



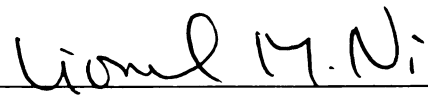
This is to certify that the
dissertation entitled

APPLICATIONS OF LOGICAL CIRCUIT EXPRESSIONS
TO CMOS VLSI DESIGN AUTOMATION
presented by

Ching-Farn Eric Wu

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science


Major professor

Date 9/21/87

**APPLICATIONS OF LOGICAL CIRCUIT EXPRESSIONS
TO CMOS VLSI DESIGN AUTOMATION**

By

Ching-Farn Eric Wu

A DISSERTATION

**Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of**

DOCTOR OF PHILOSOPHY

**Department of Computer Science
Michigan State University
East Lansing, MI 48824**

1987

ABSTRACT

APPLICATIONS OF LOGICAL CIRCUIT EXPRESSIONS TO CMOS VLSI DESIGN AUTOMATION

By

Ching-Farn Eric Wu

CMOS technology has been recognized as a leading contender for existing VLSI systems, and is projected by industry analysts as being the dominant technology for the next decade. In this thesis, a novel approach for representing CMOS logic circuit networks at the transistor level is proposed. Unlike traditional device listing approaches which represent only circuit structures, this representation combines structural data with behavioral information, and thus illustrates a way to reduce the difficulty of information transformation between behavioral and structural representations for CMOS circuits.

Functional recognition of logic components is an important issue in circuit verification. A new method based on functional expansion and logical circuit expressions is proposed, and recognition rules are described. The success of logic component recognition can help other processes such as reverse engineering, which deals with extracting logic-level components from layouts of unknown-function circuits, and the comparison of CMOS transistor schematic networks. Functional recognition enhances the schematic comparison process in that it brings the comparison up to higher levels.

Traditional approaches which use graph matching algorithms for CMOS schematic comparison have difficulty in matching circuits with the same function but different topologies. Other approaches dealing with schematic

comparison such as switch-level simulation need to exercise all possible input patterns, require a large amount of time, and thus are not practical for medium- or large-sized circuits. The approach in this thesis for CMOS schematic comparison is to represent a CMOS transistor network by a set of logical circuit expressions, so that the comparison process is not as rigid as graph matching approaches and yet is efficient enough to compare two functionally isomorphic circuits. The shift from graph connectivity to logical circuit expressions allows schematics comparison for matching functionally isomorphic structures, while most graph-based approaches can handle only topologically isomorphic circuits.

Automated CMOS design and verification using predicates is also described in this thesis. A context-free grammar and a pushdown automata are proposed so that the synthesis and verification processes for series-parallel networks can be done in linear time. ITP, an interactive theorem prover developed at Argonne National Laboratory, is used to demonstrate the capability of the approach.

To my parents and my wife

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Lionel M. Ni, for his patient advice, guidance, inspiration, and continuing support which made this thesis possible. I am in debt to Professor Anthony S. Wojcik for providing me many ideas and valuable discussions of the ITP reasoning system. I am also grateful to other members of my Ph.D. committee, Professors George C. Stockman, Michael A. Shanblatt, and Shui-Nee Chow, for their valuable suggestions, encouragement, and comments.

My thanks are due to many other people in the department for indirectly contributing to this thesis through friendship and support throughout my stay in Michigan. Special thanks are due to my colleagues and friends, who have contributed in various ways to my graduate education here at Michigan State University. In particular, I would like to express my gratitude to Dr. T. B. Gendreau, now at Vanderbilt University, C. T. King, B. McMillin, T. Znati, J. Liao, and S. W. Chen, for their personal support, friendship, and concern.

Finally, I would like to thank my wife, Yew-Huey, for her support and everlasting love over the years.

The work described here was supported in part by the National Science Foundation under grants ECS-83-04967, DCR-86-96044, and in part by the State of Michigan REED Project.

TABLE OF CONTENTS

| | |
|--|-------------|
| List of Tables | viii |
| List of Figures | ix |
| Chapter 1 Introduction | 1 |
| 1.1. Y-Chart for VLSI Design | 2 |
| 1.2. Design Starting Point | 5 |
| 1.3. Motivation and Thesis Organization | 9 |
| Chapter 2 CMOS Logic Structures and Design Styles | 11 |
| 2.1. Fully Complementary CMOS Logic | 14 |
| 2.2. Clocked CMOS Logic | 18 |
| 2.3. Pseudo-nMOS Logic | 18 |
| 2.4. Dynamic CMOS Logic | 20 |
| 2.5. Domino CMOS Logic | 22 |
| 2.6. Cascade Voltage Switch | 24 |
| 2.7. Pass Transistor Logic | 26 |
| 2.8. Differential Pass Transistor Logic | 27 |
| 2.9. Zipper CMOS | 31 |
| 2.10. Design Styles | 37 |
| Chapter 3 Logical Circuit Expressions | 42 |
| 3.1. Levels of Abstraction | 43 |
| 3.2. Series-Parallel Network | 46 |
| 3.3. Logical Circuit Representation | 49 |
| 3.4. Generating Logical Circuit Expressions | 58 |
| 3.4.1. Separating Nodes | 59 |

| | |
|--|------------|
| 3.4.2. Nonseparable Components | 66 |
| Chapter 4 Functional Recognition of Static CMOS Circuits | 71 |
| 4.1. Previous Work | 73 |
| 4.2. Functional Expansion and Recognition Rules | 76 |
| 4.3. Static Logic Component recognition | 83 |
| 4.4. Recognition of Storage Cells | 86 |
| Chapter 5 Comparison of CMOS Transistor Schematic Networks | 91 |
| 5.1. Functional Isomorphism | 93 |
| 5.2. Comparison of Boolean Expressions | 99 |
| 5.3. Comparison Beyond Logic Structures | 101 |
| 5.4. Comparison Hierarchy and Binding | 102 |
| Chapter 6 CMOS Design and Verification Using Logical Predicates | 106 |
| 6.1. Using Logical Predicates | 108 |
| 6.2. Formal Representation of CMOS Circuits | 113 |
| 6.3. Deterministic Pushdown Automata | 116 |
| 6.4. Automated Circuit Verification | 119 |
| 6.5. Automated Circuit Design | 127 |
| Chapter 7 Conclusions | 134 |
| 7.1. Summary | 134 |
| 7.2. Future Work | 137 |
| Appendix | 139 |
| Bibliography | 146 |

LIST OF TABLES

| | |
|---|-----|
| Tabel 6.1. A set of five logic predicates for CMOS circuit representation | 109 |
| Table 6.2. State transition table for recognizing n-type networks | 118 |
| Table 6.3. Verification rules for FCMOS circuits | 121 |
| Table 6.4. ITP predicate statements of the circuit in Figure 6.6 | 124 |
| Table 6.5. Demodulator list for the circuit in Figure 6.6 | 125 |
| Table 6.6. Metrics for verification of the one-bit full adder | 126 |

LIST OF FIGURES

| | |
|--|----|
| Figure 1.1. Y-chart representation of various design levels | 3 |
| Figure 1.2. Intuitive restricted gate-level design for function f | 7 |
| Figure 1.3. Intuitive gate-level design for function f | 7 |
| Figure 1.4. Minimized gate-level design for function f | 8 |
| Figure 1.5. Compound gate design for function f | 8 |
| Figure 2.1. A typical P-well CMOS inverter: (a) logic diagram, (b) transistor schematics, and (c) a simplified cross-section view | 13 |
| Figure 2.2. Fully complementary CMOS logic: (a) block diagram, and (b) a CMOS gate implementing $z = \overline{ab} + \overline{bc} + \overline{ca}$ | 15 |
| Figure 2.3. (a) Non-complementary CMOS structure for $z = \overline{c(a+b)} + \overline{ab}$, and (b) multistage CMOS gates for $z = \overline{f} + \overline{ab}$, where $f = \overline{c} + \overline{ab}$ | 17 |
| Figure 2.4. A CMOS gate using clocked CMOS logic | 19 |
| Figure 2.5. A CMOS gate using pseudo-nMOS logic | 20 |
| Figure 2.6. A dynamic CMOS logic gate with clock signal ϕ | 21 |
| Figure 2.7. Domino CMOS logic: (a) basic version, and (b) latching version | 23 |
| Figure 2.8. Two-input DCVS XOR/XNOR circuits: (a) static and (b) clocked version | 25 |
| Figure 2.9. Pass transistor logic: (a) a two-input CMOS XOR gate, and (b) a two-input CMOS multiplexer | 27 |
| Figure 2.10. Conventional CMOS pass-transistor 4-to-1 selector circuit diagram | 29 |
| Figure 2.11. Differential pass-transistor 4-to-1 selector circuit diagram | 30 |
| Figure 2.12. DCVS buffer used in differential pass transistor logic | 31 |
| Figure 2.13. Charge sharing problem in dynamic CMOS and Domino CMOS logic | 33 |
| Figure 2.14. Zipper CMOS circuit structure | 35 |
| Figure 2.15. Two stages in a (a) ZCMOS (b) FCMOS ripple carry chain | 36 |

| | |
|--|-----|
| Figure 2.16. Semi-custom design styles: (a) gate array, (b) standard cell, and (c) macrocell | 38 |
| Figure 3.1. A circuit with a Wheatstone bridge | 48 |
| Figure 3.2. Circuits with incomplete logic property: (a) a clocked CMOS inverter, and (b) a Domino CMOS gate | 51 |
| Figure 3.3. A two-stage CMOS circuit: (a) transistor schematics, (b) logic diagram, and (c) logical circuit expressions | 54 |
| Figure 3.4. A CMOS decision-making circuit | 56 |
| Figure 3.5. The virtual partitioning diagram of the circuit in Figure 3.3 | 60 |
| Figure 3.6. The virtual partitioning diagram of the circuit in Figure 3.4 | 60 |
| Figure 3.7. A two-input asynchronous arbiter design | 64 |
| Figure 3.8. Separating nodes for nonseparable components | 65 |
| Figure 3.9. Possible structures for a 2-terminal nonseparable component with less than 5 elements | 67 |
| Figure 3.10. Delta-to-wye (Δ -to-Y) transformation | 69 |
| Figure 3.11. Wye-to-delta (Y-to- Δ) transformation | 69 |
| Figure 3.12. Equivalent network by means of transformations | 70 |
| Figure 4.1. A pseudo-nMOS XNOR circuit | 84 |
| Figure 4.2. A mutual exclusion element | 84 |
| Figure 4.3. Two 2-input DCVS XOR/XNOR gates with the same logical circuit expressions | 87 |
| Figure 4.4. A pseudo-nMOS Muller-C element | 88 |
| Figure 4.5. Transistor schematics of a static CMOS D flip-flop | 89 |
| Figure 5.1. Pseudo-nMOS gates through subcircuit permutation | 95 |
| Figure 5.2. Static CMOS NAND gates through subcircuit repetition | 96 |
| Figure 5.3. Pseudo-nMOS XOR gates through functional transformation | 97 |
| Figure 5.4. A two-stage static CMOS XNOR gate | 103 |
| Figure 5.5. A two-stage pseudo-nMOS XNOR gate | 103 |
| Figure 6.1. An FCMOS gate implementing $z = \overline{ab + bc + ca}$ | 110 |
| Figure 6.2. A pseudo-nMOS gate realizing $z = \overline{ab + cd}$ | 112 |
| Figure 6.3. A Domino CMOS circuit with a clock signal ϕ | 112 |
| Figure 6.4. The parse tree of node z in Figure 6.1 | 116 |
| Figure 6.5. State transition diagram for recognizing n-type transistor networks | |

| | |
|---|------------|
| Figure 6.6. Transistor schematics of a one-bit full adder | 123 |
| Figure 6.7. Alternative D flip-flop design using basic synthesis procedure | 132 |

CHAPTER I

INTRODUCTION

Recent advances in integrated circuit fabrication technology have increased the complexity of integrated circuits to such an extent that it is possible to fabricate a VLSI chip containing hundreds of thousands of transistors [Youn86]. The growth of IC complexity has had a dramatic impact on the time needed to design a circuit. The design methods which were adequate in the MSI and LSI periods require extensive improvement. In the early days of IC design, almost unlimited freedom was used in order to achieve the most compact results. This approach is no longer possible because the time complexity for solving the design and layout problems grows exponentially as the number of gates increases.

To tackle the complexity, automated design methods have become an indispensable tool for designing chips in a reasonable time frame. Many efforts have been devoted to various aspects of automated design and verification to reduce the design time and cost. Given that the process of designing a system on silicon is complicated, the role of VLSI design aids is to reduce this complexity and raise the level of confidence for the correctness of the design. Recently, VLSI design tools have been used in many ways to help reduce high design costs. VLSI design tools not only assist an engineer in circuit design, simulation, testing, and layout, but also help alleviate human errors caused by the manual design process. The average turn-around time has decreased significantly in the past ten years due to the use of automated tools.

On the other hand, in order to reduce the VLSI design burden and make the whole design process manageable, hierarchical design methods [Nies83] are usually considered

as a means of dealing with the VLSI design problem. The hierarchy, when properly implemented, is the self-evident implementation of the “divide and conquer” principle. The use of hierarchy involves dividing a chip into modules and repeating this operation on the modules until the complexity of a submodule is at an appropriate and comprehensible level of detail [Sequ83]. For instance, a chip may be decomposed into several modules with the I/O pads residing around the boundaries, and modules can be further decomposed into a number of cells.

In addition to hierarchical design methodology, circuit representation also plays an important role. Once a high-level specification is available, the design process can be viewed as a translation process from a higher-level description into a lower one.

1.1. Y-Chart for VLSI Design

A tripartite representation for moving a design from a high-level specification to the low-level mask data has been proposed by Gajski and Kuhn [GaKu83]. The representation is partitioned into three sub-domains, namely, the *functional* (or *behavioral*) *representation*, the *structural representation*, and the *geometrical representation*. Multiple levels of detail are represented along each of the three axes. The levels of abstraction are increasing as one moves away from the vertex.

In the functional (or behavioral) domain one is interested in what the chip does and not how it is built. The design is treated as a black box with a specified set of inputs, a set of outputs and a set of functions describing the behavior of each output as a function of inputs and time. For example, the Boolean expression $z = \overline{ab} + cd$ indicates only the function of the design whose inputs are a , b , c , and d , and whose output is z . It does not say anything about the implementation or the structure of the cell. The functional representation of a design may be captured at several levels, such as systems,

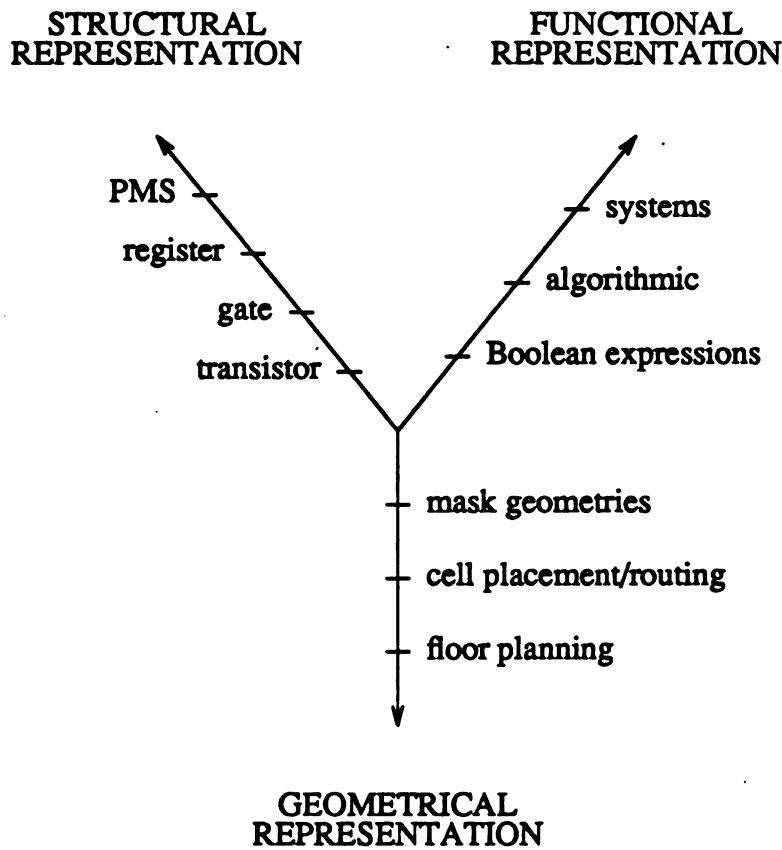


Figure 1.1. Y-chart representation of various design levels

algorithmic, and Boolean expressions. The system or architectural description defines gross operational characteristics with performance specifications without being concerned about how data is manipulated or what algorithm is used. At the algorithmic level, the system may be represented using state transition diagrams. Variables or data structures are not bound to registers or memories at this level, and operations are not bound to any functional units or control state. For the logic level, Boolean expressions are chosen to represent the circuit function.

The structural representation is the bridge between the functional representation and

the geometrical representation. It is a mapping of a functional representation onto a set of components and connections under constraints such as cost, area, and time. This representation does not specify any physical parameters, such as the locations of circuit components and their sizes. In fact the structural representation specifies the connectivity among various components. The most commonly used levels of structural representation can be identified with the basic structural elements used. At the transistor level the basic elements are transistors, while gates and flip-flops are at the logic level. ALUs, registers, RAMs and ROMs can be used to represent structures at the register level, in which the communication among the components is of more concern than the implementation of each individual component. Processors, memories, and switches are used at the system level.

At the lowest level in this hierarchy, the geometrical representation deals with physical layout design. The geometrical representation ignores, as much as possible, what the design is supposed to do and binds its structure in space or to silicon. Components in the structural representation are built, placed, and interconnected using provided primitives. The structure-to-geometry mapping can be defined as a two step process. The first step, usually called symbolic or topological layout, determines relative or approximate positions for all structural elements. The absolute positions are determined in the second step after substitution of layouts for symbols and compaction. The most commonly used levels in geometric representation are mask geometries, cell placement/routing and floor planning with arbitrary size blocks. Cell placement/routing deals with the location of each individual cell and connections among those cells within a block, while floor planning is concerned with the placement of rectangular modules of varying sizes and shapes such that the required electrical performance can be achieved and the total area occupied by the modules and the interconnections is minimized.

1.2. Design Starting Point

Based on the Y-chart representation, hierarchical design is an efficient approach to reduce the high design cost of a VLSI circuit. The design procedure for developing a chip can be entered at various starting points. In other words, it is the designer's responsibility to decide the entry point of the design process, which is an important factor in the tradeoff between the performance and design cost of the chip [VaSh85]. Since complementary metal oxide semiconductor (CMOS) technology has played an increasingly important role in the integrated circuit industry over the past several years, CMOS technology is selected in the following example to illustrate different starting points for designing a logic circuit.

Given a function f with a control signal C , assume $f = \overline{ab}$ if $C = 0$ and $f = \overline{x + ab}$ if $C = 1$. There are several different ways to implement the circuit using the prevalent CMOS technology.

Intuitively one can use a two-input multiplexer and simple logic gates to implement the function f . However, in order to simplify the automated synthesis process, one might have only a limited number of available components, such as two-input NAND gates or four-input multiplexers. Thus the basic problem of circuit design becomes finding a way to connect those primitives so that the overall circuit behaves in the desired way [WoOL84]. Figure 1.2 shows a logic circuit with eight two-input NANDs, in which the two-input multiplexer is formed by four two-input NAND gates and the longest path contains five stages. A total of thirty-two transistors are used. Note that the logic circuit is not minimized.

By increasing the number of available components, one may be able to improve the design. If two-input multiplexers and simple logic gates are available in the designer's cell library, a simplified version of the same design can be obtained. Figure 1.3 shows

the circuit with one two-input multiplexer using pass transistor logic, one NAND, one NOR and one inverter. A total of sixteen transistors are used, and the number of stages along the longest path is three. If the chip area is proportional to the device count and the delay time of a circuit is proportional to the number of gates along the longest path [Hwan79], then the chip performance AT^2 is improved by a factor of 5.5.

Logic minimization is another way to improve the circuit. Applying simple logic simplification rules, one notices that the function f is actually defined by $\overline{ab + Cx}$. Thus three NAND gates and one inverter are sufficient to build the circuit. Figure 1.4 shows the logic diagram of the simplified circuit. Fourteen transistors are used, and the number of stages along the longest path is only three. Thus, the chip performance is improved by another factor of 1.14.

If the transistor level is selected instead of the gate level as the entry point of the design, significant improvement can be achieved. Expertise at the transistor level is useful in realizing a better circuit structure. A compound gate whose output is f using true CMOS technology is shown in Figure 1.5. Only eight transistors are used. Since compound gates are basically one-level circuits, they also have significant improvement in terms of delay times. The delay time of such a circuit, estimated from SPICE [Oklo82], is about 1.5 times that of a standard 2-input CMOS NAND gate. Thus, the chip performance AT^2 is further improved by a factor of 7.

In the best case, simple logic gates available in a VLSI standard cell library are locally optimized. Even with logic minimization at the gate level, a large number of transistors in a circuit are usually wasted because of the lack of a global viewpoint.

The key to moving beyond this local minimum is the expertise of lower level design, namely, transistor level expertise. The use of transistor level expertise significantly improves the chip performance. Knowledge at the transistor level is usually

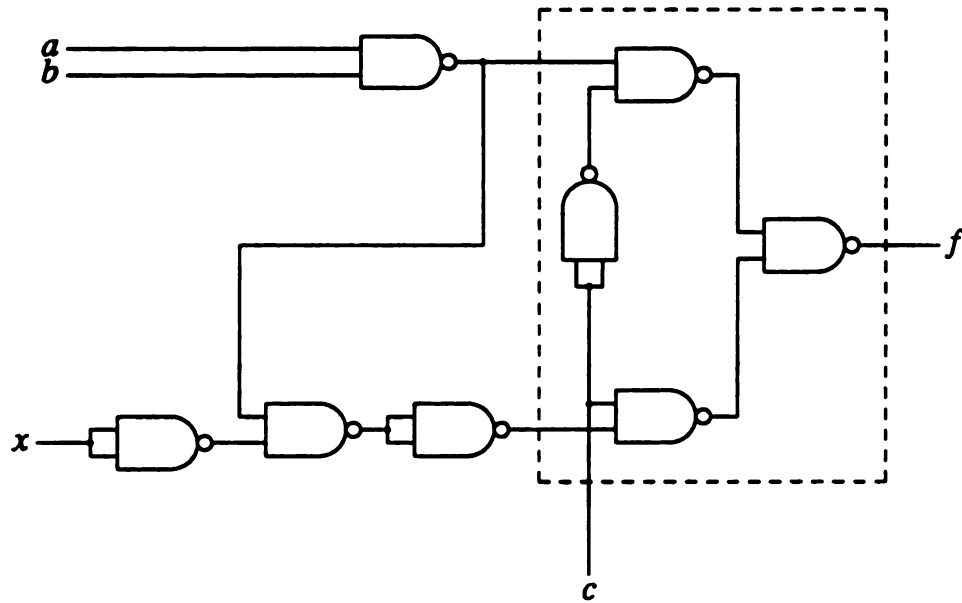


Figure 1.2. Intuitive restricted gate-level design for function f

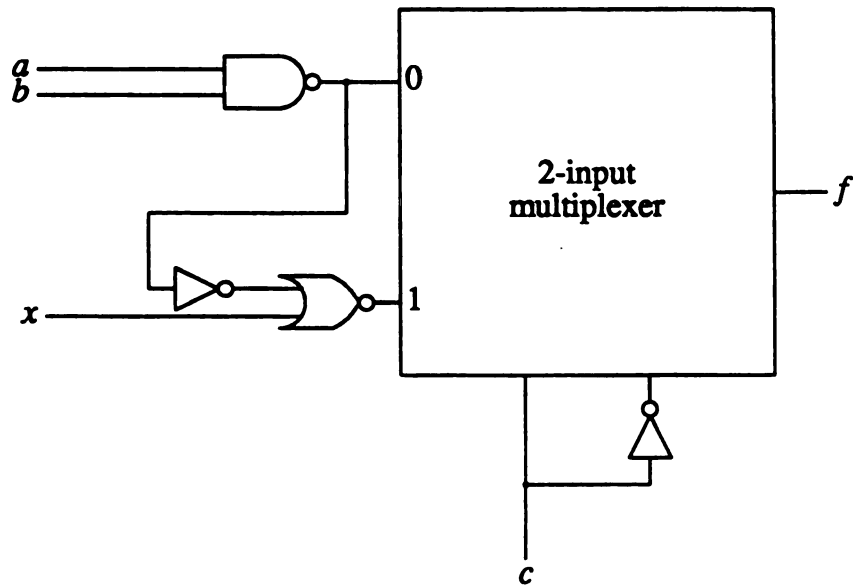


Figure 1.3. Intuitive gate-level design for function f

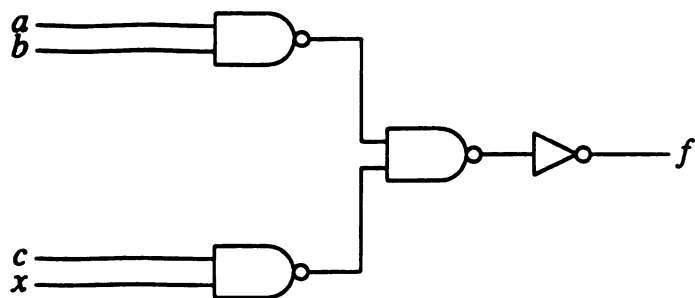


Figure 1.4. Minimized gate-level design for function f

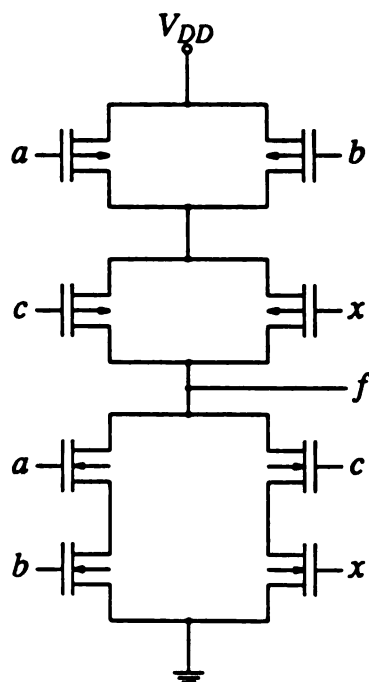


Figure 1.5. Compound gate design for function f

used mainly inside the designers' cell library. This is true especially when standard cell design is used. In order to speed up the design process in the conventional design environment, designers usually use those cells they need from the cell library and interconnect them to form new custom circuits in spite of the sacrifice in the chip performance. The tools a designer might have are merely a collection of programs that perform logic minimization at the gate level, placement of selected components on the floor plan, and routing of the interconnections.

1.3. Motivation And Thesis Organization

Since the designs at the transistor level usually result in smaller area and better performance, CMOS logic circuits are considered at the transistor level as the target area. The goal of this thesis is to find a CMOS circuit representation which combines both functional and structural information, so that the difficulty associated with information transformation between these two design domains can be reduced or released.

Once this representation is available, a practical problem which deals with CMOS schematic comparison becomes evident. Traditional approaches use either direct graph-matching algorithms or indirect switch-level simulation. As graph-matching algorithms suffer from rigid graph isomorphism checking, switch-level simulation suffers from tremendous computation overhead due to exhaustive exercises of input patterns and usually does not provide any information for error location. From the representation, an alternative is proposed and a better solution for CMOS schematic comparison may be obtained. One would also like to be able to recognize the logic functions of CMOS logic elements, so that the comparison can be performed at the Boolean level, and thus logic circuits with the same function but different topology may still be matched. In order to provide solutions for these problems, one needs a circuit partitioning scheme, and that is

where the idea for the logical circuit expressions arises.

In this thesis, a circuit representation for CMOS transistor networks is proposed and its applications to CMOS VLSI design automation is examined. After this introductory chapter, various CMOS logic structures and circuit design methodologies such as standard cell, gate array, and full-custom designs are described in Chapter 2. The approach to CMOS circuit representation, the *Logical Circuit Expressions*, is described in Chapter 3, along with approaches for generating logical circuit expressions. In Chapter 4, a novel approach to functional recognition for CMOS logic circuits based on logical circuit expressions and functional expansion is proposed. In addition to the contribution of recognizing logic functions of CMOS circuits, this approach also enhances the schematic comparison process in that it brings the comparison process up to the gate, or even, block level, provided that function recognition for individual or consecutive stages is successful. Chapter 5 shows how logical circuit expressions can be used for CMOS schematic comparison. Although it is different from the conventional direct and indirect approaches, it is shown that this approach gives more freedom when compared with the rigid graph-matching algorithms, and provides much more information when compared to switch-level simulation approaches. In Chapter 6 a rule-based system for automating CMOS design and verification using predicates is described. A context-free grammar and a pushdown automata are proposed so that the synthesis and verification processes for series-parallel networks can be done in linear time. ITP (Interactive Theorem Prover), which was developed at Argonne National Laboratory and written in Pascal, is used to demonstrate the capability of the approach. Finally, Chapter 7 concludes the thesis with a summary and directions for future research.

CHAPTER II

CMOS LOGIC STRUCTURES AND DESIGN STYLES

Basically this chapter serves as common background for CMOS VLSI designs. The general characteristics of CMOS technology are described, various CMOS logic structures are examined, and several VLSI design styles are discussed. Both advantages and disadvantages of various CMOS logic structures and design styles are illustrated.

The technology of semiconductor devices is advancing at a rapid rate. Bipolar devices have been widely used for a long time. In general, bipolar transistors are characterized by lower ON resistances and higher current capabilities for a given size of device than are MOS devices. The most widely used semiconductor technology was *TTL* (Transistor-Transistor Logic). *ECL* (Emitter-Coupled Logic) and I^2L (Integrated Injection Logic) are also popular in some applications. To date, for most high-speed and low-noise applications, such as high-speed mainframe processors, bipolar technology is still used. However, the heat dissipation in bipolar circuits is rather large and requires an elaborate cooling arrangement for its operation.

The fastest growing technology has been MOS since the last decade. MOS transistors are known as FETs (Field-Effect Transistors) and are also referred to as MOSFETs. The MOS technology is growing dramatically fast because of high-volume commercial applications. The minimum feature size of MOS technology is decreasing rapidly, resulting in high-density complex custom chips with proven reliability of the two MOS technologies, nMOS and pMOS. CMOS is comprised of both n-type and p-type MOSFETs. To fully appreciate CMOS, it is necessary to understand the properties of MOS in general.

Detailed descriptions of MOS technology can be found in [MeCo80, HoJa83, GlDo85].

The nMOS technology has had some advantages in terms of size or area of silicon needed to produce equivalent functionality, but with decreasing feature size for CMOS, that advantage is rapidly evaporating. The past several years have seen a rapid shift in the technology of choice for high-complexity digital microelectronics from nMOS to CMOS. This shift has occurred because CMOS offers high performance at low power and scales extremely well to small feature size. The greatest advantage of CMOS over nMOS is its static power consumption, which is an order of magnitude smaller than nMOS static power requirements. CMOS technology has been recognized as a leading contender for existing VLSI systems, and is projected by industry analysts as being the dominant technology for the next decade [Mukh86]. It provides an inherently low power static circuit technology that has the capability of providing a lower power-delay product than comparable design-rule nMOS or pMOS technologies.

CMOS technology provides two types of transistors, an n-type transistor and a p-type transistor. These transistors are fabricated in silicon by using negatively doped silicon that is rich in electrons and positively doped silicon that is rich in holes, respectively. For the n-transistor, the structure consists of a section of p-type silicon separating two diffused areas of n-type silicon. The area separating the n regions is capped with a sandwich consisting of an insulator and a conducting electrode called a *gate*. Similarly, for the p-transistor, the structure consists of a section of n-type silicon separating two p-type diffused areas. Each transistor has two additional connections which are designated the *drain* and the *source*. In fact, the drain and source may be viewed as two switched terminals. The terminals are physically equivalent, and the name assignment depends on the direction of current flow [WeEs85]. Figure 2.1 shows a CMOS inverter with its logic diagram, transistor schematics, and a simplified cross-section view.

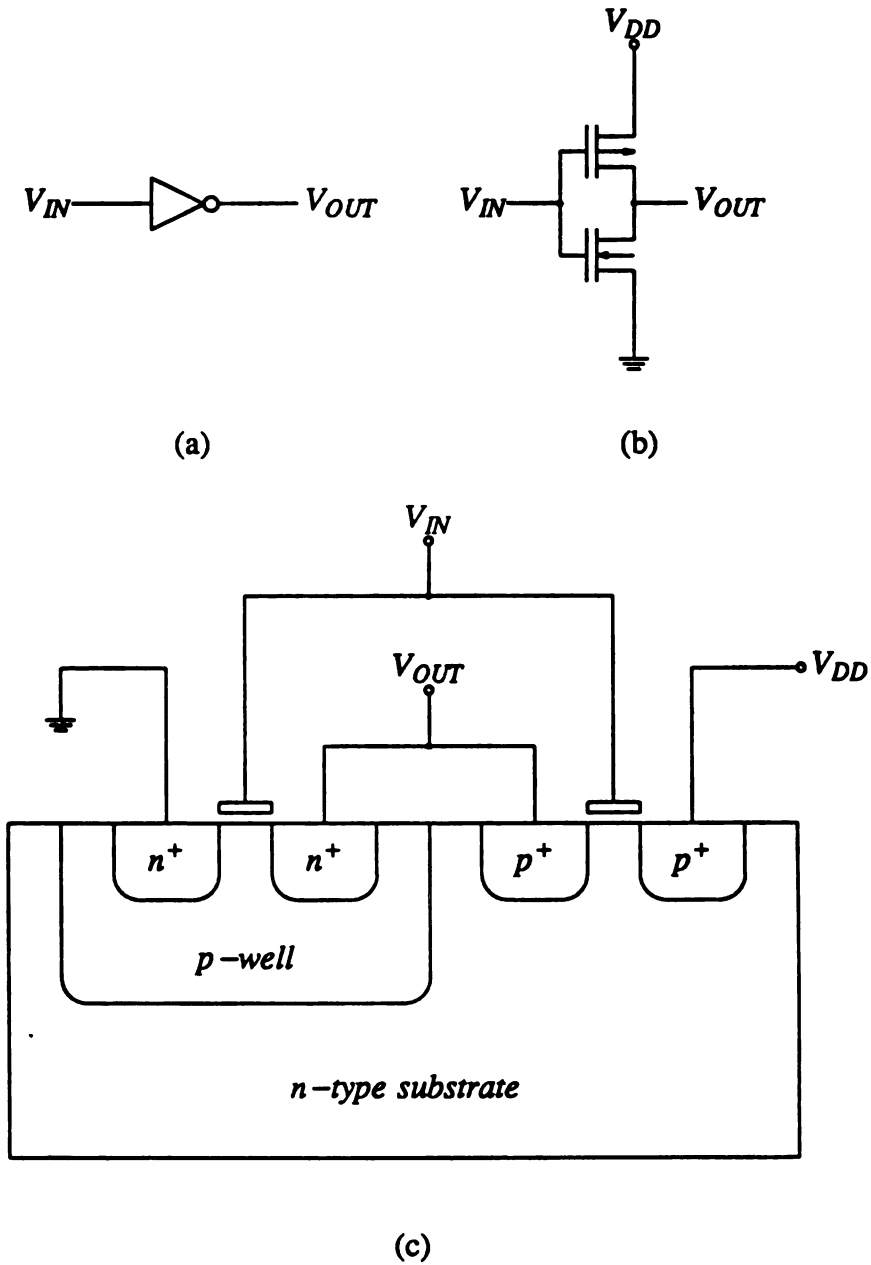


Figure 2.1. A typical P-well CMOS inverter: (a) logic diagram, (b) transistor schematics, and (c) a simplified cross-section view.

The use of both polarity devices on the same substrate creates various types of CMOS circuits. Many designers prefer to use static logic whenever possible to simplify their designs. On the other hand, dynamic CMOS circuits in which dynamic charges play an important role in circuit behavior usually results in significant area savings. In this chapter alternative CMOS logic configurations are examined and various approaches by which CMOS circuits are constructed are illustrated. The CMOS logic structures covered in this chapter are by no means complete; but through this chapter, one should be able to understand how a CMOS circuit works and how it is constructed.

2.1. Fully Complementary CMOS Logic

Fully complementary CMOS (also referred to as FCMOS or true CMOS) logic is the most common logic structure. An FCMOS gate consists of a network of p-type transistors called the *load circuit* and a network of n-type transistors called the *driver circuit*. If an input combination for which the function realized by an FCMOS gate is to be 1 is applied, a path from V_{DD} to the output node of the gate is established through conducting p-type transistors, and all paths from output to V_{SS} through the n-type transistor network are cut off. Similarly, if an input combination for which the output of the gate is 0 is applied, a path from V_{SS} to the output is established through conducting n-type transistors, and all paths from output to V_{DD} are disconnected. Consequently, there is no static current path between V_{DD} and V_{SS} , and fully complementary CMOS ICs dissipate power only to charge and discharge circuit capacitance. Figure 2.2(a) shows the block diagram of a CMOS gate with p-type and n-type transistor networks, and Figure 2.2(b) shows a fully complementary CMOS gate which is used as the carry generator of a full adder.

Basically there are three different physical structures of fully complementary

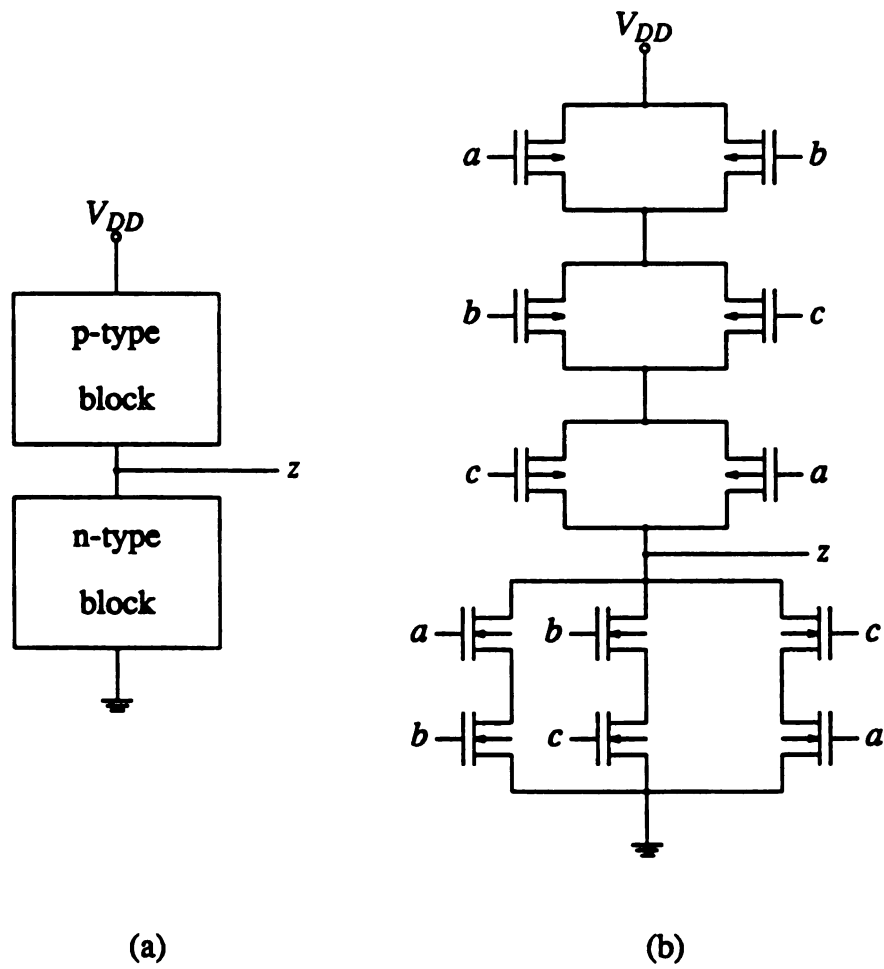
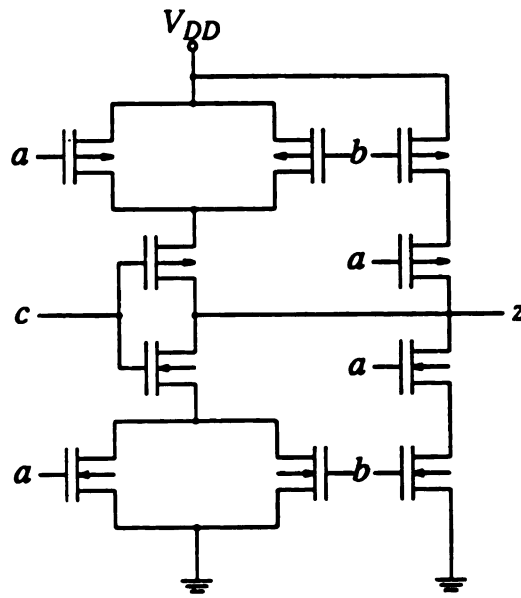


Figure 2.2. Fully complementary CMOS logic: (a) block diagram, and (b) a CMOS gate implementing $z = \overline{ab + bc + ca}$

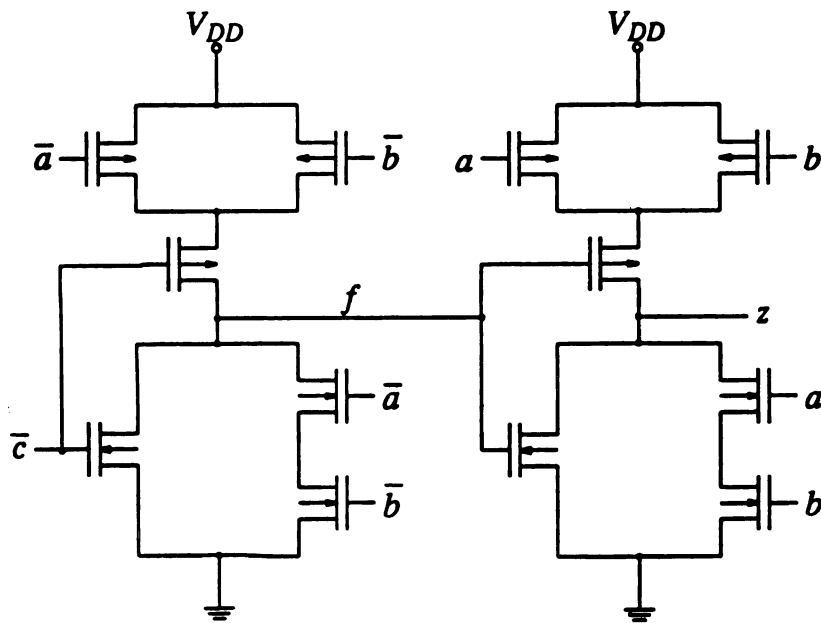
CMOS combinational gates. *Complementary CMOS structure* refers to FCMOS gates whose p-type and n-type transistor networks are complementary in physical structure. Parallel n-type devices in the driver circuit imply serial p-type devices with the same gate signals in the load circuit, and serial n-type devices in the driver circuit imply parallel p-type devices with the same gate signals in the load circuit. The complementary CMOS structure is simple and has been widely used in many designs, especially for simple logic functions. The CMOS gate in Figure 2.2(b) which implements $z = \overline{ab + bc + ca}$ is a typical example.

Non-complementary CMOS structure refers to FCMOS gates whose p-type and n-type networks are not complementary in physical structure. Since both p-type and n-type transistor networks are separated from each other, they can be implemented independently in order to compact the physical layout. Thus, unnecessary long serial structures may be avoided. Figure 2.3(a) shows a fully complementary CMOS realization of the function $z = \overline{ab + bc + ca}$, in which p-type and n-type networks are symmetric instead of complementary in physical structure.

The third approach to implement Boolean functions using FCMOS logic is the use of multistage design. Compound gates are basically one-stage circuits. As the complexity of a logic function increases, the delay time of the gate also increases. Figure 2.3(b) shows two CMOS gates concatenated in series to implement the same function as the one in Figure 2.2(b). The output node z implements the function $\overline{f + ab}$, where $f = \overline{\overline{ab} + \overline{c}} = c(a + b)$. Although it becomes a two-stage circuit with both true and complementary input signals, its first stage can be easily shared by other functions.



(a)



(b)

Figure 2.3. (a) Non-complementary CMOS structure for $z = \overline{c(a+b)} + ab$,
and (b) multistage CMOS gates for $z = \overline{f + ab}$, where $f = \overline{c + ab}$

2.2. Clocked CMOS Logic

Clocked CMOS logic is a variant of the fully complementary CMOS logic. It was originally used to build low power dissipation CMOS circuits [SuOA73]. Since only part of a circuit is active at a time, it is not necessary to enable all logic gates all the time. By introducing a clock signal in the circuit, gates not in use can be cut off to reduce dynamic power consumption. Figure 2.4 shows a clocked CMOS compound gate which implements the function $z = \overline{ab + c(d + e)}$.

Gates using clocked CMOS logic have the same input capacitance as fully complementary CMOS gates but larger rise and fall times due to the series clocking transistors.

2.3. Pseudo-nMOS Logic

A common criticism of CMOS is that equivalent CMOS circuits have more transistors than nMOS circuits. As a point of reference, CMOS designs are commonly quoted as being 20-30 percent larger than the nMOS equivalent designs [WeEs85]. This increase results from “logical redundancy” [MyIv85] – the need in fully complementary CMOS for evaluating the logic function and its complement to maintain good logic levels. Thus, circuits need two devices per logic variable in use, leading to area wasteful interconnections and a consequent degradation in performance.

Pseudo-nMOS is a CMOS variation which uses a p-type transistor to mimic an nMOS pull-up. The load device is a single p-type transistor, with the gate connected to V_{SS} . This structure is equivalent to a conventional nMOS gate except that the depletion nMOS load is replaced by a p-device. The design of this style of gate thus involves ratioed transistor sizes to ensure correct switching.

A typical pseudo-nMOS gate is shown in Figure 2.5, which performs the same func-

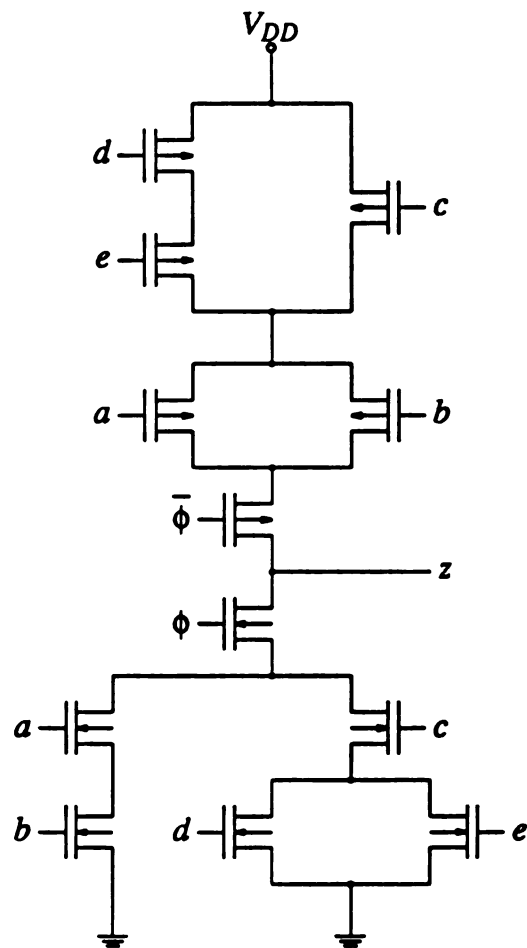


Figure 2.4. A CMOS gate using clocked CMOS logic

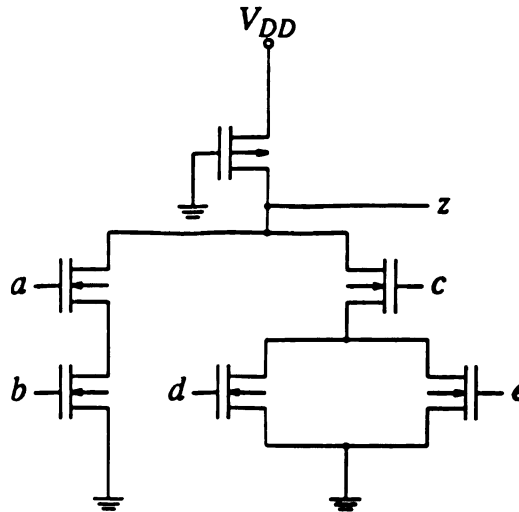


Figure 2.5. A CMOS gate using pseudo-nMOS logic

tion as the one in Figure 2.4, except that it is a static circuit. There is a definitive gain in area, but its low-to-high transition is slower and the static power consumption cancels the main advantage of CMOS technology. One possible advantage of the pMOS load is that it does not suffer from body effect as the nMOS depletion load does. Thus in a CMOS process it gives a method of emulating nMOS circuits.

2.4. Dynamic CMOS Logic

Dynamic CMOS gates require a precharge interval to generate dynamic charges at internal nodes. A basic dynamic CMOS gate is shown in Figure 2.6. It consists of an n-transistor logic structure whose output node is precharged to V_{DD} by a p-transistor in the precharge phase and conditionally discharged by an n-transistor connected to V_{SS} during the evaluate phase. Input signal ϕ is a single phase clock. The precharge phase occurs when $\phi = 0$. The path to V_{SS} supply is closed via the n-transistor “ground switch”

during the evaluate phase when $\phi = 1$. The input capacitance of this gate is the same as the pseudo-nMOS gate shown in Figure 2.5. Its pull-up time is improved by virtue of the active switch, but the pull-down time is increased due to the ground switch.

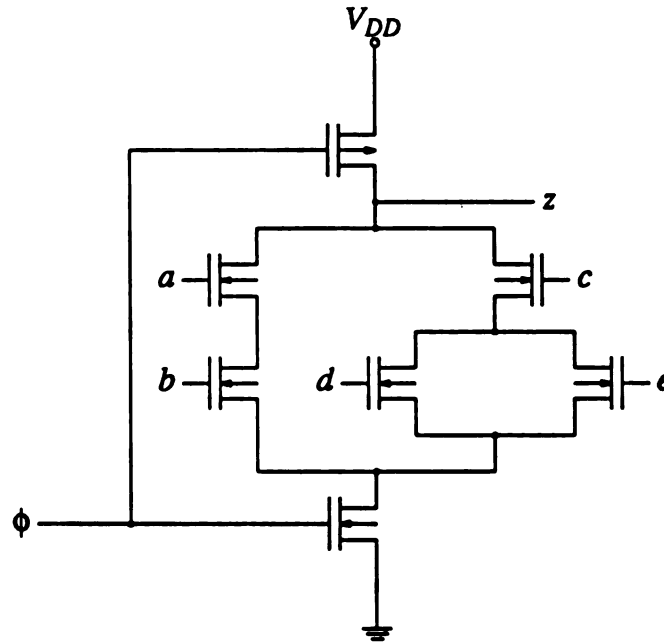


Figure 2.6. A dynamic CMOS logic gate with clock signal ϕ

Precharged logic requires a clocking scheme which implies the need for some extra global control and communication. The inputs of a gate must remain steady once the precharge phase is finished. Simple single-phase dynamic CMOS gates, therefore, cannot be cascaded due to the conditional discharge at the output of the previous gate during the evaluate phase. A possible modification of dynamic CMOS logic is presented in [KrLL82], which leads to Domino CMOS logic.

Friedman and Liu [FrLi84] proposed another dynamic CMOS logic structure based on the direct interconnection of p-type logic and n-type logic dynamic gates. Adjacent

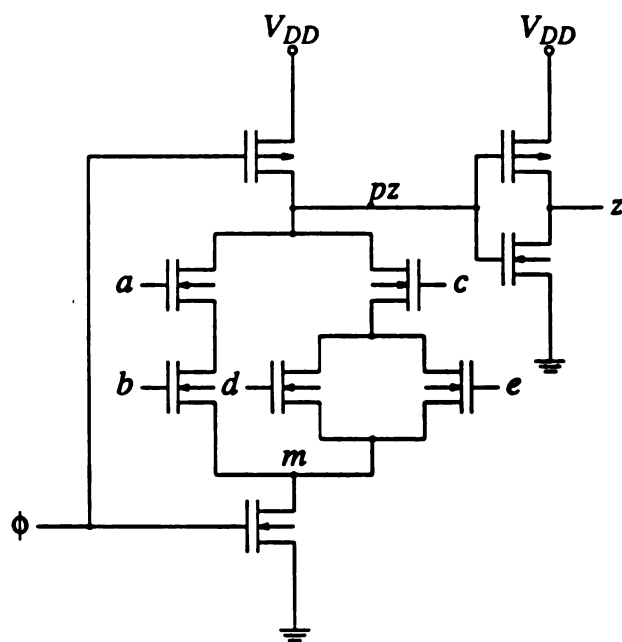
dynamic CMOS gates utilize complementary clock signals and complementary logic networks so that cascaded structures are allowed. Further improvements to dynamic CMOS logic use the forms of two and four phase logic that have been developed for earlier types of MOS design [MyIv85, WeEs85].

2.5. Domino CMOS Logic

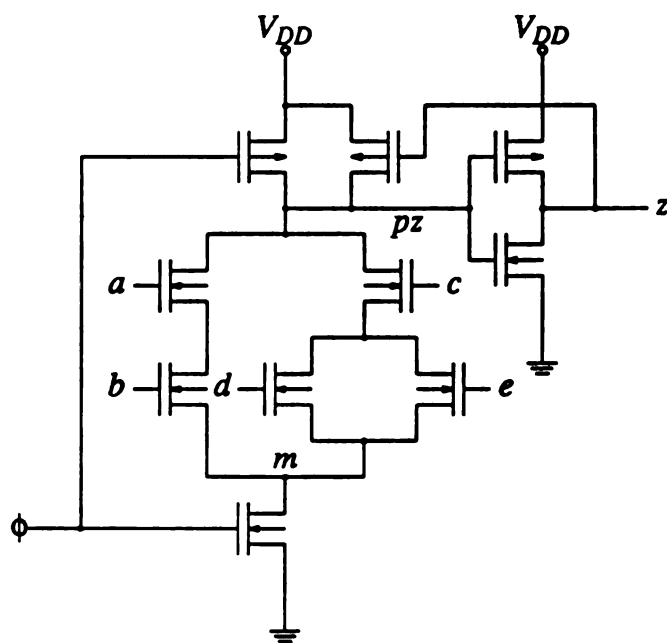
One of the best known approaches to the design of combinational logic in CMOS that avoids logical redundancy without suffering from the side effect of increased power dissipation is Domino CMOS logic. Domino CMOS circuits share some characteristics with dynamic circuits. In particular, each output is precharged high while the path to ground is opened, and the precharge is stopped while the path to ground is activated. Transitions from precharge to evaluation are accomplished by means of a single clock edge applied simultaneously to all gates in the circuit. This simplifies the clocking scheme and permits utilization of the full inherent speed of the gates.

A single Domino CMOS gate is depicted in Figure 2.7(a). It consists of a single phase dynamic gate and a static CMOS buffer. During precharge ($\phi = 0$), the output node of the dynamic gate is precharged high and the output of the buffer is low. As subsequent logic stages are fed from this buffer, transistors in subsequent logic blocks will be turned off during the precharge phase. In addition, during evaluation, a Domino gate can make only a single transition ($0 \rightarrow 1$). Because of the nature of the dynamic gate which drives it, it is impossible for the buffer to go from high to low. As a result there can be no glitches at any nodes in a cascaded set of logic blocks. Each stage evaluates and causes the next stage to evaluate in the same manner that a stack of dominos fall.

The Domino CMOS gate may be made static for low frequency circuits by including a weak p-transistor. A weak p-transistor is one that has a small W/L ratio. The ratio



(a)



(b)

Figure 2.7. Domino CMOS logic (a) basic version, and (b) latching version

is chosen small enough so that there is no significant impact on the pull-down current and so that the power consumed during the evaluation phase is tolerable. If the time between evaluation phases is relatively long, the clocked precharged p-transistor can be eliminated, and precharge can be accomplished by the weak static pull-up transistor. Charge redistribution in Domino CMOS can be reduced by placing a weak p-type feedback transistor or increasing the capacitance of the precharge node [OkMo86]. Figure 2.7(b) shows a circuit with a weak p-type feedback transistor which leads to the latching version of Domino CMOS gates, and increasing the capacitance of a precharge node can be accomplished by making the size of the transistors in the output inverter larger.

2.6. Cascade Voltage Switch

Cascade voltage switch (CVS) is a circuit implementation technique for complex single or dual output switching functions formed by cascading n-type MOS transistors. The CVS circuits yielding a dual output are called *differential CVS* (DCVS) *circuits* [HeGD84], whereas the other CVS circuits are called *single-ended CVS* (SCVS) *circuits* [BrCM84]. The differential style of logic usually requires both true and complement signals to be routed to gates. Both static and dynamic circuits can be generated. DCVS circuits are obtained by realizing the complement function with another set of n-type transistors. The two n-type networks are connected to the positive supply voltage node by a pair of cross-coupled p-type transistors.

Figure 2.8(a) shows a two-input static DCVS XOR circuit with complementary n-type networks. The static version is usually slower than a conventional complementary gate employing a p-tree and n-tree, since during the switching action, the pull-ups have to “fight” the n pull-down trees. Note that this is not a very efficient implementation of this gate.

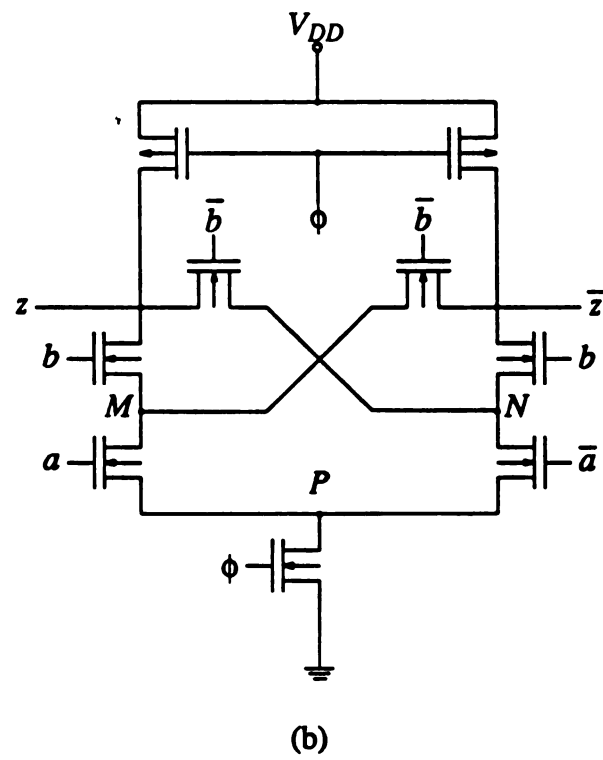
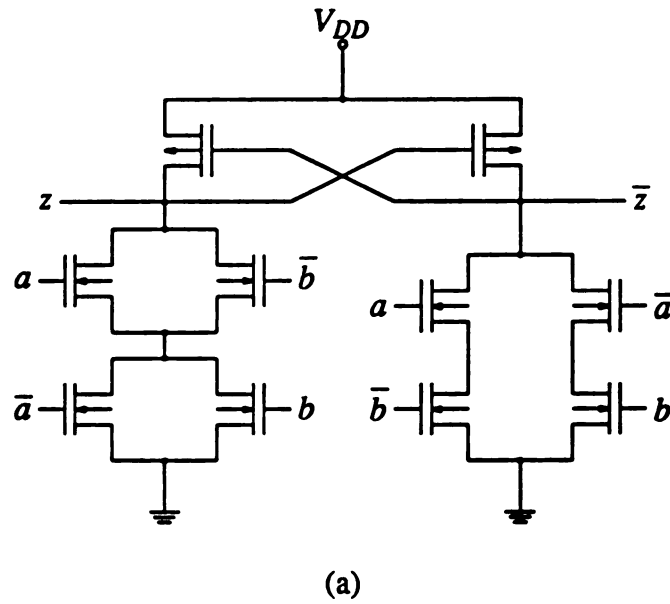


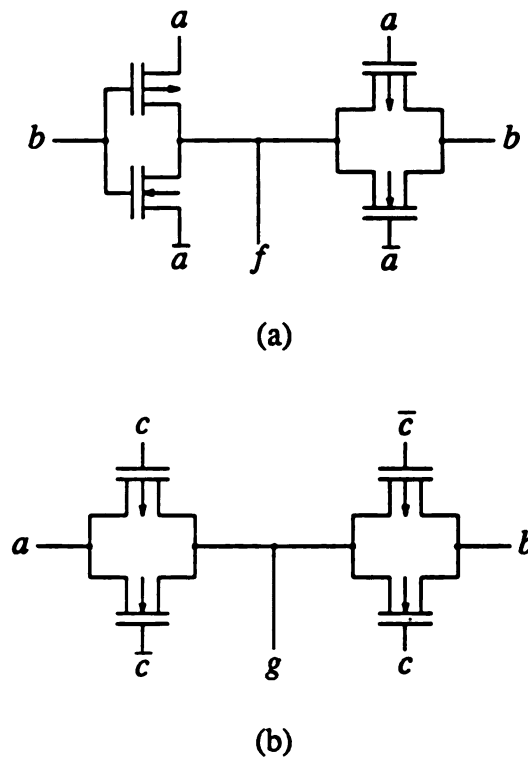
Figure 2.8. Two-input DCVS XOR/XNOR circuits (a) static and (b) clocked version

Further refinement leads to a clocked version of the DCVS gate, as shown in Figure 2.8(b). A dynamic DCVS circuit is formed simply by adding one discharge n-type transistor and substituting the two cross-coupled p-type transistors with two precharge p-type transistors. The logic tree is minimized from the full differential form using logic minimization. Logic minimization results in a n-type network with only six transistors rather than eight. Outputs from dynamic DCVS circuits are usually buffered using static CMOS inverters, which are not shown in Figure 2.8(b). This DCVS structure can be easily expanded for multiple-input parity generator and other complex switching circuits. A four-way DCVS XOR gate [HeGD84], for instance, needs only fourteen transistors in its n-type network. Significant area saving is therefore achieved.

2.7. Pass Transistor Logic

One form of logic that is popular in nMOS circuits is pass transistor logic. Pass transistor realizations using fewer transistors usually result in area savings and higher operating speed when compared with the corresponding gate logic realizations. CMOS transmission gates used in many CMOS designs such as XOR gates and multiplexers are simple examples. A CMOS XOR gate and a two-input CMOS multiplexer are depicted in Figure 2.9. In total, four transistors are used for both the two-input CMOS XOR gate and the two-input multiplexer. If they are implemented using gate-level design, four two-input CMOS NAND gates are required for a two-input XOR gate, and three CMOS NAND gates are required for a two-input multiplexer.

CMOS pass-transistor gates are composed of nMOS and pMOS transistors. Since the transistor gate controls the passage of current between the drain and source, a simplified scheme [MeCo80, Brya81] which allows the MOS transistors to be viewed as simple on/off switches is proposed. In an n-transistor, for instance, the switch is closed,



**Figure 2.9. Pass transistor logic (a) A two-input CMOS XOR gate;
and (b) a two-input CMOS multiplexer**

or “ON”, when there is a “1” on the gate. Formal methods for deriving pass-transistor logic have been presented using this model [Whit83]. Basic design procedures for pass transistor logic using modified Karnaugh map and modified Quine-McCluskey tabular approach can be found in [RaWM85].

2.8. Differential Pass Transistor Logic

In traditional CMOS pass transistor logic, a p-type transistor is used to pass logic one efficiently, while the n-type transistor is used to pass logic zero efficiently. Besides

using both p- and n-type transistors, substantial p-to-n plane interconnects and the required well-to-device channel spacing reduce area efficiency. Furthermore, the number of drain-source connections at the network output is doubled, hence doubling the output capacitance and making conventional CMOS pass transistor networks inherently slow. Pasternak et. al. [PaSS87] proposed a new logic design structure called *differential pass transistor logic* by encoding the pass variables differentially, passing this signal through a differential pass network, and then decoding the output to normal logic levels. The differential signal can be restored to normal logic levels by using a static differential buffer, which is typically a cascade voltage switch logic inverter. Thus, both true and complementary values of the function can be obtained due to the differential buffer.

The differential pass logic element consists of two n-type transistors, controlled by the same gate signal, to pass the input and its complement to the output. Since n-type transistors pass logic zero efficiently, one of these values will be zero. The other output voltage level will be the maximum output voltage of an n-type transistor $V_{DD} - V_{TH}$ where V_{TH} is the ON threshold voltage of the transistor. A conventional CMOS pass-transistor four-to-one selector circuit diagram and a differential pass-transistor four-to-one selector network are depicted in Figure 2.10 and Figure 2.11, respectively.

Note that in Figure 2.11 all the positive row outputs are connected together to form the signal $OUT+$, and all the negative row outputs are connected together to form the signal $OUT-$. The differential signal pair, $OUT+$ and $OUT-$, is then restored to normal logic levels by using a static differential buffer, which is simply a cascade voltage switch logic (DCVS) inverter and is depicted in Figure 2.12. In addition to replacing the p-type transistors of conventional CMOS pass transistor logic with smaller n-type transistors, differential pass transistor logic eliminates the p-to-n plane interconnect, and effectively halves the output capacitance. This results in increased area efficiency and operating speed.

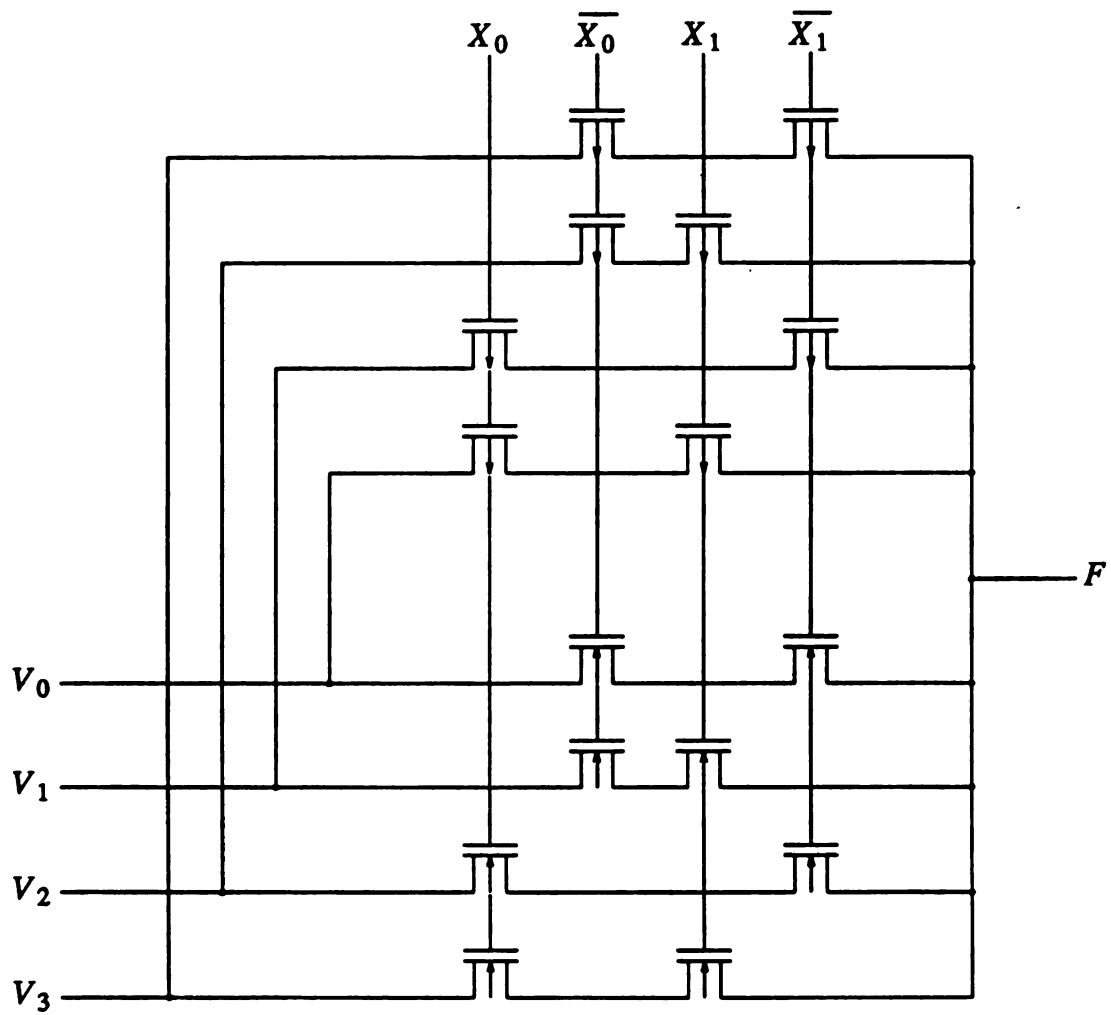


Figure 2.10. Conventional CMOS pass-transistor 4-to-1 selector circuit diagram

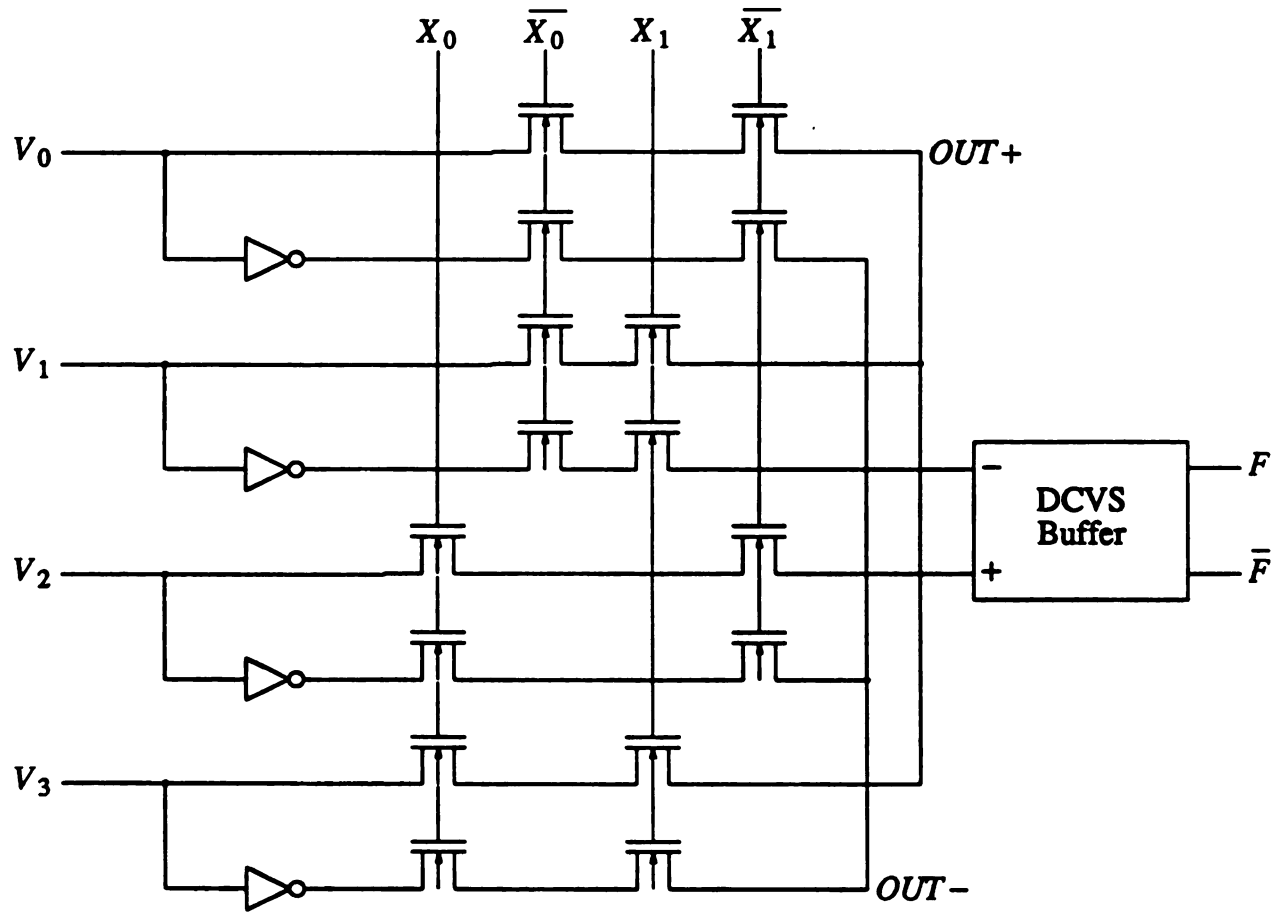


Figure 2.11. Differential pass-transistor 4-to-1 selector circuit diagram

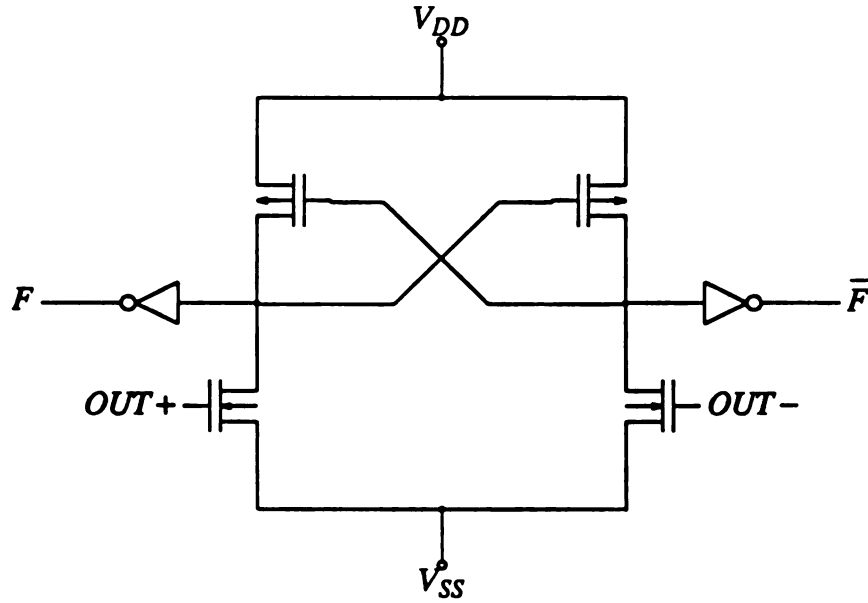


Figure 2.12. DCVS buffer used in differential pass transistor logic

Both DCVS logic and differential pass transistor logic structures have substantial area savings over fully complementary CMOS. However, when compared with corresponding DCVS circuits which suffer from longer rise time and fall time due to the fact that the p-type pull-ups have to fight the bigger n-type pull-down trees, differential pass transistor logic has shorter delay by using a fixed-size DCVS inverter.

2.9. Zipper CMOS

Charge-sharing is a prevalent problem in dynamic CMOS and Domino CMOS circuits. The problem arises due to the fact that not all internal nodes in the logic block are fully precharged. Figure 2.13 shows a Domino CMOS circuit for implementing a Boolean function f with clock signal ϕ to illustrate the charge sharing problems. During the precharge phase of the clock, the p-type transistor Q_1 is ON while the n-type

transistor Q_8 is OFF. Node N_4 is charged to V_{DD} and the output from the inverter is at the voltage level close to 0 V. This situation occurs at that time at every logic block including those whose outputs are connected to the inputs X_i of this particular block. As a consequence, the inputs X_i and those of all other blocks are close to 0 V during the precharge phase. Therefore there is no electrical path from node N_4 to node N_1 . When the clock turns to the evaluation phase, transistor Q_8 is ON conditionally creating the path from the node N_4 through the switch network to the ground, which in turn makes the output of the inverter F the logic ONE. This value is the input to the subsequent logic blocks and can cause the output of those blocks to switch to ONE in the “domino” fashion.

Obviously, the charge is stored at the node N_4 during the precharge phase. The nodes N_3, N_2, N_1 might have been discharged during the previous cycle and thus have no charge. Therefore, during the evaluation phase, there may be an electrical path to several discharged nodes without an electrical path to the ground, causing so-called charge redistribution problems. If the capacitance at the storage node N_4 is C_s , and the uncharged capacitance internal to the switching network is C_i , it is possible that the resulting voltage at the node N_4 is reduced below the inverter threshold I_{TH} ; i.e.

$$V_{DD} \times \frac{C_s}{C_i + C_s} \leq I_{TH}$$

where the maximum original voltage V_{DD} at the node N_4 is assumed. This charge redistribution will cause the inverter at the output of the switching network to falsely switch, thus placing the incorrect value on the line causing other circuits to discharge falsely.

All dynamic CMOS and Domino CMOS circuits suffer from signal degradation caused by charge redistribution and leakage current. In contrast to these circuits, Zipper CMOS (ZCMOS for short) is inherently immune to the problems of instability and charge-sharing. ZCMOS is proposed by Lee and Szeto [LeSz86], and the basic ZCMOS

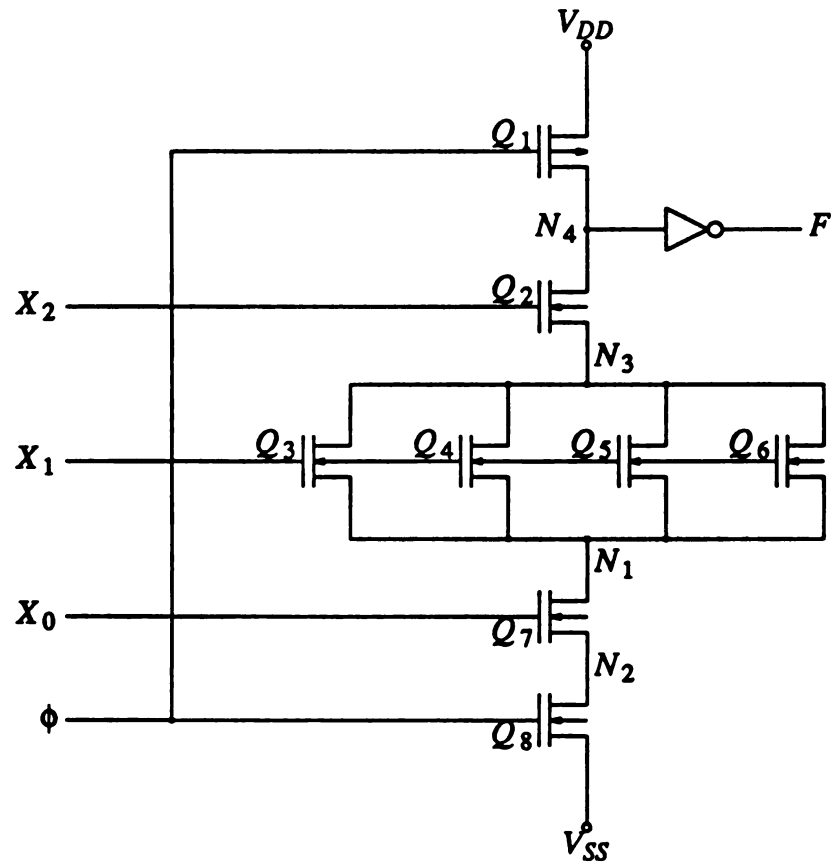


Figure 2.13. Charge sharing problem in dynamic CMOS and Domino CMOS logic

structure is depicted in Figure 2.14. The major components in a ZCMOS circuit are the Zipper Driver circuit and the alternate n-type and p-type logic blocks. The Zipper Driver generates four strobe signals, ST , \overline{ST} , ST' , and $\overline{ST'}$, at various voltage levels based on a single clock to drive subsequent dynamic blocks.

During precharge, the output of every n-type block is high and that of every p-type block is low. This ensures that the transistors driven by the output of each dynamic stage will be off. During evaluation, strobe signal ST' is at its “poor 1” voltage level, and $\overline{ST'}$ is at its “poor 0” voltage level. The precharge transistors are, therefore, partially on to produce a residual current for sustaining the precharged value of the internal nodes and overcoming the charge redistribution problem. The output of each n-stage can make at most one transition from high to low, and the output of each p-stage can undergo only one transition from low to high. This “staggered” fashion in which signals propagate through n-type and p-type blocks gives rise to the name “Zipper CMOS”.

Figure 2.15(a) shows the ZCMOS implementation of two stages of a ripple carry chain. A static FCMOS implementation of the same circuit is shown in Figure 2.15(b). P_i and G_i represent the propagation and generation terms at the i th stage. The carry-in (C_i) and carry-out (C_{out}) signals at the i th stage are related by

$$C_i = P_{i-1}C_{i-1} + G_{i-1}.$$

Generally speaking, ZCMOS circuits lead to the use of multivalued logic due to various levels of clock signals. It improves the delay time of a CMOS circuit at the expense of increasing its dynamic power consumption. Thus, design tradeoffs exist for various logic structures, and the selection of an appropriate logic structure is basically a choice among a number of factors such as area, power dissipation, speed, noise margin, and even fault susceptibility.

In general, pseudo-nMOS, dynamic CMOS, and pass transistor logic provide sub-

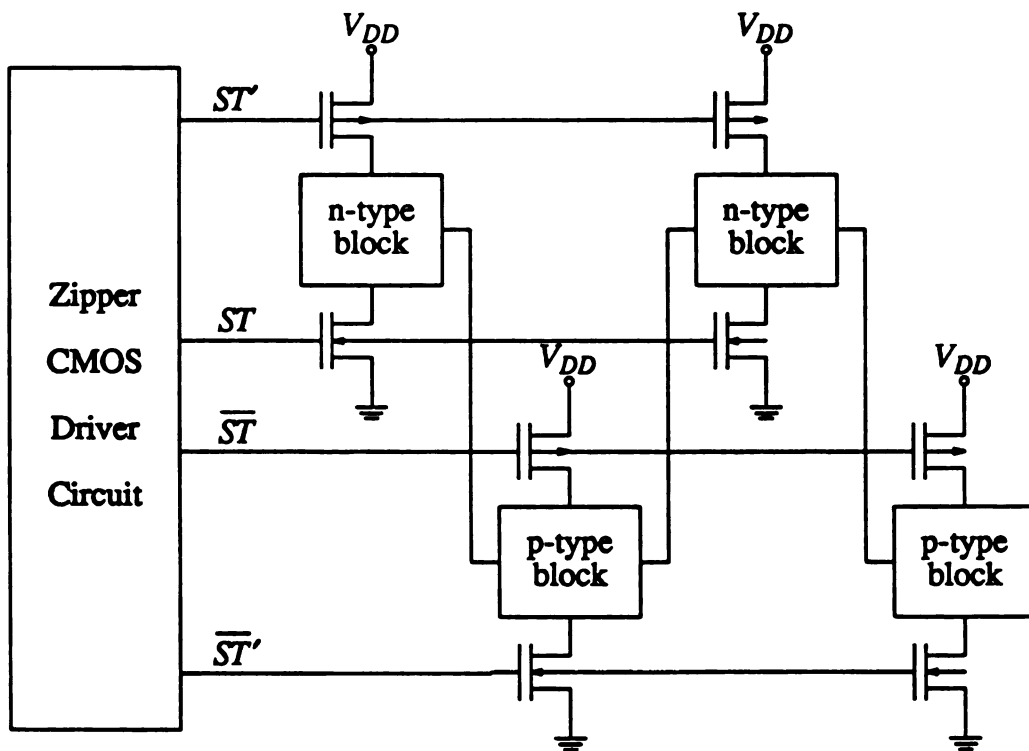


Figure 2.14. Zipper CMOS circuit structure

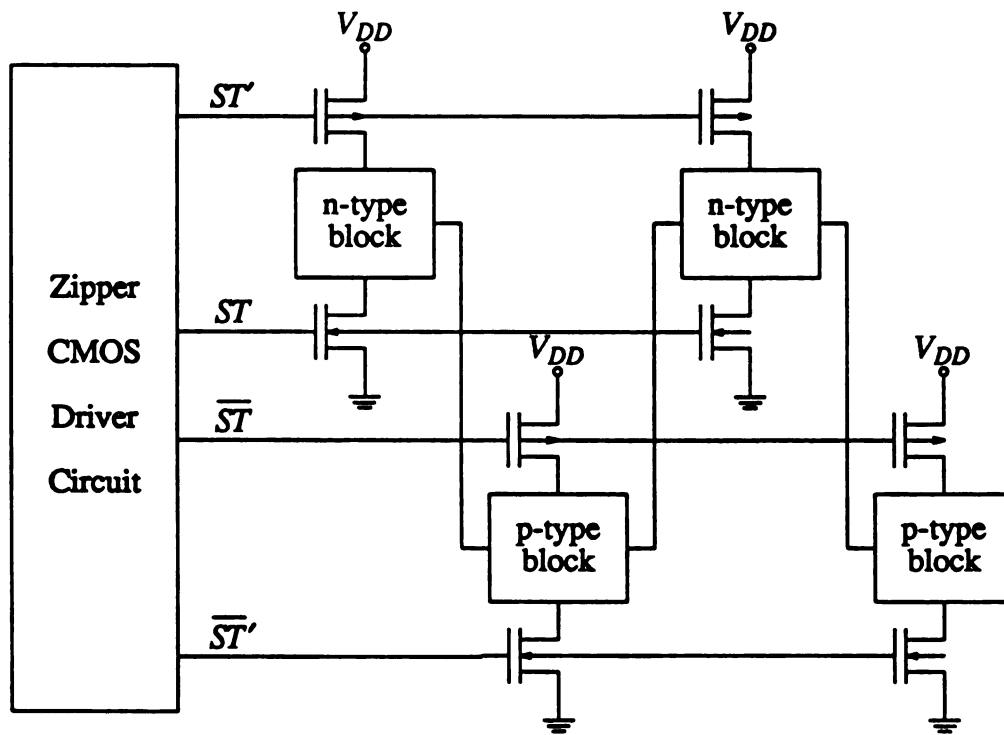
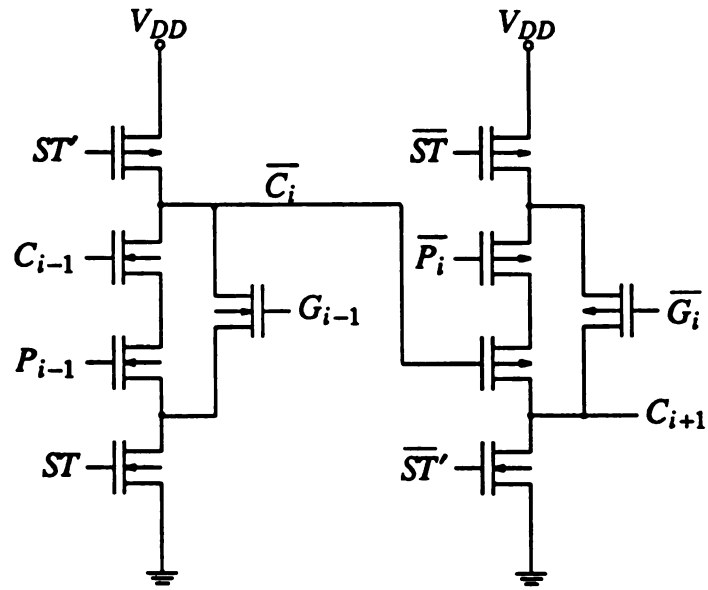
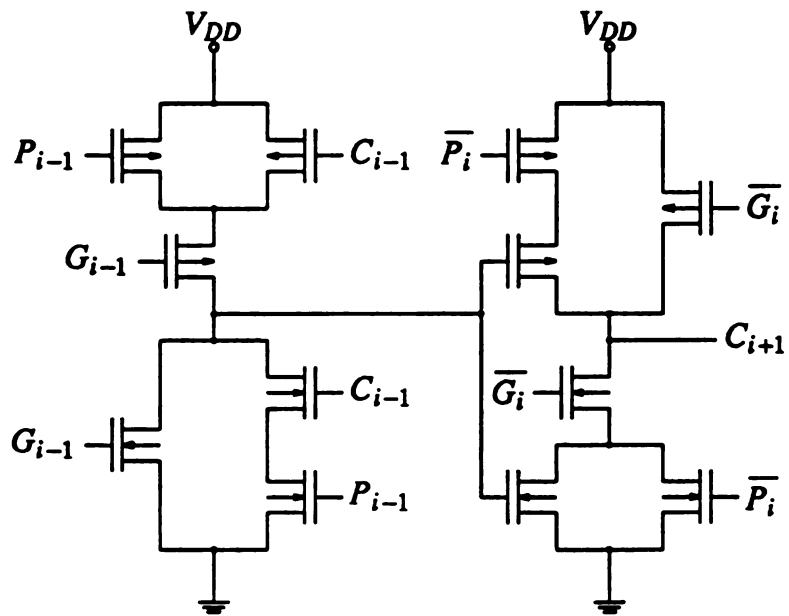


Figure 2.14. Zipper CMOS circuit structure



(a)



(b)

Figure 2.15. Two stages in a (a) ZCMOS (b) FCMOS ripple carry chain

stantial savings in terms of area, which can be estimated to a first order by the number of transistors and also affected by routing considerations. Note that dynamic gates require a precharge interval, and this may or may not impact performance. As a rule of thumb, small logic blocks are usually implemented statically while large logic cascaded structures might be implemented with dynamic or domino CMOS logic.

2.10. Design Styles

VLSI design styles are undergoing rapid evolution. As the density and the number of devices on a VLSI chip increase, methods of dealing with complex design problems have to be developed. Several different methodologies have been developed for designing integrated circuit chips, ranging from full-custom approach, in which the entire IC is designed from scratch, to semi-custom approaches, in which the designer builds the chip from a collection of pre-designed components. One of the advantages with various design styles is to allow system designers the option of implementing high performance systems directly in silicon. Figure 2.16 illustrates three semi-custom approaches -- gate array, standard cell, and macrocell.

The *standard cell* approach provides a cell library for functions to be interconnected. The cells in a design are arranged in rows with area for making interconnect between the rows. Usually the cells are constrained to be of uniform height but variable width, and the designer retains the flexibility of both placement and interconnect.

The *macrocell* approach can be treated as an extension of the standard cell approach, with the additional feature that designers can either use their own local cells or circuits generated from automatic tools. Some macrocells generated from automatic tools are parameterizable, rectilinear blocks that are customized from a library of templates. These cells are either connected by abutment or by automatic routing.

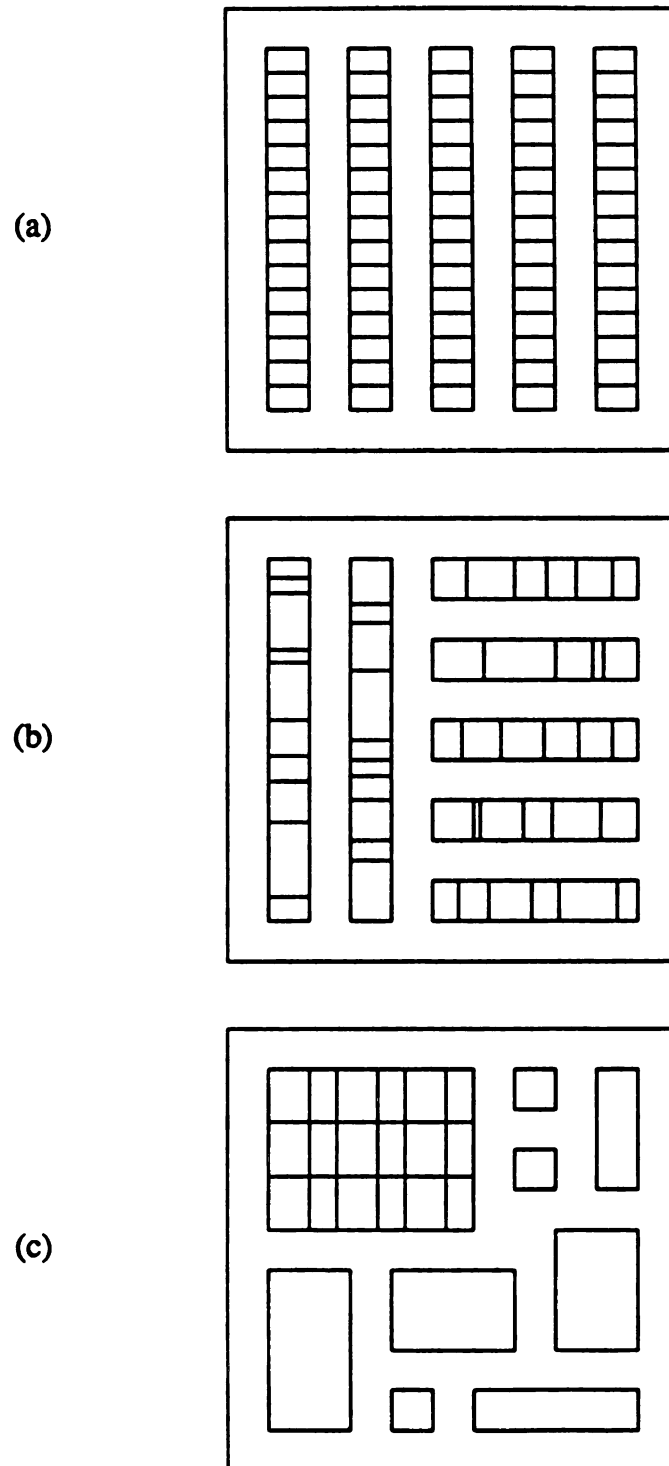


Figure 2.16. Semi-custom design styles (a) gate array, (b) standard cell, and (c) macrocell

Obviously, designs using macrocells offer still more flexibility than standard cell designs.

Gate arrays are semi-custom integrated circuits, which are mostly made ahead of time and which are customized to the user's needs by defining one or more layers of metal on the die itself. This method provides macros with predefined placement on the chip. The interconnection of these macros is customized for specific applications. Because the gate array is mostly made ahead of time, it offers significant savings in both cost and time. On the other hand, in the standard cell approach the wafers are not built ahead of time. Therefore, in the standard cell approach, the designs of all the mask levels of each cell have to be stored in a computer, in contrast to typically one or two such mask levels for gate array macros. This implies that the tooling cost for a standard cell library is considerably higher than that of a similar library of macros for gate arrays. It is true that the die cost of a gate array is higher than that of a full custom die. However, in low-volume applications, the nonrecurring development costs are the dominating factors, not the differential piece-part costs. It is here that the gate array approach has a clear edge over the full-custom and standard cell approaches. At the end of 1985, gate arrays outsold standard cell ICs by a four-to-one margin. By 1990 it is estimated [Holl87] that more than half of all semiconductors sold will be semi-custom designs of which gate arrays and standard cell ICs are a major part.

There are also a number of major arguments in favor of the standard cell approach compared with gate arrays. The ability to put an extra 20 to 30% or more of functionality on a die using the standard cell approach often enables the system to be partitioned in such a way as to minimize the connections to the world outside the chip. Because the bulk of the power dissipated is often in the I/O buffers, minimization of their numbers can drastically cut the power dissipated by the chip. Speed will also be enhanced by the elimination of the capacitance associated with the no-longer-needed I/O buffers. Thus, given a standard cell library and gate array macros, some researchers are working on

converting gate arrays to standard cell designs.

Today, standard cells and gate arrays sometimes compete for a given customization project. However, due to the shrinking feature sizes and larger die sizes, there is nothing to prevent standard cells from incorporating gate arrays. One or more of the standard cells in the library could simply be a gate array cell of a given size. Such an arrangement has the advantage of permitting a chip to be designed so as to accommodate a rapidly changing market or to accommodate a low-entry-cost product. In the future, one will see combinations of customization techniques on one chip.

The *full-custom* design style incorporates maximum flexibility, and a full-custom IC is completely made with no interruption in the manufacturing cycle. Theoretically, a full-custom IC is one in which every transistor is designed, sized and placed appropriately for the function that particular IC performs. However, in the VLSI era with as many as two million or more transistors on a chip, a relatively small number of transistor designs are usually used over and over again, and groups of circuitry are replicated where possible. Because of its flexibility, full-custom design offers a number of advantages. Full-custom chips can often be made denser and faster than their semi-custom counterparts, as well as offering better yield and lower power consumption. Sadly, these advantages are frequently offset by the greater time it takes to design custom chips. A survey of the IC industry [ReSS85] shows that even moderately small (2,000 gates) full-custom chips take approximately three times as long to design as comparable semi-custom chips. For larger chips, the differences are even more extreme.

Chips designed using the full-custom approach take longer to design mainly because more aspects of their layout are under designer control, and the verification of layout pieces is much more difficult. The designer creates each circuit transistor-by-transistor, rather than from a collection of logic gates. The layout of a given circuit is not standard, but depends on where that circuit is used on the chip. Components are often

connected by abutment or local hand-routing to compact the design, instead of through standard routing channels by an automatic router. The lack of automatic tools for full-custom design make these cells time-consuming to enter, and even more time-consuming to verify or change.

From the above introductory sections, it is clear that there exist many logic structures and various design methodologies by which a VLSI design can be implemented. To automate a design or verification process, one needs to have a base, namely a design representation, to represent and record the design. Many design representations have been proposed in the literature and actually used at different levels. In the next chapter, a brief discussion on how a design can be represented at various levels, followed by the logical circuit expressions at the switch level, will be given.

CHAPTER III

LOGICAL CIRCUIT EXPRESSIONS

In this chapter, the essence of abstraction and several levels of abstraction that various VLSI design representations use are discussed. The representation scheme proposed in this thesis, namely the *logical circuit representation*, is described in detail. Approaches for generating logical circuit expressions are proposed. Unlike traditional device listing approaches, which represent only circuit structures, logical circuit expressions combine structural data with behavioral information, and thus illustrate a way to reduce the difficulty of information transformation between behavioral and structural representations of CMOS circuits.

The VLSI design process spans a broad spectrum of disciplines in many different fields. Because of the diversity of tasks and design issues, a systematic approach to breaking the design process into a number of design levels and subtasks is essential. While the design of single chips with the complexity of a microprocessor is a relatively recent phenomenon, people have been designing software systems of similar complexity for many years. Thus, one might naturally wonder whether the experience and tools used in software design could be carried over directly to the world of VLSI. Certainly there are several factors in common. In both the circuit world and program world, the problem can be characterized as one of implementing algorithms [Ullm84]. For example, the Least-Recently Used (LRU) processing unit designed for hierarchical memory systems [WuNi86] is principally a hardware design for the LRU algorithm. Both large hardware and large software designs can be carried out by several designers, and coordinating pieces of a project is always a difficult job. Another important similarity between

software design and the world of VLSI is that designers are much more productive when they are given high-level languages in which to express their designs. For example, programmers almost never work in machine code, and they generally prefer higher level languages such as C and Pascal. Similarly, working at a higher-level circuit representation will relieve the designer's burden, and therefore improve the designer's productivity and reduce the turn-around time of the design. Of course, it is a dangerous practice to oversimplify the abstractions of electronic circuit behavior. Thus, the selection of an appropriate abstraction level for individual projects is basically a trade-off between accuracy and productivity, and is a matter of art.

3.1. Levels of Abstraction

The virtue of abstraction is data reduction. It is a method to replace an object by a simplified one that defines the interactions of the object with its environment, while deleting the internal organization of the object. There are a number of different levels of symbolic representation for MOS VLSI circuits. In order to understand the overall design system, it helps to be aware of the many levels of abstraction that various design representations use. The principal sorts of design representations that appear are the following.

Geometry Representation

Geometry languages, such as the popular layout description languages CIF (Caltech Intermediate Form) [McCo80] and Calma GDS II, use colored rectangles or colored shapes as their primitives. Once a layout has been drawn it can then be digitized, or translated into some machine-readable form. Silicon foundries take designs expressed in

these languages as inputs to fabricate the chips.

Stick Representation

In a stick diagram, transistors and vias are represented by points of a grid, and wires are represented by widthless lines. Such stick diagrams may be annotated with important circuit parameters such as length-to-width ratios, if needed. Information regarding the thickness of wires and the exact positions of the points is supplied by the compiler that translates the stick representation into geometry. A typical language can be found in [MaNE82]. One of the significant advantages of this sort language is that it is almost impossible to make a design rule error, since it is the responsibility of the compiler to position elements and wires due to the higher-level abstraction.

Switch-Level Representation

Switch-level primitives are transistors and nodes, which are points connected to one of the three terminals of one or more transistors. These three terminals are generally called the source and drain, which are the two ends, and the gate, which separates the source and drain. ESIM [BaTe80], a switch-level language for event-driven switch-level simulations, is a typical example at this level. The *Logical Circuit Expressions* representation to be proposed later in this chapter falls into this category.

PLA Personalities and Logic Representation

The programmable logic array, or PLA, is a specialized layout style for implementing switching logic and sequential machines. Truth tables and tiles are usually used to generate PLAs in various styles [KaVa81, OuSM85]. This representation leads to regular layouts and allows design to be carried out at a relatively high level. Similar to PLA per-

sonalities, the logic level of the design process uses an abstraction of the underlying electrical circuit in which the currents and voltages are limited to discrete levels. In digital circuits, the two levels of signals that are permissible are those that represent logic ZERO and logic ONE. Thus, ordinary Boolean logic, while augmented with a notion of sequentiality such as using the key word “LAST” in LGEN [John83] so one can refer to the “current” and “previous” values of a variable, also forms a high-level way to specify designs. However, with a language at the logic level, the designer has lost all control over the layout and relative positioning of circuit elements.

Finite State Machine Languages

Finite state machine languages such as SLIM [Henn81] and PEG [Hama83] are designed to specify the control of a microprocessor or similar chips. They can be compiled into logic or PLA personalities, for example. While a PLA is often a fast approach for implementing a sequential machine, the compiler for the language must decide on the coding of states for the machine in a way that minimizes the area of the PLA.

Procedure Languages

The extreme of high-level design languages is an ordinary programming language, in which the algorithm to be performed by the circuit is written. The sequencing rules for statements of the program are embedded into the control of the chip, while the variables are represented by registers. An attempt at compiling Pascal programs into silicon can be found in [Tric85]. Procedural languages that are somewhat more specialized to circuit design than are ordinary programming languages also exist. These languages, often called register transfer languages, deal with registers and specify the sequencing of actions involving these registers in terms of events at the inputs and events at other registers.

Generally speaking, it is important to simplify the model of integrated circuitry, so as to more quickly and easily analyze or explain the function of a given circuit, and more easily visualize and invent new circuit structures without drifting too far away from physically realizable and workable solutions. An extremely simple model based on transistor “switches” is proposed in [MeCo80], in which transistors are treated as valves or switches. Many switch-level simulators such as ESIM [BaTe80, Term83] and MOSSIM [Brya81] have similar models for analyzing circuit behavior.

3.2. Series-Parallel Network

The focus in this chapter is on circuit representation at the transistor level. Not only are switch-level simulators based upon transistor networks, but also symbolic layout systems such as TOPOLOGIZER [KoWe85] use a transistor connection description as input to generate geometric layouts. As far as circuit connectivity is concerned, one of the first attempts to list all electrical networks meeting certain specified conditions was made in 1892 by P. A. MacMahon [MacM92] who investigated combinations of resistances. Among various switching circuits, the simplest structure is the so-called *series-parallel* network.

A series-parallel network is a simple type of connection which occurs frequently in both theoretical and applied electrical engineering. The concept of series-parallel connection is intuitive, but this type of network plays a prominent role in Shannon’s well known application of Boolean algebra to switching circuits [Shan38]. One reason for the importance of series-parallel networks stems from the fact that the joint resistance is easily evaluated by the following two rules due to Ohm:

- (1) Resistance is additive for resistors in series, and

- (2) Reciprocal resistance is additive for resistors in parallel.

The following inductive definition of a series-parallel network is given in [RiSh42]:

Definition: A single element is a series-parallel network. A network N is series-parallel if it is either a series or a parallel connection of two series-parallel networks.

There are also several alternative definitions of series-parallel networks [Duff65]:

- (1) A network is series-parallel if it is a direct construction of the series operation and the parallel operation.
- (2) Two edges x and y are said to be *confluent* if there do not exist two closed curves C_1 and C_2 such that C_1 meets x and y in the same direction but C_2 meets x and y in the opposite direction. A network is series-parallel if it is confluent; i.e., every pair of its edges is confluent.
- (3) A network is series-parallel if it does not have an embedded Wheatstone bridge.

Figure 3.1 shows a circuit with a Wheatstone bridge [Nils83]. Suppose that the directions of f and c are chosen so that they have the same sense relative to the loop (f, b, c, d) . It then results that f and c have opposite sense relative to the loop (f, a, c, e) .

The series-parallel combinations do not exhaust the possible networks since they exclude all bridge arrangements like the Wheatstone bridge, but they are an important class because of their simplicity. When the number of elements is less than five, all networks are series-parallel. When the number of elements is exactly five, there is only one bridge-type network, the Wheatstone bridge [RiSh42]. As the number of elements rises, both series-parallel and bridge-type networks increase.

The series-parallel networks are interesting in themselves in another setting, namely the design of switching circuits. In fact, most switching circuits are series-parallel in structure, and all existing low-level synthesis tools use series-parallel networks to implement logic functions. Another property of a series-parallel network is that the direction

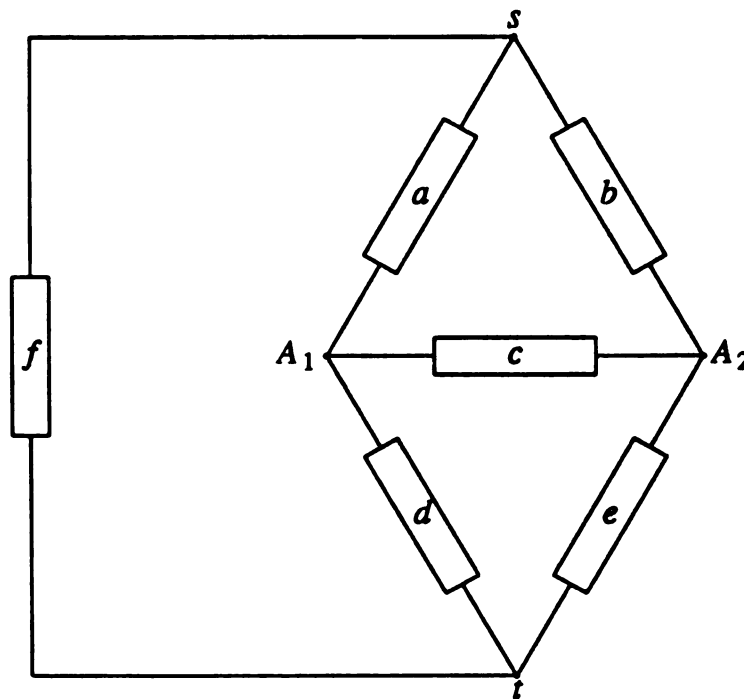


Figure 3.1. A circuit with a Wheatstone bridge

of the current flow through a given edge is independent of the resistance of the edges of the network [Duff65]. In Figure 3.2, the edges f and c can be replaced by a battery and a meter, respectively, to measure the current flow through c so as to estimate the resistance of the resistors. Since the circuit is not series-parallel in structure, the measurement obtained from the meter might be either positive or negative, indicating the direction of the current flow through the edge.

A series-parallel CMOS switching network can be described using a symbolic representation. Such a circuit is represented by a set of expressions, the terms of the expressions corresponding to the various n-channel and p-channel devices in the circuit. As far as the circuit function is concerned, a calculus is used for manipulating these expressions by simple mathematical processes, most of which are similar to rules in the

ordinary algebra. This calculus associated with *logical circuit expressions* is analogous to the calculus of propositions used in the symbolic study of logic. Once a symbolic representation is available, the desired functions of a synthesis problem can be written as a system of expressions, and the expressions are then manipulated into the form representing the simplest circuit. The circuit may then be immediately drawn from the expressions.

Based on series-parallel networks, a CMOS switching circuit can be partitioned into a number of components, most of which are series-parallel in structure. The notation to be used is taken chiefly from symbolic logic, and will be shown in the following sections.

3.3. Logical Circuit Representation

In terms of representing transistor connectivities, traditional approaches for low-level circuit representation are mainly device listing approaches, in which circuit elements are listed one by one using nodes. For example, the commonly used circuit simulator SPICE [Nage75] uses nodes and circuit elements to perform nodal analysis. The switch-level event-driven simulator ESIM uses statements in the form of

$$\langle type \rangle \langle gate \rangle \langle source \rangle \langle drain \rangle$$

to represent transistors, where $\langle type \rangle$ can be either p or n for p-type transistor and n-type transistor, respectively.

Device listing approaches used at either circuit level or transistor level are really simple, but are not suitable for describing system functions merely because they do not supply any information about how a circuit works. In fact, the difficulty of information transformation among various representations makes design automation systems for VLSI circuits less mature. Much unnecessary effort is spent in transforming information from one representation to another. In order to overcome this difficulty, we need a

representation which combines both structural and functional information so that circuit functionality can be easily extracted from geometric layouts and/or transistor schematics.

On the other hand, circuit schematics at the transistor level cannot be completely specified in terms of Boolean expressions. Many parts of a circuit may not be recognized as simple logic primitives. Some circuit blocks such as mutual exclusion elements [McCo80] used in arbiter designs are difficult to specify in terms of Boolean expressions due to their nondeterministic behavior. Given a circuit schematic which contains memory elements, pass transistor logic, and combinational gates of different logic structures, a complete description based on gate-level Boolean expressions is impossible. The difficulty in developing a Boolean description is due to the “*incomplete logic property*” of transistor-level circuit schematics.

In many CMOS logic structures, a signal at a p-n block junction of a circuit might be in the high impedance state for some input combinations. This means that the output of a circuit might depend upon the previous value of the gate. The clocked CMOS circuit shown in Figure 3.2(a) is a typical example, in that its output is floating if $\phi = 0$. Thus, the function of the circuit is somewhat different from the Boolean equation $z = \bar{a}$ or $z = \phi \bar{a}$. Similarly, node f in the dynamic CMOS circuit of Figure 3.2(b) cannot be completely specified in terms of its inputs using a Boolean equation.

This property found in many circuits is referred to as the *incomplete logic property*, which keeps the behavior of the circuits away from Boolean descriptions, and sometimes makes a combinational gate behave like a sequential element. Thus, a representation which is one-level below the level of Boolean description is needed so that it carries both structural and functional information as much as possible.

Incomplete logic properties of digital circuits have been handled in many simulation tools by means of a high-impedance state Z [Brya81, Haye84], either implicitly or expli-

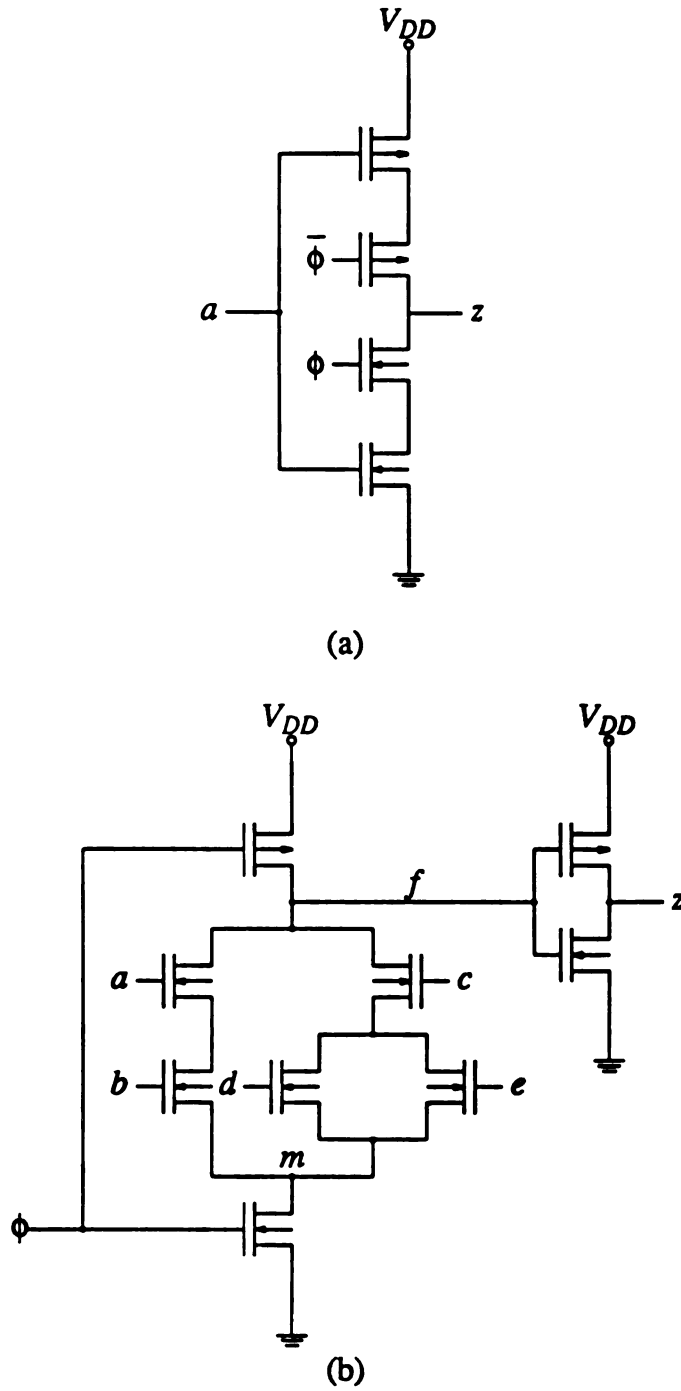


Figure 3.2. Circuits with incomplete logic property: (a) a clocked CMOS inverter, and (b) a Domino CMOS gate

citly. Although Boolean expressions are not suitable for describing circuit functionality at the transistor level due to incomplete logic properties, they can be used to represent logical paths from signal sources to stage outputs.

Simple transistor switch models for describing transistor states have been widely used for switch-level simulation. An n-type transistor is switched on if its gate signal is logically ONE, and a p-type transistor is conducted if its gate signal is logically ZERO. In general, these simple models offer useful information about circuit functionality. Since the logic value at an output node of a circuit comes from its signal sources such as V_{DD} and V_{SS} , the function of a circuit can be represented in terms of paths. The function of an output node in a circuit is partitioned into several parts, each of which has a signal source and a simple path from the node to the source. The logic expression of a path is determined by the gate signals of the transistors along the path.

A circuit schematic can be specified in terms of a number of nodes and their relationships. *Primary source nodes* are V_{DD} , V_{SS} , and input sources. The connection between two data terminals (source or drain), one from a p-type transistor and the other from an n-type transistor, defines a *junction node*. However, if connections exist for both pair of data terminals, i.e., if they are actually a pair of transistors in a transmission gate, then both junctions are not considered as junction nodes. A *gate node* is defined as a connection to the gate terminal of a transistor. All output terminals connected to the outside world are initially considered as gate nodes. If a gate node merges with a junction node, the result is a junction node. Thus, the function of a CMOS circuit is specified in terms of a set of *logical circuit expressions* for all gate nodes and junction nodes.

The logical circuit expression associated with a node z can be expressed in the form of

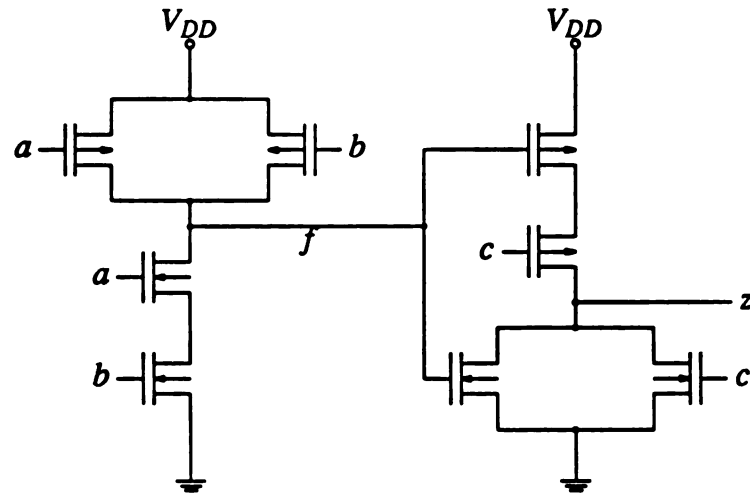
$$z = \sum_j [n_1 n_2 \cdots n_{r(j)}] v_j \cdot \sum_k [p_1 p_2 \cdots p_{s(k)}] v_k$$

where z is either a gate node or a junction node, v_j (v_k) is either a primary source node or a junction node reachable from z , and n_i (p_m), $1 \leq i \leq r(j)$ ($1 \leq m \leq s(k)$), is the gate signal of a n-type (p-type) transistor along the path from z to v_j (v_k). Note that a transistor in a path is represented by its gate signal, and logic variables involved in simple paths are the same as variables in Boolean expressions. Boolean laws, such as associative and distributive laws, are applicable. In general, v_j is reachable from z if and only if there exists an *n-type logic path* $[n_1 n_2 \cdots n_{r(j)}]$ such that the corresponding n_i , $1 \leq i \leq r(j)$, is neither a junction node nor a gate node. A *p-type logic path* can be similarly defined. After all junction nodes and gate nodes are identified, the entire transistor circuit is virtually divided into a number of disjoint partitions.

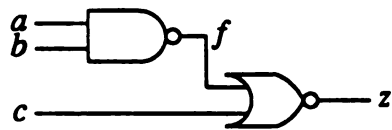
Conventional logic primitives such as NOT, NAND, and NOR used as elements in fully complementary CMOS circuits can be represented as $\bar{a} = [a]0 \cdot [a]1$, $\overline{ab} = [ab]0 \cdot [a + b]1$, and $\overline{a + b} = [a + b]0 \cdot [ab]1$, respectively. The circuit schematic of a CMOS NAND gate followed by a CMOS NOR gate is depicted in Figure 3.3(a). Since there is one simple n-type path from the junction node f to V_{SS} and two simple p-type paths from f to V_{DD} in Figure 3.3(a), only one term ab is associated with 0 (V_{SS}) and two singular terms a and b are associated with 1 (V_{DD}). Logical paths in the circuit can be easily realized. The equivalent logic diagram is shown in Figure 3.3(b), and the logical circuit expressions are given in Figure 3.3(c).

It is clear that other CMOS logic structures have similar structural rules for establishing similar relationships. For example, the logical circuit expressions of NOT, NAND, and NOR logic primitives in pseudo-nMOS logic are represented as $\bar{a} = [a]0 \cdot [0]1$, $\overline{ab} = [ab]0 \cdot [0]1$, and $\overline{a + b} = [a + b]0 \cdot [0]1$, respectively.

Figure 3.4 shows a CMOS decision-making module containing a cross-couple circuit structure. It is a typical circuit for which the output cannot be logically defined in



(a)



(b)

$$f = [ab]0 \cdot [a + b]1$$

$$z = [c + f]0 \cdot [cf]1$$

(c)

Figure 3.3. A two-stage CMOS circuit: (a) transistor schematics, (b) logic diagram, and (c) logical circuit expressions

terms of its input signals, and switch-level simulation is difficult to conduct due to its asynchronous timing-sensitive circuit behavior. The function of the circuit can be found elsewhere [WuNi87]. To simplify the transistor network and provide a hierarchical viewpoint, logic primitives such as NOR and OR gates are used as well as transistors to represent the CMOS circuit. The symbol D is used as the output of the two-input NOR gate. In this mixed-mode circuit schematic, N_1 , and N_2 are junction nodes, and N_3 , N_4 , and N_5 are gate nodes.

The logical circuit expressions of the circuit are listed below. Note that G_1 , G_2 , D , and R_c are represented by Boolean expressions.

$$G_1 = \overline{(\overline{G_c} + N_1 + \overline{R_1})}$$

$$G_2 = \overline{(\overline{G_c} + N_2 + \overline{R_2})}$$

$$D = \overline{(\overline{G_c} + N_5)}$$

$$R_c = N_3 + N_4 + D$$

$$N_1 = [R_1 \overline{R_2} \overline{G_c} + N_2]0 \cdot [N_2]1$$

$$N_2 = [R_2 \overline{R_1} \overline{G_c} + N_1]0 \cdot [N_1]1$$

$$N_3 = [R_1]N_2 + [\overline{R_1}]0 \cdot$$

$$N_4 = [R_2]N_1 + [\overline{R_2}]0 \cdot$$

$$N_5 = [R_1]R_2 + [\overline{R_1}]\overline{R_2} \cdot$$

This mixed-level representation combines circuit structure and logic expressions while most device listing approaches only provide the structure information of a circuit. It illustrates a way to reduce the difficulty of data transformation between behavior representation and structural representation of CMOS circuits. Compared with Boolean expressions, logical circuit expressions can be used to describe physical CMOS circuit structures and logical paths from output nodes to signal sources.

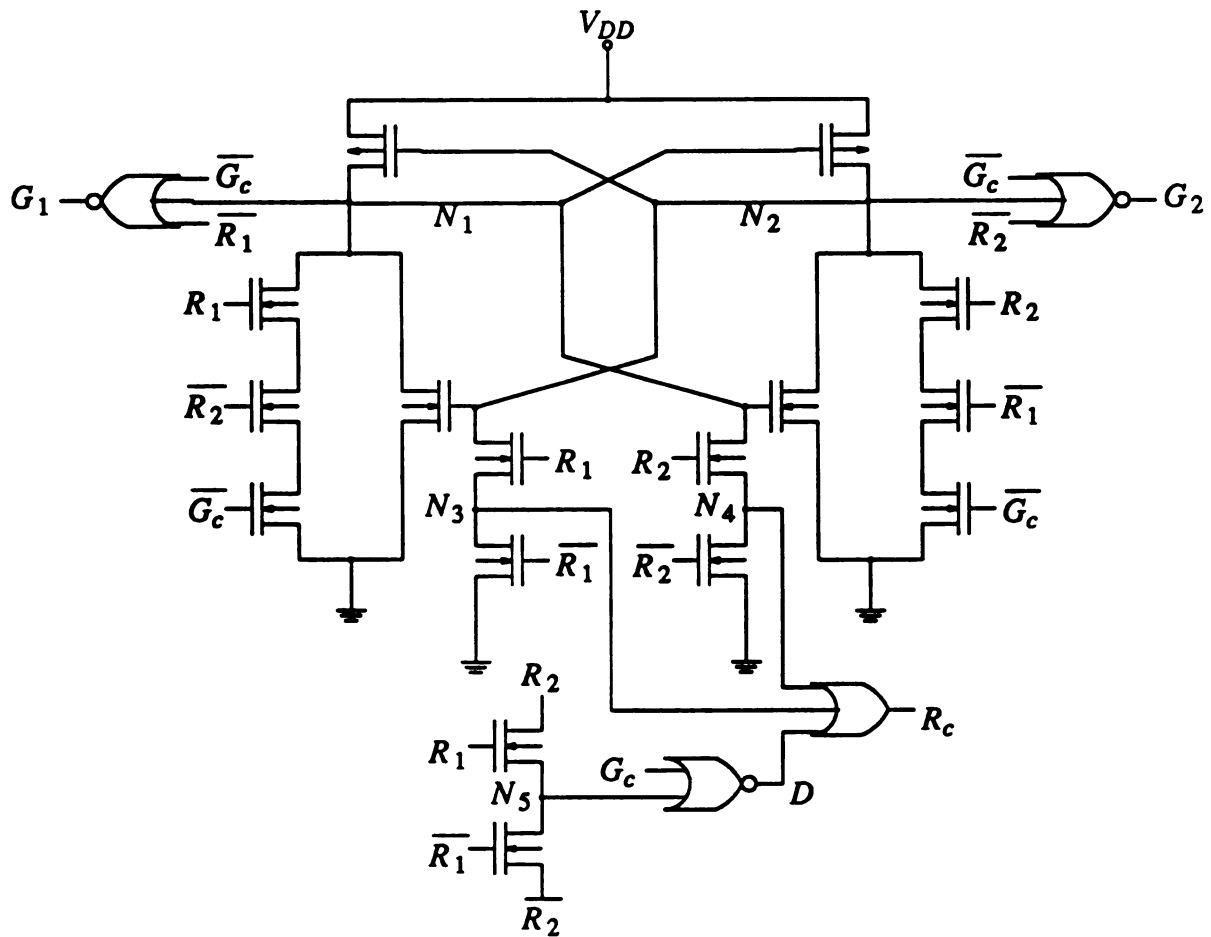


Figure 3.4. A CMOS decision-making circuit

When logical circuit expressions are used to describe transistor connectivities, they are referred to as *structural circuit expressions*, and the internal structure must be in the topology of *series-parallel* networks. For non-series-parallel structures such as the Wheatstone Bridge in Figure 3.1, *auxiliary nodes* are required to make the structure separable. Consider the Wheatstone bridge in Figure 3.1. If each edge of it is, say, an n-type transistor, then we have a n-type path between the starting node s and the node t :

$$s = [ad + ace + bcd + be]t \cdot$$

In the mean time, we have

$$s = [a]A_1 + [b]A_2 \cdot, \text{ where}$$

A_1 and A_2 are auxiliary nodes, $A_1 = [c]A_2 + [d]e \cdot$ and $A_2 = [c]A_1 + [e]t \cdot$. Like junction nodes and gate nodes, each auxiliary node is used as a starting point from which a logical circuit expression is constructed. However, paths from an auxiliary node to any junction node can be discarded, because the actual conducting path is always the other way around (from the junction node to the auxiliary node).

Similarly, while the logical circuit expressions for z and \bar{z} in Figure 2.8(b) are

$$z = [(ba + \bar{b}\bar{a})p]0 \cdot [\phi]1 \quad \text{and}$$

$$\bar{z} = [(b\bar{a} + \bar{b}a)p]0 \cdot [\phi]1,$$

we have

$$z = [b]M + [\bar{b}]N \cdot [\phi]1 \quad \text{and}$$

$$\bar{z} = [\bar{b}]M + [b]N \cdot [\phi]1$$

in terms of structural circuit expressions, where $M = [a]P \cdot$, $N = [\bar{a}]P \cdot$. The node P is a *separating node*, which will be described further in the next section.

The procedure for generating logical circuit expressions is given in Section 3.4. Generally speaking, a logical circuit expression could be a structural circuit expression;

but a transistor network need not to be series-parallel in structure. Although conventional logic operators remain unchanged in structural circuit expressions, Boolean associative and distributive laws are no longer applicable. Concatenated variables represent devices connected in series, and terms separated by “+” represent devices connected in parallel. Thus, two sets of structural circuit expressions may be the same in terms of functions but different in terms of circuit structures.

3.4. Generating Logical Circuit Expressions

Whenever a transistor is extracted from a given mask artwork, its data terminals (source and drain) are checked for p-n block junctions and its gate terminal is marked as a gate node. Output nodes connected to the outside world are initially considered to be gate nodes, and input nodes of the circuit are typically signal sources and are as strong as junction nodes. When a bidirectional path exists, two logical circuit expressions can be obtained. During the circuit extraction process, a gate node might be combined with other gate nodes and/or eventually linked to some node at a p-n block junction and thus deleted from the gate node list. After all junction nodes and gate nodes are identified, the entire transistor circuit is virtually partitioned into a number of components. Each component is bounded by *terminate nodes*, which are the collection of primary nodes, gate nodes, and junction nodes. Each gate node is then used as the starting point for finding symbolic paths to its signal sources in the component. When a node such as V_{SS} , V_{DD} , input source, or a junction node is found along a path, it is considered to be a signal source for the path. Paths having both true and complementary signals of the same input in series are discarded. If a simple path is found from the starting gate node to a junction node, edges involved in the path are in both virtual components and are therefore marked out in the other side. After all gate nodes are finished, the searching procedure continues

for each junction node.

Formally, a transistor network is treated as a connected graph $G(V, E)$, where V is the set of vertices (nodes) and E is the set of edges (transistors). It is clear that not all vertices in a graph need to be exploited. In order to construct the logical circuit representation of a circuit, only junction nodes and gate nodes are used. Figure 3.5 and Figure 3.6 show the corresponding graph diagram of the circuits in Figure 3.3 and Figure 3.4, respectively. All terminate nodes, except junction and gate nodes to be expressed, are split as many times as needed to help visualize disjoint components and recognize parallel structures, since terminate nodes form component boundaries. N-type edges are represented by solid lines, while p-type edges are represented by dashed lines. Fill-in bullets are junction nodes, and concentric circles are gate nodes. Note that all labels (gate signals) associated with edges are ignored in both diagrams for simplicity, and all dotted lines will be marked out after all gate nodes are traced.

3.4.1. Separating Nodes

Before the procedure for generating logical circuit expressions is presented, there are a number of terms that must be defined. A connected graph $G(V, E)$ is said to have a *separating vertex* v if there exist vertices a and b , $a \neq v$ and $b \neq v$, such that all the paths connecting a and b pass through v . A graph which has a separating vertex is called *separable*, and one which has none is called *nonseparable*. Let $V' \subseteq V$. The induced subgraph $G'(V', E')$ is called a *nonseparable component* if G' is nonseparable and if for every larger V'' , $V' \subset V'' \subseteq V$, the induced subgraph $G''(V'', E'')$ is separable.

In a given circuit, terminate nodes, including all primary source nodes, gate nodes, and junction nodes, are physical limits of logical paths. Logical paths start from either a gate node or a junction node, and end when they reach another terminate node. In order

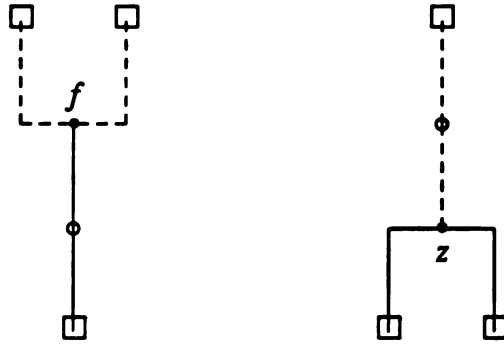


Figure 3.5. The virtual partitioning diagram of the circuit in Figure 3.3.

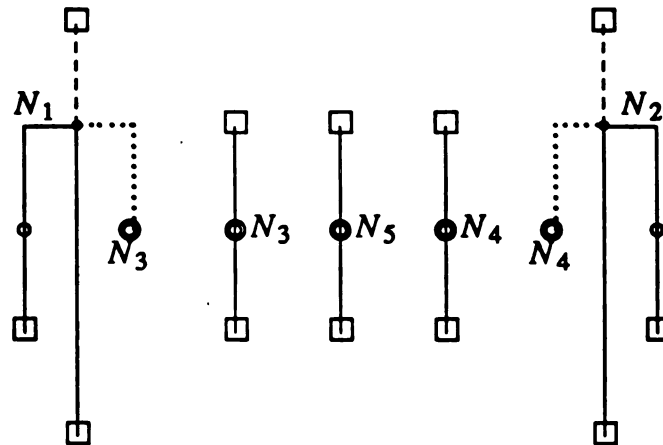


Figure 3.6. The virtual partitioning diagram of the circuit in Figure 3.4.

to generate logical circuit expressions, a modified depth-first search is conducted to find separating nodes in the circuit. After all separating nodes are identified, a general procedure for generating logical circuit expressions for each nonseparable component is then applied.

Let $G(V, E)$ be the graph representing the transistor schematic, $e(v, u)$ be an edge between two nodes v and u , and $P \subset V$ be the set consisting of only two nodes, V_{DD} and GND . During the search process, the edge set E is partitioned into a *tree edge* set T , which leads to nodes that have not yet been visited, and a *back edge* set B , where $T \cap B = \emptyset$ and $T \cup B = E$. Let s be the vertex from which we start the search. Either V_{DD} or GND is a good choice for s . The tree edge set T and the vertex set V form a tree. For each vertex v , other than s , the vertex $f(v)$ from which v is discovered is called the *father* of the node v . The *low-point* of a node v , $L(v)$, is defined as the least number, $k(u)$, of a node u which can be reached from v via a possible empty, directed path consisting of tree edges followed by at most one back edge. Note that $k(v)$ is the node number assigned by the algorithm for the node v .

An algorithm for finding separating vertices can be found in [Even79]. It is modified for finding separating vertices in a transistor network. The reason for modification is that the nodes for the power and ground lines should be treated differently from other nodes. Otherwise it is most likely that the whole circuit is nonseparable due to the fact that most of stages are connected to both V_{DD} and GND . Assume that s is the node for the power line. The algorithm is as follows.

Separating Algorithm:

- (1) Mark all the edges “unused”. Empty the stack S . For every $v \in V$ let $k(v) \leftarrow 0$.

Let $i \leftarrow 0$ and $v \leftarrow s$.

- (2) $i \leftarrow i + 1$, $k(v) \leftarrow i$, $L(v) \leftarrow i$ and put v on S . If v is the node for GND , go to Step (5).
- (3) If v has no unused incident edges, go to Step (5).
- (4) Choose an unused incident edge $e(v, u)$. Mark $e(v, u)$ "used". If $k(u) \neq 0$ which implies that $e(v, u)$ is a back edge rather than a tree edge, let $L(v) \leftarrow \text{Min} \{ L(v), k(u) \}$ and go to Step (3). Otherwise ($k(u) = 0$, a new node is found) let $f(u) \leftarrow v$, $v \leftarrow u$ and go to Step (2).
- (5) If $k(f(v)) = 1$, go to Step (9).
- (6) It is known that $f(v)$ is not the starting node at this point. If $L(v) < k(f(v))$, then $L(f(v)) \leftarrow \text{Min} \{ L(f(v)), L(v) \}$ and go to Step (8).
- (7) Now $L(v) \geq k(f(v))$ and $f(v)$ is a separating vertex. All the vertices on S down to and including v are now removed from S ; this set, with $f(v)$, forms a nonseparable component.
- (8) $v \leftarrow f(v)$ and go to Step (3).
- (9) All vertices in S down to and including v are removed from S . They form with s a nonseparable component.
- (10) If s has no unused incident edges then halt. Otherwise s is a separating vertex. Let $v \leftarrow s$ and go to Step (4).

Even [Even79] has shown that if $e(u, v)$ is a tree edge, $k(u) > 1$ and $L(v) \geq k(u)$, then u is a separating vertex of G . It has also been shown that in the case that $k(u) = 1$ (u is the starting node), u is a separating vertex if and only if there are at least two tree edges out of u .

Although this algorithm is more complicated than the depth-first search algorithm proposed in [HoTa73], its time complexity is still $O(|E|)$. This follows easily from the

fact that each edge is still scanned exactly once in each direction and that the number of operations per edge is bounded by a constant.

Figure 3.7 shows a two-input arbiter design. Its function can be found in [DiCl85, WuNi87]. The element marked “C” in Figure 3.7 is called a *Muller C-element*, or a *C-element* for short [MeCo80]. It is a binary storage device whose output becomes ONE when both its inputs are ONE, and becomes ZERO when both inputs are ZERO. A portion of the design is illustrated at the transistor level in Figure 3.8, which includes the mutual exclusion (ME) element, the logic elements G_1 , G_2 , G_3 , and a Muller-C element. The symmetric part (on the top of the schematic) in Figure 3.8 is the ME element, which takes I_1 and I_2 as inputs, and produces O_1 and O_2 as outputs. Each separating node except V_{DD} and GND is shown by a circle, and junction nodes are indicated by squares. Note that there is no gate node in Figure 3.8.

It can be found that most nonseparable components in Figure 3.8 have only one or two transistors. Although it is not always the case, it will be shown in the next section that most nonseparable components are series-parallel, and therefore the partitioning scheme is very efficient. Assume that there are n nonseparable components ($n - 1$ separating vertices) along the path from a junction node (or a gate node) z to one of its signal sources s . If the logical circuit expression E_i is found for each individual nonseparable component C_i , the resulting expression for the logical path P_{z-s} is the AND operation of all the logical circuit expressions. That is,

$$P_{z-s} = \prod_{i=1}^n E_i.$$

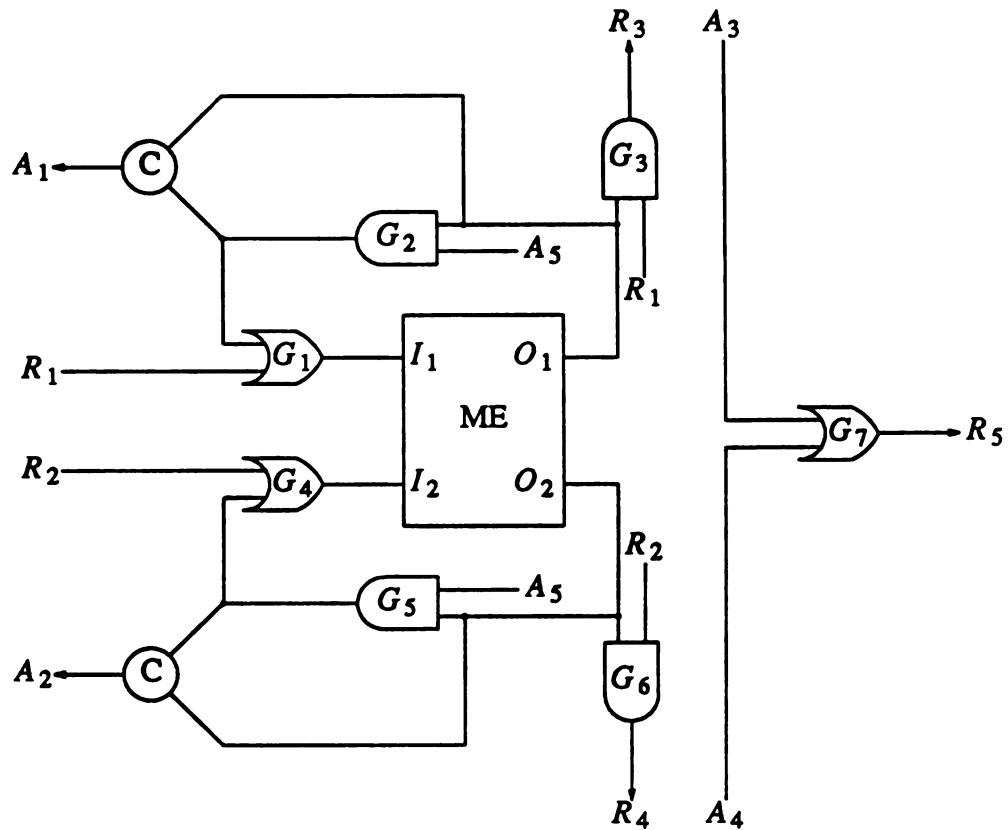


Figure 3.7. A two-input asynchronous arbiter design

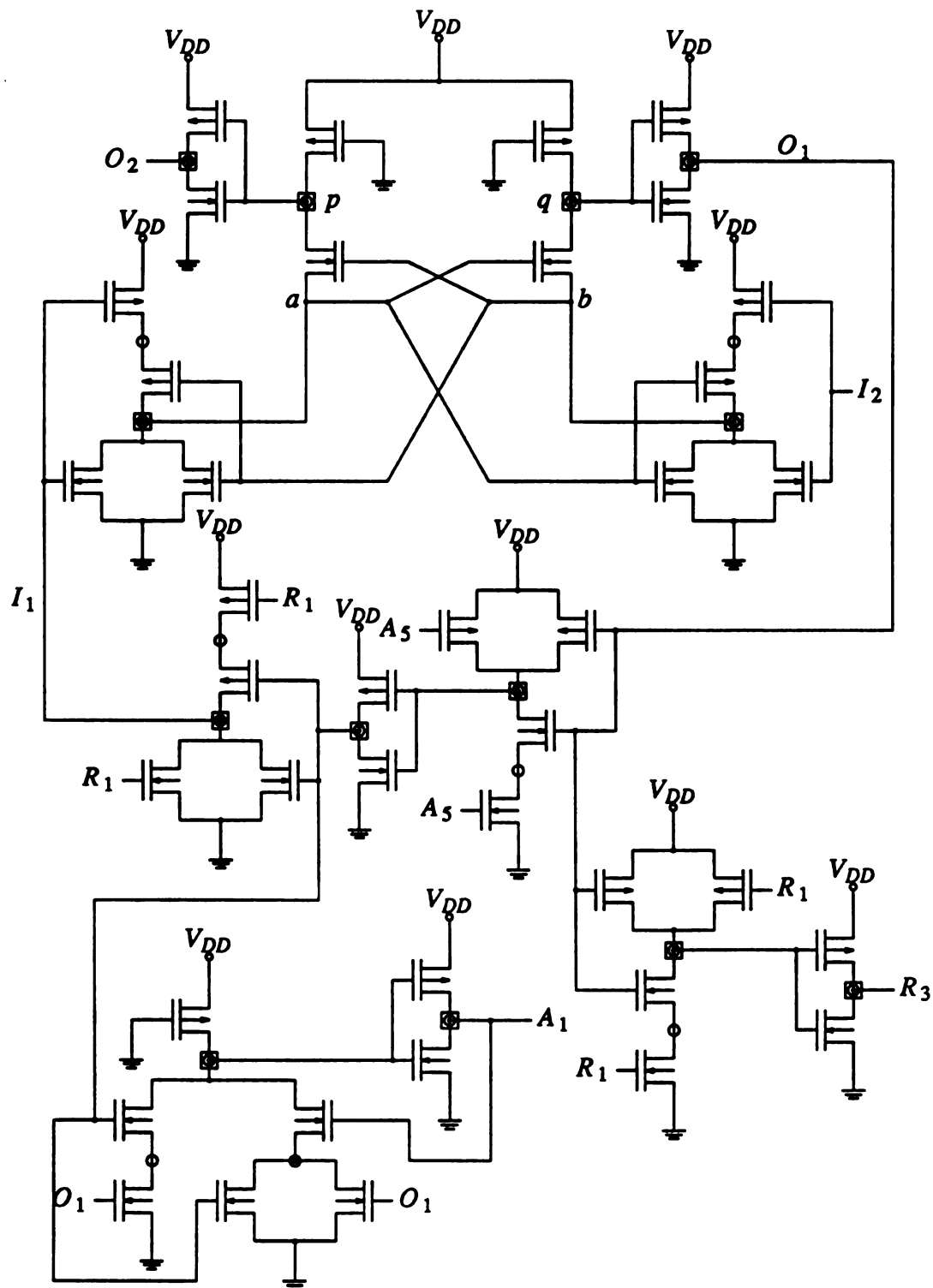


Figure 3.8. Separating nodes for nonseparable components

3.4.2. Nonseparable Components

After the separating nodes are all identified, each nonseparable component is typically small. The reason for small nonseparable components is due to the fact that circuits are different from arbitrary graphs. Circuit structures are limited by many factors, such as area, speed, charge sharing problems, and body effects. It is these constraints which make circuit structure different from arbitrary graphs. For those components which have fewer than five edges, the network connectivity must be series-parallel. In addition, there are only ten possible structures for a two-terminal (refer to the starting node and terminating node) nonseparable component with fewer than five edges. The ten structures include the trivial case with a single edge. Figure 3.9 shows the ten possible structures, in which s stands for starting node and t for terminating node.

If each nonseparable component in Figure 3.9 is checked for parallel structures with respect to the starting node and terminating node, all nonseparable components will eventually become the trivial case with only one single edge. In fact, this is true for a series-parallel structure with any number of edges, as long as the edges connected in parallel can be separated. Thus, a nonseparable component with five elements can be compared with the Wheatstone bridge for checking series-parallel structure, and larger components with six or more elements may be separated by detecting parallel structures in linear time complexity.

Although most switching circuits presented in Chapter 2 and Chapter 3 have series-parallel structures, there are nonseparable components whose structures are not series-parallel. In order to derive logical circuit expressions from a non-series-parallel nonseparable component, a definition of *equivalence* between two n -terminal networks is needed.

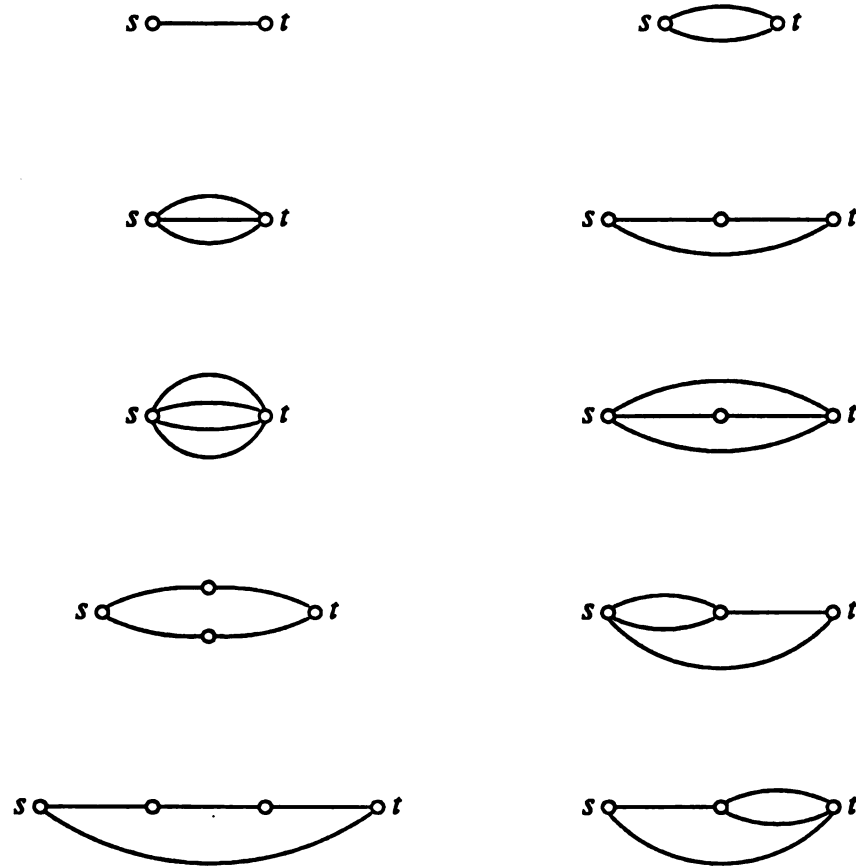


Figure 3.9. Possible structures for a 2-terminal nonseparable component with less than 5 elements

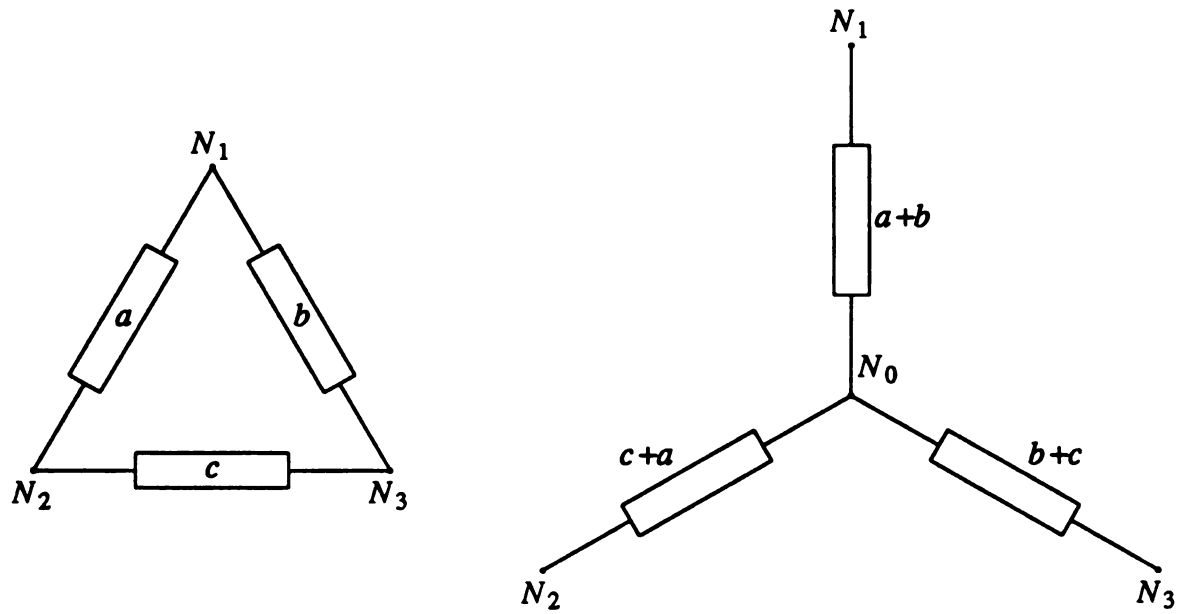
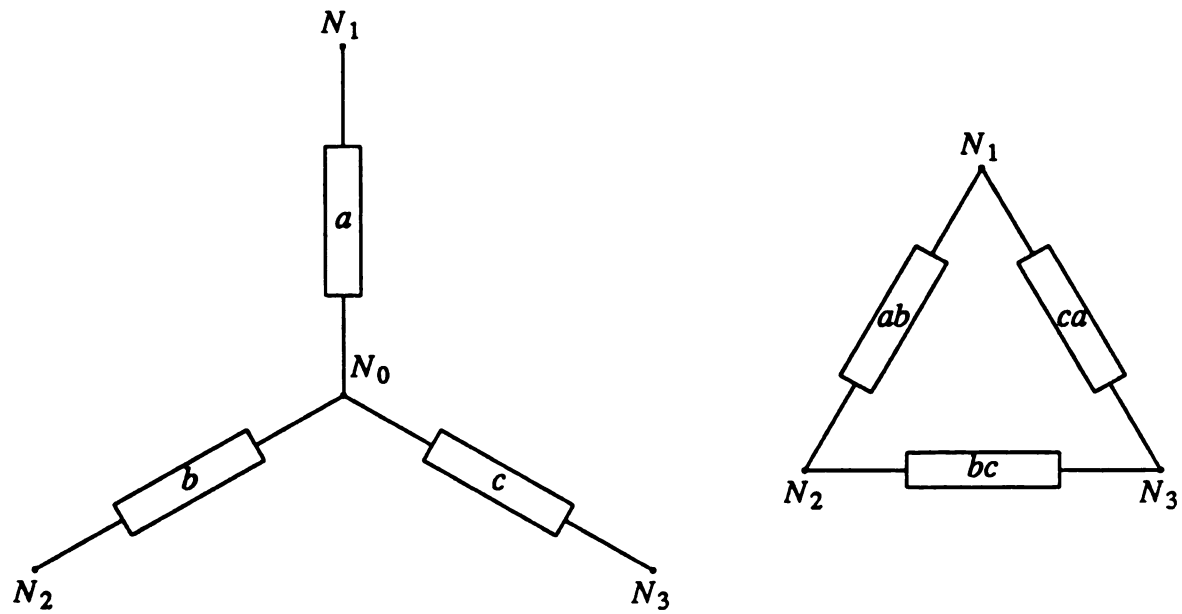
For two n -terminal networks, the equivalence with respect to those n terminals is established if and only if the logical path between each pair of terminals is preserved. As in ordinary impedance theory, there exist delta-to-wye (Δ -to-Y) and wye-to-delta (Y-to- Δ) transformations; similar transformations exist for generating logical circuit expressions. The Δ -to-Y and Y-to- Δ transformations are redefined and depicted in Figure 3.10 and Figure 3.11, respectively.

The two networks in Figure 3.10 are equivalent with respect to the three terminals N_1, N_2 , and N_3 , since we have $a + bc = (a + b)(c + a)$ between N_1 and N_2 , and similarity for the other pairs of terminals N_2-N_3 and N_3-N_1 . Similarly the two networks in Figure 3.11 are equivalent with respect to N_1, N_2 , and N_3 . This follows from the fact that $ab = ab + abc$, etc.

An n -point star also has a mesh equivalent with the central junction eliminated. Shannon [Shan38] has shown that there always exists a equivalent mesh network for a star network. Thus, one can always apply the transformations until the network is of the series-parallel type. To apply this to the Wheatstone bridge of Figure 3.1, first the node A_2 may be eliminated by applying the Y-to- Δ transformation to the star $s-A_2, A_1-A_2, t-A_2$. This gives the network of Figure 3.12. The logical path from s to t is then given by

$$\begin{aligned} P_{s \rightarrow t} &= (a + bc)(d + ce) + be \\ &= ad + be + ace + bcd. \end{aligned}$$

Let S be a subset of vertices with the starting node $s \in S$ and the terminating node $t \in \bar{S} = V - S$. Then the set of edges connecting vertices of S with \bar{S} is called the *cut* defined by S . Logical circuit expressions of non-series-parallel structures can be also derived from drawing all possible lines (cuts) which would break the circuit between the points under consideration. The expression for the path is written as a product of results

Figure 3.10. Delta-to-wye (Δ -to-Y) transformationFigure 3.11. Wye-to-delta (Y-to- Δ) transformation

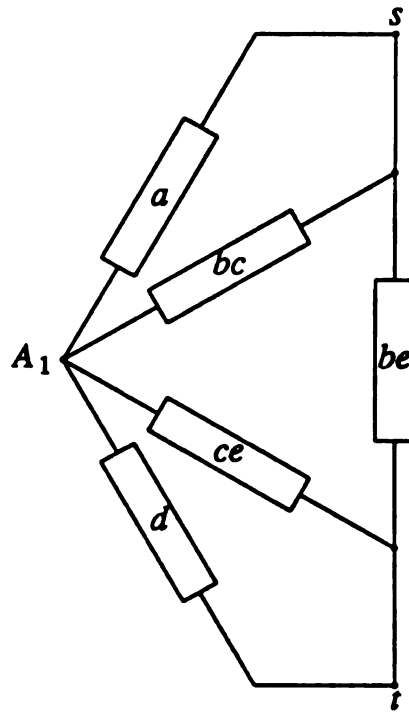


Figure 3.12. Equivalent network by means of transformations

obtained from those cuts, each of which is a sum. By applying this method to the Wheatstone bridge in Figure 3.1, we have

$$\begin{aligned}
 s &= (a + b) (d + e) (a + c + e) (b + c + d) \\
 &= (ad + be + ae + bd) (a + c + e) (b + c + d) \\
 &= (ad + ae + bcd + be) (b + c + d) \\
 &= ad + be + ace + bcd
 \end{aligned}$$

which is the same as the result obtained by using transformations.

CHAPTER IV

FUNCTIONAL RECOGNITION OF STATIC CMOS CIRCUITS

Functional recognition is an important step toward symbolic circuit verification. In this chapter, a new approach for logic component recognition is proposed. A number of recognition rules are developed for symbolic verification of CMOS logic circuits. Based on functional expansion and logical circuit expressions, this approach can be applied to various CMOS logic structures to verify logic functions.

To shorten the design cycle and to decrease design costs, it is crucial to eliminate as many errors as possible before manufacturing an integrated circuit. There are many chances for errors to occur in the design of a VLSI chip. The reasons why a particular design may not work are numerous. They range from very low-level problems, such as short circuits in the design, to high-level bugs in algorithms. Thus, many approaches proposed in the past have been used at various levels to detect design errors. For example, design rule checking is usually conducted in the early stage of the design process. Physical layout rules or design rules specify the legal or illegal relationships among the polygons used in the IC mask-making and fabrication process to implement the final circuit. During the design process of a circuit, different forms of simulation and timing analysis can be used for the verification of the design at various stages.

One of the most common methods of computer-aided verification is simulation [BaTe80, Bray84]. Simulation approaches verify the correct functionality of a logic design only for the particular data simulated. The designer provides the design description in a form that can be simulated, specifies a set of simulation runs, and compares the

simulation results with predicted responses. In practice, a large fraction of the errors in a design can be successfully detected using simulators, either because the user has some intuition about what data patterns are most likely to cause difficulty, because the error causes improper behavior for many different patterns, or because the user simulated a large number of patterns. Even so, the user is left with the difficult task of choosing data that will detect every design error and deciding when a sufficient set of tests have been applied. The complete verification of a design by simulation requires exhaustive simulation covering all possible combinations of states and inputs. In many cases, errors remain undetected until well into the design process or even after the circuit has been manufactured. Therefore, formal verification techniques which are independent of input patterns are required in order to raise the level of confidence for the correctness of the design and to save time and memory space.

Verification of a hardware design is the process by which a logic design is shown to be consistent with a functional or behavioral specification of the design [Roth77, CoVa80]. To verify the correctness of a digital circuit in a more rigorous fashion, we need to recognize logic functions of individual circuit components so that verification or comparison can be conducted in a higher level. For circuits described in terms of logic gate networks, the verification process at the logic level is conceptually straightforward, since the functions computed by a network is given by the composition of the functions computed by the gates. Our interest, however, is in verifying CMOS circuits represented at the switch level as networks of nodes connected by resistive transistor switches. This is done by obtaining circuit behavior in terms of Boolean functions. By doing so, we are able to recognize logic components in a circuit, and therefore comparison of CMOS circuits can be conducted at the Boolean logic level.

4.1. Previous Work

Circuit verification at the algorithmic level has attracted much attention since software verification techniques such as inductive assertions [Floy67] and temporal logic [ReUr71, Boch82] were proposed. Many researchers [CaJB79, Darr79, CoVa80, PiSt83, SuFr86] have tried to verify the correctness of microcode in microprogrammed machines, and computation components such as a multiplier, either by generating arithmetic equations from a given specification or by comparing two specifications such as state transition diagrams. This architectural description of the target component or machine is typically independent of the underlying technology. Therefore the verification is conducted based on the assumptions that all basic components and/or specifications are functionally correct.

Logic component recognition is a crucial step toward symbolic circuit verification. Several approaches to logic component recognition for verifying functional correctness in hardware designs have been proposed. A logic gate recognition technique for fully complementary CMOS circuits using series parallel reduction is proposed in [TaMC82]. A circuit is first partitioned into a number of disjoint subcircuits using so-called *direct current path* (or d.c. path for short) techniques. Each subcircuit is a *d.c. path block*, which is defined as a collection of devices connected within the user-specified boundary nodes. In general, V_{DD} and GND are the boundary nodes, and therefore a pass transistor at the output node of a gate is part of a path rather than being a path itself.

Using this approach, the transistors in each subcircuit are classified into a driver/pulldown part, a load/pullup part, and a transmission part. Then, series parallel connected transistors are replaced by a newly generated transistor, and trees are constructed for both p-type and n-type transistor networks. In CMOS circuits, both trees are examined and the primitive logic gates are recognized if the trees are complementary to

each other. The complementary check is performed by examining whether or not the pair of trees are isomorphic under the conditions that (1) corresponding parent vertices belong to the opposite category (series versus parallel) and (2) corresponding leaf vertices have the same labels. These constraints imply that transistor networks have to be series-parallel in structure in order to be recognizable, and only fully complementary CMOS gates whose p-type and n-type transistor networks are complementary to each other can be identified. It is therefore too limited to apply for recognizing various CMOS circuits.

Symbolic simulation is another possible alternative for conventional simulation. In conventional simulation, signals are assumed to have some initial values. The simulation proceeds in a deterministic fashion, and the signals have known values at most times, if not all times, during the simulation. In symbolic simulation, by contrast, no assumptions are made about initial values. The initial values of signals are represented symbolically. They could be Boolean variables as well as the constants 0 and 1. The advantage of symbolic simulation is that it fully tests a system, while conventional simulation tests only specific cases. On the other hand, symbolic simulation is difficult due to the need for symbolic formula manipulation and logical deduction capable of establishing desirable equations from original descriptions.

Bryant [Brya85], while advocating the use of Boolean algebra for symbolic analysis, proposed a verification technique for symbolic simulation at the switch level using path relations and Boolean matrices. A switch-level network consists of a set of nodes and a set of transistors. Each storage node is assigned an integer size from the set $\{ 1, 2, \dots, k \}$ in a highly simplified way to indicate its capacitance relative to nodes with which it may share charge. Each transistor has a *type* indicating the conditions under which it will become conducting and a *strength* indicating its conductance relative to other transistors. Transistor conductances are modeled by assigning each transistor an integer strength from the set $\{ k+1, k+2, \dots, w-1 \}$, where w is the strength of input

nodes. A *rooted path* p is a directed path in a switch graph originating at vertex $Root(p)$, terminating at vertex $Destination(p)$ and consisting of a possibly empty set of edges $Edges(p)$, in which an edge is actually a transistor. A rooted path represents a source of charge from its root to its destination with driving power indicated by its strength, which is determined by the weakest transistor or node along the path. In general, the steady state response of a node depends on a subset of paths to the corresponding vertex in the switch graph, namely those which are not blocked. A *definite* path is defined as a rooted path p such that no edge in $Edges(p)$ corresponds to a transistor in the open state. In addition, a path p is said to be *blocked* if for some initial segment p' of p and for some definite path q , $Destination(p') = Destination(q)$ and the strength of p' is less than that of q .

Suppose we wish to compute the steady state values of a component containing nodes $\{ n_1, n_2, \dots, n_q \}$ and transistors collected into sets of the form $T_s(i,j)$ consisting of those transistors of strength greater than or equal to s having source and drain nodes n_i and n_j . To account for the different strength levels, this Boolean matrix approach solves three Boolean equations for each node n for each strength s : $steady1_s(n)$ (respectively $steady0_s(n)$) indicating the conditions under which the node is driven to 1 (respectively 0) by a path of strength greater than or equal to s with no blocking path of strength greater than s , and $block_s(n)$ indicating the conditions under which the node will be the destination of a definite path of strength greater than or equal to s . The *path relation* P is defined as $m P n$ if there is an unblocked path p in the switch graph with $Root(p)$ corresponding to node m and with $Destination(p)$ corresponding to node n . Then the steady state response on node n is given by the equation

$$steady(n) = l.u.b. \{ state(m) \mid m P n \}$$

where *l.u.b.* represents the least upper bound over the ordering $0 < X$ and $1 < X$, and $0, 1$,

X are three possible states for each node. The steady state response for a component in such a network is then expressed in terms of Boolean matrix and operations for all internal nodes.

This verification approach tries to capture circuit behavior for a given time instant, since the steady state response of a node depends upon a subset of the paths to the corresponding vertex in the switch graph rather than all possible paths. In addition, the assignment of storage sizing and transistor strength is not only very difficult but also lacks accuracy. However, it is claimed that the same effect as the switch-level simulator MOSSIM II [Brya84] does is achieved by solving three equations in an algebra of path strengths for each node.

4.2. Functional Expansion and Recognition Rules

In this section, a new method for logic component recognition of CMOS circuits is proposed. This approach can be applied to various CMOS logic families, such as fully complementary CMOS, pseudo-nMOS, pass transistor logic, and DCVS (Differential Cascode Voltage Switch) structures. In addition to simple logic gates, storage elements and circuit components using tightly-coupled structures can also be recognized.

To recognize various CMOS logic circuits, the following rules are used to extend logical circuit expressions for further applications. Justifications for the rules are also provided.

Rule 1:

If a node $z = [a]b \bullet$, does not have high-impedance states and signal conflicts for any its input combination, then it is equivalent to $z = [a\bar{b}]0 \bullet [\bar{a}b]1$, i.e.,

$$R1: \quad z = [a]b \bullet = [a\bar{b}]0 \bullet [\bar{a}b]1.$$

Justification: If a is zero, the n-type transistor is disconnected, and the logic value of z depends on other simple paths. In this case both the paths $[a\bar{b}]$ from z to V_{SS} (0) and $[\bar{a}b]$ to V_{DD} (1) are open, thus the logic value of z is independent of b . If a is one, the n-type transistor is conducting, and the logic value of z depends on the value of b . In this case z is rewritten as $z = [\bar{b}]0 \cdot [b]1$, which shows $z = b$.

Rule 2:

If a node $z = \cdot [a]b$, $a \neq b$, does not have high-impedance states and signal conflicts for any its input combination, then it is equivalent to $z = [\bar{a}b]0 \cdot [ab]1$; i.e.,

$$\text{R2: } z = \cdot [a]b = [\bar{a}b]0 \cdot [ab]1.$$

Justification: If a is one, the p-type transistor is disconnected, and the logic value of z depends on other simple paths. In this case both the paths $[\bar{a}b]$ from z to V_{SS} (0) and $[ab]$ from z to V_{DD} (1) are open, which makes z independent from b . If a is zero, the p-type transistor is conducting, and the logic value of z depends on b . In this case z is rewritten as $z = [\bar{a}b]0 \cdot [ab]1 = [\bar{b}]0 \cdot [b]1$, which is equivalent to $z = b$.

If the gate and source signals of a transistor are the same or complementary to each other, results from Rule 1 and Rule 2 can be further simplified. For example, in the case of $a = b$, we have

$$\begin{aligned} z &= [a]b \cdot \\ &= [ab]0 \cdot [\bar{a}b]1 \\ &= \cdot [\bar{a}]1 \end{aligned}$$

for n-type transistors, and

$$\begin{aligned}
z &= \bullet [a]b \\
&= [\bar{a}b]0 \bullet [ab]1 \\
&= [\bar{a}]0 \bullet
\end{aligned}$$

for p-type transistors. Similarly, in the case of $a = \bar{b}$, we have

$$\begin{aligned}
z &= [a]b \bullet \\
&= [ab]0 \bullet [\bar{a}b]1 \\
&= [a]0 \bullet
\end{aligned}$$

for n-type transistors, and

$$\begin{aligned}
z &= \bullet [a]b \\
&= [\bar{a}b]0 \bullet [ab]1 \\
&= \bullet [a]1
\end{aligned}$$

for p-type transistors.

By using these two rules, one can always perform functional expansion for a given logical circuit expression to see if the circuit component is recognizable. In order to generalize the recognition procedures, one needs to exploit pseudo-nMOS and fully complementary CMOS logic structures.

Pseudo-nMOS logic plays an important role in many CMOS designs such as PLA implementation for finite state machines in order to minimize body effects and nodal capacitances due to long interconnections. This CMOS logic structure introduces a *pullup node* [Brya81], which is defined as a node connected via a pullup transistor (a depletion-mode n-type transistor in nMOS technology or a grounded p-type transistor in pseudo-nMOS logic) to V_{DD} . A pullup node is never trapped into a high-impedance state. It is considered to have a logic value one unless grounded. Thus, pseudo-nMOS circuits do not have high-impedance states as is found in other CMOS logic structures, and the function of a pseudo-nMOS logic gate depends upon its n-type network. The fol-

lowing rules are used to formally indicate properties of pseudo-nMOS logic, and the state of a pullup node can be expressed in terms of the associated rules:

Rule 3:

If there is a grounded p-type transistor which connects a node z to the node V_{DD} , other p-type paths to the same source node (V_{DD}) is of no use in terms of logic functions; i.e.,

R3: $z = \bullet [0 + P_1 + P_2 + \cdots + P_n]1 = \bullet [0]1$ where P_i is a simple path reachable from z to V_{DD} .

Justification: As far as the logic function is concerned, one conducting path will do the same thing as two or more conducting paths emitting from the same signal source. Thus Rule 3 is true in the sense that logic value ONE is the same as logic value ONE plus logic value ONE.

Similarly, a conducting n-type transistor which connects a node z to the node V_{SS} has the same effect as two or more conducting paths. Therefore we have the similar rule:

Rule 4:

If there is a conducting n-type transistor which connects a node z to the node V_{SS} , other n-type paths to the same source node (V_{SS}) is of no use in terms of logic functions; i.e.,

R4: $z = [1 + N_1 + N_2 + \cdots + N_n]0 \bullet = [1]0 \bullet$ where N_i is a simple path reachable from z to V_{SS} .

The nice thing about logical circuit expressions is that they are “compatible” with Boolean expressions: the expressions for n-type paths have the same properties as Boolean expressions, and those for p-type paths also have the same properties except that each variable is evaluated by its complement. The other important property of pseudo-

nMOS logic is that the function of a pseudo-nMOS logic gate depends upon its n-type network. This property is expressed in Rule 5.

Rule 5:

If a node z has a conducting p-type path to V_{DD} , then a Boolean expression can be directly obtained from its n-type paths. The Boolean function is actually the negation of the expression for its n-type block; i.e.

R5: $z = [f_0]0 \cdot [0]1 = \overline{f_0}$ where f_0 is the sum of expressions for all n-type paths from z to V_{SS} .

Justification: If f_0 is 0 (logic value ZERO), which implies that there is no conducting path from the node z to the node V_{SS} , then the voltage at the node z will be at the high-voltage range (logic value ONE) due to the grounded p-type transistor. On the other hand, if f_0 is 1 (logic value ONE), which implies that there is a conducting path from node z to V_{SS} , then the voltage at the node z will be pulled down to the low-voltage range (logic value ZERO). Thus, the logic function at node z depends upon the n-type paths and it is the negation of the expressions for the n-type paths.

Fully complementary CMOS structures have many advantages, such as low power consumption and high noise margins, over other CMOS logic families. The “logical redundancy” property [MyIv85] of fully complementary CMOS circuits can be applied to recognize logic functions after the logical circuit expression of each circuit component is expanded.

Rule 6:

The *complement expression* f' of a Boolean expression f is defined as an expression which is the same as f except that all logic variables and constants in f are substituted by their complements in f' . Let f_0 and f_1 be the

expressions for the n-type and p-type paths of a logical circuit expression, respectively ($z = [f_0]0 \cdot [f_1]1$). If $\overline{f_0} = f_1'$, then the function at the output node z is given by $z = \overline{f_0} = f_1'$; i.e.,

$$\text{R6: } z = [f_0]0 \cdot [f_1]1 = \overline{f_0} \quad \text{if } \overline{f_0} = f_1'$$

Justification: Since a p-type transistor is conducting when its gate is connected to a logic signal ZERO, the expressions for p-type paths are evaluated in a way similar to Boolean expressions but using complement variables and constants. The Boolean expression f_1' actually specifies the close/open (connected/disconnected) state of the p-type paths, while the other expression f_0 specifies the close/open state of the n-type paths. Thus, the necessary condition $\overline{f_0} = f_1'$ implies that the p-type and n-type paths reach their close states in a mutual-exclusive fashion: for each conducting n-type path the same input pattern will disconnect all possible p-type paths, and for each conducting p-type path the same input pattern will disconnect all possible n-type paths. Therefore the function of z is completed without signal conflicts.

There exist cases in which logic gates are tightly-coupled and none can be recognized as a logic primitive using previous rules from a single logical circuit expression. This situation arises especially in DCVS logic structures. By converting all p-type transistors of a fully complementary CMOS gate to n-type transistors with complementary gate signals and adding two cross-coupled p-type transistors, it is always possible to obtain both true and complementary values of the original logic function. Sometimes the number of transistors and/or the cell area can be reduced using this technique. Based on logical circuit expressions, this design technique is described below.

Rule 7:

Given $f = \bar{g}$ and a cross-coupled structure with logical circuit expressions $x = [f]0 \cdot [y]1$ and $y = [g]0 \cdot [x]1$, the expressions of x and y are functionally equivalent to two separate logical circuit expressions given by $x = [f]0 \cdot [f]1 = \bar{f}$ and $y = [g]0 \cdot [g]1 = \bar{g}$; i.e.,

R7: $x = [f]0 \cdot [y]1 = [f]0 \cdot [f]1 = \bar{f}$ and $y = [g]0 \cdot [x]1 = [g]0 \cdot [g]1 = \bar{g}$ provided that $f = \bar{g}$

Justification: The expression $x = [f]0 \cdot [y]1$ indicates that the output node x has logic value ONE if both y and f are ZERO, and has logic value ZERO when the Boolean expression f is ONE. By examining the signal sources of the expression $y = [g]0 \cdot [x]1$, we know that y will have logic value ZERO when the Boolean expression g is ONE. This implies that x may have logic value ONE if g is ONE. In this case, if f is ZERO, then x is ONE. From $f = \bar{g}$, it is clear that x will have logic value ONE if f is ZERO. Thus, we have $x = [f]0 \cdot [y]1 = [f]0 \cdot [f]1 = \bar{f}$. The same argument is applied for $y = [g]0 \cdot [x]1 = [g]0 \cdot [g]1 = \bar{g}$.

Both expressions for x and y are in the form of pseudo-nMOS logic. In the case of $f \neq \bar{g}$, the voltage at node x and node y might be in the low-level voltage range at the same time but never in the high-level voltage range at the same time, since a p-type transistor is ON when its gate voltage is 0 and a pullup node is considered to have a logic value ONE unless grounded.

Rules derived in this section are generally applicable, rather than being useful only for certain kind of logic structure like the approach given in [TaMC82]. This approach based on logical circuit expressions is also much simpler than approaches such as the Boolean matrix method mentioned in the previous section. A number of examples given in the next section are used to illustrate the usefulness of these rules.

4.3. Static Logic Component Recognition

To illustrate these techniques for static CMOS circuits, consider the logical circuit expressions of the circuits shown in Figure 4.1 and Figure 4.2. By applying R1, R3, and R5, the function of the logic circuit in Figure 4.1 can be verified through a sequence of steps:

$$\begin{aligned}
 z &= [b]a + [a]b \cdot [0]1 \\
 &= [b\bar{a} + a\bar{b}]0 \cdot [0 + \bar{b}\bar{a} + \bar{a}\bar{b}]1 \\
 &= [b\bar{a} + a\bar{b}]0 \cdot [0]1 \\
 &= \overline{b\bar{a} + a\bar{b}}.
 \end{aligned}$$

The mutual exclusion element depicted in Figure 4.2 contains two logical circuit expressions for nodes p and q , respectively. Since the circuit has a cross-coupled structure with two NOR gates (the same as a basic RS flip-flop), the outputs A_1 and A_2 cannot be expressed in terms of R_1 and R_2 . Instead, the rest of the circuit is verified using logical circuit expressions:

$$\begin{aligned}
 p &= [b]a \cdot [0]1 \\
 &= [b\bar{a}]0 \cdot [0 + \bar{b}\bar{a}]1 \\
 &= [b\bar{a}]0 \cdot [0]1 \\
 &= \overline{b\bar{a}} = a + \bar{b}
 \end{aligned}$$

and

$$\begin{aligned}
 q &= [a]b \cdot [0]1 \\
 &= [a\bar{b}]0 \cdot [0 + \bar{a}\bar{b}]1 \\
 &= [a\bar{b}]0 \cdot [0]1 \\
 &= \overline{a\bar{b}} = b + \bar{a}.
 \end{aligned}$$

Since node a and node b will never be one at the same time, node p and node q will not

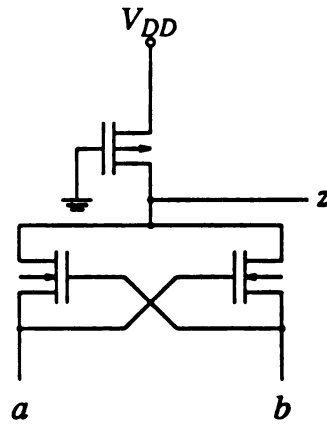


Figure 4.1. A pseudo-nMOS XNOR circuit

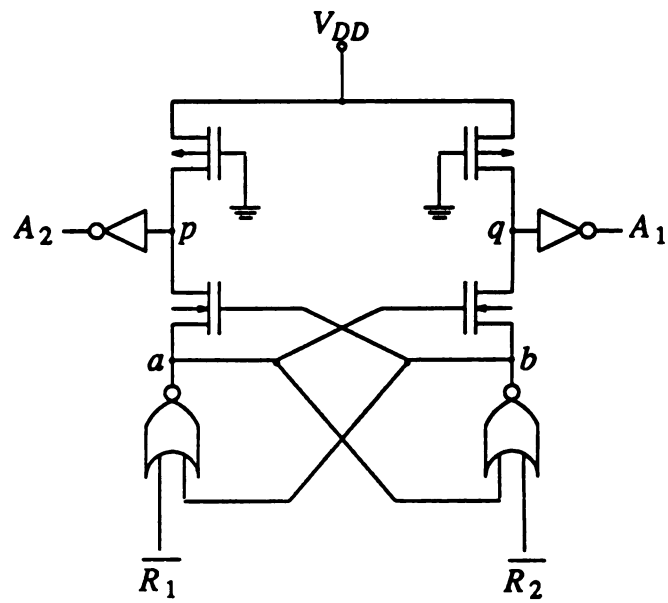


Figure 4.2. A mutual exclusion element

be zero simultaneously. This in turn ensures that node A_1 and node A_2 will not rise to high at the same time; thus, the mutual exclusion operation of the circuit is achieved.

Pass transistor logic is commonly used in many CMOS designs. Transmission gates, for example, are popular in that they maintain good logic levels while reducing chip area. Two CMOS circuits containing transmission gates have been depicted in Figure 2.9. The circuit in Figure 2.9(a) is a XOR gate consisting of a transmission gate and an inverter structure. This circuit is used to perform the exclusive-OR function when both true and complementary values of one variable are available. The other circuit in Figure 2.9(b) is a two-input multiplexer consisting of two transmission gates.

To validate the circuits in Figure 2.9, they are specified in terms of logical circuit expressions, and then the n-type and p-type paths are manipulated using general Boolean algebra and rules derived in Section 4.2. Node f is verified by:

$$\begin{aligned}
 f &= [b]\bar{a} + [\bar{a}]b \cdot [b]a + [a]b \\
 &= [ba + \bar{a}\bar{b} + \bar{b}\bar{a} + \bar{a}\bar{b}]0 \cdot [b\bar{a} + a\bar{b} + \bar{b}a + a\bar{b}]1 \\
 &= [ba + \bar{a}\bar{b}]0 \cdot [b\bar{a} + a\bar{b}]1 \\
 &= \overline{ba + \bar{a}\bar{b}}.
 \end{aligned}$$

Node g can also be verified through a sequence of steps:

$$\begin{aligned}
 g &= [\bar{c}]a + [c]b \cdot [c]a + [\bar{c}]b \\
 &= [\bar{c}\bar{a} + c\bar{b} + \bar{c}\bar{a} + c\bar{b}]0 \cdot [c\bar{a} + \bar{c}\bar{b} + c\bar{a} + \bar{c}\bar{b}]1 \\
 &= [\bar{c}\bar{a} + c\bar{b}]0 \cdot [c\bar{a} + \bar{c}\bar{b}]1 \\
 &= \overline{\bar{c}\bar{a} + c\bar{b}}.
 \end{aligned}$$

Two tightly-coupled two-input static XOR/XNOR gates using CMOS DCVS logic structures are depicted in Figure 4.3. In fact, the circuit in Figure 4.3(b) is a simplified version of the circuit in Figure 4.3(a) such that the number of transistors is reduced. These circuits offers both the true and complementary values, in a way similar to that a

flip-flop does.

These two circuits have the same static function, and their logical circuit expressions are also the same. On the other hand, their circuit connectivities are different. Therefore they have different structural circuit expressions. Note that the circuit in Figure 4.3(b) has two auxiliary nodes, but the one in Figure 4.3(a) with series-parallel n-type networks does not have any auxiliary nodes.

To recognize the function of these two circuits, one first checks if $ba + \overline{b}\overline{a}$ is the same as $\overline{b\overline{a}} + \overline{\overline{b}a}$. Since they are the same, rule R7 is applicable:

$$x = [ba + \overline{b}\overline{a}]0 \cdot [y]1 = \overline{b\overline{a}} + \overline{\overline{b}a}$$

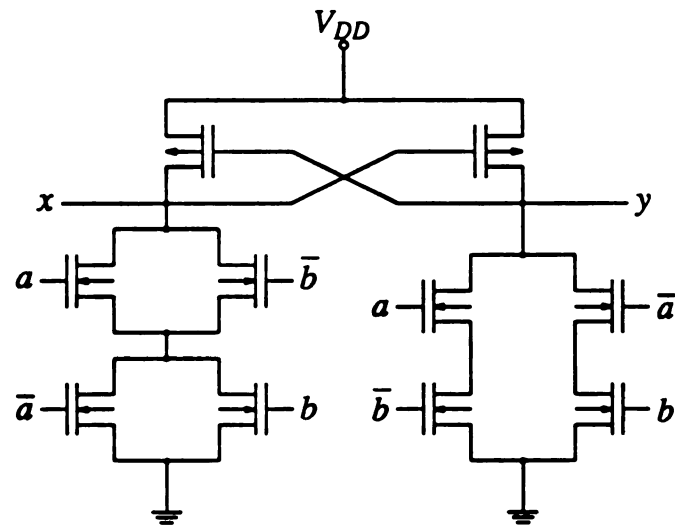
$$y = [\overline{b\overline{a}} + \overline{\overline{b}a}]0 \cdot [x]1 = \overline{b\overline{a}} + \overline{\overline{b}a}.$$

which validates the functions of x and y .

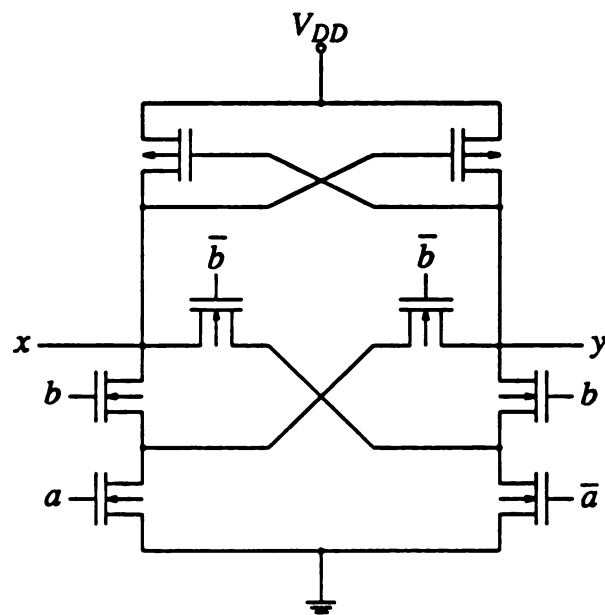
4.4. Recognition of Storage Cells

Static storage cells such as flip-flops, buffers, and memory elements are partitioned into two subcircuits during the circuit partitioning step. Each subcircuit forms a logical circuit expression and takes the output of the other as one of its inputs. Thus storage cell recognition is achieved by finding nested logical circuit expressions in consecutive stages.

Figure 4.4 shows a two-input Muller-C element using pseudo-nMOS logic structure. The Muller-C element found in many self-timed systems is a binary storage device. Its output becomes ONE only after all of its inputs are ONE, and becomes ZERO only after all of its inputs are ZERO. The logical circuit expressions for the two subcircuits of the Muller-C element are



(a)



(b)

Figure 4.3. Two 2-input DCVS XOR/XNOR gates with the same logical circuit

expressions $x = [ba + \bar{b}\bar{a}]0 \cdot [y]1$ and $y = [b\bar{a} + \bar{b}a]0 \cdot [x]1$

$$p = [ab + q(a + b)]0 \cdot [0]1, \text{ and}$$

$$q = [p]0 \cdot [0]1,$$

respectively. It can be seen that the junction node p is taken as an input in the logical circuit expression of node q and vice versa. If the logic function of each subcircuit can be recognized, the storage cell function can also be recognized. In this example, we have $p = \overline{ab + q(a + b)}$ and $q = \overline{p}$. By replacing p by its logic function and making q a function of time, we have $q_{n+1} = ab + q_n(a + b)$.

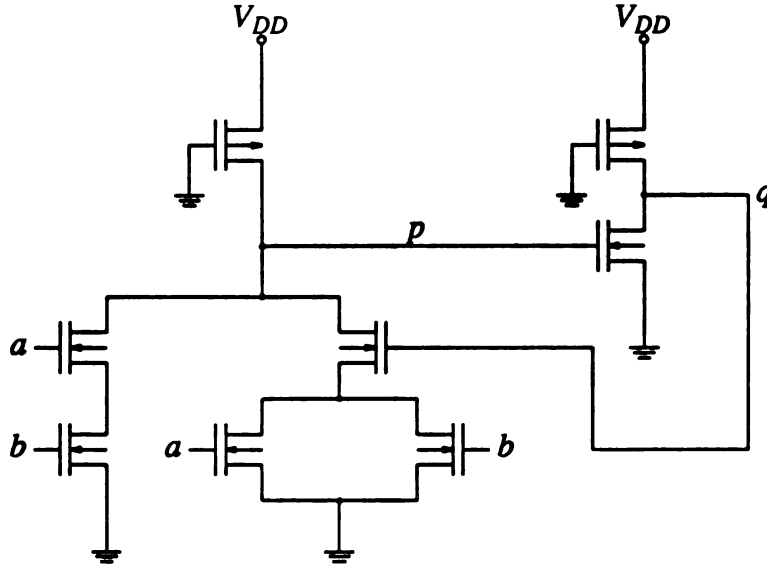


Figure 4.4. A pseudo-nMOS Muller-C element

D flip-flops are commonly used as storage elements in many static CMOS circuits. The transistor structure of a D flip-flop is depicted in Figure 4.5.

Once the logic function of each subcircuit is recognized (in Figure 4.5, $q = \overline{\phi d + p}$ and $p = \overline{\phi d + q}$), we can recognize the function of the binary storage cell by substituting one variable and making the other a function of time. Thus, for the D flip-flop in Figure

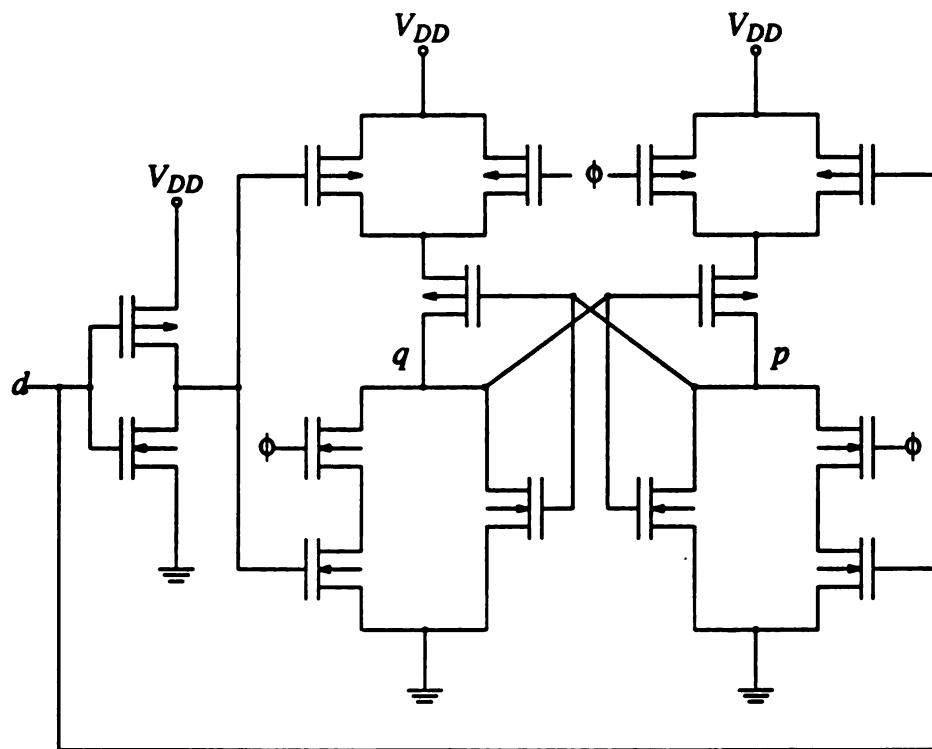


Figure 4.5. Transistor schematics of a static CMOS D flip-flop

4.5, we have

$$q = \overline{\phi \bar{d} + p} = (\bar{\phi} + d) \bar{p},$$

and therefore

$$q_{n+1} = (\bar{\phi} + d) (\phi d + q_n) = \bar{\phi} q_n + d \phi.$$

Based on logical circuit expressions, recognition of many other storage cells which were difficult to recognize before due to tightly-coupled structures becomes possible.

In general, this recognition technique can be applied to the schematic comparison problem described in the previous chapter, and is also useful in the field of reverse engineering. Reverse engineering in the field of VLSI design is a process which takes an existing but unknown circuit artwork as input, and generates a higher-level description, such as a schematic capture. With the power of this recognition technique, the reverse engineering process can be significantly simplified.

CHAPTER V

COMPARISON OF CMOS TRANSISTOR SCHEMATIC NETWORKS

Schematic comparison deals with comparing two circuit schematics, in which one is extracted from the layout, and the other specified by the designer. This chapter shows how logical circuit expressions can be used for CMOS transistor schematic comparison. Traditional graph matching algorithms for schematic comparison are too rigid to match functionally isomorphic schematics, while other approaches such as switch-level simulation suffer from tremendous computation overhead. In this chapter it is shown that the use of logical circuit expressions can help significantly to reduce the difficulty of the comparison process.

Verifying IC layouts against their schematics is a tedious task. Many methods have been developed for verifying the extracted circuit connectivity. One of the common approaches is to simulate the extracted circuit using a switch level simulator and to compare the results with the expected input-output responses [BaTe80, Brya81]. Although no reference connectivity description is needed and no restrictions are imposed on the design methodologies, this method provides little indication for error locations. In addition, this technique becomes impractical when circuits consist of more than a few thousand transistors, especially when many layout errors exist. Thus, circuit comparison between the layout and the specification of the circuit is considered essential for validating layouts created manually or by, for example, a silicon compiler system.

When circuits are verified as they are laid out through circuit comparison, errors may be found early in the design cycle, and thus the impact of layout change is more

easily managed. To date, most existing netlist comparison methods based on graph isomorphism are capable of matching topologically isomorphic circuits [EbZa83, SpNe83]. Hierarchical modeling and randomization techniques are used [TyEl85] to speed up the comparison process, and graph automorphism introduced by swappable terminals is also considered [TyEl85, KoMc86]. However, these capabilities are limited since a good circuit comparison approach should be able to recognize other types of disagreement which do not change the electrical or logical isomorphism of the circuits. Recently, another approach [Shir86] considers functionally isomorphic circuits as well as topologically identical structures. This approach assumes that two functionally isomorphic networks may differ topologically through subcircuit permutation and/or repetition. An algorithm based on graph connectivity is used in which each subcircuit is initially represented by an arc, and permutation/repetition is performed on the graph to match two transistor networks.

The approach for circuit comparison presented in this thesis is different from others in that comparison between an extracted circuit and its specification is based on logical circuit expressions instead of graphs. This representation is able to handle both transistor and gate-level circuit schematics and can help to reduce the difficulty of data transformation between behavior representation and structural representation of CMOS circuits. By using logical circuit expressions which consist of Boolean expressions, the designer can represent a CMOS transistor schematic network in the logical circuit representation. As far as circuit functionality is concerned, logic expressions are independent of circuit structures, design styles and underlying technologies. The Boolean comparison approach developed at IBM [SmBH82] has been recognized as a useful tool for verifying logic designs. Although IBM's approach aims at high-level circuit verification, it demonstrates the usefulness of logic expressions for circuit verification. Since a logic expression imposes no constraints on how a circuit is built, it can not only handle all possible

subcircuit permutations and repetitions, but it can also match circuits with identical functions but different logic structures.

5.1. Functional Isomorphism

The approach at the circuit level is to compare the extracted transistor network with the intended structure. In doing the comparison at this level, a designer should take certain permutations into account so that circuits with different topologies can be recognized to be the same as the original specification. Pins on elements, for instance, can be permuted with no change in the electrical or logical function of the network. Therefore pin permutation is allowed in circuit comparison.

Most existing methods for netlist comparison handle topologically isomorphic netlists based on graph connectivity. For example, Takashima et. al. [TaMC82] use a digraph representation for graph isomorphism testing. All vertices of the two graphs are partitioned into several vertex groups according to the in-degree, out-degree and the type of each vertex. In general, since the graph isomorphism problem is believed to be intractable [Hoff82], it would seem that development of an efficient algorithm for netlist comparison is infeasible. Thus, heuristics based on signature analysis are usually used for graph isomorphism checking [EbZa83, SpNe83].

In addition to the intractable graph isomorphism problem, two functionally isomorphic networks may differ topologically in the following ways:

- (a) subcircuit permutation,
- (b) subcircuit repetition, and
- (c) functional transformation.

First of all, devices connected in series may exchange their positions. Figure 5.1 shows two circuits which differ from each other through subcircuit permutation. The two

pseudo-nMOS gates are functionally identical but topologically different. Permutation might occur at various levels. In this example the input signals a and b are exchanged, and two pieces of subcircuits in series are also permuted.

In addition to subcircuit permutations, different counts of the same subcircuit in two circuit schematics might be encountered due to subcircuit repetition. Figure 5.2 shows two static CMOS NAND gates. Their functions are identical, although they are topologically non-isomorphic.

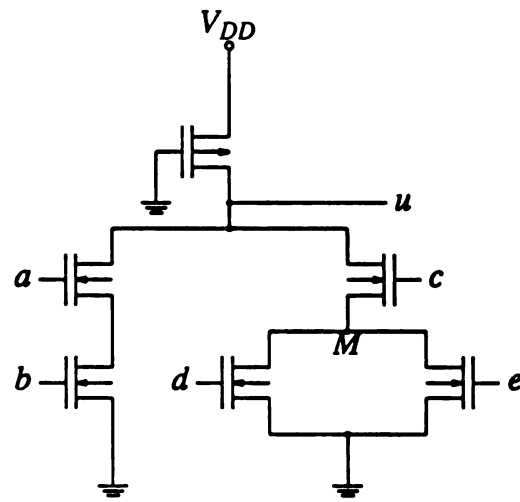
A general case comes from functional permutation, which is a super set of subcircuit permutations/repetitions. In general, it is possible that when translating the original circuit diagram into mask artwork, the designer performs a logically equivalent translation in order to reduce chip area or to improve electrical performance. Figure 5.3 illustrates two pseudo-nMOS XOR circuit schematics. Their transistor connectivities are obviously different, but their functions are the same.

The approach using logical circuit representation decomposes a circuit based on symbolic paths from internal nodes to signal sources. Since paths are grouped together in terms of signal sources, comparison between two circuits is shifted to the logic domain, and Boolean logic comparison is therefore applicable.

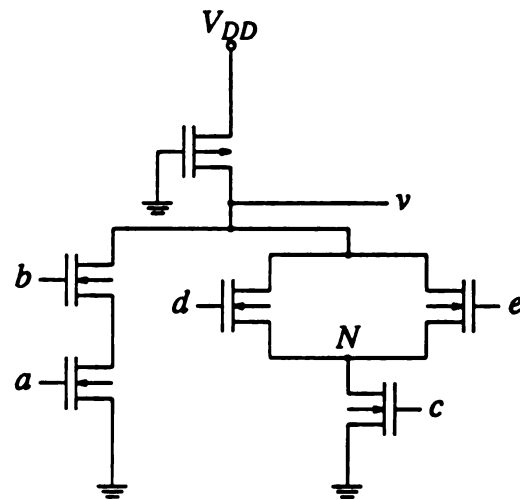
Logical circuit expressions for the schematics shown in Figure 5.1(a) and Figure 5.1(b) are

$$\begin{aligned}
 u &= [ab]0 + [c]M \cdot [0]1 \\
 &= [ab + cd + ce]0 \cdot [0]1, \text{ and} \\
 v &= [ba]0 + [d + e]N \cdot [0]1 \\
 &= [ba + dc + ec]0 \cdot [0]1,
 \end{aligned}$$

respectively, where $M = [d + e]0 \cdot$ and $N = [c]0 \cdot$.

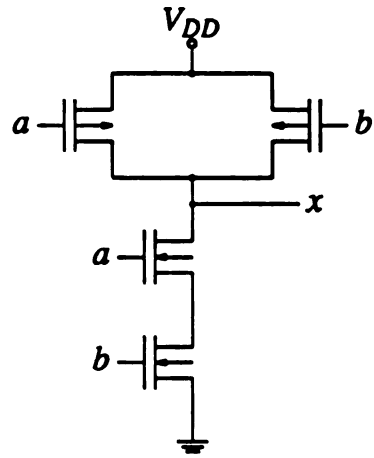


(a)

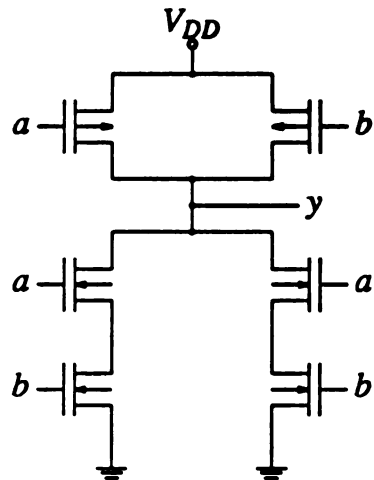


(b)

Figure 5.1. Two pseudo-nMOS gates through subcircuit permutations

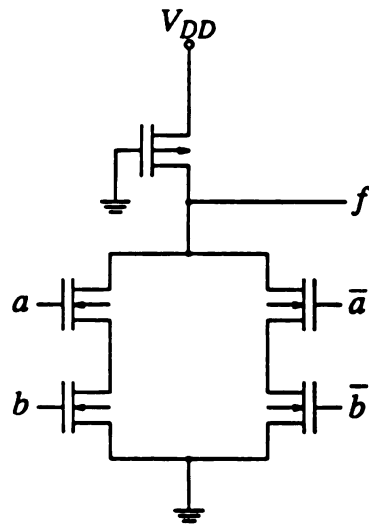


(a)

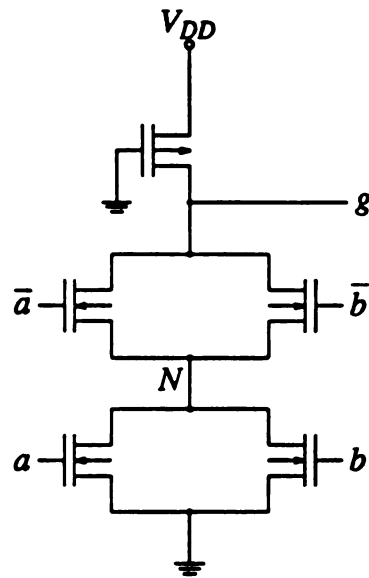


(b)

Figure 5.2. Two static CMOS NAND gates through subcircuit repetition



(a)



(b)

Figure 5.3. Two pseudo-nMOS XOR gates through functional transformation

Similarly, the static CMOS NAND gates in Figure 5.2(a) and Figure 5.2(b) are represented as

$$x = [ab]0 \cdot [a + b]1, \text{ and}$$

$$y = [ab + ab]0 \cdot [a + b]1.$$

After internal separating nodes, such as nodes M and N in Figure 5.1, have been expanded along the logical paths, comparisons between two networks of the same type are performed using logic manipulation. Boolean expressions in logical circuit representations indicate paths from an output or intermediate node to various signal sources. The corresponding expressions in two networks of the same type are then checked for logical equivalence.

The logical circuit expressions of the CMOS XOR gates in Figure 5.3 are

$$f = [ab + \bar{a}\bar{b}]0 \cdot [0]1$$

and

$$\begin{aligned} g &= [\bar{a} + \bar{b}]N \cdot [0]1 \\ &= [(\bar{a} + \bar{b})(a + b)]0 \cdot [0]1, \end{aligned}$$

respectively, where $N = [a + b]0 \cdot$. Since

$$ab + \bar{a}\bar{b} = (\bar{a} + \bar{b})(a + b)$$

we have $f = g$, which matches the outputs in Figure 5.3(a) and Figure 5.3(b). From this example, we know that there are many different ways to construct such a circuit to perform the exclusive-OR operation, and the approach using logical circuit expressions is exactly the right choice for comparing their functions.

In order to speed up the comparison process, one can use hierarchical processing if extra information, such as Signal Correspondence Record lists [SmBH82], is available. Hierarchical processing allows one to compare parts of the circuits without having the

schematic and layout completed for the entire design. If each level of a hierarchy contains information about no more than k junction nodes and gate nodes, and $F(k)$ represents the expected required time to compare two descriptions containing k structural logic expressions, then the time for comparison reduces to $F(k) \log_k(n)$, where n is the total number of structural logic expressions in the circuit.

5.2. Comparison of Boolean Expressions

There are several different ways to verify logical equivalence. The Differential Boolean Analyzer (DBA) in IBM's Boolean comparison approach [SmBH82], for example, examines the NAND operation of two gate-level Boolean equations. Assume f and g are the two functions to be compared. They are obtained from a segment builder for the reference model and the hardware function, respectively. Let $z = \overline{fg}$. The algorithm proceeds by assigning constant values (zero or one) to the inputs of the two functions, one at a time, in a way similar to Binary Decision Diagrams [Aker78]. Thus a binary tree can be constructed. Any input combination which leads to $f \neq g$ will have $z = 1$, indicating a counterexample to equivalence. An input combination which contains "don't care" terms and leads to $z = 0$ will be examined further. It is reported that this technique has been successfully used on the IBM 3081 with approximately 500,000 circuit elements, in which Boolean comparison runs are made for individual hardware modules of approximately 30,000 circuit elements each.

Another simple approach is to expand the two Boolean equations with the same source signal into truth tables and check corresponding rows at the same time. To speed up the process one can express one function in sum-of-products form and the other function in product-of-sums form. While the former is compared to logic one (1), the latter is checked for logic zero (0). Roughly half of the computation time can be saved. A coun-

terexample to equivalence is then detected when a mismatch is found in a row.

Since the Boolean comparison problem is known to be NP-complete [LePa81], one might anticipate that execution times would increase exponentially with the input size. However, circuit structures are limited not only by performance constraints such as area and speed, but also by electrical constraints such as the charge sharing problem and body effects. When complex structures with a large number of inputs have to be implemented, the best speed performance may be obtained by using stages where the number of inputs ranges from about two to five [WeEs85]. As the size of the transistor network increases, the circuit is subject to the charge redistribution problem and body effects due to the circuit's internal node capacitance. Many design engineers, therefore, consider circuits with more than three or four serial connected transistors to be poor designs. Thus, the number of variables involved in a logic path is typically small, and the exponential time complexity is thus affordable at this level of granularity.

Symbolic processing using simplification rules can help to reduce the time complexity [SrAg86]. Assume f and g are defined as before. Equations f and g are said to be equivalent if and only if $f \oplus g = \bar{f}g + f\bar{g} = 0$. Both f and g derived from circuit structures are expressed in sum-of-products form by applying the Boolean distributive law; and the complements of f and g are expressed in product-of-sums form using the DeMorgan's law. Simplification rules are then used to minimize $\bar{f}g$ and $f\bar{g}$. Although the resulting expressions might need to be expanded into individual product terms to see if they can be canceled out, the simplification rules are useful to speed up the Boolean comparison process since the resulting expressions are usually simpler.

5.3. Comparison Beyond Logic Structures

In many cases the functionality of a circuit is the major concern of the design. As long as circuit delays, area, and electrical parameters such as power consumption are tolerable within a certain range, various designs for the same circuit module based on different logic structures are allowed. For example, pseudo-nMOS NOR gates might be replaced by static CMOS gates and vice versa. In order to recognize and compare circuit functions of different logic structures, circuit pre-processing such as logic gate recognition is required to construct two circuits of the same type or to generate logic level descriptions.

Logical circuit representation exactly meets this requirement, since it is compatible with traditional logic descriptions at the logic level. Whenever it is convenient, the output of a logic gate can be expressed in terms of a standard logic description, such as using $f = \overline{a + b}$ instead of $f = [a + b]0 \cdot [ab]1$, so that logic manipulations at the gate level are possible.

Figure 5.4 shows a two-stage static CMOS XNOR gate. Since paths from the output of a static CMOS gate to V_{DD} and V_{SS} are complementary in logic, a single rule is used to reconstruct static CMOS circuits using pseudo-nMOS logic. Assume $f = [f_0]0 \cdot [f_1]1$, where f_0 and f_1 indicate simple paths from the output node f of a static CMOS gate to V_{SS} and V_{DD} , respectively. The *complement expression* f' of a Boolean expression f is defined as a Boolean expression which is the same as f except all logic variables and constants in f are substituted by their complements in f' . If $\overline{f_0}$ is logically the same as f_1' , then f_1 can be substituted for by a constant 0 for further comparison. Thus, we have

$$f = [f_0]0 \cdot [f_1]1 = [f_0]0 \cdot [0]1$$

provided that $\overline{f_0} = f_1'$. In fact, the function f can be obtained as $f = \overline{f_0}$ when $\overline{f_0} = f_1'$.

In Figure 5.4, the static CMOS XNOR circuit is expressed by

$$f = [ab]0 \cdot [a + b]1$$

$$z = [f(a + b)]0 \cdot [f + ab]1$$

using logical circuit representation. Since it is found that $f_1' = \overline{a} + \overline{b} = \overline{f_0}$ and $z_1' = \overline{f} + (\overline{a}\overline{b}) = \overline{z_0}$, we have

$$f = [ab]0 \cdot [0]1 = \overline{ab}, \text{ and}$$

$$z = [f(a + b)]0 \cdot [0]1 = \overline{f(a + b)}.$$

The resulting circuit structure is shown in Figure 5.5, which is comparable with other circuits using pseudo-nMOS logic.

To expand the above approach for comparison beyond logic structures, one needs to generalize the basic properties of n-type and p-type transistors. The functional recognition techniques proposed in Chapter 4 are very suitable to apply. Thus, the comparison process may be conducted at the logic level.

5.4. Comparison Hierarchy and Binding

One of the advantages of using logical circuit representation for connectivity comparison is its compatibility with higher level functional descriptions, since each simple path is represented by a traditional Boolean expression. When circuit functionality is the major concern of the design and the function of several consecutive stages have been recognized, the whole block can be treated as a black box with a number of Boolean equations for specifying its input-output relationships. Thus, our approach combines three levels of comparison:

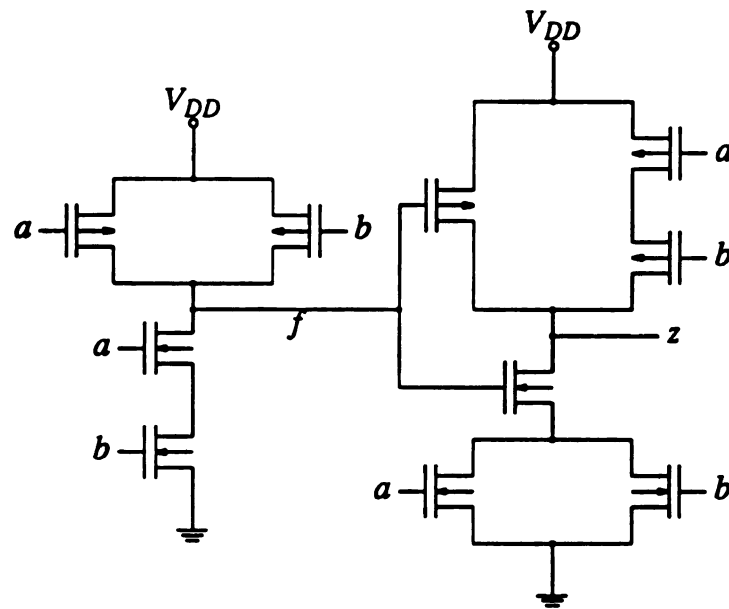


Figure 5.4. A two-stage static CMOS XNOR gate

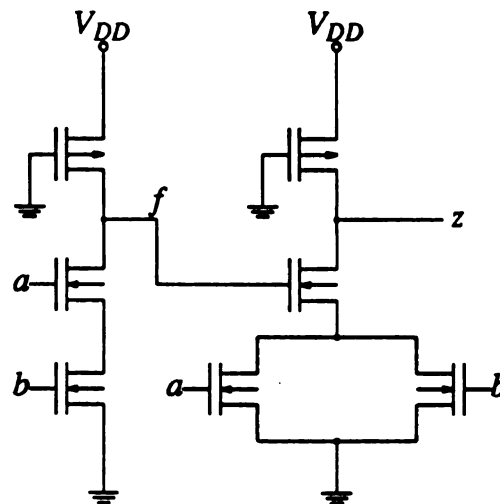


Figure 5.5. Two-stage pseudo-nMOS XNOR gate

- (1) At the transistor level, circuits to be compared are described in terms of logical circuit expressions based on individual disjoint components. Subcircuit permutations/repetitions and functional permutations are considered at this level by means of symbolic path analysis.
- (2) Different CMOS logic structures for the same function may be allowed for the gate level comparison. This usually requires additional rules for gate recognition. The number of logical circuit expressions are still the same for both circuits provided that the circuits are identical. Static CMOS gates can therefore be compared with pseudo-nMOS gates and vice versa.
- (3) The comparison at the block level checks if two corresponding black boxes are logically identical. Thus, logical circuit expressions are replaced by Boolean equations, which specify the relationship between the input and output nodes of the black box. The number of original logical circuit expressions might be different, but the number of Boolean equations (or the number of outputs for the corresponding black boxes) are the same.

Binding is an important procedure in the circuit comparison process. After two circuit schematics are specified in terms of logical circuit expressions, corresponding nodes have to be identified on a one-to-one basis. The combination of comparison and binding is basically a searching procedure, as the approach proposed in [SmBH82] for Boolean comparison. A simple heuristic is to first compare the logical circuit expression with the most number of variables in each set. Nodes with the same logical circuit expression are bound and put on a list. If a correspondence list such as the Signal Correspondence Record (SCR) list in [SmBH82] is available, the comparison process can be further simplified, and it is then possible to conduct parallel processing during the comparison process.

Errors occur when a logical circuit expression in one circuit does not occur in the other circuit. Errors are then detectable by looking through the mismatch between the schematic and the layout.

CHAPTER VI

CMOS DESIGN AND VERIFICATION

USING LOGICAL PREDICATES

Based on logical circuit expressions, a novel circuit representation using logical predicates is proposed to describe the connectivity of a CMOS series-parallel transistor network. A context-free grammar is proposed for constructing series-parallel networks, and a pushdown automata is developed for recognizing strings generated from the context-free grammar. Thus, the synthesis and verification processes for CMOS series-parallel transistor networks can be done in linear time. ITP (Interactive Theorem Prover), which was developed at Argonne National Laboratory, is used to demonstrate the capability of the approach.

Circuit synthesis and verification are major goals in design automation. A traditional approach to ensure the correctness of a physical layout is to simulate the extracted circuit using a switch level simulator and then compare the results with the expected input-output responses [BaTe80, Term83]. On the other hand, a more recent approach, formal verification, is input-pattern independent and is designed to guarantee functional equivalence between two representations of the design at different levels. Since the use of logic verification increases the level of confidence in the design, verification becomes increasingly important as the complexity of IC design grows [NeSa86].

Circuit synthesis is viewed as the process of transforming a high-level design specification into a low-level design specification that includes more structural details, leading to the physical design of the IC. Much previous work has been concentrated on the gate or higher levels. For example, local transformations [DaJB81] is a logic

synthesis approach based on gate-level circuits. Recently similar work [GrBD86] has used a set of rules to optimize the logic circuitry and to automate the design process. This approach basically considers gate-level optimization by converting gates from one type to another or by rearranging components to improve area and timing constraints.

Subrahmanyam [Subr86] has proposed a method for low-level automated synthesis using predicates. For a n -input function, a predicate requiring $2^n + 1$ arguments is needed. Such a logic predicate has different argument lengths for various logic functions, and therefore it is difficult to manipulate. In addition to the predicate representation, a designer has to derive a set of axioms for any given Boolean expression. Even a simple circuit like a CMOS inverter needs three axioms as well as eight demodulators (i.e., rewrite rules) and two support rules supplied by the designer. That is not attractive since the time spent in developing the axioms might be longer than building a cell for a cell library. It is difficult for a designer to generate axioms for all Boolean functions in his/her design. Thus it is almost impossible for a designer to build a medium- or large-sized circuit using this scheme.

An alternative way to drive the inference engine is developing general rules and describing a function in terms of Boolean expressions. What is needed is a set of axioms and/or demodulators which is generally applicable so as to generate all necessary functions for a given CMOS logic family. Different supports which describe necessary logic functions are then used to drive the inference engine in order to generate circuit schematic diagrams.

In this chapter, a novel approach using logical predicates is proposed for design and verification of CMOS logic circuits. Based on the same concepts as logical circuit expressions, logical predicate representation shows an approach when taking implementation into consideration. The Interactive Theorem Prover (ITP), written in Pascal and developed at Argonne National Laboratory [LuOv84], is used to demonstrate the

approach. Although the primary target technology is CMOS, it is believed that these techniques can be applied to other MOS technologies as well.

6.1. Using Logical Predicates

Predicates are used to specify relationships among object arguments. They are basically functions that map object arguments into TRUE or FALSE values. In this chapter, predicates are used to describe CMOS circuit connectivities at the transistor level and to form a basis for a rule-based system.

A total of five predicates are needed to describe CMOS logic circuits, as listed in Table 6.1. The drain and the source of a MOS transistor may be viewed as two switched terminals. Since MOSFETs are symmetric in nature, the drain and the source are interchangeable. They are physically equivalent, and the name assignment depends on the direction of current flow. Circuit outputs are represented using the predicate $CK(x, y)$ to emphasize primary outputs and outputs from internal stages. The predicate $CK(x, y)$ is also used to drive reasoning procedures in circuit verification. In this tree-structure representation, the number of transistors in a circuit is the same as the number of dn and dp predicates used in the circuit description.

A fully complementary CMOS gate consisting of a network of p-type transistors (the *load circuit*) and a network of n-type transistors (the *driver circuit*) is depicted in Figure 6.1. This circuit shows an implementation of the Boolean expression $z = \overline{ab+bc+ca}$ using FCMOS logic.

In representing a CMOS circuit using logic predicates, signals generated from non-separable components play an important role. Similar to the predicates PN and PD at some separating nodes to indicate the places where parallel devices start, each separating node is assigned with a node name to be referred to. On the other hand, separating nodes

Table 6.1. A set of five logic predicates for CMOS circuit representation

| predicate | explanation |
|------------|---|
| $dn(a, b)$ | drain of an n-type transistor (a : gate, b : source) |
| $dp(a, b)$ | drain of a p-type transistor (a : gate, b : source) |
| $CK(x, y)$ | output with x from n-type and y from p-type blocks |
| $PN(x, y)$ | predicates x and y connected in parallel (n-type) |
| $PD(x, y)$ | predicates x and y connected in parallel (p-type) |

between two serial devices, or those nodes along a path with a single device followed by a parallel structure, need not be exploited to eliminate unnecessary expressions.

For instance, the output of the circuit in Figure 6.1 is represented as:

$$CK(PN(dn(a, dn(b, 0)), PN(dn(b, dn(c, 0)), dn(c, dn(a, 0)))), \\ PD(dp(c, m), dp(a, m)));$$

where $m = PD(dp(b, n), dp(c, n))$ and $n = PD(dp(a, 1), dp(b, 1))$. In this case m and n are separating nodes.

If two identical transistors are connected in series, the rise and fall time will be approximately twice that of a single transistor with the same capacitive load. In circuit synthesis, separating nodes are avoided as much as possible to reduce the probability of forming long serial devices. Parallel connected transistors of the same type should be pushed towards the supply and ground rails so that internal node capacitance is minimized.

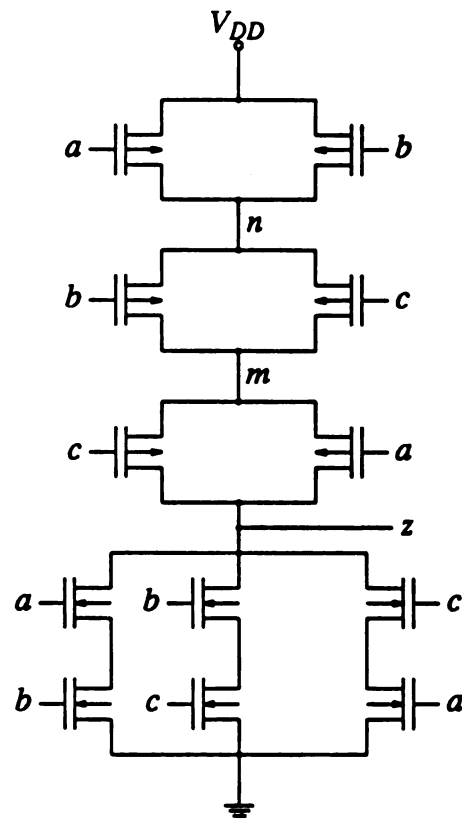


Figure 6.1. An FCMOS gate implementing $z = \overline{ab + bc + ca}$

An alternative to fully complementary CMOS logic is pseudo-nMOS logic. Pseudo-nMOS logic is a CMOS variation which uses a p-transistor to mimic an nMOS pull-up. The load device is a single p-transistor, with the gate connected to V_{SS} . This is equivalent to a conventional nMOS gate except that the depletion nMOS load is replaced by a p-device. Figure 6.2 shows a pseudo-nMOS gate realizing $z = \overline{ab} + cd$. The output signal is represented as

$$CK(PN(dn(a, dn(b, 0)), dn(c, dn(d, 0))), dp(0, 1));$$

where $dp(0,1)$ is the pull-up p-type transistor whose gate terminal is connected to the ground.

One of the best known approaches to the design of combinational logic in CMOS that avoids logical redundancy without suffering from the side effect of increased power dissipation is *Domino CMOS logic*. Each output in Domino CMOS logic is precharged high while the path to ground is open, and the precharge is stopped while the path to ground is activated. Transitions from precharge to evaluation are accomplished by means of a single clock edge applied simultaneously to all gates in the circuit. A Domino CMOS circuit is shown in Figure 6.3. The circuit consists of a single phase dynamic gate and a static CMOS buffer. Using logic predicates, the circuit is represented by

$$z = CK(dn(pz, 0), dp(pz, 1));$$

$$pz = CK(PN(dn(a, dn(b, m)), dn(c, dn(d, m))), dp(\phi, 1));$$

where $m = dn(\phi, 0)$. In fact, this output-driven representation specifies transistor connectivities in the circuit.

There are many other CMOS logic structures described in Chapter 2, including clocked CMOS, pass transistor logic, dynamic CMOS, cascade voltage switch logic, zipper CMOS, etc. The proposed set of logic predicates is flexible enough to describe all these logic structures [WuWN86, WuWN87].

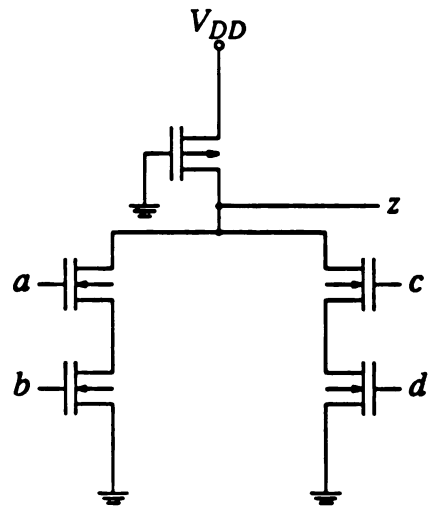


Figure 6.2. A pseudo-nMOS gate realizing $z = \overline{ab + cd}$.

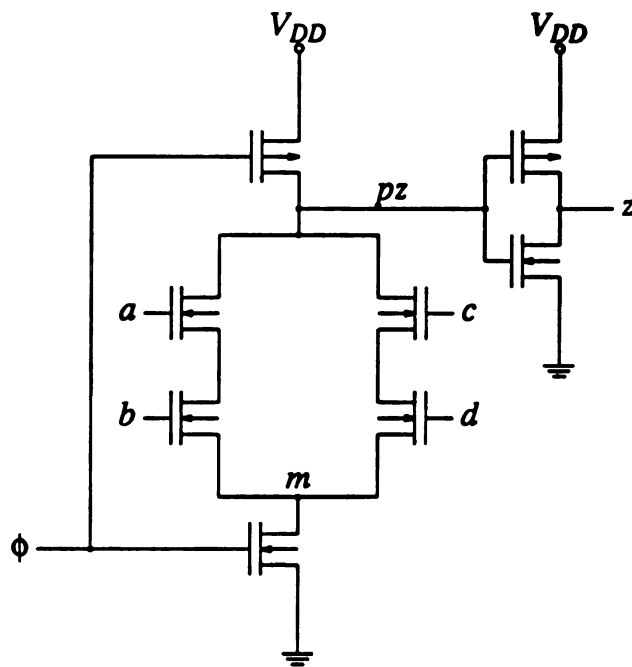


Figure 6.3. A Domino CMOS circuit with a clock signal ϕ

6.2. Formal Representation of CMOS Circuits

Fundamentally, a grammar is a finite mechanism for producing and recognizing sets of strings [Harr78]. A *phrase-structure grammar* is a 4-tuple: $G = (V, \Sigma, P, S)$, where:

- (a) V is a finite nonempty set called the vocabulary;
- (b) $\Sigma \subseteq V$ is a finite nonempty set called the terminal alphabet;
- (c) $S \in V - \Sigma = N$ is called the start symbol;
- (d) P is a finite set of rules (or productions) of the form $\alpha \rightarrow \beta$ where $\alpha \in V^*NV^*$ and $\beta \in V^*$.

A phrase-structure grammar has very general rules. The only requirement is that the left-hand side has at least one variable. It is equivalent to a nondeterministic Turing machine, and its parsing complexity is typically nonlinear.

Context-free languages are an important topic in formal language theory. The characterization of context-free languages can be expressed in terms of pushdown automata. A phrase-structure grammar $G = (V, \Sigma, P, S)$ is a context-free grammar if each rule is of the form $A \rightarrow \alpha$, where $A \in N$, $\alpha \in V^*$. The term “context-free” means that a nonterminal on the left-hand side of each rule can be replaced by whatever appears on the right-hand side, no matter the context.

A context-free grammar $G = (V, \Sigma, P, S)$ can be used to describe CMOS circuits. The terminal alphabet Σ is formed by a set of logic values, which consists of 0 and 1 in the simplest case, the input signal set I , the separating variable set A , the brackets (“[” and “]”) and the plus sign “+” for parallel structures, and a period “.” for separating n-type and p-type transistor networks. Logic value “0” denotes ground and signals in the low voltage V_L range, while logic value “1” denotes the power supply and signals in the high voltage V_H range. The number of logic levels can be expanded as necessary to

increase modeling accuracy.

The input signal set I and the separating variable set A are both finite since the number of inputs and the number of internal separating nodes in a circuit are finite. The period “.” separates the n-type network from the p-type network. Signals derived from parallel devices are surrounded by “[” and “]” and are connected by the plus sign “+”. Thus, we have $\Sigma = \{ 0, 1, \cdot, [,], +, I, A \}$. Actually the separating variable set A can be further expanded.

Nonterminal variables are crucial in constructing CMOS circuits. Two nonterminal variables $\langle dn \rangle$ and $\langle dp \rangle$ are used to denote the primitive elements, n-type transistor and p-type transistor, respectively. The variable $\langle output \rangle$ denotes an output signal observed at a given node, and also serves as the start symbol. Three variables, $\langle PN \rangle$, $\langle PD \rangle$, and $\langle CK \rangle$, are used to describe circuit structures in the transistor network. Variable $\langle PN \rangle$ indicates n-type devices connected in parallel, while variable $\langle PD \rangle$ represents p-type devices connected in parallel. Variable $\langle CK \rangle$ denotes a signal which is pulled from a junction between n-type and p-type transistor networks.

There is an alternative form of context-free grammar which is used in the specification of programming languages, namely, the *Backus normal form* (BNF) [LePa81]. The BNF form uses four meta characters which are not in the vocabulary: “<”, “>”, “::=”, and “|”. Strings are enclosed by < and > and denote variables. The symbol ::= serves as a replacement operator, and | is read “or”. Production rules are used to parse strings in the context-free language. The production set P contains six rules to represent CMOS circuits using BNF form:

$$\langle output \rangle ::= \langle CK \rangle | 0 | 1$$

$$\langle CK \rangle ::= \langle dn \rangle \cdot \langle dp \rangle | \langle dn \rangle \cdot \langle PD \rangle | \langle PN \rangle \cdot \langle dp \rangle | \langle PN \rangle \cdot \langle PD \rangle$$

$$\langle PN \rangle ::= [\langle dn \rangle + \langle dn \rangle] | [\langle dn \rangle + \langle PN \rangle]$$

$$\langle PD \rangle ::= [\langle dn \rangle + \langle dp \rangle] \mid [\langle dp \rangle + \langle PD \rangle]$$

$$\langle dn \rangle ::= I \langle dn \rangle \mid I \langle PN \rangle \mid IA \mid IO$$

$$\langle dp \rangle ::= I \langle dp \rangle \mid I \langle PD \rangle \mid IA \mid I1$$

CMOS circuit representation using the predicates described in the previous section illustrates how the “parse trees” are constructed, which in turn represent the physical transistor structures of the CMOS circuits. If we trace a parsing tree inorders (from left to right), the leaves traveled through form a string in the language $L(G)$. Strings in the language $L(G)$ look like logic equations. Separating nodes in n-type or p-type blocks can be expanded to complete the circuit representation.

Predicate representations are used to describe CMOS circuit connectivities. The representations are actually parse trees in the formal language. Each parse tree comes from its original string, which can be recognized by the grammar of the language. Thus, for the circuit in Figure 6.1, there is a corresponding string to describe the behavior of the gate:

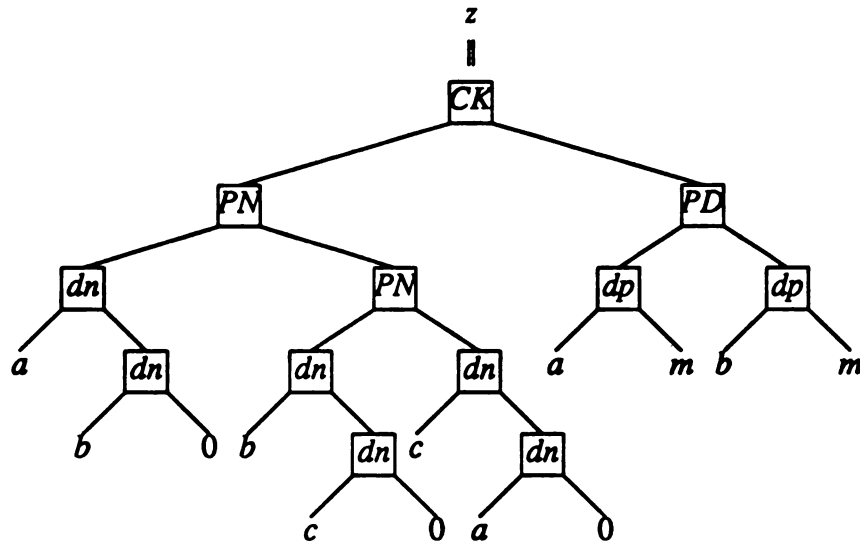
$$z = [ab0 + [bc0 + ca0]] \cdot [am + bm]$$

$$m = \cdot [bn + cn]$$

$$n = \cdot [c1 + a1].$$

The parse tree of node z is shown in Figure 6.4.

The above strings generated from the production rules P resemble Boolean expressions. With the help of such tools as automated reasoning, circuit verification and synthesis can be achieved. In the following sections, the process of how the function of a CMOS circuit can be recognized based on its transistor structure is addressed, and how such a circuit can be constructed from its corresponding Boolean expressions is also discussed.

Figure 6.4. The parse tree of node z in Figure 6.1

6.3. Deterministic Pushdown Automata

Pushdown automata (*pda* for short) form a class of devices for recognizing strings generated from grammars. A pushdown automaton is a 7-tuple [LePa81]

$$A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

- (a) Q is a finite nonempty set of states;
- (b) Σ is a finite nonempty set of input symbols;
- (c) Γ is a finite nonempty set of pushdown symbols;
- (d) $q_0 \in Q$ is the initial state;
- (e) $Z_0 \in \Gamma$ is the initial symbol on the pushdown store;
- (f) $F \subseteq Q$ is a set of final states; and

(g) $\delta: Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma \rightarrow \text{finite subsets of } Q \times \Gamma^*$, where Λ is the null symbol.

We usually write $(q', \alpha) \in \delta(q, a, Z)$ to indicate a particular transition. To ensure that a pda is deterministic, we must restrict the “choice” of a next state to at most one state. That is, for all $(q, a, Z) \in Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$ we have $|\delta(q, a, Z)| \leq 1$.

A *deterministic context-free language* can be characterized using deterministic pushdown automata. A set L is called a deterministic context-free language if there is some deterministic pda, D , such that $L = T(D)$, where $T(D)$ is the set accepted by automaton D by final state. Deterministic context-free languages are one of the most important classes of languages because it is possible to construct efficient parsers for them. They can be parsed from left to right by working from the bottom to the top. This class of grammars is rich enough to contain the syntax of many programming languages, but restrictive enough to be parsed in linear time. Moreover, these languages are all unambiguous [Harr78].

A deterministic pda which accepts the context-free language in Section 6.2 can be generated. The listing of the deterministic pda is tedious. In order to save space, we list here the transitions used to recognize n-type transistor networks only. Representation for p-type transistor networks can be recognized using similar transitions. Since representations for n-type and p-type transistor networks in a CMOS gate are always separated by a period “.”, it is easy to extend this set of transitions so that it accepts the context-free language $L(G)$ described in Section 6.2.

Let $D = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9\}$, $\Sigma = \{0, i, a, [, +,]\}$, $\Gamma = \{Z_0, [\]\}$, and $F = \{q_9\}$. Note that we use lower case letter “i” and “a” to represent input nodes and separating nodes, respectively. For $d \in \Gamma$ and $A \in \{a, i\}$, the transition set δ is listed in Table 6.2.

The pushdown automaton is clearly deterministic since for all $(q, a, Z) \in Q \times (\Sigma \cup$

Table 6.2. State transition table for recognizing n-type networks

| |
|--|
| (1) $\delta(q_0, i, d) = (q_1, d);$ |
| (2) $\delta(q_0, [, d) = (q_2, d];$ |
| (3) $\delta(q_1, i, d) = (q_1, d);$ |
| (4) $\delta(q_1, [, d) = (q_2, d];$ |
| (5) $\delta(q_2, i, d) = (q_3, d);$ |
| (6) $\delta(q_3, [, d) = (q_2, d];$ |
| (7) $\delta(q_3, i, d) = (q_3, d);$ |
| (8) $\delta(q_3, A, d) = (q_4, d);$ |
| (9) $\delta(q_4, +, d) = (q_5, d);$ |
| (10) $\delta(q_5, [, d) = (q_2, d];$ |
| (11) $\delta(q_5, i, d) = (q_6, d);$ |
| (12) $\delta(q_6, i, d) = (q_6, d);$ |
| (13) $\delta(q_6, [, d) = (q_2, d];$ |
| (14) $\delta(q_6, A, d) = (q_7, d);$ |
| (15) $\delta(q_7,], d) = (q_8, \Lambda);$ |
| (16) $\delta(q_8,], d) = (q_8, \Lambda);$ |
| (17) $\delta(q_8, +, d) = (q_5, d);$ |
| (18) $\delta(q_8, \Lambda, Z_0) = (q_9, Z_0);$ |
| (19) $\delta(q_1, A, d) = (q_9, d);$ |

$\{\Lambda\} \times \Gamma$, $|\delta(q, a, Z)| \leq 1$. Therefore $L(G) = T(D)$ is a deterministic context-free language. Notice that the above deterministic pda accepts the language which can be generated by the following production rules only:

- (a) $\langle PN \rangle ::= [\langle dn \rangle + \langle dn \rangle] \mid [\langle dn \rangle + \langle PN \rangle]$
- (b) $\langle dn \rangle ::= i \langle dn \rangle \mid i \langle PN \rangle \mid iA \mid i0$

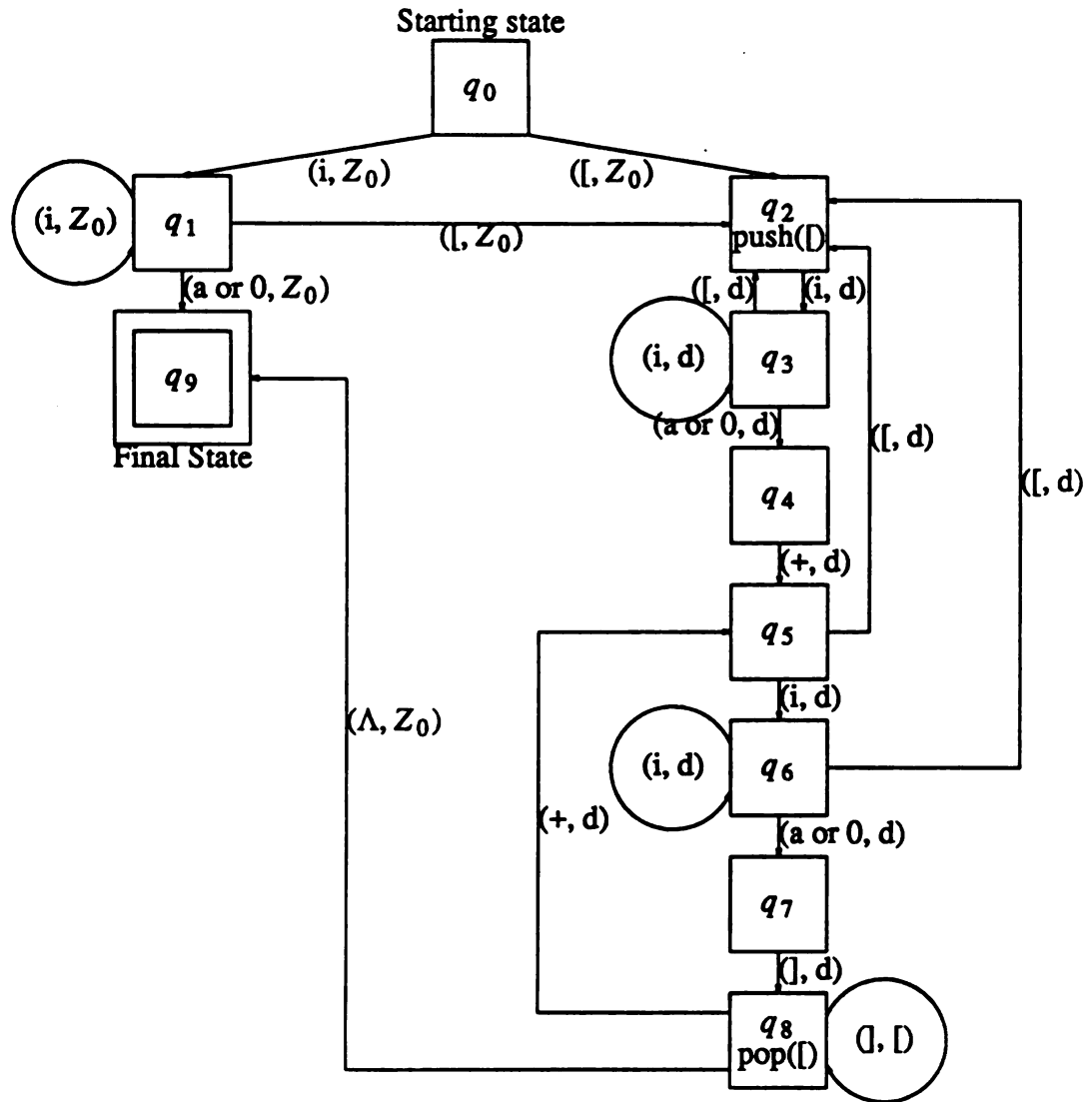
The start symbol could be either $\langle PN \rangle$ or $\langle dn \rangle$. A transition diagram is shown in Figure 6.5, by which n-type transistor blocks can be recognized. Since p-type transistor networks have similar production rules, a complete state-transition table could be established to recognize the context-free language. Thus one knows that the CMOS circuit representation is deterministic and it is recognizable in linear time.

6.4. Automated Circuit Verificaiton

Although formal verification of digital systems is still in its infancy, a number of researchers have addressed this issue at the gate and higher levels in the VLSI design hierarchy.

In this section, a set of basic demodulators are derived to guide the verification procedure for FCMOS circuits. Given a combinational circuit already designed as well as the logical specifications it supposedly satisfies, one is asked to prove that the design, in fact, meets the specification. In order to reduce the number of inference rules in use, $\langle PN \rangle$ and $\langle PD \rangle$ are merged and replaced by a single predicate $\langle PP \rangle$ to represent parallel devices. Table 6.3 lists a set of six rules for true CMOS circuit verification.

Rule R5 is actually encoded in the production rules of the context-free language defined in Section 6.2. Rules R3 and R4 are partially encoded in Section 6.2 since strings in the context-free language may carry constants 0 and 1, which represent the power and



| Symbol | Explanation |
|-----------|--------------------------------------|
| (x, y) | x =input symbol, y =top of stack |
| Λ | null character |
| d | stack symbol $d \in \Gamma$ |

Figure 6.5. State transition diagram for recognizing n-type transistor networks

Table 6.3. Verification rules for true CMOS circuits

| | Rule |
|----|---------------------------------------|
| R1 | $dn(X, 0) = X$ |
| R2 | $dp(X, 1) = \bar{X}$ |
| R3 | $dn(X, Y) = XY$ |
| R4 | $dp(X, Y) = \bar{X}Y$ |
| R5 | $PP(X, Y) = X + Y$ |
| R6 | $CK(X, Y) = \bar{X}$ if $\bar{X} = Y$ |

the ground, respectively. It is these rules that make strings in the context-free language resemble Boolean expressions. If an n-type transistor with gate signal X is connected to the ground V_{SS} , a logic value X is assigned to the drain of the transistor. If a p-type transistor with gate signal X is connected to the power line V_{DD} , a logic value X is also assigned to the drain of the transistor. Due to the true CMOS circuit structure, a path constructed by serial n-type transistors from a node toward V_{SS} is assigned the logical AND of the gate signals along the path, and a path formed by serial p-type transistors from a node toward V_{DD} is also assigned the logical AND of the gate signals along the path. Similarly, two transistors (both either n-type or p-type) connected in parallel are treated as the logical OR of their gate signals.

The last rule, R6, chooses the negated logic expression of the n-type block to verify the function of a circuit as long as it is the same as the logic expression of its p-type block. This rule reflects the “logical redundancy” property of the true CMOS circuits.

Using automated reasoning [Wojc83, WoOL84, KaWo85], logic circuit validation can be viewed as a translation between two different representations. For circuits described in terms of logic networks, the verification of the circuit function is conceptually straightforward in that the function computed by a network is given by the composition of the functions at previous stages. After the functions of the circuit have been derived at the higher level, expressions are canonicalized and compared with the specifications the circuit supposedly satisfies. The important topics of expression canonicalization and Boolean equivalence are discussed elsewhere [WoOL84, WoKS84].

The ITP (Interactive Theorem Prover) automated reasoning system [LuOV84] developed at Argonne National Laboratory is used to demonstrate the verification process. Other systems, such as Prolog [StSh86], could also be used. Figure 6.6 illustrates the transistor schematics of a one-bit full adder using true CMOS logic. Its corresponding circuit representation is shown in Table 6.4 using ITP clauses. The output signal *outs* and *outc* denote the sum and carry output, respectively. Since an output signal and its corresponding circuit has to be identified in a one-to-one manner, clauses in ITP are written in the form of

$$ck(output_signal, n-block_structure, p-block_structure);$$

instead of

$$output_signal = ck(n-block_structure, p-block_structure).$$

During the verification procedure, demodulators might be generated after a gate is verified. If the output node of a gate is connected to its next stage circuits as an input signal, we need to represent the signal in terms of primary inputs. This means clauses derived during the verifications may also become demodulators and should be dynamically added to the demodulator list. Thus, an extra demodulator which generates new inference rules is used.

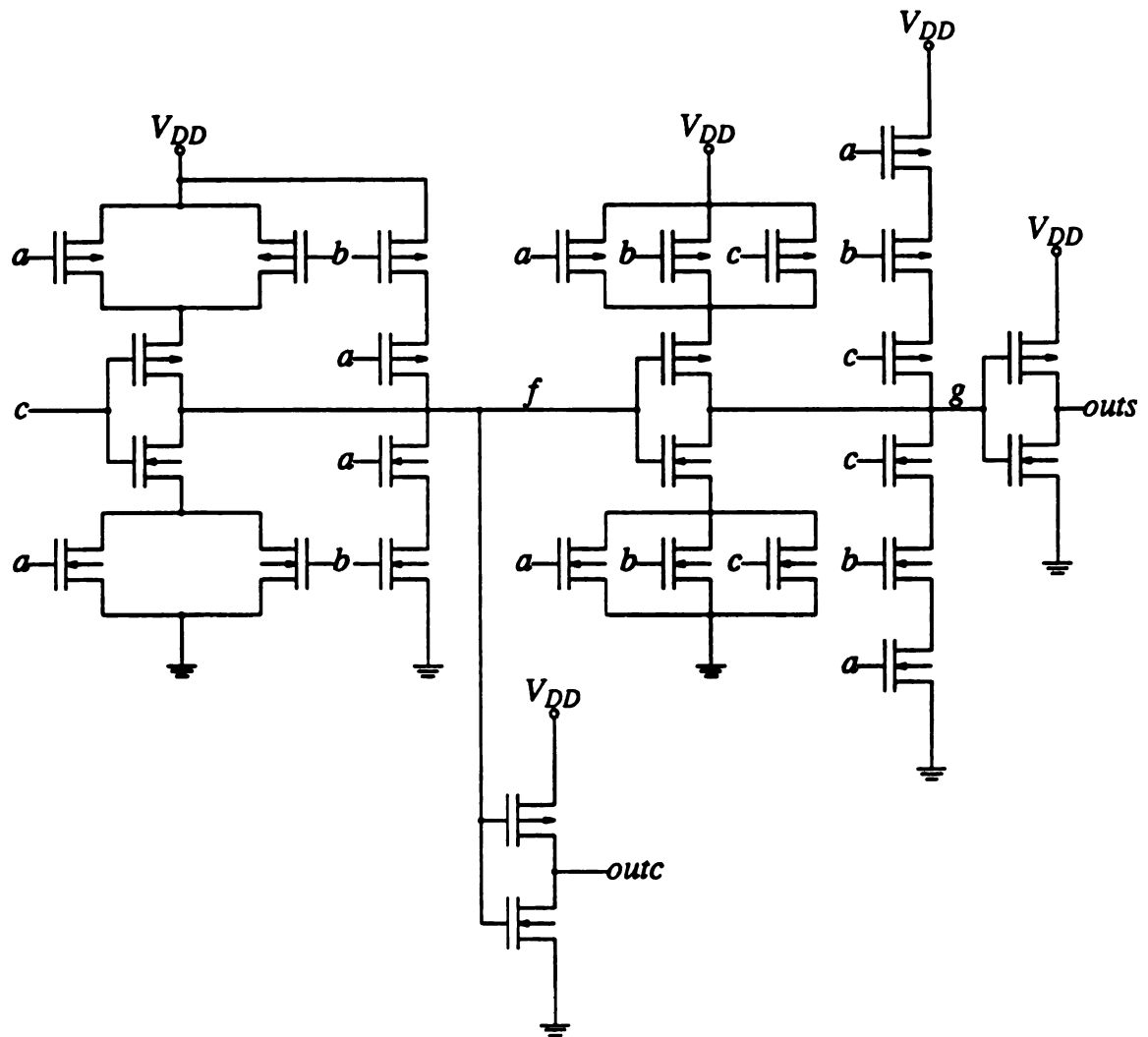


Figure 6.6. Transistor schematics of a one-bit full adder

Table 6.4. ITP predicate statements of the circuit in Figure 6.6

| |
|---|
| <i>ck</i> (<i>f</i> , <i>pp</i> (<i>dn</i> (<i>a</i> , <i>dn</i> (<i>b</i> , 0)), <i>dn</i> (<i>c</i> , <i>pp</i> (<i>dn</i> (<i>a</i> , 0), <i>dn</i> (<i>b</i> , 0)))), <i>pp</i> (<i>dp</i> (<i>a</i> , <i>dp</i> (<i>b</i> , 1)), <i>dp</i> (<i>c</i> , <i>pp</i> (<i>dp</i> (<i>a</i> , 1), <i>dp</i> (<i>b</i> , 1))))); |
| <i>ck</i> (<i>outc</i> , <i>dn</i> (<i>f</i> , 0), <i>dp</i> (<i>f</i> , 1)); |
| <i>ck</i> (<i>g</i> , <i>pp</i> (<i>dn</i> (<i>c</i> , <i>dn</i> (<i>b</i> , <i>dn</i> (<i>a</i> , 0))), <i>dn</i> (<i>f</i> , <i>pp</i> (<i>dn</i> (<i>a</i> , 0), <i>pp</i> (<i>dn</i> (<i>b</i> , 0), <i>dn</i> (<i>c</i> , 0))))), <i>pp</i> (<i>dp</i> (<i>c</i> , <i>dp</i> (<i>b</i> , <i>dp</i> (<i>a</i> , 1))), <i>dp</i> (<i>f</i> , <i>pp</i> (<i>dp</i> (<i>a</i> , 1), <i>pp</i> (<i>dp</i> (<i>b</i> , 1), <i>dp</i> (<i>c</i> , 1)))))); |
| <i>ck</i> (<i>outs</i> , <i>dn</i> (<i>g</i> , 0), <i>dp</i> (<i>g</i> , 1)); |

Although there are four clauses in Table 6.4, only two of them, *outs* and *outc*, are of interest. The outputs of the intermediate stages, *f* and *g*, are not of interest outside of the circuit. Two demodulators, which explicitly specify the output signals so that they can be easily identified, can be used to indicate the important signals in the circuit. To make the output more readable, three more demodulators are added into the demodulator list. The initial demodulator list that serves as an input list is shown in Table 6.5. Note that all the demodulators are generally applicable, except the two that are used to explicitly specify primary outputs.

The results from ITP are shown below:

lis (*outc*, *lan* (*lor* (*a*, *b*), *lor* (*c*, *lan* (*a*, *b*))));
lis (*outs*, *lan* (*lor* (*c*, *lor* (*b*, *a*)), *lor* (*lor* (*lan* (*lno* (*a*), *lno* (*b*)),
lan (*lno* (*c*), *lor* (*lno* (*a*), *lno* (*b*)))), *lan* (*a*, *lan* (*b*, *c*))));

In other words, we have

$$outc = (a + b)(c + ab) = ab + bc + ca,$$

and

$$outs = (c + b + a)(\bar{a}\bar{b} + \bar{c}(\bar{a} + \bar{b}) + abc) = a \oplus b \oplus c.$$

Table 6.5. Demodulator list for the circuit in Figure 6.6

| |
|--|
| $eq(ck(outc, x1, x2), lis(outc, x2));$ |
| $eq(ck(outs, x1, x2), lis(outs, x2));$ |
| $eq(dn(x1, 0), x1);$ |
| $eq(dp(x1, 1), lno(x1));$ |
| $eq(dn(x1, x2), lan(x1, x2));$ |
| $eq(dp(x1, x2), lan(lno(x1), x2));$ |
| $eq(pp(x1, x2), lor(x1, x2));$ |
| $eq(ck(x1, x2, x3), eq(x1, x3));$ |
| $eq(lno(lno(x1)), x1);$ |
| $eq(lno(lor(x1, x2)), lan(lno(x1), lno(x2)));$ |
| $eq(lno(lan(x1, x2)), lor(lno(x1), lno(x2)));$ |

With three demodulators for logic manipulation and two demodulators for specifying primary outputs as well as the six original inference rules, some key metrics for the verification of the circuit are shown in Table 6.6. The table shows data from an execution on a SUN-3/160 running UNIX 4.2. Owing much of its speed to the fact that no

occurs check is performed, other logic programming techniques might result in runtime improvements by at least 16-20 times [KISW86]. A complete listing created by ITP is shown in the Appendix.

Table 6.6. Metrics for verification of the one-bit full adder

| Metric | | Metric | |
|--------------------|------|------------------------------|-----|
| Runtime (sec) | 6.78 | Clauses Forward Subsumed | 3 |
| Input Clauses | 15 | Clauses Back Demodulated | 5 |
| Given Clauses | 11 | Unification Attempts | 170 |
| Clauses Generated | 5 | Unification Success | 166 |
| Clauses Integrated | 8 | Failures due to Occurs Check | 0 |

The performance of the ITP reasoning system is limited by the natural property of the combinational-explosion problem. However, the inorder search of a parse tree is linear. Since the circuit structure is represented in terms of logic predicates, dedicated programs can be generated based on the context-free language so that the above circuit verification can be done in linear time. In other words, logic expressions can be directly derived from the corresponding circuit structures in linear time. For the one-bit full adder, the logic expressions are derived as follows:

$$\begin{aligned}
 outc &= ck(dn(f, 0), dp(f, 1)) = \bar{f} \\
 &= pp(dn(a, dn(b, 0)), dn(c, pp(dn(a, 0), dn(b, 0)))) \\
 &= ab + c(a + b), \\
 outs &= ck(dn(g, 0), dp(g, 1)) = \bar{g}
 \end{aligned}$$

$$\begin{aligned}
&= pp(dn(c, dn(b, dn(a, 0))), dn(f, pp(dn(a, 0), \\
&\quad pp(dn(b, 0), dn(c, 0)))) \\
&= cba + f(a + (b + c)) \\
&= cba + \overline{ab + c(a + b)}(a + b + c)
\end{aligned}$$

Thus, we can verify the function of the one-bit full adder by the output signals *outc* and *outs*.

Rules in Table 6.3 can be partitioned into two categories: logic-family dependent rules and logic-family independent rules. For instance, rule R6 which reflects the logical redundancy property of true CMOS circuits is a logic-family dependent rule. Rules of this kind can be extended for other CMOS logic circuits. In pseudo-nMOS circuits, for example, the following three rules

- (1) $dp(0, 1) = NOT$
- (2) $PD(NOT, X) = NOT$
- (3) $CK(X, NOT) = \bar{X}$

are used. The nonterminal set $N = V - \Sigma$ is slightly modified to contain the symbol *NOT* to represent the unique structure of the pseudo-nMOS logic, in which p-type block usually has one grounded p-type transistor as the load circuit.

6.5. Automated Circuit Design

Circuit design or synthesis is viewed as the process of transforming a high-level design specification into a low-level design specification that includes more structural details, leading to the physical design of the IC. Why is automatic synthesis difficult to achieve? The major problem is that, in the search for optimal designs, a combinatorial explosion of synthesis possibilities is observed at every stage of the design process

[Shiv83, Park84]. As one proceeds down the design hierarchy, more detail is needed in the specification to describe a circuit.

The advantage of automated synthesis is to be able to design circuits faster, and more accurately, than can be done manually. Of the characteristics needed in a representation for logic synthesis, the most important one is the need to capture the functionality of a design. The predicate representation exactly meets this requirement.

Given a logic function, say $f = \overline{ab + bc + ca}$, a corresponding logic circuit is to be designed. The following procedure illustrates a simple systematic approach to design combinational circuits.

Basic Synthesis Procedure:

- (1) Negate the logic function for its n-type block while manipulating the original Boolean equation for the p-type block. At this stage, logic equations for both blocks are arranged so that the negation operator is used for only single logic variables. Thus, DeMorgan's laws, $\overline{X + Y} = \bar{X} \bar{Y}$ and $\overline{XY} = \bar{X} + \bar{Y}$, are used to arrange logic equations. Each product term at the first level (not those inside parentheses) is then concatenated with 1 or 0, which indicates the power and ground, respectively. Note that each logic variable in a logic equation corresponds to the gate signal of a certain transistor.
- (2) Using the associative law, the constant 1 or 0 at the end of each product term is merged into its last pair of parentheses, if they exist. Boolean equations are then rearranged from left to right so that brackets and symbol “+”s, which are used to represent parallel devices in the predicate representation, match the rules for parallel devices in the deterministic context-free language.
- (3) Perform parsing on both blocks from left to right. If the string cannot be parsed at

some point, introduce a separating node which is used as a signal source to cover the rest of the string.

- (4) Perform parsing for separating nodes based on the strings left from both blocks. This step is repeated many times for all separating nodes until the original strings are completely covered.

Based on the basic synthesis procedure, the design process is driven by a parsing mechanism with linear time complexity. In order to distinguish signals at separating nodes from other signals, upper case letters M and N are used in this example. The process for constructing the CMOS circuit is shown below by means of equations. The result is listed using the predicate representation which specifies its circuit connectivity:

$$\begin{aligned}
 f &= \overline{ab + bc + ca} \\
 &= CK(ab + bc + ca, (\bar{a} + \bar{b})(\bar{b} + \bar{c})(\bar{c} + \bar{a})); \\
 &= CK(ab0 + bc0 + ca0, (\bar{a} + \bar{b})(\bar{b} + \bar{c})(\bar{c} + \bar{a})1); \\
 &= CK(ab0 + bc0 + ca0, (\bar{a} + \bar{b})(\bar{b} + \bar{c})(\bar{c}1 + \bar{a}1)); \\
 &= CK([ab0 + [bc0 + ca0]], [\bar{a} + \bar{b}] [\bar{b} + \bar{c}] [\bar{c}1 + \bar{a}1]); \\
 &= CK([ab0 + [bc0 + ca0]], [\bar{a} + \bar{b}] M); \\
 &= CK([ab0 + [bc0 + ca0]], [\bar{a}M + \bar{b}M]); \\
 &= CK(PN(dn(a, dn(b, 0)), PN(dn(b, dn(c, 0)), dn(c, dn(a, 0)))), \\
 &\quad PD(dp(a, M), dp(b, M)));
 \end{aligned}$$

where

$$\begin{aligned}
 M &= \bullet [\bar{b} + \bar{c}] [\bar{c}1 + \bar{a}1] \\
 &= \bullet [\bar{b} + \bar{c}] N \\
 &= \bullet [\bar{b}N + \bar{c}N] \\
 &= PD(dp(b, N), dp(c, N));
 \end{aligned}$$

and

$$\begin{aligned} N &= \bullet [\bar{c}1 + \bar{a}1] \\ &= PD(dp(c, 1), dp(a, 1)); \end{aligned}$$

The symbol “•” in those expressions is used to indicate the device type. Thus expressions whose strings start with “•” are formed by p-type transistors, while those with “•” attached at the end are constructed by n-type transistors.

The circuit schematics generated from the parsing mechanism is the same as the circuit shown in Figure 6.1. To evaluate the precise performance of the design, one should have lower-level parameters supplied by silicon foundries, such as oxide thickness, substrate doping, zero-bias threshold voltage, etc. Thus, circuit performance is a relative term at this level. In fact, one may construct different CMOS circuits to perform the same function and to provide better performance. Algorithms for obtaining better circuit structures are currently under investigation.

The goal of logic synthesis is to accept functional specifications, such as Boolean expressions, for a hardware unit and to generate automatically a detailed, technology-specific implementation comparable in quality to that of an experienced engineer. Synthesizing combinational logic from a Boolean expression at the gate level is a relatively straightforward process. Most existing low-level synthesis procedures assume that all memory elements of the final implementation are identified in its original specification [DaJB81]. A given sequential circuit is partitioned into several disjoint combinational pieces and thus the synthesis process is simplified. Given the specification of a sequential element, the basic synthesis procedure that has been proposed can also be employed to generate its predicate representation.

Due to the existence of feedback lines in sequential circuits, pseudo signals are introduced when sequential circuit elements are required. The strategy for sequential ele-

ment synthesis is to further partition a sequential circuit or element by introducing some pseudo signal so that each partition is a combinational piece. Thus the basic synthesis procedure can be applied.

For storage elements, the transistor schematics of a static CMOS D flip-flop is depicted in Figure 5.5. The function of the D flip-flop can be expressed as a Boolean function:

$$q_{n+1} = q_n \bar{\phi} + d\phi$$

where q_n and q_{n+1} indicate the current state and next state of the D flip-flop, respectively.

By introducing the pseudo signal $M = \bar{q}$, the ambiguity between q 's on either side of the equality symbol can be eliminated. Thus, we have $M = \overline{q\bar{\phi} + d\phi}$ and $q = \bar{M}$.

These two combinational pieces can be synthesized using the basic synthesis procedure:

$$\begin{aligned} M &= \overline{q\bar{\phi} + d\phi} \\ &= CK(q\bar{\phi} + d\phi, (\bar{q} + \phi)(\bar{d} + \bar{\phi})) \\ &= CK(q\bar{\phi}0 + d\phi0, (\bar{q} + \phi)(\bar{d} + \bar{\phi})1) \\ &= CK(q\bar{\phi}0 + d\phi0, (\bar{q} + \phi)(\bar{d}1 + \bar{\phi}1)) \\ &= CK([q\bar{\phi}0 + d\phi0], [\bar{q} + \phi][\bar{d}1 + \bar{\phi}1]) \\ &= CK([q\bar{\phi}0 + d\phi0], [\bar{q} + \phi]N) \\ &= CK([q\bar{\phi}0 + d\phi0], [\bar{q}N + \phi N]) \\ &= CK(PN(dn(q, dn(\bar{\phi}, 0)), dn(d, dn(\phi, 0))), \\ &\quad PD(dp(q, N), dp(\bar{\phi}, N))); \end{aligned}$$

where

$$\begin{aligned} N &= \bullet [\bar{d}1 + \bar{\phi}1] \\ &= PD(dp(d, 1), dp(\phi, 1)); \end{aligned}$$

The other portion of the circuit is given by

$$\begin{aligned} q &= \bar{M} \\ &= CK(M, \bar{M}); \\ &= CK(M0, \bar{M}1); \\ &= CK(dn(M, 0), dp(M, 1)); \end{aligned}$$

Figure 6.7 illustrates the transistor schematic of the alternative D flip-flop design using the basic synthesis procedure and pseudo signal M . Note that the signal N in Figure 6.7 is a separating node and is generated during the basic synthesis process.

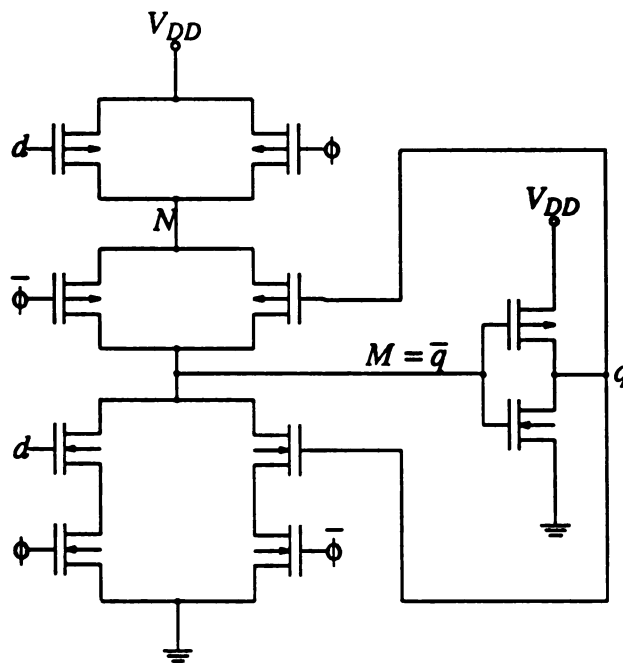


Figure 6.7. Alternative D flip-flop design using basic synthesis procedure

Compared with the D flip-flop in Figure 5.5, which has fourteen transistors, the alternative design with ten transistors uses only d (instead of both d and \bar{d}) and is smaller

and faster. This D-latch is also used by Reddy [ReRe86] as the basic structure for detecting stuck-open faults in CMOS latches and flip-flops. It has been shown that the latch is devoid of clock turn-on and turn-off hazards and glitches if the data is stable when the clock changes, and also that the latch is race free if the data is stable when the clock turns off.

CHAPTER VII

CONCLUSIONS

7.1. Summary

One reason for the success in producing VLSI devices is the availability of transistor switches. These elements, also known as the n-type and p-type field-effect transistors in silicon technology, allow large digital circuits to be made with just two types of component. Thus the transistor switches realize what may be called the physicist's dream of a single fundamental particle.

At the transistor switch level, one no longer has to worry about various types of logic gates. The simplicity allows the assemblage of large aggregates to perform powerful functions. This simplicity, it is believed, is also the power behind the switch-level techniques when it comes to many VLSI areas such as synthesis, simulation, layout, and test.

In this thesis, a novel approach for representing CMOS logic circuit networks at the transistor level has been proposed. Unlike traditional device listing approaches, which represent only circuit structures, logical circuit expressions combine structural data with behavioral information, and thus illustrate a way to reduce the difficulty of information transformation between behavioral and structural representations for CMOS circuits.

As explained in Chapter 1 and Chapter 2, CMOS technology has been recognized as a leading contender for existing VLSI systems, and is projected by industry analysts to be the dominant technology for the next decade. Various CMOS logic structures and design styles were examined to help to understand the essence of CMOS logic design and the

transistor structure of CMOS circuits. Both advantages and disadvantages of various CMOS logic structures and design styles were described. The charge sharing problem inherent in many CMOS logic structures was clearly illustrated, and possible solutions were proposed. It is also pointed out that the choices of logic structures and design styles are basically trade-offs among a number of factors, and are matters of art.

Chapter 3 introduced the definition and basic concepts of logical circuit expressions. Approaches for generating logical circuit expressions were described. A number of examples were given to demonstrate how the logical circuit expressions can be used to represent CMOS digital circuits. Among those examples, a CMOS decision-making circuit and a CMOS mutual exclusion element used in arbiter designs were proposed [WuNi87].

Functional recognition is an important step toward symbolic circuit verification. In this thesis, a new approach for logic component recognition was proposed. A number of recognition rules were developed for symbolic verification of CMOS logic circuits. Based on functional expansion and logical circuit expressions, this approach can be applied to various CMOS logic structures to verify logic functions [WNW87a]. In general, this recognition technique is able to shift the schematic comparison process from pure topology checking to logic function comparison, and is also useful in the field of reverse engineering. Reverse engineering in the field of VLSI design is a process which takes an existing but unknown circuit artwork as input, and generates a higher-level description, such as a schematic capture. With the power of this recognition technique, the reverse engineering process can be significantly simplified.

Another application of logical circuit expressions to CMOS VLSI design automation is the comparison of CMOS transistor schematic networks. Traditional approaches can be divided into two major categories: (1) direct approaches based on graph matching algorithms, and (2) indirect approaches such as switch-level simulation. Although no

restrictions are imposed on the design methodologies and no reference connectivity description is needed using switch-level simulation, the exhaustive exercise of input patterns requires a large amount of computation power and provides little indication of error locations. On the other hand, researchers have tried to incorporate techniques such as signature analysis to speed up the comparison process using graph isomorphism approaches and to allow subcircuit permutation and/or repetition to match circuits with the same function but different topologies. However, the graph isomorphism problem is believed to be intractable, and thus the development of an efficient algorithm for schematic comparison may be infeasible. In addition, techniques for functional equivalence based on graph matching are still preliminary [Shir86]. Therefore, it is difficult to compare functionally isomorphic circuits using graph matching algorithms. The approach in this thesis for CMOS schematic comparison, to the contrary, is to represent a CMOS transistor network by a set of logical circuit expressions, so that the comparison process is not as rigid as graph-isomorphic matching and yet efficient enough to compare two functionally identical circuits [WNW87b].

Based on logical circuit expressions, a novel circuit representation using logical predicates is proposed to describe the connectivity of a CMOS series-parallel transistor network. A context-free grammar is proposed for constructing series-parallel networks, and a pushdown automata is developed for recognizing strings generated from the context-free grammar. Thus, the synthesis and verification processes for CMOS series-parallel transistor networks can be done in linear time [WuWN87]. ITP (Interactive Theorem Prover), which was developed at Argonne National Laboratory, is used to demonstrate the capability of the approach.

7.2. Future Work

Several areas deserve further investigation in the field of CMOS VLSI design automation using logical circuit expressions.

First of all, how to incorporate timing information into logical circuit expressions and/or how to estimate timing delays based on logical circuit expressions is a big issue. As VLSI chips increase in size and complexity, the importance of timing analysis at the switch level increases as well. Although timing analysis at the switch level may not be as accurate as it is at the circuit level, the time required to examine the circuit is much less than that at the circuit level. Thus a result from a timing analyzer may be used as a reference for choosing the right logic structure, or for estimating the performance of a logic design.

Fault modeling is an important issue in VLSI circuits. Fault models in MOS VLSI circuits are used to model physical faults and explain the behavior of the faults. Many possible faults in MOS VLSI circuits have been proposed. The simple stuck-at fault model, for example, was developed based on the assumption that most physical defects have the same effect on the operation of the circuit as a set of gate inputs and outputs that are stuck at logic ZERO or logic ONE [FrMe71, Meik74]. However, a particularly troublesome case may arise in CMOS VLSI circuits: a break in a line or a transistor that is permanently off can make the output of a supposedly combinational logic circuit dependent on the previous output rather than the current input alone [Wads78]. Such a fault in CMOS circuits is called a *stuck-open* fault. Since logical circuit representation includes a partitioning scheme which decomposes a circuit into a number of components, possible faults in CMOS VLSI circuits might have specific expressions or patterns which can be easily examined. Stuck-open faults, on the other hand, might require extra effort to examine possible high-impedance states.

Other interesting topics include high-level circuit verification using logical circuit expressions and circuit sharing for multiple output designs. Additionally, how the logical circuit expressions can provide signal information to Temporal logic is also an interesting issue.

Based on the logical circuit representation, we have shown that the schematic comparison and functional recognition processes can be significantly simplified. CMOS logic design and verification using logical predicates has also been illustrated. As the design and verification at the switch level attracts more and more attention and symbolic processing is getting more and more popular, it is believed that the approach proposed in this thesis is very promising in the field of CMOS VLSI design automation.

APPENDIX

APPENDIX

Central to the operation of ITP are four lists of clauses [LuOv84]: the *axiom list*, the *set of support list*, the *have-been-given list*, and the *demodulator list*. Generally speaking, the set of support list contains a list of facts. The bits of knowledge given to a reasoning program at the start of a program is called *axioms* or *clauses*. The axiom list consists of a number of customized inference rules, which are used to derive new facts from existing ones. The demodulator list contains a set of demodulators, which are also called *rewrite rules*. The have-been-given list is basically a list of results. Each of these plays a specific role in the fundamental operation of the reasoning system. The fundamental operation consists of the following steps:

- (1) Choose a clause from the set of support list. Call this clause "the given clause".
- (2) Infer a set of clauses that have the given clause as one parent and other parent clauses selected from the axioms list, the have-been-given list, and the demodulator list.
- (3) For each generated clause, process it (for example, simplify it).
- (4) Move the given clause from the set of the support list to the have-been-given list.

This fundamental operation is repeated many times during the execution of the program until either the set of support list has become exhausted, or a contradiction has been found.

In the following example, ITP is run under its verbose mode so that the outputs are more readable and thus self-explanatory.

ITP - version 86

? k

option file to read? opt.fd

options read from file opt.fd

fdemod = y

demodcount = 1000

? r

filename? adder

axioms:

set of support:

1 ck(f, pp(dn(a, dn(b, 0)), dn(c, pp(dn(a, 0), dn(b, 0)))), pp(dp(a, dp(b, 1)), dp(c, pp(dp(a, 1), dp(b, 1))));

2 ck(outc, dn(f, 0), dp(f, 1));

3 ck(g, pp(dn(c, dn(b, dn(a, 0))), dn(f, pp(dn(a, 0), pp(dn(b, 0), dn(c, 0)))), pp(dp(c, dp(b, dp(a, 1))), dp(f, pp(dp(a, 1), pp(dp(b, 1), dp(c, 1))));

4 ck(outs, dn(g, 0), dp(g, 1));

have been given:

demodulators:

5 eq(ck(outc, x1, x2), lis(outc, x2));

6 eq(ck(outs, x1, x2), lis(outs, x2));

7 eq(dn(x1, 0), x1);

8 eq(dp(x1, 1), lno(x1));

9 eq(dn(x1, x2), lan(x1, x2));

10 eq(dp(x1, x2), lan(lno(x1), x2));

11 eq(pp(x1, x2), lor(x1, x2));

12 eq(ck(x1, x2, x3), eq(x1, x3));

13 eq(lno(lno(x1)), x1);

14 eq(lno(lor(x1, x2)), lan(lno(x1), lno(x2)));

15 eq(lno(lan(x1, x2)), lor(lno(x1), lno(x2)));

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

new left-to-right demodulator

state restored from file adder

? g

of given clauses to use? 0

> given clause number 1 is: 2 ck(outc, dn(f, 0), dp(f, 1));

16 lis(outc, lno(f)); ancestors: 2 8 7 5

given clause number 2 is: 16 lis(outc, lno(f));

given clause number 3 is: 4 ck(outs, dn(g, 0), dp(g, 1));

17 lis(outs, lno(g)); ancestors: 4 8 7 6

given clause number 4 is: 17 lis(outs, lno(g));

given clause number 5 is: 1 ck(f, pp(dn(a, dn(b, 0)), dn(c, pp(dn(a, 0), dn(b, 0)))),
pp(dp(a, dp(b, 1)), dp(c, pp(dp(a, 1), dp(b, 1))));

18 eq(f, lor(lan(lno(a), lno(b)), lan(lno(c), lor(lno(a), lno(b)))); ancestors: 1 8 8
11 10 8 10 11 7 7 11 9 7 9 11 12

new left-to-right demodulator

clause 16 back demodulated and deleted

clause 3 back demodulated and deleted

clause 2 back demodulated and deleted

given clause back demodulated

19 lis(outc, lan(lor(a, b), lor(c, lan(a, b)))); ancestors: 18 16 14 15 14 13 13 13 15
13 13

20 eq(g, lor(lan(lno(c), lan(lno(b), lno(a))), lan(lan(lor(a, b), lor(c, lan(a, b))),
lor(lno(a), lor(lno(b), lno(c)))); ancestors: 18 3 8 8 11 8 11 10 14 15 14 13 13 13
15 13 13 8 10 10 11 7 7 11 18 9 7 9 9 11 12

new left-to-right demodulator

clause 17 back demodulated and deleted

clause 4 back demodulated and deleted

21 eq(lor(lan(lno(a), lno(b)), lan(lno(c), lor(lno(a), lno(b)))), lor(lan(lno(a), lno(b)),
lan(lno(c), lor(lno(a), lno(b)))); ancestors: 18 1 8 8 11 10 8 10 11 7 7 11 9 7 9 11
12

new lex-dependent demodulator

22 lis(outs, lan(lor(c, lor(b, a)), lor(lor(lan(lno(a), lno(b)), lan(lno(c), lor(lno(a),
lno(b)))), lan(a, lan(b, c)))); ancestors: 20 17 14 15 14 14 13 13 13 15 14 15 14 15
15 13 13 13

given clause number 6 is: 19 lis(outc, lan(lor(a, b), lor(c, lan(a, b))));

given clause number 7 is: 18 eq(f, lor(lan(lno(a), lno(b)), lan(lno(c), lor(lno(a),
lno(b))));

given clause number 8 is: 21 eq(lor(lan(lno(a), lno(b)), lan(lno(c), lor(lno(a),
lno(b))), lor(lan(lno(a), lno(b)), lan(lno(c), lor(lno(a), lno(b))));

given clause number 9 is: 22 lis(outs, lan(lor(c, lor(b, a)), lor(lor(lan(lno(a), lno(b)),
lan(lno(c), lor(lno(a), lno(b))), lan(a, lan(b, c))));

given clause number 10 is: 20 eq(g, lor(lan(lno(c), lan(lno(b), lno(a))),
lan(lan(lor(a, b), lor(c, lan(a, b))), lor(lno(a), lor(lno(b), lno(c))));

23 eq(lor(lan(lno(c), lan(lno(b), lno(a))), lan(lan(lor(a, b), lor(c, lan(a, b))),
lor(lno(a), lor(lno(b), lno(c)))); lor(lan(lno(c), lan(lno(b), lno(a))), lan(lan(lor(a, b),
lor(c, lan(a, b))), lor(lno(a), lor(lno(b), lno(c)))); ancestors: 20 20

new lex-dependent demodulator

given clause number 11 is: 23 eq(lor(lan(lno(c), lan(lno(b), lno(a))), lan(lan(lor(a,
b), lor(c, lan(a, b))), lor(lno(a), lor(lno(b), lno(c)))); lor(lan(lno(c), lan(lno(b),
lno(a))), lan(lan(lor(a, b), lor(c, lan(a, b))), lor(lno(a), lor(lno(b), lno(c))));

no more clauses in the set of support

? t

level of report? 1

> layer 1 statistics:

applgets = 1567 applfrees = 1456 applist = 103 in use = 111

vargets = 74 varfrees = 71 varlist = 5 in use = 3

namegets = 2565 namefrees = 2474 namelist = 176 in use = 91

```

reIndgets = 3995 reIndfrees = 3752 reIndlist = 262 in use = 243
upbogets = 8 upbofrees = 8 upblist(r&o) = 14 in use = 0
upbrgets = 86 upbrfrees = 86 in use = 0
ivecchgets = 2003 ivecchfrees = 2003 ivecchlist = 129 in use = 0
cstrchgets = 774 cstrchfrees = 677 cstrchlist = 21 in use = 97
unification attempts = 170 successes = 166 failures = 4
half-match attempts = 146 half successes = 142 half failures = 4
unification failures due to occurs check = 0

```

```

runtime..... 6.78
unify time..... 0.13
build prop search tree 0.03
property search time.. 0.05
level of report? 2

```

> layer 2 statistics:

```

cstrgets = 80 cstrfrees = 80 cstrlist = 1 in use = 0
ivgets = 204 ivfrees = 204 ivlist = 17 in use = 0
stkntgets = 991 stkntfrees = 991 stkntlist = 2 in use = 0
number of simplifications = 127

```

Approximate times in seconds:

```

getlit clock..... 0.00

```

Fast inference statistics

```

unify attempts..... 0
successful unifs..... 0
logical inferences.... 0
intermediate subsumed. 0
deduction size prunes. 0
link depth prunes..... 0

```

level of report? 3

> layer 3 statistics:

number of input clauses... 15
 number of given clauses... 11
 clauses generated..... 5
 clauses integrated..... 8
 clauses forward subsumed.. 3
 (includes quick unit sub). 3
 (includes $x = x$ sub)..... 0
 clauses back demodulated.. 5
 clauses back subsumed..... 0
 rejected by user..... 0
 rejected by weight..... 0
 null clauses generated.... 0

approximate times in seconds:

inference rules..... 1.18
 process clauses..... 3.58
 simplification..... 0.33
 forward subsumption. 0.27
 back subsumption.... 0.02
 unit deletion..... 0.00
 auto factor..... 0.00
 unit conflict..... 0.25
 log inferences..... 0.00
 back demodulation... 2.27
 unit subsumption.... 0.22

subsuming clauses (subsumer:number_subsumed):

19: 1 21: 1 22: 1

level of report?

>

? s

filename? adder.out

current status saved in file adder.out

? q

exiting itp

BIBLIOGRAPHY

- [Aker78] Akers, S.B., "Binary Decision Diagrams", *IEEE Trans. Computers*, Vol. C-27, No. 6, pp. 509-516, June 1978.
- [BaTe80] Baker, C.M., and Terman, C., "Tools for Verifying Integrated Circuit Designs", *LAMBDA*, pp. 22-30, First Quarter, 1980.
- [Boch82] Bochman, G.V., "Hardware Specification With Temporal Logic: An Example", *IEEE Trans. on Computers*, Vol. C-31, No. 3, pp. 223-231, March 1982.
- [BrCM84] Brayton, R.K., Chen, C.L., McMullen, C.T., Otten, R.H., and Yamour, Y.J., "Automated Implementation of Switching Functions as Dynamic CMOS Circuits", *Proc. IEEE Custom Integrated Circuit Conf.*, pp. 346-350, 1984.
- [Brya81] Bryant, R.E., "MOSSIM: A Switch-Level Simulator for MOS LSI", *Proc. 18th ACM/IEEE Design Automation Conf.*, pp. 786-790, June 1981.
- [Brya84] Bryant, R.E., "A Switch-Level Model and Simulation for MOS Digital Systems", *IEEE Trans. Computers*, Vol. C-33, No. 2, pp. 160-177, Feb. 1984.
- [Brya85] Bryant, R.E., "Symbolic Verification of MOS Circuits", *Proc. Chapel Hill Conference on VLSI*, pp. 419-438, 1985.
- [CaJB79] Carter, W.C., Joyner, W.H., and Brand, D., "Symbolic Simulation for Correct Machine Design", *Proc. 16th Design Automation Conference*, pp. 280-286, 1979.
- [CoVa80] Cory, W.E., and VanCleemput, W.M., "Developments in Verification of Design Correctness", *Proc. 17th Design Automation Conf.*, pp. 156-164, June 1980.
- [Darr79] Darringer, J.A., "The Application of Program Verification Techniques to Hardware Verification", *Proc. 16th Design Automation Conference*, pp. 375-381, 1979.

- [DaJB81] Darringer, J.A., Joyner, W.H., Berman, C.L., and Trevillyan, L., "Logic Synthesis Through Local Transformations", *IBM J. Research and Development*, Vol. 25, No. 4, pp. 272-280, July 1981.
- [DiCl85] Dill, D.L., and Clarke, E.M., "Automatic Verification of Asynchronous Circuits Using Temporal Logic", *Proc. 1985 Chapel Hill Conf. on VLSI*, pp. 127-143, 1985.
- [Duff65] Duffin, R.J., "Topology of Series-Parallel Networks", *Journal of Mathematical Analysis and Applications*, Vol. 10, pp. 303-318, 1965.
- [EbZa83] Ebeling, C., and Zajicek, O., "Validating VLSI Circuit Layout by Wirelist Comparison", *Digest of Tech. Papers, ICCAD'83*, pp. 172-173, Nov. 1983.
- [Even79] Even, S., *Graph Algorithms*, Computer Science Press, 1979.
- [Floy67] Floyd, R.W., "Assigning Meanings to Programs", *Proc. Symposia in Applied Mathematics*, Vol. 19, pp. 19-32, 1967.
- [FrLi84] Friedman, V., and Liu, S., "Dynamic Logic CMOS Circuits", *IEEE Journal of Solid-State Circuits*, Vol. SC-19, No. 2, pp. 263-266, April 1984.
- [FrMe71] Frieman, A.D., and Memon, P.R., *Fault Detection in Digital Circuits*, Prentice-Hall, 1971.
- [GaKu83] Gajski, D.D., and Kuhn, R.H., "Guest Editor's Introduction: New VLSI Tools", *IEEE Computer*, pp. 11-14, Dec. 1983.
- [GlDo85] Glasser, L.A., and Dobberpuhl, D.W., *The Design and Analysis of VLSI Circuits*, Addison-Wesley, 1985.
- [GrBD86] Gregory, D., Bartlett, K., DeGeus, A., and Hachtel, G., "Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic", *Proc. 23rd Design Automation Conf.*, pp. 79-85, June 1986.
- [Hama83] Hamachi, G., "PEG Manual", *1983 VLSI Tools*, Report No. UCB/CSD 83/115, Computer Science Division of EECS, U.C. Berkeley, 1983.
- [Harr78] Harrison, M.A., *Introduction to Formal Language Theory*, University of California, Berkeley, 1978.
- [Haye84] Hayes, J.P., "Fault Modeling for Digital MOS Integrated Circuits", *IEEE Trans. Computer-Aided Design*, Vol. CAD-3, No. 3, pp. 200-207, July 1984.

- [HeGD84] Heller, L.G., Griffin, W.R., Davis, J.W., and Thomas, N.G., "Cascode Voltage Switch Logic: A Differential CMOS Logic Family", *Proc. 31st IEEE Int'l Solid-State Circuits Conf.*, Digest of Technical Papers, pp. 16-17, Feb. 1984.
- [Henn81] Hennessy, J.L., "SLIM: A Simulation and Implementation language for VLSI Microcode", *LAMBDA*, pp. 20-28, April 1981.
- [HoJa83] Hodges, D.A., and Jackson, H.G., *Analysis and Design of Digital Integrated Circuits*, MacGraw-Hill, 1983.
- [Hoff82] Hoffman, C., *Group-Theoretic Algorithms and Graph Isomorphism*, Springer-Verlag, 1982.
- [Holl87] Hollis, E.E., *Design of VLSI Gate Array ICs*, Prentice-Hall, 1987.
- [HoTa73] Hopcroft, J., and Tarjan, R., "Algorithm 477: Efficient Algorithms for Graph Manipulation", *Communication of ACM*, Vol. 16, pp. 372-378, 1973.
- [Hwan79] Hwang, K., *Computer Arithmetic: Principles, Architecture, and Design*, John Wiley and Sons, 1979.
- [John83] Johnson, S.C., "Code Generation for Silicon", *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983.
- [KaWo85] Kabat, W.C., and Wojcik, A.S., "Automated Synthesis of Combinational Logic Using Theorem-Proving Techniques", *IEEE Trans. Computers*, Vol. C-34, No. 7, pp. 610-632, July 1985.
- [KaVa81] Kang, S., and VanCleemput, W.M., "Automatic PLA Synthesis from a DDL-P Description", *Proc. 18th Design Automation Conference*, pp. 391-397, June 1981.
- [KIWS87] Kljaich, J., Wojcik, A.S., and Smith, B.T., "Formal Verification of Fault-Tolerance Using Theorem-Proving Techniques", *Proc. of the Sixth Annual IEEE Phoenix Conference on Computers and Communications*, Feb. 1987.
- [KoMc86] Kodandapani, K.L., and McGrath, E.J., "A Wirelist Compare Program for Verifying VLSI Layouts", *IEEE Design & Test*, Vol. 3, No. 3, pp. 46-51, June 1986.
- [KoWe85] Kollaritsch, P.W., and Weste, N.H., "Topologizer: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout", *IEEE J. Solid-*

State Circuits, Vol. SC-20, NO. 3, pp. 799-804, June 1985.

- [KrLL82] Krambeck, R.H., Lee, C.M., and Law, H.S., "High-Speed Compact Circuits with CMOS", *IEEE Journal of Solid-State Circuits*, Vol. SC-17, No. 3, pp. 614-619, June 1982.
- [LeSz86] Lee, C.M., and Szeto, E.W., "Zipper CMOS", *IEEE Circuits and Devices*, Vol. 2, No. 3, pp. 10-17, May 1986.
- [LePa81] Lewis, H.R., and Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall, 1981.
- [LuOv84] Lusk, E.L., and Overbeck, R.A., "The Automated Reasoning System ITP", Argonne National Laboratory, April 1984.
- [MacM92] MacMahon, P.A., "The Combination of Resistances", *The Electrician*, April 8, 1892.
- [MaNE82] Matthews, R., Newkirk, J., and Eichenberger, P., "A Target Language for Silicon Compilers", *IEEE COMPCON*, pp. 349-353, 1982.
- [McCo80] Mead, C., and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.
- [Meik74] Mei, K.C., "Bridging and Stuck-At Faults", *IEEE Trans. Computers*, Vol. C-23, pp. 720-726, July 1974.
- [Mukh86] Mukherjee, A., *Introduction to nMOS and CMOS VLSI Systems Design*, Prentice-Hall, 1986.
- [MyIv85] Myers, D.J., and Ivey, P.A., "A Design Style for VLSI CMOS", *IEEE Journal of Solid-State Circuits*, Vol. SC-20, No. 3, pp. 741-745, June 1985.
- [Nage75] Nagel, L.W., "SPICE: A Computer Program to Simulate Semiconductor Circuits", *Ph.D. Thesis*, Memorandum No. ERL-M520, Electronics Research Laboratory, EECS, University of California, Berkeley, 1975.
- [NeSa86] Newton, A.R., and Sangiovanni-Vincentelli, A.I., "Computer-Aided Design for VLSI Circuits", *IEEE Computer*, Vol. 19, No. 4, pp. 38-60, April 1986.
- [Nies83] Niessen, C., "Hierarchical Design Methodologies and Tools for VLSI Chips", *Proc. of the IEEE*, pp. 11-14, Jan. 1983.

- [Nils83] Nilsson, J.W., *Electric Circuits*, Addison-Wesley, 1983.
- [Oklo82] Oklobdzija, V.G., "Design for Testability of VLSI Structures Through the Use of Circuit Techniques", *Ph.D. Thesis*, University of California, Los Angeles, 1982.
- [OkMo86] Oklobdzija, V.G., and Montoye, R.K., "Design-Performance Trade-Offs in CMOS Domino Logic", *IEEE Journal of Solid-State Circuits*, Vol. SC-21, No. 2, pp. 304-306, April 1986.
- [OuSM85] Ousterhout, J.K., Scott, W.S., Mayo, R.N., and Hamachi, G., "1986 VLSI Tools: Still More Works by the Original Artists", Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California, Berkeley, Dec. 1985.
- [Park84] Parker, A.C., "Automated Synthesis of Digital Systems", *IEEE Design & Test*, Vol. 1, No. 2, pp. 75-81, Nov. 1984.
- [PaSS87] Pasternak, J.H., Shubat, A.S., and Salama, C.A., "CMOS Differential Pass-Transistor Logic Design", *IEEE J. Solid-State Circuits*, Vol. SC-22, No. 2, pp. 216-222, April 1987.
- [PiSt83] Pitchumani, V., and Stabler, E.P., "An Inductive Assertion Method for Register Transfer Level Design Verification", *IEEE Transactions on Computers*, Vol. C-32, No. 12, pp. 1073-1080, Dec. 1983.
- [RaWM85] Radhakrishnan, D., Whitaker, S.R., and Maki, G.K., "Formal Design Procedures for Pass Transistor Switching Circuits", *IEEE Journal of Solid-State Circuits*, Vol. SC-20, No. 2, pp. 531-536, April 1985.
- [ReRe86] Reddy, S.M., and Reddy, M.K., "Testable Realizations for FET Stuck-Open Faults in CMOS Combinational Logic Circuits", *IEEE Trans. Computers*, Vol. C-35, No. 8, pp. 742-754, Aug. 1986.
- [ReSS85] Reineke, P.S., Skinner, R.D., Swanson, M.C., and Winkelmann, D.A., "STATUS 1985: A Report on the Integrated Circuit Industry", Integrated Circuit Engineering Corporation, Dec. 1985.
- [ReUr71] Rescher, N., and Urquhart, A., *Temporal Logic*, Springer-Verlag, 1971.
- [RiSh42] Riordan, J., and Shannon, C.E., "The Number of Two-Terminal Series-Parallel Networks", *Journal of Mathematics and Physics*, Vol. 21, pp. 83-

93, 1942.

- [Roth77] Roth, J.P., "Hardware Verification", *IEEE Trans. on Computers*, Vol. C-26, No. 12, pp. 1292-1294, Dec. 1977.
- [Sequ83] Sequin, C.H., "Managing VLSI Complexity: An Outlook", *Proceedings of the IEEE*, Vol. 71, No. 1, pp. 149-166, Jan. 1983.
- [Shan38] Shannon, C.E., "A Symbolic Analysis of Relay and Switching Circuits", *Transactions of A. I. E. E.*, Vol. 57, pp. 713-723, 1938.
- [Shir86] Shiren, Y., "YNCC: A New Algorithm for Device-Level Comparison Between Two Functionally Isomorphic VLSI Circuits", *Digest of Tech. Papers, ICCAD'86*, pp. 298-301, Nov. 1986.
- [Shiv83] Shiva, S.G., "Automatic Hardware Synthesis", *Proceedings of the IEEE*, Vol. 71, No. 1, pp. 76-87, Jan. 1983.
- [SmBH82] Smith, G.L., Bahnser, R.J., and Halliwell, H., "Boolean Comparison of Hardware and Flowcharts", *IBM J. Research and Development*, Vol. 26, No. 1, pp. 106-116, Jan. 1982.
- [SpNe83] Spickelmier, R.L., and Newton, A.R., "WOMBAT: A New Netlist Comparison Program", *Digest of Tech. Papers, ICCAD'83*, pp. 170-171, Nov. 1983.
- [SrAg86] Srinivas, N.C., and Agrawal, V.D., "PROVE: Prolog Based Verifier", *Digest of Tech. Papers, ICCAD'86*, pp. 306-309, Nov. 1986.
- [StSh86] Sterling, L., and Shapiro, E., *The Art of Prolog: Advanced Programming Techniques*, The MIT Press, 1986.
- [Subr86] Subrahmanyam, P.A., "Synapse: An Expert System for VLSI Design", *IEEE Computer*, Vol. 19, No. 7, pp. 78-89, Jul. 1986.
- [SuFr86] Supowit, K.J., and Friedman, S.J., "A New Method for Verifying Sequential Circuits", *Proc. 23rd Design Automation Conf.*, pp. 200-207, June 1986.
- [SuOA73] Suzuki, Y., Odagawa, K., and Abe, T., "Clocked CMOS Calculator Circuitry", *IEEE J. Solid-State Circuits*, Vol. SC-8, No. 6, pp. 462-469, Dec. 1973.

- [TaMC82] Takashima, M., Mitsuhashi, T., and Chiba, T., "Programs for Verifying Circuit Connectivity of MOS/LSI Mask Artwork", *Proc. 19th Design Automation Conf.*, pp. 544-550, June 1982.
- [Term83] Terman, C.J., "Simulation Tools for Digital LSI Design", MIT Laboratory for Computer Science, TR-304, 1983.
- [Tric85] Trickey, H.W., "Compiling Pascal Programs Into Silicon", *Ph.D. Thesis*, Department of Computer Science, Stanford University, July 1985.
- [TyEl85] Tygar, J.D., and Ellickson, R., "Efficient Netlist Comparison Using Hierarchy and Randomization", *Proc. 22nd Design Automation Conf.*, pp. 702-708, June 1985.
- [VaSh85] Vai, M.K., and Shanblatt, M.A., "Performance-Design Tradeoff of Hierarchical VLSI Design Entry Points", *Master Thesis*, Michigan State University, Aug. 1985.
- [WNW87a] Wu, C.E., Ni, L.M., and Wojcik, A.S., "Functional Recognition of Static CMOS Circuits", *1987 International Conf. on Computer-Aided Design*, Nov. 1987.
- [WNW87b] Wu, C.E., Ni, L.M., and Wojcik, A.S., "Comparison of CMOS Schematics Using a Logical Circuit Representation", *1987 International Conf. on Computer Design*, Oct. 1987.
- [Wads78] Wadsack, R.L., "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits", *Bell System Tech. Journal*, Vol. 57, pp. 1449-1474, May-July 1978.
- [WeEs85] Weste, N.H., and Eshraghian, K., *Principles of CMOS VLSI Design: A Systems Perspective*, Addison-Wesley, 1985.
- [Whit83] Whitaker, S., "Pass-Transistor Networks Optimize n-MOS Logic", *Electronics*, pp. 144-148, Sep. 22, 1983.
- [Wojc83] Wojcik, A.S., "Formal Design Verification of Digital Systems", *Proc. 20th Design Automation Conference*, pp. 228-231, June 1983.
- [WoKS84] Wojcik, A.S., Kljaich, J., and Srinivas, N., "A Formal Design Verification System Based on an Automated Reasoning System", *Proc. 21st Design Automation Conference*, pp. 641-647, June 1984.

- [WoOL84] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., "Logic Circuit Design", *Automated Reasoning: Introduction and Applications*, Chap. 7, pp. 186-223, 1984.
- [WuNi86] Wu, C.E., and Ni, L.M., "VLSI Design Automation: Introduction and Experiments", *Technical Report*, MSU-ENGR-86-016, Dept. of Computer Science, Michigan State University, July 1986.
- [WuWN86] Wu, C.E., Wojcik, A.S., and Ni, L.M., "CMOS Circuit Representation, Verification, and Synthesis Using Automated Reasoning", *Technical Report*, Dept. of Computer Science, Michigan State University, Nov. 1986.
- [WuWN87] Wu, C.E., Wojcik, A.S., and Ni, L.M., "A Rule-Based Circuit Representation for Automated CMOS Design and Verification", *Proc. 24th Design Automation Conf.*, pp. 786-792, June 1987.
- [WuNi87] Wu, C.E., and Ni, L.M., "Asynchronous Arbiter Design Using CMOS DCVS Logic", *Digest of 1987 Symposium on VLSI Circuits*, pp. 96-97, May 1987.
- [Youn86] Young, J., "TRW's Superchip Passes First Milestone", *Electronics*, pp. 49-54, McGraw-Hill, July 10, 1986.

MICHIGAN STATE UNIV. LIBRARIES



31293010858912