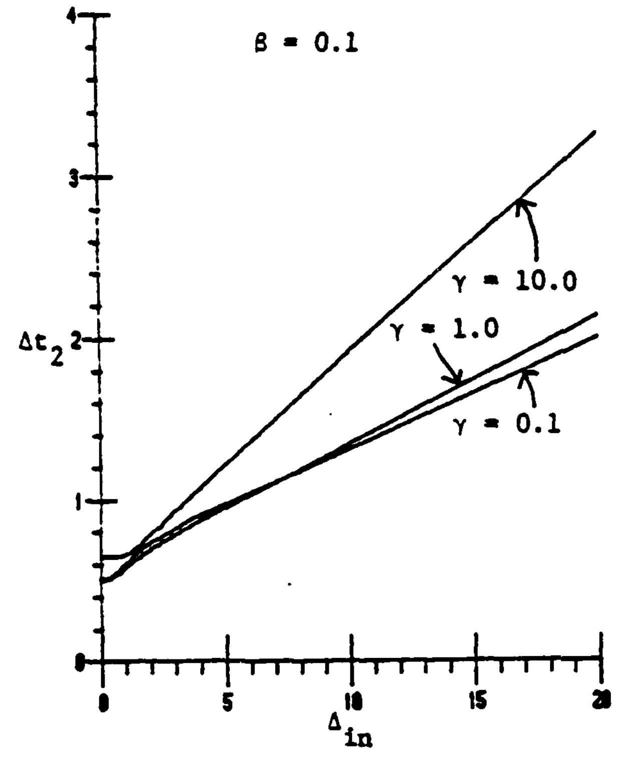
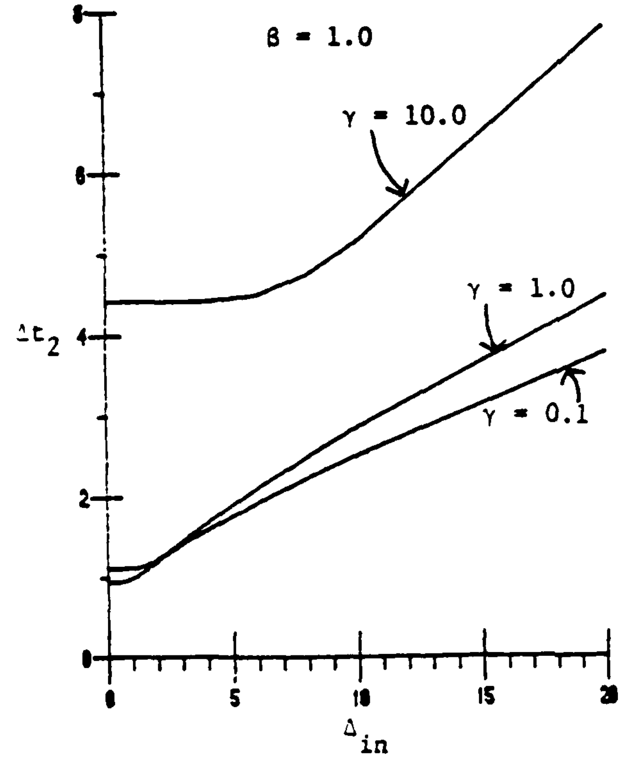


Figure A1.8 : Inertial delay for standard primitive 5, type "0"





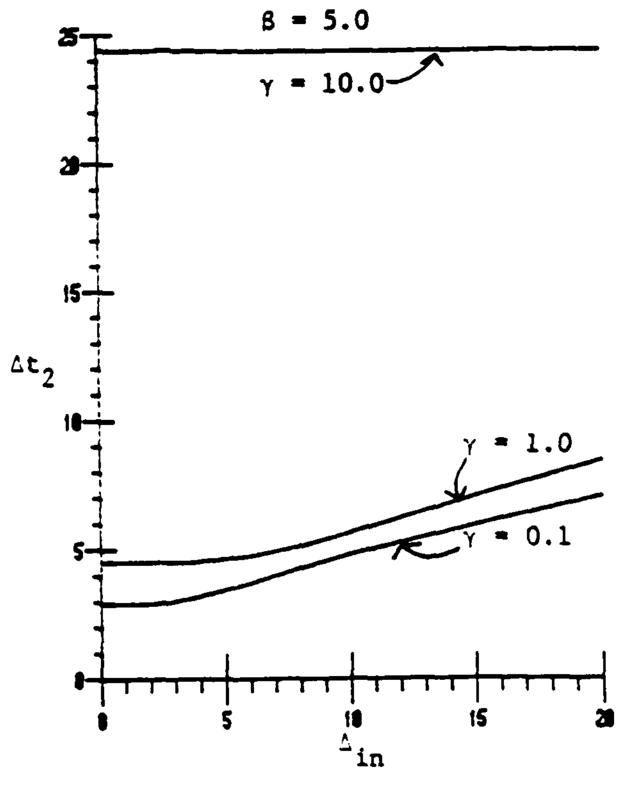


Figure A1.9 : Fall delay for standard primitive 5, type "0"

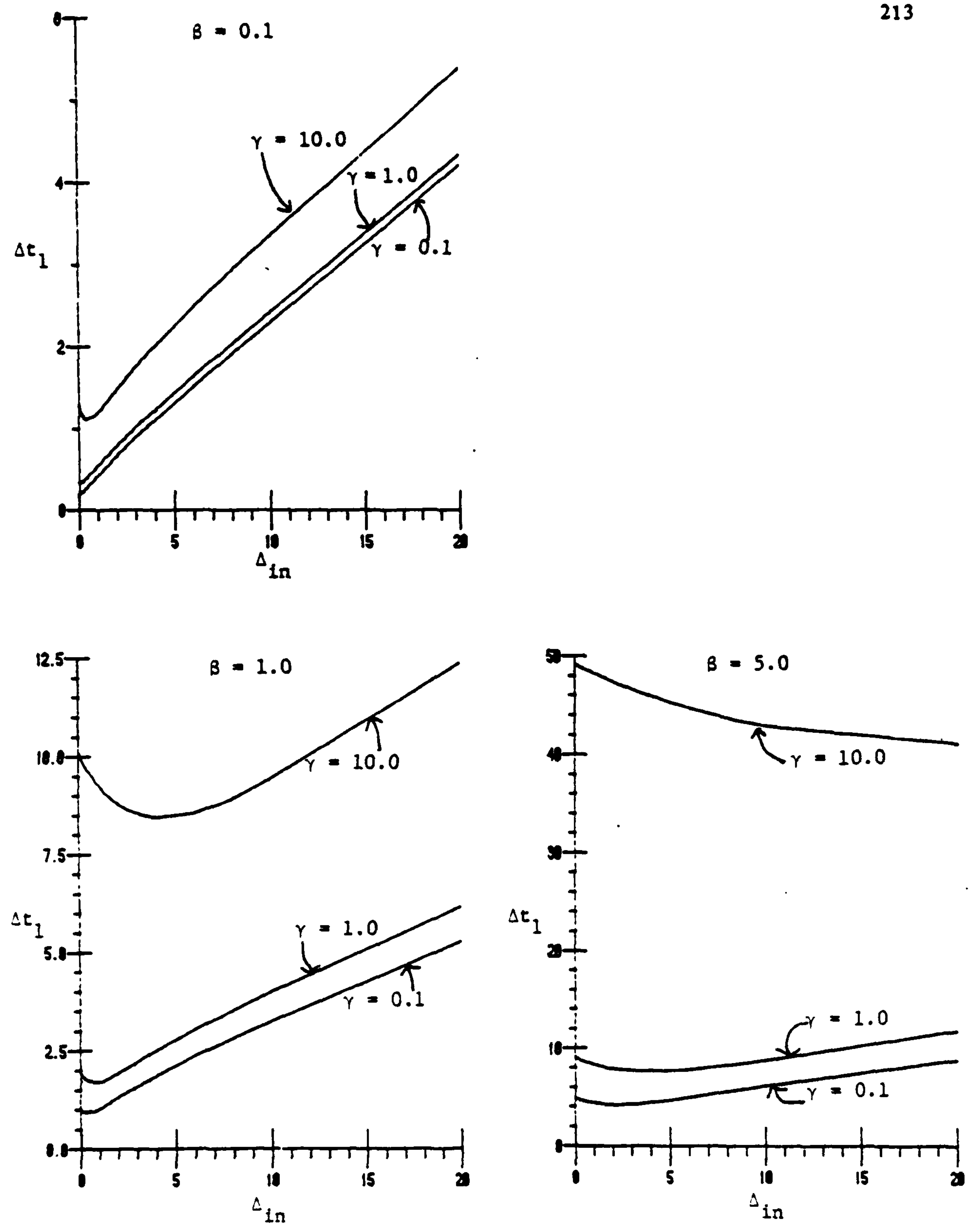
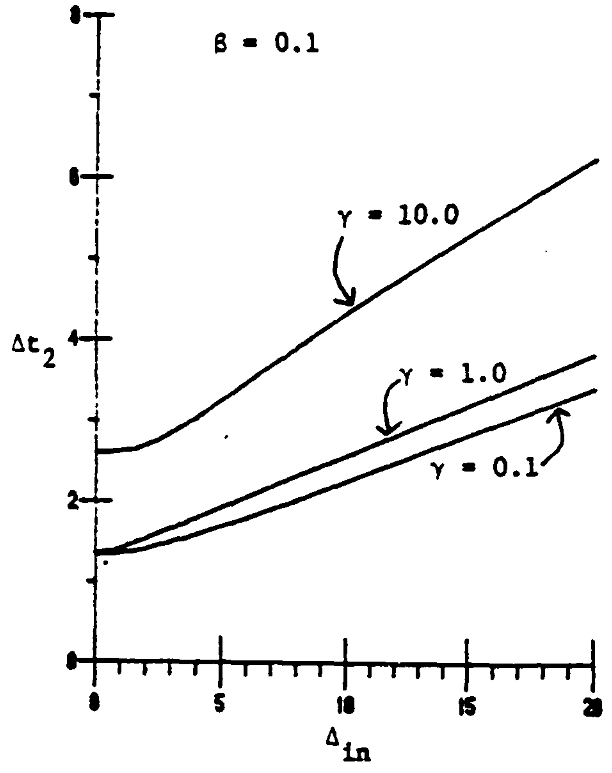
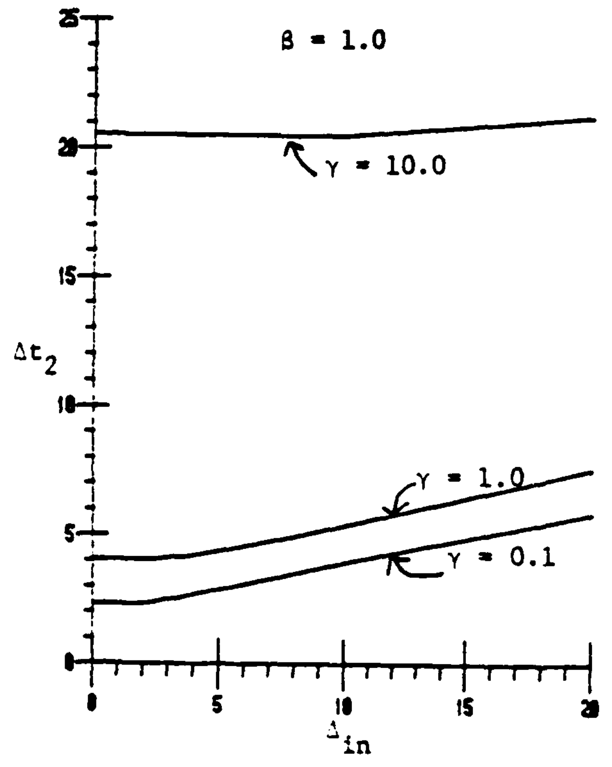


Figure A1.10 : Inertial delay for standard primitive 5, type "1"





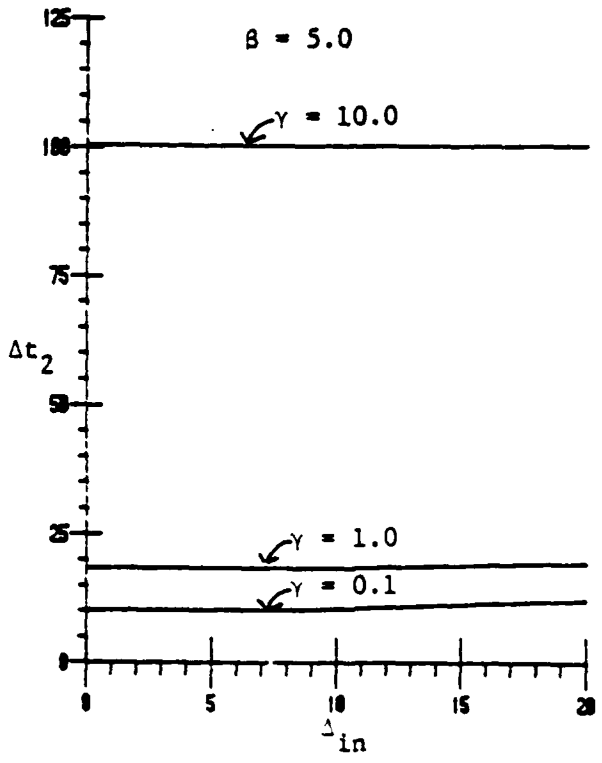


Figure A1.11 : Rise delay for standard primitive 5, type "1"

APPENDIX II

MINIMUM FEEDBACK ARC SETS FOR DIRECTED GRAPHS

A minimum feedback arc set for a directed graph is a minimum set of arcs which if removed leaves the resultant graph free of directed cycles. This problem has attracted the interest of both mathematicians and engineers over the recent years. Feedback is inherent in most engineering applications, such as sequential switching circuits, control mechanisms, regulatory devices, and large-scale systems. A good deal of success has been achieved, however, in analyzing complicated systems without feedback. Therefore, in order to analyze systems with feedback, an appropriate number of *feedback loops* are broken to reduce the system to one without feedback. The complexity of this analysis, on the other hand, increases drastically with the number of loops to be broken; hence, a knowledge of minimum feedback arc sets would be extremely useful.

The problem of finding minimum feedback arc sets (FAS) in directed graphs, when phrased as a decision problem, is known to be NP-Complete [52,53,57]. This problem remains NP-Complete even for a restricted class of graphs such as line-digraphs [58]. In this appendix we shall study an algorithm proposed by D.H.Younger [60] which attempts to solve the FAS problem by establishing a relationship between feedback arc sets and orderings on the vertices of a digraph. It will be shown that this algorithm does indeed find a minimum feedback arc set in any digraph, but could take exponential time, as should be expected, on certain digraphs. The problem of finding minimum feedback arc sets for planar graphs, however, has been shown to be solvable in polynomial time [59].

We begin by establishing a relationship between feedback arcs of a digraph and orderings on its vertex set. In fact, a minimum feedback arc set is shown to be determined by an optimum ordering **R** of vertices which minimizes the number of arcs (u,v) such that $\mathbf{R}(u) \geqslant \mathbf{R}(v)$. A key concept used to find optimum orderings is that of an *admissible ordering* [60]. While finding optimum orderings may be hard (since the problem is NP-Complete), we will show that finding admissible orderings is relatively much easier since it can be done in polynomial time. For most digraphs of interest to the practical user, admissible orderings turn out to be almost as "good" as optimum orderings in that they generate "fairly small" feedback arc sets.

We begin with some definitions and notations. For a directed graph a *feedback arc set* is a set of arcs which, if removed, leaves the resultant graph free of directed cycle. A feedback arc set is *minimum* if no other feedback arc set for that digraph has fewer arcs. For any sequential ordering **R** on the vertices of a digraph $\mathbf{G(V,A)}$, let $\mathbf{F_R=\{(u,v)\in A}$ such that $\mathbf{R(u)\geqslant R(v)\}}$ designate the feedback arc set determined by **R**. Also let $\mathbf{Q(R)=|F_R|}$.

Definition : A sequential ordering $\mathbf{R^*}$ is said to be an *optimum ordering* if $\mathbf{Q(R^*)\leqslant Q(R)}$ for all sequential orderings **R**. Given a sequential ordering **R** of a digraph, a *consecutive subgraph* is an induced subgraph on any (non-empty) set of vertices that are consecutively ordered by **R**.

We are now ready to state some properties of optimum orderings from [60].

Theorem A2.1 : A feedback arc set **F** of a digraph **G** is minimum if and only if there exists an optimum ordering **R** such that $\mathbf{F=F_R}$.

Proof : See [60].

The above theorem clearly illustrates the equivalence between optimum orderings and minimum feedback arc sets of a digraph. Hence the problem of finding optimum orderings is indeed NP-Complete.

Theorem A2.2 : The set of optimum orderings for a given digraph is invariant under the removal of self-loops and directed cycles involving two arcs.

Proof : See [60].

In accordance with the above theorem, two digraphs are said to be *order equivalent* if the removal of all self-loops and two-cycles from each digraph results in isomorphic graphs. A subgraph of a digraph obtained by removing all self-loops and two-cycles is called the *reduced graph.* Therefore, an optimum ordering for a digraph is also an optimum ordering for its reduced graph. It must be noted, however, that a minimum feedback arc set of a reduced graph is only a subset of some minimum feedback arc set of the original graph.

Theorem A2.3 : Given an optimum ordering $\mathbf{R}$ of a digraph $\mathbf{G(V,A)}$, let $\mathbf{G_1}$ be any consecutive subgraph of $\mathbf{G}$ according to $\mathbf{R}$, and define

$\mathbf{F_{1,R}=\{(u,v) : R(u) \geqslant R(v)}$ and $\mathbf{u,v \in V(G_1)\}}$ and $\mathbf{F_{2,R}=F_R-F_{1,R}}$. Then

(a) $\mathbf{F_{1,R}}$ must be a minimum feedback arc set of $\mathbf{G_1}$;

(b) $\mathbf{F_{2,R}}$ must be a minimum feedback arc set of the subgraph $\mathbf{H}$ obtained from $\mathbf{G}$ by deleting all arcs and coalescing all vertices of $\mathbf{G_1}$.

Proof : See [60].

It follows from part a) of the above theorem that for an optimum ordering $\mathbf{R}$ on a digraph $\mathbf{G}$, for any two vertices $\mathbf{u}$ and $\mathbf{v}$ such that $\mathbf{R(v)=R(u)+1}$, the number of arcs from $\mathbf{u}$ to $\mathbf{v}$ is no less than the number from $\mathbf{v}$ to $\mathbf{u}$. In fact, a much stronger result follows.

Notation : Suppose $\mathbf{G_1}$ and $\mathbf{G_2}$ are two disjoint induced subgraphs of a digraph $\mathbf{G}$. We use $\mathbf{(G_1,G_2)}$ to denote the set of arcs in $\mathbf{G}$ with tail vertex in $\mathbf{G_1}$ and head vertex in $\mathbf{G_2}$. Given an ordering $\mathbf{R}$, two disjoint consecutive subgraphs $\mathbf{G_1}$ and $\mathbf{G_2}$ are said to form an *R-adjacent pair*, denoted by $\mathbf{[G_1,G_2]}$ if

$$\min\{\mathbf{R(v) : v \in G_2}\}=\max\{\mathbf{R(u) : u \in G_1}\}+1.$$

Theorem A2.4 : Given an optimum ordering $\mathbf{R}$ for a digraph $\mathbf{G}$, let $\mathbf{[G_1,G_2]}$ be an R-adjacent pair of disjoint consecutive subgraphs of $\mathbf{n_1}$ and $\mathbf{n_2}$ vertices, respectively. Then

a) $\mathbf{|(G_1,G_2)| \geqslant |(G_2,G_1)|}$, and

b) if $\mathbf{|(G_1,G_2)|=|(G_2,G_1)|}$, then the ordering $\mathbf{R'}$, obtained from $\mathbf{R}$ as follows, is also optimum :

$\mathbf{R'(u)=R(u)}$ if $\mathbf{u}$ is neither in $\mathbf{G_1}$ nor in $\mathbf{G_2}$.

$\mathbf{R'(u)=R(u)-n_1}$ if $\mathbf{u \in G_2}$.

$R'(u)=R(u)+n_2$ if $u \in G_1$.

Proof : See [60].

Definition : A feedback arc set for a digraph is *minimal* if it contains no proper subset that is also a feedback arc set for this graph.

Definition : An ordering **R** for a digraph **G** is said to be *admissible* if

a) The condition $|(G_1,G_2)| \geq |(G_2,G_1)|$ is satisfied by all R-adjacent pairs $[G_1,G_2]$ of disjoint consecutive subgraphs of **G**, and

b) The feedback arc set F_R determined by **R** is minimal.

By definition and by Theorem A2.4 a) it is clear that all optimum orderings of **G** are also admissible. However, there might be admissible orderings that are not optimum. We shall show that starting from any arbitrary ordering of a digraph it is possible to obtain admissible orderings in polynomial time. Hence for a class of digraphs in which an admissible ordering is also an optimum ordering in each digraph, finding minimum feedback arc sets is indeed solvable in polynomial time.

The strategy we wish to employ to find optimum orderings is to start with any arbitrary ordering and first obtain an admissible ordering. The vertices of the graph are relabeled as $a',b',c',\cdots$ according to this new ordering which we will refer to as the *admissible reference* ordering. This ordering is then *selectively perturbed* to obtain a new admissible ordering with fewer feedback arcs (if one exists) which then becomes the admissible reference and the process is repeated till an optimum ordering is found. We need some more terminology and results before going into the description of the entire algorithm.

Definition : Two sequential orderings of a digraph are *F-identical* if they determine the same feedback arc set. An *F-identical class* of orderings is a set of orderings all of which are F-identical. Given an admissible reference ordering R_{ref} and an F-identical class Ψ, the ordering in Ψ that is *lexicographi-*

cally closest to $\mathbf{R}_{ref}$ is said to be the *F-representative* of Ψ. Given a digraph with vertices labeled according to some admissible reference ordering $\mathbf{R}_{ref}$ and given any arbitrary ordering $\mathbf{R}$, a *sequent* derived from $\mathbf{R}$ is an ordered pair of vertices $[\mathbf{u},\mathbf{v}]$ for which $\mathbf{R}(\mathbf{v})=\mathbf{R}(\mathbf{u})+1$. If, further, $\mathbf{R}_{ref}(\mathbf{u})<\mathbf{R}_{ref}(\mathbf{v})$ then $[\mathbf{u},\mathbf{v}]$ is an *up-sequent;* if $\mathbf{R}_{ref}(\mathbf{u})>\mathbf{R}_{ref}(\mathbf{v})$ then $[\mathbf{u},\mathbf{v}]$ is a *down-sequent.*

Theorem A2.5 : In a *reduced graph* G whose vertices are labeled according to an admissible reference ordering $\mathbf{R}_{ref}$, given an F-identical class Ψ with an admissible F-representative $\mathbf{R}_F$, there exists one or more arcs $(\mathbf{u},\mathbf{v})$ in G for every down-sequent $[\mathbf{u},\mathbf{v}]$ derived from $\mathbf{R}_F$.

Note : The arcs from **u** to **v** are forward arcs under $\mathbf{R}_F$ but are feedback arcs under $\mathbf{R}_{ref}$.

Proof : (See [60]). Since G is a reduced graph, there cannot be arcs both from **u** to **v** and from **v** to **u**. So, if we eliminate the possibilities of one or more arcs from **v** to **u**, or no arcs between **u** and **v** in G, then we are done.

Suppose G has one or more arcs from **v** to **u**. Since $[\mathbf{u},\mathbf{v}]$ is a sequent *derived from* $\mathbf{R}_F$ (i.e., $\mathbf{R}_F(\mathbf{v})=\mathbf{R}_F(\mathbf{u})+1$), reversing the order of **u** and **v** in $\mathbf{R}_F$ produces an ordering with a feedback arc set that is a proper subset of that determined by $\mathbf{R}_F$, thereby contradicting the minimality and, hence, the admissibility of $\mathbf{R}_F$. Now suppose that there are no arcs between **u** and **v** in G. The ordering produced from $\mathbf{R}_F$ by switching the positions of **u** and **v** is then lexicographically closer to the reference $\mathbf{R}_{ref}$ than $\mathbf{R}_F$, while having the same set of feedback arcs (i.e., the new ordering is also in Ψ), which is a contradiction to the designation of $\mathbf{R}_F$ as the F-representative of Ψ. □

We now begin by describing an algorithm, which, for any given directed graph $\mathbf{G}(\mathbf{V},\mathbf{A})$ and some arbitrary initial ordering $\mathbf{R}$, obtains an admissible ordering $\mathbf{R}_A$. The algorithm to find optimum orderings then treats $\mathbf{R}_A$ as a reference and selectively perturbs it to obtain a better ordering. This procedure is iterated until an optimum ordering is obtained.

Main program

INPUT : Reduced graph $G(V,A)$ and initial ordering R_{init}.

OUTPUT : An optimum ordering R_{opt} of G.

```
BEGIN
        ADMISSIBLE ($G(V,A)$ , $R_{init}$ , $R$)
        $p \leftarrow 0$
        OPTIMUM ($G(V,A)$ , $R$ , $R'$ , $p$)
        IF $p=1$ THEN
                $R \leftarrow R'$
                GO TO step 2)
        ELSE
                $R_{opt} \leftarrow R$
                STOP
        ENDIF
END
```

```
subroutine ADMISSIBLE ($G(V,A)$ , $R$ , $R_A$)
BEGIN
        1) $i \leftarrow 0$ ; $R_0 \leftarrow R$ ; $n \leftarrow |V|$
        2) CONSEC ($G(V,A)$ , $R_i$ , $R_{in}$ , $p$)
        3) MINIMAL ($G(V,A)$ , $R_{in}$ , $R_{i+1}$ , $q$)
        4) IF $p=1$ or $q=1$ THEN
                $i \leftarrow i+1$
                GO TO step 2)
        ELSE
                $R_A \leftarrow R_i$
                RETURN $R_A$
        ENDIF
END
```

```
subroutine CONSEC ($G(V,A)$ , $R$ , $R'$ , $p$)
BEGIN
        1) $p \leftarrow 0$.
        2) Relabel vertices of G as $v_1,v_2,\ldots,v_n$
           such that $R(v_i)=i$ for each $i=1,2,\ldots,n$
        3) FOR $i=1$ TO $n$ DO
                $R'(v_i) \leftarrow R(v_i)$
        4) FOR $i=1$ TO $n-2$ DO
            BEGIN
                FOR $j=i+1$ TO $n-1$ DO
                  BEGIN
                        FOR $k=j+1$ TO $n$ DO
                          BEGIN
        5)                        $V_1 \leftarrow \{v_i, v_{i+1}, \ldots, v_{j-1}\}$
                                  $V_2 \leftarrow \{v_j, v_{j+1}, \ldots, v_{k-1}\}$
                                  $n_1 \leftarrow |V_1|$
                                  $n_2 \leftarrow |V_2|$
                                  $G_1 \leftarrow G[V_1]$
                                  $G_2 \leftarrow G[V_2]$
        6)                        $S_1 \leftarrow |(G_1, G_2)|$
                                  $S_2 \leftarrow |(G_2, G_1)|$
        7)                        IF $S_1 < S_2$ THEN
                                       $p \leftarrow 1$
                                       FOR $m=i$ TO $j-1$ DO
                                         $R'(v_m) \leftarrow R(v_m) + n_2$
                                       FOR $m=j$ TO $k-1$ DO
                                         $R'(v_m) \leftarrow R(v_m) - n_1$
                                       RETURN R'
                                  ENDIF
                              END
                         END
                    END
END


subroutine MINIMAL (G(V,A), R, R', q)
BEGIN
        1) $q \leftarrow 0$
        2) FOR EACH vertex $v \in V$ DO
               $R'(v) \leftarrow R(v)$
        3) $F_1 \leftarrow \{(u,v) \in A : R(u) \geq R(v)\}$
        4) $G' \leftarrow G - F_1$
        5) $F_2 \leftarrow F_1$
        6) FOR EACH arc $a \in F_1$ DO
             BEGIN
               $G' \leftarrow G' + a$
               IF G' is still acyclic THEN
                     $F_2 \leftarrow F_2 - a$
                     $q \leftarrow 1$
               ELSE
                     $G' \leftarrow G' - a$
               ENDIF
             END
        7) IF $q=1$ THEN
               R' <– topological ordering on G'
           ENDIF
        RETURN R'
END


subroutine OPTIMUM (G(V,A), R, R', p)
BEGIN
```

```
        1) Relabel vertices of G according to R ; $R' \leftarrow R$
        2) $i \leftarrow 1$ ; $G_1 \leftarrow G$ ; $R_1 \leftarrow R$ ; $I_1 \leftarrow 0$;
           $Q_1 \leftarrow |\{(u,v) \in A(G) : R_1(u) \geqslant R_1(v)\}|$ ; $M \leftarrow Q_1$.
        3) TREE (i , M)
        4) NEXTSON (i , j , noson)
        5) IF noson=1 THEN
                IF i=1 THEN
                        $p \leftarrow 0$
                        $R' \leftarrow R$
                        RETURN R'
                ELSE
                        $i \leftarrow FATHER(i)$
                        GO TO step 4)
                ENDIF
           ELSE
                $i \leftarrow j$
           ENDIF
        6) IF $Q_i < M - I_i$ THEN
                $p \leftarrow 1$
                $R' \leftarrow R$
                RETURN R'
           ELSE
                GO TO step 3)
           ENDIF
END


subroutine TREE (i , M)
BEGIN
        1) $F \leftarrow \{(u,v) \in A(G_i) : R_i(u) \geqslant R_i(v)\}$
        2) FOR EACH arc $(u,v) \in F$ DO
                BEGIN
                        UNITE $(u,v,G_i,R_i,G',R',I')$
                        IF $M-(I_i+I') \geqslant 0$ THEN
                                $ntree \leftarrow ntree+1$
                                $j \leftarrow ntree$
                                $I_j \leftarrow I_i + I'$
                                $G_j \leftarrow G'$
                                ADMISSIBLE $(G',R',R_j)$
                                $Q_j \leftarrow |\{(x,y) \in A' : R'(x) \geqslant R'(y)\}|$
                                $FATHER(j) \leftarrow i$
                                $SON(i) \leftarrow SON(i) \cup \{j\}$
                        ENDIF
                END
           RETURN
END
```

The subroutine ADMISSIBLE starts with an ordering R_i and calls CONSEC to check if it satisfies condition a) of admissibility. If it does (indicator p=0) then there is no change; however, if not (indicator $p=1$), then an intermediate ordering $\mathbf{R}_{in}$ is produced with fewer feedback arcs. Subroutine MINIMAL is then called to check for minimality of the feedback arc set $\mathbf{F}_1$ of $\mathbf{R}_{in}$. If $\mathbf{F}_1$ is found minimal (indicator $q=0$) then, again, there is no change, otherwise (indicator $q=1$), the minimal proper subset $\mathbf{F}_2$ is found and a new ordering $\mathbf{R}_{i+1}$ with this as its feedback arc set is obtained. If either $p=1$ or $q=1$, then $Q(\mathbf{R}_{i+1})<Q(\mathbf{R}_i)$, in which case $\mathbf{i}$ is incremented by 1 and the process is repeated. In fact $Q(\mathbf{R}_{i+1})=Q(\mathbf{R}_i)$ if and only if both $p=0$ and $q=0$, in which case the program halts.

Theorem A2.6 : Given a digraph $\mathbf{G(V,A)}$ with $\mathbf{n}=|\mathbf{V}|$ and $\alpha=|\mathbf{A}|$ and any initial sequential ordering, the subroutine ADMISSIBLE halts at an admissible ordering and the number of computations involved is bounded above by a polynomial $\mathbf{P}(\mathbf{n},\alpha)$ in $\mathbf{n}$ and α.

Proof : Let $\mathbf{R}_0,\mathbf{R}_1,\mathbf{R}_2,\ldots,\mathbf{R}_i,\cdots$ be the sequence of orderings produced during each iteration of subroutine ADMISSIBLE. Let $\mathbf{m}_i=Q(\mathbf{R}_i)$ be the number of feedback arcs determined by $\mathbf{R}_i$. Since $\mathbf{m}_i \geqslant \mathbf{m}_{i+1} \geqslant 0$ for each $\mathbf{i}$, there exists a smallest integer $\mathbf{s}$ such that $\mathbf{m}_s=\mathbf{m}_{s+1}$ and $\mathbf{m}_i>\mathbf{m}_{i+1}$ for each $0 \leqslant i < s$. Therefore the program halts after $\mathbf{s}$ iterations. At this stage both indicators p and q must be 0 which means that $\mathbf{R}_s$ must be admissible. Clearly $s \leqslant \mathbf{m}_0 \leqslant \alpha$; therefore, the number of iterations is at most the number of arcs in $\mathbf{G}$.

During each call, steps 2) and 3) of CONSEC together involve at most $\mathbf{2n}$ computations, while steps 5), 6), and 7) require at most $(\mathbf{2n}+\alpha)$ computations for each R-adjacent pair $[\mathbf{G}_1,\mathbf{G}_2]$.

Lemma : Given a digraph $\mathbf{G(V,A)}$ with $\mathbf{n}=|\mathbf{V}|$, and an ordering $\mathbf{R}$, the number of R-adjacent pairs $[\mathbf{G}_1,\mathbf{G}_2]$ of disjoint consecutive subgraphs of $\mathbf{G}$ is $(\mathbf{n}+1)\mathbf{n}(\mathbf{n}-1)/6$.

Proof : Relabel the vertices of $\mathbf{G}$ as $\mathbf{v}_1,\mathbf{v}_2,\ldots,\mathbf{v}_n$ according to R. Arrange $\mathbf{n}$ dots labeled $1,2,\ldots,\mathbf{n}$ on a straight line in ascending order fr m left to right. Place dummy dots $\mathbf{0}$ on the left of $\mathbf{1}$ and $\mathbf{n+1}$ to the right of $\mathbf{n}$. We now have a linear arrangement of $\mathbf{n+2}$ dots creating $\mathbf{n+1}$ empty spaces between them. If we pick any three spaces among the $\mathbf{n+1}$ empty spaces and place a slash (/) in each of them, then we can associate $\mathbf{V}_1$ to be the vertices corresponding to dots between the first and second slashes while $\mathbf{V}_2$ to those between the second and third slashes. $\mathbf{G}_1$ and $\mathbf{G}_2$ are then the consecutive subgraphs of **G** induced by V_1 and V_2, respectively. Hence the proof of the lemma.

Therefore, the total number of computations performed during each call to CONSEC is at most $(2n+(2n+\alpha)\times(n^3-n)/6)$. In subroutine MINIMAL step 6) requires at most $n\times\alpha$ computations while the other steps would need at most $n+3\alpha$ computations. Thus each iteration of ADMISSIBLE performs at most $Q(n,\alpha)=2n+(2n+\alpha)\times(n^3-n)/6+n+3\alpha+n\alpha$ computations. Since the number of iterations is at most α we have $P(n,\alpha)=\alpha Q(n,\alpha)$ as the upper bound on the total number of computations involved in obtaining an admissible ordering for **G**. □

We now consider the algorithm to find an optimum ordering of a digraph. By Theorem A2.2 we need consider only reduced graphs. So, for a reduced graph **G(V,A)** with some arbitrary initial ordering, an admissible reference ordering **R** is first obtained. For each feedback arc of **R** a cyclic shift by one order position is performed on the vertices of a consecutive subgraph, where the subgraph has, before the shift, the feedback arc connecting its two extreme vertices. Of the two possible directions for this cyclic shift, it is convenient to choose the one which results in fewer feedback arcs. This results in a new ordering which has a down-sequent corresponding to the feedback arc of **R**. This results in **Q(R)** new orderings which are made admissible by passing them through subroutine ADMISSIBLE. If one of these admissible orderings **R'** is better than **R**, i.e., **Q(R')<Q(R)** then **R'** is treated as a new reference. If one of the initial perturbations does not establish a new reference, then each of these is selectively perturbed in a similar way and thus the search branches out. It is clear that we are only looking at orderings whose down-sequents are feedback arcs of **R**. The following result justifies this approach.

Theorem A2.7 : Given a reduced graph **G(V,A)** and an admissible reference ordering **R**. If **R** is not optimum then there exists an ordering **R'** with **Q(R')<Q(R)** such that every down-sequent of **R'** corresponds to a feedback arc of **R**.

Proof : Label the vertices of **G** according to the reference ordering **R**. Let R_o be an optimum ordering of **G**. Since **R** is not optimum $Q(R_o)<Q(R)$. Let Ψ be the F-identical class containing R_o. Let **R'** be the F-representative of Ψ. Since **R'** is optimum, it is also admissible. Also **R'** cannot be the same as **R** since

$Q(R')<Q(R)$ and so must have at least one down-sequent. But by Theorem A2.5 every down sequent $[u,v]$ of R' must correspond to an arc (u,v) in G. Since $[u,v]$ is a down-sequent, we must have $R(u) \geqslant R(v)$, by definition. Hence (u,v) is a feedback arc of R.

We now describe a limiting mechanism which keeps the search for better orderings from becoming extremely unwieldy. It is useful to imagine a tree which grows from a root vertex (labeled 1). Associated with each vertex i of this tree is a reduced graph G_i, an admissible ordering R_i on the vertices of G_i, $Q_i=Q(R_i)$ and an integer I_i which indicates the difference between the minimum number of feedback arcs of G and G_i. Initially $G_1=G$, $R_1=R$ and $I_1=0$, and $M=Q(R)$. The subroutine TREE(i,M) creates 'sons' for vertex i in the tree as follows:

> For each feedback arc of G_i according to R_i a cyclic shift is performed to establish the end points of this arc as a down sequent. The two vertices of this down-sequent are united into a single vertex and all self-loops and 2-cycles created by this union are eliminated resulting in a reduced graph G' and an ordering R'. Let I' be the number of 2-cycles thus eliminated. Each vertex of G' thus corresponds to a consecutive subgraph of the original graph G. The down-sequent, say $[u,v]$, which gets united into a single vertex, gets an equivalent label which is the label of v appended to the label of u. Thus, R' can also be treated as an ordering of G by reading off the labels of G' in order according to R'. A 'son' j is created only if $M-(I_i+I')\geqslant 0$, in which case $G_j=G'$, $R_j=R'$, and $I_j=I_i+I'$. If $M-(I_i+I')<0$ then it means that any ordering that will be derived from R' by the above procedure will have at least I_i+I' arcs of G in its feedback arc set; therefore, an ordering better than the original R can never be obtained this way.

We would now like to make a few comments about the computational complexity of this procedure. The number of iterations in the main algorithm is again at most the number of arcs of G, since successive orderings are better than the previous ones. So if computations within subroutine OPTIMUM can be performed in polynomial time then, indeed, the entire algorithm runs in polynomial time. This is impossible since by Theorem A2.7 this algorithm indeed terminates in an optimum ordering, while obtaining one is known to be NP-Complete. However, if one examines the computations within subroutine OPTIMUM, the only quantity that can grow exponentially with $n=|V|$ is the number of vertices of the tree. It would be interesting to find such a digraph on n vertices for any general n. An upper bound on the depth of the tree is $n-1$ since the leaves of the tree correspond to two-vertex digraphs and the digraph associated with a son has one vertex less than that associated with its father. So, even bounding the number of sons by k gives us at most k^n vertices in the tree which does not help.

We now illustrate with an example the use of the above algorithm. Consider the reduced graph G(V,A) shown in Figure A2.1. The natural ordering a,b,c,d,e can easily be verified to be admissible. Figure A2.2 shows the tree structure of the search for a better ordering. Note that [e,c,a] represents a consecutive subgraph of G with three vertices e, c, and a appearing in that order. The search terminates at a three-vertex digraph with indicator $p=1$ meaning that the ordering b,e,c,a,d is a better ordering. Indeed this new ordering has only two feedback arcs which is one less than that of the natural ordering. The vertices are relabeled as a',b',c',d',e' according to this new admissible reference.

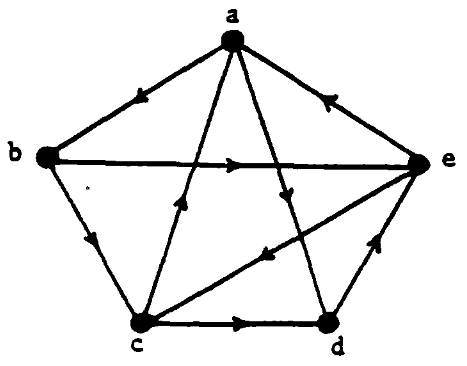


Figure A2.1 : A reduced graph G(V,A)

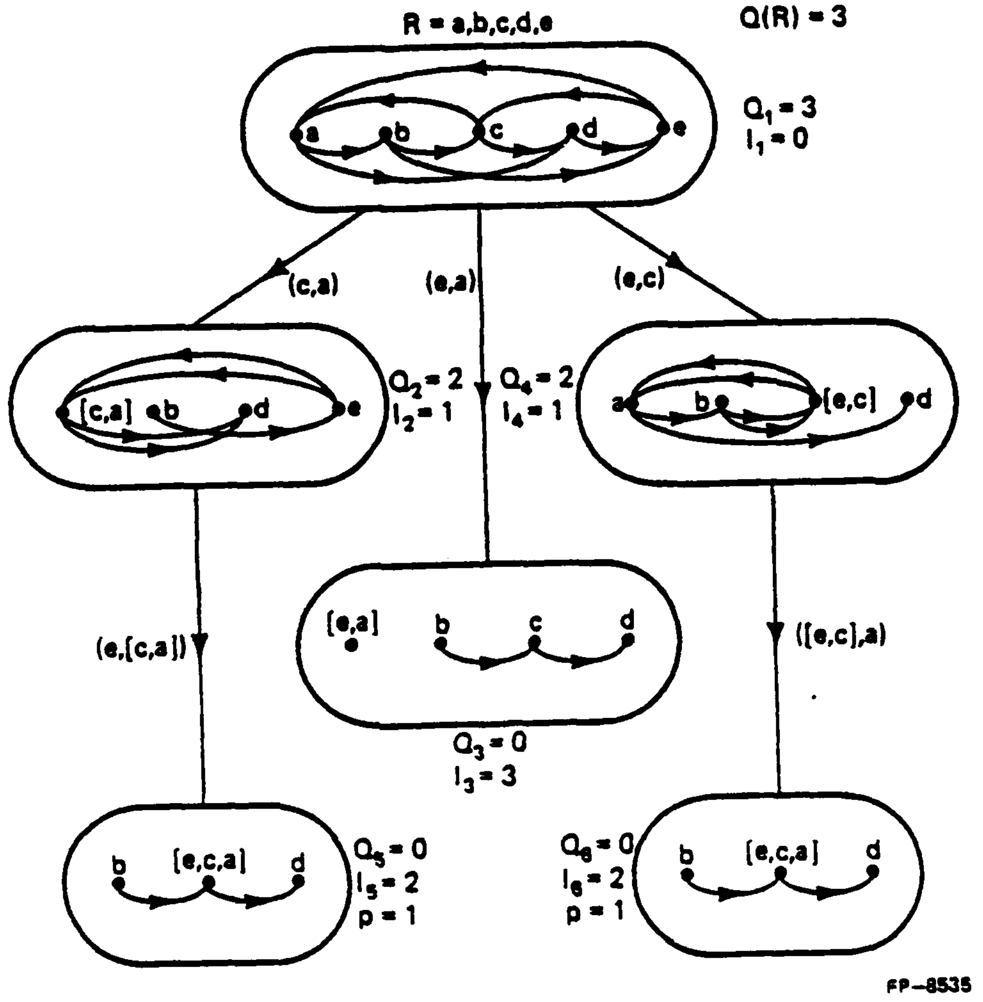


Figure A2.2 : Tree structure with R as the admissible reference

Figure A2.3 shows the tree-structure for the search for a better ordering. Since $Q_i+I_i>2$ for each vertex in the tree apart from the root, the search terminates at the root vertex with indicator $p=0$ meaning that the reference ordering is indeed optimum.

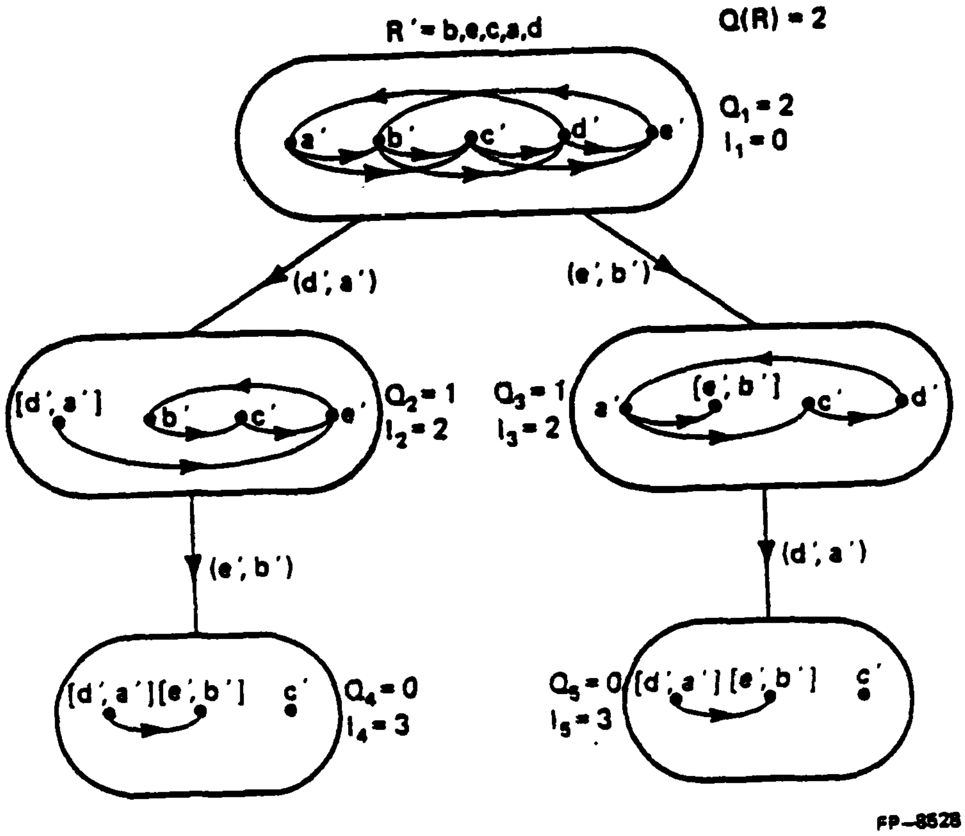


Figure A2.3 : Tree structure with R' as the admissible reference

REFERENCES

[1] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Electronics Research Laboratory Report #ERL-M520, University of California, Berkeley, May 1975.

[2] W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Qassemzadeh, and T. R. Scott, "Algorithms for ASTAP - A Network Analysis Program," *IEEE Transactions on Circuit Theory*, Vol. CT-20, pp. 628-634, November 1973.

[3] P. Yang, I. N. Hajj, and T. N. Trick, "SLATE: A Circuit Simulation Program with Latency Exploitation and Node Tearing," *Proceedings of the IEEE International Conference on Circuits and Computers*, pp. 353-355, October 1980.

[4] N. B. G. Rabbat, A. L. Sangiovanni-Vincentelli, and H. Y. Hsieh, "A Multilevel Newton Algorithm with Macromodelling and Latency for the Analysis of Large-scale Nonlinear Circuits in the Time Domain," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, pp. 733-741, September 1979.

[5] B. R. Chawla, H. K. Gummel, and P. Kozak, "MOTIS - An MOS Timing Simulator," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, pp. 901-910, December 1975.

[6] S. P. Fan, M. Y. Hsueh, A. R. Newton, and D. O. Pederson, "MOTIS-C: A New Circuit Simulator for MOS LSI Circuits," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Phoenix, Arizona, pp. 700-703, April 1977.

[7] C. F. Chen, C. Y. Lo, H. N. Nham, and P. Subramaniam, "The Second Generation MOTIS Mixed Mode Simulator," *Proceedings of the 21st Design Automation Conference*, Albuquerque, New Mexico, pp. 10-17, June 1984.

[8] Y. P. Wei, I. N. Hajj, and T. N. Trick, "A Prediction-Relaxation based Simulator for MOS Circuits," *Proceedings of the IEEE International Conference on Circuits and Computers*, New York, September 1982.

[9] E. Lelarasmee, "The Waveform Relaxation Method for the Time Domain Analysis of Large Scale Nonlinear Dynamical Systems," Ph.D. Dissertation, University of California, Berkeley, 1981.

[10] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for the Time Domain Analysis of Large Scale Integrated Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-1, No. 3, pp. 131-145, July 1982.

[11] J. White and A. L. Sangiovanni-Vincentelli, "RELAX2 : A Modified Waveform Relaxation Approach to the Simulation of MOS Digital Circuits," *Proceedings of the IEEE International Symposium on Circuits and Systems*, California, pp.756-759, May 1983.

[12] J. White and A. L. Sangiovanni-Vincentelli, "RELAX2.1 : A Waveform Relaxation based Circuit Simulation Program," *Proceedings of the IEEE Custom Integrated Circuits Conference*, Rochester, New York, pp. 232-236, May 1984.

[13] A. R. Newton, "The Simulation of Large-Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, pp. 741-749, September 1979.

[14] G. Arnout and H. De Man, "The use of Threshold Functions and Boolean-Controlled Network Elements for Macromodelling of LSI Circuits," *IEEE Journal of Solid State Circuits*, Vol. SC-13, pp. 326-332, June 1978.

[15] K. Sakallah and S. W. Director, "An Activity Directed Circuit Simulation Algorithm," *Proceedings of the IEEE International Conference on Circuits and Computers*, New York, pp. 356-360, October 1980.

[16] G. R. Case, "SALOGS - A CDC 6600 Program to Simulate Digital Logic Networks, Vol. 1 - User's Manual," *Sandia Laboratory Report SAND 74-0441*, 1975.

[17] S. A. Szygenda, "TEGAS2 - Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," *Proceedings of the Ninth ACM Design Automation Workshop*, June 1972.

[18] J. Jephson, R. McQuarrie, and R. Vogelsberg, "A Three-value Computer Design Verification System," *IBM Systems Journal*, Vol. 8, No. 3, pp. 178-188, 1969.

[19] R. E. Bryant, "An Algorithm for MOS Logic Simulation," *LAMBDA (now VLSI) magazine*, Vol. 1, No. 3, pp. 46-53, 1980.

[20] R. E. Bryant, "A Switch-level Simulation Model for Integrated Logic Circuits," Ph.D. thesis, MIT/LCS/TR-259, Massachusetts Institute of Technology, Cambridge, March 1981.

[21] R. H. Byrd, G. D. Hachtel, M. R. Lightner, and M. H. Heydemann, "Switch Level Simulation : Models, Theory and Algorithms," (to appear in) Advances in Computer-Aided Engineering Design, A. L. Sangiovanni-Vincentelli, Editor, Jai Press, 1985.

[22] V. Ramachandran, "An Improved Switch-level Simulator for MOS Circuits," *Proceedings of the 20th Design Automation Conference*, Miami Beach, Florida, pp. 293-299, June 1983.

[23] V. Ramachandran, "A Linear Time Algorithm for Race Detection in Transistor Switch-level Circuits," *Proceedings of the IEEE International Conference on Computer Design*, New York, pp. 345-348, November 1983.

[24] I. N. Hajj and D. G. Saab, "Logic and Fault Simulation of MOS Circuits Based on Symbolic Expression Generation," (submitted to) *IEEE Transactions on Circuits and Systems.*

[25] I. N. Hajj and D. G. Saab, "Symbolic Logic Simulation of MOS Circuits," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Newport Beach, California, pp. 246-249, May 1983.

[26] C. J. Terman, "RSIM - A Logic-Level Timing Simulator," *Proceedings of the IEEE International Conference on Computer Design*, New York, pp. 437-440, November 1983.

[27] C. J. Terman, "Simulation Tools for Digital LSI Design," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, 1983.

[28] V. B. Rao, T. N. Trick, and M. R. Lightner, "Hazard Detection in a Multiple Delay Logic Simulator," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Rome, Italy, pp. 72-75, May 1982.

[29] Vasant B. Rao, "Algorithms for a Multiple Delay Simulator," M.S. Thesis, Department of Electrical Engineering, University of Illinois, Urbana, March 1982.

[30] V. B. Rao, T. N. Trick, and I. N. Hajj, "A Table-driven Delay Operator Approach to Timing Simulation of MOS VLSI Circuits," *Proceedings of the IEEE International Conference on Computer Design*, New York, pp. 445-448, November 1983.

[31] R. Tarjan, "Depth First Search and Linear Graph Algorithms," *SIAM Journal of Computing*, Vol. 1, No. 2, pp. 146-160, June 1972.

[32] C. W. Ho, A. E. Ruehli, and P. A. Brennan, "The Modified Nodal Approach to Network Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, pp. 504-509, June 1975.

[33] G. D. Hachtel, R. K. Brayton, and F. G. Gustavson, "The Sparse Tableau Approach to Network Analysis and Design," *IEEE Transactions on Circuit Theory*, Vol. CT-18, pp. 101-113, January 1971.

[34] L. O. Chua and P. M. Lin, *Computer-Aided Analysis of Electronic Circuits : Algorithms and Computational Techniques*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1975, pp. 631-664.

[35] I. N. Hajj, "Sparsity considerations in Network Solution by Tearing," *IEEE Transactions on Circuits and Systems*, Vol. CAS-27, No. 5, pp. 357-366, May 1980.

[36] G. D. Hachtel and A. L. Sangiovanni-Vincentelli, "A Survey of Third Generation Simulation Techniques," *Proceedings of the IEEE*, Vol. 69, No. 10, pp. 1264-1280, October 1981.

[37] W. K. Chia, T. N. Trick, and I. N. Hajj, "Stability and Convergence Properties of Relaxation Methods for Hierarchical Simulation of VLSI Circuits," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Montreal, Canada, pp. 848-851, May 1984.

[38] D. G. Luenberger, *Optimization by Vector Space Methods.* New York: John-Wiley & Sons Inc., 1969, pp. 144-145.

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

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

[41] P. Wilcox, "Digital Logic Simulation at the Gate and Functional Level," *Proceedings of the IEEE Design Automation Conference*, New York, pp. 242-248, 1979.

[42] S. Seshu and D. N. Freeman, "The Diagnosis of Asynchronous Sequential Switching Systems," *IRE Transactions on Electronic Computing*, Vol. EC-11, No. 4, pp. 459-465, August 1962.

[43] S. A. Szygenda and E. W. Thompson, "Modelling and Digital Simulation for Design Verification and Diagnosis," *IEEE Transactions on Computers*, Vol. C-25, pp. 1242-1253, December 1976.

[44] H. N. Nham and A. K. Bose, "A Multiple Delay Simulator for MOS LSI Circuits," *Proceedings of the 17th Design Automation Conference*, pp. 610-617, June 1980.

[45] E. G. Ulrich, "Exclusive Simulation of Activity in Digital Networks," *Communications of the ACM*, Vol. 12, No. 2, pp. 102-110, February 1969.

[46] W. C. Elmore, "The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers," *Journal of Applied Physics*, Vol. 19, pp. 55-64, January 1948.

[47] P. Penfield and J. Rubinstein, "Signal Delays in RC Tree Networks," *Proceedings of the 18th Design Automation Conference*, pp. 613-617, 1981.

[48] C. Chicoix, J. Pedoussat, and N. Giambiasi, "An Accurate Time Delay Model for Large Digital Network Simulation," *Proceedings of the 13th Design Automation Conference*, pp. 54-60, June 1976.

[49] C. M. Baker, "Artwork Analysis Tools for VLSI Circuits," M.S. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Camgridge, June 1980.

[50] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. New York: North-Holland Publishing Company, 1982.

[51] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1974.

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

[53] S. Even, *Graph Algorithms*. Rockville, Maryland: Computer Science Press, 1979.

[54] Z. Kohavi, *Switching and Finite Automata Theory*. Second Edition. New York: McGraw-Hill Book Company, 1978.

[55] A. R. Newton and D. O. Pederson, "Analysis Time, Accuracy and Memory Requirement Tradeoffs in SPICE2," *Proceedings of the Eleventh Annual Asilomar Conference on Circuits, Systems and* Computers, Asilomar, California, pp. 6-9, November 1977.

[56] M. Y. Tsai, "Pass Transistor Networks in MOS Technology : Synthesis, Performance, and Testing," *Proceedings of the IEEE International Symposium on Circuits and Systems*, Newport Beach, California, pp. 509-512, May 1983.

[57] R. M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations*. New York: Plenum Press, 1972, pp. 85-103.

[58] F. Gavril, "Some NP-Complete problems on Graphs," *Proceedings of the Eleventh Conference on Information Sciences and Systems*, Johns Hopkins University, Baltimore, Maryland, pp. 91-95, 1977.

[59] C. L. Luchesi, "A Minimax Equality for Directed Graphs," Doctoral Thesis, University of Waterloo, Canada, 1976.

[60] D. H. Younger, "Minimum Feedback Arc Sets for a Directed Graph," *IEEE Transactions on Circuit Theory*, pp. 238-245, June 1963.

[61] A. Gupta, "ACE : A Circuit Extractor," *Proceedings of the ACM-IEEE 20th Design Automation Conference*," Miami Beach, Florida, pp. 721-725, June 1983.

[62] R. E. Bryant, "Race Detection in MOS Circuits by Ternary Simulation," *VLSI 83*, F. Anceau, Editor. New York: North-Holland Publishing Company, August 1983.

[63] R. E. Bryant, "A Switch-level Model and Simulator for MOS Digital Circuits," *IEEE Transactions on Computers*, Vol. C-33, No. 2, pp. 160-177, February 1984.

[64] J. A. Brzozowski and M. Yoeli, "On a Ternary Model of Gate Networks," *IEEE Transactions on Computers*, Vol. C-28, No. 3, pp. 178-183, March 1979.

[65] T. Lengauer and S. Naher, "Delay-independent Switch-level Simulation of Digital MOS Circuits," (a pre-print from) *VLSI : Algorithms and Architectures*, Amalfi, Italy, May 1984.

[66] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms : Theory and Practice*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1977.

[67] D. Coppersmith and S. Winograd, "On the Asymptotic Complexity of Matrix Multiplication," *SIAM Journal on Computing*, Vol. 11, No. 3, pp. 472-492, August 1982.

VITA

Vasant Rao was born in Bangalore, India, on July 25, 1959. He received his Bachelor of Technology degree in Electrical Engineering (Electronics) from the Indian Institute of Technology, Madras, India, in June 1980. In August 1980 he entered the University of Illinois at Urbana-Champaign and received his M.S. degree in Electrical Engineering in March 1982. From August 1980 to December 1984 he worked as a Research Assistant at the Coordinated Science Laboratory, Urbana, and as a Teaching Assistant with the Department of Electrical Engineering at the University of Illinois. He has accepted a position as an Assistant Professor in Electrical Engineering at the University of Illinois at Urbana-Champaign. His research interests include the areas of simulation of VLSI circuits, computer-aided design, semiconductor device modeling, and combinatorial and graph algorithms.